

Python for Engineers and Scientists

A Complete Hands-On Journey from Core Python to Numerical and Scientific Applications

—

Carlos Márcio da Silva Freitas

Apress®

Carlos Márcio da Silva Freitas

Python for Engineers and Scientists
A Complete Hands-On Journey from Core Python to Numerical and Scientific Applications

Apress®

Carlos Márcio da Silva Freitas
São Paulo, Brazil

ISBN 979-8-8688-3164-5 e-ISBN 979-8-8688-3165-2
https://doi.org/10.1007/979-8-8688-3165-2

© Carlos Márcio da Silva Freitas 2026

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Unless otherwise expressly agreed by the Publisher in writing, the Publisher does not authorize use or reproduction of all or any part of this work in any manner for the purpose of training, fine-tuning, evaluating, or developing artificial intelligence models, technologies, or systems. In accordance with Article 4(3) of the EU Directive 2019/790 on copyright and related rights in the Digital Single Market, the Publisher expressly reserves all rights with respect to this work under the text and data mining exception of the directive.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.comwww.springeronline.com, or visit . Apress Media, LLC is a Delaware LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.This Apress imprint is published by the registered company APress Media, LLC, part of Springer Nature.
The registered company address is: 1 New York Plaza, New York, NY 10004, U.S.A.

I dedicate this work to my beloved wife, Josenir, for supporting me unconditionally in all my projects and at every stage of my life. You are God's greatest gift in my entire life. Te amo, princesa!!!!!!

Introduction

About This Book

Python for Engineers and Scientists provides a structured, hands-on path for learning Python and applying it to real engineering and scientific problems. The book begins with the foundations of the language and gradually progresses toward numerical computation, scientific analysis, data processing, visualization, graphical application development, control systems, and database management.

The main purpose of this book is not simply to teach programming syntax. It is to show how Python can become a practical engineering tool. Throughout the chapters, programming concepts are connected to tasks such as processing measurements, performing mathematical calculations, analyzing experimental data, creating technical graphs, simulating dynamic systems, building interactive applications, and storing engineering information.

Python is especially valuable in technical fields because it combines readable syntax with a large ecosystem of scientific and engineering libraries. This makes it suitable for rapid prototyping, automation, simulation, research, data analysis, and the development of complete technical applications. The book presents these capabilities progressively so that readers can understand not only how to write Python code but also how to organize, extend, and apply it effectively.

Who This Book Is For

Since the basic concepts of the Python language are presented in an instructional manner rich in examples, this book is intended for anyone wishing to learn programming; however, as the chapters progress, the focus shifts primarily to engineers, scientists, researchers, students, and technical professionals who want to use Python for scientific computing, engineering analysis, simulation, automation, and data-driven applications.

It is suitable for readers from various engineering disciplines, including electrical, mechanical, civil, chemical, control, automation, mechatronics, robotics, aerospace, and related fields. It may also prove useful to scientists and researchers requiring a practical introduction to programming for numerical analysis, experimentation, visualization, and technical data management.

The content is designed for readers ranging from beginner to intermediate levels. No prior Python experience is required, as the book begins by covering installation, environment setup, and the language's fundamental elements. Readers who already possess basic Python knowledge can use the early chapters as a review and move more quickly to scientific libraries and applied engineering topics.

Basic familiarity with mathematics and technical problem-solving is advantageous, especially in the final chapters; however, programming concepts are introduced gradually and accompanied by practical examples.

Structure of the Book

The book is organized into three broad stages.

The first stage, covering Chapters 1–8, develops a solid foundation in Python programming. It introduces the language, variables, collections, modules, functions, control flow, object-oriented programming (OOP), and file handling. These chapters provide the knowledge required to write organized, reusable, and reliable programs.

The second stage, covering Chapters 9–12, focuses on mathematics, visualization, and scientific computing. It introduces Python's mathematical tools and the Matplotlib, NumPy, and SciPy libraries. These chapters show how Python can be used to perform numerical calculations, work efficiently with arrays and matrices, solve scientific problems, process signals, and communicate results through technical visualizations.

The third stage, covering Chapters 13–16, applies Python to broader engineering workflows. It explores graphical user interfaces (GUIs), structured data analysis with Pandas, dynamic system modeling and control, and database development with SQLAlchemy. Together, these chapters demonstrate how the concepts presented earlier can be combined to create more complete engineering applications. Chapters 1–3 include practical exercises to help readers reinforce the fundamental concepts introduced at the beginning of the book. Chapters 4–16 contain numerous worked examples that demonstrate how to apply the concepts presented. These examples serve as the main resource for practical learning in those chapters, so separate exercise sets are not included. Readers are encouraged to run, modify, and extend the examples to deepen their understanding.

Readers who are new to Python are encouraged to follow the chapters in sequence. Readers with previous programming experience may study individual chapters as references, although the applied chapters assume familiarity with the core concepts introduced earlier.

Before You Begin

This is a hands-on book. Readers are encouraged to type, execute, modify, and test the examples rather than only reading them. Changing values, introducing new conditions, adding data, and experimenting with alternative solutions are important parts of the learning process.

Using a virtual environment is strongly recommended. It keeps project dependencies isolated, reduces conflicts between library versions, and makes examples easier to reproduce. Because scientific Python libraries continue to evolve, readers should also verify the

installed library versions and consult the official documentation when working with a newer release.

The examples are designed to explain individual concepts clearly, but they also represent building blocks for larger projects. A numerical routine may later become part of a graphical application, a Pandas analysis may use data retrieved from a database, and a control system simulation may combine NumPy, SciPy, Matplotlib, and the control systems library.

By the end of the book, readers should have more than a collection of isolated Python commands. They should have a practical foundation for designing programs, processing technical data, performing scientific calculations, visualizing results, creating interactive applications, modeling engineering systems, and managing persistent information with confidence.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

Acknowledgments

I cannot begin without thanking God for granting me the health, peace, and perseverance needed to fulfill my dream of writing this book. I would also like to thank everyone involved in the preparation and publication of this work, whose dedication was essential in turning a cherished dream into reality.

Table of Contents

Chapter 1: Introduction

What Is Python?

A Brief History of Python

Ecosystem and Community

Inner Workings of Python

Lexical and Syntax Analysis

Bytecode and the Python Virtual Machine

CPython Implementation and Memory Management

Python Enhancement Proposal

Python Files and Folders

.py Files

pyproject.toml

.ini/.cfg/.config Files

setup.py

.yaml/.yml

requirements.txt

.gitignore

MANIFEST.in

.env Files

.ipynb

.json

Makefile

README.md and CHANGELOG.md

Folder Organization
Installing Python
Installing Python on Windows
Installing Python on macOS via the .pkg File
Installing Python on Linux
The Python Virtual Environment
How Does venv Work?
Creating a Virtual Environment
Choosing a Code Editor
The First Program
More About the print Function
Newline (\n)
Separator (sep)
Line Ending (end)
Adding Symbols and Emojis
Getting Keyboard Data
The Importance of Structure: Indentation
Formatting Your Output with F-Strings
Formatting Your Output with T-Strings
Table of Useful Commands: Windows and macOS/Linux
Python Project Management with UltraViolet (UV)
Installing UV
Using UV for Dependency Management
Practice Exercises

Answer Key

Chapter 2: Variables

Introduction

Variables and Types

Integers and Floating Numbers

Floating-Point Arithmetic Error

Mathematical and Utility Operators

Arithmetic Operations

Bitwise Operations

Conversion Methods

Representation Methods

Utility Methods

Built-In Functions

Making Comparisons

Working with Strings

Declaring Strings

Immutability of Strings

Indexing and Slicing Strings

String Operations

Converting Data Types

Explicit Typing with Type Hints

Explicit Type Conversion

Local and Global Variables in Python

Access Modifiers

Introduction to Collections
Overview of Collection Types
Summary of the Main Differences Between Collections
Practice Exercises
Answer Key
Chapter 3: Handling Collections
Lists
Creating Lists
Accessing and Modifying Elements
Methods for Lists in Python
Operations with Lists
len()
index()
append() and remove()
extend()
insert()
pop()
clear() and del
sort() and reverse()
copy()
Remove Duplicates
count()
min() and max()
sum()

Membership Operators
Concatenation Operator
Repetition Operator
Slicing Operator
Equality Operators
Lists and the Equals Sign
Operations Between Lists
Nested List
Common Error Messages for List Methods
List Comprehension
Tuples
Accessing Elements
Methods for Tuples in Python
len()
index()
Concatenation (+)
Repetition (*)
Membership Operators
Slicing (:)
count()
sorted ()
Sum (sum())
min() and max()
Nested Tuples

Converting a Tuple to a List
Operations Between Different Tuples
Set
Common Operations with Sets
Adding and Removing Elements
Updating Elements
Operations Between Sets
Dictionary
Creating Dictionaries
Accessing and Modifying Data
Common Operations in a Dictionary
Add/Update an Item
Delete an Item
Check for a Key
Get Length
get()
keys()
values()
items()
update()
pop()
popitem()
clear()
copy()

setdefault()
Comprehension
Operations Between Dictionaries
Practice Exercises
Answer Key

Chapter 4: Python Modules and Packages
Python Path and Modules Location
Types of Python Modules
Built-In Modules
Standard Library Modules
Third-Party Modules
Dynamic Modules
Custom Modules
Compiled Modules
Importing Modules
Dynamic Import
Relative and Absolute Import
Importing Custom Modules in Different Directories

Chapter 5: Functions
Parameters and Arguments
Positional Arguments
Keyword Arguments
Default Arguments
Variable-Length Positional Arguments (*args)

Variable-Length Keyword Arguments **kwargs
Positional-Only Arguments
Keyword-Only Arguments
Unpacking Arguments
Combining Argument-Passing Methods
Lambda Expressions
Assigning a Lambda to a Variable
Using a Lambda with Built-In Functions
Using a Lambda in Sorting
Using Lambda Expressions in List Comprehensions
Using Lambda Expressions in Conditional Statements
Using Lambda Expressions in Dictionaries
Using a Lambda in Functions
Using any() and all()
Using Lambda Expressions in sorted() with Multiple Keys
Decorators (@)

Chapter 6: Conditionals and Loops
Conditional Statements
Simplified if Statement
The if Statement with else
The if Statement with elif
While Loops
Loops with for
Jump Statements

Control Flow Statement with “break”
Control Flow Statement with “continue”
Using Jump Statements in Loops
Exception Handling
Chapter 7: Object-Oriented Programming
Object-Oriented Programming
Classes
Subclasses
OOP Concepts
Encapsulation
Abstraction
Inheritance
Polymorphism
Relationships Between Objects
Association
Aggregation
Composition
Attributes
Private and Protected Attributes
Dynamic Attributes and Static Attributes
Property Attributes
Methods
Instance Methods
Class Method

Static Method
Abstract Method
Dunder Methods
Constructor Method
super() Built-In Function
self and cls
Property Method
Decorators with OOP Principles
Dependency Injection
Dependency
Design Patterns
Dependency Injection
Walking Further in OOP

Chapter 8: Files and Paths
Overview of File Types in Python
Working with the open() Function
CSV Files
Creating and Editing Word and Excel Documents
Handling .xlsx Files
Filesystem Paths with pathlib

Chapter 9: Math
Math Built-In Modules
Math
Floating-Point Arithmetic

Floating-Point Manipulation Functions
Power, Exponential, and Logarithmic Functions
Summation and Product Functions
Angular Conversion
Trigonometric Functions
Hyperbolic Functions
Special Functions
Cmath
Decimal
Arithmetic Context
Fractions
Statistics
Random
Chapter 10: Matplotlib
Creating a Graph with Matplotlib
An Example of a Simple Graph
Graph Customizations
Lines and Spines
Text and Labels
Legends
Grid and Ticks
Scales
Linear Scale
Logarithmic Scale

Symmetric Logarithmic Scale
Logit Scale
Inverse Sine Hyperbolic Scale
Custom Scales
Chapter 11: NumPy
Introduction
Installation and Version Check
NumPy and SciPy
NumPy Subpackages
Data Types
NumPy Data Types: Examples
Python Objects in NumPy Arrays
Multidimensional Arrays and Matrices
Array Creation Patterns in NumPy
Accessing Elements in NumPy Arrays
Indexing in Different Dimensions
Indexing Errors
Array Properties
Array Inspection and Utility Functions
Operations on Arrays
Array Operation Families in NumPy
Arithmetic Operations
Unary Arithmetic
Rounding and Approximation

Arithmetic Helpers for Numerical Stability

Binary Arithmetic with Broadcasting

Complex Number Arithmetic

Aggregation and Reduction

Trigonometric, Hyperbolic, Exponential, and Logarithmic Functions

Logical, Boolean, Comparison, Sorting, Searching, and Set Operations

Linear Algebra Operations

Array Manipulation

NumPy: Common User Errors and Best Practices

Common User Errors in NumPy

Best Practices for NumPy

Chapter 12: SciPy

A Look into Key Subpackages

Constants

Physical and Mathematical Constants

SI Prefixes

Unit Conversion

Differentiate

Derivative

Jacobian

Hessian

FFT

1D Fourier Transforms

2D Fourier Transforms
N-Dimensional Fourier Transforms
Discrete Sine and Cosine Transforms (DST and DCT)
Fast Hankel Transforms
Helper Functions
Backend Control
FFT-Based Filtering
Integrate
Improving Integration Performance
Calculating a Definite Double Integral
Solving ODEs
Interpolate
Univariate Interpolation with Pchip
Interpolating Cubic Polynomial with CubicSpline
Multivariate Interpolation with griddata
B-spline Representation of 1D Data
2D Spline Interpolation
Rational Approximation and Additional Tools
Linalg
Basic Operations
Eigenvalue Problems
Decompositions
Matrix Functions and Equation Solvers
Other Special Functions

Optimize
Local Multivariate Optimization
Global Optimization
Curve Fitting/Least Squares
Signal
MATLAB Compatibilities
Example Codes
Design of a FIR Filter
LTI System Conversions
Chapter 13: Graphical User Interfaces (GUIs)
User Interface Fundamentals
Command-Line Interface (CLI)
Graphical User Interface (GUI)
The Python GUI Ecosystem
Comparison of Python Libraries
Why Choose Kivy for Engineering?
Getting Started with Kivy
Installation and Modules
Architecture: Separating Logic (.py) and Design (.kv)
First Application: “Hello engineer!”
Layouts and Positioning Systems
Understanding Layouts
Common Layout Managers
Advanced Layouts

Absolute Positioning (Core Properties)
Relative Positioning (Hints)
Layout-Specific Positioning
Kivy Widgets Catalog
Overview of Widgets and Events
Button
CheckBox
Progress Bar
Label
Slider
Text Input
Switch
Popup
Spinner
Image
Practical Project: Temperature Converter
Project Goal and Architecture
Complete Application Code
Code Walkthrough
Chapter 14: Introduction to the Pandas Library for Engineering
Installing the Pandas Library
Fundamental Pandas Data Structures
Reading and Importing Data
Reading CSV Files

Reading Excel Files
Inspecting and Exploring Data
Selecting Data in a DataFrame
Column Selection (Series vs. DataFrame)
Row Filtering with Logical Conditions
Data Manipulation and Cleaning
Creating New Columns
Removing Columns
Handling Missing Values (NaN)
Changing Data Types
Statistical Analysis and Grouping
Descriptive Statistics
Grouping Data with groupby()
Sorting and Combining DataFrames
Sorting Data
Combining DataFrames (Merge and Join)
Concatenating DataFrames
Exporting Data
Exporting Data to CSV
Exporting Data to Excel
Exporting to Other Formats
Basic Data Visualization with Pandas
Line and Scatter Plots
Engineering Line and Scatter Plots

Histograms and Bar Charts

Chapter 15: Control Systems with Python

Dynamic System Control

System Creation

Create a State-Space System

Create a Transfer Function

Frequency Response Data

Create a Nonlinear System

Create a Transfer Function from Zeros, Poles, and Gain

Model Conversions and Interconnections

Convert a System into a Canonical Form

Series Connection of I/O Systems

Parallel Connection of I/O Systems

Time-Domain Analysis

Forced Response of a Linear System

Compute the Impulse Response for a Linear System

Compute the Step Response for a Linear System

Plot a Phase Plane Diagram

Frequency-Domain Analysis

Bode Plot for a System

Nyquist Plot for a System

Nichols Plot for a System

Find the First Frequency at Which Gain Drops by 3 dB

Return the Zero-Frequency (or DC) Gain

Calculate Gain and Phase Margins and Associated Crossover Frequencies

Calculate Stability Margins and Associated Crossover Frequencies

System Properties

Compute System Poles

Compute System Zeros

Controller Design

Root Locus Plot

Pole Placement Using Ackermann's Method

Nonlinear Systems

Find an Operating Point for a Nonlinear System

Linearize an Input/Output System at a Given State and Input

Control Theory

Calculate the Controllability Matrix

PID Controller Design

PID Design with Pole Allocation

PID Design with the Root Locus Method

PID Design by Frequency Response

Chapter 16: Databases Using SQLAlchemy

Databases and SQLAlchemy

Databases in Engineering Applications

Types of Databases and Engineering Use Cases

Why Use an ORM in Engineering Projects

Overview of SQLAlchemy

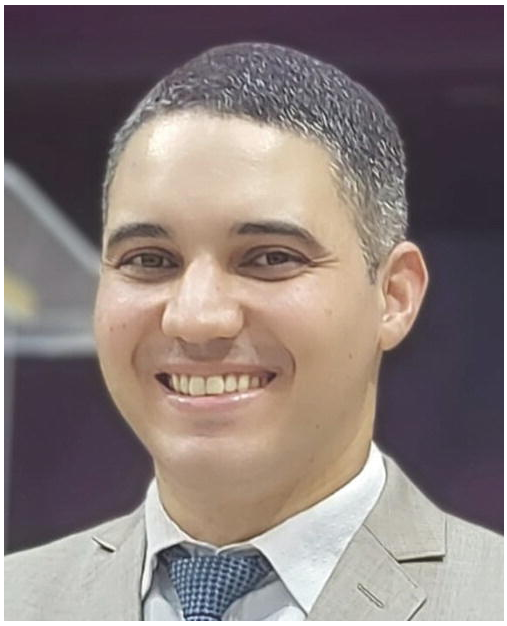
Getting Started with SQLAlchemy
Installing SQLAlchemy with pip
Database Connection and Engine Creation
The SQLAlchemy Engine Concept
Connection Strings and Database Dialects
SQLite As a Prototyping Database
Engineering Example: Wind Turbine Monitoring Database
Sessions: The Gateway to Your Database
Understanding Sessions and Transactions
Creating Sessions with sessionmaker
Declarative Base and ORM Infrastructure
Defining Data Models for Engineering Applications
ORM Models and Table Mapping
Engineering Example: Turbine and SensorReading Models
Turbine Model Definition
SensorReading Model Definition
Relationships and Table Creation
CRUD Operations for Engineering Data
Create: Logging New Sensor Readings
Read: Querying Engineering Data
Update: Correcting Erroneous Measurements
Delete: Removing Obsolete Records
Complete SQLAlchemy Example
Full Wind Turbine Monitoring Script

Bibliographic References

Index

About the Author

Carlos Freitas



is an electrical engineer with a master's degree in Computational Systems Applied to Engineering and specializations in Software Engineering and Control Systems. He began his career as an electromechanical technician, building a solid foundation in electrical systems, electronics, and industrial automation. His work spans machine automation, industrial systems, and AI applications using Python and TensorFlow. Carlos is currently pursuing a PhD in Electrical Engineering with a focus on Robotics and teaches engineering and technical programs at the Federal Institute of Education in Brazil.

About the Technical Reviewer

Shoumik Chakravarty



is a Product Development leader at Verizon, building the next chapter of the ecosystem—connected devices, kid-safe experiences, and the engineering that ties them together. He pitches, prototypes, and pressure-tests what ships next, working across product, engineering, design, and UX. He is the named inventor on US Patent Application US20240329939A1, an author of peer-reviewed publications, a holder of multiple product and technology certifications, and a regular hackathon competitor. He is passionate about the STEM student community—judging STEM competitions and early-stage entrepreneur pitches and reading and reviewing technical books.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_1

1. Introduction

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

What Is Python?

Python is a general-purpose, open source programming language. It is free to use and comes at no cost to the user. Unlike MATLAB used in scientific applications, VHDL used in electronic chip design, or LaTeX used to create texts and documents, Python does not have a single application niche. Instead, it can be applied to any field where its features are useful. It is also a high-level language, meaning that code written in Python is more human-readable compared with low-level languages like Assembly and even compared with languages such as C and C++.

These factors have contributed to the rapid popularization of Python in many areas of computing, including web development, artificial intelligence, data analysis, task automation, and scientific and engineering applications. Its widespread recognition is largely due to its strong developer community that creates valuable libraries, its ease of learning, and its open availability at no cost.

This balance between simplicity and rich functionality is one of the main reasons for Python's popularity in both the academic community and professional engineering applications.

A Brief History of Python

According to the Python documentation, the language was created by Dutch programmer Guido van Rossum in the late 1980s while working at the Centrum Wiskunde & Informatica (CWI) in Amsterdam. It began as a personal project and was developed during his Christmas vacation in 1989, becoming one of the most widely used and beloved languages worldwide today.

Contrary to what most people imagine, the name "Python" was not inspired by the snake but rather by Guido's love for the British comedy group *Monty Python's Flying Circus*.

The first official release, Python 0.9.0, occurred in 1991. Since then, Python has undergone continuous development in both features and runtime performance.

On January 1, 2020, the Python community officially ended support for Python 2, closing a chapter in its history. The focus is now fully on Python 3, which continues to evolve rapidly with contributions from the Python Software Foundation (PSF) and its massive global community.

Python is commonly described as a bytecode-interpreted language. In CPython, source code is compiled to intermediate bytecode rather than directly to a native machine-code executable, and the Python Virtual Machine (PVM) executes that bytecode. This execution model makes

Python flexible, interactive, and beginner-friendly because developers can test small pieces of code and receive immediate feedback. These qualities make Python especially valuable for engineering, prototyping, education, data analysis, and research, where rapid development may be more important than raw speed. By contrast, languages such as C, C++, Rust, and Go are typically compiled to native machine code before execution, which can provide better runtime performance. The additional build step can make development cycles less interactive, but these languages are well suited to performance-critical applications such as operating systems and game engines.

At its core, Python is object-oriented, encouraging developers to structure code around reusable and modular "objects" that combine both data and behavior. However, Python is also multi-paradigm: it supports procedural programming (step-by-step instructions) and functional programming (writing clean, mathematical-style functions), giving developers the freedom to choose the best style for their problem.

Another defining feature of Python is that it is dynamically typed. This means you don't have to explicitly declare the type of a variable; Python figures it out at runtime.

While this flexibility speeds up development, Python is also strongly typed, which prevents unintended operations, such as adding a number to a string unless you explicitly handle the conversion. This combination of dynamic typing with strong rules makes Python both flexible and safe to use.

Python is also cross-platform, running smoothly on Windows, macOS, and Linux and even on mobile devices and embedded systems. This portability ensures that code written on one system can typically run on another with little or no modification.

Due to its simplicity and versatility across many areas of computing, the Python programming language has become one of the most widely used languages in recent years. This trend is evident in major research studies and market reports, including the following:

Stack Overflow Developer Survey: An extensive annual survey covering programming languages and tools
GitHub Octoverse Report: An annual analysis of activity on GitHub
TIOBE Index: A monthly ranking based on search engine query volumes for different programming languages
PYPL (Popularity of Programming Language) Index: A monthly index that measures how often programming language tutorials are searched on Google
JetBrains State of Developer Ecosystem Survey: An annual report from JetBrains, focusing on language usage, tools, and industry trends

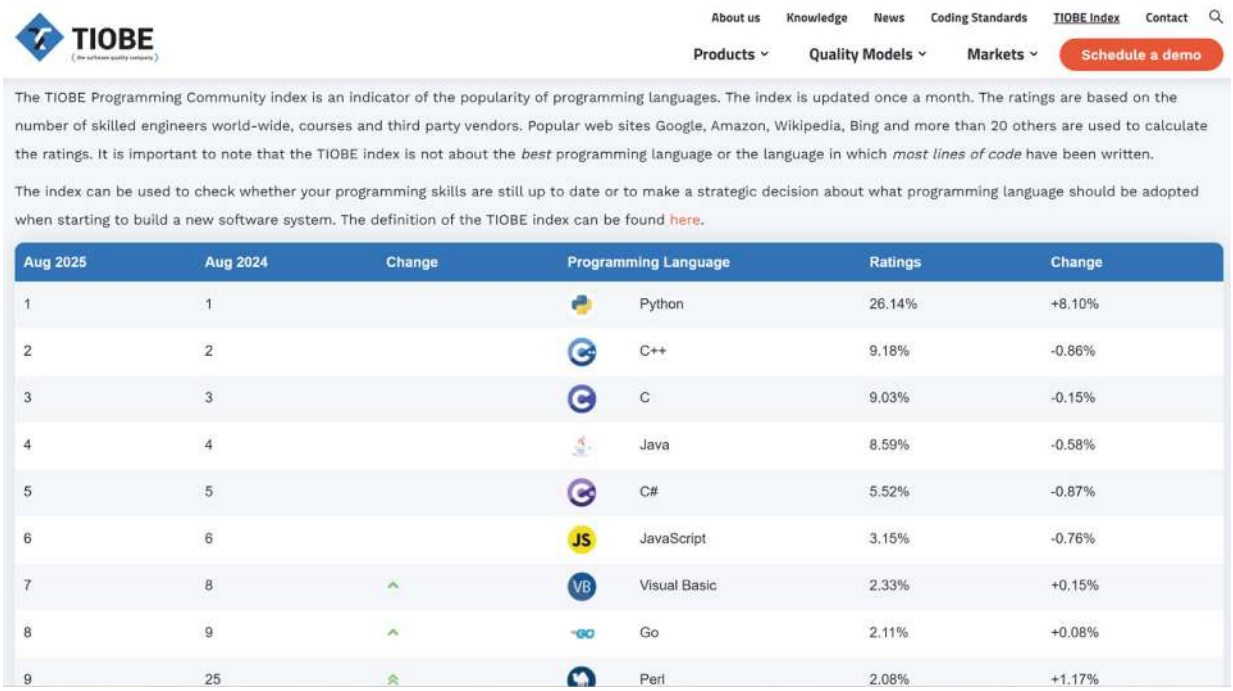


Figure 1-1 TIOBE website index that measures the popularity of programming languages monthly based on search results. Available at https://www.tiobe.com/tiobe-index/, accessed on August 15, 2025

Ecosystem and Community

One of Python's greatest strengths is its ecosystem and community. It is supported by a massive, active, and collaborative global network of developers who contribute to its growth, create tutorials, answer questions, and continuously improve the language. Thanks to this community, Python provides a vast collection of libraries and frameworks, giving developers pre-written and optimized code for nearly any field imaginable.

For engineering applications, Python has become one of the most powerful and widely used tools. Engineers across disciplines from mechanical and electrical to civil, chemical, and aerospace rely on Python not just for quick prototyping, but also for solving highly complex problems. Its accessibility and versatility allow it to bridge the gap between theory, simulation, and practical implementation.

Some key areas where Python empowers engineers include

- Numerical computing and simulation
- Data analysis and visualization
- Control systems and automation
- Machine learning (ML) and artificial intelligence
- Computer-Aided Engineering (CAE)
- Embedded systems and robotics
- Civil and structural engineering

Inner Workings of Python

Lexical and Syntax Analysis

When Python code is created and saved with the .py extension, execution begins with a stage called lexical analysis. During this phase, the Python interpreter scans the code and breaks it into smaller components called tokens. These tokens include keywords, variable names, constants, operators, and punctuation symbols, which serve as the fundamental building blocks of the Python language. The interpreter also checks for illegal characters or invalid patterns; if any are found, it raises a lexical error.

The next step is syntactic analysis, or parsing, where the interpreter checks whether the code complies with Python's syntax rules, such as verifying that the instructions are valid. This process uses a hierarchical structure that represents the program's logical flow, called the Abstract Syntax Tree (AST). If no issues are detected, the AST is compiled into bytecode, a low-level platform-independent representation of the code.

Bytecode and the Python Virtual Machine

Bytecode is also used by other modern language implementations, including Java and C#. It serves as an intermediate representation of Python code. In CPython, reusable bytecode is typically cached in .pyc files inside the __pycache__ directory, a convention used since Python 3.2, which can speed up subsequent program runs.

The bytecode is then executed by the Python Virtual Machine (PVM), which manages tasks such as memory allocation, communication with the operating system, and interaction with external libraries. The PVM also handles Python's dynamic typing, meaning that variable types are determined at runtime rather than in advance, as in compiled languages that translate source code directly into machine-code executables. As the bytecode executes, the PVM performs operations such as calling functions, updating variables, and manipulating data structures.

CPython Implementation and Memory Management

The most widely used Python interpreter is CPython, the reference implementation of the language written in C. CPython provides the complete core functionality: the interpreter, the PVM, and Python's standard library. It executes code, manages resources, and exposes operating system features to Python programs.

A distinctive feature of CPython is its reference counting memory management. Each time an object is created, CPython tracks how many references point to it. Assigning the object to another variable increases this count, and when no references remain, CPython automatically frees the memory. While this system is efficient, poorly structured code can still lead to memory leaks.

Another point of CPython is its comprehensive standard library, offering ready-to-use modules for file handling, networking, mathematics, data structures, and more. Since much of the library is written in C, it combines speed with portability, allowing the same Python code to run across multiple platforms without modification. CPython also includes a foreign function interface (FFI), enabling Python programs to call C functions directly and work with C data—a capability particularly valuable in scientific computing, machine learning, and data processing, where performance is critical.

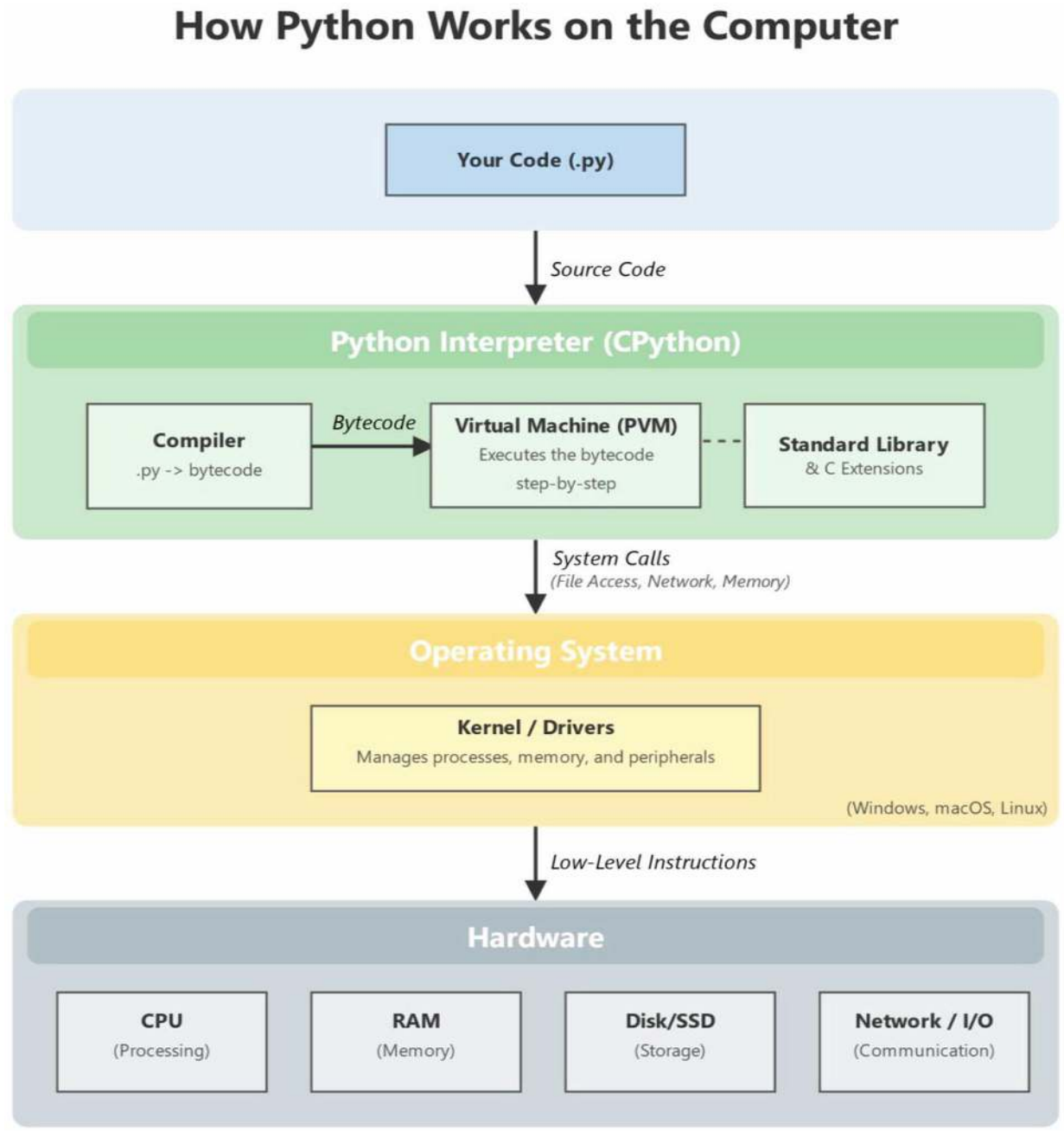


Figure 1-2 Basic Python operation flowchart

Python Enhancement Proposal

A PEP, short for Python Enhancement Proposal, is central to Python's evolution and acts as a roadmap for new features, improvements, and governance decisions. It helps keep the language organized, consistent, and responsive to the community's needs.

A PEP is an official document that proposes a new idea, feature, or improvement for the Python language. It is a formal proposal in which someone from the community identifies a problem, explains a potential solution, and discusses its possible impact on Python as a whole.

The purpose of the PEP system is to provide the Python community with a structured way to review and refine ideas before they are incorporated into the language. This collaborative process ensures that changes are carefully considered and genuinely beneficial.

Python Enhancement Proposals (PEPs) are categorized into several types. Standards Track PEPs propose changes or additions to the Python language itself or its standard library, aiming to enhance its functionality. Informational PEPs provide guidelines, recommendations, or background information without altering the language, serving as valuable resources for developers. Process PEPs, on the other hand, define or update Python's development workflows and decision-making processes, ensuring the language's evolution is well-managed and community-driven. These categories help organize and prioritize PEPs, facilitating Python's ongoing development and improvement.

Table 1-1 lists some examples of community-defined PEPs.

Table 1-1 Small example of community-defined PEPs



See the complete list on https://peps.python.org/#.

Python Files and Folders

When developing software in any programming language, understanding the typical file and folder structure is crucial for creating maintainable and scalable projects. In Python, different types of files serve different purposes, and knowing how to organize them effectively can significantly improve code readability and project management.

The most common files in Python are those with the .py extension, which contain the source code executed by the Python interpreter.

In addition to Python source files, projects often include external files that serve specific purposes depending on the application's requirements. For example, .csv files are widely used to store tabular data, .json files are commonly used for configuration or structured data, and .txt files might contain documentation or logs. Leveraging these external files allows developers to separate data from code, making programs more flexible and easier to maintain.

.py Files

Python source files (.py) are the fundamental building blocks of any Python project, containing the core program logic written by the developer, including functions, classes, calculations, and executable statements. These files are interpreted and executed directly by the Python interpreter. Developers can fully edit .py files using any text editor or a modern integrated development environment (IDE), which offers advanced features such as debugging, code completion, and error detection. For larger projects, code can be organized into multiple .py files or modules, promoting modular design, maintainability, and code reuse.

Python source files can also appear in several forms that serve different purposes in a project. Compiled Python files (.pyc) store bytecode derived from the original .py code; they are normally created automatically in the __pycache__ directory and are not meant to be edited. Since Python 3.5, optimized bytecode also uses .pyc files with an optimization tag, and .pyo files

are no longer used. .pyd files are compiled extension modules, generally written in C or C++, that can be imported like standard modules on Windows to provide high-performance functionality or access to system-level resources. Lastly, .pyi files are stub files containing type hints and function or class signatures. Static type checkers such as mypy use them to detect type errors before runtime without affecting program execution.

pyproject.toml

The pyproject.toml file is the central configuration file for modern Python projects, often replacing the traditional setup.py and setup.cfg files. It defines project metadata, build system requirements, and tool-specific settings, enabling developers to manage dependencies, packaging, and project tools such as Poetry or Flit, which are designed specifically for dependency management and packaging. By editing this file, developers can specify important project details, including the project name, version, and authors, as well as configure build tools, dependencies, and other project-specific settings. This standardized approach simplifies project configuration, improves maintainability, and makes it easier to manage complex Python projects.

.ini/.cfg/.config Files

Files with extensions like .ini, .cfg, or .config are traditional configuration files used to store application settings. They organize information into sections, with each option defined as a key–value pair, making them simple for humans to read and edit. Developers often use these files to define how an application or library should behave, such as setting paths, enabling or disabling features, or adjusting default values. In Python, these files are commonly handled with the built-in configparser module, which makes reading and writing configuration data straightforward.

setup.py

The setup.py file is a traditional build script used for packaging and distributing Python projects. It usually contains important information about the project, such as its name, version, author, dependencies, and instructions for installation. Developers edit this file to prepare their code so it can be installed with tools like pip and shared on repositories such as the Python Package Index (PyPI). Although setup.py is still supported, many modern projects now prefer pyproject.toml, which provides a cleaner and more standardized way to manage packaging and dependencies.

.yaml/.yml

YAML (YAML Ain't Markup Language) files (.yaml/.yml) are human-readable configuration files that represent structured data using an indentation-based syntax. They are widely used in modern development for application settings, deployment configurations, and automation tasks such as continuous integration and delivery (CI/ CD) pipelines. Developers can freely edit YAML files with any text editor, and in Python they are commonly processed using libraries like PyYAML or ruamel.yaml. Compared with JSON (JavaScript Object Notation), YAML emphasizes readability and simplicity, allowing developers to define complex configurations in a clear and organized way.

requirements.txt

requirements.txt is a text file used to list a project's dependencies, often including specific version numbers to ensure compatibility. It's important for creating reproducible environments across different machines, allowing all required packages to be installed consistently.

To do this, simply use the command pip install -r requirements.txt. This file can be generated automatically or written and edited by developers with commands like pip freeze > requirements.txt. Due to its widespread adoption, it's one of the most common files included in Python projects for configuration and deployment.

.gitignore

The .gitignore file is used in Git version control systems to specify which files or directories should be excluded from tracking. Common examples include virtual environments, compiled bytecode, log files, and other sensitive data that should not be committed to a repository.

MANIFEST.in

The MANIFEST.in file is used in Python packaging to specify additional files that should be included in a project's distribution, such as documentation, configuration files, license texts, or data assets. It works alongside setup.py (or modern alternatives) to ensure that everything required beyond the source code is properly packaged. Developers can freely edit this file using simple inclusion rules, making it flexible and straightforward to use. Proper configuration of MANIFEST.in ensures that distributed packages are complete and functional and include all the necessary resources for users.

.env Files

Environment variable (.env) files are text files used to store sensitive configuration values, such as API keys, database credentials, and special settings. Because they are separate from visible source code, .env files help keep projects more secure and easier to configure. In Python, libraries like python-dotenv are commonly used to load .env values into application environment variables, enabling secure and flexible configuration management.

.ipynb

Jupyter Notebook (.ipynb) files are interactive, cell-based documents that combine executable Python code, text, equations, and visual output in a single location. Because they are organized into cells, notebooks are especially popular in data science, machine learning, and research. Tools like Jupyter Notebook, JupyterLab, Visual Studio (VS) Code, and Google Colab are commonly used to edit .ipynb files.

.json

JSON (JavaScript Object Notation) is a text-based format for structuring data in a way that is both human-readable and machine-friendly. It is widely used for configuration files and as the standard format for APIs, enabling communication between different systems. In Python, the built-in json module makes it simple to read, write, and process JSON data.

Makefile

A Makefile is a plain-text file that defines automation tasks executed with the make command. It can be used to compile code, run tests, or deploy applications. Developers can tailor the file

to fit project workflows, and it is often integrated into CI/CD pipelines to streamline repetitive processes.

README.md and CHANGELOG.md

Among many other important files in a project are README.md and CHANGELOG.md. The README.md file provides an overview of the project, including installation instructions and usage examples, serving as a first reference point for users and contributors. The CHANGELOG.md file, on the other hand, keeps a record of project changes, such as bug fixes, new features, and significant changes, providing transparency and helping users understand what's new in each release. Both files are fully editable and essential for open source projects, especially those hosted on platforms like GitHub.

Table 1-2 summarizes the essential file extensions found within a Python project, categorizing each file by its function, from source code and configuration to documentation and packaging, and describing its specific role.

Table 1-2 List of essential files used by Python

Folder Organization

Python itself does not enforce a strict project structure; following a well-organized folder hierarchy is highly recommended. Separating project files into dedicated folders improves readability, facilitates collaboration, and supports long-term maintenance and expansion. A typical Python project might include folders for source code (src or app), tests (tests), configuration files (config), data files (data), and documentation (docs). This separation not only helps developers locate and manage files efficiently but also ensures that the project can scale smoothly as new features are added.

Furthermore, adopting a consistent folder structure enhances portability and team collaboration, as new contributors can quickly understand how the project is organized.

Careful attention to file and folder organization is a fundamental aspect of Python development. By combining a clear folder structure with appropriate use of external files, developers can create projects that are easier to navigate, maintain, and expand, ensuring both short-term efficiency and long-term sustainability.

See Figure 1-3 for an example of a basic Python project containing the most commonly used folders and files.

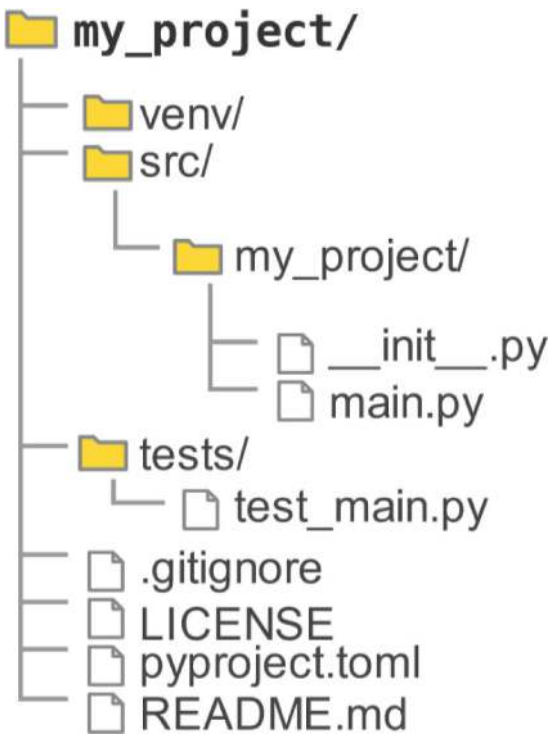


Figure 1-3 Example of folder structure used in a Python project

Table 1-3 details these common folders, explaining their primary contents, purpose, and typical use cases. Think of this as a highly recommended starting point, not an unbreakable rule. The best structure is one that fits your project's specific needs. The ultimate goal is consistency, which ensures that you and any future collaborators can easily understand the project's layout and contribute effectively.

Table 1-3 List of essential folders used by Python

Installing Python

Python programs normally require a compatible Python interpreter at runtime. Unlike applications distributed as native executables, a Python script is processed by the interpreter, which in CPython compiles it to bytecode and executes that bytecode. Installing Python is therefore a fundamental first step for most developers and users.

Official installers are readily available for all major operating systems, but how you approach installation can differ:

- *Windows*: Python is usually not pre-installed. Download it from the official page at https://www.python.org/downloads/. When using the standalone installer, selecting "Add python.exe to PATH" makes Python easier to access from the command line.
- *macOS and Linux*: Do not assume that a current, user-managed Python installation is available. Recent macOS systems may expose an Apple-controlled /usr/bin/python3 associated with developer tools, while Linux distributions vary in what they install by

default. Do not modify an operating system–managed interpreter; install a separate current version for development when needed.

Regardless of the operating system, it's always best practice to install the latest stable version of Python to benefit from new features, performance improvements, and essential security updates.

To avoid conflicts with the system Python or other projects, developers use virtual environments. A virtual environment is an isolated directory containing a specific version of Python and its own set of libraries, ensuring that each project's dependencies are kept separate and manageable.

Installing Python on Windows

Go to https://www.python.org/downloads/. Under Windows, choose the standalone installer for the latest stable release to follow the steps shown here; the Python install manager may also be offered on the download page.

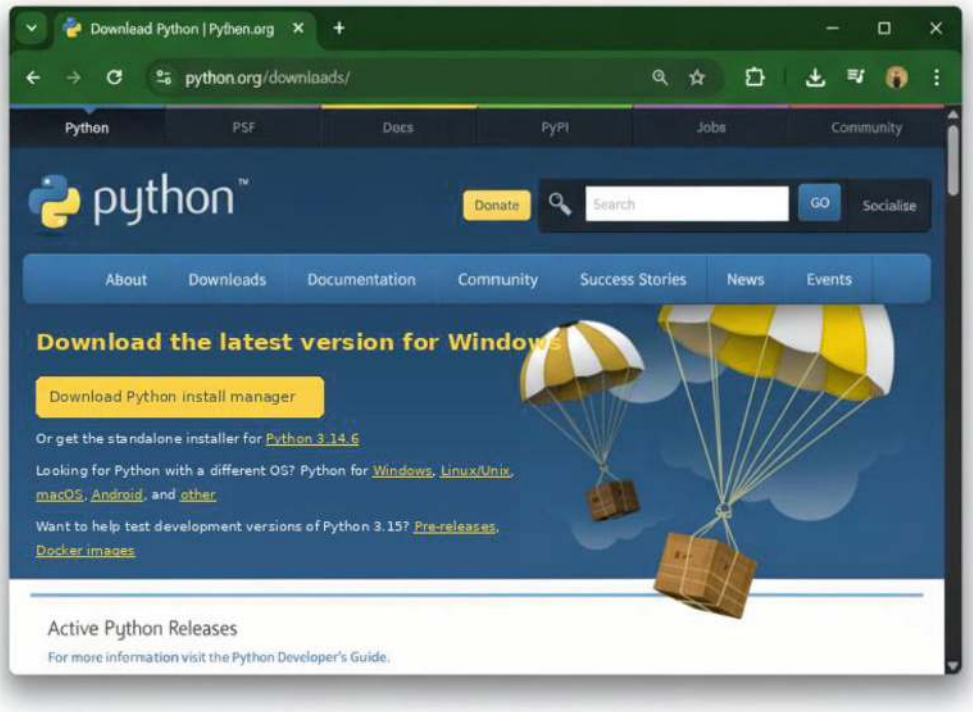


Figure 1-4 Python download page for Windows. The available buttons and version numbers may change by release. Accessed on July 16, 2026

Then open the file and start the installation process.

In the installer window, select the "Use administrator privileges when installing py.exe" and "Add python.exe to PATH" checkboxes, and if you want to use the express installation, click "Install Now." To explore the other installation options, select the "Customize Installation" option. This option gives you full control over the installation process.

Let's break down the reasons for each option:

- "Add python.exe to PATH"

The "PATH" is a Windows environment variable that contains a list of directories. When you type a command in the terminal like pip, Windows searches for the corresponding executable in those directories. Checking this box adds the folder where Python was installed to this list.

If you don't check this option, you'll only be able to run Python if you're in the exact folder where it's installed or by typing the full path to the file. By checking this box, you can open any terminal in any folder on your computer and simply type python to launch the interpreter or pip install <package> to install libraries. This makes using Python infinitely more convenient.

- "Use administrator privileges when installing py.exe"

  py.exe is the "Python Launcher" for Windows. It's a utility that helps you manage and run multiple versions of Python you may have installed. Checking this box installs py.exe in a system directory (C:\Windows\), which requires administrator privileges.

  When installed with administrator privileges, the py.exe launcher is globally available to all users on the system and works correctly even in terminals run as administrator.



Figure 1-5 Python installer window for Windows

After selecting the "Customize Installation" option and accessing the "Optional Features" window, it is not necessary to change any checkboxes; just click the "Next" button.



Figure 1-6 Python installer window with optional features

In the “Advanced Options” window, select “Install Python for all users” only when a system-wide installation is appropriate, and then click “Install.” Other options can remain at their defaults unless your environment requires changes.

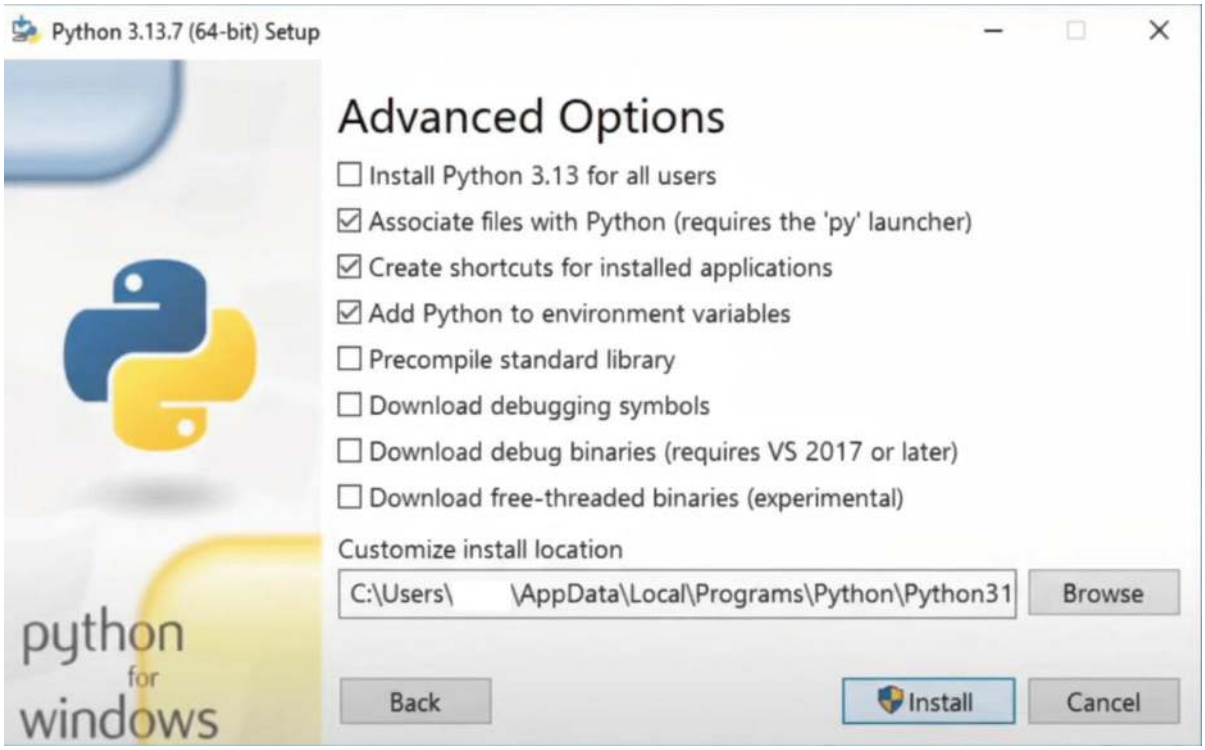


Figure 1-7 Python installer window with advanced options

After completing the installation process, a completion message will be displayed in the installer window as shown in Figure 1-8.

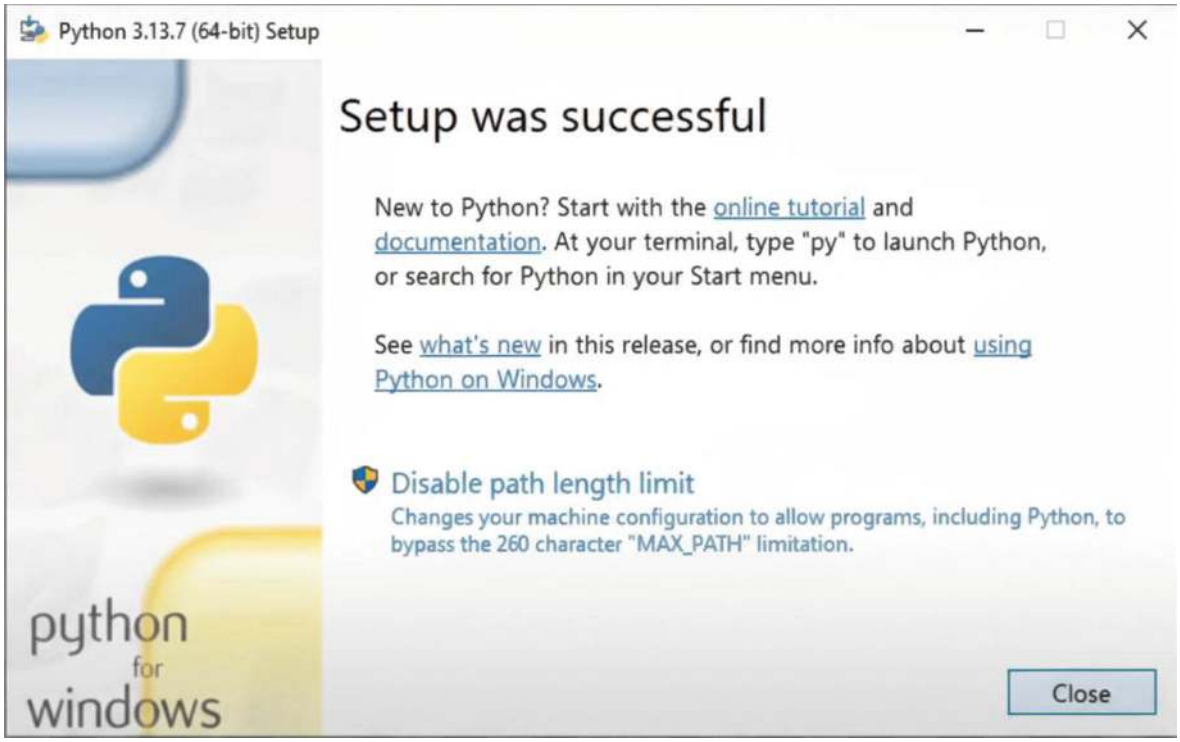


Figure 1-8 Python installer window indicating installation completion

After completing the installation, verify the installed Python version. The most convenient method is to use command prompt or PowerShell.

1. Open the terminal:
   Click the Start menu and type cmd for command prompt or powershell.
   Press Enter to open the terminal window.
2. Type the command:
   In the window that appears, type one of the commands below and press Enter.
   First Option:

```
py --version
```

   or

```
py -V
```

   Second Option:

```
python --version
```

   or

```
python -V
```

Now, see the result:
The terminal will display the installed Python version, something like Figure 1-9.

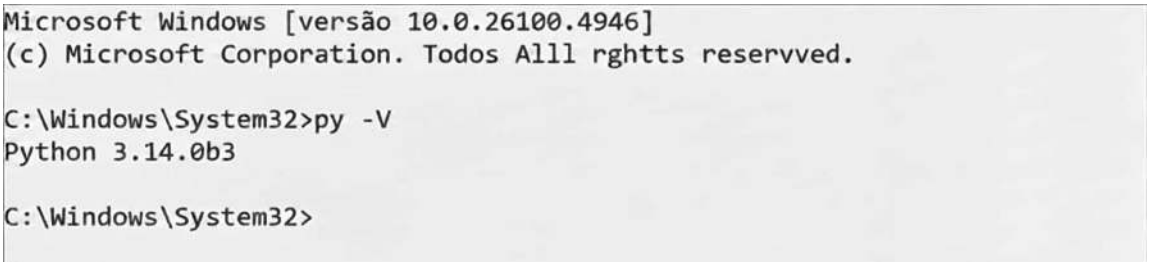


Figure 1-9 Screenshot showing the terminal output after the command to check the installed Python version

To list all installed versions, use

```
py -0p
```

The version that appears with the asterisk * indicates the default version.

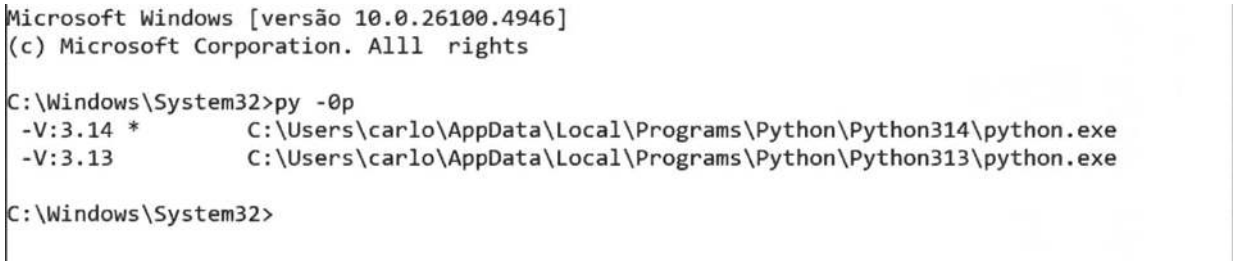


Figure 1-10 Screenshot showing the terminal output after the command to check the Python versions installed on the computer

Solving PATH and Alias Issues on Windows



On Windows 11, sometimes typing python in the terminal does not work, while the py command does. This happens because of Microsoft's execution aliases and the way the system manages the PATH variable.

How do you solve the problem of Python commands not working on Windows 11?
A very useful command to know the installation location of Python through the terminal is the command

```
where python
```

As the "python" command may not be working, use the "py" command at this time:

```
where py
```

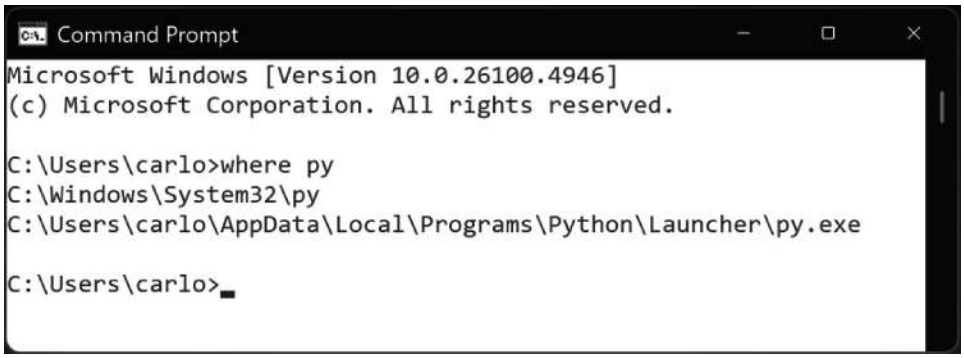


Figure 1-11 Screenshot showing the terminal output after running the command to check the location of the Python installation

Below are two effective solutions:

Method 1: Disable the alias.

This is the simplest and most direct method. You just need to disable the execution shortcuts that Windows uses.

- Open the Start menu.
- Type "Manage app execution aliases" and click the option that appears.
- A settings window will open with a list of applications.
- Find python.exe and python3.exe (app installer).
- Turn both of them off.
- Close and reopen your terminal CMD or PowerShell for the changes to take effect.
- When you type python, Windows will ignore Microsoft's fake shortcuts and continue searching in your PATH until it finds your real Python installation.

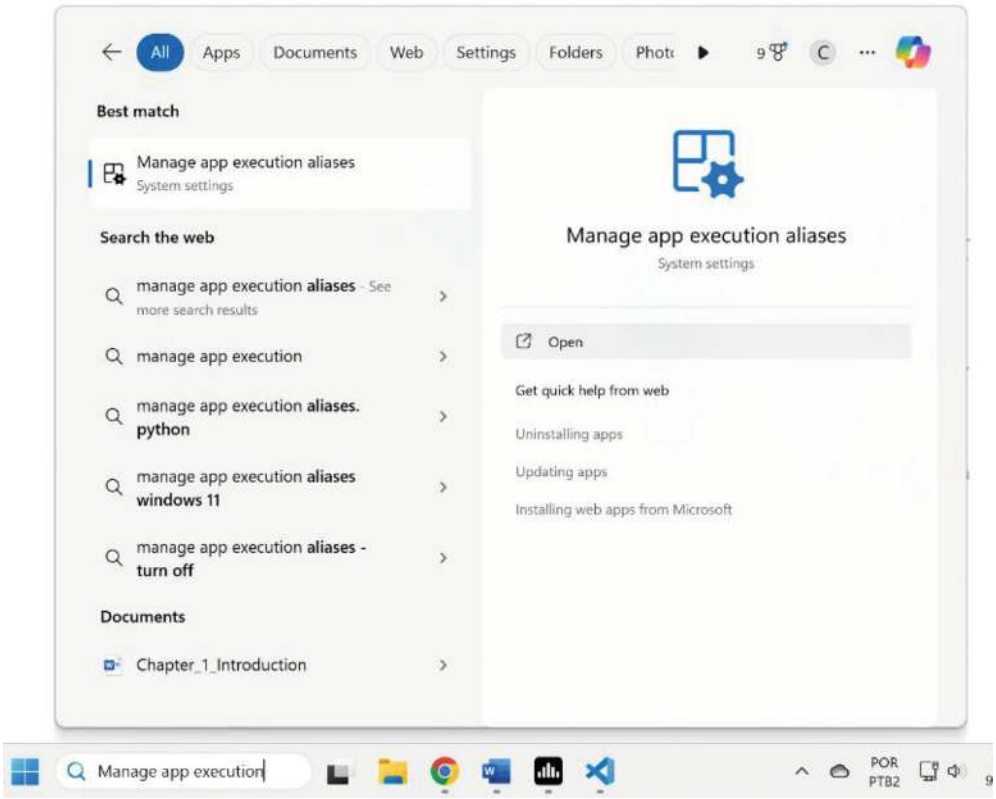


Figure 1-12 Accessing the Manage app execution aliases configuration in Windows 11

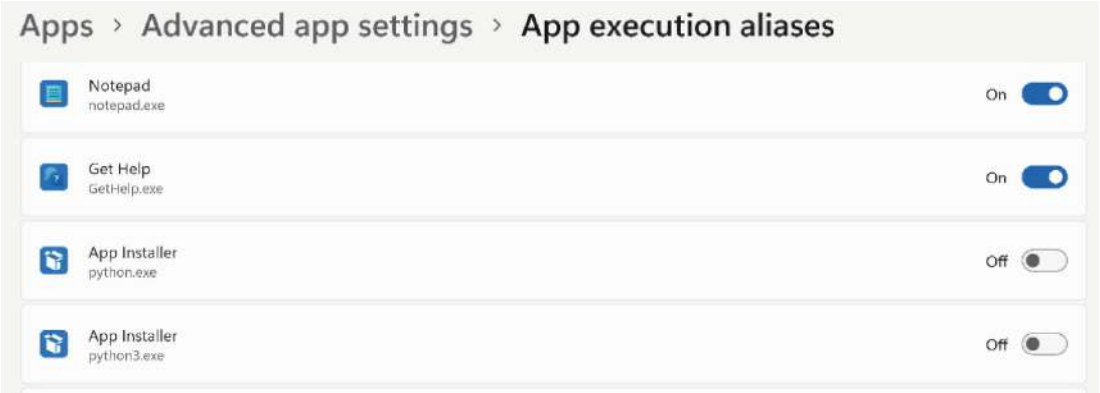


Figure 1-13 Changing the App execution aliases setting in Windows 11

Method 2: Fix the PATH order.

This method ensures your real Python installation is always prioritized over the Microsoft shortcuts.

Step 1: Check installation settings.

- If you install Python from python.org, always check the box “Add Python to PATH” during setup.
- This automatically adds the correct folders to your PATH variable.

Step 2: Fix the PATH manually.

If you forgot to check the box, you can add the paths yourself:

1. Open the Start menu and search for “Edit the system environment variables” for your account.
2. In the window that opens, find the variable Path and double-click it.
3. Click New and add the paths to your Python installation, for example:
   - C:\Python311\
   - C:\Python311\Scripts\
4. Use the Move Up button to place these entries above the line ending with \Microsoft\WindowsApps.
5. Click OK on all windows to save.
6. Restart your terminal for the changes to apply.

As an example, see the configuration sequence in the following figures.

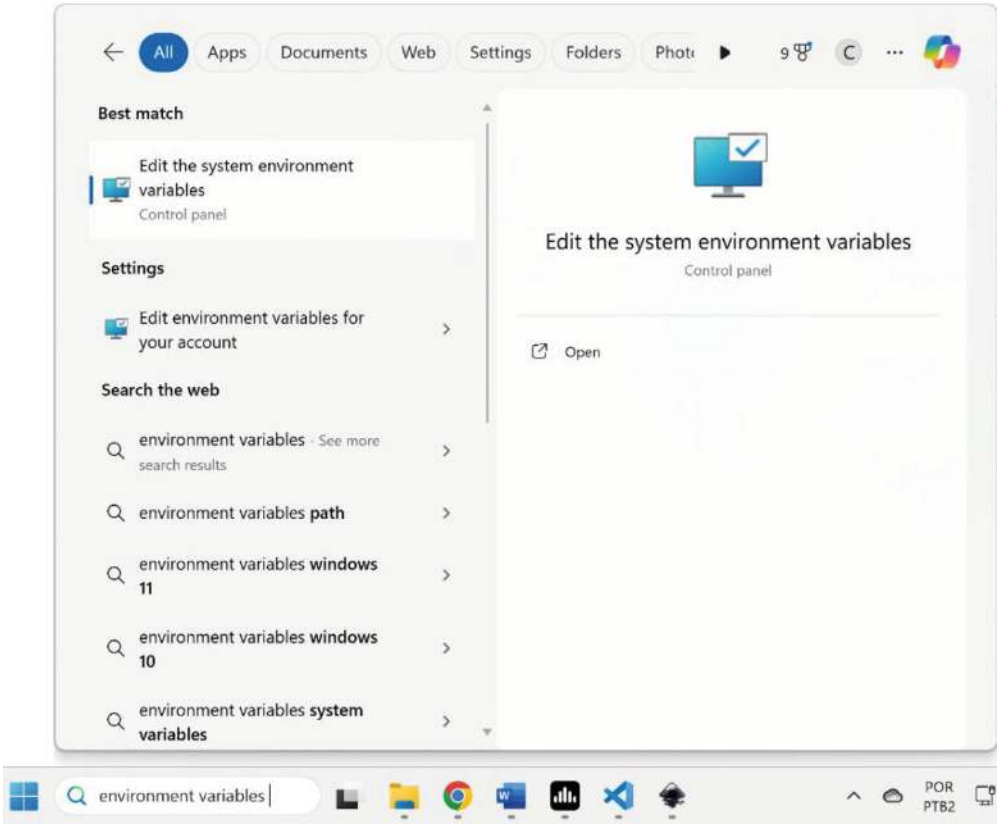


Figure 1-14 Accessing the environment variable configuration in Windows 11

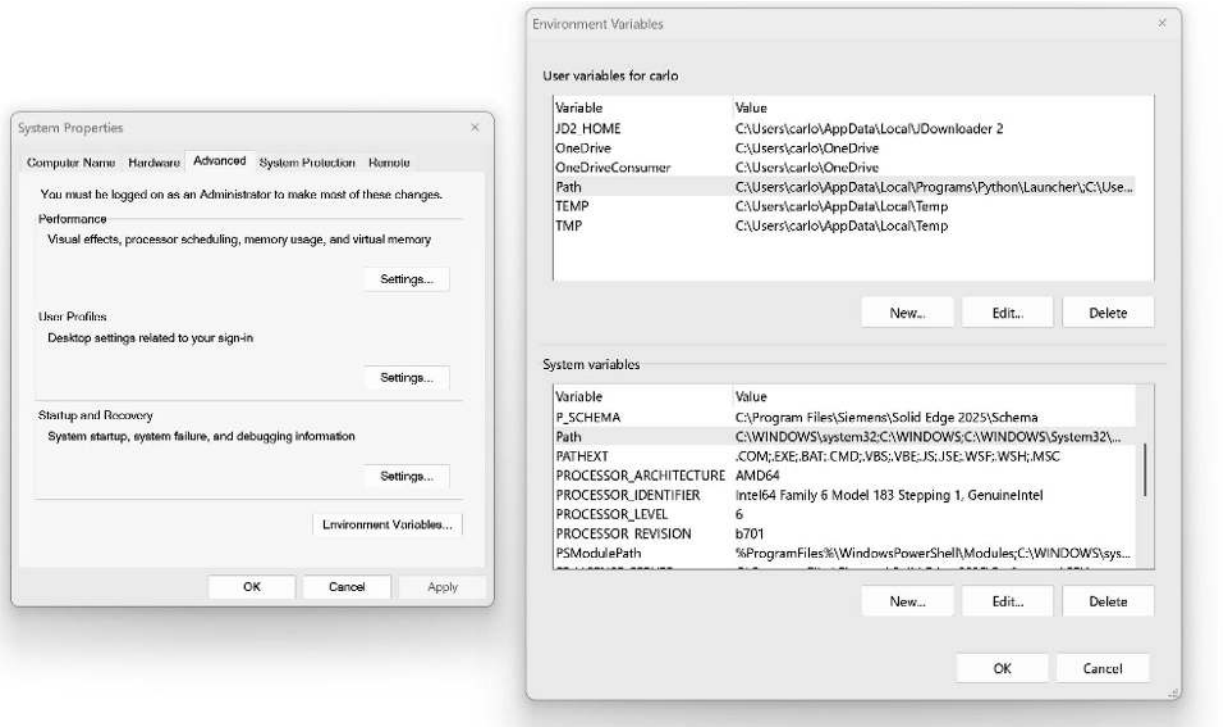


Figure 1-15 Environment variable configuration windows in Windows 11

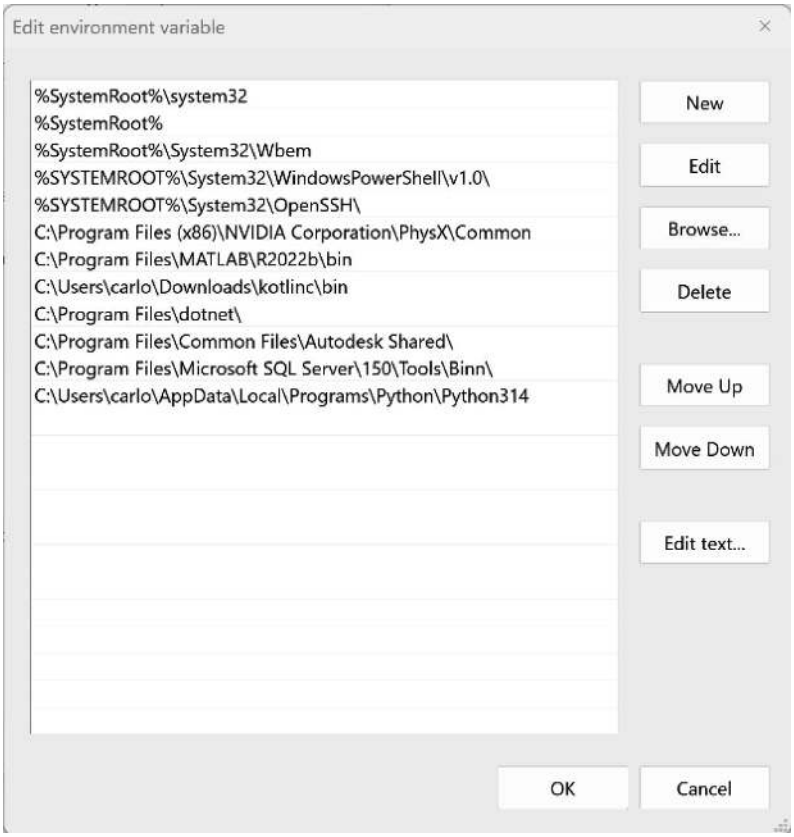


Figure 1-16 Adding the Python environment variable entry for Windows 11

Installing Python on macOS via the .pkg File

The installation procedure was performed on macOS 14 Sonoma. Follow these installation steps:

Go to the official download page at https://www.python.org/downloads/macos/. Select the latest stable macOS installer and follow the on-screen instructions. The interface and version number may differ from the screenshot.

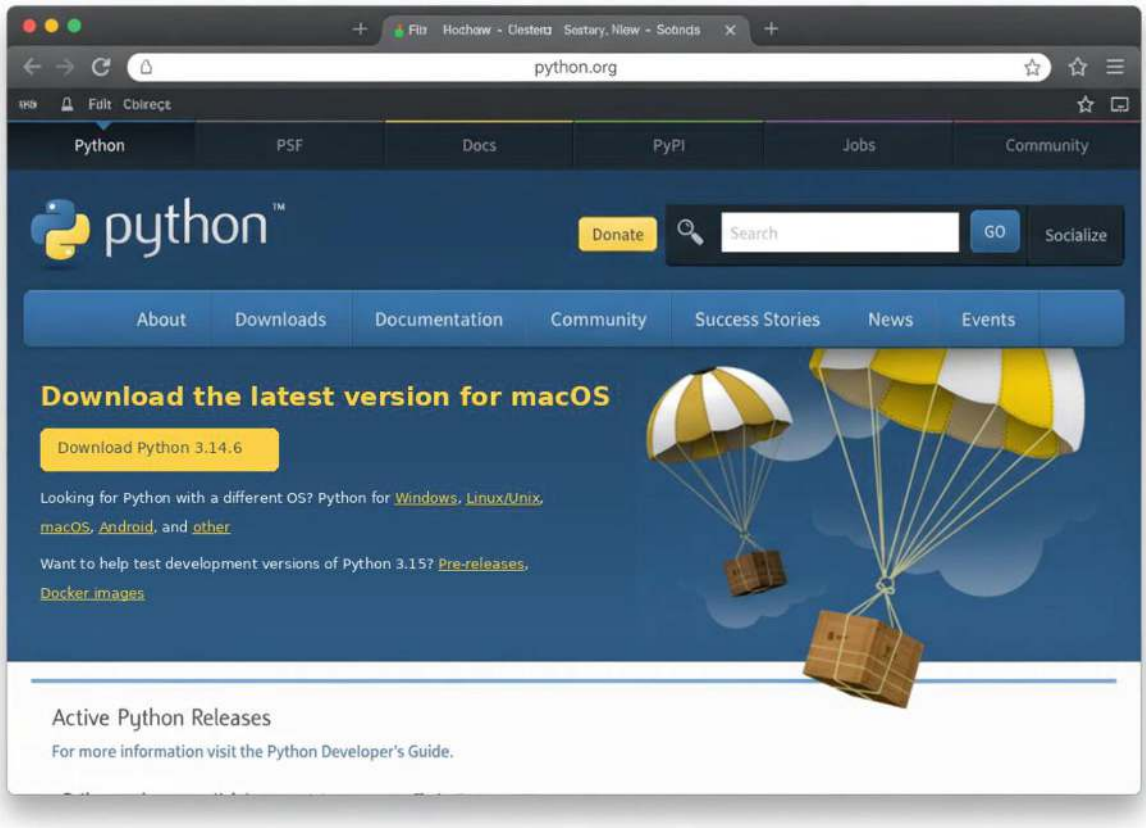


Figure 1-17 Python download page for macOS. The interface and version numbers may change by release. Accessed on July 16, 2026

Once the download is complete, you'll need to run the installer and follow the on-screen prompts, which include agreeing to the license and clicking "Install."



Figure 1-18 Python installer window for macOS

Accept the software license terms and allow the installation to continue by providing the access password.

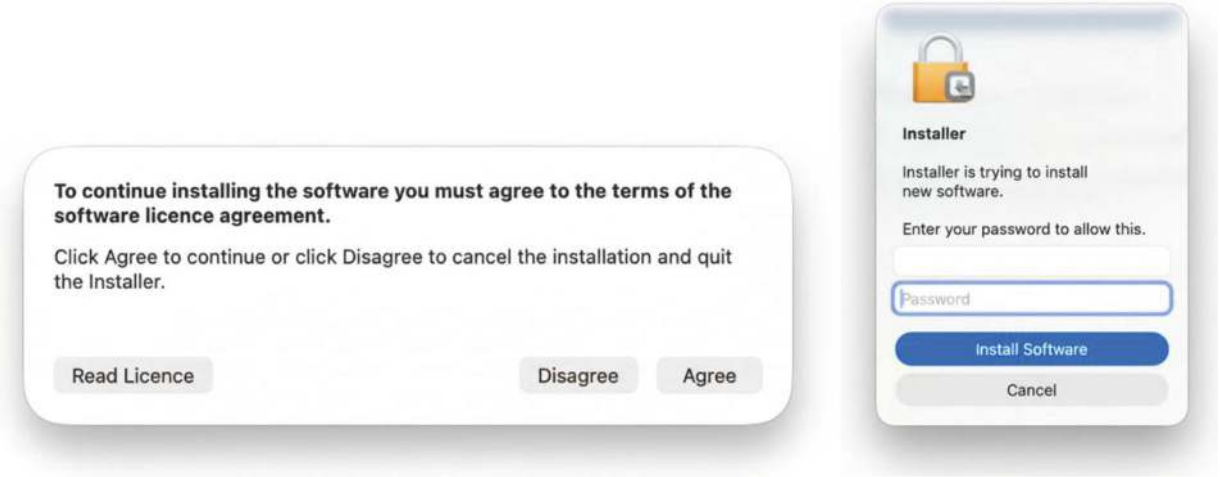


Figure 1-19 Software license terms window and software installation permission request

After completing the entire installation process, a completion message will be displayed in the installer window.



Figure 1-20 Python installer window indicating installation completion on macOS

When you click "Close," a new window will open asking if you want to move the installer contents to the trash. Click "Move to Bin," and the installer file will be deleted to free up space on your computer. The installed Python program will continue to function normally.

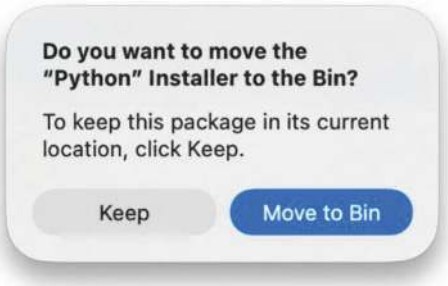


Figure 1-21 Window asking if you want to delete the installation file you used

Installing Python on macOS via Terminal

Check if Python is already installed.

The Terminal installation notes in this section are based on the official Python documentation at https://docs.python.org/3/using/mac.html, accessed on July 16, 2026. Check the current documentation because installation guidance may change between macOS and Python releases.

Do not assume that a current Python installation is already available on macOS. Open Terminal and check with

```
which python3
```

If the command returns a path such as /usr/bin/python3, a Python executable is present; check its version and remember that it may be managed by macOS developer tools.

Install command-line tools required by Homebrew and compilers:

```
xcode-select --install
```

Install Homebrew, a package manager for macOS.

Homebrew is a common package manager for installing and managing development tools on macOS:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

After installation, update Homebrew:

```
brew update
```

Install Python 3 with Homebrew:

```
brew install python
```

This will install the latest Python 3 version.

You can check it with

```
python3 --version
```

Install and update pip (Python package manager).

pip is included with modern Python versions, but you can ensure it's up to date:

```
python3 -m pip install --upgrade pip
```

Installing Python on Linux

Given the wide variety of Linux distributions available today, it is difficult to cover all of them in detail without making the topic overly brief. Therefore, we will use Ubuntu as our example for installation and updating. Ubuntu is not only widely adopted in academic settings but also highly popular in professional environments, making it a practical choice for engineers and developers. Its large community, extensive documentation, and long-term support releases ensure that users can reliably install, update, and maintain Python and other development tools with minimal friction.

Check the current Python version.

Before installing, it is important to check which version is already installed:

```
python3 --version
```

Or on some distributions:

```
python --version
```

On many Linux systems, the command "python" may be unavailable or may refer to a different version. Use "python3" to explicitly invoke Python 3.

Update your system's package list.
Ubuntu:

```
sudo apt update
sudo apt upgrade
```

Install Python.
Ubuntu:
To install the latest version available in the repositories:

```
sudo apt install python3 python3-pip
```

To install a specific version, you can use the deadsnakes PPA:

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.11  # replace with the desired version
number
```

To install a specific version:

```
sudo dnf install python3.11
```

The Python Virtual Environment

A virtual environment is an isolated directory that contains a specific Python installation along with its own libraries and packages. This tool is fundamental for organized and efficient Python development, as it resolves common issues related to dependencies and version conflicts.

The main purpose of a virtual environment is to create a contained, independent space for each project. These environments can even use different Python versions.

If you install these libraries globally on your system, one version will overwrite the other, causing conflicts and breaking one of your projects. A virtual environment solves this problem. By creating a separate environment for each project, you can have version 1.1 of the library installed in Project A's environment and version 2.9 installed in Project B's environment without one interfering with the other.

Virtual environments keep your development workspace organized by isolating project-specific packages and leaving the global Python installation clean. They also make collaboration and portability easier, ensuring the project runs consistently everywhere.

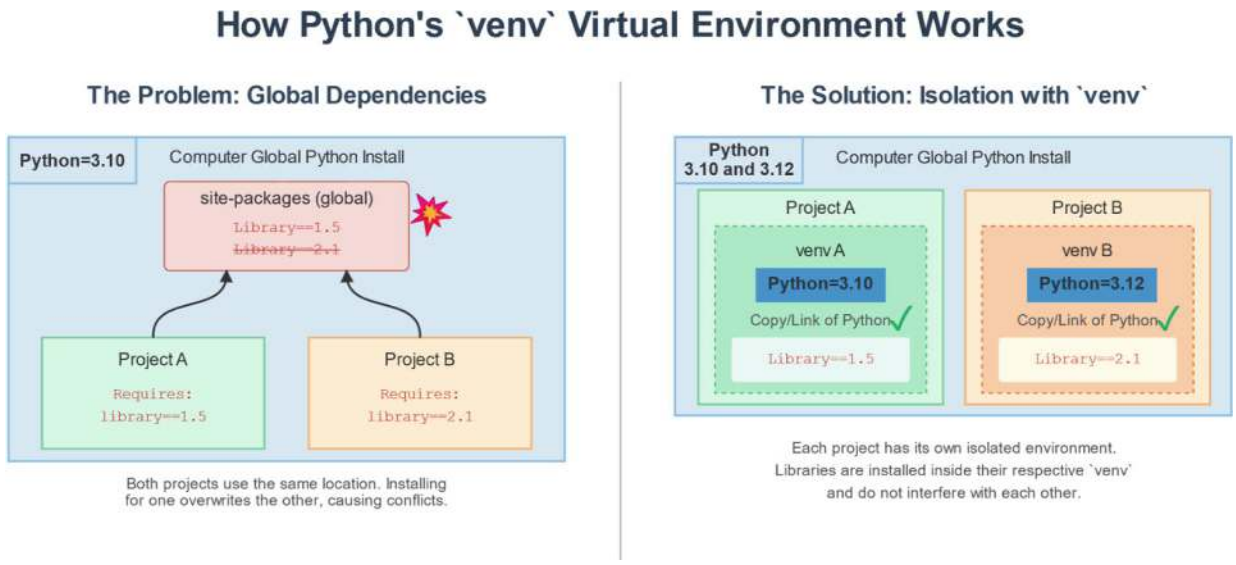


Figure 1-22 Graphical comparison between Python's global operating mode and isolated virtual environment mode

How Does venv Work?

When you create a virtual environment with venv, it essentially does the following:

- Creates a new directory to store all the files for the virtual environment. By convention, it is usually named venv or .venv.
- Sets up the Python executable by copying or creating a symbolic link to the Python interpreter that was used to create the environment.
- Creates a folder structure to hold installed packages (site-packages), activation scripts, and other necessary files.
- When you activate the environment, your system starts using the Python interpreter and packages from within that isolated directory instead of the global ones. In practice, this works by temporarily modifying your terminal's PATH environment variable.
- When you deactivate the environment, everything returns to normal.

Creating a Virtual Environment

There are several options for creating a virtual environment in Python. In this book, we'll use the venv module to create virtual environments because it's simple to use and already built into the Python installation, eliminating the need for additional installation. Using it is simple and involves three main steps: creation, activation, and package management.

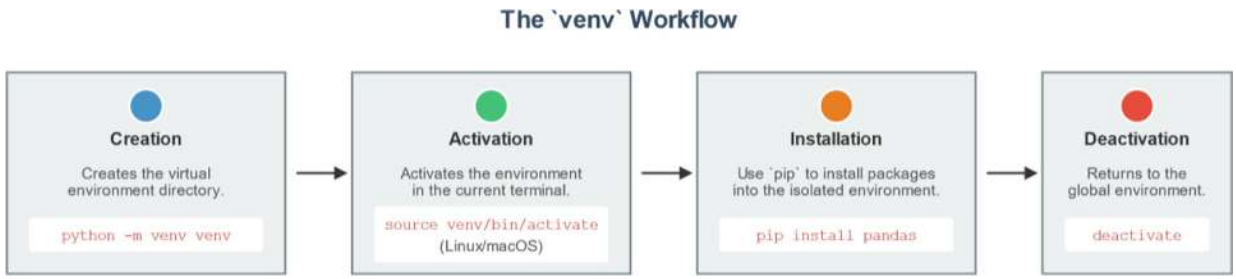


Figure 1-23 Cycle of creating and activating a Python virtual environment using the venv module from the standard library

1. *Creating the Virtual Environment*
   Open your terminal or command prompt, navigate to your project folder, and run

   ```
   python -m venv name_of_environment
   ```

   The -m instruction makes Python search for and execute some module, which in this case is venv.
   Or use

   ```
   py -m venv name_of_environment
   ```

   - python -m venv or py -m venv: Invokes the venv module
   - name_of_environment: The name of the virtual environment's directory, commonly just venv

   For example, the next command creates a folder named venv inside your project directory:

   ```
   py -m venv venv
   ```

   It is important to note that in some cases the command "python3" is required instead of "python".
2. *Activating the Virtual Environment*
   After creating it, you need to activate the environment. The command varies by operating system:
   Windows (Command Prompt or PowerShell):

   ```
   .\venv\Scripts\activate
   ```

   Linux or macOS:

   ```
   source venv/bin/activate
   ```

   When active, the environment's name usually appears at the beginning of your terminal prompt, for example:

   ```
   (venv) C:\Users\YourUser\Desktop\MyProject>
   ```

3. *Installing Packages*
   With the environment active, install packages using pip as usual. They will only be installed inside the virtual environment:

   ```
   pip install pandas requests
   ```

4. *Generating a requirements.txt File*
   To make the environment reproducible, generate a list of all installed dependencies:

   ```
   pip freeze > requirements.txt
   ```

```
# ...                    ...
```

Later, you or someone else can recreate the same environment with a single command:

```
pip install -r requirements.txt
```

5. *Deactivating the Virtual Environment*
When you're done working, deactivate the environment with

```
deactivate
```

Your terminal will then return to using the system's global Python interpreter and libraries.

Choosing a Code Editor

To start developing in Python, a simple text editor is sufficient. However, for greater efficiency and access to advanced features, an integrated development environment (IDE) is highly recommended. An IDE is a specialized software application that combines a wide range of tools into a single interface, enabling developers to write, debug, and test code more productively.

An IDE typically includes features such as

- *Code Editor*: A text editor with syntax highlighting, auto-completion, and code refactoring capabilities
- *Debugger*: A tool that allows you to navigate through code, set breakpoints, and inspect variables
- *Project Explorer*: A feature that helps you manage your project's files and directories
- *Version Control Integration*: Integration with version control systems such as Git

Table 1-4 highlights some of the most popular code editors and IDEs available today. Since the choice of development environment often comes down to personal preference, readers are encouraged to explore the options and select the one that best matches their workflow and needs. Over the course of his career, the author has worked with many IDEs, but currently relies primarily on PyCharm and Visual Studio Code. The latter, in particular, stands out for its extensive ecosystem of thousands of plugins that significantly extend its capabilities while also being open source and free to use.

Table 1-4 Comparison between the main code editors and IDEs available for the Python language

After selecting a code editor or IDE, it is advisable to consult the official website of the chosen tool for installation instructions and basic usage guidelines. A quick search in any reliable search engine will easily lead you to the official documentation and resources you need.

The First Program

In programming tradition, the first program you write in a new language is often one that simply displays the message "Hello, World!" on the screen.

Creating this program is straightforward. Open your code editor or IDE, create a new file named hello.py, and type the following line of code:

```
print("Hello, World!")
```

The print() function is one of the most commonly used commands in Python. It outputs text or data to the screen. When you run this program, Python will display the message Hello, World!. The word “print” is a built-in function provided by Python, and “Hello, World!” is a string, which means a sequence of characters enclosed in quotation marks. To print the text, you must enclose it in single ‘ or double “ quotes.

Running a simple program like this is essential to verify that Python and the IDE you are using are working correctly. If the message appears in the console, it confirms that Python is installed and working correctly.

Congratulations, this is your first Python program!

It is important to note that in programming, a function is a block of code that performs a specific task or set of tasks; functions will be explained comprehensively in a dedicated chapter.

More About the print Function

The print() function in Python can be customized using keyword arguments. Below are some useful resources available.

Newline (\n)

Using the escape sequence \n inside a string inserts a newline character, which forces the text to break and continue on the next line:

```
print("Hello,\nWorld!")
# Output:
# Hello,
# World!
```

Separator (sep)

The sep keyword defines the string used to separate multiple items. By default, items are separated by a space. To use a different separator, provide multiple items and specify the separator:

```
print("Hello", "World", sep="---")  # Output: Hello---World
```

Line Ending (end)

The end keyword specifies what is printed at the end of the output. By default, it is a newline character (\n):

```
print("Hello World", end="--") # Output: Hello World--
```



For a complete list of escape characters you can use with the print() function, see the official Python documentation.: https://python-reference.readthedocs.io/en/latest/docs/str/escapes.html.

Adding Symbols and Emojis

In Python, symbols can be created using Unicode escape sequences, an international standard for character encoding. For example, the \u prefix is used with four-digit hexadecimal codes to insert special characters or emojis directly into terminal messages with the print function. The Unicode standard is extensive, with over 140,000 defined characters, including symbols and emojis.

Python supports multiple escape formats to represent Unicode characters in strings. See the three main formats in Table 1-5.

Table 1-5 Comparing Unicode characters

The next example code demonstrates how to use Unicode escape sequences to display engineering symbols and emojis:

```
# Displaying engineering symbols using Unicode
print("Delta (Δ):", "\u0394")  # Output: Delta (Δ): Δ
print("Ohm (Ω):", "\u03A9")    # Output: Ohm (Ω): Ω
print("Micro (µ):", "\u03BC")  # Output: Micro (µ): µ
print("Approximately equal (≈):", "\u2248")  # Output:
Approximately equal (≈): ≈
print("Degree (°):", "\u00B0")  # Output: Degree (°): °

# Displaying emojis using Unicode
print("Robot:", "\U0001F916")  # Output: Robot:
print("Gear:", "\u2699\uFE0F")  # Output: Gear: ⚙
print("Lightbulb:", "\U0001F4A1")  # Output: Lightbulb:

# Example usage in a string
temperature = 25
unit = "\u00B0C"
print(f"The temperature is {temperature} {unit} \u2600\uFE0F")
# Output: The temperature is 25 °C ☀
```

The code output is shown in Figure 1-24.

```
c:/Users/carlo/Downloads/test_Python/main.py
Delta (Δ): Δ
Ohm (Ω): Ω
Micro (µ): µ
Approximately equal (≈): ≈
Degree (°): °
Robot: 🤖
Gear: ⚙
Lightbulb: 💡
The temperature is 25 °C 🌡
PS C:\Users\carlo\Downloads\test_Python>
```

Figure 1-24 Console output using Unicode characters

Table 1-6 highlights some examples that are particularly useful in the field of engineering.

Table 1-6 Unicode examples

Getting Keyboard Data

While displaying static text is a great start, programming becomes much more powerful when you can interact with the user. The next step is to learn how to receive user input via the keyboard using the input() function. This function pauses your program and waits for the user to type something and press the Enter key. The text the user types is then returned by the function as a sequence of characters, which you can store in a variable.

It's common practice to provide a helpful message inside the input() parentheses to prompt the user for the kind of information you want.

```
name = input("What is your name? ")
print("Hello,", name)
```

In this example, the program will first display "What is your name?". After the user types their name and presses Enter, their input is stored in the name variable. The program then proceeds to print "Hello," followed by the name that was entered.

The Importance of Structure: Indentation

One of Python's biggest differences from other programming languages is the way it groups statements, creating small blocks of code that perform a specific task, such as counting data or performing mathematical calculations. While most languages use curly braces or keywords to denote code blocks, Python uses an indentation-based system to define its code structure.

This means that the number of spaces or tabs at the beginning of a line is not just a matter of style but a syntax requirement that affects code interpretation.

Indentation is generally done using four spaces at the beginning of the line, and you must be mindful of the indentation levels, which is essential to avoid syntax errors and ensure correct code execution.

Throughout this book, necessary indentation in code will be discussed as needed.

Formatting Your Output with F-Strings

One of Python's most convenient features is the use of f-strings, or formatted string literals, which make it easy to embed variables directly into text without relying on complex string concatenation.

Although you can combine strings with the + operator, a more modern, readable, and efficient approach is to use f-strings. An f-string is simply a string literal prefixed with the letter f or F. Within an f-string, you can embed variables or expressions by placing them inside curly braces ({ }). Python will automatically evaluate the content inside the braces and replace it with its corresponding value.

This feature makes your print statements and your code in general much cleaner, more expressive, and easier to maintain.

Let's improve our previous input example using an f-string:

```
name = input("Please enter your name: ")
age = input("Please enter your age: ")
print(f"Hello, {name}! You are {age} years old.")
```

When you run this code, the values you entered for name and age will be seamlessly inserted into the final sentence. The expression inside the curly braces is evaluated in real time and converted to a string, making f-strings a powerful and flexible tool for creating dynamic output.

Figure 1-25 is the output of the code using f-string.



Figure 1-25 Screen with the result of the code using f-string

As a widely used feature among developers, f-strings have undergone several improvements in recent Python versions, including performance improvements. Starting with version 3.12, code enclosed in curly braces {} in f-strings has become more flexible, with some improvements: the use of the same quotation marks inside and outside the braces, the use of multi-line expressions, and the ability to add comments. It's also now possible to use backslashes (\) like any other normal Python string.

Let's look at the example that uses the same quotes inside and outside the braces. This feature makes it easier, for example, to use a type of dataset called a dictionary inside an f-

string without generating a SyntaxError.

```
my_dict = {'key': 'value'}
print(f"Accessing the value: {my_dict['key']}")
```

Before Python 3.12, any logic needed to be done outside of the f-string. Now, simple logic can be done right within the text. As an example, the code below determines a person's status based on their age using a conditional expression within an f-string. It checks whether the age is 18 or older and labels the person as "Adult" or "Minor" accordingly. For an age of 30, the code returns "Status: Adult." This approach allows for direct embedding of logic into the string formatting, making the code concise and readable.

```
age = 30
print(f"""Status: {
    'Adult'
    if age >= 18
    else 'Minor'
}""") # Output: Adult
```

It is also possible to add comments to explain a calculation inside an f-string. This is very useful in engineering applications, as it is great for documenting a complex calculation.

```
price = 70
print(f"Final price: {price * 1.15}  ( Adds a 15% tax )")
#Output: Final price: 80.5  ( Adds a 15% tax )
```

Formatting Your Output with T-Strings

Python version 3.14 added a new feature similar to the f-string: the t-string.

T-strings, or template strings, use the lowercase t prefix and resemble f-strings syntactically, but they produce a Template object instead of a final string.

Unlike an f-string, which immediately creates a final string, a t-string creates a Template object that preserves the fixed text, the evaluated interpolation values, and metadata about each interpolation. Applications can inspect or transform these components before producing final output. A Template has three main components:

- *Strings*: Fixed parts of the text that don't change.
- *Values*: Evaluated values of the expressions within {}.
- *Interpolations*: These are like "information boxes" created for each part between curly braces {} used in the t-string. Inside each box is information about the expression you wrote, the result it generated, whether you requested any data conversion, and the chosen format (such as the number of decimal places).

Because t-strings preserve static text separately from interpolated values, libraries can inspect, validate, or escape those values before rendering structured output. This does not make every t-string automatically secure, but it supports safer templating workflows when the renderer applies the required validation and escaping rules.

The example code demonstrates how t-strings work. The variable tpl is created with t"Hello {name}, the price of the product is: {price:.2f}", which produces a Template object rather than a regular string. The fixed text segments are stored in tpl.strings, the evaluated values are available through tpl.values, and tpl.interpolations contains objects describing each expression, its value, any conversion, and its format specification. A renderer can use this information to produce the final string in a controlled way.

The :.2f notation is a format specification: the colon introduces the specification, .2 requests two decimal places, and f selects fixed-point formatting. In an f-string it is applied immediately; in a t-string it is stored with the interpolation so that a renderer can apply it later. For example, a renderer applying .2f to 600.5 can produce 600.50.

```
name = "Carlos"
price = 600.5

tpl = t"Hello {name}, the price of the product is: {price:.2f}"

print(tpl)
# Template(strings=('Hello ', ', the price of the product is: ',
''),
#          interpolations=(Interpolation('Carlos', 'name', None,
''),
#                          Interpolation(600.5, 'price', None,
'.2f')))
print(tpl.strings)  # ('Hello ', ', the price of the product is: ',
'')
print(tpl.values)   # ('Carlos', 600.5)
print(tpl.interpolations)
# (Interpolation('Carlos', 'name', None, ''),
#  Interpolation(600.5, 'price', None, '.2f'))
```

For a more direct comparison, see the table below.

Table

Use f-strings when the goal is to produce a final formatted string immediately. Use t-strings when an application needs to inspect, validate, escape, or otherwise process interpolated values before rendering structured output.

Table of Useful Commands: Windows and macOS/Linux

Table 1-7 presents commonly used commands for both Windows and macOS/Linux. By providing a side-by-side comparison, it helps readers quickly reference and recall these commands during everyday programming tasks. To this end, it's important to highlight the following information.

Table 1-7 List of useful commands for Windows, Linux, and macOS

 On macOS and Linux, use “python3” when you need to explicitly select Python 3; the unqualified “python” command may be unavailable or may refer to another interpreter. Use “python3 -m pip” to install packages for that same interpreter.

Python Project Management with UltraViolet (UV)

The Python community is constantly evolving and seeking ways to make the language more efficient. Among the recent innovations is UltraViolet (UV), a unified tool for full Python project management. UV provides modern, high-performance features for installing packages, creating virtual environments, managing multiple Python versions, and running scripts in isolation, all with exceptional speed. UV is a command-line tool, used through the terminal (such as command prompt, PowerShell, or any system shell), which makes it easy to integrate into development workflows. As a complementary tool to Python, users are not required to adopt UV in every project; however, its capabilities offer excellent productivity benefits, especially in medium- and large-scale projects.

Thanks to its broad approach, UV can replace several traditional tools commonly used in everyday Python development—for example, pip, used to install packages and libraries; venv, responsible for creating virtual environments that isolate each project’s dependencies; pipx, used to install and run Python command-line tools globally while keeping them isolated; poetry install, a command frequently used to install dependencies in projects managed with Poetry; and even parts of Conda, a package and environment manager widely used in data science.

UV brings together features that would normally require all these tools separately, simplifying workflows and reducing the complexity involved in setting up the development environment.

Although it shares some similarities, Poetry is a tool focused on managing dependencies and organizing Python projects, but it does not offer all the capabilities provided by UV. It combines functions similar to pip for package installation and venv for automatic creation of virtual environments, in addition to structuring projects with configuration and control files. However, UV covers a broader scope, including Python version management and isolated script execution with superior performance.

Developed by the company Astral and written in Rust, UV benefits from the characteristics of this language, known for its memory-safety features and for delivering performance comparable to C and C++. In many scenarios, Rust has been adopted specifically to replace these languages in low-level implementations. The primary goal of UV is to harness Rust’s performance to provide an extremely fast, reliable, and modern experience for Python developers.

Installing UV

UV can be installed easily on any operating system. On Linux and macOS, you can use the official installer through the terminal using curl or wget, or, in the case of macOS, choose to install it through Homebrew. On Windows, installation is done via PowerShell by running the official script provided by Astral. As an alternative on all platforms, UV can also be installed

with pipx or pip, although these options may require compilation if prebuilt binaries are not available.

These different methods ensure that UV can be set up quickly and conveniently in any development environment. In any case, it is important to check the official documentation for possible updates to the installation process. You can do this by visiting https://docs.astral.sh/uv/getting-started/installation/.

To install UV on Linux, macOS, or Windows, follow the steps below according to your preferred installation method and operating system. After installation, it is always recommended to verify that everything is working correctly by checking the installed version of UV. You can do this with the command

```
uv --version
```

UV Installation via a Terminal

1. For any Linux distribution or macOS:
   Open a terminal and run this command using curl:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

If you don't have curl, you can use wget:

```
wget -qO- https://astral.sh/uv/install.sh | sh
```

To install globally with superuser permission:

```
curl -LsSf https://astral.sh/uv/install.sh | sudo sh
```

Verify the installation:

```
uv --version
```

After running this command, the installed UV version should appear.

2. For macOS with Homebrew:
   If you use Homebrew, you can install UV directly through it:

```
brew install uv
```

Then confirm the installed version:

```
uv --version
```

3. For Windows:

Open PowerShell as administrator and run the following command:

```
powershell -ExecutionPolicy ByPass -c "irm
```

```
https://astral.sh/uv/install.ps1 | iex"
```

After installation, check the version:

```
uv --version
```

If the version appears, everything is ready.

To update the UV later, use the command below, which works for all operating systems:

```
uv self update
```

Alternative Installation via pip/pipx

Another practical option is to install UV using pip or pipx. In this case, the installer typically downloads a file called a wheel, which is a precompiled package. This wheel already contains the program ready to use, without requiring any compilation.

However, if your platform or architecture does not have a wheel available, for example, very new operating system versions, uncommon ARM machines, or highly specific Linux distributions, then Python must compile UV directly from the source code. In such cases, it is recommended to check the UV documentation for the necessary steps and prerequisites.

Keep in mind that this does not happen on most common systems such as Windows, macOS, and popular Linux distributions like Ubuntu, because these already have official wheels available. Compilation is required only in specific scenarios.

If you already have a Python environment, you can install UV with pipx:

```
pipx install uv
```

Or with pip:

```
pip install uv
```

After installation, check the version:

```
uv --version
```

Using UV for Dependency Management

The uv tool manages the virtual environment in a fully automatic, integrated, and intelligent way, eliminating the need to manually create, activate, or maintain a venv. Whenever a project requires dependencies, for example, when you use uv add or execute any command with uv run, uv checks whether a virtual environment already exists; if not, it immediately creates a managed virtual environment inside the .venv directory, using the appropriate Python version defined in the pyproject.toml or the .python-version file. This environment is kept perfectly synchronized with the lockfile (uv.lock): when running uv run, uv automatically verifies whether there are missing or extra packages and installs, removes, or updates everything before executing the command, ensuring full consistency. As a result, you never need to activate the environment manually—there is no source venv/bin/activate, because uv always runs commands inside the correct environment. Additionally, uv can automatically recreate the environment if it becomes corrupted, and it supports temporary environments that are

created, used, and discarded to execute isolated scripts. This means uv provides a reproducible, fast, and fully self-managed virtual environment system, integrating dependency management, Python versions, and command execution into a unified and highly reliable workflow.

Main Commands in the uv Workflow

1. *Starting a New Project*

```
uv init <name>
```

This command creates the entire initial structure of a Python project, including

- Creating a folder with the specified name
- Creating a pyproject.toml file containing
  - The project name
  - The initial version (0.1.0)
  - An empty dependency list
  - Build system configuration
- Creating a basic .gitignore file
- Preparing the structure for a managed virtual environment
- Making the project ready to receive other uv commands such as uv add and uv run

Result:

```
my_project/
  ├── pyproject.toml
  ├── README.md
  └── .venv/      (created automatically when you add something)
```

2. *Adding and Installing Dependencies Automatically*
   The command below installs the specified package into your environment and automatically adds it to your project's dependency files:

```
uv add <package>
```

This command does two things at the same time:

1. Updates the pyproject.toml
   It adds the requested package to the dependency list.
   In this example, we will install the Matplotlib library, a popular library for creating graphs and visualizations in Python:

```
uv add matplotlib
```

This generates inside pyproject.toml

```
dependencies = ["matplotlib"]
```

2.
Creates or updates the project's virtual environment

- If .venv/ does not exist yet, uv automatically creates it.
- Installs the package with exact versions.
- Resolves dependencies much faster than pip or poetry.

The command to remove a package using uv is
uv remove <package>
To remove the Matplotlib library from your project and its dependency records, you would run

```
uv remove matplotlib
```

3.
Executing Commands Within the Isolated Environment
The command below executes a command, script, or library within your project's isolated environment:

```
uv r.un <command>
```

This is the core of uv's modern workflow.

```
uv run python s.cript.py
```

What uv does automatically is

1.
Ensures the virtual environment exists
If .venv does not exist, it creates it immediately.
2.
Checks if dependencies are synchronized with the lockfile

- If packages are missing → installs them immediately
- If extra packages exist → removes them
- If nothing changed → simply runs the command

3.
Uses the correct Python version
If you previously ran

```
uv python pin 3.11
```

uv uses that version.
Executing the command within the environment, for example:

```
uv run python      # runs the Python interpreter inside .venv
uv run pytest      # runs pytest installed in the project
uv run myapp       # runs a CLI app installed as a dependency
```

This means you never need to activate the venv manually.

4. *Synchronizing the Environment with the Lockfile*
   This command makes the .venv/ environment identical to the lockfile:

```
uv sync
```

   What it does is

- Installs missing packages
- Removes packages not listed in the lockfile
- Adjusts versions to exactly match the lockfile
- Ensures your machine has the same environment as another developer

   It's very useful when

- You cloned a project with pyproject.toml + uv.lock.
- Someone added or removed dependencies.
- You switched Python versions.

   Typical Example:

```
git clone project
cd project
uv sync
# done — environment identical to the original author's
```

   5. Generating or Updating the Lockfile

```
uv.lock
```

   This command forces the creation or update of

```
uv.lock
```

   It contains all dependencies, including subdependencies with explicit version numbers.
   The lockfile guarantees full reproducibility and prevents "dependency breakage" when reinstalling packages.
   Use it when

- You manually edit the pyproject.toml.
- You switch Python versions.
- You want to guarantee that everything is consistent.

About the Lockfile

uv.lock is the lockfile automatically generated and maintained by the uv package manager to guarantee fully reproducible Python environments. It records the exact versions, dependency trees, hashes, and metadata of all packages both direct and indirect that your project uses. Whenever you run commands like uv add, uv lock, or uv sync, uv resolves dependencies and updates this file so that any machine can recreate the same environment with complete accuracy. The uv.lock file ensures consistency, avoids version conflicts, and acts as the definitive source of truth for installing the correct package versions in the project's virtual environment.

uv automatically generates or updates the file:

```
uv.lock
```

This file stores exact versions, ensuring that someone else can reproduce your environment identically.

Table 1-5 presents the main uv commands.

Table

Practice Exercises

Just like in any field of study, practice is essential for reinforcing the knowledge gained in programming. Use the exercises below to strengthen the concepts introduced in this chapter, applying each idea gradually and thoughtfully.

1. Analyze the code below and describe what this program does.

```
name = input("What is your name? ")
age = input("How old are you? ")

print("\n=== User Information ===")
print("Name:", name)
print("Age:", age)
print("Welcome,", name + "!")
```

2. Create code that prints two lines of text using only one output statement, taking advantage of the newline escape character \n.
3. Analyze output using sep and end in print().

```
print("Python", "is", "fun", sep="-")
print("Version:", end=" ")
print("3.12")
```

4. Find out which Python version is installed on your system and record that version as a comment at the top of your file.
5. Write a program that asks for the user's name and shows a personalized greeting.
6. Complete the code below, correctly using the f-string features.

```
# Ask for user data
name = input("Please enter your name: ")
age = input("Please enter your age: ")

# Display result using an f-string
```

```
message = f"Hello, {____}! You are {____} years old."
print(____)
```

7. Use formatting parameters in the print function to separate values with a chosen symbol.
8. Analyze and fix indentation error.

```
print("Welcome!")
print("This program will ask for your name.")

name = input("Enter your name: ")

print("Nice to meet you,", name)
  print("Let's begin the tutorial!")
print("Python is fun!")
```

9. Rewrite the previous exercise by asking for the user's name and age again, but this time display the final message using an f-string.
10. Write a short Python program that displays a temperature value on the screen and includes the degree symbol (°) by using the appropriate Unicode escape in the string.
11. Modify the code below and add the degree symbol and emojis.

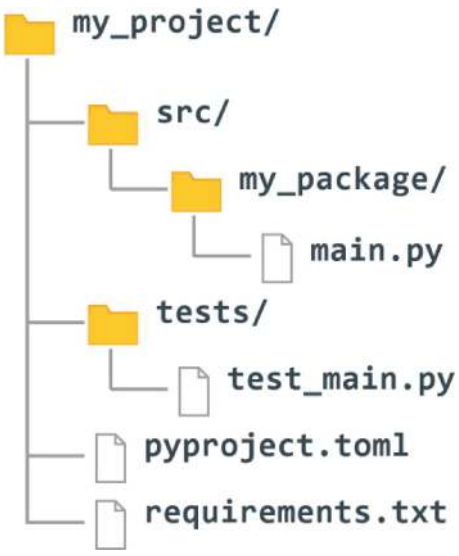
```
temperature = 25
print("The temperature is", temperature, "C")
```

12. Create a code that displays a sentence on the screen containing emojis using their respective Unicode codes.
13. Create a short program that asks the user for their name and then displays a friendly message that includes Unicode characters (such as emojis or special symbols). Choose at least one Unicode symbol and include it in the final output.
14. Complete the code with emoji and username.

```
name = input("What is your name? ")
rocket = "____" # Unicode code for the rocket emoji
message = f"Nice to meet you, {____}! {____}"
print(message)
```

15. Demonstrate two different ways of formatting text and explain which one is clearer.
16. Classify the following filenames into their correct categories based on the file types introduced in this chapter: .py, .txt, .json, and .md.

17. Analyze project structure: where does each file belong?



18. Show where new modules, files, and tests should be placed in a basic Python project layout.
19. Create a virtual environment, activate it, and briefly explain why virtual environments matter.
20. Summarize the Python installation process for your operating system and explain what the PATH variable is.
21. Choose two beginner-friendly PEPs and explain their purpose.
22. Briefly describe what happens internally when Python runs a file.
23. Explain the difference between template strings (string.Template) and f-strings, including situations where templates are safer.
24. Complete the code—concatenation vs. f-strings.

```
name = "Alex" age = 30

# Version 1: concatenation
text_concat = "Name: " + name + ", Age: " + str(age)

# Version 2: f-string (incomplete)
text_fstring = f"Name: ____, Age: ____"

print(text_concat)
print(text_fstring)
```

25. Fill in the commands.
Create and activate a virtual environment.

```
python -m venv _______

# Activation
# Windows:
_______/Scripts/Activate.ps1

# Linux/macOS:
source _______/bin/activate
```

Answer Key

1.
Analyze the code and describe what the program does.

```
name = input("What is your name? ")
age = input("How old are you? ")

print("\n=== User Information ===")
print("Name:", name)
print("Age:", age)
print("Welcome,", name + "!")
```

This program asks the user for their name and age using the input() function. After collecting the information, it prints a formatted section labeled “User Information” displaying the values entered, followed by a personalized welcome message that includes the user’s name. This demonstrates how to read input from the keyboard and then output personalized text.

2.
Create code that prints two lines using \n.

```
print("First line\nSecond line")
```

This program uses the newline escape sequence \n, which forces the text after it to appear on the next line. Even though only one print() function is used, the escape sequence allows the output to appear as two separate lines.

3. Analyze output using sep and end.

```
print("Python", "is", "fun", sep="-")
print("Version:", end=" ")
print("3.12")
```

In the first line, the sep parameter replaces the default space with a hyphen, producing the output Python-is-fun. In the second case, the end parameter replaces the default line

break with a space, causing the next print operation to appear on the same line, resulting in Version: 3.12.

4. Write the Python version as a comment.

```
import sys
print("Running Python version:", sys.version)
```

The goal of this task is simply to record the installed version of Python.

5. Program that asks for a name and greets.

```
name = input("What is your name? ")
print(f"Hello, {name}! Nice to meet you.")
```

This program reads the user's name and prints a personalized greeting. It demonstrates how to combine literal text with a stored variable to create dynamic output.

6. Complete the f-string.

Completed Version:

```
name = input("Please enter your name: ")
age = input("Please enter your age: ")
message = f"Hello, {name}! You are {age} years old."
print(message)
```

The completed script uses an f-string to embed variables directly inside a formatted message. The curly braces allow Python to insert the variable values automatically, making the code clean and easy to read.

7. Use formatting parameters in print.

```
print(10, 20, 30, sep=" | ")
```

Here the sep parameter changes how values are separated when printed, replacing the default space with " | ". This allows the output to appear more organized and readable.

8. Fix indentation error.

Corrected Version:

```
print("Welcome!")
print("This program will ask for your name.")

name = input("Enter your name: ")

print("Nice to meet you,", name)
print("Let's begin the tutorial!")
print("Python is fun!")
```

The original error occurs because one of the lines had unnecessary indentation, making Python expect a block structure that did not exist. Removing the extra spaces aligns the statements correctly and prevents the error.

9. Rewrite using an f-string.

```
name = input("Enter your name: ")
age = input("Enter your age: ")
print(f"Nice to meet you, {name}. You are {age} years old.")
```

This updated version prints the same message but uses an f-string instead of concatenation, making the code more readable and reducing the need for type conversions.

10. Display temperature with the degree symbol.

```
print("Temperature:", 24, "\u00B0C")
```

The Unicode sequence \u00B0 generates the degree symbol, which allows Celsius values to be displayed correctly. Python interprets the escape sequence and prints the symbol in the final output.

11. Modify code with emojis.

Final Version:

```
temperature = 25
print("The temperature is", temperature, "\u00B0C 🌞🌡️")
```

The improved version adds both the Unicode degree symbol and emojis directly in the string, making the output more expressive and visually meaningful.

12. Display emojis using Unicode.

```
print("Python is fun! \U0001F600 \U0001F680")
```

This program displays emojis by using their Unicode escape codes, which allows emoji usage even when they cannot be typed directly.

13. Program with a username and emoji.

```
name = input("What is your name? ")
print(f"Hello, {name}! \U0001F44B")
```

This code asks for the user's name and prints a personalized greeting with an emoji, demonstrating how to combine Unicode and f-strings to make output more engaging.

14. Complete the emoji code.

Solution:

```
name = input("What is your name? ")
rocket = "\U0001F680"
print(f"Nice to meet you, {name}! {rocket}")
```

The program stores an emoji in a variable and prints it alongside the user's name using an f-string, showing that Unicode symbols can be treated like regular text values.

15. Two formatting methods.

Example:

```
print("Name: " + "Ana")
print(f"Name: {'Ana'}")
```

Both outputs display the same text, but the f-string format is clearer and more scalable because it avoids manual concatenation and type conversions, especially when variables are involved.

16. Classify file types.

Answer:

.py files contain Python source code, .txt refers to simple plain-text documents, .json stores structured data using JavaScript Object Notation, and .md files are Markdown format commonly used for documentation.

17. Analyze project structure.

Answer:

In a basic Python project, source code files typically belong in a folder such as src/, test files belong in a tests/ directory, documentation (such as README.md) stays in the project root, and configuration or data files like .json are placed in dedicated subfolders such as config/ or data/.

18. Where modules and tests go.

Answer:

New Python modules should be placed alongside existing source code, while test files should be placed inside the test folder using matching names so that the structure remains consistent and easy to understand.

19. Create and activate a virtual environment.

Answer:

A virtual environment isolates project-specific dependencies from the global Python installation. After creating it with python -m venv, it must be activated so that any new packages install only inside the environment rather than affecting the entire system.

20. Summarize the installation process and PATH.

Answer:

Installing Python typically involves downloading the installer and enabling the option to add Python to the system PATH. The PATH variable tells the operating system where executable programs are located, allowing Python to be run from the terminal without specifying the full folder path.

21. Pick two beginner-friendly PEPs.

Answer:

PEP 8 defines style guidelines to ensure that Python code is consistent and readable, while PEP 20 summarizes the design philosophy behind Python, encouraging clarity, simplicity, and maintainability.

22. What happens when Python runs a file.

Answer:

When a Python script is executed, Python reads the file, compiles the code to bytecode, and runs that bytecode inside the Python Virtual Machine. During execution, variables, objects, and functions are created as needed and instructions are processed sequentially.

23. Template strings vs. f-strings.

Answer:

F-Strings embed evaluated values and expressions directly into a final string, making them concise and readable. Template strings also evaluate their expressions when the Template is created, but they retain the values, original expressions, conversions, and format specifications as structured data. This structure is useful when a renderer must validate or escape values before producing output.

24. Complete concatenation vs. f-string.

Solution:

```
name = "Alex"
age = 30
text_concat = "Name: " + name + ", Age: " + str(age)
text_fstring = f"Name: {name}, Age: {age}"
print(text_concat)
print(text_fstring)
```

Both approaches produce the same result, but the f-string version is easier to read and maintain, as it automatically handles type conversion and avoids lengthy string concatenation.

25. Fill in the virtual environment commands.

```
python -m venv venv
venv/Scripts/Activate.ps1
source venv/bin/activate
```

Answer:

These commands create and activate a Python virtual environment, ensuring that installed dependencies remain isolated from the system's global Python installation.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_2

2. Variables

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Introduction

All data in a Python program is stored in constants, variables, or collections of variables. When managing large amounts of data, such as a list containing thousands of store clients, it is common to use database systems.

In Python, constants don't inherently have immutable values, but resources exist to prevent reassignment, which will be explored in subsequent chapters.

Variables are temporary pieces of information, usually stored in RAM, which hold values during program execution. As in other programming languages, these variables are declared using the assignment operator (=).

Collections are data structures that store and manage multiple elements and will be introduced in this chapter and explored in more detail in the following chapters.

Variables and Types

In Python, variables are classified into types as shown in Table 2-1.

Table 2-1 Types of variables

The international IEEE 754 standard is used to represent floating-point numbers in computers, ensuring consistency between processors and programming languages. It stores numbers using a sign bit, an exponent, and a significand, commonly called a mantissa, allowing the representation of very large, very small, and fractional values, but with limited precision, which can lead to rounding errors.



The Python interpreter does not require explicit declaration of the type of variables.

In Python, it is not necessary to explicitly declare the type of a variable because Python is a dynamically typed language. Unlike languages like C or C++, the Python interpreter automatically determines and assigns the variable's type based on its value.

To determine the type of a variable assigned by the Python interpreter, use the type() method as shown in the example below:

```
name = "John"
age = 30
height = 1.75 # height in meters
print(type(name))  # Output: <class 'str'>
print(type(age))  # Output: <class 'int'>
print(type(height)) # Output: <class 'float'>
```

Integers and Floating Numbers

In Python, numbers can be represented using two types, mainly int for integer values and float for floating-point numbers. These two types are used to manipulate numeric values, but they have distinct characteristics.

An int variable can be positive, negative, or zero and is used to represent a whole number without a decimal point. A float represents a real number that can have a fractional part or a decimal point, using finite binary precision. Float variables can also be positive, negative, or zero.

Large numbers can be represented using an underscore (_) as a visual separator to improve the readability. This change improves the readability of large numbers without altering their value, as shown in the following example:

```
# Using underscores for readability in large
numbers
annual_revenue = 1_000_000_000  # 1 billion
population_of_earth = 7_888_000_000  # 7.888
billion
distance_to_sun_km = 149_597_870  # 149,597,870 km

# Printing the values
print(f"Annual Revenue: {annual_revenue}")  #
Output: Annual Revenue: 1000000000
print(f"Population of Earth:
{population_of_earth}")  # Output: Population of
Earth: 7888000000
print(f"Distance to the Sun: {distance_to_sun_km}
km")  # Output: Distance to the Sun: 149597870 km
```

Underscores can also be used in non-decimal number systems such as binary, hexadecimal, and octal to improve readability:

```
# Binary, hexadecimal, and octal numbers
binary_value = 0b1010_1011_1100_1101  # Binary
representation
hex_value = 0xFF_FF_FF  # Hexadecimal
representation
octal_value = 0o12_345  # Octal representation

# Printing the values
print(f"Binary Value: {binary_value}")  # Output:
Binary Value: 43981
```

```
print(f"Hexadecimal Value: {hex_value}")  #
Output: Hexadecimal Value: 16777215
print(f"Octal Value: {octal_value}")  # Output:
Octal Value: 5349
```

Floating-Point Arithmetic Error

 Operations with float numbers can generate rounding errors and data loss.

Computers use binary floating-point arithmetic to represent decimal numbers. Most decimal fractions cannot be represented exactly in binary, which leads to small rounding errors. In this example, the numbers 0.7 and 0.1 cannot be represented exactly in binary, so their sum is also affected by this imprecision. See the example code:

```
a: float = 0.7
b: float = 0.1
c= a + b
print(c)
# Output: 0.7999999999999999
```

The output 0.7999999999999999 instead of the expected 0.8 is due to the way floating-point numbers are represented in computers.

The round() function can be used when a rounded result is sufficient, although it does not make the stored binary value exact. For applications that require exact decimal representation, use the decimal module. We will use decimal in the chapters on advanced mathematical operations.

Mathematical and Utility Operators

Arithmetic Operations

Arithmetic operations in Python are useful for basic calculations. The // operator is recommended when decimal results are unnecessary, avoiding the creation of floats. The modulus % is commonly used to check divisibility, such as verifying if a number is even. Exponentiation with ** should be applied carefully when dealing with large numbers, since it can consume a lot of memory.

```
# 1- Arithmetic Operations
x = 42
y = 7
# Basic mathematical operations on integers
print("Addition (x + y):", x + y)          #
Output: 49 (Adds x and y)
print("Subtraction (x - y):", x - y)       #
Output: 35 (Subtracts y from x)
print("Multiplication (x * y):", x * y)    #
Output: 294 (Multiplies x by y)
print("Floor Division (x // y):", x // y) #
Output: 6 (Divides x by y and returns the largest
whole number less than or equal to the result)
print("Modulus (x % y):", x % y)           #
Output: 0 (Returns the remainder of x divided by
y)
print("Exponentiation (x ** 2):", x ** 2) #
Output: 1764 (Raises x to the power of 2)
```

Bitwise Operations

Bitwise operations are typically applied in low-level programming such as communication protocols, cryptography, and electronics. They allow direct manipulation of the binary representation of integers, making them very fast and efficient, though often less readable. For instance, shifting operators << and >> are especially useful for creating and managing bit masks.

```
# 2- Bitwise Operations (Integer-Specific)
# Perform bit-level manipulation
print("Bitwise AND (x & y):", x & y)      #
Output: 2 (Performs AND operation on binary
representations of x and y)
print("Bitwise OR (x | y):", x | y)       #
Output: 47 (Performs OR operation on binary
representations of x and y)
print("Bitwise XOR (x ^ y):", x ^ y)      #
Output: 45 (Performs XOR operation on binary
representations of x and y)
print("Left Shift (x << 1):", x << 1)  # Output:
84 (Shifts bits of x to the left by 1 position)
print("Right Shift (x >> 1):", x >> 1) # Output:
21 (Shifts bits of x to the right by 1 position)
```

Conversion Methods

Conversion methods like to_bytes and from_bytes are crucial in network programming and handling binary files. Always specify the endianness ('big' or 'little') to ensure correct interpretation of the data. The method .as_integer_ratio() for floats is very practical for representing numbers as exact fractions. Explicit conversions are recommended to avoid type errors and ensure clarity in the code.

```
# 3- Conversion Methods
# Convert integer to bytes and back
x = 42
bytes_val = x.to_bytes(2, 'big')  # Convert
integer to bytes

print("Integer to Bytes:", bytes_val) # Output:
b'\x00*'

print("Bytes to Integer:",
int.from_bytes(bytes_val, 'big'))  # Output: 42
```

```
# Conversion for floats
f = 3.14159  # A float example
print("Float as Integer Ratio
(f.as_integer_ratio()):", f.as_integer_ratio())
# Output: (3537115888337719, 1125899906842624)
(Represents f as a fraction)
```

Representation Methods

Representation methods such as .hex() are valuable when serializing floats or debugging their internal values. With float.fromhex(), it is possible to recover the exact float from its hexadecimal representation. These conversions are widely used in scientific computing, graphics, and low-level systems. As a best practice, always explain why hexadecimal is being used instead of decimal, to keep the code more understandable.

```
# 4- Representation Methods
# Convert float to hexadecimal and back

hex_val = f.hex()  # Convert float to hexadecimal
string
print("Float to Hexadecimal (f.hex()):",
hex_val)  # Output: '0x1.921f9f01b866ep+1'

print("Hexadecimal to Float
(float.fromhex(hex_val)):",
float.fromhex(hex_val))  # Output: 3.14159
```

Utility Methods

Utility methods make operations more direct and readable. The abs() function is useful for calculating distances, measures, or error values. The bit_length() method helps in fields like compression, cryptography, and data storage optimization. The method .is_integer() is a good way to check if a float contains an integer value before rounding. Using these

built-in methods is preferable to writing manual checks, as they improve clarity.

```
# 5- Utility Methods
# Perform utility operations on integers and
floats
print("Absolute Value (abs(-x)):", abs(-x))  #
Output: 42 (Returns the absolute value of -x)
print("Bit Length (x.bit_length()):",
x.bit_length())  # Output: 6 (Returns the number
of bits required to represent x in binary)
print("Is Integer (f.is_integer()):",
f.is_integer())  # Output: False (Checks if f is
an integer)
```

Built-In Functions

Built-in functions provide several useful numeric operations. The round() function rounds to a specified number of digits, divmod() returns the quotient and remainder together, and pow(x, y, z) performs modular exponentiation without first creating x ** y.

```
# 6- Built-in Functions
# Perform built-in operations
print("Rounded (f, 2):", round(3.1492653589, 2)) #
Output: 3.15 (Rounds f to 2 decimal places)

print("Divmod (x, y):", divmod(18, 3)) # Output:
(6, 0) (Returns a tuple with the quotient and
remainder)

print("Power (x ** y):", pow(3, 2)) # Output: 9
(Calculates x raised to the power of y)
```

The function modular exponentiation is a highly efficient way to calculate (base ** exponent) % modulus. In simple terms, it raises a

number to a power, divides the result by another number (the modulus), and gives you the remainder.

```
# Using the pow(x, y, z) function
print("Modular power:", pow(3, 4, 5))  # Output: 1

# --- For comparison, here is the manual
calculation ---
base = 3
exponent = 4
modulus = 5
# 1. First, calculate the exponentiation
power_result = base ** exponent  # 3**4 = 81
# 2. Then, calculate the remainder (modulus)
manual_result = power_result % modulus  # 81 % 5 =
1
```

Now look at Table 2-2 that summarizes all the mathematical and utility operations presented.

Table 2-2 Mathematical and utility operations

Making Comparisons

Comparison operators are widely used in decision-making structures, especially when a condition must evaluate to True or False. Ordering comparisons such as < and > require compatible types; for example, a string cannot be ordered directly against a float. Chained comparisons, such as 4 < y < 20, are also supported.

Python provides several comparison operators where the result is a boolean variable.

Table 2-3 Comparison operators

It's important to note that the symbol = is an assignment operator and == is a comparison operator. The “is” operator checks whether two variables point to the same object in memory, not just if their values are equal.

The codes below can demonstrate some application examples:

```
# Comparison Operations
# Comparing integers and floats
x = 4
y = 8
print("Equal (x == y):", x == y)                #
Output: False
print("Not Equal (x != y):", x != y)            #
Output: True
print("Less Than (x < y):", x < y)              #
Output: True
print("Greater Than (x > y):", x > y)           #
Output: False
print("Less or Equal (x <= y):", x <= y)        #
Output: True
print("Greater or Equal (x >= y):", x >= y)     #
Output: False
```

In Python, object identity and equality are two distinct concepts. When you assign a list to a variable, like x = [1, 2, 3], a new list object is created in memory. If you then assign y = x, both x and y point to the same list object in memory, meaning they have the same object identity. However, if you create a new list with the same content, like z = [1, 2, 3], it's a different object in memory, even though it has the same content as x.

The is operator checks for object identity, so x is y would return True because they point to the same object, but x is z would return False because z is a different object. On the other hand, the == operator checks for equality of content, so x == z would return True because

both lists have the same content, [1, 2, 3]. Understanding the difference between object identity and equality is crucial in Python programming, especially when working with mutable objects like lists and dictionaries.

```
# Object Identity
x = [1, 2, 3]
y = x  # y points to the same object as x
z = [1, 2, 3]  # z is a different object with the
same content

print(x is y)  # True
print(x is z)  # False
print(x == z)  # True (same content)
```

This code demonstrates how to check if an element is present in a list using the in and not in operators:

Membership Check with Strings

```
# Membership Check
fruits = ['apple', 'banana', 'cherry']

print('apple' in fruits)  # True
print('grape' not in fruits)  # True
```

Membership Check with Numbers

```
# Membership Check
numbers = [1, 2, 3, 4, 5]

print(3 in numbers)  # True
print(6 not in numbers)  # True
```

Working with Strings

Declaring Strings

A sequence of characters is called a string and is used to represent text. To declare a string, write text between single quotes ('), double quotes ("), or triple quotes (''' or """) for multi-line strings.

```
# Defining strings in Python
# Single quotes
single_quote_string = 'Hello'
print(single_quote_string)  # Output: Hello

# Double quotes
double_quote_string = "World"
print(double_quote_string)  # Output: World

# Triple quotes allow you to create multi-line
strings easily.
multi_line_string = '''This is a multi-line
string.'''
print(multi_line_string)    # Output:
                            # This is a multi-line
                            # string.

#Double quotes can include a single
quote/apostrophe.
quote_in_string = "It's a sunny day"
print(quote_in_string)  # Output: It's a sunny day
```

Immutability of Strings

It's important to note that strings are immutable, meaning that once a string is created, it cannot be changed, as shown in the example below:

```
s = "apple"
# Attempting to modify a single character
s[0] = "A"  # This would raise a TypeError

# Instead, you can create a new string
```

```
s = "A" + s[1:]
print(s)  # Output: "Apple"
```

Indexing and Slicing Strings

As shown in the example above, string elements can be accessed using indexing by placing the index between brackets:

```
s = "Apple"
print(s[0])  # Output: A (first character)
print(s[-1]) # Output: e (last character)
```

To slice a string, use the colon operator (:) between the start and end indexes, as shown in the example below:

```
s = "Apple"
print(s[0:3])  # Output: App (characters 0, 1, 2)
print(s[2:])   # Output: ple (from index 2 to end)
print(s[:3])   # Output: App (up to index 3,
exclusive)
```

String Operations

Python offers many methods to handle strings; the most used types of handling are listed in Table 2-4.

Table 2-4 Method for string operations

The code below lists examples of these important methods:

```
s = "apple, orange"

# 1 - Converting to uppercase or lowercase
print(s.upper())  # Output: APPLE, ORANGE
(converts all characters to uppercase)
print(s.lower())  # Output: apple, orange
(converts all characters to lowercase)
```

```
print(s.capitalize())  # Output: Apple, orange
(capitalizes the first character)
print(s.title())  # Output: Apple, Orange
(capitalizes the first character of each word)
print(s.swapcase())  # Output: APPLE, ORANGE
(swaps the case of each character)

# 2 - Finding and replacing
# Finding substrings
print(s.find("orange"))  # Output: 7 (index where
the substring "orange" starts)
print(s.replace("orange", "banana"))  # Output:
apple, banana (replaces "orange" with "banana")

# 3 - Checking string properties
print(s.startswith("apple"))  # Output: True
(checks if the string starts with "apple")
print(s.endswith("orange"))  # Output: True
(checks if the string ends with "orange")
print(s.isalpha())  # Output: False (checks if all
characters are alphabetic, false because of comma)
print(s.isdigit())  # Output: False (checks if all
characters are numeric digits (0-9))
print(s.isalnum())  # Output: False (checks if all
characters are alphanumeric)
print(s.isspace())  # Output: False (checks if the
string contains only whitespace)
print(s.islower())  # Output: True (checks if all
characters are lowercase)
print(s.isupper())  # Output: False (checks if all
characters are uppercase)

# 4 - Splitting and joining strings
print(s.split(", "))  # Output: ['apple',
'orange'] (splits the string into a list by ", ")
```

```
print("-".join(["apple", "orange"]))  # Output:
apple-orange (joins the list elements with "-")

# 5 - Stripping whitespace and characters
print(s.strip())  # Output: apple, orange (removes
leading/trailing spaces)
print(s.lstrip())  # Output: apple, orange
(removes leading spaces)
print(s.rstrip())  # Output: apple, orange
(removes trailing spaces)

# 6 - Padding and justifying
print(s.zfill(20))  # Output: 0000000apple, orange
(pads the string with zeros to the left)
print(s.ljust(20))  # Output: apple,
orange       (left-justifies the string)
print(s.rjust(20))  # Output:       apple, orange
(right-justifies the string)
print(s.center(20))  # Output:    apple,
orange    (centers the string in a field of width
20)

# 7 - Working with multi-line strings
multi_line_string = """apple,
orange"""
print(multi_line_string.splitlines())  # Output:
['apple', 'orange'] (splits the string into lines)

# 8 - Formatting strings
formatted_string = "Hello, {}!".format("apple")
print(formatted_string)  # Output: Hello, apple!
(formats the string using placeholders)

# 9 - Checking for substrings
```

```
print(s.find("banana"))  # Output: -1 (returns -1
since "banana" is not in the string)
```

Converting Data Types

In some cases, you may need to explicitly specify the data type of a variable. This can be done by using two basic ways.

Explicit Typing with Type Hints

```
name: str = "Maria"
age: int = 30
height: float = 1.68
is_student: bool = False
```

Type hints in Python are optional and do not change runtime behavior. They document the intended types and can be used by type checkers, IDEs, and linters to identify potential type inconsistencies before execution.

Explicit Type Conversion

Type conversions can be performed through explicit declaration, generally done by declaring the type of the destination variable and placing the variable to be converted in parentheses, as in the examples in Tables 2-5 and 2-6.

Table 2-5 Methods for type conversion

To demonstrate that the conversion was done correctly, a line was added to write the type of the converted variable on the screen, but it is not necessary to perform the conversion:

```
# 1 - Integers and floating numbers
# Convert a float to an integer (removes the
decimal part)
```

```
result_int = int(3.14)
print(result_int)  # Output: 3
print(type(result_int))  # Output: <class 'int'>

# Convert an integer to a float
result_float = float(42)
print("float(42):", result_float)  # Output: 42.0
print(type(result_float))  # Output: <class
'float'>

# Convert an integer to a string
result_str = str(42)
print("str(42):", result_str)  # Output: "42"
print(type(result_str))  # Output: <class 'str'>

# 2 - Representation for binary, octal and
hexadecimal
# Convert an integer to its binary representation
result_bin = bin(42)
print("bin(42):", result_bin)  # Output:
'0b101010'
print(type(result_bin))  # Output: <class 'str'>

# Convert an integer to its octal representation
result_oct = oct(42)
print("oct(42):", result_oct)  # Output: '0o52'
print(type(result_oct))  # Output: <class 'str'>

# Convert an integer to its hexadecimal
representation
result_hex = hex(42)
print(result_hex)  # Output: '0x2a'
print("Type:", type(result_hex))  # Output: <class
'str'>
```

```
#3 - Strings
# Convert a string to bytes
result_bytes = bytes("abc", "utf-8")
print("bytes('abc', 'utf-8'):", result_bytes)  #
Output: b'abc'
print(type(result_bytes))  # Output: <class
'bytes'>

# Convert a string to a mutable bytearray
result_bytearray = bytearray("abc", "utf-8")
print("bytearray('abc', 'utf-8'):",
result_bytearray)  # Output: bytearray(b'abc')
print(type(result_bytearray))  # Output: <class
'bytearray'>

# 4 - Complex number
# Convert two numbers to a complex number
result_complex = complex(2, 3)
print("complex(2, 3):", result_complex)  # Output:
(2+3j)
print(type(result_complex))  # Output: <class
'complex'>

# 5 - Boolean
# Convert 0 to a boolean (0 is considered False)
result_bool = bool(0)
print("bool(0):", result_bool)  # Output: False
print(type(result_bool))  # Output: <class 'bool'>

# 6 - Unicode of a character
# Get the Unicode code point of a character
result_ord = ord('A')
print("ord('A'):", result_ord)  # Output: 65
print(type(result_ord))  # Output: <class 'int'>
```

```
# Convert a Unicode code point to a character
result_chr = chr(65)
print("chr(65):", result_chr)  # Output: 'A'
print(type(result_chr))  # Output: <class 'str'>

# 7 - List, tuple and dictionary
# Convert a string to a list (each character
becomes an element)
result_list = list("abc")
print("list('abc'):", result_list)  # Output:
['a', 'b', 'c']
print(type(result_list))  # Output: <class 'list'>

# Convert a list to a tuple
result_tuple = tuple([1, 2, 3])
print(result_tuple)  # Output: (1, 2, 3)
print(type(result_tuple))  # Output: <class
'tuple'>

# Convert a list to a set (removes duplicates)
result_set = set([1, 2, 2, 3])
print("set([1, 2, 2, 3]):", result_set)  # Output:
{1, 2, 3}
print(type(result_set))  # Output: <class 'set'>

# Convert a list of key-value pairs to a
dictionary
result_dict = dict([('a', 1), ('b', 2)])
print("dict([('a', 1), ('b', 2)]):",
result_dict)  # Output: {'a': 1, 'b': 2}
print(type(result_dict))  # Output: <class 'dict'>

# Convert a list to a frozenset (immutable set)
result_frozenset = frozenset([1, 2, 3])
```

```
print("frozenset([1, 2, 3]):",
result_frozenset)  # Output: frozenset({1, 2, 3})
print(type(result_frozenset))  # Output: <class
'frozenset'>
```

Table 2-6 Type conversion methods in Python

See some examples below:

```
## Basic Numeric and String Conversions ##
print("## Basic Numeric and String Conversions
##")
# Output: ## Basic Numeric and String Conversions
##

# int(): Converts a value to an integer,
truncating decimals.
val_float = 3.14
res_int = int(val_float)
print(f"int({val_float}) -> Result: {res_int},
Type: {type(res_int)}")
# Output: int(3.14) -> Result: 3, Type: <class
'int'>

# float(): Converts a value to a floating-point
number.
val_int = 42
res_float = float(val_int)
print(f"float({val_int}) -> Result: {res_float},
Type: {type(res_float)}")
# Output: float(42) -> Result: 42.0, Type: <class
'float'>

# str(): Converts a value to a string.
res_str = str(val_int)
```

```
print(f"str({val_int}) -> Result: '{res_str}',
Type: {type(res_str)}")
# Output: str(42) -> Result: '42', Type: <class
'str'>

# complex(): Creates a complex number.
res_complex = complex(2, 3)
print(f"complex(2, 3) -> Result: {res_complex},
Type: {type(res_complex)}")
# Output: complex(2, 3) -> Result: (2+3j), Type:
<class 'complex'>

## Boolean Conversion ##
print("## Boolean Conversion ##")
# Output: ## Boolean Conversion ##

# bool(): Converts a value to a boolean. Non-
zero/non-empty values are True.
val_zero = 0
res_bool_false = bool(val_zero)
print(f"bool({val_zero}) -> Result:
{res_bool_false}, Type: {type(res_bool_false)}")
# Output: bool(0) -> Result: False, Type: <class
'bool'>

val_text = "hello"
res_bool_true = bool(val_text)
print(f"bool('{val_text}') -> Result:
{res_bool_true}, Type: {type(res_bool_true)}")
# Output: bool('hello') -> Result: True, Type:
<class 'bool'>

## Base and Character Conversions ##
print("## Base and Character Conversions ##")
# Output: ## Base and Character Conversions ##
```

```
num_to_convert = 42

# bin(): Converts an integer to a binary string.
res_bin = bin(num_to_convert)
print(f"bin({num_to_convert}) -> Result:
'{res_bin}', Type: {type(res_bin)}")
# Output: bin(42) -> Result: '0b101010', Type:
<class 'str'>

# hex(): Converts an integer to a hexadecimal
string.
res_hex = hex(num_to_convert)
print(f"hex({num_to_convert}) -> Result:
'{res_hex}', Type: {type(res_hex)}")
# Output: hex(42) -> Result: '0x2a', Type: <class
'str'>

# chr(): Converts a Unicode integer to its
character representation.
int_val_for_char = 65
res_chr = chr(int_val_for_char)
print(f"chr({int_val_for_char}) -> Result:
'{res_chr}', Type: {type(res_chr)}")
# Output: chr(65) -> Result: 'A', Type: <class
'str'>
```

Local and Global Variables in Python

Unlike languages like C++, Java, or C#, in Python, variables do not need to be explicitly declared as global, local, public, or private. The scope of a variable is determined automatically: variables created within functions are local, while those created outside of functions are global. You don't need to worry about the definition and usage of functions in Python now, as we will have a chapter dedicated to explaining functions

and their characteristics. If you need to modify a global variable from within a function, the global keyword should be used.

```
x = 10  # global variable

def change():
    global x
    x = 20  # modifies the global variable
```

Access Modifiers

Following the same philosophy applied to local and global variables, many programming languages provide mechanisms to restrict access to certain variables. Python, however, adopts a different approach by not enforcing strict scope-based access restrictions. Instead of using keywords such as *public* or *private*, Python relies on naming conventions to indicate the intended level of access and encapsulation.

By default, names are publicly accessible. A single leading underscore (_name) is a convention that marks a name as non-public or intended for internal use. Inside a class definition, a name beginning with two underscores (__name) triggers name mangling: Python rewrites the name to include the class name. This mechanism mainly avoids accidental name clashes; it does not provide strict privacy.

```
# Public attribute
name = "Charles"
print(name)  # Output: Charles

# Protected attribute (convention)
_age = 30
print(_age)  # Output: 30

# Double underscore: name mangling inside a class
class Person:
    __ssn = "123-45-6789"
```

```
    def show_ssn(self):
        return self.__ssn

person = Person()
print(person.show_ssn())  # Output: 123-45-6789
```

This concept will become easier to grasp as we explore the language in greater depth, especially when working with more advanced constructs such as functions and classes.

Introduction to Collections

Overview of Collection Types

A collection refers to a group of variables that may or may not be of the same type. It is typically represented by specialized container data types, such as lists, tuples, sets, or dictionaries, as outlined in Table 2-7.

Table 2-7 List of Python collections

Collections in programming can be broadly classified based on how they store elements, primarily as ordered or unordered. The key difference lies in predictability and access. Ordered collections, like Python's lists and tuples, maintain a specific, predictable sequence. This means each element has a fixed position, or index, which allows you to access it directly. The order in which you add items is the same order in which you retrieve them, and you can insert or remove elements at precise locations. In contrast, unordered collections, such as Python's sets, do not guarantee any particular sequence. Since there is no stable position, you cannot access elements by an index. Their internal order might change as items are added or removed, but this lack of structure often allows for faster performance when searching for or adding elements, as the collection doesn't need to maintain a specific sequence.

Summary of the Main Differences Between Collections

Lists are ordered and mutable, making them suitable for general-purpose sequences that require frequent changes. Tuples are ordered but immutable, ideal for representing fixed, related data. Sets are unordered and mutable, perfect for storing unique items and performing fast membership checks. Dictionaries are ordered by insertion and mutable, with unique keys, making them perfect for storing related information in key–value pairs for quick lookups. Choosing the right collection type depends on the specific requirements of your project.

Table 2-8 Comparison of Python collections

Practice Exercises

1. *Predict the Output—Understanding Numeric Types*:

   Without running the code, predict the output. Then execute and explain why Python prints those types/results.

```
x = 50
y = 2.5
print(type(x))
print(type(y))
print(x + y)
```

2. *Write a Program Using Underscores in Large Numbers*:

   Define at least three large numeric values using underscores (e.g., population, distance, earnings).

   Print each value with a descriptive message.

3. *Floating-Point Behavior—Investigate Precision*:

   Predict the result of the comparison, and then run it and print values.

```
a = 0.3
b = 0.1 * 3
```

```
b = 0.1 * 3

print(a == b)
print(____)
```

4. *Practice Arithmetic Operations*:
   Using 15 and 4, perform +, -, *, /, //, %, and **. Print each result with labels.
5. *Explore Bitwise Operations*:
   Run and describe logic behind each result.

```
x = 12
y = 5
print(x & y)
print(x | y)
print(x ^ y)
print(x << 1)
print(x >> 1)
```

6. *Type Conversion Demonstration*:
   Convert one number into int, float, string, bin, hex, and oct. Show type of each result.
7. *String Indexing and Slicing Exploration*:
   Demonstrate positive index, negative index, and slicing variations.
8. *Fix the String Modification Attempt*:
   The code tries to modify a string directly. Correct it to print 'Hello'.

```
text = 'hello'
text[0] = 'H'
print(text)
```

9. *Apply String Methods*:
   Choose a sentence and apply at least six string methods. Print before and after.
10. *Analyze and Fix Logic Error*:
    The comparison is incorrect due to data type. Fix it.

```
age = '18'
print(age > 10)
```

11. *Number Comparison Program*:
    Ask the user for a number and check >10, <100, and ==50. Print readable messages.
12. *Rewrite Using F-Strings*:
    Rewrite exercise #11 but using one formatted f-string.
13. *Membership Testing in Lists*:
    Make a list of five fruits. Ask for user input and check if the fruit exists.
14. *Apply Type Hints*:
    Add the correct type annotation.

```
name: _____ = 'Ana'
score: _____ = 9.5
is_active: _____ = True
```

15. *Converting User Input to Different Types*:
    Request input and convert to int, float, and (if numeric) hexadecimal.

Understand Variable Scope:

Understand variable scope.

16. Run the code. Explain why it fails.

```
count = 0
def update():
    count = count + 1
update()
print(count)
```

17. *Fix the Scope Issue*:
    Rewrite exercise #16 using the global keyword.
18. *Operator Precedence—Expression Challenge*:
    Write two expressions (one with parentheses). Describe the change in result.
19. *Valid and Invalid Variable Names*:
    Write five valid and two invalid names. Explain at least two.
20. *Final Mini-project*:
    Ask for a name and age, convert types, perform one arithmetic operation, apply two string methods, and display the result using an f-string.

Answer Key

1. *Predict the Output—Understanding Numeric Types*:

```
x = 50
y = 2.5
print(type(x))
print(type(y))
print(x + y)
```

Expected Output:

```
<class 'int'>
<class 'float'>
52.5
```

Here, x is an int and y is a float. When they are added, Python uses floating-point arithmetic for the operation, so the result is 52.5 and its type is float.

2. *Write a Program Using Underscores in Large Numbers*:

Example Solution:

```
# Large numbers using underscores for
readability
population = 215_000_000      # e.g., country
population
distance_to_moon = 384_400_000  # distance in
meters
annual_earnings = 75_000_000   # company
annual earnings in dollars

print("Population:", population)
print("Distance to the Moon (m):",
distance_to_moon)
print("Annual earnings (USD):",
annual_earnings)
```

Underscores in numeric literals are digit separators used for readability. They do not change the value: 215_000_000 and 215000000 represent the same integer.

3. *Floating-Point Behavior—Investigate Precision*:

```
a = 0.3
b = 0.1 * 3
```

```
print(a == b)
print(a, b)
```

Typical Output:

```
False
0.3 0.30000000000000004
```

Values such as 0.3 and 0.1 cannot always be represented exactly in binary floating point. Therefore, 0.1 * 3 is stored as a value slightly different from 0.3, so the equality test returns False.

4. *Practice Arithmetic Operations*:

```
a = 15
b = 4
print("a + b =", a + b)    # Addition
print("a - b =", a - b)    # Subtraction
print("a * b =", a * b)    # Multiplication
print("a / b =", a / b)    # True division
(float)
print("a // b =", a // b) # Floor division
(integer)
print("a % b =", a % b)    # Remainder (modulo)
print("a ** b =", a ** b) # Exponentiation
```

/ performs true division and, for these integer operands, returns a float. The // operator performs floor division, % returns the remainder, and ** performs exponentiation.

5. *Explore Bitwise Operations*:

```
x = 12  # in binary: 1100
y = 5   # in binary: 0101

print(x & y)
print(x | y)
```

```
print(x | y)

print(x ^ y)
print(x << 1)
print(x >> 1)
```

Expected Output:

```
4
13
9
24
6
```

With x = 12 (1100 in binary) and y = 5 (0101), &, |, and ^ compare corresponding bits. The results are 4, 13, and 9. Shifting x left by one bit gives 24; shifting it right by one bit gives 6.

6. *Type Conversion Demonstration*:

```
n = 42
as_int = int(n)
as_float = float(n)
as_str = str(n)
as_bin = bin(n)
as_hex = hex(n)
as_oct = oct(n)

print(as_int, type(as_int))
print(as_float, type(as_float))
print(as_str, type(as_str))
print(as_bin, type(as_bin))
print(as_hex, type(as_hex))
print(as_oct, type(as_oct))
```

Expected Output:

```
42 <class 'int'>
```

```
42 <class 'int'>

42.0 <class 'float'>
42 <class 'str'>
0b101010 <class 'str'>
0x2a <class 'str'>
0o52 <class 'str'>
```

Python provides several built-in functions to convert values between different data types and base representations. The int() function preserves the value as a standard integer, while float() transforms it into a floating-point number, such as converting 42 into 42.0, and str() converts the value into a text string like "42". For representing integers in different numeral systems, the functions bin(), hex(), and oct() produce string outputs corresponding to binary, hexadecimal, and octal formats, respectively; for example, calling bin(42) returns the binary string "0b101010".

7. *String Indexing and Slicing Exploration*:

   Example Solution:

```
text = "Python"

# Positive indexing
print("First character:", text[0])      # 'P'
print("Third character:", text[2])      # 't'

# Negative indexing
print("Last character:", text[-1])      # 'n'
print("Second to last:", text[-2])      # 'o'

# Slicing
print("First three letters:", text[0:3])   #
'Pyt'
print("From index 2 to end:", text[2:])    #
'thon'
```

```
print("Every second character:",
text[0::2])  # 'Pto'
```

In Python, sequences support a flexible indexing system where positive indexes start at 0 to reference elements from the beginning, while negative indexes start at –1 to access elements counting backward from the end. To extract specific subsections of data, the slicing syntax start:stop:step is used. A key characteristic of this operation is that the stop parameter is exclusive, meaning the slice retrieves elements up to that index but does not include the element at the index itself.

8. *Fix the String Modification Attempt*:

Original (Incorrect) Code:

```
text = 'hello'
text[0] = 'H'
print(text)
```

This fails because strings are immutable; you cannot change a single character by assignment.

Correct Version:

```
text = 'hello'
text = 'H' + text[1:]
print(text)
```

Output:

```
Hello
```

Strings are immutable, so text[0] = 'H' raises a TypeError. The expression text = 'H' + text[1:] creates a new string and rebinds text to 'Hello'.

Apply String Methods:

9. *Example Solution*:

```
sentence = "  Python is Amazing!  "
print("Original:", repr(sentence))

result = sentence.strip()       # remove
leading/trailing spaces
result = result.lower()         # all
lowercase
result = result.replace("amazing",
"powerful")  # replace word
result = result.capitalize()    # first letter
uppercase
result = result.replace("python",
"Python")     # fix proper name
length = len(result)            # length of
final string

print("Processed:", repr(result))
print("Length of final string:", length)
print("Uppercase version:", result.upper())
```

Output:

```
Original: '  Python is Amazing!  '
Processed: 'Python is powerful!'
Length of final string: 19
Uppercase version: PYTHON IS POWERFUL!
```

Python provides a variety of built-in methods for efficient string manipulation and analysis. The .strip() method is used to clean text by removing leading and trailing whitespace, while .replace(old, new) substitutes occurrences of a specific substring with another. For adjusting letter case, .lower() and .upper() convert the entire string, whereas .capitalize() ensures only the

first character is uppercase. Finally, the len() function returns the total length of the string, counting every character including spaces and punctuation.

10. *Analyze and Fix Logic Error*:

    Original Code:

```
age = '18'
print(age > 10)
```

    This code is incorrect in two ways:

    1. In Python 3, comparing a string ('18') with an integer (10) is not allowed and raises a TypeError.
    2. Even if it were allowed, we want to compare numeric values, not string text.

    Correct Version:

```
age = '18'
age_number = int(age)        # convert to
integer
print(age_number > 10)
```

    age is a string, so it cannot be ordered against the integer 10. Converting it with int(age) produces the integer 18, and 18 > 10 evaluates to True.

11. *Number Comparison Program—Using Basic Comparisons Only*:

```
number = float(input("Enter a number: "))
print("Is the number greater than 10?", number
> 10)
print("Is the number less than 100?", number <
100)
print("Is the number equal to 50?", number ==
```

```
print("Is the number equal to 50?", number ==
50)
```

Example (User Enters 75):

```
Enter a number: 75
Is the number greater than 10? True
Is the number less than 100? True
Is the number equal to 50? False
```

This program converts the input to float and evaluates three comparisons directly. Each comparison returns True or False, which is printed with a descriptive label.

12. *Rewrite Using F-Strings—Single Formatted Output*:

```
number = float(input("Enter a number: "))
message = (
    f"For the number {number}:\n"
    f" - Greater than 10? {number > 10}\n"
    f" - Less than 100? {number < 100}\n"
    f" - Equal to 50? {number == 50}"
)
print(message)
```

Example (User Enters 75):

```
Enter a number: 75
For the number 75.0:
 - Greater than 10? True
 - Less than 100? True
 - Equal to 50? False
```

This version performs the same comparisons as exercise 11 and places their results in one f-string. The \n escape sequence inserts line breaks in the formatted message.

13. *Membership Testing in Lists—Checking If a Fruit Exists*:

```
fruits = ["apple", "banana", "orange",

"grape", "kiwi"]
user_fruit = input("Enter the name of a fruit:
")
is_in_list = user_fruit.lower() in fruits
print("List of fruits:", fruits)
print("Is the chosen fruit in the list?",
is_in_list)
```

Example (User Enters Banana):

```
Enter the name of a fruit: Banana
List of fruits: ['apple', 'banana', 'orange',
'grape', 'kiwi']
Is the chosen fruit in the list? True
```

The input is converted to lowercase before the membership test, so an entry such as Banana matches 'banana' in the list. The in operator returns a boolean value, stored in is_in_list and printed.

14. *Apply Type Hints—Adding Basic Annotations*:

```
name: str = "Ana"
score: float = 9.5
is_active: bool = True

print(name, type(name))
print(score, type(score))
print(is_active, type(is_active))
```

Type hints state the intended types of the variables: str, float, and bool. Python does not enforce these annotations at runtime, but type checkers and IDEs can use them to detect

inconsistencies.

15. *Converting User Input to Different Types—int, float, and Hexadecimal*:

```
user_input = input("Enter an integer number:
")

as_int = int(user_input)
as_float = float(user_input)
as_hex = hex(as_int)

print("Original text:", user_input)
print("As integer:", as_int)
print("As float:", as_float)
print("As hexadecimal:", as_hex)
```

Example (User Enters 255):

```
Enter an integer number: 255
Original text: 255
As integer: 255
As float: 255.0
As hexadecimal: 0xff
```

Explanation: input() returns a string. int(user_input) converts that text to an integer, float(user_input) converts it to a floating-point number, and hex(as_int) returns a hexadecimal string such as "0xff".

16. *Understand Variable Scope—Why This Code Fails*:

```
count = 0
def update():
    count = count + 1
update()
print(count)
```

Expected Error:

```
UnboundLocalError: cannot access local
variable 'count' where it is not associated
with a value
```

The function fails with UnboundLocalError because assigning to count inside update() makes count local to that function. Python therefore tries to read the local count before it has a value.

17. *Fixing the Scope Issue—Using global*:

```
count = 0
def update():
    global count
    count = count + 1
update()
print(count)
```

Output:

```
1
```

The global declaration tells Python to rebind the module-level count inside update(). After the function runs once, count is 1.

18. *Operator Precedence—With and Without Parentheses*:

```
a = 2 + 3 * 4
b = (2 + 3) * 4

print(a)
print(b)
```

Output:

```
14
20
```

```
20
```

The first expression follows Python's standard operator precedence, where multiplication is evaluated before addition.

The second expression changes the evaluation order using parentheses, resulting in a different final value.

19. *Valid and Invalid Variable Names*:

```
valid1 = 10
_total = 50
value2025 = "OK"
userName = "Ana"
PI = 3.14

# invalid examples (not executed):
# 2value = 10
# user-name = "Ana"
```

Valid variable names may include letters, digits, and underscores, but must not begin with a digit or include characters such as hyphens. The invalid examples illustrate common naming mistakes.

20. *Final Mini-project—Input, Conversion, and String Formatting*:

```
name = input("Enter your name: ")
age = input("Enter your age: ")

age_number = int(age)
next_age = age_number + 1

clean_name = name.strip()
formatted_name = clean_name.title()

print(f"Hello, {formatted_name}! Next year you
```

```
will be {next_age}.")
```

Example:

```
Enter your name:   maria clara
Enter your age: 18
Hello, Maria Clara! Next year you will be 19.
```

This final activity combines input handling, type conversion, arithmetic processing, and string formatting. After conversion and formatting, the final message is produced using an f-string for a clear and readable result.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_3

3. Handling Collections

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

As discussed in the previous chapter, a collection refers to a grouping of elements that may or may not share the same data type. In Python, these structures are represented by specialized containers, such as lists, tuples, sets, and dictionaries. In this chapter, we will learn about each of them in order to select the correct type according to the requirements of the code to be developed by the user.

Lists

Lists are capable of storing sequences of mutable variables of diverse data types, including strings, integers, and boolean values. Elements may appear multiple times within a list. To instantiate a list, assign a variable to a list enclosed in square brackets, with elements delineated by commas.

Creating Lists

This code demonstrates the creation and printing of different types of lists in Python, including lists of numbers, strings, and mixed data types and nested lists:

```
# Create a list of numbers
numbers = [1, 2, 3, 4, 5]
print(numbers) # Output: [1, 2, 3, 4, 5]
```

```
# A list of strings
fruits = ["apple", "banana", "cherry", "orange"]
print(fruits)  # Output: ['apple', 'banana',
'cherry', 'orange']

# A list with different types of elements
mixed_list = [42, "hello", 3.14, True]
print(mixed_list)  # Output: [42, 'hello', 3.14,
True]

# A list containing other lists or nested List
nested_list = [[1, 2, 3], ["a", "b", "c"], [True,
False]]
print(nested_list)  # Output: [[1, 2, 3], ['a',
'b', 'c'], [True, False]]
```

Each element within a list is assigned a unique index. The initial element possesses an index of 0, the subsequent element an index of 1, and so forth.

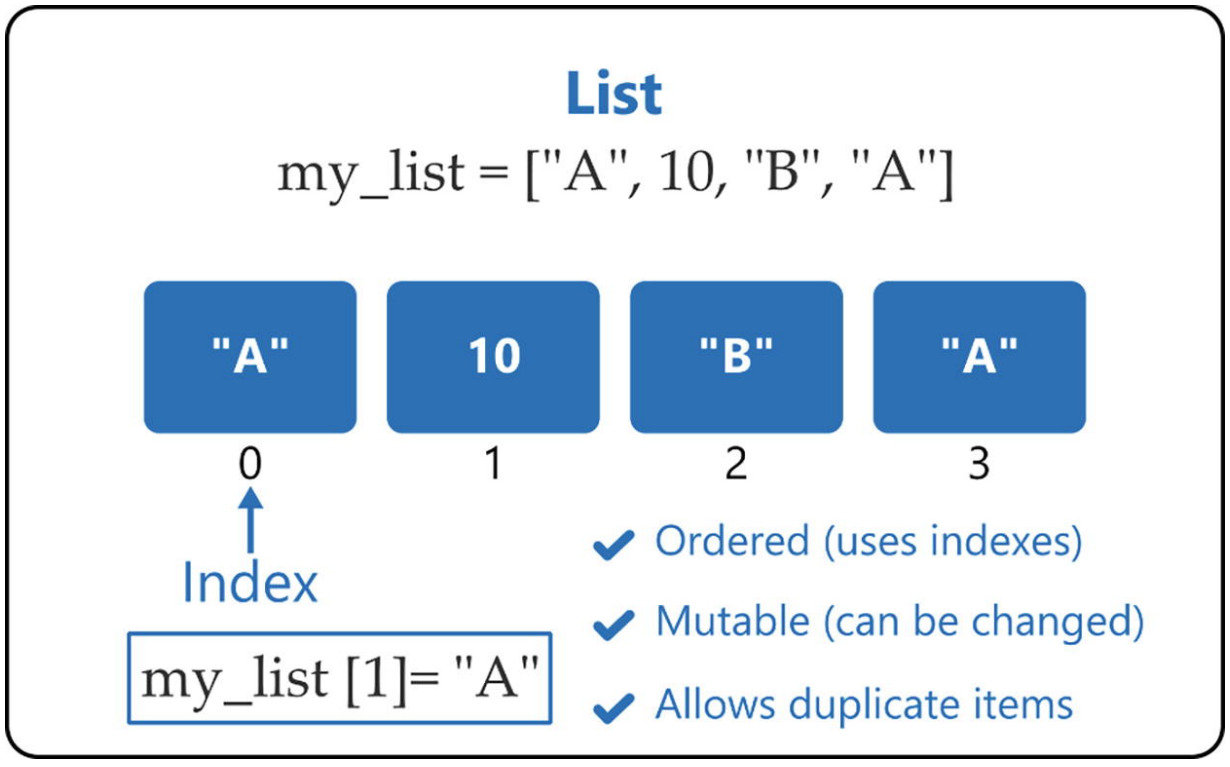


Figure 3-1 Visual representation of a Python list, showing its indexed, mutable, and ordered nature

Accessing and Modifying Elements

To access an element in the list, simply declare the name of the list and then use square brackets [ ] containing the element's index:

```
numbers = [1, 2, 3, 4, 5]
print(numbers[3]) # Output: 4
```

Elements are updated by assigning a new value to a specific index:

```
# Updating elements in a list
fruits = ["apple", "banana", "cherry"]  # Define a
list of fruits
fruits[2] = 'orange'  # Replace the third element
('cherry') with 'orange'
print(fruits)  # Output: ['apple', 'banana',
'orange']
```

Methods for Lists in Python

Table 3-1 Methods and operations for Python lists.Source: Adapted from the Python Software Foundation, Python Tutorial, section 5.1, "More on Lists" (*https://docs.python.org/3/tutorial/datastructures.html*)

Operations with Lists

Table 3-2 List operators

len()

The len() method is useful for determining the number of elements in a list.

```
fruits = ["apple", "banana", "cherry"]
```

```
list_length = len(fruits)
print(list_length) # Output: 3
```

index()

To find the index of an element, simply call the index() method.

```
# Find an element's index in a list
fruits = ["apple", "banana", "cherry"]

# Find the index of the element 'banana' in the
list
index = fruits.index('banana')

# Print the index of 'banana'
print(index) # Output: 1
```

To define the range of indexes to be analyzed, simply specify the starting and ending indexes after the element declaration within the parentheses of the index() method.

```
# Create a list with duplicate elements
fruits = ["apple", "banana", "cherry", "apple",
"banana", "cherry"]

# Find the index of 'banana' within the range [1,
5)
first_index = fruits.index('banana', 1, 5)
print(first_index)  # Output: 1

# Find the index of 'banana' within the range [2,
len(fruits))
second_index = fruits.index('banana', 2,
len(fruits))
print(second_index)  # Output: 4
```

index() for Nested Lists

```
# Find element index in a nested list
nested_list = [[1, 2, 3], ["a", "b", "c"], [True,
False]]

# Find the index of the element 2 in the first
sublist (nested_list[0])
index = nested_list[0].index(2)

print(index)  # Output: 1
```

Finding Elements

```
# Find elements in nested list
nested_list = [[1, 2, 3], ["a", "b", "c"], [True,
False]]

# Loop through each sublist to search for 2
for sublist in nested_list:
    if 2 in sublist:
        index = nested_list.index(sublist)  # Get
the index of the sublist
        print(f"Found 2 in sublist: {sublist}, at
index {index}")
        break  # Exit the loop once 2 is found
```

append() and remove()

To append elements to a list, employ the append() method. To eliminate elements, utilize the remove() method, like the example below:

```
# Adding, removing, and updating elements
# Create an initial list
fruits = ["apple", "banana", "cherry"]

fruits.append("orange")  # Add an element
```

```
print(fruits)  # Output: ['apple', 'banana',
'cherry', 'orange']

fruits.remove("banana")  # Remove an element
print(fruits)  # Output: ['apple', 'cherry',
'orange']
```

extend()

The extend() method can be used to concatenate two different lists.

```
# Create an initial list
user_list = [1, 2, 3]

# Create an iterable to extend the list
new_items = [4, 5, 6]

# Use list.extend to add items from the iterable
to the list
user_list.extend(new_items)

# Print the updated list
print(user_list) # Output: [1, 2, 3, 4, 5, 6]
```

insert()

To insert an element at a specific position in a list, use the insert() method. Its syntax is insert(index, element), where index is the position and element is the value to be inserted.

```
# Create a list
user_list = [10, 20, 30, 40]

# Inserting an element at the beginning
user_list.insert(0, 5)  # Insert 5 at index 0
print("After inserting at the beginning:",
user_list)
```

```
# Inserting an element in the middle
user_list.insert(2, 15)  # Insert 15 at index 2
print("After inserting in the middle:", user_list)

# Inserting an element at the end
user_list.insert(len(user_list), 50)  # Equivalent
to append(50)
print("After inserting at the end:", user_list)
```

pop()

The pop() method removes an element at a specific position in the list and returns its value.

```
# Create a list
fruits = ['apple', 'banana', 'orange', 'mango']

# Print the original list
print("Original List:", fruits) # Output: Original
List: ['apple', ....., 'mango']

# Remove the element at index 2 (orange), and show
the value.
removed=fruits.pop(2)
print(removed) # Output: orange

# Print the list after removing the element
print("List after removing element:", fruits)
# Output: List after removing element: ['apple',
'banana', 'mango']
```

clear() and del

To remove all elements from a list, use the clear() method, or delete the list using del.

```
# Create a list
```

```
fruits = ['apple', 'banana', 'orange', 'mango']  # Initialize a list with some fruit names

# Print the original list
print("Original List:", fruits)  # Display the original list

# Clear the list using del
del fruits[:]  # Deletes all elements in the list without deleting the list itself
# Print the list after clearing it
print("After del:", fruits)  # Output: After del: []

# Create a list
fruits = ['apple', 'banana', 'orange', 'mango']  # Reinitialize the list with the same elements

# Print the original list
print("Original List:", fruits)  # Display the original list again

# Clear the list using clear()
fruits.clear()  # Removes all elements from the list using the clear() method
# Print the list after clearing it
print("After clear():", fruits)  # Output: After clear(): []
```

sort() and reverse()

The sort() method can sort a list alphabetically or numerically in ascending order.

```
# Create a list
fruits = ['apple', 'banana', 'orange', 'mango']
```

```
# Sort the list alphabetically
fruits.sort()
print("Sorted list:", fruits)  # Output: Sorted
list: ['apple', 'banana', 'mango', 'orange']

# Reverse the order of the sorted list
fruits.reverse()
print("Reversed list:", fruits)  # Output:
Reversed list: ['orange', 'mango', 'banana',
'apple']
```

Alternatively, the sorted() function can be used.

```
# Create a list
fruits = ['apple', 'banana', 'orange', 'mango']

# Sort the list alphabetically using sorted()
sorted_fruits = sorted(fruits)

print("Sorted list:", sorted_fruits)
# Output: Sorted list: ['apple', 'banana',
'mango', 'orange']
```

The reverse() method reverses the order of elements in a list, but it does not sort them alphabetically or numerically.

```
# Define a list of fruits
fruits = ['apple', 'banana', 'orange', 'mango']

# Reverse the list (not sorting alphabetically)
# This will reverse the original order, not sort
alphabetically
fruits.reverse()

# Print the reversed list
print("Reversed list:", fruits)
```

```
# Output: Reversed list: ['mango', 'orange',
'banana', 'apple']
```

copy()

To create a copy of a list, use the copy() method. This ensures that modifications to one list do not affect the other.

```
# Create a list
fruits = ['apple', 'banana', 'orange', 'mango']

# Create a copy of the list
fruits_copy = fruits.copy()

# Modify the original list
fruits.append('grape')

print("Original list:", fruits)  # Output:
Original list: ['apple', 'banana', 'orange',
'mango', 'grape']
print("Copied list:", fruits_copy)  # Output:
Copied list: ['apple', 'banana', 'orange',
'mango']

# As you can see, the copied list remains
unchanged even after modifying the original list.
```

Remove Duplicates

```
my_list = [1, 2, 3, 2, 1, 4]
my_list = list(set(my_list))  # Remove duplicates
and create a new list
my_list.sort()  # Sort the list
print(my_list)  # Output: [1, 2, 3, 4]
```

count()

The count() method returns the number of times a specified element appears in a list.

```
# Create a list
fruits = ['apple', 'banana', 'orange', 'mango',
'banana']  # List with a duplicate item

# Print the original list
print("Original List:", fruits)

# Count the occurrences of 'banana'
banana_count = fruits.count('banana')  # Count how
many times 'banana' appears in the list

# Print the result
print("Count of 'banana':", banana_count)  #
Output: Count of 'banana': 2
```

min() and max()

The methods max() and min() are very useful to find the largest or smallest element in a list.

```
numbers = [10, 20, 5, 40]
print(min(numbers))  # Output: 5
print(max(numbers))  # Output: 40
```

When used on strings without a key argument, min() and max() compare values lexicographically. To compare strings by length, use key=len. If several strings have the same length, the first one encountered is returned.

```
# Define a list with some fruit names
fruits = ["apple", "banana", "apple", "cherry",
"orange", "grape"]
```

```
# Find and print the smallest element in the list
based on string length
print(min(fruits, key=len))  # Output: apple
# The min() function returns the shortest string;
ties are resolved by first occurrence.

# Find and print the largest element in the list
based on string length
print(max(fruits, key=len))  # Output: banana
# The max() function returns the longest string;
ties are resolved by first occurrence.
```

sum()

To calculate the sum of all numeric elements in a list, use the method sum().

```
numbers = [1, 2, 3, 4]
print(sum(numbers))  # Output: 10
```

Membership Operators

It's possible to use the in operator to check if an element exists in a list and not in to check if an element does not exist.

```
# Define a list
fruits = ["apple", "banana", "cherry", "orange"]

# Check for membership
print("apple" in fruits)      # Output: True
print("grape" not in fruits)  # Output: True
```

Concatenation Operator

Use the operator plus sign (+) to concatenate two lists into a new list.

```
# Define two lists of fruits
fruits1 = ["apple", "banana", "cherry"]
```

```
fruits2 = ["orange", "grape", "mango"]

# Concatenate the two lists
all_fruits = fruits1 + fruits2

# Print the concatenated list
print(all_fruits) # Output: ['apple', 'banana',
'cherry', 'orange', 'grape', 'mango']
```

Repetition Operator

To repeats a list a specified number of times, use the operator asterisk (*); it can be used with numbers or strings.

Example with Numbers:

```
numbers = [1, 2, 3]
print(numbers * 2)  # Output: [1, 2, 3, 1, 2, 3]
```

Example with Strings:

```
fruits = ["apple", "banana", "cherry"]

# Repeat the list of fruits 2 times
repeated_fruits = fruits * 2

# Print the repeated list
print(repeated_fruits)# Output: ['apple',
'banana', 'cherry', 'apple', 'banana', 'cherry']
```

Slicing Operator

The slicing operation allows to extract a portion of the list using the : operator.

```
# Define a list
numbers = [10, 20, 30, 40, 50]

# Slice the list
```

```
subset = numbers[1:4]  # Extract elements from
index 1 to 3 (4 is excluded)

print(subset)  # Output: [20, 30, 40]

# Other slicing examples:
print(numbers[:3])  # Output: [10, 20, 30] (from
start to index 2)
print(numbers[2:])  # Output: [30, 40, 50] (from
index 2 to the end)
print(numbers[-3:]) # Output: [30, 40, 50] (last
three elements)
```

Equality Operators

Use the operator == to check if two lists are equal or use != to check if there is difference between two lists.

```
list1 = [10, 20, 30]
list2 = [10, 20, 30]
print(list1 == list2)  # Output: True
print(list1 != list2)  # Output: False
```

Lists and the Equals Sign



Using the equals sign (=) does not create a copy of a list; it only creates a new reference to the original list.

When you write list_b = list_a, you're creating a second reference to the same list object, not a new list, meaning both list_a and list_b point to the same data, and changes made through one will affect the other.

```
# Create an original list
original_list = [1, 2, 3]
```

```
# Assign it to a new variable using =
reference_list = original_list

# Now, modify the "new" list
reference_list.append(4)

# Print both lists
print(f"Reference List: {reference_list}") #
Output: [1, 2, 3, 4]
print(f"Original List:  {original_list}")  #
Output: [1, 2, 3, 4] The original changed too.
```

Operations Between Lists

To handle multiple lists, you can utilize methods originally designed for sets, another Python collection type. To perform these operations, lists must be converted to sets, as shown in Table 3-3.

Table 3-3 Set-like operations between lists

The examples in the code below demonstrate how these methods can be used:

```
list1 = [1, 2, 3]
list2 = [3, 4, 5]

union_list = list(set(list1) | set(list2))
intersection_list = list(set(list1) & set(list2))
difference_list = list(set(list1) - set(list2))
symmetric_difference_list = list(set(list1) ^
set(list2))

print(union_list)  # [1, 2, 3, 4, 5]
print(intersection_list)  # [3]
print(difference_list)  # [1, 2]
print(symmetric_difference_list)  # [1, 2, 4, 5]
```

Nested List

Elements in a nested list are updated this way:

```
# Create a list
nested_list = [[1, 2, 3], ["a", "b", "c"], [True,
False]]

# Update an element
nested_list[0][1] = 99  # Change element at index
1 in the first sublist to 99
nested_list[1][0] = "apple"  # Change element at
index 0 in the second sublist to "apple"

# Print the updated nested list
print(nested_list)
```

The following example shows how to update, append, and remove elements in a nested list:

```
# Update, append and remove in a nested list
nested_list = [[1, 2, 3], ["a", "b", "c"], [True,
False]]  # Create a list

nested_list[0][1] = 99  # Change element at index
1 in the first sublist to 99
nested_list[1][0] = "apple"  # Change element at
index 0 in the second sublist to "apple"

nested_list[0].append(100) # Add 100 to the first
sublist
nested_list.append([1.23, 4.56]) # Add a new
sublist [1.23, 4.56] to the end of the list

nested_list[1].remove("b") # Remove "b" from the
second sublist
```

```
nested_list.remove([True, False])  # Remove the
sublist [True, False]

# Print the updated nested list
print(nested_list)
```

Common Error Messages for List Methods

The methods can raise various error messages when they fail to execute. These error messages can help solve problems quickly.

Table 3-4 Common error messages for list methods

List Comprehension

List comprehensions are a compact syntax built on the concepts of for loops and if statements, which will be covered in detail in the following chapters. Don't worry if this seems confusing at first; it will make perfect sense once you're familiar with these basic programming tools. If you wish, you can directly study Chapter 6, which deals with loops and conditional tests.

List comprehension in Python provides a compact and readable way to create a new list by transforming or filtering values from an existing list or another iterable.

It consists of brackets containing an expression followed by a for clause and, optionally, if clauses for filtering. This concise syntax makes your code more readable and efficient, eliminating the need for explicit loops and conditional statements.

```
fruits = ["apple", "banana", "cherry", "kiwi",
"mango"]

newlist = [x for x in fruits if "i" in x]

print(newlist) # Output: ['kiwi']
```

This code creates a new list called newlist by filtering the original fruits list using a list comprehension. It iterates through each element in fruits, temporarily calling it x, and checks if the letter "i" is present within that string. If the condition is true, the element x is added to the newlist. The process evaluates "apple", "banana", "cherry", and "mango" as false, but finds "i" in "kiwi", resulting in newlist containing only ['kiwi'].

- *Expression (*x*)*: This is the value that will be added to the new list. In this case, it's the fruit's name (x) itself.
- *Loop (*for x in fruits*)*: This part goes through each item in the fruits list, one by one. Each item (like "apple" or "banana") is temporarily assigned to the variable x.
- *Condition (*if "i" in x*)*: This is a filter that checks if the fruit's name (x) contains the letter "i". If the condition is true, the fruit's name is added to the new list. If it's false, the fruit is skipped.

Think of it like a pipeline:

- Take each fruit from the list (for x in fruits).
- Check if it meets the condition (if "i" in x).
- If it does, add it to the new list (x is added).

This process repeats for each fruit in the original list, resulting in a new list with only the fruits that contain the letter "i".

The code works like Table 3-5.

Table 3-5 List comprehension filtering example

Tuples

Tuples store sequences of immutable variables of different types, such as strings, integers, and booleans. Since tuples are immutable, methods like append() and remove() cannot be used. However, it is possible to rewrite all the values of a tuple by assigning a new value to the tuple variable.

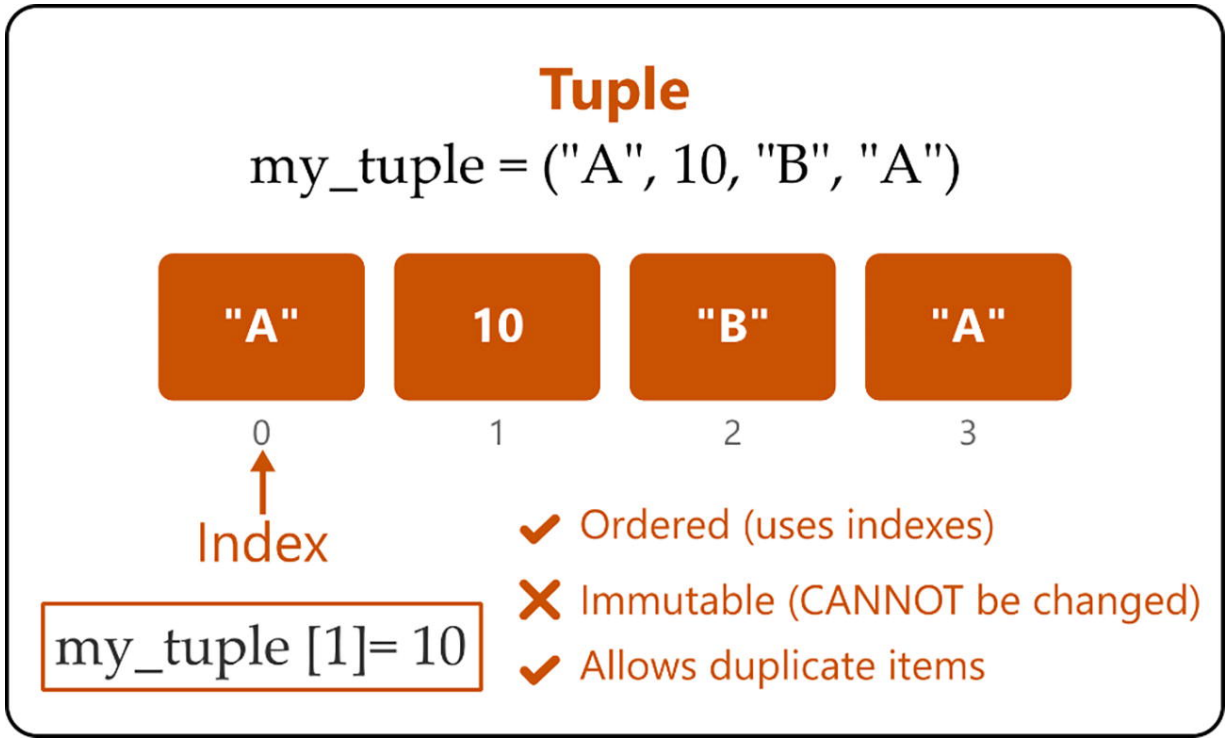


Figure 3-2 Visual representation of a Python tuple, emphasizing its immutability and ordered structure

To create a tuple, assign a variable to a comma-separated set of values enclosed in parentheses. Tuples are more memory-efficient than lists and can serve as dictionary keys, unlike lists.

```
# A tuple of numbers
numbers = (1, 2, 3, 4, 5)
print(numbers)  # Output: (1, 2, 3, 4, 5)

# A tuple of strings
fruits = ("apple", "banana", "cherry", "orange")
print(fruits)  # Output: ('apple', 'banana',
'cherry', 'orange')

# A tuple with different types of elements
mixed_tuple = (42, "hello", 3.14, True)
print(mixed_tuple)  # Output: (42, 'hello', 3.14,
True)
```

```
# A tuple containing other tuples or nested tuples
nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True,
False))
print(nested_tuple)  # Output: ((1, 2, 3), ('a',
'b', 'c'), (True, False))
```

Accessing Elements

In a tuple, each element has a unique index; the first element receives the index 0, the second element receives the index 1, and the next elements continue the sequence.

See the example code:

```
# Access elements by index
fruits = ("apple", "banana", "cherry", "orange")

# Access the first element
print(fruits[0])  # Output: apple

# Access the last element
print(fruits[-1])  # Output: orange
```

Methods for Tuples in Python

Although tuples are immutable and do not support methods for modifying their content, they do provide several useful methods.

Table 3-6 Methods and operations for Python tuples

len()

The len() function returns the number of elements in a tuple.

```
# Define a tuple with some fruit names
fruits = ("apple", "banana", "cherry")
tuple_length = len(fruits)
print(tuple_length)  # Output: 3
```

index()

To find the index of an element in a tuple, use the index() method. This method can be used in two ways. The first mode searches for the first occurrence of the element and returns its index. In the second mode, you can define the starting and stopping indexes for the search. It is important to note that in this case, the search stops at the index just before the specified stop index. For example, if the stop index is 4, the search will go up to index 3 only.

```
# Find element index in a tuple
fruits = ("apple", "banana", "cherry", "banana",
"orange")

# Without specifying start and stop, it finds the
first occurrence of 'banana'
index_first = fruits.index("banana")
print("First 'banana' found at index:",
index_first)  # Output: 1

# Find the index of 'banana' starting from index 2
index_after_first = fruits.index("banana", 2)
print("Second 'banana' found at index:",
index_after_first)  # Output: 3

# Find the index of 'banana' within a specific
range (from index 1 to index 4)
# This will search only between indices 1 and 4,
inclusive of 1 and exclusive of 4
index_in_range = fruits.index("banana", 1, 4)
print("Found 'banana' at index within range 1 to
4:", index_in_range)  # Output: 1
```

The example below demonstrates that the search with a range does not extend to the maximum value of the requested index:

```
# Define a tuple with some fruit names
fruits = ("banana", "apple", "cherry", "orange",
"banana")

# Find the index of the first occurrence of
"banana" within the range from index 2 to 4
index_in_range = fruits.index("banana", 2, 5)
print("Found 'banana' at index within range 2 to
4:", index_in_range)  # Output: 4

# Try to find the index of the first occurrence of
"banana" within the range from index 2 to 3
try:
    index_in_range = fruits.index("banana", 2, 4)
except ValueError as e:
    print(f"ValueError: {e}")
# Output: ValueError: tuple.index(x): x not in
tuple
```

Concatenation (+)

To combine two or more tuples into one, just use the plus sign (+) operator, like in the example below:

```
# Define a tuple with some fruit names
fruits = ("apple", "banana", "apple", "cherry")

# Define another tuple with additional fruit names
more_fruits = ("orange", "grape", "apple")

# Concatenate the two tuples
all_fruits = fruits + more_fruits

# Print the concatenated tuple and the count of
"apple"
```

```
print(all_fruits)  # Output: ('apple', 'banana',
'apple', 'cherry', 'orange', 'grape', 'apple')
print(all_fruits.count("apple"))  # Output: 3
```

Repetition (*)

To repeat the elements of an entire tuple for a specified number of times, use the operator asterisk (*).

```
# Define a tuple with some fruit names
fruits = ("apple", "banana", "apple", "cherry")

# Repeat the tuple multiple times
repeated_fruits = fruits * 2  # The tuple will be
repeated twice

# Print the repeated tuple and the count of
"apple"
print(repeated_fruits)
# Output: ('apple', 'banana', 'apple', 'cherry',
'apple', 'banana', 'apple', 'cherry')
```

Membership Operators

The membership method using "in" returns True if an element is found in the tuple and returns False if it does not exist. The membership method using "not in" returns True if an element is not found in the tuple and returns False if it does exist, as shown in the example below:

```
# Define a tuple
fruits = ("apple", "banana", "cherry", "banana",
"orange")

# Check if an element is in the tuple
print("apple" in fruits)  # Output: True

# Check if an element is not in the tuple
```

```
print("grape" not in fruits)  # Output: True
```

Slicing (:)

To extract a portion of the tuple using slicing syntax, use the colon operator (:).

```
# Define a tuple with some fruit names
fruits = ("apple", "banana", "apple", "cherry",
"orange", "grape")

# Slice the tuple to get a subset of elements
sliced_fruits = fruits[1:4]  # Get elements from
index 1 to 3 (4 is excluded)

# Print the sliced tuple and the count of "apple"
print(sliced_fruits)
# Output: ('banana', 'apple', 'cherry')
```

count()

The count() method returns the number of occurrences of a specific element.

```
# Define a tuple with some fruit names
fruits = ("apple", "banana", "apple", "cherry")
# Count occurrences of an element
apple_count = fruits.count("apple")
print(apple_count)  # Output: 2
```

sorted ()

To sort a tuple by the number sequence or alphabetical sequence, use the method sorted(). It's important to note that the sorted() function returns a list, not a tuple, so the result of sorting is a list. To convert the sorted result back to a tuple, you can use the tuple() function.

```
# Define a tuple with some fruit names
```

```
fruits = ("cherry", "banana", "apple", "grape",
"apple", "orange")

# Sort the tuple using sorted() (it returns a
list)
sorted_fruits = sorted(fruits)

# Print the sorted list and the count of "apple"
print(sorted_fruits)
# Output: ['apple', 'apple', 'banana', 'cherry',
'grape', 'orange']

# Identifying the type of data generated by the
sorted() method.
print(type(sorted_fruits)) # Output: <class
'list'>
```

Here is an example of numerical ordering:

```
# Define a tuple with numbers
numbers = (42, 7, 19, 3, 99, 8)

# Sort the tuple using sorted() (it returns a
list)
sorted_numbers = sorted(numbers)

# Print the sorted list and the count of the
number 7
print(sorted_numbers) # Output: [3, 7, 8, 19, 42,
99]

# Identifying the type of data generated by the
sorted() method.
print(type(sorted_numbers)) # Output: <class
'list'>
```

Converting Back to a Tuple

Since the result of the sorted() method is a list, it is possible to use the reverse() method on it. It is also possible to convert the list back to a tuple with the tuple() method. To do this, add the following code to the end of the previous example:

```
sorted_numbers_tuple = tuple(sorted_numbers)
print(sorted_numbers_tuple)
# Output: (3, 7, 8, 19, 42, 99)
```

Sum (sum())

To calculate the sum of all numeric elements in the tuple, use the method sum().

```
numbers = (1, 2, 3, 4)
print(sum(numbers))  # Output: 10
```

min() and max()

Like in lists, the methods max() and min() are very useful for finding the smallest or largest element in a tuple.

```
numbers = (10, 20, 5, 40)
print(min(numbers))  # Output: 5
print(max(numbers))  # Output: 40
```

For strings, min() and max() compare lexicographically unless a key is supplied. To compare by length, use key=len; if several strings have the same length, the first one encountered is returned.

```
# Define a tuple with some fruit names
fruits = ("apple", "banana", "apple", "cherry",
"orange", "grape")

# Find and print smallest element in the tuple
print(min(fruits, key=len))  # Output: apple
```

```
# With key=len, min() returns the shortest string;
ties are resolved by first occurrence.

# Find and print the largest element in the tuple
print(max(fruits, key=len))  # Output: banana
# With key=len, max() returns the longest string;
ties are resolved by first occurrence.
```

Nested Tuples

Tuples can contain other tuples, and elements in nested tuples can be accessed using multiple indexes.

```
# Accessing elements in a nested tuple
nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True,
False))

# Access the second element of the first tuple
print(nested_tuple[0][1])  # Output: 2

# Access the first element of the second tuple
print(nested_tuple[1][0])  # Output: a
```

Converting a Tuple to a List

If you need to modify a tuple, you can convert it to a list first.

```
# Convert tuple to list
fruits_tuple = ("apple", "banana", "cherry")
fruits_list = list(fruits_tuple)

# Modify the list
fruits_list[1] = "orange"

# Print the modified list
print(fruits_list)  # Output: ['apple', 'orange',
'cherry']
```

```
# Creating a new tuple with modifications
fruits = ("apple", "banana", "cherry")
fruits = fruits + ("orange",)
print(fruits)  # Output: ('apple', 'banana',
'cherry', 'orange')
```

Operations Between Different Tuples

Because tuples are immutable, the possible operations between two tuples are more restricted than between lists. Python offers two main types of operations between two tuples: concatenation and comparison.

Concatenation (+)

Concatenation joins two tuples into a new one.

```
tuple_1 = ('a', 1, 'b')
tuple_2 = (2, 'c', 3)
tuple_3 = tuple_1 + tuple_2
print(tuple_3)  # Output: ('a', 1, 'b', 2, 'c', 3)
```

Comparisons (==, !=, <, >)

Comparison is done element by element, from left to right.

```
tuple_1 = ('a', 10, 20)
tuple_2 = ('a', 10, 30)

# Comparing tuple_1 and tuple_2
print(f"Comparing {tuple_1} and {tuple_2}:")
print(tuple_1 == tuple_2) # False, because 20 is
not equal to 30
print(tuple_1 != tuple_2) # True, because 20 is
not equal to 30
print(tuple_1 < tuple_2)  # True, because 20 is
less than 30
```

```
print(tuple_1 > tuple_2)  # False, because 20 is
not greater than 30
print(tuple_1 <= tuple_2) # True, because 20 is
less than 30
print(tuple_1 >= tuple_2) # False, because 20 is
not greater than 30
```

Set

Sets can store a collection of unique and immutable elements of different types, such as str, int, or bool. To create a set, simply assign a variable to a set contained in curly brackets with the values separated by commas. Sets do not maintain the order of elements and do not allow duplicate values; therefore, the printed order shown in the examples is illustrative and may vary. Their main purpose is to handle collections where you need to perform mathematical set operations or quickly check for the presence of an item.

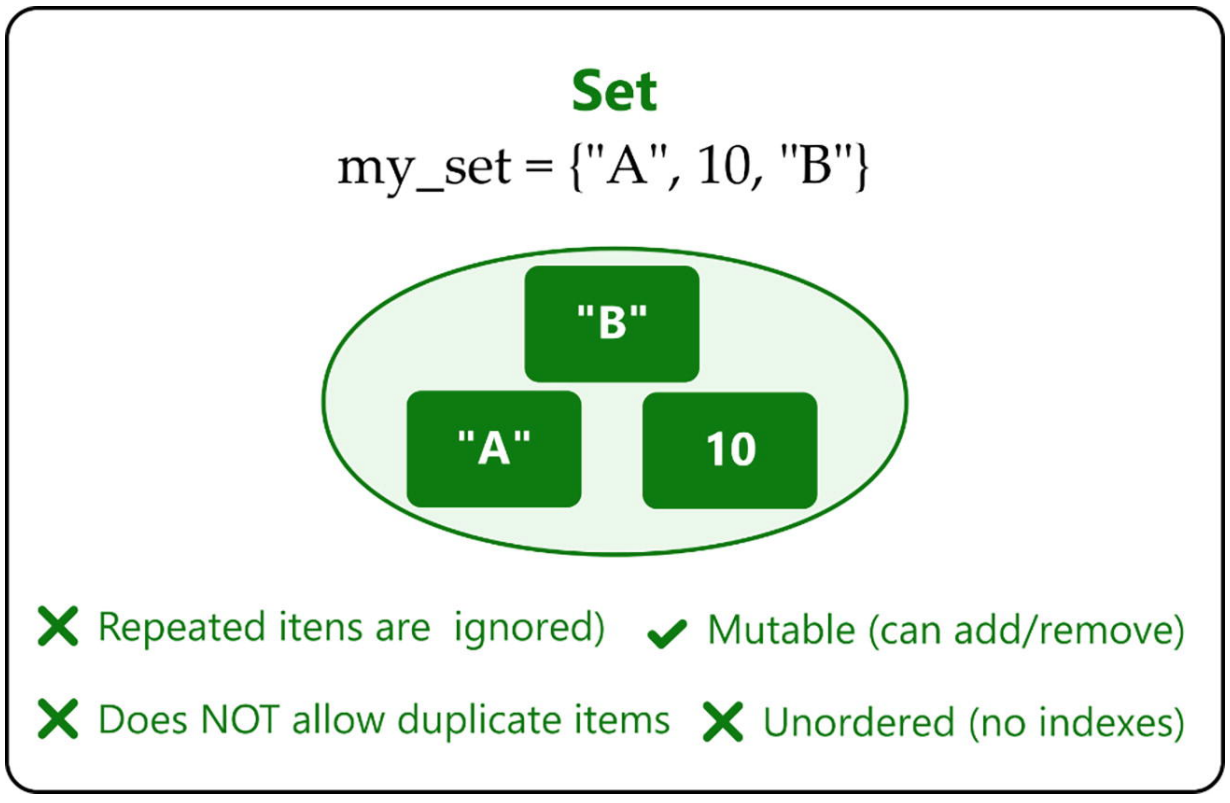


Figure 3-3 Diagram of a Python set, illustrating its unordered nature and storage of unique elements

It is possible to create a set using curly braces {}, as represented in the code below:

```
# A set of numbers
numbers = {1, 2, 3, 4, 5}
print(numbers)  # Example (order may vary): {1, 2,
3, 4, 5}

# A set of strings
fruits = {"apple", "banana", "cherry", "orange"}
print(fruits)  # Example (order may vary):
{'apple', 'banana', 'cherry', 'orange'}

# A set with different types of elements
mixed_set = {42, "hello", 3.14, True}
print(mixed_set)  # Example (order may vary):
{True, 42, 'hello', 3.14}

# Creating a set from different iterables
set_from_list = set([1, 2, 3, 4, 5])
set_from_tuple = set((1, 2, 3, 4, 5))
```

You can also create a set using the set() constructor, which is particularly useful for converting other data types, like lists, into sets. This process automatically removes duplicates, making it a convenient way to get unique elements:

```
# Creating a set from a list with duplicates
my_list = ["apple", "banana", "apple", "cherry"]
fruits = set(my_list)
print(fruits) # Example (order may vary):
{'cherry', 'apple', 'banana'}
```

Common Operations with Sets

Sets in Python support various operations for modifying their contents. You can add a single element using the .add() method. To remove elements, .remove() deletes an item but raises a KeyError if the item doesn't exist, whereas .discard() removes an item without raising an error if it's not found. Additionally, .clear() removes all elements from the set, leaving it empty.

```
my_list = ["apple", "banana", "apple", "cherry"]
my_set = set(my_list)

print("Original Set:", my_set)  # Example (order
may vary): {'apple', 'banana', 'cherry'}

# Add an element
my_set.add("mango")
print("After adding 'mango':", my_set) # Example
(order may vary):
{'apple','banana','cherry','mango'}

# Remove an element
my_set.remove("banana")
print("After removing 'banana':", my_set)  #
Example (order may vary): {'apple', 'cherry',
'mango'}

# Remove an element with .discard()
my_set.discard("apple")
print("After discarding 'apple':", my_set)  #
Example (order may vary): {'cherry', 'mango'}

# Discard a non-existent element (no error)
my_set.discard("grape")
print("After discarding 'grape':", my_set)  #
Example (order may vary): {'cherry', 'mango'}
```

```
# Clear the set
my_set.clear()
print("After clearing the set:", my_set)  #
Output: set()
```

Adding and Removing Elements

To add an element to a set, use the add() method. To remove an element, use the remove() or discard() method.

```
# Adding and removing elements
fruits = {"apple", "banana", "cherry"}
fruits.add("orange")  # Add an element
print(fruits)  # Example (order may vary):
{'apple', 'banana', 'cherry', 'orange'}

fruits.remove("banana")  # Remove an element
print(fruits)  # Example (order may vary):
{'apple', 'cherry', 'orange'}
```

Note that remove() raises a KeyError if the element is not found, while discard() does not raise an error.

Updating Elements

Since sets only store unique elements, you can't update an element in the classical sense. However, you can remove an element and add a new one.

```
# Updating elements
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")  # Remove an element
fruits.add("orange")  # Add a new element
print(fruits)  # Example (order may vary):
{'apple', 'cherry', 'orange'}
```

Operations Between Sets

Several mathematical operations can be performed on sets. The union operation (|) combines two sets, excluding duplicates. The intersection operation (&) returns elements present in both sets. The difference operation (-) returns elements in the first set, but not in the second. Finally, the symmetric difference operation (^) returns elements in either set, but not in their intersection.

Table 3-7 Mathematical set operations

See the example using these four operations:

```
# Define two sets of fruits
fruits_a = {"apple", "banana", "cherry", "date"}
fruits_b = {"cherry", "date", "elderberry", "fig"}

# Union
union_fruits = fruits_a | fruits_b
print("Union:", union_fruits)
# Example (order may vary): {'apple', 'banana',
'cherry', 'date', 'elderberry', 'fig'}

# Intersection
intersection_fruits = fruits_a & fruits_b
print("Intersection:", intersection_fruits)
# Example (order may vary): {'cherry', 'date'}

# Difference
difference_fruits = fruits_a - fruits_b
print("Difference (fruits_a - fruits_b):",
difference_fruits)
# Example (order may vary): {'apple', 'banana'}

# Symmetric Difference
symmetric_difference_fruits = fruits_a ^ fruits_b
```

```
print("Symmetric Difference:",
symmetric_difference_fruits)
# Example (order may vary): {'apple', 'banana',
'elderberry', 'fig'}
```

Dictionary

A Python dictionary is a mutable collection that stores data in key–value pairs, allowing you to retrieve data using meaningful identifiers per key instead of numeric indexes. Dictionaries have unique keys, which can appear only once and, if reassigned, replace the old value. They are ordered and support adding, changing, and removing key–value pairs. Keys must be immutable types, such as strings, numbers, or tuples, while values can be any data type.

Use a dictionary in Python when you need to associate related pieces of information, look up data based on a unique identifier, or count the frequency of items in a sequence. Dictionaries are particularly useful for storing and retrieving data with meaningful keys, making them one of the most important and versatile data structures in Python. In engineering applications, dictionaries are valuable for mapping sensor readings to equipment IDs, storing configuration data with descriptive keys, parsing JSON data, and counting event occurrences.

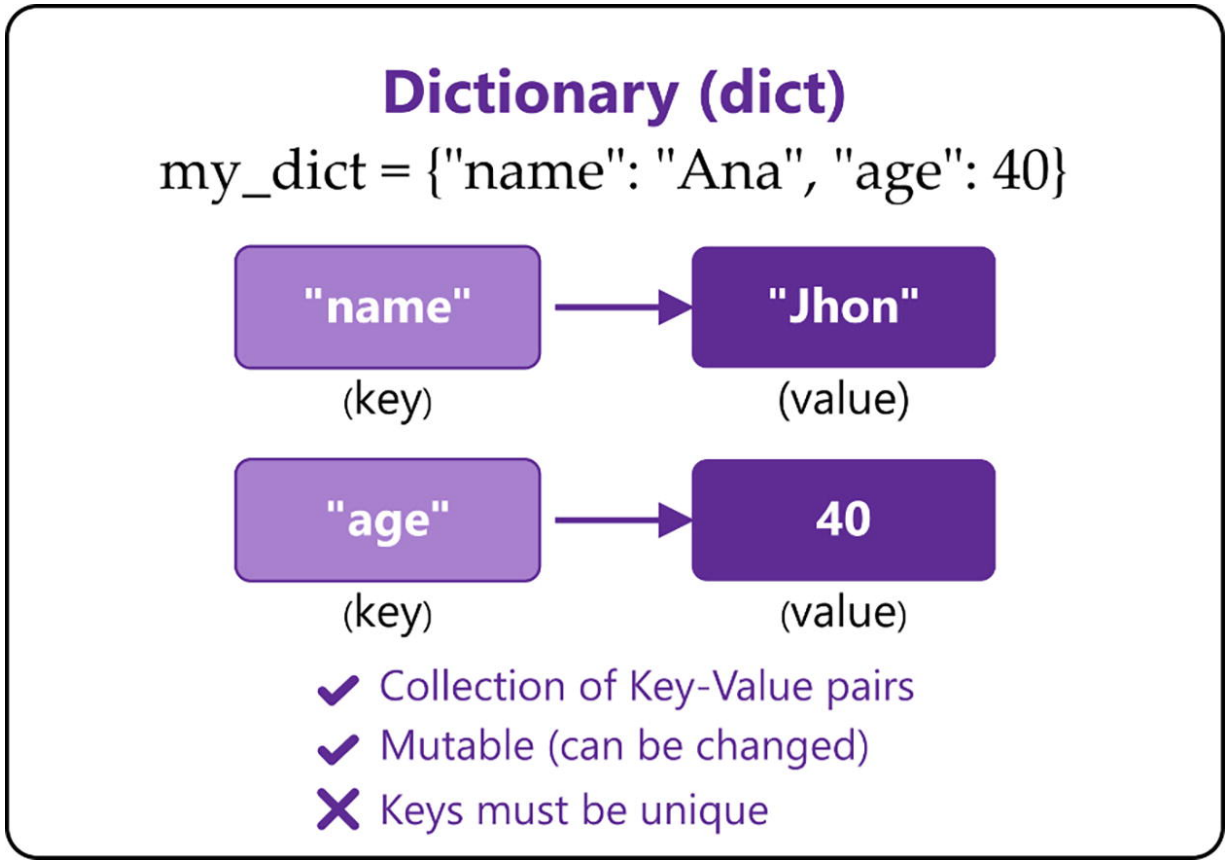


Figure 3-4 Diagram of a Python dictionary, showing the key–value pair structure and its core properties

Creating Dictionaries

A dictionary is created using curly braces {}.

```
# A dictionary representing electric motor
specifications
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
```

Now see the keys and their corresponding values from the example code in Table 3-8.

Table 3-8 Example of dictionary key–value pairs

Accessing and Modifying Data

Accessing a dictionary value is done by querying its key between square brackets []. This is useful for retrieving a specific parameter for calculations.

```
# A dictionary representing electric motor
specifications
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
# Get the motor's RPM for a performance
calculation
speed = motor_specs["rpm"]
print(f"Motor speed: {speed} RPM")  # Output:
Motor speed: 1500 RPM
```

A safer way to access a dictionary is to use the .get() method. This is important for handling components that may not have optional parameters, preventing your program from crashing, such as when handling keys that don't exist in the dictionary.

```
# A dictionary representing electric motor
specifications
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
```

```
# Safely get the torque, providing a default of
0.0 if it's not specified
torque = motor_specs.get("torque_nm", 0.0)
print(f"Motor torque: {torque} Nm") # Output:
Motor torque: 0.0 Nm
```

The code above defines a dictionary motor_specs containing specifications for an electric motor. It then attempts to retrieve the value associated with the key "torque_nm" using the .get() method.

The .get() method is used to safely retrieve a value from a dictionary. If the key is not found, it returns a default value instead of raising a KeyError. In this case, the default value is 0.0.

Since "torque_nm" is not a key in the motor_specs dictionary, the .get() method returns the default value 0.0, which is then assigned to the torque variable. This way, the motor torque is printed on the screen as 0.0 Nm, as the "torque_nm" key was not found in the dictionary.

Common Operations in a Dictionary

Python provides several operations for handling dictionaries; see the most used ones in Table 3-9.

Table 3-9 Common operations and methods for dictionaries

Add/Update an Item

Adding or updating a key–value pair in a dictionary is a simple operation. You can add a new key–value pair or update an existing key's value by assigning a new value to the key. If the key already exists, its value will be overwritten. This operation is useful when you need to add new data or update existing data in a dictionary.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
```

```
    "power_rating_kw": 1.5
}
motor_specs["efficiency"] = 0.85  # Add new key-
value pair
motor_specs["voltage"] = 230  # Update existing
key's value
print(motor_specs)  # Output: {'part_number': 'MX-
2001', 'voltage': 230, 'rpm': 1500,
'power_rating_kw': 1.5, 'efficiency': 0.85}
```

Delete an Item

Removing a key-value pair from a dictionary can be done using the del statement. If the key doesn't exist in the dictionary, a KeyError is raised. This operation is useful when you need to remove unnecessary data from a dictionary.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
del motor_specs["rpm"]
print(motor_specs)  # Output: {'part_number': 'MX-
2001', 'voltage': 220, 'power_rating_kw': 1.5}
```

Check for a Key

When working with dictionaries, it's often necessary to check if a key exists before trying to access its value. This can be done using the "in" keyword. By checking if a key exists, you can avoid raising a KeyError and instead handle the situation as needed. This operation is useful when you're not sure if a key is present in the dictionary.

```
motor_specs = {
    "part_number": "MX-2001",
```

```
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
if "rpm" in motor_specs:
    print("RPM is specified")  # Output: RPM is
specified
```

Get Length

The len() function can be used to get the number of key–value pairs in a dictionary. This can be useful when you need to know the size of the dictionary or iterate over it. By knowing the length of the dictionary, you can perform operations that depend on the number of key–value pairs.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
num_specs = len(motor_specs)
print(f"Number of specs: {num_specs}")  # Output:
Number of specs: 4
```

get()

The get() method allows you to safely retrieve a value from a dictionary without raising a KeyError if the key is not found. Instead, you can specify a default value to return if the key is not present. This operation is useful when you're not sure if a key exists in the dictionary and want to provide a fallback value.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
```

```
    "rpm": 1500,
    "power_rating_kw": 1.5
}
speed = motor_specs.get("rpm", "Not specified")
print(f"Motor speed: {speed} RPM")  # Output:
Motor speed: 1500 RPM
```

keys()

The keys() method returns a view object that displays a list of all the keys in the dictionary. This view object is dynamic, meaning it reflects changes to the dictionary. You can use this method to iterate over the keys of the dictionary or perform other operations that require access to the keys.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
spec_keys = motor_specs.keys()
print(spec_keys)  # Output:
dict_keys(['part_number', 'voltage', 'rpm',
'power_rating_kw'])
```

values()

The values() method returns a view object that displays a list of all the values in the dictionary. Like the keys() method, this view object is dynamic and reflects changes to the dictionary. You can use this method to iterate over the values of the dictionary or perform other operations that require access to the values.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
```

```
    "rpm": 1500,
    "power_rating_kw": 1.5
}
spec_values = motor_specs.values()
print(spec_values)  # Output: dict_values(['MX-
2001', 220, 1500, 1.5])
```

items()

The items() method returns a view object that displays a list of all key–value pairs in the dictionary as tuples. This view object is dynamic and reflects changes to the dictionary. You can use this method to iterate over the key–value pairs of the dictionary or perform other operations that require access to both the keys and values.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
spec_items = motor_specs.items()
print(spec_items)  # Output:
dict_items([('part_number', 'MX-2001'),
('voltage', 220), ('rpm', 1500),
('power_rating_kw', 1.5)])
```

update()

The update() method allows you to merge another dictionary or iterable into the current dictionary. If a key already exists in the dictionary, its value will be updated with the new value. This operation is useful when you need to combine data from multiple sources.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
```

```
    "rpm": 1500,
    "power_rating_kw": 1.5
}
additional_specs = {
    "efficiency": 0.85,
    "weight_kg": 10
}
motor_specs.update(additional_specs)
print(motor_specs)  # Output: {'part_number': 'MX-
2001', 'voltage': 220, 'rpm': 1500,
'power_rating_kw': 1.5, 'efficiency': 0.85,
'weight_kg': 10}
```

pop()

The pop() method removes and returns the value of a specified key. If the key doesn't exist in the dictionary, a KeyError is raised. When a key may be absent, use pop(key, default) to avoid the exception.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
removed_spec = motor_specs.pop("rpm", None)
print(f"Removed spec: {removed_spec}")  # Output:
Removed spec: 1500
```

popitem()

The popitem() method removes and returns the last inserted key–value pair as a tuple. This operation is useful when you need to remove and retrieve the most recently added data from a dictionary.

```
motor_specs = {
    "part_number": "MX-2001",
```

```
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
removed_spec = motor_specs.popitem()
print(f"Removed spec: {removed_spec}")  # Output:
Removed spec: ('power_rating_kw', 1.5)
```

clear()

The clear() method removes all items from the dictionary, leaving it empty. This operation is useful when you need to reset a dictionary to an empty state.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
motor_specs.clear()
print(motor_specs)  # Output: {}
```

copy()

The copy() method creates a shallow copy of the dictionary, which means it creates a new dictionary with references to the original values. This operation is useful when you need to create a copy of a dictionary without modifying the original.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
copied_specs = motor_specs.copy()
```

```
print(copied_specs)  # Output: {'part_number':
'MX-2001', 'voltage': 220, 'rpm': 1500,
'power_rating_kw': 1.5}
```

setdefault()

The setdefault() method returns the value of a key if it exists in the dictionary. If the key doesn't exist, it inserts the key with a default value and returns the default value. This operation is useful when you need to retrieve a value from a dictionary and provide a default value if the key is not present.

```
motor_specs = {
    "part_number": "MX-2001",
    "voltage": 220,
    "rpm": 1500,
    "power_rating_kw": 1.5
}
efficiency = motor_specs.setdefault("efficiency",
0.85)
print(efficiency)  # Output: 0.85
```

Comprehension

Dictionary comprehensions provide a concise way to create dictionaries from iterables. You can use a dictionary comprehension to create a new dictionary with a specific structure and data.

Dictionary comprehension is a slightly more advanced topic, so you may want to study Chapter 6 before diving into this important Python feature. By familiarizing yourself with these foundational concepts, you'll be better equipped to understand and effectively utilize dictionary comprehensions in your own code.

```
motor_specs = {f"param_{i}": i*10 for i in
range(5)}
print(motor_specs)
```

```
# Output: {'param_0': 0, 'param_1': 10, 'param_2':
20, 'param_3': 30, 'param_4':   40}
```

Just as with list comprehension, the code above uses a dictionary comprehension to create a dictionary called motor_specs.

Here's a breakdown of what's happening:

- range(5): This generates a sequence of numbers from 0 to 4.
- f"param_{i}": This is a formatted string literal (f-string) that creates a string key for each iteration. The {i} is replaced by the current value of i from the range.
- i*10: This calculates the value for each key–value pair. The value is the current value of i multiplied by 10.
- {... for i in range(5)}: This is dictionary comprehension syntax. It creates a new dictionary with the key–value pairs generated by the iteration.

So when you run this code, it creates a dictionary with five key–value pairs:

param_0 with value 0 (0 * 10)
param_1 with value 10 (1 * 10)
param_2 with value 20 (2 * 10)
param_3 with value 30 (3 * 10)
param_4 with value 40 (4 * 10)

The resulting dictionary is then printed to the console.

Operations Between Dictionaries

There are several operations to perform when working with two dictionaries. The dictionary union operator | requires Python 3.9 or later; in earlier versions, use update() or dictionary unpacking.

Table 3-10 Set-like operations between dictionaries

A union merges two dictionaries—if any key overlaps, the value from the second dictionary is retained—while an intersection finds the

keys that exist in both dictionaries. To do this, a set intersection is performed on the dictionary keys, and then a new dictionary is constructed from the result.

A difference finds the keys that are in the first dictionary but not the second. This is useful for identifying which parameters were not included in a dictionary update.

A symmetric difference finds the keys that are in one or the other dictionary, but not both. It identifies the parameters that are unique to each dataset.

```
dict1 = {'a': 1, 'b': 2, 'c': 3}
dict2 = {'b': 20, 'c': 30, 'd': 40}

# Union (works directly)
union_dict = dict1 | dict2
print("Union:", union_dict)
# Output: {'a': 1, 'b': 20, 'c': 30, 'd': 40}

# Intersection (on keys)
intersection_dict = {k: dict1[k] for k in
dict1.keys() & dict2.keys()}
print("Intersection:", intersection_dict)
# Output: {'b': 2, 'c': 3}

# Difference (keys in dict1 but not in dict2)
difference_dict = {k: dict1[k] for k in
dict1.keys() - dict2.keys()}
print("Difference (dict1 - dict2):",
difference_dict)
# Output: {'a': 1}

# Symmetric Difference (keys in either but not
both)
symmetric_difference_dict = {
    k: dict1[k] if k in dict1 else dict2[k]
```

```
    for k in dict1.keys() ^ dict2.keys()
}
print("Symmetric Difference:",
symmetric_difference_dict)
# Output: {'a': 1, 'd': 40}
```

Practice Exercises

1. *Predict the Output—Understanding List Indexing*:
   Without running the code, analyze the instructions and write down what you expect each line to print.

   ```
   items = [10, 20, 30, 40, 50]
   print(items[0])
   print(items[-1])
   print(items[2])
   print(items[1:4])
   ```

2. *Create and Print Three Different Lists*:
   Create a list with integers, a list with mixed elements, and a nested list. Then print each with a descriptive sentence.
3. *Modify an Existing List*:
   Using fruits = ['apple', 'banana', 'cherry'], replace 'banana' with 'kiwi', append 'grape', and print the final list.
4. *Asking the User—Membership Test*:
   Using a predefined list of fruits, ask the user for input and test if the fruit exists in the list.
5. *Sorting Comparison—Understanding sort() vs. sorted()*:
   Demonstrate the difference using data = [5, 1, 9, 3, 7].
6. *Predict and Explain—Assignment vs. Copy*:
   Predict what prints and explain why:

```
A = [1, 2, 3]
B = A
B.append(4)
print(A, B)
```

7. *List Methods Practice*:
   Apply append(), remove(), insert(), and pop() on numbers = [10,20,30,40] and print results.
8. *Eliminating Duplicates with Sets*:
   Convert values = [2,2,5,7,7,9,9,9] to a set and then back to sorted list.
9. *Accessing Nested Collections*:
   Access 'b' from nested = [[1,2,3],['a','b','c'],[True,False]].
10. *Filtering Using List Comprehension*:
    Create a filtered list of fruits containing 'e'.
11. *Predict the Output—Understanding Tuple Indexing*:
    Tuples store values in a fixed order, and each element can be accessed using its index just like lists.
    Before running the code below, observe it carefully and predict what each print() statement will display.
    Negative indexing allows accessing elements from the end of the tuple.

```
colors = ("red", "green", "blue", "yellow")

print(colors[1])      # Positive index
print(colors[-2])     # Negative index
```

    After writing your prediction, run the code and compare the result.
    If your prediction was different, explain what part of indexing logic you misunderstood. This reflection helps reinforce the

concept.

12. *Fix the Tuple Modification Error*:

    Tuples in Python are immutable, which means their elements cannot be changed directly after creation.

    The code below tries to modify an element and therefore causes an error:

```
t = ("apple", "banana", "cherry")
t[1] = "orange"
```

    a. Explain in your own words why this code raises an error.

    b. Rewrite the code to achieve the same desired result ("banana" → "orange") using this strategy:

        i. Convert the tuple to a list.

        ii. Modify the desired element in the list.

        iii. Convert the list back to a tuple.

        iv. Print the new tuple and verify the change.

13. *Working with Tuple Operations*:

    Tuples are useful to represent groups of values that should not be modified.

    a. Create a tuple called numbers containing at least four integers.

    b. Use the * operator to create a new tuple that repeats numbers three times.

    c. Create another tuple called extra with a few additional values.

d. Concatenate the repeated tuple with extra using the + operator and store the result in a new variable.

Print

a. The original tuple
b. The repeated tuple

c. The final concatenated tuple

Include clear labels in your output so that each result is easy to identify.

14. *Convert a Tuple*:

Sometimes you need to "change" data that was originally stored in a tuple. A common solution is to temporarily convert the tuple to a list.

Consider

```
shape = ("circle", "square", "triangle")
```

a. Convert the tuple shape to a list.
b. Replace the value "square" with "rectangle".
c. Convert the list back into a tuple and store the result in a new variable, for example, new_shape.

Print

a. The original tuple
b. The modified tuple

Explain briefly what changed and why this approach is

necessary when working with tuples.

15. *Set Operations*:

    Sets store unique values and are ideal for membership and mathematical operations.

    Using the initial set

```
colors = {"red", "blue", "green"}
```

    do the following:

    a. Add the color "yellow" to the set using .add().

    b. Remove the color "blue" using .remove().

    c. Try to remove the color "pink" using .discard() and notice that no error is raised, even if the element does not exist.

    d. Add comments to your code explaining the difference between .remove() and .discard().

    e. Print the set after each modification with descriptive messages.

16. *Mathematical Set Relationships—Union, Intersection, Difference, Symmetric Difference*:

    Sets provide powerful mathematical operations to compare collections of values.

    Consider

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
```

    Using set operators, compute and print

    a. *Union (*A | B*)*: All elements that appear in A or B

    b.

*Intersection (*A & B*)*: Elements that appear in both A and B

c. *Difference (*A - B*)*: Elements that are in A but not in B

d. *Symmetric Difference (*A ^ B*)*: Elements that are in A or B, but not in both at the same time

For each operation:

a. Store the result in a variable.

b. Print the name of the operation and the resulting set.

c. Add a short explanation as a comment describing what that operation represents.

17. *Accessing Dictionary Values Safely—Using get() with Default Values*:

Dictionaries store data as key–value pairs. Sometimes, a key may not exist, and direct access would raise an error. The .get() method allows safe access with a default value.

Consider

```
motor = {"voltage": 220, "rpm": 1500,
"power_kw": 1.5}
```

a. Access and print the value associated with the "rpm" key using motor["rpm"].

b. Access the "torque" key using motor.get("torque", "Not available") and print the result.

c. Explain briefly (in a comment or sentence) why using .get() is safer than accessing the key directly when you are not sure it exists.

18. *Updating a Dictionary—Adding and Modifying Key–Value Pairs*:

Dictionaries are mutable and can be expanded or updated as the program runs.

Using again:

```
motor = {"voltage": 220, "rpm": 1500,
"power_kw": 1.5}
```

a. Print the original dictionary.

b. Add a new key "efficiency" with the value 0.85.

c. Update the value of "voltage" from 220 to 230.

d. Print the dictionary after the modifications, highlighting the changes.

e. Write a short comment explaining the difference between adding a new key and updating an existing key.

19. *Inspecting a Dictionary*:

To better understand what is stored in a dictionary, it is useful to inspect its keys, values, and key–value pairs separately.

Using the motor dictionary from the previous exercise:

```
motor = {"voltage": 220, "rpm": 1500,
"power_kw": 1.5}
```

1. Use motor.keys() to get and print all keys.

2. Use motor.values() to get and print all values.

3. Use motor.items() to get and print all key–value pairs.

20. *Final Mini-project—Simple Product Inventory System with Dictionaries*:

In this mini-project, you will simulate a small inventory system using a dictionary, where

- The keys are product names
- The values are their quantities in stock

Your program must

1. Start with a dictionary containing at least five products and their quantities
2. Ask the user for the name and quantity of a new product and add it to the dictionary
3. Allow the user to update the quantity of an existing product (e.g., by entering the product name and the new quantity)
4. Allow the user to remove a product from the dictionary using del or .pop()
5. At the end, print a clear and organized summary of the inventory, such as

```
Current Inventory:
- apples: 10 units
- oranges: 5 units
- bananas: 12 units
...
```

Use meaningful variable names and friendly messages so the user clearly understands what to enter and what the program is doing.

Answer Key

1. *Predict the Output—Understanding List Indexing*:

   Code:

```
items = [10, 20, 30, 40, 50]
print(items[0])
print(items[-1])
print(items[2])
print(items[1:4])
```

   Output:

```
10
50
30
[20, 30, 40]
```

   Explanation:

- items[0] → first element → **10**
- items[-1] → last element → **50**
- items[2] → index 2 → **30**
- items[1:4] → from index 1 to 3 → **[20, 30, 40]**

2. *Create and Print Three Different Lists*:

   Example Solution:

```
int_list = [1, 2, 3, 4]
mixed_list = ["Alice", 25, 1.75, True]
nested_list = [[1, 2], ["a", "b"], [True,
False]]

print("Integer list:", int_list)
print("Mixed data list:", mixed_list)
print("Nested list:", nested_list)
```

Multiple answers are acceptable, as long as they include

- One integer list
- One mixed-type list
- One nested list

3. *Modify an Existing List*:

```
fruits = ["apple", "banana", "cherry"]

fruits[1] = "kiwi"      # replace
fruits.append("grape")  # add at the end

print(fruits)
```

Final Result:

```
['apple', 'kiwi', 'cherry', 'grape']
```

4. *Asking the User—Membership Test*:
   Example Solution:

```
fruits = ["apple", "banana", "orange",
"mango"]
user_fruit = input("Enter a fruit name: ")

# The tuple stores the messages:
# Index 0 (False): Fruit not found
# Index 1 (True): Fruit found
messages = (
    "Sorry, this fruit is not available.",
    "Yes, we have that fruit!"
)
# The result of the expression (user_fruit in
fruits) is True (1) or False (0),
# which is used to select the message directly
```

```
from the tuple.
print(messages[user_fruit in fruits])
```

5. *Sorting Comparison*:

   Example:

```
data = [5, 1, 9, 3, 7]

# Using sort() - modifies the original list
data.sort()
print("After sort():", data)

# Using sorted() - returns a new sorted list,
original can be kept
data2 = [5, 1, 9, 3, 7]
sorted_data = sorted(data2)
print("Original data2:", data2)
print("sorted(data2):", sorted_data)
Possible output:
After sort(): [1, 3, 5, 7, 9]
Original data2: [5, 1, 9, 3, 7]
sorted(data2): [1, 3, 5, 7, 9]
```

6. *Predict and Explain—Assignment vs. Copy*:

   Code:

```
A = [1, 2, 3]
B = A
B.append(4)
print(A, B)
```

   Output:

```
[1, 2, 3, 4] [1, 2, 3, 4]
```

Explanation:

B = A does not create a copy—it only creates another reference to the same list.

Changing B also changes A.

7. *List Methods Practice*:

Example:

```
numbers = [10, 20, 30, 40]
numbers.append(50)
print("After append:", numbers)         # [10,
20, 30, 40, 50]

numbers.remove(20)
print("After remove(20):", numbers)     # [10,
30, 40, 50]

numbers.insert(1, 25)
print("After insert(1, 25):", numbers) # [10,
25, 30, 40, 50]

popped = numbers.pop()
print("After pop():", numbers)          # [10,
25, 30, 40]
print("Popped value:", popped)          # 50
```

Exact values may differ depending on inserted/removed elements, but logic must match.

8. *Eliminating Duplicates with Sets*:

```
values = [2, 2, 5, 7, 7, 9, 9, 9]

unique_set = set(values)
unique_list = sorted(unique_set)
```

```
print("Original list:", values)
print("Unique sorted list:", unique_list)
```

Output:

```
Original list: [2, 2, 5, 7, 7, 9, 9, 9]
Unique sorted list: [2, 5, 7, 9]
```

9. *Accessing Nested Collections*:

```
nested = [[1, 2, 3], ["a", "b", "c"], [True,
False]]

value = nested[1][1]
print(value)
```

Output:

```
b
```

- nested[1] → ["a", "b", "c"]
- nested[1][1] → "b"

10. *Filtering Using List Comprehension*:
    Example:

```
fruits = ["apple", "banana", "cherry", "kiwi",
"pear"]

filtered = [fruit for fruit in fruits if "e"
in fruit]
print("Original list:", fruits)
print("Fruits containing 'e':", filtered)
```

Output:

```
Original list: ['apple', 'banana', 'cherry',
```

```
'kiwi', 'pear']
Fruits containing 'e': ['apple', 'cherry',
'pear']
```

11. *Predict the Output—Tuple Indexing*:

```
colors = ("red", "green", "blue", "yellow")

print(colors[1])     # Positive index
print(colors[-2])    # Negative index
```

Output:

```
green
blue
```

12. *Fix the Tuple Modification Error*:
Original Code (Produces Error):

```
t = ("apple", "banana", "cherry")
t[1] = "orange"
Corrected solution:
t = ("apple", "banana", "cherry")

# Convert to list
temp = list(t)

# Modify element
temp[1] = "orange"

# Convert back to tuple
t = tuple(temp)

print(t)
```

Output:

```
('apple', 'orange', 'cherry')
```

Explanation:

Tuples are immutable—modification requires converting to a list.

13. *Working with Tuple Operations*:

Output:

```
numbers = (1, 2, 3, 4)

repeated = numbers * 3
extra = (5, 6)

combined = repeated + extra
print("Original tuple:", numbers)
print("Repeated tuple:", repeated)
print("Final concatenated tuple:", combined)

Original tuple: (1, 2, 3, 4)
Repeated tuple: (1, 2, 3, 4, 1, 2, 3, 4, 1, 2,
3, 4)
Final concatenated tuple: (1, 2, 3, 4, 1, 2,
3, 4, 1, 2, 3, 4, 5, 6)
```

14. *Convert a Tuple*:

```
shape = ("circle", "square", "triangle")
print("Original tuple:", shape)

shape_list = list(shape)
shape_list[1] = "rectangle"
new_shape = tuple(shape_list)

print("Modified tuple:", new_shape)
```

Output:

```
Original tuple: ('circle', 'square',
'triangle')

Modified tuple: ('circle', 'rectangle',
'triangle')
```

15. *Set Operations*:

```
colors = {"red", "blue", "green"}
colors.add("yellow")
print("After add('yellow'):", colors)

colors.remove("blue")
print("After remove('blue'):", colors)

colors.discard("pink")  # no error even if
'pink' is not in the set
print("After discard('pink'):", colors)
```

Key Notes:

- .remove("x") → raises KeyError if "x" is missing
- .discard("x") → does **not** raise an error

Order may vary because sets are unordered.

16. *Mathematical Set Relationships*:

```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}

union_ab = A | B
intersection_ab = A & B
difference_ab = A - B
sym_diff_ab = A ^ B
```

```
print("Union (A | B):", union_ab)
print("Intersection (A & B):",
intersection_ab)
print("Difference (A - B):", difference_ab)


print("Symmetric difference (A ^ B):",
sym_diff_ab)
```

Comments:
Union: Elements in A or B
Intersection: Elements in both A and B
Difference: Elements only in A
Symmetric Difference: Elements in A or B, but not both

17. *Accessing Dictionary Values Safely—get()*:

```
motor = {"voltage": 220, "rpm": 1500,
"power_kw": 1.5}

rpm_value = motor["rpm"]
print("Motor speed (rpm):", rpm_value)

torque_value = motor.get("torque", "Not
available")
print("Torque specification:", torque_value)
```

Explanation:
motor["torque"] would raise a KeyError, but .get() safely returns a default value.

18. *Updating a Dictionary—Adding and Modifying*:

```
motor = {"voltage": 220, "rpm": 1500,
"power_kw": 1.5}
print("Original dictionary:", motor)
motor["efficiency"] = 0.85      # add new key
```

```
motor["voltage"] = 230          # update
existing key

print("Updated dictionary:", motor)
```

Output:

```
Original dictionary: {'voltage': 220, 'rpm':
1500, 'power_kw': 1.5}
Updated dictionary: {'voltage': 230, 'rpm':
1500, 'power_kw': 1.5, 'efficiency': 0.85}
```

Rule:

- If key exists → value is overwritten.
- If key does not exist → new entry is created.

19. *Inspecting a Dictionary*:

```
motor = {"voltage": 230, "rpm": 1500,
"power_kw": 1.5, "efficiency": 0.85}

print("Keys:", list(motor.keys()))
print("Values:", list(motor.values()))
print("Items:", list(motor.items()))

for key, value in motor.items():
    print(f"Parameter: {key} -> Value:
{value}")
```

Typical Output (Order May Vary):

```
Keys: ['voltage', 'rpm', 'power_kw',
'efficiency']
Values: [230, 1500, 1.5, 0.85]
Items: [('voltage', 230), ('rpm', 1500),
('power_kw', 1.5), ('efficiency', 0.85)]
```

```
Parameter: voltage -> Value: 230
Parameter: rpm -> Value: 1500
Parameter: power_kw -> Value: 1.5
Parameter: efficiency -> Value: 0.85
```

20. *Final Mini-project—Inventory System*:
    Example Solution (Many Variations Are Valid):

```
# Initial inventory (Requirement 1: uses a
dictionary with 5 products)
inventory = {
    "apples": 10,
    "oranges": 5,
    "bananas": 12,
    "grapes": 8,
    "pears": 6
}

print("Initial inventory:", .inventory)

# 1. Add new product (Requirement 2: Adds a
new product)
new_product = input("Enter new product name to
add: ")
new_quantity = int(input(f"Enter quantity for
{new_product}: "))
# Direct assignment adds a new item
inventory[new_product] = new_quantity
print(f"Confirmation: Added {new_product} to
inventory.")

# 2. Update existing product (Requirement 3:
Updates an existing product)
product_to_update = input("Enter an existing
product to update: ")
new_qty = int(input(f"Enter new quantity for
{product_to_update}: "))
```

```
{product_to_update}.") ))
# Direct assignment updates the value if the
key exists (or adds it if it doesn't)
inventory[product_to_update] = new_qty
print(f"Confirmation: Updated quantity for
{product_to_update}.")

# 3. Remove a product (Requirement 4: Removes
a product)
product_to_remove = input("Enter a product to
remove: ")
# The pop() method removes and returns the
quantity (or None if not found)
removed_qty = inventory.pop(product_to_remove,
None)

# TUPLE INDEXING TRICK TO REPLACE IF/ELSE
# The result of the expression 'removed_qty is
not None' is True (1) or False (0)
messages = (
    f"Product '{product_to_remove}' not found,
nothing removed.",  # Index 0 (False)
    f"Removed '{product_to_remove}' from
inventory."              # Index 1 (True)
)
# We use the boolean result as an index to
select the correct message
print(messages[removed_qty is not None])

# 4. Final report (Requirement 5: Prints a
clear final report)
print("\nCurrent Inventory:")
# Prints the dictionary directly, without
using a 'for' loop
print(inventory)
```

Requirements for Correctness:

- Uses a dictionary with at least five products
- Adds a new product
- Updates an existing product
- Removes a product

- Prints a clear final report

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_4

4. Python Modules and Packages

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Modules and packages are organizational structures in Python that enable efficient code management across multiple programs, files, and directories. By promoting a modular programming approach, they allow developers to organize, reuse, and distribute code with simplicity and maintainability.

A Python module is a single file containing functions, classes, variables, and other code elements. Modules help organize code by separating supplementary parts from the main code, providing better project structure and organization.

A package is a collection of modules grouped together in a directory or subdirectories, providing additional functionalities to Python programs. The Python community offers a large number of packages that are available for a wide variety of applications. Table 4-1 lists some Python packages and their applications.

Table 4-1 Examples of Python packages by application

It's important to note that there are many more Python packages available by the Python community. In order to find an updated list of Python packages, visit the PyPI website at `https://pypi.org/`.

The appearance of the PyPI website is like Figure 4-1.

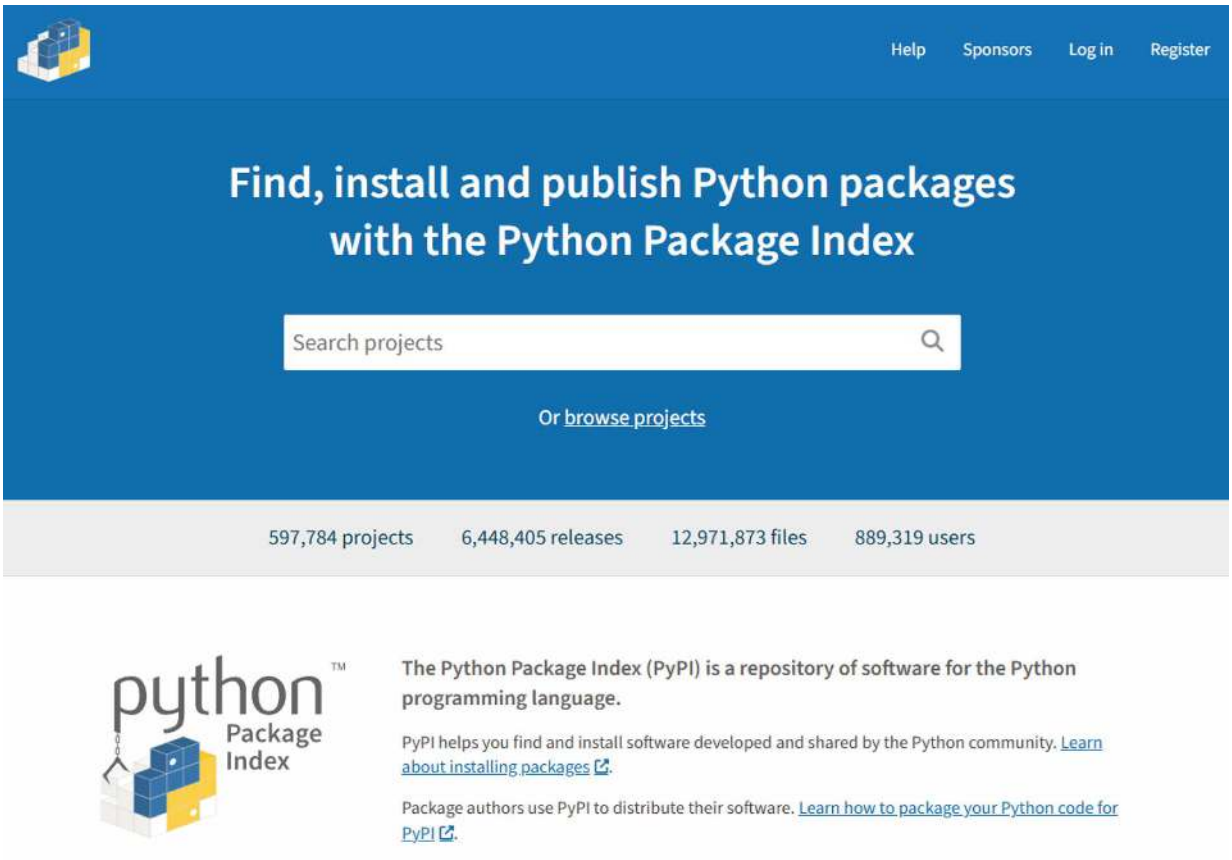


Figure 4-1 The Python Package Index (PyPI) website homepage

Python Path and Modules Location

Before studying the types and the use of modules, it is important to know the Python path also called sys.path that is a list with directories that Python searches to find modules and packages.

To access the current Python path list, use the code below:

```
# Importing the 'sys' module, which provides access to system-specific parameters and functions
import sys

# Printing the list of directories where Python searches for modules to import
# 'sys.path' contains the paths that Python checks to locate imported modules
print(sys.path)
```

The output in the terminal console will be a list of various locations that Python uses to search for modules and packages. These paths are a location recorded in the environment variables where Python is installed. It is important to note that custom

modules located in the project's root directory are automatically included in the Python path. Therefore, keeping custom modules in that directory allows you to use them without modifying environment variables. It is also possible to dynamically add a custom module path via code, as shown in the cross-platform example below:

```
# Importing the 'sys' module to interact with the Python
interpreter
import sys

# Importing the 'os' module to interact with the operating
system
import os

# Constructing a custom path in a platform-independent way
using 'os.path.join'
# This ensures compatibility across different operating systems
(Windows, Linux, macOS)
custom_path = os.path.join("home", "user", "my_project")  #
Joins 3 directory names into a valid path

# Adding the custom path to the Python module search path
(sys.path)
# This allows Python to locate and import modules from the
specified directory
sys.path.append(custom_path)
```

An important point when working with packages is the "__init__.py" file. This file allows to assign various functionalities to a package, as described below:

- Executing initialization code
- Controlling what is exported by the package
- Organizing subpackages
- Indicating that the directory is a package

The "__init__.py" file can be empty, but it is possible to define code that initializes every time the package is imported:

```
# __init__.py
print("Package is being initialized")
# Initialize package-level variables
VERSION = '1.0.0'
```

It is possible to define which modules and names should be available when using the import * package. To do this, use the __all__ statement before the modules inside square brackets:

```
# __init__.py
__all__ = ['function1', 'function2']
```

If your package contains subpackages, you can use __init__.py to manage imports and initialization for the subpackages. For example, alongside the package structure shown in Figure 4-2, here is the code used to initialize the subpackage:

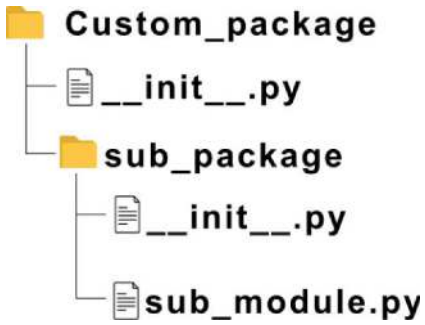


Figure 4-2 Package directory requiring __init__.py in older Python versions

```
import my_package.sub_package.sub_module
```

In Python versions lower than 3.3, the presence of the "__init__.py" file indicates that a directory should be treated as a Python package; this happens even if the file is empty.



Figure 4-3 Package directory with __init__.py

Starting with Python 3.3, the __init__.py file is no longer strictly necessary to mark a directory as a package.

Types of Python Modules

There are several types of Python modules that can be used to organize the code, expand its functionalities, or improve its execution speed. The main types of Python modules are listed as follows:

1. Built-in modules
2. Standard library modules
3. Third-party modules
4. Dynamic modules

5. Custom modules
6. Compiled modules

Built-In Modules

Python's built-in modules come pre-installed with the interpreter, eliminating the need for separate installation. As part of the standard library, they're often coded in C or other low-level languages for optimal performance. This tight integration with the Python runtime enables direct interaction with the system or hardware.

The only requirement before using the module is to import it using the import command, as in the example below:

```
import sys

# Print the version of the Python interpreter
print("Python version:", sys.version)

# Print the path where Python looks for modules
print("Module search paths:", sys.path)

# Exit the program with status code 0 (success)
sys.exit(0)
```

This code demonstrates how to use Python's built-in sys module to interact with the interpreter: it imports sys, prints the current Python version using sys.version, and shows the list of directories where Python searches for modules through sys.path. The program then terminates using sys.exit(0), with 0 indicating successful execution, while non-zero values typically signify errors.

Standard Library Modules

The standard library is a collection of codes with many functionalities for several tasks and comes with the Python distribution, but are not loaded into memory by default nor are they included in the core of Python like built-in modules. To use standard library modules, import them explicitly using an import statement, as shown in the example below.

```
import datetime  # Import the datetime module for working with
dates and times

# Get the current date and time
current_datetime = datetime.datetime.now()
```

```
# Print the complete current date and time
print("Current date and time:", current_datetime)

# Extract and print the current date
print("Current date:", current_datetime.date())

# Extract and print the current time
print("Current time:", current_datetime.time())
```

Third-Party Modules

Third-party modules are usually modules to specific tasks and are not part of the standard library. Generally they are developed by the Python community and always need to be installed. There are several ways to install them; the main ways are listed as follows:

1. Package manager like pip, apt, yum, or brew
2. Using anaconda by the terminal or anaconda navigator
3. From source code of the package
4. Using a dependency manager called Poetry
5. Using a .whl file

Package Manager

Table 4-2 lists the package managers and the operating systems that can be used. Additionally, a column is provided with examples for installing a common Python package called NumPy. To install other modules, simply replace the name "numpy" with the desired module.

Table 4-2 Comparison of common package managers

The following examples demonstrate how to use pip across different operating systems to install numpy a library designed for numerical computations and array operations.

To use the pip command, open the Command Prompt, PowerShell, or terminal, and execute the command corresponding to your operating system:

Windows

In command prompt or PowerShell, write the command below:

```
pip install numpy
```

Linux
In the terminal write

```
sudo pip install numpy or sudo pip3 install numpy
```

The pip3 command is the package installer for Python 3. Alternatively, you can use system package managers depending on your Linux distribution, as shown below:

- *Ubuntu/Debian*: sudo apt install python3-numpy
- *Red Hat/CentOS*: sudo yum install python3-numpy
- *Fedora*: sudo dnf install python3-numpy
- *Arch Linux*: sudo pacman -S python-numpy

macOS
To install a package in macOS, open Terminal and write the command below:

```
pip install numpy
```

Anaconda

Another option of package installation is using anaconda, an open source distribution platform for Python. Through anaconda it is possible to install several Python versions and packages that can be managed by an interface called anaconda navigator or by the command line in command prompt, PowerShell, or the terminal. Anaconda is available on Windows, macOS, or Linux.

Source Code

You can install packages directly from their source code hosted on a version control platform like GitHub or Bitbucket or by downloading the source code archive from the project's official website or repository. Below are some examples:

1. *Using Git Repositories*:
   You can clone the repository using Git and install the package manually using these commands:

   ```
   git clone https://github.com/username/repository.git
   cd repository
   pip install .
   ```

   For example, to install the NumPy package, use these commands:

   ```
   # Clone the NumPy repository from GitHub
   git clone https://github.com/numpy/numpy.git

   # Navigate into the cloned repository
   cd numpy
   ```

```
# Install NumPy in editable/development mode
pip install -e .
```

2. *Using pip with Git and Other Version Control Systems*:
   You can install directly from a Git repository using pip.
   The syntax is

```
pip install git+https://github.com/username/repository.git
```

   An example is

```
pip install git+https://github.com/numpy/numpy.git
```

   pip supports other version control systems like Subversion, Mercurial, and Bazaar.

3. *Source Archives*:
   Download the source code as a .zip or .tar.gz file, extract it, and install it.
   The syntax is

```
wget https://example.com/package.tar.gz
tar -xvzf package.tar.gz
cd package
pip install .
```

   An example to install NumPy is

```
# Download the source archive (example using wget)
wget
https://github.com/numpy/numpy/archive/refs/heads/main.tar.gz

# Extract the archive
tar -xvzf main.tar.gz

# Navigate into the extracted directory
cd numpy-main

# Install NumPy
pip install .
```

4. *Local Source Code*:
   If you have the source code locally, you can install it directly:

```
cd /path/to/source/code
pip install .
```

For the NumPy example, use

```
# Navigate to the directory where the source code is located
cd /path/to/numpy/source/code

# Install NumPy using setup.py
pip install .
```

5. *Editable Install (*pip install -e*)*:

   For development purposes, you can install a package in "editable" mode, allowing you to make changes to the source code and use them without reinstalling:

```
pip install -e /path/to/source/code
```

Using Poetry

Poetry is a dependency management and packaging tool for Python that simplifies the process of managing project dependencies, creating virtual environments, and building and publishing Python packages. With Poetry, it is possible to install Python packages from source code. As this topic is quite extensive and beyond the scope of this book, for more details, visit the project's website at https://python-poetry.org/, which contains comprehensive documentation.

Using a Wheel File

A wheel file is a binary distribution format for Python packages with the extension .whl. It is essentially a prebuilt package that can be installed using pip, Python's package manager. .whl files are part of the Python wheel standard, which is designed to make package installation faster and more efficient compared to building packages from source.

While .whl files are primarily designed to be installed with pip, they can also be installed using other Python package managers that support the wheel standard, such as Poetry or Conda.

To install the .whl file, use this command:

```
pip install package_name.whl
```

Dynamic Modules

Dynamic modules are modules that are loaded or created at runtime and are not statically linked during compilation or initial program loading. They can be created or modified while a program is running and loaded only when needed, which can improve startup time and memory use in large applications.

To create a dynamic module, follow these steps:

1. Import sys and types, which provide access to the module cache and utilities for creating module objects.

```
import sys, types
```

2. Specify a name for the new module as a string.

```
module_name = "my_dynamic_module"
```

3. Use types.ModuleType to create a module object.

```
dynamic_module = types.ModuleType(module_name)
```

4. Add variables, functions, or other attributes to the module.

```
dynamic_module.greeting = "Hello, Dynamic Module!"  # Adding
a variable
dynamic_module.say_hello = lambda name: f"Hello, {name}!"  #
Adding a function
```

5. Register the module in sys.modules so that Python's import system can find it.

```
sys.modules[module_name] = dynamic_module
```

6. The dynamic module can now be imported and used.

An example of a dynamic module can be seen below:

```
import sys, types  # Import the modules required to create and
register a module

# Create a module dynamically
module_name = "dynamic_module"  # Define the name of the
dynamic module as a string.
dynamic_module = types.ModuleType(module_name)  # Create a new
module object with the specified name.
dynamic_module.greeting = "Hello, World!"  # Add an attribute
'greeting' to the dynamic module with a string value.

# Register the dynamically created module in the import cache
sys.modules[module_name] = dynamic_module
```

```
# Import and use the dynamic module
import dynamic_module  # Import the newly created module by its
name.
print(dynamic_module.greeting)  # Access and print the
'greeting' attribute. Output: Hello, World!
```

Custom Modules

Customized modules are created by the user and can be used in other Python files; this type of module is created from a Python code and used in the main.py file or other files.

To create a custom module in Python, create a .py file such as custom_module.py and define the variables or functions that will be reused. The module can then be imported by other Python files when it is in the project directory, inside an importable package, or in a directory listed in sys.path.



Figure 4-4 Custom module in the same directory as the main script

The code below is an example of a custom module:

```
# Define a constant for the year
YEAR = 2025  # Represents the year, which can be used elsewhere
in the program

# Define a constant for the name
NAME = "John"  # Represents a name, which can be used as an
identifier or label

# Function to calculate the sum of two numbers
def sum(a, b):
    """
    Adds two numbers and returns the result.
     """
    return a + b

# Function to calculate the difference between two numbers
def sub(a, b):
    """
    Subtracts the second number from the first and returns the
result.
     """
```

```
    return a - b
```

To use the content of the custom module, just import in the code like the example below:

```
# Importing the custom module created earlier
import custom_module

# Accessing and printing the constant YEAR from the custom
module
print(custom_module.YEAR)  # Output: 2025

# Using the sum function from the custom module to add two
numbers
x = custom_module.sum(2, 3)  # Calls the sum function with
arguments 2 and 3
print(x)  # Output: 5

# Using the sub function from the custom module to subtract two
numbers
y = custom_module.sub(5, 3)  # Calls the sub function with
arguments 5 and 3
print(y)  # Output: 2
```

The next pages will cover the forms and types of importing modules and packages.

Compiled Modules

Python supports the creation and use of modules written in other languages such as C, C++, and Rust. This allows developers to leverage the performance and features of these languages while integrating seamlessly with Python. An example of creation of a compiled module using C++ is listed below.

To build a new Python module with C++, the following dependencies are required:

– Install setuptools as the build backend.
– Use a compatible C++ compiler such as GCC, Clang, MinGW, or Microsoft Visual C++.

Setuptools is a Python library that supports the development and packaging of Python projects. To install it on your computer and verify the installation, use the following commands:

On Windows, use command prompt or PowerShell:

```
python -m pip install --upgrade setuptools
```

To verify the installation, use this command—this should show a message with the setuptools module version:

```
python -m pip show setuptools
```

On Linux use the terminal to write the installation command:

```
python3 -m pip install --upgrade setuptools
```

To verify the installation on Linux, write

```
python3 -m pip show setuptools
```

On macOS use Terminal to write

```
python3 -m pip install --upgrade setuptools
```

To verify the installation on macOS, write

```
python3 -m pip show setuptools
```

After installing setuptools and verifying the C++ compiler, follow the sequence below:

1. *Write the C++ Code*:
   Create the C++ source file that contains the extension code, for example, Compiled_module.cpp.
2. *Create the pyproject.toml File*:
   Declare setuptools as the build backend using the following configuration:

   ```
   [build-system]
   requires = ["setuptools"]
   build-backend = "setuptools.build_meta"
   ```

3. *Build the Extension*:
   From the directory containing Compiled_module.cpp and pyproject.toml, install the project in editable mode. This lets pip invoke the configured build backend instead of running setup.py directly:
   Windows

   ```
   py -m pip install -e .
   ```

   Linux

   ```
   sudo apt update && sudo apt install build-essential
   python3 -m pip install -e .
   ```

   macOS

```
xcode-select --install  # Install Xcode command-line tools

python3 -m pip install -e .
```

4. *Use the New Module*:
   Import the compiled module in Python and call the function defined in the C++ extension.
5. Run the Python code with the new module imported.

Importing Modules

Modules and submodules can be imported in several ways. The best method depends on which names are needed and where the module or package is located.

Table 4-3	Basic module import methods

Table 4-4	Additional module import methods

Table 4-5	Advanced and special-case import methods

Importing a specific name from a module (e.g., from math import sqrt) does not normally reduce the module's memory use compared with import math. In both cases, Python loads and caches the module. The main difference is namespace binding: import math requires math.sqrt, whereas from math import sqrt binds sqrt directly.

The following example confirms that importing a specific function still loads and caches the math module. sys.getsizeof should not be used to compare total import memory because it reports only the shallow size of an object:

```
import sys
from math import sqrt

# The full math module has been loaded and cached
print("math" in sys.modules)  # Output: True
print(sqrt(16))  # Output: 4.0
```

Two common styles are importing a module object and importing selected names from it. The first style is shown below:

```
# Importing the entire math module, which provides a wide range
of mathematical functions
import math
```

```
# Using the math.sqrt function to calculate the square root of
16
# math.sqrt is part of the math module and can be accessed
using the 'math.' prefix
x = math.sqrt(16)

# Printing the result of the square root calculation
# The result of math.sqrt(16) is 4.0, since 4 * 4 = 16
print(x)  # Output: 4.0
```

To import a module, write import followed by the module name.

The imported module is accessed through its name and a period, as in math.sqrt(16).

A wildcard import, such as from math import *, binds the module's public names directly in the current namespace. Because this can hide where names came from or create collisions, explicit imports are generally clearer.

```
# Importing all functions and constants from the math module
from math import *

# Using sqrt directly, without the need for the 'math.' prefix
x = sqrt(16)

# Printing the result of the square root calculation
# The result of sqrt(16) is 4.0, since 4 * 4 = 16
print(x)  # Output: 4.0
```

To control which modules are exposed to wildcard imports, use the __all__ variable. This is a list of strings where each string represents the name of a variable, function, class, or module that you want to expose to wildcard imports.

```
# Example of wildcard imports in a custom module.
__all__ = ["sum", "sub"]  # Only these will be imported with
`*`

YEAR = 2025
NAME = "John"

def sum (a, b):
    return a + b

def sub (a, b):
    return a - b
```

It is also possible to import selected names with the from command. For example, from math import sqrt binds sqrt directly in the current namespace. This can improve

readability, but it does not mean that only part of the module is loaded.

```
# Importing only the sqrt function from the math module
from math import sqrt

# Using sqrt directly, without the 'math.' prefix
x = sqrt(16)

# Printing the result of the square root calculation
# The result of sqrt(16) is 4.0, since 4 * 4 = 16
print(x)  # Output: 4.0
```

To import several components of a module, use commas "," to separate the components.

```
# Importing specific components from the math module
from math import sqrt, pi

# Using sqrt to calculate the square root of 25
result = sqrt(25)
# Using pi to calculate the circumference of a circle with
radius 5
radius = 5
circumference = 2 * pi * radius

# Printing the results
print("The square root of 25 is:", result) # Output: The square
root of 25 is: 5.0
print("The circumference is:", circumference) # Output: The
circumference is: 31.41592653589793
```

In order to write a shorter name of the module or submodule, it is possible to import the module with an alias using the keyword "as".

```
import module_name as alias  # Syntax for assigning an alias to
the module name
```

An example can be seen below:

```
# Importing the math module with an alias 'm'
import math as m

# Using the alias 'm' to call the sqrt() function from the math
module
result = m.sqrt(16)  # Calculates the square root of 16
```

```
# Printing the result
print("The square root of 16 is:", result) # Output:The square
root of 16 is: 4.0
```

The keyword "as" also can be used with specific components.

```
from math import sqrt as square_root
```

Dynamic Import

Python can import a module at runtime instead of at the beginning of a script. This is called a dynamic import and is useful when the module depends on user input, configuration, or another runtime condition. Use importlib.import_module() to load the required module.

```
# Importing the importlib module
import importlib

# Dynamically importing a module based on a condition
module_name = 'math'  # This could be user input or a
configuration value
math_module = importlib.import_module(module_name)

# Using the dynamically imported module
result = math_module.sqrt(16)  # Calculates the square root of
16
print("The square root of 16 is:", result) # Output: The square
root of 16 is: 4.0
```

Relative and Absolute Import

Python provides absolute and relative imports, which differ in how a module or package is referenced. Absolute imports can use either the import or from ... import syntax, whereas explicit relative imports use from ... import.

Relative imports refer to modules within the current package hierarchy and are commonly used to organize packages and subpackages.

Relative imports use leading dots to indicate the package level. One dot refers to the current package, two dots move one level up, and each additional dot moves up one more level.

```
from .package import module_A  # Current package
from ..package import module_B  # One level up
from ...package import module_C  # Two levels up
```

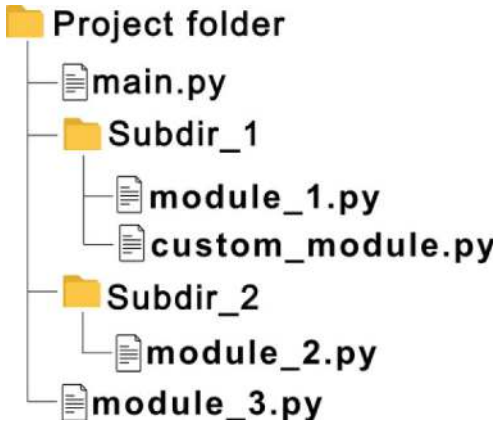


Figure 4-5 Directory hierarchy for relative and absolute import examples

Based on the directory structure shown in Figure 4-5, relative imports can be used from custom_module.py to access modules in other directories within the project hierarchy.

Table 4-6 Examples of relative imports

To understand how Python imports modules, it's important to know the directories it searches, which are listed in sys.path. Python locates modules based on their type and location. User-created modules are typically found in the current project directory. Built-in modules are compiled into the Python interpreter itself, while standard library modules and third-party modules installed via package managers like pip are stored in designated directories within your Python environment.

The location of the built-in modules can be found with the code below:

```
import sys
import os

# Check if a module is built-in
print(sys.builtin_module_names)  # List of all built-in modules

# Example: Check if 'os' is built-in
print('sys' in sys.builtin_module_names)  # True if built-in

# Get the location of a module
print(os.__file__)  # Will raise an AttributeError if it's
built-in
```

An **absolute import** specifies the full path to the module or package starting from the root of the project or package hierarchy.

Table 4-7 Examples of absolute imports

The code of custom_module.py is written below:

```
# Example of the code of custom_module.py
YEAR = 2025
NAME = "John"

def sum (a, b):
    return a + b
```

Third-party modules are installed in the site-packages directory. To find their location, use the command below:

```
python -m site
```

To find the location of a specific third-party module, use this code as an example:

```
import module_name  # Replace 'module_name' with the module
name

# Get the location of the module
print(module_name.__file__)
```

If the module being searched for is not found, an error message should appear accompanied by some other information:

```
> ModuleNotFoundError: No module named 'numpy'
```

In the case of custom modules, the interpreter of Python searches for these modules in the directories listed in sys.path, generally in the project directory. To find out the location of sys.path, just type and run the code below:

```
import sys

# List all directories where Python looks for modules
print(sys.path)
```

Importing Custom Modules in Different Directories

Customs modules need to be located in the same directory of main code, but it is possible to call and utilize these modules inside different subdirectories as shown in Figure 4-6.

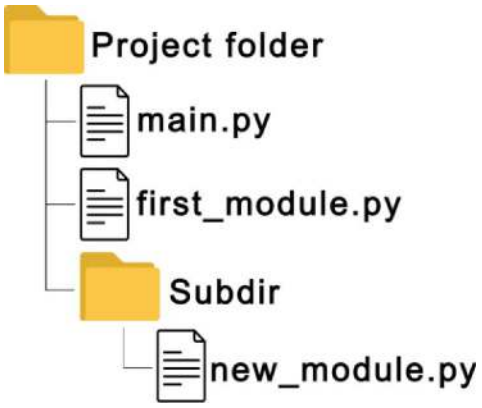


Figure 4-6 Importing a module from a subdirectory

In this example the file first_module.py is located in the same directory of main.py; in this case, first_module.py can be accessed by main.py, just using the import command.

To import the file new_module.py to use in main.py, only the import command is not enough; in this case it is necessary to use some options as follows:

1. Adding the directory to sys.path
2. Using relative import
3. Using an Import Hook with importlib

To add a subdirectory containing Python modules to sys.path, use the sys module as shown below. This relative path is resolved from the current working directory:

```
import sys
sys.path.append('subdir')  # Add the folder to sys.path

import custom_module  # Import the custom module
custom_module.my_function()  # Now is possible to call a
function from the module
```

To build the path relative to the current script instead, use this cross-platform version:

```
import sys
import os

# Dynamically construct the path to the 'subdir' folder
subdir_path = os.path.join(os.path.dirname(__file__), 'subdir')

# Add the folder to sys.path
sys.path.append(subdir_path)
```

```
# Import the custom module
import custom_module

# Call a function from the module
custom_module.my_function()
```

The "os" module provides functions that enable the creation of cross-platform paths. The function "os.path.join" combines the home directory path with another directory, while the function "os.path.dirname(__file__)" returns the directory of the current script, ensuring that the subdir folder is correctly located relative to the script. This approach makes it possible to use external folders on multiple platforms, such as Windows, Linux, and macOS.

For a module stored outside the project, add its absolute directory to sys.path. pathlib provides a concise, cross-platform way to construct the path:

```
import sys
from pathlib import Path

modules_dir = Path.home() / "Documents" / "Python_modules"
sys.path.append(str(modules_dir))

from custom_module import sum
x = sum(2, 3)
print(x)  # Output: 5
```

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_5

5. Functions

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Functions are a common feature across programming languages in computer programming. They enhance the readability, reusability, and modularity of Python code.

. Functions are blocks of reusable code that perform a specific task. They can be called by Python code and used an unlimited number of times.

The syntax of a Python function is illustrated in Figure 5-1.

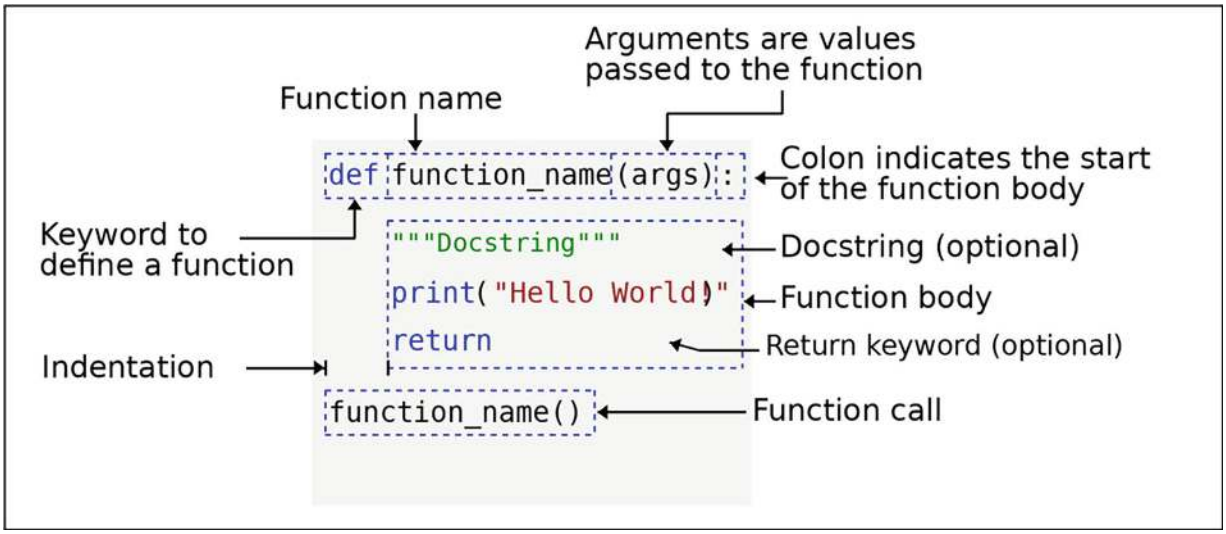


Figure 5-1 The syntax of a Python function, illustrating its core components such as the def keyword, function name, arguments, and body

This is an example of a function in Python:

```
# Function definition.
def greet():
    # This function prints a greeting message
"Hello!" when called.
    print("Hello World!")

# This calls the greet() function, which executes
the code inside it.
greet() # Output: Hello World!
```

To define a function in Python, start with the “def” keyword, followed by the function name and a pair of parentheses. End the line with a colon “:”. The function body, which contains the code that performs the function’s task, must typically be indented with four spaces. This indentation is essential as it indicates where the function’s code begins and ends.

Since Python executes code sequentially, line by line, from top to bottom, a function must be defined before it is called. If a function is called before its definition, Python will raise a NameError because it does not yet know what the function is.

Optionally, a function can return a value to its caller. This is especially useful when the result of the function needs to be used in the code outside the function. To return a value, use the return keyword, followed by the variable or expression you want to send back to the function call. The value returned by the function will be assigned to the function call, allowing the function call to be used as a variable holding the returned value. A function may also begin with a docstring enclosed in triple quotes. Ordinary comments are introduced with the # character.

```
# Function definition with return value.
def greet():
```

```
    # This function prints a greeting message
"Hello!" when called.
    print("Hello")  # Prints "Hello" to the
console.
    return("World!")  # Returns the string
"World!" to the caller.

# This calls the greet() function, which executes
the code inside it.
returned_value = greet()  # The function is
called, "Hello" is printed, and "World!" is
returned.
# Output: Hello!

# The returned value ("World!") is stored in the
variable 'returned_value'.
print(returned_value)  # Prints the value stored
in 'returned_value', which is "World!"
# Output: World!
```

Parameters and Arguments

When defining a function, you can declare parameters that receive the arguments supplied when the function is called.

Parameters and arguments work together in a function. In the example below, the variable “name” is the parameter of the function, while the value “Carlos” is the argument passed to the function during the function call. Any type of data can be used as a parameter or argument in a function, including built-in types such as int, float, list, set, dict, and even custom objects:

```
# Function definition with a parameter.
def greet_user(name):
    """
```

```
    This function takes one parameter, 'name', and
prints a personalized greeting message.
    """
    # Print a greeting message using the provided
'name' parameter.
    print(f" Hello {name} !")  # Output: " Hello
Carlos !"
    # Return statement (not necessary in this
case, as it doesn't return any value).
    return()

# Function call with the argument "Carlos".
# This calls the 'greet_user' function and passes
the string "Carlos" as the 'name' parameter
greet_user("Carlos")  # Output: " Hello Carlos !"
```

The **return** statement is used inside a function to exit the function immediately and optionally send a value back to the caller of the function.

In the following example, the **return** keyword is used to exit the function and return a specific value:

```
def process_numbers():
    for i in range(1, 6):  # Loop through numbers
from 1 to 5.
        if i == 3:         # Check if the current
number is 3.
            return         # Exit the function
when i is 3.
        else:              # If the number is not
3.
            print(i)       # Print the current
number.
            # Output: 1, 2 (before the function
exits)
```

```
process_numbers()
```

In this example, the **return** keyword is used to send a value back to the calling function:

```
# Define a function that adds two numbers and
returns the result
def add_numbers(a, b):
    return a + b  # Return the sum of a and b

# Call the function and store the result
result = add_numbers(3, 5)

# Print the result
print(result)  # Output: 8
```

Table 5-1 outlines the different ways to pass arguments to a function. These methods offer flexibility in defining and calling functions, making Python functions highly adaptable to a wide range of use cases.

Table 5-1 An overview of different methods for passing arguments to functions in Python, including positional, keyword, and variable-length arguments

Positional Arguments

A function can have multiple parameters and accept multiple arguments at the same time. To define or pass them, simply separate each parameter or argument with a comma, as shown in the example below. In this case, it is important to use the parameters in the same order as the arguments. If the order of the parameters is changed, the function will not correctly assign the arguments to the corresponding parameters, which can lead to unexpected results. This method of passing arguments is known as positional arguments:

```
# Function definition with multiple parameters
```

```
(Positional Arguments).
def greet_user(name, age, location):
    """
    This function takes three parameters: 'name',
'age', and 'location',
    and prints a personalized greeting message.
    """
    # Print a greeting message using the provided
parameters.
    print(f"Hello {name}! You are {age} years old
and live in {location}.")
    # Return statement (not necessary in this
case, as it doesn't return any value).
    return()

# Function call with multiple arguments.
# This calls the 'greet_user' function and passes
values for 'name', 'age', and 'location'.
greet_user("John", 25, "New York")
# Output: Hello John! You are 25 years old and
live in New York.

# Calling the function again and passing new
arguments.
greet_user("Alice", 30, "London")
# Output: Hello Alice! You are 30 years old and
live in London.
```

Keyword Arguments

Arguments can be passed by explicitly specifying the parameter names, regardless of their order or sequence. To achieve this, assign a value to each parameter name when calling the function, as shown in the example below:

```
# Define a function with two parameters: 'name'
```

```
and 'age'.
def greet(name, age):
    # Print a greeting message using the provided
'name' and 'age' arguments.
    print(f"Hello {name}, you are {age} years
old.")

# Call the 'greet' function using keyword
arguments.
# Here, the argument 'age' is explicitly assigned
the value 25,
# and the argument 'name' is explicitly assigned
the value "John".
greet(age=25, name="John")  # Output: Hello John,
you are 25 years old.
```

Default Arguments

Parameters can have default values assigned in the function definition. These default values are used if no argument is provided for the corresponding parameter. In the example below, the “age” parameter is a default argument, making it an optional parameter with a default value of 30:

```
# Define a function where 'name' is a required
parameter, and 'age' is an optional.
def greet(name, age=30):
    # Print a greeting message using the provided
'name' and 'age' arguments.
    print(f"Hello {name}, you are {age} years
old.")

# Call the 'greet' function with only the 'name'
argument.
# Since 'age' is not provided, the default value
of 30 is used.
```

```
greet("John")  # Output: Hello John, you are 30
years old.

# Call the 'greet' function with both 'name' and
'age' arguments.
# The default value of 'age' is overridden with
the provided value of 25.
greet("Alice", 25)  # Output: Hello Alice, you are
25 years old.
```

Variable-Length Positional Arguments (*args)

A function can accept a variable number of arguments, which are collected into a tuple. To use this functionality, place an asterisk (*) before the parameter name in the function definition, allowing multiple arguments to be passed to the function. Variable-length positional arguments are called *args by convention, but it is possible to use another name for the parameter; just remember to write the * before it.

A simple example of *args is shown below:

```
# The parameter '*args' allows the function to
accept multiple arguments.
def greet(*args):
    # Print the 'args' tuple containing all the
arguments passed to the function.
    print(args)

# Call the 'greet' function with multiple
arguments.
greet("John", "Alice", "Bob") # Output: ('John',
'Alice', 'Bob').
```

To pass additional arguments, write them before the *args parameter:

```
# 'age' is a required parameter and '*names'
```

```
parameter collects all extra arguments into a
tuple.
def greet(age, *names):
    # Print the 'age' value and the 'names' tuple
containing all additional arguments.
    print(age, names)

# The 'age' parameter gets the value 25, and the
remaining arguments are packed into the 'names'
tuple.
greet(25, "John", "Alice", "Bob")  # Output: 25
('John', 'Alice', 'Bob')
```

A common way to use *args is to iterate over the collected arguments, as shown below:

```
# The parameter *names allows the function to
accept multiple arguments as a tuple.
def greet(*names):
    # Iterate through each name in the 'names'
tuple.
    for name in names:
        # Print a greeting message for each name.
        print(f"Hello {name}!")

# Call the 'greet' function with multiple
arguments as a tuple.
greet("John", "Alice", "Bob")  # Output: # Hello
John! # Hello Alice! # Hello Bob!
```

Variable-Length Keyword Arguments **kwargs

To pass multiple keyword arguments to a function, use two asterisks before the name of the parameter; these are commonly called **kwargs. Consider the following example:

```
# The parameter '**kwargs' allows the function to
accept multiple keyword arguments.
def greet(**kwargs):
    # Print the 'kwargs' dictionary.
    print(kwargs)

# Call the 'greet' function with multiple keyword
arguments.
greet(name="John", age=25, city="New York") #
Output: {'name': 'John', 'age': 25, 'city': 'New
York'}
```

As seen with *args, it is possible to combine more arguments with **kwargs; simply write the standard arguments first, followed by **kwargs:

```
# 'age' is a required parameter, and '** kwargs'
collects all additional keyword arguments into a
dictionary.
def greet(age, **kwargs):
    # Print the 'age' value and the 'info'
dictionary containing all additional keyword
arguments.
    print(age, kwargs)

# The 'age' parameter gets the value 25, and the
remaining keyword arguments are packed into the
'kwargs' dictionary.
greet(25, name="John", city="New York",
hobby="Reading")
# Output: 25 {'name': 'John', 'city': 'New York',
'hobby': 'Reading'}
```

To expand the functionality, use iteration to loop through all items inside **kwargs:

```
# The parameter '**info' allows the function to
accept multiple keyword arguments as a dictionary.
def greet(**info):
    # Iterate through each key-value pair in the
'info' dictionary.
    for key, value in info.items():
        # Print a message for each key-value pair.
        print(f"{key.capitalize()}: {value}")

# Call the 'greet' function with multiple keyword
arguments: name="John", age=25, and city="New
York".
greet(name="John", age=25, city="New
York")  #Output: Name: John #Age: 25 #City: New
York
```

Positional-Only Arguments

Parameters can be restricted to accept only positional arguments by placing a "/" in the function definition after the desired parameters. Positional-only arguments were introduced in Python 3.8 and are useful for preventing users from passing these arguments as keyword arguments. This ensures that the function is called in a specific and predictable way. Additionally, positional-only arguments can be slightly faster than keyword arguments because Python does not need to process or match parameter names.

```
def greet(name, age, /):
    print(f"Hello {name}, you are {age} years
old.")

greet("John", 25)  # This line works properly.
greet(name="John", age=25)  # Error: Cannot use
keyword arguments.
```

Keyword-Only Arguments

Following the same philosophy as positional-only arguments, it is also possible to use keyword-only arguments to restrict certain parameters to be passed only as keyword arguments. To achieve this, place an asterisk "*" before the desired parameters in the function definition.

```
def greet(*, name, age):
    print(f"Hello {name}, you are {age} years
old.")

greet(name="John", age=25)  # This line works
properly.
greet("John", 25)  # Error: Must use keyword
arguments.
```

Unpacking Arguments

Unpack a tuple or list into individual elements, which are passed as positional arguments to a function. Just write an asterisk "*" before the arguments in the function call. This is useful for dynamic and flexible function calls.

An example of unpack using *args is shown below:

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years
old.")

args = ("John", 25)
# Unpacks the tuple into positional arguments.
greet(*args)  # Output: Hello John, you are 25
years old.
```

Here is the same example, but now using **kwargs:

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years
old.")
```

```
kwargs = {"name": "John", "age": 25}
# Unpacks the dictionary into keyword arguments.
greet(**kwargs)  # Output: Hello John, you are 25
years old.
```

Combining Argument-Passing Methods

It is also possible to combine positional arguments, keyword arguments, variable-length positional arguments *args, and variable-length keyword arguments **kwargs in a single function.

```
def greet(greeting, *args, **kwargs):
    print(greeting)
    for name in args:
        print(f"Hello {name}!")
    for key, value in kwargs.items():
        print(f"{key}: {value}")

greet("Welcome!", "John", "Alice", age=25,
location="New York") # Output: Welcome! #Hello
John!  # Hello Alice! # age: 25 #location: New
York.
```

The code above prints a greeting, greets multiple names, and displays additional key–value information.

Lambda Expressions

Lambda expressions, also known as anonymous functions, can be understood as a concise way to define a function without a name. They can be treated as values, passed as arguments, or returned from other functions, but they are limited to containing a single expression.

Lambda expressions have their roots in functional programming, which dates back to the 1950s, and are now available in several modern programming languages, such as Python, Java, JavaScript, C#, and others.

A lambda function must be called with arguments and expressions to produce a result, as illustrated in Figure 5-2.

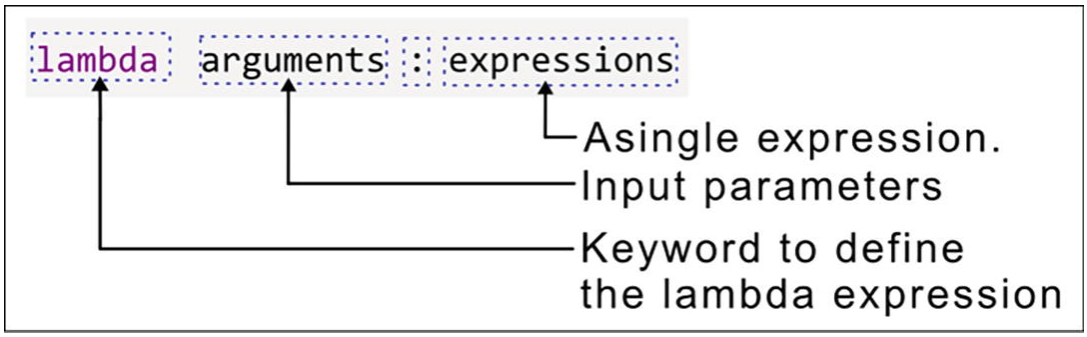


Figure 5-2 The syntax of a Python lambda expression, showing the keyword, arguments, and single-expression structure

The following is an example of code that uses a lambda function:

```
# A lambda function to multiply two numbers.
multiply = lambda x, y: x * y
# Using the lambda function.
result = multiply(2, 5)
print(result)  # Output: 10
```

There are several ways to use lambda expressions; the most used ways are listed in Table 5-2.

Table 5-2 Common methods and use cases for lambda expressions in Python, such as their use with built-in functions and in sorting operations

Assigning a Lambda to a Variable

To use a lambda expression as a regular function, just assign a lambda expression to a variable and use this variable in the code. The arguments can be passed to the lambda expression like a regular function.

```
# Example: Adding two numbers
add = lambda x, y: x + y
```

```
print(add(8, 7))  # Output: 15
```

Using a Lambda with Built-In Functions

Lambda expressions are very useful when working with Python built-in functions; the most common are map(), filter(), zip(), and reduce().

The **map()** function is used to execute a function on all items in an iterable data sequence or collection, like lists and tuples.

```
# Define a list of numbers.
numbers = [1, 2, 3, 4, 5]

#The lambda function squares each number (x**2).
#The result is then converted into a list.
squared = list(map(lambda x: x**2, numbers))

# Print the resulting list of squared numbers.
print(squared)  # Output: [1, 4, 9, 16, 25]
```

The **filter()** function iterates through each item in a collection, such as a list or tuple, and keeps only those items for which a given condition evaluates to true.

```
# Define a list of numbers.
numbers = [1, 2, 3, 4, 5]

# Use the filter() function to filter out even
numbers from the list.
even_numbers = list(filter(lambda x: x % 2 == 0,
numbers))

# Print the filtered list of even numbers.
print(even_numbers)  # Output: [2, 4]
```

The **zip()** function combines multiple sequences, such as lists or tuples, into a single sequence of tuples, where each tuple contains one element from each sequence.

```
# Define a list of names.
names = ['John', 'Emma', 'Sophia']

# Define a list of ages corresponding to the
names.
ages = [20, 25, 40]

# Use the map() function to increment each age by
1.
# The lambda function (lambda age: age + 1) adds 1
to each age in the ages list.
# Then, use the zip() function to pair each name
with the incremented age.
zipped = list(zip(names, map(lambda age: age + 1,
ages)))

# Print the resulting list of tuples, where each
tuple contains a name and the incremented age.
print(zipped)  # Output: [('John', 21), ('Emma',
26), ('Sophia', 41)].
```

The **reduce()** function repeatedly applies a two-argument function to a sequence of items, combining the results step by step until only a single final value remains. Although it is part of Python's standard library, it must be explicitly imported from the *functools* package before use. The syntax of the reduce() function is *reduce(function, iterable, initializer=None)*. The initializer is optional and serves as the starting value for the computation.

```
from functools import reduce

# Example: Calculating the product of all numbers
in a list.
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
```

```
print(product)  # Output: 120.
```

Using a Lambda in Sorting

Lambda expressions can be used as the key argument in the sorted() function to sort a collection, such as a list, tuple, or dictionary.

```
# Define a list of dictionaries, where each
dictionary represents a student.
students = [
    {'name': 'Alice', 'grade': 85},
    {'name': 'Bob', 'grade': 92},
    {'name': 'Charlie', 'grade': 78}
]

# The lambda function (lambda x: x['grade'])
extracts the 'grade' value from each dictionary.
# The reverse=True argument sorts the list in
descending order (highest grade first).
sorted_students = sorted(students, key=lambda x:
x['grade'], reverse=True)

# Print the sorted list of students.
print(sorted_students)
# Output: [{'name': 'Bob', 'grade': 92}, {'name':
'Alice', 'grade': 85}, {'name': 'Charlie',
'grade': 78}].
```

Using Lambda Expressions in List Comprehensions

List comprehension is a compact way to create lists. It can generate a new list by applying an expression to each item in an iterable collection. Additionally, it is possible to filter items using a conditional statement.

```
# 4. Using Lambda in List Comprehensions

# Define a list of numbers
```

```
numbers = [1, 2, 3, 4, 5]

# Use a list comprehension to create a list of
lambda functions
squared = [lambda x=x: x**2 for x in numbers]

# The result is a new list containing the squares
of the numbers in the original 'numbers' list
print([f() for f in squared])  # Output: [1, 4, 9,
16, 25]
```

Using Lambda Expressions in Conditional Statements

Lambda expressions can help to create a compact and short simple condition using the "if" and "else" conditional statement.

```
# Define a lambda function to check if a number is
even or odd
check_even_odd = lambda x: "Even" if x % 2 == 0
else "Odd"

# Test the lambda function with the number 4
print(check_even_odd(4))  # Output: Even

# Test the lambda function with the number 7
print(check_even_odd(7))  # Output: Odd
```

The following example shows code using two else statements:

```
# Lambda to determine if a number is positive,
negative, or zero
check_number_sign = lambda x: "Positive" if x > 0
else "Negative" if x < 0 else "Zero"

# Test the lambda function
print(check_number_sign(5))   # Output: Positive
print(check_number_sign(-3))  # Output: Negative
```

```
print(check_number_sign(0))   # Output: Zero
```

Using Lambda Expressions in Dictionaries

Dictionaries can have their functionality extended with the use of lambda expressions. It is possible to associate a lambda expression with a dictionary, where each key in the dictionary corresponds to an operation. This is useful for storing operations or logic directly within the dictionary.

```
# Dictionary with lambda functions as values
operations = {
    "add": lambda x, y: x + y,
    "subtract": lambda x, y: x - y,
    "multiply": lambda x, y: x * y,
    "divide": lambda x, y: x / y if y != 0 else
"Division by zero"
}

# Use the dictionary to perform operations
print(operations["add"](5, 3))        # Output: 8
print(operations["subtract"](10, 4))  # Output: 6
print(operations["multiply"](2, 3))   # Output: 6
print(operations["divide"](8, 2))     # Output:
4.0
```

Using a Lambda in Functions

A regular function can return a lambda expression to create customizable behavior.

```
# A function that returns a lambda expression
def power_function(exponent):
    return lambda x: x ** exponent

# Create specific functions using the returned
lambda
```

```
square = power_function(2)  # Returns a lambda
that squares a number

# Use the returned lambda expressions
print(square(4))  # Output: 16
```

Using any() and all()

The built-in functions **any()** and **all()** are used to operate on iterable collections like lists, tuples, or sets. The difference between them is that any() compares each element and all() compares all elements, but they both stop evaluating as soon as the result is determined.

The **any()** function compares each element in the collection, returning True if at least one element of an iterable is true and False if all elements are false.

```
# Example usage of any()
bools = [False, False, True, False]
result = any(bools)
print(result)  # Output: True

# Another example with numbers
numbers = [1, 3, 5, 7, 9]
result = any(num % 2 == 0 for num in numbers)
print(result)  # Output: False
```

The **all()** function returns True if **all** elements of the iterable are true or if the iterable is empty. If there is one false element, it returns False.

```
# Example usage of all()
bools = [True, True, False, True]
result = all(bools)
print(result)  # Output: False

# Another example with numbers
```

```
numbers = [2, 4, 6, 8, 10]
result = all(num % 2 == 0 for num in numbers)
print(result)  # Output: True
```

The following examples use generator expressions with any() and all(). Each expression evaluates the condition for one item at a time.

```
# List of numbers
numbers = [2, 4, 6, 8, 10]

# Check if any number is odd
is_any_odd = any(x % 2 != 0 for x in numbers)
print(is_any_odd)  # Output: False

# Check if all numbers are even
are_all_even = all(x % 2 == 0 for x in numbers)
print(are_all_even)  # Output: True
```

Using Lambda Expressions in sorted() with Multiple Keys

It is possible to sort a list by multiple criteria using a lambda expression that returns a tuple of sorting values.

In the following example the loop iterates through each part in the parts list and calculates the total price by multiplying the quantity by the price per unit. The result is stored in a new key total_price for each part. The "key" parameter uses a lambda function to determine the sorting order.

The negative sign in the expression "-part["total_price"]" ensures descending order for the total price. If two parts have the same total price, they are sorted alphabetically by name in ascending order through "*part["name"]*".

```
# List of parts for a project
parts = [
    {"name": "Resistor", "quantity": 10,
"price_per_unit": 0.1},
```

```
    {"name": "Capacitor", "quantity": 5,
"price_per_unit": 0.2},
    {"name": "Transistor", "quantity": 3,
"price_per_unit": 0.5},
    {"name": "Diode", "quantity": 8,
"price_per_unit": 0.15},
]

# Calculate total price for each part
for part in parts:
    part["total_price"] = part["quantity"] *
part["price_per_unit"]

# Sort parts by total price (descending) and then
by name (ascending)
sorted_parts = sorted(parts, key=lambda part: (-
part["total_price"], part["name"]))

# Print the sorted list of parts with their total
prices
print("Sorted parts by total price (descending)
and name (ascending):")
for part in sorted_parts:
    print(f"{part['name']}: Quantity =
{part['quantity']}, Total Price =
${part['total_price']:.2f}")
    # Output:
    # Transistor: Quantity = 3, Total Price =
$1.50
    # Diode: Quantity = 8, Total Price = $1.20
    # Capacitor: Quantity = 5, Total Price = $1.00
    # Resistor: Quantity = 10, Total Price = $1.00
```

Decorators (@)

Decorators are a powerful way to add or extend the functionality of functions. They are widely used and promote dynamic behavior, code reusability, and flexibility, making the code cleaner, more modular, and easier to maintain. Decorators can be applied in a wide range of domains and use cases, including web development, graphical user interfaces (GUIs), machine learning, data science, testing, authentication, authorization, and much more.

In addition to functions, decorators can also be applied to methods and classes, originating from the object-oriented programming (OOP) paradigm. This approach is discussed in detail across two chapters in this book, providing a comprehensive understanding of how decorators enhance code structure and functionality.

A decorator is a function that takes another function as its parameter and extends or modifies its behavior.

To create a decorator, you define a new function that accepts the original function as a parameter and includes a nested function, commonly named wrapper to add extra behavior before or after the original function is executed. The nested function wraps around the original function, hence the name wrapper.

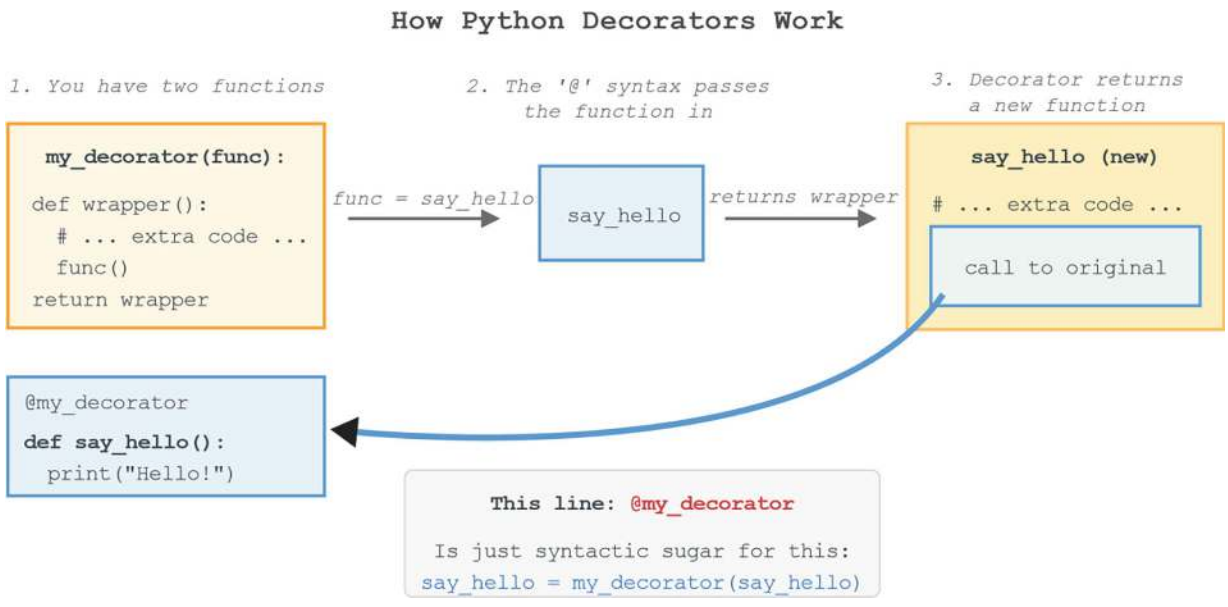


Figure 5-3 Decorator operation flowchart

To use a decorator, simply place the @ symbol followed by the decorator's name directly above the function definition. Every time the decorated function is called, the decorator will also be executed, allowing it to perform the desired additional behavior.

```
# Define a decorator
def simple_decorator(func):
    def wrapper():
        print("Before the function is called.")
        func()
        print("After the function is called.")
    return wrapper

# Use the decorator
@simple_decorator
def say_hello():
    print("Hello, world!")

# Call the decorated function
say_hello()
#Output: Before the function is called.
Hello, world!
After the function is called.
```

In the example above, the word func is used to represent the original function say_hello passed as a parameter to the decorator. The decorator then returns the wrapper function, which is executed in place of the original function when the decorated function is called.

Another example code using mathematical operations is provided below:

```
# Define a decorator
def math_decorator(func):
    def wrapper(*args, **kwargs):
        print("Performing a math operation...")
```

```
        result = func(*args, **kwargs)
        print(f"The result of the operation is:
{result}")
        return result
    return wrapper

# Use the decorator
@math_decorator
def add_numbers(a, b):
    return a + b
# Call the decorated functions
add_numbers(5, 7)
# Output: Performing a math operation...
# The result of the operation is: 12
```

The example code above defines a decorator called math_decorator, which adds messages before and after the wrapped mathematical operation and prints its result. The add_numbers function, decorated with math_decorator, takes two arguments, adds them, and returns the sum.

When the function add_numbers(5, 7) is called, the decorator first prints a message indicating a math operation, then executes the function to compute the sum (12), afterward prints the result (12), and finally returns the result. The decorator adds functionality without modifying the original function's logic.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_6

6. Conditionals and Loops

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Two important features found in most programming languages are loop control and conditional statements. Both are essential for guiding code execution using control flow statements. In this chapter, the keywords "while" and "for" will be studied to create loops for repeated execution, as well as the "if" keyword for performing conditional tests. Additionally, this chapter covers exception handling, a mechanism for responding to errors and unexpected situations during program execution.

Conditional Statements

One of the most fundamental control flow statements in programming is the **if** statement.

> The **if** statement allows you to execute a block of code only when a specified condition evaluates to True.

It is particularly useful for making decisions in your program based on certain conditions or for comparing a value against a set of parameters.

The syntax of the **if** statement is shown in Figure 6-1.

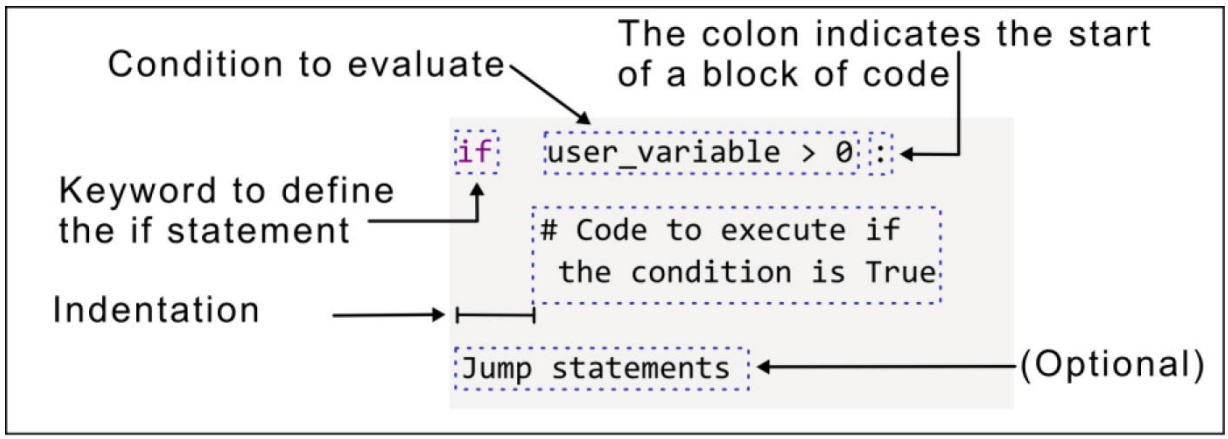


Figure 6-1 The syntax of a basic if statement in Python, highlighting the keyword, condition, and indented code block

For a better explanation of the variations of the **if** statement, the topic has been divided into the following sub-items.

Simplified if Statement

Let's start with the simplest form of the if statement. To write it, place a condition after the keyword if, followed by a colon ":"; the colon tells the Python interpreter that the following indented lines belong to the block of code associated with the statement. On the next line, indent the code block consistently; four spaces are the standard convention. Write the code to execute if the condition evaluates to True. Before the if statement, ensure that any variables used in the condition are properly declared and initialized. Inside loops, if statements are often combined with control statements such as break and continue; these are discussed later in this chapter. The syntax of the basic if statement is shown in Figure 6-1.

The following example demonstrates the simplest form of an if statement:

```
number = 5                                  # Assign a
value to the variable.

if number > 0:                              # Check if
the number is greater than 0.
```

```
                                        #This will be
executed if the condition is True.
   print("The number is positive.")  # Output: The
number is positive.
```

It is important to note in this example that if the variable number is assigned a negative value or zero, the code will execute nothing because there is no condition to handle such cases. For example, with "number = -1", the condition "number > 0" is False, so the print statement is skipped and no output is produced.

The if Statement with else

To handle false conditions in an if statement, it is possible to use the keyword else. The syntax is shown in Figure 6-2.

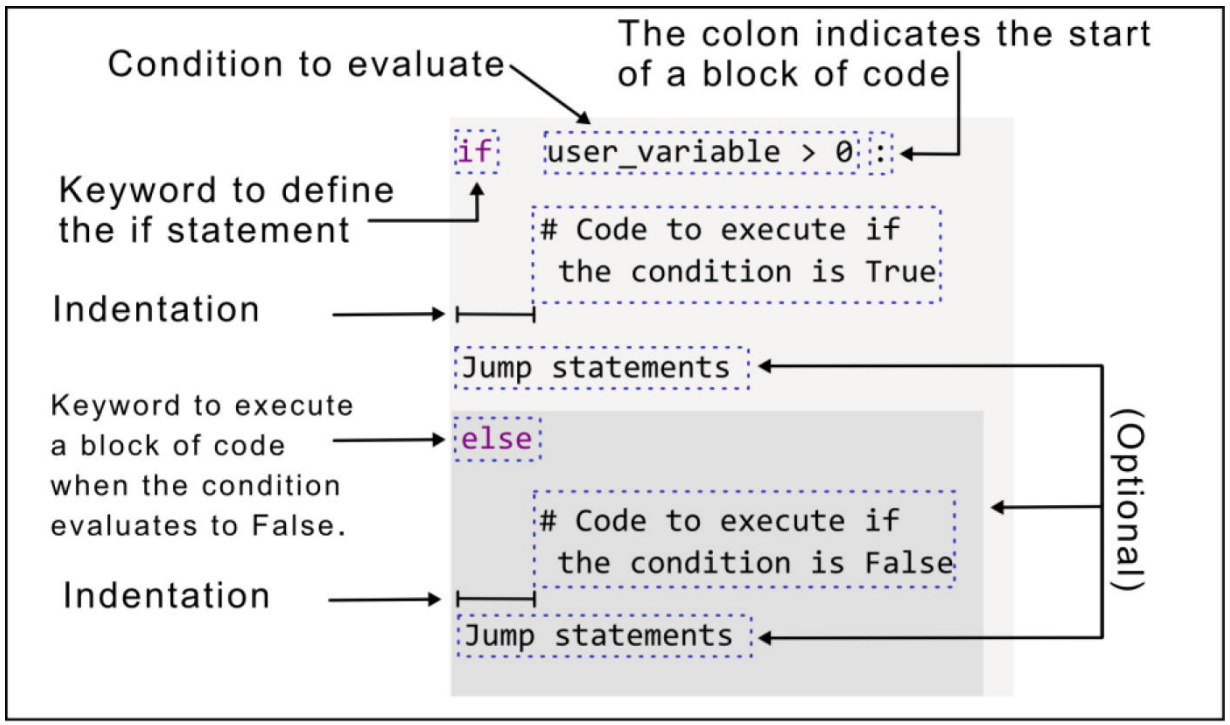


Figure 6-2 The syntax of an if…else statement, showing how to execute alternative blocks of code

```
number = 5  # Assign a value to the variable.
```

```
if number > 0:  # Check if the number is greater than 0.
    # This will be executed if the condition is True.
    print("The number is positive.")  # Output: The number is positive.
else:
    # This will be executed if the condition is False, if number <= 0.
    print("The number is not positive.")  # Output: The number is not positive.
```

The if Statement with elif

An **if** statement can test multiple conditions by adding one or more **elif** branches before an optional **else** branch. An example is shown in the code below:

```
age = 25  # Assign a value to the variable.

if age < 2:  # Check if the person is less than 2 years old.
    print("The person is a baby.")  # Output: The person is a baby.
elif 2 <= age < 12:  # Check if the person is between 2 and 12 years old.
    print("The person is a child.")  # Output: The person is a child.
elif 12 <= age < 18:  # Check if the person is between 12 and 18 years old.
    print("The person is young.")  # Output: The person is young.
elif 18 <= age < 60:  # Check if the person is between 18 and 60 years old.
```

```
    print("The person is an adult.")  # Output:
The person is an adult.
else:  # If none of the above conditions are True,
the person is 60 or older.
    print("The person is old.")  # Output: The
person is old.
```

While Loops

When you need to perform repetitive tasks, such as printing numbers from 1 to 10, writing the same code multiple times is both inefficient and error-prone. Instead, loops allow you to execute a block of code repeatedly with minimal effort. In Python, one of the commonly used loop structures is the "while" loop.

The basic syntax for using the "while" statement is illustrated in Figure 6-3.

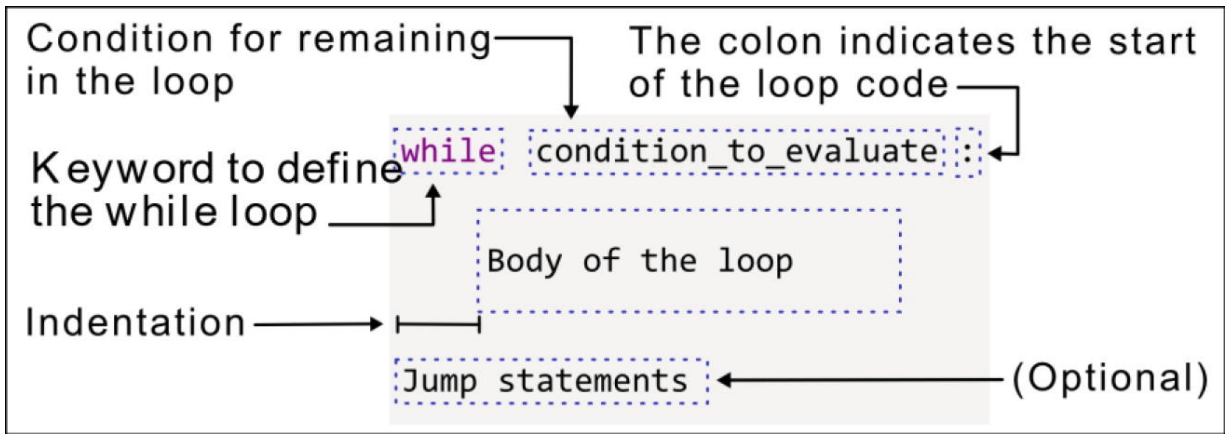


Figure 6-3 The syntax of a while loop, illustrating the keyword, condition, and loop body

To create a while loop, write the keyword while followed by a condition to be evaluated. Then, on a new indented line, write the code that will execute in each iteration. The next example code prints on the screen a sequence of numbers from 1 to 5 and demonstrates basically how the while loop works. Additionally, it is possible to use jump statements in a while loop. These will be discussed later in this chapter.

```
i = 1                       # Initialization: Start
with i = 1

while i <= 5:               # While keyword, and
condition to evaluate.
    print(i)                # Body with the code.
    i += 1                  # Increment to eventually
make the condition False.

#Output: #1 #2 #3 #4 #5 (These numbers will be
printed one at a time on the screen.)
```

The loop in the example code starts with "i" = 1, where the value 1 is assigned to the variable "i" before the while statement is executed. In the first iteration, the value of the variable "i" is 1. During each iteration, the variable i is incremented by 1. The loop ends when i becomes 6 because the condition "i" <= 5 evaluates to False.

Table 6-1 provides a detailed breakdown of each iteration, showing the value of the variable "i" and whether the condition "i" <= 5 is True or False.

Table 6-1 An iteration-by-iteration breakdown of a basic while loop that counts from 1 to 5

Just as with an if statement, the else keyword can be used with a while loop to execute a block of code once the loop condition evaluates to false.

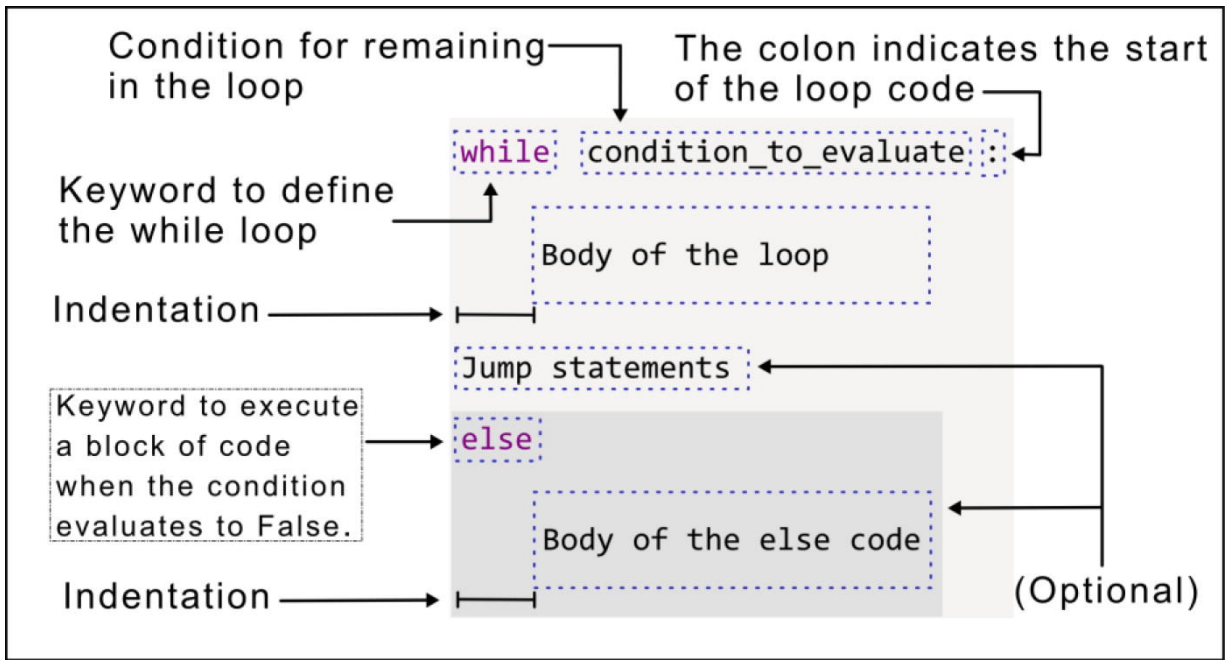


Figure 6-4 The syntax of a while loop with an else block, which executes after the loop completes normally

The following example code demonstrates the use of **else** in a while loop:

```
i = 1                          # Initialization:
Start with i = 1.

while i <= 5:                  # While loop:
Continue as long as i is less than or equal to 5.
    print(i)                   # Print the current
value of i.
    i += 1                     # Increment i by 1 to
eventually make the condition False.

else:                          # This block executes
when the while loop condition becomes False.
    print("Loop finished!")    # Indicate that the
loop has completed.
```

```
# Output:                          # 1 # 2 # 3 # 4 # 5 #
Loop finished!
```

Table 6-2 shows the step-by-step iteration trace of the example code using the else keyword.

Table 6-2 A trace of a while loop with an else block, showing when the else statement is executed

It is also possible to combine conditional structures like if–elif–else within loops to handle multiple logical conditions during execution.

The following code example, along with Table 6-3 illustrates its step-by-step execution trace:

```
i = 1                              # Initialization:
Start with i = 1.

while i <= 5:                      # While loop:
Continue as long as i is less than or equal to 5.
    if i < 3:                      # Check if i is less
than 3.
        print(f"{i} is less than 3")  # Action for
i < 3.
    elif i == 3:                   # Check if i is equal
to 3.
        print(f"{i} is equal to 3")  # Action for
i == 3.
    else:                          # This block executes
if i > 3.
        print(f"{i} is greater than 3")  # Action
for i > 3.
    i += 1                         # Increment i by 1 to
eventually make the condition False.
```

```
else:                           # This block executes
when the while loop condition becomes False.
    print("Loop finished!")   # Indicate that the
loop has completed.

# Output:
# 1 is less than 3
# 2 is less than 3
# 3 is equal to 3
# 4 is greater than 3
# 5 is greater than 3
# Loop finished!
```

Table 6-3 A detailed trace of a while loop containing a nested if–elif–else structure

Loops with for

Another commonly used loop in programming is the **for** loop.

> The **for** loop is used to iterate over a sequence, such as a range of numbers, characters in a string, or elements in a list, tuple, or other iterable data structures.

The basic syntax of the **for** loop is shown in Figure 6-5.

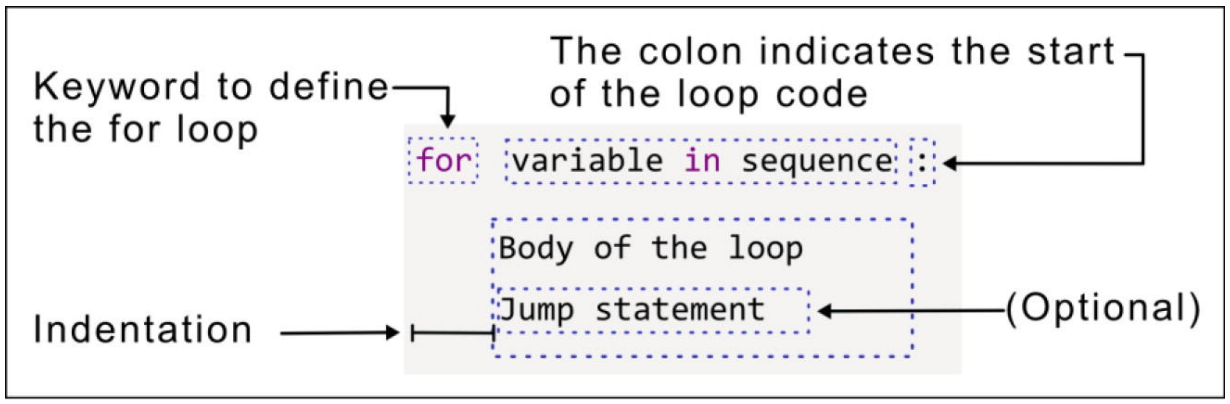


Figure 6-5 The syntax of a for loop used for iterating over a sequence

As shown in Figure 6-5, the for loop requires a variable and a sequence of data to work properly. Its syntax is very similar to while and if statements, as it includes a keyword, a colon, and a block of code to execute during each iteration of the loop.

The simplest example of iterating with a for loop is shown in the following code:

```
# Define a list of numbers
numbers = [2, 4, 6]

# Iterate through each number in the list
for number in numbers:
    # Print the current number
    print(number)

# Output:
# 2
# 4
# 6
```

The iteration for this example code is shown in Table 6-4.

Table 6-4 A trace of a for loop iterating over a list of numbers

An example of a for loop iterating over a list of names is shown in the following code:

```
# Define a list of fruits
fruits = ["apple", "banana", "cherry"]

# Iterate through each fruit in the list
for fruit in fruits:
    # Print the current fruit
    print(fruit)
```

```
# Output:
# apple
# banana
# cherry
```

For this example, the iteration is shown in Table 6-5.

Table 6-5 A trace of a for loop iterating over a list of strings (fruits)

Table 6-6 An iteration trace for a for loop that uses range(5) to generate a sequence of numbers

Before presenting the next way to use the for loop, it is necessary to understand the range() function, which is used to generate a sequence of numbers. The range() function accepts up to three parameters: "start", "stop", and "step".

The **start** parameter is optional and specifies the starting number of the sequence. If no value is provided, the default is 0.

The **stop** parameter specifies the number at which the sequence stops. However, this value is not included in the sequence. For example, if the stop parameter is 5, the function will generate numbers up to 4 (i.e., 5 is excluded).

The **step** parameter is optional and specifies the difference between each number in the sequence. If no value is provided, the function uses a default value of 1.

The code below is an example of how to use the range() function:

```
# Convert range to a list and print it, range(5)
generates numbers from 0 to 4.
numbers = list(range(5))
print(numbers)  # Output: [0, 1, 2, 3, 4].

# Create a range starting from 1, stopping before
10, with a step of 2.
numbers_range = list(range(1, 10, 2))
```

```
print(numbers_range)  # Output: [1, 3, 5, 7, 9].
```

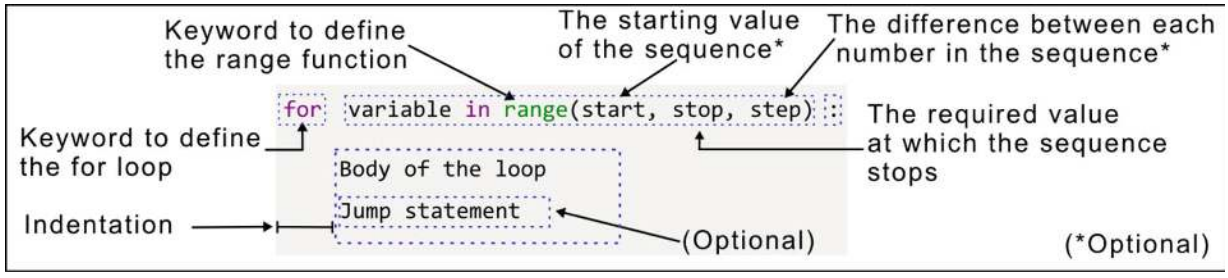


Figure 6-6 The syntax of a for loop combined with the range() function, detailing the start, stop, and step parameters

Another way to use the for loop is with the range() function. This allows you to define a range of numbers to run the iterations multiple times. Remember that the stop value is excluded from the generated sequence. When only the stop argument is provided, the start value defaults to 0. The syntax for this case, along with the example code, is shown in Figure 6-7 and the code below.

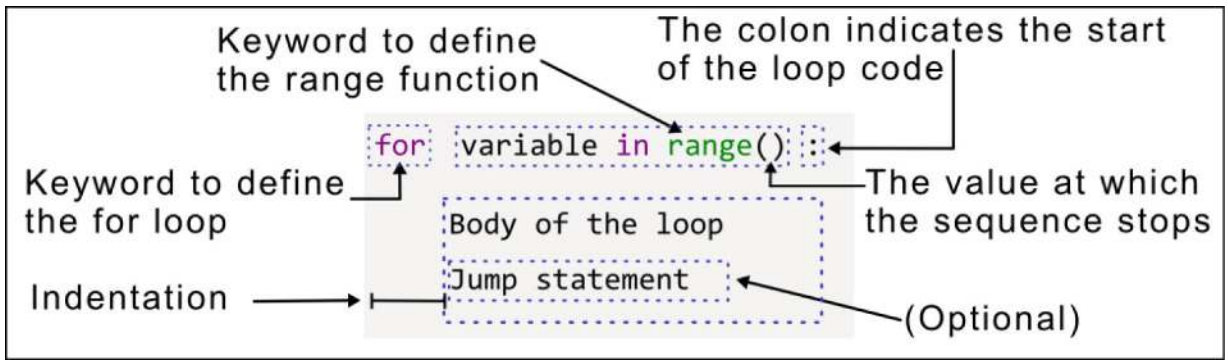


Figure 6-7 The syntax of a for loop using the range() function with a single stop parameter

```
# Iterate through numbers generated by range(5)
# range(5) generates numbers from 0 to 4 (5 is
excluded)
for i in range(5):
    # Print the current value of i
    print(i)

# Output:
```

```
# 0
# 1
# 2
# 3
# 4
```

The iteration is similar to the first example.

It is also possible to iterate through a string, as shown in the example below:

```
# Iterate through each character in the string
"Hello!"
for char in "Hello!":
    # Print the current character
    print(char)

# Output:
# H
# e
# l
# l
# o
# !
```

The iteration for this example code is shown in Table 6-7.

Table 6-7 A trace showing a for loop iterating through the individual characters of a string

To use the **else** keyword in a **for** loop, simply write the **else** keyword after the loop's block of code, ensuring it is not indented:

```
# range(3) generates numbers from 0 to 2 (3 is
excluded)
for i in range(3):
    # Print the current value of i
```

```
    print(i)
# The else block executes after the loop finishes
(if no break occurs)
else:
    print("Loop completed!")

# Output:
# 0
# 1
# 2
# Loop completed!
```

Jump Statements

Control statements can alter the execution flow of loops by skipping an iteration or exiting a loop early. The examples in this section show how these statements affect while and for loops.

The statements discussed here are "break", "continue", and "pass". The first two alter loop flow, while pass is a placeholder that performs no action. Each is explained with examples below.

Control Flow Statement with "break"

The keyword "break" is used to exit the loop immediately, regardless of the loop condition.

With the same example shown above, but changing to use the keyword "break" and the "if" condition, the loop will stop when "i" is equal to 3:

```
i = 1              # Initialization.

while i <= 5:      # 1. while keyword, 2. condition
to evaluate.
    if i == 3:     # Break the loop when i equals 3.
        break
    print(i)       # 4. Body with the code.
```

```
    i += 1        # Increment to eventually make
the condition False.

# Output:
# 1
# 2
```

Table 6-8 provides a detailed breakdown of each iteration of the example code above.

Table 6-8 An iteration trace demonstrating how the break statement immediately terminates a while loop

Control Flow Statement with "continue"

The statement "continue" is used to skip the current iteration and move to the next iteration of the loop.

The code in the next example prints only the odd numbers from 1 to 5. It skips even numbers using the continue statement, which moves to the next iteration without executing the "print(i)" statement for even values:

```
i = 0 # Start at 0 to check 1 first
while i < 5:
    i += 1 # Increment once at the start of the
loop
    if i % 2 == 0: # Check if the number is even
        continue   # Skip the print for even
numbers
    print(i)      # This line is only reached for
odd numbers
 # Output:
 # 1
 # 3
 # 5
```

Table 6-9 shows each iteration of the example code above.

Table 6-9 An iteration trace demonstrating how the continue statement skips the current iteration in a while loop

Using Jump Statements in Loops

Within loops, conditional tests are often combined with control statements. The break statement exits a loop, continue skips the rest of the current iteration, and pass performs no action. The following examples show these statements in for loops and in conditions inside loops.

Jump Statement with the break Keyword

The break statement is used to exit a loop prematurely based on a specific condition. In the example below, the keyword break is used to exit the loop when the number 3 is found. This way, the loop does not need to perform all iterations, saving computational effort:

```
numbers = [1, 2, 3, 4, 5]  # A list of numbers to
iterate through

for num in numbers:  # Start a loop to go through
each number in the list
    if num == 3:  # Check if the current number is
equal to 3
        print("Found 3!")  # Output: Found 3!
(when the condition is True)
        break  # Exit the loop immediately when 3
is found
    print(num)  # Output: 1, 2 (prints the number
if it is not 3)

# Output: # 1 # 2 # Found 3!
```

To better understand this last code, the iteration is shown in Table 6-10.

Table 6-10 A trace showing how the break statement exits a for loop prematurely

Jump Statement with the continue Keyword

The continue statement is used to skip the rest of the current iteration and pass control to the next iteration of the loop. This is useful for ignoring unnecessary cases without stopping the entire loop:

```
for i in range(1, 6):     # Loop through numbers
from 1 to 5
    if i % 2 == 0:        # Check if the current
number is even (i % 2 == 0)
        continue          # Skip the rest of the
loop for even numbers and move to the next
iteration
    print(i)              # Output:# 1 # 3 # 5
```

The iteration in Table 6-11 helps illustrate each iteration and its corresponding action.

Table 6-11 A trace showing how the continue statement skips specific iterations in a for loop

Jump Statement with the pass Keyword

The pass statement is used when Python requires a statement but no action is needed. It acts as a placeholder and does not skip the current iteration; execution continues normally after pass. In the example below, the else branch prevents print(i) from running when i is 3:

```
for i in range(1, 6):       # Loop through numbers
from 1 to 5.
    if i == 3:              # Check if the current
number is 3.
```

```
        pass                    # Do nothing for now.
    else:                       # If the number is not
3.
        print(i)                # Output: # 1 # 2 # 4 #
5.
```

To understand each iteration of the example code above, refer to Table 6-12.

Table 6-12 A trace illustrating the behavior of the pass statement within a for loop

Exception Handling

Exception handling is a mechanism available in Python and many other programming languages for responding to errors and unexpected situations during program execution. When an exception is handled, the program can take a defined action instead of leaving the error unhandled.

In engineering software, exception handling can support reliability by allowing failures to be handled in a controlled way. It does not by itself guarantee a safe system; validation, fault detection, logging, and appropriate recovery logic are also required. For example, a program may log an error, release a resource, notify the operator, or follow a predefined recovery path.

Examples in engineering applications include

- *Robotics*: After invalid data or a failed operation is detected, an exception handler can stop a command or report the problem to higher-level control software.
- *Industrial Automation*: Communication or hardware-related exceptions can be logged and passed to recovery or fallback logic instead of being left unhandled.
- *Embedded Software*: In systems that use exception mechanisms, handlers can support controlled error reporting or recovery; overall safety still depends on the complete system design.

Let's consider a situation where the code prompts the user to input a number, attempts to convert the input to an integer, and then prints twice that number. If the user enters a valid number, the program calculates and displays the result. However, if the input is not a valid number, such as a letter or symbol, the program crashes with a ValueError. To improve the code, error handling could be added to catch and handle this exception, allowing the program to provide a more user-friendly response instead of crashing.

```
user_text = input("Please enter a number: ")

# This line will raise a ValueError if the user
doesn't enter a number.
number = int(user_text)

print(f"The double of your number is {number *
2}")
```

See this example in the code shown in Figure 6-8.

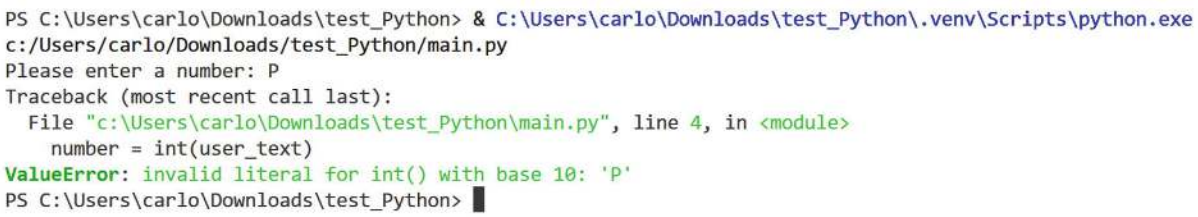


Figure 6-8 A terminal screenshot showing a ValueError traceback, which occurs when attempting to convert a non-numeric string to an integer

The basic structure of exception handling in Python involves a try block, where code that may raise an exception is placed, and one or more except blocks that handle matching exceptions. If an exception raised in the try block matches an except block, execution transfers to that handler so the program can respond to the error.

Optional else and finally blocks can also be used. The else block runs when the try block completes without an exception, while the finally

block runs whether or not an exception occurs and is commonly used for cleanup tasks.

Figure 6-9 shows Python's exception handling syntax.

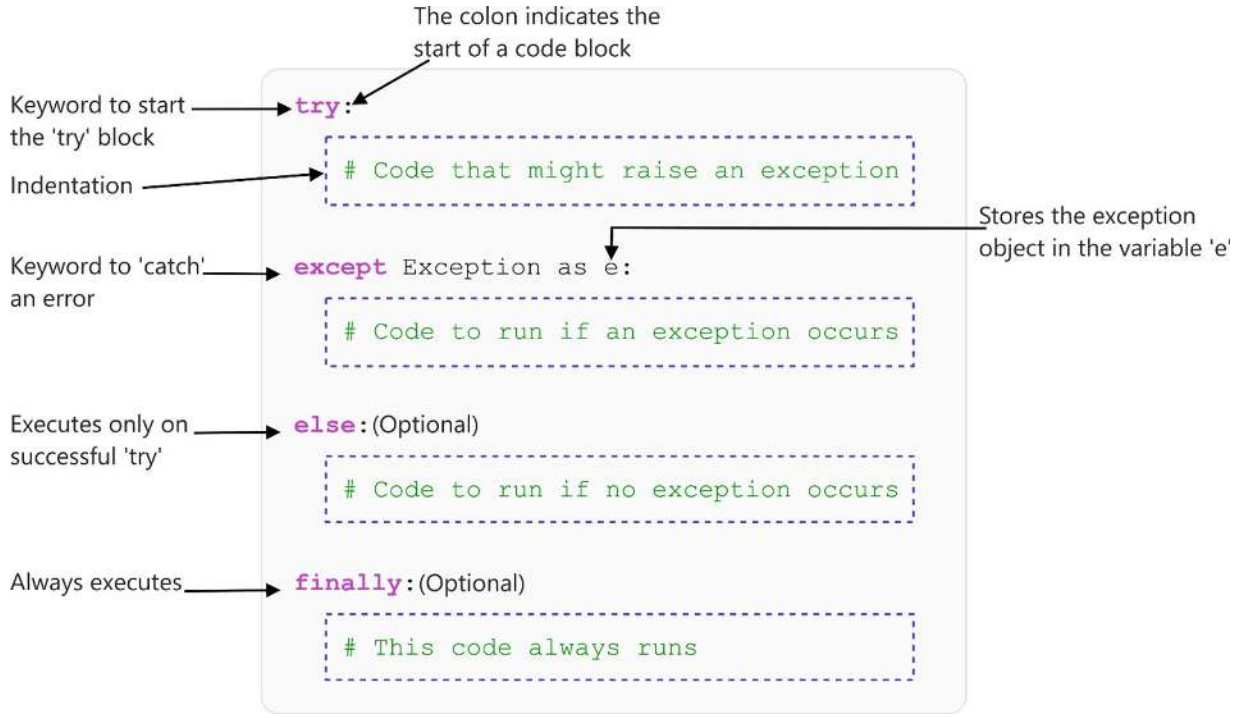


Figure 6-9 The syntax of Python's exception handling structure, including the try, except, else, and finally blocks

Note the following line used in the syntax:

```
except Exception as e:
```

The line "except Exception as e:" catches exceptions that inherit from Exception and stores the exception object in the variable e. The variable can then be used to inspect, print, or log details about the error. In larger programs, catching a more specific exception type is generally preferable when the expected error is known.

The code below solves the initial problem—if a letter or symbol is entered instead of a number, the ValueError is handled rather than left unhandled:

```
user_text = input("Please enter a number: ")
```

```
try:
    # 1. It tries to convert the text into a
number.
    number = int(user_text)

    # 2. If the conversion works, the code
continues and prints the result.
    print(f"The double of your number is {number *
2}.")

except ValueError:
    # 3. If the conversion fails, the program
jumps to this block.
    print("That was not a valid number.")
```

For a more complete version of the code using the else and finally keywords, see the example below:

```
user_text = input("Please enter a number: ")

try:
    # Code that might raise an exception
    number = int(user_text)
except ValueError as e:
    # Code to run if an exception occurs
    print(f"That was not a valid number.")
    print(f"Error details: {e}")
else: # This is optional.
    # Code to run if no exception occurs
    print(f"Success! The double of your number is
{number * 2}.")
finally: # This is optional.
    # This code always runs
    print("Input attempt finished.")
```

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_7

7. Object-Oriented Programming

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

This chapter focuses on object-oriented programming (OOP), a fundamental paradigm that shapes how modern software is designed and organized. We will explore how OOP provides a systematic framework for structuring code around the concept of "objects," which encapsulate both data and behavior. As Python is a language built entirely on object-oriented principles, understanding this paradigm is crucial for leveraging the full power and versatility of the language.

Throughout this chapter, you will study the core building blocks and principles of OOP. We will begin with the basics of classes and objects, explaining how a class serves as a blueprint for creating instances. You will then learn about the four pillars of OOP: encapsulation, abstraction, inheritance, and polymorphism, which together create a powerful and flexible approach to programming. The chapter will also delve into practical components such as attributes and methods, the different relationships between objects—association, aggregation, and composition—and more advanced topics like special "dunder" methods and the dependency injection design pattern.

Upon completing this chapter, you will be able to design and implement your own classes and objects in Python. You will have the skills to apply the principles of inheritance to build specialized subclasses, use polymorphism to create flexible and reusable code, and effectively manage the relationships between different objects in your programs. Ultimately, you will gain a solid foundation for writing code that is more

modular, maintainable, and scalable, enabling you to build more complex and robust applications.

In computer programming, a very common topic is the concept of programming paradigms. These paradigms represent fundamental methodologies or techniques to organize code. They provide a systematic framework for developing and organizing software, defining the logic of the code, and shaping approaches to solve problems.

There are various programming paradigms, each suited to different types of problems. While some paradigms have become outdated and are rarely used today, others remain widely adopted and influential. Choosing the right programming paradigm is a critical decision in software engineering, as it significantly impacts how the software is designed and developed and how it meets its requirements.

> A **programming paradigm** is a methodology for structuring and implementing computer software. It offers a framework for designing and developing software tailored to the specific problem being addressed.

This chapter does not aim to describe all programming paradigms. However, Tables 7-1 and 7-2 highlight six of the most common paradigms compatible with Python, providing relevant details for each.

Table 7-1 Common programming paradigms and examples

For example, C and Pascal are primarily procedural programming languages created in the early 1970s. They focus on procedures or routines that perform specific tasks, following a step-by-step approach. On the other hand, C# is a language primarily centered on the object-oriented programming (OOP) paradigm, emphasizing the use of objects and classes to structure code. In contrast to these languages, Python is a versatile, multi-paradigm language that supports various programming styles, including procedural, object-oriented, functional, and more,

making it suitable for a wide range of applications across different domains.

Table 7-2 Key concepts and applications of common programming paradigms

Object-Oriented Programming

In this chapter, we will delve into the fundamentals of object-oriented programming (OOP), one of the most popular and practical software development techniques. This programming paradigm structures code by organizing it into objects. The fundamental concept of OOP is the class, which serves as a blueprint for creating reusable blocks of code that encapsulate both data and behavior. A class is a core building block of OOP, used to create objects. Organizing code into classes improves its manageability, maintainability, and debuggability. Additionally, it facilitates code reuse across different parts of a program, promoting scalability and flexibility.

Object-oriented programming (OOP) aims to simplify the representation of real-world entities in the programming world. To achieve this, OOP is based on four fundamental principles: abstraction, encapsulation, inheritance, and polymorphism. Each of these principles contributes to making OOP a powerful and flexible paradigm for designing and organizing code. These four concepts will be explained after the topic of the class object because a basic understanding of classes is essential to grasp these concepts. To begin exploring these concepts, refer to the block diagram in Figure 7-1.

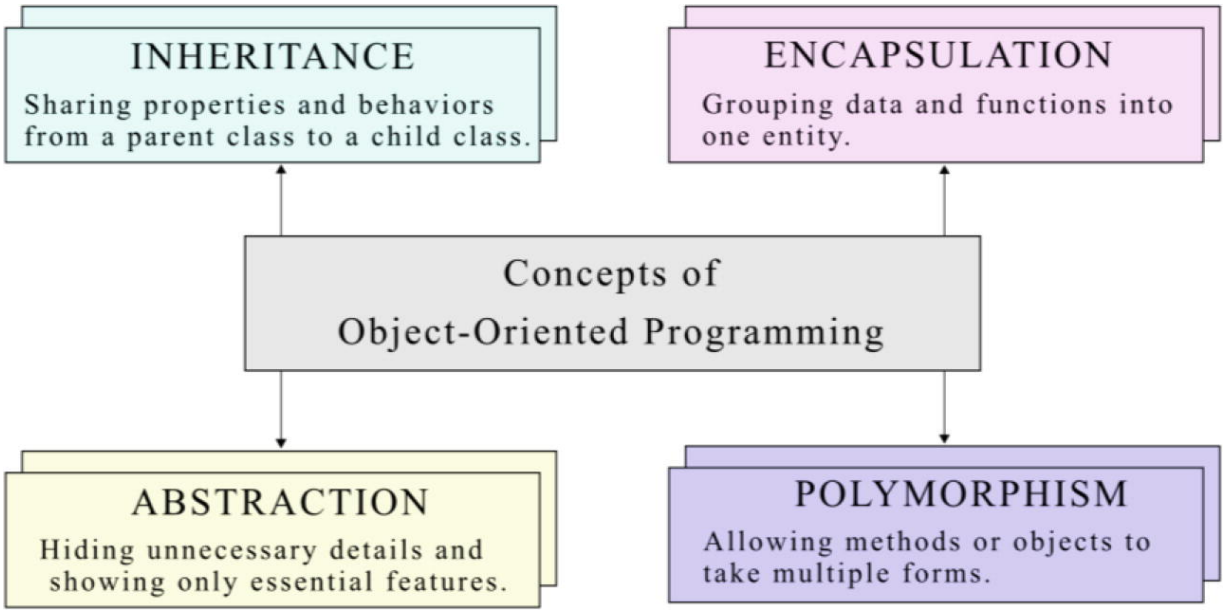


Figure 7-1 The four fundamental concepts of object-oriented programming

Python supports multiple programming paradigms, but its object model is central to the language. Almost everything exposed to the developer is an object, including numbers, strings, lists, functions, and modules. Each object is an instance of a class and participates in Python's type hierarchy.

A string, for example, is an object of the str class and has attributes and methods associated with its operation. To demonstrate this functionality, a simple example code has been added below:

```
#Converting a string to uppercase using a
string(str) method.

# Define a string
original_string = "hello, world!"

# Convert string to uppercase using a method.
uppercase_string = original_string.upper()

# Print the result
print("Original String:", original_string) #
Output: Original String: hello, world!
```

```
print("Uppercase String:", uppercase_string) #
Output: # Uppercase String: HELLO, WORLD!
```

In the line “uppercase_string = original_string.upper()”, the period after the name of the string variable is used to call a method. In this example, the upper() method is called to convert the string to uppercase. However, there are many other methods available for working with strings, each providing different functionalities. These methods were covered in Chapter 2.

Object-oriented programming (OOP) is a programming paradigm that structures code by organizing it into objects, which are instances of classes. These objects encapsulate data, known as attributes, and behavior, known as methods.

Classes

A class is like a model or blueprint that serves as a reference for creating objects based on the initial model defined by the user.

New instances of a class are created using the original blueprint or model, and these instances are referred to as objects in Python. Data and behaviors can be assigned to a class. The data is referred to as **attributes**, while the behavior is implemented through methods. A **method** is similar to a function but is specifically designed to operate on an object’s data. The syntax for defining a class is shown in Figure 7-2.

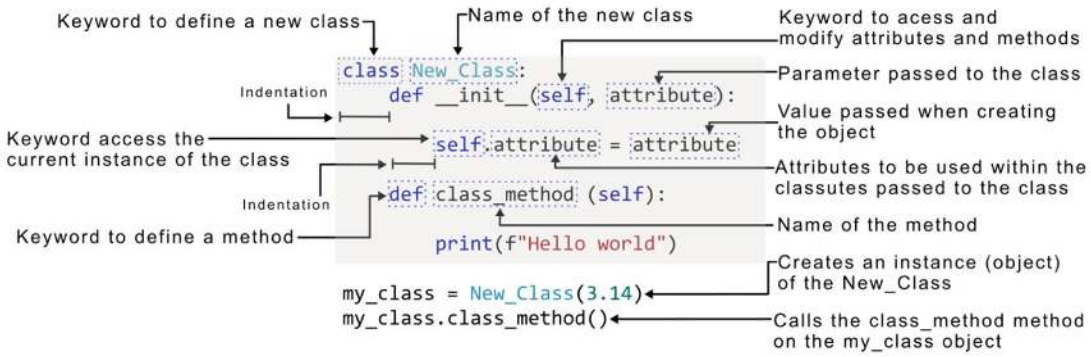


Figure 7-2 Syntax for defining a class in Python

To create an object of a class, simply call the name of the desired class and, if necessary, provide the parameters that will be used as attributes. Then, assign the resulting object to a variable.

The blueprint used to create multiple instances is simply called a class. The terms *base class*, *parent class*, and *superclass* are used when another class inherits from it. Figure 7-3 illustrates a class and its possible instances. In this case, only the value of the color attribute differs between instances, while the constructor and methods remain the same for all instances.

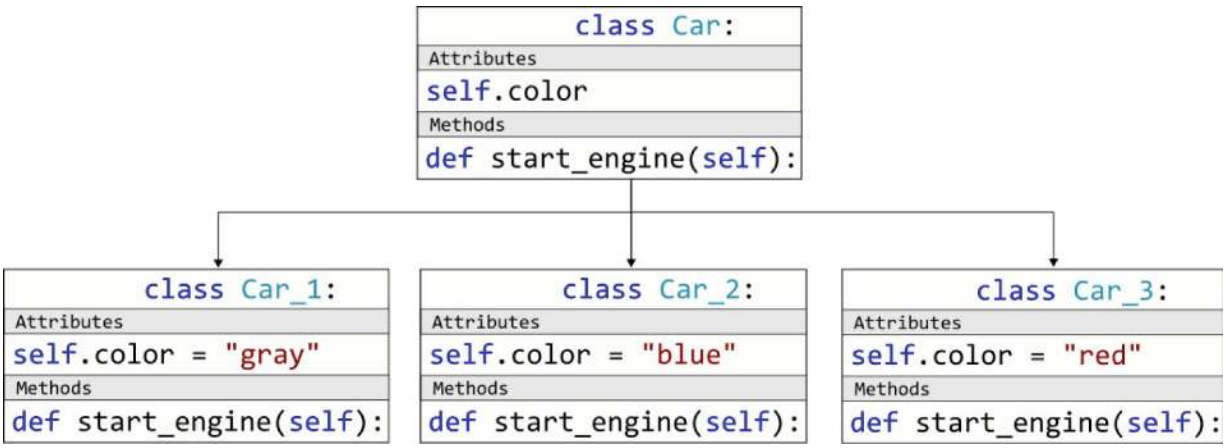


Figure 7-3 Illustration of a class and its instances

The name of a Python class does not need to be capitalized, but by convention, class names typically start with a capital letter, and each subsequent word in the name is also capitalized. This naming style is

known as PascalCase, which is commonly followed in Python to improve code readability.

Here is an example of code that creates and accesses a class:

```
#Creating a class with Car name.
class Car:
    def __init__(self, color_parameter):

        #Initialize the Car object with a single
attribute: color.
        self.color = color_parameter

    def start_engine(self):

        #Start the car's engine.
        print(f"The engine of the {self.color} car
is now running.")

# Example usage of the class Car:
my_car = Car("red")    # Sends the value parameter
"red" to the class.
my_car.start_engine()  # Output: The engine of the
red car is now running.
print (my_car.color)   # Output: red
```

The first line defines a new class named Car:

```
class Car:
```

The next line defines the __init__ method, which is automatically called when a new instance of the class is created. It takes two parameters: self, which refers to the instance being created, and color_parameter, which is passed when creating the instance in the line my_car = Car(“red”):

```
def __init__(self, color_parameter):
```

The line

```
self.color = color_parameter
```

is used to assign the value of the color_parameter parameter to the color attribute of the object. The value of color_parameter is passed when the object is created, and the "color" attribute is encapsulated within the object using self.color.

The method is defined using the line

```
def start_engine(self):
```

Just like defining a function, a method is defined using the same keyword **def**.

The next line in the class is

```
print(f"The engine of the {self.color} car is now
running.")
```

This line prints a message indicating that the car's engine is running and displays the value of the "color" attribute used in the "Car" class. It is important to note that the keyword self in {self.color} is used to refer to the color attribute of the specific instance of the Car class that received the parameter "red":

```
print(f"The engine of the {self.color} car is now
running.")
my_car = Car("red")     # Sends the value parameter
"red" to the class.
my_car.start_engine()  # Output: The engine of the
red car is now running.
print (my_car.color)    # Output: red
```

Outside of the class, we have the following three lines.

```
my_car = Car("red")     # Sends the value parameter
```

```
"red" to the class.
my_car.start_engine()  # Output: The engine of the
red car is now running.
print (my_car.color)   # Output: red
```

The line my_car = Car("red") creates a new instance of the Car class, and "red" is passed as the color_parameter to the __init__ method. The next line, my_car.start_engine(), calls the start_engine method of the Car instance named my_car. Finally, the line print(my_car.color) demonstrates how to access the color attribute outside of the class and print its value to the screen.

It is very important to clarify the path of the "red" argument as it becomes the color attribute. This process is illustrated in Figure 7-4, where it is possible to see the three steps taken by the color parameter.

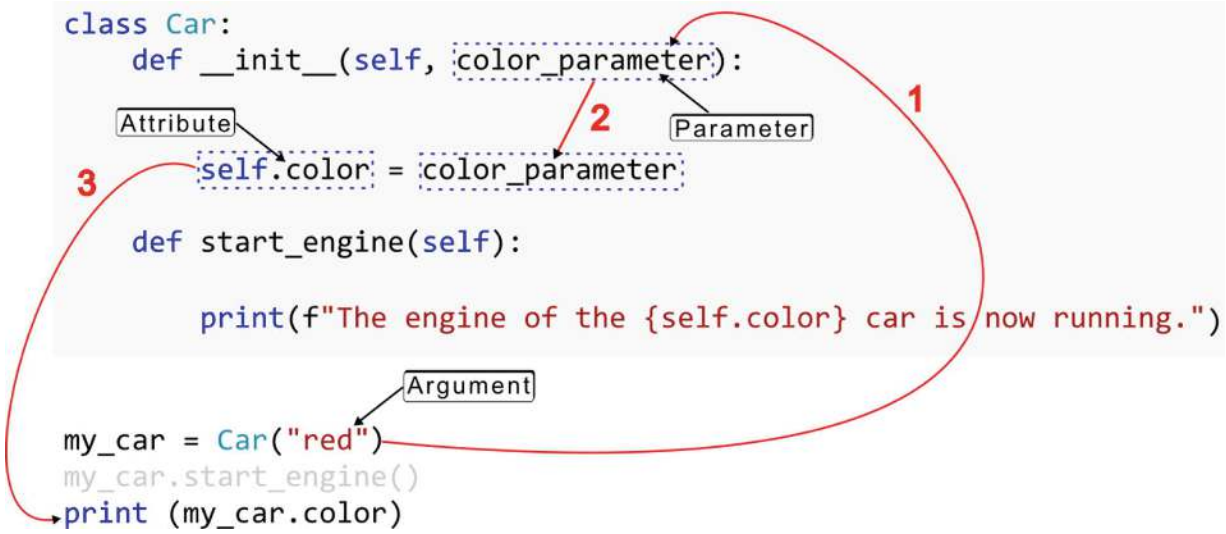


Figure 7-4 Flow of an argument to an attribute within a class

1) In step 1, the argument "red" is passed as the color_parameter to the __init__ method of the class during the creation of the object.
2) In step 2, inside the __init__ method, the value of color_parameter (which is "red") is assigned to the "self.color" attribute.
3) In step 3, when my_car.color is accessed, it retrieves the value "red".

A class attribute can be accessed through the class or an instance, whereas an instance attribute is accessed through its instance. Variables defined outside the class can be passed as arguments when an instance is created or assigned to instance attributes later.

To view the contents of an instance, you can use the special attribute __dict__ or the built-in function vars(). The __dict__ attribute, also called "dunder dict," is a dictionary that contains the instance attributes and can be used to read or write them. Similarly, vars() provides a convenient way to access an object's __dict__:

```
class Car:
    def __init__(self, color_parameter,
year_parameter):
        self.color = color_parameter
        self.year = year_parameter

# Create car instance with both parameters
my_car = Car("red", 2023)

# Show attributes using both methods
print(my_car.__dict__)  # Output: {'color': 'red',
'year': 2023}
print(vars(my_car))     # Output: {'color': 'red',
'year': 2023}
```

Subclasses

One class can be derived from another class, known as a superclass or parent class. The derived class, or subclass, can inherit all the attributes and methods of the parent class and may also have its own additional attributes and methods. The subclass can override the attributes and methods inherited from the parent class.

The different ways of working with attributes and methods will be explored in the following sections of this chapter.

```
class Car:
    def __init__(self, color):
        self.color = color

class ElectricCar(Car):
    def __init__(self, color, battery_capacity):
        super().__init__(color)
        self.battery_capacity = battery_capacity

    def start_engine(self):
        print(f"The {self.color} electric car is
running with {self.battery_capacity} kWh
battery.")

# Example usage:
my_electric_car = ElectricCar("blue", 50)
my_electric_car.start_engine()  # Output: The blue
electric car is running with 50 kWh battery.
```

In the example above, the ElectricCar class can inherit attributes and methods from the Car class.

An improved version of the “Car” class is used in the cited examples, as shown below. In this case, additional attributes and methods were added to expand the functionality of the class. For example, the methods used to start and stop the engine can now check its current status. Before starting the engine, the method verifies whether the engine is already running, as it is not possible to start an engine that is already on:

```
class Car:
    def __init__(self, color, car_type, year):
        self.color = color
        self.car_type = car_type  # e.g., sedan,
hatchback, pickup
```

```
        self.year = year  # Manufacturing year of
the car
        self.engine_running = False  # Tracks
whether the engine is running

    def start_engine(self):
        if not self.engine_running:
            self.engine_running = True
            print(f"The engine of the {self.color}
{self.year} {self.car_type} is now running.")
        else:
            print("The engine is already
running.")

    def stop_engine(self):
        if self.engine_running:
            self.engine_running = False
            print(f"The engine of the {self.color}
{self.year} {self.car_type} has been stopped.")
        else:
            print("The engine is already off.")

    def move(self):
        if self.engine_running:
            print(f"The {self.color} {self.year}
{self.car_type} is moving.")
        else:
            print("You need to start the engine
first before the car can move.")

my_car = Car("Blue", "Hatchback", 2022)
my_car.start_engine()  # Output: The engine of the
Blue 2022 Hatchback is now running.
my_car.move()          # Output: The Blue 2022
Hatchback is moving.
```

```
my_car.stop_engine()   # Output: The engine of the
Blue 2022 Hatchback has been stopped.
```

OOP Concepts

Now that the class object has been presented, it is possible to understand the four concepts or pillars of object-oriented programming (OOP). The examples for each concept are useful for understanding how it works.

Encapsulation

The first principle of object-oriented programming that we will study is encapsulation, which involves gathering data and its related methods within a single unit, typically a single class. It does not necessarily hide the data from the user or some part of the code but rather creates a capsule or wrapper around the attributes. This approach allows one to control access to the data, shielding it from accidental or unauthorized access and modification.

The color attribute is encapsulated within the classes, and we access it through methods or directly.

```
# Class: Car (for encapsulation)
class Car:
    def __init__(self, color):
        self.__color = color  # Private attribute

    def get_color(self):
        return self.__color

    def start_engine(self):
        print(f"The engine of the {self.__color}
car is now running.")

# Example usage
my_car = Car("blue")
```

```
print(my_car.get_color())  # Output: blue
my_car.start_engine()  # Output: The engine of the
blue car is now running.
```

Abstraction

The term *abstraction* is an object-oriented programming principle closely related to encapsulation and information hiding. Abstraction aims to hide unnecessary details from the user and make only the essential features of an object available. It allows developers to focus on high-level functionality without worrying about the underlying implementation details. By using abstraction, complex systems can be broken down into smaller, more manageable components, making the code easier to understand, maintain, and extend.

The Vehicle class provides a basic interface for starting an engine, hiding the internal details of how the engine works. The user only needs to call the start_engine method to start the engine, while the implementation details of this operation are encapsulated within the class and the method.

```
# Base class: Vehicle (for abstraction)
class Vehicle:
    def __init__(self, color):
        self.color = color

    def start_engine(self):
        print(f"The engine of the {self.color}
vehicle is now running.")

# Example usage
my_vehicle = Vehicle("red")
my_vehicle.start_engine()  # Output: The engine of
the red vehicle is now running.
```

Inheritance

In object-oriented programming, one class can inherit attributes and methods from another class. This is called inheritance. Inheritance promotes code reusability and establishes a relationship between parent and child classes, similar to a family tree where children inherit traits or skills from their parents, such as blue eyes, tall stature, or an ability for a specific sport.

Car and Motorcycle inherit from Vehicle, reusing its attributes and methods.

```
# Base class: Vehicle (for inheritance)
class Vehicle:
    def __init__(self, color):
        self.color = color

    def start_engine(self):
        print(f"The engine of the {self.color}
vehicle is now running.")

# Derived class: Car (for inheritance)
class Car(Vehicle):
    def __init__(self, color, model):
        super().__init__(color)
        self.model = model

    def honk(self):
        print(f"The {self.color} {self.model} car
is honking.")

# Example usage
my_car = Car("red", "Sedan")
my_car.start_engine()  # Output: The engine of the
red vehicle is now running.
my_car.honk()  # Output: The red Sedan car is
honking.
```

Polymorphism

Polymorphism is the ability of a class to take on multiple forms, depending on which subclass is implemented. It allows a single interface or method to represent different underlying data types or behaviors. Polymorphism enables objects of different classes to be treated as objects of a common base class, thereby promoting flexibility and reusability in code.

The code below is a clear demonstration of polymorphism, where objects of different classes—Vehicle, Car, Motorcycle—can be treated uniformly through a common interface, the start_engine() method, yet exhibit different behaviors based on their specific class implementations.

The start_engine() method is indeed overridden in the Motorcycle class, as well as in the Car class, allowing for different behavior for different types of vehicles while using the same method name.

```
class Vehicle:
    def __init__(self, color):
        self.color = color
    def start_engine(self):
        print(f"The engine of the {self.color}
vehicle is running.")

class Car(Vehicle):
    def start_engine(self):
        print(f"The engine of the {self.color} car
is running.")

class Motorcycle(Vehicle):
    def start_engine(self):
        print(f"The engine of the {self.color}
motorcycle is running.")

Vehicle("black").start_engine()
Car("red").start_engine()
```

```
Motorcycle("blue").start_engine()
```

Relationships Between Objects

In object-oriented programming (OOP), the objects of different classes can interact with each other. Python provides mechanisms to model these relationships based on three base relationships, aggregation, association, and composition. These relationships can be either unidirectional or bidirectional and can take various forms, including one-to-one, one-to-many, many-to-one, and many-to-many. The following sections explain these relationships in detail, accompanied by example code to enhance understanding and provide better illustration.

Table 7-3 Comparison of relationships between objects: association, aggregation, and composition

Association

Association is the most fundamental way to relate instances of two classes. It represents a relationship between two or more objects where one object uses or interacts with another, describing a “uses-a” relationship. This is a weak relationship, allowing objects to exist independently of each other.

In this example code a car can be associated with a driver, but they can exist independently. Car and Driver are associated; however, the driver can drive any car, and the car can be driven by any driver. They are not dependent on each other for existence:

```
class Driver:
    def __init__(self, name):
        self.name = name

    def drive(self, car):
        print(f"{self.name} is driving the
{car.color} car.")
```

```
class Car:
    def __init__(self, color_parameter):
        self.color = color_parameter

    def start_engine(self):
        print(f"The engine of the {self.color} car
is now running.")

my_car = Car("red")
my_driver = Driver("Alice")

my_car.start_engine()  # Output: The engine of the
red car is now running.
my_driver.drive(my_car)  # Output: Alice is
driving the red car.
```

Aggregation

Aggregation is a type of association that represents a relationship where one class acts as a collection or container for another class. In other words, where an object as a whole is composed of other objects. However, it is considered a weak relationship because the parts can exist independently of the whole. It describes a “has-a” relationship, where objects can exist independently of the other existing or not.

In the example code below, the wheels can exist independently of the car, but the car can contain multiple wheels. It is possible to extend the Car class to include a Wheel class:

```
class Wheel:
    def __init__(self, size):
        self.size = size

    def __str__(self):
        return f"Wheel of size {self.size}"

class Car:
```

```
    def __init__(self, color_parameter):
        self.color = color_parameter
        self.wheel = None  # Aggregation: Car can
have a wheel, but the wheel can exist
independently

    def add_wheel(self, wheel):
        self.wheel = wheel  # Add a single wheel
to the car

    def display_wheel(self):
        if self.wheel:
            print(self.wheel)
        else:
            print("No wheel attached to the car.")

my_car = Car("red")
front_wheel = Wheel(17)  # Create a wheel object
my_car.add_wheel(front_wheel)  # Add the wheel to
the car
my_car.display_wheel()  # Output: Wheel of size 17
```

Composition

Composition is a way of combining simple objects or data types to create more complex objects. In other words, it refers to an object that relies on other objects to form its structure. Composition is considered a stronger form of association, where an object often referred to as the “container” owns or manages the lifecycle of another object that it contains. This relationship is also known as a “part-of” relationship, meaning the contained object cannot exist independently without the container.

In this example code, a car is composed of an engine, and the engine cannot exist without the car. Then the engine is created within the car and cannot exist independently of it:

```
class Engine:
    def __init__(self):
        self.running = False

    def start(self):
        self.running = True
        print("Engine started.")

    def stop(self):
        self.running = False
        print("Engine stopped.")

class Car:
    def __init__(self, color_parameter):
        self.color = color_parameter
        self.engine = Engine()  # Composition: Car
owns the Engine

    def start_engine(self):
        self.engine.start()
        print(f"The engine of the {self.color} car
is now running.")

    def stop_engine(self):
        self.engine.stop()
        print(f"The engine of the {self.color} car
is now stopped.")

# Example usage
my_car = Car("red")
my_car.start_engine()  # Output (two lines):
Engine started. / The engine of the red car is now
running.
my_car.stop_engine()  # Output (two lines): Engine
stopped. / The engine of the red car is now
```

```
stopped.
```

Attributes

In Python, attributes can be classified into several categories. In this section, we will focus on the main types: class attributes, instance attributes, properties, and public attributes. An attribute can belong to more than one category. For example, a property is often used with a private attribute to control access to its data.

Table 7-4 Types of attributes in Python classes

In Python the default type of attribute in a class is **public**; they are used without any underscore prefix and are accessible from anywhere, both inside and outside the class.

Class attributes are used to share data or variables for all instances of a class, so they need to be defined directly within a class. Class attributes can be reached via the class name or instances, and all alteration impacts all instances.

Here is an example code with a class attribute:

```
class Car:
    # Class attribute
    wheels = 4  # All cars have 4 wheels

    def __init__(self, color_parameter):
        self.color = color_parameter

my_car = Car("red")    # Sends the value parameter
"red" to the class.
# Accessing the class attribute
print(f"All cars have {Car.wheels} wheels.")  #
Output: All cars have 4 wheels.

# Modifying the class attribute
```

```
Car.wheels = 6
print(f"Now all cars have {Car.wheels}
wheels.")  # Output: Now all cars have 6 wheels.
```

In this code, declaring a variable directly below the class definition and outside of any methods creates a class attribute. Class attributes are shared among all instances of the class, allowing data to be consistent across all objects. While it is possible to modify the value of a class attribute, it is generally recommended to access or modify it through the class itself rather than through individual instances for clarity and consistency.

Instance attributes, as the name suggests, are created within the constructor method of a class. The constructor method is typically the first method defined in a class and is most commonly implemented using the __init__ method. The constructor method is executed automatically when an instance of the class is created, allowing to initialize the instance with specific attributes or values. More details about the constructor method will be covered in the following sections of this chapter.

Instance attributes are unique to each instance of the class and are not shared among other instances.

Unlike class attributes, instance attributes are not copied from the parent class but are instead created and stored within the specific instance of the class.

Here is an example code using instance attributes, such as a variable or parameter called color_parameter:

```
class Car:
    def __init__(self, color_parameter):

        #Initialize the Car object with a single
attribute: color.
        self.color = color_parameter
```

```
my_car = Car("red")     # Sends the value parameter
"red" to the class.
print (my_car.color)    # Output: red
```

Table 7-5 offers a comparison of the class attributes and instance attributes.

Table 7-5 Comparison between class and instance attributes

Private and Protected Attributes

The attributes private and protected may be similar, but we will study the differences between the written form and application of each one.

Several programming languages adopt a way to enforce strict access control of attributes. These methods are called access modifiers and are represented by a keyword like private, public, or protected. Python does not enforce strict access control of attributes; instead it represents access modifiers by relying on conventions and name mangling to indicate that an attribute is intended to be private or protected.

Private attributes in Python are created by prefixing an attribute name with double underscores (__), such as __attribute or __attribute_name. This convention signals to other developers that these attributes are intended to be private and should not be accessed directly from outside the class. However, this is not a strict enforcement of privacy, as it is still possible to access these attributes using their mangled names.

The protected attributes are created by prefixing an attribute name with a single underscore (_), such as _attribute or _attribute_name. This convention signals to other developers that the attribute is intended for internal use within the class and its subclasses. However, this is not a strict enforcement of protection, as it is still possible to access these attributes directly from outside the class. The single underscore prefix serves as a warning to other developers to treat the attribute as protected, but it does not implement any technical restrictions on access.

In Python, the @property decorator can be used with private and protected attributes to control access through getter and setter methods, as discussed in the section "Methods."

The difference between protected and private attributes in Python lies in their level of access control and how they are intended to be used. Table 7-6 summarizes the difference between the two.

Table 7-6 Comparison between protected and private attributes

Dynamic Attributes and Static Attributes

Dynamic attributes and static attributes are types of attributes that can be defined and managed in a class.

Dynamic attributes are added to an object at runtime rather than being defined in the class itself. They can be created by directly assigning a value to an attribute on an object after it has been instantiated. This allows new attributes to be added to an object during code execution, even if those attributes are not defined in the class.

Dynamic attributes need to be created carefully; they can lead to unpredictable behavior of the code.

Static attributes, also known as class attributes, as explained earlier in this section are used to share data or variables for all instances of a class and ensure a predictable structure. They are defined directly in the class body, outside of any method, and are not tied to any specific instance of the class.

Table 7-7 summarizes the difference between dynamic attributes and static attributes.

Table 7-7 Comparison between dynamic and static attributes

Here is an example code using dynamic attributes:

```
class Car:
    def __init__(self, color, model):
        self.color = color
```

```
        self.model = model

    def honk(self):
        print(f"The {self.color} {self.model} car
is honking.")

my_car = Car("red", "Sedan")
my_car.honk()  # Output: The red Sedan car is
honking.

# Adding a dynamic attribute at runtime
my_car.mileage = 10000  # This creates a new
attribute 'mileage' for my_car
print(my_car.mileage)  # Output: 10000

# We can modify the dynamic attribute
my_car.mileage += 500
print(my_car.mileage)  # Output: 10500
```

Property Attributes

Property attributes are linked to methods and allow developers to control access to an attribute. This topic is covered in the next section.

Methods

As stated earlier, a method is similar to a function but is specifically designed to operate on an object's data, also called attributes. It is worth remembering that a class can be instantiated multiple times, as needed. Each instantiation creates a new and independent instance with its own attributes and methods. This enables the same class blueprint to instantiate multiple objects, each possessing distinct data and functionality.

First of all, it is necessary to explain the difference between an argument and a parameter. An **argument** is the actual value or variable passed when calling a class instance or a method.

A **parameter** is a variable specified in a method's definition that receives a value when the method is called.

In Python, methods can be categorized into six types, as outlined in Table 7-8.

Table 7-8 Types of methods in Python classes

The character @ is used to denote a feature called a decorator. Decorators are used with functions to extend or modify their behavior. More details about decorators will be studied in another chapter.

Instance Methods

The most common and simplest type of method used in Python is the instance method, as shown in the code example earlier. This type of method is tied to a specific instance of a class, allowing it to interact with and modify the instance's attributes. Instance methods are typically defined inside the class, usually after the constructor method __init__.

The following example illustrates a simple way to create and use an instance method:

```
class Car:
    def __init__(self, color_parameter):
        # Initialize the Car object with a single
attribute: color
        self.color = color_parameter

    def start_engine(self):  # Instance method
        print(f"The engine of the {self.color} car
is now running.")

my_car = Car("red")  # Creates an instance of the
Car class
my_car.start_engine()  # Calls the instance method
on the object
```

In this example, the “red” argument is passed to the class instance when it is created. The constructor method __init__ then initializes the color attribute of the instance with the value “red”.

Attributes of a class instance can be accessed and modified within instance methods using the keyword self. If an attribute is initialized by the __init__ constructor method, it becomes available to all instance methods through self. Alternatively, additional variables can be passed directly to an instance method as arguments when the method is called.

Methods with Arguments

It is also possible to pass arguments directly to a method, rather than passing them to the __init__ method. In this way, the arguments are received by the method and processed internally.

Here is an example code using an argument as a parameter in a method. In this case, the year 2020 is passed as an argument to the year_manufactory method, where it becomes the parameter year and is then printed:

```
class Car:
    def __init__(self, color_parameter):
        # Initialize the Car object with a single
attribute: color
        self.color = color_parameter

    def year_manufactory(self, year):  # Accept
year as a parameter
        # Print the car color and the year of
manufacture
        print(f"The {self.color} car was
manufactured in {year}.")

my_car = Car("red")  # Creates an instance of the
Car class with color
my_car.year_manufactory(2020)  # Pass the year
directly to the method
```

Class Method

Class methods allow you to use the methods without creating an instance of the class, operating directly on the class itself. They are methods bound only to the class. Class methods can modify class-level attributes and can be used as alternative constructors. To define a class method, use the @classmethod decorator, and the first parameter should be cls.

Class methods can be used in several scenarios, as described in Table 7-9.

Table 7-9 Common use cases for class methods

Class Method: Alternative Constructors

The next example code uses the class method default_car as an alternative constructor to create a Car object with a default color.

It allows the creation of a Car instance with the color set to "blue" without needing to explicitly pass the color parameter when instantiating the object. This is demonstrated by the line default_car = Car.default_car(), which calls the class method to create a new Car object with the default color:

```
class Car:
    def __init__(self, color_parameter):
        # Initialize the Car object with a single
attribute: color.
        self.color = color_parameter

    @classmethod
    def default_car(cls):
        # Class method to create a Car object with
a default color.
        return cls("blue")  # Creates a Car object
with the color "blue"
```

```
# Create a Car object with a specific color
my_car = Car("red")  # Sends the value parameter
"red" to the class.
print(my_car.color)  # Output: red

# Create a Car object using the class method
default_car = Car.default_car()  # Calls the class
method to create a default car
print(default_car.color)  # Output: blue
```

Class Method: Factory Methods

A factory method is used to create a new instance of a class, as demonstrated in the example below:

```
Factory Method:
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    @classmethod
    def create_electric_car(cls, brand):
        return cls(brand, "Electric")

electric_car = Car.create_electric_car("Tesla")
print(electric_car.brand, electric_car.model)  #
Output: Tesla Electric
```

This code above uses the class method called create_electric_car, marked with the @classmethod decorator, to provide an alternative way to create a Car instance.

Class Method: Class-Level Operations

Class methods can be used to access or modify class-level attributes used across all instances. The example code below includes a class

method called set_default_color that allows changing the default_color for all instances of the class:

```
class Car:
    default_color = "white"

    @classmethod
    def set_default_color(cls, color):
        cls.default_color = color

Car.set_default_color("blue")
print(Car.default_color)  # Output: blue
```

Another example is used below to find the name of the class:

```
class Car:
    @classmethod
    def get_class_name(cls):
        return cls.__name__

print(Car.get_class_name())  # Output: Car
```

Class Method: Inheritance and Polymorphism

Class methods provide a powerful mechanism for implementing inheritance and polymorphism:

```
class Car:
    def __init__(self, color):
        self.color = color
    def start_engine(self): print(f"The engine of
the {self.color} car is now running.")
    @classmethod
    def class_method(cls):
        return f"This is a {cls.__name__} class
method."
```

```
class ElectricCar(Car):
    @classmethod
    def class_method(cls):
        return f"This is an {cls.__name__} class
method."

my_car = Car("red")
print(Car.class_method())
my_electric_car = ElectricCar("blue")
print(ElectricCar.class_method())
```

In the given example, the ElectricCar class inherits from the Car class, gaining access to its methods and attributes, such as the color attribute and the start_engine method. The concept of polymorphism is demonstrated through the overriding of the class_method in the ElectricCar class. This allows ElectricCar to provide a different implementation of class_method while maintaining the same method name and interface. This polymorphic behavior is evident when calling class_method on Car and ElectricCar, resulting in different outputs based on the class type. Class methods, being called on the class itself rather than on instances, facilitate this inheritance and polymorphic behavior effectively.

Class Method: Utility Methods

Utility methods in class methods are particularly useful for providing functionality related to the class without the need for an instance. These methods can be used to perform operations that are relevant to the class as a whole, such as creating instances in different ways, managing class-level data, or providing helper functions that don't require instance-specific data.

The next example code uses the Car class to include a utility method that provides information about the class without needing an instance:

```
class Car:
    def __init__(self, color):
```

```
        self.color = color
    def start_engine(self):
        print(f"The engine of the {self.color} car
is now running.")
    @classmethod
    def info(cls):
        return f"{cls.__name__} class for car
objects."

my_car = Car("red")
my_car.start_engine()  # Output: The engine of the
red car is now running.
print(my_car.color)    # Output: red
print(Car.info())      # Output: Car class for car
objects.
```

This last example code demonstrates the use of a class method called info to provide utility information about the class without needing an instance.

Static Method

Static methods are used to define functions that are logically related to a class but do not require access to the class itself or its instances. To define a static method, the @staticmethod decorator is placed above the function. This type of method does not use the cls parameter (for the class) or the self parameter (for an instance).

Whenever it is necessary to keep functionality organized within a class but the method does not need to access or modify any attributes of the class or its instances, a static method is a good option.

In the following example, the static method is_valid_color is used to check if a given color is valid for a car:

```
class Car:
    def __init__(self, color_parameter):
        self.color = color_parameter
```

```
    @staticmethod
    def is_valid_color(color):
        valid_colors = ["red", "blue", "green"]
        return color in valid_colors

# Example usage
print(Car.is_valid_color("red"))    # Output: True
print(Car.is_valid_color("purple")) # Output:
False
```

Abstract Method

An **abstract method** is a method that is designed to be inherited and implemented by subclasses. In other words, it is a method that is declared in a base class but is not implemented in that class. Instead, it serves as a blueprint for subclasses, which are required to provide their own implementation of the method. Abstract methods ensure that certain functionality is defined in all subclasses, enforcing a consistent interface across them.

To create an abstract method, you must import the abstract base class ABC and use the @abstractmethod decorator just above the method to be used as abstract:

```
from abc import ABC, abstractmethod

# Abstract Base Class
class Car(ABC):
    def __init__(self, color_parameter):
        self.color = color_parameter

    @abstractmethod
    def start_engine(self):
        pass

# Concrete Subclass
class ElectricCar(Car):
```

```
    def start_engine(self):
        print(f"The {self.color} electric car's
engine is running silently.")

# Example usage
my_car = ElectricCar("blue")
my_car.start_engine()  # Output: The blue electric
car's engine is running silently.
```

In this last example the start_engine method is decorated with @abstractmethod, meaning it must be implemented in any subclass of Car.

Dunder Methods

Dunder methods are special methods that are written with a word or acronym surrounded by double underscores on both sides and are also called magic methods. They allow to customize the behavior of the classes in specific situations and can be used for several applications, as listed below:

- Object lifecycle
- Attribute access
- Container methods
- Numeric operations
- Comparison operators
- Pickling (process of serializing and deserializing objects)
- Miscellaneous (callable objects, context management, descriptors)

Some of these categories have already been used and explained in this chapter. To illustrate the use of dunder methods, example code will be provided.

Table 7-10 Dunder methods in Python, categorized by functionality

Object Lifecycle

Use __bool__ to get the boolean value of the object.

```
class Car:
    def __init__(self, color_parameter):
        self.color = color_parameter

    def __bool__(self):
        return bool(self.color)

my_car = Car("")
print(bool(my_car))  # Output: False
```

Attribute Access

The method __getattr__ is used to define behavior for accessing attributes that do not exist in an object.

```
class Car:
    def __init__(self, color_parameter):
        self.color = color_parameter

    def __getattr__(self, name):
        return f"{name} attribute not found!"

my_car = Car("red")
print(my_car.speed)  # Output: speed attribute not
found!
```

Container Methods

The __len__ method allows you to define the “length” of an object, typically for collections such as lists, dictionaries, or custom classes that are designed to behave like collections.

```
class Car:
    def __init__(self, passengers):
        self.passengers = passengers
```

```
    def __len__(self):
        return len(self.passengers)

my_car = Car(["Alice", "Bob"])
print(len(my_car))  # Output: 2
```

The __getitem__ method allows you to retrieve the item associated with a specified key or index.

```
class Car:
    def __init__(self, passengers):
        self.passengers = passengers

    def __getitem__(self, index):
        return self.passengers[index]

my_car = Car(["Alice", "Bob"])
print(my_car[0])  # Output: Alice
```

Numeric Operations

The __add__ method defines the behavior of the addition operator (+).

```
class Car:
    def __init__(self, speed):
        self.speed = speed

    def __add__(self, other):
        return Car(self.speed + other.speed)

    def __repr__(self):
        return f"Car(speed={self.speed})"

car1 = Car(50)
car2 = Car(70)
car3 = car1 + car2
print(car3)  # Output: Car(speed=120)
```

Comparison Operators

The __lt__ method is used to define the behavior of the less than (<) operator for custom objects.

```
class Car:
    def __init__(self, speed):
        self.speed = speed

    def __lt__(self, other):
        return self.speed < other.speed

car1 = Car(50)
car2 = Car(70)
print(car1 < car2)  # Output: True
```

Callable Objects

The __call__ method allows an instance of a class to be called as if it were a function.

```
class Car:
    def __init__(self, color):
        self.color = color

    def __call__(self):
        return f"The car is {self.color}"

my_car = Car("red")
print(my_car())  # Output: The car is red
```

Constructor Method

Constructors are a fundamental concept in OOP paradigm; they are automatically called when class instances, also called objects, are created before they are used. The constructor is a special method or function used to initialize an object when it is created and is

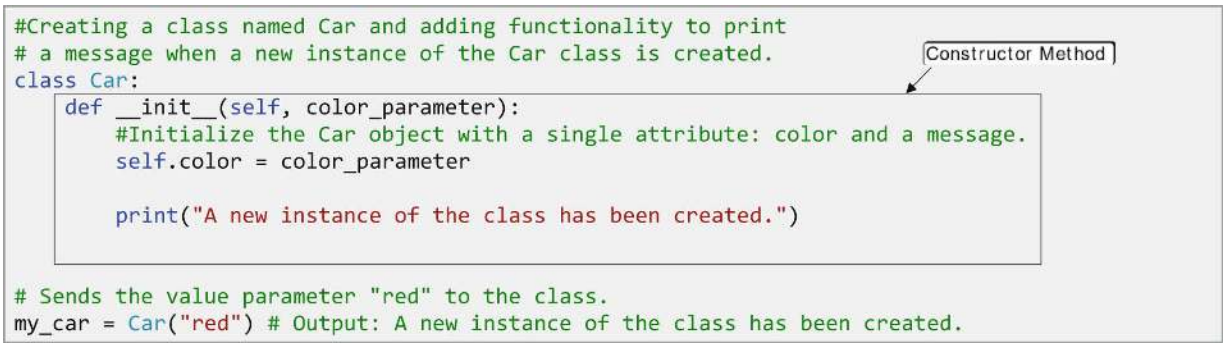
responsible for setting up the initial state of the object by assigning values to its attributes or executing a specific code.

In Python there are several special methods to initialize the attributes of an object created from an instance of the class. The main methods are listed in Table 7-11.

Table 7-11 Special methods for object construction and lifecycle management

These methods are written with a word or name surrounded by double underscores on both sides, which is why they are often referred to as “dunder” methods. The term dunder is short for double underscore.

The __init__ method is the most commonly used constructor; it is used to initialize the attributes of an object or perform any setup tasks when an object is created. It is possible to add several functionalities to execute during the creation of the object, such as adding a print statement to indicate that a new instance of the class has been created and setting up the initial state of the object, as shown in the example below:




```
#Creating a class named Car and adding functionality to print
# a message when a new instance of the Car class is created.
class Car:
    def __init__(self, color_parameter):
        #Initialize the Car object with a single attribute: color and a message.
        self.color = color_parameter

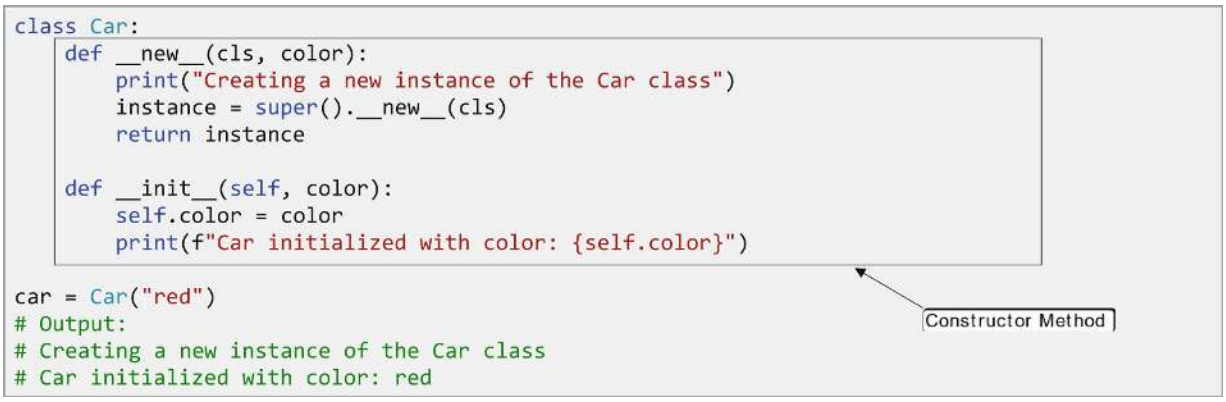
        print("A new instance of the class has been created.")

# Sends the value parameter "red" to the class.
my_car = Car("red") # Output: A new instance of the class has been created.
```

The color_parameter inside the __init__ method is received as a parameter, and the next line converts it into an attribute of the object. The __init__ method is called after __new__ has returned an instance.

The __new__ is a class method used for creating and returning a new instance of the class; this method allocates memory for the object, is called before __init__, and receives the class as its first argument. This is

made usually with cls, a convention to refer to the class itself. The difference between ___new__ and __init__ is __new__ is called before __init__ and __new__ creates the instance, while __init__ only initializes it. In other words __new__ is for instance creation, and __init__ is for instance initialization.



Notice the word super(); it is a function that gives access to methods and properties of a parent or sibling class. The line super().__new__(cls) is responsible for creating the new instance.

The __del__ method may be called when an object is being finalized, but its timing is not guaranteed. It should not be relied on for deterministic cleanup of external resources; context managers (with statements) are preferred for files, network connections, and similar resources. The method can still be useful in limited debugging demonstrations:

```
class Car:
    def __init__(self, color):
        self.color = color
        print(f"Car initialized with color:
{self.color}")

    def __del__(self):
        print(f"Car with color {self.color} is
being destroyed")
```

```
car = Car("red")
del car
# Output:
# Car initialized with color: red
# Car with color red is being destroyed
```

The __call__ method is used to create objects that can be called like functions. This is useful for creating function-like objects with state. It can be used to return new instances of a class or modify existing ones based on the arguments passed to it, enabling the creation of objects with varying attributes or behaviors without hard-coding them:

```
class Car:
    def __init__(self, color):
        self.color = color

    def __call__(self, new_color):
        print(f"Changing car color from
{self.color} to {new_color}")
        self.color = new_color

car = Car("red")
car("blue")  # Output: Changing car color from red
to blue
```

The __init_subclass__ method can alter the behavior of the sub-classes:

```
class Car:
    def __init_subclass__(cls, **kwargs):
        super().__init_subclass__(**kwargs)
        print(f"{cls.__name__} is a subclass of
Car")

class SportsCar(Car):
```

```
    pass
# Output: SportsCar is a subclass of Car

class Car:
    def __new__(cls, color_parameter):
        # Create a new instance of the class
        instance = super().__new__(cls)

        # Set the color attribute directly in
__new__
        instance.color = color_parameter

        # Print the message
        print("A new instance of the class has
been created.")

        return instance

# Create a new Car object
my_car = Car("red")  # Sends the value parameter
"red" to the class.

# Access the color attribute
print(my_car.color)  # Output: red
```

super() Built-In Function

The word super() used in the code above is a built-in function that allows you to call methods from a parent class or superclass without explicitly naming the parent class and ensure that the parent class's methods are properly invoked.

Here is an example code using super() that allows to extend the functionality of the Car class while still maintaining its original behavior:

```
class Car:
```

```
    def __init__(self, color):
        self.color = color;
        print("Car created.")

class ElectricCar(Car):
    def __init__(self, color, battery):
        super().__init__(color)
        self.battery = battery
        print(f"Electric car, {battery} kWh
battery.")

my_electric_car = ElectricCar("blue", 75)
print(f"Color: {my_electric_car.color}, Battery:
{my_electric_car.battery} kWh")

# Output: # Car created. # Electric car, 75 kWh
battery. # Color: blue, Battery: 75 kWh
```

The subclass ElectricCar receives a color parameter common to all cars and a battery parameter that is specific to electric cars. In the sequence, the line super().__init__(color) calls the parent class Car and passes the color parameter; after these steps the parameter color becomes an attribute of the class, and it is possible to use this attribute as my_electric_car.color.

self and cls

The word **self** refers to the current instance of the class and is used to access the attributes and methods of the instance. On the other hand, **cls** refers only to the class itself, rather than an instance of the class.

To refer to instances or classes, self and cls are conventions, which means that they are names commonly used by convention, but are not mandatory in Python and can even be replaced by other words or

acronyms, as long as reserved words or language keywords such as False, True, if, for, while, def, class, etc. are not used.

The word self is used in instance methods to access or modify the attributes and methods of a specific instance or object, and cls is used in class methods to access or modify class-level attributes or methods. With cls it also is possible to access only the class-level attributes, but it is possible to explicitly pass an instance to it.

```
class Car:
    def __init__(self, color_parameter):
        self.color = color_parameter
        print("A new instance of the class has
been created.")

    def change_color(self, new_color):
        self.color = new_color
        print(f"Car color changed to
{self.color}")

my_car = Car("red")  # Output: A new instance of
the class has been created.
my_car.change_color("blue")  # Output: Car color
changed to blue
```

In the code example above, self.color assigns the value of the color_parameter parameter to an instance attribute. The keyword self references the current instance, making the attribute accessible within the class or its subclasses.

The code below defines a Car class that can track the total number of instances created and allows you to retrieve that count. Each time a new instance of the Car class is created, the __init__ method increments the total_cars attribute by 1. The get_total_cars method is defined to retrieve the value of the total_cars attribute. However, it is a regular instance method, not a class method, so it does not automatically receive the class as an argument. Instead, when calling

Car.get_total_cars(Car), the Car class is explicitly passed as the cls argument, allowing the method to access the total_cars attribute of the class:

```
class Car:
    total_cars = 0
    def __init__(self, color_parameter):
        self.color = color_parameter
        Car.total_cars += 1
        print("A new instance of the class has
been created.")

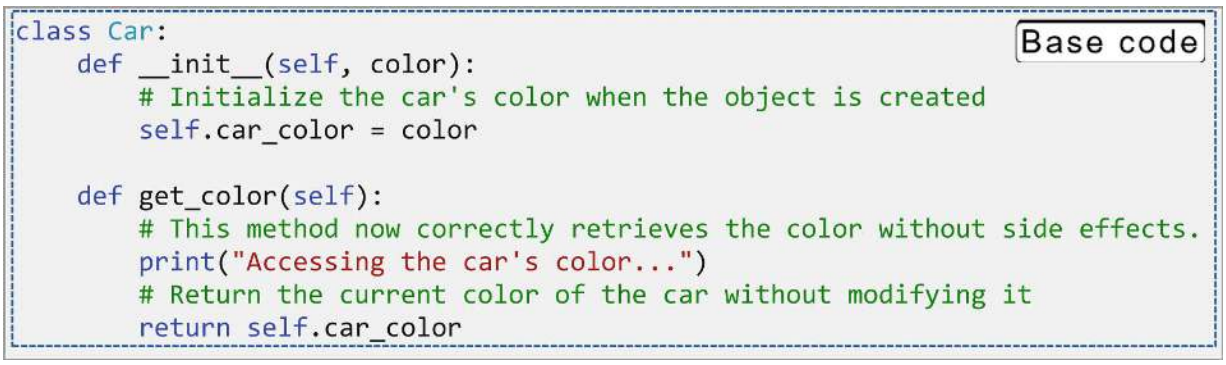
    def get_total_cars(cls):
        return cls.total_cars

my_car1 = Car("red")  # Output: A new instance of
the class has been created.
my_car2 = Car("blue")  # Output: A new instance of
the class has been created.
print(Car.get_total_cars(Car))  # Output: 2
```

Property Method

In Python, it is possible to use a method that acts like an attribute, providing a way to control access to the attribute while maintaining a clean and intuitive interface.

To understand how a property method works, see the code example below:



```
car = Car("Red")                                                    Cliente code

# 1. Access the car's color directly.
print(f"Initial color (direct access): {car.car_color}")
# Output: Initial color (direct access): Red

# 2. Access the car's color using the corrected get_color method.
print(f"First call to get_color(): {car.get_color()}")
# Output:First call to get_color(): Red

# 3. Subsequent calls now return the original, unmodified color.
print(f"Second call to get_color(): {car.get_color()}")
# Output: Second call to get_color(): Red

print(f"Color after all calls (direct access): {car.car_color}")
# Output: Color after all calls (direct access): Red
```

In programming, the concepts of **base code** and **client code** are very common. These terms refer to the relationship between a reusable module or library called the base code and the code that uses it, called the client code.

In order to illustrate this concept, two rectangles were placed in the code above. The first rectangle represents the base code, which is reusable and can be accessed by multiple client codes. The second rectangle represents the client code, which may be used by different users or computers.

An interesting aspect of this code is the get_color method. It retrieves and returns the car's color while also printing "Accessing the car's color …" to the console. You might be wondering why this is useful.

The answer is simple: the get_color method allows you to modify the variable name only in the base code. This means that client code does not need to be updated if the variable name changes, as it interacts with the method instead of accessing the car_color variable directly. This approach improves code maintainability and avoids the rework of updating variable names in multiple places.

In Python, the @property decorator is available to define a method that can act like an attribute. This decorator provides encapsulation capabilities, offering data hiding and protection for class attributes. It

allows controlled access to read or modify the data within these attributes. The @property decorator also enhances abstraction, as users of the class don't need to know how the attribute is stored or managed internally; they only need to understand how to interact with it through the property interface.

The @property decorator in Python allows for indirect access to an attribute through a method, which is referred to as a getter. Similarly, it enables indirect modification of an attribute, known as a setter. The getter is implemented using a method decorated with @property directly above it. In contrast, the setter is implemented using a method decorated with @<attribute_name>.setter, where <attribute_name> is the name of the property defined by the getter.

Within these getter and setter methods, attributes often follow the convention of using double underscores before the attribute name, such as __attribute_name, which is a common convention for indicating private attributes in Python. Both getter and setter methods allow for additional code to be executed when an attribute is accessed or modified. This capability is useful for performing operations such as data validation, ensuring the data is in the correct format, like uppercase or lowercase, checking for empty strings, and raising a ValueError if the input is invalid. These features contribute to data integrity and encapsulation while providing a clean and intuitive interface for interacting with the attributes.

Here is an example of the @property decorator using getter and setter operations:

```
# Class: Car with @property for encapsulation.
class Car:
    def __init__(self, color):
        self.__color = color  # Private attribute

    @property
    def color(self):
        """Getter for the car's color."""
```

```
        return self.__color

    @color.setter
    def color(self, new_color):
        """Setter for the car's color."""
        print(f"Changing color from {self.__color}
to {new_color}")
        self.__color = new_color

    def start_engine(self):
        print(f"The engine of the {self.__color}
car is now running.")

my_car = Car("blue")
# Accessing the color using the getter.
print(my_car.color)  # Output: blue.

# Changing the color using the setter.
my_car.color = "red"
print(my_car.color)  # Output: red.

my_car.start_engine()  # Output: The engine of the
red car is now running.
```

In the Car class, the @property decorator is used to create a getter method for the color attribute, allowing it to be accessed like a regular attribute called my_car.color, while internally calling the getter method to return the private __color attribute. The @color.setter decorator is used to define a setter method, enabling controlled modification of the color attribute.

Another example using validation code is shown in the example code below. In this case, the attributes are filtered by two conditions: the string should not be a number, nor should it be a string in uppercase. A ValueError is raised if the input is invalid, thus ensuring

data integrity and encapsulation while providing a clean and intuitive interface for interacting with the color attribute:

```
# Class: Car with validation in @property.
class Car:
    def __init__(self, color):
        self.color = color

    @property
    def color(self):
        """Getter for the car's color."""
        return self.__color

    @color.setter
    def color(self, new_color):
        """Setter with validation for the car's
color."""
        if not isinstance(new_color, str):
            raise ValueError("Color must be a
string.")
        if not new_color.islower():
            raise ValueError("Color must be in
lowercase letters only.")
        if any(char.isdigit() for char in
new_color):
            raise ValueError("Color cannot contain
numbers.")
        self.__color = new_color

    def start_engine(self):
        print(f"The engine of the {self.__color}
car is now running.")

try:
    my_car = Car("blue")  # Valid color.
```

```
    print(my_car.color)  # Output: blue.

    my_car.color = "red"  # Valid color.
    print(my_car.color)  # Output: red.

    my_car.color = "Red"  # Invalid: uppercase.
except ValueError as e:
    print(e)  # Output: Color must be in lowercase
letters only.

try:
    my_car.color = "blue123"  # Invalid: contains
numbers.
except ValueError as e:
    print(e)  # Output: Color cannot contain
numbers.

try:
    my_car.color = 123  # Invalid: not a string.
except ValueError as e:
    print(e)  # Output: Color must be a string.

my_car.start_engine()  # Output: The engine of the
red car is now running.
```

Decorators with OOP Principles

The decorator introduced in Chapter 5 can also be implemented in object-oriented programming (OOP) as a class that wraps another object. However, there are differences between function-based decorators and class-based decorators. In function-based decorators, the original function is typically represented by a variable, often named func, and the additional behavior is implemented in a nested function, commonly called wrapper. In contrast, class-based decorators use the __call__ method to make the object callable, eliminating the need for a

wrapper function. Table 7-12 compares these two approaches to implementing decorators.

Table 7-12 Comparison between function-based and class-based decorators

The next example code uses the MathDecorator class, which modifies the behavior of the add_numbers function. The decorator wraps the function and, when called, computes the sum of the two input numbers by executing the original function. It then calculates the square (power of 2) of the result and prints it.

However, the decorator returns the original result of the function, not the squared value. When add_numbers(5, 7) is called, the sum (12) is computed, the square of the result (144) is printed, and the original result (12) is returned and printed separately. This demonstrates how a class-based decorator can add additional functionality like calculating and printing the result without altering the original function's return value:

```
class MathDecorator:
    def __init__(self, func):
        self.func = func  # Store the original
function

    def __call__(self, *args, **kwargs):
        result = self.func(*args, **kwargs)
        power_of_2 = result ** 2
        print(f"The power of 2 of the result is:
{power_of_2}")
        return result

@MathDecorator
def add_numbers(a, b):
    return a + b

# Call the decorated function
```

```
y = add_numbers(5, 7)
print(f"The result of the operation is: {y}")
# Output:
# The power of 2 of the result is: 144
# The result of the operation is: 12
```

Dependency Injection

A very important topic used in Python and OOP is dependency injection, but to understand what it is and how to use it, it is necessary to know other two topics, dependency and design pattern.

Dependency

A dependency occurs when a class or object needs another class to function. For example, if the principal class uses a second class to perform some task, we say that the principal class depends on the second class. In this example case, the Car class depends on an Engine class. The Car class creates an instance of Engine internally, making it tightly coupled to that specific implementation of Engine.

```
class Engine:
    def start(self):
        print("The engine is now running.")

# Define a Car class that depends on the Engine
class
class Car:
    def __init__(self, color_parameter):
        # The Car class takes a color as a
parameter and assigns it to the car
        self.color = color_parameter
        # The Car class directly creates an
instance of the Engine class
        # This is an example of tight coupling, as
the Car class is dependent on the Engine class
```

```
        self.engine = Engine()

    def start_engine(self):
        print(f"The {self.color} car is preparing
to start the engine.")
        self.engine.start()

# Call the start_engine method to start the car's
engine
my_car = Car("red")
my_car.start_engine()
# Output:
# "The red car is preparing to start the engine."
# "The engine is now running."
```

Design Patterns

Over time, developers noticed that certain problems were recurring, even in different projects; then automatically they remembered how they solved similar challenges in the past codes. Then design patterns emerged as a high-level definition of how a common problem can be solved; it is not a framework or ready-made code but a template or guideline that can be adapted to solve specific design challenges. In other words, a design pattern is a reusable solution to a common problem that occurs in software design. It can be applied to various programming paradigms, but its functionalities are most effectively utilized in object-oriented programming–based code.

The origin of design patterns can be traced back to 1977 with the publication of the book *A Pattern Language: Towns, Buildings, Construction*. In this book, the authors cataloged 253 types of problems or design challenges, analyzed the underlying principles of each situation, described them in their essence, and proposed standard solutions. The number of standard solutions or design patterns in software engineering has grown significantly since the concept was introduced. The book *Design Patterns: Elements of Reusable Object-*

Oriented Software by the Gang of Four (GoF), published in 1994, cataloged 23 design patterns, making the design pattern even better known. It's challenging to pinpoint the precise count of design patterns since they are constantly evolving, with new ones emerging and established ones being modified or tailored for different situations.

Dependency Injection

Dependency injection is a method in software design where a class's required components are supplied externally, rather than the class generating them on its own. Instead of a class producing the necessary objects internally, these objects are provided to the class, typically via its constructor or through method arguments. In other words, dependency injection is a computer program development pattern used to avoid high level of code coupling within an application. This approach promotes more modular and testable code, making it easier to change or replace dependencies without modifying the dependent class. It provides flexibility, reusability, and scalability of the code.

In the example below, the classes Engine and ElectricEngine are injected into the Car class, allowing us to change the behavior of the car without modifying its code. This demonstrates the flexibility and testability that dependency injection provides:

```
class Engine:
    def start(self):
        return "vroom"

class ElectricEngine:
    def start(self):
        return "silent hum"

class Car:
    def __init__(self, color_parameter, engine):
        self.color = color_parameter
        self.engine = engine
```

```
    def start_engine(self):
        engine_sound = self.engine.start()
        print(f"The {engine_sound} of the
{self.color} car is now running.")

gas_engine = Engine()
electric_engine = ElectricEngine()

gas_car = Car("red", gas_engine)
gas_car.start_engine()  # Output: The vroom of the
red car is now running.

electric_car = Car("blue", electric_engine)
electric_car.start_engine()  # Output: The silent
hum of the blue car is now running.
```

Walking Further in OOP

The universe of object-oriented programming (OOP) is vast. As this book is not focused solely on OOP, this chapter covers the core fundamentals in Python as a foundation for more advanced topics. The following topics are recommended for further study:

- Design patterns: Templates or guidelines that can be adapted to solve recurring software design challenges.
- SOLID principles: A set of five design principles intended to make software architecture more understandable and flexible.
- UML diagrams: Standardized graphical representations used to visualize, model, and document software systems.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_8

8. Files and Paths

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Overview of File Types in Python

Since the focus of almost all engineering or scientific work is the manipulation of file data, in this chapter, we will explore how to work with the various file types supported by the Python ecosystem and the most efficient ways to access a file or directory path.

File-path syntax differs by operating system. Windows usually uses backslashes (\) and drive letters (e.g., C:\), whereas Linux and macOS use forward slashes (/) and a single root directory. Windows paths are usually case-insensitive, while Linux and macOS paths are generally case-sensitive. Relative names such as data.txt are portable, but Python resolves them against the current working directory, which must be writable for write operations. Later examples use pathlib to construct paths without hard-coding separators.

Python provides several tools for handling the most common file types used worldwide.

Table 8-1 lists some file types and their respective Python libraries or frameworks.

Table 8-1 A summary of common file types, their descriptions, and the primary Python libraries or frameworks used to handle them

In this chapter, we will explore how to handle files commonly used in engineering applications, including text, spreadsheets, PDFs, and DXFs.

Working with the open() Function

The built-in open() function provides byte- and text-level access to files. Table 8-2 lists common text-based formats that can be opened directly; parsing, validation, or conversion may require an additional library. Formats such as .docx and .xlsx are covered later in the chapter.

Table 8-2 Common text-based formats that can be accessed with Python's open() function and typical operations performed with supporting parsers

The open() function uses seven basic mode characters, which can also be combined. The most common are ‘r’ for reading and ‘w’ for writing.

Table 8-3 File manipulation modes available in the open() function

The open() function can also be used without a context manager, but the file must then be closed explicitly. Failing to close a writable file can leave buffered data unwritten and may keep operating system resources or file locks in use.

```
file = open('data.txt', 'r')
print(file.read())
file.close()
```

It is also possible to use the try and finally statements to close a file. These statements will be covered in the next chapters, but their usage has declined due to the convenience of the with statement.

The most commonly used and safest way to handle files is by using the open() function in combination with the with statement. The with statement ensures that resources are properly managed and is commonly used when working with files, network connections, or database connections, guaranteeing that the resource is closed or released after use, even if an error occurs.

Another use of the with statement is to release locks in multithreaded programs. It is an essential tool in Python for managing resources, preventing errors, and making code more readable and easier to understand.

Inside a with open() block, several methods and attributes are available. Tables 8-4 to 8-8 summarize the most common ones.

Table 8-4 Common commands for reading files

Table 8-5 Common commands for writing to files

Table 8-6 Common commands for positioning the cursor within a file

Table 8-7 Czommon commands for closing files

Table 8-8 Other useful file-handling commands

A relative filename is resolved from the current working directory. A full Windows path can be written as a raw string so that backslashes are not interpreted as escape sequences. The example below is Windows-specific. The example below demonstrates

how to specify a Windows file path using a raw string to ensure backslashes are treated as literal characters:

```
# Windows-specific raw string example
file_path = r"C:\Users\Eng\Python_codes\data.txt"
with open(file_path, 'r', encoding='utf-8') as file:
    content = file.read()
    print(content)
```

This matters because backslashes are used in escape sequences such as newline (\n) and tab (\t). A raw string keeps those backslashes literal.

An example of each operation using a .txt file is shown in Table 8-9.

Table 8-9 A summary of basic file operations demonstrated in the chapter for .txt files

1. *Write Mode*

   The code below opens data.txt in write mode and writes the text “Hello engineers!” to the file.

   If the file does not exist, Python creates it; if it already exists, write mode replaces its previous contents.

   The with statement closes the file automatically when the write operation finishes, including when an exception occurs.

   ```
   # Open 'data.txt' in write mode
   with open('data.txt', 'w', encoding='utf-8') as file:
       # Write content to the file
       file.write("Hello engineers!")
   ```

   After the code runs, data.txt contains the text “Hello engineers!”.

   Write Mode with a Specific Path

   For portable code, pathlib can build a path from operating system–independent components. This example stores the file in a Python_codes directory inside the current user’s home directory and creates the directory if needed.

   ```
   from pathlib import Path
   file_path = Path.home() / "Python_codes" / "data.txt"
   file_path.parent.mkdir(parents=True, exist_ok=True)
   with file_path.open('w', encoding='utf-8') as file:
       file.write("Hello engineers!")
   ```

2. *Read Mode*

   This example opens data.txt for reading, stores the complete contents in x, and

prints the result. The context manager closes the file automatically.

```
# Use the 'with' statement to open 'data.txt' in read mode.
with open('data.txt', 'r') as file:
    # Read the entire content of the file and assign it to
variable 'x'
    x = file.read()
    # Print the content of the file stored in variable 'x'
    print(x)
```

3. *Exclusive Creation Mode*

This code demonstrates the use of exclusive creation mode 'x' when opening a file in Python. When using the 'x' mode, the file is created only if it does not already exist. If the file is already present, a FileExistsError is raised.

The example attempts to create newfile.txt and write a short message. If the file already exists, FileExistsError is caught so that existing data is not overwritten.

```
try:
    with open('newfile.txt', 'x') as file:
        file.write("This file is created exclusively.")
except FileExistsError:
    print("File already exists!")
```

4. *Append Mode*

Append mode ('a') adds new content to the end of data.txt without replacing its current contents. If the file does not exist, Python creates it.

```
with open('data.txt', 'a') as file:
    file.write("\nThis text is appended to the existing
content.")
```

The leading "\n" starts the appended text on a new line.

Unlike write mode, append mode preserves the existing contents and is useful for logs or incremental updates.

5. *Binary Mode*

The binary example writes the ASCII bytes for "Hello" to data.bin. The b prefix creates a bytes literal, and each \xNN escape represents one hexadecimal byte.

```
with open('data.bin', 'wb') as file:
    file.write(b'\x48\x65\x6C\x6C\x6F')  # Writing "Hello"
in binary
```

6. *Text Mode*

Text Mode (Reading)

Opening data.txt with ‘rt’ reads and decodes the file as text. Because the example specifies UTF-8 and text mode is the default, ‘r’ would behave the same way.

```
with open('data.txt', 'rt', encoding='utf-8') as file:  #
'r' is equivalent
    print(file.read())  # Reads file in text mode
```

Text Mode (Writing)
With ‘wt’, Python writes UTF-8 text to example.txt, replacing existing content or creating the file when necessary. The shorter mode ‘w’ is equivalent.

```
with open('example.txt', 'wt', encoding='utf-8') as
file:  # 'w' is equivalent
    file.write("Hello, this is a text file in write mode.")
```

Text Mode (Appending)
With ‘at’, the new UTF-8 text is added to the end of example.txt without replacing earlier content. The mode ‘a’ is equivalent, and either form creates the file if it is missing.

```
with open('example.txt', 'at', encoding='utf-8') as
file:  # 'a' is equivalent
    file.write("\nThis line is appended to the file.")
```

Text Mode (Reading and Writing)
The mode ‘r+t’ permits both reading and writing without truncating the file; ‘r+’ is its shorter equivalent.

```
# Use r+t or r+ for text reading and writing.
with open('example.txt', 'r+t') as file:
    content = file.read()
    print("Original Content:", content)

    file.write("\nAdding new text while reading.")  # The
previous read left the pointer at the end
    print("Original Content:", content)
```

The code reads and prints the original contents. Because read() leaves the file pointer at the end, the following write() appends new text. The content variable is unchanged because it still contains the string returned by the earlier read.

7. *Read and Write Mode*
The ‘r+’ example also reads and writes without truncation. The file pointer begins at

The 'r+' example also reads and writes without truncation. The file pointer begins at the start, but after read() it is positioned at the end of the content.

```
with open('data.txt', 'r+') as file:  # Read and write
without truncating
    content = file.read()
    print(content)
    file.write("\nAdditional text written after reading the
content.")
    print(content)
```

The code first reads the entire file content using file.read() and stores it in the content variable. It then prints the original content.

However, when writing to the file using file.write(), the file pointer is already at the end of the file due to the previous read() operation. Therefore, the new text is appended to the end of the file, rather than overwriting the existing content.

The last line prints the original content again, showing that the content variable still holds the original content, as the file's content has been modified after it was read.

The following section presents additional examples of file manipulation operations, starting with writing to a specific line.

8. *Writing to a Specific Line*

A text file has no direct "replace line" operation, so the usual approach is to read the lines, modify the list, and write the complete list back.

The example replaces the third line (lines[2]). If the file contains fewer than three lines, blank lines are added first.

```
# Write to a specific line in a file
filename = 'data.txt'

# Read existing lines
with open(filename, 'r') as file:
    lines = file.readlines()

# Extend lines if needed
while len(lines) < 3:
    lines.append('\n')

# Write to the specific line (line 3 in this example)
lines[2] = "New content for line 3\n"

# Write back to the file
with open(filename, 'w') as file:
    file.writelines(lines)
```

The procedure has four steps:

1. Read the existing lines with readlines().
2. Add blank lines until the list contains at least three entries.
3. Replace the third entry, including its newline character.
4. Open the file in write mode and save the updated list with writelines().

9. *Reading a Specific Line*

This example reads the third line from data.txt.

list(file)[2] uses zero-based indexing, and strip() removes leading and trailing whitespace before the line is printed.

```
# Read a specific line from a file
filename = 'data.txt'

# Read specific line (line 3 in this example)
with open(filename, 'r') as file:
    # Use list comprehension or indexing to get the
specific line
    specific_line = list(file)[2].strip()
    print(f"Content of line 3: {specific_line}")
```

10. *Reading Multiple Lines*

This example reads lines 2–4 from a text file.

The slice [1:4] selects those lines because Python indexing starts at zero. Each selected line is printed after strip() removes surrounding whitespace.

```
# Read multiple lines from a file
filename = 'data.txt'

# Read lines 2 to 4
with open(filename, 'r') as file:
    lines = list(file)[1:4]
    print("Lines 2-4:")
    for line in lines:
        print(line.strip())
```

CSV Files

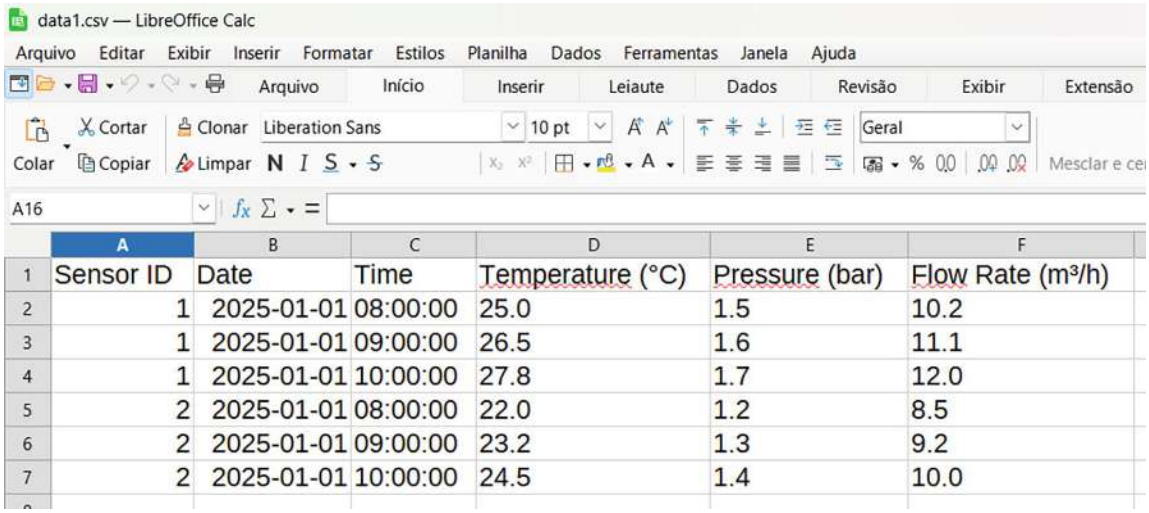
After the basic text-file modes, the next examples apply open() and the csv module to comma-separated values files.

A comma-separated values (CSV) file is a plain-text format for tabular data. Each physical line normally represents a row, and a delimiter—commonly a comma—separates the fields.

CSV is widely used to exchange data among applications and to store datasets in research, web development, finance, machine learning, and database workflows.

```
"Sensor ID","Date","Time","Temperature (°C)","Pressure
(bar)","Flow Rate (m³/h)"
"1","2025-01-01","08:00:00","25.0","1.5","10.2"
"1","2025-01-01","09:00:00","26.5","1.6","11.1"
"1","2025-01-01","10:00:00","27.8","1.7","12.0"
"2","2025-01-01","08:00:00","22.0","1.2","8.5"
"2","2025-01-01","09:00:00","23.2","1.3","9.2"
"2","2025-01-01","10:00:00","24.5","1.4","10.0"
```

CSV readers must account for quoting because a field can contain the delimiter itself. For example, a comma inside a quoted value should remain part of that field rather than create another column.



	A	B	C	D	E	F
1	Sensor ID	Date	Time	Temperature (°C)	Pressure (bar)	Flow Rate (m³/h)
2	1	2025-01-01	08:00:00	25.0	1.5	10.2
3	1	2025-01-01	09:00:00	26.5	1.6	11.1
4	1	2025-01-01	10:00:00	27.8	1.7	12.0
5	2	2025-01-01	08:00:00	22.0	1.2	8.5
6	2	2025-01-01	09:00:00	23.2	1.3	9.2
7	2	2025-01-01	10:00:00	24.5	1.4	10.0

Figure 8-1 An example of a .csv file containing sensor data opened in LibreOffice Calc

Python provides the csv module for reading and writing tabular data. Import it as shown below:

```
import csv
```

Writing to .csv files using open() follows a similar approach to that used for .txt files.

Use \n to start a new line. Inside a double-quoted Python string, an embedded double quote can be escaped as \"; alternatively, enclose the outer string in single quotes.

Consider the following sentence:

```
"He said "Hello" to me"
```

In this invalid expression, the quote before Hello ends the string too early, so the remaining text cannot be parsed as Python code.

The following line therefore raises SyntaxError:

```
print("He said "Hello" to me")  # SyntaxError: invalid syntax
```

Escape the inner quotation marks:

```
print("He said \"Hello\" to me")  # Correct
```

Or use single quotation marks around the complete string:

```
print('He said "Hello" to me')  # Correct
```

The next example creates a small CSV file directly with open():

```
# Create a CSV file
with open('data.csv', 'w', newline='', encoding='utf-8') as
file:
    file.write("\"Sensor ID\",\"Date\",\"Time\",\"Temperature
(°C)\",\"Pressure (bar)\",\"Flow Rate (m³/h)\"\n")
    file.write("\"1\",\"2025-01-
01\",\"08:00:00\",\"25.0\",\"1.5\",\"10.2\"\n")
```

The code creates data.csv with a header and one data record. Each call to write() supplies a complete CSV row followed by a newline character.

Although CSV content can be written manually, the csv module is safer for structured data because it handles quoting, delimiters, and embedded newlines. It does not infer or preserve numeric data types automatically; values read by csv.reader() are strings unless the program converts them.

Using open() alone often leads to manual splitting, which fails when a quoted field contains a delimiter such as the comma in “Hello, World”.

Table 8-10 summarizes the CSV operations demonstrated below.

Table 8-10 An index of common operations for .csv files

Table 8-11 A summary of commands used for various .csv file operations

1. *Read an Entire CSV File As a String.*

```
# Read entire CSV file as a string
with open('data.csv', 'r') as file:
    content = file.read()
print("Entire File Content:", content)
```

file.read() returns one string containing the complete file. This is useful for inspection, but it does not parse rows or columns.

2. *Read All Lines.*

```
# Read all lines into a list
with open('data.csv', 'r') as file:
    lines = file.readlines()
print("All Lines:", [line.strip() for line in lines])
```

readlines() returns one string per physical line. It preserves line boundaries but does not interpret CSV quoting or delimiters.

3. *Read the First Line.*

```
# Read first line
with open('data.csv', 'r') as file:
    first_line = file.readline().strip()
print("First Line:", first_line)
```

readline() reads only the first line, which is useful for previewing a header without loading the complete file.

4. *Read a Specific Line.*

```
# Read a specific line
line_index = 2
with open('data.csv', 'r') as file:
    line = file.readlines()[line_index].strip()
print("Specific Line:", line)
```

readlines()[line_index] retrieves a line by zero-based index. Because readlines() loads the file into memory, this approach is best for small files.

5. *Read a Specific Cell.*

```
# Read a specific cell
import csv
row_idx, col_idx = 1, 1
with open('data.csv', 'r') as file:
```

```
with open('data.csv', 'r') as file:
    rows = list(csv.reader(file))

    cell = rows[row_idx][col_idx].strip()
print(f"Cell ({row_idx+1},{col_idx+1}):", cell)
```

csv.reader() parses the file into rows, allowing access by row and column index. Validate both indexes before using values supplied at runtime.

6. *Iterate Through Lines.*

```
# Iterate through lines
import csv
with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

csv.reader() can be iterated directly, so each row is processed without loading the entire file into memory.

7. *Read a Specific Row.*

```
# Read a specific row
import csv
row_number = 2
with open('data.csv', 'r') as file:
    reader = list(csv.reader(file))
    print("Specific Row:", reader[row_number])
```

Converting csv.reader() to a list makes indexed row access simple, but stores the whole file in memory.

8. *Convert .csv to a List.*

```
# Convert CSV file to a list
import csv
with open('data.csv', 'r') as file:
    data_list = list(csv.reader(file))
print("CSV as List:", data_list)
```

The same list conversion produces reusable rows for small datasets. For large files, iterate over the reader instead.

9. *Append a New Row.*

```
# Append a new line
```

```
import csv
with open('data.csv', 'a', newline='') as file:
    writer = csv.writer(file)


    writer.writerow(["3", "2025-01-01", "08:00:00", "24.0",
"1.4", "9.8"])
```

Opening the file in append mode and calling writerow() adds one complete row without altering existing records.

10. *Modify a Specific Row.*

```
# Modify a specific row
import csv
row_index = 1
new_row = ["1", "2025-01-01", "08:00:00", "26.0", "1.6",
"11.0"]
with open('data.csv', 'r', newline='') as file:
    rows = list(csv.reader(file))
if 0 <= row_index < len(rows):
    rows[row_index] = new_row
    with open('data.csv', 'w', newline='') as file:
csv.writer(file).writerows(rows)
```

The rows are loaded, the entry at row_index is replaced when the zero-based index is valid, and the complete dataset is written back.

11. *Insert a New Row at a Specific Position.*

```
# Insert a new row at the desired position
import csv
row_index = 2
new_row = ["2", "2025-01-01", "08:00:00", "23.5", "1.3",
"9.4"]
with open('data.csv', 'r', newline='') as file:
    rows = list(csv.reader(file))
if 0 <= row_index <= len(rows):
    rows.insert(row_index, new_row)
    with open('data.csv', 'w', newline='') as file:
        csv.writer(file).writerows(rows)
    print("File has been updated successfully!")
else:
    print("Row position is out of range.")
```

list.insert() adds a complete row at the selected zero-based position. The

updated rows are then written back to the file.

12. *Delete a Specific Row.*

```
# Delete a specific row
import csv
line_number_to_delete = 3
with open('data.csv', 'r', newline='') as file:
    rows = list(csv.reader(file))
if 1 <= line_number_to_delete <= len(rows):
    del rows[line_number_to_delete - 1]
    with open('data.csv', 'w', newline='') as file:
        csv.writer(file).writerows(rows)
```

The code reads the rows, removes the requested one-based line number when valid, and writes the remaining rows back.

13. *Write to a Specific Cell.*

```
# Write to a specific cell
import csv
row_idx = 2  # Third row (0-indexed)
col_idx = 3  # Fourth column (0-indexed)
new_value = "29.0"

# Read, modify, write
with open('data.csv', 'r', newline='') as file:
    rows = list(csv.reader(file))

    if 0 <= row_idx < len(rows) and 0 <= col_idx <
len(rows[row_idx]):
        rows[row_idx][col_idx] = new_value

        with open('data.csv', 'w', newline='') as outfile:
            csv.writer(outfile).writerows(rows)
        print(f"Updated cell ({row_idx+1},{col_idx+1}) to:
{new_value}")
    else:
        print("Cell position is out of range.")
```

After validating the row and column indexes, the selected cell is updated, and the rows are written back to the file.

Creating and Editing Word and Excel Documents

External libraries are required for .docx and .xlsx files. Legacy .xls workbooks use different libraries; the examples here focus on python-docx and openpyxl.

```
pip install python-docx
```

Table 8-12 summarizes common .docx operations, which are demonstrated in detail in the following code examples.

Table 8-12 A summary of commands used for common .docx file operations

1. *Reading a .docx File*
   The code loads an existing Word document and prints each paragraph.

```
from docx import Document
file_path = 'document.docx'
doc = Document(file_path)
for paragraph in doc.paragraphs:
    print(paragraph.text)
```

2. *Writing a .docx File*
   The next code creates a new .docx file or overwrites an existing file with the same name.

   A Document object is created in memory, a title and two paragraphs are added, and save() writes the result to new_document.docx.

```
from docx import Document

# 1. Create a new document object in memory
doc = Document()

# 2. Add content to the document object
doc.add_heading('Document Title', level=0)
doc.add_paragraph('This is the first paragraph.')
doc.add_paragraph('This is another paragraph.')

# 3. Save the document to a file. This is when the file is
actually created.
doc.save('new_document.docx')

print("Document 'new_document.docx' created successfully.")
```

In the add_heading method, the level parameter (level=0) determines the heading style. Here's what the different levels typically represent:

- level=0: Title (the main title of the document)
- level=1: Heading 1 (main headings)
- level=2: Heading 2 (subheadings)
- level=3 *and Above*: Further subheadings (Heading 3, Heading 4, etc.)

3. *Appending to a .docx File*
To append content, load the existing document, add a paragraph, and save it to the same path.

```
from docx import Document
file_path = 'document.docx'
doc = Document(file_path)
doc.add_paragraph('This is additional content added to the
document.')
doc.save(file_path)
print("Content appended to document.docx")
```

4. *Getting Font Information*
This example inspects the font properties applied directly to each run in a Word document.

```
from docx import Document  # Import the library to work
with .docx files

# Open the Word document in binary mode
with open('document.docx', 'rb') as file:
    doc = Document(file)  # Load the document into memory

    # Loop through each paragraph in the document
    for i, paragraph in enumerate(doc.paragraphs):
        # Loop through each "run" (a portion of text with
the same formatting)
        for run in paragraph.runs:
            font = run.font  # Get the font properties of
the run

            # Display paragraph index and text
            print(f"Paragraph {i+1}: {run.text}")
            # Display font information
            print(f"  - Font: {font.name}")  # Font family
(e.g., Arial, Times New Roman)
```

```
            print(f"  - Size: {font.size.pt if font.size
else 'Default'}")  # Font size in points (pt)
            print(f"  - Bold: {font.bold}")  # True if
bold, False otherwise
            print(f"  - Italic: {font.italic}")  # True if
italic, False otherwise
            print(f"  - Underline: {font.underline}")  #
True if underlined, False otherwise
```

5.

Translating Text

The following example translates paragraph text into Spanish and writes a new document.

```
# Install: pip install googletrans==4.0.2 python-docx
import asyncio
from docx import Document
from googletrans import Translator

async def translate_document():
    source_doc = Document("document.docx")
    translated_doc = Document()

    async with Translator() as translator:
        for paragraph in source_doc.paragraphs:
            text = paragraph.text.strip()
            if text:
                result = await translator.translate(text,
dest="es")
                translated_doc.add_paragraph(result.text)
            else:
                translated_doc.add_paragraph("")

    translated_doc.save("translated.docx")
    print("Translation complete: translated.docx")

if __name__ == "__main__":
    asyncio.run(translate_document())
```

This version follows the asynchronous API used by googletrans 4.0.2. Empty paragraphs are preserved, and translated.docx is saved after all translations finish. Because googletrans is an unofficial service wrapper, production applications should account for possible service changes or interruptions.

6. *Reading a Specific Paragraph from a .docx File*

python-docx exposes paragraphs rather than visual screen lines. This example reads the third paragraph.

```
# Read a specific paragraph from a .docx file
from docx import Document

# Specify the file and paragraph number
file_path = 'document.docx'
paragraph_number = 3

doc = Document(file_path)
if 0 < paragraph_number <= len(doc.paragraphs):
    text = doc.paragraphs[paragraph_number - 1].text
    print(f"Paragraph {paragraph_number}: {text}")
else:
    print(f"Paragraph {paragraph_number} does not exist")
```

The requested paragraph number is converted to a zero-based list index by subtracting one.

If the paragraph number is outside the valid range, the program prints an explanatory message instead of indexing the list.

7. *Replacing a Specific Paragraph in a .docx File*

The example replaces the contents of a selected paragraph and saves the document.

A paragraph can span several visual lines, so the paragraph number—not the displayed line number—is the relevant index.

Calling clear() removes the paragraph's existing runs before add_run() inserts the replacement text.

```
# Replace a specific paragraph
from docx import Document

file_path = 'document.docx'
paragraph_number = 3
new_text = "Replacement paragraph text."

doc = Document(file_path)

if 0 < paragraph_number <= len(doc.paragraphs):
    paragraph = doc.paragraphs[paragraph_number - 1]
    paragraph.clear()
    paragraph.add_run(new_text)
    doc.save(file_path)
```

```
else:
    print(f"Paragraph {paragraph_number} does not exist")
```

8. *Appending Text to a Specific Paragraph*
This example adds text to the end of an existing paragraph without replacing its current runs.

```
# Append text to a specific paragraph
from docx import Document
file_path = 'document.docx'
paragraph_number = 3
additional_text = " - Additional text."

doc = Document(file_path)
if 0 < paragraph_number <= len(doc.paragraphs):
    doc.paragraphs[paragraph_number -
1].add_run(additional_text)
    doc.save(file_path)
    (f"Text appended to paragraph {paragraph_number}")
else:
    print(f"Paragraph {paragraph_number} does not exist")
```

After validating the paragraph number, add_run() appends the new text and save() writes the updated document. The existing paragraph formatting is otherwise retained.

9. *Counting Paragraphs, Tables, and Images*

```
from docx import Document

with open('document.docx', 'rb') as file:  # Open file in
binary mode
    doc = Document(file)  # Load the document

    num_paragraphs = len(doc.paragraphs)
    num_tables = len(doc.tables)
    num_images = sum(1 for rel in doc.part.rels.values() if
"image" in rel.reltype)

    print(f"Paragraphs: {num_paragraphs}")
    print(f"Tables: {num_tables}")
    print(f"Images: {num_images}")
```

The example loads a Word document and prints basic counts for paragraphs, tables, and image relationships.

10. *Extracting All Paragraphs*

```
from docx import Document
doc = Document('document.docx')
for i, paragraph in enumerate(doc.paragraphs):
    print(f"Paragraph {i + 1}: {paragraph.text}")
```

enumerate() supplies the paragraph number while the loop prints each paragraph's text.

11. *Identifying Heading Paragraphs*

```
from docx import Document
doc = Document('document.docx')
print("Heading paragraphs:")
for paragraph in doc.paragraphs:
    if paragraph.style.name.startswith("Heading"):
        print(paragraph.text)
```

The loop prints paragraphs whose style names begin with Heading, such as Heading 1 or Heading 2.

Handling .xlsx Files

The .xlsx file format is commonly used by office software to store spreadsheet data. This format is an Office Open XML file format introduced in Microsoft Excel 2007 and is used in later versions. It is also supported by most modern spreadsheet software.

For new spreadsheets, it is recommended to use the .xlsx format to ensure compatibility and security. Additionally, .xls files are limited to 65,536 rows and 256 columns, whereas .xlsx files support up to 1,048,576 rows and 16,384 columns, making them more suitable for handling large datasets.

The openpyxl library is used for the .xlsx examples below.

```
pip install openpyxl
```

The first example uses a cross-platform path and handles a missing workbook explicitly. Subsequent examples assume that the workbook and any referenced sheet names already exist; adjust the names for your files.

```
from pathlib import Path
from openpyxl import load_workbook
file_path = Path.home() / "Python_codes" / "example.xlsx"
try:
```

```
    workbook = load_workbook(file_path)
except FileNotFoundError:
    print(f"Workbook not found: {file_path}")
else:
    sheet = workbook.active
    for row in sheet.iter_rows(values_only=True):
        print(row)
```

Table 8-13 summarizes the spreadsheet operations demonstrated in the remaining examples.

Table 8-13 A summary of commands used for common .xlsx file operations with the openpyxl library

1. *Read an .xlsx File.*
   The code below reads the contents of an existing .xlsx file and prints them.

```
from openpyxl import load_workbook
# Load the workbook
workbook = load_workbook('example.xlsx')

# Select the active worksheet
sheet = workbook.active

# Loop through each row and print the cell values
for row in sheet.iter_rows(values_only=True):
    print(row)
```

   The example loads the workbook, selects its active worksheet, and prints each row as a tuple of cell values.

   load_workbook() opens the specified file, workbook.active returns the active worksheet, and iter_rows() traverses its rows.

2. *Read All Sheets.*
   To read data from all sheets in the workbook, you can loop through the sheet names.

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('example.xlsx')

# Loop through all sheets
for sheet_name in workbook.sheetnames:
    sheet = workbook[sheet_name]  # Select the sheet
    print(f"\nSheet: {sheet_name}")
```

```
    print(f"\nSheet: {sheet_name}")
    # Read all rows in the sheet
    for row in sheet.iter_rows(values_only=True):
        print(row)
```

This code reads data from all sheets in the workbook. It loads the workbook and then loops through each sheet name, selecting the sheet and printing its contents.

The method sheetnames returns a list of all sheet names in the workbook. workbook[sheet_name] selects the specified sheet.

3. *Read a Specific Sheet.*
To read all the data from a specific sheet, you can iterate through the rows and columns.

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('example.xlsx')

# Select a specific sheet
sheet = workbook['Sheet1']  # Replace 'Sheet1' with the
name of your sheet

# Read all rows in the sheet
for row in sheet.iter_rows(values_only=True):  #
`values_only=True` returns cell values instead of cell
objects
    print(row)
```

workbook['Sheet1'] selects the named worksheet, and iter_rows() traverses its rows.

4. *Read a Specific Row.*
To read a specific row, index the worksheet with its one-based row number.

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('example.xlsx')

# Select a specific sheet
sheet = workbook['Sheet1']  # Replace 'Sheet1' with the
name of your sheet

# Read a specific row (e.g., row 2)
row_number = 2
row_values = [cell.value for cell in sheet[row_number]]
```

```
row_values = [cell.value for cell in sheet[row_number]]

print(f"Values in row {row_number}: {row_values}")
```

Here, a list comprehension extracts the values from the requested row.

The method sheet[row_number] returns a tuple of cells in the specified row, and the list comprehension extracts the cell values from the tuple.

5. *Read a Specific Column.*
To read a specific column, index the worksheet with its column letter.

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('example.xlsx')

# Select a specific sheet
sheet = workbook['Sheet1']  # Replace 'Sheet1' with the
name of your sheet

# Read a specific column (e.g., column B)
column_letter = 'B'
column_values = [cell.value for cell in
sheet[column_letter]]
print(f"Values in column {column_letter}: {column_values}")
```

Here, a list comprehension extracts the values from the requested column.

The method sheet[column_letter] returns a tuple of cells in the specified column, and the list comprehension extracts the cell values from the tuple.

6. *Read Non-empty Rows or Columns.*
To read only non-empty rows or columns, you can filter out rows or columns where all cells are None.

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('example.xlsx')

# Select a specific sheet
sheet = workbook['Sheet1']  # Replace 'Sheet1' with the
name of your sheet

# Read non-empty rows
print("Non-empty rows:")
for row in sheet.iter_rows(values_only=True):
    if any(cell is not None for cell in row):  # Check if
```

```
    if any(cell is not None for cell in row):  # Check if
any cell in the row is not empty

        print(row)

# Read non-empty columns
print("\nNon-empty columns:")
for col in sheet.iter_cols(values_only=True):
    if any(cell is not None for cell in col):  # Check if
any cell in the column is not empty
        print(col)
```

The two loops print only rows and columns that contain at least one non-empty cell.

iter_rows() and iter_cols() traverse the worksheet; any() tests whether the current row or column contains a value.

7. *Read a Specific Cell.*

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('example.xlsx')

# Select a specific sheet
sheet = workbook['Sheet1']  # Replace 'Sheet1' with the
name of your sheet

# Read a specific cell (e.g., cell A1)
cell_value = sheet['A1'].value
print(f"Value of cell A1: {cell_value}")

# Alternatively, use row and column indices (e.g., row 1,
column 1)
cell_value = sheet.cell(row=1, column=1).value
print(f"Value of cell at row 1, column 1: {cell_value}")
```

A cell can be accessed either by its coordinate or by numeric row and column indexes.

To read the value of a specific cell, you can use the cell's coordinates like A1, B2. Use sheet['A1'].value or sheet.cell(row=1, column=1).value to read a specific cell.

8. *Read a Range of Cells.*

To read a specific range of cells (e.g., A1:B10), you can use slicing.

```
from openpyxl import load_workbook
```

```
# Load the workbook
workbook = load_workbook('example.xlsx')

# Select a specific sheet
sheet = workbook['Sheet1']  # Replace 'Sheet1' with the
name of your sheet

# Read a range of cells (e.g., A1:B10)
for row in sheet['A1:B10']:  # Iterate through the range
    for cell in row:
        print(cell.value)
```

After loading Sheet1, the nested loop traverses the range A1:B10 and prints each cell's value.

9. *Write to an .xlsx File.*
The next code creates a new .xlsx file or overwrites an existing one with new content.

```
from openpyxl import Workbook

# Create a new workbook
workbook = Workbook()

# Select the active worksheet
sheet = workbook.active

# Add content to the worksheet
sheet['A1'] = 'New Excel Document'
sheet['A2'] = 'This is a new row in the document.'

# Save the workbook
workbook.save('new_document.xlsx')

print("Workbook written and saved as 'new_document.xlsx'")
```

The Workbook class creates a new workbook. The example writes values to A1 and A2, saves the file as new_document.xlsx, and prints a confirmation.

10. *Append to an .xlsx File.*
To append data without removing existing content, load the workbook, add a row, and save it again.

```
from openpyxl import load_workbook

# Load the existing workbook
workbook = load_workbook('data.xlsx')
```

```
workbook = load_workbook('data.xlsx')

# Select the active worksheet
sheet = workbook.active
# Append new data to the worksheet
sheet.append(['This is additional content added to the
document.'])

# Save the changes to the same workbook
workbook.save('data.xlsx')

print("Content appended and saved to 'data.xlsx'")
```

The example opens data.xlsx, appends one row to the active worksheet, and saves the changes to the same file.

11. *Get Information from an .xlsx File.*

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('data.xlsx')

# Get the number of sheets in the workbook
num_sheets = len(workbook.sheetnames)
print(f"Number of sheets: {num_sheets}")

# Loop through each sheet in the workbook
for sheet_name in workbook.sheetnames:
    sheet = workbook[sheet_name]  # Select the sheet

    # Get the number of rows and columns in the sheet
    num_rows = sheet.max_row  # Total number of rows
    num_columns = sheet.max_column  # Total number of
columns

    print(f"\nSheet: {sheet_name}")
    print(f"  - Number of rows: {num_rows}")
    print(f"  - Number of columns: {num_columns}")
```

This example reports the number of worksheets and, for each one, the maximum row and column indexes recorded by openpyxl.

It prints the worksheet name together with those dimensions.

12. *Count Non-empty Rows and Columns.*

If you want to count only non-empty rows and columns, you can use the following code:

code:

```
from openpyxl import load_workbook

# Load the workbook
workbook = load_workbook('data.xlsx')

# Select a specific sheet
sheet = workbook['Sheet1']

# Count non-empty rows
non_empty_rows = 0
for row in sheet.iter_rows(values_only=True):
    if any(cell is not None for cell in row):  # Check if
any cell in the row is not empty
        non_empty_rows += 1

# Count non-empty columns
non_empty_columns = 0
for col in sheet.iter_cols(values_only=True):
    if any(cell is not None for cell in col):  # Check if
any cell in the column is not empty
        non_empty_columns += 1

print(f"Non-empty rows: {non_empty_rows}")
print(f"Non-empty columns: {non_empty_columns}")
```

The counters increase only when a row or column contains at least one value. The final values therefore exclude rows and columns that are completely empty.

Filesystem Paths with pathlib

The pathlib module provides an object-oriented, cross-platform interface for working withfilesystem paths. It avoids manual separator handling and works with both relative and absolute paths.

pathlib is part of the Python standard library, so no installation is required; import Path as shown below.

```
from pathlib import Path
```

The main pathlib operations used in this chapter are summarized in the following table.

Table

1. *Creating a Path Object*
   Path represents a filesystem path without requiring platform-specific string manipulation.

```
from pathlib import Path
path = Path.home() / "documents" / "file.txt"
print(path)  # The separator is selected for the current
operating system
```

   Path.home() returns the current user's home directory, and the / operator appends the remaining components.

   The resulting object can be inspected, joined with other components, or used directly by many file APIs.

2. *Joining Paths*
   Use the / operator to join path components clearly and without hard-coded separators.

```
from pathlib import Path
base_path = Path.home()
new_path = base_path / "documents" / "file.txt"
print(new_path)
```

   Starting from the home directory, the code appends documents and file.txt to form a complete path.

   This is clearer and more portable than concatenating path strings manually.

3. *Getting the Parent Directory*
   The .parent attribute returns the directory that contains a path.

```
from pathlib import Path
# Get the parent directory of a path
path = Path("example_directory/example_file.txt")
print(path.parent)  # Output: example_directory
```

   For example_directory/example_file.txt, path.parent returns example_directory.

4. *Getting Path Components*
   The .name, .stem, and .suffix attributes return the filename, the name without its extension, and the extension, respectively.

```
from pathlib import Path

# Get file name, stem (name without extension), and suffix
(extension)
```

```
path = Path("example_directory/example_file.txt")
print(path.name)    # Output: example_file.txt
print(path.stem)    # Output: example_file
print(path.suffix) # Output: .txt
```

These attributes retrieve example_file.txt, example_file, and .txt without manual string splitting.

5. *Checking Whether a Path Exists*
The .exists() method reports whether a file or directory is present.

```
from pathlib import Path

# Check if a path exists
path = Path("example_directory/example_file.txt")
print(path.exists())  # Output: True or False
```

path.exists() returns True when the specified path exists and False otherwise. For operations that require a particular type, prefer is_file() or is_dir().

6. *Checking Whether a Path is a File or Directory*
Use .is_file() and .is_dir() to distinguish files from directories.

```
from pathlib import Path

# Check if a path is a file or directory
path = Path("example_directory/example_file.txt")
print(path.is_file())  # Output: True or False
print(path.is_dir())   # Output: True or False
```

The two methods return boolean values and help validate a path before a type-specific operation.

7. *Creating Directories*
The .mkdir() method creates directories.

```
from pathlib import Path

# Create a single directory
Path("new_directory").mkdir(exist_ok=True)

# Create nested directories
Path("parent_directory/child_directory").mkdir(parents=True,
exist_ok=True)
```

exist_ok=True avoids an error when a directory already exists. For nested paths, parents=True also creates any missing parent directories.

8. *Reading and Writing Files*
Path provides concise text-file methods for simple reads and writes.

```
from pathlib import Path

# Write to a file
path = Path("example.txt")
path.write_text("Hello, pathlib!", encoding="utf-8")

# Read from a file
content = path.read_text(encoding="utf-8")
print(content)  # Output: Hello, pathlib!
```

write_text() creates or replaces the file, and read_text() returns its contents as a string. Specifying UTF-8 makes the encoding independent of the platform default.

9. *Renaming and Moving Files*
The next examples verify that the source exists before renaming or moving it.

```
from pathlib import Path

# Rename a file
file_path = Path("example.txt")
new_name = "renamed_example.txt"

if file_path.exists():
    file_path.rename(new_name)
    print(f"File renamed to {new_name}")
else:
    print("File does not exist.")
```

After exists() confirms the source file, rename() changes its name. Otherwise, the program prints an explanatory message.

```
from pathlib import Path

# Move a file to a new directory
file_path = Path("example.txt")
new_directory = Path("new_directory")

# Create the new directory if it doesn't exist
new_directory.mkdir(parents=True, exist_ok=True)

# Define the new path for the file
new_path = new_directory / file_path.name
```

```
if file_path.exists():
    file_path.replace(new_path)
    print(f"File moved to {new_path}")
else:
    print("File does not exist.")
```

The destination directory is created if necessary, and replace() moves example.txt to new_directory using the original filename. If a file already exists at the destination, replace() replaces it.

10. *Deleting Files and Directories*
Use unlink() for files and rmdir() for empty directories.

```
from pathlib import Path

# Delete a file
file_path = Path("example.txt")
if file_path.is_file():
    file_path.unlink()

# Delete an empty directory
dir_path = Path("empty_directory")
if dir_path.is_dir():
    dir_path.rmdir()
```

The type checks prevent calling a file operation on the wrong kind of path. rmdir() only removes an empty directory and raises an error if it contains entries.

The following examples demonstrate common methods for retrieving file metadata, such as file size and modification time.

11. *Getting File Size*
path.stat().st_size returns a file size in bytes.

```
from pathlib import Path

# Get the size of a file in bytes
path = Path("example.txt")
if path.is_file():
    print(path.stat().st_size)  # Output: File size in bytes
```

After is_file() validates the path, stat().st_size prints the number of bytes in the file.

12. *Getting the Last Modified Time*

path.stat().st_mtime returns the last modification time as a timestamp; time.ctime() formats it for display.

```
from pathlib import Path
import time

# Get the last modified time of a file
path = Path("example.txt")
if path.is_file():
    last_modified_time = path.stat().st_mtime
    print(time.ctime(last_modified_time))  # Output: Human-
readable timestamp
```

The example validates the file, obtains its modification timestamp, and converts it to a readable local-time string.

13. *Finding Files with Glob*
The .glob() method searches one directory for names that match a pattern.

```
from pathlib import Path
path = Path.home() / "documents"
matching_files = list(path.glob("*.txt"))  # Finds all .txt
files in the directory
```

path.glob("*.txt") finds text files directly inside the documents directory and stores the matches in a list.

```
from pathlib import Path

# Find all .txt files in a directory
for txt_file in Path("example_directory").glob("*.txt"):
    print(txt_file)
```

The loop processes each path returned by Path("example_directory").glob("*.txt") without first creating a list.

14. *Recursive Glob*
The .rglob() method applies the pattern to a directory and all of its subdirectories.

```
from pathlib import Path
path = Path.home()
all_txt_files = list(path.rglob("*.txt"))  # Finds all .txt
files recursively
```

path.rglob("*.txt") returns text files found anywhere below the user's home directory.

```
from pathlib import Path

# Recursively find all .txt files in a directory and its
subdirectories
for txt_file in Path("example_directory").rglob("*.txt"):
    print(txt_file)
```

The loop prints each text file found below example_directory, including files in nested subdirectories.

15. *Getting an Absolute Path*

```
from pathlib import Path
path = Path("documents/file.txt")
absolute_path = path.resolve()  # Absolute, normalized path
```

resolve() converts the relative path into an absolute, normalized path. By default, it can also resolve symbolic links where supported.

16. *Getting a Relative Path*

The .relative_to() method expresses a path relative to a specified base when the path is contained within that base.

```
from pathlib import Path

# Get the relative path of a file with respect to a base
directory
path = Path.home() / "example_directory" /
"example_file.txt"
base = Path.home()
print(path.relative_to(base))  # Output:
example_directory/example_file.txt
```

relative_to(base) removes the shared home-directory prefix and returns example_directory/example_file.txt.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_9

9. Math

Carlos Márcio da Silva Freitas[1]
(1) São Paulo, Brazil

This chapter explains how Python performs mathematical computation, from basic arithmetic to more advanced tools used in engineering and science.

Figure 9-1 summarizes the main built-in modules covered in the chapter and the purpose of each one.

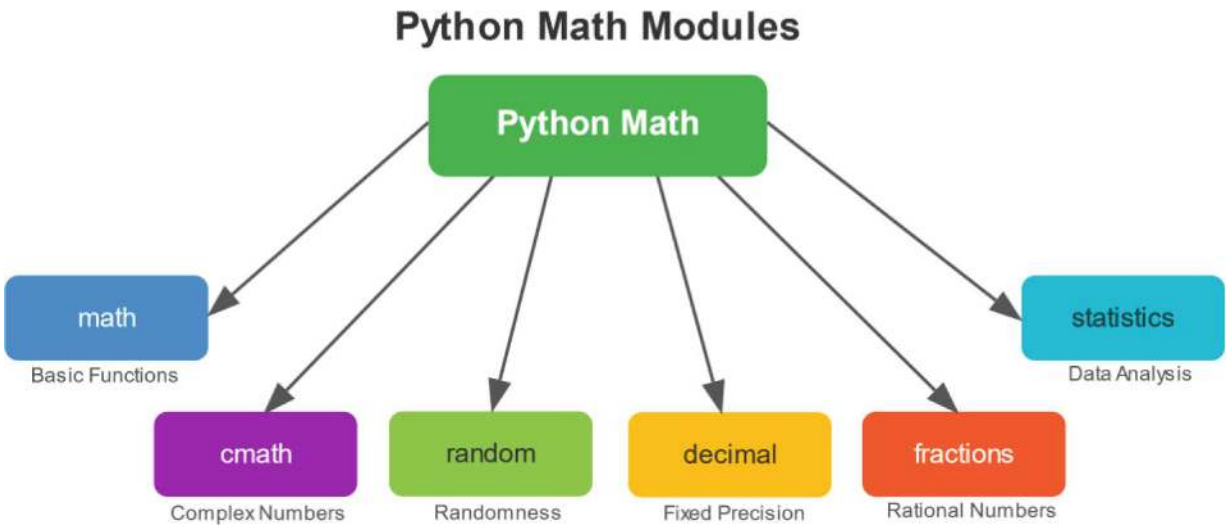


Figure 9-1 Mind map of Python's built-in math modules

The diagram serves as a roadmap for the mathematical tools introduced in the following sections.

Table 9-1 summarizes Python's native arithmetic operators and several built-in functions that do not require an external module.

Table 9-1 Python's native math operations

For built-in int and float operands, the single slash operator, /, performs true division and returns a float, even when the mathematical result is an integer. Other numeric types may define different return types.

Python operators and built-in functions handle many routine calculations, while standard library and third-party modules extend these capabilities.

The sections below focus first on modules included with Python; external packages installed with pip are discussed elsewhere.

Math Built-In Modules

A large number of common operations are available in the standard Python library. These operations can be useful for simple calculations or used to form more complex functions.

Table 9-2 contains the main built-in modules for math operation purposes.

Table 9-2 Main built-in modules for mathematical operations

Math

The built-in Python module math provides a wide range of capabilities, including number-theoretic functions, floating-point arithmetic and manipulation, power/exponential/logarithmic functions, summation and product functions, angular conversion, trigonometric and hyperbolic functions, special functions, and constants (pi, e, tau, infinity, Not a Number (NaN)).

Table 9-3 Main functions of the math module

math.isclose(a, b) compares floating-point values using relative and absolute tolerances. It is preferable to == when calculations such as 0.1 + 0.2 introduce a small binary-representation error.

Import math before using its functions. The complete API is available in the official Python documentation: https://docs.python.org/3/library/math.html.

The examples below cover combinatorics, rounding, floating-point classification, powers and logarithms, trigonometry, coordinate conversion, vector distance, and special functions. Together they illustrate the range of routine scientific calculations supported by math:

```
import math

# Combination formula.
comb_result = math.comb(5, 3)
print(comb_result)  # Output: 10

# Factorial of a non-negative integer.
factorial_result = math.factorial(5)
print(factorial_result)  # Output: 120

# Least common multiple.
lcm_result = math.lcm(4, 5, 6)
print(lcm_result)  # Output: 60

# Permutation formula
perm_result = math.perm(5, 3)
print(perm_result)  # Output: 60
```

Floating-Point Arithmetic

```
# Smallest integer >= x.
ceil_result = math.ceil(9.2)
print(ceil_result)  # Output: 10

# Absolute value of x.
fabs_result = math.fabs(-101.5)
```

```
print(fabs_result)  # Output: 101.5

# Largest integer <= x.
floor_result = math.floor(9.8)
print(floor_result)  # Output: 9

# Remainder of x / y.
fmod_result_pos = math.fmod(10, 3)
print(fmod_result_pos)  # Output: 1.0

# Fractional and integer parts of x.
modf_result = math.modf(12.34)
print(modf_result)  # Output: (0.3399999999999986,
12.0)

# Removes the fractional part.
trunc_result_pos = math.trunc(12.7)
print(trunc_result_pos)  # Output: 12
```

Floating-Point Manipulation Functions

```
# Mantissa and exponent of x.
frexp_result = math.frexp(16.5)
print(frexp_result)  # Output: (0.515625, 5)

# Compare values that are mathematically equal but
stored slightly differently.
val_a, val_b = 0.1 + 0.2, 0.3
# Default tolerance
print(math.isclose(val_a, val_b))  # Output: True

# Relative tolerance of 1%
print(math.isclose(val_a, val_b, rel_tol=0.01))  #
Output: True

# Absolute tolerance of 1e-6
```

```
print(math.isclose(val_a, val_b, abs_tol=1e-6))  #
Output: True

# Check if x is finite.
isfinite_result_true = math.isfinite(123)
print(isfinite_result_true)  # Output: True
isfinite_result_false = math.isfinite(math.inf)
print(isfinite_result_false)  # Output: False

# Check if x is NaN (Not a number).
isnan_result_true = math.isnan(math.nan)
print(isnan_result_true)  # Output: True
isnan_result_false = math.isnan(123)
print(isnan_result_false)  # Output: False
```

Power, Exponential, and Logarithmic Functions

```
# Cube root of x.
cbrt_result = math.cbrt(27)
print(cbrt_result)  # Output: 3.0

# e raised to the power x.
exp_result = math.exp(2)
print(f"{exp_result:.4f}")  # Output: 7.3891

# 2 raised to the power x.
exp2_result = math.exp2(5)
print(exp2_result)  # Output: 32.0

# Logarithm of x.
log_result_e = math.log(10)
print(f"{log_result_e:.4f}")  # Output: 2.3026
log_result_10 = math.log(100, 10)
print(log_result_10)  # Output: 2.0

# Base-2 logarithm of x.
log2_result = math.log2(32)
```

```
print(log2_result)  # Output: 5.0

# Base-10 logarithm of x.
log10_result = math.log10(1000)
print(log10_result)  # Output: 3.0

# x raised to the power y.
pow_result = math.pow(3, 4)
print(pow_result)  # Output: 81.0

# Square root of x.
sqrt_result = math.sqrt(64)
print(sqrt_result)  # Output: 8.0
```

Summation and Product Functions

```
# Euclidean distance between two points.
p, q = (1, 2, 3), (4, 6, 8)
dist_result = math.dist(p, q)
print(f"{dist_result:.4f}")  # Output: 7.0711

# Sum of products from two iterables.
p_sp, q_sp = [1, 2, 3], [4, 5, 6]
sumprod_result = math.sumprod(p_sp, q_sp)
print(sumprod_result)  # Output: 32
```

Angular Conversion

```
# Convert radians to degrees.
degrees_result = math.degrees(math.pi)
print(degrees_result)  # Output: 180.0

# Convert degrees to radians.
radians_result = math.radians(180)
print(f"{radians_result:.4f}")  # Output: 3.1416
```

Trigonometric Functions

```
# Arc cosine/sine/tangent of x.
acos_result = math.acos(0.5)
print(f"{acos_result:.4f} radians")  # Output:
1.0472 radians

# Cosine/Sine/Tangent of x radians,
math.cos(x)/sin(x)/tan(x).
cos_result = math.cos(math.pi)
print(cos_result)  # Output: -1.0
```

Hyperbolic Functions

```
# Inverse hyperbolic of x,
math.acosh(x)/asinh(x)/atanh(x).
acosh_result = math.acosh(1)
print(acosh_result)  # Output: 0.0

# Hyperbolic of x, math.cosh(x)/sinh(x)/tanh(x).
cosh_result = math.cosh(1)
print(f"{cosh_result:.4f}")  # Output: 1.5431
```

Special Functions

```
# math.gamma(x) - Gamma function at x.
gamma_result = math.gamma(5)
print(gamma_result)  # Output: 24.0
```

Mathematical constants are also available in the math module and can be useful in the areas of mathematics and physics.

```
import math

print("\n--- Constants ---")
pi_const = math.pi
print(pi_const)  # Output: 3.141592653589793
e_const = math.e
print(e_const)  # Output: 2.718281828459045
```

```
tau_const = math.tau
print(tau_const)  # Output: 6.283185307179586
inf_const = math.inf
print(inf_const)  # Output: inf
nan_const = math.nan
print(nan_const)  # Output: nan
```

Cmath

The built-in Python module cmath offers a comprehensive set of functions for handling complex numbers, including complex-aware versions of mathematical functions, conversions to and from polar coordinates, power and logarithmic functions, trigonometric and hyperbolic functions, classification functions, and constants (pi, e, tau, infinity, Not a Number (NaN), infinity for complex numbers (infj)).

In complex analysis, functions such as sqrt(z) and acos(z) can have more than one mathematical value. For computation, cmath selects a principal value and defines a branch cut, a curve in the complex plane across which that selected value is discontinuous. On a cut, the sign of zero in the relevant component can determine which side is used.

Table 9-4 Main functions of the cmath module

Figure 9-2 represents the complex number z = a + bi in the complex plane, with a on the real axis and b on the imaginary axis.

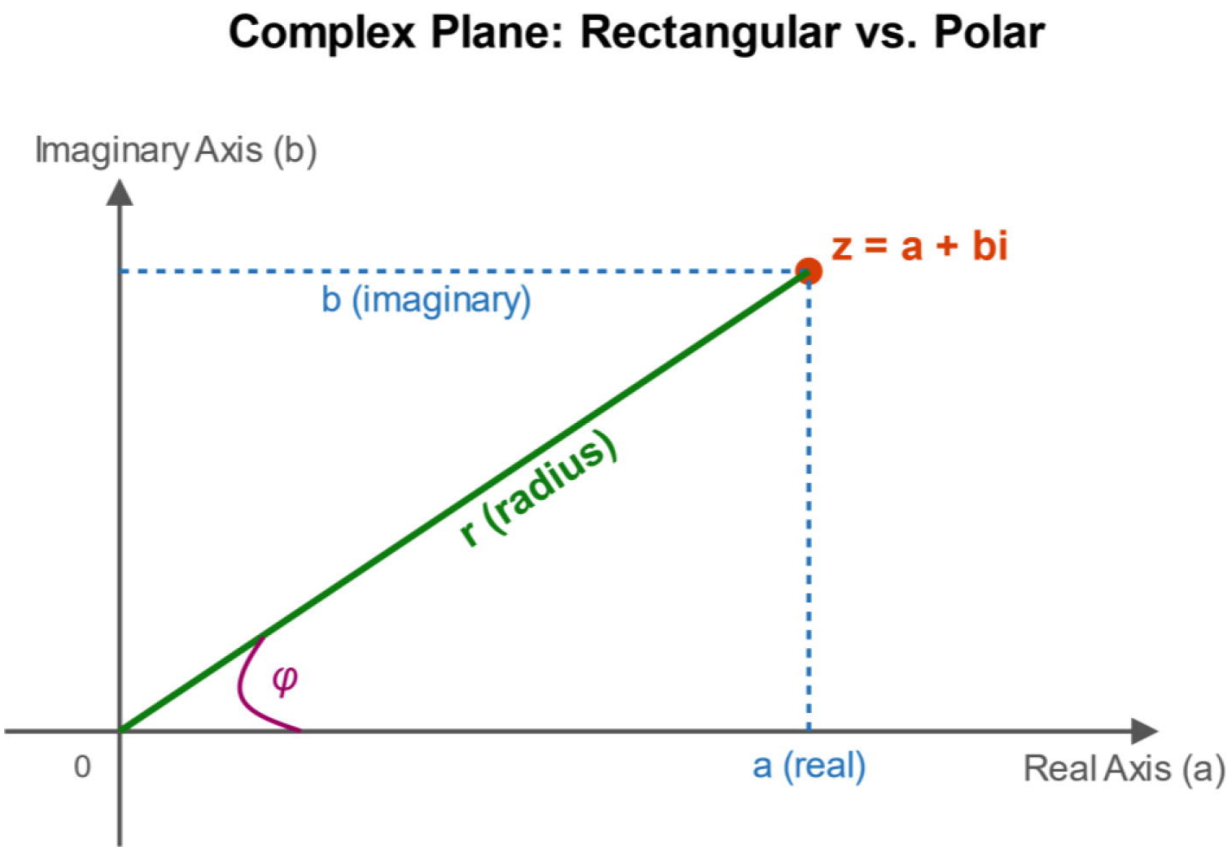


Figure 9-2 Representation of a complex number in the complex plane

The example below shows the branch-cut behavior of cmath.sqrt() with signed zeros:

```
import cmath

# Case 1: Positive zero imaginary part (above cut)
z_above = complex(-2.0, 0.0)
sqrt_above = cmath.sqrt(z_above)
print(f"sqrt({z_above})  : {sqrt_above}") #
Expected: positive imaginary result
# Result: sqrt((-2+0j))  : 1.4142135623730951j

# Case 2: Negative zero imaginary part (below cut)
z_below = complex(-2.0, -0.0)
sqrt_below = cmath.sqrt(z_below)
```

```
print(f"sqrt({z_below}) : {sqrt_below}") # Expected:
negative imaginary result
# Result: sqrt((-2-0j)) : (-1.4142135623730951j)
```

Rectangular coordinates (x, y) represent points using horizontal and vertical distances from the origin, while polar coordinates (r, θ) use distance from the origin (radius) and angle from the positive x-axis. Conversions between the two systems use formulas like $x = r \times \cos(\theta)$ and $r = \sqrt{(x^2 + y^2)}$. Both systems are useful in different applications, with rectangular coordinates suited for graphing and linear motion and polar coordinates suited for navigation, circular motion, and wave patterns.

Complex numbers are closely related to both rectangular and polar coordinates. In rectangular coordinates, a complex number z = a + bi is represented as a point (a, b) in the complex plane. In polar coordinates, z = r(cos(θ) + i sin(θ)) or z = re^(iθ), where r is the magnitude and θ is the angle. This connection enables powerful representations and operations with complex numbers, making them essential in many areas of engineering.

The Python cmath library provides the essential tools to convert between these two forms. The cmath.polar() function takes a complex number in rectangular form and returns its full polar coordinates (r, ϕ), while the more specific cmath.phase() function returns only the phase (ϕ). Conversely, the cmath.rect() function takes a modulus (r) and a phase (ϕ) and reconstructs the original complex number in its rectangular form (a + bj):

```
import cmath

# --- Conversions to and from polar coordinates ---
print("--- Conversions to and from polar coordinates
---")
# Create a complex number
z = 1 + 2j

# phase(z): Return the phase of z, in radians.
phase_result = cmath.phase(z)
```

```
print(f"Phase of {z}: {phase_result:.4f}
radians")  # Output: Phase of (1+2j): 1.1071 radians

# polar(z): Return the representation of z in polar
coordinates as a pair (r, phi).
# r is the modulus (absolute value) of z, and phi is
the phase of z.
polar_result = cmath.polar(z)
print(f"Polar coordinates of {z}: (r=
{polar_result[0]:.4f}, phi={polar_result[1]:.4f}
radians)")  # Output: Polar coordinates of (1+2j):
(r=2.2361, phi=1.1071 radians)

# rect(r, phi): Return the complex number z with
polar coordinates r and phi.
r, phi = 2, cmath.pi / 4  # Modulus 2, phase 45
degrees
rect_result = cmath.rect(r, phi)
print(f"Complex number from polar(r={r}, phi=
{phi:.4f}): {rect_result:.4f}")  # Output: Complex
number from polar(r=2, phi=0.7854): 1.4142+1.4142j
```

The next block covers complex exponentials, natural and base-10 logarithms, and the principal square root of a negative number:

```
import cmath

# --- Power and logarithmic functions ---
print("\n--- Power and logarithmic functions ---")
z = 1 + 1j

# exp(z): Return e raised to the power z.
exp_result = cmath.exp(z)
print(f"e^{z}: {exp_result:.4f}")  # Output:
e^(1+1j): 1.4687+2.2874j
```

```
# log(z[, base]): Return the logarithm of z to the
given base.
log_result_e = cmath.log(z)
print(f"Natural log of {z}: {log_result_e:.4f}")  #
Output: Natural log of (1+1j): 0.3466+0.7854j
log_result_10 = cmath.log(100 + 0j, 10)
print(f"Base-10 log of 100: {log_result_10:.4f}") #
Output: Base-10 log of 100: 2.0000+0.0000j

# log10(z): Return the base-10 logarithm of z.
log10_result = cmath.log10(1000 + 0j)
print(f"Base-10 log of 1000: {log10_result:.4f}")  #
Output: Base-10 log of 1000: 3.0000+0.0000j

# sqrt(z): Return the square root of z.
sqrt_result = cmath.sqrt(-1 + 0j)
print(f"Square root of -1: {sqrt_result:.4f}")  #
Output: Square root of -1: 0.0000+1.0000j
```

The trigonometric example applies direct and inverse functions to complex inputs. Unlike the real-valued math functions, these operations can return complex results:

```
import cmath

# --- Trigonometric functions ---
print("\n--- Trigonometric functions ---")
z = 0.5 + 0.5j

# acos(z): Return the arc cosine of z.
acos_result = cmath.acos(z)
print(f"Arc cosine of {z}: {acos_result:.4f}")  #
Output: Arc cosine of (0.5+0.5j): 1.1185-0.5306j

# asin(z): Return the arc sine of z.
asin_result = cmath.asin(z)
```

```
print(f"Arc sine of {z}: {asin_result:.4f}")  #
Output: Arc sine of (0.5+0.5j): 0.4523+0.5306j

# atan(z): Return the arc tangent of z.
atan_result = cmath.atan(z)
print(f"Arc tangent of {z}: {atan_result:.4f}")  #
Output: Arc tangent of (0.5+0.5j): 0.5536+0.4024j

# cos(z): Return the cosine of z.
cos_result = cmath.cos(cmath.pi + 0j)
print(f"Cosine of pi: {cos_result:.4f}") # Output:
Cosine of pi: -1.0000+0.0000j

# sin(z): Return the sine of z.
sin_result = cmath.sin(cmath.pi / 2)
print(f"Sine of pi/2: {sin_result:.4f}") # Output:
Sine of pi/2: 1.0000+0.0000j

# tan(z): Return the tangent of z.
tan_result = cmath.tan(cmath.pi / 4)
print(f"Tangent of pi/4: {tan_result:.4f}") #
Output: Tangent of pi/4: 1.0000-0.0000j
```

The next block applies the direct and inverse hyperbolic functions to complex inputs and formats each result to four decimal places:

```
import cmath

# --- Hyperbolic functions ---
print("\n--- Hyperbolic functions ---")
z = 1 + 0j

# acosh(z): Return the inverse hyperbolic cosine of
z.
acosh_result = cmath.acosh(z)
print(f"Inverse hyperbolic cosine of {z}:
{acosh_result:.4f}") # Output: Inverse hyperbolic
```

```
cosine of (1+0j): 0.0000+0.0000j

# asinh(z): Return the inverse hyperbolic sine of z.
asinh_result = cmath.asinh(z)
print(f"Inverse hyperbolic sine of {z}:
{asinh_result:.4f}") # Output: Inverse hyperbolic
sine of (1+0j): 0.8814+0.0000j

# atanh(z): Return the inverse hyperbolic tangent of
z.
atanh_result = cmath.atanh(0.5 + 0j)
print(f"Inverse hyperbolic tangent of 0.5:
{atanh_result:.4f}") # Output: Inverse hyperbolic
tangent of 0.5: 0.5493+0.0000j

# cosh(z): Return the hyperbolic cosine of z.
cosh_result = cmath.cosh(z)
print(f"Hyperbolic cosine of {z}:
{cosh_result:.4f}") # Output: Hyperbolic cosine of
(1+0j): 1.5431+0.0000j

# sinh(z): Return the hyperbolic sine of z.
sinh_result = cmath.sinh(z)
print(f"Hyperbolic sine of {z}: {sinh_result:.4f}")
# Output: Hyperbolic sine of (1+0j): 1.1752+0.0000j

# tanh(z): Return the hyperbolic tangent of z.
tanh_result = cmath.tanh(z)
print(f"Hyperbolic tangent of {z}:
{tanh_result:.4f}") # Output: Hyperbolic tangent of
(1+0j): 0.7616+0.0000j
```

The classification functions test complex values rather than transforming them. cmath.isfinite() requires both components to be finite, while cmath.isinf() and cmath.isnan() detect infinity or NaN in either

component. cmath.isclose() compares two complex numbers within configurable tolerances:

```
import cmath

# --- Classification functions ---
print("\n--- Classification functions ---")

# isfinite(z): Check if both real and imaginary
parts are finite.
print(f"Is 1+2j finite? {cmath.isfinite(1+2j)}") #
Output: Is 1+2j finite? True
print(f"Is cmath.inf+2j finite?
{cmath.isfinite(cmath.inf+2j)}") # Output: Is
cmath.inf+2j finite? False

# isinf(z): Check if either real or imaginary part
is an infinity.
print(f"Is 1+cmath.infj infinite?
{cmath.isinf(1+cmath.infj)}") # Output: Is
1+cmath.infj infinite? True
print(f"Is 1+2j infinite? {cmath.isinf(1+2j)}") #
Output: Is 1+2j infinite? False

# isnan(z): Check if either real or imaginary part
is a NaN.
print(f"Is cmath.nan+2j a NaN?
{cmath.isnan(cmath.nan+2j)}") # Output: Is
cmath.nan+2j a NaN? True
print(f"Is 1+2j a NaN? {cmath.isnan(1+2j)}") #
Output: Is 1+2j a NaN? False

# isclose(a, b): Check if two complex numbers are
close.
a = 1.0 + 1.0j
b = 1.0000001 + 1.0000001j
```

```
print(f"Are {a} and {b} close? {cmath.isclose(a,
b)}")  # Output: Are (1+1j) and
(1.0000001+1.0000001j) close? False
c = 1.1 + 1.1j
print(f"Are {a} and {c} close? {cmath.isclose(a,
c)}") # Output: Are (1+1j) and (1.1+1.1j) close?
False
```

Like math, cmath defines mathematical constants and special floating-point values. Two complex-specific shortcuts are

cmath.infj: A complex number with a real part of zero and an imaginary part of infinity (0.0 + inf*j). It is a convenient shortcut for representing a purely imaginary infinity.

cmath.nanj: A complex number with a real part of zero and an imaginary part of NaN (0.0 + nan*j). It is a shortcut for representing a purely imaginary undefined value.

```
import cmath

# --- Constants ---
print("\n--- Constants ---")
print(f"pi: {cmath.pi}")       # Output: pi:
3.141592653589793
print(f"e: {cmath.e}")         # Output: e:
2.718281828459045
print(f"tau: {cmath.tau}")     # Output: tau:
6.283185307179586
print(f"inf: {cmath.inf}")     # Output: inf: inf
print(f"infj: {cmath.infj}")   # Output: infj: infj
print(f"nan: {cmath.nan}")     # Output: nan: nan
print(f"nanj: {cmath.nanj}")   # Output: nanj: nanj
```

Decimal

Most decimal fractions cannot be represented exactly by a binary float. The stored approximation is usually harmless, but it can make direct

equality tests fail, as 0.1 + 0.2 illustrates:

```
# Example demonstrating floating point precision
issues
# Mathematically identical values
a = 0.1 + 0.2
b = 0.3

print("a =", a)  # Output: a = 0.30000000000000004
print("b =", b)  # Output: b = 0.3

# Equality check
print("a == b:", a == b)  # Output: a == b: False
```

The decimal module represents base-10 values and lets the programmer control precision, rounding, and exceptional conditions through a Context object. Values created exactly can retain significant trailing zeros, while arithmetic is rounded only when the active context requires it.

Create Decimal values from strings when the intended decimal literal must be preserved exactly. Passing a float instead preserves that float's approximate binary value.

The example compares construction from a string, an integer, and a float so that the difference is visible in the printed values:

```
from decimal import Decimal

# From a string (recommended)
a = Decimal('3.14')
print("From string:", a)
# Output: From string: 3.14

# From an integer
b = Decimal(10)
print("From integer:", b)
# Output: From integer: 10
```

```
# From a float (can be imprecise)
c = Decimal(3.14)
print("From float:", c)
# Output: From float:
3.140000000000000124344978758017532527446746826171875
```

The later examples use string inputs whenever an exact decimal literal is required.

Arithmetic Context

The arithmetic context defines the precision, rounding mode, exponent limits, flags, and traps used by decimal operations.

getcontext() returns the context for the current thread. Temporary changes can be isolated with localcontext().

Table 9-5 Attributes of the decimal arithmetic context

The most frequently used rules are precision (prec), which determines the total number of significant digits that a result can have, and rounding, which defines how to handle cases where a result has more digits than the precision allows.

Table 9-6 summarizes the most commonly used rounding modes in the decimal module.

Table 9-6 Common rounding modes in the decimal module

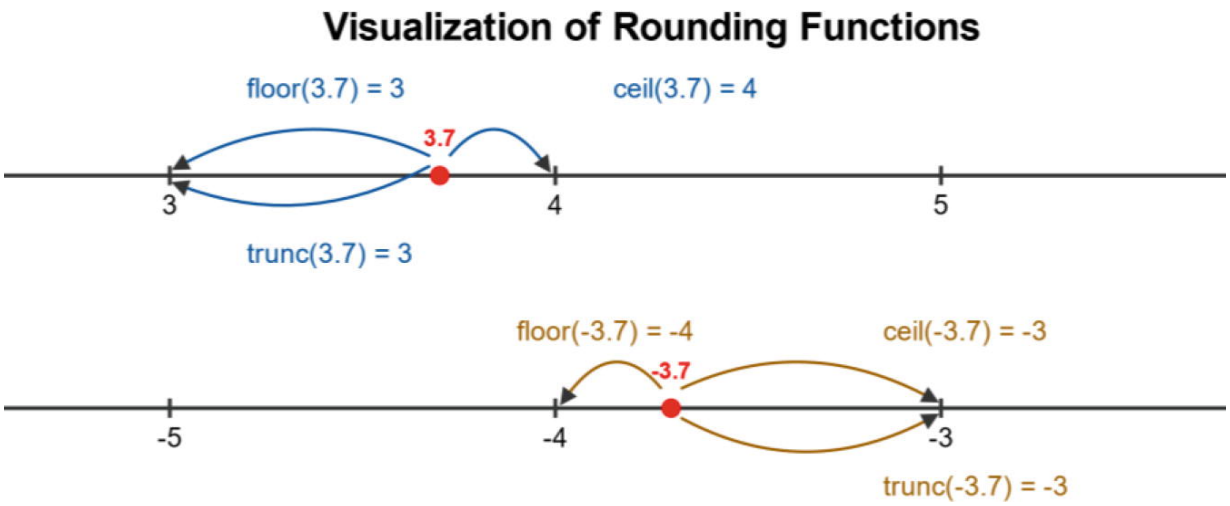


Figure 9-3 Illustration of rounding functions floor, ceil, and trunc

This example changes precision and rounding inside a temporary context, preventing those settings from leaking into later calculations:

```
from decimal import Decimal, localcontext, ROUND_UP

# Use a temporary context so these settings do not
affect later examples.
with localcontext() as ctx:
    ctx.prec = 6

    result = Decimal(1) / Decimal(7)
    print(result)  # Output: 0.142857

    ctx.rounding = ROUND_UP
    result2 = Decimal('3.14159') +
Decimal('2.71828')
    print(result2)  # Output: 5.85987
```

The module also provides preconfigured contexts for common situations. Two examples are summarized in Tables 9-7 and 9-8.

Table 9-7 Preconfigured decimal.BasicContext

Table 9-8 Preconfigured decimal.ExtendedContext

The next seven examples cover common uses of Decimal.

Example 1: Basic Precision and Arithmetic
Changing getcontext().prec alters the number of significant digits retained by subsequent decimal calculations. The same division is shown at precisions 6 and 30:

```
from decimal import Decimal, getcontext

# The default precision is 28 places. We can change
it globally.
# Set precision to 6 significant digits
getcontext().prec = 6

# Performing a division
result = Decimal(1) / Decimal(7)
print(f"1 / 7 with precision 6: {result}")
# Output: 1 / 7 with precision 6: 0.142857

# Now, let's increase the precision for more detail
getcontext().prec = 30
result_high_prec = Decimal(1) / Decimal(7)
print(f"1 / 7 with precision 30:
{result_high_prec}")
# Output: 1 / 7 with precision 30:
0.142857142857142857142857142857
```

Example 2: Avoiding Float Inaccuracy
The comparison below shows why Decimal values should be created from strings when exact base-10 arithmetic is required—the float calculation leaves a small residual, while the Decimal calculation returns zero:

```
from decimal import Decimal

# Using standard floats can lead to small
representation errors because computers
```

```
# cannot store some base-10 fractions exactly in
binary.
float_sum = 0.1 + 0.1 + 0.1 - 0.3
print(f"Using floats (0.1 + 0.1 + 0.1 - 0.3):
{float_sum}")
# Output: Using floats (0.1 + 0.1 + 0.1 - 0.3):
5.551115123125783e-17

# Decimals represent these numbers exactly as they
are written.
# We create Decimals from strings to avoid the
float's binary representation.
decimal_sum = Decimal('0.1') + Decimal('0.1') +
Decimal('0.1') - Decimal('0.3')
print(f"Using Decimals (from strings):
{decimal_sum}")
# Output: Using Decimals (from strings): 0.0
```

Example 3: Monetary Calculations

For monetary calculations, quantize() rounds a value to a fixed exponent. Here, a Decimal('0.01') template and ROUND_HALF_UP round the tax to cents before the total is calculated:

```
from decimal import Decimal, ROUND_HALF_UP

# quantize() is perfect for rounding to a fixed
number of decimal places, like cents.

price = Decimal('19.99')
tax_rate = Decimal('0.075') # 7.5% tax

# Calculate tax, which might have many decimal
places
tax = price * tax_rate
print(f"Calculated tax (unrounded): {tax}")
# Output: Calculated tax (unrounded): 1.49925
```

```
# Round the tax to the nearest cent (2 decimal
places)
# '0.01' defines the desired exponent/number of
places.
# ROUND_HALF_UP is a common rounding mode for money.
cents = Decimal('0.01')
rounded_tax = tax.quantize(cents,
rounding=ROUND_HALF_UP)
print(f"Rounded tax: {rounded_tax}")
# Output: Rounded tax: 1.50

total_price = price + rounded_tax
print(f"Total price: {total_price}")
# Output: Total price: 21.49
```

Example 4: Using Rounding Modes

Rounding to a whole number requires quantize() (or another explicit rounding operation); context precision alone does not remove the fractional part. The example compares ROUND_HALF_EVEN with ROUND_DOWN without changing the global context:

```
from decimal import Decimal, localcontext,
ROUND_HALF_EVEN, ROUND_DOWN

# ROUND_HALF_EVEN sends ties to the nearest even
integer.
with localcontext() as ctx:
    ctx.rounding = ROUND_HALF_EVEN
    num_even_tie = Decimal('2.5')
    num_odd_tie = Decimal('3.5')

    print(f"ROUND_HALF_EVEN: 2.5 ->
{num_even_tie.quantize(Decimal('1'))}")
    # Output: ROUND_HALF_EVEN: 2.5 -> 2
    print(f"ROUND_HALF_EVEN: 3.5 ->
{num_odd_tie.quantize(Decimal('1'))}")
```

```
    # Output: ROUND_HALF_EVEN: 3.5 -> 4

# ROUND_DOWN truncates toward zero.
with localcontext() as ctx:
    ctx.rounding = ROUND_DOWN
    num_to_round = Decimal('8.7')
    rounded_down =
num_to_round.quantize(Decimal('1'))
    print(f"ROUND_DOWN: 8.7 -> {rounded_down}")
    # Output: ROUND_DOWN: 8.7 -> 8
```

Example 5: Temporary Context

localcontext() confines a temporary precision change to its with block. When the block ends, the previous context is restored automatically:

```
from decimal import Decimal, getcontext,
localcontext

# You can temporarily change settings for a specific
block of code.
# The original context is restored automatically.
getcontext().prec = 6 # Set a global precision
print(f"Precision before localcontext:
{getcontext().prec}")
# Output: Precision before localcontext: 6

with localcontext() as ctx:
    ctx.prec = 50 # High precision for just this one
calculation
    # Example: A simplified calculation of Pi
    pi_approx = Decimal(355) / Decimal(113)
    print(f"Pi approximation (prec=50):
{pi_approx}")
# Output: Pi approximation (prec=50):
3.1415929203539823008849557522123893805309734513274
```

```
# The original precision is automatically restored
outside the 'with' block.
print(f"Precision after localcontext:
{getcontext().prec}")
# Output: Precision after localcontext: 6
```

Example 6: Mathematical Functions
Decimal also provides methods such as sqrt() and exp(); their results follow the active context precision:

```
from decimal import Decimal, getcontext

getcontext().prec = 28 # Reset to default precision
for this example.

square_root_of_2 = Decimal(2).sqrt()
print(f"Square root of 2: {square_root_of_2}")
# Output: Square root of 2:
1.414213562373095048801688724

# 'exp()' calculates e to the power of the number.
e_squared = Decimal(2).exp()
print(f"e squared (e**2): {e_squared}")
# Output: e squared (e**2):
7.389056098930650227230427461
```

Example 7: Special Values
Decimal can represent Infinity and NaN. Because 0 / 0 normally raises InvalidOperation, the example disables that trap only inside a local context and then continues to the NaN comparison:

```
from decimal import Decimal, localcontext,
InvalidOperation

# Decimal supports Infinity and NaN.
infinity = Decimal('Infinity')
nan = Decimal('NaN')
```

```
print(f"Infinity + 100: {infinity + 100}")
# Output: Infinity + 100: Infinity

# Disable the InvalidOperation trap only inside this
block.
with localcontext() as ctx:
    ctx.traps[InvalidOperation] = False
    undefined = Decimal(0) / Decimal(0)
    print(f"0 / 0 with the trap disabled:
{undefined}")
# Output: 0 / 0 with the trap disabled: NaN
print(f"Is NaN == NaN? {nan == nan}")
# Output: Is NaN == NaN? False
```

Fractions

The fractions module represents rational numbers exactly. Fraction objects are reduced automatically, can be constructed from several input types, and support ordinary arithmetic. A float such as 1.1 may produce a long ratio because the float is already approximate; limit_denominator() can recover a nearby ratio with a smaller denominator.

Here's its official Python documentation: https://docs.python.org/3/library/fractions.html.

Table 9-9 Main operations and methods of the fractions module

Fraction can be constructed from integers, strings, other Fraction objects, Decimal values, and floats. The examples below show how each source affects the result:

```
from fractions import Fraction
from decimal import Decimal

# --- Creating Fraction instances ---
# Fraction values are automatically reduced to their
simplest form.
```

```
F1 = Fraction(16, -10)
print(f"From integers (16, -10): {F1}")  # Output:
-8/5

# From an integer (the denominator defaults to 1)
F2 = Fraction(123)
print(f"From a single integer (123): {F2}")  #
Output: 123

# With no arguments, Fraction defaults to 0/1.
F3 = Fraction()
print(f"From no arguments: {F3}")  # Output: 0

# From another Fraction instance
F4 = Fraction(F1)
print(f"From another fraction ({F1}): {F4}")  #
Output: -8/5

# From a string
F5 = Fraction('3/7')
print(f"From a string ('3/7'): {F5}")  # Output: 3/7

# From a float (the exact stored float ratio may be
long)
F6 = Fraction(1.1)
print(f"From a float (1.1): {F6}")  # Output:
2476979795053773/2251799813685248

# From a Decimal object for exact decimal
representation
F7 = Fraction(Decimal('1.1'))
print(f"From a Decimal object (Decimal('1.1')):
{F7}")  # Output: 11/10

# From a float-like string
F8 = Fraction('-.125')
```

```
print(f"From a float-like string ('-.125'):
{F8}")  # Output: -1/8

# From a string with scientific notation
F9 = Fraction('7e-6')
print(f"From scientific notation string ('7e-6'):
{F9}")  # Output: 7/1000000

# Using underscores in strings
F10 = Fraction('1_000/2_000')
print(f"From string with underscores
('1_000/2_000'): {F10}")  # Output: 1/2
```

Use the numerator and denominator attributes to access a fraction in lowest terms:

```
from fractions import Fraction
# --- Accessing Numerator and Denominator ---
f_simple = Fraction(15, 25)
print(f"Original fraction: 15/25")
print(f"The fraction is: {f_simple}") #
Automatically simplified
print(f"Numerator: {f_simple.numerator}")
print(f"Denominator: {f_simple.denominator}")
```

Fraction objects support the usual arithmetic operators:

```
from fractions import Fraction
# --- Arithmetic Operations ---
a = Fraction(1, 2)
b = Fraction(1, 3)
print(f"a = {a}, b = {b}") # Output: a = 1/2, b =
1/3
print(f"Addition (a + b): {a + b}")  # Output: 5/6
print(f"Subtraction (a - b): {a - b}")  # Output:
1/6
```

```
print(f"Multiplication (a * b): {a * b}")  # Output:
1/6
print(f"Division (a / b): {a / b}")  # Output: 3/2
print(f"Power (a ** 2): {a ** 2}")  # Output: 1/4
print(f"Power (a ** -2): {a ** -2}")  # Output: 4
```

math.floor(), math.ceil(), and round() also accept Fraction objects. The example includes round-half-to-even cases and rounding to three decimal places:

```
import math
# --- Mathematical Functions (floor, ceil, round) --
-
f_num = Fraction(355, 113) # ~3.14159
print(f"The number is {f_num}") # Output: The number
is 355/113
print(f"math.floor({f_num}) =
{math.floor(f_num)}")  # Output: math.floor(355/113)
= 3
print(f"math.ceil({f_num}) = {math.ceil(f_num)}") #
Output: math.ceil(355/113) = 4
print(f"round({f_num}) = {round(f_num)}") # Output:
round(355/113) = 3
print(f"round({Fraction(5, 2)}) = {round(Fraction(5,
2))}")  # Output: round(5/2) = 2
print(f"round({Fraction(7, 2)}) = {round(Fraction(7,
2))}")  # Output: round(7/2) = 4
# Rounding to ndigits
f_round_target = Fraction(1, 10)
print(f"round({f_num}, 3) = {round(f_num, 3)}")  #
Output: round(355/113, 3) = 1571/500
```

Fraction supports ordinary and float-style formatting as well as several conversion and inspection methods:

- limit_denominator(): Finds the closest fraction with a limited denominator
- from_float() and from_decimal(): Create fractions from floating-point numbers and decimal objects
- as_integer_ratio(): Returns the numerator and denominator as a tuple
- is_integer(): Checks whether a fraction represents an integer value

These methods are useful when converting approximate values to rational forms or inspecting the exact ratio.

Statistics

The standard library statistics module focuses on foundational calculations. It complements, rather than replaces, broader packages such as NumPy and SciPy and is well suited to calculator-scale analysis.

Its functions cover measures of central location, spread, and relationships between paired datasets, including covariance, correlation, and simple linear regression.

Here's its official Python documentation: https://docs.python.org/3/library/statistics.html.

Table 9-10 Main functions of the statistics module

The first script calculates common means, medians, modes, and quantiles for numeric, categorical, weighted, rate, and grouped datasets:

```
import statistics
from collections import Counter

# --- Sample Datasets ---
data_numeric = [15, 8, 22, 15, 4, 31, 8, 19, 22, 12,
8]
data_strings = ['red', 'blue', 'green', 'blue',
'yellow', 'red', 'blue', 'purple']
data_rates = [80, 110]
```

```
grades, weights = [78, 95, 88, 91, 82], [0.15, 0.25,
0.20, 0.30, 0.10]
data_even = [10, 20, 30, 40]
grouped_data = list(Counter({25: 88, 35: 201, 45:
150, 55: 45, 65: 12}).elements())

# --- Averages and Measures of Central Location ---
print(f"Data: {data_numeric}")
print(f"Mean: {statistics.mean(data_numeric):.2f}")
print(f"Median: {statistics.median(data_numeric)}")
print(f"Geometric Mean:
{statistics.geometric_mean(data_numeric):.2f}")
print(f"Most Common Value(s):
{statistics.multimode(data_numeric)}")
print(f"Quartiles: {[round(q, 2) for q in
statistics.quantiles(data_numeric, n=4)]}\n")

print("--- Specialized Averages & Medians ---")
print(f"Weighted Mean (grades):
{statistics.fmean(grades, weights):.2f}")
print(f"Harmonic Mean (rates):
{statistics.harmonic_mean(data_rates):.2f}")
print(f"Low Median of {data_even}:
{statistics.median_low(data_even)}")
print(f"High Median of {data_even}:
{statistics.median_high(data_even)}")
print(f"Grouped Median (ages):
{statistics.median_grouped(grouped_data,
interval=10):.2f}\n")

print("--- Mode for Nominal Data ---")
print(f"Data: {data_strings}")
print(f"Most Common Value:
'{statistics.mode(data_strings)}'")
```

The output is:

```
Data: [15, 8, 22, 15, 4, 31, 8, 19, 22, 12, 8]
Mean: 14.91
Median: 15
Geometric Mean: 12.85
Most Common Value(s): [8]
Quartiles: [8.0, 15.0, 22.0]

--- Specialized Averages & Medians ---
Weighted Mean (grades): 88.55
Harmonic Mean (rates): 92.63
Low Median of [10, 20, 30, 40]: 20
High Median of [10, 20, 30, 40]: 30
Grouped Median (ages): 37.96

--- Mode for Nominal Data ---
Data: ['red', 'blue', 'green', 'blue', 'yellow',
'red', 'blue', 'purple']
Most Common Value: 'blue'
```

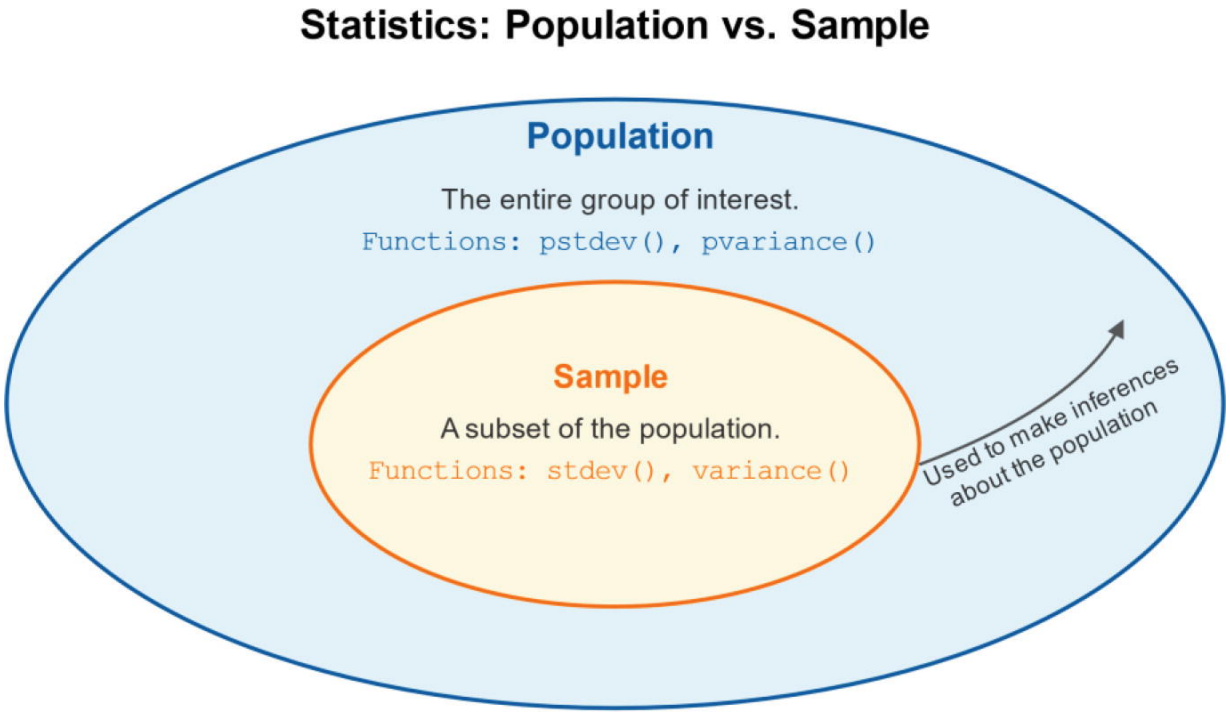


Figure 9-4 Difference between population and sample in statistics

The next script distinguishes population variance and standard deviation from their sample counterparts:

```
import statistics
# --- Sample Datasets ---
# A numeric list representing a sample of data.
data_sample = [15, 8, 22, 15, 4, 31, 8, 19, 22, 12,
8]
# A numeric list representing an entire population.
data_population = [1, 2, 3, 4, 5, 6]

# pvariance(): Calculates the variance of an entire
population.
pop_var = statistics.pvariance(data_population)
print(f"pvariance(): The population variance is
{pop_var:.2f}")

# pstdev(): Calculates the standard deviation of an
entire population.
pop_stdev = statistics.pstdev(data_population)
print(f"pstdev(): The population standard deviation
is {pop_stdev:.2f}\n")

# variance(): Calculates the variance of a sample.
# This is the one to use when your data is a sample
of a larger population.
samp_var = statistics.variance(data_sample)
print(f"variance(): The sample variance is
{samp_var:.2f}")

# stdev(): Calculates the standard deviation of a
sample.
samp_stdev = statistics.stdev(data_sample)
print(f"stdev(): The sample standard deviation is
{samp_stdev:.2f}")
```

The resulting dispersion measures are

```
pvariance(): The population variance is 2.92
pstdev(): The population standard deviation is 1.71

variance(): The sample variance is 64.69
stdev(): The sample standard deviation is 8.04
```

The choice between population and sample formulas matters because their denominators differ.

When the data comprise the entire population, pvariance() divides by n. For a sample used to estimate population variance, variance() applies Bessel's correction and divides by n – 1; using pvariance() on that sample would be biased downward on average.

The final statistics example examines two paired variables. It reports covariance, Pearson's correlation, and the slope and intercept of a simple regression line:

```
import statistics
# --- Sample Datasets ---
temperature_celsius = [20, 25, 30, 35, 40, 45, 50,
55, 60, 65]
sensor_voltage = [4.98, 4.97, 4.96, 4.95, 4.94,
4.93, 4.92, 4.91, 4.90, 4.89]

# covariance(): Measures how two variables change
together.
cov = statistics.covariance(temperature_celsius,
sensor_voltage)
print(f"covariance(): The sample covariance is
{cov:.2f}")

# correlation(): Measures the strength and direction
of a linear relationship.
corr = statistics.correlation(temperature_celsius,
sensor_voltage)
```

```
print(f"correlation(): The Pearson's correlation
coefficient is {corr:.2f}")

# linear_regression(): Calculates the slope and
intercept for a simple linear regression line.
regression =
statistics.linear_regression(temperature_celsius,
sensor_voltage)
slope = regression.slope
intercept = regression.intercept

print(f"linear_regression():")
print(f"  - Slope: {slope:.4f}") # Increased
precision for small slope
print(f"  - Intercept: {intercept:.2f}")
print(f"  - The regression line is: y = {slope:.4f}x
+ {intercept:.2f}")
```

The output is

```
covariance(): The sample covariance is -0.46
correlation(): The Pearson's correlation coefficient
is -1.00
linear_regression():
  - Slope: -0.0020
  - Intercept: 5.02
  - The regression line is: y = -0.0020x + 5.02
```

Random

The random module generates pseudo-random values for simulation, sampling, and similar non-cryptographic tasks.

Its main capabilities fall into three areas:

- *Generating Numbers*: Integers, floating-point values, and values from common probability distributions.

- *Working with Sequences*: Selecting elements with choice() or sample() and reordering mutable sequences with shuffle().
- *Reproducibility*: random.seed() fixes the initial generator state so the same sequence can be reproduced.

Because the generator is predictable, random must not be used for passwords, tokens, or other security-sensitive values; use secrets for those tasks.

This section focuses on the functions used most often. The full API appears in the official documentation: https://docs.python.org/3/library/random.html.

Table 9-11 Most frequently used functions of the random module

The example fixes the seed at 101, shuffles a list, samples elements, and generates integer and floating-point values.

random.seed(101) initializes the deterministic pseudo-random generator to a known state, so repeated runs produce the same sequence.

A different seed produces a different reproducible sequence. When no seed is supplied, Python uses operating system randomness when available and otherwise falls back to the system time.

```
import random
# 1. seed(): Remove or comment out random.seed() to
obtain a new sequence on each run.
random.seed(101)
# --- Setup Data ---
my_list = ['A', 'B', 'C', 'D', 'E', 'F']
print(f"Original list:    {my_list}")
# 2. shuffle(): Reorders the list in-place.
random.shuffle(my_list)
print(f"Shuffled list:    {my_list}")
# 3. choice(): Picks one random element.
print(f"Random choice:    {random.choice(my_list)}")
# 4. sample(): Picks 3 unique elements.
```

```
print(f"Random sample(k=3): {random.sample(my_list,
k=3)}")
# 5. randint(): A random integer from 1 to 100.
print(f"Random integer:   {random.randint(1, 100)}")
# 6. uniform(): A random float in a specific range.
print(f"Random uniform float: {random.uniform(20.5,
75.5):.2f}")
# 7. random(): A random float between 0.0 and 1.0.
print(f"Random float (0-1): {random.random():.2f}")
```

With random.seed(101), the output is reproducible:

```
Original list:    ['A', 'B', 'C', 'D', 'E', 'F']
Shuffled list:    ['D', 'A', 'F', 'C', 'B', 'E']
Random choice:    E
Random sample(k=3): ['B', 'A', 'E']
Random integer:   37
Random uniform float: 47.20
Random float (0-1): 0.78
```

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_10

10. Matplotlib

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Graphs are one of the primary tools for presenting information across various fields of engineering. They play a crucial role in visualizing data, identifying patterns, and communicating complex results in a clear and effective manner.

It is important to note that this chapter was created based on Matplotlib Release 3.10.3 and Python 3.13.5.

Table 10-1 lists various engineering disciplines and the corresponding graphical representations or diagrams they use.

Table 10-1 Common graphs used across various engineering disciplines

Python offers a wide range of libraries specifically designed for creating graphs and data visualizations. Each of these libraries comes with its own set of features, strengths, and ideal use cases, making them suitable for different types of applications and datasets.

Table 10-2 highlights some of the most popular free and open source Python libraries for graph generation. These libraries vary in complexity, customization capabilities, and suitability for interactive or static visualizations, allowing engineers and data scientists to choose the right tool for their specific needs.

Table 10-2 Popular Python libraries for data visualization

Among the plotting tools available for Python, Matplotlib is particularly useful when the engineer needs detailed control over the final figure. Axis limits, scales, line properties, annotations, legends, and figure dimensions can all be adjusted directly, which is important when preparing technical reports or comparing simulation and experimental results.

In practice, Matplotlib can be used in small Python scripts as well as in larger numerical workflows. For example, the same plotting approach may be applied to a motor speed response, a temperature profile, a stress–strain dataset, or the frequency response of an electrical system.

This chapter uses Matplotlib as a general plotting tool for engineering analysis. The emphasis is not only on producing a graph, but also on organizing the figure so that the physical meaning of the data can be understood by the reader.

To install Matplotlib use the following command:

```
pip install matplotlib
```

This command instructs pip to download Matplotlib and all of its required dependencies.

The functions selected in Table 10-3 represent plotting operations that frequently appear in engineering work. Rather than listing every plotting command available in Matplotlib, the functions are grouped here according to the type of technical data they help represent. This organization will also make it easier to relate each command to practical applications discussed later in the chapter.

Table 10-3 Matplotlib plotting tools frequently used in engineering applications

To illustrate some basic types of graphs useful in engineering, see the examples in Figures 10-1 to 10-4 plotted with Matplotlib.

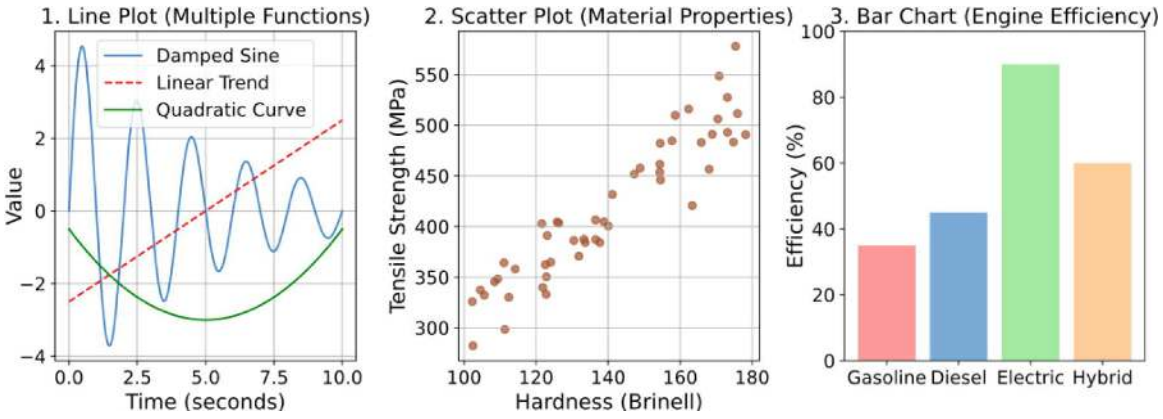


Figure 10-1 Examples of basic Matplotlib plots: line plot, scatter plot, and bar chart

1. *Line Plot*: This plot displays three different functions over time: a damped sine wave, a straight line, and a parabolic curve. It displays multiple functions over time, with the vertical axis representing "Value" and the horizontal axis representing "Time (seconds)."
2. *Scatter Plot*: This plot shows the relationship between two variables by placing a dot at each data point's (x, y) coordinate. Here, it visualizes the correlation between the hardness and tensile strength of a material. An engineer would use this to see if there's a trend, which in this case suggests that as hardness increases, so does strength.
3. *Bar Chart*: This chart uses rectangular bars to compare the efficiency of different engine types. The height of each bar is proportional to the value it represents.

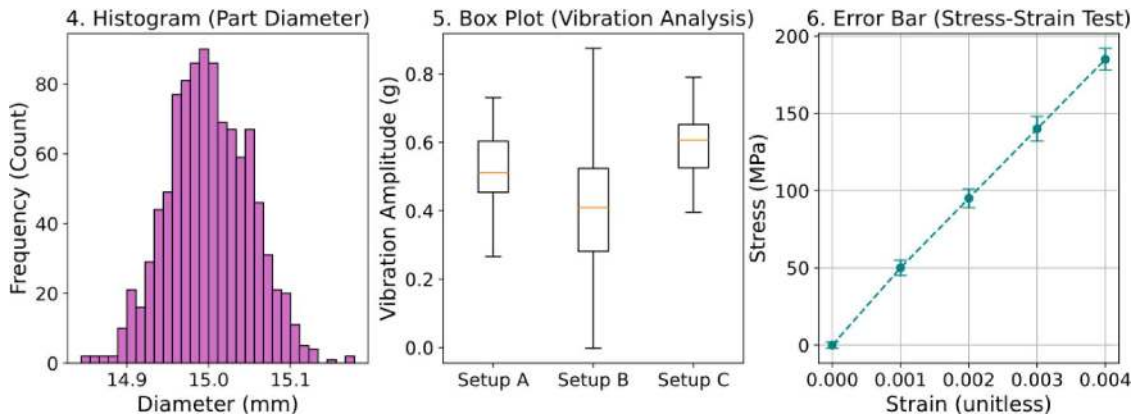


Figure 10-2 Examples of statistical plots in Matplotlib: histogram, box plot, and error bar plot

4. *Histogram*: This plot shows how often different sizes of manufactured parts occur by dividing the data into ranges or "bins" and counting how many parts fit into each range. It's useful for quality control engineers to see if the parts are being made correctly and meet the required standards.
5. *Box Plot*: This plot compares data from different groups. The "box" shows the middle 50% of the data, the line inside is the middle value (median), and the "whiskers" show how spread out the data is. In this case, it's used to compare vibration levels in different machine setups, helping engineers identify which setup is most stable or has the most variation.
6. *Error Bar*: This plot displays data points with an added visual indicator for the uncertainty or error in each measurement. In this stress–strain curve example, the vertical error bars show the confidence range for each stress measurement.

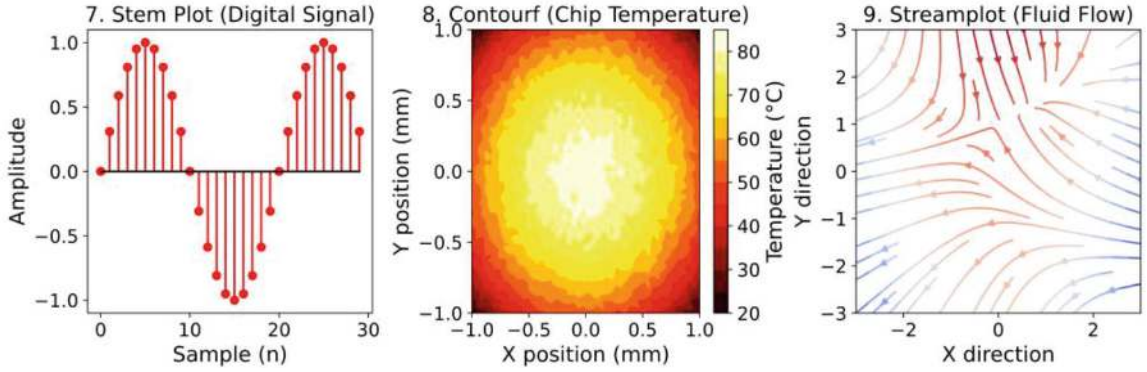


Figure 10-3 Examples of specialized Matplotlib plots: stem plot, contourf, and streamplot

7. *Stem Plot*: This plot is used to display discrete data. Each data point is shown as a marker with a vertical line (the "stem") connecting it to a baseline. This is a standard way to visualize discrete-time signals in digital signal processing (DSP), where a continuous signal has been sampled at regular intervals.
8. *Contourf*: This plot shows a 2D representation of a 3D surface, using colors to represent

different temperature ranges. It's useful for thermal engineers to identify hotspots or analyze temperature distribution on a surface, like a computer chip.

9. *Streamplot*: This plot visualizes a vector field using streamlines, which show the path that particles would take in the flow. It's essential in fluid dynamics for understanding flow patterns around an object, helping engineers design more aerodynamic or hydrodynamic shapes.

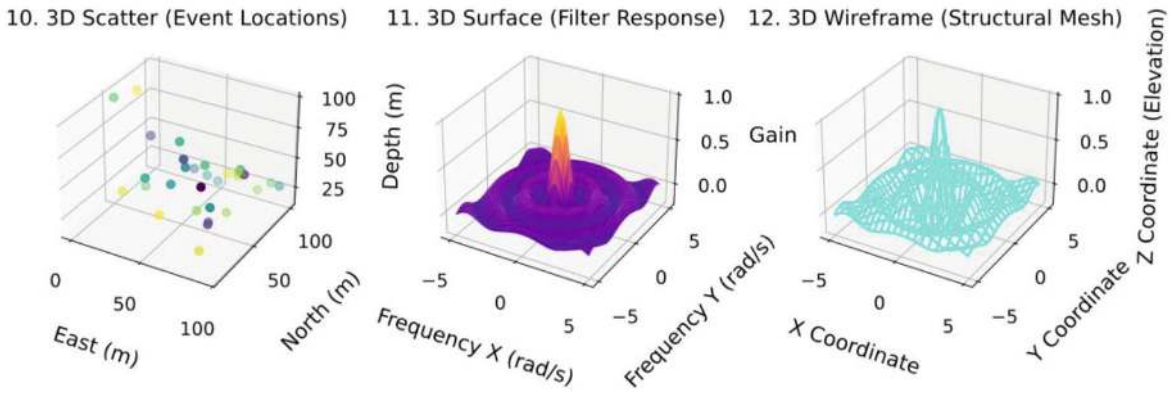


Figure 10-4 Examples of 3D plots in Matplotlib: 3D scatter, 3D surface, and 3D wireframe

10. *3D Scatter*: This plot shows data points in 3D space, used here to map event locations (like micro-earthquakes) with color indicating a fourth variable (energy). It's useful for analyzing spatial data in fields like geophysics and civil engineering.

11. *3D Surface*: This plot creates a 3D surface to visualize data points on a grid, showing the frequency response of a 2D filter. Electrical engineers use it to understand how a filter behaves across different input frequencies.

12. *3D Wireframe*: This plot displays a 3D surface as a grid of lines, like a mesh. It's used in mechanical and civil engineering for finite element analysis (FEA) to visualize a model's structure or represent terrain data.

Creating a Graph with Matplotlib

A Matplotlib program can essentially be written in two ways.

The first, the object-oriented (OO) approach, relies on explicit control of visual elements through the manual creation of figure and axis instances commonly instantiated via "fig, ax = plt.subplots()". This methodology is highly robust, facilitating the maintenance of complex code and the reuse of scripts.

The second alternative is the Pyplot style, which operates under a state-based paradigm. In this approach, instructions are sent directly to the global module (such as "plt.plot()" or "plt.title()"), with the library implicitly managing which plot is being modified. Although less flexible for advanced customization, Pyplot is ideal for quick analysis and prototyping.

Throughout this chapter, the object-oriented approach will be preferred. The reason is practical: as an engineering plot becomes more detailed, explicit access to its axes makes the program easier to extend and maintain.

For basic applications the code difference between the two styles is small. See the example below that compares the two styles to create the same graph.

Table

Matplotlib allows the creation of highly customized graphs, which are very useful in engineering applications. To simplify the display of information, the customization levels will be divided into two groups: The first will be the customization of the graph, such as line colors, grids, and texts in general. The second will analyze the scales available to represent the data in linear, logarithmic, and other ways.

The basic process of creating graphics can be divided into six steps as listed in Table 10-4.

Table 10-4 The six basic steps to create a plot with Matplotlib

Note In Jupyter notebooks, Matplotlib backends such as %matplotlib inline, %matplotlib notebook, or %matplotlib widget automatically call show() at the end of each cell. Therefore, plt.show() is usually optional in notebooks, although it remains useful in standalone scripts and examples.

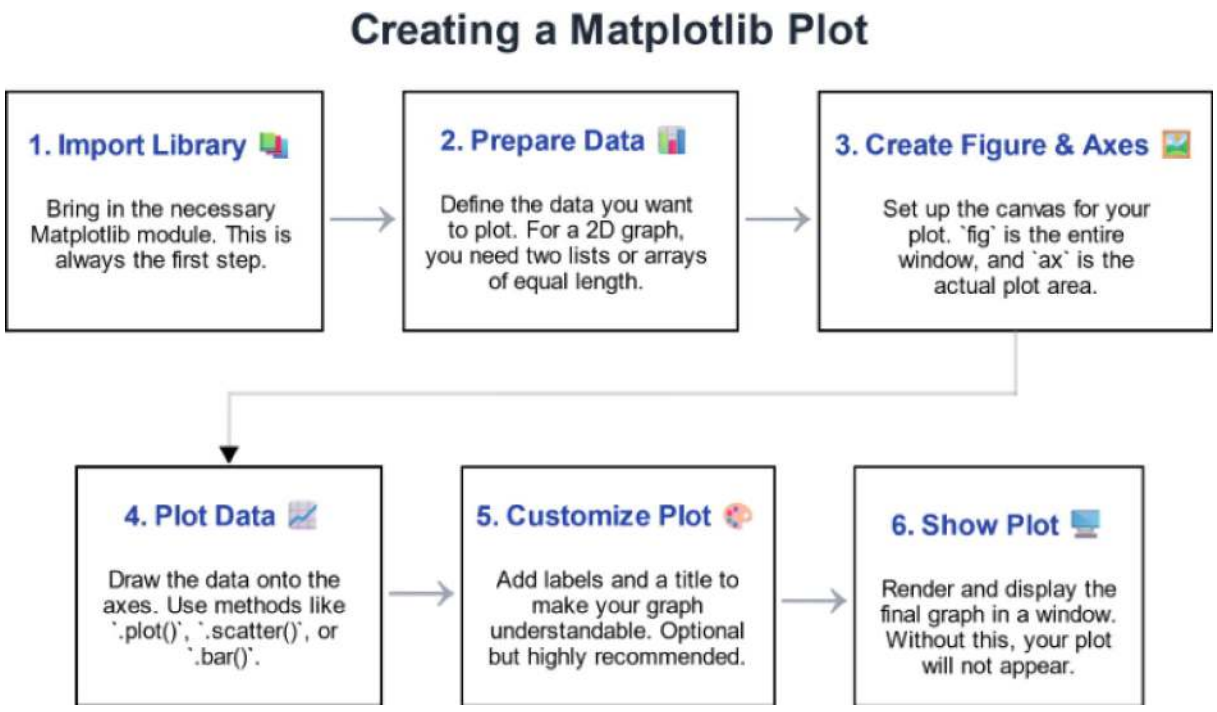


Figure 10-5 Flowchart illustrating the process of creating a Matplotlib plot

An Example of a Simple Graph

This example code uses the Matplotlib library to generate a simple line chart. It starts by importing the library and preparing two lists of data (`x_data and y_data`). Then, it creates a figure and axis object using plt.subplots(). The data is plotted on the axis using ax.plot() and customized with a title and labels for the x- and y-axes using ax.set_title(), ax.set_xlabel(), and ax.set_ylabel(). Finally, plt.show() renders and displays the plot in a window:

```
# Step 1: Import Library
import matplotlib.pyplot as plt

# Step 2: Prepare Data
x_data = [1, 2, 3, 4]
y_data = [10, 25, 18, 33]

# Step 3: Create Figure & Axes
fig, ax = plt.subplots()

# Step 4: Plot Data
ax.plot(x_data, y_data)

# Step 5: Customize Plot
ax.set_title("A Simple Plot")
ax.set_xlabel("X-Axis")
ax.set_ylabel("Y-Axis")

# Step 6: Show Plot
plt.show()
```

The result of the code is the graph shown in Figure 10-6.

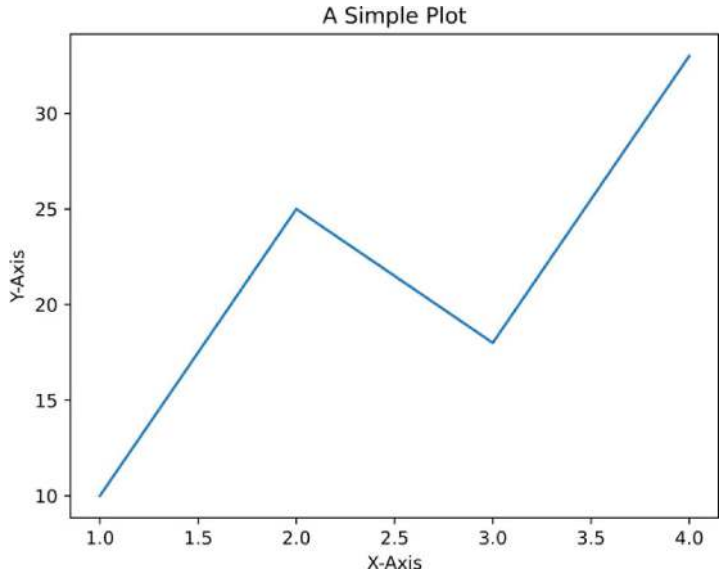


Figure 10-6 Result of the simple plot example code

The previous code used pre-populated datasets. In contrast, the next code generates a sine wave graph using Matplotlib and NumPy. It creates an array of x values from 0 to 2π with np.linspace() and calculates the corresponding y values using the np.sin() function. The data is then plotted on a graph using ax.plot(), with a title and labels for the x- and y-axes. The plt.show() command displays the graph, showing a single cycle of the sine wave with amplitude ranging from –1 to 1.

To enable the graph grid in Matplotlib, simply add the line ax.grid(True) before running the plt.show() command:

```
# Step 1: Import Library
import matplotlib.pyplot as plt
import numpy as np

# Step 2: Prepare Data
x_data = np.linspace(0, 2 * np.pi, 100)
y_data = np.sin(x_data)

# Step 3: Create Figure & Axes
fig, ax = plt.subplots()

# Step 4: Plot Data
ax.plot(x_data, y_data)

# Step 5: Customize Plot
ax.set_title("Sine Wave")
ax.set_xlabel("Angle (radians)")
ax.set_ylabel("Amplitude")
ax.grid(True) # Add a grid

# Step 6: Show Plot
plt.show()
```

The sine wave graph using Matplotlib and NumPy by calculating the sin() function is represented in Figure 10-7.

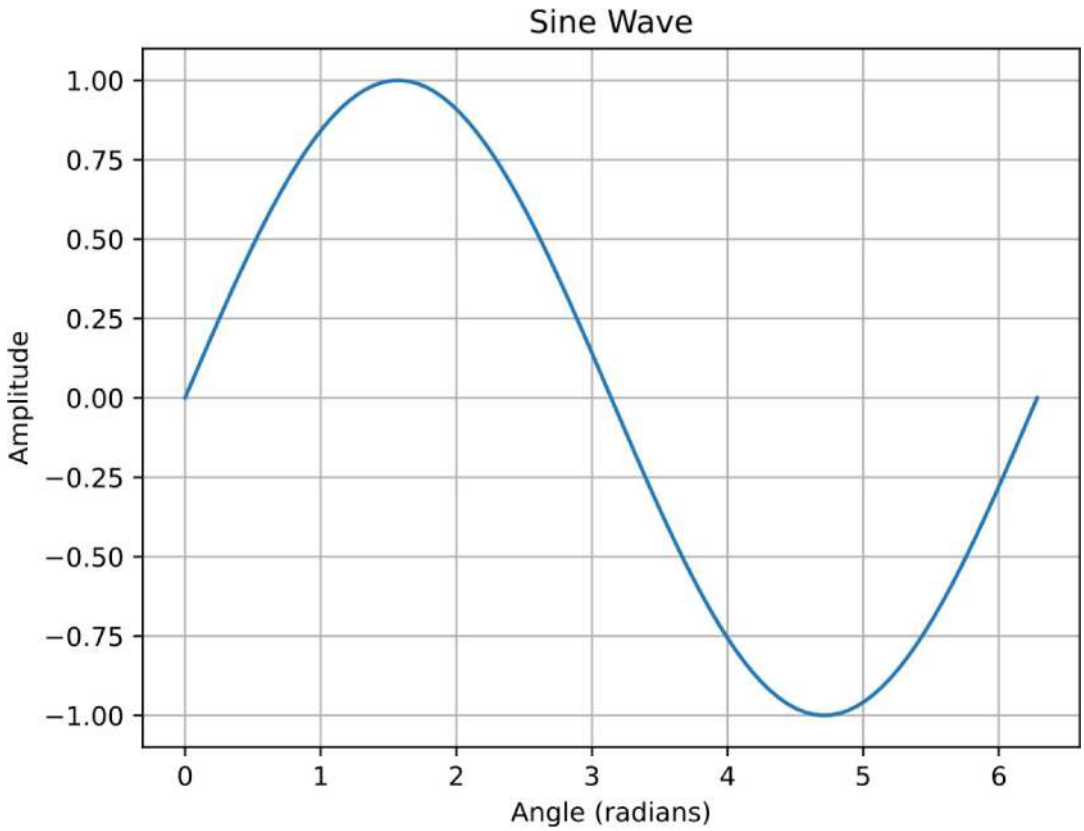


Figure 10-7 Sine wave plot generated with Matplotlib and NumPy

Graph Customizations

Table 10-5 Common customization options for Matplotlib chart elements

The sample code has been divided into five parts to explain each category of customizations available in the chart.

This first part of the code sets up the foundation for the plot by first importing the necessary libraries, Matplotlib for plotting and NumPy for numerical operations. It then prepares the data by using NumPy to create a smooth sequence of 100 x values from 0 to 2π and then calculates the corresponding y values for both a sine and a cosine wave, which will be plotted as two separate lines. It initializes the main plot window (fig) and the drawing area (ax) with a larger size of 10 × 8 inches to ensure the final chart is spacious and easy to read:

```
import matplotlib.pyplot as
plt                                                    1/5
import numpy as np

# --- 1. Prepare Data ---
# We'll create two datasets to demonstrate the legend effectively.
x_data = np.linspace(0, 2 * np.pi, 100)
y_data_sin = np.sin(x_data)
```

```
y_data_cos = np.cos(x_data)

# --- 2. Create Figure & Axes ---
fig, ax = plt.subplots(figsize=(10, 8)) # Make the figure larger
```

Lines and Spines

The code customizes the appearance of the data itself. Inside each ax.plot() call, several arguments are used: color sets the line color, linewidth makes it thicker, linestyle changes it from a solid line to dashed (--) or dotted (:), and marker with markersize adds small circles to the sine wave's data points. This makes each line visually distinct. Afterward, the code modifies the chart's frame by targeting the spines. The lines ax.spines['right'].set_visible(False) and ax.spines['top'].set_visible(False) remove the top and right borders of the plot, creating a cleaner, more open look:

```
# --- 3. Plot Data with Line and Marker Customizations --
-                    2/5
# Plot the sine wave with specific styling
ax.plot(x_data, y_data_sin,
        color='blue',                 # Set line color
        linewidth=2,                  # Set line width
        linestyle='--',               # Use a dashed line
        marker='o',                   # Add circle markers at data
points
        markersize=4,                 # Set the size of the markers
        label='Sine Wave')            # Add a label for the legend

# Plot the cosine wave with different styling
ax.plot(x_data, y_data_cos,
        color='red',
        linewidth=2,
        linestyle=':',                # Use a dotted line
        label='Cosine Wave')
```

Text and Labels

This section of the code is responsible for adding all the descriptive text that gives the chart context. First, ax.set_title() and fig.suptitle() are used to create two levels of titles—one for the specific subplot and a larger, main title for the entire figure. The code customizes these with different font sizes and weights. Then, ax.set_xlabel() and ax.set_ylabel() are used to label the axes, making it clear what the plotted values represent. Finally, ax.annotate() adds a specific note directly onto the plot area. It places the text "Peak of Sine" at one coordinate and draws an arrow from the text to the exact data point of the sine wave's peak, making it a powerful tool for highlighting specific features:

```
# --- 4. Apply All Customizations --
-                                        3/5
# -- Text & Labels --
```

```
ax.set_title("Fully Customized Matplotlib Chart", fontsize=18,
fontweight='bold')
ax.set_xlabel("Angle (radians)", fontsize=12)
ax.set_ylabel("Amplitude", fontsize=12)
fig.suptitle("Demonstration of Chart Customizations", fontsize=22)
# Add a main title for the whole figure

# Add an annotation with an arrow
ax.annotate('Peak of Sine',
            xy=(np.pi/2, 1),           # The point to annotate
            xytext=(np.pi/2, 1.3),     # The position of the text
            arrowprops=dict(facecolor='black', shrink=0.05))
```

Legends

To help differentiate between the two plotted lines, a legend is created. This is a two-step process. First, inside each ax.plot() function, the label argument is given a name (e.g., 'Sine Wave'). This registers the line with Matplotlib so it can be included in the legend. Second, the ax.legend() function is called to actually display the legend box. The code further customizes it by setting its location with loc, giving it a title, and making its frame semi-transparent with framealpha:

```
# -- Legend -
-                                                                    4/5
ax.legend(loc='upper right',  # Position the legend
          ncol=1,             # Number of columns
          frameon=True,       # Add a frame
          framealpha=0.8,     # Set frame transparency
          title='Functions')  # Add a title to the legend
```

Grid and Ticks

This part of the code focuses on the reference markers of the chart. The ax.grid() function is called with True to display a grid, and its appearance is customized with a light gray color and a dash–dot style to make it less intrusive. The x-axis ticks, which are normally numerical, are explicitly set to specific positions corresponding to key points on a sine wave using ax.set_xticks(). Then, ax.set_xticklabels() replaces the numerical labels with mathematical symbols (like 'π/2'), making the chart more readable for this specific data. Finally, ax.tick_params() is used to change the color and size of the labels on the y-axis, demonstrating how to style individual axes:

```
# -- Grid & Ticks -
-                                                                    5/5
ax.grid(True, color='gray', linestyle='-.', linewidth=0.5) #
Customize the grid
ax.set_xticks([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi]) # Set
specific tick locations
```

```
ax.set_xticklabels(['0', 'π/2', 'π', '3π/2', '2π']) # Set custom
labels for those ticks
ax.tick_params(axis='y', labelcolor='purple', labelsize=10) #
Customize y-axis tick labels

# -- Spines --
ax.spines['right'].set_visible(False) # Hide the right spine
ax.spines['top'].set_visible(False)   # Hide the top spine

# --- 5. Show Plot ---
# Adjust layout to prevent the suptitle from overlapping the plot
title
plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

The resulting graph from this example code that has been split into five parts is shown in Figure 10-8.

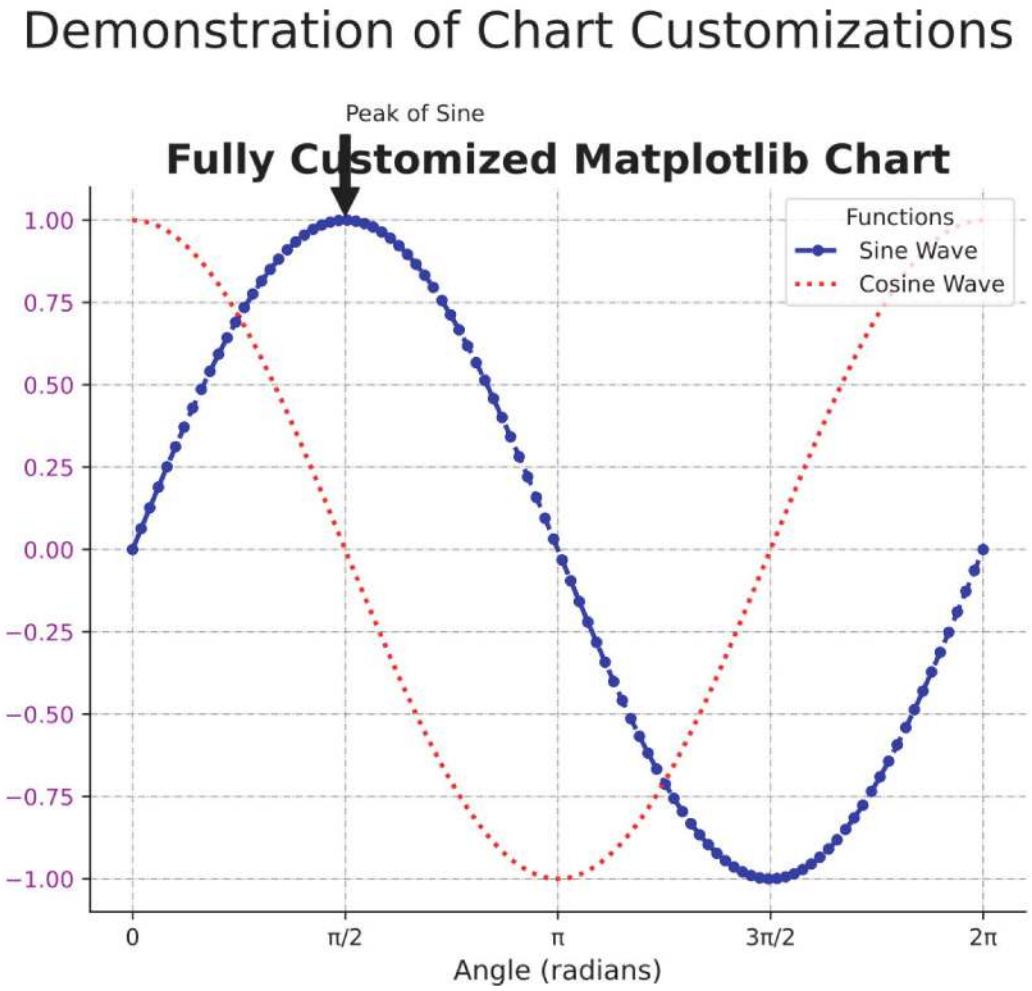


Figure 10-8 Example of a fully customized Matplotlib chart

Scales

Choosing an axis scale is part of the engineering interpretation of a graph. The same numerical dataset may emphasize very different features depending on how distances along the axis are transformed.

Matplotlib offers several scales to customize the data visualization in order to adapt the presentation of the graph according to the demands of the application.

- **Linear scale** is the default, suitable for standard data and general-purpose graphs.
- **Logarithmic scale** compresses a wide range of positive values, ideal for visualizing exponential growth and scientific data.
- **Symlog scale** handles positive, negative, and zero values with a logarithmic scale that becomes linear near the origin, useful for signal processing data.
- **Logit scale** is designed for data between 0 and 1, extending the scale near the ends, and is often used for probabilities and model accuracy metrics.
- **Inverse hyperbolic sine** (asinh) scale behaves linearly around zero and logarithmically for larger values, commonly used in astronomy and physics.
- **Custom scale** for highly specific engineering or academic research scenarios. This option allows the developer to define their own mathematical transformation to shape the chart axis.

Table 10-6 Axis scales interpreted from an engineering data perspective

Table 10-7 Function prototypes and key parameters for Matplotlib scales

base: For log and symlog, this sets the logarithm base (default is 10). For asinh, base is used by the tick locator to round tick locations; it does not change the asinh transformation.
nonpositive: For log and logit, this determines how to handle non-positive values ('clip' to a small positive number or 'mask' to hide them).
linthresh: For symlog, this sets the range around zero (–linthresh to +linthresh) where the scale is linear.
linscale: For symlog, this allows you to adjust the space that the linear section occupies relative to the logarithmic sections.
linear_width: For asinh, this defines the range around zero where the scale is approximately linear.
functions: For the function scale, you must provide a tuple containing two functions: the forward transformation and its inverse.

Linear Scale

The next example code visualizes a simple linear relationship using Matplotlib's default linear axis scale. It begins by importing the necessary libraries, matplotlib.pyplot for plotting and numpy for numerical operations. It then generates data for a straight line defined by the equation y=2x+5. A figure and a single subplot are created, and the linear data is plotted onto the axes. A title, axis labels, and a grid are also added to improve readability before displaying the final chart, which is best suited for showing data where changes are constant and additive.

The function prototype of the linear scale is

```
ax.set_xscale('linear')
```

Linear scaling does not require any special parameters.

```
import matplotlib.pyplot as plt
import numpy as np

# --- 1. Linear Scale (Default) ---
# Create a figure and a single subplot
fig, ax = plt.subplots(figsize=(6, 5))

# Generate data
x_linear = np.linspace(-10, 10, 100)
y_linear = 2 * x_linear + 5  # A simple straight line

# Plot the data
ax.plot(x_linear, y_linear)

# Set titles and labels
ax.set_title('Linear Scale')
ax.set_xlabel("X-axis")
ax.set_ylabel("Y-axis")
ax.grid(True)

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```

The graph generated by the code using the linear scale is shown in Figure 10-9.

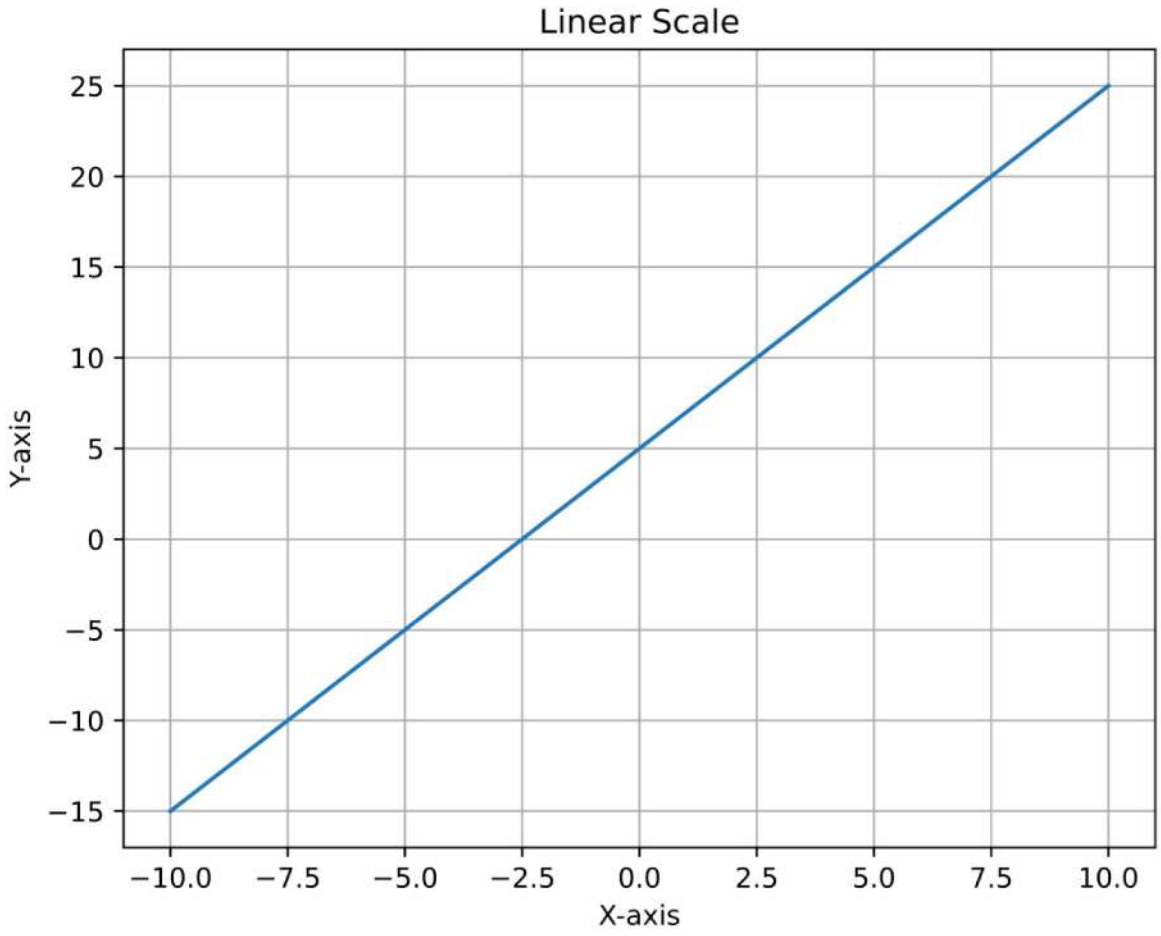


Figure 10-9 Example of a plot using the default linear scale

Logarithmic Scale

This code demonstrates a logarithmic (log) scale, which is ideal for displaying data that spans several orders of magnitude. After importing the required libraries, it generates exponential data where y is 10x. This type of data would appear as a sharp, hard-to-read curve on a linear scale. By setting the y-axis scale to 'log', the exponential curve is transformed into a straight line, making the underlying trend much clearer. The script adds titles, labels, and a grid that shows both major and minor tick marks (which="both"), which is helpful on log scales, before showing the final plot.

The function prototype of logarithmic scale is

```
ax.set_xscale('log', base=10, nonpositive='clip')
```

To use the function specify only the scale name 'log'; all other parameters are optional. The parameter "base" sets the base of the logarithm for the axis scaling. The default value is 10. And "nonpositive" determines how to handle non-positive values; use 'clip' to a small positive number or 'mask' to hide them.

```
import matplotlib.pyplot as plt
import numpy as np

# --- 2. Log Scale ---
# Create a figure and a single subplot
fig, ax = plt.subplots(figsize=(6, 5))

# Generate data
x_log = np.linspace(1, 5, 100)
y_log = 10**x_log  # Exponential data

# Plot the data
ax.plot(x_log, y_log)

# Set the y-axis to a logarithmic scale
ax.set_yscale('log')

# Set titles and labels
ax.set_title('Log Scale')
ax.set_xlabel("X-axis (Linear)")
ax.set_ylabel("Y-axis (Log)")
ax.grid(True, which="both")

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```

The graph generated by the code using the log scale on the y-axis is shown in Figure 10-10.

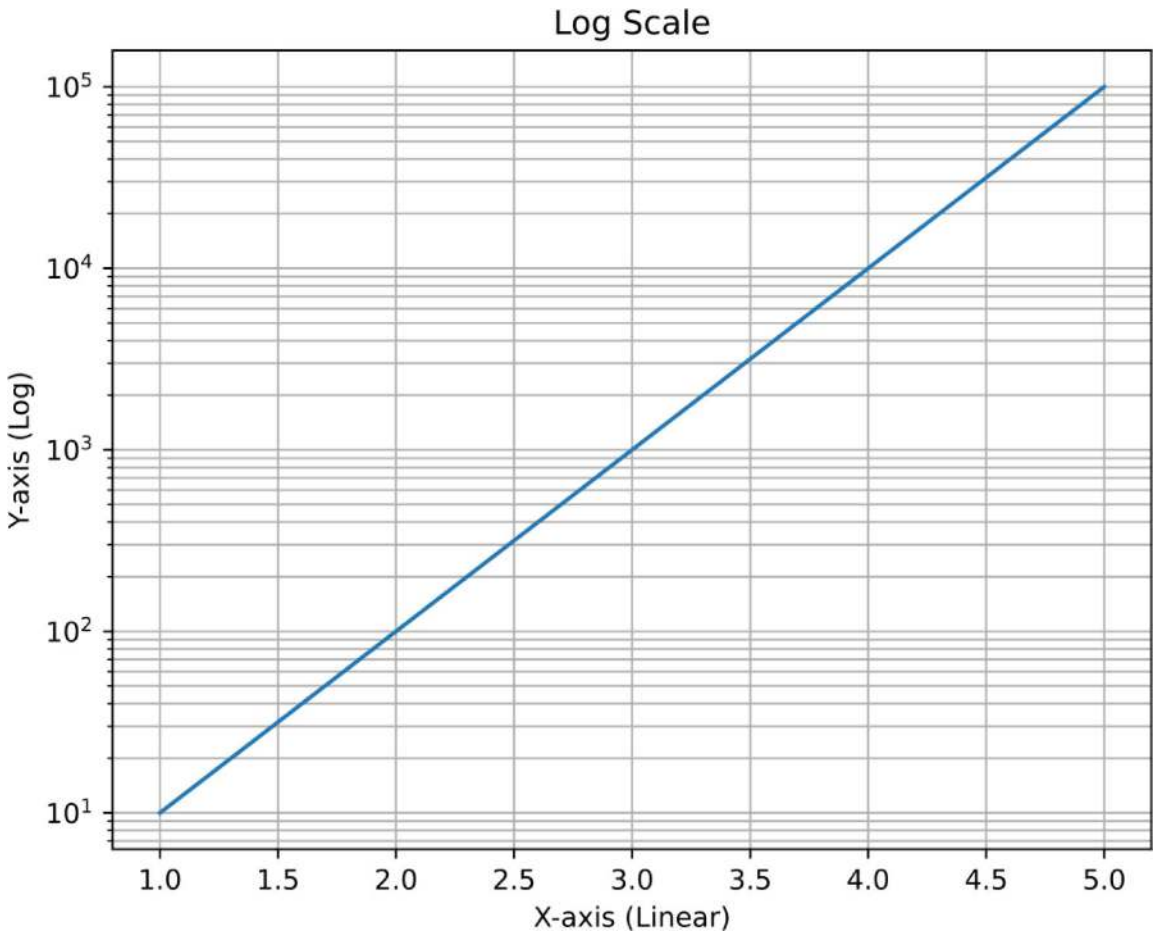


Figure 10-10 Example of a plot using a logarithmic scale on the y-axis

Symmetric Logarithmic Scale

We now introduce the symmetric logarithmic (symlog) scale, a specialized scale for visualizing data that, like a logarithmic scale, spans a wide range but also includes zero or negative values. The code generates data for a cubic function (y = x3), which has both positive and negative values and crosses zero. The command set_yscale('symlog', linthresh=10) applies the symlog scale to the y-axis, creating a logarithmic scale for values far from zero while maintaining a linear scale in the region near zero (within a limit of ±10). This hybrid approach avoids the problems that arise when taking the logarithm of zero or negative numbers, allowing for effective visualization of data with a wide dynamic range centered around the origin.

```
ax.set_xscale('symlog', linthresh=1.0, base=10, linscale=1.0)
```

To use the function, simply type the name of the scale; all other parameters are optional. The "linthresh" parameter defines the range around zero (–linthresh to +linthresh) where the scale is linear. As with the logarithmic scale function, the "base" parameter defines the base of the logarithm for the axis scale, with the default value being 10. And "linscale" allows you to adjust the space that the linear section occupies relative to the logarithmic sections.

```
import matplotlib.pyplot as plt
```

```
import numpy as np

# --- 3. Symmetric Log (Symlog) Scale ---
# Create a figure and a single subplot
fig, ax = plt.subplots(figsize=(6, 5))

# Generate data
x_symlog = np.linspace(-100, 100, 200)
y_symlog = x_symlog**3  # Data that crosses zero

# Plot the data
ax.plot(x_symlog, y_symlog)

# Set y-axis to symlog scale with a linear threshold around zero
ax.set_yscale('symlog', linthresh=10)

# Set titles and labels
ax.set_title('Symmetric Log (Symlog) Scale')
ax.set_xlabel("X-axis (Linear)")
ax.set_ylabel("Y-axis (Symlog)")
ax.grid(True)

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```

The result of the code using the symmetric log scale is seen in Figure 10-11.

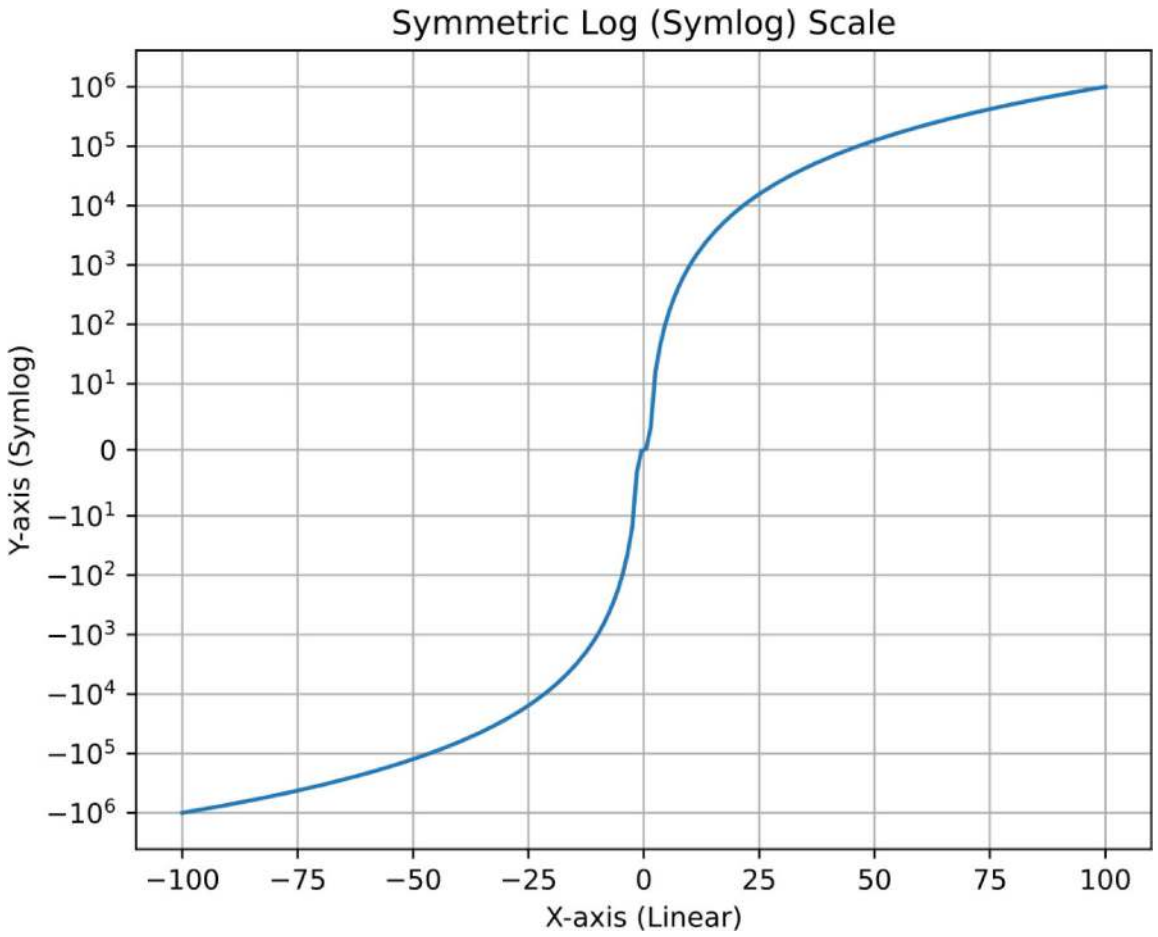


Figure 10-11 Example of a plot using a symmetric log (symlog) scale

Logit Scale

The next code creates a plot using a logit scale, which is specifically designed for visualizing data that represents proportions or probabilities, confined to the range between 0 and 1. The script generates data using the logistic (or sigmoid) function, a common S-shaped curve that models probabilities. When plotted, the set_yscale('logit') command transforms the y-axis to effectively stretch out the regions near 0 and 1. This transformation makes it much easier to discern differences in data points that are very close to the extremes (e.g., distinguishing between 0.99 and 0.999), which would be compressed and indistinguishable on a linear scale.

```
ax.set_xscale('logit', nonpositive='clip')
```

As with the logarithmic scale function, the "nonpositive" parameter is not mandatory but determines how to handle non-positive values; use 'clip' for a small positive number or 'mask' to hide them.

```
import matplotlib.pyplot as plt
import numpy as np
# --- 4. Logit Scale ---
# Create a figure and a single subplot
```

```
fig, ax = plt.subplots(figsize=(6, 5))

# Generate data
x_logit = np.linspace(-6, 6, 100)
y_logit = 1 / (1 + np.exp(-x_logit))  # Logistic (sigmoid) function

# Plot the data
ax.plot(x_logit, y_logit)

# Set the y-axis to a logit scale
ax.set_yscale('logit')

# Set titles and labels
ax.set_title('Logit Scale')
ax.set_xlabel("X-axis (Linear)")
ax.set_ylabel("Y-axis (Logit)")
ax.grid(True)

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```

The code plot using the logit scale is shown in Figure 10-12.

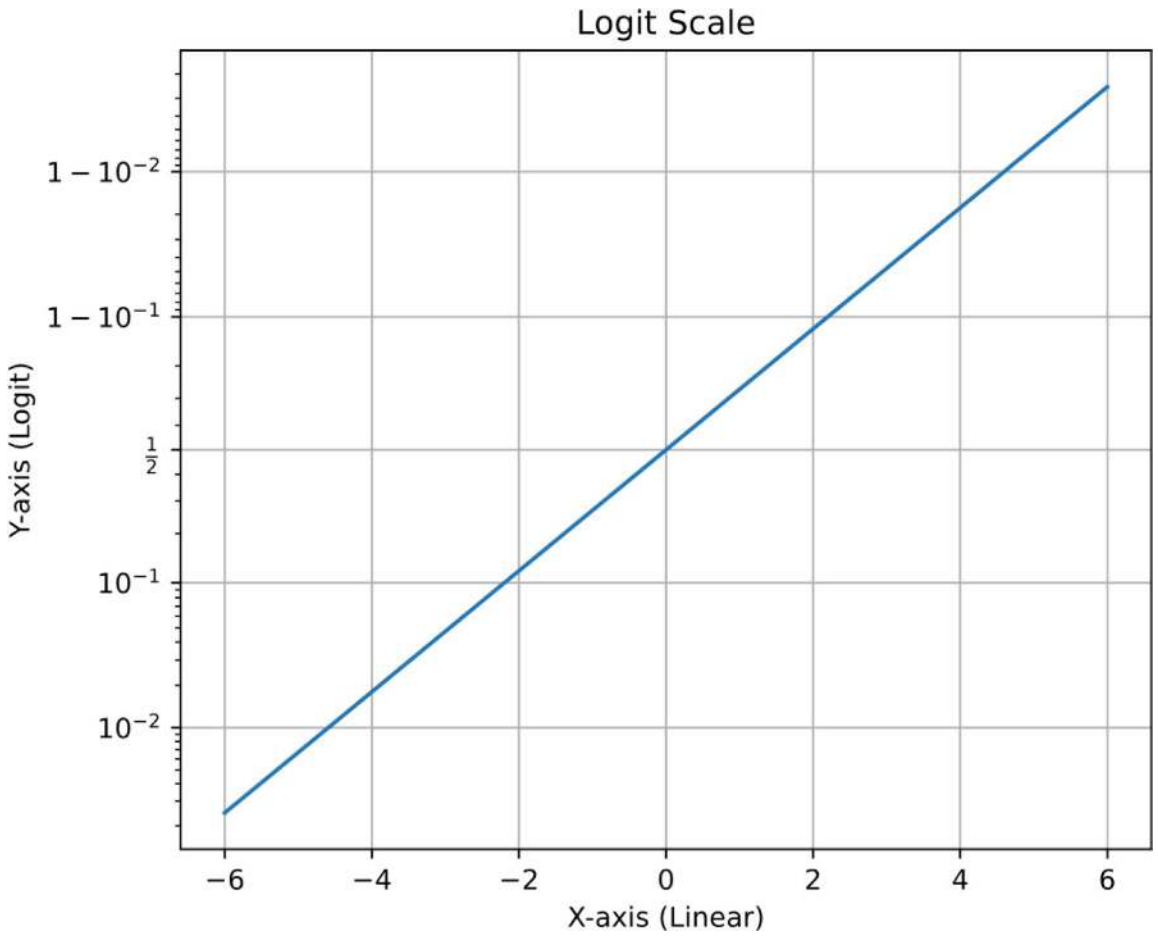


Figure 10-12 Example of a plot using a logit scale for probability data

Inverse Sine Hyperbolic Scale

The inverse sine hyperbolic (asinh) scale is a clever hybrid scale that combines the best features of both linear and logarithmic scales. The asinh scale is the perfect tool for those who have very small and very large values that they want to see clearly on the same graph. This is because the asinh scale has two distinct behaviors, depending on the data value:

Near Zero: For values close to zero (both positive and negative), the scale behaves linearly. This is its main advantage over a standard logarithmic scale, which cannot handle zero or negative numbers. It avoids the infinite decay that you would get with log(0).
Far from Zero: As the absolute value of the data increases, the scale smoothly transitions to behaving like a logarithmic scale. This means that it compresses large values, making it excellent for visualizing data that spans several orders of magnitude.

The prototype of the function asinh scale is

```
ax.set_yscale('asinh', linear_width=1.0, base=10)
```

To use the scale, provide the scale name to set_xscale() or set_yscale(). The linear_width parameter defines the effective width of the quasi-linear region around zero. The base

parameter controls how tick locations are rounded; it does not change the inverse hyperbolic sine transformation. The default base is 10.

```
# --- 1. Import Library ---
import matplotlib.pyplot as plt
import numpy as np

# --- 2. Prepare Data ---
x_data = np.linspace(-8, 8, 400)
y_data = x_data**3

# --- 3. Create Figure & Axes ---
fig, ax = plt.subplots(figsize=(8, 6))

# --- 4. Plot Data and Set Scale ---
ax.plot(x_data, y_data)

# Apply the asinh scale to the y-axis
ax.set_yscale('asinh')

# --- 5. Customize the Plot ---
ax.set_title('Asinh (Inverse Hyperbolic Sine) Scale')
ax.set_xlabel("Linear Input")
ax.set_ylabel("Asinh Scaled Output")
ax.grid(True)

# --- 6. Show the Plot ---
plt.show()
```

The graph using the asinh scale generated by the above code is represented in Figure 10-13.

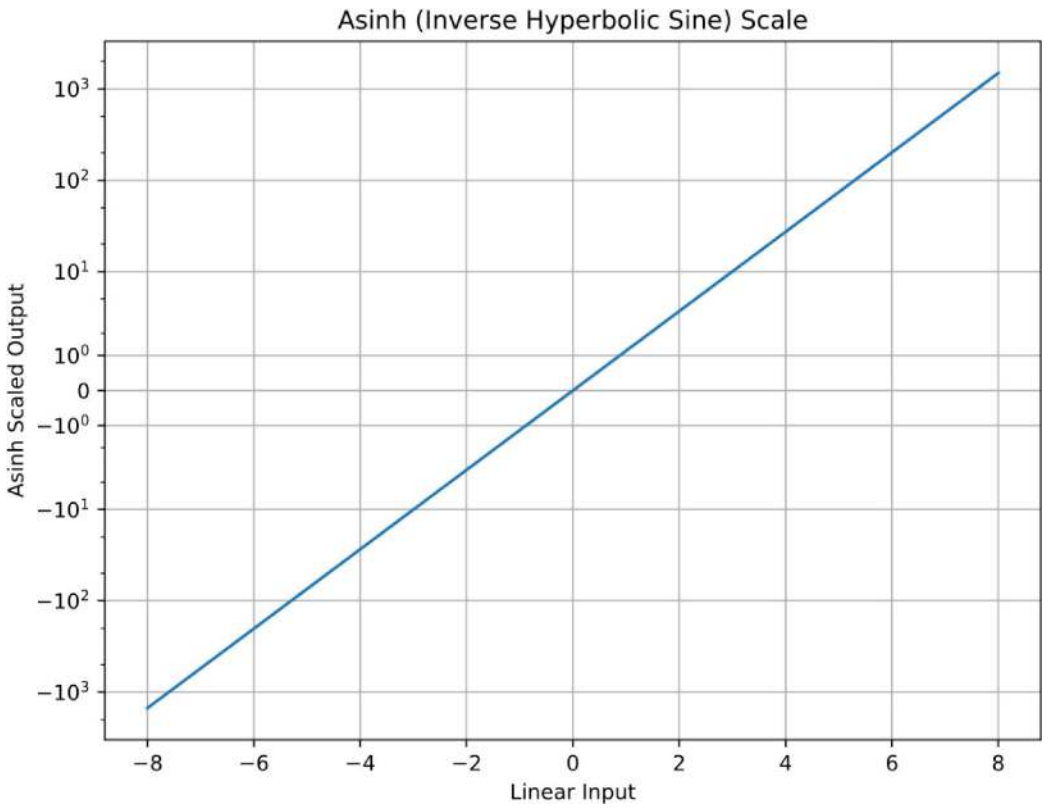


Figure 10-13 Example of a plot using an inverse hyperbolic sine (asinh) scale

Custom Scales

Custom scales are powerful tools for specialized situations where standard scales don't adequately represent the data's meaning. Matplotlib's custom function scale allows you to transform the axis of a plot using your own mathematical formulas. Instead of being limited to standard scales like linear or log, you can create a visualization that perfectly matches the nature of your data. To create a custom scale, you must provide two key components:

A Forward Function: This is the mathematical transformation you want to apply to your data values before they are plotted.
An Inverse Function: This function reverses the forward transformation. Matplotlib needs this to correctly calculate the positions of tick marks and interact with the plot.

The prototype of the function scale is

```
ax.set_xscale('function', functions=(forward_func, inverse_func))
```

The parameter functions provides a tuple containing two functions: the forward transformation and its inverse. To use the function you need the scale name and the functions keyword argument, which requires a tuple containing your two custom functions (forward_func, inverse_func). Table 10-8 describes various custom scale functions for data visualization, each with a forward function and its corresponding inverse function. These scales can be used to emphasize specific patterns or relationships in the data.

Table 10-8 Examples of custom scale transformations

This code creates a plot with a custom “squared” scale on the x-axis, emphasizing differences between larger x values. It defines the forward_square() and inverse_square() functions, prepares the data, plots it, and applies the custom scale. Manual axis limit setting ensures the plot displays correctly, and customization includes a title, labels, and a grid.

```
import matplotlib.pyplot as plt
import numpy as np
# --- 5. Custom Scale ---

# --- 1. Define the Custom Scale Functions ---
def forward_square(x):
    return x**2

# The inverse function reverses the transformation.
def inverse_square(x):
    return np.sqrt(x)

# --- 2. Prepare Data ---
# Data must be valid for the inverse function (e.g., no negative
numbers for sqrt).
x_data = np.linspace(0, 10, 100)
y_data = np.cos(x_data) # Some data to plot against the custom
scale

# --- 3. Create Figure & Axes ---
fig, ax = plt.subplots(figsize=(8, 6))

# --- 4. Plot Data ---
ax.plot(x_data, y_data)

# --- 5. Set Scale ---
ax.set_xscale('function', functions=(forward_square,
inverse_square))

# --- 6. Manually Set Axis Limits (The Fix) ---
# Explicitly setting the limits prevents the plot from appearing
empty.
# Set x-limits based on the original data's range.
ax.set_xlim(x_data.min(), x_data.max())
# Set y-limits based on the plotted data's range, with a little
padding.
ax.set_ylim(y_data.min() - 0.1, y_data.max() + 0.1)

# --- 7. Customize the Plot ---
ax.set_title('Custom "Function" Scale (x-axis is squared)')
```

```
ax.set_xlabel("X-axis (Squared Scale)")
ax.set_ylabel("Y-axis (Linear Scale)")
ax.grid(True)

# --- 8. Show the Plot ---
plt.show()
```

It's important to note that after applying a custom function scale, Matplotlib's automatic axis scaling may not work correctly. Therefore, you must often set the axis limits manually using ax.set_xlim() and ax.set_ylim() to ensure the data is displayed properly.

The result is a unique visualization that highlights specific patterns in the data.

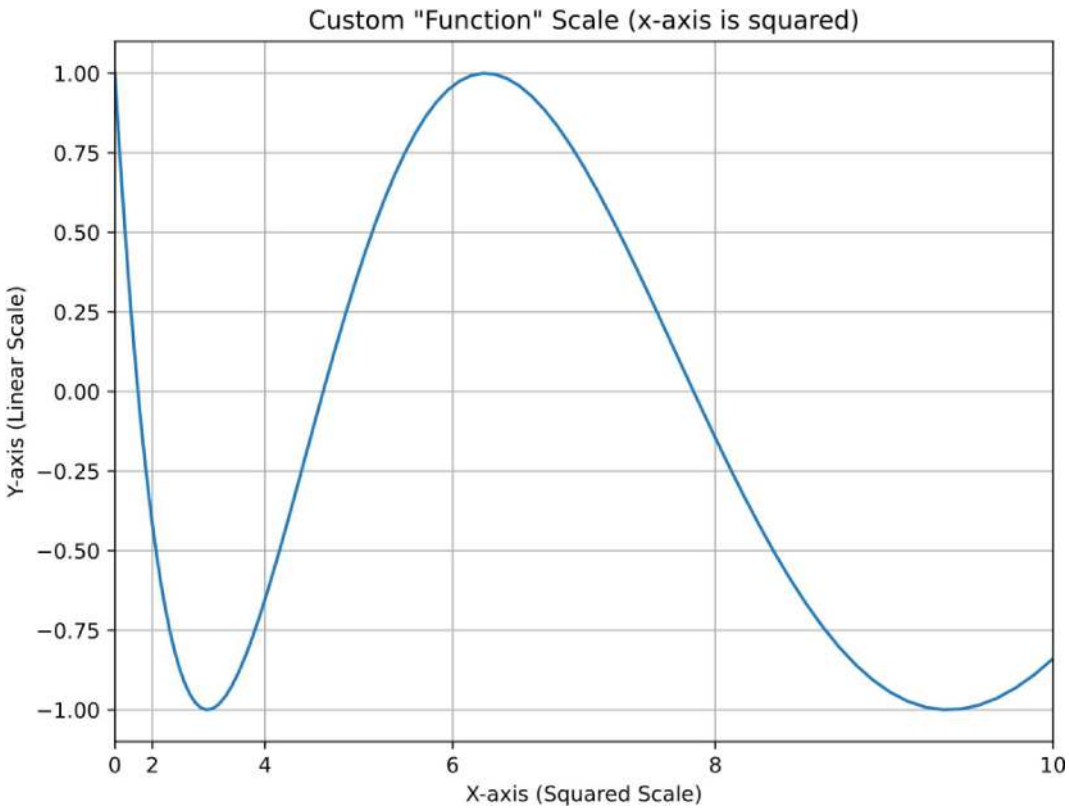


Figure 10-14 Example of a plot using a custom "function" scale (squared x-axis)

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_11

11. NumPy

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Introduction

NumPy is an acronym for Numerical Python, a library for working with arrays and mathematical operations. An array is a data structure that stores a collection of elements, each identified by an index or subscript. The elements are stored in contiguous memory locations, allowing for efficient access and manipulation, and typically they are of the same data type.

A key feature of NumPy is that it's written using the C programming language, with additional components in Python and Fortran. The C implementation of the NumPy core is justified by performance reasons, which is why NumPy operations are significantly faster than equivalent pure Python code, executing at the speed of compiled C rather than interpreted Python. Python provides the user-facing API, while Fortran is used for specific components, particularly linear algebra operations. This hybrid architecture combines high performance with ease of use.

Table 11-1 lists all the families of operations available in NumPy.

Table 11-1 Overview of main operation categories in NumPy

Installation and Version Check

The examples in this chapter target NumPy 2.x and were originally tested with version 2.3.5. Before installing, check the official NumPy website for the current stable release.

NumPy can be installed easily using pip. To install the most recent stable release, you simply need to run

```
pip install numpy
```

As seen in Chapter 1, the use of a virtual environment is highly recommended, as it isolates the Python installation from the system and avoids version conflicts with other packages. After installation, NumPy can be imported into any Python script using the conventional alias "import numpy as np", which exposes all its array functionalities, numerical tools, data types, and computational routines.

When working on larger engineering or scientific projects, it is important to verify that your environment is indeed using the intended version of NumPy. Sometimes multiple Python environments may exist on the same machine, and IDEs such as PyCharm and VS Code may activate a different interpreter than expected. A simple way to confirm the active NumPy version is to print np.__version__ directly within the script or interpreter:

```
import numpy as np
print("NumPy version:", np.__version__)  # Output
depends on the installed version
```

To create a new array, simply pass a Python list to np.array():

```
import numpy as np

# Creating a simple 1-D array (vector)
a = np.array([1, 2, 3, 4, 5])
print(a) # Output: [1 2 3 4 5]
```

NumPy and SciPy

By offering a large number of robust mathematical operations, NumPy serves as a fundamental layer for many other scientific libraries, such as SciPy, Pandas, scikit-learn, TensorFlow, and others. Because SciPy is built directly on top of NumPy, several operations exist in both libraries, including FFT routines, basic linear algebra, interpolation, statistical functions, and more. In practice, the same type of operation can sometimes be performed using either NumPy or SciPy, which may create confusion for beginners.

To help you choose the most recommended library for each task, the comparison table below clearly shows when you should prefer NumPy (usually for basic, fast, array-based operations) and when SciPy is the better choice (for

more complete, stable, and scientifically accurate implementations). This way, you can always select the most appropriate tool for your application.

Table 11-2 Feature comparison and selection guide between NumPy and SciPy

To make the most of both NumPy and SciPy, this book presents the strengths of each library in a complementary way. This chapter on NumPy focuses on the core and foundational features that are unique to NumPy, the building blocks of the entire scientific Python ecosystem. Chapter 12 on SciPy, on the other hand, covers the advanced or "premium" features that extend NumPy's capabilities with more complete and scientifically robust tools.

NumPy Subpackages

NumPy is organized into several specialized subpackages, each responsible for a specific set of functionalities that extend the capabilities of the ndarray array. These subpackages include modules for linear algebra, Fourier Transforms, random number generation, polynomial manipulation, string operations, modern type systems, integration with C code, and much more. Table 11-3 below presents provides an overview of the main subpackages available in NumPy 2.x, highlighting their purposes and areas of application, facilitating an understanding of how the library is structured internally.

Table 11-3 Main subpackages available in NumPy and their functional descriptions

Data Types

The following table presents the main data types (dtypes) supported by NumPy, each optimized to efficiently represent information in vector operations. These dtypes range from boolean and integer values with different widths (8, 16, 32, and 64 bits) to floating-point numbers in various precisions, complex numbers, strings, bytes, generic objects, and specialized types such as dates, time intervals, and structured records. Each category offers a well-defined range of values, ensuring consistent behavior across platforms, except in specific cases such as float128 and complex256, whose availability depends on the architecture. Understanding these types is essential for choosing appropriate representations, optimizing performance, controlling memory

usage, and ensuring numerical precision in scientific and engineering applications.

Table 11-4 Main data types (dtypes), descriptions and value ranges supported by NumPy

Although a boolean value theoretically requires only 1 bit to represent True or False, NumPy stores each numpy.bool_ element using a full byte (8 bits). This design choice offers performance gains and simplicity.

Modern processors are optimized to read and write data in units of at least 1 byte and, in practice, often in larger blocks (such as 4 or 8 bytes). Storing boolean values as individual bits would require extra calculations to locate the correct bit within a byte and extract its value, significantly reducing performance. By using a full byte per boolean, NumPy ensures fast, straightforward, and properly aligned memory access. Furthermore, allocating 1 byte per element keeps boolean arrays consistent with NumPy's fixed-size data type model. This simplifies indexing, interoperability with C/C++, and integration with the rest of the NumPy numerical infrastructure.

Starting with NumPy 2.0, StringDType provides variable-width UTF-8 strings. Unlike fixed-width np.str_ and np.bytes_ arrays, it stores strings without requiring one common character width, reducing truncation risk and supporting modern vectorized text processing.

StringDType is available as numpy.dtypes.StringDType and works with vectorized operations in numpy.strings.

NumPy Data Types: Examples

1. *Boolean*: bool_

   Useful for threshold-based engineering decisions or sensor state evaluations. Booleans are extremely useful in engineering for threshold analysis, fault detection, alarm triggering, filter masks, and any situation requiring a true/false decision.

```
import numpy as np
temperatures=np.array([22,35,40,28,31])
mask=temperatures>30
print(mask)# Output: [False  True  True False  True]
```

2. *Signed and Unsigned Integers*: int8, int16, int32, int64, uint8, uint16,

uint32, uint64

Integers are widely used in engineering to represent sensor readings, counters, state indexes, and others. Signed integers can store negative and positive values.

Unsigned integers (uintX) are ideal for raw data, such as sensor buffers, images, or communication bytes.

```
import numpy as np
analog_values=np.array([0,128,255],dtype=np.uint8)
print(analog_values)# Output: [  0 128 255]
```

3. *Floating Point*: float16, float32, float64

Floating-point numbers are fundamental in engineering to represent physical measurements, engineering simulations, and calculations. float64 is the most common and recommended type due to its precision in scientific and control calculations.

```
import numpy as np
forces=np.array([12.5,15.3,9.81],dtype=np.float32)
print(forces)# Output: [12.5  15.3  9.81]
```

4. *Complex Numbers*: complex64, complex128

Complex numbers allow natural representation of magnitude and phase. Essential in signal processing, AC circuit analysis, and control systems.

```
import numpy as np
signal=np.array([1+2j,3+4j],dtype=np.complex128)
print(signal.real)# Output: [1. 3.]
print(signal.imag)# Output: [2. 4.]
```

5. *String Type*: str_

While less common than numerical values, strings are essential to label engineering data.

```
import numpy as np
labels=np.array(["Motor","Sensor"],dtype=np.str_)
print(labels)# Output: ['Motor' 'Sensor']
```

6.

Bytes Types: bytes_

Used in serial communication, binary protocols, or microcontroller messages. These are perfect for representing data exactly as received from a serial or bus interface.

```
import numpy as np
packet=np.array([b'H',b'I'],dtype=np.bytes_)
print(packet)# Output: [b'H' b'I']
```

7.

Modern UTF-8 Strings: numpy.dtypes.StringDType

StringDType is NumPy's new string type, fully compatible with UTF-8 and without a fixed size. It overcomes the limitations of the older np.str_ and np.bytes_, allowing for efficient storage of text of any length without wasting memory or risking truncation. This allows NumPy to better handle component names, metadata, file paths, and general descriptions, offering behavior much closer to that of strings used in high-level languages.

```
import numpy as np
from numpy.dtypes import StringDType

components = np.array(
    ["Valve", "Pump"],
    dtype=StringDType()
)
print(components)       # Output: ['Valve' 'Pump']
print(components.dtype) # Output: StringDType()
```

8. *Object Type*: object_

Useful for heterogeneous engineering datasets. Allows storing any Python object: lists, strings, numbers, dictionaries.

Useful when engineering datasets are heterogeneous, such as mixed logs, imported raw data, non-standardized datasets. Because it is slower and not vectorized, it should be used with caution.

```
import numpy as np
mixed=np.array([42,"Rotor",[1,2]],dtype=object)
print(mixed)# Output: [42 'Rotor' list([1, 2])]
```

9.

Datetime: datetime64

Used in logging, monitoring, and timestamping engineering events. Supports multiple time resolutions, from nanoseconds to days.

```
import numpy as np
log_times=np.array(["2025-01-01","2025-01-
10"],dtype="datetime64[D]")
print(log_times[1]-log_times[0])# Output: 9 days
```

10.

Timedelta: timedelta64

Represents durations such as time differences in system measurements. Ideal for time-series analysis and temporal engineering models.

```
import numpy as np
durations=np.array([1,5,10],dtype="timedelta64[D]")
print(durations.astype('timedelta64[D]').astype(str))
# Output: ['1 days' '5 days' '10 days']
```

11.

Structured Types: custom dtype([...])

Structured dtypes allow creating table-like structures inside NumPy arrays, perfect for organized engineering data, like sensors with fields like name, unit, offset, value, and imported CSV technical tables. Ideal for organizing datasets without needing specialized external data libraries like Pandas.

```
import numpy as np
dt=np.dtype([("name","U10"),("mass","f4")])
parts=np.array([("Bolt",0.02),("Gear",1.5)],dtype=dt)
print(parts["name"])# Output: ['Bolt' 'Gear']
```

Void Type: void

12. *Void Type*: void

The void dtype is how NumPy represents a whole row of a structured array. Instead of returning each field separately, NumPy packs all the fields into a single object when you access a full record (e.g., row = table[0]). You can think of it like a small "box" that contains all the values of that row together. This makes it easier and faster to store or move complete engineering records, especially when they contain several fields.

```
import numpy as np

dt=np.dtype([("x","i4"),("y","i4")])
coords=np.array([(3,4)],dtype=dt)
print(type(coords[0])) # Output: <class 'numpy.void'>
```

Python Objects in NumPy Arrays

NumPy can also store arbitrary Python objects inside an array by using the object data type. This makes it possible to mix integers, strings, lists, dictionaries, or even user-defined classes within a single array. However, arrays with dtype=object do not benefit from NumPy's optimized, contiguous memory layout. Instead of compact numeric storage, each element becomes a pointer to a standalone Python object. As a consequence, performance is significantly reduced: vectorized ufuncs and broadcasting cannot be applied efficiently, and operations fall back to standard Python loops.

For these reasons, object arrays should be used only when truly necessary, typically for heterogeneous data that cannot be represented with standard NumPy dtypes.

Here is an example showing how to store Python objects:

```
import numpy as np
# A NumPy array containing Python objects of different
types
arr = np.array([1, "hello", [1, 2, 3]], dtype=object)
print(arr) # Output: [1 'hello' list([1, 2, 3])]
```

Multidimensional Arrays and Matrices

NumPy provides efficient support for large, multidimensional arrays and matrices. Its vectorized operations enable concise and highly optimized numerical code.

The number of dimensions determines how data is structured in a NumPy array. A 0D array (scalar) contains a single value; a 1D array (vector) stores a sequence of values accessed with one index; a 2D array (matrix) arranges data into rows and columns, requiring two indexes; and a 3D array (tensor) represents a stack of 2D matrices, similar to image layers or slices. Arrays with four or more axes—often called N-D or higher-dimensional tensors—are used to model more complex datasets such as simulation grids or batches of images. This unified multidimensional structure allows NumPy to represent nearly any type of numerical data while maintaining consistent indexing rules and efficient mathematical operations.

See the table below for the different dimensions accepted in NumPy arrays.

Table 11-5 Summary of array dimensions, usage purposes, and code examples

Higher-dimensional tensors play a central role in modern applications such as deep learning, numerical simulations, and physical system modeling, where multidimensional data naturally arises.

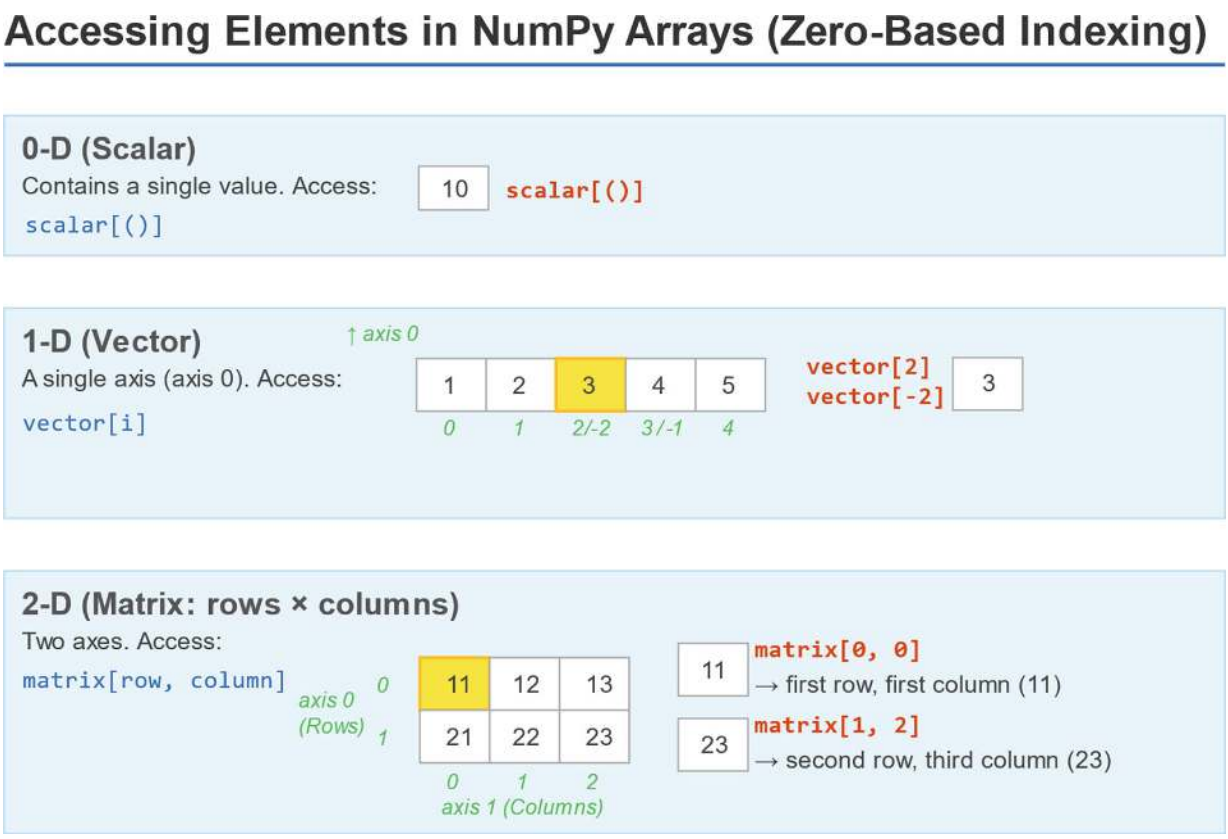


Figure 11-1 Conceptual representation of NumPy array dimensions (0D to N-D)

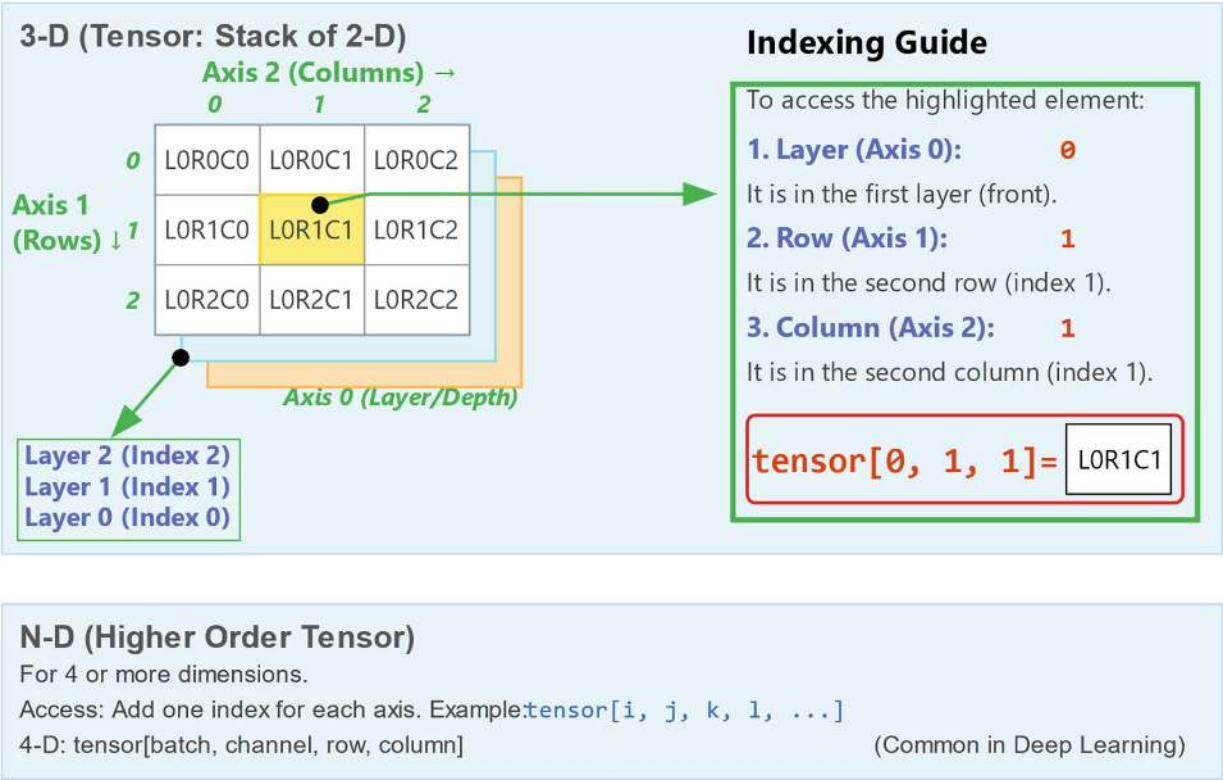


Figure 11-2 Memory layout and indexing structure of multidimensional NumPy arrays

The code below demonstrates how NumPy represents data in different dimensions using arrays. It creates a 0D scalar, a 1D vector, a 2D matrix, a 3D tensor composed of stacked matrices, and, finally, an N-D (4D) tensor generated with random values. When possible, the code prints the array along with its format, allowing you to visualize how NumPy organizes data, from a single value to high-dimensional tensors:

```
import numpy as np

# ----------------------------------------
# 0-D ARRAY (Scalar)
# ----------------------------------------
scalar = np.array(42)
print("0-D array (scalar):", scalar)
print("Shape:", scalar.shape)
# Output:
# 0-D array (scalar): 42
# Shape: ()
```

```
# ----------------------------------------
# 1-D ARRAY (Vector)
# ----------------------------------------
vector = np.array([1, 2, 3, 4])
print("\n1-D array (vector):", vector)
print("Shape:", vector.shape)
# Output:
# 1-D array (vector): [1 2 3 4]
# Shape: (4,)

# ----------------------------------------
# 2-D ARRAY (Matrix: rows × columns)
# ----------------------------------------
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
print("\n2-D array (matrix):")
print(matrix)
print("Shape:", matrix.shape)
# Output:
# [[1 2 3]
#  [4 5 6]]
# Shape: (2, 3)

# ----------------------------------------
# 3-D ARRAY (Tensor: stack of 2-D matrices)
# ----------------------------------------
tensor_3d = np.array([
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]]
])
print("\n3-D array (tensor):")
print(tensor_3d)
print("Shape:", tensor_3d.shape)
# Output:
# [[[1 2]
#   [3 4]]
#
#  [[5 6]
```

```
#   [7 8]]]
# Shape: (2, 2, 2)

# ----------------------------------------
# N-D ARRAY (Higher-D tensor: 4-D example)
# ----------------------------------------
tensor_nd = np.random.rand(2, 3, 4, 5)
print("\nN-D array (4-D tensor):")
print("Shape:", tensor_nd.shape)
# Output (dimensions may vary):
# Shape: (2, 3, 4, 5)
```

Array Creation Patterns in NumPy

NumPy provides a wide range of creation functions that support everything from converting Python sequences into efficient numerical arrays to generating structured grids, initialized matrices, random data, and arrays with specific memory layouts. These creation methods allow users to control dtype, shape, initialization strategy, and distribution of values whether through deterministic patterns such as zeros, ones, linspace, and arange or stochastic generation using NumPy's modern random API.

Table 11-6 summarizes the most common and essential ways to create specialized NumPy arrays.

Table 11-6 Array creation functions, commands, and initialization patterns

See examples of the features listed above in the code below:

```
import numpy as np

# ----------------------------------------------------
------
# From Python list / tuple
# ----------------------------------------------------
------
a1 = np.array([1, 2, 3])
print(a1)
# Output: [1 2 3]
```

```
# ------------------------------------------------------
------
# From Python data (no copy if possible)
# ------------------------------------------------------
------
my_list = [10, 20, 30]
a2 = np.asarray(my_list)
print(a2)
# Output: [10 20 30]

# ------------------------------------------------------
------
# With explicit dtype
# ------------------------------------------------------
------
a3 = np.array([1, 2, 3], dtype=np.float64)
print(a3)
# Output: [1. 2. 3.]

# ------------------------------------------------------
------
# Zeros
# ------------------------------------------------------
------
z = np.zeros((3, 3))
print(z)
# Output:
# [[0. 0. 0.]
#  [0. 0. 0.]
#  [0. 0. 0.]]

# ------------------------------------------------------
------
# Ones with dtype
# ------------------------------------------------------
------
o = np.ones((2, 4), dtype=np.int32)
print(o)
```

```
# Output:
# [[1 1 1 1]
#  [1 1 1 1]]

# ------------------------------------------------------
------
# Constant value (full)
# ------------------------------------------------------
------
full = np.full((2, 3), 7.5)
print(full)
# Output:
# [[7.5 7.5 7.5]
#  [7.5 7.5 7.5]]

# ------------------------------------------------------
------
# Like another array
# ------------------------------------------------------
------
a = np.array([[1, 2], [3, 4]])
like = np.zeros_like(a)
print(like)
# Output:
# [[0 0]
#  [0 0]]

# ------------------------------------------------------
------
# Identity matrix
# ------------------------------------------------------
------
I = np.eye(3)
print(I)
# Output:
# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]
```

```
# ------------------------------------------------------
------
# Diagonal from 1-D array
# ------------------------------------------------------
------
diag = np.diag([1, 2, 3])
print(diag)
# Output:
# [[1 0 0]
#  [0 2 0]
#  [0 0 3]]

# ------------------------------------------------------
------
# Arange (integer steps)
# ------------------------------------------------------
------
ar = np.arange(1, 10, 2)
print(ar)
# Output: [1 3 5 7 9]

# ------------------------------------------------------
------
# Linspace (evenly spaced)
# ------------------------------------------------------
------
lin = np.linspace(0, 10, 5)
print(lin)
# Output: [ 0.   2.5  5.   7.5 10. ]

# ------------------------------------------------------
------
# Geometric spacing
# ------------------------------------------------------
------
geo = np.geomspace(1, 1000, 4)
print(geo)
# Output: [   1.   10.  100. 1000.]
```

```
# ------------------------------------------------------
------
# Logarithmic spacing
# ------------------------------------------------------
------
log = np.logspace(0, 3, 4)
print(log)
# Output: [   1.   10.  100. 1000.]

# ------------------------------------------------------
------
# From function (index-based)
# ------------------------------------------------------
------
func = np.fromfunction(lambda i, j: i + j, (3, 3),
dtype=int)
print(func)
# Output:
# [[0 1 2]
#  [1 2 3]
#  [2 3 4]]

# ------------------------------------------------------
------
# Random uniform – modern Generator API
# ------------------------------------------------------
------
rng = np.random.default_rng(42)
rand_uniform = rng.random((3, 3))
print(rand_uniform)
# Output:
# [[0.77395605 0.43887844 0.85859792]
#  [0.69736803 0.09417735 0.97562235]
#  [0.7611397  0.78606431 0.12811363]]

# ------------------------------------------------------
------
# Random integers – modern API
```

```
# ------------------------------------------------------
------
rand_int = rng.integers(0, 10, size=(2, 4))
print(rand_int)
# Output:
# [[8 4 5 3]
#  [1 9 7 6]]

# ------------------------------------------------------
------
# Random normal – modern API
# ------------------------------------------------------
------
rand_norm = rng.normal(loc=0.0, scale=1.0, size=(2, 2))
print(rand_norm)
# Output:
# [[ 1.12724121  0.46750934]
#  [-0.85929246  0.36875078]]

# ------------------------------------------------------
------
# Random with same shape as an existing array
# ------------------------------------------------------
------
rand_like = rng.random(a.shape)
print(rand_like)
# Output:
# [[0.06381726 0.82763117]
#  [0.6316644  0.75808774]]

# ------------------------------------------------------
------
# Uninitialized array (unsafe)
# ------------------------------------------------------
------
empty_arr = np.empty((4, 4))
print(empty_arr)
# Output (garbage, unpredictable):
```

```
# [[4.67e-307 1.39e-309 ...]
#  [... arbitrary memory values ...]]

# ------------------------------------------------------
------
# Reshape after creation
# ------------------------------------------------------
------
reshaped = np.arange(6).reshape(2, 3)
print(reshaped)
# Output:
# [[0 1 2]
#  [3 4 5]]
```

Accessing Elements in NumPy Arrays

NumPy uses a powerful indexing system that allows accessing elements from arrays of any dimension, from scalars (0D) to high-dimensional tensors (N-D).

Indexing always starts at zero, just like standard Python, but NumPy expands this behavior to multiple axes.

Indexing in Different Dimensions

0D (Scalar)

A 0D array contains a single value.

It behaves like a regular number, but it is still a NumPy array.

To access it:

scalar[()] → standard way to access a NumPy scalar

1D (Vector)

A vector has a single axis (axis 0).

To access an element, you use one index:

vector[i] → accesses element at position *i*

vector[-1] → accesses the last element

Negative indexes work the same as in Python lists.

2D (Matrix: Rows × Columns)

A matrix has two axes:

axis 0 → rows

axis 1 → columns

To access an element:
matrix[row, column]
Examples:
matrix[0, 0] → first row, first column
matrix[1, 2] → second row, third column

3D (Tensor: Stack of 2D Matrices)
A 3D array has **three axes**:
axis 0 → depth layer
axis 1 → row
axis 2 → column
Access pattern:
tensor[layer, row, column]

N-D (Tensor of Higher Order)
For tensors with four or more dimensions (N-D tensors), additional axes and corresponding indices are appended as required:
Access pattern:
tensor[i, j, k, ...]
Simply add one index per axis.

```
import numpy as np

# ----------------------------------------
# 0-D ARRAY (Scalar)
# ----------------------------------------
scalar = np.array(42)
print("0-D array (scalar):", scalar) # Output: 0-D array
(scalar): 42
print("Shape:", scalar.shape) # Output: Shape: ()

# Accessing the value
print("Access:", scalar[()])
# Output:
# Access: 42

# ----------------------------------------
# 1-D ARRAY (Vector)
# ----------------------------------------
```

```
vector = np.array([1, 2, 3, 4])
print("\n1-D array (vector):", vector)
print("Shape:", vector.shape)

# Accessing individual elements
print("vector[0]:", vector[0])   # First element
print("vector[-1]:", vector[-1]) # Last element
# Output:
# 1-D array (vector): [1 2 3 4]
# Shape: (4,)
# vector[0]: 1
# vector[-1]: 4

# ----------------------------------------
# 2-D ARRAY (Matrix: rows × columns)
# ----------------------------------------
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
print("\n2-D array (matrix):")
print(matrix)
print("Shape:", matrix.shape)

# Accessing by row and column
print("matrix[0, 0]:", matrix[0, 0])  # Row 0, Column 0
print("matrix[1, 2]:", matrix[1, 2])  # Row 1, Column 2
# Output:
# 2-D array (matrix):
# [[1 2 3]
#  [4 5 6]]
# Shape: (2, 3)
# matrix[0, 0]: 1
# matrix[1, 2]: 6

# ----------------------------------------
# 3-D ARRAY (Tensor: stack of 2-D matrices)
# ----------------------------------------
tensor_3d = np.array([
    [[1, 2], [3, 4]],   # Layer 0
    [[5, 6], [7, 8]]    # Layer 1
```

```
])
print("\n3-D array (tensor):")
print(tensor_3d)
print("Shape:", tensor_3d.shape)

# Access pattern: [layer, row, column]
print("tensor_3d[0, 0, 0]:", tensor_3d[0, 0, 0])  # 1
print("tensor_3d[1, 1, 1]:", tensor_3d[1, 1, 1])  # 8
# Output:
# 3-D array (tensor):
# [[[1 2]
#   [3 4]]
#
#  [[5 6]
#   [7 8]]]
# Shape: (2, 2, 2)
# tensor_3d[0, 0, 0]: 1
# tensor_3d[1, 1, 1]: 8

# ----------------------------------------
# N-D ARRAY (Higher-D tensor: 4-D example)
# ----------------------------------------
tensor_nd = np.random.rand(2, 3, 4, 5)
print("\nN-D array (4-D tensor):")
print("Shape:", tensor_nd.shape)

# Example access: [a, b, c, d]
print(f"tensor_nd[0, 1, 2, 3]: {tensor_nd[0, 1, 2,
3]:.6f}")
# Output (example values):
# N-D array (4-D tensor):
# Shape: (2, 3, 4, 5)
# tensor_nd[0, 1, 2, 3]: 0.374829  # representative
value
```

Indexing Errors

Indexing errors occur when attempting to access an element at an index that is out of the valid range.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
value = arr[4]
# IndexError: index 4 is out of bounds for axis 0 with
size 4
```

Array Properties

NumPy arrays offer a comprehensive set of attributes known as ndarray properties that describe how the array is structured in memory and how it should be interpreted. These properties allow developers, engineers, and scientists to inspect dimensions, element types, memory usage, and data layout, making NumPy arrays powerful and transparent. Understanding these attributes is essential when working with large datasets or optimizing performance.

Table 11-7 lists the most important array properties.

Table 11-7 Key ndarray properties and structural memory attributes

---

Array Inspection and Utility Functions

Similarly, NumPy offers a set of practical functions for inspecting arrays and querying their structural or numerical properties. These tools are essential for analyzing dataset characteristics, validating inputs, and debugging calculations. The following table summarizes the most commonly used inspection and information functions.

Table 11-8 Array inspection and structural query utility functions

---

Below are two code examples, one that uses the Array Inspection features and another that uses the properties:

Example 1: Array Properties and Inspection

```
import numpy as np

# Sample 2-D array
arr = np.array([[1.0, 2.0, 3.0],
                [4.0, 5.0, 6.0]])
```

```
print("Array:")
print(arr)
# Output:
# [[1. 2. 3.]
#  [4. 5. 6.]]

print("\n--- Array Properties ---")

print("shape:", arr.shape)
# Output: (2, 3)

print("ndim:", arr.ndim)
# Output: 2

print("size:", arr.size)
# Output: 6

print("dtype:", arr.dtype)
# Output: float64

print("itemsize:", arr.itemsize)
# Output: 8

print("nbytes:", arr.nbytes)
# Output: 48

print("strides:", arr.strides)
# Output on most systems: (24, 8)

print("T:\n", arr.T)
# Output:
# [[1. 4.]
#  [2. 5.]
#  [3. 6.]]

print("real:\n", arr.real)
# Output:
# [[1. 2. 3.]
#  [4. 5. 6.]]
```

```
print("imag:\n", arr.imag)
# Output:
# [[0. 0. 0.]
#  [0. 0. 0.]]

print("flags:\n", arr.flags)
# Output (abbreviated):
#   C_CONTIGUOUS : True
#   F_CONTIGUOUS : False
#   WRITEABLE    : True
#   ... (other flags)

print("flat iterator:", list(arr.flat))
# Output in NumPy 2.x: [np.float64(1.0),
np.float64(2.0), np.float64(3.0), np.float64(4.0),
np.float64(5.0), np.float64(6.0)]
```

Example 2: Utility Functions and Inspection

```
import numpy as np

arr = np.array([[1.0, 2.0, 3.0],
                [4.0, 5.0, 6.0]])

print("--- Utility Functions ---")

print("numpy.shape(arr):", np.shape(arr))
# Output: (2, 3)

print("numpy.ndim(arr):", np.ndim(arr))
# Output: 2

print("numpy.size(arr):", np.size(arr))
# Output: 6

print("numpy.isfinite(arr):\n", np.isfinite(arr))
# Output:
# [[ True  True  True]
#  [ True  True  True]]
```

```
print("numpy.isnan(arr):\n", np.isnan(arr))
# Output:
# [[False False False]
#  [False False False]]

print("numpy.iscomplex(arr):\n", np.iscomplex(arr))
# Output:
# [[False False False]
#  [False False False]]

print("numpy.isreal(arr):\n", np.isreal(arr))
# Output:
# [[ True  True  True]
#  [ True  True  True]]

print("numpy.isfortran(arr):", np.isfortran(arr))
# Output: False

print("numpy.isscalar(arr):", np.isscalar(arr))
# Output: False

print("numpy.array_equal(arr, arr.copy()):",
np.array_equal(arr, arr.copy()))
# Output: True

print("\n--- numpy.info(arr) ---")
np.info(arr)
# Output: (Displays ndarray documentation)
```

Operations on Arrays

Once you understand how NumPy arrays are created and how their shapes, dtypes, and memory layout work, the next step is to explore what truly makes NumPy powerful: operations on arrays. Rather than operating on one element at a time in slow Python loops, NumPy applies mathematical and logical operations to entire arrays at once, using highly optimized vectorized routines implemented in C. This design allows you to write code that is both short and expressive while still achieving performance close to low-level compiled languages.

In this section, we will organize the vast collection of NumPy operations into families, such as arithmetic, aggregation, trigonometric functions, linear algebra, and more. Each family groups together functions that share a common purpose, making it easier to find the right tool for a given problem. Although SciPy and other libraries build on top of NumPy and provide more advanced routines, almost every scientific Python workflow starts with these core NumPy array operations.

Array Operation Families in NumPy

Table 11-9 below summarizes the main categories of operations available in NumPy. It is not an exhaustive list of every single function, but a practical map of the most important operation families and their typical use cases.

Table 11-9 Detailed summary of operation families and representative functions in NumPy

Arithmetic Operations

As mentioned earlier, NumPy performs all arithmetic operations in a highly optimized and vectorized way, executed at the speed of compiled C. These operations work element by element on entire arrays and fully support broadcasting, allowing operations between arrays of different shapes. The table below summarizes the main arithmetic tools available in NumPy, including the symbolic operators (+, -, *, /) and their corresponding universal functions (ufuncs), which offer greater flexibility, performance, and broadcast support.

Table 11-10 Elementwise arithmetic operations, operators, and corresponding ufuncs

The following example demonstrates the core elementwise arithmetic operations in NumPy. We use small 1D arrays to show how addition, subtraction, multiplication, division, floor division, modulo, and exponentiation work, both with operators and with their corresponding ufuncs:

```
import numpy as np

a = np.array([1, 2, 3], dtype=float)
b = np.array([10, 20, 30], dtype=float)

print("a:", a)
print("b:", b)
```

```
# Output:
# a: [1. 2. 3.]
# b: [10. 20. 30.]

print("\n--- Addition ---")
print("a + b:", a + b)
# Output: [11. 22. 33.]
print("np.add(a, b):", np.add(a, b))
# Output: [11. 22. 33.]

print("\n--- Subtraction ---")
print("a - b:", a - b)
# Output: [-9. -18. -27.]
print("np.subtract(a, b):", np.subtract(a, b))
# Output: [-9. -18. -27.]

print("\n--- Multiplication ---")
print("a * b:", a * b)
# Output: [10. 40. 90.]
print("np.multiply(a, b):", np.multiply(a, b))
# Output: [10. 40. 90.]

print("\n--- Division (true division) ---")
print("a / b:", a / b)
# Output: [0.1 0.1 0.1]
print("np.divide(a, b):", np.divide(a, b))
# Output: [0.1 0.1 0.1]

print("\n--- Floor division ---")
print("b // a:", b // a)
# Output: [10. 10. 10.]
print("np.floor_divide(b, a):", np.floor_divide(b, a))
# Output: [10. 10. 10.]

print("\n--- Remainder (modulo) ---")
print("b % a:", b % a)
# Output: [0. 0. 0.]
print("np.mod(b, a):", np.mod(b, a))
# Output: [0. 0. 0.]
```

```
print("np.remainder(b, a):", np.remainder(b, a))
# Output: [0. 0. 0.]

print("\n--- Power / Exponentiation ---")
print("a ** 2:", a ** 2)
# Output: [1. 4. 9.]
print("np.power(a, 2):", np.power(a, 2))
# Output: [1. 4. 9.]
```

Unary Arithmetic

Unary arithmetic operations operate on a single array and are indispensable for fast transformations, sign manipulation, magnitude calculations, exponential growth, and reciprocal calculations. The table below lists the main unary operations supported by NumPy, each implemented as a fast ufunc.

Table 11-11 Unary arithmetic operations, operators, and corresponding ufuncs

This example illustrates unary arithmetic operations that act on a single array: applying a sign, computing the absolute value, calculating the reciprocal, and evaluating the exponential. These operations are applied elementwise and are especially common in engineering formulas and signal transformations:

```
import numpy as np

a = np.array([-2.0, -1.0, 0.0, 1.0, 2.0])

print("a:", a)
# Output: [-2. -1.  0.  1.  2.]

print("\n--- Unary plus and minus ---")
print("+a:", +a)
# Output: [-2. -1.  0.  1.  2.]
print("-a:", -a)
# Output: [ 2.  1. -0. -1. -2.]

print("\n--- Absolute value ---")
print("np.abs(a):", np.abs(a))
# Output: [2. 1. 0. 1. 2.]
```

```
print("\n--- Reciprocal (1 / a) ---")
# Avoid dividing by zero; use a mask
mask = a != 0
reciprocal = np.empty_like(a)
reciprocal[mask] = np.reciprocal(a[mask])
reciprocal[~mask] = np.inf  # convention for 1/0
print("np.reciprocal(a) (with 1/0 -> inf):", reciprocal)
# Output: [-0.5        -1.
inf  1.          0.5        ]

print("\n--- Exponential ---")
print("np.exp(a):", np.exp(a))
# Output: [0.13533528 0.36787944 1.         2.71828183
7.3890561 ]
```

Rounding and Approximation

Rounding and approximation functions are essential in the transition between floating-point and integer domains or in preparing numerical data for engineering calculations, formatting, or comparison. NumPy offers rounding to the nearest number, truncation, or rounding up/down. The table below summarizes the most commonly used rounding tools.

Table 11-12 Common rounding and approximation functions in NumPy

The following code shows the main rounding and approximation functions provided by NumPy. These functions are useful when converting floating-point results to integers, formatting results, or enforcing discrete grid values in simulations and numerical models:

```
import numpy as np

values = np.array([1.2, 1.5, 1.8, -1.2, -1.5, -1.8])
print("values:", values)
# Output: [ 1.2  1.5  1.8 -1.2 -1.5 -1.8]

print("\n--- np.round (to nearest) ---")
print("np.round(values):", np.round(values))
# Output: [ 1.  2.  2. -1. -2. -2.]
```

```
print("\n--- np.floor (round down) ---")
print("np.floor(values):", np.floor(values))
# Output: [ 1.  1.  1. -2. -2. -2.]

print("\n--- np.ceil (round up) ---")
print("np.ceil(values):", np.ceil(values))
# Output: [ 2.  2.  2. -1. -1. -1.]

print("\n--- np.trunc (toward zero) ---")
print("np.trunc(values):", np.trunc(values))
# Output: [ 1.  1.  1. -1. -1. -1.]
```

Arithmetic Helpers for Numerical Stability

Certain numerical operations, such as calculating exp(x) − 1 or log(1 + x), can lose precision when x is very small. To avoid this, NumPy includes specialized "stable math" helper functions that maintain high precision even under extreme numerical conditions. These functions are crucial in scientific simulations and optimization routines where numerical stability is essential. Table 11-13 below highlights these high-precision arithmetic helper functions.

Table 11-13 Arithmetic helper functions for numerical stability in NumPy

Some operations, such as exp(x) - 1 or log(1 + x), can suffer from significant rounding errors when x is very small. NumPy provides numerically stable versions of these expressions (expm1 and log1p), along with helpers for rounding (rint) and computing roots. The example below shows how to use these functions in practice:

```
import numpy as np

x = np.array([1e-10, 1e-5, 1e-2])

print("x:", x)
# Output: [1.e-10 1.e-05 1.e-02]

print("\n--- exp(x) - 1 vs np.expm1(x) ---")
print("np.exp(x) - 1:", np.exp(x) - 1)
# Output (approx): [1.00000008e-10 1.00000500e-05
1.00501671e-02]
```

```
print("np.expm1(x):   ", np.expm1(x))
# Output (approx): [1.00000000e-10 1.00000500e-05
1.00501671e-02]

print("\n--- log(1 + x) vs np.log1p(x) ---")
print("np.log(1 + x):", np.log(1 + x))
# Output (approx): [9.99999972e-11 9.99995000e-06
9.95033085e-03]
print("np.log1p(x):  ", np.log1p(x))
# Output (approx): [1.00000000e-10 9.99995000e-06
9.95033085e-03]

print("\n--- np.rint (round to nearest integer, keep
float) ---")
values = np.array([1.2, 1.5, 1.8, -1.2, -1.5, -1.8])
print("np.rint(values):", np.rint(values))
# Output: [ 1.  2.  2. -1. -2. -2.]

print("\n--- Square root and cube root ---")
data = np.array([1.0, 8.0, 27.0])
print("np.sqrt(data):", np.sqrt(data))
# Output: [1. 2.82842712 5.19615242]
print("np.cbrt(data):", np.cbrt(data))
# Output: [1. 2. 3.]
```

Binary Arithmetic with Broadcasting

Broadcasting is one of NumPy's most powerful features, allowing operations between arrays of different shapes without the need for manual replication. This capability enables concise, elegant, and efficient mathematical expressions that adapt to the shape of the operands. The examples below illustrate how NumPy automatically aligns dimensions during arithmetic operations:

Table 11-14 Operations on complex-valued arrays

```
import numpy as np

a = np.array([1.0, 2.0, 3.0])
print("a:", a)
```

```
# Output: [1. 2. 3.]

print("\n--- Add scalar (broadcasting) ---")
print("a + 5:", a + 5)
# Output: [6. 7. 8.]

print("\n--- Matrix + row vector (broadcasting along
rows) ---")
M = np.array([[1.0, 2.0, 3.0],
              [4.0, 5.0, 6.0]])
row = np.array([10.0, 20.0, 30.0])
print("M:\n", M)
# Output:
# [[1. 2. 3.]
#  [4. 5. 6.]]
print("row:", row)
# Output: [10. 20. 30.]

print("M + row:\n", M + row)
# Output:
# [[11. 22. 33.]
#  [14. 25. 36.]]

print("\n--- Elementwise square ---")
print("a ** 2:", a ** 2)
# Output: [1. 4. 9.]
```

Complex Number Arithmetic

NumPy treats complex numbers as numeric types, which facilitates its applications in signal processing, control systems, and any domain that uses magnitude/phase representations. The table below summarizes the essential operations for extracting real/imaginary components, calculating magnitudes, retrieving phase angles, and working with conjugates, all efficiently executed on integer arrays.

Table 11-15 Operations for working with complex-valued NumPy arrays

The example below shows how to access the real and imaginary parts of a complex array, compute the complex conjugate, obtain the magnitude, and calculate the phase angle:

```
import numpy as np

c = np.array([1 + 2j, 3 + 4j])

print("c:", c)
# Output: [1.+2.j 3.+4.j]

print("\n--- Real and imaginary parts ---")
print("c.real:", c.real)
# Output: [1. 3.]
print("c.imag:", c.imag)
# Output: [2. 4.]

print("\n--- Complex conjugate ---")
print("np.conj(c):", np.conj(c))
# Output: [1.-2.j 3.-4.j]

print("\n--- Magnitude (absolute value) ---")
print("np.abs(c):", np.abs(c))
# Output: [2.23606798 5.        ]
# (sqrt(1^2 + 2^2) = sqrt(5), sqrt(3^2 + 4^2) = 5)

print("\n--- Phase angle (radians) ---")
print("np.angle(c):", np.angle(c))
# Output (approx): [1.10714872 0.92729522]
```

Aggregation and Reduction

Aggregation and reduction operations are at the core of numerical analysis in NumPy. They take one or more axes of an array and collapse them into summary values, such as sums, products, means, minima, or maxima. Instead of writing explicit Python loops, NumPy performs these reductions in optimized C code, making them both much faster and more concise. Nearly every data analysis or scientific workflow uses these tools to compute statistics over time, features, samples, or spatial dimensions. The functions in this family support options such as axis, keepdims, and dtype to control which dimensions are

reduced, whether to preserve singleton dimensions for broadcasting, and which numeric precision to use—features that become especially important when working with large arrays, high-precision calculations, or multidimensional data in NumPy.

Table 11-16 below shows the main aggregation and reduction functions in NumPy. They are used to summarize array values, such as summing, calculating averages, finding minimums and maximums, and much more.

Table 11-16 Aggregation and reduction functions in NumPy

The following code demonstrates, in practice, how each of these functions works and what result they produce:

```
import numpy as np

# Base array for all examples
a = np.array([
    [1, 2, 3],
    [4, 5, 6]
], dtype=float)

print("Array a:")
print(a)
# Output:
# [[1. 2. 3.]
#  [4. 5. 6.]]

# ------------------------------------------
# Total sum and sum along axes
# ------------------------------------------
print("\n--- Sum ---")
print("np.sum(a):", np.sum(a))
# Output: 21.0

print("np.sum(a, axis=0):", np.sum(a, axis=0))  #
column-wise
# Output: [5. 7. 9.]
```

```
print("np.sum(a, axis=1):", np.sum(a, axis=1))  # row-
wise
# Output: [ 6. 15.]

# ----------------------------------------
# Product and product along axes
# ----------------------------------------
print("\n--- Product ---")
print("np.prod(a):", np.prod(a))
# Output: 720.0

print("np.prod(a, axis=0):", np.prod(a, axis=0))
# Output: [ 4. 10. 18.]

print("np.prod(a, axis=1):", np.prod(a, axis=1))
# Output: [  6. 120.]

# ----------------------------------------
# Mean, standard deviation, variance
# ----------------------------------------
print("\n--- Mean / Std / Var ---")
print("np.mean(a):", np.mean(a))
# Output: 3.5

print("np.std(a):", np.std(a))
# Output: 1.707825127659933

print("np.var(a):", np.var(a))
# Output: 2.9166666666666665

# ----------------------------------------
# Min, max, argmin, argmax
# ----------------------------------------
print("\n--- Min / Max and Argmin / Argmax ---")
print("np.min(a):", np.min(a))
# Output: 1.0

print("np.max(a):", np.max(a))
# Output: 6.0
```

```
print("np.argmin(a):", np.argmin(a))  # flattened index
# Output: 0

print("np.argmax(a):", np.argmax(a))
# Output: 5

# ----------------------------------------
# Cumulative sum and cumulative product
# ----------------------------------------
print("\n--- Cumulative Sum / Product ---")
print("np.cumsum(a):", np.cumsum(a))
# Output: [ 1.  3.  6. 10. 15. 21.]

print("np.cumprod(a):", np.cumprod(a))
# Output: [  1.   2.   6.  24. 120. 720.]

# ----------------------------------------
# Successive differences
# ----------------------------------------
print("\n--- Difference (np.diff) ---")
print("np.diff(a):")
print(np.diff(a))
# Output:
# [[1. 1.]
#  [1. 1.]]

# ----------------------------------------
# Trace of a matrix
# ----------------------------------------
print("\n--- Trace ---")
print("np.trace(a):", np.trace(a))
# Output: 6.0

# ----------------------------------------
# Median and quantiles
# ----------------------------------------
print("\n--- Median / Quantiles ---")
print("np.median(a):", np.median(a))
# Output: 3.5
```

```
print("np.quantile(a, 0.25):", np.quantile(a, 0.25))
# Output: 2.25

print("np.quantile(a, 0.75):", np.quantile(a, 0.75))
# Output: 4.75

# ----------------------------------------
# Logical reductions
# ----------------------------------------
b = np.array([True, True, False])

print("\n--- Logical Reductions ---")
print("np.all(b):", np.all(b))
# Output: False

print("np.any(b):", np.any(b))
# Output: True
```

Trigonometric, Hyperbolic, Exponential, and Logarithmic Functions

NumPy's trigonometric, hyperbolic, exponential, and logarithmic functions are essential in fields such as physics, engineering, and signal processing. They allow you to model periodic phenomena, handle exponential growth, apply scaling, calculate rates, and perform mathematical transformations in a vectorized and highly efficient way. All these operations are performed element by element.

Table 11-17 below summarizes the main commands of these families in NumPy.

Table 11-17 Trigonometric, hyperbolic, exponential, and logarithmic functions in NumPy

The code below shows how to perform the calculations:

```
import numpy as np

a = np.array([0.0, 0.5, 1.0])

print("Array a:", a)
# Output: [0.  0.5 1. ]
```

```
# ------------------------------------------
# Trigonometric functions
# ------------------------------------------
print("\n--- Trigonometric ---")
print("np.sin(a):", np.sin(a))
# Output: [0.         0.47942554 0.84147098]

print("np.cos(a):", np.cos(a))
# Output: [1.         0.87758256 0.54030231]

print("np.tan(a):", np.tan(a))
# Output: [0.         0.54630249 1.55740772]

# ------------------------------------------
# Hyperbolic functions
# ------------------------------------------
print("\n--- Hyperbolic ---")
print("np.sinh(a):", np.sinh(a))
# Output: [0.         0.52109531 1.17520119]

print("np.cosh(a):", np.cosh(a))
# Output: [1.         1.12762597 1.54308063]

print("np.tanh(a):", np.tanh(a))
# Output: [0.         0.46211716 0.76159416]

# ------------------------------------------
# Exponential functions
# ------------------------------------------
print("\n--- Exponential ---")
print("np.exp(a):", np.exp(a))
# Output: [1.         1.64872127 2.71828183]

print("np.expm1(a):", np.expm1(a))
# Output: [0.         0.64872127 1.71828183]

# ------------------------------------------
# Logarithmic functions
# ------------------------------------------
```

```
b = np.array([1.0, 10.0, 100.0])

print("\n--- Logarithmic ---")
print("np.log(b):", np.log(b))
# Output: [0.         2.30258509 4.60517019]

print("np.log10(b):", np.log10(b))
# Output: [0. 1. 2.]

print("np.log2(b):", np.log2(b))
# Output: [0.         3.32192809 6.64385619]

print("np.log1p(a):", np.log1p(a))
# Output: [0.         0.40546511 0.69314718]

# ------------------------------------------
# Power and roots
# ------------------------------------------
print("\n--- Power / Root ---")
print("np.sqrt(b):", np.sqrt(b))
# Output: [ 1.          3.16227766 10.        ]

print("np.cbrt(b):", np.cbrt(b))
# Output: [1.         2.15443469 4.64158883]

print("np.square(a):", np.square(a))
# Output: [0.   0.25 1.  ]

print("np.power(a, 2):", np.power(a, 2))
# Output: [0.   0.25 1.  ]
```

Logical, Boolean, Comparison, Sorting, Searching, and Set Operations

NumPy offers a powerful set of functions for testing conditions, comparing values, building boolean masks, and applying filters to multidimensional arrays. These operations are fundamental in data analysis and conditional selection tasks. Furthermore, NumPy includes multiple functions for sorting, searching, and set operations, allowing you to locate indexes, sort results, find unique values, and combine or compare numerical collections extremely

efficiently. These tools replace manual loops and provide fast and consistent mechanisms for manipulating large data collections in NumPy.

This table brings together the main comparison, logical, sorting, and set operations, providing a quick overview of the most commonly used functions for efficient array analysis and manipulation.

Table 11-18 Comparison, logical, sorting, searching, and set operations in NumPy

This example code demonstrates how to use logical operations, comparisons, conditional selection, sorting, and set functions:

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([2, 2, 1, 5])

print("Array a:", a)    # Output:  a: [1 2 3 4]
print("Array b:", b)    # Output:  b: [2 2 1 5]

# ----------------------------------------
# Comparison and Boolean operations
# ----------------------------------------

print("\n--- Comparison ---")
print("a == b:", a == b) # Output: [False  True False
False]

print("a > b:", a > b)   # Output: [False False  True
False]

print("np.greater(a, b):", np.greater(a, b)) # Output:
[False False  True False]
print("\n--- Logical Operations ---")
cond1 = a > 1
cond2 = b < 5

print("cond1:", cond1) # Output:
[False  True  True  True]
```

```
print("cond2:", cond2) # Output: [ True  True  True
False]

print("np.logical_and(cond1, cond2):",
np.logical_and(cond1, cond2))
# Output: [False  True  True False]

print("np.logical_or(cond1, cond2):",
np.logical_or(cond1, cond2))
# Output: [ True  True  True  True]

print("np.logical_not(cond1):", np.logical_not(cond1))
# Output: [ True False False False]
print("\n--- Conditional Selection (np.where) ---")
print("np.where(a > b, a, b):", np.where(a > b, a,
b))   # Output: [2 2 3 5]

# ------------------------------------------
# all() / any()
# ------------------------------------------

print("\n--- All / Any ---")
print("np.all(a > 0):", np.all(a > 0))      # Output:
True

print("np.any(a == 3):", np.any(a == 3))    # Output:
True

# ------------------------------------------
# Sorting and Searching
# ------------------------------------------
print("\n--- Sorting ---")
print("np.sort(a):", np.sort(a))            # Output: [1
2 3 4]

print("np.argsort(b):", np.argsort(b))      # Output: [2
0 1 3]

print("\n--- Partition (top-k behavior) ---")
print("np.partition(a, 1):", np.partition(a, 1))
```

```
# Output: [1 2 3 4]   # 1st smallest in place; others
not guaranteed sorted

print("\n--- Searchsorted ---")
c = np.array([10, 20, 30, 40])
print("np.searchsorted(c, 25):", np.searchsorted(c,
25))    # Output: 2

# ----------------------------------------
# Set operations
# ----------------------------------------
print("\n--- Set Operations ---")
print("np.unique(a):",
np.unique(a))                          # Output: [1 2 3
4]

print("np.intersect1d(a, b):", np.intersect1d(a,
b))         # Output: [1 2]

print("np.union1d(a, b):", np.union1d(a,
b))                # Output: [1 2 3 4 5]

print("np.setdiff1d(a, b):", np.setdiff1d(a,
b))             # Output: [3 4]

print("np.isin(a, b):", np.isin(a,
b))                       # Output: [ True  True False
False]
```

Linear Algebra Operations

NumPy also offers a robust set of linear algebra functions, widely used in scientific computing, machine learning, optimization, physical simulations, and signal processing. These operations include matrix and vector multiplication, norms, determinants, linear systems, eigenvalues, matrix decompositions, and fundamental transformations for numerical models. Most of these functions are available in the numpy.linalg module, which uses optimized routines from low-level libraries such as BLAS and LAPACK, ensuring high performance and accuracy even in complex calculations.

BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage) are essential libraries for high-performance numerical computing, originally written in Fortran. BLAS serves as the foundation, providing highly optimized low-level routines for fundamental linear algebra operations such as matrix–vector and matrix–matrix products. Building upon this foundation, LAPACK offers higher-level algorithms capable of solving complex problems, including linear systems, least-squares regressions, matrix decompositions, and the calculation of eigenvalues and eigenvectors.

Table 11-19 presents the main linear algebra commands in NumPy.

Table 11-19 Linear algebra operations and functions in NumPy

The following example demonstrates the main linear algebra operations in NumPy, including matrix multiplication, inner and outer products, norms, determinants, inversion, solving linear systems, eigenvalues/eigenvectors, decompositions, and other essential features of the np.linalg module:

```
import numpy as np

# Base matrices
A = np.array([[1., 2.],
              [3., 4.]])
B = np.array([[5., 6.],
              [7., 8.]])
v = np.array([1., 2.])

print("Matrix A:\n", A)
print("Matrix B:\n", B)
print("Vector v:", v)
# Output:
# Matrix A:
# [[1. 2.]
#  [3. 4.]]
# Matrix B:
# [[5. 6.]
#  [7. 8.]]
# Vector v: [1. 2.]
```

```
# ----------------------------------------
# Matrix multiplication
# ----------------------------------------
print("\n--- Matrix Multiplication ---")
print("A @ B:\n", A @ B)
# Output:
# [[19. 22.]
#  [43. 50.]]

print("np.matmul(A, B):\n", np.matmul(A, B))
# Output (same as above)

# ----------------------------------------
# Dot products
# ----------------------------------------
print("\n--- Dot Products ---")
print("np.dot(v, v):", np.dot(v, v))
# Output: 5.0

print("np.inner(v, v):", np.inner(v, v))
# Output: 5.0

print("np.outer(v, v):\n", np.outer(v, v))
# Output:
# [[1. 2.]
#  [2. 4.]]

# ----------------------------------------
# Norm
# ----------------------------------------
print("\n--- Norm ---")
print("np.linalg.norm(v):", np.linalg.norm(v))
# Output: 2.23606797749979

# ----------------------------------------
# Determinant, Inverse
# ----------------------------------------
print("\n--- Determinant and Inverse ---")
print("np.linalg.det(A):", np.linalg.det(A))
```

```
# Output: -2.0000000000000004

print("np.linalg.inv(A):\n", np.linalg.inv(A))
# Output:
# [[-2.   1. ]
#  [ 1.5 -0.5]]

# ------------------------------------------
# Solving linear systems Ax = b
# ------------------------------------------
b_vec = np.array([1., 2.])
print("\n--- Solve Ax = b ---")
print("Solution x:", np.linalg.solve(A, b_vec))
# Output: [0.  0.5]

# ------------------------------------------
# Eigenvalues and eigenvectors
# ------------------------------------------
print("\n--- Eigenvalues and Eigenvectors ---")
eigvals, eigvecs = np.linalg.eig(A)
print("Eigenvalues:", eigvals)
# Output approx: [-0.37228132  5.37228132]

print("Eigenvectors:\n", eigvecs)
# Output approx:
# [[-0.82456484 -0.41597356]
#  [ 0.56576746 -0.90937671]]

# ------------------------------------------
# Singular Value Decomposition (SVD)
# ------------------------------------------
print("\n--- SVD ---")
U, S, Vt = np.linalg.svd(A)
print("U:\n", U)
# Typical output (singular-vector signs may vary):
# U:
# [[-0.40455358 -0.9145143 ]
#  [-0.9145143   0.40455358]]
print("S:", S)
```

```
# Output approx: [5.4649857  0.36596619]
print("Vt:\n", Vt)
# Vt:
# [[-0.57604844 -0.81741556]
#  [ 0.81741556 -0.57604844]]

# ------------------------------------------
# Rank and Trace
# ------------------------------------------
print("\n--- Rank and Trace ---")
print("np.linalg.matrix_rank(A):",
np.linalg.matrix_rank(A))
# Output: 2

print("np.trace(A):", np.trace(A))
# Output: 5.0

# ------------------------------------------
# QR decomposition
# ------------------------------------------
print("\n--- QR Decomposition ---")
Q, R = np.linalg.qr(A)
print("Q:\n", Q)
print("R:\n", R)
# Typical output (column signs may vary):
# Q:
# [[-0.31622777 -0.9486833 ]
#  [-0.9486833   0.31622777]]
# R:
# [[-3.16227766 -4.42718872]
#  [ 0.         -0.63245553]]

# ------------------------------------------
# Cholesky (requires symmetric positive-definite matrix)
# ------------------------------------------
C = np.array([[4., 2.],
              [2., 2.]])
print("\nMatrix C:\n", C)
print("np.linalg.cholesky(C):\n", np.linalg.cholesky(C))
```

```
# Output:
# [[2.          0.          ]
#  [1.          1.          ]]
```

Array Manipulation

Array manipulation functions allow you to reshape, reorganize, combine, split, and transform arrays without changing their underlying data. These operations are essential in data preprocessing, scientific simulations, and any task that involves changing the structure of a dataset. NumPy provides a wide set of tools for reshaping arrays, flattening them, adding or removing dimensions, stacking multiple arrays together, or dividing them into smaller parts. Understanding these transformations is crucial, as most numerical workflows require adjusting array shapes to match the expected format of algorithms and mathematical operations.

The following table summarizes the main operations for transforming, rearranging, and combining arrays in NumPy, allowing you to change shapes, add or remove dimensions, stack, concatenate, split, and replicate data in a practical and efficient way.

Table 11-20 Array reshaping, reorganization, and combination operations in NumPy

The following example code demonstrates the main shape and structure manipulation operations in NumPy arrays, including reshaping, flattening, transposition, dimension expansion and reduction, stacking, concatenation, division, repetition, and pattern replication:

```
import numpy as np

a = np.array([[1, 2, 3],
              [4, 5, 6]])

print("Array a:")
print(a)
# Output:
# [[1 2 3]
#  [4 5 6]]

# ----------------------------------------
```

```
# Reshape
# ----------------------------------------
print("\n--- Reshape ---")
reshaped = np.reshape(a, (3, 2))
print("reshape (3,2):\n", reshaped)
# Output:
# [[1 2]
#  [3 4]
#  [5 6]]

# ----------------------------------------
# Flatten and ravel
# ----------------------------------------
print("\n--- Flatten / Ravel ---")
print("a.flatten():", a.flatten())
# Output: [1 2 3 4 5 6]

print("a.ravel():", a.ravel())
# Output: [1 2 3 4 5 6]

# ----------------------------------------
# Transpose and swap axes
# ----------------------------------------
print("\n--- Transpose / Swapaxes ---")
print("np.transpose(a):\n", np.transpose(a))
# Output:
# [[1 4]
#  [2 5]
#  [3 6]]

print("np.swapaxes(a, 0, 1):\n", np.swapaxes(a, 0, 1))
# Output (same as transpose)

# ----------------------------------------
# Expand dims / squeeze
# ----------------------------------------
print("\n--- Expand / Squeeze ---")
b = np.expand_dims(a, axis=0)
print("expand_dims (axis=0):\n", b)
```

```
# Output:
# [[[1 2 3]
#   [4 5 6]]]

print("squeeze:\n", np.squeeze(b))
# Output:
# [[1 2 3]
#  [4 5 6]]

# ----------------------------------------
# Stacking
# ----------------------------------------
print("\n--- Stacking ---")
c = np.array([[7, 8, 9]])

print("np.vstack([a, c]):\n", np.vstack([a, c]))
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]

print("np.hstack([a, a]):\n", np.hstack([a, a]))
# Output:
# [[1 2 3 1 2 3]
#  [4 5 6 4 5 6]]

# ----------------------------------------
# Concatenate
# ----------------------------------------
print("\n--- Concatenate ---")
print("np.concatenate([a, c], axis=0):\n",
np.concatenate([a, c], axis=0))
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]

# ----------------------------------------
# Split
```

```
# ----------------------------------------
print("\n--- ARRAY SPLIT (unequal parts allowed) ---")
parts = np.array_split(a, 2, axis=1)
print("Part 1:\n", parts[0])
print("Part 2:\n", parts[1])
# Output:
# Part 1:
# [[1 2]
#  [4 5]]
# Part 2:
# [[3]
#  [6]]
# ----------------------------------------
# Repeat and tile
# ----------------------------------------
print("\n--- Repeat / Tile ---")
print("np.repeat([1, 2, 3], 2):", np.repeat([1, 2, 3],
2))
# Output: [1 1 2 2 3 3]

print("np.tile([1, 2], 3):", np.tile([1, 2], 3))
# Output: [1 2 1 2 1 2]
```

NumPy: Common User Errors and Best Practices

NumPy is designed for efficient numerical computation, but shape mismatches, indexing mistakes, casting, and memory limits can still cause errors. The following sections summarize common problems and practical ways to avoid them.

Common User Errors in NumPy

Broadcasting and Shape Compatibility Errors

Broadcasting is the process of aligning arrays with different shapes and sizes to perform elementwise operations. NumPy automatically broadcasts arrays to make them compatible for operations.

Ensuring shape compatibility is crucial when performing operations with NumPy arrays. Broadcasting errors occur when attempting to operate on

arrays with incompatible shapes. To avoid these errors, NumPy requires specific conditions to be met for elementwise operations.

Similarly, matrix operations, such as multiplication, demand compatible dimensions. For instance, in matrix multiplication, the number of columns in the first matrix must match the number of rows in the second. Failing to meet these conditions results in errors, as demonstrated in the following examples.

Broadcasting works only when the arrays' dimensions are compatible starting from the trailing dimension; their sizes must either match or be equal to 1. If the shapes are incompatible, NumPy raises a ValueError, as shown in the code snippet below:

```
import numpy as np
# Attempting to use incompatible shapes
a = np.array([1, 2, 3])
b = np.array([[1, 2], [3, 4]])
result = a + b  # ValueError: operands could not be
broadcast together
```

Understanding the broadcasting rules and ensuring shape compatibility are necessary to prevent errors and write more robust NumPy code.

Similarly to the example above, when performing elementwise addition, subtraction, multiplication, or division, the two arrays must either have the same shape or be broadcastable. If they are not, NumPy raises a ValueError due to shape incompatibility.

The following example demonstrates this issue. In matrix d, multiplication occurs between a square matrix a and a rectangular matrix b, resulting in a rectangular matrix—this is a valid mathematical operation.

However, in the last operation (e), multiplication is attempted between a rectangular matrix b and a square matrix a, which raises a ValueError due to incompatible dimensions:

```
import numpy as np
# Matrix multiplication with incompatible dimensions.
a = np.array([[1, 2], [3, 4]])
b = np.array([[1, 2, 3], [4, 5, 6]])
# Matrix multiplication with compatible dimensions.
c = np.dot(a, a)  # Works: (2,2) and (2,2) -> (2,2)
d = np.dot(a, b)  # Works: (2,2) and (2,3) -> (2,3)
```

```
# Matrix multiplication with incompatible dimensions.
e = np.dot(b, a)  # ValueError: shapes (2,3) and (2,2)
not aligned: 3 (dim 1) != 2 (dim 0)
```

Understanding matrix multiplication rules and ensuring proper shape alignment make it possible to prevent such errors and write more robust NumPy code.

Indexing Errors

For a basic out-of-range indexing example, see the subsection "Indexing Errors" under the section “Accessing Elements in NumPy Arrays.” The remaining examples focus on errors not covered there.

Type Conversion Issues

NumPy arrays use a fixed data type. When an element of a different type is assigned, it may be implicitly converted through truncation or casting, potentially resulting in data loss during the operation.

```
import numpy as np
# Unexpected type conversion
a = np.array([1, 2, 3])
a[0] = 3.1415  # The float will be truncated to 3
print(a[0]) # Output: 3
```

Memory Errors

```
An ArrayMemoryError occurs when NumPy cannot allocate
enough memory for the requested array.
import numpy as np
# Creating arrays too large for memory
huge_array = np.ones((1000000, 1000000))  # May cause
MemoryError
```

The maximum feasible array size depends on dtype, available RAM, operating system overhead, and other processes. For example, a 10,000 × 10,000 float64 array alone requires about 800 MB, before temporary arrays and other overhead are considered.

View vs. Copy

In NumPy, slicing an array does not create a new independent array; it creates a view, which is a reference to the original data. Modifying a view affects the original array. However, using .copy() explicitly creates a new independent copy, preventing unintentional modifications.

```
import numpy as np
# Create a 2D NumPy array
a = np.array([[1, 2, 3], [4, 5, 6]])

# Create a slice of 'a' from column 1 to 3 (excluding
column 0)
# This creates a VIEW of 'a', not a copy
b = a[:, 1:3]

# Modify the value at position (0, 0) in 'b'
# Since 'b' is a view of 'a', this modification also
affects 'a'
b[0, 0] = 10

# Print the modified 'a' to demonstrate the effect
print("Modified 'a':")
print(a)

# Create a COPY of the slice of 'a' from column 1 to 3
c = a[:, 1:3].copy()

# Modify the value at position (0, 0) in 'c'
# Since 'c' is a copy of 'a', this modification does not
affect 'a'
c[0, 0] = 20

# Print 'a' and 'c' to demonstrate the difference
print("\n'a' remains unchanged:")
print(a)
print("\nModified 'c':")
print(c)
# Output:
# Modified 'a':
# [[ 1 10  3]
```

```
#  [ 4  5  6]]
# 'a' remains unchanged:
# [[ 1 10  3]
#  [ 4  5  6]]
# Modified 'c':
# [[20  3]
#  [ 5  6]]
```

Best Practices for NumPy

These practices improve performance, memory use, readability, and reliability.

Use Vectorized Operations

Vectorized operations are much faster than elementwise operations using Python loops. The next example code compares the performance of loop-based and vectorized operations in NumPy. It creates a large array of random numbers and measures the time taken to multiply each element by 2 using both methods.

The loop-based approach uses a traditional for loop to iterate over the array elements, while the vectorized approach leverages NumPy's ability to perform operations on entire arrays at once:

```
import numpy as np
import time

arr = np.random.rand(1000000)

# Measure the time taken by loop-based operation
start_time = time.time()   # Record the start time
for i in range(len(arr)):  # Iterate over the array
elements
    arr[i] = arr[i] * 2    # Multiply each element by 2
print(f"Loop time: {time.time() - start_time:.6f}
seconds")

# Create another large array of random numbers
arr = np.random.rand(1000000)

# Measure the time taken by vectorized operation
start_time = time.time()
```

```
arr = arr * 2
print(f"Vectorized time: {time.time() - start_time:.6f}
seconds")
Example output (timings vary by system):
# Loop time: 0.118138 seconds
# Vectorized time: 0.001505 seconds
```

Manage Memory Carefully

When working with NumPy, cautious memory management is essential for effective data processing. Large numerical operations can quickly consume available RAM, particularly when handling datasets of significant size.

It is possible to drastically reduce memory consumption when full precision is not required by selecting appropriate data types, such as np.float32 instead of np.float64.

Use Appropriate Data Types

```
import numpy as np
# np.float64 (default) vs. np.float32 uses half the
memory.
large_array_float64 = np.ones((10000, 10000),
dtype=np.float64)
print(f"Memory usage (float64):
{large_array_float64.nbytes / 1e6:.2f} MB")

large_array_float32 = np.ones((10000, 10000),
dtype=np.float32)
print(f"Memory usage (float32):
{large_array_float32.nbytes / 1e6:.2f} MB")

# Output:
# Memory usage (float64): 800.00 MB
# Memory usage (float32): 400.00 MB
```

To avoid issues, it's essential to specify the attribute dtypes explicitly when reading data, performing operations, or creating arrays—especially when reading data or performing operations that might change types.

```
import numpy as np
```

```
# Create an array with inferred dtype (an integer list
usually produces the platform's default integer dtype)
implicit_array = np.array([1, 2, 3, 4, 5])
print("Implicit dtype:", implicit_array.dtype)

# Create an array with explicit dtype (int32)
explicit_array = np.array([1, 2, 3, 4, 5],
dtype=np.int32)
print("Explicit dtype:", explicit_array.dtype)

# Demonstrate the difference in memory usage
print("Implicit array memory usage:",
implicit_array.nbytes, "bytes")
print("Explicit array memory usage:",
explicit_array.nbytes, "bytes")
Typical output on a 64-bit system is shown below.
# Output:
# Implicit dtype: int64
# Explicit dtype: int32
# Implicit array memory usage: 40 bytes
# Explicit array memory usage: 20 bytes
```

Use Memory-Mapped Arrays

A memory-mapped array (np.memmap) stores array data in a file on disk and accesses portions of it as needed. It is useful for datasets that are too large to load fully into RAM.

The np.memmap constructor provides parameters for the filename, data type, mode, shape, and byte offset.

Here are the parameters for np.memmap.

Table 11-21 Parameters of the np.memmap constructor

The next code creates a large array using NumPy's memory-mapped array functionality.

It first specifies a filename ("large_array.dat") to store the array on disk. Then, it creates a memory-mapped array with a shape of 10,000 × 10,000 elements, using 32-bit floating-point numbers. The mode="w+" argument allows the array to be both read and written.

The array is then filled with random values using np.random.rand:

```
# Using memory-mapped arrays for large datasets.
import numpy as np
filename = "large_array.dat"
memmap_array = np.memmap(filename, dtype=np.float32,
mode="w+", shape=(10000, 10000))
memmap_array[:] = np.random.rand(10000, 10000)  #
Populate the array
print(f"Memory-mapped array created: {filename}")
# Output: Memory-mapped array created: large_array.dat
```

Delete Arrays When No Longer Needed

Large arrays retain memory while references to them remain. Using del removes a reference, but Python's memory allocator and the operating system determine when memory is returned, so process memory may not decrease immediately.

The code below illustrates the effect. Measured values vary by platform, allocator, and other activities in the process.

It starts by defining a function to measure memory usage in megabytes using the psutil library. The psutil library is an acronym for Python System Utilities, a cross-platform library for retrieving information about running processes and system utilization. In this code psutil is specifically used to track memory consumption.

The script first measures baseline memory consumption; then creates a large NumPy array with 10 million elements of type float64, requiring approximately 76 MB; and records the new memory usage.

After del removes the array reference, the script measures resident memory again. The reported change is an observation for that run, not a guarantee that the operating system immediately reclaims the memory.

The import os statement in the code is used for the os.getpid() function, which gets the current process ID for the running Python script. This process ID is required by the psutil.Process() function to identify which process's memory usage to measure:

```
# Deleting arrays when no longer needed
import numpy as np
import os
```

```
import psutil

def get_memory_usage():
    """Return the current memory usage in MB."""
    process = psutil.Process(os.getpid())
    return process.memory_info().rss / (1024 * 1024)

print("DELETING ARRAYS WHEN NOT NEEDED")

# Check initial memory
start_mem = get_memory_usage()
print(f"Initial memory: {start_mem:.2f} MB")

# Create one large array (about 76MB for float64)
large_array = np.ones(10000000, dtype=np.float64)
mem_with_array = get_memory_usage()
print(f"Memory with large array: {mem_with_array:.2f}
MB")
print(f"Memory used by array: {mem_with_array -
start_mem:.2f} MB")

# Delete the array
del large_array
mem_after_delete = get_memory_usage()
print(f"Memory after deletion: {mem_after_delete:.2f}
MB")
print(f"Memory freed: {mem_with_array -
mem_after_delete:.2f} MB")
```

Preallocate Arrays

NumPy array preallocation is a key optimization technique that significantly improves performance and memory management. Rather than dynamically growing arrays or converting Python lists to arrays after filling them, preallocation reserves memory space in advance using functions such as "np.zeros()", "np.empty()", or "np.full()".

This approach eliminates costly repeated memory allocations, prevents memory fragmentation, and dramatically improves execution speed, especially for large datasets.

The next code compares two methods of building NumPy arrays to demonstrate why preallocation is critical for performance.

The first method creates an empty array and repeatedly uses np.append() to add elements one at a time, forcing NumPy to allocate new memory and copy all existing data with each addition.

The second method preallocates a zeros array of the required size of 100,000 elements and then simply assigns values to each position without any reallocation:

```
import numpy as np
import time
# Compare array growing vs preallocation
size = 100000

# Growing array (slow)
t1 = time.time()
a = np.array([])
for i in range(size): a = np.append(a, i)
t1 = time.time() - t1

# Preallocated array (fast)
t2 = time.time()
b = np.zeros(size)
for i in range(size): b[i] = i
t2 = time.time() - t2

print(f"Growing: {t1:.4f}s | Preallocated: {t2:.4f}s |
Speedup: {t1/max(t2, 0.0001):.0f}x")
# Output example: Growing: 1.3679s | Preallocated:
0.0055s | Speedup: 248x
```

Use Built-In Functions

NumPy's built-in functions are optimized and usually faster than equivalent Python code; they are implemented in optimized C code rather than interpreted Python.

The code below compares the performance of Python's built-in sum() function and NumPy's optimized np.sum() function when calculating the sum of a large array of 5 million random numbers. It measures the execution time of

both functions and prints the results, highlighting the time difference and speedup factor achieved by using NumPy's optimized function:

```
import numpy as np
import time

data = np.random.rand(5000000)

# Time Python's built-in sum() function
t1 = time.time(); sum(data); t1 = time.time() - t1
# Time NumPy's optimized np.sum() function
t2 = time.time(); np.sum(data); t2 = time.time() - t2

# Print results showing the time difference and speedup
factor
print(f"Python: {t1:.4f}s | NumPy: {t2:.4f}s | Speedup:
{t1/t2:.1f}x")
# Output example: Python: 0.1819s | NumPy: 0.0030s |
Speedup: 60.6x
```

Keep Arrays Contiguous

Contiguous arrays can improve performance. A contiguous array in NumPy is an array where the elements are stored in a single, continuous block of memory. This allows for faster access and computation because data is efficiently read in sequence. Use np.ascontiguousarray() to ensure your arrays are contiguous.

The next code demonstrates the importance of keeping NumPy arrays contiguous using np.ascontiguousarray():

```
import numpy as np

# Create a non-contiguous array (using slicing)
a = np.arange(10).reshape(2, 5)
b = a[:, ::2]  # Non-contiguous slice

print("Is 'b' contiguous?", b.flags['C_CONTIGUOUS'])

# Convert to a contiguous array
b_contiguous = np.ascontiguousarray(b)
```

```
print("Is 'b_contiguous' contiguous?",
b_contiguous.flags['C_CONTIGUOUS'])
```

The results of this code are the messages below:

```
# Is 'b' contiguous? False
# Is 'b_contiguous' contiguous? True
```

Additional Tips

Avoid using np.append() repeatedly, because it reallocates the array. Prefer building a list and converting once, or use np.concatenate() or np.vstack() when appropriate.

```
import numpy as np
# Inefficient way
a = np.array([1, 2, 3])
b = np.append(a, [4, 5])  # Creates a new array, slow
for large data
print("Using np.append:", b)

# Efficient alternative using np.concatenate()
c = np.concatenate((a, [4, 5]))  # Merges arrays
efficiently
print("Using np.concatenate:", c)

# Efficient alternative for stacking vertically
d = np.vstack((a, [4, 5, 6]))  # Stacks row-wise
print("Using np.vstack:\n", d)
The output is shown below.
# Using np.append: [1 2 3 4 5]
# Using np.concatenate: [1 2 3 4 5]
# Using np.vstack:
# [[1 2 3]
# [4 5 6]]
```

Use np.isnan() and np.isinf(): when checking for NaN or Inf values, use the dedicated functions np.isnan() and np.isinf() instead of relying on == comparisons.

```
import numpy as np
# Creating an array with NaN and Inf values
arr = np.array([1, np.nan, 2, np.inf, -np.inf, 3])

# Checking for NaN values
nan_mask = np.isnan(arr)
print("NaN mask:", nan_mask)  # [False  True False False
False False]

# Checking for Inf values
inf_mask = np.isinf(arr)
print("Inf mask:", inf_mask)  # [False False
False  True  True False]

# Replacing NaN and Inf values with 0
cleaned_arr = np.where(np.isnan(arr) | np.isinf(arr), 0,
arr)
print("Cleaned array:", cleaned_arr)  #
[1.  0.  2.  0.  0.  3.]
```

The result of this code is the messages below:

```
# NaN mask: [False  True False False False False]
# Inf mask: [False False False  True  True False]
# Cleaned array: [1. 0. 2. 0. 0. 3.]
```

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_12

12. SciPy

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

SciPy (Scientific Python) is a library that builds on top of NumPy to provide functions for scientific and engineering applications. It includes modules for signal processing, optimization, statistics, and more. SciPy's functions are designed to be efficient and easy to use, making it a popular choice for tasks such as data analysis, numerical simulations, and data visualization. In this chapter, we use SciPy 1.15.2 with Python 3.13.

To learn more about the SciPy library, visit the official website: https://scipy.org/.

Source note: Function signatures, parameter descriptions, and return-value summaries in this chapter are adapted from the official SciPy 1.15.2 documentation (https://docs.scipy.org/doc/scipy-1.15.2/).

Table 12-1 provides a comprehensive overview of these modules, each tailored for a specific domain such as signal processing (signal), linear algebra (linalg), optimization (optimize), and statistics (stats). This modular structure provides scientists and engineers with a powerful toolkit, offering high-level functions for a wide range of complex computational problems.

Table 12-1 SciPy subpackages and their primary functions

Here are the SciPy subpackages from the list that can be particularly useful for engineering:

1. *constants*: Provides physical and mathematical constants, which are essential for engineering calculations.
2. *differentiate*: Useful for finite difference differentiation tools, often used in numerical simulations and engineering analysis.
3. *fft*: Fourier Transforms are critical for signal processing and systems analysis in engineering.
4. *integrate*: For performing integration, which is common in many engineering problems.
5. *interpolate*: Essential for interpolation tasks in data analysis, often used in engineering for estimating unknown values between known data points.
6. *linalg*: Linear algebra routines are crucial for solving systems of equations, which appear in many engineering fields.
7. *optimize*: Optimization is often used in engineering for finding the best parameters or solutions to problems.
8. *signal*: Signal processing techniques are highly relevant in electrical engineering, communications, and control systems.
9. *sparse*: Sparse arrays can be useful when working with large systems, which is common in many engineering problems.
10. *spatial*: Spatial data structures and algorithms are important for computational geometry and spatial analysis.
11. *special*: Special functions can be used in various engineering disciplines for solving complex mathematical models.
12. *stats*: Statistical methods are frequently used in engineering for

data analysis, quality control, and experimental design.

A Look into Key Subpackages

The power of the SciPy library lies in its collection of subpackages, each focused on a specific task. Instead of an exhaustive list, we'll take a more practical tour of the highlights. Through a series of examples, we will see how these tools work.

Constants

The scipy.constants subpackage provides a collection of physical and mathematical constants and can also be used for unit conversions. It is a valuable resource for scientific calculations that require precise constant values, offering access to constants with varying levels of precision.

The following sections present three topics in the scipy.constants subpackage: constants, SI prefixes, and unit conversion.

Physical and Mathematical Constants

The constants in this subpackage can be divided into six families, listed below:

- Mathematical constants
- Fundamental physical constants
- Atomic and nuclear constants
- Electromagnetic constants
- Thermodynamic constants
- Astronomical units and constants

The next code snippet shows an example of using mathematical constants:

```
# SciPy mathematical constants
from scipy.constants import pi, golden, e
```

```
print(f"Pi (π): {pi}")                    # Output: 3.141592653589793
print(f"Golden Ratio (φ): {golden}")  # Output: 1.618033988749895
print(f"Euler's number (e): {e}")     # Output: 2.718281828459045
print(f"Tau (τ = 2π): {2*pi}")        # Output: 6.283185307179586

# SciPy physics constants
from scipy import constants

# Fundamental constants
print(f"Speed of light in vacuum (c): {constants.c} m/s")
# Output: 299792458.0 m/s
print(f"Gravitational constant (G): {constants.G} m³/(kg·s²)")
# Output: 6.67430e-11 m³/(kg·s²)
print(f"Elementary charge (e): {constants.e} C")
# Output: 1.602176634e-19 C
print(f"Vacuum electric permittivity (ε₀): {constants.epsilon_0} F/m")  #Output: 8.8541878128e-12 F/m
print(f"Vacuum magnetic permeability (μ₀): {constants.mu_0} H/m") # Output: 1.25663706212e-06 H/m

# Atomic and nuclear constants
print(f"Electron mass (mₑ): {constants.electron_mass} kg")  # Output: 9.1093837015e-31 kg
print(f"Proton mass (mₚ): {constants.proton_mass} kg")      # Output: 1.67262192369e-27 kg
```

```
print(f"Neutron mass (m_n): {constants.neutron_mass}
kg")    # Output: 1.67492749804e-27 kg

# Electromagnetic constants
print(f"Conductance quantum (G_0):
{constants.value('conductance quantum')} S")
# Output: 7.748091729863649e-05 S
print(f"Magnetic flux quantum (Φ_0):
{constants.value('mag. flux quantum')} Wb")
# Output: 2.067833848e-15 Wb
```

SI Prefixes

The constants module in SciPy offers a wide array of SI prefixes, which act as convenient scaling factors for handling values across various magnitudes. This feature is especially useful in scientific and engineering applications, where measurements can range from extremely small to very large scales. Instead of working with less intuitive numbers, such as 0.001 meters or 0.000000038 amperes, you can use these predefined constants to express the values in more readable and compact forms, like 1 mm (millimeter) or 38 μA (microamperes).

The following example shows some examples of SI prefixes available in SciPy:

```
# List all SI prefixes available in
scipy.constants
from scipy import constants

# List all SI prefixes available in
scipy.constants
print("Available SI prefixes in scipy.constants:")
print("-" * 40)

# Dictionary to store prefixes and their values
si_prefixes = {
```

```
    # Examples of SI prefixes (powers of 10)
    'exa': constants.exa,      # 10^18
    'peta': constants.peta,    # 10^15
    'tera': constants.tera,    # 10^12
    'giga': constants.giga,    # 10^9
    'mega': constants.mega,    # 10^6
    'kilo': constants.kilo,    # 10^3
    'hecto': constants.hecto,  # 10^2
    'deka': constants.deka,    # 10^1
    'deci': constants.deci,    # 10^-1
    'centi': constants.centi,  # 10^-2
    'milli': constants.milli,  # 10^-3
    'micro': constants.micro,  # 10^-6
    'nano': constants.nano,    # 10^-9
    'pico': constants.pico,    # 10^-12
}
# Print all SI prefixes
for prefix, value in sorted(si_prefixes.items()):
    print(f"{prefix}: {value}")

print("\n\nExample usage of SI prefixes:")
print("-" * 40)
```

The conversion between prefixes is exemplified below:

```
# SI prefixes conversion available in
scipy.constants
from scipy import constants

# Define a value in base units
bytes_value = 1500000000  # bytes

# Convert to different prefixes
print(f"Original value: {bytes_value} bytes")
```

```
print(f"In kilobytes: {bytes_value /
constants.kilo} kB")
# Output: 1500000.0 kB
print(f"In megabytes: {bytes_value /
constants.mega} MB")
# Output: 1500.0 MB
print(f"In gigabytes: {bytes_value /
constants.giga} GB")
 # Output: 1.5 GB

# Working with different magnitudes
small_current = 0.000000025  # amperes
print(f"\nSmall current: {small_current} A")
print(f"In microamperes: {small_current *
constants.mega} µA")
# Output: 0.024999999999999998 µA
print(f"In nanoamperes: {small_current *
constants.giga} nA")
# Output: 25.0 nA

# Distance example
distance = 299792458  # meters (speed of light in
1 second)
print(f"\nDistance: {distance} m")
print(f"In kilometers: {distance / constants.kilo}
km")
# Output: 299792.458 km
print(f"In megameters: {distance / constants.mega}
Mm")
# Output: 299.792458 Mm

# Time example
time_value = 0.000001  # seconds
print(f"\nTime: {time_value} s")
```

```
print(f"In microseconds: {time_value /
constants.micro} µs")
# Output: 1.0 µs
print(f"In nanoseconds: {time_value /
constants.nano} ns")
# Output: 1000.0 ns
```

Unit Conversion

The scipy.constants module provides a variety of unit conversion tools, allowing users to easily switch between different measurement units commonly used in physics and engineering. It offers straightforward conversion factors for various quantities, such as length (e.g., converting meters to feet), pressure (such as converting bars to psi), mass (like converting grams to ounces), and time (e.g., converting hours to seconds). Additionally, the convert_temperature() function enables seamless temperature conversions between Celsius, Fahrenheit, and Kelvin.

The code below provides various unit conversions for physical quantities such as length, mass, pressure, power, and temperature. It utilizes the scipy.constants module to access conversion factors between different units. The code then applies these factors to perform conversions and prints the results:

```
from scipy import constants

print("\nLENGTH CONVERSIONS:")       # Length
conversions
inches_value = 42
print(f"{inches_value} inches = {inches_value *
constants.inch} meters")
# Output: 42 inches = 1.0668 meters
print(f"{inches_value} inches = {inches_value *
constants.inch / constants.mile} miles")
# Output: 42 inches = 0.0006628787878787879 miles
```

```
print("\nMASS CONVERSIONS:")          # Mass
conversions
mass_kg = 75
print(f"{mass_kg} kg = {mass_kg / constants.pound}
pounds")
# Output: 75 kg = 165.34661739734547 pounds

print("\nPRESSURE CONVERSIONS:")      # Pressure
conversions
pressure_pa = 101325                  # Standard
atmospheric pressure
print(f"{pressure_pa} Pa = {pressure_pa /
constants.bar} bar")  # Output: 101325 Pa =
1.01325 bar
print(f"{pressure_pa} Pa = {pressure_pa /
constants.psi} psi")  # Output: 101325 Pa =
14.695950254342185 psi

print("\nPOWER CONVERSIONS:")         # Power
conversions
power_w = 745.7                       #
Approximately 1 horsepower
print(f"{power_w} W = {power_w /
constants.horsepower} horsepower")
# Output: 745.7 W = 1.0000001722110126 horsepower
print(f"{power_w} W = {power_w / 1000} kilowatts")
# Output: 745.7 W = 0.7457 kilowatts
```

To use the temperature conversion, see the code below:

```
from scipy.constants import convert_temperature

# Initial temperature in Celsius
celsius = 25
```

```
# Use the SciPy function for seamless conversions
fahrenheit = convert_temperature(celsius,
'Celsius', 'Fahrenheit')
kelvin = convert_temperature(celsius, 'Celsius',
'Kelvin')

print(f"{celsius} °C = {fahrenheit} °F")       #
Output: 25 °C = 77.0 °F
print(f"{celsius} °C = {kelvin} K")            #
Output: 25 °C = 298.15 K
```

Differentiate

SciPy's differentiation module provides functions to numerically differentiate functions using finite differences, especially in cases where the underlying function is unknown or opaque, often referred to as "black-box" functions. This means the function has an unspecified or hidden internal structure or algorithm, enabling numerical differentiation without the need to know the analytical form of the function or its derivatives.

Basically, three operations are available in this subpackage, as shown in Table 12-2:

- *Derivative*: Calculates the rate at which a single-variable function changes
- *Jacobian Matrix*: Evaluates the partial derivatives of a multi-variable function, resulting in a matrix of first-order derivatives
- *Hessian Matrix*: Computes the second-order partial derivatives of a single-variable function, yielding a curvature

Table 12-2 Summary of numerical differentiation operations available in scipy.differentiate

Derivative

The derivative function is designed to estimate the numerical derivative of a function. It approximates the rate of change of a function at a given

point or set of points, with options to control accuracy, step size, and other aspects of the computation.

The function has several parameters, as shown below:

```
derivative(f, x, *, args=(), tolerances=None,
maxiter=10, order=8, initial_step=0.5,
step_factor=2.0, step_direction=0,
preserve_shape=False, callback=None)
```

The * symbol after (f,x,* ...) in a function signature is used to indicate the end of positional arguments and the start of keyword-only arguments. This means that any parameters listed after the * must be passed as keyword arguments and cannot be passed positionally.

The parameters of the derivative function and the default values are explained in Table 12-3.

Table 12-3 Parameters and descriptions for the scipy.differentiate.derivative function

The derivative function returns a RichResult object. This object has attributes that provide information about the derivative approximation.

For more details see Table 12-4, describing the attributes of the RichResult object returned by the derivative, jacobian, and hessian functions in scipy.differentiate.

Table 12-4 Attributes of the RichResult object returned by SciPy's differentiation functions

The following example code numerically approximates the derivative of a function. It defines a function f(x) that returns x2 and then calculates the derivative of this function at the point x=2. The derivative function estimates the derivative, and the code prints both the approximated derivative value and an estimate of the error in the approximation:

```
import numpy as np
```

```
from scipy.differentiate import derivative

# Define the function whose derivative we want to
approximate
def f(x):
    return x**2  # Example function: x^2

# Define the point at which to evaluate the
derivative
x = np.array([2])

# Calculate the derivative using
scipy.differentiate.derivative
res = derivative(f, x)

# Print the results
print("Approximated derivative values:", res)
print("Error estimate:", res.error)
```

Running the example produces

```
Approximated derivative values:      success: [
True]
      status: [0]
          df: [ 4.000e+00]
       error: [ 2.132e-14]
         nit: [2]
        nfev: [11]
           x: [ 2.000e+00]
Error estimate: [2.13162821e-14]
```

Error Estimate

The error estimate printed in the screen is optional, but useful to check the precision of the operation. It is important to know that the error estimate doesn't directly measure how close the approximated

derivative is to the true derivative. Instead, it reflects the difference between successive approximations of the derivative as the step size is reduced.

The function employs an iterative finite difference method, which involves

1. Initializing a step size
2. Calculating the derivative
3. Reducing the step size
4. Recalculating the derivative

The error estimate is derived from the difference between successive approximations. This estimate serves as a convergence criterion: when the error becomes sufficiently small, the approximation stabilizes, indicating higher accuracy.

In the example, an error of 2.13162821e-14 doesn't necessarily mean the approximation is that far from the true value; rather, it indicates the difference between successive approximations. 10^-14 means the algorithm has converged to a very precise result, as the difference between the final approximations is extremely small.

To print only the value of the function's derivative, use this code below. The code accesses the df attribute of the res object, which contains the derivative value:

```
import numpy as np
from scipy.differentiate import derivative

# Define the function
def f(x):
    return np.exp(x)
```

```
# Define the point at which to evaluate the
derivative
x = np.array([2])

# Calculate the derivative
res = derivative(f, x)

# Print only the derivative values
print("df:", res.df)    # Output: df: [7.3890561]
```

Also it is possible to pass additional arguments to the function whose derivative we want to calculate. The example below demonstrates how to do this:

```
import numpy as np
from scipy.differentiate import derivative

def f(x, a, b):
    return a * x**2 + b * x

a = 2
b = 3
x = np.array([1, 2, 3])

# Calculate the derivative of f with respect to x,
passing a and b as arguments
res = derivative(f, x, args=(a, b))

print("df:", res.df)   # Output: df: [ 7. 11. 15.]
```

Controlling the Direction of Finite Difference Approximations

The step_direction parameter controls the direction of the finite difference approximation.

A value of –1 uses backward differences, 0 uses central differences, and 1 uses forward differences, thus enabling the computation of

derivatives using different finite difference methods by specifying the direction in which the function is sampled around the evaluation point.

```
import numpy as np
from scipy.differentiate import derivative

def f(x):
    return x**3

x = np.array([1])
step_directions = np.array([-1, 0, 1])  # -1:
backward, 0: central, 1: forward

results = derivative(f, x,
step_direction=step_directions)

print("Derivatives (backward, central, forward):",
results.df)
# Output: Derivatives (backward, central,
forward): [3. 3. 3.]
```

So step_direction lets you choose whether to use the points on both sides, only the front, or only the back of the point you're interested in to calculate the "slope."

By choosing the direction, you can compute derivatives using different methods, which can be useful depending on the problem you're trying to solve.

Imagine you're trying to figure out how steep a hill is at a certain point. You can do this by looking at how much the hill changes:

- *Backward*: Looking behind you, using steps in the negative direction (–1)
- *Central*: Looking both behind and ahead of you, using steps in both positive and negative directions (0)
- *Forward*: Looking ahead of you, using steps in the positive direction (1)

The step_direction parameter in scipy.differentiate.derivative lets you choose which direction to look.

Figure 12-1 provides a clear visual explanation of the three primary finite difference methods used to approximate the derivative of a function. This concept is directly implemented in SciPy's derivative function through the step_direction parameter, which controls how the function is sampled to calculate the slope. The solid blue curve represents the function f(x), and the dashed red line indicates the true tangent, representing the actual derivative at point x.

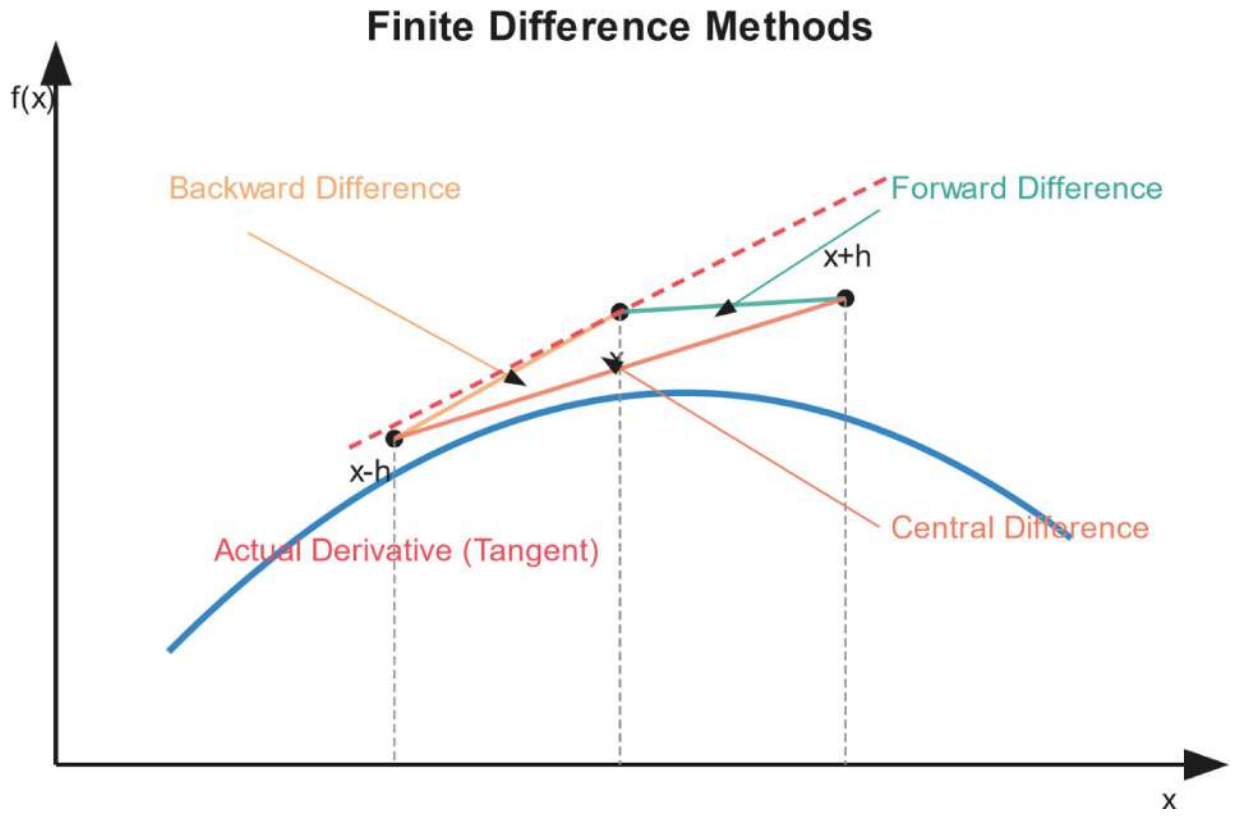


Figure 12-1 Forward, backward, and central finite difference approximations to the derivative of a function

The three methods illustrated are

- *Forward Difference (*step_direction=1*)*: This method approximates the derivative by calculating the slope of the line connecting the point of interest, x, and a point just ahead of it, x+h. It essentially "looks ahead" to estimate the function's rate of change.

- *Backward Difference (*step_direction=-1*)*: This approach estimates the derivative using the slope of the line between the point of interest, x, and a point just behind it, x-h. It “looks backward” to determine the rate of change.
- *Central Difference (*step_direction=0*)*: This method calculates the slope using points symmetrically placed on either side of x (specifically, x-h and x+h). As the visualization suggests, the slope of the central difference line is often a more accurate approximation of the actual tangent. For this reason, it is the default method used in the scipy.differentiate.derivative function.

Preserve Shape

There may be cases where a function whose derivative is being calculated does not handle input arrays with broadcastable shapes correctly. To resolve this issue, the preserve_shape parameter is used.

In the next example code, the function f(x) returns a list, which can lead to shape mismatches during the derivative calculation. Setting preserve_shape=True ensures that the function f(x) is called with input arrays that have consistent shapes, thus preventing potential errors due to broadcasting issues:

```
import numpy as np
from scipy.differentiate import derivative

def f(x):
    return [x, np.sin(3*x), x + np.sin(10*x),
np.sin(26*x) * (x - 1)**2]

x = np.array([0])

# Calculate the derivative with
preserve_shape=True
res = derivative(f, x, preserve_shape=True)

print("df:", res.df)
```

```
The output is:
df: [[ 1.]
 [ 3.]
 [11.]
 [26.]]
```

Jacobian

It is possible to perform a numerical approximation of the Jacobian matrix of a given function f at a specific point x.

The Jacobian matrix is a matrix of all first-order partial derivatives of a vector-valued function. The function f(x) in this example takes a 1D array x and returns a 1D array where the elements are a combination of the input array elements. The jacobian function calculates the partial derivatives of each output of f with respect to each input in x, arranged in a matrix.

```
import numpy as np
from scipy.differentiate import jacobian

# Define the function whose Jacobian is desired
def f(x):
    return np.array([x[0]**2 * x[1], x[0] +
np.sin(x[1])])

# Define the point at which to evaluate the
Jacobian
x = np.array([1, 2])

# Calculate the Jacobian using
scipy.differentiate.jacobian
res = jacobian(f, x)

# Print the Jacobian
print("Jacobian:", res.df)
The output is:
```

```
Jacobian: [[ 4. 1. ]
 [ 1. -0.41614684]]
```

Hessian

To numerically calculate the Hessian matrix of a function, use scipy.differentiate.hessian. The Hessian matrix is a square matrix of second-order partial derivatives of a scalar-valued function. In the code below, the Hessian of the Rosenbrock function is computed at a specific point.

The code compares the numerically calculated Hessian with the analytically derived Hessian, obtained using scipy.optimize.rosen_hess, to verify the accuracy of the numerical approximation.

The Hessian matrix is essential in optimization for determining the curvature of a function, which helps identify the nature of stationary points—such as minima, maxima, or saddle points—and is also used in algorithms like Newton's method for faster convergence:

```
import numpy as np
from scipy.differentiate import hessian
from scipy.optimize import rosen, rosen_hess

# Example from the documentation
m = 3
x = np.full(m, 0.5)  # Create an array filled with
0.5

# Calculate the Hessian of the Rosenbrock function
using scipy.differentiate.hessian
res = hessian(rosen, x)

ref = rosen_hess(x)  # reference value of the
Hessian

print("Hessian:", res.ddf)
print("Reference Hessian:", ref)
```

```
# Verify if the calculated Hessian is close to the
reference Hessian
print("Hessian close to reference:",
np.allclose(res.ddf, ref))
```

The result of the code for calculating the Hessian matrix is shown below:

```
Hessian: [[ 1.02000000e+02 -2.00000000e+02
-2.28888497e-12]
 [-2.00000000e+02  3.02000000e+02 -2.00000000e+02]
 [ 2.64722950e-10
-2.00000000e+02  2.00000000e+02]]
Reference Hessian: [[ 102. -200.    0.]
 [-200.  302. -200.]
 [   0. -200.  200.]]
```

Hessian close to reference: True

FFT

The scipy.fft submodule provides functions for performing Fast Fourier Transforms (FFTs). These functions enable the analysis of data across multiple spatial or temporal dimensions and are widely used in fields such as signal processing, image and video analysis, control systems, communication systems, and data science.

It is important to note that (x) in lowercase represents the input signal, while (X) in uppercase represents the output signal. This is a notational convention to distinguish between the input and output of Fourier Transform functions.

The following overview summarizes the main FFT functions before they are discussed in groups according to signal dimensionality.

Specialized Fourier Transforms

These functions handle special cases of Fourier Transforms, such as real-valued inputs or Hermitian symmetry. The “rfft(x)” and “irfft(X)” functions are optimized for real-valued signals, reducing computational complexity by returning only positive frequencies. The “hfft(x)” and “ihfft(X)” functions are designed for signals with “Hermitian symmetry,” a property often encountered in real-valued signals, ensuring efficient computation and reconstruction. These specialized functions are particularly useful in applications like audio processing, where real-valued signals are common.

Summary of FFT Functions

This topic provides a quick reference for all the functions, grouped by their dimensionality and special cases. It includes 1D, 2D, and N-dimensional transforms, as well as specialized functions for real-valued inputs and Hermitian symmetry. Table 12-5 helps users quickly identify the right function for their needs, whether they are working with simple 1D signals, 2D images, or complex N-dimensional data. It serves as a handy guide for both beginners and advanced users of Fourier Transforms.*Fast Fourier Transforms (FFTs)*

Table 12-5 Summary of Fast Fourier Transform (FFT) functions in scipy.fft categorized by dimensionality

1D Fourier Transforms

The first group encompasses fundamental operations on single-dimensional signals. This group efficiently processes signals such as

- Audio signals, including speech and music
- Time-series data, such as heart rate measurements and stock prices
- Sensor data, including temperature and humidity readings from weather sensors
- Communication signals, such as those used in wireless and wired communication systems

The function fft(x) computes the Discrete Fourier Transform (DFT) of a one-dimensional signal, transforming it from the time domain to the frequency domain, while ifft(X) does the inverse, reconstructing the original signal. For real-valued signals, rfft(x) computes the DFT but returns only the positive frequency terms, making it more efficient, and irfft(X) reconstructs the original signal. The hfft(x) and ihfft(X) functions handle signals with Hermitian symmetry, a property often encountered in real-valued signals, ensuring efficient computation and reconstruction.

Function FFT

The Fast Fourier Transform (FFT) operation is used to find how much of each frequency is present in the signal.

The scipy.fft.fft function computes the 1D Discrete Fourier Transform (DFT) of a sequence using the Fast Fourier Transform (FFT) algorithm. The FFT is an efficient way to compute the DFT, which transforms a signal from its time domain or spatial domain representation into its frequency domain representation.

The function signature is shown below:

```
fft(x, n=None, axis=-1, norm=None,
overwrite_x=False, workers=None, *, plan=None)
[source]
```

As seen earlier, parameters listed before the * can be passed either positionally or as keyword arguments.

Table 12-6 Parameters and descriptions for the scipy.fft.fft function

The following example code generates a 60 Hz sine wave signal and computes its frequency spectrum using the Fast Fourier Transform (FFT) from SciPy and then prints the first ten frequency components of the resulting spectrum.

This code uses the linspace function from the NumPy library to create a one-dimensional array representing a sequence of time points or values. The array, named t, consists of 500 evenly spaced numbers within the interval [0, 1), meaning it includes 0 but excludes 1. This exclusion occurs because the parameter endpoint=False explicitly instructs np.linspace not to include the stop value (1 in this case) as the final element in the generated array.

Here are the parameters of the linspace function:

- *0 (*start*)*: This is the starting value of the sequence. It means that the first element in the array t will be 0.
- *1 (*stop*)*: This is the ending value of the sequence. However, because the endpoint parameter is set to False, this value will *not* be included in the array t. The last element in the array will be very close to 1 but not exactly 1.
- *500 (*num*)*: This integer specifies the number of evenly spaced samples to be generated within the specified interval. In this case, the linspace function will create an array containing exactly 500 numbers.
- endpoint=False: This boolean parameter determines whether the stop value should be included in the resulting array. When endpoint=True, the stop value is included as the last element. If endpoint is not specified, its default value is True. When endpoint=False, the stop value is excluded from the array.

```
import numpy as np
from scipy.fft import fft

# Create a simple signal (sine wave)
t = np.linspace(0, 1, 500, endpoint=False)
signal = np.sin(2 * np.pi * 60 * t)  # 60 Hz sine
wave

# Compute the FFT
spectrum = fft(signal)
```

```
print(spectrum[:10])  # Print first 10 frequency
components
```

Running the example produces

```
[ 2.74518957e-14-0.00000000e+00j  4.49909016e-14-
3.76995520e-14j
 -8.77203106e-14-3.60596157e-15j  2.14935729e-13-
7.20504470e-14j
  4.64068792e-14+6.46132454e-14j -1.70130222e-13-
2.99317735e-14j
  5.59153833e-14-1.10088165e-13j  2.82225218e-
14+9.05004605e-14j
 -1.98077068e-14+3.42188935e-14j  3.33801084e-14-
2.33615856e-14j]
```

Note that the result was a sequence of complex numbers. The complex number has the a + bj, where a is the real part and b is the imaginary part.

- The real part a represents the amplitude of the frequency component.
- The imaginary part b represents the phase of the frequency component.

In this case, the output array has ten complex numbers, which represent the frequency components of the input signal.

The output array does not directly display frequency values. However, it is possible to determine the frequency corresponding to each complex number in the output array using the fftfreq() function, which generates the frequency bins for a given Discrete Fourier Transform (DFT) output. An example is shown in the following:

```
import numpy as np
from scipy.fft import fftfreq
```

```
# Given FFT result (frequency domain signal)
signal = [
    2.74518957e-14 - 0.00000000e+00j, 4.49909016e-
14 - 3.76995520e-14j,
    -8.77203106e-14 - 3.60596157e-15j,
2.14935729e-13 - 7.20504470e-14j,
    4.64068792e-14 + 6.46132454e-14j,
-1.70130222e-13 - 2.99317735e-14j,
    5.59153833e-14 - 1.10088165e-13j, 2.82225218e-
14 + 9.05004605e-14j,
    -1.98077068e-14 + 3.42188935e-14j,
3.33801084e-14 - 2.33615856e-14j
]

# Sampling interval (d) is 0.002 seconds
d = 0.002

# Compute frequency bins
freq_bins = fftfreq(len(signal), d=d)

# Print the first 10 frequency bins
print(freq_bins[:10])

# Output: [   0.   50.  100.  150.  200. -250.
-200. -150. -100.  -50.]
```

The frequency bins range from 0 Hz or DC component to the Nyquist frequency of 250 Hz and then mirror back down to –250 Hz.

The positive frequency bins 0–250 Hz represent the actual frequency components present in the signal.

The negative frequency bins –250 to 0 Hz are a mirror image of the positive frequency bins, due to the symmetry property of the DFT.

Since a 60 Hz sine wave was generated, we would expect to see a peak in the frequency domain spectrum at 60 Hz. However, the output of fftfreq() does not show a range at exactly 60 Hz because the FFT

produces discrete frequency intervals, so the 60 Hz component is captured by the interval with the closest center frequency.

The Nyquist frequency is a fundamental concept in digital signal processing and is defined as half of the sampling rate. In this case, the sampling rate is 500 Hz, which means that you're taking 500 samples per second. This can be seen in the part of code made earlier:

```
# Create a simple signal (sine wave)
t = np.linspace(0, 1, 500, endpoint=False)
```

According to the Nyquist–Shannon sampling theorem, a continuous-time signal can be perfectly reconstructed from its samples if the sampling rate is greater than twice the highest frequency component in the signal.

If you want to accurately capture a signal with a frequency component of 250 Hz, it is needed to sample it at a rate of at least twice the frequency 500 Hz.

The Nyquist frequency (250 Hz) represents the highest frequency that can be accurately captured by the sampling process. Any frequency components above the Nyquist frequency will be aliased, meaning that they will be distorted and appear as lower-frequency components.

The following figure illustrates the concept of aliasing, a fundamental pitfall in digital signal processing that occurs when a signal is sampled at an insufficient rate. Aliasing leads to the misinterpretation of a high-frequency signal as one of a much lower frequency. This phenomenon is a direct consequence of violating the Nyquist–Shannon sampling theorem.

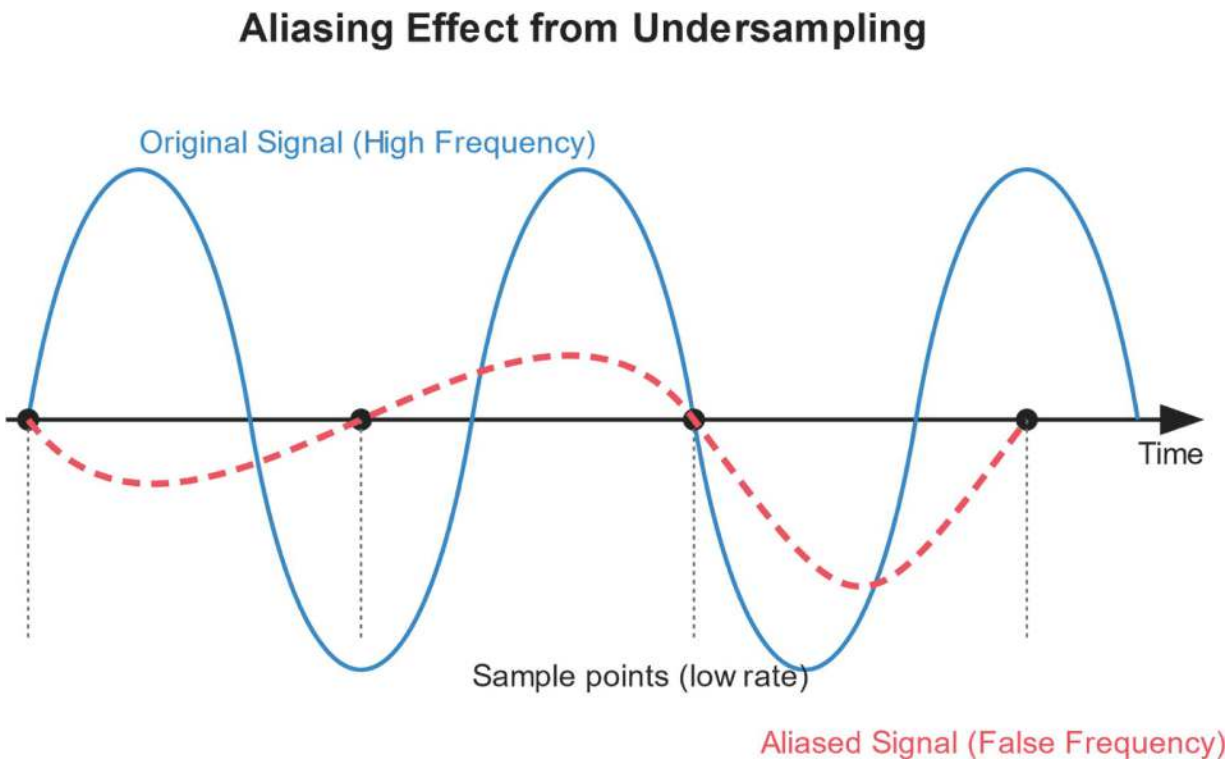


Figure 12-2 Aliasing caused by undersampling a high-frequency signal

This graphic provides a clear visual indication of the importance of selecting a sufficiently high sampling rate before performing a Fast Fourier Transform (FFT) to avoid erroneous conclusions about a signal's frequency content.

The key components in the visualization are

- *Original Signal (High Frequency)*: The solid blue line represents the true, continuous signal, which oscillates at a high frequency.
- *Sample Points (Low Rate)*: The black dots represent the discrete measurements taken from the signal. The critical aspect shown here is that the time between samples is too long, meaning the sampling rate is too low to capture the signal's rapid changes.
- *Aliased Signal (False Frequency)*: The dashed red line is the incorrect, lower-frequency signal that would be reconstructed from the sparse sample points. This "false frequency" is an artifact of the undersampling process.

Aliasing occurs when the original signal's frequency exceeds the Nyquist frequency—half the sampling rate. According to the sampling theorem, a signal can be perfectly reconstructed only if the sampling rate is more than double the signal's highest frequency component.

In your case, since you're generating a 60 Hz sine wave, which is well below the Nyquist frequency, you can accurately capture the signal without aliasing.

The last code above does not generate a graphic, but for a better understanding of the signal and the frequency spectrum, the following graph was created.

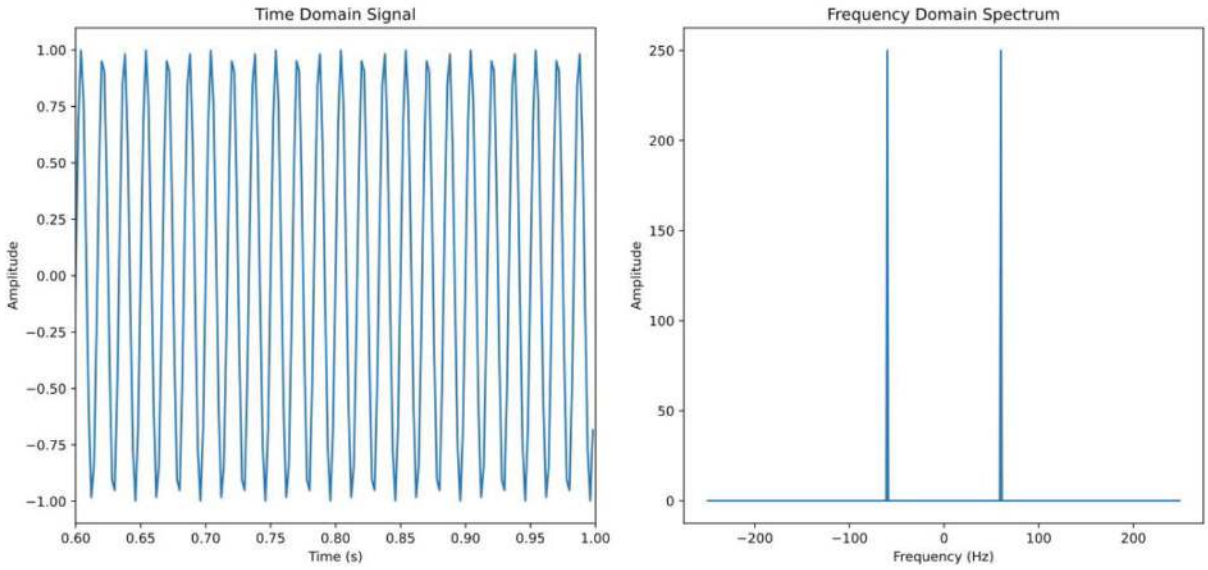


Figure 12-3 Time-domain waveform and two-sided Fourier magnitude spectrum of a 60 Hz sinusoidal signal

The Time Domain Signal graph shows how the signal changes over time. The x-axis represents time in seconds, and the y-axis represents the amplitude or magnitude of the signal.

In this case, the signal is a 60 Hz sine wave, which means it completes 60 cycles per second. The graph shows the signal's oscillations over time, with the amplitude varying between –1 and 1. The zoomed-in view, x-axis limited to 0.6–1.0 seconds allows to see the signal's details more clearly.

The Frequency Domain Spectrum graph shows the signal's frequency content. The x-axis represents frequency in Hz, and the y-axis represents the amplitude of each frequency component.

The graph shows a single peak at 60 Hz, indicating that the signal consists of a single frequency component at 60 Hz. The amplitude of this peak represents the strength of the 60 Hz component.

In other words, this graph tells us that the signal is primarily composed of a 60 Hz sine wave, which is consistent with the time domain signal observed.

1D Inverse Fast Fourier Transform (IFFT)

The **Inverse Fast Fourier Transform (IFFT)** is the reverse operation of the **Fast Fourier Transform (FFT)**. It takes a frequency-domain representation—such as the output of the FFT—and reconstructs the original time-domain signal. Mathematically, the IFFT ensures that the transformation between the time and frequency domains is reversible. This means that applying the FFT followed by the IFFT should ideally recover the original signal, with only minor numerical precision errors. These errors occur due to the limitations of floating-point arithmetic in computers, which represent real numbers with finite precision, leading to small rounding or truncation errors. However, in most practical applications, these errors are negligible, as they are much smaller than typical noise or measurement errors in real-world signals.

Together, the FFT and IFFT form the foundation of many modern signal processing techniques, enabling efficient analysis, manipulation, and synthesis of signals in both domains.

The following example demonstrates the use of the Fast Fourier Transform (FFT) and its inverse (IFFT) to analyze and reconstruct a signal. First, as in the previous example, a 60 Hz sine wave is generated. The fft function transforms the signal from the time domain to the frequency domain.

Next, the Inverse FFT (IFFT) is applied to the spectrum to reconstruct the original time-domain signal. The first ten values of the reconstructed signal are printed, which should closely match the

original sine wave, confirming the accuracy of the FFT and IFFT operations. This process is commonly used in signal processing to analyze and manipulate signals in the frequency domain:

```
import numpy as np
from scipy.fft import fft, ifft

# Create a simple signal (sine wave)
t = np.linspace(0, 1, 500, endpoint=False)
signal = np.sin(2 * np.pi * 60 * t)  # 60 Hz sine
wave
# Compute the FFT
spectrum = fft(signal)

# Compute the inverse FFT of the previous spectrum
reconstructed_signal = ifft(spectrum)
# Print first 10 time-domain values
print(reconstructed_signal[:10])
```

The output of this code is shown below, an array with ten complex numbers:

```
[ 1.51461294e-30-4.03896783e-31j  6.84547106e-
01+1.21169035e-30j
  9.98026728e-01+1.42108547e-17j  7.70513243e-01-
5.68434189e-17j
  1.25333234e-01+3.02922588e-30j -5.87785252e-01-
5.68434189e-17j
 -9.82287251e-01-5.68434189e-17j -8.44327926e-
01+1.13686838e-16j
 -2.48689887e-01+5.55358077e-31j  4.81753674e-
01+1.38396317e-30j]
```

1D FFT for Real Input (RFFT)

The Real-input Fast Fourier Transform (RFFT) is a specialized version of the Fast Fourier Transform (FFT) tailored for signals that are real-valued. While the standard FFT calculates a complete spectrum of complex numbers, including both real and imaginary parts, the RFFT leverages the inherent symmetry in the frequency spectra of real-valued signals. This symmetry allows the RFFT to compute and return only the non-redundant half of the frequency components, significantly improving efficiency in both computation speed and memory usage.

Figure 12-4 illustrates the concept of **Hermitian symmetry**, a fundamental property of the Fourier Transform that arises whenever the input signal is purely real-valued. This symmetry imposes a specific, predictable structure on the resulting complex frequency spectrum, meaning the negative frequency components are entirely determined by the positive frequency components.

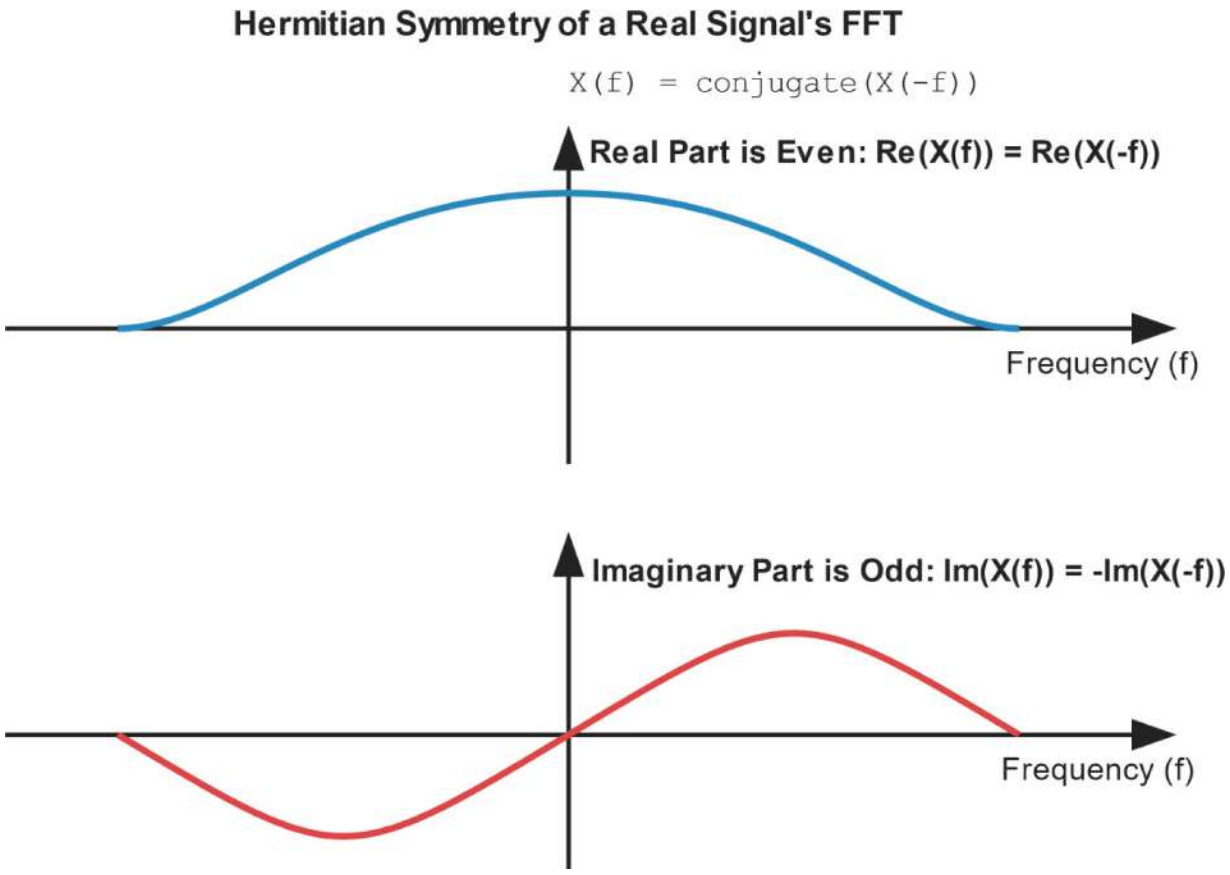


Figure 12-4 Hermitian symmetry in the FFT of a real-valued signal: an even real part and an odd imaginary part

The overall relationship is that the spectrum at a frequency f is the complex conjugate of the spectrum at –f. This is broken down into two simpler rules for the real and imaginary parts:

- *The Real Part Is Even*: As shown in the top plot, the real component of the frequency spectrum is perfectly symmetric around the vertical axis. The value at any positive frequency +f is identical to the value at the corresponding negative frequency –f.
- *The Imaginary Part Is Odd*: The bottom plot shows that the imaginary component of the spectrum is anti-symmetric. The value at a positive frequency +f is the exact negative of the value at –f.

This inherent redundancy is exploited by specialized functions in the scipy.fft subpackage to achieve greater efficiency. For example, the rfft() function is optimized for real-valued signals by only calculating and returning the positive frequency terms, which saves both computation time and memory. The hfft() function is designed specifically to work with signals that have this Hermitian property.

RFFT is widely used in signal processing, audio analysis, data compression, and other fields where real-valued signals are common and computational efficiency is critical.

The following example computes the Real-input Fast Fourier Transform (RFFT) of a signal using scipy.fft.rfft, which is optimized for real-valued input signals, and prints the first ten frequency components of the resulting one-sided spectrum:

```
import numpy as np
from scipy.fft import rfft

# Create a signal - using a sine wave as an
example
fs = 1000  # Sample rate (Hz)
t = np.arange(0, 1, 1/fs)  # 1 second time array
f1, f2 = 50, 120  # Frequencies of components (Hz)
signal = np.sin(2 * np.pi * f1 * t) + 0.5 *
np.sin(2 * np.pi * f2 * t)
```

```
# Compute the real-input FFT (more efficient for
real signals)
real_spectrum = rfft(signal)

print(real_spectrum[:10])  # Print first 10
frequency components
```

The printed result is a complex-valued array with ten complex numbers:

```
[-1.34264606e-13+0.00000000e+00j  1.47945457e-14-
1.80425085e-14j
 -1.05040578e-13-2.71965446e-13j  2.75195556e-13-
6.63933779e-14j
  7.85294964e-14+1.51964849e-13j -2.01002995e-
13+9.55840221e-14j
  8.90207843e-15-8.16760194e-14j  4.25451299e-14-
3.43601976e-14j
  1.41849841e-13-1.57365092e-13j  4.58216015e-14-
1.31616405e-13j]
```

2D Fourier Transforms

The second group performs operations on two-dimensional signals, which are typically functions of two spatial dimensions, such as

- *Images*: Grayscale or color images
- *Medical Imaging*: X-ray or MRI scans
- *Radar Data*: 2D radar images for object detection
- *Seismic Data*: 2D seismic waves recorded for geological analysis
- *Computer Vision*: Depth maps or edge-detected images

These functions are designed for two-dimensional arrays, such as images or matrices. The fft2(x) function computes the 2D Discrete Fourier Transform (DFT), transforming a 2D signal (such as an image) into its frequency representation, while ifft2(X) reconstructs the original 2D signal. For real-valued 2D inputs, rfft2(x) computes the DFT

efficiently by returning only positive frequencies, and irfft2(X) reconstructs the original 2D signal. The hfft2(x) and ihfft2(X) functions handle 2D arrays with Hermitian symmetry, ensuring efficient computation and reconstruction for complex-valued inputs.

2D Fast Fourier Transform (FFT2)

The 2D Discrete Fourier Transform (FFT2) is a powerful tool for analyzing and processing 2D signals and images and is used for efficient application of filters, convolution, and other operations. FFT2 has numerous applications in image and signal processing, including image filtering, de-noising, compression, and feature extraction. It is also used in various fields such as medical imaging, radar, and seismic data analysis.

The following example uses the 2D Fast Fourier Transform to analyze the frequency components of a 2D signal, such as an image. It begins by creating a simple 8 × 8 matrix filled with random values, simulating a grayscale image where each pixel value represents intensity. The fft2 function is then applied to compute the 2D FFT of this matrix, transforming it from the spatial domain (pixel intensities) into the frequency domain (complex numbers) representing the amplitude and phase of spatial frequencies. The result contains the frequency components of the image, which can be used for tasks like filtering, compression, or feature extraction in image processing:

```
from scipy.fft import fft2
import numpy as np

# Create a simple 2D signal (as an image-like
matrix)
image = np.random.rand(8, 8)

# Compute the 2D FFT
image_spectrum = fft2(image)

print(image_spectrum)
```

The result of the code is a complex-valued matrix with eight rows and eight columns. The numbers in the matrix are represented in scientific notation of complex numbers. Thus, there are 64 numbers in the matrix.

N-Dimensional Fourier Transforms

N-dimensional Fourier Transforms are used for data with three or more dimensions, including the following examples:

- *3D Volumetric Data*: 3D MRI or CT scans
- *Video Data*: A sequence of 2D images (frames) over time
- *Scientific Simulations*: 3D fluid dynamics or weather data
- *Hyperspectral Imaging*: 3D data cubes with two spatial dimensions and one spectral dimension
- *4D Data*: 3D volumetric data over time

These functions extend Fourier Transforms to N-dimensional arrays, such as 3D volumes or higher-dimensional data. The fftn(x) computes the N-dimensional DFT, transforming the input into its frequency representation, while ifftn(X) reconstructs the original N-dimensional signal. For real-valued N-dimensional inputs, rfftn(x) computes the DFT efficiently by returning only positive frequencies, and irfftn(X) reconstructs the original signal. The hfftn(x) and ihfftn(X) functions handle N-dimensional arrays with Hermitian symmetry, ensuring efficient computation and reconstruction for complex-valued inputs.

N-Dimensional Discrete Fourier Transform (FFTN)

The N-dimensional Discrete Fourier Transform (FFTN), often referred to as the multidimensional FFT, is a generalization of the Discrete Fourier Transform to N-dimensional arrays or signals.

The example code demonstrates the use of the N-dimensional Fast Fourier to analyze a 3D volumetric dataset in the frequency domain. It begins by creating a 3D array volume of random values using numpy, simulating a 4 × 4 × 4 volumetric dataset. The fftn function from

scipy.fft is then applied to compute the N-dimensional FFT of the volume, transforming it from the spatial-domain voxel intensities into the frequency-domain complex numbers representing amplitude and phase of spatial frequencies. The resulting volume_spectrum contains the frequency components of the 3D signal, which can be used for tasks like filtering, compression, or feature extraction in applications such as medical imaging or scientific simulations:

```
from scipy.fft import fftn
import numpy as np

# Create a 3D signal (e.g., a volumetric dataset)
rng = np.random.default_rng(0)
volume = rng.random((4, 4, 4))

# Compute the N-D FFT
volume_spectrum = fftn(volume)

print("Spectrum shape:", volume_spectrum.shape)
print(volume_spectrum[0, :2, :2])
Spectrum shape: (4, 4, 4)
[[31.5931766 -0.j          -0.131419  -2.54506118j]
 [-1.90273793-
1.00512699j  0.42764253+0.63349858j]]
```

Discrete Sine and Cosine Transforms (DST and DCT)

The Discrete Cosine Transform (DCT) and Discrete Sine Transform (DST) are mathematical techniques used to transform a signal or sequence of data points into a sum of cosine or sine functions and are widely used in signal processing, image compression, and solving partial differential equations.

Table 12-7 lists all types of operations that can be performed with Discrete Sin and Cosine Transforms.

Table 12-7 Functions for Discrete Sine and Cosine Transforms (DST and DCT) in scipy.fft

Discrete Cosine Transform (DCT)

The following example demonstrates how to use the DCT function from scipy.fft to compute the Discrete Cosine Transform (DCT) of a given input array.

The function signature of the Discrete Cosine Transform is showed below:

```
dct(x, type=2, n=None, axis=-1, norm=None,
overwrite_x=False, workers=None,
orthogonalize=None)
```

Table 12-8 outlines the parameters for the Discrete Cosine Transform (DCT) function.

The input array x is a required parameter with no default value, representing the data to be transformed.

The type parameter specifies the DCT variant as 1, 2, 3, or 4, with a default of Type 2, the most commonly used.

The n parameter defines the length of the transform, defaulting to the size of the input array along the specified axis, which defaults to the last axis at –1.

The norm parameter controls normalization, with options like backward as the default, ortho, and forward.

The overwrite_x parameter, defaulting to False, allows in-place computation by potentially modifying the input array.

The workers parameter, defaulting to None, sets the number of parallel workers for computation, wrapping around the system CPU count if negative.

Finally, orthogonalize determines whether to use the orthogonalized DCT variant, defaulting to True if norm is ortho and False otherwise.

These parameters .provide flexibility for customizing the DCT computation based on specific needs.

Table 12-8 Parameters and descriptions for the scipy.fft.dct function

There are four types of the Discrete Cosine Transform (DCT), highlighting their key characteristics:

DCT Type 1 corresponds to an even extension of the input at both boundaries and is defined for input arrays longer than one element.

DCT Type 2, the most commonly used variant, features even symmetry around sample points and is widely applied in image compression such as JPEG and audio processing.

DCT Type 3 serves as the inverse of DCT Type 2 unnormalized and is used to reconstruct signals, with odd symmetry around sample points.

DCT Type 4 is an orthogonal variant with even symmetry around half-sample points, often used in applications requiring orthogonality. Each type has unique symmetry properties and use cases, making them suitable for different signal processing tasks.

Figure 12-5 displays the set of eight basis functions for a Type 2 Discrete Cosine Transform (DCT-II) with a length of N=8. The DCT operates by representing an input signal as a weighted sum of these fundamental cosine waveforms. DCT-II is the most commonly used variant of this transform and is the default type calculated by the scipy.fft.dct function.

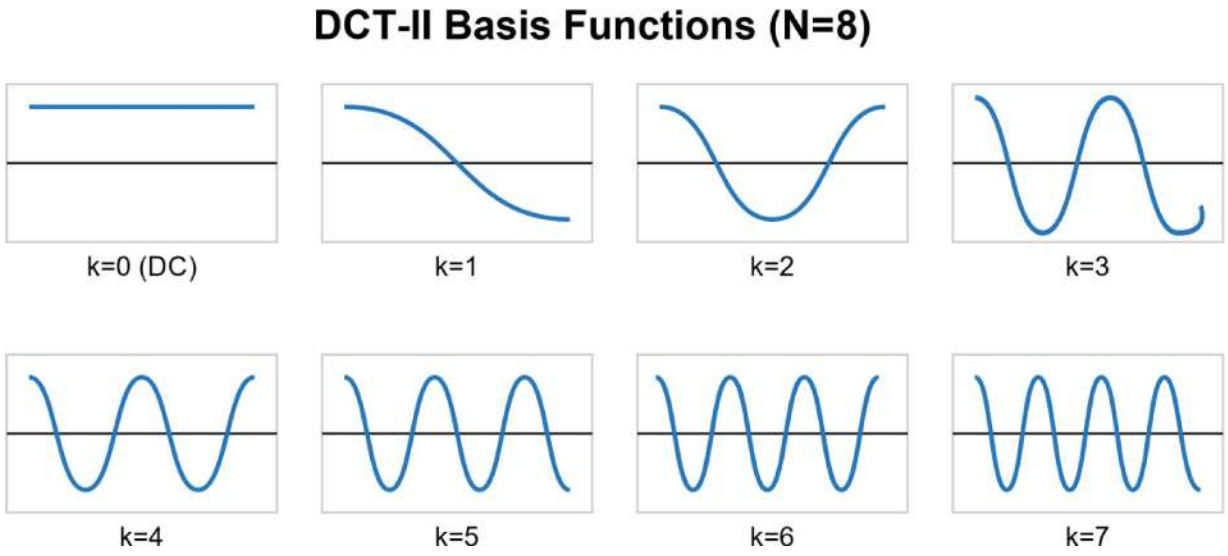


Figure 12-5 The eight DCT-II basis functions for a transform length of N = 8

The basis functions, indexed by k, are essentially cosine waves of increasing frequency:

- The **k=0** basis function is a flat, constant line. It represents the average value, or "DC component," of the entire signal.
- As the index **k** increases from 1 to 7, the basis functions become cosine waves of progressively higher frequencies.

When you apply the scipy.fft.dct() function to a signal, the output is a set of coefficients. Each coefficient represents the amplitude or "strength" of the corresponding basis function shown in Figure 12-5 that is present in the original signal. This transform is particularly effective at concentrating most of a signal's energy into the first few low-frequency coefficients, a property that makes it extremely valuable for compression algorithms like JPEG.

Table 12-9 describes the four types of Discrete Cosine Transforms, each characterized by its symmetry properties and commonly used in specific applications such as signal processing, image compression, and reconstruction.

Table 12-9 Comparison of the four types of Discrete Cosine Transforms (DCTs) available in SciPy

The following example demonstrates how to compute the Discrete Cosine Transform (DCT) and Fast Fourier Transform (FFT).

It starts by defining an input array x and computes its FFT, taking only the real part of the result since the input is real-valued. Next, it calculates the DCT for different types (1, 2, 3, and 4) using the dct function.

DCT Type 1 is applied to the first half of the array in this example only to illustrate a different input length; the transform itself does not require an even-symmetric input. Types 2, 3, and 4 use the full array, with Type 2 as the default:

```
from scipy.fft import fft, dct
import numpy as np
```

```
# Example input array.
x = np.array([4., 3., 5., 10., 5., 3.])

# Compute the FFT of the input array
fft_result = fft(x).real  # Take the real part of
the FFT result.
print("FFT Result (real part):", fft_result)

# Compute the DCT (Type 1) of the input array
# DCT Type I accepts any input array with length
greater than 1.
dct_result_type1 = dct(x[:4], type=1)  # Use the
first half of the input array.
print("DCT Type 1 Result:", dct_result_type1)

# Compute the DCT (Type 2) of the input array.
dct_result_type2 = dct(x, type=2)  # Default DCT
type.
print("DCT Type 2 Result:", dct_result_type2)

# Compute the DCT (Type 3) of the input array.
dct_result_type3 = dct(x, type=3)  # Inverse DCT
type.
print("DCT Type 3 Result:", dct_result_type3)

# Compute the DCT (Type 4) of the input array.
dct_result_type4 = dct(x, type=4)  # DCT Type 4
print("DCT Type 4 Result:", dct_result_type4)
```

The results are printed to show the transformed arrays, illustrating the differences and similarities between the FFT and DCT outputs:

```
FFT Result (real part): [30. -8.  6. -2.  6. -8.]
DCT Type 1 Result: [30. -8.  6. -2.]
```

```
DCT Type 2 Result: [ 60. -3.48476592 -13.85640646
11.3137085 6. -6.31319305]
DCT Type 3 Result: [ 39.15085889
-20.14213562  -6.45392043   7.13341236   8.14213562
  -3.83035081]
DCT Type 4 Result: [ 38.19358955
-24.15218404   4.41711757   9.26349518   0.81975981
  -7.74121742]
```

Fast Hankel Transforms

The Fast Hankel Transform, or FHT, is a computational method used to efficiently approximate the continuous Hankel transform, which is particularly useful for problems involving radial or cylindrical symmetry.

This transform is widely applied in fields such as astrophysics for optics for modeling diffraction patterns in circular systems, medical imaging for processing ultrasound data, and seismology for studying wave propagation in cylindrical structures.

The FHT simplifies the Hankel transform by using a special way of sampling the input and output data, spreading them out on a logarithmic scale. This turns the problem into a convolution, which can be solved quickly with the Fast Fourier Transform (FFT). To keep the results accurate, the method fine-tunes the output grid using techniques like the low-ringing condition, which can be adjusted with the fhtoffset function. There is also an Inverse FHT that can bring the transformed data back to its original form.

Unlike the conventional Fourier Transform, which decomposes signals into sinusoidal components, the Hankel transform utilizes Bessel functions, making it more appropriate for axisymmetric problems like heat diffusion in circular objects or acoustic waves in cylindrical structures. Its effectiveness and precision make the FHT an essential tool in scientific computing, engineering simulations, and data analysis, particularly in cases where radial symmetry is a significant factor.

Figure 12-6 contrasts the fundamental building blocks or basis functions that are used by the Fast Fourier Transform (FFT) and the Fast Hankel Transform (FHT). A transform analyzes a complex signal by decomposing it into a sum of these simpler basis functions. The appropriate transform to use depends on the geometry of the problem being studied.

- *FFT Basis (Sinusoids)*: The left panel shows that the FFT uses sinusoids (sine and cosine waves) as its basis. The Fourier Transform is designed to decompose signals into these sinusoidal components. This makes it perfectly suited for analyzing signals that evolve over time or are described in standard Cartesian coordinates.
- *Hankel Basis (Bessel J_0)*: The right panel shows that the Hankel transform uses Bessel functions as its basis. The text explicitly states that "the Hankel transform utilizes Bessel functions," which makes it more appropriate for problems that have "radial or cylindrical symmetry." Unlike the uniformly repeating sinusoids, Bessel functions oscillate with a decaying amplitude as the radius increases, which is ideal for modeling physical phenomena that spread outward from a central point, like waves in a pond or heat diffusion in a circular disk.

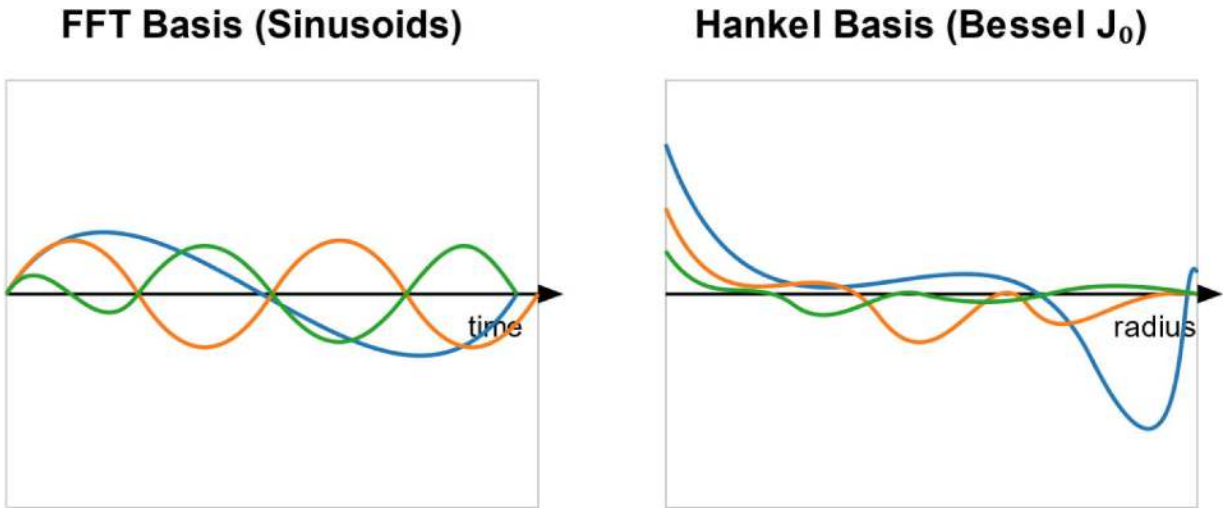


Figure 12-6 Sinusoidal Fourier basis functions and the Bessel-function kernel of the Hankel transform

Both of these powerful transforms are available in the scipy.fft subpackage, implemented in functions like fft and fht. This visualization provides a clear, intuitive reason for choosing one over the other: For a typical time-series signal, the FFT is the standard choice. However, for a problem defined by a radius and exhibiting circular symmetry, the Hankel transform is the more natural and efficient tool.

The scipy.fft subpackage supports Fast Hankel Transform operations with two functions, which are the transform and its inverse.

Table 12-10 Functions for the Fast Hankel Transform (FHT) and its inverse

The function signature of the Fast Hankel Transform is

```
fht(a, dln, mu, offset=0.0, bias=0.0)
```

The Inverse Fast Hankel Transform uses a similar signature. The only difference between them is the letter written in capital letters.

Table 12-11 demonstrates the parameters used in this function.

Table 12-11 Parameters and descriptions for the scipy.fft.fht function

The default value will be used for a parameter if no value is provided when calling the function, such as 0.0 for offset and bias.

The example code demonstrates the Fast Hankel Transform (FHT) using scipy.fft.fht to analyze a radially symmetric power-law function ($a = r^{-1.3}$) defined on a logarithmic radial grid from 0.01 to 10, which is useful for studying phenomena like galaxy light profiles or acoustic waves in spherical systems.

The transform is computed with $\mu = 0.5$ (appropriate for spherical symmetry) by leveraging logarithmic spacing dln to convert the problem into a convolution that can be efficiently solved via FFT, while the inverse transform ifht verifies accuracy by reconstructing the original signal:

```
import numpy as np
from scipy.fft import fht, ifht
import matplotlib.pyplot as plt

# 1. Create a logarithmically spaced input signal
(e.g., a power-law function)
n = 512                          # Number of points
r = np.logspace(-2, 1, n)        # Logarithmic
radial coordinates
a = r**-1.3                      # Example power-
law function (r^-1.3)

# 2. Compute the Fast Hankel Transform using the
SciPy function
dln = np.log(r[1]/r[0])          # Required log
spacing parameter
mu = 0.5                         # Order of the
transform (Bessel function order)
fht_result = fht(a, dln, mu)

# 3. Compute the inverse transform to verify the
result
reconstructed = ifht(fht_result, dln, mu)

# 4. Plot the results to show the original,
reconstructed, and transformed signals
plt.figure(figsize=(12, 5))

# Plot Original vs. Reconstructed
plt.subplot(1, 2, 1)
plt.loglog(r, a, label='Original Function',
linewidth=2)
plt.loglog(r, reconstructed, '--',
label='Reconstructed via ifht', color='orange')
plt.title('Original vs. Reconstructed Signal')
```

```
plt.xlabel('r (Spatial Domain)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True, which="both", ls="--")

# Plot the Hankel Transform result
plt.subplot(1, 2, 2)
plt.loglog(r, fht_result, 'r-', label='FHT
Result')
plt.title('Fast Hankel Transform')
plt.xlabel('k (Frequency Domain)')
plt.ylabel('Transformed Amplitude')
plt.grid(True, which="both", ls="--")

plt.tight_layout()
plt.show()
```

The results are visualized in two plots: one comparing the original and reconstructed $r^{-1.3}$ curves, validating the transform's precision, and another showing the transformed spectrum, which helps identify hidden patterns in the data. This approach is particularly valuable in astrophysics for analyzing density distributions, in engineering for studying wave propagation in cylindrical structures, and in physics for solving problems with radial symmetry where traditional Fourier Transforms would be less effective.

The output graph of this code is shown below.

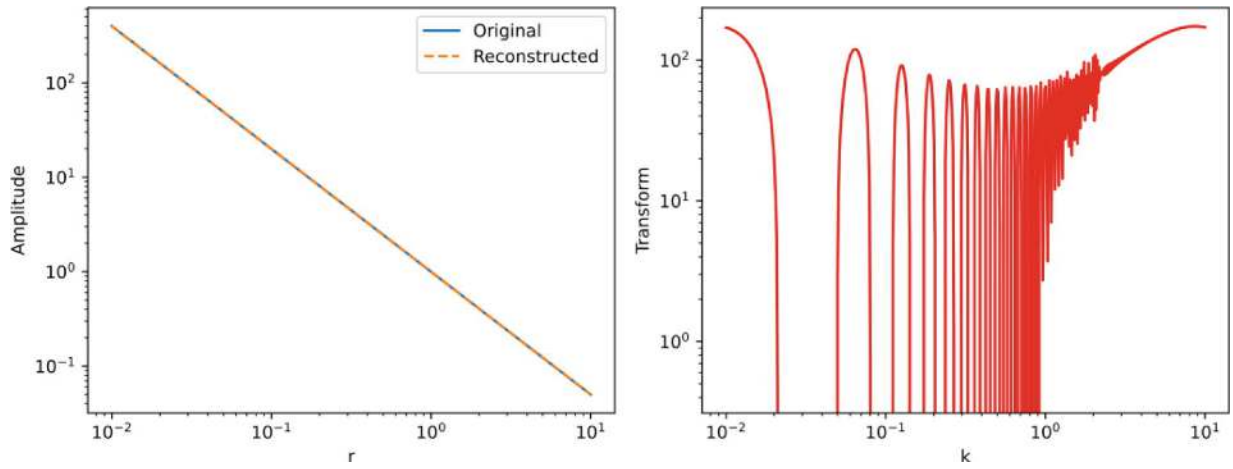


Figure 12-7 Fast Hankel transform of a power-law input and reconstruction using the inverse transform

Helper Functions

Helper functions are useful tools designed to support and optimize Fourier and related transforms, like the Hankel transform. They handle tasks such as frequency mapping, array rearrangement, performance tuning, and parallel processing. These functions serve three main goals.

First, they provide clarity by automatically determining frequency values for FFT results through functions like fftfreq and rfftfreq, removing the need for manual calculations and preventing mistakes when interpreting data.

Second, they enhance performance by including tools such as next fast len, which finds the most efficient array size for faster computations, and set workers, which enables parallel processing to handle large datasets more quickly.

Third, they ensure accuracy by offering solutions like fhtoffset that reduces errors in Hankel transforms for circular data patterns and fftshift that properly positions frequency components for correct visual representation.

These useful functions are commonly applied in different technical areas, such as analyzing sound and vibrations in signal processing, reconstructing medical images, studying space data in astrophysics, and performing fast mathematical calculations in machine learning.

They play an important role in both scientific research and real-world engineering solutions.

These functions are divided and described in the table below.

Table 12-12 Summary of SciPy FFT helper functions, their descriptions, and key use cases

The next code performs a frequency analysis of a noisy 60 Hz sine wave signal using the Fast Fourier Transform (FFT) with helper functions.

First, it generates a one-second test signal containing a 60 Hz sine wave (common in power systems) mixed with Gaussian noise, sampled at 1,000 Hz. The next_fast_len function optimizes the FFT computation by finding the most efficient array size, while fftfreq creates the correct frequency axis labels. The FFT result is centered using fftshift for intuitive visualization, with 0 Hz at the center revealing the dominant 60 Hz component amid noise:

```
import numpy as np
from scipy.fft import fft, fftfreq, fftshift,
next_fast_len
import matplotlib.pyplot as plt

# Generate a signal: 60Hz sine wave + noise
fs = 1000  # Sampling frequency (Hz)
t = np.linspace(0, 1, fs, endpoint=False)  # 1
second duration
signal = np.sin(2 * np.pi * 60 * t) + 0.5 *
np.random.randn(fs)  # 60Hz + noise

# Optimize array size for FFT
n = next_fast_len(len(signal))  # Get efficient
FFT size (fixed extra parenthesis)

# Compute FFT and prepare frequency axis
fft_result = fft(signal, n=n)
```

```
freqs = fftfreq(n, d=1/fs)           # Get proper
frequency bins
fft_shifted = fftshift(fft_result)  # Center the
spectrum
freqs_shifted = fftshift(freqs)     # Center
frequency axis

# Plot
plt.figure(figsize=(10, 4))
plt.plot(freqs_shifted, np.abs(fft_shifted))
plt.title("Frequency Spectrum (60Hz signal +
noise)")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.xlim(-100, 100)          # Zoom around our
60Hz signal
plt.grid()
plt.show()
```

The plot will show a peak at ±60 Hz centered via fftshift, representing the clean 60 Hz sine wave, with random noise visible across other frequencies. The xlim(-100, 100) focuses the view around the 60 Hz component.

This is a common analysis for power systems, as 60 Hz is the standard AC frequency in many countries.

This code analyzes a wavy electrical signal (like from a power outlet) mixed with random noise. It uses special math tools (FFT) to break down the signal and find its hidden frequencies. The graph shows the results, with a clear spike at 60 Hz—proving the signal's main frequency—while the noise appears as smaller bumps. The helper functions make this easy by

- Labeling frequencies correctly (fftfreq)
- Centering the graph for clarity (fftshift)
- Speeding up calculations (next_fast_len)

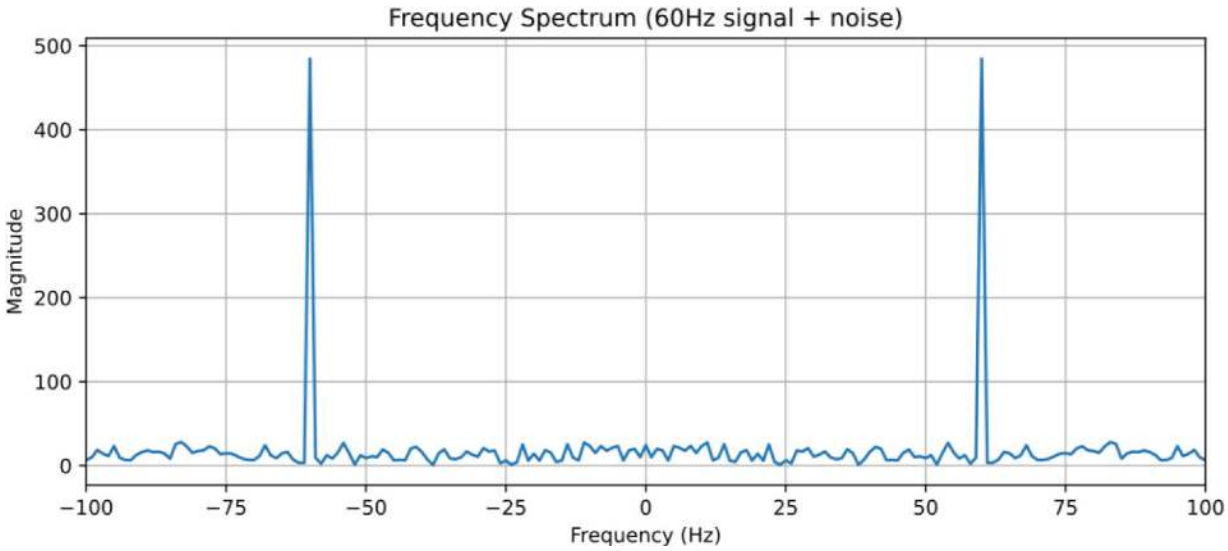


Figure 12-8 Centered Fourier magnitude spectrum of a noisy 60 Hz signal, with peaks at -60 Hz and +60 Hz

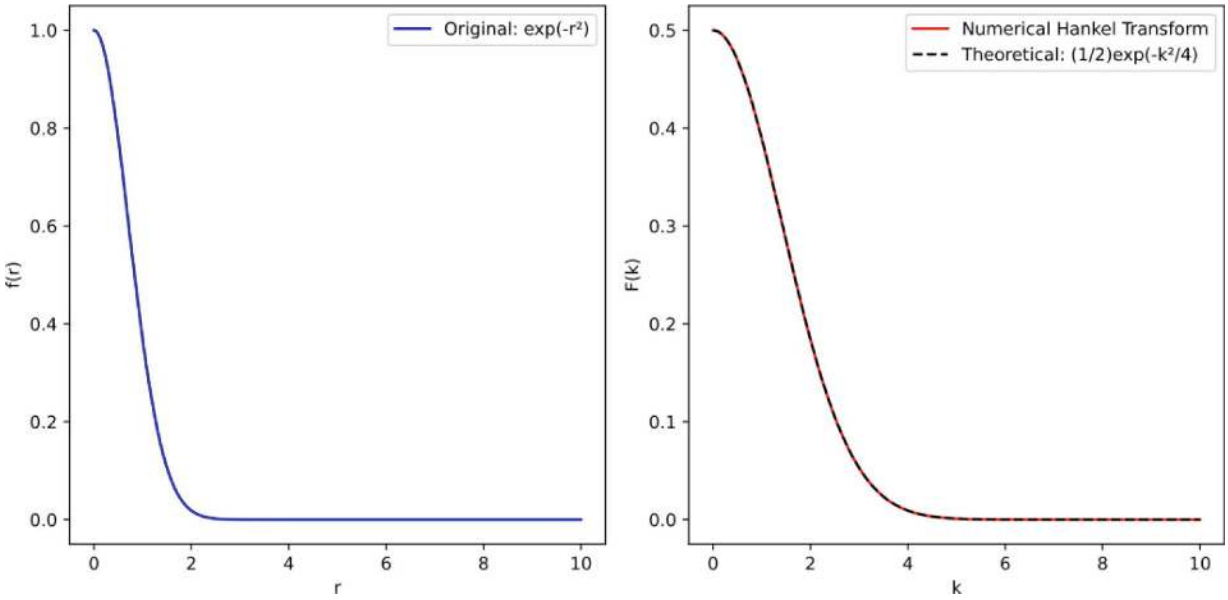


Figure 12-9 Gaussian radial function and comparison of its numerical and analytical zeroth-order Hankel transforms

Centering Frequencies with fftshift and ifftshift

The helper functions in scipy.fft assist in managing and optimizing Fourier and related transforms by providing essential utilities for frequency-domain analysis, spectral manipulation, and computational efficiency. These functions play a crucial role in signal processing, image analysis, and scientific computing, where Fourier Transforms are

widely used. These utility functions improve the functionality, speed, and precision of Fourier-based transformations in scientific and engineering fields.

Functions like fftshift and ifftshift are particularly helpful for repositioning the zero-frequency component in spectral data, making it easier to interpret symmetric frequency distributions in both 1D signals and 2D images.

The functions fftfreq and rfftfreq assist in mapping Discrete Fourier Transform (DFT) coefficients to their corresponding physical frequencies, which is crucial for applications such as spectral analysis, filtering, and feature extraction.

Additionally, next_fast_len and prev_fast_len enhance FFT performance by determining optimal array sizes that take advantage of efficient radix-based algorithms, thereby reducing computation time when zero-padding or truncating data.

The fhtoffset function adjusts parameters for Hankel transforms, which are commonly used in polar coordinate systems and various physics-related computations.

Lastly, set_workers and get_workers manage parallel processing, improving efficiency in large-scale FFT operations.

Table 12-13 Summary of helper functions in scipy.fft for optimizing and interpreting transform results

The function signature of the fftshift is represented below:

```
fftshift(x, axes=None)
```

Details of parameters can be seen in Table 12-14.

Table 12-14 Parameters for the scipy.fft.fftshift function

The following example visualizes the Fast Fourier Transform (FFT) of a composite signal in two different ways. It first creates a time array

of 1,000 points over one second and then generates a signal by combining a 10 Hz sine wave with a half-amplitude 25 Hz sine wave.

After calculating the FFT of this signal, the code creates a figure with two subplots: the left subplot shows the raw FFT with frequency components arranged with zero frequency at the left edge and both positive and negative frequencies spread out from there, while the right subplot shows the centered FFT using fftshift() to rearrange the output so that zero frequency appears in the center, making the spectrum more intuitive to interpret:

```
# Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
from scipy.fft import fft, fftshift

# Generate a time array from 0 to 1 second with
1000 samples
t = np.linspace(0, 1, 1000, endpoint=False)
# Create a signal composed of two sine waves with
frequencies 10 Hz and 25 Hz
sig = np.sin(2*np.pi*10*t) +
0.5*np.sin(2*np.pi*25*t)

# Apply the Fast Fourier Transform (FFT) to the
signal
fft_vals = fft(sig)
# Create a new figure with a specified size
plt.figure(figsize=(10,4))
# Plot the raw FFT output
plt.subplot(121)
plt.plot(np.abs(fft_vals))
plt.title("Raw FFT")
# Plot the centered FFT output using fftshift
plt.subplot(122)
plt.plot(fftshift(np.abs(fft_vals)))
```

```
plt.title("Centered FFT")
# Display the plot instead of saving it
plt.tight_layout()
plt.show()
```

The code opens a window to visualize the results using Matplotlib to display the magnitude of the frequency components in both formats.

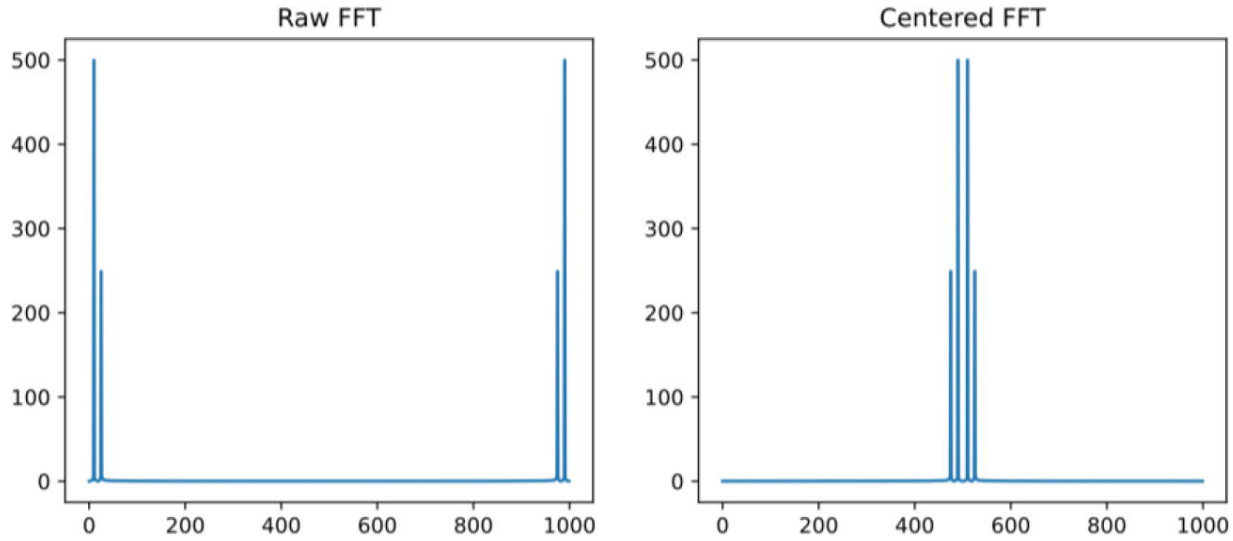


Figure 12-10 Fourier magnitude spectra before and after rearranging frequency bins with fftshift

Analysis of the Frequency Spectrum with fftfreq

To use the fftfreq function, use its signature according to the syntax pattern below:

```
fftfreq(n, d=1.0, *, xp=None, device=None)
```

The parameters are listed in Table 12-15.

Table 12-15 Parameters for the scipy.fft.fftfreq function

The example code with fftfreq generates and analyzes a synthetic signal composed of two sine waves 10 Hz and 25 Hz using the Fast Fourier Transform (FFT) to visualize its frequency spectrum.

It first creates a one-second time series sampled at 1,000 Hz and then computes the FFT to transform the time-domain signal into frequency-domain components. The resulting frequencies and their corresponding magnitudes, absolute values of the FFT output, are plotted, showing peaks at 10 Hz and 25 Hz, with the latter having half the amplitude 0.5 of the former, as specified in the signal generation. The plot is limited to the Nyquist frequency 500 Hz to avoid displaying redundant negative frequencies:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.fft import fft, fftfreq

fs = 1000  # Sampling frequency
t = np.linspace(0, 1, fs, endpoint=False)
sig = np.sin(2*np.pi*10*t) +
0.5*np.sin(2*np.pi*25*t)

fft_vals = fft(sig)
freqs = fftfreq(len(sig), 1/fs)

plt.figure(figsize=(8,4))
plt.plot(freqs[:len(freqs)//2],
np.abs(fft_vals[:len(freqs)//2]))
plt.title("FFT Spectrum")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.show()
```

The generated graph is a frequency spectrum displaying the magnitude of the signal's frequency components. The synthesized signal consists of two sine waves at 10 Hz and 25 Hz. This graph confirms that the FFT correctly identifies these frequencies and their relative strengths. The stronger peak at 10 Hz has a magnitude of approximately 500, corresponding to the dominant sine wave

(np.sin(2π*10*t)). The smaller peak at 25 Hz, with a magnitude of about 250, corresponds to the second sine wave (0.5*np.sin(2π*25*t)), which has half the amplitude of the first.

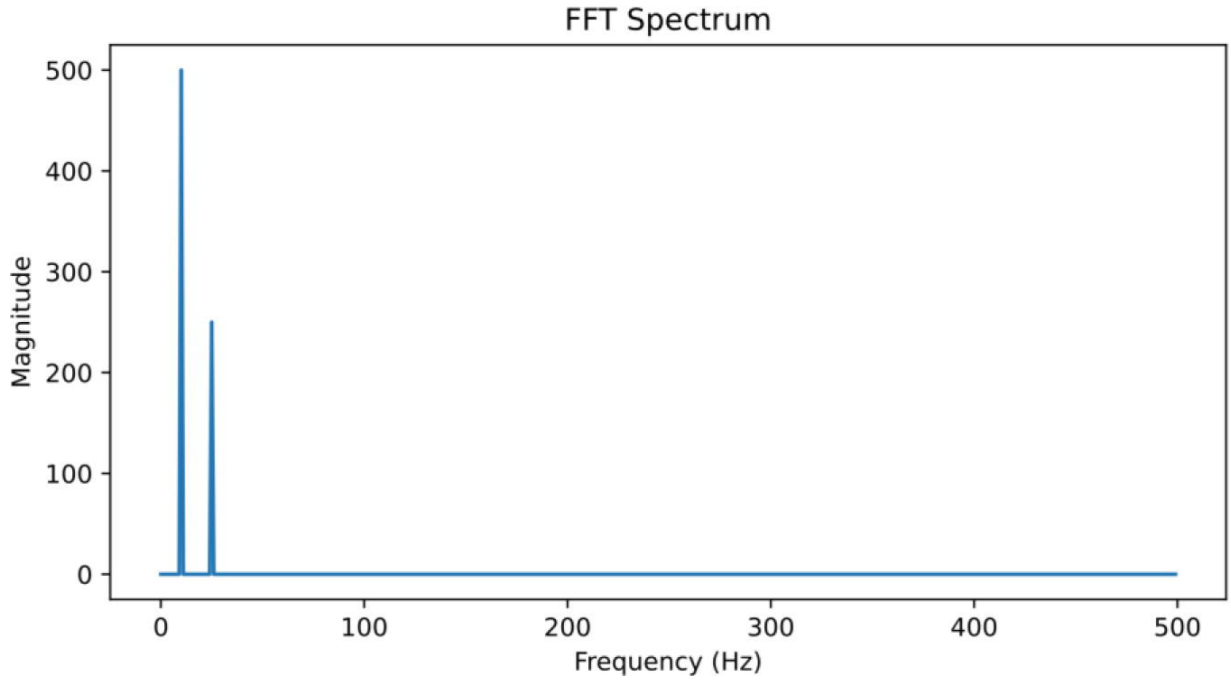


Figure 12-11 Fourier magnitude spectrum of a signal containing 10 Hz and 25 Hz sinusoidal components

Backend Control

The scipy.fft backend control functions allow to manage and switch between different Fast Fourier Transform backends for optimized performance or compatibility. The Fast Fourier Transforms are computed by selecting different computational engines for optimized performance or compatibility.

The set_backend() function allows temporary switching between different computational engines like numpy, scipy, or pyfftw within specific code blocks. For cases where certain backends should be avoided, the skip_backend() function can exclude problematic implementations like “mkl_fft” when needed.

When permanent configuration changes are required, set_global_backend() establishes a new default choice among supported options such as scipy or pyfftw. The register_backend()

function extends functionality by adding custom implementations including GPU-accelerated options like CuPy. These controls are particularly valuable in scientific computing where different scenarios may call for specific backends—numpy for compatibility, pyfftw for performance, or specialized options for hardware acceleration.

Figure 12-12 illustrates the flexible and modular backend architecture of the scipy.fft subpackage. It shows that the functions a user calls are part of a high-level interface that can delegate the actual intensive computations to one of several underlying libraries, known as "backends." This architecture allows users to optimize for performance or compatibility based on their specific needs.

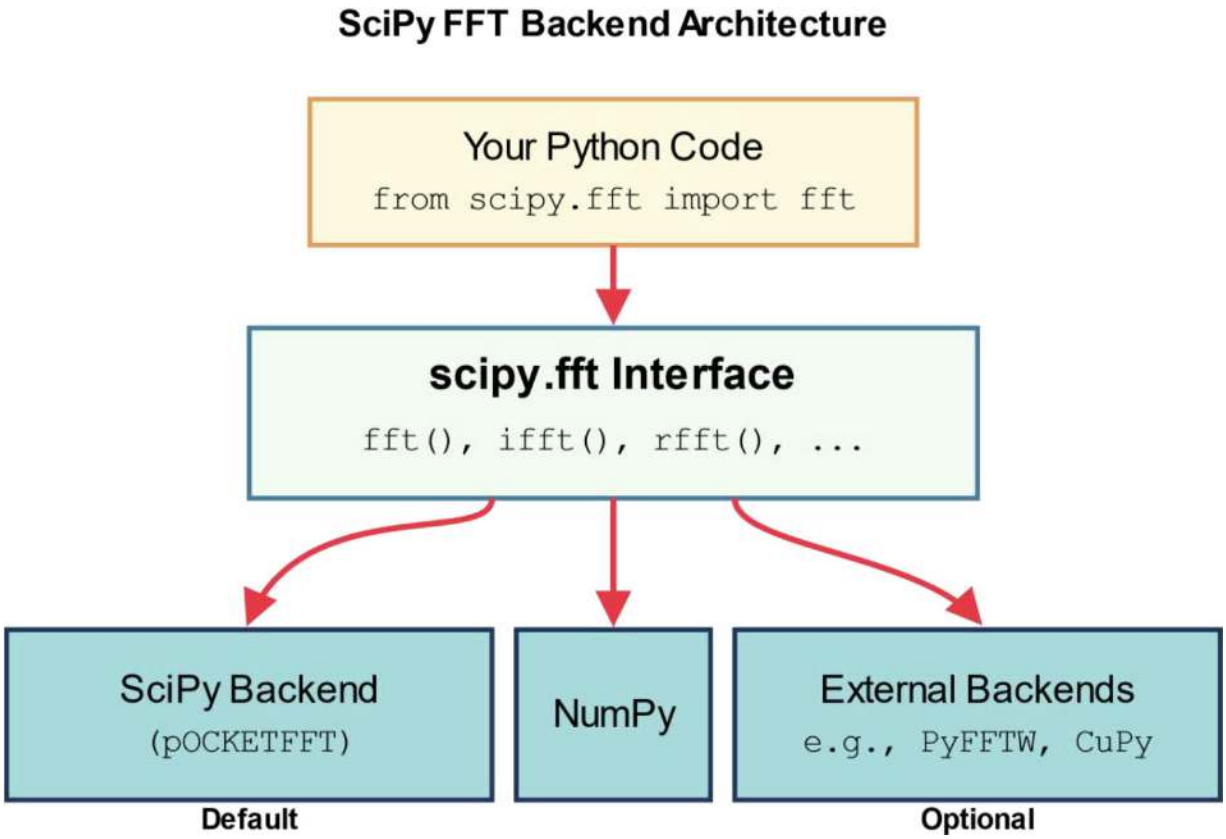


Figure 12-12 Architecture of the scipy.fft interface and its computational backends

The data flow shown is as follows:

1. *Your Python Code*: The process begins when a user imports and calls a standard function from the scipy.fft library, such as fft().

2. *scipy.fft Interface*: The call is first handled by a consistent public interface. This layer ensures that the function calls remain the same for the user, regardless of which backend is performing the computation.
3. *Backends*: The interface then dispatches the task to a specific backend. The options include

   SciPy Backend (pOCKETFFT): The default, built-in FFT implementation that comes with SciPy.

   NumPy: The interface can be configured to use the FFT implementation from the NumPy library.

   External Backends: Users can install and register optional, specialized libraries. The text provides examples such as PyFFTW for high-performance computing and CuPy for GPU acceleration.

This ability to switch between computational engines is managed by the "backend control" functions described in the text, such as set_backend(), set_global_backend(), and register_backend().

For example, the function signature for set_backend is shown below, followed by Table 12-16 detailing its parameters:

```
set_backend(backend, coerce=False, only=False)
```

Table 12-16 Parameters for the scipy.fft.set_backend context manager

There are several functions available for backend control.

Table 12-17 Summary of functions used for managing FFT computation backends

The example code below compares two Fast Fourier Transform implementations by generating a test signal containing 50 Hz and 120 Hz sine wave components and then computing and visualizing their frequency spectra side by side.

It first forces SciPy's native FFT backend to process the signal and plots the result on the left subplot and then attempts to use NumPy's backend through SciPy's interface on the right subplot, falling back to direct NumPy FFT if unavailable.

The code uses set_backend() to explicitly select computational backends, creates proper frequency axes in Hz, and displays both spectra with identical scaling for fair comparison.

Error handling ensures the code runs even if the NumPy backend isn't registered, and professional plot styling makes the comparison clear and publication-ready:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import fft

# Generate test signal
fs = 1000  # Sampling frequency
t = np.linspace(0, 1, fs, endpoint=False)
signal = np.sin(2*np.pi*50*t) +
0.5*np.sin(2*np.pi*120*t)  # 50Hz + 120Hz

# Prepare figure
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12,
4))
freqs = np.fft.fftfreq(fs, 1/fs)[:fs//2]

# Compute and plot with SciPy backend
with fft.set_backend('scipy'):  # Force SciPy's
native backend
    fft_sp = np.abs(fft.fft(signal)[:fs//2])  #
Fixed: Added closing parenthesis
    ax1.plot(freqs, fft_sp, 'b-', linewidth=1.5,
label='SciPy Native')
    ax1.set_title('SciPy Native Backend')
    ax1.set_xlabel('Frequency (Hz)')
```

```
    ax1.set_ylabel('Magnitude')
    ax1.grid(True, linestyle=':', alpha=0.7)
    ax1.legend()

# Compute and plot comparison
try:
    with fft.set_backend('numpy'):  # Try to force
NumPy backend
        fft_np = np.abs(fft.fft(signal)
[:fs//2])  # Fixed: Added closing parenthesis
        ax2.plot(freqs, fft_np, 'r-',
linewidth=1.5, label='NumPy Backend')
        ax2.set_title('NumPy Backend via SciPy')
except ValueError:
    fft_np = np.abs(np.fft.fft(signal)[:fs//2])  #
Fallback to direct NumPy
    ax2.plot(freqs, fft_np, 'r-', linewidth=1.5,
label='Direct NumPy')
    ax2.set_title('Direct NumPy Implementation')

ax2.set_xlabel('Frequency (Hz)')
ax2.grid(True, linestyle=':', alpha=0.7)
ax2.legend()

plt.suptitle('FFT Backend Comparison (50Hz + 120Hz
signal)')
plt.tight_layout()
plt.show()
```

The resulting plot demonstrates that both implementations produce identical frequency peaks at 50 Hz and 120 Hz, verifying their mathematical equivalence while showcasing SciPy's backend control system.

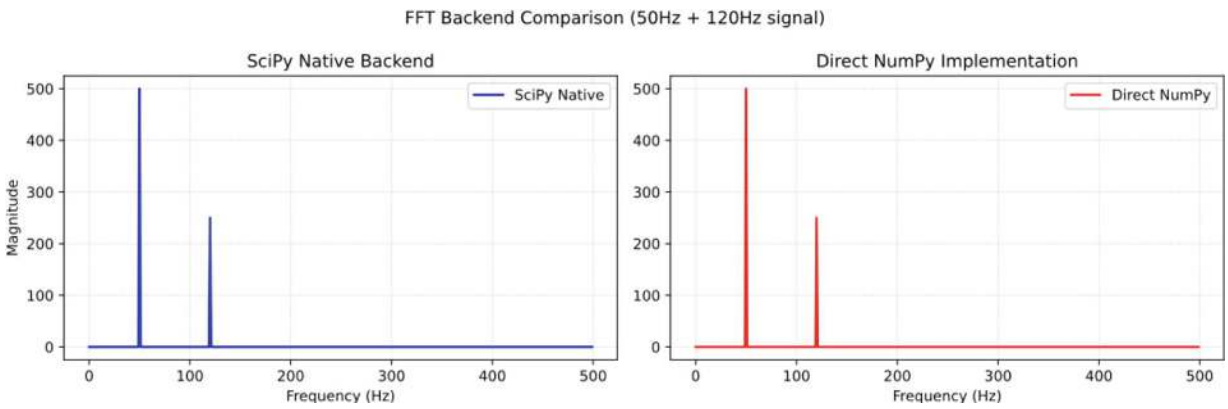


Figure 12-13 Comparison of Fourier magnitude spectra computed with SciPy and a direct NumPy implementation for a signal containing 50 Hz and 120 Hz components

Table 12-18 Common applications of Fast Fourier Transform operations

Table 12-19 Comprehensive list of FFT functions in scipy.fft

```
import numpy as np
import scipy.fft

# 1. fft - Compute the 1D Discrete Fourier
Transform
x = np.array([1, 2, 1, -1, 1.5])
X = scipy.fft.fft(x)
print("fft:", X)
# Output: [ 4.5       +0.j          2.08155948-
1.65109876j
#          -1.83155948+1.60822041j -1.83155948-
1.60822041j
#           2.08155948+1.65109876j]

# 2. ifft - Compute the 1D Inverse Discrete
Fourier Transform
x_recovered = scipy.fft.ifft(X)
print("ifft:", x_recovered)
```

```
# Output: [ 1. +0.j  2. +0.j  1. +0.j -1.
+0.j  1.5+0.j]

# 3. fft2 - Compute the 2D Discrete Fourier
Transform
x2 = np.array([[1, 2], [3, 4]])
X2 = scipy.fft.fft2(x2)
print("fft2:", X2)
# Output: [[10.+0.j -2.+0.j]
#          [-4.+0.j  0.+0.j]]

# 4. ifft2 - Compute the 2D Inverse Discrete
Fourier Transform
x2_recovered = scipy.fft.ifft2(X2)
print("ifft2:", x2_recovered)
# Output: [[1.+0.j 2.+0.j]
#          [3.+0.j 4.+0.j]]

# 5. fftn - Compute the N-Dimensional Discrete
Fourier Transform
x3 = np.random.rand(3, 3, 3)
X3 = scipy.fft.fftn(x3)
print("fftn:", X3)
# Output: Complex values in a 3x3x3 matrix

# 6. ifftn - Compute the N-Dimensional Inverse
Discrete Fourier Transform
x3_recovered = scipy.fft.ifftn(X3)
print("ifftn:", x3_recovered)
# Output: Approximately the original x3 values

# 7. rfft - Compute the 1D Discrete Fourier
Transform for real input
xr = np.array([1, 2, 3, 4, 5])
Xr = scipy.fft.rfft(xr)
```

```
print("rfft:", Xr)
# Output: [15. +0.j -2.5+3.44095j -2.5+0.8123j]

# 8. irfft - Compute the inverse of rfft
xr_recovered = scipy.fft.irfft(Xr, n=len(xr))
print("irfft:", xr_recovered)
# Output: [1. 2. 3. 4. 5.]

# 9. rfft2 - Compute the 2D Discrete Fourier
Transform for real input
xr2 = np.array([[1, 2], [3, 4]])
Xr2 = scipy.fft.rfft2(xr2)
print("rfft2:", Xr2)
# Output: [[10.+0.j -2.+0.j]
#          [-4.+0.j  0.+0.j]]

# 10. irfft2 - Compute the inverse of rfft2
xr2_recovered = scipy.fft.irfft2(Xr2, s=xr2.shape)
print("irfft2:", xr2_recovered)
# Output: [[1. 2.]
#          [3. 4.]]

# 11. rfftn - Compute the N-Dimensional Discrete
Fourier Transform for real input
xr3 = np.random.rand(3, 3, 3)
Xr3 = scipy.fft.rfftn(xr3)
print("rfftn:", Xr3)
# Output shape: (3, 3, 2)

# 12. irfftn - Compute the inverse of rfftn
xr3_recovered = scipy.fft.irfftn(Xr3, s=xr3.shape)
print("irfftn:", xr3_recovered)
# Output: Approximately the original xr3 values
```

```
# 13. hfft - Compute the FFT of a signal with
Hermitian symmetry (real spectrum)
xh = np.array([1, 2, 3, 2, 1])
Xh = scipy.fft.hfft(xh)
print("hfft:", Xh)
# Output: Real-valued frequency spectrum

# 14. ihfft - Compute the inverse FFT of a signal
with Hermitian symmetry
xh_recovered = scipy.fft.ihfft(Xh)
print("ihfft:", xh_recovered)
# Output: Approximately the original xh values

# 15. hfft2 - Compute the 2D FFT of a Hermitian
complex array
xh2 = np.array([[1, 2], [2, 1]])
Xh2 = scipy.fft.hfft2(xh2)
print("hfft2:", Xh2)
# Output: Real-valued frequency domain
representation

# 16. ihfft2 - Compute the 2D inverse FFT of a
real spectrum
xh2_recovered = scipy.fft.ihfft2(Xh2, s=xh2.shape)
print("ihfft2:", xh2_recovered)
# Output: Approximately the original xh2 values

# 17. hfftn - Compute the N-D FFT of Hermitian
symmetric complex input
xh3 = np.random.rand(3, 3, 3)
Xh3 = scipy.fft.hfftn(xh3)
print("hfftn:", Xh3)
# Output: Real-valued frequency domain
representation
```

```
# 18. ihfftn - Compute the N-D inverse Discrete
Fourier Transform for a real spectrum
xh3_recovered = scipy.fft.ihfftn(Xh3, s=xh3.shape)
print("ihfftn:", xh3_recovered)
# Output: Approximately the original xh3 values

import numpy as np
from scipy.fft import dct, idct, dctn, idctn, dst,
idst, dstn, idstn, fht, ifht

# 1. dct - Discrete Cosine Transform
x = np.array([1, 2, 3, 4, 5])
X_dct = dct(x, type=2, norm='ortho')
print("dct:", X_dct)
# Output: Transformed coefficients in the DCT
domain

# 2. idct - Inverse Discrete Cosine Transform
x_recovered_dct = idct(X_dct, type=2,
norm='ortho')
print("idct:", x_recovered_dct)
# Output: [1. 2. 3. 4. 5.]

# 3. dctn - Multidimensional Discrete Cosine
Transform
x_dctn = np.array([[1, 2], [3, 4]])
X_dctn = dctn(x_dctn, type=2, norm='ortho')
print("dctn:", X_dctn)
# Output: 2D DCT coefficients

# 4. idctn - Inverse Multidimensional Discrete
Cosine Transform
x_recovered_dctn = idctn(X_dctn, type=2,
norm='ortho')
print("idctn:", x_recovered_dctn)
```

```
# Output: [[1. 2.]
#          [3. 4.]]

# 5. dst - Discrete Sine Transform
X_dst = dst(x, type=2, norm='ortho')
print("dst:", X_dst)
# Output: Transformed coefficients in the DST
domain

# 6. idst - Inverse Discrete Sine Transform
x_recovered_dst = idst(X_dst, type=2,
norm='ortho')
print("idst:", x_recovered_dst)
# Output: [1. 2. 3. 4. 5.]

# 7. dstn - Multidimensional Discrete Sine
Transform
X_dstn = dstn(x_dctn, type=2, norm='ortho')
print("dstn:", X_dstn)
# Output: 2D DST coefficients

# 8. idstn - Inverse Multidimensional Discrete
Sine Transform
x_recovered_dstn = idstn(X_dstn, type=2,
norm='ortho')
print("idstn:", x_recovered_dstn)
# Output: [[1. 2.]
#          [3. 4.]]

# 9. fht - Fast Hankel Transform
a = np.array([1, 2, 3, 4, 5])
dln = 0.1  # Logarithmic step size
mu = 0      # Order of transform
A_fht = fht(a, dln, mu)
print("fht:", A_fht)
```

```
# Output: Fast Hankel Transform coefficients

# 10. ifht - Inverse Fast Hankel Transform
a_recovered_fht = ifht(A_fht, dln, mu)
print("ifht:", a_recovered_fht)
# Output: [1. 2. 3. 4. 5.]

import numpy as np
from scipy.fft import (
    fftshift, ifftshift, fftfreq, rfftfreq,
fhtoffset,
    next_fast_len, prev_fast_len, set_workers,
get_workers,
    set_backend, skip_backend, set_global_backend,
register_backend
)

# 1. fftshift - Shift zero-frequency component to
the center
x = np.array([0, 1, 2, 3, 4, 5, 6, 7])
X_shifted = fftshift(x)
print("fftshift:", X_shifted)
# Output: [4 5 6 7 0 1 2 3]

# 2. ifftshift - Inverse of fftshift
X_unshifted = ifftshift(X_shifted)
print("ifftshift:", X_unshifted)
# Output: [0 1 2 3 4 5 6 7]

# 3. fftfreq - Compute DFT sample frequencies
n = 8
frequencies = fftfreq(n, d=1.0)
print("fftfreq:", frequencies)
# Output: [ 0.    0.125  0.25  0.375 -0.5  -0.375
-0.25  -0.125]
```

```
# 4. rfftfreq - Compute sample frequencies for
real-input FFT
real_frequencies = rfftfreq(n, d=1.0)
print("rfftfreq:", real_frequencies)
# Output: [0.    0.125 0.25  0.375 0.5  ]

# 5. fhtoffset - Compute optimal offset for fast
Hankel transform
dln = 0.1
mu = 0
offset = fhtoffset(dln, mu)
print("fhtoffset:", offset)
# Output: Optimal offset value

# 6. next_fast_len - Find the next optimal FFT
size
size = 100
fast_size = next_fast_len(size)
print("next_fast_len:", fast_size)
# Output: The next efficient size for FFT
computation

# 7. prev_fast_len - Find the previous optimal FFT
size
prev_size = prev_fast_len(size)
print("prev_fast_len:", prev_size)
# Output: The previous efficient size for FFT
computation

# 8. set_workers - Set default number of workers
for parallel FFT computation
with set_workers(4):
    num_workers = get_workers()
    print("set_workers:", num_workers)
# Output: 4
```

```
# 9. get_workers - Get current number of FFT
workers
num_workers = get_workers()
print("get_workers:", num_workers)
# Output: Current number of FFT workers

# 10. Backend control (set_backend, skip_backend,
set_global_backend, register_backend)
try:
    with set_backend("scipy"):
        print("set_backend: Successfully set to
scipy")
except ValueError:
    print("set_backend: Backend not available")

try:
    with skip_backend("scipy"):
        print("skip_backend: Skipped scipy
backend")
except ValueError:
    print("skip_backend: Backend not available")

try:
    set_global_backend("scipy")
    print("set_global_backend: Successfully set to
scipy")
except ValueError:
    print("set_global_backend: Backend not
available")

try:
    register_backend("scipy")
    print("register_backend: Successfully
registered scipy backend")
except ValueError:
```

```
    print("register_backend: Backend already
registered or not available")
```

FFT-Based Filtering

Here are some examples of using scipy.fft to create several types of filters.

Low-Pass Filter (FFT)

A low-pass filter allows low-frequency signals to pass through while attenuating high-frequency signals.

```
import numpy as np
from scipy.fft import fft, ifft
import matplotlib.pyplot as plt

# Generate a sample signal
t = np.linspace(0, 1, 1000, endpoint=False)
x = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 *
np.pi * 50 * t)

# Perform FFT
X = fft(x)

# Create a low-pass filter
freq = np.fft.fftfreq(len(x), d=0.001)
cutoff_freq = 20
X_filtered = X.copy()
X_filtered[np.abs(freq) > cutoff_freq] = 0

# Perform inverse FFT
x_filtered = ifft(X_filtered).real

# Plot the results
plt.plot(t, x, label='Original signal')
plt.plot(t, x_filtered, label='Filtered signal')
```

```
plt.legend()
plt.show()
```

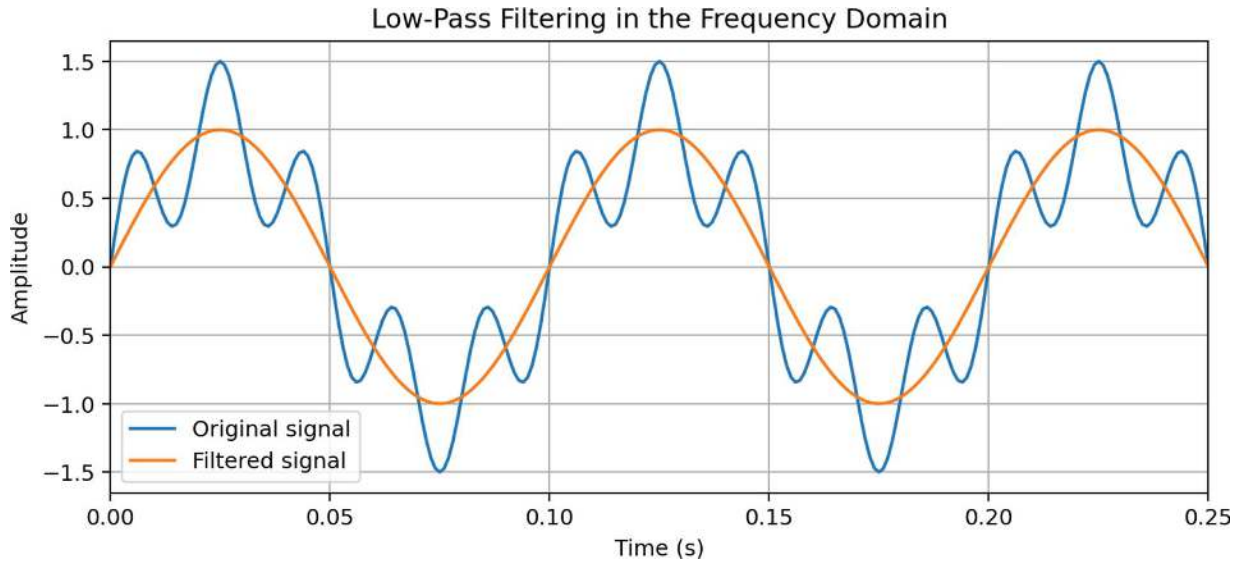


Figure 12-14 Original signal and output of FFT-based low-pass filtering

High-Pass Filter (FFT)

A high-pass filter allows high-frequency signals to pass through while attenuating low-frequency signals.

```
import numpy as np
from scipy.fft import fft, ifft
import matplotlib.pyplot as plt

# Generate a sample signal
t = np.linspace(0, 1, 1000, endpoint=False)
x = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 *
np.pi * 50 * t)

# Perform FFT
X = fft(x)

# Create a high-pass filter
freq = np.fft.fftfreq(len(x), d=0.001)
cutoff_freq = 20
```

```
X_filtered = X.copy()
X_filtered[np.abs(freq) < cutoff_freq] = 0

# Perform inverse FFT
x_filtered = ifft(X_filtered).real

# Plot the results
plt.plot(t, x, label='Original signal')
plt.plot(t, x_filtered, label='Filtered signal')
plt.legend()
plt.show()
```

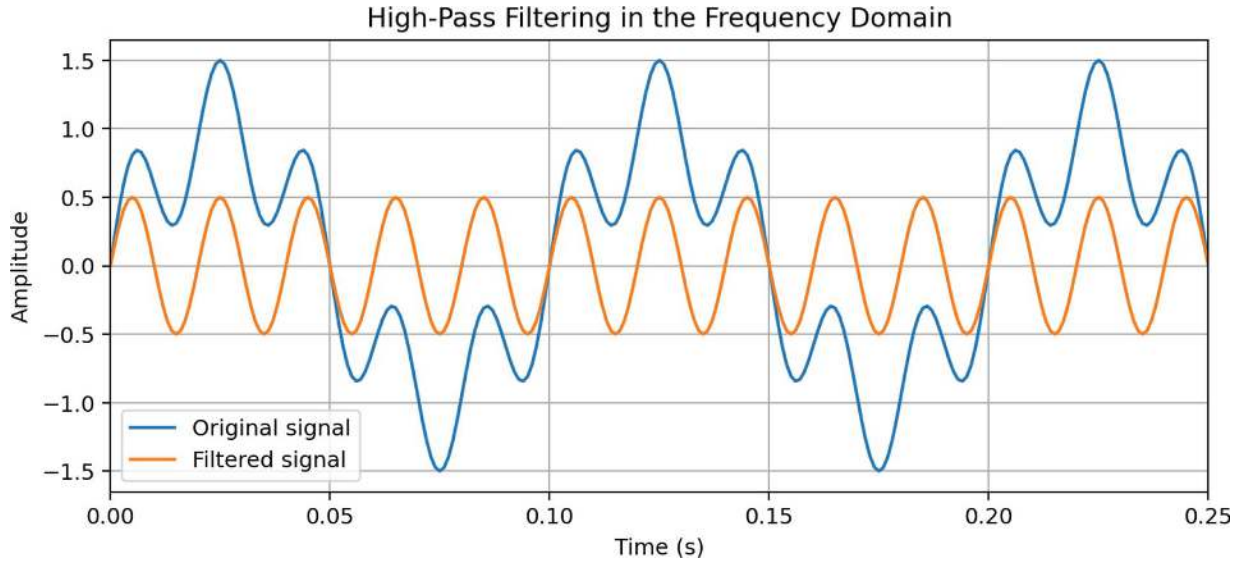


Figure 12-15 Original signal and output of FFT-based high-pass filtering

Band-Pass Filter (FFT)

A band-pass filter allows signals within a specific frequency range to pass through while attenuating signals outside that range.

```
import numpy as np
from scipy.fft import fft, ifft
import matplotlib.pyplot as plt

# Generate a sample signal
t = np.linspace(0, 1, 1000, endpoint=False)
```

```
x = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 *
np.pi * 50 * t)

# Perform FFT
X = fft(x)

# Create a band-pass filter
freq = np.fft.fftfreq(len(x), d=0.001)
cutoff_freq_low = 40
cutoff_freq_high = 60
X_filtered = X.copy()
X_filtered[np.abs(freq) < cutoff_freq_low] = 0
X_filtered[np.abs(freq) > cutoff_freq_high] = 0

# Perform inverse FFT
x_filtered = ifft(X_filtered).real

# Plot the results
plt.plot(t, x, label='Original signal')
plt.plot(t, x_filtered, label='Filtered signal')
plt.legend()
plt.show()
```

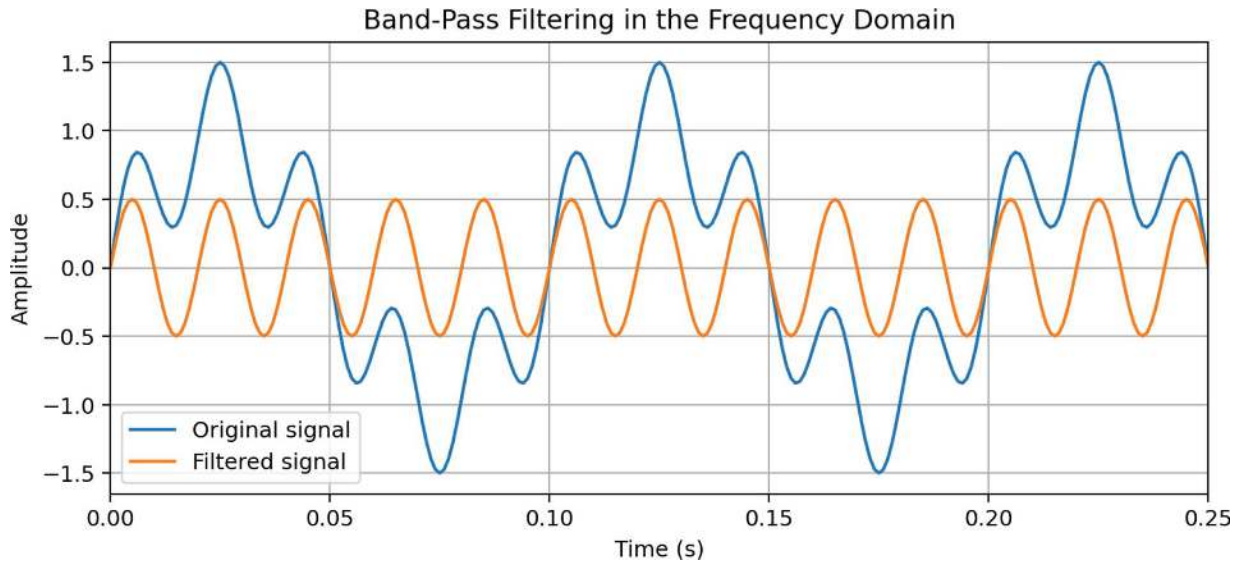


Figure 12-16 Original signal and output of FFT-based band-pass filtering

Notch Filter (FFT)

A notch filter attenuates signals within a specific frequency range while allowing signals outside that range to pass through.

```
import numpy as np
from scipy.fft import fft, ifft
import matplotlib.pyplot as plt

# Generate a sample signal
t = np.linspace(0, 1, 1000, endpoint=False)
x = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 *
np.pi * 50 * t)

# Perform FFT
X = fft(x)

# Create a notch filter
freq = np.fft.fftfreq(len(x), d=0.001)
cutoff_freq_low = 40
cutoff_freq_high = 60
X_filtered = X.copy()
X_filtered[(np.abs(freq) >= cutoff_freq_low) &
(np.abs(freq) <= cutoff_freq_high)] = 0

# Perform inverse FFT
x_filtered = ifft(X_filtered).real

# Plot the results
plt.plot(t, x, label='Original signal')
plt.plot(t, x_filtered, label='Filtered signal')
plt.legend()
plt.show()
```

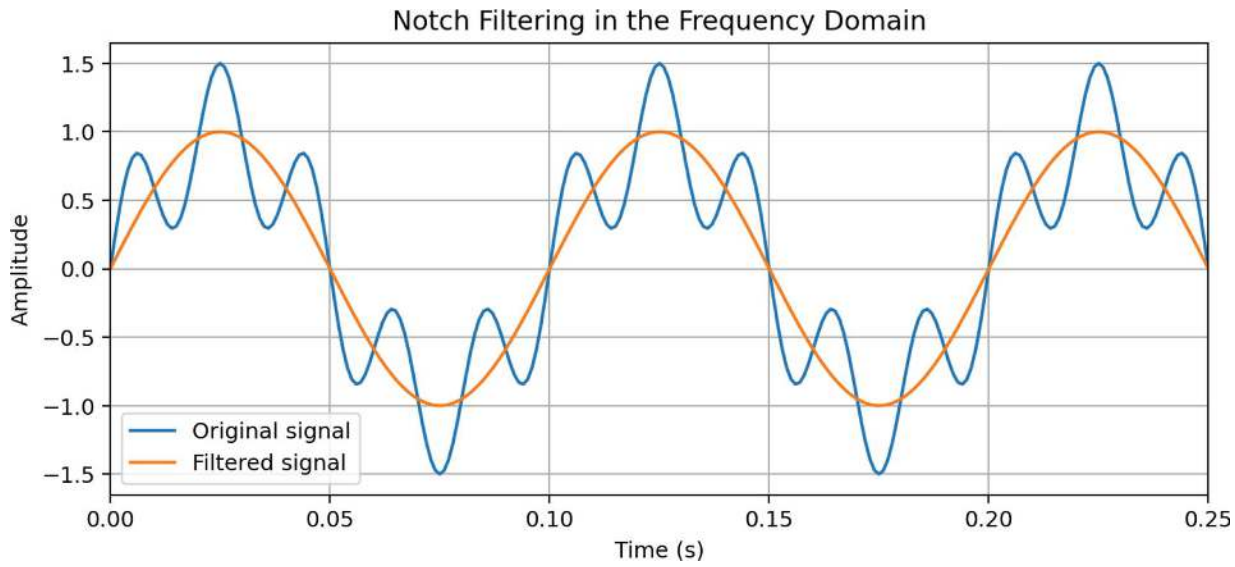


Figure 12-17 Original signal and output of FFT-based notch filtering

Table 12-20 Overview of common digital signal processing (DSP) operations and their corresponding Python libraries

Integrate

The scipy.integrate module provides powerful tools for numerical integration and solving ordinary differential equations (ODEs). It includes functions for computing definite integrals, multiple integrals, and integrals over multiple variables. Also it provides various ODE solvers, including explicit and implicit Runge–Kutta methods, as well as Adams/BDF methods. The module also includes functions for evaluating convergent series and performing Romberg integration.

Figure 12-18 illustrates the geometric principle behind the trapezoidal rule, a fundamental method for performing numerical integration. The primary goal of a definite integral is to calculate the total area under the curve of a function, f(x), between two points on the x-axis, a and b.

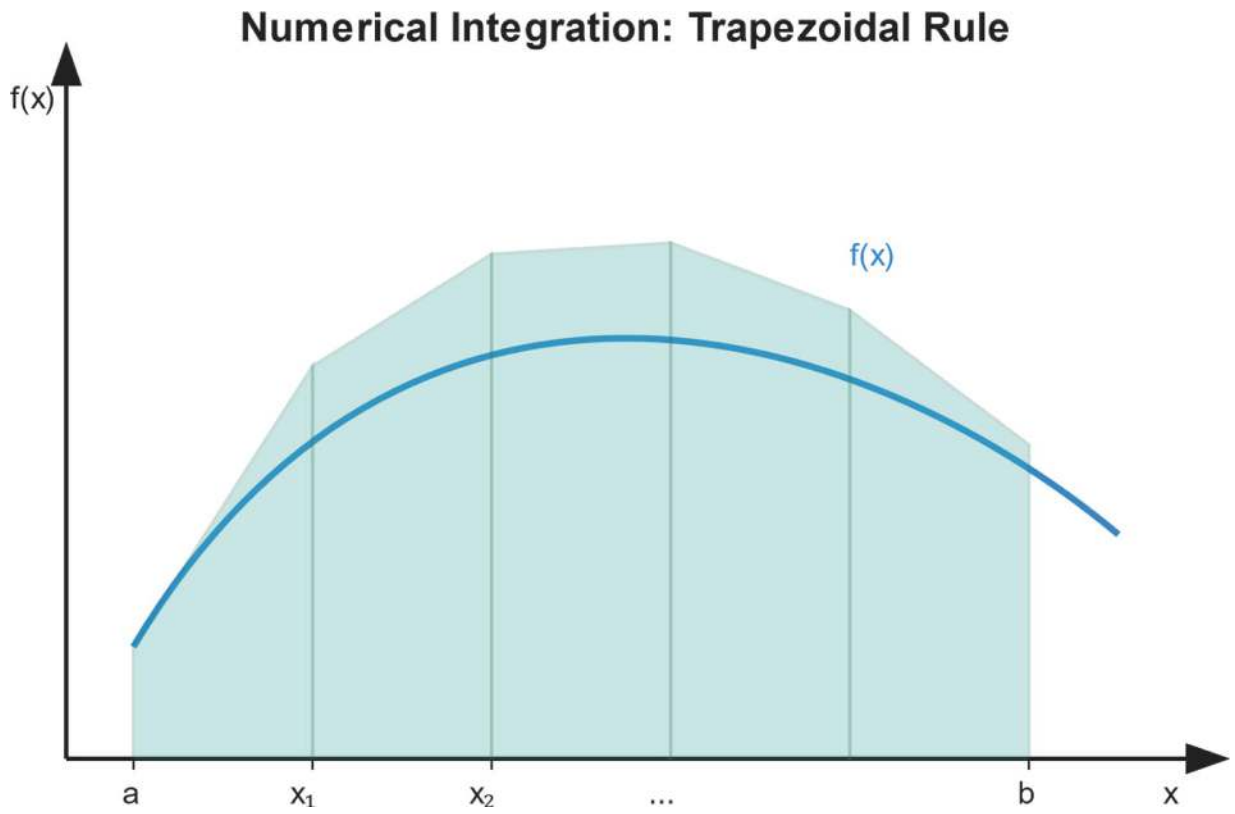


Figure 12-18 Trapezoidal approximation of the area under a curve for numerical integration

The trapezoidal rule approximates this area through the following steps:

1. The total interval from a to b is divided into several smaller, equal-width subintervals (from a to x_1, x_1 to x_2, and so on).
2. Within each subinterval, the area under the curved function is approximated by the area of a simple trapezoid, as shown by the shaded vertical strips. The top edge of each trapezoid is a straight line that connects the function's values at the beginning and end of the subinterval.
3. The total area under the curve is then estimated by summing the areas of all these individual trapezoids.

This method, known as the composite trapezoidal rule, is directly implemented in SciPy through the scipy.integrate.trapezoid function.

This function takes the y values (the height f(x)) and x values of a set of data points and computes the total area, providing an efficient way to perform numerical integration.

The main functions of scipy.integrate are listed in Table 12-21.

Table 12-21 Key functions for numerical integration and solving ODEs in scipy.integrate

Compute a definite integral:

```
quad(func, a, b, args=(), full_output=0,
epsabs=1.49e-08, epsrel=1.49e-08, limit=50,
points=None, weight=None, wvar=None, wopts=None,
maxp1=50, limlst=50, complex_func=False)
```

Table 12-22 shows the parameters with default values.

Table 12-22 Parameters and descriptions for the scipy.integrate.quad function

The function integrate.quad() can return several data, like those listed in Table 12-23.

Table 12-23 Return values for the scipy.integrate.quad function

By default full_output=False, in this case integrate.quad returns only the first two values.

Table 12-24 Simplified return values for scipy.integrate.quad when full_output=False

The simplest code for the operation of numerical integration is shown below:

```
# Import necessary libraries.
from scipy import integrate  # SciPy's integration
module.
```

```
import numpy as np  # Numerical computing library.

# Define the function we want to integrate using a
lambda function.
# Here we're defining f(x) = x^2 (a quadratic
function).
x2 = lambda x: x**2

# Perform the numerical integration using
quadrature method.
# Arguments:
# 1. The function to integrate (x2).
# 2. Lower limit of integration (0).
# 3. Upper limit of integration (4).
result = integrate.quad(x2, 0, 4)

# Print the result.
print(result)
```

The result of this code is a tuple containing

```
(21.333333333333336, 2.368475785867001e-13)
```

The first number 21.33 … is the estimated value of the integral and the second number 2.36…e-13 is an estimate of the absolute error in the numerical integration.

To return just the estimated value of the integral, you can split the tuple and get the value and error separately. The next code example demonstrates how to do this:

```
from scipy import integrate
import numpy as np

# Define the function to integrate.
integrand = lambda x: x**2
```

```
# Perform the integration
result, error = integrate.quad(integrand, 0, 4)

# Print the numerical, analytical and error.
print("Numerical integration result:", result)
print("Numerical integration error:", error)
print("Analytical result:", 4**3 / 3.)
```

The example code above demonstrates numerical integration using the integrate.quad function to compute the definite integral of x^2 from 0 to 4. First, it defines the integrand as a lambda function x^2 and then uses integrate.quad() to compute the numerical result and an error estimate.

Finally, it prints the numerical result ~21.333; the estimated error that is extremely small, ~2.37e-13; and the analytical solution $\frac{4^3}{3} = 21.33$ showing excellent agreement between the numerical and exact mathematical solutions. This example code illustrates how SciPy can accurately approximate integrals that would otherwise require symbolic computation.

Improving Integration Performance

It is possible to improve the integration performance using compiled callback functions avoiding wrapping data in Python objects.

The following example integrates the function x**2 from 0 to 1 using scipy.integrate.quad. To improve integration performance, it wraps the function in a LowLevelCallable object, which allows SciPy to call the function directly without the overhead of a Python function call. The ctypes library is used to create a C-compatible function pointer to the integrand function, which is then passed to LowLevelCallable to create the wrapped function. The wrapped function is then passed to quad to perform the integration, returning the result and an estimate of the error:

```
import ctypes
```

```
from scipy import LowLevelCallable, integrate
import numpy as np

# Define the function to integrate
def integrand(x):
    return x**2

# Create a ctypes wrapper for the function
integrand_c = ctypes.CFUNCTYPE(ctypes.c_double,
ctypes.c_double)
integrand_ptr = integrand_c(integrand)

# Create a scipy.LowLevelCallable object
llc = LowLevelCallable(integrand_ptr)

# Integrate the function using
scipy.integrate.quad
result, error = integrate.quad(llc, 0, 1)

print("Result:", result)
print("Error:", error)
Result: 0.33333333333333337
Error: 3.700743415417189e-15
```

Calculating a Definite Double Integral

The function integrate.dblquad() can be used to evaluate double integrals numerically with high precision:

```
dblquad(func, a, b, gfun, hfun, args=(),
epsabs=1.49e-08, epsrel=1.49e-08)
```

Table 12-25 Parameters for the scipy.integrate.dblquad function

The dblquad function returns two values.

Table 12-26 Return values for the scipy.integrate.dblquad function

```
import numpy as np
from scipy import integrate

# Define a function f(y,x) that returns 1
# This function will be integrated over a 2D
region
f = lambda y, x: 1

# Perform a double integral of f(y,x) over the
region
# bounded by x=0, x=π/4, y=sin(x), and y=cos(x)
result = integrate.dblquad(f, 0, np.pi/4, lambda
x: np.sin(x), lambda x: np.cos(x))

# Print the result of the double integral
print(result)
(0.4142135623730950 3, 1.1083280054755938e-14)

import numpy as np
from scipy import integrate

# Define the function to integrate
f = lambda y, x: x * y**2

# Perform the double integral
result = integrate.dblquad(f, 0, 2, 0, 1)

# Print the result
print(result)
(0.6666666666666667, 7.401486830834377e-15)
```

Solving ODEs

Figure 12-19 provides a conceptual illustration of how a numerical solver for an ordinary differential equation (ODE), such as scipy.integrate.solve_ivp, finds a solution curve for an initial value problem. The goal is to determine the unknown function y(t) when given its starting point and the equation that defines its rate of change (dy/dt).

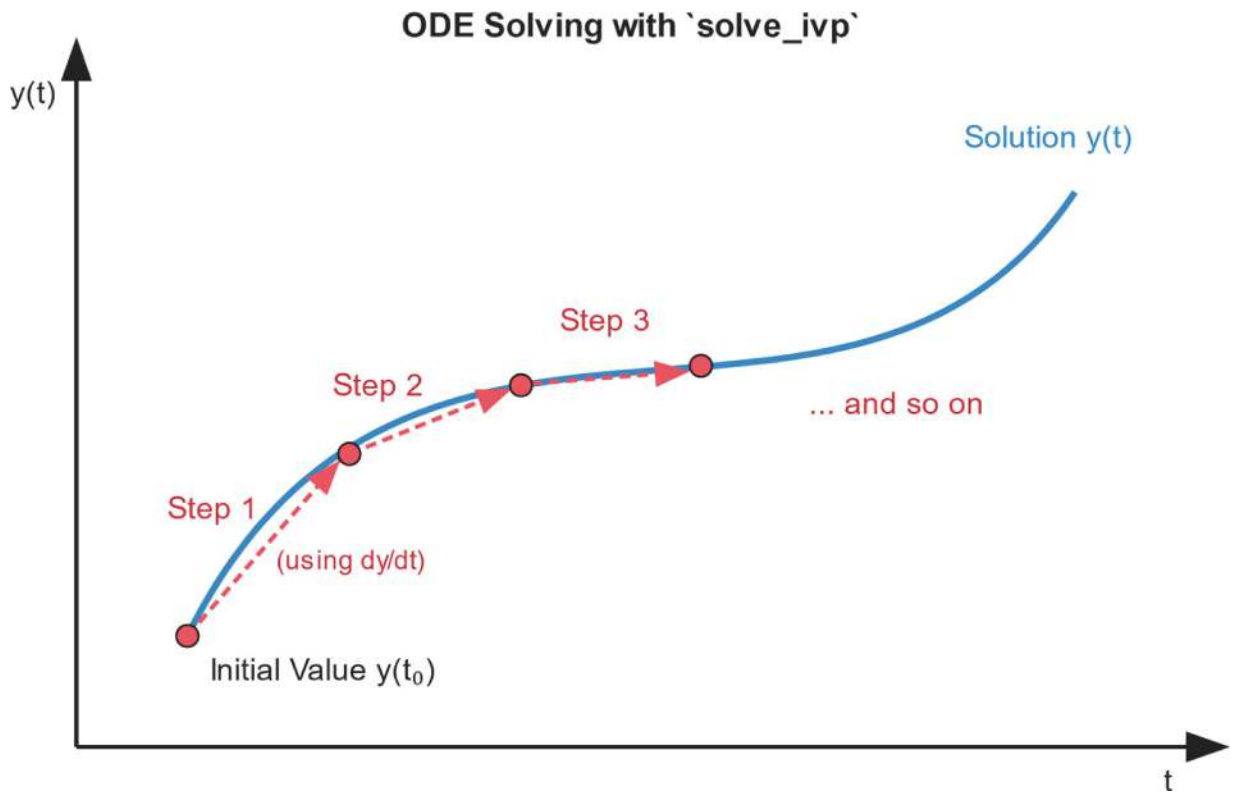


Figure 12-19 Schematic progression of a numerical solver for an ordinary differential equation initial value problem

The solver approximates the true, continuous solution (the solid blue curve) by taking a series of iterative steps:

1. The process begins at a known **initial value**, $y(t_0)$. This corresponds to the y0 parameter required by the solve_ivp function.
2. In **Step 1**, the solver uses the provided ODE function to calculate the slope of the curve (the derivative dy/dt) at the initial point. It

then takes a small step forward in that direction to predict the next point on the solution curve.

3. This process is repeated for **Step 2**, **Step 3**, and so on. At each new point, the slope is recalculated from the ODE, and another incremental step is taken.

4. The sequence of calculated points (the red dots) collectively forms a numerical approximation of the true solution.

This iterative "follow the slope" method is the fundamental principle behind the solve_ivp function. While solve_ivp uses sophisticated and highly accurate algorithms to intelligently adjust the step sizes and minimize error, the core idea of starting at an initial condition and progressively building the solution based on the derivative remains the same. This visualization demystifies the process by showing how the solver constructs the solution piece by piece.

First-Order ODE

Let's get started with a very simple differential equation:

$$\frac{dy}{dt} = -2\mathrm{y}$$

The example code below numerically solves and plots the first-order ordinary differential equation (ODE) described above, which represents exponential decay with a rate of 2 per second. The solution is computed over a time span of 0–5 seconds, starting from the initial condition y(0)=1y(0) = 1. Using integrate.solve_ivp, the code calculates the solution at 100 evenly spaced time points.

After the numerical processing, it visualizes the results with Matplotlib, labeling the time axis in seconds and titling the plot to reflect the equation and its units:

```
import numpy as np
```

```
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def ode_func(t, y):
    return -2 * y  # Decay rate of 2 per second
(if t is in seconds)

# Time span (start, end) in seconds
t_span = (0, 5)  # 0 to 5 seconds
# Initial condition
y0 = [1]
# Solve the ODE
solution = solve_ivp(ode_func, t_span, y0,
t_eval=np.linspace(0, 5, 100))

# Plot the solution
plt.plot(solution.t, solution.y[0], label='y(t)')
plt.xlabel('Time t (seconds)')  # Explicit unit
(seconds)
plt.ylabel('y')
plt.title('Solution of dy/dt = -2y (t in
seconds)')  # Clarified in title
plt.legend()
plt.grid()
plt.show()
```

The output illustrates the decay of y(t)y(t) over time.

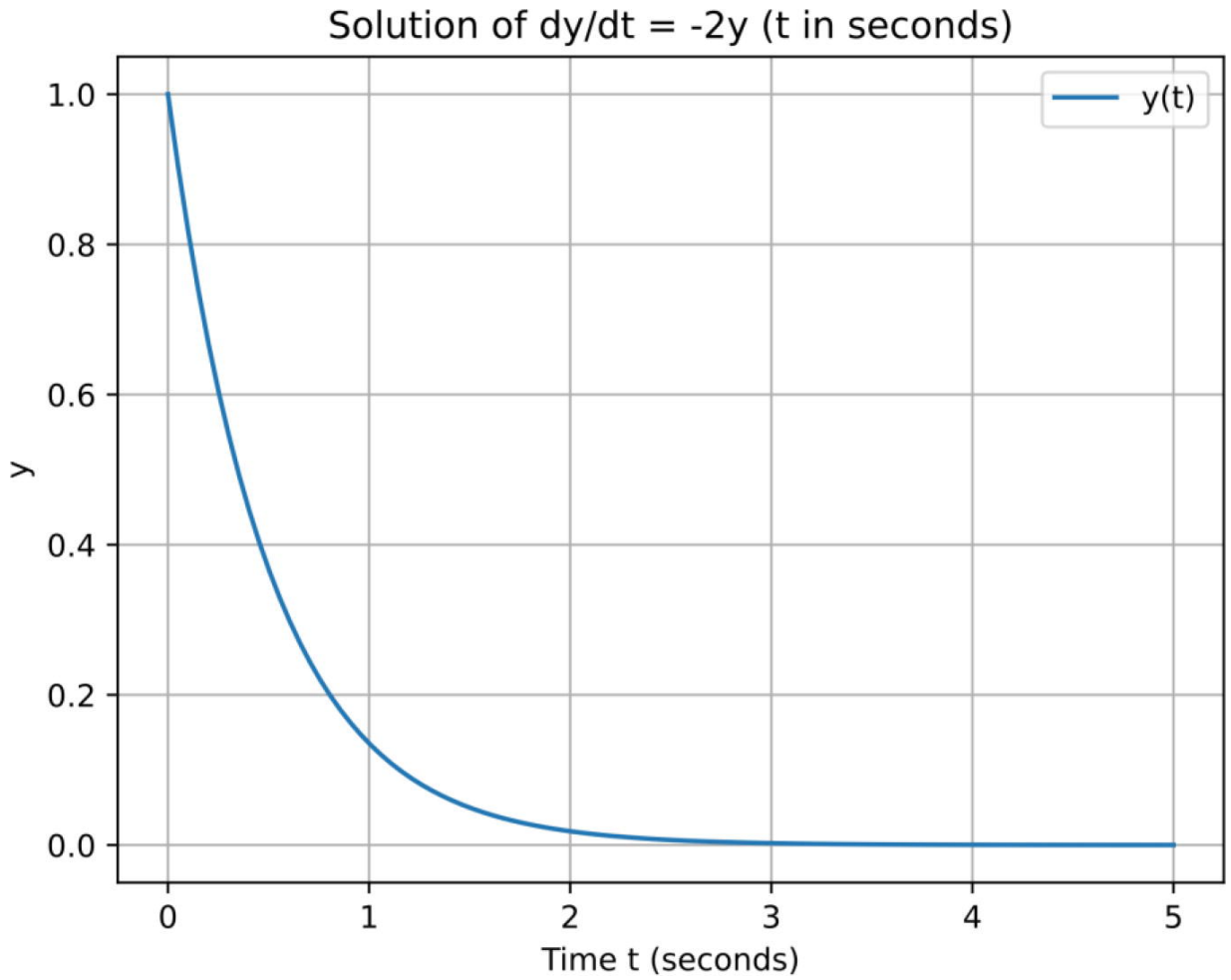


Figure 12-20 Numerical solution of dy/dt = -2y with y(0) = 1 using solve_ivp

This is another example using a slightly larger differential equation:

$$\frac{dy}{dt} = \sin(t) - y^2 + e^{-t}$$

This example numerically solves the first-order ordinary differential equation (ODE) given by dy/dt = sin(t) - y^2 + exp(-t) with the initial condition y(0)=1y(0) = 1. The solution is computed over a time span of 0–10 seconds using solve_ivp, evaluated at 100 evenly spaced time points. The results are then plotted, showing the time evolution of y(t)y(t):

```
import numpy as np
```

```
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def first_order_ode(t, y):
    return np.sin(t) - y**2 + np.exp(-t)

# Time span (start, end) in seconds
t_span = (0, 10)  # 0 to 10 seconds
# Initial condition: y(0) = 1
y0 = [1]
# Solve the ODE
solution = solve_ivp(first_order_ode, t_span, y0,
t_eval=np.linspace(0, 10, 100))

# Plot the solution
plt.plot(solution.t, solution.y[0], label='y(t)')
plt.xlabel('Time t (seconds)')  # Explicitly
stating the unit
plt.ylabel('y')
plt.title('Solution of dy/dt = sin(t) - y^2 +
exp(-t)')
plt.legend()
plt.grid()
plt.show()
```

The plot visualizes how the solution y(t) behaves dynamically under the given ODE.

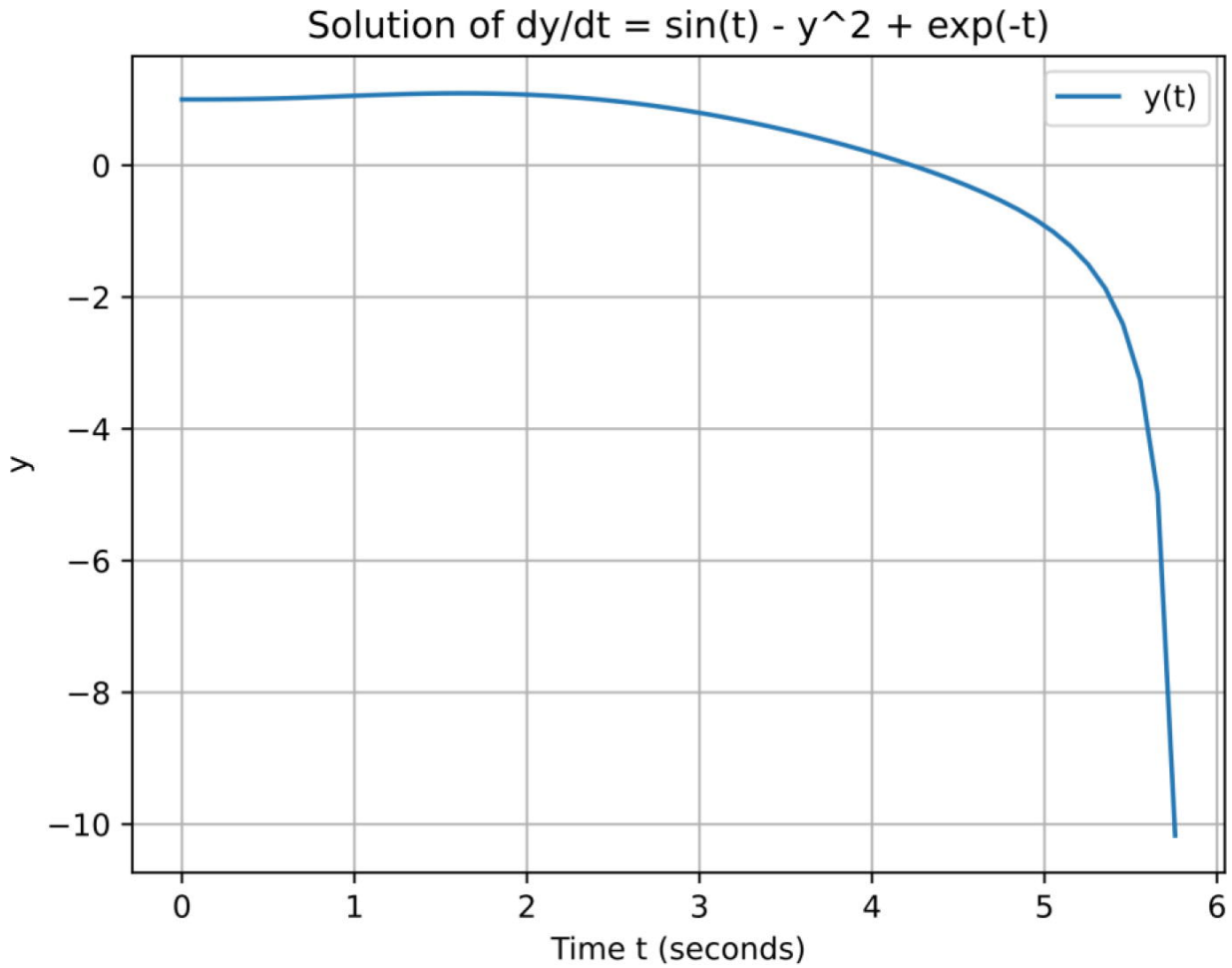


Figure 12-21 Numerical solution of dy/dt = sin(t) - y^2 + exp(-t) with y(0) = 1 using solve_ivp

Second-Order ODE

$$\frac{d^2y}{dt^2} + 3\frac{dy}{dt} + 6\mathrm{y} = 0$$

This code numerically solves a second-order ordinary differential equation defined by the equation represented above. It first converts it into a system of two first-order ODEs, where y(0) = y and y(1) = $\frac{dy}{dt}$, and then computes the solution over the time span t = 0 to 10 seconds using solve_ivp from SciPy with initial conditions y(0) = 1 and y'(0) = 0 .

The solution is evaluated at 100 evenly spaced time points, and the resulting displacement y(t) is plotted:

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def second_order_ode(t, y):
    dydt = y[1]  # dy/dt (velocity if y is
position)
    d2ydt2 = -3*y[1] - 6*y[0]  # d²y/dt²
(acceleration)
    return [dydt, d2ydt2]

# Time span (start, end) in seconds
t_span = (0, 10)  # 0 to 10 seconds
# Initial conditions: y(0) = 1, y'(0) = 0
y0 = [1, 0]
# Solve the ODE
solution = solve_ivp(second_order_ode, t_span, y0,
t_eval=np.linspace(0, 10, 100))

# Plot the solution
plt.plot(solution.t, solution.y[0], label='y(t)')
plt.xlabel('Time t (seconds)')  # Explicit unit
(seconds)
plt.ylabel('y')
plt.title('Solution of d²y/dt² + 3dy/dt + 6y = 0 (t
in seconds)')  # Clarified
plt.legend()
plt.grid()
plt.show()
```

The plot visualizes the system's damped oscillatory behavior due to the coefficients in the ODE.

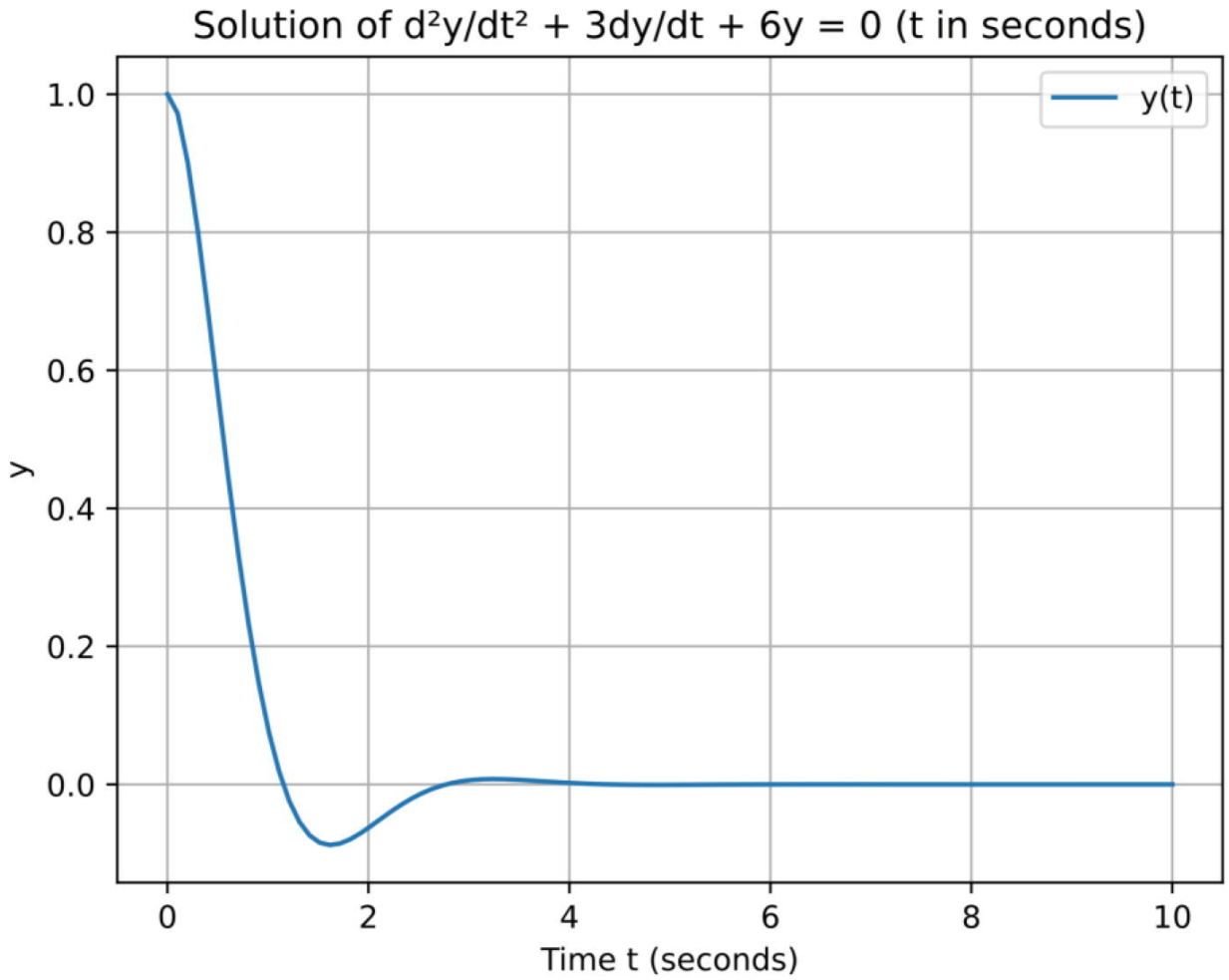


Figure 12-22 Numerical solution of $d^2y/dt^2 + 3dy/dt + 6y = 0$ with $y(0) = 1$ and $dy/dt(0) = 0$ using solve_ivp

Interpolate

The interpolate subpackage offers a rich collection of tools for estimating unknown data points within a known range. It supports both univariate and multivariate interpolation, offering classes and functions for splines, polynomial interpolation, and specialized methods like Lagrange and Taylor polynomials.

Whether you're smoothing noisy data, reconstructing surfaces, or solving differential equations, this subpackage delivers a powerful and flexible solution. It combines high performance with easy-to-use convenience functions and object-oriented interfaces, making it suitable for a wide range of applications.

Table 12-27 summarizes the key interpolation methods available in this subpackage, categorizing them by their use cases and dimensionality.

Univariate interpolation focuses on one-dimensional (1D) data used, for example, on time series in applications like engineering to approximate experimental measurements like pressure or temperature variations over time, analysis in finance to fill missing prices, and signal processing to reconstruct signals.

Multivariate interpolation extends this to higher dimensions like 2D, 3D, or even N-dimensional data, handling data points that are arranged in a regular, predictable grid pattern known as gridded data and data points that are irregularly placed, with no predictable pattern, where the spacing between data points varies and the layout is not aligned to a regular grid known as scattered data.

In simpler terms, when data is organized in a grid pattern, it's easier to fill in the gaps and estimate values between points using straightforward algorithms. However, when data is scattered randomly, more advanced and complex techniques, such as radial basis functions or Delaunay triangulation, are needed to accurately estimate values between points.

Figure 12-23 illustrates the fundamental components that define a **B-spline** or basis spline, which is a powerful mathematical method for generating smooth, flexible curves. B-splines are widely used in computer graphics, design, and data fitting. They are defined by a set of control points and a sequence known as knots. The key components are

- *B-spline Curve (Red Line)*: This is the final smooth curve that is generated. It is a *piecewise polynomial*, meaning it is constructed from multiple individual polynomial segments that are joined together in a way that ensures smoothness.
- *Control Points (Blue Dots)*: These points form a control polygon that guides the overall shape of the curve. The B-spline curve is "pulled" towards these points, but unlike a simple interpolating spline, it does not necessarily pass through them. The locations of these points dictate the curve's path and curvature.

- *Knots (Green Ticks)*: These are a sequence of parameter values that determine where the polynomial pieces of the curve connect to each other. The number and placement of knots have a significant influence on the shape of the resulting B-spline.

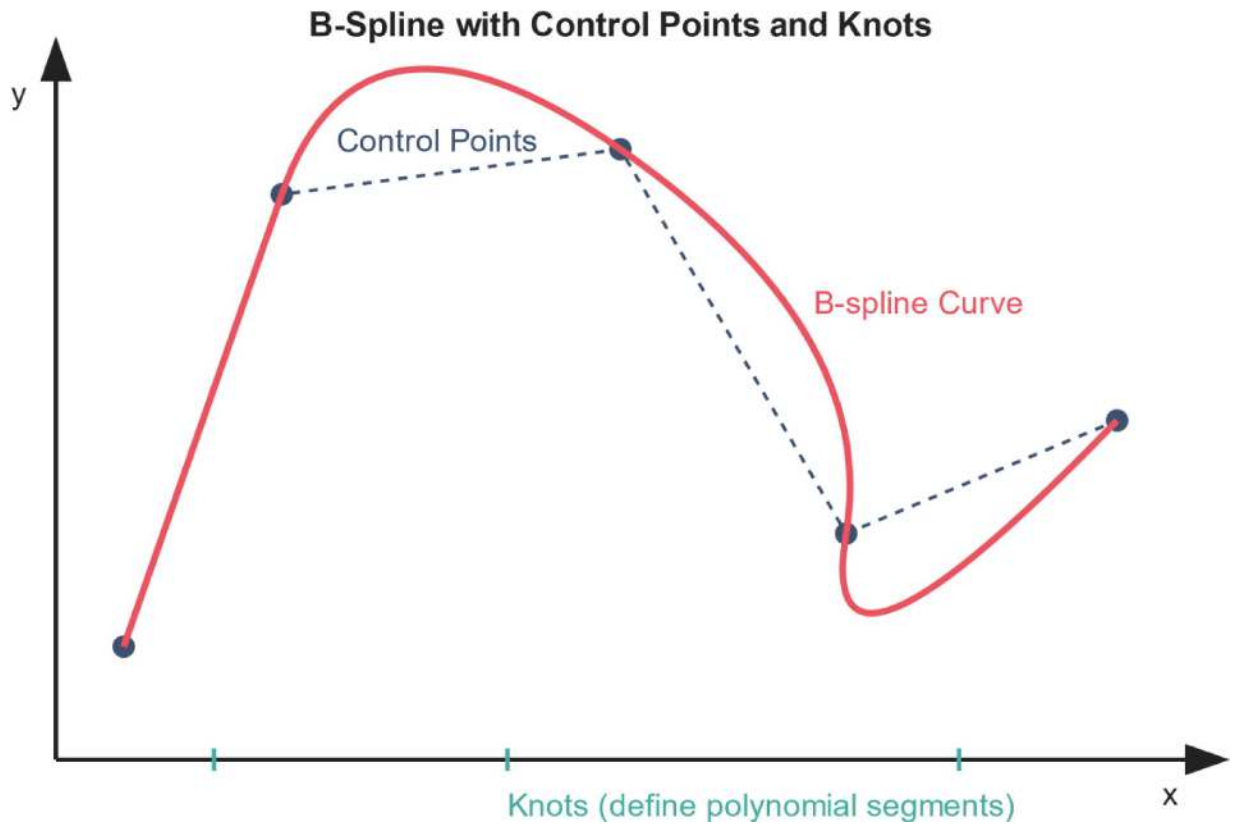


Figure 12-23 B-spline curve, control polygon, and knot locations

The scipy.interpolate subpackage provides a rich set of tools for working with B-splines. Functions like make_splrep and make_splprep are used to find the B-spline representation, that is, the control points and knots, that best fits a given set of data. This visualization provides a clear conceptual model, distinguishing the role of control points in shaping the curve from the role of knots in defining its underlying structure.

These are crucial in geospatial analysis for interpolating elevation data from scattered GPS points, 3D medical imaging, and climate modeling.

1D splines provide smooth, adjustable curves for one-dimensional (1D) data. For example, robotics applications use them to smooth joint trajectories, and data analysis uses them to reduce noise in sensor readings.

2D splines fit smooth surfaces to 2D data, enabling applications in computer graphics for image correction, automotive design for smoothing car body surfaces, and geology for mapping underground rock layers.

Figure 12-24 provides a conceptual overview of how radial basis function (RBF) interpolation operates. RBF is a powerful method for creating a smooth and continuous surface from a set of scattered, unstructured data points. This technique is particularly useful for multivariate data and is implemented in SciPy via the RBFInterpolator class.

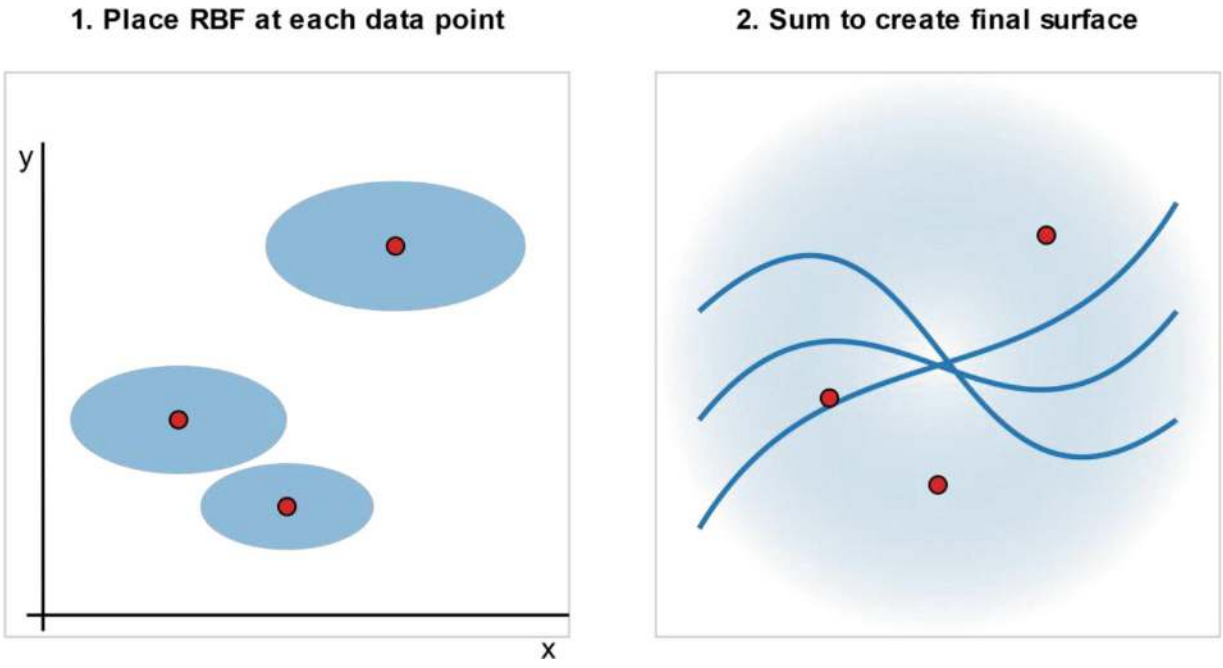


Figure 12-24 Conceptual construction of an interpolated surface from radial basis functions centered at data points

The process can be understood in two main steps:

1. *Place RBF at Each Data Point*: The method begins by centering a radial basis function at the location of each known data point (red

dots). An RBF is a function whose value depends only on the distance from its center. This is conceptually similar to placing a smooth “hill” or “mound of influence” at each data point.

2. *Sum to Create Final Surface*: The final interpolated surface is then constructed by taking a weighted sum of all these individual radial basis functions. The weights are mathematically determined so that the resulting surface passes exactly through the original data points. This creates a smooth surface where the value at any given location is a blended influence of all the known points.

Rational approximation avoids instability in high-degree polynomials by using ratios of polynomials. It proves valuable when traditional polynomial interpolation fails, particularly in circuit design for modeling impedance responses and numerical analysis. These methods use ratios of polynomials to improve stability in challenging scenarios.

The interpolation subpackage also includes advanced tools like Lagrange and Taylor approximations—these methods cover a wide range of interpolation needs, from simple curve fitting to complex multidimensional surface modeling, which are especially useful in physics simulations and optimization problems where straightforward yet precise interpolations are needed.

Figure 12-25 provides a comparative visualization of different one-dimensional (univariate) interpolation methods available in scipy.interpolate. Interpolation is the technique of estimating unknown values that lie between a set of known data points. The goal is to construct a new continuous function that passes exactly through the original points (red dots).

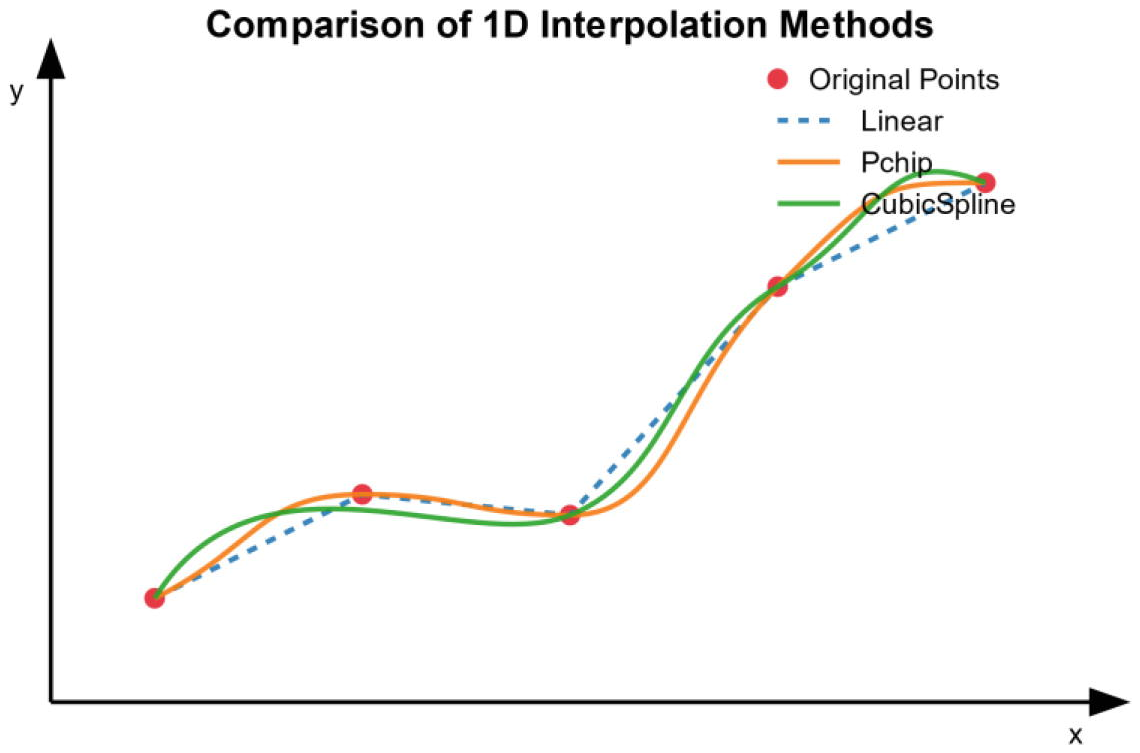


Figure 12-25 Comparison of linear, PCHIP, and cubic spline interpolation for one-dimensional data

The three methods compared in the graph are

- *Linear (Dashed Blue Line)*: This is the most fundamental interpolation method. It simply connects consecutive data points with straight lines. The resulting function is continuous, but it is not smooth, having sharp corners at each of the original data points.
- *Pchip (Solid Orange Line)*: This represents the Piecewise Cubic Hermite Interpolating Polynomial. It fits a special type of cubic polynomial between data points, creating a smooth curve. A key feature of Pchip is that it is designed to preserve the shape and monotonicity of the original data, meaning it avoids overshooting or creating new peaks and valleys between points.
- *CubicSpline (Solid Green Line)*: This method also fits a piecewise cubic polynomial to the data, resulting in a smooth and continuous curve that passes through every point. As shown in the graph, a cubic spline produces a very smooth path, though it may sometimes

introduce slight “overshoots” (as seen between the first and second points) that Pchip interpolation is designed to prevent.

This visualization serves as a practical guide for selecting an appropriate interpolation tool from SciPy. It clearly demonstrates that while linear interpolation is simple, methods like PchipInterpolator and CubicSpline offer superior, smoother results suitable for a wide range of scientific and engineering applications.

See Table 12-27 for a summary of the various interpolation categories available in scipy.interpolate.

Table 12-27 Summary of interpolation categories in scipy.interpolate

Table 12-28 is a comprehensive list of available operations.

Table 12-28 Comprehensive list of functions and classes available in scipy.interpolate

Univariate Interpolation with Pchip

Pchip, or Piecewise Cubic Hermite Interpolating Polynomial, is a widely used interpolation method in fields such as engineering, physics, and data analysis. It estimates values between known data points using piecewise cubic polynomials and Hermite basis functions, which are defined by both the function values and their derivatives at the data points. This approach ensures a smooth, continuous curve that preserves the shape and monotonicity of the original data.

The scipy.interpolate submodule provides the PchipInterpolator class; its signature is shown below:

```
class PchipInterpolator(x, y, axis=0,
extrapolate=None)
```

Table 12-29 summarizes the parameters of the PchipInterpolator class.

Table 12-29 Parameters for the PchipInterpolator class

The following example demonstrates the use of the Pchip (Piecewise Cubic Hermite Interpolating Polynomial) interpolator to interpolate irregularly spaced data points. The code first generates a set of x values with clusters and gaps and corresponding y values with added noise to simulate real-world data. It then creates a Pchip interpolator object and uses it to interpolate the data over a finer grid of x values.

In the code x is an array of irregularly spaced values, intentionally clustered and gapped to demonstrate the effectiveness of Pchip interpolation, and y values are generated by applying the sine function to x and adding a small amount of random noise to simulate real-world data:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import PchipInterpolator

# Irregularly spaced x values (clusters and gaps)
x = np.array([1, 1.2, 2.8, 3.1, 5.5, 6.8, 7.0,
8.5])
y = np.sin(x) + 0.1 * np.random.randn(len(x))  #
Add slight noise

# Interpolation domain
xx = np.linspace(1, 8.5, 200)

# Create PCHIP interpolator
pchip = PchipInterpolator(x, y)
yy = pchip(xx)

# Plot
plt.plot(x, y, 'o', label='Data Points')
plt.plot(xx, yy, '-', label='PchipInterpolator',
linewidth=2, color='red')
```

```
plt.title("PCHIP Interpolation with Irregularly
Spaced Points")
plt.xlabel("x")
plt.ylabel("Interpolated y")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

The code plots the original data points and the interpolated curve, showcasing the ability of the Pchip interpolator to smoothly connect the data points while preserving the underlying shape of the data.

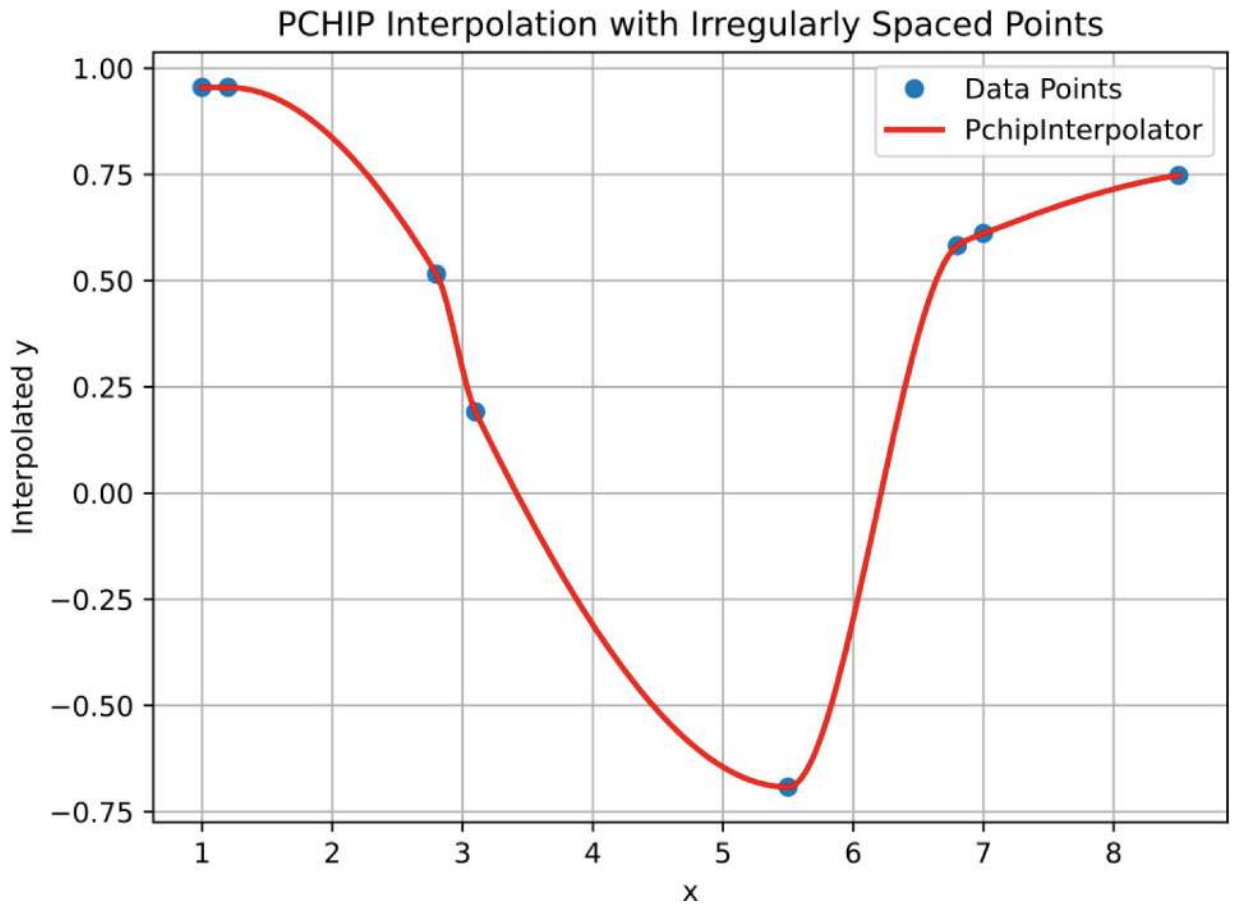


Figure 12-26 Shape-preserving PCHIP interpolation of irregularly spaced data points

Interpolating Cubic Polynomial with CubicSpline

The class called CubicSpline creates a cubic spline interpolator object. This class uses the parameters of Table 12-30 to construct a piecewise cubic polynomial that smoothly interpolates the input data points, allowing for various boundary conditions and extrapolation behaviors:

```
class CubicSpline(x, y, axis=0, bc_type='not-a-
knot', extrapolate=None)
```

Table 12-30 Parameters for the CubicSpline class

Table 12-30 summarizes the parameters of the CubicSpline class.

One important aspect of this class is the bc_type parameter, also known as the boundary condition type. It determines how the cubic spline behaves at its boundaries, such as the start and end points of the data. Table 12-31 summarizes the possible bc_type values.

Table 12-31 Boundary condition types (bc_type) for the CubicSpline class

The example code creates a cubic spline interpolation of a set of data points using the CubicSpline class. The data points are defined by the arrays x and y, which are then used to create a cubic spline interpolator object cs. This object is used to generate new y-values at a finer grid of x-values defined by x_new, resulting in a smooth curve that passes through the original data points:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline

# Same data points
x = np.array([0, 1, 2, 3, 4])
y = np.array([0, 2, 1, 3, 7])

# Create cubic spline interpolator
```

```
cs = CubicSpline(x, y)

# New points
x_new = np.linspace(0, 4, 100)
y_new = cs(x_new)

# Plot
plt.plot(x, y, 'o', label='Data points')
plt.plot(x_new, y_new, label='Cubic Spline',
color= "red")
plt.legend()
plt.title("Cubic Spline Interpolation")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

The original data points and the interpolated curve are then plotted, with the data points shown as blue circles and the interpolated curve shown as a red line, demonstrating the effectiveness of cubic spline interpolation in creating a smooth and continuous representation of the data.

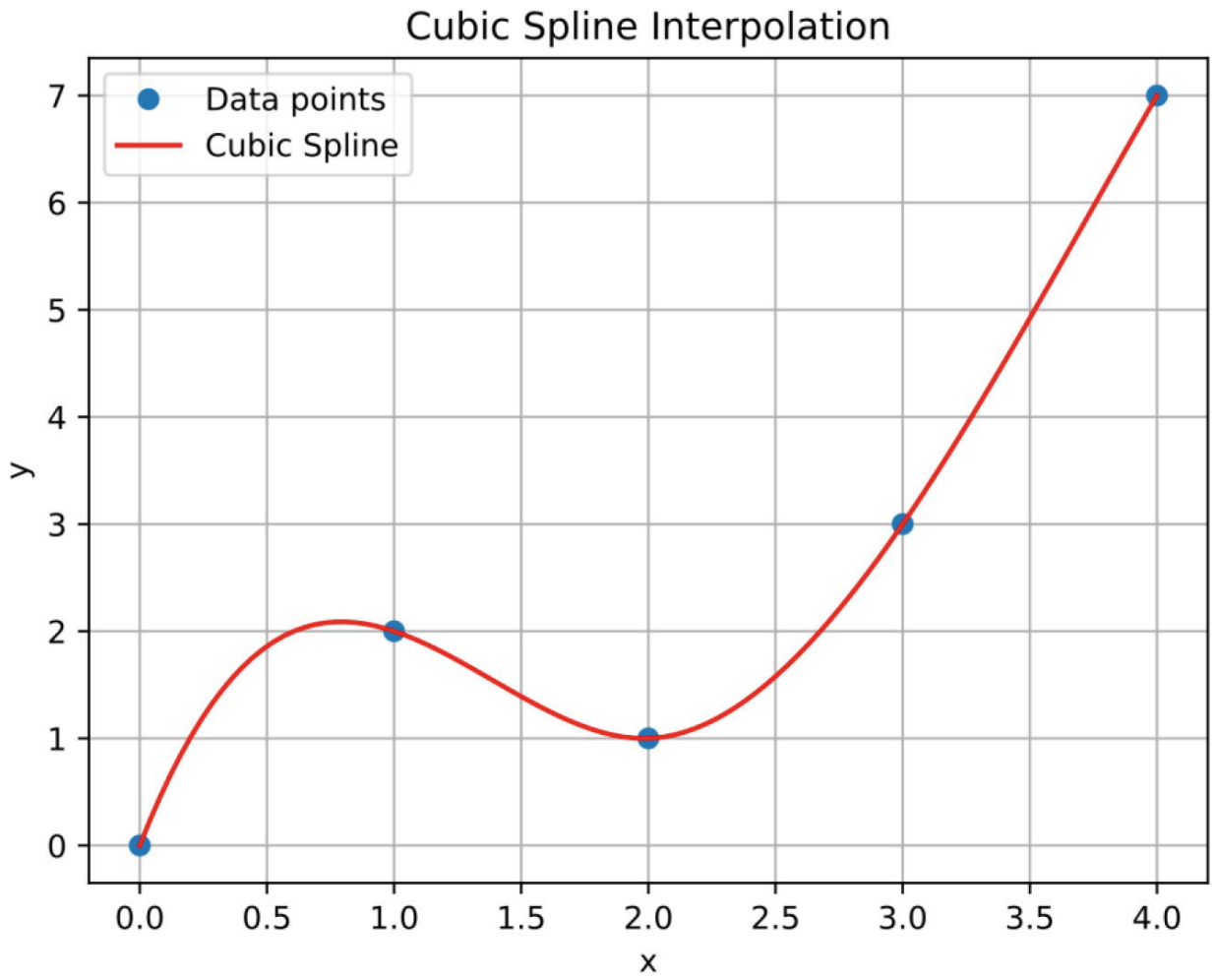


Figure 12-27 Cubic spline interpolation through a set of one-dimensional data points

Multivariate Interpolation with griddata

For multivariate interpolation with unstructured data, the most commonly used operation is griddata(), as it efficiently handles scattered points in multiple dimensions and supports various interpolation methods such as nearest, linear, and cubic.

The example code provided performs 2D interpolation using the nearest-neighbor method. It starts by generating 50 random data points in a 2D space from 0 to 8 on both axes and assigning each point a value based on a mathematical function with some random noise.

Then it creates a regular grid of 100 × 100 points where interpolated values will be calculated. The griddata() function

calculates values for each grid point by finding the value of the nearest original data point:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import griddata

# Generate data points
np.random.seed(42)
points = np.random.rand(50, 2) * 8
values = np.sin(points[:, 0]) * np.cos(points[:,
1]) + 0.1 * np.random.randn(len(points))

# Create grid and interpolate using nearest
neighbor
grid_x, grid_y = np.mgrid[0:8:100j, 0:8:100j]
result = griddata(points, values, (grid_x,
grid_y), method='nearest')

# Plot
plt.figure(figsize=(8, 6))
im = plt.imshow(result.T, extent=(0, 8, 0, 8),
origin='lower', cmap='viridis')
plt.scatter(points[:, 0], points[:, 1], c='red',
s=20, edgecolor='black')
plt.title('Nearest Neighbor Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar(im)
plt.tight_layout()
plt.show()
```

The code visualizes the results with Matplotlib, showing the interpolated surface as a color map and overlaying the original data points as red dots with black edges. This visualization helps illustrate

how the nearest-neighbor interpolation creates a discontinuous surface with distinct regions of constant values around each original data point.

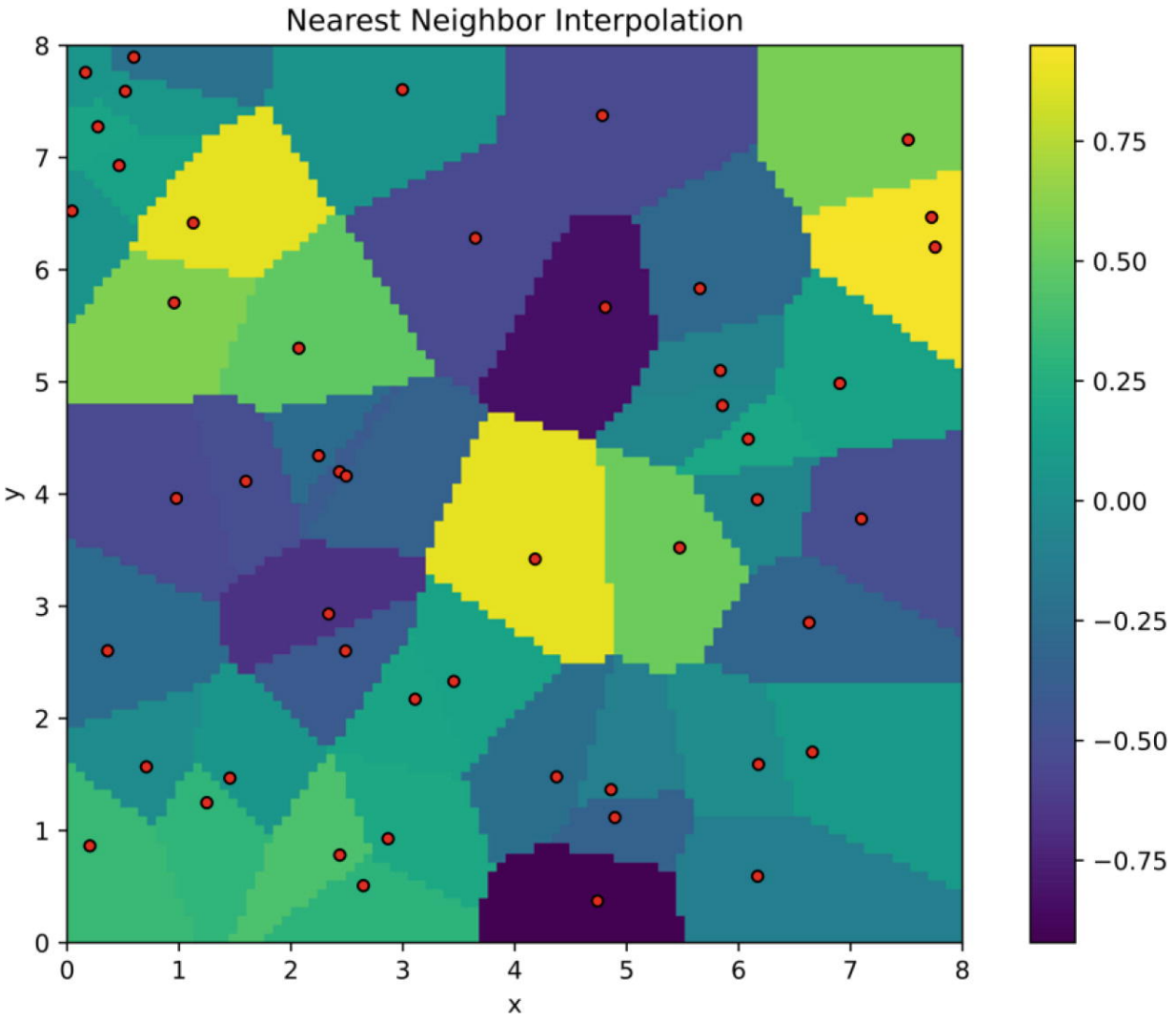


Figure 12-28 Nearest-neighbor interpolation of scattered two-dimensional data using griddata

Figure 12-29 compares three common methods for the multivariate interpolation of scattered and unstructured data, which is the process of creating a continuous surface from a sparse set of known data points (the black dots). These three approaches are directly available as options in the scipy.interpolate.griddata function.

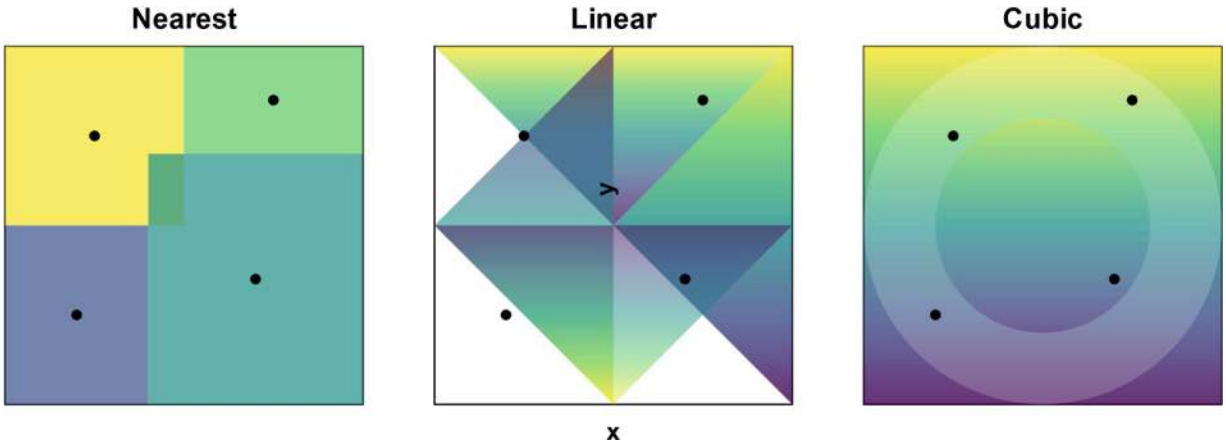


Figure 12-29 Comparison of nearest-neighbor, linear, and cubic interpolation of scattered two-dimensional data

The methods illustrated are

- *Nearest*: In this method, any location on the surface is assigned the value of the single closest known data point. This results in a discontinuous surface made of distinct, block-like regions with sharp boundaries, as shown in the leftmost panel.
- *Linear*: This method works by first connecting the known points to form a mesh of triangles. It then performs linear interpolation within each triangle, creating a surface composed of flat, tilted planes. The resulting surface is continuous, but its gradient is not smooth across the edges where the triangles meet.
- *Cubic*: This method fits a smoother, curved surface that passes through the known data points. This produces a result that is continuous in both value and its first derivative, leading to the smooth and gradual color transitions seen in the rightmost panel.

This visualization serves as an excellent guide for choosing the appropriate method parameter in the griddata function. It clearly shows the trade-offs between the techniques: “nearest” is the fastest but produces a coarse result, “linear” ensures continuity but is not smooth, and “cubic” provides the smoothest, most visually continuous surface, though it is typically more computationally intensive.

B-spline Representation of 1D Data

The make_splprep function computes a smooth parametric B-spline approximation of an N-dimensional curve from a set of scattered data points, allowing precise control over smoothing, knot placement, and parameterization. This powerful tool is particularly valuable for applications requiring curve fitting in multiple dimensions, such as computer-aided design (CAD), motion planning in robotics, and data approximation tasks, or scientific data visualization:

```
make_splprep(x, *, w=None, u=None, ub=None,
ue=None, k=3, s=0, t=None, nest=None)
```

The asterisk (*) symbol has already been covered in this chapter and is a special syntax in Python called a “keyword-only parameter separator” or “parameter separator” and is used to indicate that all parameters after it must be specified by keyword and not by position.

Table 12-32 summarizes the parameters of the make_splrep function.

Table 12-32 Parameters for the make_splprep function

The return values are listed in Table 12-33.

Table 12-33 Return values for the make_splprep function

The following example uses make_splrep to fit a smooth B-spline through noisy one-dimensional data. The returned BSpline object is evaluated directly on a denser grid; s=0 requests exact interpolation of the sample points.

This technique proves valuable for applications like signal smoothing, trajectory fitting, or data approximation in cases where both smoothness and exact interpolation remain essential requirements:

```
import numpy as np
```

```
import matplotlib.pyplot as plt
from scipy.interpolate import make_splrep

# Sample data (noisy sine wave)
x = np.linspace(0, 10, 20)
np.random.seed(42)
y = np.sin(x) + 0.1 * np.random.randn(len(x))

# Create the B-spline representation of the curve
# s=0 forces the spline to go through all points
(interpolation)
spl = make_splrep(x, y, s=0)

# Evaluate the spline at new x values for a smooth
curve
x_new = np.linspace(0, 10, 200)
y_new = spl(x_new)

# Plot original data and the spline curve
plt.figure(figsize=(8, 5))
plt.plot(x, y, 'o', label='Data points')
plt.plot(x_new, y_new, '-', label='Spline
interpolation')
plt.title('Spline Interpolation using
make_splrep')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

The original data plotted as circles and the interpolated spline shown as a solid line are visualized, demonstrating how B-spline interpolation effectively reconstructs the underlying sine wave while eliminating noise.

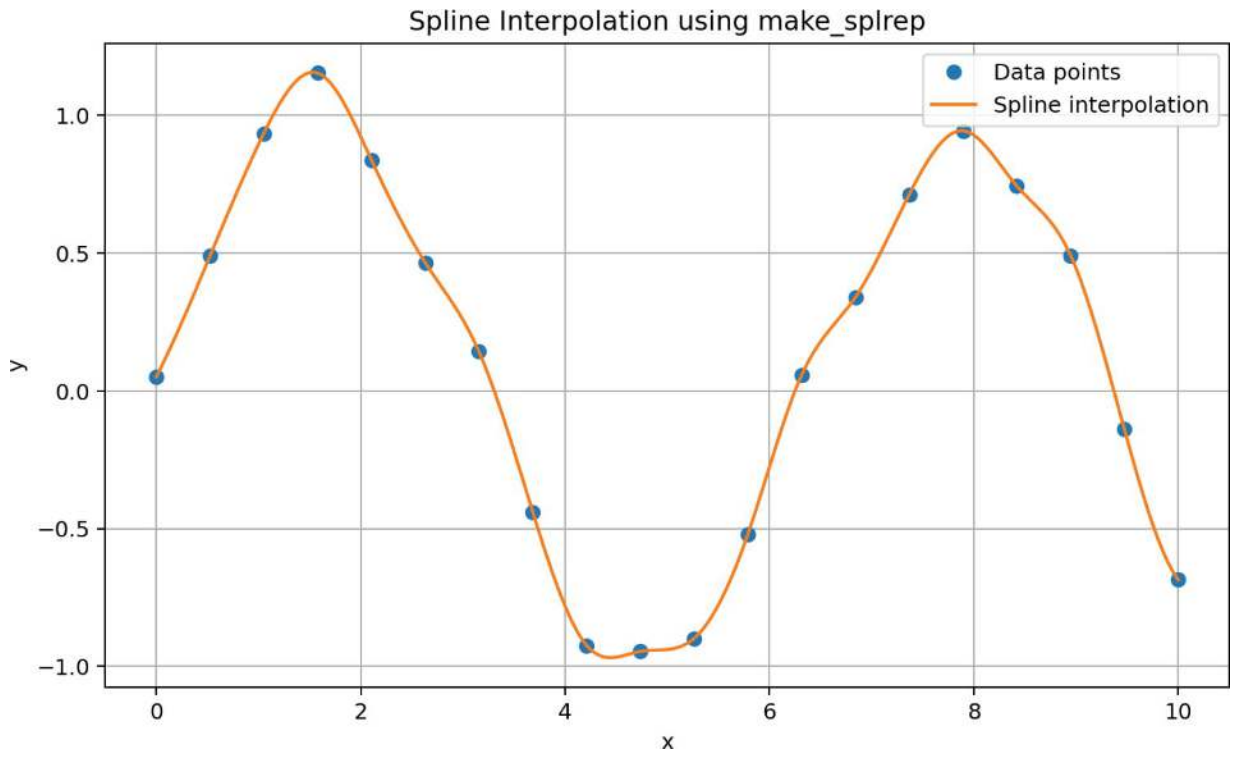


Figure 12-30 One-dimensional spline interpolation of sampled data using make_splrepr

2D Spline Interpolation

Table 12-34 summarizes the parameters of the RectBivariateSpline class.

Table 12-34 Parameters for the RectBivariateSpline class

2D spline interpolation using RectBivariateSpline is a computational technique for estimating values across a two-dimensional grid based on a given set of discrete data points. This method constructs a smooth surface by fitting piecewise polynomial functions typically cubic between the known points in both the x and y directions.

Depending on the chosen settings, the interpolation can either exactly pass through all input points, known as interpolation, or approximate a smoother fit that may not match the data points precisely, known as smoothing.

The output is a continuous and differentiable surface, making it useful for applications like image processing, scientific data analysis,

and geometric modeling where smooth approximations of scattered or gridded data are required.

This code demonstrates how to use RectBivariateSpline from SciPy to perform 2D spline interpolation over a rectangular grid. It first defines a coarse grid (x and y) and computes values Z using the function sin(x) * cos(y). The RectBivariateSpline object is then created to fit a spline surface to this data. A finer grid (x_fine, y_fine) is used to evaluate the interpolated values (Z_fine), resulting in a smooth approximation of the original function:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import RectBivariateSpline

# Sample grid and data
x = np.linspace(0, 5, 6)
y = np.linspace(0, 5, 6)
X, Y = np.meshgrid(x, y, indexing='ij')
Z = np.sin(X) * np.cos(Y)

# Create spline interpolator
spline = RectBivariateSpline(x, y, Z)

# Interpolate on a finer grid
x_fine = np.linspace(0, 5, 100)
y_fine = np.linspace(0, 5, 100)
Z_fine = spline(x_fine, y_fine)

# Plot result
plt.contourf(x_fine, y_fine, Z_fine,
cmap='viridis')
plt.colorbar(label='Interpolated Value')
plt.title("2D Spline Interpolation")
plt.xlabel("x")
plt.ylabel("y")
```

```
plt.show()
```

The interpolated surface is visualized as a filled contour plot.

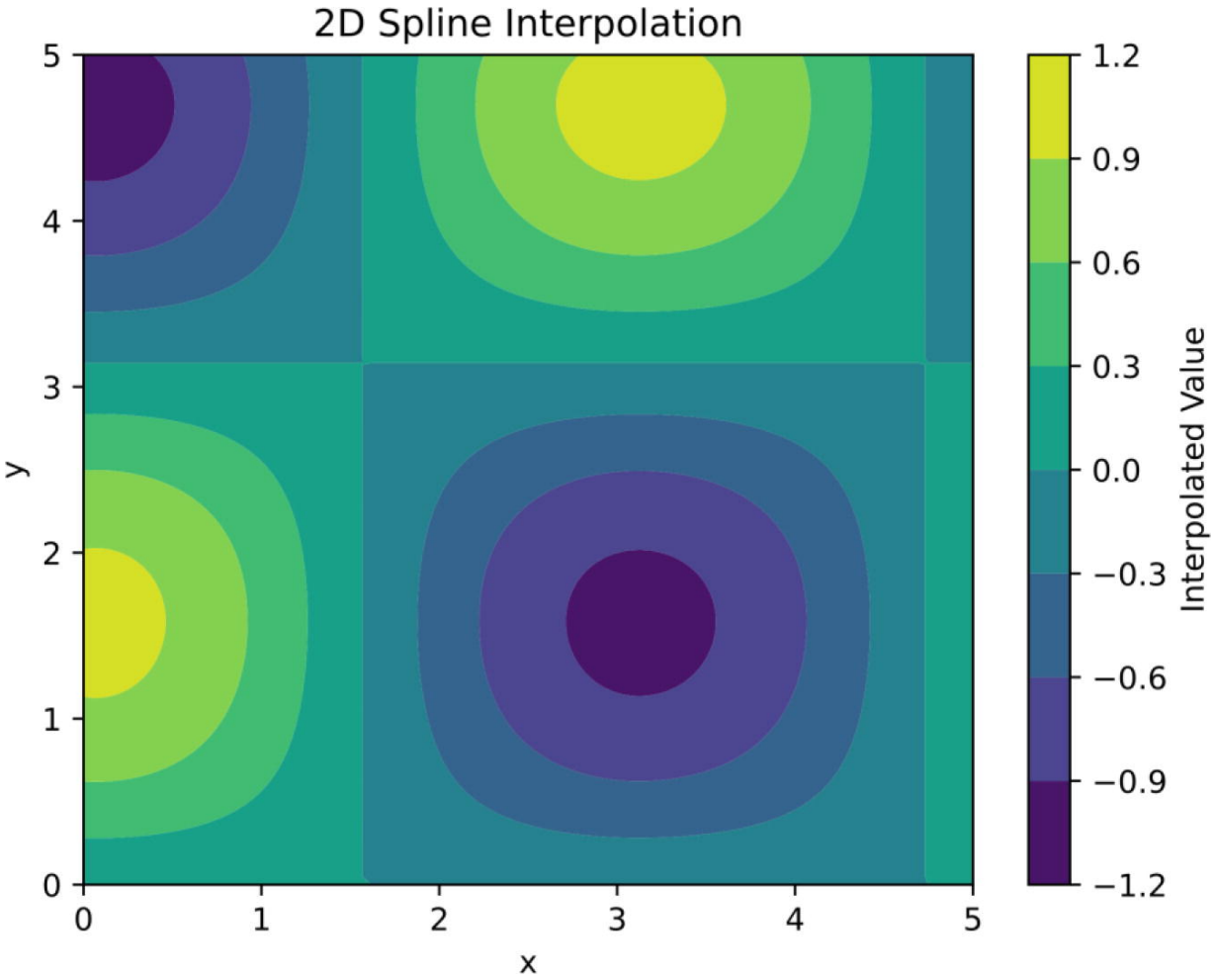


Figure 12-31 Filled contour plot of a two-dimensional spline interpolant constructed with RectBivariateSpline

Rational Approximation and Additional Tools

In addition to the functions studied above, there are two groups of approximation functions and tools. These functions offer various methods for approximating functions using either rational or polynomial representations.

Under rational approximation, there are the functions pade and AAA. The pade function returns a Pade approximation, which represents a function as the ratio of two polynomials, often providing a

more compact and accurate representation than a standard polynomial. The AAA function performs a rational approximation real or complex using an adaptive Antoulas–Anderson algorithm (AAA).

Under additional tools, there are lagrange and approximate_taylor_polynomial. The lagrange function generates a Lagrange interpolating polynomial, which is a polynomial that passes through a given set of points. The approximate_taylor_polynomial function estimates the Taylor polynomial of a function at a specific point by fitting a polynomial to sampled values, providing a local approximation of the function.

These tools are useful in a variety of applications, such as simplifying complex functions, creating easier-to-evaluate representations, or estimating function behavior near a particular point.

Linalg

The linalg subpackage is organized into eight groups. Together, these groups offer a comprehensive set of linear algebra tools for a wide range of applications, from basic matrix operations to advanced eigenvalue problems and matrix decompositions.

1. Basics
2. Eigenvalue Problems
3. Decompositions
4. Matrix Functions
5. Matrix Equation Solvers
6. Sketches and Random Projections
7. Special Matrices

8. Low-Level Routines

Each group will be studied in more detail, but to provide an overview Table 12-35 summarizes the eight groups of the scipy.linalg subpackage.

Table 12-35 Summary of function groups within the scipy.linalg subpackage

Basic Operations

The Basics group in the scipy.linalg module provides fundamental linear algebra functions that serve as the foundation for many mathematical and scientific applications. This group acts as the core toolkit for linear algebra in SciPy, offering robust and efficient solutions for a wide range of tasks from basic matrix operations to solving complex linear systems.

These functions provide the essential building blocks for developing advanced numerical algorithms with dense or structured matrices.

Table 12-36 summarizes the functions in the Basics group of the scipy.linalg module.

Table 12-36 Summary of fundamental functions in the Basics group of scipy.linalg

In addition to the functions presented in Table 12-36, there are two more dedicated operations, LinAlgError and LinAlgWarning, which are not functions, but rather an exception and a warning that can be generated by linear algebra functions.

Now see the following examples using some functions of the Basics group.

Inverse Matrix

It's worth remembering that an inverse matrix is a type of square matrix, meaning it has the same number of rows and columns. This occurs when the product of two matrices results in an identity matrix of the same order.

To calculate the inverse matrix, the function inv() will be used; the signature of this function is presented below:

```
inv(a, overwrite_a=False, check_finite=True)
```

Table 12-37 explains each parameter of the inv() function.

Table 12-37 Parameters for the scipy.linalg.inv function

The parameters overwrite_a and check_finite are optional. The returned value is ainv, which is a ndarray (N-dimensional array) representing the inverse of the input matrix a.

This code computes the inverse of a 3 × 3 matrix A using the inv function. It first imports the necessary libraries, NumPy and SciPy, and defines matrix A using NumPy's array function. The inv function is then used to calculate the inverse of A, which is stored in A_inv:

```
import numpy as np
from scipy.linalg import inv
# Define matrix A
A = np.array([[1, 3, 2],
              [4, 5, 7],
              [2, 4, 8]])
# Compute the inverse
A_inv = inv(A)
print("Inverse of A:\n", A_inv)
```

The code prints the inverse matrix, allowing for verification of the result:

```
Inverse of A:
```

```
[[-0.4          0.53333333 -0.36666667]
 [ 0.6         -0.13333333 -0.03333333]
 [-0.2         -0.06666667  0.23333333]]
```

Solving Systems

The function solve() will be utilized in this example, with its signature detailed below:

```
solve(a, b, lower=False, overwrite_a=False,
overwrite_b=False, check_finite=True,
assume_a=None, transposed=False)
```

The parameters available in this function are shown in Table 12-38.

Table 12-38 Parameters for the scipy.linalg.solve function

The lower parameter is not mandatory, but can be used to determine which part of the matrix a is used in the calculation, but only when a is considered symmetric ('sym'), hermitian ('her'), or positive definite ('pos').

When lower=True, only the lower triangle of a is used (elements on and below the diagonal). Elements above the diagonal are ignored.

When lower=False (default), only the upper triangle of a is used (elements on and above the diagonal). Elements below the diagonal are ignored.

This parameter helps optimize calculations by utilizing the symmetry of the matrix, reducing unnecessary computations.

The assume_a parameter is also not mandatory, but can be used when the data matrix type is known; you can specify it using the assume_a key, which selects a specialized solver, adapted to that matrix type. Supported options include those shown in Table 12-39.

Table 12-39 Supported matrix structures for the assume_a parameter in scipy.linalg.solve

The example code solves a system of linear equations Ax = b, where A is a 3 × 3 matrix and b is a three-element vector. It uses the solve function to find the solution vector x. The matrix A and vector b are defined using NumPy arrays, and the solution x is then printed out:

```
import numpy as np
from scipy.linalg import inv, solve, det
# Define matrix A
A = np.array([[1, 3, 2],
              [4, 5, 7],
              [2, 4, 8]])
# Define right-hand side vector b
b = np.array([1, 2, 3])
# Solve the linear system
x = solve(A, b)
print("Solution x of Ax = b:\n", x)
```

The result of this code is printed below:

```
Solution x of Ax = b:
 [-0.43333333  0.23333333  0.36666667]
```

Calculate the Determinant

The signature of the det() function is shown below, and the parameters are detailed in Table 12-40:

```
det(a, overwrite_a=False, check_finite=True)
```

Table 12-40 Parameters for the scipy.linalg.det function

This code below calculates the determinant of a 3 × 3 matrix A using the det function from SciPy's linear algebra module. The matrix A is defined using a NumPy array, and the determinant is computed and stored in the variable d:

```
import numpy as np
from scipy.linalg import inv, solve, det
# Define matrix A
A = np.array([[1, 3, 2],
              [4, 5, 7],
              [2, 4, 8]])
# Compute the determinant
d = det(A)
print(f"Determinant of A: {d:.2f}")
```

The result is then printed out with two decimal places, providing a measure of the matrix's linear independence and invertibility:

```
Determinant of A: -30.00
```

Eigenvalue Problems

The Eigenvalue Problems group is a collection categorized by their specific applications and matrix types. It outlines eight distinct functions, each tailored to solve particular types of eigenvalue problems, including ordinary, generalized, symmetric, Hermitian, band, and tridiagonal matrices.

Table 12-41 provides a concise overview of the different functions available for addressing various eigenvalue problems, highlighting the diversity of tools for tackling complex mathematical and scientific applications.

Table 12-41 Functions for solving standard and specialized eigenvalue problems in scipy.linalg

Compute Eigenvalues and Eigenvectors

The eig() function solves an ordinary or generalized eigenvalue problem for a square matrix, finding eigenvalues and right or left eigenvectors. It takes in a matrix a and optionally a right-hand-side matrix b and returns the eigenvalues w and/or the left and right eigenvectors vl and vr, depending on the specified parameters. This

function is useful for various applications in linear algebra and engineering.

The function signature is presented below, along with a description of each parameter in Table 12-42:

```
eig(a, b=None, left=False, right=True,
overwrite_a=False, overwrite_b=False,
check_finite=True, homogeneous_eigvals=False)
```

Table 12-42 Parameters for the scipy.linalg.eig function

The eig() function returns

1. *Eigenvalues (*w*)*: The values that represent how much change occurs in a linear transformation.
2. *Left Eigenvectors (*vl*)*: Returned only if left=True. These are vectors that satisfy a specific equation.
3. *Right Eigenvectors (*vr*)*: Returned by default (right=True). These are vectors that satisfy another specific equation.

Figure 12-32 provides an intuitive, geometric visualization of the concepts of eigenvectors and eigenvalues within linear algebra. It demonstrates how different vectors behave when a linear transformation, represented by a matrix **A**, is applied to the space they exist in.

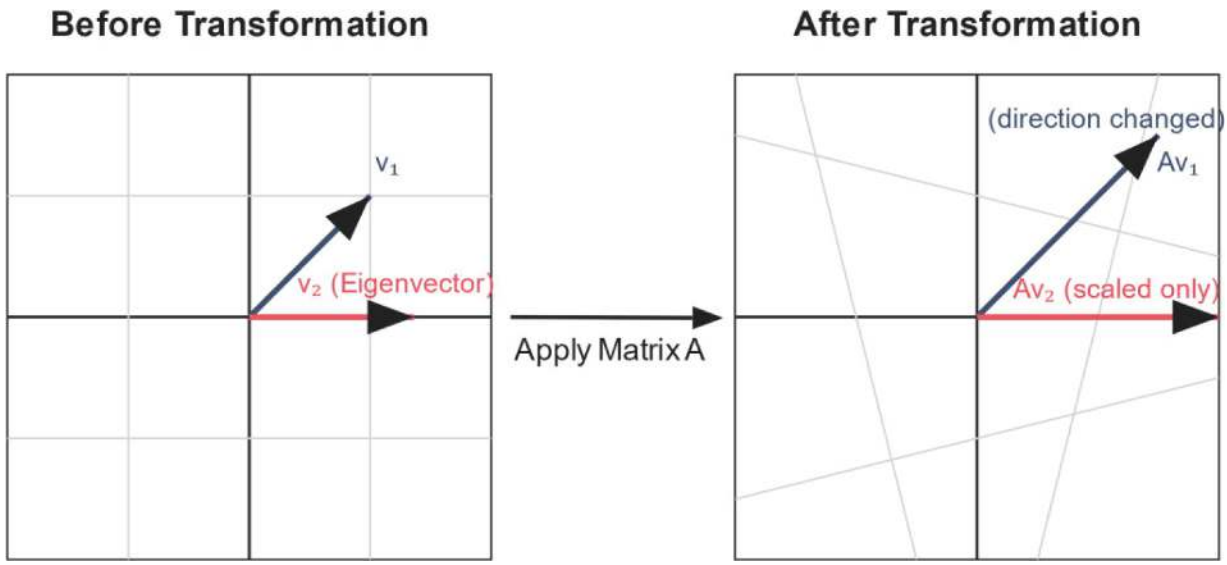


Figure 12-32 Effect of a linear transformation on an eigenvector and a vector whose direction changes

The process shown can be understood as follows:

- *Before Transformation*: The panel on the left depicts a standard grid with two vectors, $\mathbf{v_1}$ and $\mathbf{v_2}$. The red vector, $\mathbf{v_2}$, is specifically identified as an **eigenvector**.
- *After Transformation*: When the transformation **A** is applied, the grid itself is warped. The effect on the vectors is as follows:

  The vector $\mathbf{v_1}$ is transformed into $\mathbf{Av_1}$. Its direction has clearly changed, indicating it is **not** an eigenvector of this specific transformation.
  The eigenvector $\mathbf{v_2}$ is transformed into $\mathbf{Av_2}$. Crucially, its direction remains exactly the same; it has only been scaled along its original line.

This visualization perfectly illustrates the core definition of these terms: An **eigenvector** of a matrix is a special, nonzero vector that does not change its direction when the matrix's linear transformation is applied. The corresponding **eigenvalue** is the scalar value that describes how much the eigenvector is scaled (stretched, shrunk, or flipped). Functions like scipy.linalg.eig are designed to compute exactly

these properties, returning the set of eigenvalues and corresponding eigenvectors for a given matrix.

As an example, consider the following code that calculates the eigenvalues and eigenvectors of a 3 × 3 matrix A using the eig() function from SciPy's linear algebra module. The eigenvalues represent scalar values by which the matrix stretches or shrinks its eigenvectors, while the eigenvectors are the directions of these transformations:

```
from scipy.linalg import eig
import numpy as np
# Define matrix A
A = np.array([[1, 3, 2],
              [4, 5, 7],
              [2, 4, 8]])
# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = eig(A)

# Print eigenvalues
print("Eigenvalues:")
for val in eigenvalues:
    print(f"{val:.3f}")

# Print eigenvectors
print("\nEigenvectors (columns correspond to
eigenvalues):")
#Prints the eigenvectors with all elements rounded
to 3 decimal places.
print(np.round(eigenvectors, 3))
```

The code prints out the eigenvalues and eigenvectors, with the latter being rounded to three decimal places for readability. The eigenvectors are arranged column-wise, with each column corresponding to a specific eigenvalue:

```
Eigenvalues:
```

```
13.141+0.000j
-1.141+0.000j
2.000+0.000j

Eigenvectors (columns correspond to eigenvalues):
[[-0.281 -0.776 -0.577]
 [-0.701  0.622 -0.577]
 [-0.655 -0.103  0.577]]
```

Decompositions

The functions of the Decompositions collection provide various matrix decomposition techniques, which are essential tools in linear algebra and its applications. These decompositions enable the solution of systems of linear equations, computation of eigenvalues and singular values, and construction of orthonormal bases. They include LU, SVD, Cholesky, QR, polar, RQ (Triangular–Orthogonal), QZ (Quasi-triangular), and Schur decompositions, as well as specialized functions for banded and tridiagonal matrices. These functions facilitate efficient and accurate computations in fields such as control theory and signal processing.

Table 12-43 Summary of matrix decomposition functions available in scipy.linalg

More details about the operations of these decomposition functions are described below.

LU (Lower–Upper) Decomposition: A factorization of a matrix into a lower triangular matrix (L) and an upper triangular matrix (U), often used for solving systems of linear equations and finding matrix inverses.

Figure 12-33 provides a visual representation of LU (Lower–Upper) decomposition, a common and useful technique in linear algebra for factoring a square matrix. The method decomposes a single matrix **A** into the product of two specialized matrices: a lower triangular matrix **L** and an upper triangular matrix **U**.

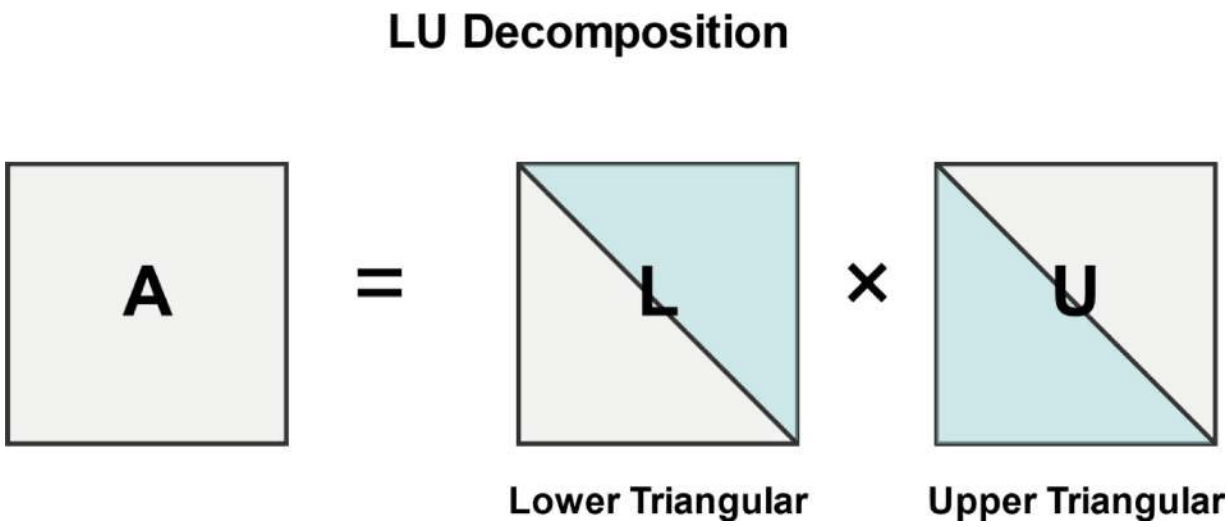


Figure 12-33 Lower and upper triangular factors in an LU decomposition, shown schematically without pivoting

The components of the decomposition are

- *L (Lower Triangular)*: A square matrix where all the elements located above the main diagonal are zero
- *U (Upper Triangular)*: A square matrix where all the elements located below the main diagonal are zero

This factorization is particularly valuable for the efficient solving of systems of linear equations and for calculating matrix inverses. In SciPy, this operation is performed by functions such as scipy.linalg.lu() and lu_factor().

SVD (Singular Value Decomposition): A factorization of a matrix into three matrices: U, Σ, and V^T, where U and V are orthogonal matrices and Σ is a diagonal matrix containing the singular values of the original matrix. SVD is widely used in applications like data compression, image processing, and latent semantic analysis.

Figure 12-34 visually represents Singular Value Decomposition (SVD), a fundamental factorization of a matrix that is widely used in applications like data compression and image processing. The process breaks down any rectangular matrix **A** of size m × n into the product of three specific matrices:

- *U*: An m × m **orthogonal** matrix. Its columns contain the left singular vectors of the original matrix **A**.
- *Σ (Sigma)*: An m × n **diagonal** matrix. The values along its main diagonal are the singular values of **A**, while all off-diagonal elements are zero.
- V^T *(V Transpose)*: An n × n **orthogonal** matrix, which is the transpose of a matrix V whose columns contain the right singular vectors of **A**.

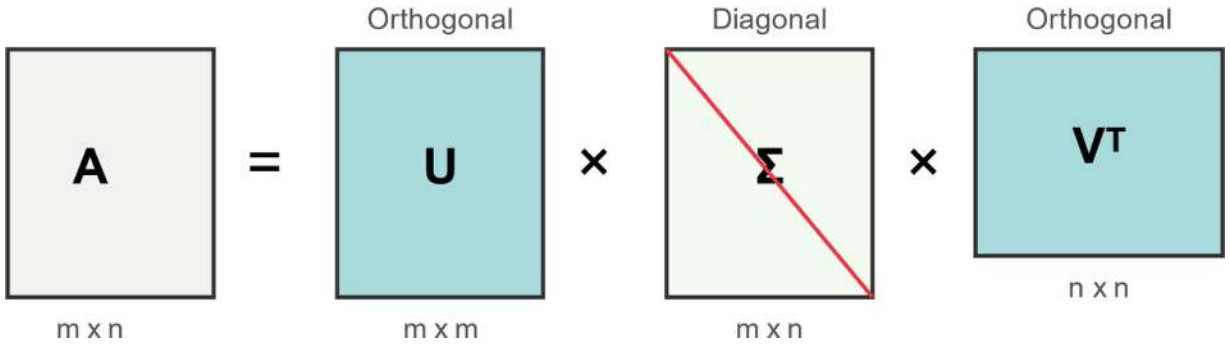


Figure 12-34 Singular value decomposition of a real matrix into orthogonal factors and a diagonal singular-value matrix

This decomposition is directly implemented in the scipy.linalg.svd function. When calling this function, it returns the matrix U, a one-dimensional array s containing the singular values that form the diagonal of Σ, and the matrix Vh that corresponds to $\boldsymbol{V^T}$. This visualization provides a clear and essential blueprint for understanding the structure and properties of the three matrices that result from the SVD computation.

LDLt (LDL Transpose) Decomposition: A factorization of a symmetric matrix into a lower triangular matrix (L), a diagonal matrix (D), and the transpose of L (L^T). It's used for solving systems of linear equations and is particularly efficient for symmetric matrices.

QR (Orthogonal–Triangular) Decomposition: A factorization of a matrix into an orthogonal matrix (Q) and an upper triangular matrix (R). QR decomposition is commonly used for solving least-squares problems, eigenvalue decomposition, and QR algorithms.

RQ (Triangular–Orthogonal) Decomposition: Similar to QR decomposition but with the order reversed, factorizing a matrix into an

upper triangular matrix (R) and an orthogonal matrix (Q). It's useful in certain applications where the RQ form is more convenient.

QZ (Quasi-triangular) Decomposition: Also known as the generalized Schur decomposition, it's a factorization of two matrices into quasi-triangular matrices. QZ decomposition is used for solving generalized eigenvalue problems, where you have a pair of matrices.

Schur Decomposition: A factorization of a matrix into a unitary matrix (Q) and a triangular matrix (T), such that A = QTQ^T, where the diagonal elements of T are the eigenvalues of A. Schur decomposition is used for solving eigenvalue problems and is particularly useful for matrices that are not diagonalizable.

Cholesky Decomposition: A factorization of a symmetric, positive-definite matrix into a lower triangular matrix (L) such that A = LL^T. Cholesky decomposition is used for solving systems of linear equations, generating random samples from multivariate normal distributions, and Monte Carlo simulations.

Singular Value Decomposition

```
svd(a, full_matrices=True, compute_uv=True,
overwrite_a=False, check_finite=True,
lapack_driver='gesdd')
```

Table 12-44 Parameters for the scipy.linalg.svd function

When computing Singular Value Decomposition (SVD), you can choose the parameter lapack_driver between two methods:

'gesdd' (divide-and-conquer approach), which is generally faster. Used by default
'gesvd' (general rectangular approach), used by MATLAB and Octave

The svd() function returns the values shown in Table 12-45.

Table 12-45 Return values for the scipy.linalg.svd function

Note K = min(M, N) and shapes depend on the full_matrices parameter. If compute_uv=False, only s is returned.

This example code performs Singular Value Decomposition (SVD) on a 3 × 3 matrix A using the svd function. SVD breaks down the matrix A into three components: U (left singular vectors), s (singular values), and VT (right singular vectors, transposed):

```
from scipy.linalg import svd
import numpy as np

# Define matrix A
A = np.array([[1, 3, 2],
              [4, 5, 7],
              [2, 4, 8]])

# Perform Singular Value Decomposition
U, s, VT = svd(A)

# Display results with 3 decimal places
print("Matrix U:")
print(np.round(U, 3))

print("\nSingular values:")
print(np.round(s, 3))

print("\nMatrix V^T:")
print(np.round(VT, 3))
```

The results are then printed out with elements rounded to three decimal places, showcasing the decomposed matrices U, singular values, and VT:

```
Matrix U:
[[-0.255 -0.584 -0.771]
```

```
 [-0.697 -0.441  0.565]
 [-0.67   0.681 -0.295]]

Singular values:
[13.521  1.98   1.121]

Matrix V^T:
[[-0.324 -0.513 -0.795]
 [-0.498 -0.622  0.604]
 [ 0.804 -0.592  0.053]]
```

Matrix Functions and Equation Solvers

These matrix operations enable various linear algebra computations, including computing matrix exponentials and logarithms, trigonometric and hyperbolic functions, and solving equations like Sylvester, Riccati, and Lyapunov equations. Additionally, randomized matrix sketching techniques like the Clarkson–Woodruff transform are used for dimensionality reduction. These operations are fundamental in applications such as control theory, signal processing, and machine learning.

Table 12-46 Summary of matrix functions and equation solvers in scipy.linalg

The Sylvester equation is a matrix equation of the form AX + XB = Q, where A and B are square matrices and Q is a matrix of compatible dimensions. It is used in various applications, including control theory, signal processing, and image processing. It has been widely studied in linear algebra.

Riccati equations are quadratic matrix equations that arise in control theory, filtering, and estimation problems. They are commonly used to solve Linear Quadratic Regulator (LQR) problems and Kalman filtering. Riccati equations can be either continuous-time or discrete-time, and their solutions provide the optimal control or estimation

gains. These equations are fundamental and have numerous applications in engineering.

Lyapunov equations are matrix equations used to analyze the stability of dynamical systems, signal processing, mechanics, and electrical engineering. They come in two forms: continuous-time (AX + XA^T = Q) and discrete-time (AXA^T - X = Q). Lyapunov equations are essential in control theory, and their solutions provide a way to determine the stability of a system.

Matrix Exponential

The function expm() is very simple and has only one parameter as shown in the function signature below:

```
expm(A)
```

The parameter A is a ndarray with input where the last two dimensions are square (..., n, n).

The example code with this function calculates the matrix exponential of a 3 × 3 matrix A using the expm() function. The matrix exponential is a fundamental concept in linear algebra and is used in various applications, such as solving systems of differential equations:

```
from scipy.linalg import expm
import numpy as np

# Define matrix A
A = np.array([[1, 3, 2],
              [4, 5, 7],
              [2, 4, 8]])

# Compute the matrix exponential
expA = expm(A)

# Print the result with 3 decimal places
print("Matrix exponential of A (exp(A)):")
```

```
print(np.round(expA, 3))
```

The result is then printed out with elements rounded to three decimal places for better readability:

```
Matrix exponential of A (exp(A)):
[[ 44384.873  75019.068 119396.551]
 [110699.86  187108.044 297800.515]
 [103384.896 174746.482 278138.767]]
```

Solving the Continuous Lyapunov Equation

The solve_continuous_lyapunov() function solves the continuous Lyapunov equation, a type of matrix equation, using the Bartels–Stewart algorithm.

This function takes two mandatory parameters:

a: A square matrix (array type)
q: Right-hand square matrix (array type)
It returns a ndarray:
x: The solution to the continuous Lyapunov equation

```
solve_continuous_lyapunov(a, q)
```

This code below solves the continuous Lyapunov equation, a type of matrix equation used in control theory and stability analysis. It starts by defining two matrices, A (a 2 × 2 system matrix) and Q (a positive semi-definite matrix), and then uses the solve_continuous_lyapunov function to find the solution matrix X that satisfies the equation A X + X A^T = -Q, where A^T is the transpose of A. The Lyapunov equation is often used to analyze the stability of linear systems and to design control systems:

```
from scipy.linalg import solve_continuous_lyapunov
import numpy as np

# Define matrix A (example 2x2 system)
```

```
A = np.array([[1, 2],
              [3, 4]])

# Define matrix Q (positive semi-definite matrix)
Q = np.array([[1, 0],
              [0, 1]])

# Solve the continuous Lyapunov equation: A X + X
A^T = -Q
X = solve_continuous_lyapunov(A, Q)

# Display the solution
print("Solution X to the continuous Lyapunov
equation:")
print(np.round(X, 3))
```

The solution X is then printed out, rounded to three decimal places:

```
Solution X to the continuous Lyapunov equation:
[[-0.9  0.7]
 [ 0.7 -0.4]]
```

Other Special Functions

The scipy.linalg module offers tools for working with other types of matrices and linear algebra operations. It includes methods for creating special matrices, such as circulant, Hankel, and Toeplitz matrices, which have specific structures and applications. It also provides low-level routines for interacting with optimized BLAS and LAPACK libraries, enabling efficient computations. These tools are valuable in fields like signal processing, statistics, and numerical analysis, where matrix operations and linear algebra are essential.

Figure 12-35 uses color-coded cells to visually demonstrate the defining patterns of three special types of matrices that are important in linear algebra and signal processing. Each distinct color represents a unique value, making the underlying structure of each matrix clear. The

scipy.linalg subpackage provides dedicated functions to construct these matrices due to their useful properties.

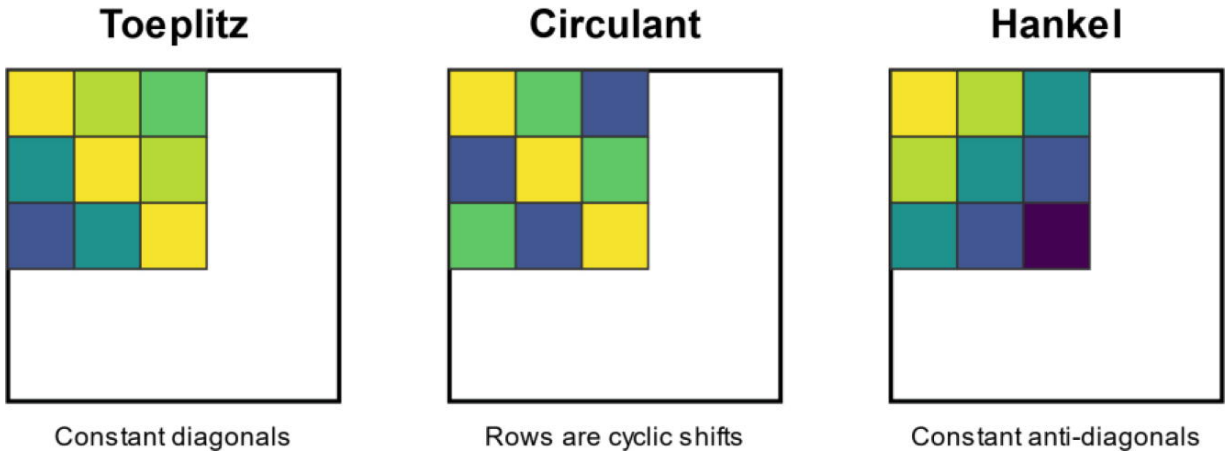


Figure 12-35 Structural patterns of Toeplitz, circulant, and Hankel matrices

The three matrix types shown are

- *Toeplitz Matrix*: This matrix is defined by having constant values along each of its diagonals. As Figure 12-35 shows, every element on a given diagonal from top left to bottom right is the same. This structure is created using the scipy.linalg.toeplitz function.
- *Circulant Matrix*: This is a special case of a Toeplitz matrix where each row is a cyclic shift of the row above it. This circular pattern is generated using the scipy.linalg.circulant function.
- *Hankel Matrix*: This matrix is defined by having constant values along its anti-diagonals (from top right to bottom left). The scipy.linalg.hankel function is used to create these matrices.

The predictable patterns are the reason these matrices are considered "special" and have optimized functions for their creation and, in the case of Toeplitz and circulant matrices, for solving systems of linear equations within SciPy.

Table 12-47 Summary of special matrix functions and low-level routines in scipy.linalg

Optimize

The SciPy optimize submodule is a comprehensive library for solving optimization and root finding problems. It provides a wide range of functions for minimizing or maximizing objective functions, possibly subject to constraints, as well as for finding roots of scalar or vector functions. This module is particularly useful for tasks such as curve fitting, least-squares optimization, and linear programming.

The library includes various algorithms for local and global optimization, including methods for unconstrained and constrained optimization, as well as specialized methods for specific problem types, such as nonlinear least squares and root finding. Additionally, it provides tools for approximating derivatives, performing line searches, and updating Hessian matrices. With its extensive set of features and flexible interface, the SciPy optimize is an essential tool for scientists, engineers, and researchers who need to solve complex optimization problems.

The SciPy optimize module can be applied in a variety of fields, including data analysis, machine learning, physics, and engineering. For example, it can be used to fit models to experimental data, optimize the parameters of a complex system, or find the minimum or maximum of a function subject to certain constraints. By providing a robust and efficient way to solve optimization problems, the SciPy optimize helps to gain insights, make predictions, and drive decision-making in their respective domains.

The scipy.optimize submodule contains many functions, which for ease of use are divided into six families of more than 70 functions.

Table 12-48 Overview of topics and function families within the scipy.optimize subpackage

To exemplify the use of the optimize submodule, we will use three of the most commonly used functions as examples.

Local Multivariate Optimization

For local multivariate optimization, we will use as an example the minimize function, a powerful tool for finding the minimum of a function that offers several algorithms and customization options. The function signature is

```
minimize(fun, x0, args=(), method=None, jac=None,
hess=None, hessp=None,
bounds=None,constraints=(), tol=None,
callback=None, options=None)
```

The minimize function requires only two mandatory parameters: fun, the objective function to minimize, and x0, the initial guess.

Table 12-49 Parameters and descriptions for the scipy.optimize.minimize function

Figure 12-36 provides a clear visual summary of the workflow for using the scipy.optimize.minimize function. It outlines the essential inputs a user must provide, the function that processes them, and the structured object that is returned as the output. The workflow consists of three main parts:

1. *Inputs*: To begin the optimization, the user must provide two mandatory arguments:
   *Objective Function (*fun*)*: A function that accepts a vector of input variables (e.g., a NumPy array) and returns the scalar value that needs to be minimized
   *Initial Guess (*x0*)*: An array or list that serves as the starting point for the algorithm's search for the minimum
2. *The* minimize() *Function*: This is the core SciPy function that takes the objective function and the initial guess, along with other optional settings (like the optimization method), and runs an iterative algorithm to find a minimum.
3. *Output (*OptimizeResult *Object)*: The function returns a single, convenient OptimizeResult object that contains all the information

about the outcome. As shown in Figure 12-36, this object has several useful attributes, including

- .x: The coordinates of the minimum that the algorithm found
- .fun: The value of the objective function at the found minimum
- .success: A boolean flag indicating whether the optimizer converged successfully
- .message: A string with more details about the termination status

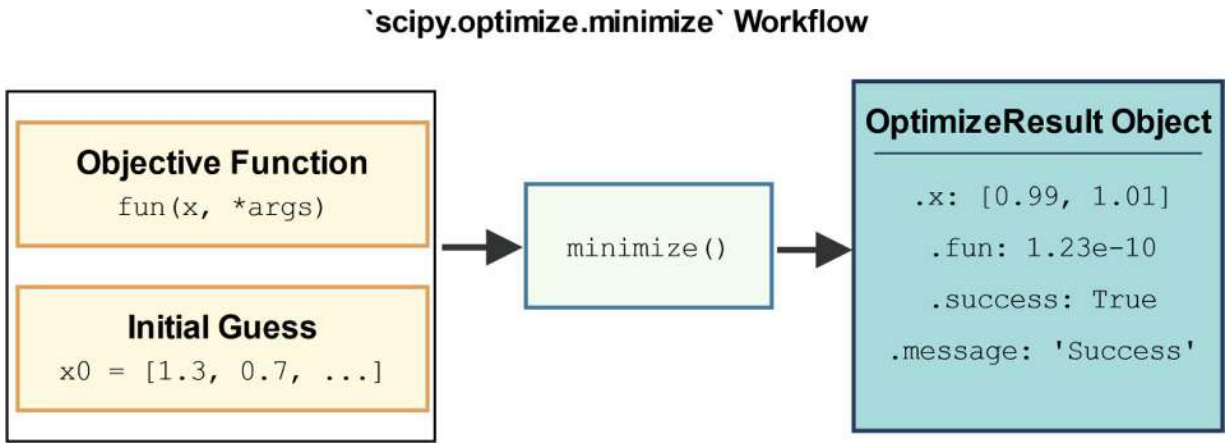


Figure 12-36 Inputs, optimization call, and OptimizeResult output of scipy.optimize.minimize

This workflow is demonstrated exactly in the text's example of minimizing the Rosenbrock function, where the rosenbrock function and an initial guess x0 are passed to minimize and the returned result object is queried for the solution.

The optimization methods available via the "method" parameter can be categorized by their underlying principles, each suited to different problem types. This overview highlights key approaches, including derivative-free methods, gradient-based techniques, and constrained optimization strategies, helping users efficiently select the best method for their needs. The method parameter has over a dozen options available, and Table 12-50 lists four of the most common ones.

Table 12-50 A selection of commonly used optimization methods available in scipy.optimize.minimize

To execute this example, we will use the Rosenbrock function, a classic test function for optimization algorithms that highlights common challenges:

Slow Convergence: The function's shape can cause algorithms to get stuck in the valley or converge slowly.
Sensitivity to Initial Conditions: A poor initial guess can significantly impact the algorithm's convergence.

The example code minimizes the Rosenbrock function, using the BFGS (Broyden–Fletcher–Goldfarb–Shanno) method. The Rosenbrock function is defined as a sum of squares of differences between consecutive elements in the input array, with a global minimum at (1, 1) for a 2D input. The code starts with an initial guess of [1.3, 0.7] and then uses the minimize function to find the optimal solution, printing the resulting values and the minimum function value achieved:

```
from scipy.optimize import minimize

# Objective function
def rosenbrock(x):
    return sum(100.0 * (x[1:] - x[:-1]**2.0)**2.0
+ (1 - x[:-1])**2.0)

# Initial guess
x0 = [1.3, 0.7]

# Minimize using BFGS method
result = minimize(rosenbrock, x0, method='BFGS')

print("Optimal solution:", result.x)
print("Minimum value:", result.fun)
```

The result shows that the optimization was successful. The optimal solution, [0.99999552 0.99999102], is very close to the global minimum of the Rosenbrock function, which is at (1, 1). The minimum value achieved, 2.0115094474352237e-11, is extremely close to zero, indicating that the optimizer effectively minimized the function. The tiny value suggests that the BFGS method worked well for this problem, converging to the global minimum with high accuracy:

```
Optimal solution: [0.99999552 0.99999102]
Minimum value: 2.0115094474352237e-11
```

Figure 12-37 provides a visual representation of how a numerical optimization algorithm works to find the minimum of a function. It uses the Rosenbrock function, which is a classic and challenging test case for optimization algorithms due to its narrow, parabolic valley.

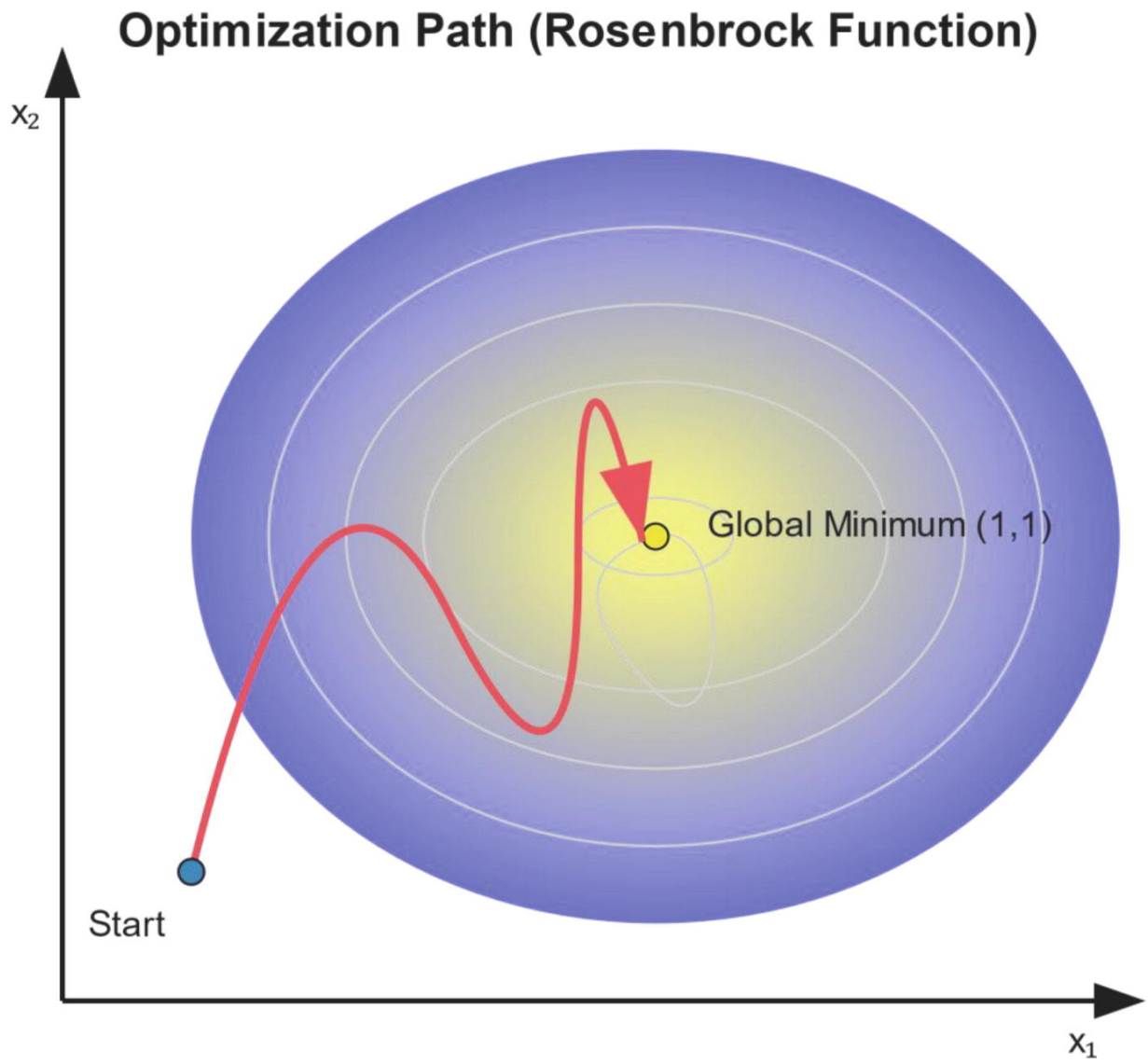


Figure 12-37 Schematic optimization path toward the minimum of the Rosenbrock function at (1, 1)

The key elements in this plot are

- *The Contour Map*: The colored background represents the value of the Rosenbrock function at every (x_1, x_2) point. The blue areas are higher values, while the yellow area represents the bottom of the function's valley where the values are lowest.
- *Start Point*: The blue dot represents the initial guess (x0) that is provided to the optimization algorithm. This is the algorithm's starting position in its search.

- *Global Minimum*: The yellow dot at coordinates (1,1) is the point where the function has its true lowest value, which is the target of the optimization process.
- *Optimization Path*: The winding red arrow illustrates the iterative steps the algorithm takes to get from the start point to the minimum. Notice the path is not a straight line; the algorithm must carefully navigate the curve of the valley to successfully converge on the solution.

This visualization is a graphical depiction of what occurs when using a function like scipy.optimize.minimize. The algorithm begins at the x0 coordinates and follows a computed path, like the one shown in red, until it finds the optimal solution result.x. The winding path visually explains why slow convergence is a known challenge when optimizing this function.

OPTIMIZATION METHODS

The optimization methods can be categorized into several groups based on their underlying principles and requirements.

These categories include methods that don't require derivatives, such as Nelder–Mead and Powell, which are useful when the objective function's gradient is unknown or difficult to compute.

Gradient-based methods, like CG, require the gradient of the objective function and are often used for large-scale optimization.

Quasi-Newton methods, such as BFGS, approximate the Hessian matrix and are efficient for smooth functions.

Hessian-based methods, like Newton-CG, utilize the Hessian matrix and are suitable for problems where the Hessian is available.

Trust-region methods, including dogleg, trust-ncg, trust-exact, and trust-krylov, are efficient for small- to medium-sized problems and use a trust region to restrict the step size.

Bound constraints methods, such as L-BFGS-B and TNC, are designed for optimization problems with bound constraints.

Finally, constrained optimization methods, including COBYLA, SLSQP, and trust-constr, handle problems with linear or nonlinear

constraints, and some of these methods are derivative-free or efficient for specific types of constraints.

Figure 12-38 provides a geometric illustration of the fundamental difference between unconstrained and constrained optimization problems. The concentric circles represent the contour lines of a function that is to be minimized, with the center of the circles being the ideal point with the lowest value.

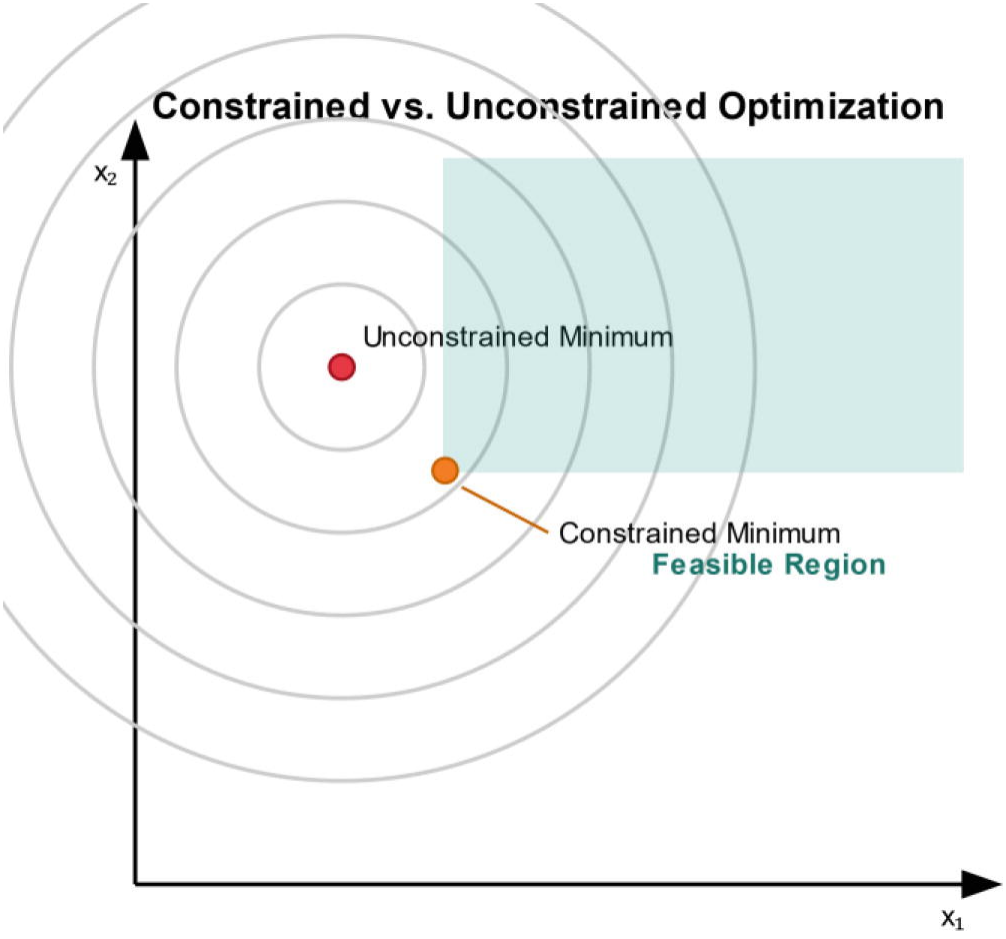


Figure 12-38 Unconstrained minimum and constrained minimum within a feasible region

The key concepts shown are

- *Unconstrained Minimum*: The red dot at the center represents the true minimum of the function. This is the optimal solution an

algorithm would find if there were no restrictions on the values of the variables (x_1, x_2).

- *Feasible Region*: The shaded blue area represents the set of all valid or "allowed" solutions as defined by a series of constraints (e.g., bounds on the variables). In a constrained problem, any acceptable solution *must* lie within this region.
- *Constrained Minimum*: The orange dot is the optimal solution for the constrained problem. It is the point that is within the feasible region and has the lowest possible function value. Often, as is the case here, the constrained minimum lies on the boundary of the feasible region, representing the closest possible point to the true minimum that still satisfies all the required conditions.

The scipy.optimize.minimize function is designed to handle both scenarios. By default, it performs unconstrained optimization. However, when the user provides arguments to the bounds and constraints parameters, the function switches to solving a constrained problem, using specialized methods like "SLSQP" or "trust-constr" to find the optimal point within the defined feasible region.

Global Optimization

As an example of global optimization, the differential_evolution function will be used. The differential_evolution function is a global optimization algorithm that uses a stochastic population-based approach to find the minimum of a multivariate function, particularly useful for complex, non-convex functions. It features a stochastic nature, population-based evolution, and derivative-free optimization, making it suitable for functions with unknown or hard-to-compute derivatives.

Figure 12-39 illustrates the fundamental difference between a local minimum and a global minimum, a crucial concept in the field of numerical optimization. The distinction between these two types of minima determines the appropriate category of optimization algorithm to use for a given problem.

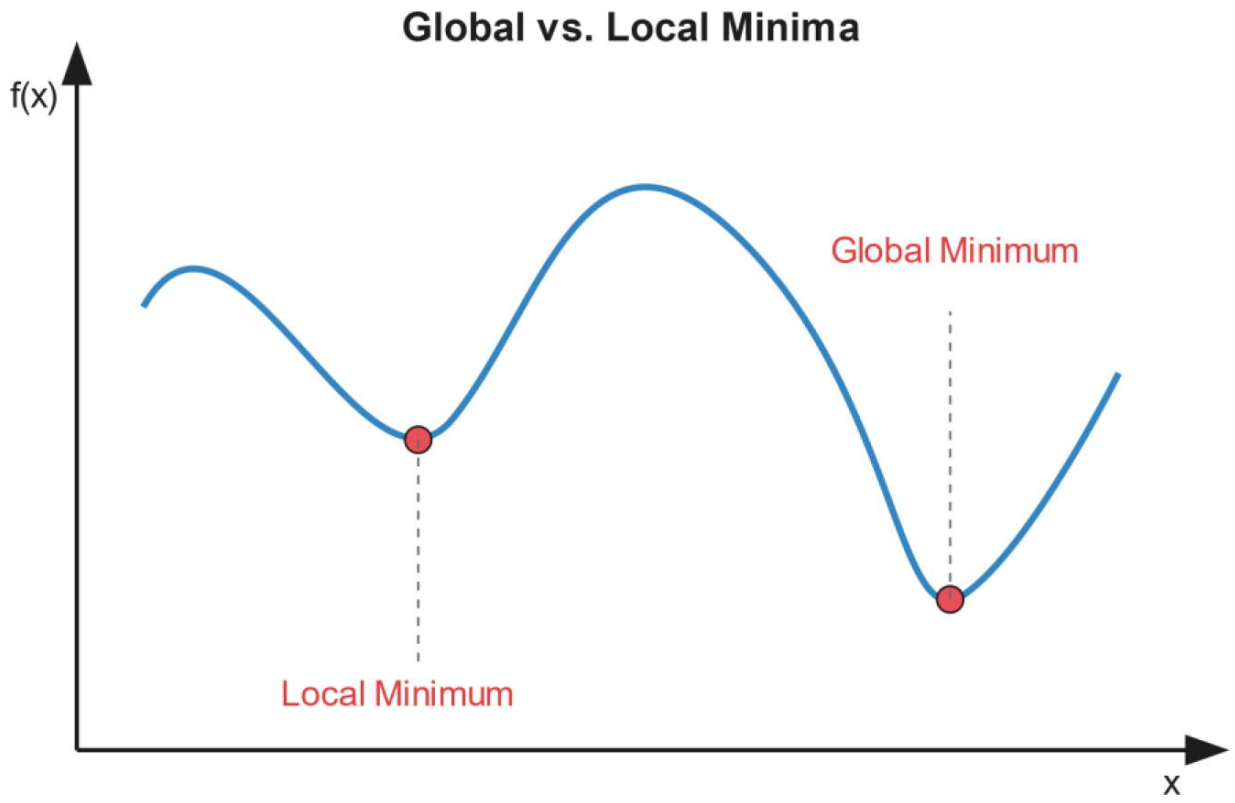


Figure 12-39 Local and global minima of a one-dimensional objective function

The scipy.optimize subpackage is structured to address this very challenge, offering distinct categories of functions for "local (multivariate) optimization" and "global optimization":

- Local optimizers, such as the minimize function, are highly efficient at finding the bottom of a single valley. However, if their starting point is within the basin of a local minimum, they will likely converge there and fail to find the global minimum.
- Global optimizers, such as the differential_evolution function, are specifically designed to avoid getting "stuck" in a local minimum. They use stochastic, population-based methods to explore the entire search space, making them better suited for complex functions where multiple minima are suspected.

The function is customizable, allowing users to tailor parameters like strategy, population size, mutation, and recombination. It also supports parallelization and vectorization, significantly speeding up

optimization for computationally expensive functions. The function returns an OptimizeResult object with the solution array, success flag, and other attributes, making it a powerful tool for solving complex optimization problems.

```
differential_evolution(func, bounds, args=(),
strategy='best1bin', maxiter=1000, popsize=15,
tol=0.01, mutation=(0.5, 1), recombination=0.7,
rng=None, callback=None, disp=False, polish=True,
init='latinhypercube', atol=0,
updating='immediate', workers=1, constraints=(),
x0=None, *, integrality=None, vectorized=False)
```

The differential_evolution function requires two mandatory parameters: “func,” the objective function to be minimized, and “bounds,” the bounds for variables specified as a sequence of (min, max) pairs for each element in x or as an instance of the Bounds class.

These two variables must be provided when calling the differential_evolution function. All other variables have default values and are optional.

Table 12-51 Parameters for the scipy.optimize.differential_evolution function

The “strategy” parameter in the differential_evolution function determines how the algorithm generates new trial solutions. It’s a crucial component of the differential evolution algorithm, as it controls the way the population evolves and converges toward the optimal solution.

In essence, the strategy defines how to combine existing solutions to create new ones. Different strategies can affect the convergence rate, exploration–exploitation trade-off, and overall performance of the algorithm.

The “strategy” parameter can be specified as a string, choosing from a set of predefined strategies, or as a custom callable function that

implements a user-defined strategy. In total there are 12 options available for the strategy parameter. Table 12-52 lists the four most used options.

Table 12-52 A selection of commonly used strategies for the differential_evolution function

The box below shows the options available for the strategy parameter.

DIFFERENTIAL EVOLUTION STRATEGIES

The differential evolution strategies can be grouped into several categories based on their characteristics.

One group includes strategies that utilize the best solution as a base vector, such as 'best1bin', 'best1exp', 'best2bin', and 'best2exp', which are suitable for general-purpose optimization and problems where convergence toward the best solution is crucial.

Another group includes strategies that use random base vectors, like 'rand1bin', 'rand1exp', 'rand2bin', and 'rand2exp', which are useful for multi-modal problems and high-dimensional spaces where exploration is essential.

Additionally, there are strategies that employ a random-to-best or current-to-best approach, such as 'randtobest1bin', 'randtobest1exp', 'currenttobest1bin', and 'currenttobest1exp', which balance exploration and exploitation and are suitable for problems where leveraging current good solutions is important.

These groups of strategies can be chosen based on the specific characteristics of the optimization problem at hand.

The following example uses the differential_evolution function to find the global minimum of a sphere function, which is a simple objective function that calculates the sum of squares of its input variables. The function is defined over a 2D space with bounds of (–5, 5) for each variable. The differential_evolution algorithm is a global optimization

technique that searches for the minimum value within the specified bounds:

```
from scipy.optimize import differential_evolution

# Objective function
def sphere(x):
    return sum(xi**2 for xi in x)

# Bounds for each variable
bounds = [(-5, 5), (-5, 5)]

# Run global optimizer
result = differential_evolution(sphere, bounds)

print("Global minimum at:", result.x)
print("Minimum value:", result.fun)
```

The code prints the location of the global minimum (the values of the variables that achieve the minimum) and the minimum value itself:

```
Global minimum at: [0. 0.]
Minimum value: 0.0
```

Curve Fitting/Least Squares

The curve_fit function uses nonlinear least squares to fit a user-defined function to a set of data points. It takes in the function to be fitted, the independent variable data, and the dependent variable data and returns the optimal parameters for the function that minimize the sum of the squared residuals. The function also allows for various options, such as specifying initial guesses for the parameters, bounds for the parameters, and the method to use for optimization.

Additionally, it can handle uncertainties in the dependent variable data and provide estimates of the covariance of the fitted parameters.

The function is useful for fitting complex models to data and estimating the parameters of the model.

```
curve_fit(f, xdata, ydata, p0=None, sigma=None,
absolute_sigma=False, check_finite=None, bounds=(-
inf, inf), method=None, jac=None, *,
full_output=False,
nan_policy=None, **kwargs)
```

The following parameters are mandatory for the curve_fit function:

1. f (the model function to fit)
2. xdata (the independent variable data)
3. ydata (the dependent variable data)

These three parameters must be provided when calling the curve_fit function. All other parameters are optional and have default values.

Table 12-53 Parameters for the scipy.optimize.curve_fit function

The dictionary of optional outputs, infodict, contains additional information about the optimization process. Its keys include nfev, which stores the number of function calls; fvec, which contains the residual values evaluated at the solution; fjac, a permutation of the R matrix of a QR factorization of the final approximate Jacobian matrix; ipvt, an integer array defining a permutation matrix; and qtf, the vector resulting from the product of the transpose of q and fvec. These values provide insight into the optimization process and can be useful for debugging or further analysis.

Table 12-54 Return values and optional outputs for the scipy.optimize.curve_fit function

DICTIONARY WITH ADDITIONAL INFORMATION
The dictionary of optional outputs, "infodict," contains additional information about the optimization process when "full_output=True". For method "lm", its keys include "nfev", "fvec", "fjac", "ipvt", and "qtf". However, for other methods like "trf" and "dogbox", not all of these keys may be present, and "nfev" might not count function calls for numerical Jacobian approximation. These values provide insight into the optimization process and can be useful for debugging or further analysis.

The example code of curve fitting defines a model function “model(x, a, b) = a * exp(b * x)” and generates sample data with added noise. It then uses curve_fit to fit the model to the data, determining the optimal parameters a and b that minimize the sum of the squared residuals:

```
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# Define the model function
def model(x, a, b):
    return a * np.exp(b * x)

# Sample data
x_data = np.linspace(0, 4, 50)
y_data = model(x_data, 2.5, -1.3) + 0.2 *
np.random.normal(size=len(x_data))

# Fit the model
params, _ = curve_fit(model, x_data, y_data)

print("Fitted parameters: a =", params[0], ", b
=", params[1])

# Plot
```

```
plt.scatter(x_data, y_data, label='Data')
plt.plot(x_data, model(x_data, *params),
color='red', label='Fitted curve')
plt.legend()
plt.show()
```

The fitted parameters are printed, showing a ≈ 2.50 and b ≈ –1.28, which are close to the original values a = 2.5 and b = –1.3. The output parameters indicate a good fit, with values close to the actual parameters used to generate the data:

```
Fitted parameters: a = 2.4960527915025317 , b =
-1.2794349654602735
```

The code plots the original data points and the fitted curve, visualizing the quality of the fit.

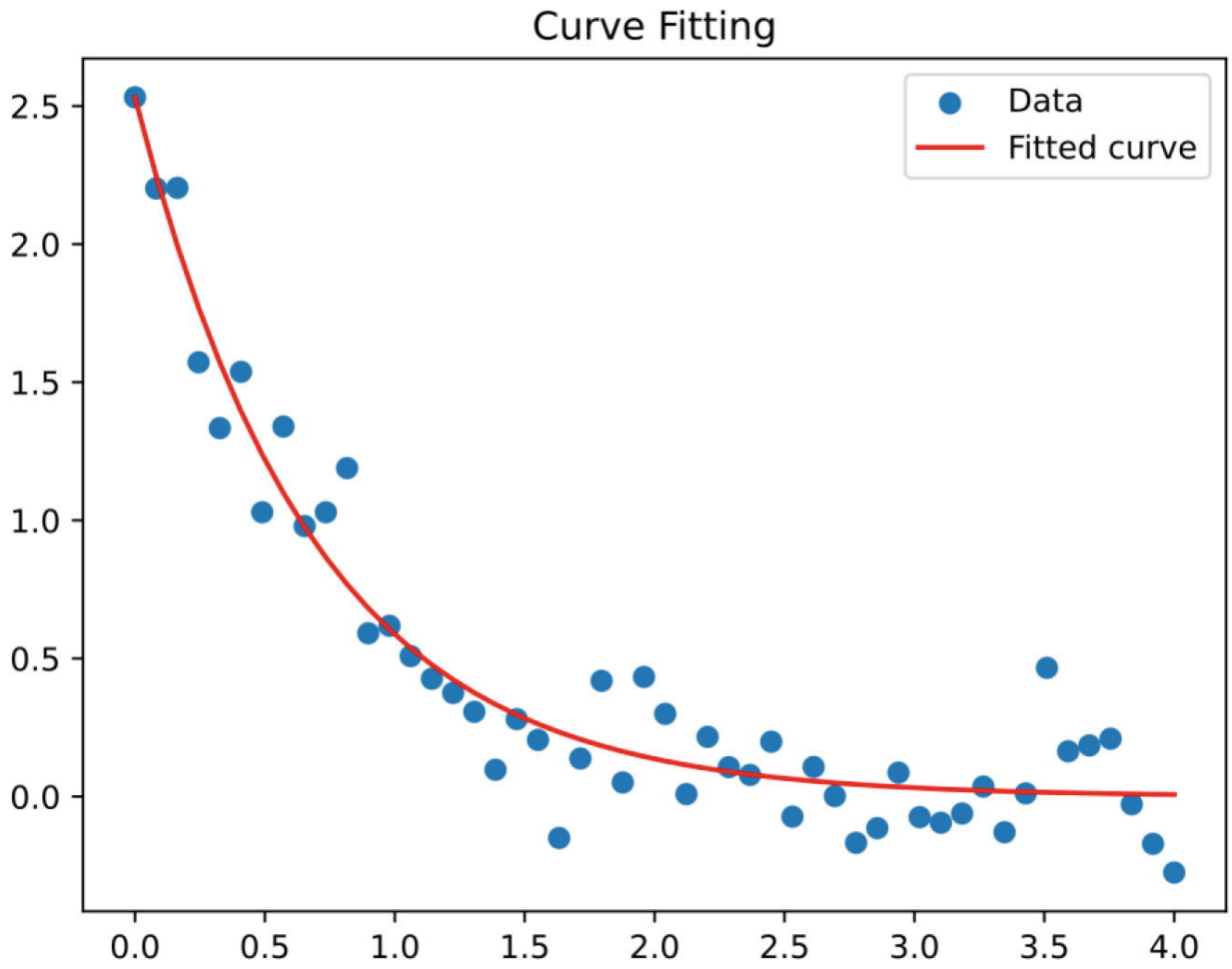


Figure 12-40 Exponential model fitted to noisy data using scipy.optimize.curve_fit

Signal

The scipy.signal submodule provides a wide range of functions for signal processing. It includes tools for tasks like convolution and correlation, filtering using both finite impulse response (FIR) and infinite impulse response (IIR) filters, filter design, spectral analysis, and more.

In total, there are approximately over 100 functions listed in the scipy.signal submodule. Therefore, the functions will be grouped to facilitate comprehension and coverage of the functions in this important submodule.

Figure 12-41 provides a clear, step-by-step visualization of one-dimensional convolution, a fundamental operation in signal processing.

Convolution is the process of applying a kernel (g), which can be thought of as a filter or an effect to an input signal (f) to produce an output. This operation is central to many applications, such as applying audio effects, smoothing data, and filtering images.

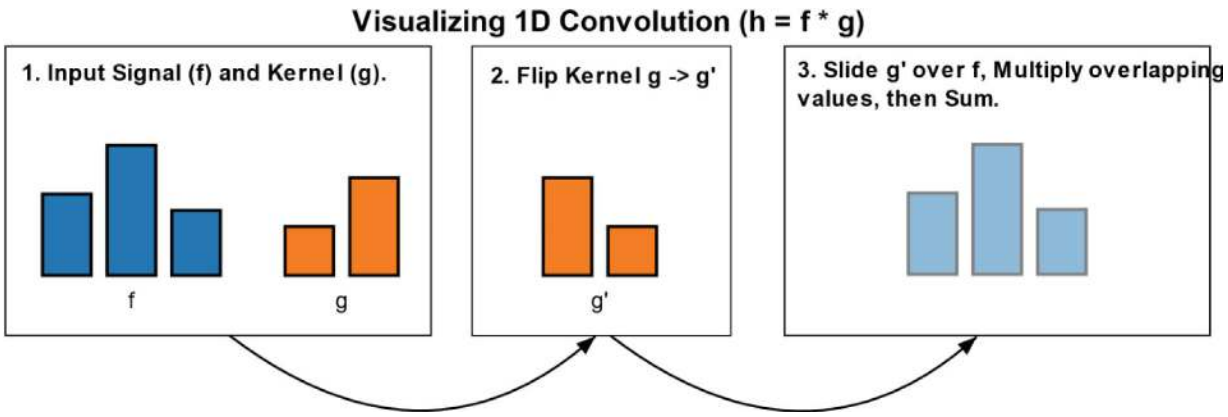


Figure 12-41 One-dimensional convolution illustrated by reversing the kernel, shifting it, and summing products of overlapping samples

The process, as depicted, can be broken down into three stages:

1. *Start with Inputs*: The operation begins with two discrete signals: the input signal (f) and the kernel (g).
2. *Flip the Kernel*: In the first mathematical step of convolution, the kernel (g) is flipped or reversed in order.
3. *Slide, Multiply, and Sum*: The flipped kernel is then slid across the input signal from one end to the other. At each position, the values of the signal and the kernel that overlap are multiplied together, and these products are summed up to create a single point in the output signal. This process is repeated at every position to generate the complete convolved signal.

This entire “flip, slide, and sum” operation is performed by functions within the scipy.signal subpackage, such as convolve, fftconvolve, and oaconvolve. Figure 12-41 helps demystify the abstract mathematics of convolution by showing it as a tangible, mechanical

process, providing an intuitive understanding of what happens when these SciPy functions are called.

Table 12-55 Main categories of functions available in the scipy.signal subpackage

Figure 12-42 illustrates the relationship between a signal whose frequency changes over time (a non-stationary signal) and its representation in the time-frequency domain.

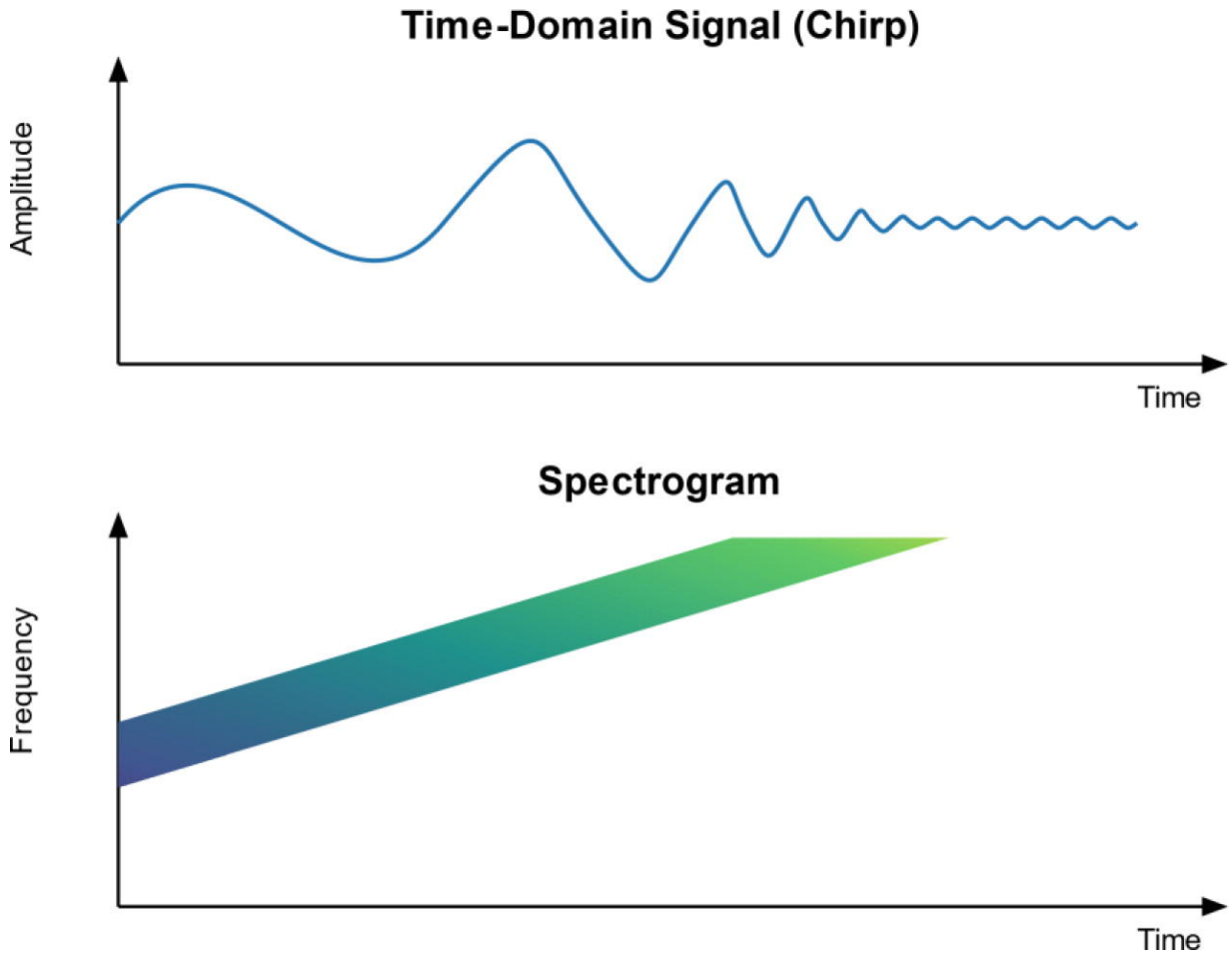


Figure 12-42 Time-domain chirp signal and schematic spectrogram showing an increasing frequency over

The two plots show

- *Time-Domain Signal (Chirp)*: The top plot displays a “chirp” signal. Its defining characteristic is that its frequency changes over time; in this

case, the frequency increases, which can be seen by the oscillations becoming progressively faster and closer together. Waveforms like this can be generated in SciPy using the scipy.signal.chirp function.
- *Spectrogram*: The bottom plot is a spectrogram, a powerful tool that visualizes the frequency content of a signal as it evolves. The horizontal axis is time, the vertical axis is frequency, and the color intensity represents the magnitude of a frequency component at a particular instant. The rising diagonal band of color clearly shows that the signal's primary frequency increases from low to high over the time period.

A spectrogram is a visual representation of the Short-Time Fourier Transform (STFT) and is essential for analyzing signals whose frequency is not constant. This type of analysis can be performed in SciPy using the scipy.signal.spectrogram function. Figure 12-42 effectively demonstrates why a spectrogram is more informative than a standard FFT for such signals, as it reveals the dynamic, time-varying nature of the frequency content.

Figure 12-43 illustrates two fundamental methods for analyzing the dynamic behavior of a continuous-time linear time–invariant (LTI) system: the step response and the impulse response. These are standard characterization techniques in fields like control theory and signal processing, and they can be computed directly using functions within SciPy's signal subpackage.

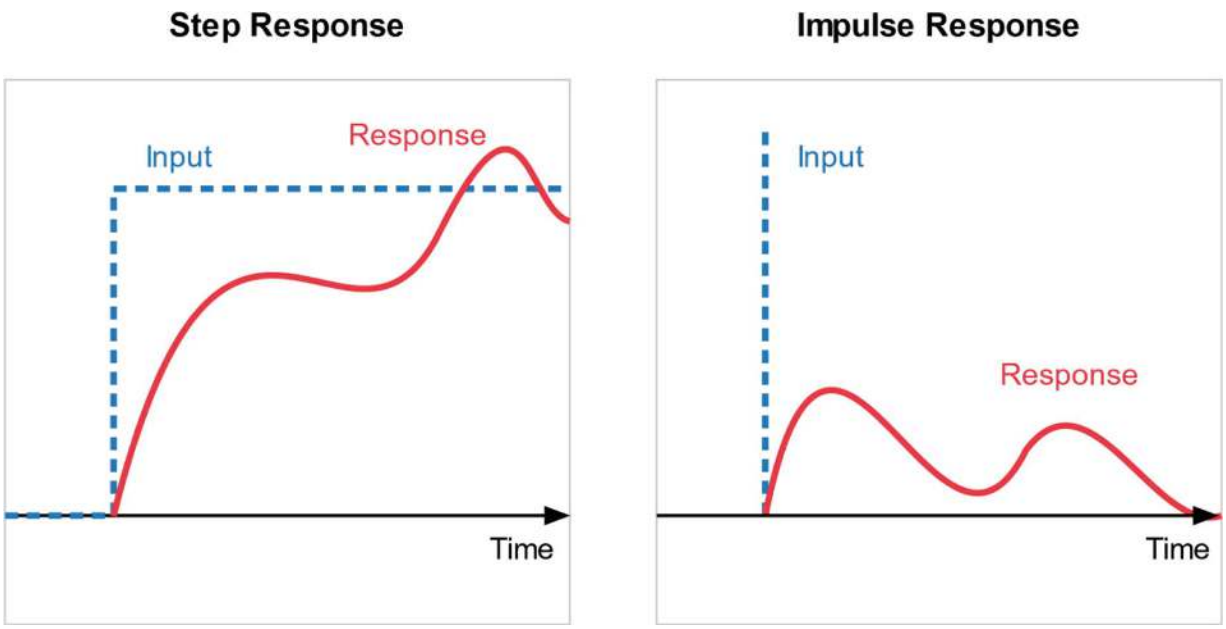


Figure 12-43 Step and impulse responses of a continuous-time linear time-invariant system

The two plots show

- *Step Response (Left)*: This graph shows the system's output (the red "Response" curve) when the input is a step function (the blue dashed line). A step input is a signal that abruptly switches from zero to a constant value and stays there. The response reveals critical system characteristics like how fast it responds, whether it overshoots its target, and how long it takes to settle at a steady state.
- *Impulse Response (Right)*: This graph shows the system's output when the input is an impulse function—a theoretical, infinitesimally short and infinitely high spike. The impulse response reveals the system's natural dynamics in reaction to a sudden "kick" or shock, showing how it dissipates that initial energy.

The scipy.signal module provides dedicated tools for this type of analysis. The scipy.signal.step and scipy.signal.impulse functions are used to compute the step and impulse responses, respectively, for a given LTI system model.

MATLAB Compatibilities

MATLAB and SciPy's signal processing functions share many similarities. There are several functions compatible with MATLAB in the sense that they implement similar filter design algorithms and have similar functionality. However, the syntax and some of the specific options may differ.

Table 12-56 shows some examples of equivalent functions.

Table 12-56 A list of MATLAB-compatible filter design functions in scipy.signal

In general, the SciPy functions follow the same basic syntax and structure as their MATLAB counterparts, but may have some differences in terms of optional arguments, default values, and output formats. If you're familiar with MATLAB's filter design functions, you should be able to adapt to the SciPy functions with some minor adjustments.

Example Codes

In the first example using the submodule signal, we will study the function "butter". This function designs a Butterworth digital or analog filter, returning filter coefficients. It takes parameters such as filter order (N), critical frequency (Wn), filter type (lowpass, highpass, bandpass, or bandstop), and output format ('ba', 'zpk', or 'sos'). The function allows for both digital and analog filter design, with options for specifying sampling frequency (fs).

Filter Design (Butterworth)

The function signature of the function butter is presented below:

```
butter(N, Wn, btype='low', analog=False,
output='ba', fs=None)
```

The mandatory parameters are N (filter order, integer type) and Wn (critical frequency, scalar or length-2 sequence). The other parameters, such as btype, analog, and output, are optional and have default values.

Table 12-57 Parameters for the scipy.signal.butter function

Figure 12-44 graphically illustrates the ideal frequency responses for the four fundamental types of filters used in signal processing. Each plot shows the filter's magnitude response across a range of frequencies. The sections where the red line is high represent the "passband," indicating the frequencies the filter allows to pass through. Conversely, sections where the line is low represent the "stopband," where frequencies are attenuated or blocked.

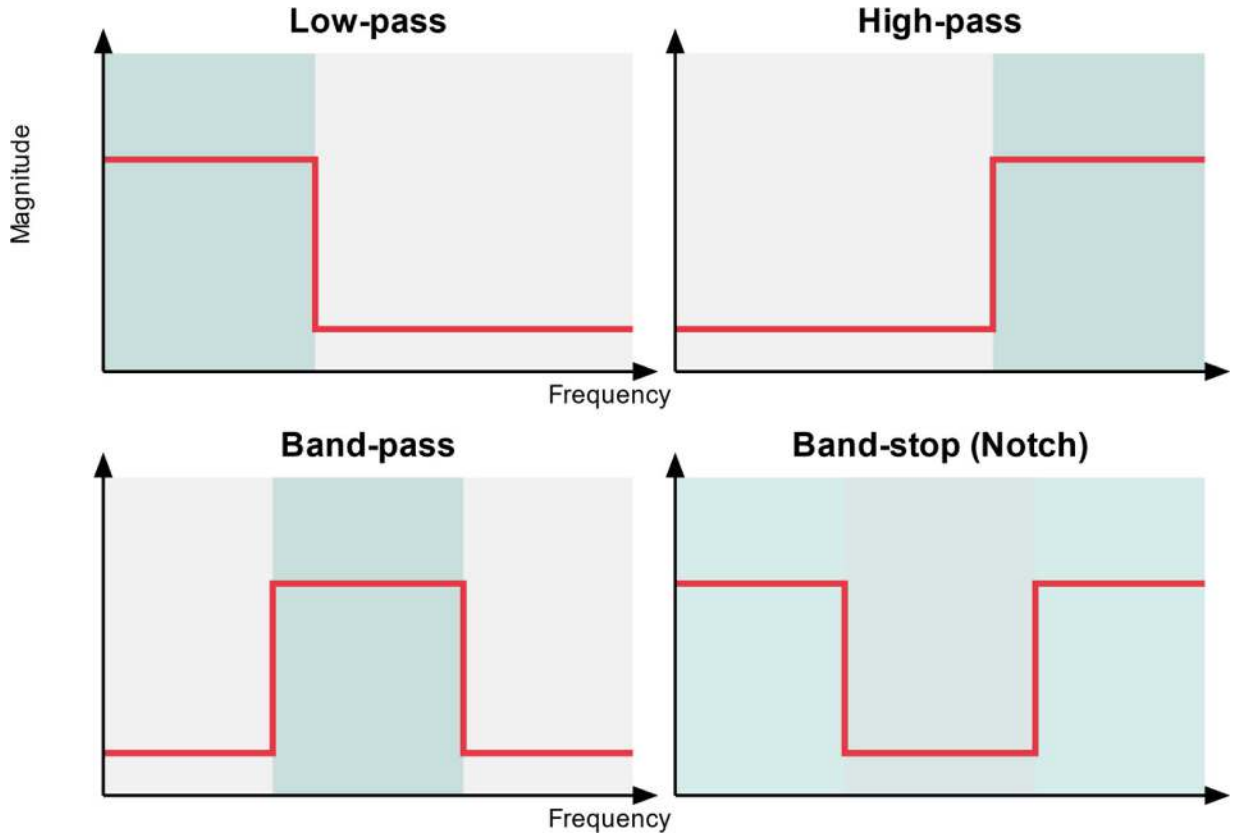


Figure 12-44 Ideal magnitude responses of low-pass, high-pass, band-pass, and band-stop filters

The four filter types are

- *Low-Pass*: This filter allows low-frequency signals to pass through while attenuating high-frequency signals.
- *High-Pass*: This filter performs the opposite function, allowing high-frequency signals to pass while attenuating low-frequency ones.

- *Band-Pass*: This filter allows signals only within a specific frequency range to pass through, attenuating signals that are outside of that defined band.
- *Band-Stop (Notch)*: This filter, often called a notch filter, attenuates signals within a specific frequency range while allowing all other frequencies (both lower and higher) to pass through.

These filter behaviors are central to the scipy.signal subpackage. Functions like butter and firwin allow you to design these specific filters by setting the btype parameter to 'lowpass', 'highpass', 'bandpass', or 'bandstop'. This visualization provides an essential conceptual guide for understanding the purpose of each filter before implementing them in code.

The following example generates a noisy sine wave signal with a frequency of 5 Hz, sampled at 500 Hz, and then applies a fourth-order low-pass Butterworth filter with a cutoff frequency of 10 Hz to remove high-frequency noise. The original noisy signal, filtered signal, and original clean sine wave are then plotted for comparison, demonstrating the effectiveness of the filter in reducing noise while preserving the underlying signal. The code utilizes NumPy for numerical computations, SciPy for signal processing, and Matplotlib for visualization:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt

# Create noisy sine wave
fs, f, t = 500, 5, np.linspace(0, 1, 500,
endpoint=False)
signal = np.sin(2 * np.pi * f * t) + 0.5 *
np.random.randn(500)

# Low-pass Butterworth filter
b, a = butter(4, 10 / (0.5 * fs), btype='low')
```

```
filtered = filtfilt(b, a, signal)

# Plot
plt.plot(t, signal, label='Noisy', alpha=0.6)
plt.plot(t, filtered, label='Filtered',
linewidth=2)
plt.plot(t, np.sin(2 * np.pi * f * t), '--',
label='Original')
plt.legend(), plt.title('Butterworth Low-Pass
Filter'), plt.grid()
plt.xlabel('Time [s]'), plt.ylabel('Amplitude')
plt.show()
```

The code's result is a plot that effectively demonstrates the application of a low-pass Butterworth filter to remove high-frequency noise from a signal. The filtered signal closely follows the original clean sine wave, showcasing the filter's success in reducing noise while preserving the underlying signal's characteristics. The comparison between the noisy, filtered, and original signals visually illustrates the filter's effectiveness in signal processing.

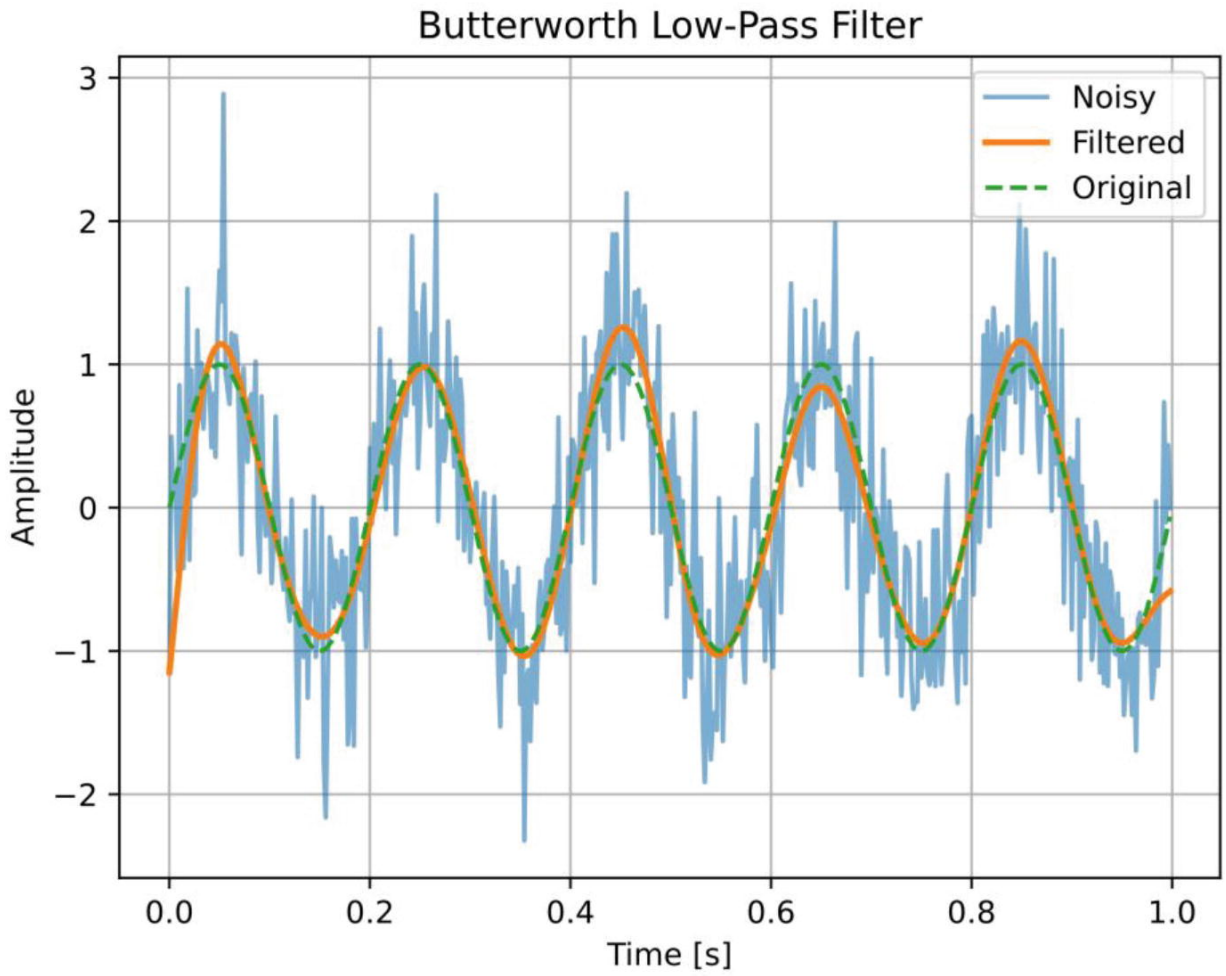


Figure 12-45 Original, noisy, and Butterworth low-pass-filtered signals

Power Spectral Density with Periodogram

The periodogram function estimates the power spectral density (PSD) of a signal, providing insight into its frequency content. It takes parameters such as the input signal, sampling frequency, window type, and scaling options and returns the sample frequencies and corresponding PSD values. The function allows for customization, including detrending and choice of scaling (density or spectrum), making it a versatile tool for spectral analysis of signals.

Although we have already introduced the FFT submodule of SciPy, this module provides spectral analysis capabilities, but differs in approach: the periodogram is a specialized function for estimating power spectral density with features such as windowing and

detrending, while scipy.fft offers a more basic Fast Fourier Transform functionality, requiring manual handling of these aspects.

```
periodogram(x, fs=1.0, window='boxcar', nfft=None,
detrend='constant', return_onesided=True,
scaling='density', axis=-1)
```

The only mandatory parameter is “x”, the time series of measurement values. All other parameters are optional and have default values.

Table 12-58 Parameters for the scipy.signal.periodogram function

The example code for this function generates a composite signal consisting of two sine waves (5 Hz and 20 Hz) and analyzes its power spectrum using the periodogram function from SciPy. The resulting power spectral density is plotted on a semi-log scale, highlighting the peaks at 5 Hz and 20 Hz, which correspond to the frequencies of the original signal components:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import periodogram

fs, t = 500, np.linspace(0, 1, 500,
endpoint=False)
s = np.sin(2*np.pi*5*t) + 0.5*np.sin(2*np.pi*20*t)
f, p = periodogram(s, fs)

plt.semilogy(f, p)
plt.axvline(5, color='r', ls='--', alpha=0.7)
plt.axvline(20, color='g', ls='--', alpha=0.7)
plt.text(5.5, max(p)/3, '5 Hz', color='r')
plt.text(20.5, max(p)/5, '20 Hz', color='g')
plt.title('Power Spectrum of Composite Signal')
```

```
plt.xlabel('Frequency [Hz]'); plt.ylabel('Power')
plt.grid(); plt.tight_layout()
plt.show()
```

The plot is annotated with vertical lines and labels to emphasize the frequency components.

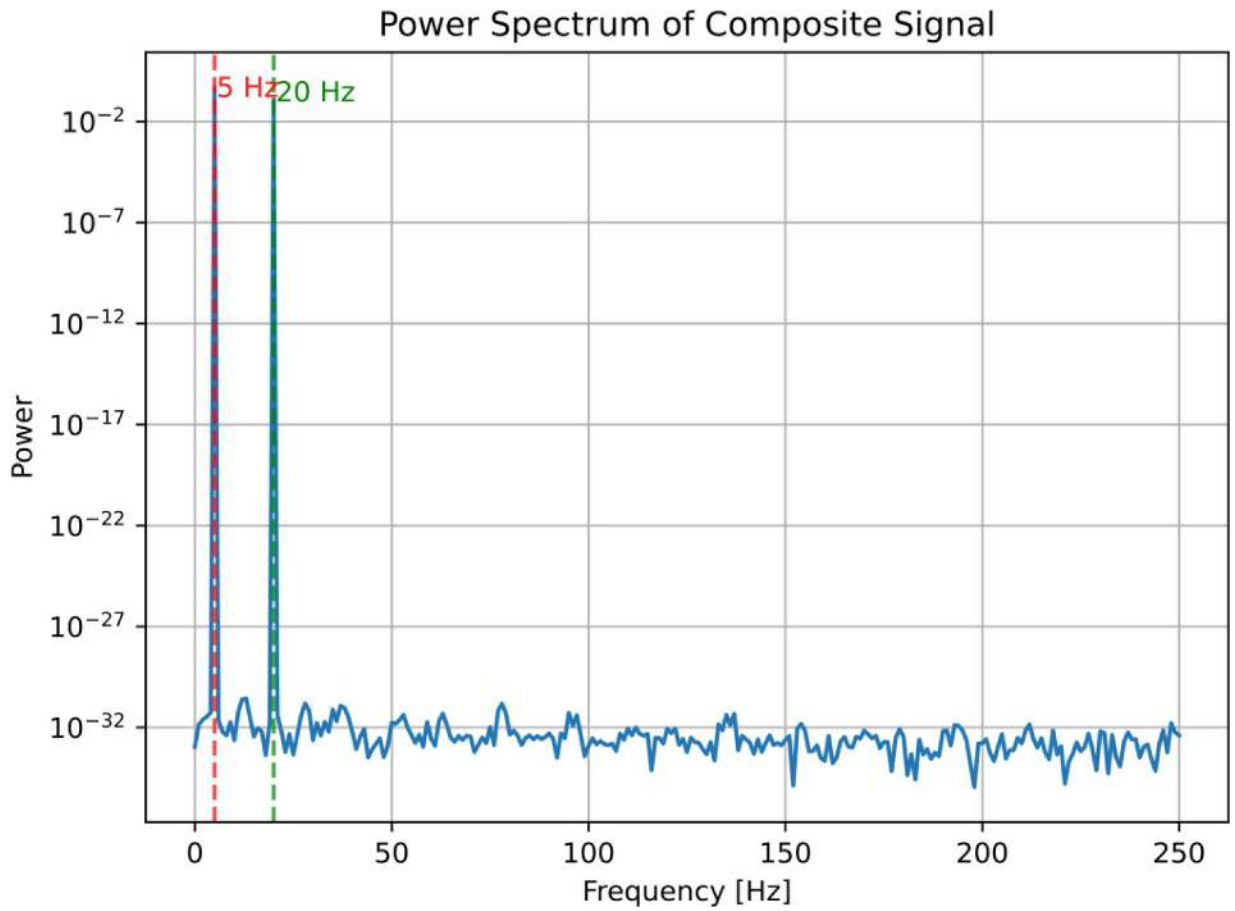


Figure 12-46 Periodogram of a composite signal containing 5 Hz and 20 Hz sinusoidal components

Design of a FIR Filter

```
firwin(numtaps, cutoff, *, width=None,
window='hamming',pass_zero=True,
scale=True, fs=None)
```

The firwin function requires two mandatory parameters: “numtaps”, which specifies the length of the filter, and “cutoff”, which defines the cutoff frequency or frequencies of the filter. All other parameters, such as width, window, pass_zero, scale, and fs, are optional and have default values.

The “*” in the function signature indicates that all parameters after cutoff are keyword-only arguments. This means that width, window, pass_zero, scale, and fs can only be specified using their keyword names, not by positional arguments. For example, you would need to use firwin(numtaps, cutoff, window=’hamming’) instead of firwin(numtaps, cutoff, ‘hamming’).

Table 12-59 Parameters for the scipy.signal.firwin function

The example code below designs a low-pass FIR (finite impulse response) filter using the firwin function with a Hamming window and 51 coefficients. The filter’s cutoff frequency is set at 0.3 (normalized, where 1.0 corresponds to the Nyquist frequency). The frequency response of the designed filter is computed using the freqz function, which returns both the frequency values and the corresponding complex frequency response:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import firwin, freqz

# Filter parameters
numtaps = 51            # Number of filter
coefficients (filter order + 1)
cutoff = 0.3            # Normalized cutoff
frequency (0.5 corresponds to Nyquist)

# FIR low-pass filter design using Hamming window
b = firwin(numtaps, cutoff, window='hamming')
```

```
# Compute the frequency response
w, h = freqz(b)

# Plot the frequency response
plt.figure(figsize=(10, 5))
plt.plot(w / np.pi, 20 * np.log10(np.abs(h)),
label='Magnitude (dB)')
plt.title('Frequency Response of FIR Low-Pass
Filter')
plt.xlabel('Normalized Frequency (×π rad/sample)')
plt.ylabel('Magnitude (dB)')
plt.grid(True)
plt.legend()
plt.show()
```

The code plots the magnitude of the response in decibels (dB) against the normalized frequency, providing a visual representation of the filter's behavior across the frequency spectrum.

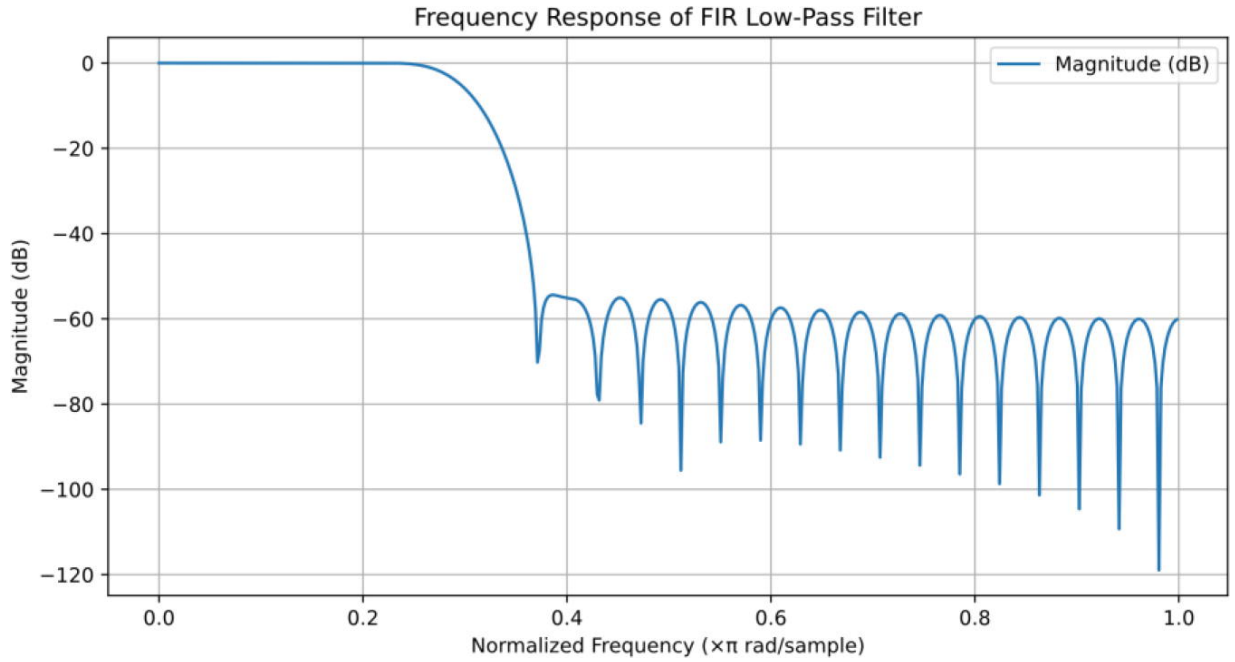


Figure 12-47 Magnitude response in decibels of a 51-tap FIR low-pass filter designed with a Hamming window

LTI System Conversions

SciPy provides several functions for converting representation forms of linear systems. Figure 12-48 serves as a valuable roadmap, visually organizing the different LTI models and clearly showing which SciPy function to use for each conversion task.

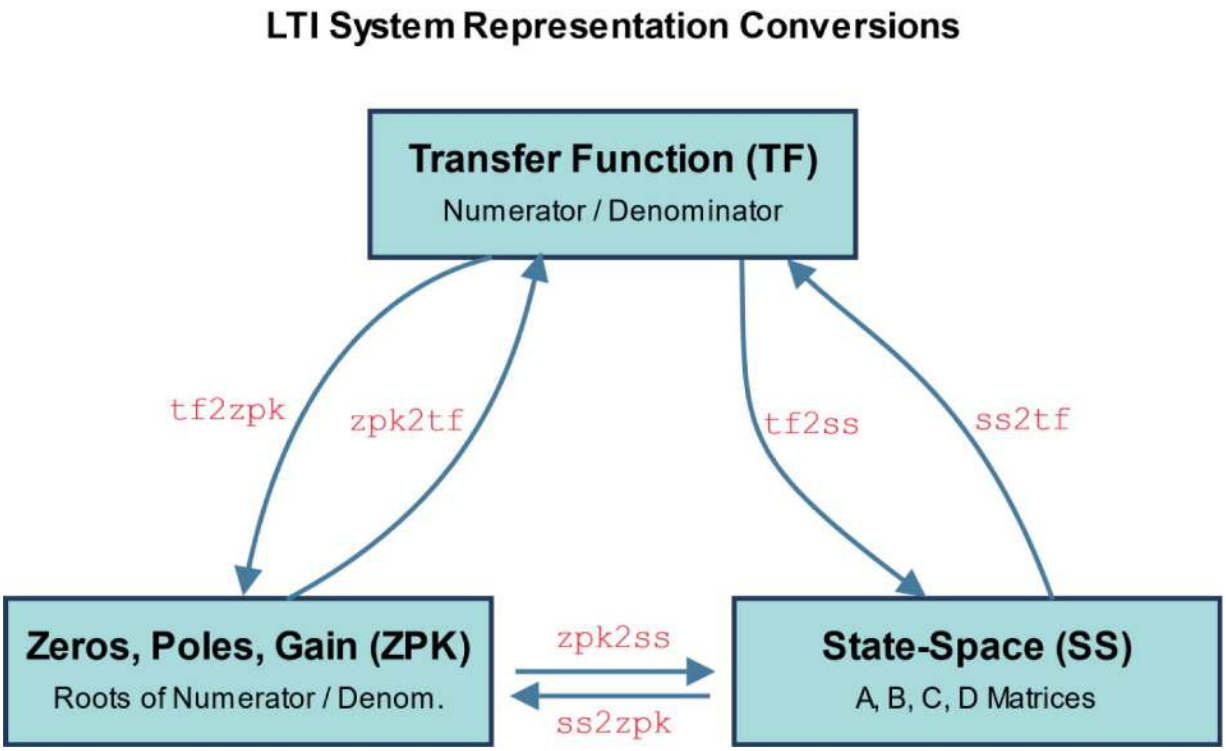


Figure 12-48 Conversions between transfer-function, zero-pole-gain, and state-space representations of linear time-invariant systems

The flowchart illustrates the three primary mathematical models used to represent a linear time–invariant (LTI) system and the functions available in scipy.signal to convert between them. Each representation provides a different perspective on a system's behavior and is advantageous for different types of analysis in signal processing and control theory.

The three system representations are

- *Transfer Function (TF)*: Describes the system's input–output relationship using a ratio of two polynomials: a numerator and a denominator

- *Zeros, Poles, Gain (ZPK)*: Represents the system using the roots of the transfer function's numerator (the "zeros") and the roots of the denominator (the "poles"), along with an overall system gain factor
- *State-Space (SS)*: A matrix-based model that describes the internal dynamics of a system using four matrices: A, B, C, and D

The arrows in the diagram are labeled with the names of the specific functions provided in the scipy.signal subpackage to convert from one form to another. For instance, to convert a transfer function model into its state-space equivalent, one would use the tf2ss function.

Convert a Transfer Function to Zero–Pole–Gain Form

The scipy.signal submodule can handle transfer functions and state-space representations. The function tf2zpk converts a transfer function representation of a linear filter, given by numerator (b) and denominator (a) polynomial coefficients, into its zero–pole–gain (z, p, k) representation. It returns the zeros (z), poles (p), and gain (k) of the filter. This is useful for analyzing filter characteristics and stability. The function signature is

```
tf2zpk(b, a)
```

Both "b" and "a" are required to be specified when calling the function.

Table 12-60 Parameters for the scipy.signal.tf2zpk function

As shown in Table 12-61, the tf2zpk function returns three values: z, a matrix of zeros of the transfer function; p, a matrix of poles of the transfer function; and k, the system gain as a floating point. These values represent the characteristics of the filter, with the zeros and poles determining its frequency response and the gain representing the overall amplification of the system.

Table 12-61 Return values for the scipy.signal.tf2zpk function

This simple example code defines a transfer function with numerator coefficients b = [1, 2] and denominator coefficients a = [1, 3, 2] and then uses tf2zpk to convert it to zero–pole–gain form, calculating the zeros, poles, and gain, which are then printed out, providing a compact representation of the filter's characteristics:

```
import numpy as np
from scipy import signal

# Define a transfer function
b = [1, 2]
a = [1, 3, 2]

# Convert to zero-pole-gain form
z, p, k = signal.tf2zpk(b, a)

print("Zeros:", z)
print("Poles:", p)
print("Gain:", k)
```

The output is

```
Zeros: [-2.]
Poles: [-2. -1.]
Gain: 1.0
```

Convert a Transfer Function to State-Space Form

To convert a transfer function to state-space form, use tf2ss. It converts numerator and denominator polynomial coefficients into the state-space matrices A, B, C, and D.

```
tf2ss(num, den)
```

Table 12-62 Parameters for the scipy.signal.tf2ss function

The tf2ss function returns four state-space matrices: A, the state matrix; B, the input matrix; C, the output matrix; and D, the feedthrough matrix, all as ndarrays, which together represent the system's dynamics and input–output behavior in state-space form.

Table 12-63 Return values for the scipy.signal.tf2ss function

The example code below defines a transfer function with numerator coefficients num = [1, 2] and denominator coefficients den = [1, 3, 2] and then uses tf2ss to convert it to state-space form, calculating the state-space matrices A, B, C, and D, which are then printed out, representing the system's dynamics in a different mathematical framework:

$$\text{Transfer Function}: \quad H(s) = \frac{Y(s)}{U(s)} = \frac{s+2}{s^2+3s+2}$$

```
import numpy as np
from scipy import signal

# Define a transfer function
num = [1, 2]
den = [1, 3, 2]

# Convert to state-space form
A, B, C, D = signal.tf2ss(num, den)

print("A:", A)
print("B:", B)
print("C:", C)
print("D:", D)
```

Remember that the standard format of the state-space equation is as represented below:

$\dot{x} = Ax + Bu$

y = Cx + Du

The result is the four matrices A, B, C, and D, below:

```
A: [[-3. -2.]
 [ 1.  0.]]
B: [[1.]
 [0.]]
C: [[1. 2.]]
D: [[0.]]
```

The state-space form is

X(t) = [[-3., -2.], [1., 0.]] x(t) + [[1.], [0.]] u(t)

y(t) = [[1., 2.]] x(t) + [[0.]] u(t)

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_13

13. Graphical User Interfaces (GUIs)

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

There are many ways for users to interact with a computer. The two primary methods of interacting with software are the Command-Line Interface (CLI) and the graphical user interface (GUI).

User Interface Fundamentals

Command-Line Interface (CLI)

The CLI was once the main way people interacted with computers. It is a text-based interface where users type commands to instruct the software to perform specific tasks.

By entering precise commands into a terminal or command prompt, users can perform tasks, manage files, and run programs. This method is particularly powerful and efficient for those familiar with command syntax. It can be faster for certain operations and supports easy automation of repetitive tasks through scripting.

Common examples of CLIs include command prompt in Windows and the terminal in macOS and Linux. System administrators and other technical users often prefer the CLI because it offers greater power, flexibility, and control.

```
--- Engineering CLI Calculator ---
Select a calculation to perform:
1. Ohm's Law Calculator (V = IR)
2. Mechanical Stress Calculator (σ = F/A)
3. Beam Deflection Calculator (Simply Supported, Center Load)
4. Exit
-----------------------------------
Enter your choice (1-4): 1

--- Ohm's Law Calculator ---
Which variable do you want to calculate?
1. Voltage (V)
2. Current (I)
3. Resistance (R)
Enter your choice (1-3): 1
Enter current in Amperes (A): 3
Enter resistance in Ohms (Ω): 2

Result: Voltage (V) = 6.0000 Volts

Press Enter to return to the main menu...
```

Graphical User Interface (GUI)

The GUI is the most common way people interact with software today. It relies on visual elements such as icons, windows, menus, and buttons to facilitate interaction. Users navigate using a mouse, touchpad, or

touchscreen, often performing tasks by clicking, tapping, or dragging and dropping.

GUIs are designed to be intuitive and user-friendly, making them accessible even to non-technical users. They are used in nearly all modern applications, including web browsers, word processors, and photo editors.

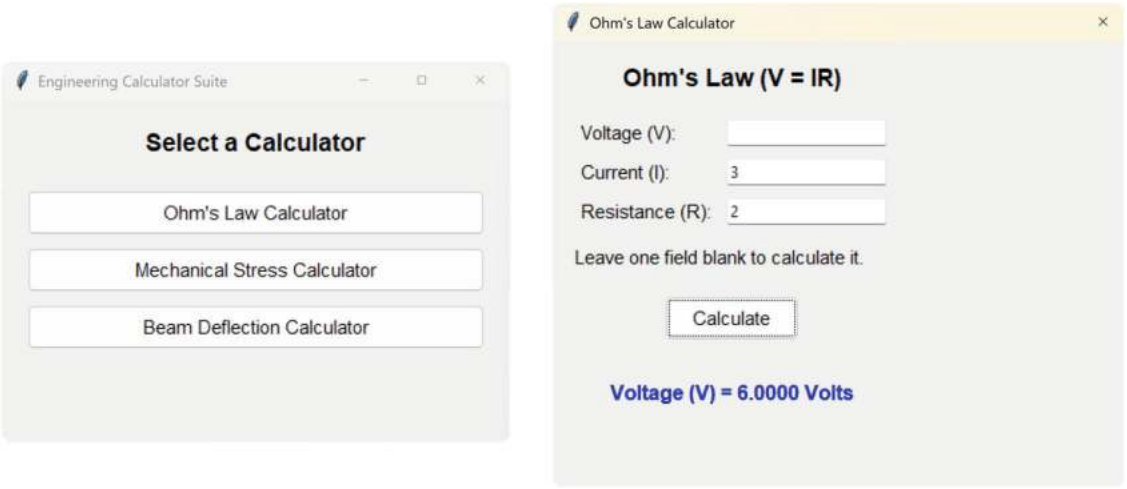


Figure 13-1 Engineering calculator interface with calculator selection and an Ohm's law calculation

With the rapid advancement of computing, especially the rise of complex websites such as social networks and online banking platforms, and the significant technological progress in smartphones and tablets, there has been a growing need for interactive software that is intuitive and easy to use, even for non-technical users. As a result, the development of efficient graphical user interfaces has become essential to ensure usability, accessibility, and a comfortable user experience.

The Python GUI Ecosystem

Comparison of Python Libraries

The following table compares several Python graphical user interface (GUI) libraries, highlighting their descriptions and mobile device compatibility. It lists popular libraries such as Tkinter, Kivy, and PyQt, indicating which ones support mobile development on Android and iOS. Some libraries, such as Kivy, Flet, Toga, PySide, and PyQt, offer cross-platform compatibility, including desktop support for Windows, Linux, and macOS operating systems, as well as mobile devices, while others, such as Tkinter and wxPython, are limited to desktop applications. This table helps developers choose the best GUI library for their projects based on their needs and target platforms.

Table 13-1 Comparison of Python GUI libraries and mobile-platform support

Scope note: This table focuses on general-purpose GUI toolkits. Toga already represents the BeeWare ecosystem, while Briefcase is primarily a packaging tool. Web-first frameworks such as NiceGUI, Streamlit, Dash, and Panel, as well as the iOS development environment Pythonista, serve different purposes and are not listed as equivalent desktop/mobile GUI toolkits.

Why Choose Kivy for Engineering?

In this chapter, we will use the Kivy framework due to its maturity, strong community support, and multiplatform compatibility including native support for mobile devices.

While other libraries have gained popularity in recent years, such as Flet, which is based on the popular Flutter framework, Kivy remains a solid choice. Flet also allows the creation of cross-platform applications for desktop and mobile and is even used by the author in robotics and automation projects. However, Kivy

stands out for its long-standing development, extensive documentation, and active user community. It also offers many built-in features and capabilities that are not commonly found in other frameworks, making it a reliable and powerful tool for building interactive applications.

Kivy is a free and open source Python library that allows developers to create multi-touch applications that can run seamlessly across different platforms, including desktop and mobile devices. With Kivy, developers can write code once and deploy it on various operating systems, including macOS, Linux, Windows, iOS, and Android. It is designed to be business-friendly and flexible, with a permissive MIT license that allows for use in both open source and commercial projects. It is still possible to use an aesthetic variation of Kivy, called KivyMD, which is a collection of Material Design widgets for Kivy.

Kivy's architecture is modular and feature-rich, with a graphics engine that leverages GPU acceleration for high-performance graphics. The library also includes a wide range of widgets and layouts, making it easy to build complex user interfaces.

One of Kivy's standout features is the Kv language, a declarative language that allows developers to separate their UI design from application logic, resulting in cleaner and more maintainable code.

Getting Started with Kivy

Based on the Kivy documentation, there are various options to install Kivy using pip.

Installation and Modules

It is possible to install Kivy with specific sets of dependencies using the following selectors:

kivy[base]: This installs the essential dependencies required for Kivy to run, excluding audio and video support.

```
python -m pip install "kivy[base]"
```

kivy[media]: This installs only the dependencies needed for audio and video playback.

```
python -m pip install "kivy[media]"
```

kivy[full]: This option installs all the typical dependencies needed for Kivy, which include both the base and media dependencies.

```
python -m pip install "kivy[full]"
```

kivy[dev]: This adds dependencies used for development tasks such as running tests and generating documentation. For a regular pip installation, it can be combined with the full dependency set.

```
python -m pip install "kivy[dev,full]"
```

To include Kivy examples with your installation, you can add kivy_examples.

```
python -m pip install "kivy[base]" kivy_examples
```

On Windows, Kivy provides users with the flexibility to choose specific backends for core functionalities through optional installations. These are installed as separate dependency wheels and include options for audio and video support, window, image, audio backend, and a choice between alternative OpenGL graphics backends.

For installation details and platform-specific updates, consult the official documentation: https://kivy.org/doc/stable/gettingstarted/installation.html#using-pip.

Kivy offers a vast array of widgets for crafting intuitive graphical user interfaces, which are categorized into distinct groups. At the foundation are basic widgets, including labels, buttons, text inputs, checkboxes, and more, which serve as the building blocks for user interaction. Layout widgets provide structural organization, managing the positioning and sizing of other widgets within an app. More advanced widgets

combine multiple elements to deliver sophisticated user experiences. Examples include dropdown menus, file choosers, popup dialogs, tabbed interfaces, and virtual keyboards.

Kivy's behavior widgets influence the behavior of their child elements without being visually represented themselves, enabling features like touch-based transformations and graphical effects.

The framework also includes specialized widgets for lists, such as single-line and multi-line list items, often used in conjunction with RecycleView.

The table below provides a comprehensive overview of Kivy's user interface building blocks, organizing various widgets and layouts into distinct categories. The User Experience (UX) Widgets section includes fundamental UI elements, while the Layouts section covers container components used to arrange these widgets. The table also highlights more advanced UX widgets, along with behavior widgets such as Scatter and StencilView. Additionally, it features the ScreenManager, which allows for managing multiple screens and transitions. Together, these components enable developers to build complex, interactive, and user-friendly interfaces using Kivy.

Table 13-2 Representative Kivy widgets and layout managers

This catalog is representative rather than exhaustive. Other Kivy widgets include ScrollView, Carousel, Accordion, ActionBar, TreeView, Camera, ColorPicker, CodeInput, Splitter, Settings, ModalView, and AsyncImage.

Architecture: Separating Logic (.py) and Design (.kv)

A Kivy application's code is typically split into two main parts to keep things organized. The first part is the Python file (.py), which controls the application's behavior and logic. This file defines what the application actually does, such as handling user input, processing data, and running the functions that are triggered when a button is pressed. This component represents the functional core of the application.

The second part is the Kivy language file (.kv), which is dedicated entirely to describing the application's visual appearance. Instead of writing code for every visual detail, you use this file to lay out all the user interface elements like buttons, labels, and text boxes and to define their look, size, and position on the screen. This method allows you to easily change the app's design without altering the functional code. As we will see in the example later, the codes contained in the Kivy language file (.kv) can also be located directly in the Python file (.py), but they may not be suitable for large-scale editing projects.

Kivy automatically links a Python script with its corresponding .kv file, allowing them to work together seamlessly as a complete application. This connection is established through a simple naming convention. When you run a Kivy application, it looks for a .kv file in the same directory that matches the name of your main App class converted to lowercase and with the "App" suffix removed. For example, if your main application class in Python is named EngApp, Kivy will automatically search for and load a file named eng.kv.

Once Kivy finds the matching .kv file, it parses its contents to build the user interface. There are two main things it looks for:

Rules: Any rules you've defined, like <MyWidget>:, are stored and automatically applied to every instance of that widget class created in your application. This allows you to define the look and feel for your custom widgets.

Root Widget: If your .kv file defines a root widget (a widget class name without the <> brackets), Kivy will instantiate it and set it as the main widget for your application. This root widget becomes the base of your application's entire widget tree.

This process allows Kivy to seamlessly merge the logic you've written in your Python file with the UI structure and design you've laid out in your .kv file, creating a single, complete application without you needing to manually load the file yourself.

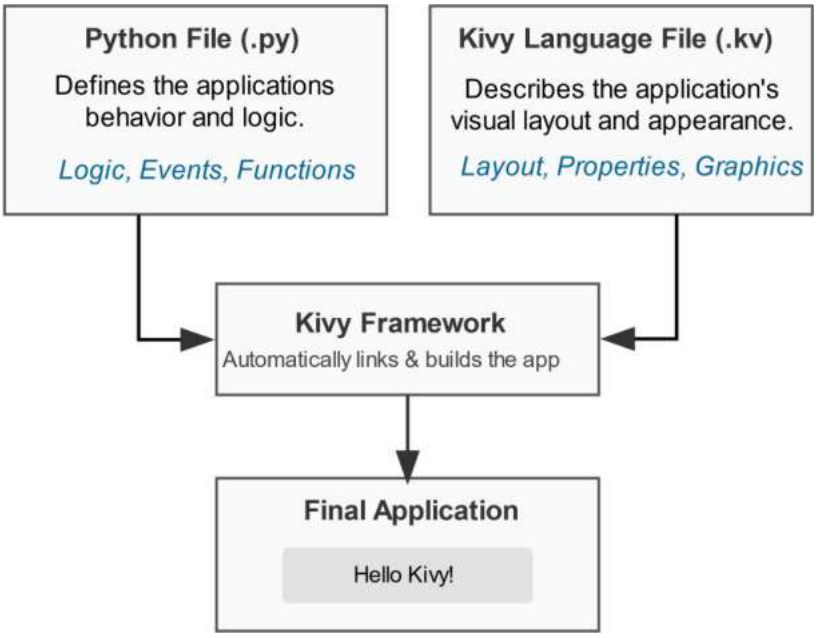


Figure 13-2 Separation of application logic in Python and user-interface design in a Kv file

First Application: "Hello engineer!"

This code is a simple Kivy application that creates a window with a centered box layout containing a button and a label. The UI is defined in a string using Kivy's language, which includes a FloatLayout as the root widget and a BoxLayout as its child. The BoxLayout is centered and has a fixed size, with a button and label arranged vertically inside it.

The application is defined in a Python class EngApp that inherits from Kivy's App class. The build method of this class loads the kv code string and returns the resulting widget, which becomes the root of the application. When the script is run, an instance of EngApp is created and its run method is called, launching the application and displaying the UI.

```
from kivy.app import App
from kivy.lang import Builder
# We define the UI and widgets in one string
kv_code = """
FloatLayout:
    BoxLayout:
        orientation: 'vertical'
        size_hint: None, None
        size: "300dp", "200dp"
        pos_hint: {'center_x': .5, 'center_y': .5}
        padding: "10dp"
        spacing: "10dp"

        Button:
            text: 'Button'
            font_size: '20sp'
        Label:
            text: 'Hello engineer!'
            font_size: '20sp'
"""
class EngApp(App):
```

```
    def build(self):
        return Builder.load_string(kv_code)

if __name__ == '__main__':
    EngApp().run()
```

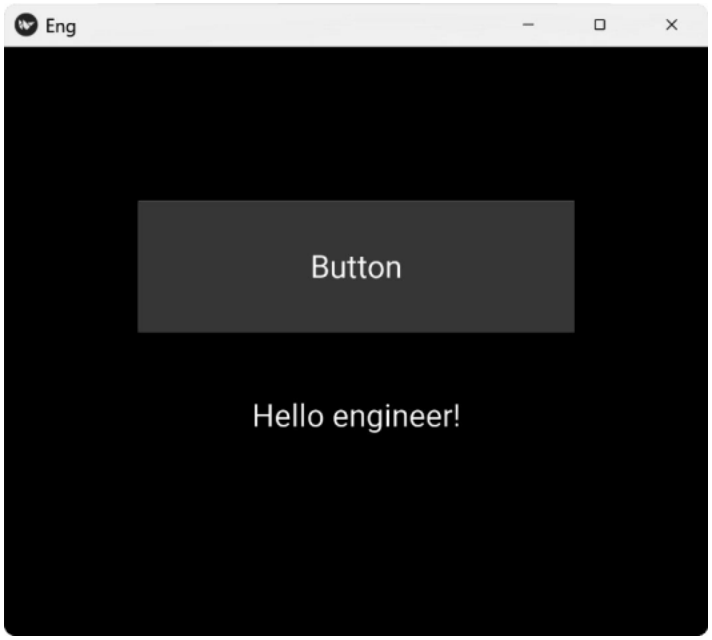


Figure 13-3 First Kivy application displaying a button and the message “Hello engineer!”

Table

Layouts and Positioning Systems

Understanding Layouts

Layouts are essential components for designing user interfaces that are both visually appealing and adaptable to different screen sizes. They automatically handle the positioning and sizing of widgets, making it easier to create responsive and scalable interfaces. By using properties like size_hint and pos_hint, layouts enable proportional and relative sizing and positioning of their child widgets.

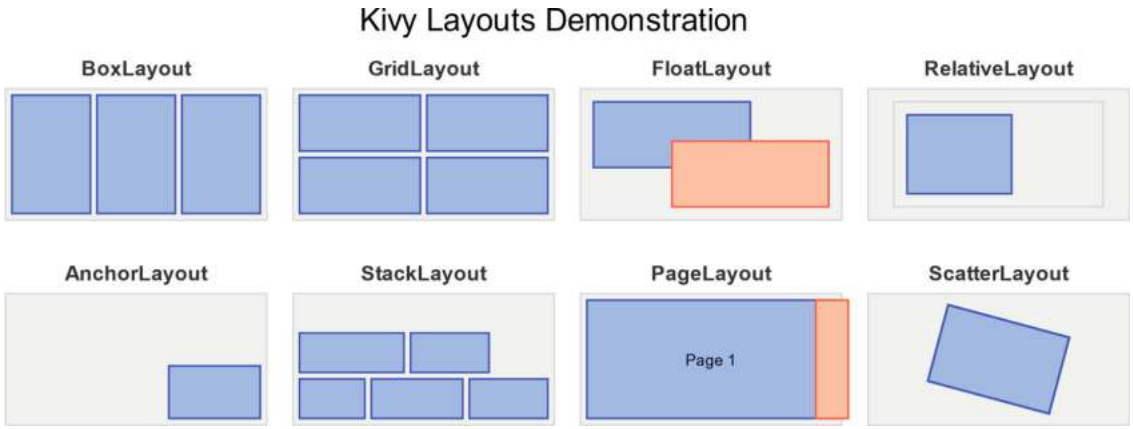


Figure 13-4 Visual overview of eight Kivy layout managers

Table 13-3 Kivy layout managers, arrangement styles, and typical use cases

Common Layout Managers

BoxLayout

This layout arranges widgets in a single row. The buttons are given a fixed size by setting their size_hint to None.

Table

GridLayout

This layout organizes widgets into a grid.

Table

FloatLayout

This layout allows placing widgets with absolute or relative coordinates.

Table

RelativeLayout

This layout positions its children relative to its own position.

Table

AnchorLayout

This layout "anchors" its children to a border or the center.

Table

Advanced Layouts

StackLayout

This layout stacks children next to one another, wrapping when the row is full.

Table

PageLayout

This layout creates a multi-page view. The buttons fill each page by default, so a fixed size is not applied.

Table

ScatterLayout

This layout allows its children to be freely manipulated.

Table

In the Kivy framework, widget positioning is achieved through a combination of absolute positioning and layout-specific "hint" properties, allowing for precise manual control or automatic management by the parent layout. In other words, widget positioning can use pixels or the range 0.0–1.0, depending on the

context. When defined directly on a widget, they use pixels. And when used as keys in the pos_hint dictionary, they use the proportional range 0.0–1.0.

Absolute Positioning (Core Properties)

Every widget instance has access to these properties, which set its position in pixels relative to its parent's coordinates, with the origin (0, 0) at the bottom-left corner. They're best utilized in layouts like FloatLayout and RelativeLayout that don't override these values. These properties are ideal for situations requiring exact, pixel-level control over a widget's position, typically in FloatLayout or when manually managing widget positions.

Table 13-4 Kivy widget position properties and coordinate values

This example code creates an application with a single, centered button using a more concise method. It leverages Kivy's kv language to define the user interface, where it creates a Button with a fixed size of 150 × 50 pixels inside a FloatLayout. The key line, center: self.parent.center, automatically binds the button's center position to the center of its parent layout, ensuring the button remains perfectly centered even if the window is resized, without needing extra Python code for event handling:

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.lang import Builder

Builder.load_string("""
<MyFloatLayout>:
    Button:
        text: 'Center'
        size_hint: None, None
        size: 150, 50
        center: self.parent.center
""")
class MyFloatLayout(FloatLayout):
    pass

class EngApp(App):
    def build(self):
        return MyFloatLayout()

if __name__ == '__main__':
    EngApp().run()
```

Relative Positioning (Hints)

Layouts use "hint" properties to automatically calculate and assign positions to their children. This is the preferred method for creating responsive and dynamic UIs.

The pos_hint property is a dictionary that enables you to position a widget in proportion to the size of its parent. This layout-driven approach, honored by FloatLayout, BoxLayout, and RelativeLayout, provides a responsive alternative to absolute positioning. While properties like pos, x, and y use exact pixel values, pos_hint uses fractional values between 0.0 and 1.0. You simply specify keys such as 'x', 'y', 'center_x', or 'top' in the dictionary to automatically align the widget as the parent's size changes.

Table 13-5 Relative positioning keys supported by pos_hint

Before reading the following code example, see the explanations for the details of using size_hint.

This line below controls the size of a widget relative to its parent layout:

```
size_hint=(0.2, 0.1)
```

It's a way to make your user interface responsive to different screen sizes. The property size_hint is a tuple (width_hint, height_hint).

- 0.2 means the widget's width will be 20% of the parent's width.
- 0.1 means the widget's height will be 10% of the parent's height.

The values for width_hint and height_hint are proportions, so they are typically set between 0.0 (0%) and 1.0 (100%). Values greater than 1.0 can be used, but this would make the widget larger than its parent, which is usually not desired. To disable the hint for a specific dimension, you can set its value to None (e.g., size_hint=(0.5, None)). If a dimension's hint is None, Kivy will use the width or height property in pixels instead.

The next line controls the position of a widget within its parent layout, also using relative coordinates:

```
pos_hint={'center_x': 0.5, 'center_y': 0.5}
```

pos_hint is a dictionary that tells the parent layout how to place the widget.

- 'center_x': 0.5 places the center of the widget at the horizontal midpoint (50%) of the parent layout.
- 'center_y': 0.5 places the center of the widget at the vertical midpoint (50%) of the parent layout.

The combined effect is to perfectly center the widget within its parent.

This code creates a simple mobile application with a single button centered on the screen, where the button is sized to 20% of the screen's width and 10% of its height, and despite being labeled "Bottom", it's actually positioned at the center of the screen due to the pos_hint properties set to 0.5 for both center_x and center_y:

```
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.button import Button

class MyFloatLayout(FloatLayout):
    def __init__(self, **kwargs):
        super(MyFloatLayout, self).__init__(**kwargs)

        button = Button(
            text='Bottom',
            size_hint=(0.2, 0.1),
            pos_hint={'center_x': 0.5, 'center_y': 0.5}
        )
        self.add_widget(button)
class EngApp(App):
    def build(self):
        return MyFloatLayout()
if __name__ == '__main__':
    EngApp().run()
```

Layout-Specific Positioning

Different layouts arrange their children automatically based on their own unique properties, often in combination with the children's size_hint.

Table 13-6 Layout-specific properties for positioning Kivy widgets

Kivy Widgets Catalog

Overview of Widgets and Events

The Kivy framework offers a variety of widgets for user interaction, including buttons, checkboxes, progress bars, and sliders. These widgets are documented in two tables. The first table describes the widgets and their main attributes, such as color, size, and position. The second table details the methods and events available for each widget, allowing you to use them during program execution.

Following the tables, several code examples of widgets commonly used in engineering applications will be presented.

For an overview, the following table serves as a comprehensive reference guide, systematically detailing the configurable attributes, or "properties," of various Kivy widgets. It organizes these properties into logical groups, such as those that control text, background appearance, or interactive behavior, making it easy to find and understand how to customize each UI element. For each listed property, a clear description is provided, explaining its specific function, from setting the text on a button to setting the range of a slider or enabling the markup on a label.

Table 13-7 Configurable properties of common Kivy widgets

The next table provides a summary of the events and methods of the main interactive components for various Kivy widgets. Events are triggers that a widget automatically fires in response to user actions or changes in its state, such as on_press for a button or on_enter for a screen, allowing you to bind custom functions to these occurrences. Methods, on the other hand, are functions you can call on a widget instance to command it to perform a specific action, such as programmatically selecting text in a TextInput or closing a Popup. In essence, the table serves as a quick reference for reacting to and controlling the widget's behavior directly.

Table 13-8 Events and methods of common Kivy widgets

Button

In the next example, two buttons are stacked vertically. Their text, size, and color are defined in an embedded Kv string, while ButtonWidget contains the interaction logic. The first button prints messages when pressed and released; the second programmatically triggers those same actions on the first button:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import ObjectProperty

Builder.load_string("""
<ButtonWidget>:
    orientation: 'vertical'
    padding: 20
    spacing: 10
    button1: button1_id

    Button:
        id: button1_id
        text: "Press Me"
        font_size: '20sp'
        background_color: [0, .7, .2, 1]
        background_normal: ''
```

```
        on_press: root.button_pressed(self)
        on_release: root.button_released(self)

    Button:
        text: "Trigger First Button"
        on_press: root.trigger_first_button(self)
""")
class ButtonWidget(BoxLayout):
    """
    This class contains the logic for the button interactions.
    The layout itself is defined in the Kv string above.
    """
    button1 = ObjectProperty(None)

    def button_pressed(self, instance):
        # This function is called by the on_press event in the .kv file
        print(f"'{instance.text}' pressed, state is '{instance.state}'")

    def button_released(self, instance):
        # This function is called by the on_release event in the .kv file
        print(f"'{instance.text}' released, state is '{instance.state}'")

    def trigger_first_button(self, instance):
        # This method demonstrates programmatically triggering a button
press
        print("Triggering the first button's action...")
        self.button1.trigger_action(duration=0.1)

class ButtonExampleApp(App):
    def build(self):
        return ButtonWidget()

if __name__ == '__main__':
    ButtonExampleApp().run()
```



Figure 13-14 Button press and release events and programmatic triggering of a button action

CheckBox

The CheckBox example arranges labels and controls in a two-column grid. It includes one independent checkbox and two grouped checkboxes that behave like radio buttons. Each change calls a Python method that reports the control's group and active state in the console:

```
from kivy.app import App
from kivy.lang import Builder
```

```
from kivy.uix.gridlayout import GridLayout

Builder.load_string("""
<CheckBoxWidget>:
    cols: 2
    padding: 20
    spacing: 10

    Label:
        text: "Standalone CheckBox:"
        halign: 'left'
    CheckBox:
        # The on_active event calls the python method directly
        on_active: root.on_checkbox_active(self, self.active)

    Label:
        text: "Grouped (Radio Buttons):"
        halign: 'left'
    Widget:  # spacer to fill column 2 in this row

    Label:
        text: "Choice 1"
        halign: 'left'
    CheckBox:
        group: 'radio_group'
        on_active: root.on_checkbox_active(self, self.active)

    Label:
        text: "Choice 2"
        halign: 'left'
    CheckBox:
        group: 'radio_group'
        active: True  # Start this one as active
        on_active: root.on_checkbox_active(self, self.active)
""")

class CheckBoxWidget(GridLayout):
    """
    This class now only contains the logic. The layout and widgets
    are defined in the Kv language string above.
    """
    def on_checkbox_active(self, checkbox_instance, value):
        """
        This function is called when a checkbox is activated or deactivated.
        """
        # A more informative print statement
        group = checkbox_instance.group or 'Standalone'
        status = "active" if value else "inactive"
        print(f"Checkbox in group '{group}' is now {status}")

class CheckBoxExampleApp(App):
    def build(self):
        return CheckBoxWidget()
```

```
if __name__ == '__main__':
    CheckBoxExampleApp().run()
```



Figure 13-15 Standalone CheckBox and mutually exclusive grouped CheckBox widgets

Progress Bar

The ProgressBar example combines a bar, a value label, and a start/stop button. Its Kv definition handles the layout, while ProgressBarWidget uses Kivy's Clock to update the value at regular intervals. Pressing the button again cancels the scheduled updates:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.progressbar import ProgressBar
from kivy.uix.button import Button
from kivy.uix.label import Label
from kivy.clock import Clock

class ProgressBarWidget(BoxLayout):
    def __init__(self, **kwargs):
        super(ProgressBarWidget, self).__init__(**kwargs)
        self.orientation = 'vertical'
        self.padding = 20
        self.spacing = 10

        self.progress_bar = ProgressBar(max=1000)
        self.add_widget(self.progress_bar)

        self.label = Label(text="Progress: 0 / 1000")
        self.add_widget(self.label)

        self.button = Button(text="Start Progress")
        self.button.bind(on_press=self.start_progress)
        self.add_widget(self.button)

        self.event = None

    def start_progress(self, instance):
        if self.event:
            # If already running, stop it
            self.event.cancel()
            self.event = None
            self.button.text = "Start Progress"
        else:
            # Reset and start
```

```
            self.progress_bar.value = 0
            self.button.text = "Stop Progress"
            self.event = Clock.schedule_interval(self.update_progress, 0.1)

    def update_progress(self, dt):
        if self.progress_bar.value < self.progress_bar.max:
            self.progress_bar.value += 25
            self.label.text = (f"Progress: {self.progress_bar.value} /
{self.progress_bar.max}\n"
                               f"Normalized:
{self.progress_bar.value_normalized:.2f}")
        else:
            self.event.cancel()
            self.event = None
            self.button.text = "Start Progress"

class ProgressBarExampleApp(App):
    def build(self):
        return ProgressBarWidget()

if __name__ == '__main__':
    ProgressBarExampleApp().run()
```

Pay attention to the dt parameter in the definition of the update_progress(self, dt) method. It stands for *delta time*. When using Clock.schedule_interval to schedule a repetitive task, Kivy automatically passes the exact time elapsed since the last call (in seconds) to the scheduled function. Even if your function's internal logic does not require this value for calculations, it is mandatory to declare this argument in the method definition. If you forget to include dt, Python will raise a TypeError indicating that the function received an unexpected argument, as Kivy will attempt to pass it regardless.

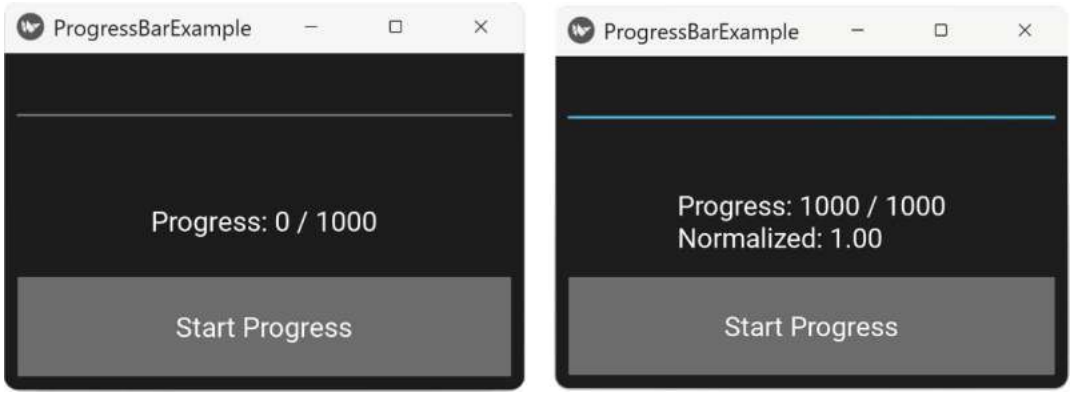


Figure 13-16 ProgressBar example before execution (left) and after completion (right)

Label

Four Label configurations are arranged vertically in a BoxLayout—plain text, styled magenta text, markup with a clickable reference, and wrapped text with controlled horizontal and vertical alignment:

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.utils import escape_markup
```

```
class LabelWidget(BoxLayout):
    def __init__(self, **kwargs):
        super(LabelWidget, self).__init__(**kwargs)
        self.orientation = 'vertical'
        self.padding = 10

        # --- Basic Label ---
        self.add_widget(Label(text="Basic Label"))

        # --- Styled Label ---
        self.add_widget(Label(
            text="Styled Label",
            font_size='24sp',
            bold=True,
            italic=True,
            color=[1, 0, 1, 1] # Magenta

        ))

        # --- Markup Label ---
        markup_text = (
            "This label uses [b]markup[/b] for styling. "
            "You can have [color=ff3333]red text[/color], "
            "and even [ref=my_link]clickable links[/ref]. "
            "Don't forget to escape special characters like "
            f"[b]{escape_markup('[ and ]')}[/b]."
        )
        markup_label = Label(text=markup_text, markup=True)
        markup_label.bind(on_ref_press=self.link_clicked)
        self.add_widget(markup_label)

        # --- Text Alignment and Wrapping ---
        wrapping_text = "This long text will wrap inside the label's
bounding box. The alignment is set to center."
        wrapping_label = Label(
            text=wrapping_text,
            text_size=(self.width - 40, None), # Constrain width
            halign='center',
            valign='middle'
        )
        # Bind text_size to width changes to re-wrap on resize
        self.bind(width=lambda instance, value: setattr(wrapping_label,
'text_size', (value - 40, None)))
        self.add_widget(wrapping_label)

    def link_clicked(self, instance, value):
        print(f"The link '{value}' was pressed!")

class LabelExampleApp(App):
    def build(self):
        return LabelWidget()

if __name__ == '__main__':
```

```
LabelExampleApp().run()
```

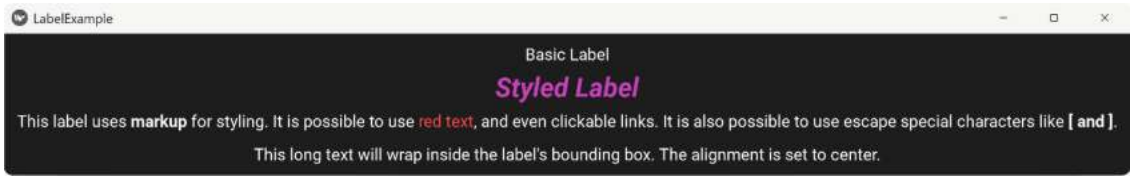


Figure 13-17 Label examples with text styling, markup, a clickable reference, and text wrapping

Slider

Three Slider configurations are defined in one Kv string: a standard horizontal slider, a stepped vertical slider, and a styled slider with a visible value track. Each accompanying label is bound directly to its slider's value:

```
from kivy.app import App
from kivy.lang import Builder

KV_STRING = """
BoxLayout:
    orientation: 'vertical'
    padding: 20
    spacing: 20
    # --- Standard Horizontal Slider ---
    BoxLayout:
        spacing: 10
        Label:
            text: f"Value: {int(h_slider.value)}"
            size_hint_x: 0.3
        Slider:
            id: h_slider
            min: -100
            max: 100
            value: 25
    # --- Vertical Slider with Steps ---
    BoxLayout:
        spacing: 10
        Label:
            text: f"Stepped Value: {int(v_slider.value)}"
            size_hint_x: 0.3
        Slider:
            id: v_slider
            orientation: 'vertical'
            step: 10 # Step by 10
    # --- Styled Slider with Value Track ---
    BoxLayout:
        spacing: 10
        Label:
            text: f"Normalized: {styled_slider.value_normalized:.2f}"
            size_hint_x: 0.3
        Slider:
            id: styled_slider
            value_track: True
```

```
            value_track_color: [1, 0, 0, 1] # Red track color
"""

class SliderExampleApp(App):
    def build(self):
        # The build method now simply loads the layout from the string.
        return Builder.load_string(KV_STRING)

if __name__ == '__main__':
    SliderExampleApp().run()
```

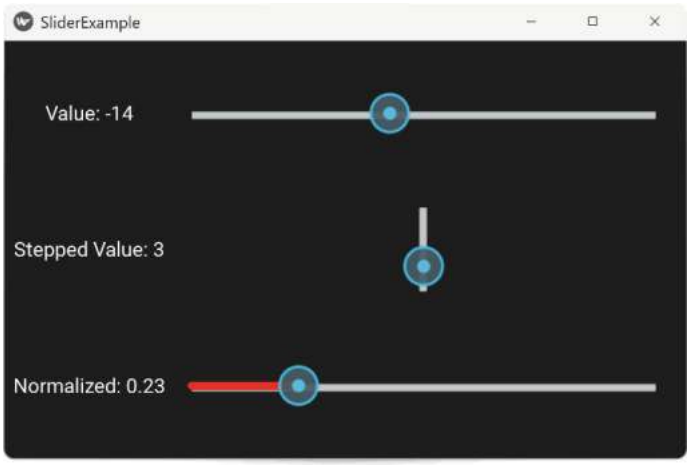


Figure 13-18 Slider examples with horizontal and vertical orientations, discrete steps, and normalized values

Text Input

The TextInput example includes a password field, a read-only text box, and buttons for selection and clipboard operations. Pressing Enter in the password field calls a Python method that prints the value and clears the input; the other controls invoke methods such as select_all(), copy(), and paste() from Kv:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import ObjectProperty

KV = """
<TextInputWidget>:
    orientation: 'vertical'
    padding: '10dp'
    spacing: '10dp'
    selection_input: selection_input_id

    Label:
        text: "Password (Press Enter to Validate)"
        size_hint_y: None
        height: self.texture_size[1]
    TextInput:
        id: password_input_id
        multiline: False
        password: True
```

```
        hint_text: "Enter password"
        on_text_validate: root.on_enter_pressed(self)

    Label:
        text: "Selection & Clipboard"
        size_hint_y: None
        height: self.texture_size[1]
    TextInput:
        id: selection_input_id
        text: "Select this text for clipboard actions."
        readonly: True

    BoxLayout:
        size_hint_y: None
        height: '40dp'
        spacing: '5dp'
        Button:
            text: "Select All"
            on_press: root.selection_input.select_all()
        Button:
            text: "Copy"
            on_press: root.selection_input.copy()
        Button:
            text: "Paste"
            on_press: password_input_id.paste()
"""
Builder.load_string(KV)

class TextInputWidget(BoxLayout):
    """
    This class contains the logic that responds to events from the UI.
    """
    selection_input = ObjectProperty(None)

    def on_enter_pressed(self, instance):
        """Called by the on_text_validate event from the password input."""
        print(f"Enter pressed! Password field contains: '{instance.text}'")
        instance.text = "" # Clear after validation

class TextInputExampleApp(App):
    def build(self):
        # The build method now returns an instance of our root widget
        return TextInputWidget()

if __name__ == '__main__':
    TextInputExampleApp().run()
```

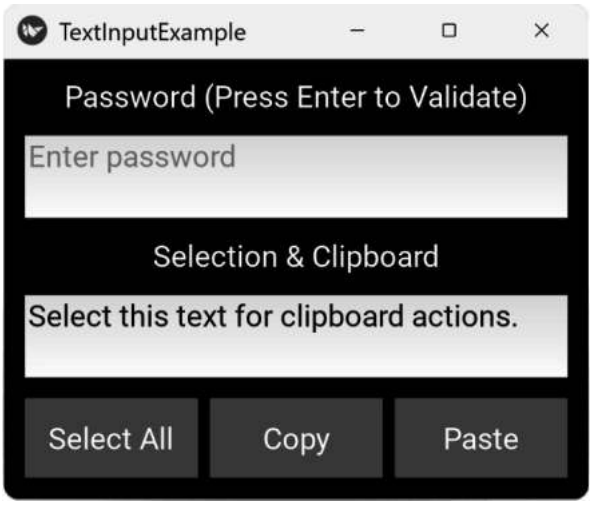


Figure 13-19 TextInput example with password entry, text selection, and clipboard actions

Switch

The Switch example places a toggle beside a Label. Because the label text is bound directly to the active state, it changes between "ON" and "OFF" without an additional Python callback:

```
from kivy.app import App
from kivy.lang import Builder

KV_STRING = """
BoxLayout:
    orientation: 'vertical'
    padding: '20dp'
    spacing: '20dp'
    BoxLayout:
        spacing: '10dp'
        size_hint_y: None
        height: '48dp'
        Label:
            text: "Power Status:"
            size_hint_x: 0.7
        Switch:
            id: main_switch
            active: True # The switch starts in the 'on' state
    Label:
        font_size: '24sp'
        text: "System is ON" if main_switch.active else "System is OFF"
        color: [0, .7, .2, 1] if main_switch.active else [1, 0, 0, 1]
"""
class SwitchExampleApp(App):
    def build(self):
        return Builder.load_string(KV_STRING)

if __name__ == '__main__':
    SwitchExampleApp().run()
```

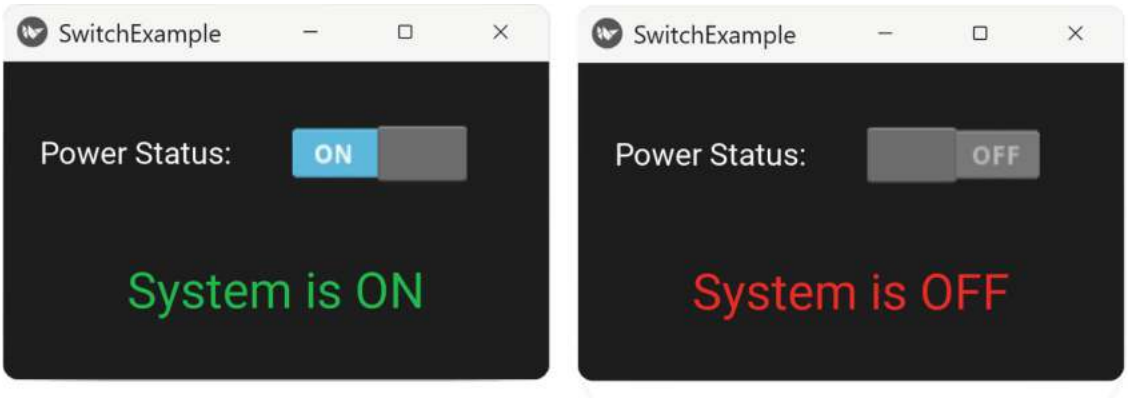


Figure 13-20 Switch example displaying the ON and OFF states and their corresponding status messages

Popup

This example code creates a simple application with a single button that opens a popup window. The main layout, containing the "Open Popup" button, is defined in a Kv language string. When this button is pressed, it calls the open_popup method from the main application class (App). This Python method then programmatically creates a Popup widget, sets its title and content (a simple Label), gives it a fixed size of 400 × 400 pixels, and then displays it on the screen by calling its open() method:

```
from kivy.app import App
from kivy.lang import Builder
from kivy.uix.popup import Popup
from kivy.uix.label import Label

KV_STRING = """
BoxLayout:
    orientation: 'vertical'
    padding: '20dp'
    spacing: '20dp'
    Button:
        text: 'Open Popup'
        on_press: app.open_popup()
"""
class PopupExampleApp(App):
    def build(self):
        return Builder.load_string(KV_STRING)

    def open_popup(self):
        popup = Popup(
            title='My Popup',
            content=Label(text='This is a popup'),
            size_hint=(None, None),
            size=(400, 400)
        )
        popup.open()

if __name__ == '__main__':
    PopupExampleApp().run()
```

In Kivy, Popup is a special type of widget that is typically created dynamically in response to some event, such as a button press. Because of this, it's not usually defined directly in the Kv language layout like other widgets. If a Popup is defined directly in the Kv language layout, it is created as soon as the layout is loaded and becomes immediately visible on the screen. However, popups are usually meant to be displayed temporarily in response to user actions.

See the image below; on the left side we see the home screen containing the button that triggers the popup message, and on the right side we have the popup message that was triggered.

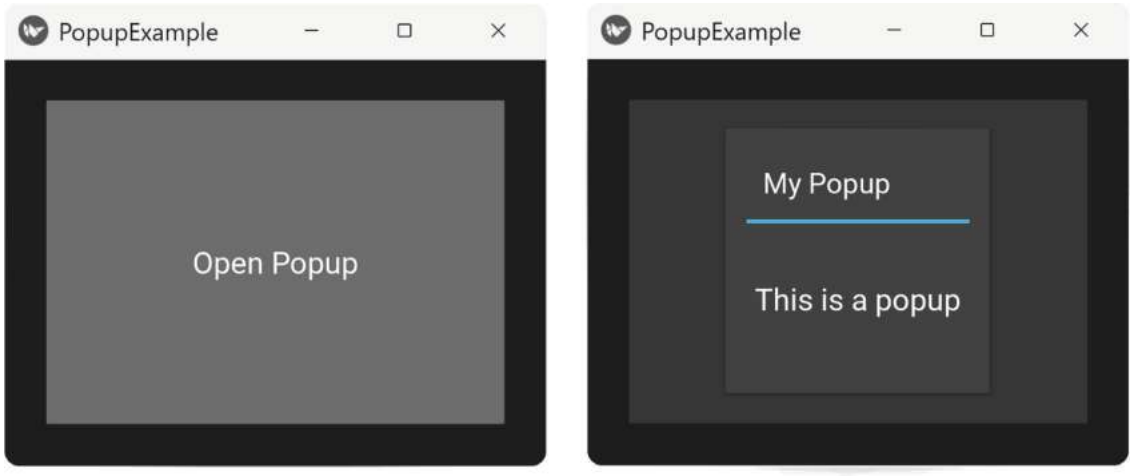


Figure 13-21 Popup example before and after opening a modal dialog

Spinner

This example combines a Spinner with a Label. Selecting one of the three options calls spinner_changed and updates the label; the initial placeholder is ignored:

```
from kivy.app import App
from kivy.lang import Builder

KV_STRING = """
BoxLayout:
    orientation: 'vertical'
    padding: '20dp'
    spacing: '20dp'
    Spinner:
        id: spinner
        text: 'Select an option'
        values: ['Option 1', 'Option 2', 'Option 3']
        on_text: app.spinner_changed(spinner.text)
    Label:
        id: label
        text: 'You selected: '
"""
class SpinnerExampleApp(App):
    def build(self):
        return Builder.load_string(KV_STRING)

    def spinner_changed(self, text):
        if text != 'Select an option':
            self.root.ids.label.text = 'You selected: ' + text
```

```
if __name__ == '__main__':
    SpinnerExampleApp().run()
```



Figure 13-22 Spinner example showing the initial prompt, available options, and selected value

Image

This example displays a local image while preserving its aspect ratio. The script directory is registered as a Kivy resource path, so the file Kivy_eng.png can be found on Windows, macOS, or Linux without a machine-specific absolute path. The fit_mode property keeps the image contained within the available space without distortion.

The most commonly used image formats are PNG and JPEG. However, Kivy supports several other image formats for use with the image widget, including GIF, BMP, TGA, and PNM.

Kivy uses the SDL_image library to load images, which supports these formats:

```
import os
from kivy.app import App
from kivy.lang import Builder
from kivy.resources import resource_add_path

HERE = os.path.dirname(os.path.abspath(__file__))
resource_add_path(HERE)

KV_STRING = """
BoxLayout:
    orientation: 'vertical'
    padding: '20dp'
    spacing: '20dp'
    Image:
        source: 'Kivy_eng.png'
        fit_mode: 'contain'
"""

class ImageExampleApp(App):
    def build(self):
        return Builder.load_string(KV_STRING)

if __name__ == '__main__':
    ImageExampleApp().run()
```

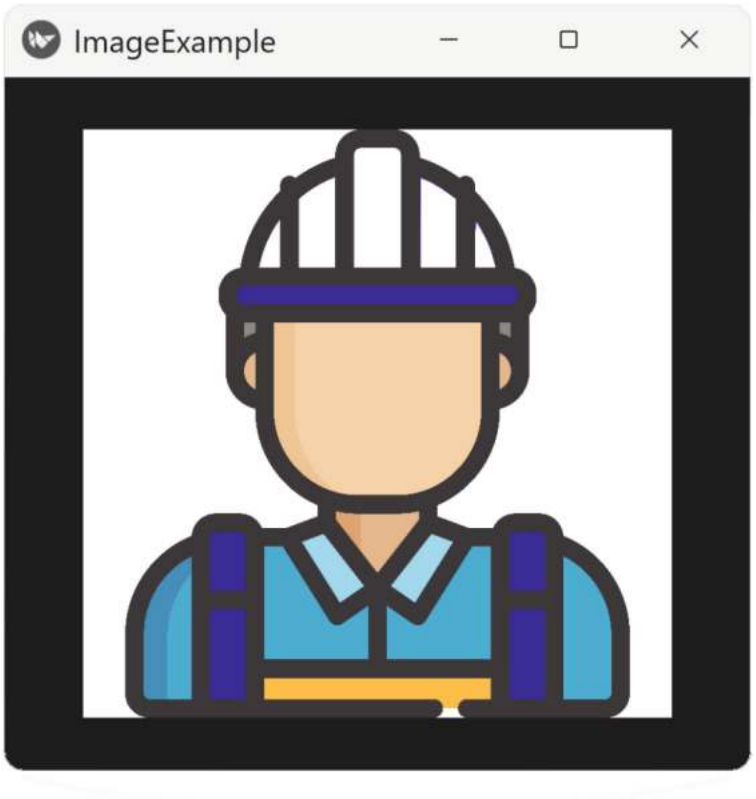


Figure 13-23 Display of a local image using the Image widget

The next variation loads an image from a web address with AsyncImage. Loading occurs asynchronously, so the interface remains responsive while the file is downloaded. An active internet connection is required, and fit_mode preserves the image proportions within the window:

```
from kivy.app import App
from kivy.lang import Builder

KV_STRING = """
BoxLayout:
    orientation: 'vertical'
    padding: '20dp'
    spacing: '20dp'
    AsyncImage:
        source: 'https://pix4free.org/assets/library/2021-05-
25/originals/engineering.jpg'
        fit_mode: 'contain'
"""

class AsyncImageExampleApp(App):
    def build(self):
        return Builder.load_string(KV_STRING)

if __name__ == '__main__':
    AsyncImageExampleApp().run()
```



Figure 13-24 Display of an image loaded from a URL using AsyncImage

This example creates a button-controlled slideshow. The image files are resolved relative to the script directory, avoiding operating system–specific absolute paths. A NumericProperty tracks the current item, and the modulo operator returns to the first image after the last one:

```
import os
from kivy.app import App
from kivy.lang import Builder
from kivy.properties import NumericProperty, ListProperty

HERE = os.path.dirname(os.path.abspath(__file__))
IMAGE_PATHS = [
    os.path.join(HERE, 'Bus.png').replace('\\', '/'),
    os.path.join(HERE, 'Car.png').replace('\\', '/'),
    os.path.join(HERE, 'motorcycle.png').replace('\\', '/')
]

KV_STRING = """
BoxLayout:
    orientation: 'vertical'
    padding: '20dp'
    spacing: '20dp'
    Image:
        id: image_display
        source: app.image_sources[app.current_image_index]
        fit_mode: 'contain'
    Button:
        text: 'Switch Image'
        size_hint_y: None
        height: '48dp'
        on_release: app.switch_image()
"""
```

```
class ImageSwitchApp(App):
    image_sources = ListProperty(IMAGE_PATHS)
    current_image_index = NumericProperty(0)

    def build(self):
        return Builder.load_string(KV_STRING)

    def switch_image(self):
        self.current_image_index = (self.current_image_index + 1) %
len(self.image_sources)

if __name__ == '__main__':
    ImageSwitchApp().run()
```



Figure 13-25 Cycling through three local images using a button

---

Practical Project: Temperature Converter

Project Goal and Architecture

To illustrate how the Kivy framework works, consider this simple temperature conversion application. It creates a graphical interface with an input field for users to enter a temperature value, two buttons to convert the temperature from Celsius to Fahrenheit and vice versa, and a label to display the result. When a button is pressed, the corresponding conversion function is triggered, which retrieves the input value, performs the conversion, and updates the result label with the converted temperature. If the input value is not a valid number, the application displays an error message. The application is structured in two classes: TemperatureConverter defines the GUI layout and conversion logic, while TemperatureConverterApp builds and runs the application.

Complete Application Code

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput

class TemperatureConverter(BoxLayout):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.orientation = 'vertical'    # Input field for temperature value
        self.temperature_input = TextInput(
```

```
            hint_text='Enter temperature value',
            size_hint_y=0.4,  # Adjusted vertical size to give it more space
            # height=80 # Removed this conflicting property to allow
size_hint_y to work
        )
        self.add_widget(self.temperature_input)
        # Buttons for conversion
        buttons_layout = BoxLayout(
            size_hint_y=0.2  # Adjusted vertical size for buttons
            # height=40 # Removed this conflicting property
        )
        celsius_to_fahrenheit_button = Button(text='Celsius to Fahrenheit')
        celsius_to_fahrenheit_button.bind(on_press=self.celsius_to_fahrenheit)
        buttons_layout.add_widget(celsius_to_fahrenheit_button)

        fahrenheit_to_celsius_button = Button(text='Fahrenheit to Celsius')
        fahrenheit_to_celsius_button.bind(on_press=self.fahrenheit_to_celsius)
        buttons_layout.add_widget(fahrenheit_to_celsius_button)
        self.add_widget(buttons_layout)
        # Label to display result
        self.result_label = Label(
            text='',
            size_hint_y=0.4  # Adjusted vertical size for result label
        )
        self.add_widget(self.result_label)

    def celsius_to_fahrenheit(self, instance):
        try:
            temperature = float(self.temperature_input.text)
            result = temperature * 9 / 5 + 32
            self.result_label.text = f'{temperature}°C is equal to {result}°F'
        except ValueError:
            self.result_label.text = 'Invalid temperature value'

    def fahrenheit_to_celsius(self, instance):
        try:
            temperature = float(self.temperature_input.text)
            result = (temperature - 32) * 5 / 9
            self.result_label.text = f'{temperature}°F is equal to {result}°C'
        except ValueError:
            self.result_label.text = 'Invalid temperature value'

class TemperatureConverterApp(App):
    def build(self):
        return TemperatureConverter()
if __name__ == '__main__':
    TemperatureConverterApp().run()
```

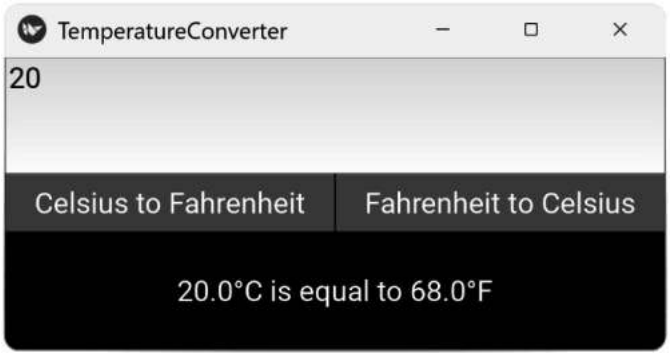


Figure 13-26 Temperature converter displaying the conversion from 20 degrees Celsius to 68 degrees Fahrenheit

Code Walkthrough

1. *Importing Modules*

   The code begins by importing the necessary modules from the Kivy framework. These modules provide the functionality to create a graphical user interface (GUI) and handle user interactions.

```
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.textinput import TextInput
```

   - App is the base class for all Kivy applications. It provides the main entry point for the application and handles the event loop.
   - BoxLayout as previously studied is a layout class that arranges its children in a single row or column.
   - Label is a widget that displays text. It's used to display the result of the temperature conversion.
   - Button is a widget that can be pressed by the user; it is used to trigger the temperature conversion functions.
   - TextInput is used to input the temperature value.

2. *Defining the TemperatureConverter Class*

   The TemperatureConverter class defines the GUI layout and conversion logic for the application.

```
class TemperatureConverter(BoxLayout):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.orientation = 'vertical'
```

   - The __init__ method is a special method that's called when an object is created from the class. It's used to initialize the object's attributes and perform any necessary setup.
   - super().__init__(**kwargs) calls the __init__ method of the parent class (BoxLayout) to perform any necessary initialization.
   - self.orientation = 'vertical' sets the orientation of the layout to vertical, which means that the children will be arranged in a column.

3. *Creating the GUI Layout*

   The GUI layout consists of a text input field, two buttons, and a label to display the result.

```
self.temperature_input = TextInput(
    hint_text='Enter temperature value',
    size_hint_y=0.4)
```

```
self.add_widget(self.temperature_input)
buttons_layout = BoxLayout(size_hint_y=0.2)
celsius_to_fahrenheit_button = Button(text='Celsius to Fahrenheit')
celsius_to_fahrenheit_button.bind(on_press=self.celsius_to_fahrenheit)
buttons_layout.add_widget(celsius_to_fahrenheit_button)

fahrenheit_to_celsius_button = Button(text='Fahrenheit to Celsius')
fahrenheit_to_celsius_button.bind(on_press=self.fahrenheit_to_celsius)
buttons_layout.add_widget(fahrenheit_to_celsius_button)
self.add_widget(buttons_layout)

self.result_label = Label(text='', size_hint_y=0.4)
self.add_widget(self.result_label)
```

- The TextInput widget is created with a hint text that prompts the user to enter a temperature value. The size_hint_y attribute sets the vertical size of the widget to 40% of the parent layout's height.
- The two conversion buttons share a BoxLayout assigned 20% of the parent's height.
- The Button widgets are created with text labels and bound to the celsius_to_fahrenheit and fahrenheit_to_celsius methods, respectively. When a button is pressed, the corresponding method will be called.
- The result label occupies the remaining 40% of the vertical space.

4. *Temperature Conversion Functions*
The celsius_to_fahrenheit and fahrenheit_to_celsius methods perform the temperature conversions.

```
def celsius_to_fahrenheit(self, instance):
    try:
        temperature = float(self.temperature_input.text)
        result = temperature * 9 / 5 + 32         self.result_label.text
= f'{temperature}°C is equal to {result}°F'     except ValueError:
        self.result_label.text = 'Invalid temperature value'

def fahrenheit_to_celsius(self, instance):
    try:
        temperature = float(self.temperature_input.text)
        result = (temperature - 32) * 5 /
9         self.result_label.text = f'{temperature}°F is equal to {result}
°C'     except ValueError:
        self.result_label.text = 'Invalid temperature value'
```

- The try–except block attempts to convert the input text to a float value. If the conversion fails (i.e., the input is not a valid number), a ValueError exception is raised, and the except block sets the result label to an error message.
- The temperature conversion formulas are applied to the input value, and the result is displayed in the result label.

5. *Defining the TemperatureConverterApp Class*
The TemperatureConverterApp class builds and runs the application.

```
class TemperatureConverterApp(App):
    def build(self):
        return TemperatureConverter()
```

- The build method is a special method that's called when the application is started. It returns the root

The build method is a special method that's called when the application is started. It returns the root widget of the application, which in this case is an instance of the TemperatureConverter class.

The application is run using the following code:

```
if __name__ == '__main__':
    TemperatureConverterApp().run()
```

The if __name__ == '__main__': block checks if the script is being run directly (i.e., not being imported as a module by another script). If so, it creates an instance of the TemperatureConverterApp class and calls its run method to start the application.

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_14

14. Introduction to the Pandas Library for Engineering

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

For engineers, the ability to manipulate, analyze, and visualize large volumes of data is crucial. Pandas is a powerful open source Python library for data manipulation and analysis that provides data structures and functions for efficiently handling structured data, including tabular data like spreadsheets and SQL tables.

The objective of this chapter is to explore the Pandas library from a practical engineering perspective, focusing on the complete data analysis workflow. Starting with fundamental data structures, we will progress through data import, inspection, selection, cleaning, statistical analysis, and visualization. Each concept is presented using realistic engineering examples, ensuring that the reader can immediately relate Pandas operations to real-world problems involving measurements, experiments, and industrial data.

Pandas is an indispensable tool for engineers, often described as the “Excel of Python” but with far greater flexibility and automation capabilities. Its utility spans numerous disciplines, from signal processing, where it’s used to filter, clean, and perform spectral analysis on sensor data, to the design of control systems, where it helps analyze system responses and calculate performance metrics.

Furthermore, Pandas is fundamental for building data-driven models and simulations, enabling engineers to prepare complex datasets for machine learning. This capability is critical for applications like predictive maintenance, where real-time sensor data is analyzed to detect anomalies and forecast equipment failures. The library is also crucial in the design and analysis of physical experiments, where it aids in collecting data, performing statistical analysis, and visualizing results to identify key trends and correlations.

For an overview of Pandas’ functionality, see the table below, which summarizes the main Pandas function families. Each family is dedicated to a specific aspect of data analysis, such as importing data, manipulating datasets, or creating visualizations. The table highlights the essential commands for each family, providing a concise overview of Pandas’ capabilities and serving as a quick reference for data analysts and scientists working with the library.

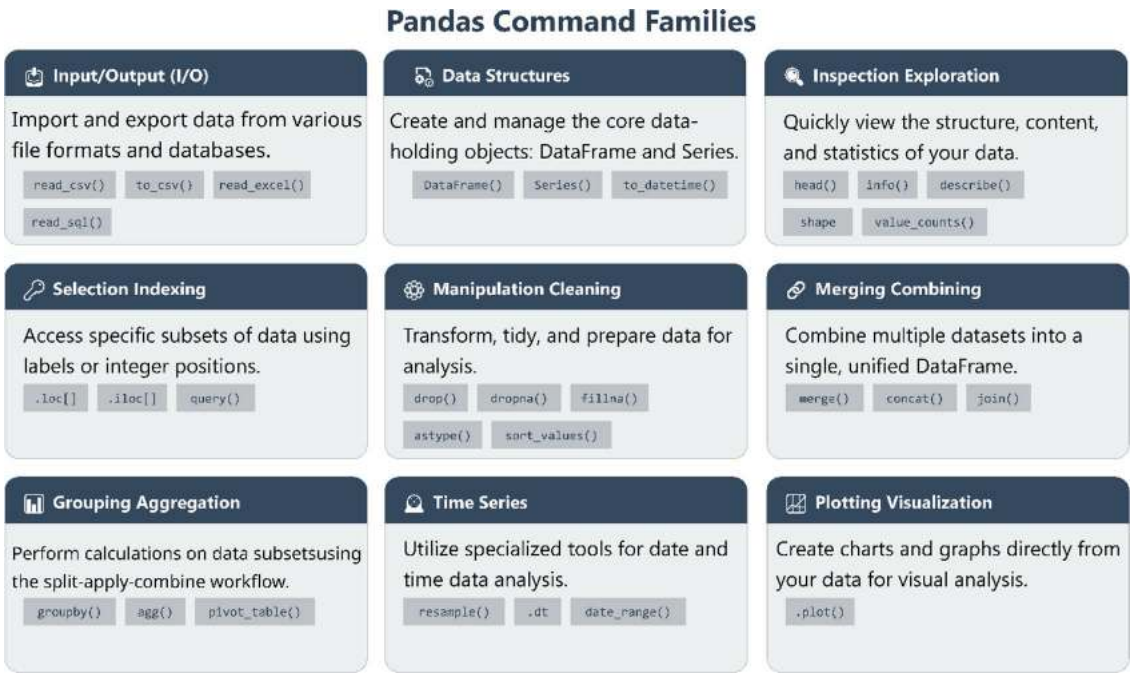


Figure 14-1 Pandas command families

Table

Installing the Pandas Library

Before we begin, we need to install Pandas in your Python environment. Open your terminal (or command prompt) and run the following command:

```
pip install pandas openpyxl matplotlib seaborn
```

- pandas: The core library for data manipulation
- openpyxl: Required to read and write .xlsx (Excel) files
- matplotlib: The core plotting library used later in the chapter
- seaborn: A high-level data visualization library built on top of Matplotlib that integrates with Pandas to create attractive, informative statistical plots with minimal code

After installation, you'll need to import Pandas into your Python scripts. It's common practice to import it with the alias pd for ease of use:

```
import pandas as pd
import numpy as np # We also import numpy, as it's frequently used with
pandas
```

Fundamental Pandas Data Structures

Pandas primarily works with two data structures: Series and DataFrames. Understanding the distinction between a Series and a DataFrame is essential for effective data manipulation.

A Series is a one-dimensional array capable of holding any data type (integers, strings, floats, Python objects, etc.). Think of it as a single column in a table or an enhanced list.

Example: Reading Data from a Temperature Sensor

Imagine a temperature sensor producing seven readings over time. The code creates a Series, prints its index–value pairs, retrieves the third reading (25.9°C), and confirms that the values use the float64 data type.

```
import pandas as pd

# Temperature readings in °C
temperatures = pd.Series([25.5, 26.1, 25.9, 27.0, 26.5, 27.2, 26.8])

print("Temperature Series:")
# Output: Temperature Series:

print(temperatures)
# Output: 0    25.5
#         1    26.1
#         2    25.9
#         3    27.0
#         4    26.5
#         5    27.2
#         6    26.8
#         dtype: float64

# You can access elements by index
print(f"\nTemperature at 3rd reading: {temperatures[2]}°C")
# Output:
#         Temperature at 3rd reading: 25.9°C

# And check the data type
print(f"Data type of the series: {temperatures.dtype}")
# Output: Data type of the series: float64
```

A DataFrame is a two-dimensional tabular data structure with labeled columns and rows. It's the most widely used structure in Pandas and closely resembles an Excel spreadsheet or a database table.

Example: Material Strength Measurement Data
Let's create a DataFrame to store strength test results for different material samples, including material type, applied load (kN), and resulting deformation (mm).

The material test data are stored in a dictionary and converted to df_materials. Printing the DataFrame shows its tabular structure; .shape confirms six rows and four columns, while .columns returns the column names.

```
import pandas as pd
material_data = {
    'Sample': ['A1', 'A2', 'A3', 'B1', 'B2', 'B3'],
    'Material': ['Steel', 'Aluminum', 'Steel', 'Copper', 'Aluminum',
'Copper'],
    'Load_kN': [150.2, 80.5, 155.0, 100.1, 82.3, 105.5],
    'Deformation_mm': [0.15, 0.08, 0.16, 0.10, 0.09, 0.11]
}

df_materials = pd.DataFrame(material_data)

print("\nMaterial Test DataFrame:")
# Output:  Material Test DataFrame:
```

```
print(df_materials)
# Output:   Sample  Material  Load_kN  Deformation_mm
#         0     A1     Steel    150.2            0.15
#         1     A2  Aluminum     80.5            0.08
#         2     A3     Steel    155.0            0.16
#         3     B1    Copper    100.1            0.10
#         4     B2  Aluminum     82.3            0.09
#         5     B3    Copper    105.5            0.11

# Checking the dimensions of the DataFrame (rows, columns)
print(f"\nDimensions of the DataFrame: {df_materials.shape}")
# Output: Dimensions of the DataFrame: (6, 4)

# Viewing the columns
print(f"Column names: {df_materials.columns.tolist()}")
# Output: Column names: ['Sample', 'Material', 'Load_kN', 'Deformation_mm']
```

Reading and Importing Data

In engineering, data is rarely created manually in scripts. It usually comes from CSV files, Excel spreadsheets, databases, etc. Pandas greatly simplifies the reading of these files. Furthermore, engineering decisions are only as reliable as the data they are based on. For this reason, correctly importing data from external sources is a crucial first step in any analysis. Pandas offers robust input/output tools that allow engineers to read data from multiple formats while preserving numerical accuracy, timestamps, and metadata.

Reading CSV Files

As seen in previous chapters, CSV (comma-separated values) is a very common plain-text file format for data exchange.

Example: Air Quality Monitoring Data

Imagine you have an air_quality.csv file with pollutant concentration data.

Let's create an example CSV file first:

```
import pandas as pd
import os

# Code to create the example CSV file
csv_content = """Date,Location,PM2.5,O3,CO
2023-01-01,Central,25.3,45.1,2.0
2023-01-01,North,30.1,50.5,2.5
2023-01-02,Central,22.0,42.0,1.8
2023-01-02,North,28.5,48.0,2.3
2023-01-03,Central,27.8,47.5,2.1
"""
with open('air_quality.csv', 'w') as f:
    f.write(csv_content)

# Reading the CSV file
df_air = pd.read_csv('air_quality.csv')

print("\nAir Quality DataFrame (CSV):")
# Output: Air Quality DataFrame (CSV):
```

```
print(df_air)
# Output:          Date Location  PM2.5    O3   CO
#         0  2023-01-01  Central   25.3  45.1  2.0
#         1  2023-01-01    North   30.1  50.5  2.5
#         2  2023-01-02  Central   22.0  42.0  1.8
#         3  2023-01-02    North   28.5  48.0  2.3
#         4  2023-01-03  Central   27.8  47.5  2.1
# Clean up the created file
os.remove('air_quality.csv')
```

This self-contained example writes sample air quality measurements to air_quality.csv, reads the file with pd.read_csv(), prints the resulting DataFrame, and then removes the temporary file. It reproduces the main steps of a simple CSV workflow without leaving residual files.

Reading Excel Files

Much engineering data is still stored in Excel spreadsheets.

Example: Equipment Energy Consumption Data

Suppose you have an energy_consumption.xlsx file with energy consumption data for various equipment across different shifts, with “Shift_1” and “Shift_2” tabs.

Let’s create an example Excel file first (you can use Excel to create it or the code below with the openpyxl library):

```
import pandas as pd
from openpyxl import Workbook
import os

# Code to create the example Excel file
wb = Workbook()

ws1 = wb.active
ws1.title = "Shift_1"
ws1.append(['Equipment', 'Consumption_kWh', 'Operating_Hours'])
ws1.append(['Motor A', 150.5, 8])
ws1.append(['Pump B', 80.2, 7])
ws1.append(['Compressor C', 200.0, 8])

ws2 = wb.create_sheet("Shift_2")
ws2.append(['Equipment', 'Consumption_kWh', 'Operating_Hours'])
ws2.append(['Motor A', 145.0, 7])
ws2.append(['Pump B', 85.0, 8])
ws2.append(['Generator D', 250.0, 6])

wb.save('energy_consumption.xlsx')

# Reading the Excel file (default tab)
df_energy_shift1 = pd.read_excel('energy_consumption.xlsx',
sheet_name='Shift_1')

print("\nEnergy Consumption DataFrame (Shift 1):")
# Output: Energy Consumption DataFrame (Shift 1):

print(df_energy_shift1)
```

```
# Output:       Equipment  Consumption_kWh  Operating_Hours
#         0       Motor A            150.5                8
#         1        Pump B             80.2                7
#         2  Compressor C            200.0                8

# Reading a specific tab
df_energy_shift2 = pd.read_excel('energy_consumption.xlsx',
sheet_name='Shift_2', dtype={'Consumption_kWh': 'float64'})

print("\nEnergy Consumption DataFrame (Shift 2):")
# Output: Energy Consumption DataFrame (Shift 2):

print(df_energy_shift2)
# Output:        Equipment  Consumption_kWh  Operating_Hours
#         0       Motor A        145.0                7
#         1        Pump B        85.0                 8
#         2   Generator D        250.0                6
# Clean up the created file
os.remove('energy_consumption.xlsx')
```

The workbook contains two worksheets, Shift_1 and Shift_2, with equipment, energy consumption, and operating hour data. After the file is saved as energy_consumption.xlsx, Pandas reads each sheet into a separate DataFrame, and the temporary workbook is removed.

Inspecting and Exploring Data

After loading your data, it is important to inspect it to gain a clear understanding of its structure, content, and overall quality. Pandas provides several built-in methods that help you quickly explore a DataFrame and identify potential issues such as missing values or incorrect data types. Because info() output is version-dependent, the examples in this chapter use Pandas 3.x conventions; memory usage may vary by environment.

- df.head(n): Displays the first n rows of the DataFrame (default is 5)
- df.tail(n): Displays the last n rows of the DataFrame (default is 5)
- df.info(): Provides a concise summary of the DataFrame, including data types, number of non-null values, and memory usage
- df.describe(): Generates descriptive statistics for numerical columns (count, mean, standard deviation, minimum, quartiles, maximum)

Example: Initial Analysis of Concrete Test Data
Consider a DataFrame with concrete strength test results over time:

```
import pandas as pd
import io

concrete_data = {
    'SampleID': [1, 2, 3, 4, 5, 6, 7],
    'Age_days': [7, 7, 28, 28, 90, 90, 180],
    'Cement_kg': [300, 320, 310, 305, 315, 330, 325],
    'Water_liters': [150, 155, 148, 152, 150, 160, 158],
    'Strength_MPa': [25.0, 26.5, 40.1, 39.5, 45.0, 42.8, 48.2]
}
df_concrete = pd.DataFrame(concrete_data)
```

```
print("\nFirst 3 rows of Concrete DataFrame:")
# Output: First 3 rows of Concrete DataFrame:

print(df_concrete.head(3))
# Output:        SampleID  Age_days  Cement_kg  Water_liters  Strength_MPa
#         0         1         7        300           150          25.0
#         1         2         7        320           155          26.5
#         2         3        28        310           148          40.1

print("\nLast 2 rows of Concrete DataFrame:")
# Output: Last 2 rows of Concrete DataFrame:

print(df_concrete.tail(2))
# Output:        SampleID  Age_days  Cement_kg  Water_liters  Strength_MPa
#         5         6        90        330           160          42.8
#         6         7       180        325           158          48.2

print("\nGeneral information about Concrete DataFrame:")
# Output:  General information about Concrete DataFrame:

# The info() method prints directly to stdout, so we capture its output
using a StringIO buffer.
buffer = io.StringIO()
df_concrete.info(buf=buffer)
info_output = buffer.getvalue()
print(info_output)
# Output: <class 'pandas.DataFrame'>
#         RangeIndex: 7 entries, 0 to 6
#         Data columns (total 5 columns):
#          #   Column        Non-Null Count  Dtype
#         ---  ------        --------------  -----
#          0   SampleID      7 non-null      int64
#          1   Age_days      7 non-null      int64
#          2   Cement_kg     7 non-null      int64
#          3   Water_liters  7 non-null      int64
#          4   Strength_MPa  7 non-null      float64
#         dtypes: float64(1), int64(4)
#         memory usage: 412.0 bytes

print("\nDescriptive statistics of Concrete DataFrame:")
# Output: Descriptive statistics of Concrete DataFrame:

print(df_concrete.describe())
# Output:       SampleID   Age_days   Cement_kg  Water_liters  Strength_MPa
#       count    7.000000   7.000000    7.000000      7.000000      7.000000
#       mean     4.000000  61.428571  315.000000    153.285714     38.157143
#       std      2.160247  63.049867   10.801234      4.498677      8.979766
#       min      1.000000   7.000000  300.000000    148.000000     25.000000
#       25%      2.500000  17.500000  307.500000    150.000000     33.000000
#       50%      4.000000  28.000000  315.000000    152.000000     40.100000
#       75%      5.500000  90.000000  322.500000    156.500000     43.900000
#       max      7.000000 180.000000  330.000000    160.000000     48.200000
```

```
buffer = io.StringIO()
```

This line creates an in-memory text stream, which is a file-like object that allows you to read and write strings as if they were files. The io.StringIO() function returns a StringIO object, which is a buffer that can be used to capture output that would normally be written to a file or the console.

```
df_concrete.info(buf=buffer)
```

The info() method of a Pandas DataFrame is used to print a concise summary of the DataFrame, including the index dtype and column dtypes, non-nullable values, and memory usage. By default, info() prints its output directly to the console (stdout).

In this case, we're passing the buffer object as the buf argument to info(), which tells Pandas to write the output to the buffer instead of the console.

```
info_output = buffer.getvalue()
```

After info() has written its output to the buffer, we use the getvalue() method to retrieve the contents of the buffer as a string. This allows us to capture the output of info() as a string, which we can then print or use in our code as needed.

By using a StringIO buffer, we're able to capture the output of info() and store it in a variable, rather than having it printed directly to the console. This can be useful if we want to further process or display the information in a specific way.

Selecting Data in a DataFrame

One of the most common and essential operations in data analysis is the selection of specific data subsets, which enables targeted analysis, improves computational efficiency, and supports more accurate interpretation of results.

Column Selection (Series vs. DataFrame)

When you select a single column from a DataFrame, Pandas returns a Series (a one-dimensional object). When you select multiple columns, Pandas returns another DataFrame (a two-dimensional table). This distinction matters because many operations behave differently depending on whether the result is a Series or a DataFrame.

Example: Selecting Process Parameters from an Industrial Plant

The process data are stored in df_process. Selecting df_process['Temperature_C'] returns a Series, whereas df_process[['Pressure_MPa', 'Flow_Lps']] returns a DataFrame. The final lines also compare label-based selection with .loc[] and position-based selection with .iloc[].

```
import pandas as pd

process_data = {
    'Time_min': [0, 10, 20, 30, 40, 50, 60],
    'Temperature_C': [80.5, 82.1, 81.0, 83.5, 84.0, 83.2, 85.0],
    'Pressure_MPa': [1.2, 1.3, 1.25, 1.35, 1.38, 1.30, 1.40],
    'Flow_Lps': [10.1, 10.5, 9.8, 10.3, 10.8, 10.0, 11.0]
}
df_process = pd.DataFrame(process_data)

# Select a single column (results in a Series)
temperature = df_process['Temperature_C']
print("\n'Temperature_C' Column:")
```

```
# Output:'Temperature_C' Column:

print(temperature)# Output:
#         0    80.5
#         1    82.1
#         2    81.0
#         3    83.5
#         4    84.0
#         5    83.2
#         6    85.0
# Name: Temperature_C, dtype: float64

# Select multiple columns (results in a DataFrame)
pressure_flow = df_process[['Pressure_MPa', 'Flow_Lps']]
print("\n'Pressure_MPa' and 'Flow_Lps' Columns:")
# Output: 'Pressure_MPa' and 'Flow_Lps' Columns:
print(pressure_flow)# Output:
#            Pressure_MPa  Flow_Lps
#         0          1.20      10.1
#         1          1.30      10.5
#         2          1.25       9.8
#         3          1.35      10.3
#         4          1.38      10.8
#         5          1.30      10.0
#         6          1.40      11.0
# Label-based selection with .loc[]
print(df_process.loc[2, ['Temperature_C', 'Pressure_MPa']])
# Position-based selection with .iloc[]
print(df_process.iloc[2, [1, 2]])
```

Row Filtering with Logical Conditions

Filtering rows allows you to select observations that satisfy one or more conditions, which is essential for tasks such as quality control, outlier removal, and selecting test cases. Row filtering is typically performed using boolean masks (expressions that evaluate to True or False).

You can combine conditions using logical operators such as AND (&) and OR (|). When combining conditions, wrap each comparison in parentheses to avoid ambiguity.

Example: Filtering Concrete Samples by Strength and Age

Boolean masks isolate concrete samples of interest. The first filter selects strengths above 40 MPa; the second combines strength and age with the AND operator (&); and the third uses OR (|) to keep samples cured for either 7 or 90 days.

```
import pandas as pd
# Re-creating the DataFrame for context
concrete_data = {
    'SampleID': [1, 2, 3, 4, 5, 6, 7],
    'Age_days': [7, 7, 28, 28, 90, 90, 180],
    'Cement_kg': [300, 320, 310, 305, 315, 330, 325],
    'Water_liters': [150, 155, 148, 152, 150, 160, 158],
    'Strength_MPa': [25.0, 26.5, 40.1, 39.5, 45.0, 42.8, 48.2]
}
df_concrete = pd.DataFrame(concrete_data)
```

```
# Filter samples with strength greater than 40 MPa
high_strength_concrete = df_concrete[df_concrete['Strength_MPa'] > 40.0]
print("\nConcrete Samples with Strength > 40 MPa:")
# Output: Concrete Samples with Strength > 40 MPa:

print(high_strength_concrete)
# Output:
#            SampleID  Age_days  Cement_kg  Water_liters  Strength_MPa
#         2         3        28        310           148          40.1
#         4         5        90        315           150          45.0
#         5         6        90        330           160          42.8
#         6         7       180        325           158          48.2

# Filter concrete samples with strength above 30 MPa AND 28 days old
condition1 = df_concrete['Strength_MPa'] > 30.0
condition2 = df_concrete['Age_days'] == 28
filtered_concrete = df_concrete[condition1 & condition2]
print("\nConcrete Samples with Strength > 30 MPa and 28 Days Old:")
# Output: Concrete Samples with Strength > 30 MPa and 28 Days Old:

print(filtered_concrete) # Output:
#            SampleID  Age_days  Cement_kg  Water_liters  Strength_MPa
#         2         3        28        310           148          40.1
#         3         4        28        305           152          39.5

# Filter concrete samples with age of 7 days OR 90 days
concrete_specific_age = df_concrete[(df_concrete['Age_days'] == 7) |
(df_concrete['Age_days'] == 90)]
print("\nConcrete Samples with Age of 7 or 90 Days:")
# Output: Concrete Samples with Age of 7 or 90 Days:

print(concrete_specific_age) # Output:
#            SampleID  Age_days  Cement_kg  Water_liters  Strength_MPa
#         0         1         7        300           150          25.0
#         1         2         7        320           155          26.5
#         4         5        90        315           150          45.0
#         5         6        90        330           160          42.8
```

Data Manipulation and Cleaning

After selecting the data you need, you often must modify the DataFrame structure and improve data quality. This section covers common operations such as creating and removing columns, handling missing values (NaN), and converting data types.

Creating New Columns

You can create new columns by performing calculations on existing ones or by assigning new values. This is widely used in engineering to derive key metrics such as stress, strain, efficiency indicators, and normalized variables.

Example: Calculating Stress, Strain, and Elastic Modulus

```
import pandas as pd
tensile_data = {
    'SampleID': ['T1', 'T2', 'T3', 'T4'],
```

```
    'Area_mm2': [100, 102, 98, 101],
    'Max_Load_N': [50000, 51000, 49000, 50500],
    'Max_Deformation_mm': [0.5, 0.52, 0.48, 0.51],
    'Initial_Length_mm': [100, 100, 100, 100]
}
df_tensile = pd.DataFrame(tensile_data)

# Calculate Stress (Stress = Load / Area) in MPa (N/mm2)
df_tensile['Stress_MPa'] = df_tensile['Max_Load_N'] / df_tensile['Area_mm2']

# Calculate Strain (Strain = Deformation / Initial Length)
df_tensile['Strain_mm_mm'] = df_tensile['Max_Deformation_mm'] /
df_tensile['Initial_Length_mm']

# Estimate Modulus of Elasticity (Modulus = Stress / Strain) in GPa
df_tensile['Elastic_Modulus_GPa'] = (df_tensile['Stress_MPa'] /
df_tensile['Strain_mm_mm']) / 1000

print("\nTensile Data with New Columns:") # Output: Tensile Data with New
Columns:
print(df_tensile)
```

Below is the resulting DataFrame with the newly calculated columns:

```
Tensile Data with New Columns:
SampleID  Area_mm2  Max_Load_N  ...  Stress_MPa       Strain_mm_mm
Elastic_Modulus_GPa
0             T1    100              50000  ...
500.0        0.0050           100.000000
1             T2    102              51000  ...
500.0        0.0052            96.153846
2             T3    98               49000  ...
500.0        0.0048           104.166667
3             T4    101              50500  ...
500.0        0.0051            98.039216

[4 rows x 8 columns]
```

Removing Columns

To remove columns, use the drop() method with the columns parameter. If you want the change applied to the original DataFrame, use inplace=True; otherwise, store the result in a new variable. Removing irrelevant columns helps simplify models, reduce memory usage, and keep datasets focused.

Example: Removing an Unnecessary Column

This example shows how to remove a column from a Pandas DataFrame after performing derived calculations, a common step in data cleaning and preparation. The code first recreates a DataFrame containing tensile test measurements, such as cross-sectional area, applied load, deformation, and initial length. Using these values, two new columns are calculated: stress, obtained by dividing the applied load by the cross-sectional area, and strain, computed as the ratio between deformation and initial length. Once these derived quantities are available, the original column Initial_Length_mm may no longer be necessary for further analysis. The drop() method is then used with the columns parameter to remove this column, returning a new DataFrame (df_tensile_modified) without altering the original one. This approach helps

simplify the dataset, reduce redundancy, and keep only the variables required for subsequent analysis or visualization.

```
import pandas as pd
# Re-creating the DataFrame from the previous step.
tensile_data = {
    'SampleID': ['T1', 'T2', 'T3', 'T4'],
    'Area_mm2': [100, 102, 98, 101],
    'Max_Load_N': [50000, 51000, 49000, 50500],
    'Max_Deformation_mm': [0.5, 0.52, 0.48, 0.51],
    'Initial_Length_mm': [100, 100, 100, 100]
}
df_tensile = pd.DataFrame(tensile_data)
df_tensile['Stress_MPa'] = df_tensile['Max_Load_N'] / df_tensile['Area_mm2']
df_tensile['Strain_mm_mm'] = df_tensile['Max_Deformation_mm'] /
df_tensile['Initial_Length_mm']

# Remove the 'Initial_Length_mm' column
df_tensile_modified = df_tensile.drop(columns=['Initial_Length_mm'])
print("\nTensile Data without 'Initial_Length_mm':")
# Output: Tensile Data without 'Initial_Length_mm':

print(df_tensile_modified)
```

The final DataFrame structure, after discarding the redundant length column, is shown as follows:

```
Tensile Data without 'Initial_Length_mm':
  SampleID  Area_mm2  Max_Load_N  Max_Deformation_mm  Stress_MPa  Strain_mm_mr
0       T1       100       50000                0.50       500.0        0.005(
1       T2       102       51000                0.52       500.0        0.005:
2       T3        98       49000                0.48       500.0        0.004{
3       T4       101       50500                0.51       500.0        0.005:
```

Handling Missing Values (NaN)

Missing values (NaN, Not a Number) are common in real-world datasets due to sensor failures, manual input errors, communication issues, or incomplete records. Pandas provides tools to detect and handle NaN values.

- df.isnull(): Returns a boolean DataFrame indicating True where values are missing
- df.dropna(): Removes rows or columns that contain missing values
- df.fillna(): Fills missing values with a specified value (e.g., mean, median, or zero)

Example: Cleaning Instrumentation Data with Failures

The sensor dataset includes several NaN readings. isnull() identifies their positions, dropna() keeps only complete rows, and fillna() replaces the missing values with the selected mean or median. A copy is filled so that the original measurements remain unchanged.

```
import pandas as pd
import numpy as np

sensor_data = {
    'Timestamp': pd.to_datetime(['2023-01-01 10:00', '2023-01-01 10:05', '2023
10:10', '2023-01-01 10:15', '2023-01-01 10:20']),
    'Sensor_1': [22.1, 22.5, np.nan, 23.0, 22.8],
```

```
    'Sensor_2': [15.0, 15.2, 14.9, 15.1, np.nan],
    'Sensor_3': [5.0, 5.1, 5.0, np.nan, 5.2]
}
df_sensor = pd.DataFrame(sensor_data)

print("\nOriginal DataFrame with Missing Values:")
# Output:  Original DataFrame with Missing Values:

print(df_sensor)
# Output:
#               Timestamp          Sensor_1  Sensor_2  Sensor_3
#         0 2023-01-01 10:00:00      22.1      15.0       5.0
#         1 2023-01-01 10:05:00      22.5      15.2       5.1
#         2 2023-01-01 10:10:00       NaN      14.9       5.0
#         3 2023-01-01 10:15:00      23.0      15.1       NaN
#         4 2023-01-01 10:20:00      22.8       NaN       5.2

# Checking where the missing values are
print("\nMissing Values (True if null):")
# Output:  Missing Values (True if null):
print(df_sensor.isnull())
# Output:
#              Timestamp  Sensor_1  Sensor_2  Sensor_3
#         0      False     False     False     False
#         1      False     False     False     False
#         2      False      True     False     False
#         3      False     False     False      True
#         4      False     False      True     False

# Removing rows that contain any missing value
df_sensor_dropna = df_sensor.dropna()
print("\nDataFrame after removing rows with missing values:")
# Output: DataFrame after removing rows with missing values:

print(df_sensor_dropna)
# Output:
#                Timestamp        Sensor_1  Sensor_2  Sensor_3
#         0 2023-01-01 10:00:00      22.1      15.0       5.0
#         1 2023-01-01 10:05:00      22.5      15.2       5.1

# Filling missing values
df_sensor_fillna = df_sensor.copy()
df_sensor_fillna['Sensor_1'] =
df_sensor_fillna['Sensor_1'].fillna(df_sensor_fillna['Sensor_1'].mean(skipna=T
df_sensor_fillna['Sensor_2'] =
df_sensor_fillna['Sensor_2'].fillna(df_sensor_fillna['Sensor_2'].median(skipna
df_sensor_fillna['Sensor_3'] =
df_sensor_fillna['Sensor_3'].fillna(df_sensor_fillna['Sensor_3'].mean(skipna=T

print("\nDataFrame after filling missing values:")
# Output:  DataFrame after filling missing values:

print(df_sensor_fillna)
```

```
# Output:
#                 Timestamp         Sensor_1  Sensor_2  Sensor_3
#         0 2023-01-01 10:00:00     22.10     15.00     5.000
#         1 2023-01-01 10:05:00     22.50     15.20     5.100
#         2 2023-01-01 10:10:00     22.60     14.90     5.000
#         3 2023-01-01 10:15:00     23.00     15.10     5.075
#         4 2023-01-01 10:20:00     22.80     15.05     5.200
```

Changing Data Types

After cleaning missing values and organizing the structure of the DataFrame, it is essential to ensure that each column has the correct data type. In many real-world datasets, Pandas may automatically infer inappropriate types, for example, numerical values being interpreted as text when data is imported from CSV files or spreadsheets. This can prevent calculations, statistical analysis, and plotting from working correctly. The astype() method allows you to explicitly convert columns to the appropriate data types, such as float or int, ensuring data consistency and enabling reliable numerical operations.

Example: Adjusting Data Types in a Maintenance Log
In the maintenance log, material costs and worked hours initially arrive as text. info() reveals the inferred data types, and astype() converts the two numeric columns to float and int before the corrected DataFrame is displayed.

```
import pandas as pd
import io

maintenance_data = {
    'Service_ID': ['S001', 'S002', 'S003', 'S004'],
    'Material_Cost': ['120.50', '85.20', '150.00', '99.90'],
    'Hours_Worked': ['5', '3', '7', '4'],
    'Status': ['Completed', 'Pending', 'Completed', 'Completed']
}
df_maintenance = pd.DataFrame(maintenance_data)

print("\nOriginal Maintenance DataFrame (types):")
buffer = io.StringIO()
df_maintenance.info(buf=buffer)
print(buffer.getvalue())

# Converting types

df_maintenance['Material_Cost'] =
df_maintenance['Material_Cost'].astype(float)
df_maintenance['Hours_Worked'] = df_maintenance['Hours_Worked'].astype(int)

print("\nMaintenance DataFrame with Converted Types (types):")
buffer = io.StringIO()
df_maintenance.info(buf=buffer)
print(buffer.getvalue())

print(df_maintenance)
```

The output below shows the structure of the DataFrame before and after the data type conversion, highlighting how numeric columns originally interpreted as text were successfully converted to their appropriate numeric types:

```
Original Maintenance DataFrame (types):
<class 'pandas.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Service_ID     4 non-null      str
 1   Material_Cost  4 non-null      str
 2   Hours_Worked   4 non-null      str
 3   Status         4 non-null      str
dtypes: str(4)
memory usage: 260.0+ bytes

Maintenance DataFrame with Converted Types (types):
<class 'pandas.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Service_ID     4 non-null      str
 1   Material_Cost  4 non-null      float64
 2   Hours_Worked   4 non-null      int64
 3   Status         4 non-null      str
dtypes: float64(1), int64(1), str(2)
memory usage: 260.0+ bytes

  Service_ID  Material_Cost  Hours_Worked     Status
0       S001          120.5             5  Completed
1       S002           85.2             3    Pending
2       S003          150.0             7  Completed
3       S004           99.9             4  Completed
```

Statistical Analysis and Grouping

Once the data is clean and properly typed, you can compute statistics and analyze performance across categories. This section introduces descriptive statistics and grouping with groupby().

Descriptive Statistics

Understanding the basic characteristics of data is often the first step in engineering analysis. Beyond describe(), you can compute targeted statistics such as mean, median, standard deviation, min/max, and frequency counts.

Example: Statistical Analysis of Production Data

The production dataset is summarized with targeted statistics. The code calculates the mean and standard deviation of Parts_Produced; the sum, minimum, and maximum of Defects; and the number of records for each shift.

```
import pandas as pd

production_data = {
    'Date': pd.to_datetime(['2023-03-01', '2023-03-02', '2023-03-03', '2023-
03-04', '2023-03-05', '2023-03-06']),
```

```
    'Shift': ['Morning', 'Afternoon', 'Night', 'Morning', 'Afternoon',
'Night'],
    'Parts_Produced': [1000, 980, 950, 1020, 990, 960],
    'Defects': [15, 20, 25, 10, 18, 22]
}
df_production = pd.DataFrame(production_data)

print("\nAverage Parts Produced:", df_production['Parts_Produced'].mean())
print("Standard Deviation of Parts Produced:",
df_production['Parts_Produced'].std())
print("Total Defects:", df_production['Defects'].sum())
print("Minimum Defects:", df_production['Defects'].min())
print("Maximum Defects:", df_production['Defects'].max())

print("\nShift Counts:")
print(df_production['Shift'].value_counts())
```

These results offer a concise statistical overview that aids in performance monitoring and operational analysis.

```
Average Parts Produced: 983.3333333333334
Standard Deviation of Parts Produced: 25.81988897471611
Total Defects: 110
Minimum Defects: 10
Maximum Defects: 25

Shift Counts:
Shift
Morning      2
Afternoon    2
Night        2
Name: count, dtype: int64
```

Grouping Data with groupby()

The groupby() method allows you to split data into groups based on one or more keys (such as shift or equipment) and then compute aggregated metrics for each group. This is widely used for performance monitoring and operational decision-making.

Example: Performance Analysis by Shift

Grouping the production data by Shift separates the calculations for Morning, Afternoon, and Night. The example computes average output, total defects, and a combined summary containing the mean, standard deviation, and defect total.

```
import pandas as pd

production_data = {
    'Date': pd.to_datetime(['2023-03-01', '2023-03-02', '2023-03-03', '2023-
03-04', '2023-03-05', '2023-03-06']),
    'Shift': ['Morning', 'Afternoon', 'Night', 'Morning', 'Afternoon',
'Night'],
    'Parts_Produced': [1000, 980, 950, 1020, 990, 960],
    'Defects': [15, 20, 25, 10, 18, 22]
}
```

```
df_production = pd.DataFrame(production_data)

# Average parts produced per shift

avg_parts_per_shift = df_production.groupby('Shift')
['Parts_Produced'].mean()
print("\nAverage Parts Produced by Shift:")
print(avg_parts_per_shift)

# Total defects per shift

total_defects_per_shift = df_production.groupby('Shift')['Defects'].sum()
print("\nTotal Defects by Shift:")
print(total_defects_per_shift)

# Multiple aggregations

full_shift_analysis = df_production.groupby('Shift').agg(
    Avg_Parts=('Parts_Produced', 'mean'),
    Std_Dev_Parts=('Parts_Produced', 'std'),
    Total_Defects=('Defects', 'sum')
)
print("\nFull Analysis by Shift:")
print(full_shift_analysis)
```

The output below summarizes production performance by shift, showing average output, variability, and total defects to enable a clear comparison between Morning, Afternoon, and Night operations:

```
Average Parts Produced by Shift:
Shift
Afternoon     985.0
Morning      1010.0
Night         955.0
Name: Parts_Produced, dtype: float64

Total Defects by Shift:
Shift
Afternoon    38
Morning      25
Night        47
Name: Defects, dtype: int64

Full Analysis by Shift:
           Avg_Parts  Std_Dev_Parts  Total_Defects
Shift
Afternoon      985.0       7.071068             38
Morning       1010.0      14.142136             25
Night          955.0       7.071068             47
```

Sorting and Combining DataFrames

After computing metrics, it is common to sort results and combine datasets from different sources. This section covers sorting, merging/joining, and concatenation.

Sorting Data

It is possible to sort a DataFrame by one or more columns using "sort_values()". When sorting by categories with a specific order (e.g., High > Medium > Low), it is generally useful to associate the categories with numbers.

Example: Sorting Projects by Priority and Deadline

The project table is first sorted by deadline. Because the priority labels have no inherent order, they are mapped to numbers and used as the primary sort key, with the deadline as the secondary key.

```
import pandas as pd

project_data = {
    'Project': ['P1', 'P2', 'P3', 'P4', 'P5'],
    'Priority': ['High', 'Medium', 'Low', 'High', 'Medium'],
    'Deadline_Days': [30, 45, 60, 20, 50],
    'Budget_Thousand_USD': [150, 100, 80, 200, 120]
}
df_projects = pd.DataFrame(project_data)

print("\nOriginal Projects:")
print(df_projects)

# Sort by Deadline_Days (ascending)

projects_sorted_deadline = df_projects.sort_values(by='Deadline_Days')
print("\nProjects Sorted by Deadline (ascending):")
print(projects_sorted_deadline)

# Sort by Priority (High to Low) then by Deadline_Days

priority_map = {'High': 1, 'Medium': 2, 'Low': 3}
df_projects['Priority_Numeric'] = df_projects['Priority'].map(priority_map)

projects_sorted_complex = df_projects.sort_values(by=['Priority_Numeric',
'Deadline_Days'])
print("\nProjects Sorted by Priority (High to Low) then by Deadline:")
print(projects_sorted_complex)
```

The output below shows the original project list followed by two sorted views—first by increasing deadline and then by project priority (from High to Low) with deadlines used as a secondary sorting criterion:

```
Original Projects:
  Project Priority  Deadline_Days  Budget_Thousand_USD
0      P1     High             30                  150
1      P2   Medium             45                  100
2      P3      Low             60                   80
3      P4     High             20                  200
4      P5   Medium             50                  120

Projects Sorted by Deadline (ascending):
  Project Priority  Deadline_Days  Budget_Thousand_USD
3      P4     High             20                  200
0      P1     High             30                  150
```

```
1       P2   Medium             45                   100
4       P5   Medium             50                   120
2       P3      Low             60                    80

Projects Sorted by Priority (High to Low) then by Deadline:
  Project Priority  Deadline_Days  Budget_Thousand_USD  Priority_Numeric
3       P4     High             20                   200                 1
0       P1     High             30                   150                 1
1       P2   Medium             45                   100                 2
4       P5   Medium             50                   120                 2
2       P3      Low             60                    80                 3
```

Combining DataFrames (Merge and Join)

In engineering, related information is often stored in different tables and must be combined. The pd.merge() function is the Pandas equivalent of SQL joins.

Example: Combining Equipment and Supplier Data

This example code demonstrates how to combine related information from two DataFrames using pd.merge(), showing the difference between an inner join, which keeps only matching supplier records, and a left join, which preserves all equipment entries while adding supplier details when available.

```
import pandas as pd

df_equipment = pd.DataFrame({
    'Equipment_ID': ['EQ001', 'EQ002', 'EQ003', 'EQ004'],
    'Equipment_Name': ['Centrifugal Pump', 'Control Valve', 'Heat
Exchanger', 'Electric Motor'],
    'Supplier_ID': ['S001', 'S002', 'S001', 'S009']
})
df_suppliers = pd.DataFrame({
    'Supplier_ID': ['S001', 'S002', 'S003', 'S005'],
    'Supplier_Name': ['ABC Industrial', 'XYZ Components', 'Delta
Engineering', 'Global Tech'],
    'Location': ['New York', 'Los Angeles', 'Chicago', 'Houston']
})

print("\nEquipment DataFrame:")
print(df_equipment)

print("\nSuppliers DataFrame:")
print(df_suppliers)

# Inner join (default): only rows with a match in both DataFrames

df_equip_supp = pd.merge(df_equipment, df_suppliers, on='Supplier_ID',
how='inner')
print("\nEquipment combined with Suppliers (Inner Join):")
print(df_equip_supp)

# Left join: keeps all equipment and adds supplier info if it exists

df_equip_supp_left = pd.merge(df_equipment, df_suppliers, on='Supplier_ID',
how='left')
```

```
print("\nEquipment combined with Suppliers (Left Join):")
print(df_equip_supp_left)
```

The output below shows the original equipment and supplier tables, followed by the results of combining them using inner and left joins based on the shared Supplier_ID column:

```
Equipment DataFrame:
  Equipment_ID    Equipment_Name Supplier_ID
0        EQ001  Centrifugal Pump        S001
1        EQ002     Control Valve        S002
2        EQ003    Heat Exchanger        S001
3        EQ004    Electric Motor        S009

Suppliers DataFrame:
  Supplier_ID      Supplier_Name     Location
0        S001     ABC Industrial     New York
1        S002     XYZ Components  Los Angeles
2        S003  Delta Engineering      Chicago
3        S005        Global Tech      Houston

Equipment combined with Suppliers (Inner Join):
  Equipment_ID    Equipment_Name Supplier_ID   Supplier_Name     Location
0        EQ001  Centrifugal Pump        S001  ABC Industrial     New York
1        EQ002     Control Valve        S002  XYZ Components  Los Angeles
2        EQ003    Heat Exchanger        S001  ABC Industrial     New York

Equipment combined with Suppliers (Left Join):
  Equipment_ID    Equipment_Name Supplier_ID   Supplier_Name     Location
0        EQ001  Centrifugal Pump        S001  ABC Industrial     New York
1        EQ002     Control Valve        S002  XYZ Components  Los Angeles
2        EQ003    Heat Exchanger        S001  ABC Industrial     New York
3        EQ004    Electric Motor        S009             NaN          NaN
```

Concatenating DataFrames

Concatenation (stacking) is used to combine DataFrames either vertically (along rows, axis=0) or horizontally (along columns, axis=1). This is useful when you have similar datasets split across different sources, such as different shifts or batches.

Example: Joining Data from Different Shifts

pd.concat() stacks the two shift DataFrames vertically because they share the same columns. The ignore_index=True parameter then creates a continuous index for the combined energy consumption dataset.

```
import pandas as pd

df_shift1_conc = pd.DataFrame({
    'Equipment': ['Motor A', 'Pump B'],
    'Consumption_kWh': [150.5, 80.2],
    'Shift': ['S1', 'S1']
})
df_shift2_conc = pd.DataFrame({
    'Equipment': ['Motor A', 'Generator D'],
    'Consumption_kWh': [145.0, 250.0],
```

```
    'Shift': ['S2', 'S2']
})

print("\nShift 1 DataFrame:")
print(df_shift1_conc)

print("\nShift 2 DataFrame:")
print(df_shift2_conc)

df_total_consumption = pd.concat([df_shift1_conc, df_shift2_conc],
ignore_index=True)
print("\nTotal Consumption DataFrame (Concatenated):")
print(df_total_consumption)
```

The output shows the individual energy consumption datasets for each shift and the resulting combined DataFrame after concatenating them into a single table:

```
Shift 1 DataFrame:
  Equipment  Consumption_kWh Shift
0   Motor A            150.5    S1
1    Pump B             80.2    S1

Shift 2 DataFrame:
     Equipment  Consumption_kWh Shift
0      Motor A            145.0    S2
1  Generator D            250.0    S2

Total Consumption DataFrame (Concatenated):
     Equipment  Consumption_kWh Shift
0      Motor A            150.5    S1
1       Pump B             80.2    S1
2      Motor A            145.0    S2
3  Generator D            250.0    S2
```

Exporting Data

After cleaning, transforming, and analyzing your data, you will often want to save it for future use or share it with others. Pandas supports exporting to multiple formats, including CSV, Excel, JSON, HTML, and SQL. This export is commonly used to store processed data for report generation, data sharing, or further analysis in external tools.

Exporting Data to CSV

Example: Saving Processed Energy Consumption Data to CSV
After the two shift tables are concatenated, to_csv() writes the processed data to processed_energy_consumption.csv. Setting index=False prevents the DataFrame index from becoming an extra CSV column.

```
import pandas as pd

df_shift1_conc = pd.DataFrame({
    'Equipment': ['Motor A', 'Pump B'],
    'Consumption_kWh': [150.5, 80.2],
    'Shift': ['S1', 'S1']
```

```
})
df_shift2_conc = pd.DataFrame({
    'Equipment': ['Motor A', 'Generator D'],
    'Consumption_kWh': [145.0, 250.0],
    'Shift': ['S2', 'S2']
})
df_total_consumption = pd.concat([df_shift1_conc, df_shift2_conc],
ignore_index=True)

df_total_consumption.to_csv('processed_energy_consumption.csv', index=False)
print("\nDataFrame 'df_total_consumption' saved as
'processed_energy_consumption.csv'")
# 'index=False' prevents the DataFrame index from being saved as a column.
# Output: DataFrame 'df_total_consumption' saved as
'processed_energy_consumption.csv'
```

Exporting Data to Excel

It is possible to export the analysis results of a Pandas DataFrame to an Excel file.

See the example code below. First, it creates a DataFrame containing tensile test measurements and calculates two derived quantities: stress, obtained by dividing the applied load by the cross-sectional area, and strain, obtained by dividing the deformation by the initial length. The to_excel() method then saves the enriched DataFrame as tensile_results.xlsx. The index=False parameter prevents the index from being written as a column, while sheet_name='Test Results' sets the worksheet name.

Example: Saving Tensile Test Results to an Excel File

```
import pandas as pd

tensile_data = {
    'SampleID': ['T1', 'T2', 'T3', 'T4'],
    'Area_mm2': [100, 102, 98, 101],
    'Max_Load_N': [50000, 51000, 49000, 50500],
    'Max_Deformation_mm': [0.5, 0.52, 0.48, 0.51],
    'Initial_Length_mm': [100, 100, 100, 100]
}
df_tensile = pd.DataFrame(tensile_data)
df_tensile['Stress_MPa'] = df_tensile['Max_Load_N'] / df_tensile['Area_mm2']
df_tensile['Strain_mm_mm'] = df_tensile['Max_Deformation_mm'] /
df_tensile['Initial_Length_mm']

df_tensile.to_excel('tensile_results.xlsx', index=False, sheet_name='Test
Results')
print("\nDataFrame 'df_tensile' saved as 'tensile_results.xlsx'")
# Output: DataFrame 'df_tensile' saved as 'tensile_results.xlsx'
```

Exporting to Other Formats

In addition to CSV and Excel, Pandas can export to many other formats, such as JSON (to_json()), SQL databases (to_sql()), and HTML tables (to_html()). Choose the format that best suits your workflow and target tools. Consult the updated official Pandas documentation for more information on other supported formats.

Basic Data Visualization with Pandas

Visualization is a crucial step for communicating results. Pandas integrates well with Matplotlib and can be combined with libraries such as Seaborn. This section includes examples of line/scatter plots and histogram/bar charts.

Line and Scatter Plots

Line and scatter plots are among the most widely used visualization techniques in engineering and scientific analysis, as they allow trends, relationships, and patterns in data to be identified quickly and intuitively. Line plots are especially useful for representing the evolution of a variable over time or another continuous parameter, while scatter plots are ideal for examining relationships between two measured quantities, such as stress and strain in material testing. In this section, these two plot types are used to visualize process monitoring data and experimental results, demonstrating how Pandas and Matplotlib can be combined to produce clear, professional-quality figures suitable for reports, publications, and automated workflows.

Example: Line and Scatter Plots

The first DataFrame is displayed as a line plot to show change over time. The second uses a scatter plot to show the relationship between paired values. Titles and axis labels make the distinction between the chart types clear.

```
import pandas as pd
import matplotlib.pyplot as plt

# Line plot example
df_line = pd.DataFrame({
    "Time": [0, 1, 2, 3, 4],
    "Value": [10, 12, 15, 14, 18]
})

plt.plot(df_line["Time"], df_line["Value"])
plt.title("Line Plot Example")
plt.xlabel("Time")
plt.ylabel("Value")
plt.show()

# Scatter plot example
df_scatter = pd.DataFrame({
    "X": [1, 2, 3, 4, 5],
    "Y": [2, 4, 5, 8, 10]
})

plt.scatter(df_scatter["X"], df_scatter["Y"])
plt.title("Scatter Plot Example")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

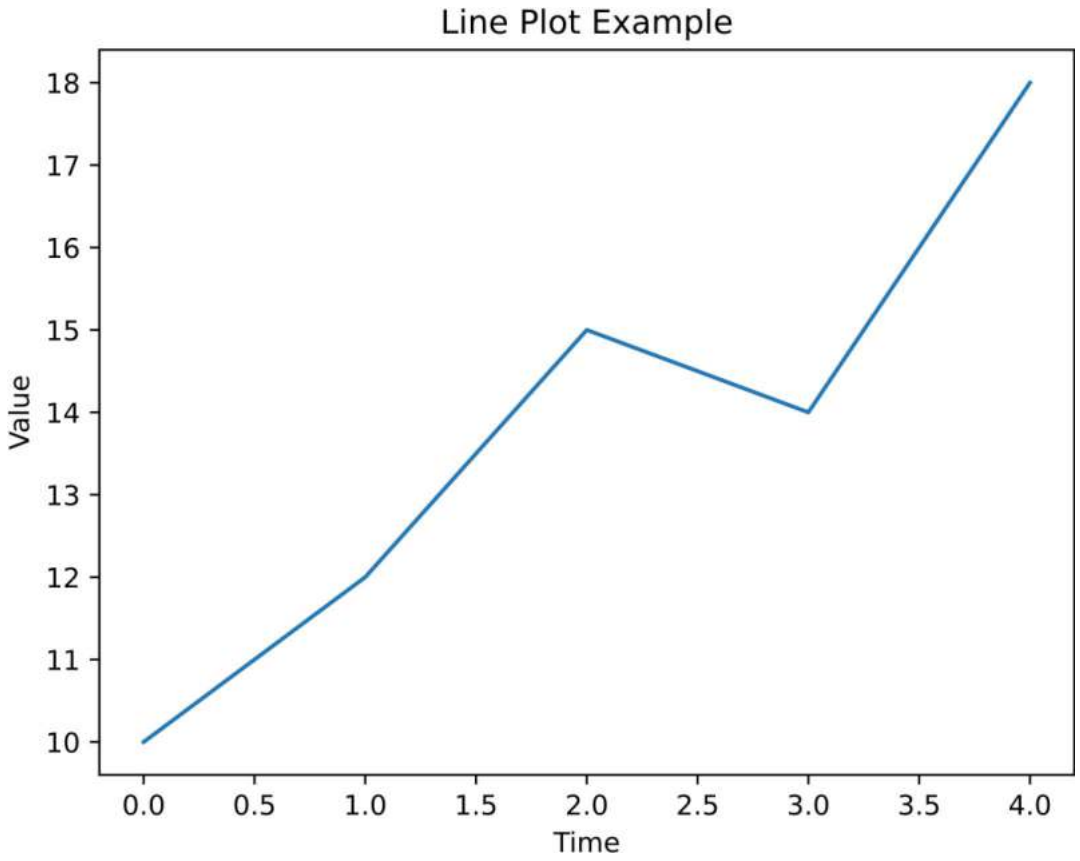


Figure 14-2 Pandas line plot example

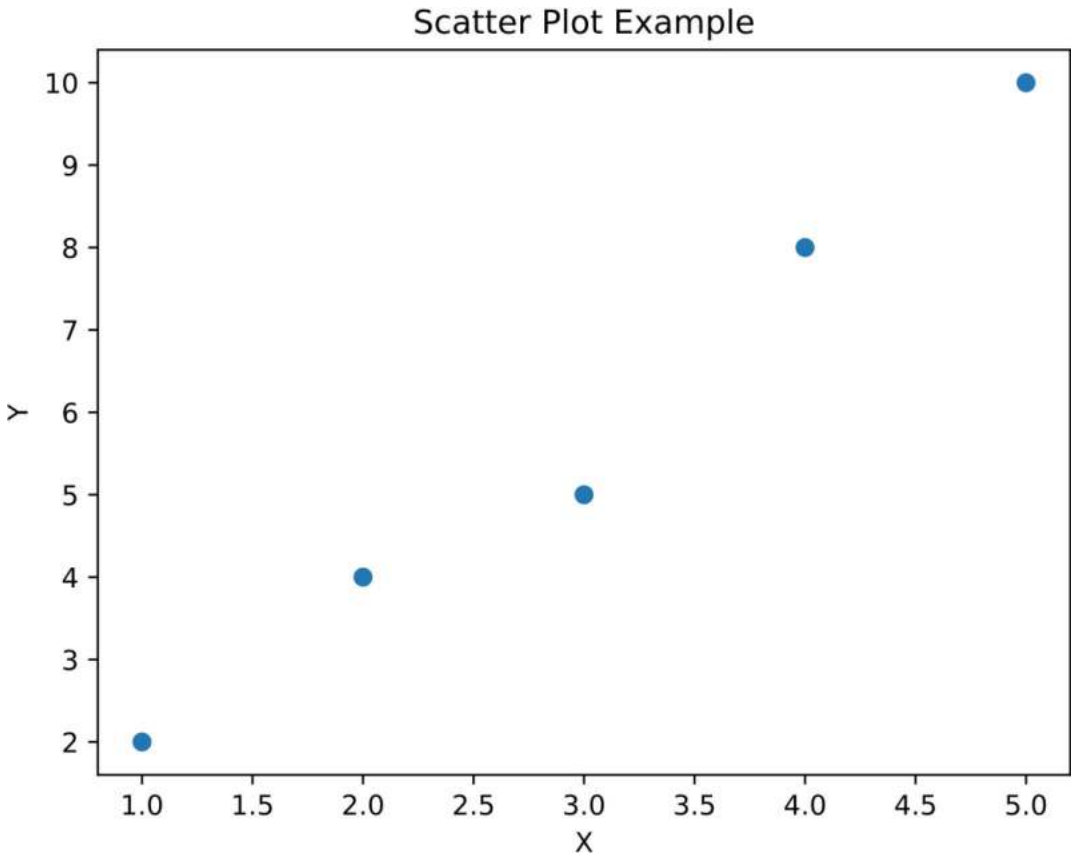


Figure 14-3 Pandas scatter plot example

Engineering Line and Scatter Plots

While this guide focuses on Pandas, data visualization is a crucial step in engineering for communicating insights. Pandas integrates well with Matplotlib and Seaborn.

Example: Temperature vs. Time and Stress vs. Strain

This extended example creates engineering line and scatter plots and saves them as PNG files. The temperature plot shows process change over time, while the stress–strain scatter plot presents the calculated peak points. plt.clf() clears each figure before the next plot is created.

```
import pandas as pd
import matplotlib.pyplot as plt

# Re-creating the df_process DataFrame for the first plot
process_data = {
    'Time_min': [0, 10, 20, 30, 40, 50, 60],
    'Temperature_C': [80.5, 82.1, 81.0, 83.5, 84.0, 83.2, 85.0],
    'Pressure_MPa': [1.2, 1.3, 1.25, 1.35, 1.38, 1.30, 1.40],
    'Flow_Lps': [10.1, 10.5, 9.8, 10.3, 10.8, 10.0, 11.0]
}
df_process = pd.DataFrame(process_data)

# Re-creating the df_tensile DataFrame for the second plot
tensile_data = {
    'SampleID': ['T1', 'T2', 'T3', 'T4'],
    'Area_mm2': [100, 102, 98, 101],
```

```
    'Max_Load_N': [50000, 51000, 49000, 50500],
    'Max_Deformation_mm': [0.5, 0.52, 0.48, 0.51],
    'Initial_Length_mm': [100, 100, 100, 100]
}
df_tensile = pd.DataFrame(tensile_data)
df_tensile['Stress_MPa'] = df_tensile['Max_Load_N'] / df_tensile['Area_mm2']
df_tensile['Strain_mm_mm'] = df_tensile['Max_Deformation_mm'] /
df_tensile['Initial_Length_mm']

# Visualizing Temperature vs. Time (from df_process)
plt.figure(figsize=(8, 5))
plt.plot(df_process['Time_min'], df_process['Temperature_C'], marker='o',
linestyle='-')
plt.title('Process Temperature Over Time')
plt.xlabel('Time (min)')
plt.ylabel('Temperature (°C)')
plt.grid(True)
plt.savefig('process_temperature.png') # Changed from plt.show()
plt.clf()

# Visualizing Stress vs. Strain (from df_tensile)
plt.figure(figsize=(8, 5))
plt.scatter(df_tensile['Strain_mm_mm'], df_tensile['Stress_MPa'])
plt.title('Stress-Strain Curve (Peak Points)')
plt.xlabel('Strain (mm/mm)')
plt.ylabel('Stress (MPa)')
plt.grid(True)
plt.savefig('stress_strain_curve.png') # Changed from plt.show()
plt.clf()
```

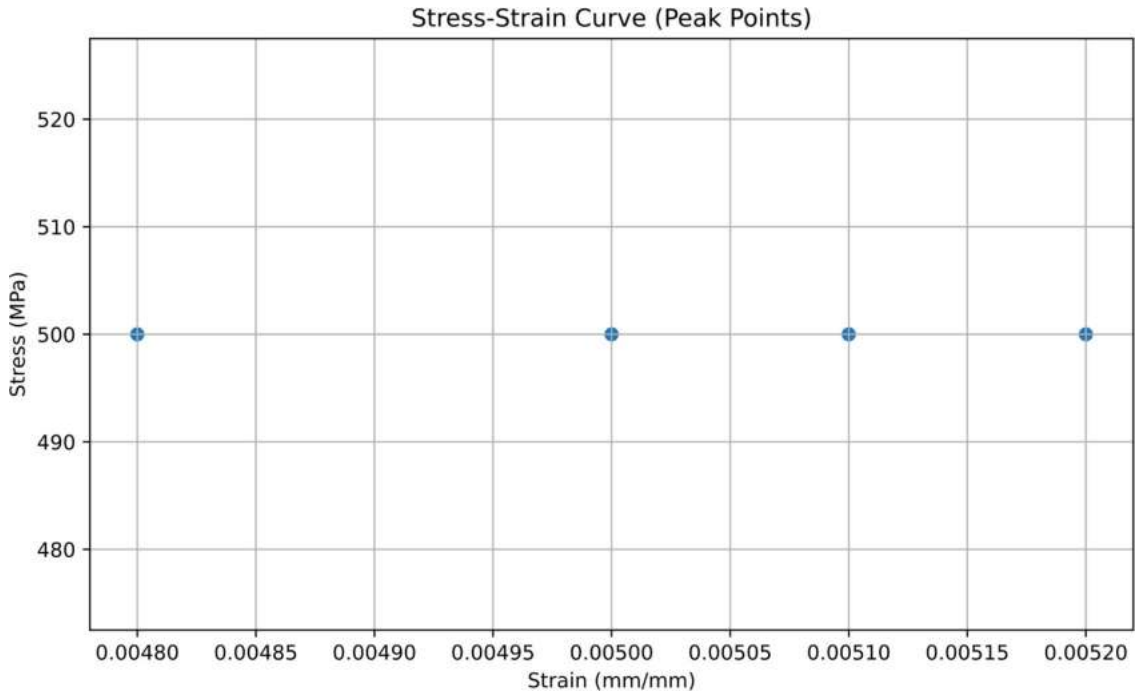


Figure 14-4 Pandas peak points plot

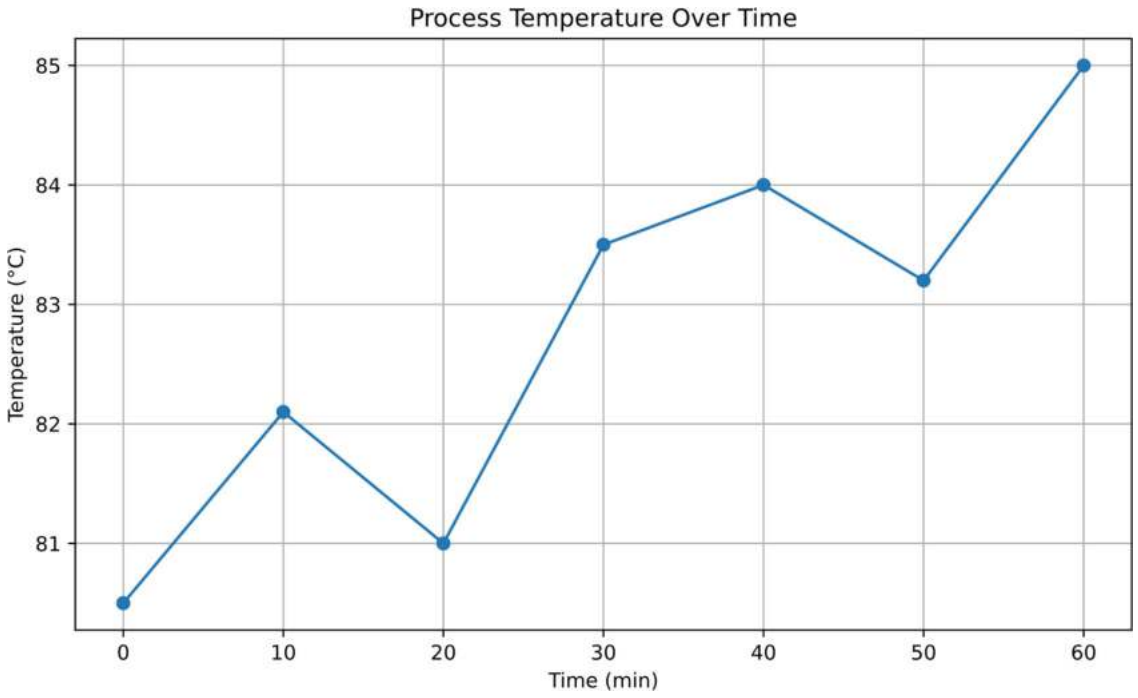


Figure 14-5 Temperature × time plot

Histograms and Bar Charts

Histograms and bar charts are essential tools for summarizing and comparing data in engineering and scientific analyses. Histograms are particularly useful for visualizing the distribution and spread of continuous variables, such as material strength, allowing patterns, variability, and outliers to be easily identified. Bar charts, on the other hand, are well suited for comparing aggregated values across discrete categories, such as average production output per work shift. In this example, these two visualization types are applied to analyze the distribution of concrete compressive strength and to compare average parts produced across different shifts, demonstrating how Pandas, Matplotlib, and Seaborn can be combined to produce clear and informative graphical representations.

Example: Concrete Strength Distribution and Parts Produced per Shift

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Re-creating the df_concrete DataFrame for the histogram
concrete_data = {
    'SampleID': [1, 2, 3, 4, 5, 6, 7],
    'Age_days': [7, 7, 28, 28, 90, 90, 180],
    'Cement_kg': [300, 320, 310, 305, 315, 330, 325],
    'Water_liters': [150, 155, 148, 152, 150, 160, 158],
    'Strength_MPa': [25.0, 26.5, 40.1, 39.5, 45.0, 42.8, 48.2]
}
df_concrete = pd.DataFrame(concrete_data)

# Re-creating the avg_parts_per_shift Series for the bar chart
production_data = {
    'Date': pd.to_datetime(['2023-03-01', '2023-03-02', '2023-03-03', '2023-
03-04', '2023-03-05', '2023-03-06']),
```

```
    'Shift': ['Morning', 'Afternoon', 'Night', 'Morning', 'Afternoon',
'Night'],
    'Parts_Produced': [1000, 980, 950, 1020, 990, 960],
    'Defects': [15, 20, 25, 10, 18, 22]
}
df_production = pd.DataFrame(production_data)
avg_parts_per_shift = df_production.groupby('Shift')
['Parts_Produced'].mean()

# Histogram of concrete strength (from df_concrete)
plt.figure(figsize=(7, 5))
sns.histplot(df_concrete['Strength_MPa'], kde=True, bins=5)
plt.title('Distribution of Concrete Strength')
plt.xlabel('Strength (MPa)')
plt.ylabel('Frequency')
plt.grid(True)
plt.savefig('concrete_strength_histogram.png') # Changed from plt.show()
plt.clf()

# Bar chart of average parts produced per shift (using the grouped analysis)
plt.figure(figsize=(7, 5))
avg_parts_per_shift.plot(kind='bar', color='skyblue')
plt.title('Average Parts Produced by Shift')
plt.xlabel('Shift')
plt.ylabel('Average Parts')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.tight_layout()
plt.savefig('avg_parts_per_shift_barchart.png') # Changed from plt.show()
plt.clf()
```

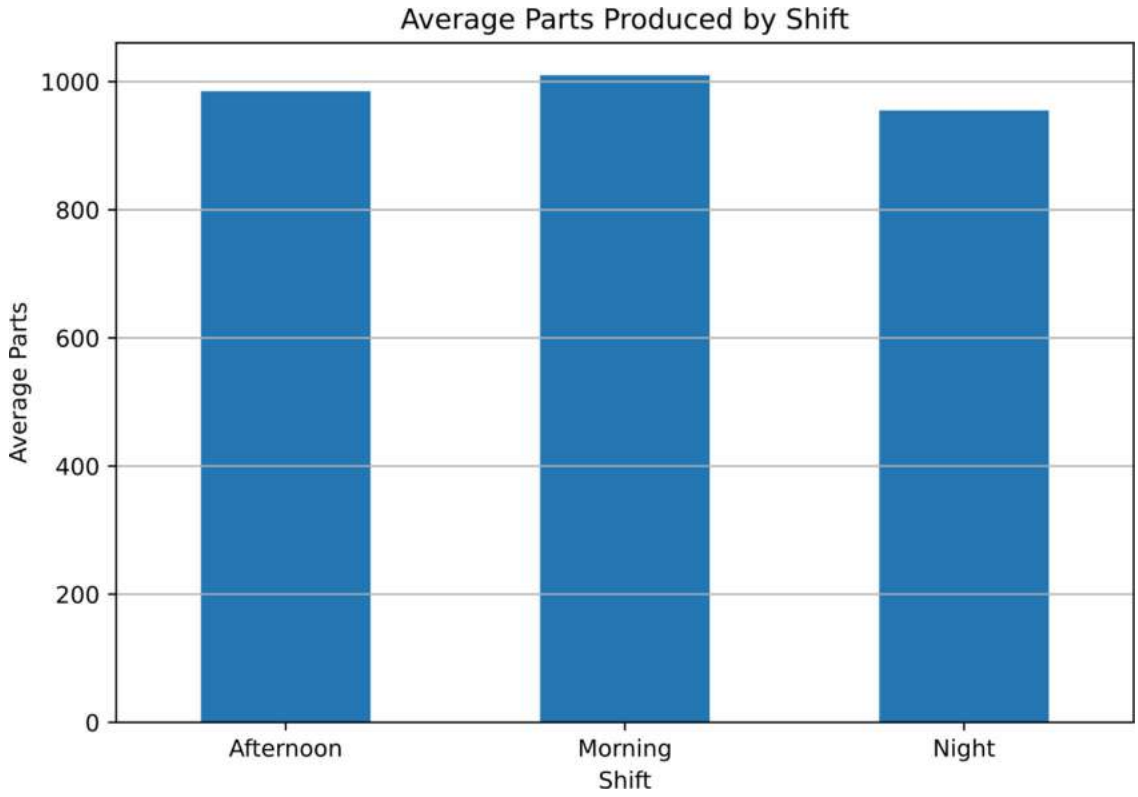


Figure 14-6 Bar chart example

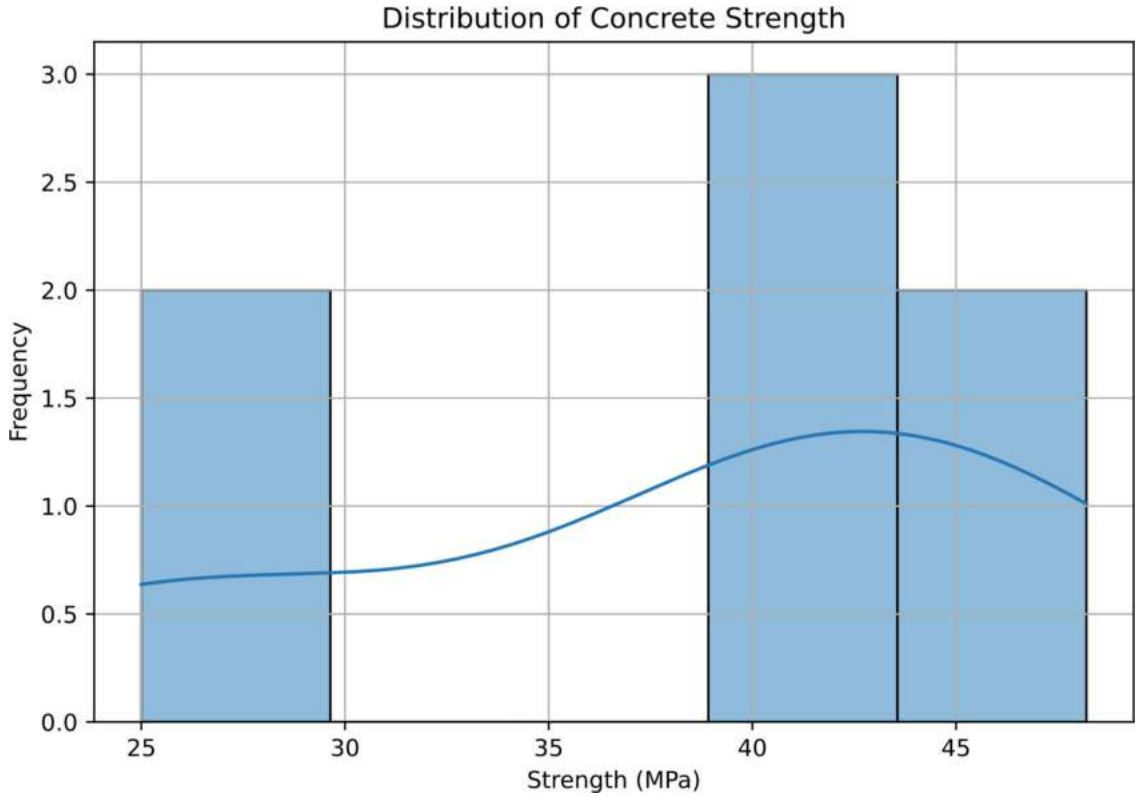


Figure 14-7 Histogram with Kernel Density Estimate

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_15

15. Control Systems with Python

Carlos Márcio da Silva Freitas[1] ✉
(1) São Paulo, Brazil

Dynamic System Control

Dynamic system control is a broad field with applications across engineering including electrical, mechatronics, automation, and mechanical systems as well as in biology, aerospace, robotics, chemical processes, renewable energy systems, and transportation. Its primary goal is to regulate system behavior to meet specified performance criteria, such as maintaining variables at desired setpoints, ensuring stability, and optimizing energy use. By implementing appropriate control strategies, engineers can influence system dynamics to achieve targeted outcomes, enhancing efficiency, reliability, and overall performance.

Python provides a variety of tools for control systems, ranging from basic implementations to advanced applications. To support both learning and professional practice, the table below presents the main libraries commonly used for the design and analysis of control systems, organized according to their application areas and primary focus.

It's worth remembering that SciPy's signal module contains functions for control systems. However, a dedicated control systems library has more functionality in this area and is easier to use.

Table

This chapter introduces the control library, a powerful open source Python package for modeling, analyzing, and designing control systems. Built on top of NumPy and SciPy, it provides an intuitive interface for working with transfer functions, state-space models, and frequency response data. Widely adopted by engineers, students, and researchers, the library offers essential functions for feedback control design, robust support for linear systems, and simulation capabilities for nonlinear models, making it a versatile resource for studying and implementing dynamic systems.

As a free and actively maintained project, the control library stands as a high-quality alternative to proprietary software. Supported by a community of more than 70 developers and thousands of users worldwide, it continues to evolve through regular updates and contributions.

With the control library, you can build mathematical models of physical systems and assemble block diagrams by connecting multiple components. It enables simulation and visualization of system behavior over time, stability analysis, and frequency response evaluation while also providing access to intrinsic model properties such as poles and zeros. Moreover, it supports controller design for specific performance goals, the handling of nonlinear models to explore optimal control solutions, and the simplification of complex models without losing essential dynamics. Altogether, these features make the library a powerful and flexible tool for designing, testing, and optimizing control systems.

Key features of the library are described in the following table.

Table

To simplify adoption, the control systems library was designed with MATLAB users in mind. It even offers a dedicated compatibility module to make the transition smoother. This is particularly useful when working with control systems books or exercises that use MATLAB as the primary reference. As a result, many of the common MATLAB

Control System Toolbox commands have direct Python equivalents, making it easier to implement familiar workflows.

In total, there are approximately 35 MATLAB commands compatible with the Python control library.

Here's the command distribution by category:

- *System Modeling and Conversion*: 5
- *Time-Domain Simulation*: 5
- *Frequency-Domain Analysis*: 6
- *System Properties*: 4
- *Interconnections (Block Diagrams)*: 5
- *State-Space Analysis and Design*: 10

See below for examples of some common commands between MATLAB and the control systems library.

Table

To install the control systems library in your Python application, run the following command in your terminal or command prompt. This command is the same for Windows, macOS, and Linux:

```
pip install control
```

The python-control package makes use of the NumPy and SciPy libraries and plotting with Matplotlib.

After installing the library, create a small Python code to check the version installed on your computer:

```
import control
print(control.__version__)
```

It should print the installed version number, for example, 0.10.2.

The version installed to run the codes in this chapter was version 0.10.2 with Python version 3.13.5.

Now, see how to use the main commands and functions of the control systems library through the following examples. Due to the large number of features available in the library, we will cover only the most commonly used commands. More information about the control library can be found in its online documentation available at https://python-control.readthedocs.io/en/latest/index.html.

System Creation

This section introduces the creation of dynamic system models using the Python control library. It demonstrates how continuous-time systems can be defined mathematically by specifying their parameters in transfer function and state-space forms. These representations serve as the starting point for subsequent analysis, simulation, and controller design, allowing engineers to describe system dynamics in a structured and computationally efficient way.

Create a State-Space System

The ss(A, B, C, D[, dt]) command creates a state-space system, where A, B, C, and D are the system matrices and dt is the optional sampling time. If dt is specified, it creates a discrete-time system.

This example is based on the mechanical system of a mass, spring, and damper.

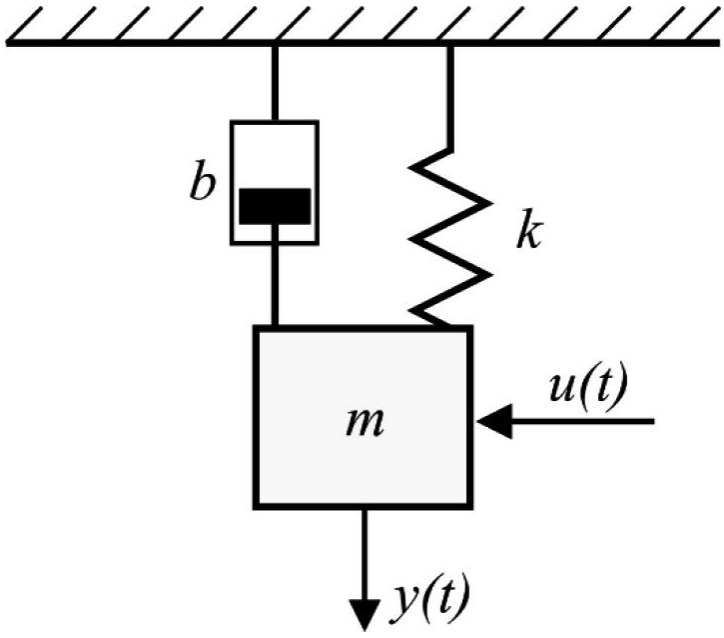


Figure 15-1 State-space representation of a mass–spring–damper system defined using matrices A, B, C, and D

```
import control as ct
import numpy as np

# System matrices for a mass-spring-damper system
m = 1    # mass
b = 0.5  # damping coefficient
k = 1    # spring constant

A = np.array([[0, 1], [-k/m, -b/m]])
B = np.array([[0], [1/m]])
C = np.array([1, 0])
D = 0

# Create the state-space system
sys_ss = ct.ss(A, B, C, D)
print("State-space model:\n", sys_ss)
```

The print(sys_ss) command will display a block of text that clearly lists the matrices A, B, C, and D that you defined, so the sys_ss object is

now ready for further analysis, such as simulating its step response, checking its stability, or designing a controller for it.

```
State-space model:
 <StateSpace>: sys[0]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']
States (2): ['x[0]', 'x[1]']

A = [[ 0.   1. ]
     [-1.  -0.5]]

B = [[0.]
     [1.]]

C = [[1. 0.]]

D = [[0.]]
```

Create a Transfer Function

The tf(num, den[, dt]) command creates a transfer function system, where num and den are the numerator and denominator coefficients and dt is the optional sampling time.

This example represents an RLC circuit as a transfer function.

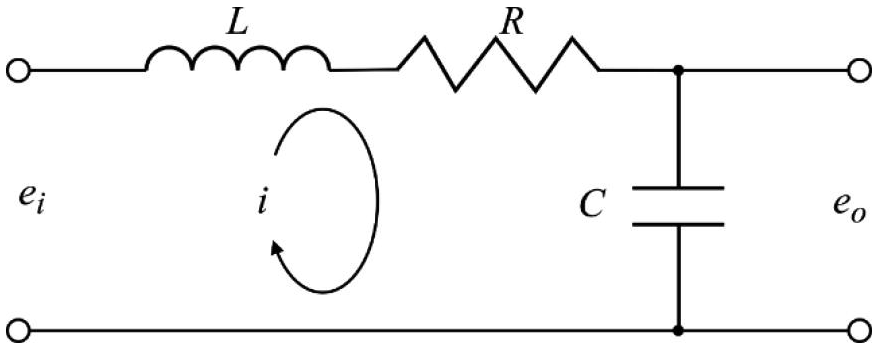


Figure 15-2 Second-order transfer function model representing an RLC circuit created using the tf() command

```
import control as ct
```

```
# Coefficients for the transfer function of an RLC
circuit
L = 1  # Inductance
C = 1  # Capacitance
R = 1  # Resistance

num = [1]
den = [L*C, R*C, 1]

sys_tf = ct.tf(num, den)
print("Transfer function model:\n", sys_tf)
```

The print(sys_tf) command will display the resulting transfer function.

```
Transfer function model:
 <TransferFunction>: sys[0]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

       1
  -----------
  s^2 + s + 1
```

This sys_tf object represents a second-order low-pass filter, and from now on it can be used for further analysis within the control library, such as plotting its frequency response or simulating its step response.

Frequency Response Data

This example creates a frequency response data (FRD) model, a way to define a system's behavior based on known data points at specific frequencies, which is useful when you have experimental measurements. The frd(frdata, omega[, dt]) command constructs a frequency response data (FRD) model, where

- omega (frequencies) is a set of specific frequencies being tested, including low, mid-range, and high frequencies
- frdata (complex response data) is the system's response at each frequency, represented as complex numbers that capture both magnitude (gain) and phase shift information
- ct.frd(...) is a function that combines the frequency and response data into a single frequency response data (FRD) object, providing a direct representation of the system's behavior

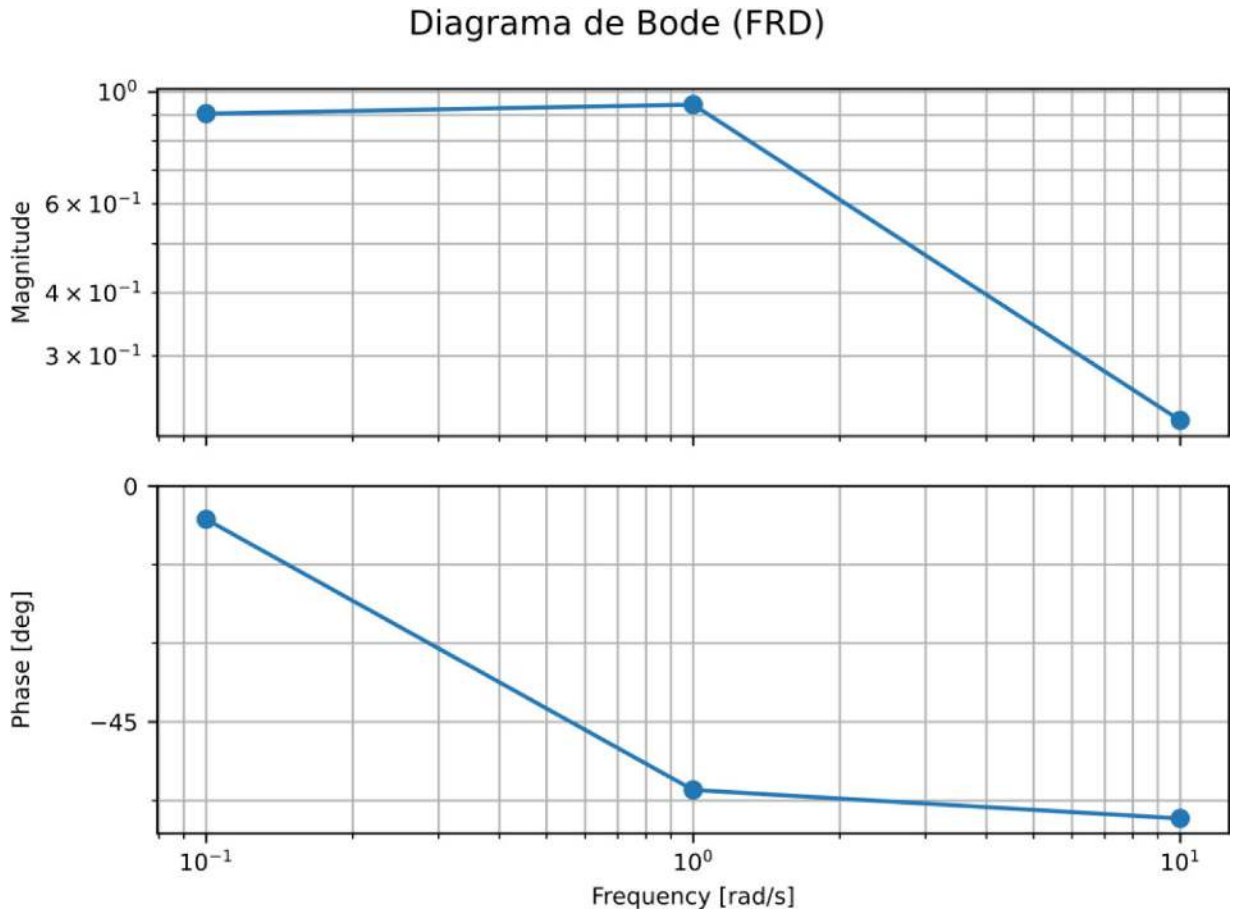


Figure 15-3 Frequency response data (FRD) model constructed from discrete experimental frequency measurements

```
import control as ct
import numpy as np

# Frequency points (in rad/s)
omega = np.array([0.1, 1, 10])
```

```
# Corresponding complex frequency response data
frdata = np.array([0.9-0.1j, 0.5-0.8j, 0.1-0.2j])

# Create the FRD model
sys_frd = ct.frd(frdata, omega)
print("Frequency response data model:\n", sys_frd)
```

The print(sys_frd) command will generate a summary of the FRD model. It will confirm that the model stored frequency response data for the three specified frequencies (0.1, 1, and 10 rad/s). Importantly, the result can now be used for system identification, where you attempt to find a transfer function that best fits these experimental points.

```
Frequency response data model:
 <FrequencyResponseData>: sys[0]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

Freq [rad/s]  Response
------------  ---------------------
       0.100          0.9      -0.1j
       1.000          0.5      -0.8j
      10.000          0.1      -0.2j
```

Create a Nonlinear System

The nlsys(updfcn[, outfcn]) command creates a nonlinear input/output system, where updfcn is the update function and outfcn is the optional output function.

This example creates a nonlinear input/output model of a physical pendulum.

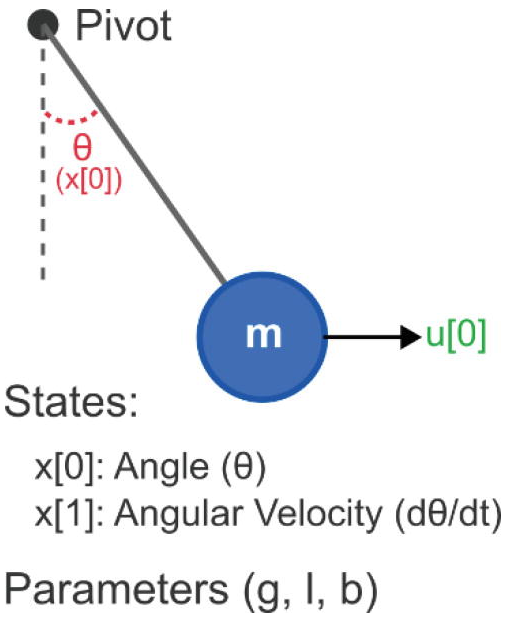


Figure 15-4 Nonlinear pendulum model defined as a nonlinear input/output system with two states and one input

```
import control as ct
import numpy as np

def pendulum_update(t, x, u, params):
    g = params.get('g', 9.8)
    l = params.get('l', 1.0)
    m = params.get('m', 1.0)
    b = params.get('b', 0.1)
    return [x[1], -b / (m * l**2) * x[1] - g / l *
np.sin(x[0]) + u[0] / (m * l**2)]

def pendulum_output(t, x, u, params):
    return x

# Create the nonlinear system
sys_nlsys = ct.nlsys(
    pendulum_update, pendulum_output,
    states=2,
    name='pendulum',
    inputs=1,          # We have one input, u[0]
```

```
    outputs=2)        # We have two outputs, x[0]
and x[1]
print("Nonlinear system model:\n", sys_nlsys)
```

The function below is the core of the model, implementing the pendulum's equation of motion:

```
def pendulum_update(t, x, u, params):
```

It defines the dynamics in terms of the angle x[0] and angular velocity x[1]. The model uses bob mass m, pendulum length l, viscous rotational damping b, gravity g, and applied torque u[0]. The sine term preserves the nonlinear restoring torque, so the model remains valid beyond the small-angle approximation sin(theta) approximately theta.

The pendulum_output function defines what is being measured from the system. By returning x, it means we are observing both the angle (x[0], or θ) and the angular velocity (x[1], or $\dot{\theta}$):

```
def pendulum_output(t, x, u, params):
```

The ct.nlsys function packages the update and output functions into a single nonlinear system object named sys_nlsys, making it easier to work with and analyze the pendulum's behavior:

```
sys_nlsys = ct.nlsys(……)
```

The print(sys_nlsys) command will display a summary of the object you created:

```
Nonlinear system model:
 <NonlinearIOSystem>: pendulum
Inputs (1): ['u[0]']
Outputs (2): ['y[0]', 'y[1]']
States (2): ['x[0]', 'x[1]']
```

```
Update: <function pendulum_update at
0x00000128E8073CE0>
Output: <function pendulum_output at
0x00000128E807D1C0>
```

It won't show matrices, but it will confirm that you have a nonlinear system named "pendulum" with two state variables, one input, and two outputs.

Create a Transfer Function from Zeros, Poles, and Gain

The zpk(zeros, poles, gain[, dt]) command creates a transfer function from zeros, poles, and gain, where zeros and poles are the system zeros and poles, gain is the system gain, and dt is the optional sampling time.

```
import control as ct

# Define zeros, poles, and gain
zeros = [-1]
poles = [-2, -3]
gain = 10

# Create the ZPK model
sys_zpk = ct.zpk(zeros, poles, gain)
print("Zero-pole-gain model:\n", sys_zpk)
```

The print(sys_zpk) command will display the assembled transfer function in its factored, zero–pole–gain form. The output will look exactly like this:

```
Zero-pole-gain model:
 <TransferFunction>: sys[0]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

    10 s + 10
  ------------
```

```
s^2 + 5 s + 6
```

Model Conversions and Interconnections

In practical control engineering, dynamic systems are often analyzed and designed using multiple mathematical representations rather than a single fixed model. Transfer functions, state-space models, and frequency-domain descriptions each offer specific advantages depending on the analysis or control task being performed. This section introduces the fundamental concepts and methods for converting between these representations, allowing engineers to integrate classical and modern control approaches within the same workflow. In addition, it addresses the interconnection of subsystems through series, parallel, and feedback configurations, which is essential for accurately modeling real-world systems composed of interacting components such as actuators, sensors, and controllers. By combining model conversion techniques with system interconnections, engineers can build comprehensive and flexible control models suitable for simulation, analysis, and controller design.

Convert a System into a Canonical Form

A canonical form is a standardized way of representing a system, facilitating the analysis and design of control systems. It's like a specific format for writing the system's equations.

The canonical_form function converts a given system into a canonical form. In this case, we're converting a state-space system into a controllable canonical form, also known as an achievable canonical form. Converting a system into a canonical form can be useful for simplifying analysis, as canonical forms can facilitate the analysis of system properties, such as stability and controllability. It also provides a more straightforward way to design control systems, such as state-feedback controllers.

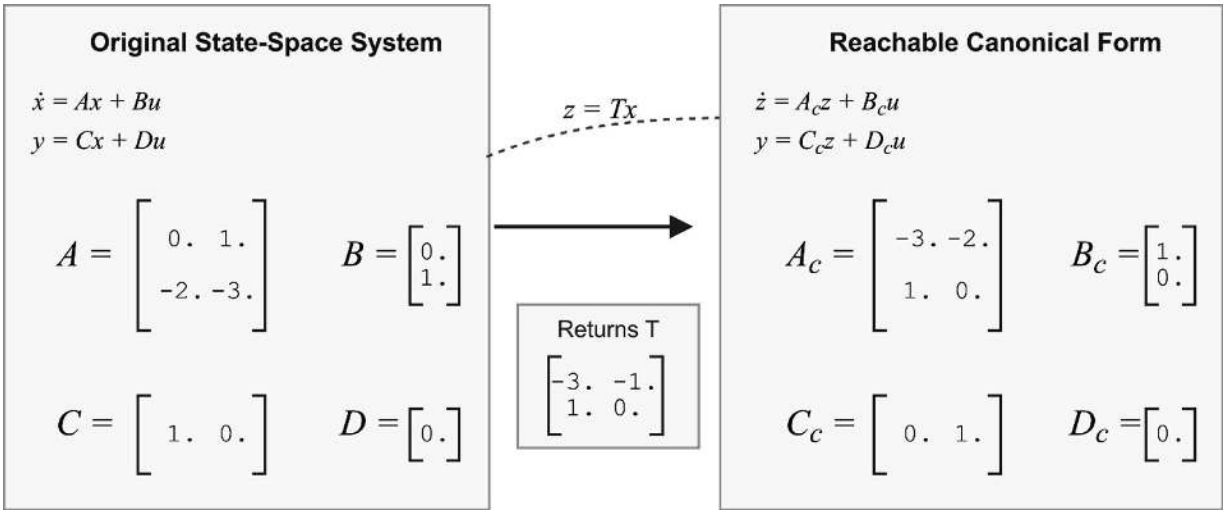


Figure 15-5 Reachable (controllable) canonical form obtained from a state-space system using a similarity transformation

A reachable canonical form is a specific type of canonical form useful for analyzing and designing control systems. It is called "reachable" because it highlights the controllable states of the system.

```
import control as ct
import numpy as np
# A state-space system
A = np.array([[0., 1.], [-2., -3.]])
B = np.array([[0.], [1.]])
C = np.array([[1., 0.]])
D = np.array([[0.]])
sys = ct.ss(A, B, C, D)

# Convert to controllable canonical form
sys_canonical, T = ct.canonical_form(sys,
form='reachable')
print("Reachable canonical form:\n",
sys_canonical)

# Print the transformation matrix that was
calculated
print("\n--- Transformation Matrix (T) ---")
```

```
print(T)
```

The code above defines a state-space system (sys) with matrices A, B, C, and D. We use ct.canonical_form to convert the system to a reachable canonical form (sys_canonical). The argument form='reachable' specifies that we want the reachable canonical form.

The function returns the transformed system (sys_canonical) and a transformation matrix (T).

```
Reachable canonical form:
 <StateSpace>: sys[0]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']
States (2): ['x[0]', 'x[1]']

A = [[-3. -2.]
     [ 1.  0.]]

B = [[1.]
     [0.]]

C = [[0. 1.]]

D = [[0.]]

--- Transformation Matrix (T) ---
[[0. 1.]
 [1. 0.]]
```

Series Connection of I/O Systems

When two systems are connected in series, the output of the first system becomes the input to the second system. The overall transfer function of the series connection is the product of the individual transfer functions.

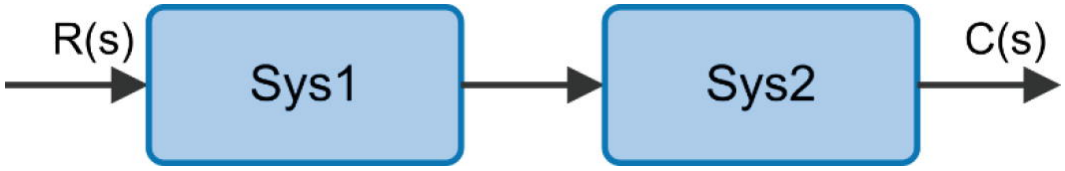


Figure 15-6 Series interconnection of two transfer function systems resulting in a combined second-order system

```
import control as ct

sys1 = ct.tf([1], [1, 1])
sys2 = ct.tf([1], [1, 2])

# Connect in series
sys_series = ct.series(sys1, sys2)
print("Series connection:\n", sys_series)
```

The ct.series call connects sys1 and sys2 so that the output of sys1 drives sys2; both models were created with ct.tf.

The print statement displays the resulting series connection, which includes the transfer function coefficients.

```
Series connection:
 <TransferFunction>: sys[2]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

        1
  -------------
  s^2 + 3 s + 2
```

Parallel Connection of I/O Systems

When two systems are connected in parallel, the inputs to both systems are the same, and the outputs are added together. The overall transfer function of the parallel connection is the sum of the individual transfer functions.

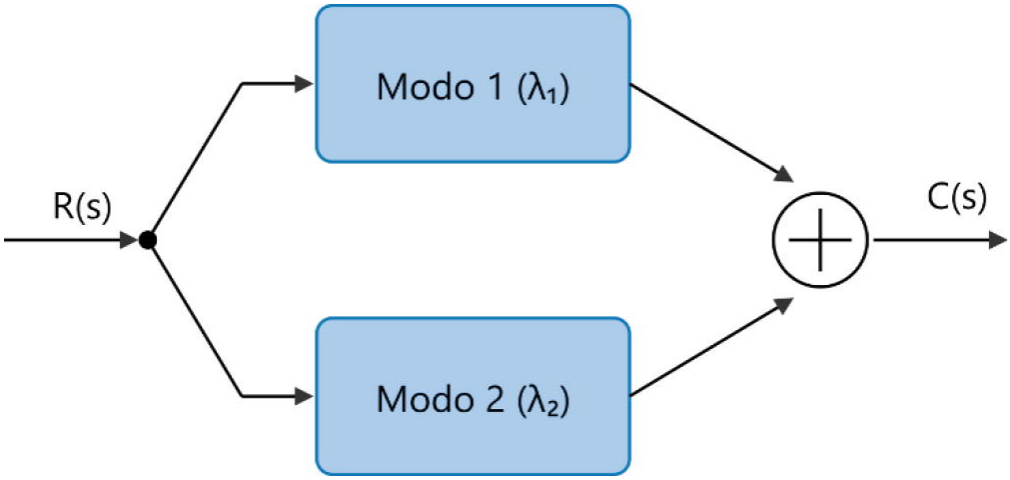


Figure 15-7 Parallel interconnection of two transfer function systems illustrating output summation behavior

```
import control as ct

sys1 = ct.tf([1], [1, 1])
sys2 = ct.tf([1], [1, 2])

# Connect in parallel
sys_parallel = ct.parallel(sys1, sys2)
print("Parallel connection:\n", sys_parallel)
```

For the parallel connection, ct.parallel applies the same input to sys1 and sys2 and sums their outputs.

- sys1 has a transfer function of 1 / (s + 1).
- sys2 has a transfer function of 1 / (s + 2).

To add the two transfer functions, we need to find a common denominator:

1 / (s + 1) + 1 / (s + 2) = ((s + 2) + (s + 1)) / ((s + 1)(s + 2))
= (2s + 3) / (s^2 + 3s + 2)
So the resulting transfer function is
(2s + 3) / (s^2 + 3s + 2)
The print statement displays the resulting parallel connection, which includes the transfer function coefficients.

```
Parallel connection:
 <TransferFunction>: sys[2]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

     2 s + 3
  -------------
  s^2 + 3 s + 2
```

Time-Domain Analysis

Time-domain analysis is a fundamental aspect of control engineering, as it allows engineers to evaluate how a dynamic system responds to inputs and disturbances over time. By examining time-dependent responses such as step, impulse, and ramp inputs, it is possible to assess key performance characteristics including stability, transient behavior, rise time, overshoot, settling time, and steady-state error. This section introduces the main concepts and tools used for time-domain analysis, demonstrating how system dynamics can be interpreted through temporal responses and how these responses provide direct insight into the effectiveness and robustness of a control system.

Forced Response of a Linear System

The forced response of a linear system describes its output resulting solely from an external input signal, assuming zero initial conditions. In this context, the system response is driven exclusively by the applied input, without any contribution from initial state values. Forced-response analysis is widely used to evaluate how a system behaves under known excitations, such as sinusoidal, step, or arbitrary input signals.

In this example, the system is defined as a second-order transfer function with numerator [1] and denominator [1,2,1], representing a stable linear dynamic system. The input signal is a sinusoidal function with a frequency of 1 rad/s, generated using np.sin(T). The system

response is evaluated over a time interval from 0 to 10 seconds using a uniformly spaced time vector containing 1,000 samples.

The ct.forced_response() function is used to compute the time response of a dynamic system subjected to an external input signal. It evaluates how the system output evolves over time based on a specified input waveform and time vector, assuming zero initial conditions unless otherwise specified. This function can be applied to systems defined either as transfer functions or as state-space models, making it a versatile tool for time-domain analysis.

When the system is defined in state-space form, ct.forced_response() directly integrates the state equations and returns the time vector, the output response, and, optionally, the state trajectories. By setting return_x=True, the function returns the internal state evolution explicitly, allowing detailed inspection of the system dynamics and providing access to physically meaningful states when they have been defined by the user.

When the system is defined as a transfer function, the library internally converts the model into an equivalent state-space realization in order to perform the time-domain simulation. In this case, the returned output response accurately represents the system behavior, while the returned state trajectory (when return_x=True is used) corresponds to the internal mathematical states of the chosen realization. These states are not unique and should be interpreted as computational variables rather than physical system states.

Depending on the modeling approach and analysis objectives, ct.forced_response() may return two or three variables. By default, it returns the time vector and output response. When the state trajectory is required, the additional argument return_x=True can be provided, in which case the function returns the time vector, output response, and state evolution. This flexibility allows the function to be used effectively for both high-level system response analysis and detailed state-level inspection.

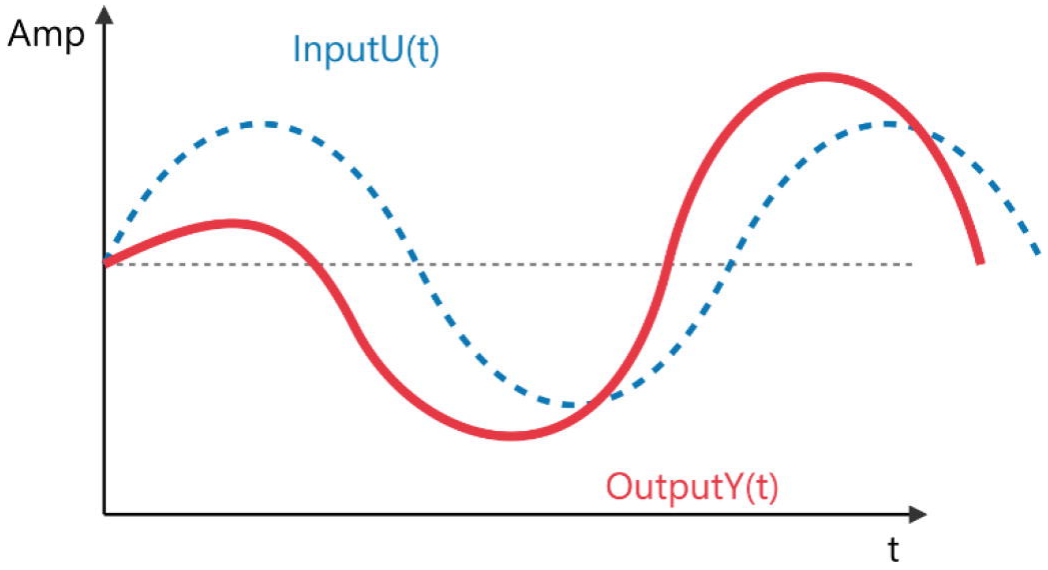


Figure 15-8 Time-domain response of a linear dynamic system showing the relationship between the input signal u(t) and the corresponding output response y(t), highlighting amplitude attenuation and phase lag

```
import control as ct
import numpy as np
import matplotlib.pyplot as plt

# Transfer function: 1 / (s^2 + 2s + 1) = 1 / (s +
1)^2
sys = ct.tf([1], [1, 2, 1])

T = np.linspace(0, 10, 1000)
U = np.sin(T)

# For TF models, forced_response typically returns
(T, yout)
T_out, yout = ct.forced_response(sys, T, U)

plt.plot(T_out, U, label="Input u(t) = sin(t)",
linestyle="--")
plt.plot(T_out, yout, label="Output y(t)")
plt.title("Forced Response (Input and Output)")
plt.xlabel("Time (s)")
```

```
plt.ylabel("Amplitude")
plt.grid(True)
plt.legend()
plt.show()
```

Using the specified sinusoidal input and time vector, ct.forced_response computes yout, the system output over time. The plot compares this response with the input and shows the resulting attenuation and phase lag.

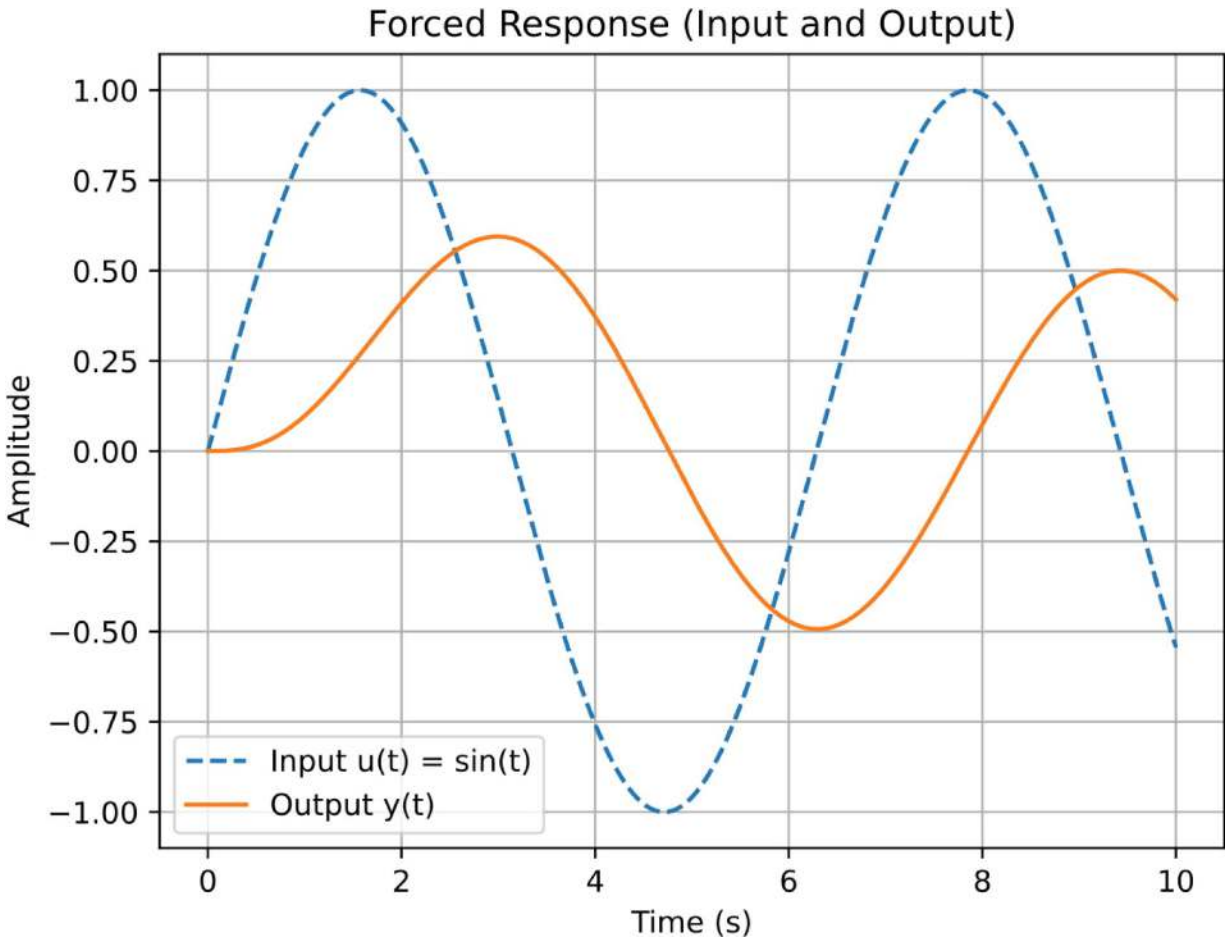


Figure 15-9 Forced response of a linear system to the sinusoidal input u(t) = sin(t), showing output attenuation and phase lag in y(t)

Compute the Impulse Response for a Linear System

The impulse response of a system is its output when subjected to a brief, isolated impulse-like input. It is a fundamental characteristic of a

system that reveals its behavior and stability.

The example system is defined as a transfer function (sys) with numerator [1] and denominator [1, 2, 1]. This represents a second-order system. The ct.impulse_response function calculates the system's impulse response (yout) in a time matrix (T).

```
import control as ct
import matplotlib.pyplot as plt

sys = ct.tf([1], [1, 2, 1])
T, yout = ct.impulse_response(sys)

plt.plot(T, yout)
plt.title("Impulse Response")
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.grid(True)
plt.show()
```

Plotting yout against T shows the transient produced by a unit impulse and helps assess the system's stability and decay rate.

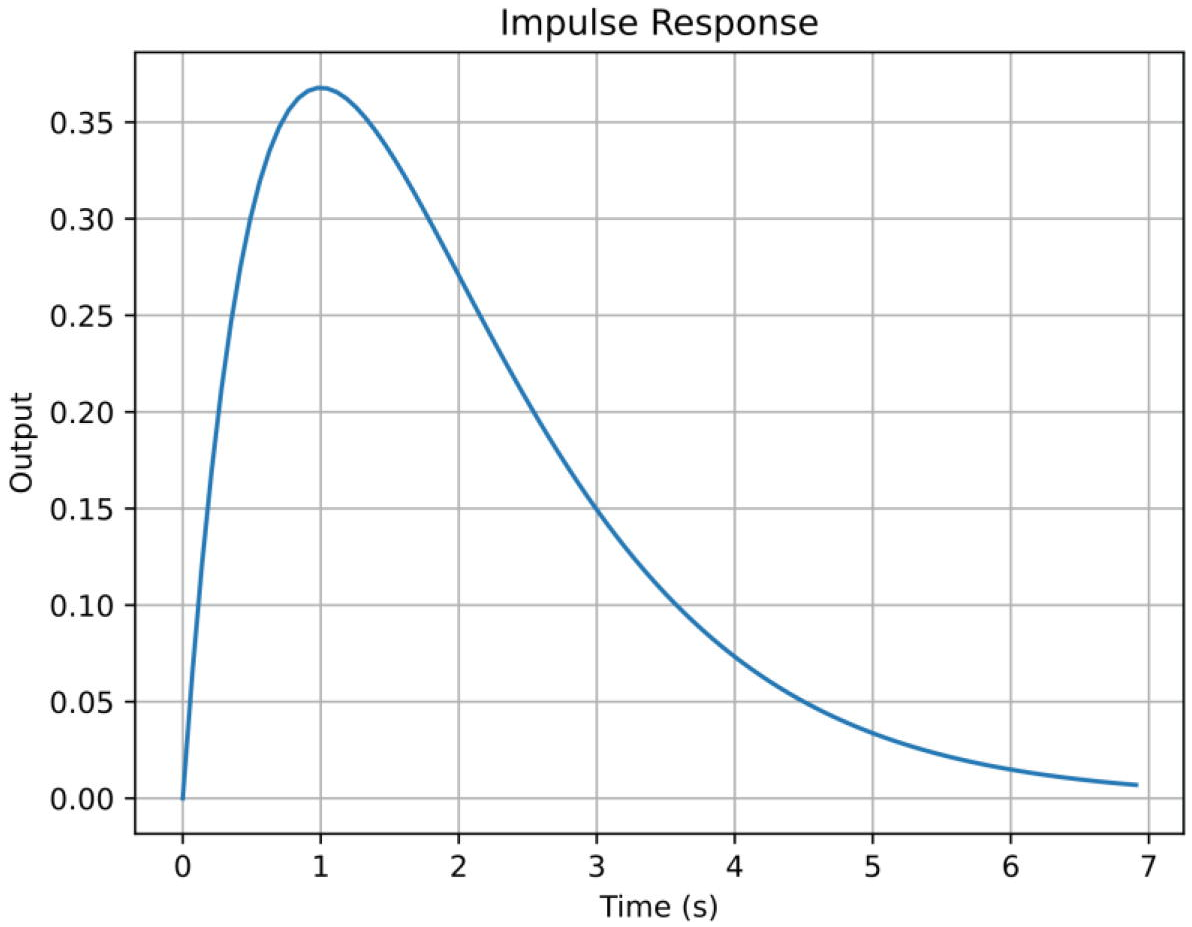


Figure 15-10 Impulse response of a linear dynamic system, illustrating the transient behavior following a unit impulse input

The impulse amplitude is not explicitly defined in the code, as the ct.impulse_response function assumes a unitary impulse (with amplitude 1) as input.

If you want to simulate an impulse with a different amplitude, you can multiply the output of the ct.impulse_response function by the desired amplitude.

```
T, yout = ct.impulse_response(sys)
yout = 2 * yout
```

Compute the Step Response for a Linear System

The step response of a system is its output when subjected to a sudden and sustained change in input, such as a step. It is especially useful for

analyzing the behavior and stability of a system.

Here, ct.step_response is applied to the second-order model with numerator [1] and denominator [1, 2, 1], and Matplotlib displays the result.

The ct.step_response function calculates the system's step response (yout) in a time matrix (T) and also automatically generates this time matrix (T) based on the system's dynamics.

```
import control as ct
import matplotlib.pyplot as plt

sys = ct.tf([1], [1, 2, 1])
T, yout = ct.step_response(sys)

plt.plot(T, yout)
plt.title("Step Response")
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.grid(True)
plt.show()
```

The step response graph can reveal several system properties, such as stability, overshoot, settling time, steady-state gain, and the final response value, which indicates the system's DC gain. This allows us to design more effective control strategies.

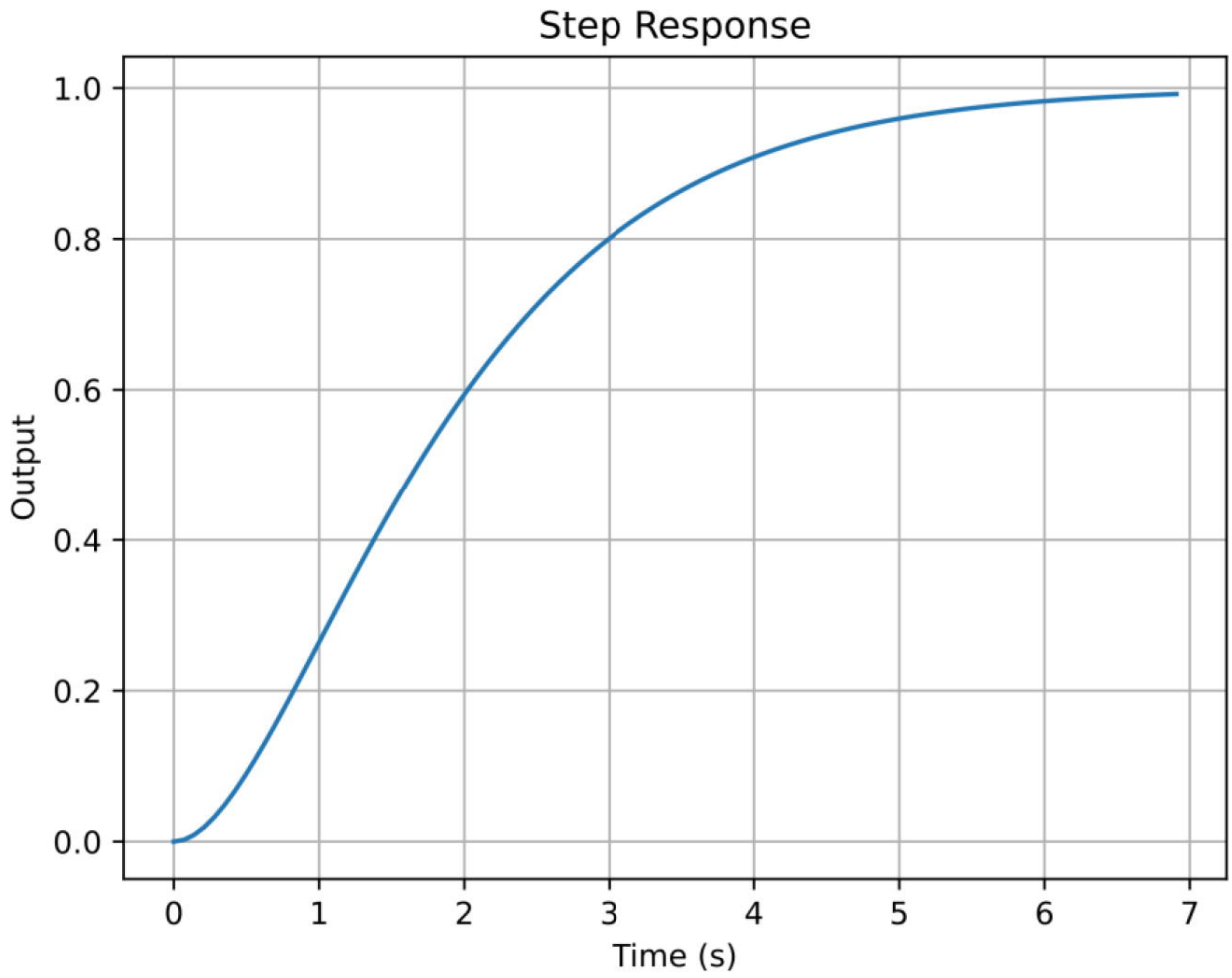


Figure 15-11 Time-domain step response of a linear system showing transient dynamics and steady-state convergence

Plot a Phase Plane Diagram

A phase plane plot is a visual tool widely used in nonlinear systems that shows how the state of a dynamical system evolves over time. Instead of plotting one variable as a function of time, as in a typical graph, it plots two state variables of the system relative to each other, for example, velocity as a function of position.

The phase plane plot can reveal various properties of the system, such as stability, oscillations, and limit cycles if the trajectories approach and enter a closed loop. By analyzing the phase plane plot, we can gain a better understanding of the behavior and dynamics of nonlinear systems like a pendulum.

```
import control as ct
import numpy as np
import matplotlib.pyplot as plt

def pendulum_update(t, x, u, params):
    g = params.get('g', 9.8)
    l = params.get('l', 1.0)
    m = params.get('m', 1.0)
    b = params.get('b', 0.1)
    return [x[1], -b / (m * l**2) * x[1] - g / l *
np.sin(x[0])]

def pendulum_output(t, x, u, params):
    return x

# Define the system with 0 inputs
sys_nlsys = ct.nlsys(pendulum_update,
pendulum_output,
                     states=2, name='pendulum',
inputs=0)

# 1. Define the plot boundaries: [xmin, xmax,
ymin, ymax]
plot_boundaries = [-2, 2, -2, 2]  # You can adjust
these values

# 2. Plot the phase portrait over the specified
region
ct.phase_plane_plot(sys_nlsys,
pointdata=plot_boundaries, plot_streamplot=True)

plt.show()
```

The example defines the nonlinear pendulum with ct.nlsys and passes the four plot boundaries as pointdata to ct.phase_plane_plot.

With plot_streamplot=True, the function draws the vector field and streamlines throughout that region.

The pendulum system is defined using two functions:

- pendulum_update: This function defines the dynamics of the pendulum, including gravity g, length l, mass m, and viscous rotational damping b.
- pendulum_output: This function defines the output of the system, which in this case is the state vector itself (x).

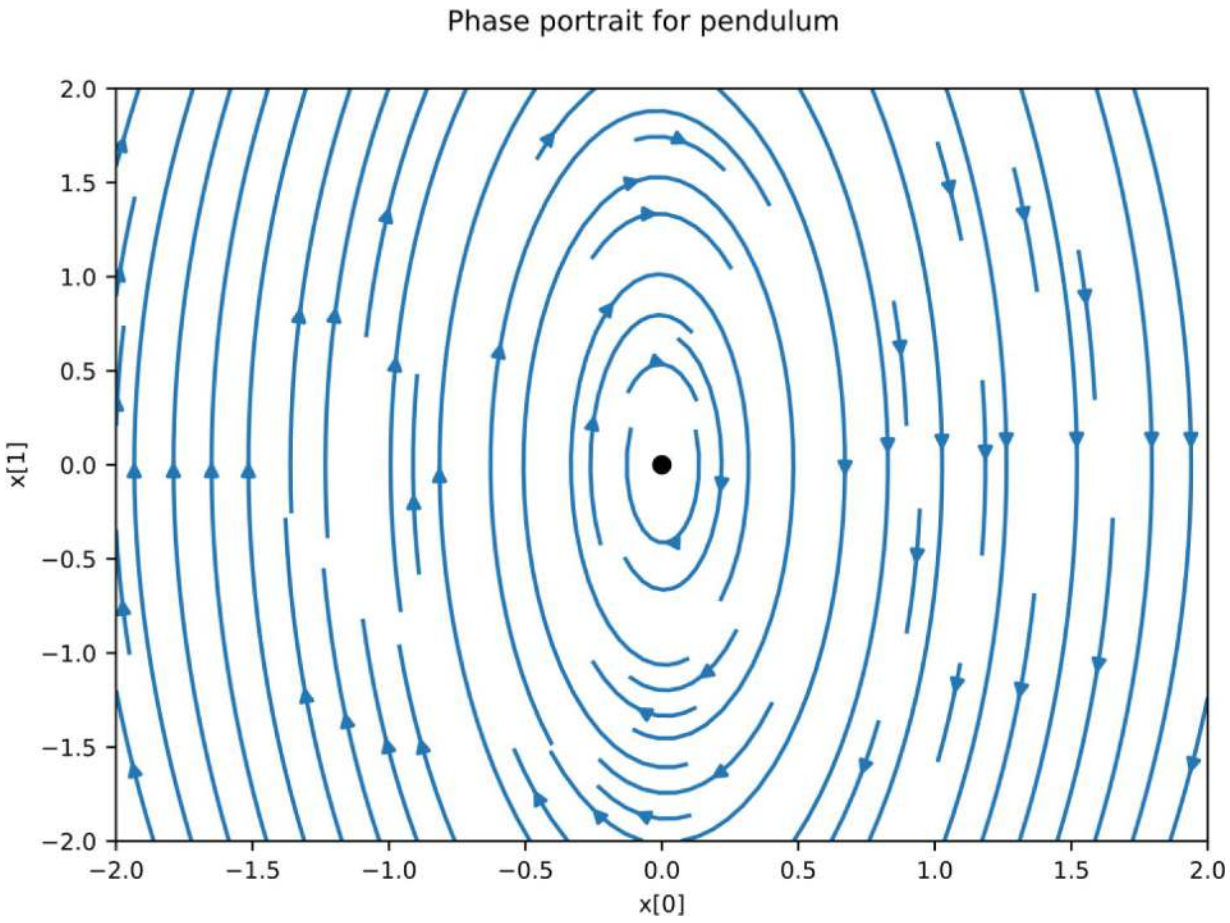


Figure 15-12 Phase portrait of a nonlinear pendulum system in the state space, illustrating the trajectories of angular position and angular velocity around an equilibrium point

The system is created using ct.nlsys, which represents a nonlinear system.

The graph shows streamlines generated throughout the specified state-space region; each streamline is tangent to the nonlinear vector

field. The limits [–2, 2, –2, 2] correspond to angular position and angular velocity, and damping causes trajectories near the equilibrium [0, 0] to converge toward it.

Frequency-Domain Analysis

This section focuses on analyzing control systems in the frequency domain using computational tools provided by the Python control library. It demonstrates how frequency response characteristics such as magnitude and phase can be evaluated through Bode plots, allowing engineers to assess system behavior across a range of frequencies. Frequency-domain analysis is particularly useful for understanding stability margins, resonance effects, and robustness, and it complements time-domain methods by providing insight into how systems respond to sinusoidal inputs.

Bode Plot for a System

A Bode plot is a graphical representation of a system's frequency response, showing the magnitude and phase of the system's output as a function of frequency. The plot consists of two subplots: the magnitude plot, which shows the system's gain or magnitude as a function of frequency, and the phase plot, which shows the system's phase shift as a function of frequency.

The Bode plot can reveal various system properties, such as bandwidth, gain, phase shift, and stability. By analyzing the Bode plot, we can better understand the system's behavior and design more effective control strategies.

The example system is defined as a second-order transfer function (sys) with numerator [10] and denominator [1, 2, 10].

```
import control as ct
import matplotlib.pyplot as plt

sys = ct.tf([10], [1, 2, 10])
ct.bode_plot(sys)
```

```
plt.show()
```

The result created by the bode_plot(sys) function and plotted by Matplotlib is as follows.

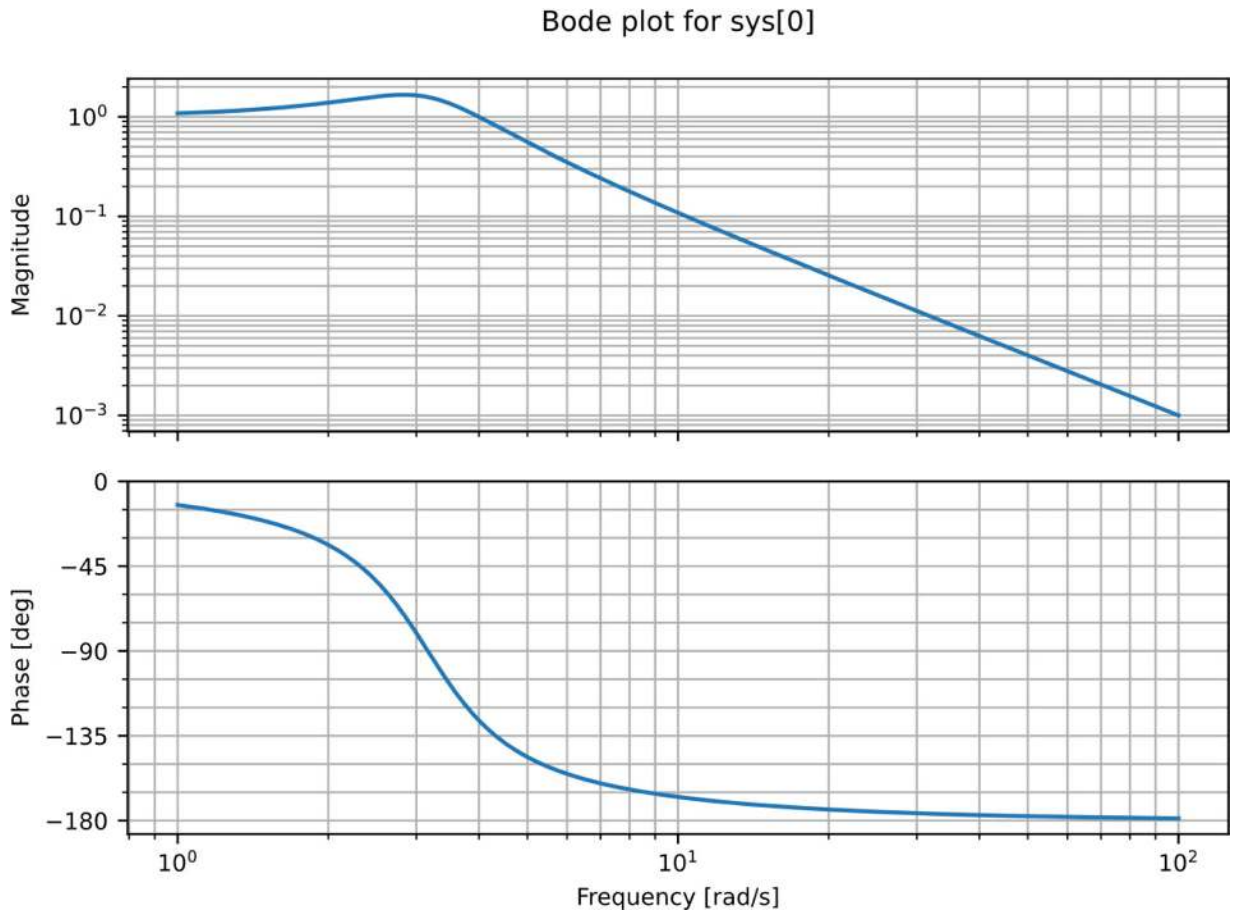


Figure 15-13 Bode magnitude and phase plots of a linear system, illustrating its frequency-domain response

Nyquist Plot for a System

A Nyquist plot is a graphical representation of a system's frequency response in the complex plane. It shows the locus of the system's transfer function as the frequency varies from 0 to infinity. The plot can be used to determine stability and stability margins.

The example system is defined as a transfer function (sys) of second-order with a numerator of [10] and a denominator of [1, 2, 10].

```
import control as ct
```

```
import matplotlib.pyplot as plt

sys = ct.tf([10], [1, 2, 10])
ct.nyquist_plot(sys)
plt.show()
```

The example code generates a Nyquist plot for a transfer function system (sys) using the ct.nyquist_plot function.

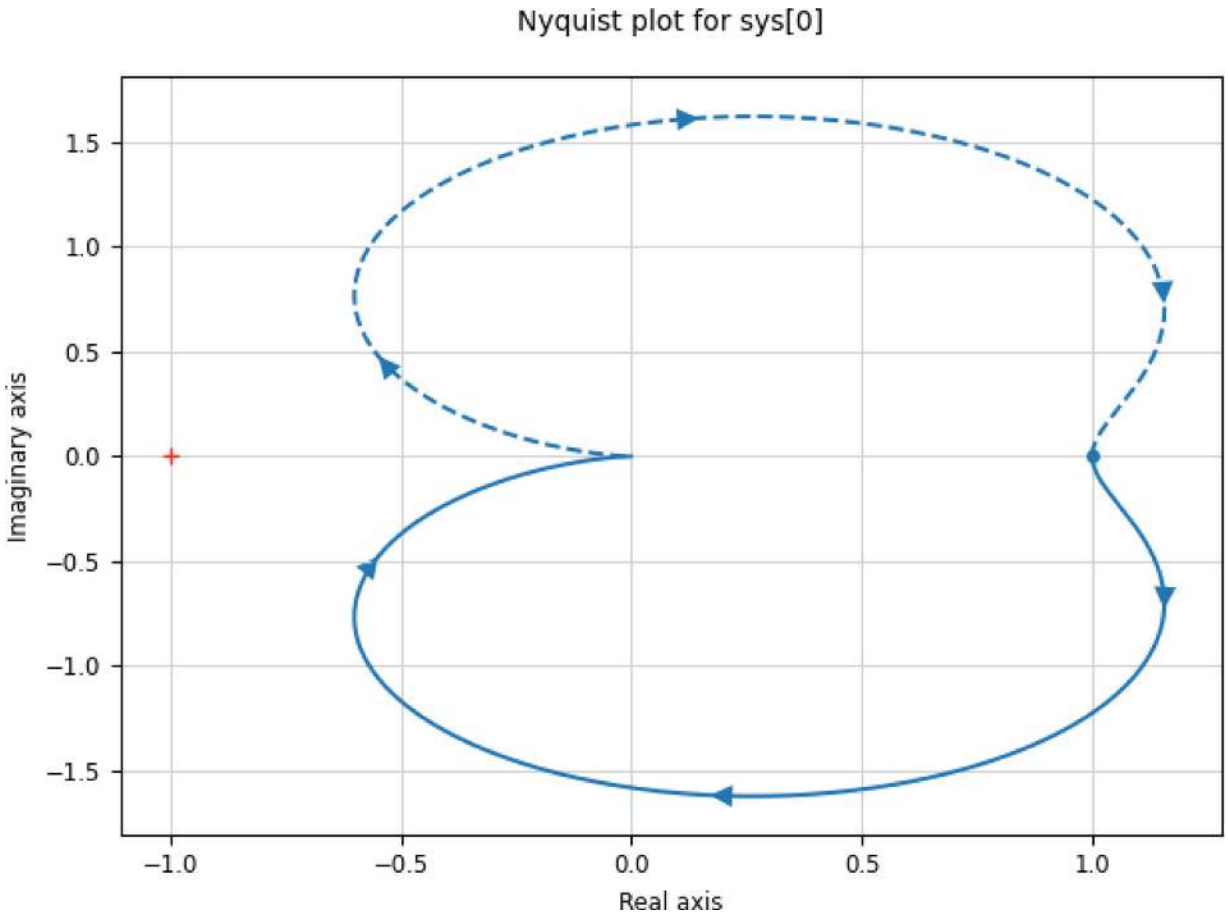


Figure 15-14 Nyquist plot of a linear system, illustrating the frequency response in the complex plane and its relation to closed-loop stability

Nichols Plot for a System

A Nichols diagram is a graphical representation of a system's frequency response, showing the gain or magnitude vs. the phase shift of the system's transfer function. The Nichols diagram can reveal various

system properties, such as gain margin, phase margin, and stability. It is therefore useful for designing controllers.

The example system is defined as a second-order transfer function (sys) with numerator [10] and denominator [1, 2, 10].

```
import control as ct
import matplotlib.pyplot as plt

sys = ct.tf([10], [1, 2, 10])
ct.nichols_plot(sys)
plt.show()
```

The ct.nichols_plot call draws the Nichols chart for sys.

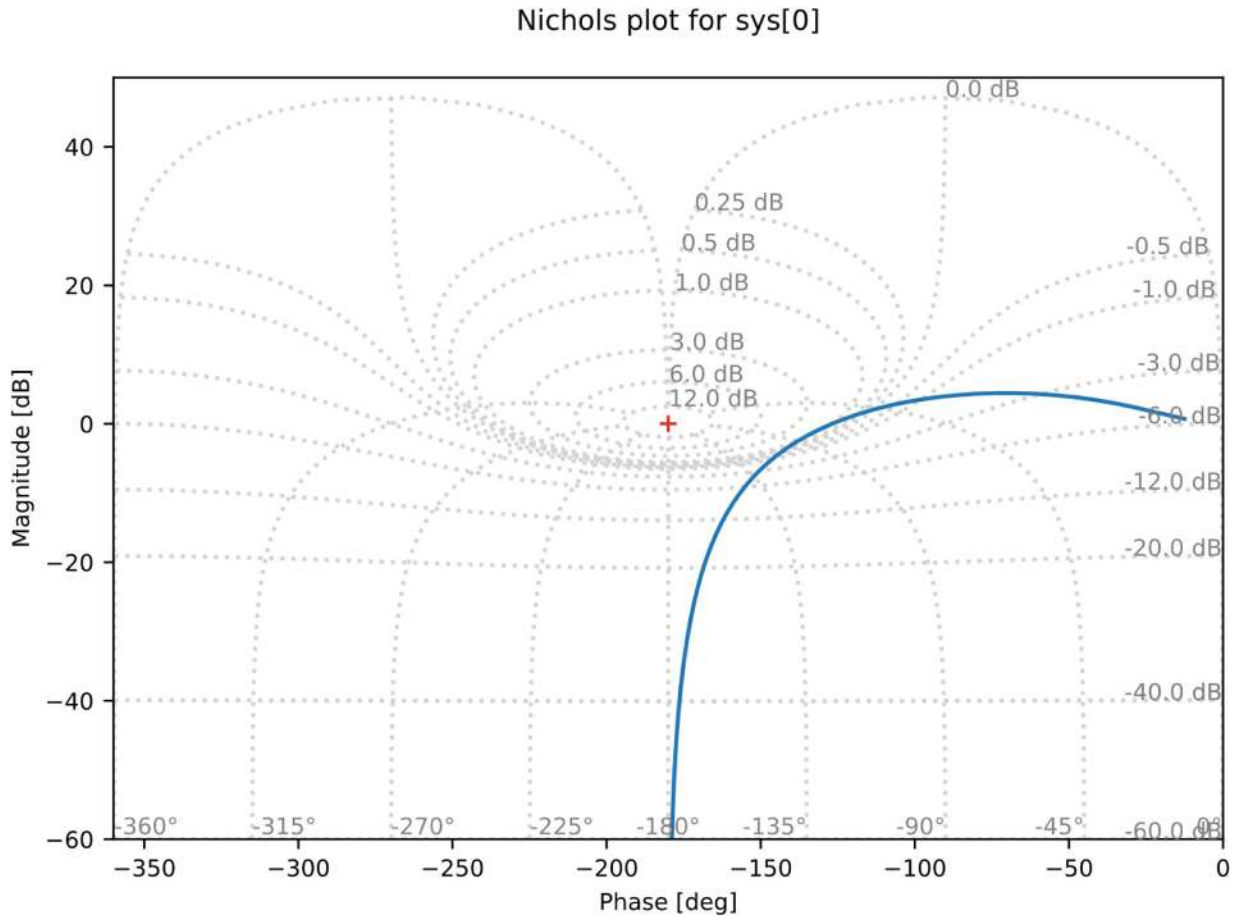


Figure 15-15 Nichols plot of a linear system, showing the relationship between magnitude and phase in the frequency domain and its implications for closed-loop stability

Find the First Frequency at Which Gain Drops by 3 dB

A 3 dB drop in gain corresponds to a reduction in amplitude by a factor of $\sqrt{2}$ (or approximately 0.707). This is often used as a reference for determining the bandwidth of a system.

The bandwidth of a system is the frequency range over which the system responds well. In this case, the ct.bandwidth function finds the first frequency at which the gain drops by 3 dB, which is a common definition of bandwidth.

As in the other examples seen so far, the system is defined as a second-order transfer function (sys) with numerator [1] and denominator [1, 2, 1], and the code is finding the bandwidth of a transfer function system (sys) using the ct.bandwidth function.

```
import control as ct

sys = ct.tf([1], [1, 2, 1])
bw = ct.bandwidth(sys)
print("Bandwidth:", bw)
```

The ct.bandwidth function returns the –3 dB bandwidth, reported below:

```
Bandwidth: 0.642290856718715
```

Return the Zero-Frequency (or DC) Gain

The DC gain is the system gain at zero frequency, or DC. It represents the ratio of the output to the input when the input is a constant or DC signal.

Calculation

The DC gain is calculated by evaluating the transfer function at s = 0. For the given system, the transfer function is

H(s) = 10 / (s + 2)

The DC gain is

H(0) = 10 / (0 + 2) = 10 / 2 = 5

In this case, the first-order model has numerator [10] and denominator [1, 2]. The ct.dcgain function computes its zero-frequency (DC) gain.

```
import control as ct

sys = ct.tf([10], [1, 2])
dc_gain = ct.dcgain(sys)
print("DC Gain:", dc_gain)
```

The ct.dcgain function returns the steady-state gain; the console output is 5.0.

```
DC Gain: 5.0
```

In this case, the DC gain is 5, which means that if the input to the system is a constant signal of amplitude 1, the output will be a constant signal of amplitude 5.

Calculate Gain and Phase Margins and Associated Crossover Frequencies

Gain and phase margins are measures of a system's stability.

- *Gain Margin (*gm*)*: The amount of gain that can be added to the system before it becomes unstable
- *Phase Margin (*pm*)*: The amount of phase shift that can be added to the system before it becomes unstable

Crossover frequencies are the frequencies at which the system's gain or phase response crosses a critical threshold.

- *Gain Crossover Frequency (*wcp*)*: The frequency at which the magnitude crosses 0 dB
- *Phase Crossover Frequency (*wcg*)*: The frequency at which the phase crosses –180 degrees

The example system is defined as a transfer function (sys) with numerator [10] and denominator [1, 2, 10]. The code then calculates the gain and phase margins, as well as the associated crossover frequencies, for a transfer function system (sys) using the ct.margin function.

```
import control as ct

sys = ct.tf([10], [1, 2, 10])
gm, pm, wcg, wcp = ct.margin(sys)
print("Gain Margin:", gm)
print("Phase Margin:", pm)
print("Phase Crossover Frequency (rad/s):", wcg)
print("Gain Crossover Frequency (rad/s):", wcp)
```

The ct.margin function returns the gain margin, phase margin, phase crossover frequency, and gain crossover frequency. All four values are printed below:

```
Gain Margin: inf
Phase Margin: 53.13010235415598
Phase Crossover Frequency (rad/s): nan
Gain Crossover Frequency (rad/s): 4.0
```

Calculate Stability Margins and Associated Crossover Frequencies

The ct.stability_margins function returns a tuple containing the following stability margins and crossover frequencies:

- *Gain Margin*: The gain that can be added to the system before it becomes unstable
- *Phase Margin*: The phase shift that can be added to the system before it becomes unstable
- *Gain Crossover Frequency*: The frequency at which the system's gain response crosses 0 dB

- *Phase Crossover Frequency*: The frequency at which the system's phase response crosses –180 degrees
- *Stability Margin*: The minimum distance from the Nyquist curve to the critical point –1
- *Stability Margin Frequency*: The frequency at which the Nyquist curve is closest to –1

The example system is a transfer function (sys) with numerator [10] and denominator [1, 2, 10].

```
import control as ct

sys = ct.tf([10], [1, 2, 10])
margins = ct.stability_margins(sys)
print("Stability Margins:", margins)
```

The six returned values are gm, pm, sm, wpc, wgc, and wms: gain margin, phase margin, stability margin, phase crossover frequency, gain crossover frequency, and the frequency associated with the stability margin.

```
Stability Margins: (inf,
np.float64(53.13010235415598),
np.float64(0.6192413689137686), nan,
np.float64(4.0), np.float64(4.921335637537115))
```

System Properties

This section examines the fundamental properties of dynamic systems that can be directly extracted from their mathematical models. It demonstrates how key characteristics such as poles, zeros, system order, and gain can be obtained using computational tools, providing insight into system behavior before performing simulations or controller design. Understanding these properties allows engineers to quickly assess stability tendencies and dynamic performance based on the system model.

Compute System Poles

The poles of a transfer function are the complex values of s that make its denominator zero. For a continuous-time LTI system, asymptotic stability requires every pole to have a strictly negative real part. A pole with a positive real part makes the system unstable; poles on the imaginary axis require separate analysis and do not imply asymptotic stability.

For sys = $1/(s^2 + 2s + 1)$, ct.poles computes the pole locations.

```
import control as ct

sys = ct.tf([1], [1, 2, 1])
p = ct.poles(sys)
print("Poles:", p)
```

The ct.poles function returns the poles of the system, which are printed to the console.

```
Poles: [-1.+0.j -1.+0.j]
```

Compute System Zeros

The zeros of a transfer function are the complex values of s that make its numerator zero. They shape transient and frequency responses and may create transmission nulls for particular exponential or sinusoidal inputs, but they are not, in general, ordinary real frequencies at which every output must vanish.

The ct.zeros(sys) call returns the zeros of the transfer function model.

```
import control as ct

sys = ct.tf([1, 1], [1, 2, 1])
z = ct.zeros(sys)
print("Zeros:", z)
```

The ct.zeros function returns the zeros of the system, which are printed to the console.

```
Zeros: [-1.+0.j]
```

Controller Design

Controller design focuses on modifying the dynamic behavior of a system in order to meet specific performance and stability requirements. By appropriately selecting controller parameters, it is possible to influence key characteristics such as response speed, damping, steady-state error, and robustness. This section introduces fundamental controller design techniques supported by the Python control library, including graphical methods and analytical approaches, which allow engineers to systematically shape the closed-loop dynamics of a system based on its mathematical model.

Root Locus Plot

A root locus plot shows how the closed-loop poles move as a scalar loop gain varies. It helps identify gain ranges that preserve stability and reveals changes in damping, response speed, and oscillatory behavior. Gain and phase margins are frequency-domain quantities and are computed separately.

The example calls ct.root_locus_plot for the third-order transfer function with numerator [1] and denominator [1, 3, 2, 0]. The zero constant term introduces a pole at the origin (an integrator).

```
import control as ct
import matplotlib.pyplot as plt

sys = ct.tf([1], [1, 3, 2, 0])
ct.root_locus_plot(sys)
plt.show()
```

The root locus graph is shown below.

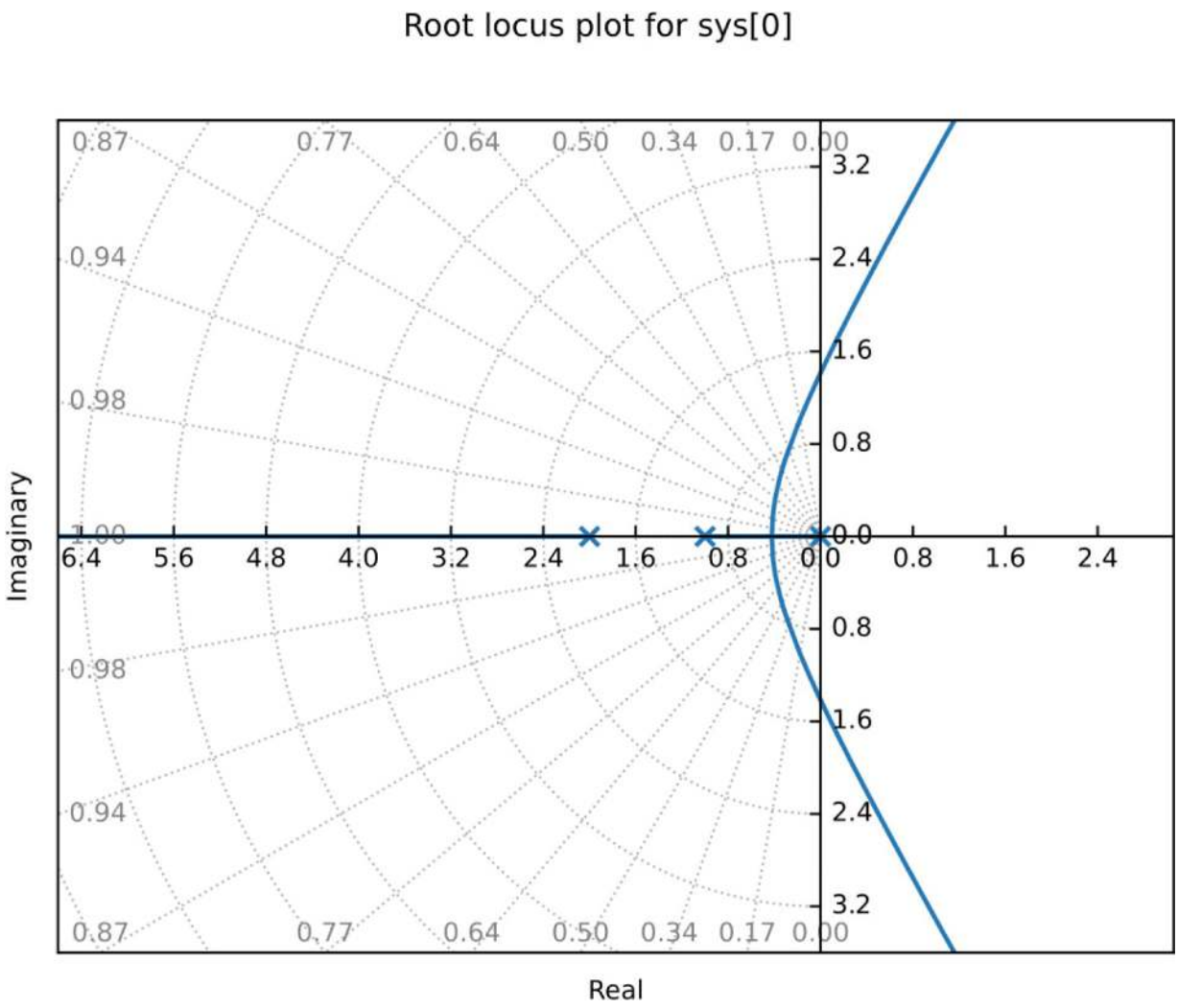


Figure 15-16 Root locus of a linear system, illustrating the trajectories of closed-loop poles as the system gain varies

Pole Placement Using Ackermann's Method

Pole placement assigns the nominal closed-loop eigenvalues of a controllable state-space model. These eigenvalues strongly influence response speed, damping, and stability, but zeros, reference scaling, actuator limits, and model uncertainty must also be checked. Ackermann's method provides a direct formula for the state-feedback gain matrix K.

The code uses Ackermann's method to design a state-based feedback controller that positions the poles of a system at desired locations. The system is defined by the state-space matrices A and B,

which represent the system dynamics and the input matrix, respectively. The desired poles are specified as [−4, −5], meaning that the closed-loop system must have poles at these locations. This will determine the stability and dynamic behavior of the system.

```
import control as ct
import numpy as np

A = np.array([[0, 1], [-2, -3]])
B = np.array([[0], [1]])
# Desired poles
poles = [-4, -5]

K = ct.place_acker(A, B, poles)
print("Feedback gain matrix K:", K)
```

The ct.place_acker function produces the feedback gain matrix K shown below:

```
Feedback gain matrix K: [18.  6.]
```

Nonlinear Systems

This section introduces the analysis of nonlinear dynamic systems using computational models that explicitly describe nonlinear relationships between system variables. Unlike linear systems, nonlinear models are defined directly through their governing equations, allowing more realistic representation of physical processes. The examples presented here demonstrate how such systems can be analyzed and simulated to observe their dynamic behavior under different operating conditions.

Find an Operating Point for a Nonlinear System

The ct.find_operating_point function is used to find an operating point for the system. An operating point is a state and input that result in a

steady-state condition, where the system's dynamics are at equilibrium. In this case, the operating point is found around the initial state [0, 0], which corresponds to the pendulum hanging straight down.

The system is created as a nonlinear system object (sys_nlsys) using the ct.nlsys function and is defined by two functions:

- pendulum_update: This function describes the dynamics of the pendulum, including the effects of gravity, damping, and an external input.
- pendulum_output: This function specifies the output of the system, which in this case is the state itself.

```
import control as ct
import numpy as np

def pendulum_update(t, x, u, params):
    g = params.get('g', 9.8)
    l = params.get('l', 1.0)
    m = params.get('m', 1.0)
    b = params.get('b', 0.1)
    return [x[1], -b / (m * l**2) * x[1] - g / l *
np.sin(x[0]) + u[0] / (m * l**2)]

def pendulum_output(t, x, u, params):
    return x

# Now we specify that there is 1 input and 2
outputs
sys_nlsys = ct.NonlinearIOSystem(
    pendulum_update, pendulum_output,
    states=2, inputs=1, outputs=2,
    name='pendulum'
)

# Find the operating point
```

```
op_point = ct.find_operating_point(sys_nlsys, [0,
0], inputs=[0])

# --- Accessing and printing data---
print("\n--- Operating Point Values ---")
print("Equilibrium States:", op_point.states)
print("Equilibrium Inputs:", op_point.inputs)
print("Equilibrium Outputs:", op_point.outputs)
```

The ct.find_operating_point function returns the operating point, which includes the state and input values that result in a steady-state condition. The operating point is printed to the console.

```
--- Operating Point Values ---
Equilibrium States: [0. 0.]
Equilibrium Inputs: [0]
Equilibrium Outputs: [0. 0.]
```

Linearize an Input/Output System at a Given State and Input

The code is linearizing a nonlinear system representing a pendulum around a given operating point.

The ct.linearizefunction is used to linearize the system around the operating point defined by the state x_op = [0, 0] and input u_op = [0]. This corresponds to the pendulum hanging straight down with no external input.

The linearization process approximates the nonlinear system dynamics with a linear system, which can be used for analysis, control design, or simulation.

```
import control as ct
import numpy as np

def pendulum_update(t, x, u, params):
    g = params.get('g', 9.8)
```

```
    l = params.get('l', 1.0)
    m = params.get('m', 1.0)
    b = params.get('b', 0.1)
    return [x[1], -b / (m * l**2) * x[1] - g / l *
np.sin(x[0]) + u[0] / (m * l**2)]

def pendulum_output(t, x, u, params):
    return x

sys_nlsys = ct.nlsys(pendulum_update,
pendulum_output, states=2, name='pendulum')
x_op = [0, 0]
u_op = [0]

sys_linearized = ct.linearize(sys_nlsys, x_op,
u_op)
print("Linearized system:\n", sys_linearized)
```

The ct.linearizefunction returns the linearized system (sys_linearized), which is printed to the console.

```
Linearized system:
 <StateSpace>: sys[0]
Inputs (1): ['u[0]']
Outputs (2): ['y[0]', 'y[1]']
States (2): ['x[0]', 'x[1]']

A = [[ 0.   1. ]
     [-9.8 -0.1]]

B = [[0.]
     [1.]]

C = [[1. 0.]
     [0. 1.]]
```

```
D = [[0.]
     [0.]]
```

Control Theory

Control theory provides the theoretical foundation for analyzing and designing systems that regulate their behavior through feedback. It establishes the principles that explain how system dynamics, feedback structures, and controller parameters influence stability, performance, and robustness. This section consolidates the key ideas introduced throughout the chapter, emphasizing the role of mathematical models and feedback in achieving desired system behavior and preparing the reader for more advanced control strategies and applications.

Calculate the Controllability Matrix

The controllability matrix is a matrix that determines whether a system is controllable, meaning whether it is possible to drive the system from any initial state to any final state using the inputs.

The system is defined by the state-space matrices:

- A: The state matrix, which describes the dynamics of the system
- B: The input matrix, which describes how the inputs affect the system

The code is calculating the controllability matrix for a linear system defined by the matrix A and matrix B.

```
import control as ct
import numpy as np

A = np.array([[0, 1], [-2, -3]])
B = np.array([[0], [1]])

# Calculate controllability matrix
ctrb_matrix = ct.ctrb(A, B)
print("Controllability Matrix:\n", ctrb_matrix)
```

The ct.ctrb function returns the controllability matrix, which is printed to the console.

```
Controllability Matrix:
 [[ 0.  1.]
 [ 1. -3.]]
```

PID Controller Design

Now that the capabilities of the control library have been explored, it is possible to proceed to the design of a PID (Proportional–Integral–Derivative) controller. A PID controller, which includes proportional, integral, and derivative actions, helps regulate system behavior to achieve desired performance goals such as stability, fast response, and minimal error. As one of the most widely used controllers in the world, it is essential to understand the design details of a PID controller using Python, which provides valuable tools and resources for this type of application.

There are three main methods commonly used to design a PID controller:

Root Locus Method

This method analyzes how the system's poles move as the controller's gains change, allowing for fine-tuning of the system's transient response and stability.

Pole Allocation Method

In this technique, nominal closed-loop pole locations are selected, and the controller parameters are calculated to reproduce the corresponding characteristic polynomial. The resulting time response must still be verified because controller zeros, reference scaling, and implementation limits also affect performance.

Frequency Response Analysis

In this method, controllers are designed from the response to sinusoidal signals at different frequencies. Bode and Nyquist diagrams are used to assess and shape stability margins, bandwidth, and robustness.

These techniques provide systematic starting points for selecting PID gains that target specified stability and performance requirements. The resulting controller must still be verified for robustness, actuator limits, noise sensitivity, and implementation effects. The table below classifies the methods to facilitate comparison.

Table

PID Design with Pole Allocation

This example code designs a Proportional–Integral–Derivative (PID) controller for a linear system using a control technique called pole allocation. The goal is to modify the system's behavior to make it faster and more stable and eliminate long-term errors. It then simulates and plots the step responses of both the original uncontrolled system and the controlled system, visually demonstrating the performance improvement achieved by the PID controller.

Simple PID Analyzer

Before proceeding with the PID controller design, it is important to establish how to analyze the results obtained from the controller design code. This analysis can be performed using software external to the Python ecosystem, such as Scilab or any other dynamic systems analysis software. It is, of course, possible to develop a complete Python application for analyzing the data obtained from the controller design that approximates dedicated simulation software. However, the complexity and size of the resulting code are beyond the scope of this chapter, limiting it to the development of applications for simplified PID controller analysis.

Therefore, we present two codes that evaluate the PID controller parameters derived from the calculations, as well as the corresponding transfer functions for the systems on which the calculations were based. Both scripts apply a step response signal to evaluate system behavior using the PID controller parameters Kp, Ki, and Kd. For ease of understanding, the scripts are divided into two applications: the first evaluates systems modeled with a transfer function, while the second deals with systems represented by state-space matrices.

These scripts are intended for basic response analysis. Results can differ from other tools when the controller form, derivative filtering, initial conditions, solver tolerances, or simulation time grids are not identical.

The examples use either an ideal PID or an explicitly filtered derivative, as indicated in each code block. Practical implementations also require saturation handling, anti-windup, sampling effects, and appropriate initialization. Differences in these details—not the programming language itself—can change the simulated response.

Using a Transfer Function

The code below simulates a control system where a PID controller is used to regulate a plant, represented by a transfer function. The user inputs the coefficients of the plant's transfer function and the PID gains. The system then calculates the step response and performance metrics like rise time and overshoot. This allows to analyze how well the PID controller is tuned for the specific plant, providing insights into stability and responsiveness. The code calculates performance metrics such as rise time and peak time and then plots the system response over time, providing insights into stability and performance.

```
# Code that uses a transfer function.
import control as ct
import matplotlib.pyplot as plt
import numpy as np

try:
```

```
    # 1. Get user input
    num = [float(n) for n in input("Enter plant
numerator coefficients (space-separated):
").split()]
    den = [float(n) for n in input("Enter plant
denominator coefficients (space-separated):
").split()]
    kp, ki, kd = map(float, input("Enter Kp, Ki,
Kd (space-separated): ").split())

    # 2. Create transfer functions and closed-loop
system
    plant_tf = ct.tf(num, den)

    # Now printing only the user-defined plant
transfer function
    print("\n--- Plant Transfer Function G(s) ---
")
    print(plant_tf)

    controller_tf = ct.tf([kd, kp, ki], [1, 0])
    closed_loop_tf = ct.feedback(controller_tf *
plant_tf, 1)

    # 3. Get step response data and performance
metrics
    # Define a longer, explicit time vector for
the simulation
    T = np.linspace(0, 40, 1000)
    info = ct.step_info(closed_loop_tf, T=T)
    t, y = ct.step_response(closed_loop_tf, T=T)

    # 4. Print results and plot
    print("\n--- Performance Metrics ---")
    for key, value in info.items():
```

```
            if isinstance(value, float):
                print(f"{key}: {value:.4f}")
            else:
                print(f"{key}: {value}")

        plt.figure(figsize=(10, 6))
        plt.plot(t, y, label='System Response')
        plt.axhline(1, color='r', linestyle='--',
label='Setpoint')
        plt.title('Step Response Analysis')
        plt.xlabel('Time (seconds)')
        plt.ylabel('Amplitude')
        plt.grid(True)
        plt.legend()
        plt.show()

except Exception as e:
        print(f"\nAn error occurred: {e}")
        print("Please check your inputs. Ensure they
are space-separated numbers.")
```

Using State-Space Matrices

Now focusing on state-space matrices, the code simulates a control system using state-space matrices, where a PID controller is designed to regulate a plant. The plant dynamics are defined by the state-space matrices A, B, C, and D, and the PID gains are specified by the user. The code creates a state-space model for the plant and the PID controller, combines them into a closed-loop system, and analyzes the step response. As with the previous code, performance metrics such as rise time and peak time are calculated, and then the system response is plotted over time.

```
# Code that uses state space matrices.
import control as ct
import matplotlib.pyplot as plt
```

```
import numpy as np

try:
    # 1. Define the plant state-space matrices and
PID gains directly in the code.
    # --- USER: MODIFY YOUR PLANT AND GAINS HERE -
--
    A_p = np.array([[0, 1], [0, -2]])
    B_p = np.array([[0], [1]])
    C_p = np.array([[1, 0]])
    D_p = np.array([[0]])

    # PID Gains
    kp, ki, kd = 40.0, 20, 25

    # --- END OF USER MODIFICATION SECTION ---
    # 2. Create the State-Space model for the
plant directly from the defined matrices
    plant_ss = ct.ss(A_p, B_p, C_p, D_p)

    print("\n--- Plant State-Space Model (A, B, C,
D) ---")
    print("A = \n", np.round(plant_ss.A, 4))
    print("B = \n", np.round(plant_ss.B, 4))
    print("C = \n", np.round(plant_ss.C, 4))
    print("D = \n", np.round(plant_ss.D, 4))

    # 3. Create a state-space model for the PID
controller directly.
    # The controller has two states: one for the
integrator and one for the
    # filtered derivative term.
    # C(s) = Kp + Ki/s + Kd*s/(Tf*s + 1)
    # A pure derivative isn't physically
realizable, so a filtered derivative
```

```
    # is used, where Tf is a small filter time
constant.
    Tf = 0.01  # Small time constant for the
derivative filter

    # State-space matrices for the controller
based on its components.
    # The state vector is x_c = [integrator_state,
derivative_filter_state]^T
    # State equation: x_c_dot = A_c * x_c + B_c *
error
    # Output equation: u = C_c * x_c + D_c * error
    A_c = np.array([[0, 0], [0, -1 / Tf]])
    B_c = np.array([[1], [1]])
    # The C and D matrices combine the
proportional, integral, and derivative actions
    C_c = np.array([[ki, -kd / (Tf ** 2)]])
    D_c = np.array([[kp + kd / Tf]])

    controller_ss = ct.ss(A_c, B_c, C_c, D_c)

    # 4. Combine the controller and plant in
state-space to create the closed-loop system
    # Open-loop system = controller connected in
series with the plant
    loop_ss = ct.series(controller_ss, plant_ss)

    # Closed-loop system with negative feedback
    closed_loop_ss = ct.feedback(loop_ss, 1)

    print("\n--- System Combination ---")
    print(f"The final closed-loop system has
{closed_loop_ss.nstates} states.")
```

```
    # 5. Get step response data and performance
metrics from the state-space model
    # Define an explicit time vector for a
detailed simulation
    T = np.linspace(0, 10, 1000)  # Adjusted time
for a potentially faster response
    info = ct.step_info(closed_loop_ss, T=T)
    t, y = ct.step_response(closed_loop_ss, T=T)

    # 6. Print results and plot
    print("\n--- Performance Metrics ---")
    for key, value in info.items():
        if isinstance(value, float):
            print(f"{key}: {value:.4f}")
        else:
            print(f"{key}: {value}")

    plt.figure(figsize=(10, 6))
    plt.plot(t, y, label='System Response')
    plt.axhline(1, color='r', linestyle='--',
label='Setpoint')
    plt.title('Step Response Analysis (State-
Space)')
    plt.xlabel('Time (seconds)')
    plt.ylabel('Amplitude')
    plt.grid(True)
    plt.legend()
    plt.show()

except Exception as e:
    print(f"\nAn error occurred: {e}")
    print("Please check your inputs.")
```

A representative console output is shown below:

```
--- Plant State-Space Model (A, B, C, D) ---
A = [[ 0.  1.]
     [ 0. -2.]]
B = [[0.]
     [1.]]
C = [[1. 0.]]
D = [[0.]]
--- System Combination ---
The final closed-loop system has 4 states.
--- Performance Metrics ---
RiseTime: 0.0601
SettlingTime: 0.1401
SettlingMin: 0.9043
SettlingMax: 1.0149
Overshoot: 1.4943
Undershoot: 0.0000
Peak: 1.0149
PeakTime: 1.6316
SteadyStateValue: 1.0000
```

PID Design by Pole Allocation with a Transfer Function

The pole allocation technique, also known as pole placement, is a method used in control system design to determine the gains of a controller such as a PID controller by assigning the poles of the closed-loop system to specific locations in the complex plane.

In a transfer function, the poles are the values of *s* that make the denominator equal to zero. These poles govern the stability and dynamic behavior of the system.

First-Order Systems

For a first-order plant, adding a PI controller introduces an integrator, so the unity-feedback closed-loop characteristic polynomial is second-order. Two desired closed-loop poles are therefore specified and used to determine Kp and Ki.

$$\mathrm{G(s)} = \frac{1}{\mathrm{s} + \mathrm{a}}$$

Second-Order Systems

For a second-order plant, an ideal PID controller introduces an integrator, producing a third-order unity-feedback characteristic polynomial. Three desired closed-loop poles are therefore specified and matched to determine Kp, Ki, and Kd.

$$\mathrm{G}(s) = \frac{1}{s^2 + 2s + a}$$

By assigning poles to specific locations, it is possible to achieve desired performance characteristics, such as

- *Stability*: Poles in the left half of the complex plane ensure a stable system.
- *Damping*: Poles with negative real parts and adequate damping ratios ensure oscillations decay quickly.
- *Settling Time*: Poles located further to the left in the complex plane result in faster settling times.

In the transfer function approach, design begins by defining performance targets such as settling time, damping, overshoot, and disturbance rejection. These targets guide the choice of desired closed-loop poles, normally in the open left half-plane for a continuous-time system.

With the desired poles defined, PID gains can be obtained by matching the coefficients of the closed-loop characteristic polynomial. This matches the nominal denominator for the assumed plant and controller structure; controller zeros and implementation effects still influence the actual response.

See the block diagram below, which outlines the algorithmic steps for obtaining controller gains using the transfer function pole allocation technique.

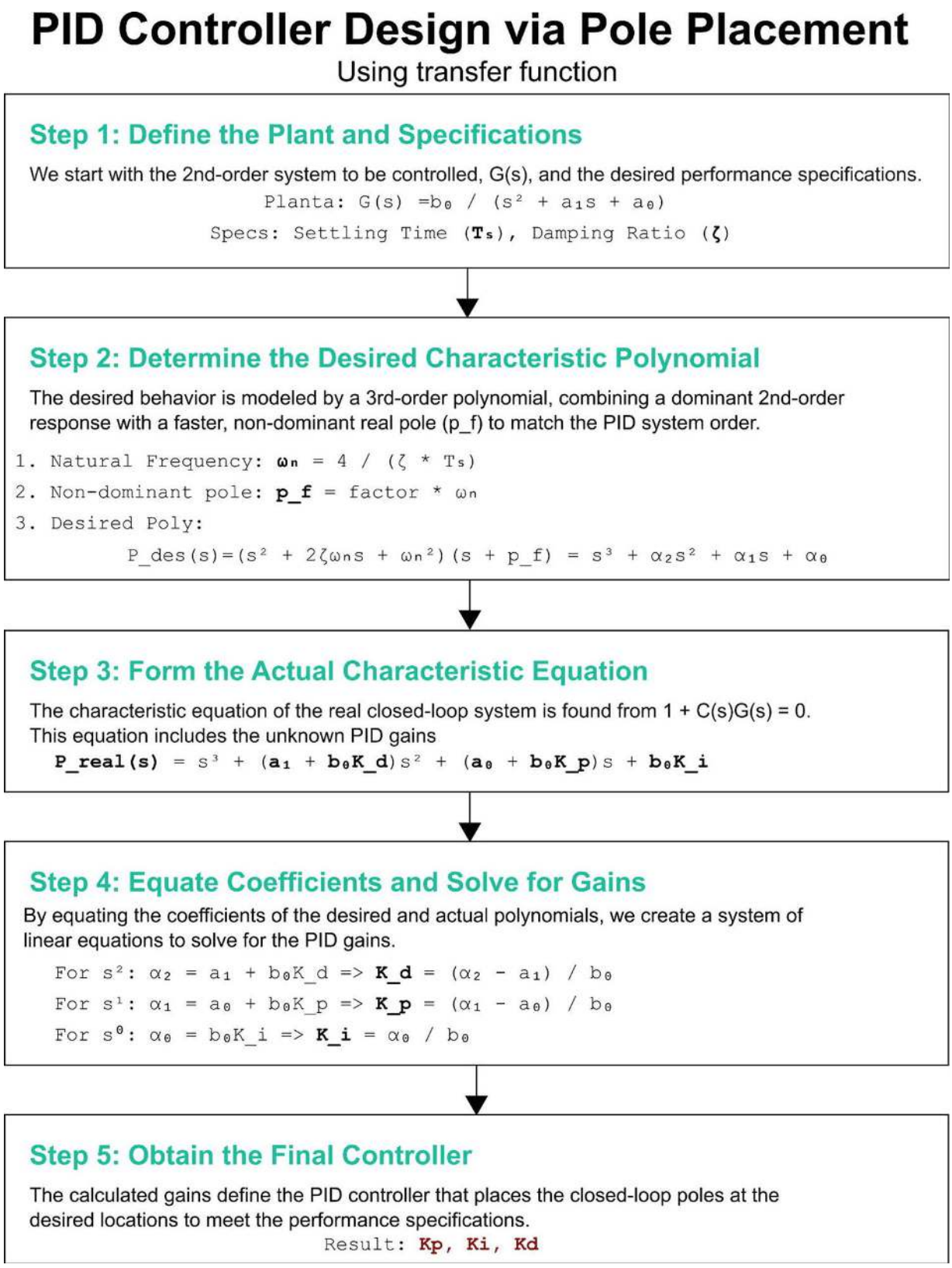


Figure 15-17 Step-by-step procedure for PID controller design using pole placement based on time-domain performance specifications

As an example, we will design a PID controller by pole allocation with a second-order transfer function. The system will have the following transfer function:

$$\mathrm{G}(s) = \frac{1}{s^2 + 2s + 5}$$

```
# Designing a PID controller for a 2nd order plant
using pole allocation.
import numpy as np
import control as ct
import matplotlib.pyplot as plt
# --- Inputs ---
# Plant: G(s) = b0 / (s^2 + a1*s + a0)
a1, a0, b0 = 2.0, 5.0, 1.0
# Design: Ts (settling time), zeta (damping ratio)
Ts, zeta, pole_factor = 2.0, 0.7, 5.0

# --- Direct Calculations ---
# Desired 3rd-order polynomial
wn = 4 / (zeta * Ts)
pf = pole_factor * wn
desired_poly = np.polymul([1, 2 * zeta * wn,
wn**2], [1, pf])

# PID gains by coefficient matching
# desired_poly = [1, alpha2, alpha1, alpha0]
Kd = (desired_poly[1] - a1) / b0
Kp = (desired_poly[2] - a0) / b0
Ki = desired_poly[3] / b0

# --- Results ---
print(f"Calculated PID Gains:\n Kp = {Kp:.4f}\n Ki
= {Ki:.4f}\n Kd = {Kd:.4f}\n")
```

```
# Transfer Functions
plant = ct.tf(b0, [1, a1, a0])
controller = ct.tf([Kd, Kp, Ki], [1, 0])
closed_loop = ct.feedback(controller * plant, 1)

print("--- Transfer Functions ---")
print("Plant G(s):\n", plant)
print("Controller C(s):\n", controller)
print("Closed-Loop System T(s):\n", closed_loop)
```

The first part of the code defines the system you want to control and the ideal behavior you want to achieve.

```
# --- Inputs ---
# Plant: G(s) = b0 / (s^2 + a1*s + a0)
a1, a0, b0 = 2.0, 5.0, 1.0
# Design: Ts (settling time), zeta (damping ratio)
Ts, zeta, pole_factor = 2.0, 0.7, 5.0
```

The Plant: The variables a1, a0, and b0 describe your system's natural behavior. This is the "uncontrolled" system, modeled by the transfer function G(s)=s2+a1s+a0b0.

The Desired Performance: You specify how you want the final, controlled system to behave using two key parameters:

- *Settling Time (*Ts*)*: How quickly the system should stop oscillating and settle at its new target after a change. A smaller Ts means a faster response.
- *Damping Ratio (*zeta*)*: This controls the damping of the dominant pole pair. A value near 0.7 is a common nominal target that usually provides moderate overshoot and a relatively fast response.

From Ts and zeta, the code constructs a nominal desired characteristic polynomial. The familiar relation wn = 4/(zeta Ts) is the usual approximate 2% settling-time relation for a dominant

underdamped second-order pair; the additional real pole is placed farther left by pole_factor.

```
# --- Direct Calculations ---
# Desired 3rd-order polynomial
wn = 4 / (zeta * Ts)
pf = pole_factor * wn
desired_poly = np.polymul([1, 2 * zeta * wn,
wn**2], [1, pf])
```

The code then creates a general mathematical model for your plant combined with an untuned PID controller. This model has the unknown gains Kp, Ki, and Kd in it.

The algorithm matches the coefficients of the nominal closed-loop denominator and solves for Kp, Ki, and Kd. The denominator poles are therefore assigned for the assumed model, but the closed-loop numerator contains controller zeros, so the complete step response must still be simulated and checked.

```
# PID gains by coefficient matching
# desired_poly = [1, alpha2, alpha1, alpha0]
Kd = (desired_poly[1] - a1) / b0
Kp = (desired_poly[2] - a0) / b0
Ki = desired_poly[3] / b0
```

The final output of the code prints the calculated PID gains and displays the transfer functions for

1. The plant (G(s))
2. The PID controller (C(s))
3. The closed-loop system (T(s))

The transfer functions are represented using the control library, and the results are printed in a readable format. The closed-loop system is created by connecting the controller and plant in a feedback loop.

```
# --- Results ---
print(f"Calculated PID Gains:\n Kp = {Kp:.4f}\n Ki
= {Ki:.4f}\n Kd = {Kd:.4f}\n")

# Transfer Functions
plant = ct.tf(b0, [1, a1, a0])
controller = ct.tf([Kd, Kp, Ki], [1, 0])
closed_loop = ct.feedback(controller * plant, 1)

print("--- Transfer Functions ---")
print("Plant G(s):\n", plant)
print("Controller C(s):\n", controller)
print("Closed-Loop System T(s):\n", closed_loop)
```

The result printed on the screen is

```
Calculated PID Gains:
Kp = 60.3061
Ki = 116.6181
Kd = 16.2857

--- Transfer Functions ---
Plant G(s):
 <TransferFunction>: sys[0]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

        1
  -------------
  s^2 + 2 s + 5
Controller C(s):
```

```
 <TransferFunction>: sys[1]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

  16.29 s^2 + 60.31 s + 116.6
  ---------------------------
               s
Closed-Loop System T(s):
 <TransferFunction>: sys[4]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

     16.29 s^2 + 60.31 s + 116.6
  ---------------------------------
  s^3 + 18.29 s^2 + 65.31 s + 116.6
```

For a first-order system, see the code below.

Here's for a simpler first-order plant control design, which is a system modeled by the transfer function

$$\mathrm{G(s)} = \frac{b_0}{\mathrm{s} + a_0}$$

For this first-order plant, a PI controller is sufficient for the illustrated pole assignment objective. Combining the plant with a PI controller gives a second-order closed-loop characteristic polynomial, so Kp and Ki are obtained by matching two coefficients and Kd is set to zero.

```
#Designing a PID controller for a 1st order plant
using pole allocation.
import numpy as np
import control as ct
import matplotlib.pyplot as plt
```

```
# --- Inputs ---
# Plant: G(s) = b0 / (s + a0)
a0, b0 = 2.0, 5.0
# Design: Ts (settling time), zeta (damping ratio)
Ts, zeta = 2.0, 0.7

# 1. Define the desired 2nd-order polynomial based
on performance specs
wn = 4 / (zeta * Ts)
desired_poly = np.array([1, 2 * zeta * wn, wn**2])

# 2. Match coefficients to solve for the PI gains
(Kp, Ki)
# The actual characteristic polynomial is: s^2 +
(a0 + b0*Kp)s + b0*Ki = 0
Kp = (desired_poly[1] - a0) / b0
Ki = desired_poly[2] / b0
Kd = 0.0  # No derivative action is needed for
this design

# --- Results ---
print(f"Calculated PI Gains:\n Kp = {Kp:.4f}\n Ki
= {Ki:.4f}\n")
# Transfer Functions
plant = ct.tf(b0, [1, a0])
controller = ct.tf([Kp, Ki], [1, 0])  # PI
Controller
closed_loop = ct.feedback(controller * plant, 1)

print("--- Transfer Functions ---")
print("Plant G(s):\n", plant)
print("Controller C(s):\n", controller)
print("Closed-Loop System T(s):\n", closed_loop)
```

The result of the code for the first-order system is

```
Calculated PI Gains:
Kp = 0.4000
Ki = 1.6327

--- Transfer Functions ---
Plant G(s):
 <TransferFunction>: sys[0]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

    5
  -----
  s + 2
Controller C(s):
 <TransferFunction>: sys[1]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

  0.4 s + 1.633
  -------------
        s
Closed-Loop System T(s):
 <TransferFunction>: sys[4]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

     2 s + 8.163
  ----------------
  s^2 + 4 s + 8.163
```

PID Design by Pole Allocation with State Space

The second approach to pole allocation is based on the state-space representation of the system. In this framework, specialized methods such as Ackermann's formula or the Bass–Gura formula are commonly used to compute the required controller gains. For this example, we will

employ Ackermann's formula, which is conveniently available as a built-in function in the control library. The main difference between these two design methods is that the transfer function method works with the system's external input–output behavior, while Ackermann's formula works with its internal state model.

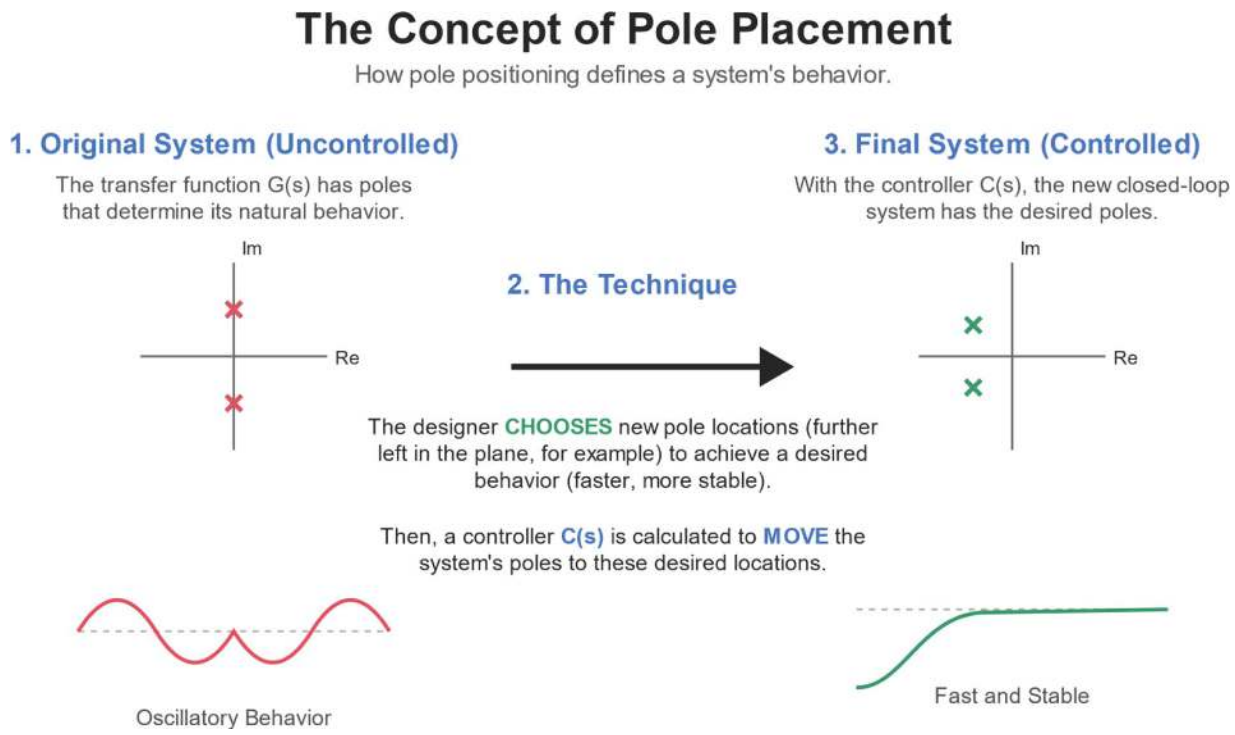


Figure 15-18 Illustration of the effect of pole placement on system dynamics, comparing the uncontrolled system behavior with the controlled closed-loop response obtained by appropriately selecting the controller C(s)

To understand the process using Ackermann's method, see the flow diagram below.

I-PD Design via Augmented-State Pole Placement

1. Define the plant

Use the position-velocity model x_dot = A x + B u and y = C x.

```
A=[[0,1],[0,-2]], B=[[0],[1]], C=[[1,0]]
```

2. Add integral tracking action

Define xi_dot = r - y. The reference is added separately when the closed-loop model is formed.

```
A_aug=[[A,0],[-C,0]], B_aug=[[B],[0]]
```

3. Assign the augmented poles

The augmented system has three states, so three poles are specified.

```
poles = [-1.0, -1.5, -0.8]
K_aug = place_acker(A_aug, B_aug, poles)
```

4. Interpret this specific realization

Because x1 is position and x2 is velocity, the law has an I-PD form.

```
u = -Kp*y - Kd*dy/dt + Ki*xi
Kp=3.5, Kd=1.3, Ki=1.2
```

5. Form the reference model

Use A_cl = A_aug - B_aug K_aug and let r enter the integrator state.

```
B_ref=[[0],[0],[1]]
poles(A_cl)={-1.5,-1.0,-0.8}
```

6. Simulate and verify

The complete closed-loop response is checked rather than inferred from pole locations alone.

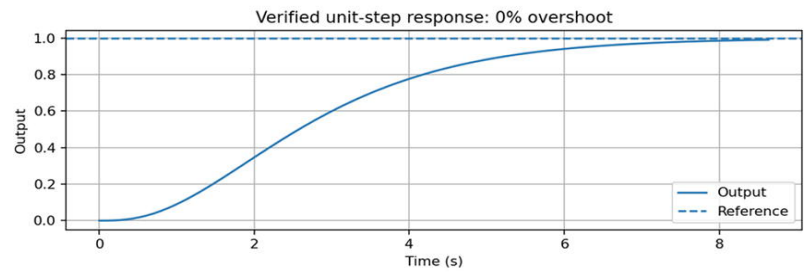


Figure 15-19 Step-by-step procedure for PID controller design via pole placement using an augmented state-space model with integral action

The code defines the plant, augments it with the integral state xi = integral(r - y) dt, and uses ct.place_acker to place the poles of the augmented system. For this position–velocity realization, the resulting law has an I-PD form: proportional and derivative feedback act on the measured output, while integral action acts on tracking error. The code then forms the closed-loop reference model and simulates its unit-step response.

See below the code and a detailed explanation of each part of the code:

```
# PID design - Pole Placement using Ackermann's
method for a second order system.
import numpy as np
import control as ct
import matplotlib.pyplot as plt

# 1. Define the original system (Open-Loop)
A = np.array([[0, 1], [0, -2]])
B = np.array([[0], [1]])
C = np.array([[1, 0]])
D = np.array([[0]])
n = A.shape[0]  # Number of states
sys_ol = ct.ss(A, B, C, D)

# 2. Build the augmented system for integral
control
# This step is ESSENTIAL for creating a PID
controller.
# It adds a new state xi whose derivative is the
tracking error (r - y).
A_aug = np.block([[A, np.zeros((n, 1))], [-C, 0]])
B_aug = np.block([[B], [0]])
```

```
C_aug = np.block([[C, 0]])

# 3. Define desired poles for the augmented system
and compute gain
# We need n+1 poles: n for the system dynamics and
1 for the integrator.
desired_poles = [-1.0, -1.5, -0.8]
K_aug = ct.place_acker(A_aug, B_aug,
desired_poles)
K_aug = np.asarray(K_aug).reshape(-1)

# 4. Extract and print the equivalent PID gains
from the feedback matrix K_aug
# The gain matrix K_aug now has 3 elements,
allowing us to find Ki.
# K_aug = [Kp, Kd, -Ki] for x_aug = [position,
velocity, xi].
Kp = K_aug[0]
Kd = K_aug[1]
Ki = -K_aug[2]

print("Pole Placement-based PID Controller
Gains:")
print(f"  Kp (Proportional): {Kp:.4f}")
print(f"  Kd (Derivative):   {Kd:.4f}")
print(f"  Ki (Integral):     {Ki:.4f}")
# 5. Form and simulate the closed-loop I-PD servo
A_cl = A_aug - B_aug @ K_aug.reshape(1, -1)
# The reference enters only the integrator state:
dxi/dt = r - y
B_ref = np.vstack((np.zeros((n, 1)), [[1.0]]))
sys_cl = ct.ss(A_cl, B_ref, C_aug, 0)
T_cl, y_cl = ct.step_response(sys_cl)
info_cl = ct.step_info(sys_cl)
```

```
print("Closed-loop poles:",
np.round(ct.poles(sys_cl), 4))
print(f"Closed-loop overshoot:
{info_cl['Overshoot']:.2f}%")
plt.plot(T_cl, y_cl, label="Closed-loop output")
plt.axhline(1, linestyle="--", label="Reference")
plt.xlabel("Time (s)")
plt.ylabel("Output")
plt.grid(True)
plt.legend()
plt.show()
```

Define the Original System with Open-Loop

This section defines the system we want to control using a state-space representation. A linear time–invariant (LTI) system represented by a state-space equation is composed of four main matrices:

- *A (State Matrix)*: Captures the system's internal behavior and dynamics
- *B (Input Matrix)*: Defines how the control inputs influence the system's states
- *C (Output Matrix)*: Specifies which states can be measured or observed
- *D (Feedforward Matrix)*: Represents any direct relationship between inputs and outputs, often zero in many systems

The command ct.ss(A, B, C, D) uses the control library to create a state-space system object named sys_ol (for "open-loop").

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -2 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 \end{bmatrix}$$

```
# 1. Define the original system (Open-Loop)
A = np.array([[0, 1], [0, -2]])
B = np.array([[0], [1]])
C = np.array([[1, 0]])
D = np.array([[0]])
n = A.shape[0]  # Number of states
sys_ol = ct.ss(A, B, C, D)
```

Build the Augmented System for Integral Control

Integral action is introduced with xi = integral(r - y) dt, so dxi/dt = r - y = -Cx + r. The matrices A_aug and B_aug describe the autonomous plant-plus-integrator dynamics used for pole placement; the reference input is added separately when the closed-loop model is formed.

```
# 2. Build the augmented system for integral
control
A_aug = np.block([[A, np.zeros((n, 1))], [-C, 0]])
B_aug = np.block([[B], [0]])
C_aug = np.block([[C, 0]])
```

Define Desired Poles and Compute Gain

```
# 3. Define desired poles for the system and
compute gain
desired_poles = [-1.0, -1.5, -0.8]
K_aug = ct.place_acker(A_aug, B_aug,
desired_poles)
```

- Poles are the roots of the characteristic equation and strongly influence stability, speed, and oscillation. Moving dominant poles farther left generally produces a faster nominal response, but it can increase control effort and noise sensitivity; the complete closed-loop model must be checked.
- *Desired Poles*: A set of predefined locations for the controlled system's poles. For a system with three states (e.g., two original

states + one integral state), three pole locations must be specified to achieve the desired performance.
- Ackermann's method (ct.place_acker) computes K_aug so that A_aug - B_aug K_aug has the specified eigenvalues. The augmented pair must be controllable; the function does not by itself establish equivalence with a conventional PID controller.

Extract and Print the Equivalent PID Gains

For this specific canonical plant, x1 is position, x2 is velocity, and xi integrates r - y. The feedback law u = -k1 x1 - k2 x2 - k3 xi can therefore be written as the I-PD form u = -Kp y - Kd dy/dt + Ki xi, with Kp = k1, Kd = k2, and Ki = -k3. This mapping is not valid for an arbitrary state-space realization whose states lack those physical meanings.

- *Kp (Proportional Gain)*: Proportional to the current error
- *Kd (Derivative Gain)*: Proportional to the rate of change of error (acts like a damper)
- *Ki (Integral Gain)*: Proportional to the accumulated error over time (eliminates steady-state error)

The sign Ki = -k3 follows from defining xi as the integral of r - y while using u = -K_aug x_aug. A parallel PID acting on the full error would additionally contain the proportional setpoint term Kp r; this changes the reference numerator but not the assigned poles.

```
# 4. Extract and print the equivalent PID gains
Kp = K_aug[0]
Kd = K_aug[1]
Ki = -K_aug[2]
```

The result of the code for the second-order system using Ackermann's method is

```
Pole Placement-based PID Controller Gains:
  Kp (Proportional): 3.5000
  Kd (Derivative):   1.3000
```

```
  Ki (Integral):     1.2000
Closed-loop poles: [-1.5+0.j -1.0+0.j -0.8+0.j]
Closed-loop overshoot: 0.00%
```

The following is an example of PI design for a first-order system, with pole placement using Ackermann's method.

A first-order plant can be augmented in the same way. The resulting feedback is an I-P realization of PI control: u = -Kp y + Ki xi with dxi/dt = r - y. Adding Kp r gives the usual parallel-PI reference path without changing the assigned poles.

Here are the matrices used:

A=[−2], B=[1], C=[1], D=[0]

The fundamental difference in the code lies in the definition of the system's order, which directly impacts the controller's complexity. A first-order system is defined with a single state (n=1), resulting in a 1 × 1 matrix A, and when augmented for integral control, it requires placing two poles to determine the Kp and Ki gains for a PI controller.

```
# PI design for a first-order system - Pole
Placement using Ackermann's method.
import numpy as np
import control as ct
import matplotlib.pyplot as plt

# 1. Define the original first-order system (Open-
Loop)
# Example system: x_dot = -2*x + u, y = x
A = np.array([[-2]])
B = np.array([[1]])
C = np.array([[1]])
D = np.array([[0]])
n = A.shape[0]  # Number of states (n=1)
sys_ol = ct.ss(A, B, C, D)
```

```
# 2. Build the augmented system for integral
control
# This step is ESSENTIAL for creating a PI
controller.
# It adds a new state xi whose derivative is the
tracking error (r - y).
# The new augmented state vector is [x,
integral_of_error].
A_aug = np.block([[A, np.zeros((n, 1))], [-C, 0]])
B_aug = np.block([[B], [0]])
C_aug = np.block([[C, 0]])

# 3. Define desired poles for the augmented system
and compute gain
# We need n+1 = 2 poles: n=1 for the system
dynamics and 1 for the integrator.
desired_poles = [-1.5, -2.5]
K_aug = ct.place_acker(A_aug, B_aug,
desired_poles)

# 4. Extract and print the equivalent PI gains
from the feedback matrix K_aug
# The gain matrix K_aug now has 2 elements,
allowing us to find Ki.
# K_aug = [Kp, -Ki] for x_aug = [x, xi], with
dxi/dt = r - y.
# We flatten the matrix to a 1D array for easier
access.
K_gains = K_aug.flatten()
Kp = K_gains[0]
Ki = -K_gains[1]

print("Pole Placement-based PI Controller Gains:")
print(f"Kp (Proportional): {Kp:.4f}")
print(f"  Ki (Integral):     {Ki:.4f}")
```

The result is

```
Pole Placement-based PI Controller Gains:
  Kp (Proportional): 2.0000
  Ki (Integral):     3.7500
```

PID Design with the Root Locus Method

The root locus shows how closed-loop poles move as one scalar loop gain varies after the controller's poles and zeros have been selected. It can guide the gain choice, but it does not independently determine all three PID gains or guarantee a specified overshoot; the complete closed-loop response must be verified.

Before studying the code, see the illustration below to get an overview of how the code executes its steps.

PID Design via Root Locus

1. Define plant and controller structure

G(s)=1/[s(s+2)] and C_struct(s)=(s+1)(s+2)/s.

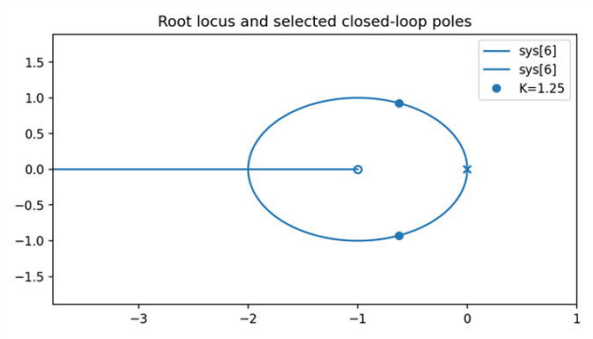
2. Reduce the nominal input-output model

Call minreal explicitly; the library does not automatically cancel the pole-zero pair at -2. Exact cancellation is sensitive to uncertainty.

```
L(s)=(s+1)/s^2
```

3. Select and verify a gain

K=1.25 is stable, but the complete response has about 26.8% overshoot. The gain is illustrative, not automatically optimal.



4. Recover ideal parallel-form gains

Compare K*C_struct(s) with (Kd s^2 + Kp s + Ki)/s.

```
Kd=1.25, Kp=3.75, Ki=2.50
```

5. Validate before implementation

Check poles, zeros, time response, robustness, derivative filtering, saturation, and anti-windup.



Figure 15-20 Step-by-step procedure for PID controller design using the root locus method, including controller zero placement, gain selection, and closed-loop performance validation

This code designs a PID controller for a dynamic system using a powerful control theory technique called the root locus method.

```
# PID design for a second-order system – Root
Locus.
import numpy as np
import control as ct
import matplotlib.pyplot as plt

# 1. Define Plant and Controller Structure
# Plant: G(s) = 1 / (s^2 + 2s)
plant = ct.tf([1], [1, 2, 0])

# Controller structure: C(s) = (s+1)(s+2)/s
# The pole-zero pair at s = -2 is reduced
explicitly for this nominal model
controller_structure = ct.tf([1, 3, 2], [1, 0])

# 2. Open-Loop System
open_loop_system = ct.minreal(controller_structure
* plant, verbose=False)
print("Open-Loop System L(s) = G(s)C(s):")
print(open_loop_system)  # Explicit minimal
realization

# 3. Root Locus and Gain Selection
plt.figure(figsize=(10, 7))
ct.root_locus_plot(open_loop_system, grid=True)
plt.title(r'Root Locus for $L(s) = \frac{K(s+1)}
{s^2}$', fontsize=16)
plt.xlabel("Real Axis")
plt.ylabel("Imaginary Axis")
```

```
plt.show()

# Choose gain K based on Root Locus for desired
damping
gain_K = 1.25
closed_loop = ct.feedback(gain_K *
open_loop_system, 1)
closed_loop_poles = ct.poles(closed_loop)
closed_loop_info = ct.step_info(closed_loop)
print("Closed-loop poles:",
np.round(closed_loop_poles, 4))
print(f"Closed-loop overshoot:
{closed_loop_info['Overshoot']:.2f}%")

# 4. Final PID Controller Parameters
final_controller = gain_K * controller_structure

# Extract numerator coefficients: [Kd, Kp, Ki]
Kd, Kp, Ki = final_controller.num[0][0]

print("\n----------------------------------------
--")
print(f"PID Controller Parameters (with K =
{gain_K}):")
print(f"  Kp (Proportional): {Kp:.4f}")
print(f"  Ki (Integral):     {Ki:.4f}")
print(f"  Kd (Derivative):   {Kd:.4f}")
print("----------------------------------------
")
```

Running this block displays the root locus and prints the closed-loop poles and step response overshoot for the selected gain.

Define Plant and Controller Structure

```
# Plant: G(s) = 1 / (s^2 + 2s)
plant = ct.tf([1], [1, 2, 0])

# Controller structure: C(s) = (s+1)(s+2)/s
controller_structure = ct.tf([1, 3, 2], [1, 0])
```

This section defines the mathematical models for the system and the controller.

```
plant = ct.tf([1], [1, 2, 0])
```

This defines the system to be controlled, called the plant. Its transfer function is $G(s) = 1/(s^2 + 2s)$. The ct.tf function takes the numerator [1] and denominator [1, 2, 0] polynomial coefficients.

```
controller_structure = ct.tf([1, 3, 2], [1, 0])
```

This defines the basic PID controller structure, $C_struct(s) = (s^2 + 3s + 2)/s$. A key design choice is the zero at s = -2, which cancels the plant pole at s = -2. This pole–zero cancellation simplifies the control problem.

Open-Loop System

```
open_loop_system = ct.minreal(controller_structure
* plant, verbose=False)
print("Open-Loop System L(s) = G(s)C(s):")
print(open_loop_system)
```

The product contains a nominal pole–zero pair at s = -2. The library does not cancel it automatically, so ct.minreal is called explicitly to obtain the reduced input–output model L(s) = (s + 1)/s^2. Exact cancellation should be used cautiously because model uncertainty can leave a residual mode.

Root Locus and Gain Selection

```
plt.figure(figsize=(10, 7))
ct.root_locus_plot(open_loop_system, grid=True)
plt.title(r'Root Locus for $L(s) = \frac{K(s+1)}
{s^2}$', fontsize=16)
plt.xlabel("Real Axis")
plt.ylabel("Imaginary Axis")
plt.show()

# Choose gain K based on Root Locus for desired
damping
gain_K = 1.25
```

This is the core design step.

```
ct.root_locus_plot(open_loop_system, grid=True)
```

The function generates the **root locus plot**. This is a graphical tool showing how the poles of the final, controlled system will move as a gain K is varied.

```
plt.show()
```

The plot is used to select a stable gain and estimate damping. The resulting closed-loop poles and step response metrics must then be computed because controller zeros can significantly alter overshoot.

```
gain_K = 1.25
```

The illustrative choice K = 1.25 gives closed-loop poles near -0.625 +/- 0.927j. It is stable, but its approximately 26.8% overshoot shows why the selected gain must be verified rather than described as automatically optimal.

Final PID Controller Parameters

```
final_controller = gain_K * controller_structure
```

```
# Extract numerator coefficients: [Kd, Kp, Ki]
Kd, Kp, Ki = final_controller.num[0][0]

print("\n----------------------------------------
--")
print(f"PID Controller Parameters (with K =
{gain_K}):")
print(f"  Kp (Proportional): {Kp:.4f}")
print(f"  Ki (Integral):     {Ki:.4f}")
print(f"  Kd (Derivative):   {Kd:.4f}")
print("----------------------------------------
")
```

This final block calculates and displays the PID controller gains.

```
final_controller = gain_K * controller_structure
```

The chosen gain K is multiplied with the initial controller structure to get the final controller transfer function.

```
Kd, Kp, Ki = final_controller.num[0][0]
```

By comparing the numerator coefficients of the final controller with the standard form of the PID controller, the code cleverly extracts the three PID gains.

The printed Kp, Ki, and Kd reproduce the ideal parallel-form controller used in this simulation. A practical implementation also requires derivative filtering, saturation handling, anti-windup, sampling, and robustness checks.

The result of the code is

```
Open-Loop System L(s) = G(s)C(s):
<TransferFunction>: sys[2]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']
```

```
  s + 1
  ------------
   s^2

----------------------------------------
PID Controller C(s) = Kp + Ki/s + Kd*s
Note: This is a PID controller in 'parallel' form.
Based on C(s) = (1.2500s^2 + 3.7500s + 2.5000)/s
----------------------------------------
  Kp (Proportional): 3.7500
  Ki (Integral):     2.5000
  Kd (Derivative):   1.2500
Closed-loop poles: [-0.625+0.9270j -0.625-0.9270j]
Closed-loop overshoot: 26.76%
----------------------------------------
```

For a first-order plant, a PI zero can be placed to shape the locus without relying on exact pole–zero cancellation. The example below uses G(s) = 1/(s + 2), places the controller zero at s = -1, and selects K = 2 from the stable root locus.

```
# PI design for a first-order system – Root Locus.
import numpy as np
import control as ct
import matplotlib.pyplot as plt

# 1. Definitions
plant = ct.tf([1], [1, 2])
# PI controller with a zero at s = -1 to shape the
locus
pi_controller_structure = ct.tf([1, 1], [1, 0])  #
C(s) = (s + 1)/s

# 2. Open-Loop System
```

```
pi_open_loop_system = pi_controller_structure *
plant
print("Open-Loop System with PI L(s):")
print(pi_open_loop_system)  # L(s) = (s + 1)/(s(s
+ 2))

# 3. Root Locus and Gain
plt.figure(figsize=(8, 6))
ct.root_locus_plot(pi_open_loop_system, grid=True)
plt.title('Root Locus for L(s) = K(s+1)/(s(s+2))',
fontsize=16)
plt.show()

gain_K = 2.0  # Selected from the stable root
locus
closed_loop_pi = ct.feedback(gain_K *
pi_open_loop_system, 1)
print("Closed-loop poles:",
np.round(ct.poles(closed_loop_pi), 4))

# 4. PI Parameters
final_pi_controller = gain_K *
pi_controller_structure
# Extract numerator coefficients: [Kp, Ki]
Kp, Ki = final_pi_controller.num[0][0]

print("\n-------------------------------------")
print(f"PI Controller Parameters (with K =
{gain_K}):")
print(f"  Kp (Proportional): {Kp:.4f}")
print(f"  Ki (Integral):     {Ki:.4f}")
print("-------------------------------------")
```

The result printed on the console is

```
Open-Loop System with PI L(s):
<TransferFunction>: sys[2]
Inputs (1): ['u[0]']
Outputs (1): ['y[0]']

     s + 1
  -----------
  s^2 + 2 s

-------------------------------------
PI Controller Parameters (with K = 2.0):
  Kp (Proportional): 2.0000
  Ki (Integral):     2.0000
Closed-loop poles: [-3.4142+0.j -0.5858+0.j]
-------------------------------------
```

PID Design by Frequency Response

Frequency response PID design starts from a target gain crossover frequency wc and phase margin PM. After choosing the relative integral strength, the controller magnitude and phase at wc can be solved simultaneously so that |C(jwc)G(jwc)| = 1 and the loop phase equals -180 degrees + PM. These are loop-shaping targets, not a guarantee of every time-domain requirement, so the completed design must be verified.



Figure 15-21 Step-by-step procedure for PID controller design in the frequency domain based on crossover frequency and phase margin specifications

The calculation applies to a continuous-time SISO plant when the required controller phase is feasible for the selected ideal-PID structure. Higher-order plants can also be treated, but additional crossover frequencies and robustness constraints may require more advanced design.

```
# PID - Frequency Response Design – for first or
second order systems.
import control as ct
import numpy as np
import matplotlib.pyplot as plt

# 1. Plant definition and performance requirements
s = ct.tf('s')
plant = 1 / (s**2 + 2 * s + 5)
desired_wc = 5      # Desired crossover frequency
in rad/s
desired_pm_deg = 60 # Desired phase margin in
degrees

# 2. Analyze the plant at the crossover frequency
response_at_wc = ct.evalfr(plant, 1j * desired_wc)
plant_mag = np.abs(response_at_wc)
plant_phase_deg =
np.rad2deg(np.angle(response_at_wc))

print(f"At the frequency of {desired_wc} rad/s,
the plant has:")
print(f"  - Magnitude: {plant_mag:.3f}")
print(f"  - Phase: {plant_phase_deg:.3f}°")

# 3. PID controller design with simultaneous
magnitude and phase matching
required_phase_deg = -180 + desired_pm_deg -
plant_phase_deg
```

```
required_phase_rad =
np.deg2rad(required_phase_deg)
integral_ratio = 0.1  # Ki/(Kp*wc); places
integral action below wc
Kp = np.cos(required_phase_rad) / plant_mag
Ki = integral_ratio * Kp * desired_wc
Kd = (Kp * np.tan(required_phase_rad) + Ki /
desired_wc) / desired_wc

print(f"\nThe controller needs to add
{required_phase_deg:.3f}° of phase lead.")
print(f"Calculated PID gains:\n  - Kp =
{Kp:.3f}\n  - Ki = {Ki:.3f}\n  - Kd = {Kd:.3f}")

# 4. Verification of the compensated system
pid_controller = Kp + Ki / s + Kd * s
open_loop_system = pid_controller * plant
gm, pm, wcg, wcp = ct.margin(open_loop_system)

print(f"\nFinal Stability Margins:")
print(f"  - Gain Margin: {'inf' if gm == np.inf
else f'{20 * np.log10(gm):.2f} dB'}")
print(f"  - Phase Margin: {pm:.2f}°")
print(f"  - Gain Crossover Frequency: {wcp:.3f}
rad/s")
```

Below is an explanation of the blocks used in the code:

1. *Plant Definition and Performance Requirements*

```
# 1. Plant definition and performance
requirements
s = ct.tf('s')
plant = 1 / (s**2 + 2 * s + 5)
desired_wc = 5      # Desired crossover
frequency in rad/s
```

```
desired_pm_deg = 60 # Desired phase margin in
degrees
```

The first line is

```
s = ct.tf('s')
```

This line creates a special object s that represents the Laplace variable, s. This allows you to define transfer functions algebraically, just like you would on paper.

```
plant = 1 / (s**2 + 2 * s + 5)
```

This defines the mathematical model (the transfer function) of the system you want to control. This is the "plant." You would change this line to match whatever system you are working with.

```
desired_wc = 5
```

This sets your first performance goal: response speed. The crossover frequency (wc or ωc) is the frequency at which the system's open-loop gain is 1 (or 0 dB). A higher ωc generally leads to a faster-reacting system.

```
desired_pm_deg = 60
```

This sets your second performance goal: stability and damping. The phase margin (PM) is a measure of how far the system is from becoming unstable. A value between 45 and 60 degrees is typical for a stable system that doesn't oscillate or overshoot too much.

2. *Analyze the Plant at the Crossover Frequency*

```
# 2. Analyze the plant at the crossover
frequency
response_at_wc = ct.evalfr(plant, 1j *
desired_wc)
```

```
plant_mag = np.abs(response_at_wc)
plant_phase_deg =
np.rad2deg(np.angle(response_at_wc))
```

The goal here is to find out how the original, uncontrolled plant behaves at your target frequency (desired_wc).

```
response_at_wc = ct.evalfr(plant, 1j *
desired_wc)
```

This is the core calculation. It evaluates the plant's transfer function at s = jω, where j is the imaginary unit and ω is your desired_wc. The result is a single complex number that contains both magnitude and phase information.

```
plant_mag = np.abs(response_at_wc)
```

This extracts the magnitude (or gain) from the complex number. It tells you how much the plant amplifies or dampens signals at your target frequency.

```
plant_phase_deg =
np.rad2deg(np.angle(response_at_wc))
```

This extracts the phase shift from the complex number. np.angle gets the phase in radians, and np.rad2deg converts it to degrees, which is more intuitive.

3. *PID Controller Design*

```
# 3. PID controller design with simultaneous
magnitude and phase matching
required_phase_deg = -180 + desired_pm_deg -
plant_phase_deg
required_phase_rad =
np.deg2rad(required_phase_deg)
integral_ratio = 0.1  # Ki/(Kp*wc)
```

```
Kp = np.cos(required_phase_rad) / plant_mag
Ki = integral_ratio * Kp * desired_wc
Kd = (Kp * np.tan(required_phase_rad) + Ki /
desired_wc) / desired_wc
```

The gains are obtained by satisfying the complete ideal-PID magnitude and phase at wc. The integral ratio is selected first; Kp is then scaled by cos(phi) so that the full controller, including Ki and Kd, has the required magnitude rather than treating Kp alone as the controller.

```
Kp = np.cos(required_phase_rad) / plant_mag
```

The required controller phase phi is calculated from the desired phase margin. With Ki/(Kp wc) fixed by integral_ratio, Kp = cos(phi)/|G(jwc)| enforces the complete loop-magnitude condition at wc.

required_phase_deg determines the controller phase needed at wc from PM = 180 degrees + angle C(jwc)G(jwc). The code checks this phase through the final margin calculation.

Kd is obtained from the imaginary part of C(jwc) = Kp + j(Kd wc - Ki/wc), so both derivative and integral contributions are included in the target phase.

Ki is set by integral_ratio. The value 0.1 is a design choice that keeps the integral contribution below the crossover region; it must be reconsidered if actuator limits or low-frequency disturbance rejection requires a different trade-off.

4. *Verification of the Compensated System*

```
# 4. Verification of the compensated system
pid_controller = Kp + Ki / s + Kd * s
open_loop_system = pid_controller * plant
gm, pm, wcg, wcp = ct.margin(open_loop_system)
```

This section builds the complete system and checks if the design was successful.

```
pid_controller = Kp + Ki / s + Kd * s
```

This assembles the PID controller's transfer function using the gains calculated in the previous step.

```
open_loop_system = pid_controller * plant
```

This creates the complete open-loop system by combining the controller and the plant. This is the system whose margins we need to check.

```
gm, pm, wcg, wcp = ct.margin(open_loop_system)
```

The ct.margin() function automatically calculates the final gain margin (gm) and phase margin (pm) of your newly designed system.

For this example, the verified gain crossover is 5.000 rad/s and the phase margin is 60.00 degrees. Other plants may require a different integral ratio, a filtered derivative, or numerical optimization if the requested phase is infeasible.

The Bode plot below verifies that the compensated loop crosses 0 dB at 5 rad/s and has a 60-degree phase margin. A practical implementation should replace the ideal derivative Kd s with a filtered derivative and recheck the margins.

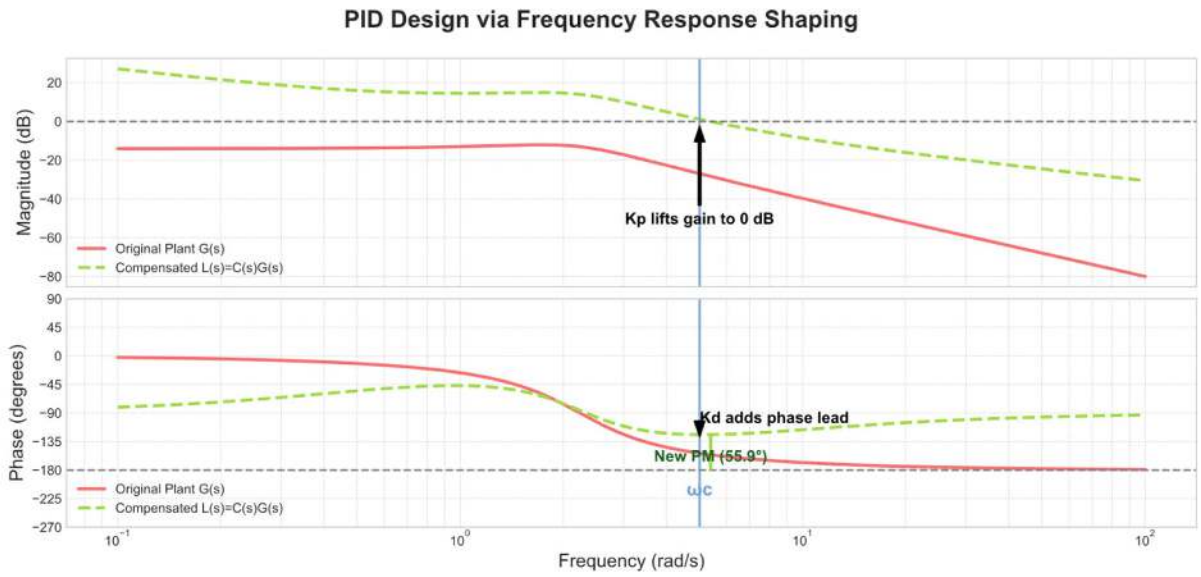


Figure 15-22 PID controller design via frequency response shaping, illustrating how proportional and derivative actions modify the Bode magnitude and phase to achieve the desired crossover frequency and phase margin

This plot is a powerful visual tool that illustrates how the PID controller reshapes the behavior of the original system to meet performance requirements.

It is divided into two parts: the magnitude plot on the top and the phase plot on the bottom. Together, they show how the controller modifies both the system's gain and phase across frequencies.

The two curves serve as a clear comparison of the system before and after control design. The red line represents the original plant G(s), showing how the system behaves without any compensation. In contrast, the green dashed line represents the compensated system L(s)=C(s)G(s), which reflects the effect of the designed PID controller. This visual contrast highlights how the controller achieves the desired performance.

In the magnitude plot, the proportional, integral, and derivative terms are solved together so that the compensated loop crosses 0 dB at wc = 5 rad/s. Kp alone does not establish this condition because Ki/s and Kd s also contribute to the controller magnitude at the crossover frequency.

The phase plot focuses on stability. It shows how much phase shift (or delay) the system introduces at each frequency. For robust stability, the system's phase at the crossover frequency must remain sufficiently above the critical –180∘ line. This vertical distance is the phase margin (PM). In the uncompensated system, represented by the red line, the phase margin is relatively small. In the uncompensated system, represented by the red line, the phase margin is relatively small. To reach the target of 60°, the derivative gain Kd is applied. This term adds phase lead, effectively pushing the curve upward around the crossover frequency.

The annotation reports a phase margin of 60.0 degrees at the verified gain crossover frequency. This confirms the two frequency-domain constraints used in the calculation, while separate time-domain and robustness checks remain necessary.

If you want to plot the Bode diagram explained above, add this code to your original PID project code. The goal is to use the numerical results from the first four sections and create an intuitive Bode plot that graphically shows the success of your PID project. It doesn't perform any new design calculations; it simply visualizes the "before" and "after" results.

```
# ------------------------------------------------
------------------------
# 5. BODE PLOT VISUALIZATION OF THE DESIGN
# This section is an addition to visualize the
results from the steps above.
# ------------------------------------------------
------------------------
print("\n--- Generating Bode Plot Visualization --
-")

# Set plot style for a white background
plt.style.use('seaborn-v0_8-whitegrid')
```

```
fig, (ax_mag, ax_phase) = plt.subplots(2, 1,
figsize=(12, 10), sharex=True)
fig.suptitle('PID Design: Before vs. After
Compensation', fontsize=18, weight='bold')

# Frequency range for the plot
omega = np.logspace(-1, 2, 500)

# Plot Bode for the original Plant (Uncompensated)
mag_plant, phase_plant, _ =
ct.frequency_response(plant, omega)
ax_mag.plot(omega, 20 * np.log10(mag_plant),
color='r', linewidth=2.5, label='Original Plant
G(s)')
ax_phase.plot(omega, phase_plant * 180/np.pi,
color='r', linewidth=2.5, label='Original Plant
G(s)')

# Plot Bode for the final Open-Loop System
(Compensated)
mag_ol, phase_ol, _ =
ct.frequency_response(open_loop_system, omega)
ax_mag.plot(omega, 20 * np.log10(mag_ol),
color='g', linewidth=2.5, linestyle='--',
label='Compensated L(s)=C(s)G(s)')
ax_phase.plot(omega, phase_ol * 180/np.pi,
color='g', linewidth=2.5, linestyle='--',
label='Compensated L(s)=C(s)G(s)')

# --- Annotations ---
# Crossover frequency vertical line
ax_mag.axvline(x=desired_wc, color='blue',
linestyle=':', lw=2)
ax_phase.axvline(x=desired_wc, color='blue',
linestyle=':', lw=2, label=f'Target ωc =
```

```
{desired_wc} rad/s')

# Magnitude plot annotation
ax_mag.axhline(y=0, color='gray', linestyle='--',
lw=1.5)
ax_mag.annotate('PID places the 0 dB crossing at
wc',
                xy=(desired_wc, 0),
xycoords='data',
                xytext=(desired_wc, -25),
textcoords='data',
                arrowprops=dict(facecolor='black',
shrink=0.05, width=1.5, headwidth=7),
                ha='center', va='top',
fontsize=12)

# Phase plot annotation
ax_phase.axhline(y=-180, color='gray',
linestyle='--', lw=1.5)
gm_final, pm_final, _, wc_final =
ct.margin(open_loop_system)
phase_final_at_wc = np.interp(wc_final, omega,
phase_ol * 180/np.pi)
ax_phase.vlines(wc_final, -180, phase_final_at_wc,
color='g', lw=2.5)
ax_phase.annotate(f'New PM = {pm_final:.1f}°',
                  xy=(wc_final, -180),
xycoords='data',
                  xytext=(wc_final*1.2, -160),
textcoords='data',
                  ha='left', va='center',
color='g', fontsize=12, weight='bold')

# Final Plot Formatting
ax_mag.set_ylabel('Magnitude (dB)', fontsize=14)
```

```
ax_mag.legend()
ax_mag.set_xscale('log')

ax_phase.set_ylabel('Phase (degrees)',
fontsize=14)
ax_phase.set_xlabel('Frequency (rad/s)',
fontsize=14)
ax_phase.legend()
ax_phase.set_xscale('log')
ax_phase.set_yticks(np.arange(-270, 91, 45))

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

The graph generated by this code block is shown below.

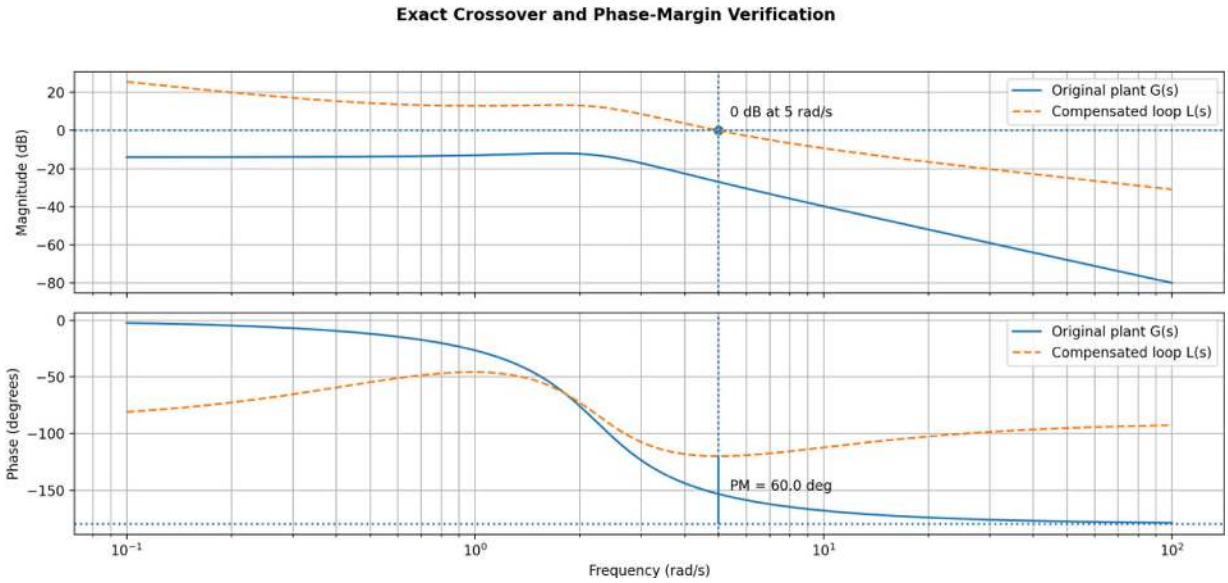


Figure 15-23 Comparison of the system frequency response before and after PID compensation, showing the specified gain crossover frequency and 60-degree phase margin

© The Author(s), under exclusive license to APress Media, LLC, part of Springer Nature 2026
C. M. d. S. Freitas, *Python for Engineers and Scientists*
https://doi.org/10.1007/979-8-8688-3165-2_16

16. Databases Using SQLAlchemy

Carlos Márcio da Silva Freitas[1]
(1) São Paulo, Brazil

Databases and SQLAlchemy

In this chapter, we will explore how to work with databases in a simple and transparent way using the resources provided by SQLAlchemy. The examples focus on the fundamental concepts needed to create, store, retrieve, and manage data in Python applications.

Databases in Engineering Applications

In modern engineering, data operations are of fundamental importance. From sensor readings in a manufacturing plant to simulation results in a design process, the ability to efficiently store, retrieve, and manage data is crucial.

It is important to emphasize that this chapter serves as an introductory overview of database usage with SQLAlchemy, focusing on fundamental concepts and practical engineering-oriented examples. The examples were written and tested with SQLAlchemy 2.0.51, ensuring alignment with the safest and most efficient practices in the Python ecosystem.

SQLAlchemy is a powerful and extensive toolkit, offering many advanced features related to performance optimization, schema migrations, concurrency control, and complex query construction that go beyond the scope of this introduction.

Readers who intend to use SQLAlchemy in production systems or advanced engineering applications are strongly encouraged to consult the official SQLAlchemy documentation, which provides comprehensive and

up-to-date guidance on all available features, best practices, and supported database backends. Official documentation is at https://docs.sqlalchemy.org/.

Types of Databases and Engineering Use Cases

A database is a collection of organized data stored in a way that allows for efficient retrieval and manipulation. There are several types of databases. Relational databases, for example, are SQL (Structured Query Language)–based systems such as MySQL, PostgreSQL, and SQLite. They store data in tables with defined relationships, and SQL is the standard language used to interact with them. It allows users to perform various operations, including creating and modifying database structures (tables, views, stored procedures); inserting, updating, and deleting data; as well as retrieving specific information through queries.

Other types include NoSQL databases, such as MongoDB, which store data in flexible, schema-less formats; time-series databases, such as InfluxDB, optimized for storing and querying large amounts of timestamped data; and graph databases, such as Neo4j, designed to represent complex relationships between data entities. Each type of database is suited for specific use cases—for example, relational databases for transactional data, NoSQL databases for large volumes of unstructured data, and graph databases for modeling intricate relationships.

In engineering, databases have a wide range of applications, including product lifecycle management, simulation data management, and sensor data storage for IoT applications. They also support tools such as computer-aided design (CAD) software, engineering analytics, and supply chain management, enabling engineers to store, retrieve, and analyze large volumes of data efficiently. For instance, databases can be used to track the performance of mechanical components, store and manage design files, or analyze data from sensors and testing equipment. By leveraging databases, engineers can make data-driven decisions, optimize product performance, and improve overall efficiency in their workflows.

The table below compares different databases and their use cases, strengths, and engineering applications. Each database's strengths are

matched with specific engineering applications, showing their suitability for various use cases.

Table

Why Use an ORM in Engineering Projects

While SQL databases are the industry standard for structured data storage, interacting with them directly using SQL queries within a Python application can be cumbersome and error-prone. This is where SQLAlchemy comes in.

Overview of SQLAlchemy

SQLAlchemy is one of the most popular tools in the Python ecosystem for database management. It offers an Object-Relational Mapper (ORM) that provides application developers with easy access to the full power of SQL. In essence, an ORM such as SQLAlchemy allows you to interact with your database using Python objects instead of raw SQL. This approach offers several advantages for engineers:

Increased Productivity: By abstracting away the SQL, you can focus on the logic of your engineering application rather than the intricacies of database queries.
Improved Code Readability: Python code that uses SQLAlchemy is often more readable and maintainable than code with embedded SQL strings.
Database Agnosticism: SQLAlchemy supports a wide range of database backends (like SQLite, PostgreSQL, and MySQL), allowing you to switch between them with minimal code changes. This is particularly useful in engineering projects where the database requirements might change over time.

This chapter will guide you through the fundamentals of SQLAlchemy, with examples tailored to common engineering use cases.

Getting Started with SQLAlchemy

Before interacting with databases in an engineering application, it is necessary to prepare the development environment and understand the basic requirements for using SQLAlchemy. This section introduces the initial setup steps required to begin working with the library, ensuring that the necessary tools and dependencies are correctly installed.

SQLAlchemy is designed to integrate seamlessly with Python, making it suitable for a wide range of engineering workflows from quick prototypes and educational examples to more structured data management systems. By starting with a proper installation and minimal configuration, engineers can quickly move from conceptual understanding to hands-on experimentation.

The following subsection focuses on installing SQLAlchemy using standard Python tools and verifying that the environment is ready for use. These steps provide the foundation for all subsequent examples in the chapter, including database connections, session management, and ORM-based data modeling.

Installing SQLAlchemy with pip

The examples in this chapter use SQLAlchemy 2.0.51. To reproduce the same environment, install that version with pip:

```
pip install SQLAlchemy==2.0.51
```

Database Connection and Engine Creation

Before any data can be stored, queried, or manipulated using SQLAlchemy, the application must define how it will access the target database. In SQLAlchemy, this access is managed through a central component called the engine, which acts as the bridge between your Python application and the database system.

In engineering applications, this abstraction is particularly important. A prototype may start with a lightweight SQLite database running locally, while later stages of the same project may require migration to a more robust system such as PostgreSQL or MySQL for handling larger datasets, concurrent access, or long-term storage. SQLAlchemy's engine design enables this transition with minimal changes to the application code.

This section introduces the concept of the SQLAlchemy engine, explains how connection strings define database dialects and access parameters, and demonstrates how to create a reliable database connection using SQLite as a practical example. The concepts presented here form the foundation for all subsequent operations, including session management, ORM model definitions, and CRUD (Create, Read, Update, and Delete) workflows used throughout the chapter.

The SQLAlchemy Engine Concept

The entry point to any SQLAlchemy application is the create_engine() function. Rather than establishing a single, rigid connection, the resulting Engine object manages database connectivity dynamically, typically through a connection pool and transaction-related infrastructure.

Instead of interacting directly with low-level database drivers, developers use the engine as a unified interface. It coordinates all communication behind the scenes, ensuring that the Python code remains clean, consistent, and independent of the specific SQL dialect in use—allowing SQLAlchemy applications to work with different database systems while maintaining a consistent programming structure.

Connection Strings and Database Dialects

In SQLAlchemy, the information required to connect to a database is provided through a connection string, sometimes referred to as a database URL. This string encodes both the database dialect and the connection details, allowing SQLAlchemy to know *what type of database is being used* and *how to access it*.

A database dialect represents a specific database system and its SQL syntax, such as SQLite, PostgreSQL, MySQL, or Oracle. Each database has its own variations in SQL commands, data types, and connection mechanisms. SQLAlchemy abstracts these differences through dialects, enabling engineers to write database-agnostic Python code while still leveraging the full capabilities of each backend.

A typical SQLAlchemy connection string follows this general structure:

```
dialect+driver://username:password@host:port/database
```

Not all components are required for every database. For example, SQLite is a file-based database and does not require a username, password, host, or port. As a result, its connection string is significantly simpler.

Examples of common connection strings include

SQLite (Local File):

```
sqlite:///example.db
```

PostgreSQL:

```
postgresql://user:password@localhost:5432/mydatabase
```

MySQL:

```
mysql+pymysql://user:password@localhost/mydatabase
```

In engineering workflows, understanding connection strings is essential for configuring applications across different environments. A development setup might use SQLite for simplicity and portability, while testing and production environments may rely on PostgreSQL or MySQL for improved performance, concurrency, and data integrity. By modifying only the connection string, engineers can adapt the same SQLAlchemy-based application to different database systems without rewriting core logic.

This flexibility makes SQLAlchemy particularly well suited for engineering projects that evolve over time, scale in complexity, or must be deployed across multiple platforms and infrastructure configurations.

SQLite As a Prototyping Database

The create_engine() function receives a database URL as its main argument and may also receive keyword arguments that configure the engine and its connection behavior. For this chapter, we will use SQLite, a simple file-based database that is well suited to prototyping and small-scale applications.

Engineering Example: Wind Turbine Monitoring Database

Let's create a database to store data from a wind turbine:

```
from sqlalchemy import create_engine

# The string 'sqlite:///wind_turbine_data.db'
specifies the database type (SQLite)
# and the name of the database file.
engine =
create_engine("sqlite:///wind_turbine_data.db",
echo=False)
```

When the engine first connects to the database, SQLite will create a file named wind_turbine_data.db in the current directory if the file does not already exist.

The create_engine() call configures a SQLAlchemy engine for a SQLite database. Let us break it down:

- engine = create_engine(...): This creates an Engine object, which provides the central database connectivity interface used by SQLAlchemy.
- "sqlite:///wind_turbine_data.db": This is the connection string that specifies the database to connect to.
  - sqlite: This specifies the database dialect, which in this case is SQLite.
  - ///: This indicates that the database file is located on the local machine. The three slashes (///) specify a relative path to the current working directory.
  - wind_turbine_data.db: This is the name of the database file that will be created or connected to.
- echo=False: This parameter controls SQL statement logging. When set to True, SQLAlchemy writes generated SQL to standard output, which is useful during debugging. In this example, logging is disabled.

The create_engine() function also accepts other keyword arguments. Common examples include pool_size, pool_recycle, pool_pre_ping, and execution_options. Their use depends on the database driver and

connection pool; the simple SQLite example in this chapter does not require additional pool configuration.

Sessions: The Gateway to Your Database

Once a database connection has been established through the SQLAlchemy engine, ORM-based interactions are commonly coordinated through a session. The session serves as the primary interface between the application's Python objects and the underlying database, acting as a controlled workspace for database operations.

Conceptually, a session represents a unit of work with the database. It manages the lifecycle of ORM objects, tracks changes made to those objects, and coordinates when pending changes are flushed to the database and when the surrounding transaction is committed. This separation allows related operations to be handled together while preserving transactional control.

In engineering applications, this transactional control is critical. Sensor measurements, simulation results, or configuration parameters are often collected and processed in batches, and partial or inconsistent writes can lead to corrupted datasets or incorrect analyses. By grouping related operations into a single session, SQLAlchemy helps ensure that data is either fully committed or safely rolled back in case of errors.

Sessions also help coordinate efficient database access. A session obtains connections through the engine and its connection pool, while ORM features such as lazy loading and relationship management simplify access to related objects. This makes sessions suitable for both small-scale prototypes and larger engineering systems with more complex data interactions.

In this section, you will learn how SQLAlchemy sessions are created, how they manage transactions, and how they serve as the gateway for Create, Read, Update, and Delete (CRUD) operations used throughout the remainder of this chapter.

Understanding Sessions and Transactions

In SQLAlchemy, a session is tightly coupled with the concept of a database transaction. A transaction represents a logical unit of work that must be executed atomically—either all operations succeed, or none of them are applied to the database. This mechanism is essential for preserving data integrity, especially in engineering systems where consistency and traceability are critical.

A newly created session does not immediately emit SQL. By default, SQLAlchemy begins transactional work automatically when the session first needs to interact with the database. Objects added, modified, or deleted are tracked by the session. Before certain ORM queries and before commit(), a flush may send pending INSERT, UPDATE, or DELETE statements to the database. Calling commit() flushes any remaining changes and then commits the current transaction.

If an error occurs before the transaction is committed, such as invalid data, a constraint violation, or an unexpected system failure, the application can call rollback(). This reverses the uncommitted transaction and returns the session to a usable transactional state. From an engineering perspective, this behavior is crucial when handling measurement data, simulation outputs, or configuration parameters, where partial updates could compromise analyses or system behavior.

Another important aspect of sessions is their role as an identity map. Within a session, each database row corresponds to a single Python object. This guarantees consistency when the same record is accessed multiple times during a transaction, preventing duplication and ensuring that all references reflect the same state.

By combining object tracking, transactional control, and error recovery, SQLAlchemy sessions provide a robust foundation for reliable data management. Understanding how sessions and transactions work is fundamental before performing CRUD operations or designing larger, transaction-heavy engineering applications.

Creating Sessions with sessionmaker

Sessions may be created directly from the Session class, but applications often use a session factory configured with sessionmaker. A factory

separates session configuration from session creation, promoting consistency, reusability, and cleaner application design.

The sessionmaker function generates a customized Session class that is bound to a specific database engine. This binding ensures that every session created by the factory uses the same connection settings, transaction behavior, and database backend. In practice, sessionmaker acts as a blueprint for creating sessions with predefined characteristics.

A typical workflow involves two steps. First, a session factory is configured and associated with the engine. Second, individual session objects are created from that factory whenever database interaction is required. This pattern is particularly useful in engineering applications where multiple independent workflows such as data acquisition, analysis, and reporting may need to interact with the same database in a controlled manner.

By using sessionmaker, engineers can easily manage session lifecycles, create and close sessions as needed, and avoid unintended sharing of transactional state across different parts of an application. This becomes especially important in long-running processes, batch computations, or applications that handle concurrent operations.

In the context of this chapter, sessionmaker provides the structured entry point for ORM interactions. Every Create, Read, Update, and Delete (CRUD) operation demonstrated later relies on sessions instantiated through this mechanism, making it a foundational component of the example application.

Declarative Base and ORM Infrastructure

For the ORM examples in this chapter, database work is coordinated through a session. The sessionmaker function creates a configured session factory, while DeclarativeBase provides the base class used by the ORM models.

```
from sqlalchemy.orm import DeclarativeBase,
sessionmaker

# Create a configured Session factory
Session = sessionmaker(bind=engine)
```

```
# Create a session
session = Session()

# Create a base class for the ORM models
class Base(DeclarativeBase):
    pass
```

Defining Data Models for Engineering Applications

In SQLAlchemy, data models form the core of the Object-Relational Mapping (ORM) approach. A data model defines how real-world entities such as machines, sensors, components, or measurements are represented as Python objects and how these objects are mapped to tables in a relational database.

For engineering applications, data models provide a structured and expressive way to represent complex systems. Instead of thinking in terms of rows and columns, engineers work with meaningful abstractions such as *turbines*, *sensor readings*, or *simulation results*. Each model encapsulates both the structure of the data and its relationships with other entities, making the code easier to understand, maintain, and extend.

SQLAlchemy models are typically defined as Python classes that inherit from a common declarative base. In SQLAlchemy 2.0, the annotated declarative style uses Mapped type annotations and mapped_column() for column-specific configuration. Each class corresponds to a database table, while mapped attributes define the table's columns, data types, and constraints.

In addition to defining individual entities, data models also capture relationships between them, such as one-to-many or many-to-one associations. These relationships are especially important in engineering systems, where measurements are often linked to specific devices, experiments, or time periods. By modeling these connections explicitly, SQLAlchemy allows engineers to navigate related data intuitively using Python objects rather than complex SQL joins.

This section introduces the principles of ORM model definition in SQLAlchemy and demonstrates how to design data models that reflect real engineering use cases. The examples that follow focus on clarity, correctness, and extensibility, which are key requirements for reliable data management in engineering projects.

ORM Models and Table Mapping

In SQLAlchemy, database tables are represented by Python classes, often referred to as models. These classes inherit from Base, the DeclarativeBase subclass created in the previous step. In the SQLAlchemy 2.0 annotated declarative style, mapped attributes are described with Mapped and, when additional column configuration is required, mapped_column().

Engineering Example: Turbine and SensorReading Models

Let's define two models for our wind turbine monitoring system:

- *Turbine*: To store information about each wind turbine
- *SensorReading*: To store the sensor readings from each turbine

```
from sqlalchemy import ForeignKey, String
from sqlalchemy.orm import Mapped, mapped_column,
relationship
```

Turbine Model Definition

```
class Turbine(Base):
    __tablename__ = "turbines"

    id: Mapped[int] =
mapped_column(primary_key=True)
    model: Mapped[str] = mapped_column(String(50))
    location: Mapped[str] =
mapped_column(String(100))

    # One turbine can have many sensor readings
```

```
    readings: Mapped[list["SensorReading"]] =
relationship(
        back_populates="turbine"
    )
```

SensorReading Model Definition

```
class SensorReading(Base):
    __tablename__ = "sensor_readings"

    id: Mapped[int] =
mapped_column(primary_key=True)
    timestamp: Mapped[float]
    wind_speed: Mapped[float]
    power_output: Mapped[float]
    turbine_id: Mapped[int] = mapped_column(
        ForeignKey("turbines.id")
    )

    # Each sensor reading belongs to one turbine
    turbine: Mapped["Turbine"] = relationship(
        back_populates="readings"
    )

# Create tables that do not already exist
Base.metadata.create_all(engine)
```

In this example:

- __tablename__ explicitly sets the name of the table in the database.
- Mapped[...] identifies an ORM-mapped attribute and provides Python typing information.
- mapped_column() supplies column-specific configuration such as primary keys, SQL types, and foreign keys.
- primary_key=True marks a column as the primary key for the table.
- relationship() defines how related ORM objects are associated and navigated in Python.

- ForeignKey creates a database-level reference from sensor_readings.turbine_id to turbines.id.

Relationships and Table Creation

The relationship construct defines how Python objects are associated with one another at the ORM level, establishing logical links between entities such as Turbine and SensorReading. While database-level relationships are enforced using foreign keys, relationship enables SQLAlchemy to represent and navigate these connections naturally using Python objects.

In this context, a one-to-many relationship is established, where a single Turbine instance can be associated with multiple SensorReading records. This allows engineers to access all sensor measurements of a turbine directly through an object attribute, without writing explicit SQL join statements.

Once the models and their relationships are defined, SQLAlchemy can create the corresponding database tables from the metadata associated with the base class. Calling Base.metadata.create_all(engine) creates tables that do not yet exist. It does not alter existing tables when a model changes; schema migrations are normally handled with a migration tool such as Alembic.

This combination of relationship definitions and metadata-based table creation simplifies initial database setup, making it especially useful for prototypes, teaching examples, and early stages of engineering applications.

A SQLite-specific detail is worth noting: foreign key enforcement is disabled by default. Applications that depend on database-level foreign key enforcement should enable PRAGMA foreign_keys=ON for each SQLite connection. The ORM relationship still provides Python-side navigation between the mapped objects.

CRUD Operations for Engineering Data

CRUD stands for Create, Read, Update, and Delete. These are the four basic functions of persistent storage in any database system. Together, these

operations define the complete lifecycle of stored data, from initial insertion to eventual removal.

In engineering applications, CRUD operations are used to log new measurements, retrieve historical or real-time data, correct erroneous values, and remove obsolete or invalid records. Whether handling sensor readings, simulation outputs, or configuration parameters, these operations form the backbone of reliable data management workflows.

SQLAlchemy provides a structured and intuitive interface for performing CRUD operations through sessions and ORM models. By abstracting low-level SQL commands, it allows engineers to focus on application logic while maintaining transactional safety, data integrity, and consistency across database interactions.

The following subsections demonstrate each CRUD operation in practical engineering scenarios, illustrating how data can be safely created, queried, modified, and deleted using SQLAlchemy.

Create: Logging New Sensor Readings

To add new data to the database, create Python objects, add them to the session, and commit the transaction.

```
# Create a new turbine
turbine1 = Turbine(model="V112-3.0", location="Site
A")
session.add(turbine1)

# Create a new sensor reading for this turbine
reading1 = SensorReading(
    timestamp=1678886400.0,
    wind_speed=12.5,
    power_output=2.5,
    turbine=turbine1,
)
session.add(reading1)
session.commit()
```

Read: Querying Engineering Data

SQLAlchemy 2.0 uses select() as the standard ORM query construct. A Select object can be executed with Session.execute(); calling scalars() on the result returns the mapped ORM objects directly.

```
from sqlalchemy import select

# Get all turbines
all_turbines =
session.execute(select(Turbine)).scalars().all()
for turbine in all_turbines:
    print(
        f"Turbine Model: {turbine.model}, "
        f"Location: {turbine.location}"
    )

# Get the first turbine with the model "V112-3.0"
specific_turbine = session.execute(
    select(Turbine).where(Turbine.model == "V112-
3.0")
).scalars().first()

if specific_turbine is not None:
    print(f"Found Turbine:
{specific_turbine.model}")

    # Get all readings for the selected turbine
    turbine_readings = session.execute(
        select(SensorReading).where(
            SensorReading.turbine_id ==
specific_turbine.id
        )
    ).scalars().all()

    for reading in turbine_readings:
        print(
            f"Timestamp: {reading.timestamp}, "
```

```
            f"Power Output: {reading.power_output}
MW"
        )
```

Update: Correcting Erroneous Measurements

To update a record, retrieve it, modify its attributes, and commit the changes. An object loaded by the session is already being tracked, so it does not need to be added again.

```
# Assume we found an error in reading 1
reading_to_update = session.get(SensorReading, 1)

if reading_to_update is not None:
    # SQLAlchemy detects the attribute change
    reading_to_update.power_output = 2.4
    session.commit()
```

Delete: Removing Obsolete Records

Deleting data involves permanently removing records that are no longer required or that may compromise the accuracy of the dataset. In SQLAlchemy, the target object is first retrieved within an active session and then marked for deletion using session.delete().

The DELETE statement is normally emitted when the session flushes. Calling commit() flushes any pending work and commits the transaction. Until the transaction is committed, the application can call rollback() if an error occurs.

In engineering applications, delete operations must be used with caution. Removing sensor measurements, test results, or configuration data can affect analyses, traceability, and system behavior. For this reason, deletions are often reserved for truly obsolete, invalid, or duplicated records and are typically combined with validation or logging mechanisms.

By managing deletions through SQLAlchemy sessions, engineers retain control over data integrity while maintaining consistency across related tables and workflows.

```
# Delete a specific reading
reading_to_delete = session.get(SensorReading, 1)

if reading_to_delete is not None:
    session.delete(reading_to_delete)
    session.commit()
```

Complete SQLAlchemy Example

The previous sections introduced the core components of SQLAlchemy step by step, including database engines, sessions, ORM models, relationships, and CRUD operations. While each concept can be understood in isolation, real engineering applications require these elements to work together as a cohesive system.

This section presents a complete, self-contained example that integrates all the concepts covered in this chapter into a single executable script. The example demonstrates how to establish a database connection, define data models, create tables, manage sessions, and perform full CRUD workflows in a realistic engineering scenario.

By following this end-to-end example, engineers can see how SQLAlchemy is typically structured in practice, from initial configuration to data persistence and cleanup. The script also serves as a practical reference that can be adapted or extended for real-world applications such as monitoring systems, experimental data logging, or simulation result storage.

This complete example reinforces the practices discussed throughout the chapter and provides a clear template for building maintainable database-driven engineering applications with SQLAlchemy.

Full Wind Turbine Monitoring Script

Here is a complete script that demonstrates all the concepts covered in this chapter:

```
from sqlalchemy import ForeignKey, String,
create_engine, select
from sqlalchemy.orm import (
```

```
    DeclarativeBase,
    Mapped,
    mapped_column,
    relationship,
    sessionmaker,
)

# 1. Database Connection and Engine Creation
engine =
create_engine("sqlite:///wind_turbine_data.db",
echo=False)

# 2. Session Factory
Session = sessionmaker(bind=engine)

# 3. Base Class for Data Models
class Base(DeclarativeBase):
    pass

# 4. Data Models
class Turbine(Base):
    __tablename__ = "turbines"

    id: Mapped[int] =
mapped_column(primary_key=True)
    model: Mapped[str] = mapped_column(String(50))
    location: Mapped[str] =
mapped_column(String(100))

    readings: Mapped[list["SensorReading"]] =
relationship(
        back_populates="turbine"
    )

class SensorReading(Base):
    __tablename__ = "sensor_readings"
```

```
    id: Mapped[int] =
mapped_column(primary_key=True)
    timestamp: Mapped[float]
    wind_speed: Mapped[float]
    power_output: Mapped[float]
    turbine_id: Mapped[int] = mapped_column(
        ForeignKey("turbines.id")
    )

    turbine: Mapped["Turbine"] = relationship(
        back_populates="readings"
    )

# Create tables that do not already exist
Base.metadata.create_all(engine)

# 5. CRUD Operations
with Session() as session:
    # Create
    new_turbine = Turbine(
        model="V112-3.0",
        location="Site B",
    )
    new_reading = SensorReading(
        timestamp=1678886400.0,
        wind_speed=12.5,
        power_output=2.5,
        turbine=new_turbine,
    )
    session.add_all([new_turbine, new_reading])
    session.commit()

    # Read
    all_turbines = session.execute(
        select(Turbine)
    ).scalars().all()
```

```
for turbine in all_turbines:
    print(
        f"Turbine Model: {turbine.model}, "
        f"Location: {turbine.location}"
    )

# Update
reading_to_update = session.execute(
    select(SensorReading).limit(1)
).scalars().first()
if reading_to_update is not None:
    reading_to_update.power_output = 2.4
    session.commit()

# Delete
reading_to_delete = session.execute(
    select(SensorReading).limit(1)
).scalars().first()
if reading_to_delete is not None:
    session.delete(reading_to_delete)
    session.commit()
```

Bibliographic References

Chapter 1: Introduction

[1] Python Software Foundation: Python 3.13 documentation. https://docs.python.org/3.13/. Accessed July 22, 2026.

[2] van Rossum, G., Warsaw, B., Coghlan, A.: PEP 8—Style Guide for Python Code. Python Software Foundation (2001). https://peps.python.org/pep-0008/. Accessed July 22, 2026.

[3] Peters, T.: PEP 20—The Zen of Python. Python Software Foundation (2004). https://peps.python.org/pep-0020/. Accessed July 22, 2026.

[4] Smith, E.V.: PEP 498—Literal String Interpolation. Python Software Foundation (2015). https://peps.python.org/pep-0498/. Accessed July 22, 2026.

[5] Baker, J., van Rossum, G., Everitt, P., Aono, K., Nikolaou, L., Peck, D.: PEP 750—Template Strings. Python Software Foundation (2024). https://peps.python.org/pep-0750/. Accessed July 22, 2026.

[6] Python Packaging Authority: Python Packaging User Guide. https://packaging.python.org/. Accessed July 22, 2026.

[7] Astral Software, Inc.: uv documentation. https://docs.astral.sh/uv/. Accessed July 22, 2026.

Chapter 2: Variables

[1] Python Software Foundation: The Python Language Reference—Data Model. https://docs.python.org/3.13/reference/datamodel.html. Accessed July 22, 2026.

[2] Python Software Foundation: Built-In Types. https://docs.python.org/3.13/library/stdtypes.html. Accessed July 22, 2026.

[3] Python Software Foundation: typing—Support for Type Hints. https://docs.python.org/3.13/library/typing.html. Accessed July 22, 2026.

[4] Goldberg, D.: What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.* 23(1), 5–48 (1991). https://doi.org/10.1145/103162.103163

[5] Ramalho, L.: *Fluent Python*, 2nd edn. O'Reilly Media, Sebastopol, CA (2022).

Chapter 3: Handling Collections

[1] Python Software Foundation: Built-In Types—Lists, Tuples, Sets, and Dictionaries. https://docs.python.org/3.13/library/stdtypes.html. Accessed July 22, 2026.

[2] Python Software Foundation: collections—Container Datatypes. https://docs.python.org/3.13/library/collections.html. Accessed July 22, 2026.

[3] Python Software Foundation: itertools—Functions Creating Iterators for Efficient Looping. https://docs.python.org/3.13/library/itertools.html. Accessed July 22, 2026.

[4] Ramalho, L.: *Fluent Python*, 2nd edn. O'Reilly Media, Sebastopol, CA (2022).

[5] Lutz, M.: *Learning Python*, 5th edn. O'Reilly Media, Sebastopol, CA (2013).

Chapter 4: Python Modules and Packages

[1] Python Software Foundation: The Python Tutorial—Modules. https://docs.python.org/3.13/tutorial/modules.html. Accessed July 22, 2026.

[2] Python Packaging Authority: Python Packaging User Guide. https://packaging.python.org/. Accessed July 22, 2026.

[3] pip maintainers: pip documentation. https://pip.pypa.io/en/stable/. Accessed July 22, 2026.

[4] Python Poetry: Poetry documentation. https://python-poetry.org/docs/. Accessed July 22, 2026.

[5] Astral Software, Inc.: uv documentation. https://docs.astral.sh/uv/. Accessed July 22, 2026.

[6] Smith, N.J., Kluyver, T.: PEP 517—A Build-System Independent Format for Source Trees. Python Software Foundation (2015). https://peps.python.org/pep-0517/. Accessed July 22, 2026.

[7] Cannon, B., Smith, N.J., Stufft, D.: PEP 518—Specifying Minimum Build System Requirements for Python Projects. Python Software Foundation (2016). https://peps.python.org/pep-0518/. Accessed July 22, 2026.

Chapter 5: Functions

[1] Python Software Foundation: The Python Tutorial—Defining Functions. https://docs.python.org/3.13/tutorial/controlflow.html#defining-functions. Accessed July 22, 2026.

[2] Python Software Foundation: The Python Language Reference—Function Definitions. https://docs.python.org/3.13/reference/compound_stmts.html#function-definitions. Accessed July 22, 2026.

[3] Python Software Foundation: functools—Higher-Order Functions and Operations on Callable Objects. https://docs.python.org/3.13/library/functools.html. Accessed July 22, 2026.

[4] Ramalho, L.: *Fluent Python*, 2nd edn. O'Reilly Media, Sebastopol, CA (2022).

[5] Slatkin, B.: *Effective Python: 90 Specific Ways to Write Better Python*, 2nd edn. Addison-Wesley Professional, Boston (2020).

Chapter 6: Conditionals and Loops

[1] Python Software Foundation: The Python Tutorial—More Control Flow Tools. https://docs.python.org/3.13/tutorial/controlflow.html. Accessed July 22, 2026.

[2] Python Software Foundation: The Python Tutorial—Errors and Exceptions. https://docs.python.org/3.13/tutorial/errors.html. Accessed July 22, 2026.

[3] Python Software Foundation: The Python Language Reference—Compound Statements. https://docs.python.org/3.13/reference/compound_stmts.html. Accessed July 22, 2026.

[4] Lutz, M.: *Learning Python*, 5th edn. O'Reilly Media, Sebastopol, CA (2013).

[5] Slatkin, B.: *Effective Python: 90 Specific Ways to Write Better Python*, 2nd edn. Addison-Wesley Professional, Boston (2020).

Chapter 7: Object-Oriented Programming

[1] Python Software Foundation: The Python Language Reference—Data Model. https://docs.python.org/3.13/reference/datamodel.html. Accessed July 22, 2026.

[2] Python Software Foundation: The Python Tutorial—Classes. https://docs.python.org/3.13/tutorial/classes.html. Accessed July 22, 2026.

[3] Python Software Foundation: abc—Abstract Base Classes. https://docs.python.org/3.13/library/abc.html. Accessed July 22, 2026.

[4] Ramalho, L.: *Fluent Python*, 2nd edn. O'Reilly Media, Sebastopol, CA (2022).

[5] Lott, S.F., Phillips, D.: *Python Object-Oriented Programming: Build Robust and Maintainable Object-Oriented Python Applications and Libraries*, 4th edn. Packt Publishing, Birmingham (2021).

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA (1994).

Chapter 8: Files and Paths

[1] Python Software Foundation: pathlib—Object-Oriented Filesystem Paths. https://docs.python.org/3.13/library/pathlib.html. Accessed July 22, 2026.

[2] Python Software Foundation: io—Core Tools for Working with Streams. https://docs.python.org/3.13/library/io.html. Accessed July 22, 2026.

[3] Python Software Foundation: csv—CSV File Reading and Writing. https://docs.python.org/3.13/library/csv.html. Accessed July 22, 2026.

[4] Python Software Foundation: json—JSON Encoder and Decoder. https://docs.python.org/3.13/library/json.html. Accessed July 22, 2026.

[5] python-docx contributors: python-docx documentation. https://python-docx.readthedocs.io/en/latest/. Accessed July 22, 2026.

[6] openpyxl maintainers: openpyxl documentation. https://openpyxl.readthedocs.io/en/stable/. Accessed July 22, 2026.

[7] Sweigart, A.: *Automate the Boring Stuff with Python*, 2nd edn. No Starch Press, San Francisco (2019).

Chapter 9: Math

[1] Python Software Foundation: Numeric and Mathematical Modules. https://docs.python.org/3.13/library/numeric.html. Accessed July 22, 2026.

[2] Python Software Foundation: math—Mathematical Functions. https://docs.python.org/3.13/library/math.html. Accessed July 22, 2026.

[3] Python Software Foundation: cmath—Mathematical Functions for Complex Numbers. https://docs.python.org/3.13/library/cmath.html. Accessed July 22, 2026.

[4] Python Software Foundation: decimal—Decimal Fixed-Point and Floating-Point Arithmetic. https://docs.python.org/3.13/library/decimal.html. Accessed July 22, 2026.

[5] Python Software Foundation: statistics—Mathematical Statistics Functions. https://docs.python.org/3.13/library/statistics.html. Accessed July 22, 2026.

[6] Kiusalaas, J.: *Numerical Methods in Engineering with Python 3*, 3rd edn. Cambridge University Press, Cambridge (2013).

[7] Johansson, R.: *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*, 2nd edn. Apress, New York (2019).

Chapter 10: Matplotlib

[1] Matplotlib Development Team: Matplotlib 3.10.3 documentation. https://matplotlib.org/3.10.3/. Accessed July 22, 2026.

[2] Hunter, J.D.: Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.* 9(3), 90–95 (2007). https://doi.org/10.1109/MCSE.2007.55

[3] VanderPlas, J.: *Python Data Science Handbook: Essential Tools for Working with Data*, 2nd edn. O'Reilly Media, Sebastopol, CA (2022).

[4] Johansson, R.: *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*, 2nd edn. Apress, New York (2019).

[5] Tufte, E.R.: *The Visual Display of Quantitative Information*, 2nd edn. Graphics Press, Cheshire, CT (2001).

Chapter 11: NumPy

[1] NumPy Developers: NumPy 2.3 documentation. https://numpy.org/doc/2.3/. Accessed July 22, 2026.

[2] Harris, C.R., Millman, K.J., van der Walt, S.J., et al.: Array programming with NumPy. *Nature* 585, 357–362 (2020). https://doi.org/10.1038/s41586-020-2649-2

[3] Johansson, R.: *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*, 2nd edn. Apress, New York (2019).

[4] VanderPlas, J.: *Python Data Science Handbook: Essential Tools for Working with Data*, 2nd edn. O'Reilly Media, Sebastopol, CA (2022).

[5] Oliphant, T.E.: *Guide to NumPy*, 2nd edn. Continuum Press, Austin, TX (2015).

Chapter 12: SciPy

[1] SciPy Community: SciPy 1.15.2 documentation. https://docs.scipy.org/doc/scipy-1.15.2/. Accessed July 22, 2026.

[2] Virtanen, P., Gommers, R., Oliphant, T.E., et al.: SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nat. Methods* 17, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2

[3] Johansson, R.: *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*, 2nd edn. Apress, New York (2019).

[4] Kiusalaas, J.: *Numerical Methods in Engineering with Python 3*, 3rd edn. Cambridge University Press, Cambridge (2013).

[5] Nocedal, J., Wright, S.J.: *Numerical Optimization*, 2nd edn. Springer, New York (2006).

Chapter 13: Graphical User Interfaces (GUIs)

[1] Kivy Organization: Kivy 2.3.1 documentation. https://kivy.org/doc/stable/. Accessed July 22, 2026.

[2] KivyMD Team: KivyMD documentation. https://kivymd.readthedocs.io/en/latest/. Accessed July 22, 2026.

[3] Python Software Foundation: tkinter—Python Interface to Tcl/Tk. https://docs.python.org/3.13/library/tkinter.html. Accessed July 22, 2026.

[4] Phillips, D.: *Creating Apps in Kivy: Mobile with Python*. O'Reilly Media, Sebastopol, CA (2014).

[5] Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S., Elmqvist, N., Diakopoulos, N.: *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 6th edn. Pearson, Boston (2016).

Chapter 14: Introduction to the Pandas Library for Engineering

[1] pandas Development Team: pandas documentation. https://pandas.pydata.org/docs/. Accessed July 22, 2026.

[2] McKinney, W.: *Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter*, 3rd edn. O'Reilly Media, Sebastopol, CA (2022).

[3] McKinney, W.: Data structures for statistical computing in Python. In: van der Walt, S., Millman, J. (eds.) *Proceedings of the 9th Python in Science Conference*, pp. 56–61 (2010). https://doi.org/10.25080/Majora-92bf1922-00a

[4] VanderPlas, J.: *Python Data Science Handbook: Essential Tools for Working with Data*, 2nd edn. O'Reilly Media, Sebastopol, CA (2022).

[5] openpyxl maintainers: openpyxl documentation. https://openpyxl.readthedocs.io/en/stable/. Accessed July 22, 2026.

Chapter 15: Control Systems with Python

[1] Python Control Systems Library Developers: Python Control Systems Library 0.10.2 documentation. https://python-control.readthedocs.io/en/0.10.2/. Accessed July 22, 2026.

[2] Fuller, S., Greiner, B., Moore, J., Murray, R.M., van Paassen, R., Yorke, R.: The Python Control Systems Library (python-control). In: *2021 60th IEEE Conference on Decision and Control (CDC)*, pp. 4875–4881. IEEE (2021). https://doi.org/10.1109/CDC45484.2021.9683368

[3] Åström, K.J., Murray, R.M.: *Feedback Systems: An Introduction for Scientists and Engineers*, 2nd edn. Princeton University Press, Princeton, NJ (2021).

[4] Ogata, K.: *Modern Control Engineering*, 5th edn. Prentice Hall, Upper Saddle River, NJ (2010).

[5] Nise, N.S.: *Control Systems Engineering*, 7th edn. Wiley, Hoboken, NJ (2015).

[6] SciPy Community: scipy.signal documentation. https://docs.scipy.org/doc/scipy-1.15.2/reference/signal.html. Accessed July 22, 2026.

Chapter 16: Databases Using SQLAlchemy

[1] SQLAlchemy Authors: SQLAlchemy 2.0 documentation. https://docs.sqlalchemy.org/en/20/. Accessed July 22, 2026.

[2] Myers, J., Copeland, R.: *Essential SQLAlchemy: Mapping Python to Databases*, 2nd edn. O'Reilly Media, Sebastopol, CA (2016).

[3] Bayer, M.: SQLAlchemy. In: Brown, A., Wilson, G. (eds.) *The Architecture of Open Source Applications*, vol. II. Lulu.com (2012). https://aosabook.org/en/v2/sqlalchemy.html. Accessed July 22, 2026.

[4] Beaulieu, A.: *Learning SQL: Generate, Manipulate, and Retrieve Data*, 3rd edn. O'Reilly Media, Sebastopol, CA (2020).

[5] SQLite Consortium: SQLite documentation. https://www.sqlite.org/docs.html. Accessed July 22, 2026.

[6] PostgreSQL Global Development Group: PostgreSQL documentation. https://www.postgresql.org/docs/. Accessed July 22, 2026.

Index

A

abs() function 77
Absolute imports 204–212
Absolute positioning 714–716
Abstraction 262, 271
Abstract method 289, 290
Abstract Syntax Tree (AST) 5
Access modifiers 95, 96, 279
Achievable canonical form 815
Ackermann's method 838, 839
add() method 144
*args parameter 220
Aggregation 275, 276
all() function 230
Anaconda 189
Anaconda navigator 189
AnchorLayout 711
Anonymous functions
  See Lambda expressions
Antoulas–Anderson algorithm (AAA) 627
any() function 230
append() method 118
Approximate_taylor_polynomial function 627
Argument-passing methods 223
Arguments 282
Arithmetic context 378, 379
Arithmetic operations 75, 102, 473–475
Arrays
  accessing elements 461–465
  creation patterns 454–461
  dimensions 451
  memory layout and indexing structure 452

multidimensional arrays and matrices 450–454
operations 470
aggregation and reduction 482–487
arithmetic operations 473–475
binary arithmetic with broadcasting 480, 481
complex numbers 481, 482
linear algebra 494–499
logical, boolean, comparison, sorting, searching and set operations 490–493
manipulation 499–504
numerical stability 478, 480
NumPy families 470–472
rounding and approximation 477, 478
trigonometric, hyperbolic, exponential and logarithmic functions 487–490
unary arithmetic operations 475–477
properties 465
information and utility functions 466–468
inspection and utility functions 469, 470
Python objects 450
Assignment operator (=) 71
Association 274, 275
Astrophysics 558
astype() method 779
Attributes 264, 282, 284
class attributes 278
dynamic and static attributes 281, 282
instance attributes 278
private and protected 279, 280
property attributes 282
public 278
types 277
Audio signals 539
Automation 178

ax.grid() function 418
Axisymmetric problems 558
ax.legend() function 418
ax.tick_params() 418

B

Backend control functions
  data flow 571
  error handling 573
  functions 572, 573
  libraries 571
  mathematical equivalence 574
  parameters 572
  scientific computing 571
  scipy.fft 570
  set_backend() function 570
  set_global_backend() 570
Backward difference 533
Band-pass filter 586, 587
Bartels–Stewart algorithm 649
Base class 264
Base code 304
Bessel functions 558
bit_length() method 77
Bitwise operations 75, 103
Black-box functions 525
Bound constraints methods 661
BoxLayout 706
Break statement 250, 252
Broadcasting 480, 481, 504, 505
Broyden–Fletcher–Goldfarb–Shanno (BFGS) method 657
B-spline 607, 608, 622–624
Built-in functions 77, 78, 226
Built-in modules 186, 187
Bytecode 5

C

Calling function 216
Canonical_form function 815–817
Central difference 533
Centrum Wiskunde & Informatica (CWI) 1
Chained comparisons 81
CHANGELOG.md file 12
Chemical engineering 182
Chirp signal 675
Cholesky decomposition 644
Circulant matrix 651
Clarkson–Woodruff transform 646
Classes
  argument 266, 267
  attributes 278, 281
  coding 265
  color_parameter 265
  defined 264
  def keyword 266
  __init__ method 265
  instances 264, 298
  lines 266
  method 264
  subclass 268–270
  syntax 264
  vars() 267
Class methods
  alternative constructor 285, 286
  class-level operations 286, 287
  @classmethod decorator 285
  factory methods 286
  inheritance and polymorphism 287, 288
  use cases 285
  utility methods 288, 289

Class *vs*. instance attributes 279
clear() method 120, 155
Client code 304
cls method 302, 303
Cmath
  classification functions test 376
  cmath.sqrt() 371
  complex numbers 369, 372
  complex plane 370, 371
  complex-specific shortcuts 376
  direct and inverse functions 374, 375
  exponentials 373
  functions 369–370
  library 372
cmath.infj() function 377
cmath.isclose() function 376
cmath.isfinite() function 376
cmath.isinf() function 376
cmath.isnan() function 376
cmath.nanj() function 377
cmath.phase() function 372
cmath.polar() function 372
cmath.rect() function 372
cmath.sqrt() function 371
Code editor 37–39
Collections
  comparison 97, 98
  types 96, 97
Colon operator (:) 84
color_parameter 279
Command-line interface (CLI) 693, 694
Comma-separated values (CSV)
  commands 329
  csv.reader() 331, 332

file.read() 330
list 332
list.insert() 333
operations 329
plain-text format 327
readlines() 330
readlines()[line_index] 331
sensor data 327
specific cell 334
structured data 328
SyntaxError 328
writerow() 332
zero-based index 333
commit() 893
Communication signals 539
Community-defined PEPs 7
Comparison operators 81–83
Comparisons (==, !=, <, >) 141
Compilation 54
Compiled modules 195–197
Compiled Python files (.pyc) 8
Complex numbers 481, 482
Composite trapezoidal rule 592
Composition 276, 277
Compression 550
Computer-aided design (CAD) 622, 886
Concatenation (+) method 136, 141
Concatenation operator 124
Conditional statements 228, 229
if statement with elif 239
if statement with else 238, 239
simplified if statement 237, 238
syntax 237
Connection strings 890

Constrained optimization 653, 661
Constrained optimization strategies 657
Constrained *vs*. unconstrained optimization 661
Constructor method 278, 298–301
Continue statement 251–253
Continuous integration and delivery (CI/CD) pipelines 10
Continuous Lyapunov equation 649
Continuous-time LTI system 676
Controllability matrix 842, 843
Controllable canonical form 815
Controller design
  characteristics 836
  pole placement 838, 839
  root locus plot 837, 838
Control points 608
Control statements
  break keyword 250
  continue keyword 251
  loops
    break keyword 252
    continue keyword 252
    pass keyword 253
Control systems 180
  command distribution 805
  features 803, 804
  frequency-domain analysis 827–835
  functionality 801, 803
  mathematical models 803
  MATLAB 805, 806
  model conversions and interconnections 815–819
  nonlinear Systems 839–842
  online documentation 807–814
  performance goals 803
  PID controller design 843–867

python-control package 806
system properties 835, 836
time-domain analysis 819–827
tools 801
Control theory 647, 649, 676, 842, 843
Conversion methods 76
convert_temperature() function 523
Convolution 673
copy() method 122, 155
Correctness 176
count() method 123, 137
CPython 5
Create, Read, Update, and Delete (CRUD)
correcting erroneous measurements 900
functions 898
logging new sensor readings 899
querying engineering data 899, 900
removing obsolete records 900, 901
structured and intuitive interface 898
create_engine() function 889, 891, 892
ct.forced_response() function 820, 821
ct.nlsys function 813
CubicSpline 611, 617–619
Curve fitting 653, 668–671
Custom modules 194, 195, 210
Custom scales 421, 432, 433, 435

D

Data analysis 181
Databases
engineering applications 885
types and use cases 886
DataFrame 763, 767
column selection, series *vs.* 770, 771
concatenation 788, 789

merge and join 786–788
row filtering with logical conditions 772, 773
sorting data 784–786
Data models
abstractions 895
declarative style 895
defined 895
ORM models and table mapping 896
relationships 895, 897, 898
sensor reading model definition 896, 897
table creation 897, 898
turbine and SensorReading models 896
turbine model definition 896
Data science 178
Data types
explicit type conversion 88–94
type hints 88
Data visualization 622
defined 791
engineering line and scatter plots 794–796
histograms and bar charts 797–800
line and scatter plots 792–794
peak points plot 796
Debugger 37
Decimal fractions
arithmetic context 378, 379
base-10 values 378
binary float 377
examples
avoiding float inaccuracy 382
mathematical functions 385
monetary calculations 383
precision and arithmetic 381
rounding modes 383

special values 385
temporary context 384
preconfigured contexts 380
preconfigured decimal.BasicContext 381
preconfigured decimal.ExtendedContext 381
printed values 378
rounding functions floor, ceil and trunc 380
rounding modes 379
Decomposition 640–646
Decorators 232–235, 283, 308, 309
Default arguments 219
delete() method 120
De-noising 550
Dependency breakage 58
Dependency injection
defined 309, 311
design patterns 310, 311
flexibility and testability 311
implementation 310
modular and testable code 311
Derivative-free methods 657
Derived class 268
Descriptive statistics 781, 782
Design patterns 310–311
det() function 634
df.dropna() 776
df.fillna() 776
df.isnull() 776
Dictionary
accessing and modifying data 147, 148
add/update item 150
checking key 151
clear() method 155
comprehensions 156, 157

copy() method 155
creation 147
defined 146
deleting item 150
get length 151
get() method 152
immutable types 146
items() method 153
keys() method 152
key–value pairs 147
key–value pair structure and core properties 146
lambda expressions 229
mapping sensor readings 146
operations 148, 149, 157–159
pop() method 154
popitem() method 155
setdefault() method 156
unique keys 146
update() method 154
values() method 153
Differential_evolution function 662, 664–667
Differentiation
derivative function
error approximation 529
error estimation 529, 530
finite difference method 531–533
parameters 526, 527
preserve shape 534
RichResult object 528
finite differences 525
operations 525, 526
Digital signal processing (DSP) 543, 589–590
discard() method 144
Discrete cosine transform (DCT) 552–557

Discrete Fourier transform (DFT) 539, 542, 550, 566
Discrete sine transform (DST) 552, 553
drop() method 775
dunder dict 267
Dunder methods 291–295
  applications 290
  attribute access 295
  callable objects 297
  comparison operations 297
  container methods 296
  defined 290
  numeric operations 297
  object lifecycle 295
Dynamic attributes 281
Dynamic import 203
Dynamic modules 192, 194
Dynamic system control 801–807

E

Eigenvalue problems 635–639
Eigenvectors 636, 638
eig() function 636, 637, 639
Electrical engineering 182
Embedded software 254
Emojis 40–42
Encapsulation 262, 270
Engineering analytics 886
Engineering calculations 179
Environment variable (.env) files 11
Equality operator 126
Error handling 573
Error messages 129
Exception handling 254–257
Explicit conversions 76
Explicit typing with type hints 88

expm() function 648
extend() method 118, 119

F

Fast Fourier Transforms (FFTs)
  applications 575
  backend control 570–583
  DST and DCT 552–557
  FHT 558–562
  filtering 583–588
  functions 537, 538, 576
  helper functions 562–570
  N-dimensional Fourier transforms 551, 552
  1D Fourier Transforms 539–549
  scipy.fft submodule 536
  specialized Fourier transforms 537
  2D Fourier transforms 550–551
Fast Hankel Transform (FHT) 558–562
Feature extraction 550
File-path syntax 313
File types 313–316
Filter design 672
filter() function 226
Finite binary precision 73
Finite difference method 530–533
Finite element analysis 181
Finite impulse response (FIR) 672, 685, 686
First-order ODE 601, 603
FloatLayout 709
Fluid dynamics 182
Flutter framework 697
Forced-response analysis 820
Foreign function interface (FFI) 6
Foreign key enforcement 898
For loop 244–249

Formatted string literals
See F-strings
Forward difference 533
Fractions
arithmetic operators 389
conversion and inspection methods 390
decimal places 389
examples 387, 388
numerator and denominator attributes 389
objects 386
operations/methods 386–387
Python documentation 386
Frequency-domain analysis
Bode plot 828, 829
characteristics 827
gain and phase margins 833, 834
gain drops 832
Nichols plot 830, 831
Nyquist plot 829, 830
stability margins and frequencies 834, 835
zero frequency 832, 833
Frequency domain spectrum 545
Frequency response analysis 844
Frequency response data (FRD) model 803, 810, 811
Frequency response PID design 874, 876–884
F-strings 44–46, 48
Function FFT 539–545
Functional programming 2
Function-based *vs*. class-based decorators 308
Functions
comments 214
decorators 232–235
def keyword 214
defined 213

example 214
lambda expressions 223–232
parameters and arguments 215–223
return keyword 214
syntax 213

G

Gain crossover frequency 833
Game development 178
get_color method 305
getcontext() 378
get() method 148, 152
Getter 305
getvalue() method 770
GitHub octoverse report 3
.gitignore file 10
Global optimization 662–667
Google Colab 11
Gradient-based methods 660
Gradient-based techniques 657
Graphical user interfaces (GUIs)
absolute positioning 714–716
ecosystem 695–697
fundamentals 693–695
Kivy 697–703, 720–751
layouts 705–714
layout-specific positioning 719
non-technical users 695
relative positioning 716–718
temperature converter 751–757
visual elements 694
Graphs 399
griddata() 619, 620, 622
Gridded data 607
GridLayout 707

groupby() method 782–784

H

Hankel matrix 651
Hankel transforms 558
Helper functions
  description 563
  fftfreq 568–570
  fftshift and ifftshift 565–568
  frequency analysis 563
  goals 562
  intuitive visualization 563
  power systems 564
  random noise 564
  scipy.fft 566
  technical areas 563
  tools 562
Hermite basis functions 614
Hermitian symmetry 547, 548, 550, 551
Hessian-based methods 660
Hessian matrix 525, 535, 536
hfft() function 549
Higher-dimensional tensors 451
High-level functionality 271
High-pass filter 584, 585

I

Image filtering 550
Image processing 550
Immutability 84
importlib.import_module() 203
Impulse response of linear system 822–824
Indentation 44
Indentation-based syntax 10
Indexing errors 465

index() method 117, 118, 134, 135
Industrial automation 254
Infinite impulse response (IIR) 672
info() method 769
Informational PEPs 7
Inheritance 259, 262, 271, 272, 287, 288
.ini/.cfg/.config files 9
input() function 44
insert() method 119
Instance attributes 278
Instance method
  attributes 284
  defined 283
  methods with arguments 284
int() function 104
integrate.dblquad() function 598
Integrated development environment (IDE) 8, 37–39
integrate.quad() function 595
International IEEE 754 standard 72
Interpolation
  advanced tools 610
  B-spline 622–624
  categories 611
  components 607, 608
  cubic polynomial 617–619
  defined 606
  functions and classes 612–614
  geospatial analysis 608
  grid pattern 607
  methods 610, 611
  multivariate 607, 619, 620, 622
  one-dimensional visual comparison 610
  rational approximation 609
  rational approximation and additional tools 627, 628

steps 609
2D spline 625–627
univariate 606, 614–616
use cases and dimensionality 606
Inverse fast Fourier transform (IFFT) 546, 547
Inverse fast Hankel transform 560
Inverse hyperbolic sine 421
Inverse sine hyperbolic scale 430–432
io.StringIO() function 769
items() method 153

J

Jacobian matrix 525, 534, 535
JavaScript Object Notation (JSON) 10, 11
JetBrains state of developer ecosystem survey 3
Jump statements
See Control statements
JupyterLab 11
Jupyter Notebook (.ipynb) files 11

K

keys() method 152
Keyword arguments 217, 218
Keyword-only arguments 222
Keyword-only parameter separator 622
Kivy framework
application 702, 703
architecture 701
button 729, 730
CheckBox 731–733
components 699, 700
image 746–750
installation and modules 697–699
label 735–737
popup 743–745

ProgressBar 733–735
purpose 696, 697
slider 737–739
spinner 745, 746
switch 742, 743
TextInput 739–741
widgets and events 720–729
KivyMD 697
Knots 607, 608
**kwargs parameter 220, 221

L

Lagrange approximations 610
Lagrange function 627
Lambda expressions
any() and all() 230, 231
arguments and expressions 224
assigning variable 225
built-in functions 226, 227
conditional statements 228, 229
defined 223
dictionaries 229
example 224
functional programming 224
list comprehension 228
methods and use cases 225
regular function 230
sorted() with multiple keys 231, 232
sorting 227
syntax 224
Layouts
advanced layouts 711–714
managers 706–711
responsive and scalable interfaces 705
screen sizes 705

Layout-specific positioning 719
LDLt (LDL transpose) decomposition 643
Least squares 668–671
Least-squares optimization 653
len() method 117, 134, 151
Lexical analysis 4
Linalg
  decomposition 640–646
  determinant calculation 634, 635
  eigenvalue problems 628, 635–639
  functions 629, 630
  groups 628
  inverse matrix 631, 632
  matrix decompositions 628
  matrix functions and equation solvers 646–650
  matrix structures 633
  operations 629
  optimization 653–671
  parameters 631, 632
  scipy.linalg subpackage 628, 629
  solving systems 632–634
  special matrix functions 650–652
Linear algebra functions 494–499
Linear interpolation method 610
Linear programming 653
Linear quadratic regulator (LQR) 647
Linear scale 420, 423, 424
Linear time–invariant (LTI) 676, 688, 862
Linspace function 540
linthresh parameter 427
List of useful commands 48–51
Lists
  accessing and modifying elements 114
  append() and remove() 118

clear() and delete() 120
comprehensions 129–131, 169, 228
concatenation operator 124
copy () 122
count() 123
creation 113, 114
data types 113
equality operator 126
error messages 129
extend() 118, 119
index() 117, 118
insert() 119
len() 117
max() and min() 123
membership operators 124
methods 115–116
nested 128
operations 127
operators 116
pop() 119
remove duplicates 122
repetition operator 125
slicing operator 125
sort () and reverse() 121
sum() 124
visual representation 114
localcontext() 378
Local multivariate optimization 655–662
Local *vs*. global minima 663
Logarithmic scale 420, 425, 426
Logit scale 421, 428–430
LowLevelCallable object 597
Low-pass filter 583, 584
LU (lower–upper) decomposition 641

Lyapunov equations 647

M

Machine learning 178, 181
Magic methods 290
Makefile 11
make_splprep function 622, 623
MANIFEST.in file 10
Mantissa 72
Many-to-many relationships 273
Many-to-one relationships 273
map() function 226
Math
  angular conversion 367
  built-in modules 359, 362
  floating-point arithmetic 365
  floating-point manipulation functions 365, 366
  functions 362–364
  hyperbolic functions 368
  operations 359–361
  power, exponential and logarithmic functions 366, 367
  routine scientific calculations 364
  special functions 368
  summation and product functions 367
  trigonometric functions 368
Math_decorator 234
Mathematical computation 359
MATLAB compatibilities 677, 678
Matplotlib 399
  applications 409
  bar chart 405
  box plot 406
  commands 401
  contourf 407
  customization 414–415, 420

engineering analysis 401
error bar 406
examples 405, 407
flowchart 411
functions 401–404
grid and ticks 418, 419
histogram 406
legends 418
line plot 405
lines and spines 416
numerical operations 416
object-oriented approach 408
scales 420–435
scatter plot 405
simple graph example 411–413
statistical plots 406
stem plot 407
steps 409, 410
streamplot 407
text and labels 417, 418
3D plots 408
3D scatter 408
3D surface 408
3D wireframe 408
workflows 401
Matrix exponential 648
Matrix operations 505
max() method 123, 139
Mechanical engineering 182
Membership operators 124
Membership testing 108
Memory management 5
data types 509
delete arrays 511, 512

memory-mapped arrays 510, 511
numerical operations 509
Memory-mapped arrays 510, 511
min() method 123, 139
Mini-project—inventory system 174
Miscellaneous 179
MongoDB 886
Monte Carlo simulations 644
Multidimensional arrays 450–454
Multivariate interpolation 607
Multivariate normal distributions 644
MySQL 886, 889

N

Naming conventions 95
N-dimensional discrete Fourier transform (FFTN) 552
N-dimensional Fourier transforms 551, 552
Nearest-neighbor method 620
Neo4j 886
Nested list 128
Nested tuples 140
Networking 178
Newline (n) 39
Nichols plot 830, 831
Nonlinear pendulum model 812
Nonlinear pendulum system 827
Nonlinear restoring torque 813
Nonlinear systems 812–814
input/output system 841
operating point 839, 840
physical processes 839
NoSQL databases 886
Notch filter 587, 588
Numerical analysis 180
Numerical integration 590

Numerical Python (NumPy) 188
arrays
See Arrays
best practices
additional tips 514, 516
built-in functions 513, 514
contiguous arrays 514
memory management 509–512
preallocation arrays 512, 513
vectorized operations 508
boolean and integer values 443
components 437
data types
boolean 446
bytes types 447
complex numbers 447
datetime 448
floating-point numbers 447
object type 448
signed and unsigned integers 446
string type 447
structured types 449
timedelta 449
UTF-8 strings 448
void types 449
defined 437
installation and version check 439
modern processors 445
numerical infrastructure 446
operations 437, 438
scientific and engineering applications 444, 445
SciPy 439–442
StringDType 446
subpackages 442, 443

user errors
broadcasting and shape compatibility 504, 505
indexing errors 506
memory errors 506
type conversion issues 506
view *vs*. copy 506, 507
Nyquist frequency 543, 544
Nyquist plot 829, 830
Nyquist–Shannon sampling theorem 543

O

Object-oriented programming (OOP) 232
aims 262
attributes 277–282
C# language 261
classes 264–270
coding 263
concepts 259, 262
decorator 308, 309
dependency injection 309–312
methods 263, 282–308
pillars 259, 270–273
principles 262
problems 260
relationships between objects 273–277
Object-relational mapping (ORM) 887, 895
Objects 264, 298
One-to-many relationships 273, 898
One-to-one relationships 273
open() function
append mode 323
binary mode 323
byte- and text-level access 316
commands 319, 320
context manager 318

exclusive creation mode 322
file manipulation modes 318
multiple line 326
multithreaded programs 319
operations 320, 321
read and write mode 324, 325
read mode 322
specific line 325, 326
text-based formats 317
text mode 323, 324
write mode 321, 322
Operator precedence 111
Optimization 181
Ordered collections 97
Ordinary differential equation (ODE)
first-order 601, 603
goals 600
second-order 604, 605
slope method 601
steps 600, 601
Ordinary differential equations (ODEs) 590
Orthogonality 555
os.getpid() function 511

P

Pade function 627
PageLayout 713
Pandas library
command families 760, 761
data-driven models and simulations 759
DataFrame 770–773, 784–789
data manipulation and cleaning
changing data types 779, 780
creating new columns 773–775
missing values (NaN) 776–778

removing columns 775, 776
data visualization 791–799
exporting data
CSV 790
Excel 790, 791
formats 791
multiple formats 789
functionality 760
inspecting and exploring data 767–770
installation
commands 761
data structures 762–764
overview 759
practical engineering perspective 759
reading and importing data
external sources 764
reading CSV files 764, 765
reading Excel files 765–767
statistical analysis and grouping 781–784
tools 759
Parallel connection of I/O systems 818, 819
Parameters 282
Parameter separator 622
Parent class 264, 268
Parsing 5
Part-of relationship 276
PascalCase 265
Pass statement 253, 254
path.exists() 353
pathlib
absolute path 358
checking file/directory 353
checking path exists 353
creating directories 354

creating path object 352
deleting files and directories 355
filesystem paths 351
finding files with glob 357
getting file size 356
getting last modified time 356
getting parent directory 352
getting path components 353
joining paths 352
operations 351
reading and writing files 354
recursive glob 357
relative path 358
renaming and moving files 354, 355
Pchip 610
PchipInterpolator class 615
pd.concat() 788
pd.merge() function 786
Periodogram function 682–684
Phase crossover frequency 833
Phase margin (PM) 882
Piecewise cubic hermite interpolating polynomial (Pchip) 614–616
Piecewise cubic polynomials 614
plt.show() command 412
plt.subplots() 411
Poetry 52
Pole allocation
augmented system 863
defined 844
desired poles and compute gain 864
equivalent PID gains 864, 865, 867
first-order systems 851
PID design 850
second-order systems 851, 853–858

simple PID analyzer 845
state-space 858, 860–863
state-space matrices 847–850
system dynamics 859
transfer function 845–847
Pole allocation method 844
Pole allocation technique 850
Pole placement 850
Polymorphism 259, 262, 272, 273, 287, 288
popitem() method 155
pop() method 119, 154
Popularity of programming language (PYPL) Index 3
pos_hint property 716
Positional arguments 217
Positional-only arguments 222
PostgreSQL 886, 889
Power spectral density (PSD) 682–684
print() function 39–42
Private attributes 280
Procedural programming 2
Programming paradigms 260, 261
Project explorer 38
Property attributes 282
@property decorator 305, 306
Property method 304–308
Proportional–integral–derivative (PID) controller
defined 843
methods 843
pole allocation 844–867
root locus method 867–884
Protected *vs.* private attributes 280
Psutil library 511
psutil.Process() function 511
pvariance() 394

PyCharm 38
.py files 8
Pyplot 409
pyproject.toml file 9
Python
  choosing code editor 37–39
  community 177
  data keyboard 44
  definition 1
  ecosystem and community 4
  exercises 60–70
  files and folders 8–13
  first program 39
  folder organization 13–16
  history 1–3
  inner workings 4–6
  installation
    aliases configuration 25
    aliases setting 26
    environment variable configuration 27, 28
    environment variable entry 28
    Linux 33
    location 24
    macOS 29–32
    operating systems 17
    PATH and alias issues on Windows 24, 26
    software license and permission request 30
    terminal output 23
    virtual environment 34–37
    Windows 17–23
  libraries 400, 695, 696
  modules
    See Python modules
  multi-paradigm 2

operation flowchart 6
packages 178–182
path and modules location 183, 185, 186
print() function 39–42
project management 51–59
source files (.py) 8
structure 44
Python Enhancement Proposals (PEPs) 7
Python Launcher 19
Python modules 177
absolute and relative import 204–212
advanced and special-case import methods 200
components 202
dynamic import 203
import methods 197–199
namespace binding 200
styles 201
types 186
built-in modules 186, 187
compiled modules 195–197
custom modules 194, 195
dynamic modules 192, 194
standard library 187
third-party modules 188–192
wildcard import 201, 202
Python Package Index (PyPI) 9, 183
Python Software Foundation (PSF) 2
Python Virtual Machine (PVM) 2, 5

Q

QR (orthogonal–triangular) decomposition 643
Quasi-Newton methods 660
QZ (quasi-triangular) decomposition 644

R

Radial basis function (RBF) 609
Random module 396–398
range() function 246, 247
Reachable canonical form 816
README.md file 12
Real-input fast Fourier transform (RFFT) 547–549
RectBivariateSpline class 625
reduce() function 227
register_backend() function 570
Relational databases 886
Relative imports 204–212
RelativeLayout 710
Relative positioning 716–718
remove() method 118, 144
Repetition (*) method 136
Repetition operator 125
Representation methods 76
requirements.txt 10
Return keyword 215
reverse() method 121, 138
rfft() function 549
.rglob() method 357
Riccati equations 647
Robotics 181, 254
rollback() 893
Root locus method 843
  closed-loop poles 867
  control theory technique 869, 870
  frequency response PID design 874, 876–884
  gain selection 871
  PID controller design 868
  PID controller parameters 872–874
  plant and controller structure 870, 871
Root locus plot 837, 838

Rosenbrock function 657–659
round() function 74, 77
Row filtering 772, 773
RQ (triangular–orthogonal) decomposition 644
Rs 566

S

Scales
  axis 421
  custom scales 432, 433, 435
  data visualization 420
  function prototypes and parameters 422, 423
  inverse sine hyperbolic scale 430–432
  linear scale 423, 424
  logarithmic scale 425, 426
  logit scale 428–430
  symmetric logarithmic scale 426–428
ScatterLayout 713, 714
Schur decomposition 644
Scientific computing 178
Scientific Python (SciPy) 439–441
  defined 517
  FFTs
    See Fast Fourier Transforms (FFTs)
  integration
    absolute error 596
    coding 595
    double integral 598, 599
    functions 592–593
    ODEs 600–605
    ODE solvers 590
    parameters 594
    performance 597
    return values 595
    trapezoidal rule 591

interpolate
See Interpolation
linalg 628–651
overview 517
signal 672–692
subpackages 518, 519
constants 519–525
differentiation 525–536
Hessian matrix 535, 536
Jacobian matrix 534, 535
tasks 517
scipy.constants subpackage
defined 519
physical and mathematical constants 520, 521
SI prefixes 521–523
unit conversion 523–525
scipy.fft.dct function 554
scipy.fft.dct() function 555
scipy.fft.fftfreq function 569
scipy.fft.fft function 540
scipy.fft.fftshift function 567
scipy.fft.fht function 560
scipy.integrate.dblquad function 598
scipy.optimize.minimize function 655, 656
Scripting 178
Second-order ODE 604, 605
Security 178
self method 302, 303
Sensor data 539
Series connection of I/O systems 817, 818
Sessionmaker function 894
Sessions
declarative base and ORM infrastructure 894
defined 892

lifecycle 892
ORM features 892
sessionmaker 894
transactional control 892
transactions 893
set() constructor 142
set_backend() function 570
setdefault() method 156
Sets
adding and removing elements 144
coding 142
operations 143–145
purpose 141
types 141
unique elements 142
unordered nature and storage 142
updating elements 144
Setter 305
setup.py file 9
Setuptools 196
Short-time Fourier transform (STFT) 676
Signal processing 180
1D convolution process 673
categories 674
chirp signal and spectrogram 675
convolution and correlation 672
filter design 679–682
FIR filter 685, 686
functions 672
LTI system conversions 687–692
MATLAB compatibilities 677, 678
methods 676
parameters 679
plots 675–677

PSD 682–684
stages 673
Simulations 179
Singular value decomposition (SVD) 642, 643, 645
Slicing method 137
Slicing operator 125
Smoothing 626
sort() method 121, 138
Sorting 227
Sorting comparison 167
Spectral analysis 672
Spectrogram 675
Spinner 745, 746
SQLAlchemy 885
abstraction 889
components 901
connection strings and database dialects 890, 891
CRUD 898–901
data models 895–898
defined 885
engine 889
engineering workflows 888
full wind turbine monitoring script 902–904
installation 888
overview 887, 888
production systems 885
self-contained example 901
sessions 892–895
SQLite 891
wind turbine monitoring database 891, 892
SQLite 886, 889, 891
Stability analysis 649
Stable math helper functions 478
StackLayout 711

Stack overflow developer survey 3
Standard library modules 187
start_engine() method 273
Start parameter 246
State-based paradigm 409
State-feedback controllers 815
State-space (SS) 688, 690–692, 803, 807, 808, 815, 847–850, 858, 860–863
Static attributes 281
Static methods 289
Statistics
  datasets 392, 393
  dispersion measures 394
  functions 391
  output 395
  packages 390
  population *vs*. sample 393
  Python documentation 390
  simple regression line 395
Step parameter 246
Stop parameter 246
StringDType 446
Strings
  declaration 83
  examples 85, 87
  immutability 84
  indexing and slicing 84
  methods 106
  operations 84, 85
.strip() method 106
Structural engineering 182
Structured query language (SQL) 886
Subclass 268–270, 289
sum() method 124, 139

Superclass 264, 268
super() function 301, 302
Supply chain management 886
Sylvester equation 647
Symbols 40–42
Symmetric logarithmic (symlog) scale 421, 426–428
Syntactic analysis 5
System creation
  continuous-time systems 807
  FRD model 810, 811
  nonlinear system 812–814
  state-space system 807, 808
  transfer function 809, 810
  zeros, poles and gain 814

T

Taylor approximations 610
Temperature converter
  complete application code 752, 753
  functions 756
  GUI layout 755, 756
  importing modules 754
  project goal and architecture 751
  TemperatureConverterApp class 757
  TemperatureConverter class 754
TemperatureConverter 751, 754
TemperatureConverterApp 751, 757
Temperature sensor 762
Template strings
  See T-strings
2D discrete Fourier transform (FFT2) 550, 551
Thermodynamics 182
Third-party modules
  anaconda 189
  defined 188

package managers 188, 189
source code
editable mode 192
Git and version control systems 190
Git repository 190
local source code 191
poetry 192
source archives 191
version control platform 190
wheel file 192
Time-domain analysis
control engineering 819
forced response of linear system 820–822
impulse response of linear system 822–824
performance characteristics 820
phase plane diagram 825–827
step response 824, 825
Time domain signal graph 545
Time-frequency domain 675
Time-series data 539
TIOBE index 3
Toeplitz matrix 651
Tokens 4
Transfer functions (TF) 688, 803, 809, 810, 815, 845–847, 853
Trust-region methods 660
T-strings 46–48
Tuples
accessing elements 132
concatenation (+) 136
converting list 140
count() 137
immutability and ordered structure 131
index() 134, 135
len() 134

max() and min() 139
membership operators 137
memory-efficient 131
methods 131–134
modification error 170
nested 140
operations 140, 141, 171
repetition (*) 136
slicing 137
sorted() 138
sum() 139
types 131
Type conversions 88–94, 103
Type 2 discrete cosine transform (DCT-II) 555
Type hints 109
type() method 73

U

UltraViolet (UV)
benefits 52
command-line tool 51
commands 59, 60
innovations 51
installation
command 52
dependency management 54, 55
Linux distribution/macOS 53
macOS with Homebrew 53
operating systems 54
pip/pipx 54
PowerShell 52
Windows 53
lockfile 58
medium- and large-scale projects 51
tools 51

workflow
adding and installing dependencies 56
generating/updating lockfile 58
isolated environment 56, 57
starting new project 55
synchronizing lockfile 57, 58
Unary arithmetic operations 475–477
Unconstrained optimization 653
Unicode characters 41, 42
Unicode examples 42–43
Unit conversion 523–525
Univariate interpolation 606
Unpacking arguments 222, 223
Unsigned integers (uintX) 446
update() method 154
upper() method 263
User-created modules 208
User—membership test 167
Utility methods 77, 288, 289

V

ValueError 255, 306, 505
values() method 153
Variable-length arguments 217
Variable-length keyword arguments 220, 221
Variable-length positional arguments 219, 220
Variables
access modifiers 95, 96
collections 96–98
comparison operators 81–87
converting data types 88–95
defined 71
exercises 98–112
floating-point arithmetic error 74
integers and floating numbers 73, 74

languages 72
local and global 95
mathematical and utility operators 75–80
types 72
Vectorized operations 508
Version control integration 38
Virtual environments 17, 192
creation 35–37
vs. global operating mode 34
purpose 34
Python development 34
with venv 35
workspace development 34
Visualization 544
Visual Studio (VS) Code 11, 38

W

Web development 178
While loops 240–244
Word and Excel documents
.docx operations 335
appending 336
appending specific paragraph 340
counting paragraph, tables and images 341
extracting paragraph 341
getting font information 337, 338
identifying heading paragraph 341
reading 335
reading specific paragraph 339
replacing specific paragraph 339
translating text 338
writing 336
external libraries 334

X

.xlsx file format
  appending 348, 349
  compatibility and security 342
  count non-empty rows and columns 350
  examples 342
  getting information 349
  non-empty rows/columns 346, 347
  openpyxl library 342, 343
  range of cells 347
  reading 343
  reading all sheets 344
  reading specific cell 347
  reading specific column 345, 346
  reading specific row 345
  reading specific sheets 344
  spreadsheet data 342
  writing 348

Y

YAML (YAML Ain't Markup Language) 10
.yaml/.yml files 10

Z

Zero frequency 832, 833
Zero–pole–gain 688–690
zip() function 226

Table 12-24 Simplified return values for scipy.integrate.quad when full_output=False

Return Value	Data Type	Description
y	float	The computed integral result
abserr	float	Estimate of the absolute error

Table 11-6 Array creation functions, commands, and initialization patterns

Operation Type	Command	Description
From a Python list/tuple	np.array([1, 2, 3])	Creates a NumPy array from a Python sequence. NumPy chooses an appropriate dtype automatically.
From Python data (no copy if possible)	np.asarray(my_list)	Converts input to an array, avoiding unnecessary copies; useful when wrapping existing data.
With an explicit dtype	np.array([1, 2, 3], dtype=np.float64)	Creates an array with a specified numeric dtype (e.g., for precision control in engineering).
Zeros	np.zeros((3, 3))	Creates a 3 × 3 array filled with 0.0. Often used for initialization of matrices or state vectors.
Ones	np.ones((2, 4), dtype=np.int32)	Creates a 2 × 4 array filled with 1, with a given integer dtype.
Constant value (full)	np.full((2, 3), 7.5)	Creates a 2 × 3 array where every element is 7.5. Useful for offsets, bias terms, or masks.
Like another array	np.zeros_like(a)	Creates an array of zeros with the same shape and dtype as a.
Identity matrix	np.eye(3)	Creates a 3 × 3 identity matrix (ones on the diagonal, zeros elsewhere).
Diagonal from a 1D array	np.diag([1, 2, 3])	Builds a square matrix with the given 1D array on the main diagonal.
Arange (integer steps)	np.arange(1, 10, 2)	Creates values from 1 to 9 with step 2. Good for integer ranges and indexing patterns.
Linspace (evenly spaced)	np.linspace(0, 10, 5)	Creates five evenly spaced values from 0.0 to 10.0 (inclusive). Useful for time grids, sampling, etc.
Geometric spacing	np.geomspace(1, 1000, 4)	Creates values spaced geometrically (multiplicative steps), for example, for logarithmic scales.
Logarithmic spacing	np.logspace(0, 3, 4)	Creates values from 10^0 to 10^3 in log scale. Helpful for gain, frequency, or scale studies.
From a function (index-based)	np.fromfunction(lambda i, j: i + j, (3, 3), dtype=int)	Creates an array by calling a function on indexes; good for structured patterns (e.g., grids, coefficients).
Random uniform (modern API)	rng = np.random.default_rng()	Creates a modern Generator instance (recommended API in NumPy 2.x).
	rng.random((3, 3))	3 × 3 array of random floats in [0.0, 1.0).

Operation Type	Command	Description
Random integers (modern API)	rng.integers(0, 10, size=(2, 4))	2 × 4 array of random integers in [0, 10).
Random normal (modern API)	rng.normal(loc=0.0, scale=1.0, size=(2, 2))	2 × 2 array of normally distributed values (μ = 0, σ = 1).
From an existing shape with random	rng.random(a.shape)	Random array with the same shape as a.
Reshape after creation	np.arange(6).reshape(2, 3)	Creates a 1D array and reshapes it to (2, 3), keeping the same data in a new structure.

Table 12-44 Parameters for the scipy.linalg.svd function

Parameter	Type	Description	Default
a	(M, N) array_like	Matrix to decompose.	–
full_matrices (optional)	bool	Determine shape of U and Vh.	True
compute_uv (optional)	bool	Compute U and Vh in addition to s.	True
overwrite_a (optional)	bool	Overwrite a for performance gain.	False
check_finite (optional)	bool	Check for finite numbers in input matrix.	True
lapack_driver (optional)	{'gesdd', 'gesvd'}	Choose SVD computation approach.	'gesdd'

Table 12-38 Parameters for the scipy.linalg.solve function

Parameter	Type	Description	Default
a	(N, N) array_like	Square input data.	–
b	(N, NRHS) array_like	Input data for the right-hand side.	–
lower (optional)	bool	Use only lower/upper triangle of a (depends on assume_a).	False
overwrite_a (optional)	bool	Allow overwriting data in a (may enhance performance).	False
overwrite_b (optional)	bool	Allow overwriting data in b (may enhance performance).	False
check_finite (optional)	bool	Check if input matrices contain only finite numbers.	True
assume_a	str, optional	Specify structure of a (e.g., symmetric, positive).	None (auto-detect)
transposed	bool	Solve a.T @ x == b instead of a @ x == b.	False

```
from kivy.app import App
from kivy.lang import Builder
# We define the UI and widgets in one string
kv_code = """
GridLayout:
 cols: 2
 spacing: 10
 padding: 10
 Button:
  text: 'Button 1'
  size_hint: None, None
  size: '100dp', '36dp'
 Button:
  text: 'Button 2'
  size_hint: None, None
  size: '100dp', '36dp'
 Button:
  text: 'Button 3'
  size_hint: None, None
  size: '100dp', '36dp'
 Button:
  text: 'Button 4'
  size_hint: None, None
  size: '100dp', '36dp'
"""
class EngApp(App):
 def build(self):
  return Builder.load_string(kv_code)
if __name__ == '__main__':
 EngApp().run()
```

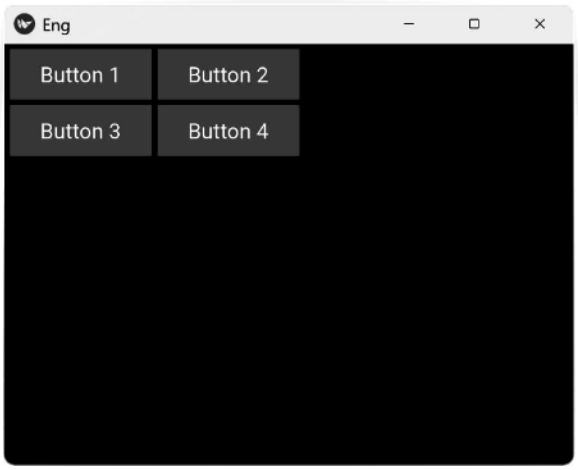


Figure 13-6 Arrangement of four buttons in a two-by-two GridLayout

Table 12-29 Parameters for the PchipInterpolator class

Param.	Description	Default Value
x	1D array of monotonically increasing real values (no duplicates allowed)	–
y	N-D array of real values (length along axis must match length of x)	–
axis	Axis in y array corresponding to x-coordinate values	0
extrapolate	Whether to extrapolate to out-of-bound points (bool)	None (extrapolate based on first and last intervals or return NaNs)

Table 11-12 Common rounding and approximation functions in NumPy

Operation Type	Command	Description
Round	np.round(a)	Rounds to nearest integer
Floor	np.floor(a)	Rounds down
Ceil	np.ceil(a)	Rounds up
Truncate	np.trunc(a)	Drops decimal part (toward zero)

Table 8-7 Czommon commands for closing files

Command	Description	Example
close()	Closes the file	file.close()
closed	Checks if the file is closed	file.closed

Table 1-2 List of essential files used by Python

Category	Extension	Description	Developer editable?
Source code	.py	Main application logic files written in Python	Yes
	.pyc	Precompiled code files generated for faster execution	No
	.pyd	Compiled module files built from C/C++ code	No
	.pyi	Type definition files for static analysis tools	Yes
	.ipynb	Cell-based interactive computing files with code, text, and visual elements	Yes
Configuration	pyproject.toml	Standardized project settings file	Yes
	.ini/.cfg/.config	Classic configuration format with sections and key–value pairs	Yes
	.yaml/.yml	Settings format for modern projects	Yes
	.json	Lightweight configuration data exchange format and communication APIs	Yes
	.env	Environment variable definitions to hide sensitive data	Yes
	requirements.txt	List of packages required for installation	Yes
Documentation and metadata	README.md	Project instructions in Markdown format	Yes
	LICENSE/LICENSE.txt/COPYING	Software license terms and redistribution conditions	Yes
	CHANGELOG.md	Version history	Yes
	.gitignore	File and directory exclusions for version control	Yes

Category	Extension	Description	Developer editable?
Testing	test_*.py/*_test.py	Code test files	Yes
Packaging and distribution	setup.py	Package build script	Yes
	MANIFEST.in	Additional file inclusions for distribution packages	Yes

Table 7-12 Comparison between function-based and class-based decorators

Aspect	Function-Based Decorators	Class-Based Decorators
Implementation	Crafted as a function accepting another function and yielding a wrapper function	Constructed as a class featuring a **__call__** method to enable object invocation
State management	Offers restricted state handling, necessitating the use of closures or global variables	Facilitates straightforward state preservation through instance attributes
Flexibility	Simpler and more concise for straightforward use cases	Offers greater adaptability for intricate scenarios, like state preservation or layering
Readability	Easier to read and understand for simple use cases	May be more detailed but advantageous for sophisticated situations

Table 12-40 Parameters for the scipy.linalg.det function

Parameter	Type	Description	Default
a	array_like	Input array to compute determinants for.	–
overwrite_a (optional)	bool	Allow overwriting data in a (may enhance performance).	False
check_finite (optional)	bool	Check if input matrix contains only finite numbers (disabling may cause issues with infinities/NaNs).	True

Table 6-5 A trace of a for loop iterating over a list of strings (fruits)

Iteration Number	Current Value of "fruit"	Output
1	apple	apple
2	banana	banana
3	cherry	cherry

Table 12-12 Summary of SciPy FFT helper functions, their descriptions, and key use cases

Function	Description	Key Use Cases
fftshift/ifftshift	Rearrange FFT outputs to move zero-frequency to the center and undo this operation.	Visualization of frequency spectra
fftfreq/rfftfreq	Generate frequency bins for FFT results; rfftfreq is optimized for real-valued inputs.	Labeling frequency axes
fhtoffset	Calculate optimal shifts to minimize numerical artifacts in Hankel transforms.	Reducing transform artifacts
next_fast_len/prev_fast_len	Find optimal array sizes for FFT efficiency.	Signal padding and performance optimization
set_workers/get_workers	Manage parallel processing threads for FFT operations.	Improving computational performance

Table 12-14 Parameters for the scipy.fft.fftshift function

Parameter	Data Type	Description
x	array_like	Input array.
axes	int or shape tuple	Axes over which to shift. Default is None, which shifts all axes.

Table 13-3 Kivy layout managers, arrangement styles, and typical use cases

Layout	Arrangement Style	Ideal Use Cases
BoxLayout	Arranges widgets in a single row (horizontal) or column (vertical)	Simple toolbars or lists
GridLayout	Organizes widgets into a grid of rows and columns	Table-like formats
FloatLayout	Allows widgets to be placed at arbitrary or proportional coordinates	Overlapping widgets or creating complex, layered designs
RelativeLayout	Positions children relative to the layout's own position	Grouping widgets that need to move together as a single unit
AnchorLayout	"Anchors" or snaps its single child widget to a border or the center	Placing widgets in corners or center
StackLayout	Stacks widgets next to each other, wrapping to a new line when full	Displaying collections of items with uniform sizes, like a gallery
PageLayout	Creates a multi-page view where users can swipe between pages	Creating simple wizards, carousels, or book-like interfaces
ScatterLayout	Allows its children to be freely moved, scaled, and rotated with touch	Interactive elements

Feature	F-String	T-String
Output	An immediate, final string.	A Template object.
Evaluation	Expressions are evaluated and inserted instantly.	Expressions are evaluated when the template is created; their values and metadata are stored for later processing.
Security	It produces final text immediately and does not provide a separate inspection or sanitization step for interpolated values.	It is safer for structured-output workflows because values can be inspected, escaped, or validated before rendering.
Use case	It should be used for simple string formatting, such as print instructions or just logging.	Useful for structured text processing, such as templating systems that validate or escape values before producing HTML or SQL.

Table 11-13 Arithmetic helper functions for numerical stability in NumPy

Operation Type	Command	Description
exp(x) - 1 (accurate for small x)	np.expm1(a)	Prevents precision loss
log(1 + x) (accurate for small x)	np.log1p(a)	Stable version of log(1 + x)
Floating-point ➤ integer	np.rint(a)	Rounds to nearest integer (float preserved)
Square root/cube root	np.sqrt(a)	Elementwise √a
	np.cbrt(a)	Elementwise cube root

Table 11-11 Unary arithmetic operations, operators, and corresponding ufuncs

Operation Type	Command	Description
Positive	+a	Unary plus (returns a copy).
Negative	-a	Unary negation (changes signs).
Absolute value	np.abs(a) or np.absolute(a)	Elementwise magnitude (works for complex numbers).
Reciprocal	np.reciprocal(a)	Computes the multiplicative inverse elementwise. Integer inputs use integer arithmetic, so values may truncate; use floating-point input for fractional reciprocals.
Exponent	np.exp(a)	e^a elementwise.
Square	np.square(a)	Squares each element.
Square root	np.sqrt(a)	Computes square roots elementwise; real inputs must be nonnegative.
Sign	np.sign(a)	Returns −1, 0, or 1 according to the sign of each real value.
Natural Logarithm	np.log(a)	Computes the natural logarithm elementwise; real inputs must be positive.

Table 3-6 Methods and operations for Python tuples

Method/Operation	Description	Example Usage
count(value)	Returns the number of times a specified value appears in the tuple.	my_tuple = (1, 2, 2, 3) print(my_tuple.count(2)) # Output: 2
index(value, [start, stop])	Returns the index of the first occurrence of a value. Optionally, search within a range using start and stop.	my_tuple = (1, 2, 3, 2) print(my_tuple.index(2)) # Output: 1 print(my_tuple.index(2, 2)) # Output: 3
Concatenation (+)	Combines two or more tuples into one.	tuple1 = (1, 2) tuple2 = (3, 4) result = tuple1 + tuple2 print(result) # Output: (1, 2, 3, 4)
Repetition (*)	Repeats the elements of a tuple a specified number of times.	my_tuple = (1, 2) result = my_tuple * 3 print(result) # Output: (1, 2, 1, 2, 1, 2)
Membership (in, not in)	Checks if an element exists in the tuple.	my_tuple = (1, 2, 3) print(2 in my_tuple) # Output: True print(4 not in my_tuple) # Output: True
Length (len())	Returns the number of elements in the tuple.	my_tuple = (1, 2, 3) print(len(my_tuple)) # Output: 3
Slicing [:]	Extracts a portion of the tuple using slicing syntax.	my_tuple = (1, 2, 3, 4, 5) print(my_tuple[1:4]) # Output: (2, 3, 4)
Sorting (sorted())	Returns a sorted list from the tuple elements (requires conversion back to a tuple if needed).	my_tuple = (3, 1, 2) sorted_tuple = tuple(sorted(my_tuple)) print(sorted_tuple) # Output: (1, 2, 3)
Sum (sum())	Calculates the sum of all numeric elements in the tuple.	sum(my_tuple)

Method/Operation	Description	Example Usage
min() and max()	Finds the smallest or largest element in the tuple.	min(my_tuple) max(my_tuple)

Table 11-3 Main subpackages available in NumPy and their functional descriptions

Subpackage	Description
numpy._core (private)	Internal implementation of ndarray, universal functions (ufuncs), broadcasting rules, and the numeric type system. Application code should use NumPy's public top-level API rather than importing this private module directly.
numpy.linalg	High-level linear algebra engine built on optimized BLAS (Basic Linear Algebra Subprograms)/LAPACK (Linear Algebra PACKage) routines, enabling efficient matrix computations for engineering and scientific workflows.
numpy.fft	Fast Fourier Transform routines for converting signals between time and frequency domains, essential for signal processing, vibration analysis, and spectral studies.
numpy.random	Modern random number generation framework based on PCG64, offering high-quality, reproducible stochastic sampling for simulations, Monte Carlo methods, and statistics. PCG64 is the random number generator algorithm used by modern NumPy.
numpy.ma	Implements masked arrays that allow computations while marking elements as invalid or missing, useful for real-world datasets and sensor data with gaps.
numpy.polynomial	Provides polynomial classes and numerical tools for evaluation, differentiation, integration, and fitting using multiple bases (Chebyshev, Legendre, etc.).
numpy.testing	Utility suite for validating numerical algorithms with robust array comparisons, tolerances, and automated verification of floating-point computations.
numpy.typing	Type-hinting support for better IDE assistance and static analysis, defining standard types for arrays, dtypes, shapes, and array-like objects.
numpy.dtypes	Starting with NumPy version 2.0, the modern and extensible data type system allows for richer scalar types, enhanced string types, custom data representations, and future data type extensions.
numpy.strings	High-performance UTF-8 vectorized string operations that replace older byte-based string handling, supporting full Unicode and efficient text manipulation.
numpy.char	Legacy string manipulation module providing vectorized operations over Unicode arrays; maintained for backward compatibility with older NumPy code.
numpy.ctypeslib	Bridges NumPy arrays with C libraries through the ctypes module, enabling zero-copy memory sharing and interfacing with compiled C code.

Subpackage	Description
numpy.lib	Collection of high-level utilities: file I/O helpers, indexing tricks, histogramming, interpolation tools, and numerical helper functions.
numpy.matlib	MATLAB-style matrix creation utilities providing convenience functions for users transitioning from matrix-centric environments.

Table 12-19 Comprehensive list of FFT functions in scipy.fft

Function	Description
fft(x)	Computes the 1D Discrete Fourier Transform (DFT) of a signal
ifft(X)	Reconstructs the original signal from its DFT
fft2(x)	Computes the 2D DFT of a 2D array
ifft2(X)	Reconstructs the original 2D signal from its 2D DFT
fftn(x)	Computes the N-dimensional DFT of an array
ifftn(X)	Reconstructs the original N-dimensional signal from its DFT
rfft(x)	Computes the 1D DFT for real input, returning only positive frequency terms
irfft(X)	Reconstructs the original real-valued signal from its DFT
rfft2(x)	Computes the 2D DFT for real 2D input
irfft2(X)	Reconstructs the original real-valued 2D signal from its 2D DFT
rfftn(x)	Computes the N-dimensional DFT for real input
irfftn(X)	Reconstructs the original real-valued N-dimensional signal from its DFT
hfft(x)	Computes the FFT of a signal with Hermitian symmetry
ihfft(X)	Reconstructs the original signal from its Hermitian-symmetric FFT
hfft2(x)	Computes the 2D FFT of a Hermitian-symmetric complex array
ihfft2(X)	Reconstructs the original 2D signal from its Hermitian-symmetric 2D FFT
hfftn(x)	Computes the N-D FFT of a Hermitian-symmetric complex input
ihfftn(X)	Reconstructs the original N-dimensional signal from its Hermitian-symmetric FFT

Table 10-3 Matplotlib plotting tools frequently used in engineering applications

Function	Description
Curves and Discrete Data	
plot(x, y)	Connects numerical coordinates with lines and may also place markers at individual data points
scatter(x, y)	Represents independent observations as points whose appearance can reflect additional data properties
bar(x, height)	Compares numerical quantities by representing each value with a rectangular bar
stem(x, y)	Displays sampled or discrete values using markers connected to a reference baseline
fill_between(x, y1, y2)	Highlights the region bounded by two curves over a common horizontal coordinate
stackplot(x, y)	Places several area datasets on top of one another to show their accumulated contribution
stairs(values)	Represents values that remain constant over successive intervals using a stepped curve
Statistical Analysis and Measurement Uncertainty	
hist(x)	Groups numerical observations into intervals and shows how frequently values occur in each range
boxplot(X)	Summarizes the distribution and spread of one or more datasets using boxes, medians, and whiskers
errorbar(x, y, yerr, xerr)	Adds horizontal or vertical uncertainty ranges to measured or calculated data points
violinplot(D)	Shows the shape and density of a numerical distribution for one or more datasets
eventplot(D)	Marks the positions of discrete events along parallel reference lines
hist2d(x, y)	Counts observations within two-dimensional intervals to reveal the joint distribution of two variables
hexbin(x, y, C)	Groups nearby observations into hexagonal regions, making dense two-dimensional datasets easier to inspect
pie(x)	Divides a circular chart into sectors that represent the relative contribution of different categories
ecdf(x)	Displays the fraction of observations that are less than or equal to each numerical value
Images, Scalar Fields, and Vector Fields	

Function	Description
imshow(Z)	Converts a two-dimensional numerical array into a raster image for visual inspection
pcolormesh(X, Y, Z)	Maps numerical values onto cells of a two-dimensional mesh using a pseudocolor representation
contour(X, Y, Z)	Draws curves connecting locations associated with equal values of a scalar quantity
contourf(X, Y, Z)	Separates a scalar field into filled value regions defined by contour levels
barbs(X, Y, U, V)	Represents two-dimensional vector magnitude and direction using barb-style graphical symbols
quiver(X, Y, U, V)	Places arrows on a two-dimensional domain to indicate vector direction and relative magnitude
streamplot(X, Y, U, V)	Traces continuous paths through a two-dimensional vector field to illustrate its overall flow pattern
Triangular Mesh and Unstructured Data	
tricontour(x, y, z)	Generates equal-value contour curves from data distributed over a triangular mesh
tricontourf(x, y, z)	Creates filled contour regions from scalar values associated with triangulated data
tripcolor(x, y, z)	Assigns pseudocolor values to an unstructured triangular domain
triplot(x, y)	Displays the edges and points that form a two-dimensional triangulation
Three-Dimensional and Volumetric Visualization	
bar3d(x, y, z, dx, dy, dz)	Uses three-dimensional rectangular bars to compare quantities positioned in 3D space
plot(xs, ys, zs)	Connects coordinates in three dimensions to form spatial curves or marked trajectories
quiver(X, Y, Z, U, V, W)	Displays three-dimensional vectors as arrows located at selected spatial coordinates
scatter(xs, ys, zs)	Places individual observations at specified three-dimensional coordinates
stem(x, y, z)	Represents discrete three-dimensional values with markers and stems extending from a reference plane
plot_surface(X, Y, Z)	Builds a shaded three-dimensional surface from coordinate values arranged on a grid
plot_trisurf(x, y, z)	Constructs a three-dimensional surface using triangular connections between data points
voxels([x, y, z], filled)	Displays occupied cells within a three-dimensional volume using voxel elements

Function	Description
plot_wireframe(X, Y, Z)	Represents the geometry of a three-dimensional surface using intersecting mesh lines

Table 2-3 Comparison operators

Operator	Description	Example	Result
>	Greater than	5 > 3	True
<	Less than	5 < 3	False
>=	Greater than or equal to	5 >= 3	True
<=	Less than or equal to	5 <= 3	False
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
is	Object identity (same object)	b=10 a = b b is a	True
is not	Object identity (not the same)	b=10 a = b a is not b	False
in	Membership (element in container)	'a' in 'apple'	True
not in	Membership (element not in)	'z' not in 'apple'	True

Table 12-3 Parameters and descriptions for the scipy.differentiate.derivative function

Parameter	Description	Def. Value
f	The function to differentiate. It should take at least one argument x and can take additional arguments if needed. f should be elementwise. It must not change the input arrays.	
x	The point(s) at which to evaluate the derivative. It can be a single value or an array of values.	
args	If your function f requires additional parameters, you can pass them as a tuple to args.	( )
tolerances	Controls the accuracy of the derivative calculation. You can specify absolute tolerance (atol) and relative tolerance (rtol). The algorithm stops when the error estimate is less than atol + rtol * abs(res.df).	None
maxiter	The maximum number of iterations the algorithm will perform.	10
order	The order of the finite difference formula used for the approximation. Higher order generally leads to more accurate results, but also requires more function evaluations.	8
initial_step	The initial step size used by the finite difference method. It influences the accuracy and computational cost.	0.5
step_factor	The factor by which the step size is reduced in each iteration. If less than 1, subsequent steps will be greater than the initial step.	2.0
step_direction	Determines whether to use central, forward, or backward differences. 0 for central, –1 for backward, and 1 for forward.	0
preserve_shape	Deals with functions that may have issues with broadcasting. If True, it ensures that the function f is called with arguments of consistent shapes.	False
callback	A function that can be called before the first iteration and after each iteration. It allows you to monitor the progress of the algorithm.	None

```
#File engapp.py
from kivy.app import App
from kivy.uix.floatlayout import FloatLayout
# Kivy will automatically look for a file named "eng.kv".
class EngApp(App):
 def build(self):
  # The root widget is defined in the .kv file, so we don't need to return anything here.
  pass
if __name__ == '__main__':
 EngApp().run()
```

```
#File eng.kv - kivy 2.2.0
FloatLayout:
# This BoxLayout will be a centered window.
BoxLayout:
orientation: 'vertical'
size_hint: None, None
size: "300dp", "200dp"
pos_hint: {'center_x': .5, 'center_y': .5}
padding: "10dp"
spacing: "10dp"
Button:
text: 'Button'
font_size: '20sp'
Label:
text: 'Hello engineer!'
font_size: '20sp'
```

Table 10-8 Examples of custom scale transformations

Scale Name	Forward Function	Inverse Function	Use Case
Squared	$f(x) = x^2$	$f^{-1}(x) = \sqrt{x}$	Emphasizes differences between large values; compresses small values
Cubic	$f(x) = x^3$	$f^{-1}(x) = x^{(1/3)}$	Provides even stronger emphasis on large values than a square scale
Reciprocal	$f(x) = 1/x$	$f^{-1}(x) = 1/x$	For inverse relationships (e.g., frequency vs. wavelength); makes small values large
Exponential (base-2)	$f(x) = 2^x$	$f^{-1}(x) = \log_2(x)$	Visualizing data with binary doubling patterns or exponential growth
Tangent	$f(x) = \tan(x)$	$f^{-1}(x) = \arctan(x)$	For angular data within a monotonic interval; expands the scale near tangent asymptotes at $\pi/2 + k\pi$
Offset	$f(x) = x + c$	$f^{-1}(x) = x - c$	Shifts the entire axis by a constant value, c

Table 11-8 Array inspection and structural query utility functions

Function/Method	Description
numpy.info(obj)	Displays documentation for an object
numpy.shape(a)	Returns array shape
numpy.ndim(a)	Number of dimensions
numpy.size(a)	Number of elements
numpy.isnan(x)	Elementwise NaN test
numpy.dtype(obj)	Creates a dtype object
numpy.isfortran(a)	Checks Fortran-contiguous layout
numpy.isscalar(obj)	Checks if value is scalar
numpy.isfinite(x)	Tests finiteness
numpy.iscomplex(x)	Tests complex values
numpy.isreal(x)	Tests real values
numpy.array_equal(a1, a2)	Checks if arrays have same shape and values

Table 12-42 Parameters for the scipy.linalg.eig function

Parameter	Type	Description	Default
a	(M, M) array_like	Matrix whose eigenvalues and eigenvectors will be computed.	–
b (optional)	(M, M) array_like	Right-hand-side matrix in generalized eigenvalue problem.	None (identity matrix assumed)
left (optional)	bool	Calculate and return left eigenvectors.	False
right (optional)	bool	Calculate and return right eigenvectors.	True
overwrite_a (optional)	bool	Overwrite a to improve performance.	False
overwrite_b (optional)	bool	Overwrite b to improve performance.	False
check_finite (optional)	bool	Check if input matrices contain only finite numbers.	True
homogeneous_eigvals (optional)	bool	Return eigenvalues in homogeneous coordinates.	False

```
import matplotlib.pyplot as plt
# Data
x = [1, 2, 4, 6]
y = [7, 18, 13, 30]
# Pyplot-style
plt.plot(x, y)
plt.title("My Pyplot Plot")
plt.xlabel("X-values")
plt.show()
```

```
import matplotlib.pyplot as plt
# Data
x = [1, 2, 4, 6]
y = [7, 18, 13, 30]
# OO-style
fig, ax = plt.subplots()
ax.plot(x, y)
ax.set_title("My OO Plot")
ax.set_xlabel("X-values")
plt.show()
```

Table 8-2 Common text-based formats that can be accessed with Python's open() function and typical operations performed with supporting parsers

File Format	Description	Possible Operations
Text files (.txt)	Plain-text files containing unformatted text	Read, write, append, search, replace, manipulate lines, count lines/words/characters.
CSV files (.csv)	Comma-separated values files used for tabular data	Read, write, append, parse, manipulate rows/columns, convert to/from other formats, data analysis.
JSON files (.json)	JavaScript Object Notation files used for structured data	Read, write, parse, serialize/deserialize JSON objects, manipulate data, validate JSON structure, convert to/from other formats.
HTML files (.html)	HyperText Markup Language files used for web pages	Read, write, edit HTML content, parse HTML elements, web scraping, generate HTML reports.
XML files (.xml)	eXtensible Markup Language files used for data interchange	Read, write, parse, manipulate XML elements/attributes, validate XML structure, transform XML (XSLT), convert to/from other formats.
INI files (.ini)	Initialization files used for configuration settings	Read, write, parse, manipulate key–value pairs, update configuration settings.
Log files (.log)	Log files containing plain-text logs of events or actions	Read, write, append, search, filter, analyze log entries, generate log reports.
Markdown files (.md)	Markdown files used for formatted text with markup	Read, write, edit Markdown content, convert to HTML/PDF, preview/render Markdown, manipulate Markdown elements.
Python files (.py)	Python script files containing Python code	Read, write, edit Python code, execute scripts, analyze code, extract documentation (docstrings), linting/formatting.
YAML files (.yaml or .yml)	YAML Ain't Markup Language files used for configuration data	Read, write, parse, manipulate key–value pairs, update configuration settings, validate YAML structure.
Properties files (.properties)	Properties files used for configuration settings in key–value pairs	Read, write, parse, manipulate key–value pairs, update configuration settings.

Table 11-15 Operations for working with complex-valued NumPy arrays

Operation	Command
Real part	a.real
Imag part	a.imag
Conjugate	np.conj(a)
Magnitude	np.abs(a)
Angle (phase)	np.angle(a)

Table 2-8 Comparison of Python collections

Collection Type	Ordered	Mutable (Changeable)	Allows Duplicates	Primary Use Case
List	Yes	Yes	Yes	A general-purpose, ordered sequence of items that can change
Tuple	Yes	No	Yes	An unchangeable (read-only) sequence of related data
Set	No	Yes	No	Storing unique items and performing high-speed membership checks
Dictionary	Yes (insertion order)	Yes	Keys must be unique	Storing related information in key–value pairs for quick lookups

Table 12-8 Parameters and descriptions for the scipy.fft.dct function

Parameter	Type	Description	Default Value
x	array_like	The input array.	–
type	{1, 2, 3, 4}	Type of the DCT.	2
n	int	Length of the transform.	n=x.shape[axis]
axis	int	Axis along which the DCT is computed.	–1
norm	{“backward”, “ortho”, “forward”}	Normalization mode.	"backward"
overwrite_x	bool	If True, the contents of x can be destroyed.	False
workers	int	Maximum number of workers for parallel computation. If negative, the value wraps around from os.cpu_count().	None
orthogonalize	bool	Whether to use the orthogonalized DCT variant.	True if norm="ortho", else False

Table 11-2 Feature comparison and selection guide between NumPy and SciPy

Category	NumPy	SciPy	Best Overall Choice
Arrays	Create, manipulate, and operate on arrays.	–	NumPy
Dtypes/memory layout	Understand numeric types, shape, strides.	–	NumPy
Indexing/slicing/broadcasting	Structural manipulation	–	NumPy
Math operations (sine, cosine, exponential, logarithm)	Basic elementwise math	Special mathematical functions (e.g., Bessel, gamma, and error functions)	NumPy for standard elementwise math; SciPy for special functions
Linear algebra	Matrix product, dot()	SVD (Singular Value Decomposition), robust inverse, eigenvalues, QR (Orthogonal–Triangular decomposition), LU (Lower–Upper decomposition)	SciPy (more accurate and stable)
FFT/Transforms (FFT = Fast Fourier Transform)	Simple FFT on arrays	Multidimensional FFT, faster and more modern implementation	SciPy
Statistics	Mean, sum, std, median (simple stats)	PDFs (probability density functions), CDFs (cumulative distribution functions), statistical tests	SciPy.stats
Interpolation	np.interp (1D linear interpolation)	Splines, B-splines, RBFs (radial basis functions), griddata, N-D (N-dimensional) interpolation	SciPy.interpolate

Category	NumPy	SciPy	Best Overall Choice
Integration/differentiation	Not available in NumPy	quad (1D numerical integration), dblquad (double integration), solve_ivp (solve an initial value problem for ODEs (ordinary differential equations)), derivatives	SciPy.integrate
Optimization	Not available	minimize, curve_fit, least_squares	SciPy.optimize
Signal processing	Not available	Filters, spectra, convolution, resampling	SciPy.signal
N-D image processing (N-D = N-dimensional)	Not available	Gaussian filters, rotation, edge detection	SciPy.ndimage
Sparse matrices	Not available	Sparse matrices and sparse solvers	SciPy.sparse
Special functions	Not available	Bessel, airy, gamma, beta, and other special mathematical functions	SciPy.special
Random numbers	Basic random number generation (Generator API)	Advanced statistical distributions	NumPy for simple random data/SciPy for statistics
Scientific constants	Not available	scipy.constants	SciPy
Input/output	.npy, .npz, loadtxt	Specialized scientific file formats	NumPy (basic), SciPy (advanced)

Table 7-9 Common use cases for class methods

Use Case	Description
Alternative constructors	Create objects with default or specific values without using __init__.
Factory methods	Create and return new instances of the class with specific configurations.
Class-level operations	Modify or access class-level attributes shared across all instances.
Inheritance and polymorphism	Override class methods in subclasses for polymorphic behavior.
Utility methods	Provide utility functions related to the class but not requiring an instance.

Table 12-7 Functions for Discrete Sine and Cosine Transforms (DST and DCT) in scipy.fft

Function	Description
dct(x)	Returns the Discrete Cosine Transform (DCT) of an arbitrary-type sequence x
idct(x)	Returns the Inverse Discrete Cosine Transform (IDCT) of an arbitrary-type sequence x
dctn(x)	Returns the multidimensional Discrete Cosine Transform along the specified axes
idctn(x)	Returns the multidimensional Inverse Discrete Cosine Transform along the specified axes
dst(x)	Returns the Discrete Sine Transform (DST) of an arbitrary-type sequence x
idst(x)	Returns the Inverse Discrete Sine Transform (IDST) of an arbitrary-type sequence x
dstn(x)	Returns the multidimensional Discrete Sine Transform along the specified axes
idstn(x)	Returns the multidimensional Inverse Discrete Sine Transform along the specified axes

Table 3-7 Mathematical set operations

Operation	Description
Union (|)	Combines both sets, excluding duplicates
Intersection (&)	Gets elements present in both sets
Difference (-)	Gets elements in the first set but not in the second
Symmetric difference (^)	Gets elements in one set or the other, but not in both

Table 6-8 An iteration trace demonstrating how the break statement immediately terminates a while loop

Iteration	Value of "i"	Condition (`"i <= 5"`)	Action
1	1	True	Print 1, and increment "i" to 2.
2	2	True	Print 2, and increment "i" to 3.
3	3	True	i == 3, so break is executed.
–	–	–	The loop ends.

Table 13-6 Layout-specific properties for positioning Kivy widgets

Layout	Key Properties for Positioning	Description
BoxLayout	orientation, spacing, padding	orientation ('horizontal' or 'vertical') determines the primary axis of arrangement. spacing adds space between children, and padding adds space around the group of children. pos_hint is partially supported for the axis perpendicular to the orientation.
GridLayout	cols, rows, spacing, padding	spacing and padding control the space between and around cells.
StackLayout	orientation, spacing, padding	Stacks children from left to right or top to bottom (lr-tb, tb-lr, etc.), wrapping to a new row/column when the edge is reached.
AnchorLayout	anchor_x, anchor_y	Anchors its single child to a specific point. anchor_x can be 'left', 'center', or 'right', and anchor_y can be 'top', 'center', or 'bottom'.
RelativeLayout	pos_hint, pos	Similar to FloatLayout, but all child positions are *relative to the layout's own position*. A child at pos=(0, 0) will be at the layout's bottom-left corner, not the window's.
Scatter/ScatterLayout	pos, rotation, scale	Allows for interactive translation, rotation, and scaling of its children via touch input. pos can be set initially, but is typically modified by user interaction.
PageLayout	page, border	Arranges children as pages. Swiping from the border area transitions between pages. The page property controls which child is currently visible.

Table 6-1 An iteration-by-iteration breakdown of a basic while loop that counts from 1 to 5

Iteration	Value of "i"	Condition ("i <= 5")	Action
1	1	True	Print "1", and increment "i" to "2".
2	2	True	Print "2", and increment "i" to "3".
3	3	True	Print "3", and increment "i" to "4".
4	4	True	Print "4", and increment "i" to "5".
5	5	True	Print "5", and increment "i" to "6".
6	6	False	The loop ends.

Table 12-43 Summary of matrix decomposition functions available in scipy.linalg

Function	Description
Decompositions	
lu(), lu_factor()	Lower–Upper decomposition
svd(), svdvals()	Singular Value Decomposition
cholesky(), cho_factor()	Cholesky decomposition
qr(), qr_multiply()	QR decomposition
rq(), qz(), ordqz()	Other decompositions (RQ, QZ, ordered QZ)
schur(), rsf2csf()	Schur decomposition
hessenberg()	Hessenberg form
polar()	Polar decomposition
ldl()	LDLt (LDL Transpose) factorization
Solving and Basis	
lu_solve(), cho_solve()	Solving linear equations
orth(), null_space()	Orthonormal basis construction
Updates and Conversions	
qr_update(), qr_delete(), qr_insert()	QR (Orthogonal–Triangular) updates
cdf2rdf()	Complex to real eigenvalue conversion
cossin()	Cosine–sine decomposition

Table 2-7 List of Python collections

Data Type	Example	Description
list	[1, 2, 3]	Represents an ordered collection of elements that can be modified. It can store repeated elements and allows access to elements by index.
tuple	(1, 2, 3)	Represents an ordered collection of elements that cannot be modified. It can store repeated elements and allows access to elements by index.
set	{1, 2, 3}	Represents an unordered collection of unique elements that can be modified. Sets have no indexes for their elements and automatically remove duplicate elements.
dict	{“key”: “value”}	Represents an ordered collection of key–value pairs that can be modified. Duplicate keys are not allowed.
range	range(10)	Represents a sequence of numbers generated by a range function.

Table 7-1 Common programming paradigms and examples

Paradigm Name	Description	Examples
Object-oriented programming (OOP)	Structures code around objects, which are instances of classes	Python, Java, C++, C#, Ruby, Julia
Procedural programming	Organizes code into functions that manipulate data, using a sequential method to address issues, following a step-by-step approach	C, Pascal, Fortran, Python, Julia
Functional programming	Treats computation as the application of mathematical functions and steers clear of state changes or mutable data	Haskell, Lisp, Scala, Elixir, Python, Julia
Multi-paradigm programming	Integrates various paradigms in a single language, allowing developers to choose the best approach for a specific problem	Python, JavaScript, C++, Julia
Imperative programming	Emphasizes detailing the process of program execution through statements that modify the program's state	C, C++, Java, Python, Julia
Declarative programming	Defines what the program should accomplish without describing how to achieve it	SQL, HTML, CSS, Prolog

Table 1-4 Comparison between the main code editors and IDEs available for the Python language

Editor/IDE	Type	Free/Paid	Debugging	Code Completion	Project Management
PyCharm	IDE	Free (Community), paid (Pro)	Excellent	Excellent	Excellent
VS Code	Code editor	Free	Excellent	Excellent	Good
Sublime Text	Code editor	Paid (license)	Good	Excellent	Good
Spyder	IDE	Free	Excellent	Excellent	Good
Thonny	IDE	Free	Good	Good	Limited
IDLE	IDE	Free	Limited	Limited	Limited
Jupyter Notebook	Interactive (cell-based)	Free	Limited	Limited	Limited
GNU Emacs	Text editor	Free	Good	Excellent	Good
Vim	Text editor	Free	Good	Excellent	Good

Table 9-10 Main functions of the statistics module

Function	Description
Averages and Measures of Central Location	
mean()	Calculates the arithmetic average
fmean()	A faster, floating-point version of mean(), with optional weighting
geometric_mean()	Computes the geometric mean
harmonic_mean()	Calculates the harmonic mean, suitable for averaging rates
median()	Determines the middle value of a dataset
median_low() and median_high()	Provide the lower or higher of the two middle values for even-sized datasets
median_grouped()	Estimates the median for grouped data
mode()	Finds the single most frequent value
multimode()	Returns a list of the most frequent values
quantiles()	Divides data into intervals with equal probability (e.g., quartiles, deciles, percentiles)
Measures of Spread	
pstdev() and pvariance()	Calculate the population standard deviation and variance
stdev() and variance()	Calculate the sample standard deviation and variance
Statistics for Relations Between Two Inputs	
covariance()	Measures the joint variability of two variables
correlation()	Calculates Pearson's and Spearman's correlation coefficients to assess the strength and direction of a linear or monotonic relationship
linear_regression()	Determines the slope and intercept for a simple linear regression model

Table 12-56 A list of MATLAB-compatible filter design functions in scipy.signal

MATLAB/SciPy	Application
butter	Design Butterworth digital or analog filters for applications like audio processing, image filtering, and biomedical signal processing.
buttord	Determine the order of a Butterworth filter required to meet specific frequency response characteristics.
cheby1	Design Chebyshev Type I digital or analog filters for applications where a steeper roll-off is needed, such as in audio equalization.
cheb1ord	Determine the order of a Chebyshev Type I filter required to meet specific frequency response characteristics.
cheby2	Design Chebyshev Type II digital or analog filters for applications where a flat passband and stopband ripple are acceptable.
cheb2ord	Determine the order of a Chebyshev Type II filter required to meet specific frequency response characteristics.
ellip	Design elliptic (Cauer) digital or analog filters for applications where a very sharp transition between passband and stopband is required.
ellipord	Determine the order of an elliptic filter required to meet specific frequency response characteristics.
bessel	Design Bessel/Thomson digital or analog filters for applications where a maximally flat group delay is required, such as in audio processing.
iirnotch	Design second-order IIR notch digital filters for removing specific frequency components, such as power line noise in biomedical signals.

Table 12-9 Comparison of the four types of Discrete Cosine Transforms (DCTs) available in SciPy

Type	Description	Symmetry	Common Use Case
1	Equivalent to FFT for real, even-symmetrical inputs	Even symmetry at boundaries	Signals with symmetry at both ends
2	Most commonly used DCT variant	Even symmetry at sample points	Image compression (such as JPEG)
3	Inverse of DCT Type 2 (unnormalized)	Odd symmetry at sample points	Reconstructing signals
4	Orthogonal variant with even symmetry around half-sample points	Even symmetry at half-samples	Applications requiring orthogonality

Table 8-8 Other useful file-handling commands

Command	Description	Example
name	Returns the filename	file.name
mode	Returns the file mode	file.mode
encoding	Returns the file encoding	file.encoding
newlines	Returns the type of newlines used in the file	file.newlines
buffer	Returns the underlying buffer object	file.buffer
errors	Returns the error handling mode	file.errors

Table 12-15 Parameters for the scipy.fft.fftfreq function

Parameter	Data Type	Description
n	int	Window length.
d	scalar	Sample spacing (inverse of the sampling rate). Defaults to 1.
xp	array_namespace	The namespace for the return array. Default is None (NumPy).
device	device	The device for the return array. Only valid when xp.fft.fftfreq implements the device parameter.

Table 6-7 A trace showing a for loop iterating through the individual characters of a string

Iteration Number	Current Value of "char"	Output
1	H	H
2	e	e
3	l	l
4	l	l
5	o	o
6	!	!

Table 2-1 Types of variables

Data Type (Dtype)	Example	Description	Value Limits
Int (integer)	42	Represents an integer (whole number)	Limited only by the available machine memory.
float	3.14	Represents a floating-point (decimal) number	Typically a 64-bit IEEE 754 binary64 value. The maximum finite positive value is approximately 1.79 × 10^308.
complex	2 + 3j	Represents a complex number with real and imaginary parts	The real and imaginary parts each have the same limits as a float.
Bool (boolean)	True, False	Represents a boolean value (logical true or false)	Only two possible values: True or False.
Str (string)	“Hello, World!”	Represents a string (sequence of characters)	Limited only by the available machine memory.

Family	Description	Key Commands
Input/Output (I/O)	Import and export data from various file formats and databases.	read_csv(), to_csv(), read_excel(), read_sql(), read_parquet(), to_parquet(), read_json()
Data Structures	Fundamental data containers for storing and manipulating data.	DataFrame(), Series(), Index, Categorical
Inspection and Exploration	Get insights into data structure, content, and statistics.	head(), tail(), info(), describe(), shape, value_counts(), sample()
Selection and Indexing	Access specific data elements using labels or positions.	.loc[], .iloc[], .at[], .iat[], query(), filter()
Manipulation and Cleaning	Transform, clean, and prepare data for analysis.	drop(), dropna(), fillna(), astype(), sort_values(), rename(), replace(), assign()
Merging and Combining	Combine multiple datasets into a unified whole.	merge(), concat(), join()
Grouping and Aggregation	Perform calculations on data subsets using the split–apply–combine approach.	groupby(), agg(), transform(), pivot_table()
Time Series	Work with date and time data using specialized tools.	resample(), .dt accessor, date_range(), to_datetime()
Plotting and Visualization	Create visual representations of data for analysis and insights.	.plot(), .plot.line(), .plot.bar(), .plot.hist()

Table 2-5 Methods for type conversion

Category	Methods	Description
1. Integers and floating numbers	int(), float(), str()	These methods convert between integers, floats, and strings. Used to remove decimals or convert to a float or to a string.
2.Representation for binary, octal, and hexadecimal	bin(), oct(), hex()	Convert an integer to its binary, octal, or hexadecimal string representation, prefixed with"0b," "0o," or "0x."
3. Strings	bytes(), bytearray()	Converts a string into a byte object or a mutable bytearray, useful for working with binary data.
4. Complex number	complex()	Converts two numbers (real and imaginary) into a complex number in the form real + imagj.
5. Boolean	bool()	Converts a value to a boolean, where 0 becomes False and any nonzero value becomes True.
6. Unicode of a character	ord(), chr()	Converts a character to its Unicode code point using ord() and converts a code point back to a character with chr().
7. List, tuple, and dictionary	list(), tuple(), set(), dict(), frozenset()	Converts data types into list, tuple, set (unique elements), dictionary (key–value pairs), or frozenset (immutable set).

Table 7-3 Comparison of relationships between objects: association, aggregation, and composition

Feature	Association	Aggregation	Composition
Relationship strength	Weak relationship.	Weak relationship.	Strong relationship.
Object independence	Objects can exist independently.	Parts can exist independently of the whole.	Parts cannot exist without the whole (part-of).
Lifecycle management	No lifecycle management.	Lifecycle of parts is not managed by the whole.	Lifecycle of parts is managed by the whole.
Example case	Driver and car.	Wheels and car.	Car and engine.
Use case	When objects interact but are independent (uses-a).	When parts can be shared among multiple wholes (has-a).	When parts are integral to the whole and cannot be shared (part-of).

Table 12-32 Parameters for the make_splprep function

Par	Description	Default Value
x	Sampled data points representing the curve in ndim dimensions.	–
w	Strictly positive 1D array of weights.	np.ones(m)
u	Array of parameter values for the curve in parametric form.	Calculated automatically
ub, ue	Endpoints of the parameters interval.	u[0] and u[-1]
k	The degree of the spline determines the smoothness of the curve, with cubic splines being the recommended choice.	3
s	The smoothing condition governs the trade-off between closeness to data points and overall curve smoothness.	0
t	Spline knots.	None
nest	Target length of the knot vector.	None

Table 5-1 An overview of different methods for passing arguments to functions in Python, including positional, keyword, and variable-length arguments

Method	Description
Positional arguments	Pass arguments in the same order as the parameters are defined.
Keyword arguments	Pass arguments by explicitly naming the parameters, regardless of their order.
Default arguments	Use default values for parameters if no argument is provided.
Variable-length positional	Use *args to accept any number of positional arguments, passed as a tuple.
Variable-length keyword	Use **kwargs to accept any number of keyword arguments, passed as a dictionary.
Positional-only arguments	Restrict arguments to positional-only by using / in the function definition.
Keyword-only arguments	Restrict arguments to keyword-only by using * in the function definition.
Unpacking arguments	Use * or ** to unpack sequences (e.g., list, tuple) or dictionaries into arguments.

Table 10-6 Axis scales interpreted from an engineering data perspective

Scale Name	Description	Common Use Cases
linear	The default, evenly spaced scale.	Plotting standard data, showing absolute changes, most general-purpose graphs
log	Compresses a wide range of values. Data must be positive.	Visualizing exponential growth, scientific data spanning several orders of magnitude (e.g., decibel scales)
symlog	"Symmetric log." A logarithmic scale that can handle positive, negative, and zero values by being linear near the origin.	Plotting data with a wide dynamic range that also includes zero or negative values, such as financial profit/loss charts or signal processing data
logit	Specifically for data that exists between 0 and 1. Stretches the scale near the ends.	Visualizing probabilities, model accuracy metrics (like precision/recall), or population proportions
asinh	"Inverse hyperbolic sine." Behaves linearly around zero and logarithmically for larger values.	An alternative to symlog for visualizing data with a wide range that includes zero, often used in astronomy and physics
function/functionlog	Allows for a completely custom scale defined by the user with forward and inverse functions.	Creating highly specialized plots for niche scientific or engineering fields

Table 13-5 Relative positioning keys supported by pos_hint

pos_hint Key	Description	Range
'x'	Positions the widget's x value based on the parent's width. Aligns the widget's left edge	0.0 is the left edge of the parent; 1.0 is the right edge.
'y'	Positions the widget's y value based on the parent's height. Aligns the widget's bottom edge	0.0 is the bottom edge of the parent; 1.0 is the top edge.
'right'	Positions the widget's right edge based on the parent's width	0.0 is the parent's left edge; 1.0 is the right edge.
'top'	Positions the widget's top edge based on the parent's height	0.0 is the parent's bottom edge; 1.0 is the top edge.
'center_x'	Positions the widget's center_x based on the parent's width. Aligns the widget's horizontal center	0.0 is the parent's left edge, 0.5 is the center, and 1.0 is the right edge.
'center_y'	Positions the widget's center_y based on the parent's height. Aligns the widget's vertical center	0.0 is the parent's bottom edge, 0.5 is the center, and 1.0 is the top edge.

Table 8-3 File manipulation modes available in the open() function

Character	Meaning	When to Use
r	Open for reading (default)	Use when you only need to read the contents of a file without modifying it.
w	Open for writing, truncating the file first	Use when you want to overwrite the file with new content. If the file doesn't exist, it will be created.
x	Open for exclusive creation, failing if the file already exists	Use when you want to create a new file and ensure it doesn't already exist.
a	Open for writing, appending to the end of the file if it exists	Use when you want to add new content to the end of an existing file without overwriting it.
b	Binary mode	Use when working with non-text files like images, videos, or other binary data.
t	Text mode (default)	Use when working with text files (.txt, .csv, .log).
+	Open for updating (reading and writing)	Use when you need to both read and write to the same file.

Table 3-9 Common operations and methods for dictionaries

Operation/Method	Syntax Example	Description
Access an item	d['name']	Retrieves the value associated with a key; raises KeyError if the key doesn't exist
Add/update an item	d['age'] = 31	Adds a new key–value pair or updates an existing key's value
Delete an item	del d['name']	Removes a key–value pair; raises KeyError if the key doesn't exist
Check for a key	'age' in d	Returns True if the key exists and otherwise False
Get length	len(d)	Returns number of key–value pairs in dictionary
get()	d.get('city', 'N/A')	Safely retrieves a value; returns a default value if the key is not found
keys()	d.keys()	Returns a view object with all dictionary keys
values()	d.values()	Returns a view object with all dictionary values
items()	d.items()	Returns a view object with all key–value pairs as tuples
update()	d.update({'city': 'NY'})	Merges another dictionary or iterable into the current one
pop()	d.pop('age')	Removes a specified key and returns its value
popitem()	d.popitem()	Removes and returns the last inserted key–value pair as a tuple
clear()	d.clear()	Removes all items from a dictionary
copy()	new_d = d.copy()	Returns a shallow copy of a dictionary
setdefault()	d.setdefault('city', 'NY')	Returns a key's value; inserts the key with a default value if it doesn't exist
Comprehension	{x: x**2 for x in range(3)}	Creates a dictionary from an iterable in a concise way

MATLAB Command	Python control Equivalent	Description
tf(num, den)	tf(num, den)	Create a transfer function model.
zpk(z, p, k)	zpk(z, p, k)	Create a zero–pole–gain model.
ss(A, B, C, D)	ss(A, B, C, D)	Create a state-space model.
c2d(sys, Ts)	c2d(sys, Ts)	Convert continuous to discrete system.
d2c(sys)	No direct equivalent in control 0.10.2	Convert discrete to continuous system.
step(sys)	step_response(sys)	Plot step response.
stepinfo(sys)	step_info(sys)	Compute step response characteristics.
impulse(sys)	impulse_response(sys)	Plot impulse response.
lsim(sys, u, t)	forced_response(sys, T=t, U=u)	Simulate time response to arbitrary input.
initial(sys, x0)	initial_response(sys, X0=x0)	Time response with initial condition.
bode(sys)	bode_plot(sys)	Generate Bode magnitude/phase plot.
nyquist(sys)	nyquist_plot(sys)	Generate Nyquist plot.
nichols(sys)	nichols_plot(sys)	Generate Nichols chart.
rlocus(sys)	root_locus_plot(sys)	Generate root locus plot.
margin(sys)	margin(sys)	Compute gain/phase margins.
dcgain(sys)	dcgain(sys)	Compute steady-state (DC) gain.
freqresp(sys, w)	frequency_response(sys, w)	Frequency response at specified points.
pole(sys)	poles(sys)	Return poles of system.
zero(sys)	zeros(sys)	Return zeros of system.
pzmap(sys)	pole_zero_plot(sys)	Pole–zero map.
acker(A, B, p)	place_acker(A, B, p)	Pole placement using Ackermann's formula.
ctrb(A, B)	ctrb(A, B)	Controllability matrix.
obsv(A, C)	obsv(A, C)	Observability matrix.

Table 12-2 Summary of numerical differentiation operations available in scipy.differentiate

Operation	Purpose	Description
Finite difference differentiation	Compute the derivative of a scalar function numerically.	Approximates the rate of change of a function at a given point by analyzing small changes in the input
Jacobian calculation	Compute the Jacobian matrix of a function numerically.	Evaluates the gradient of a vector-valued function by computing the partial derivatives of each component with respect to the input variables
Hessian calculation	Compute the Hessian matrix of a function numerically.	Analyzes the curvature of a scalar-valued function by computing its second-order partial derivatives at a given point

Table 11-16 Aggregation and reduction functions in NumPy

Operation Type	Command/Pattern	Description
Total sum	np.sum(a)	Sum of all elements in the array.
Sum along axis	np.sum(a, axis=0)	Sum along a given axis (e.g., columns of a 2D array).
Total product	np.prod(a)	Product of all elements in the array.
Product along axis	np.prod(a, axis=1)	Product along a given axis (e.g., row-wise products).
Mean (average)	np.mean(a)	Arithmetic mean of all elements.
Mean along axis	np.mean(a, axis=0)	Mean along a specified axis.
Standard deviation	np.std(a)	Population standard deviation of the array.
Variance	np.var(a)	Population variance of the array.
Minimum value	np.min(a)	Smallest value in the array.
Maximum value	np.max(a)	Largest value in the array.
Index of minimum	np.argmin(a)	Index of the smallest element (flattened index by default).
Index of maximum	np.argmax(a)	Index of the largest element (flattened index by default).
Cumulative sum	np.cumsum(a)	Cumulative sum along a flattened array or a given axis.
Cumulative product	np.cumprod(a)	Cumulative product along a flattened array or a given axis.
Successive differences	np.diff(a)	Discrete difference between successive elements.
Trace of a matrix	np.trace(a)	Sum of the main diagonal of a 2D array.
Median	np.median(a)	Median (50th percentile) of the array.
Quantiles/percentiles	np.quantile(a, q)	General quantiles (e.g., quartiles) of the data.
All elements true?	np.all(a)	Returns True if all elements are non-zero/True.
Any element true?	np.any(a)	Returns True if any element is non-zero/True.
Axis control	np.sum(a, axis=..., keepdims=False)	axis selects which dimensions to reduce; keepdims=True keeps reduced axes with size 1.
Higher-precision reduction	np.sum(a, dtype=np.float64)	Performs the reduction in a specified dtype for better numerical stability.

Table 9-3 Main functions of the math module

Function	Description
fabs(x)	Absolute value (makes a number positive).
fmod(x, y)	Gets the remainder of a division.
ceil(x)	Rounds up to the next whole number.
floor(x)	Rounds down to the previous whole number.
trunc(x)	Cuts off the decimal part completely.
modf(x)	Splits a number into its decimal and integer parts.
pow(x, y)	Calculates x to the power of y (xy).
sqrt(x)	Calculates the square root of x.
cbrt(x)	Calculates the cube root of x.
isqrt(n)	The integer part of the square root.
exp(x)	Calculates e to the power of x (ex).
exp2(x)	Calculates 2 raised to the power of x.
log(x, base)	Calculates the logarithm of x. Base is e if not specified.
log2(x)	Calculates the base-2 logarithm of x.
log10(x)	Calculates the base-10 logarithm of x.
factorial(n)	Calculates n factorial (n!).
comb(n, k)	Combinations: how many ways to choose k items from n.
perm(n, k)	Permutations: like combinations, but order matters.
gcd(*integers)	Greatest common divisor (the largest number that divides all inputs).
lcm(*integers)	Least common multiple (the smallest number they all divide into).
degrees(x)	Converts an angle from radians to degrees.
radians(x)	Converts an angle from degrees to radians.
sin(x), cos(x), tan(x)	Sine, cosine, tangent. The input angle x must be in radians.
asin(x), acos(x), atan(x)	Inverse sine, cosine, tangent. Finds the angle for a given ratio.
dist(p, q)	Calculates the straight-line distance between two points.
hypot(x, y)	Calculates the distance from the origin (0,0) to a point.
isclose(a, b)	Safely checks if two numbers are almost equal (avoids float errors).
fsum(iterable)	Adds up a list of numbers with high precision.

Function	Description
pi, e, tau	Provides famous mathematical constants.
gamma(x), erf(x)	Special advanced functions used in statistics and calculus.
sumprod(p, q)	Multiplies corresponding items from two lists and then adds the results (added in Python 3.12).

Table 9-4 Main functions of the cmath module

Function	Description
polar(z)	Converts a complex number (a+bj) to its polar form (radius, angle).
phase(z)	Gets just the angle (phase) of a complex number, in radians.
rect(r, phi)	Converts polar coordinates (radius, angle) back to a complex number.
exp(z)	Calculates e to the power of a complex number (ez).
log(z, base)	Calculates the logarithm of a complex number.
log10(z)	Calculates the base-10 logarithm of a complex number.
sqrt(z)	Calculates the square root, works for negative numbers.
sin(z), cos(z)	Sine, cosine, etc., for complex numbers. The results are complex.
asin(z), acos(z)	Inverse sine, cosine, etc. Finds the complex angle for a given ratio.
sinh(z), cosh(z)	Hyperbolic sine, cosine, etc., for complex numbers.
asinh(z), acosh(z)	Inverse hyperbolic functions for complex numbers.
isfinite(z)	Returns True only when both the real and imaginary parts are finite.
isinf(z)	Checks if any part of the number is infinity.
isnan(z)	Checks if any part of the number is "Not a Number."
isclose(a, b)	Safely checks if two complex numbers are almost equal.
pi, e, tau	Provide the famous mathematical constants π, e, and τ.
inf, nan	Represent positive infinity and "Not a Number."
infj, nanj	Shortcuts for pure imaginary infinity (0+inf*j) and NaN (0+nan*j).

Table 4-2 Comparison of common package managers

Package Manager	Purpose	Platform	Example
pip	Python package management	Windows, Linux, Mac	pip install numpy
apt	System package management	Debian-based Linux	sudo apt install python3-numpy
yum	System package management	Red Hat–based Linux	sudo yum install numpy
brew	System package management	macOS, Linux	brew install numpy

Table 9-9 Main operations and methods of the fractions module

Operation/Method	Description
Constructors	
Fraction(numerator, denominator)	Creates a fraction from two integers
Fraction(other_fraction)	Creates a copy of another Fraction object
Fraction(float)	Creates a fraction from a float. Can be imprecise
Fraction(decimal)	Creates a fraction from a decimal.Decimal object
Fraction(string)	Creates a fraction from a string representation
Properties	
.numerator	The numerator of the fraction in its simplest form
.denominator	The denominator of the fraction in its simplest form
Class Methods	
from_float(flt)	Alternative constructor to create a Fraction from a float
from_decimal(dec)	Alternative constructor to create a Fraction from a Decimal
Instance Methods	
limit_denominator(max_d)	Finds the closest fraction with a denominator less than or equal to max_d
as_integer_ratio()	Returns a (numerator, denominator) tuple of the fraction in lowest terms
is_integer()	Returns True if the fraction is a whole number and otherwise False
Mathematical Functions	
Arithmetic (+, -, *, /)	Performs standard arithmetic operations between fractions
math.floor(f)	Returns the greatest integer less than or equal to the fraction
math.ceil(f)	Returns the smallest integer greater than or equal to the fraction
round(f)	Rounds the fraction to the nearest integer (rounding halves to the nearest even number)
round(f, ndigits)	Rounds the fraction to the nearest multiple of 1/10**ndigits
Formatting	
format(format_spec)	Allows advanced formatting via format() or f-strings

Table 12-22 Parameters and descriptions for the scipy.integrate.quad function

Parameter	Data Type	Description	Def.
func	function, scipy.LowLevelCallable	A Python function or method to integrate	–
a	float	Lower limit of integration	–
b	float	Upper limit of integration	–
args	tuple, optional	Extra arguments to pass to function	()
full_output	int, optional	Nonzero to return a dictionary of integration information	0
limit	float or int, optional	Upper bound on the number of subintervals	No limit
points	sequence of floats, ints, optional	Break points in the bounded integration interval	[]
weight	float or int, optional	String indicating weighting function	None
wvar	optional	Variables for use with weighting functions	None
wopts	optional	Optional input for reusing Chebyshev moments	None
maxp1	float or int, optional	Upper bound on the number of Chebyshev moments	50
limlst	int, optional	Upper bound on the number of cycles	50
epsabs	float or int, optional	Absolute error tolerance	1.49e-8
epsrel	float or int, optional	Relative error tolerance	1.49e-8
complex_func	bool, optional	Indicates if the function's return type is real or complex	False

Table 12-31 Boundary condition types (bc_type) for the CubicSpline class

bc_type	Description
not-a-knot	The first and second segments of the spline use the same polynomial, and this behavior is applied by default.
periodic	Data is assumed to be periodic, with first and last points connected.
clamped	First derivative of the spline is set to zero at the boundaries.
natural	Second derivative of the spline is set to zero at the boundaries.

Database	Use Case and Strength	Engineering Applications
MySQL	Web apps, general-purpose; fast, simple, stable	Sensor logging, inventory systems, web dashboards for simulations
PostgreSQL	Complex queries, data integrity; extensible and SQL-compliant	GIS (geographic information systems) for civil engineering, FEA (finite element analysis)/CFD (computational fluid dynamics) results, research data management
SQLite	Mobile/IoT, lightweight; zero-config, portable	On-device storage for drones, robotics, handheld diagnostic tools
MS SQL Server	Enterprise BI (business intelligence) and analytics; strong Microsoft ecosystem	SCADA (Supervisory Control and Data Acquisition), MES (manufacturing execution systems), production line analytics
Oracle	Mission-critical, large-scale; top performance and reliability	Aerospace datasets, utility grid management, large-scale supply chain logistics

Family	Main Purpose
System creation	Establishes the foundational mathematical models of dynamic systems. This involves translating a physical process into a standard representation, such as state-space equations (ss) or transfer functions (tf).
Interconnections	Assembles complex system architectures by programmatically defining the connections between individual subsystem models. This family implements the principles of block diagram algebra to construct feedback loops and other common control configurations.
Time domain	Evaluates the system's dynamic performance by simulating its response to standardized inputs over time. It is used to analyze transient characteristics like rise time, overshoot, and settling time, providing direct insight into real-world behavior.
Frequency domain	Analyzes system stability and performance by examining its response to sinusoidal inputs across a spectrum of frequencies. This approach is crucial for determining stability margins and understanding properties like bandwidth and resonance.
System analysis	Extracts and quantifies the intrinsic dynamic properties and performance metrics directly from the system model. This includes identifying fundamental characteristics such as poles, zeros, and damping ratios that govern the system's behavior.
Controller design	Synthesizes a control algorithm (a controller) intended to modify the behavior of a given system (the plant) to meet specific performance requirements. This family provides algorithms for classic and modern control strategies, from pole placement to optimal control.
Nonlinear and optimal	Addresses the challenges of systems whose dynamics are not linear. This includes tools for linearizing a model around an operating point and for solving optimal control problems to find the best possible performance, via the optimal module.
Model reduction	Creates simplified, lower-order models that approximate the input–output behavior of more complex, high-order systems. This is essential for reducing computational load and for designing controllers for systems that are too complex to be handled directly.

Table 8-9 A summary of basic file operations demonstrated in the chapter for .txt files

	Operation	**Used Command**
1	Write mode and write for a specific path.	file.write()
2	Read mode.	file.read()
3	Exclusive creation mode.	file.write()
4	Append mode.	file.write()
5	Binary mode (writing).	file.write()
6	Text mode (reading, writing, appending, reading and writing).	file.read(), file.write()
7	Read and write mode.	file.read(), file.write()
8	Write to a specific line.	file.readlines(), file.writelines()
9	Read a specific line.	list(), strip()
10	Read multiple lines.	list(), print()

Table 13-2 Representative Kivy widgets and layout managers

Element Name	Description
UX Widgets	
Label	A widget for rendering text
Button	A label with associated actions that are triggered when pressed
CheckBox	A two-state button that can be either checked or unchecked
Image	A widget used to display an image
Slider	A widget that looks like a scrollbar and allows selecting a value in a range
ProgressBar	A widget to visualize the progress of a task
TextInput	A box for editable plain text
ToggleButton	A button that toggles between a 'normal' and 'down' state
Switch	A two-state switch that can be active or inactive
Video	A widget used to display video files and streams
Spinner	A dropdown list for selecting one option
Layouts	
AnchorLayout	Aligns its children to a border or the center
BoxLayout	Arranges children in a vertical or horizontal box
FloatLayout	Allows placing children with arbitrary locations and sizes
GridLayout	Arranges children in a matrix (grid)
PageLayout	A simple multi-page layout that allows flipping between pages
RelativeLayout	Positions child widgets relative to the layout's position
ScatterLayout	A layout that allows for multi-touch translation, rotation, and scaling of its children
StackLayout	Arranges children vertically or horizontally, filling the space
Complex UX Widgets	
Bubble	A small popup with an arrow, often used for menus
DropDown	A versatile list that can display custom widgets
FileChooser	Provides classes for displaying and browsing the file system
Popup	A widget used to create modal popups
Spinner	A widget that provides a quick way to select one value from a set
RecycleView	A flexible view for displaying large datasets efficiently

Element Name	Description
TabbedPanel	Manages different widgets in tabs
VideoPlayer	A widget to play video with user controls
VKeyboard	An onscreen virtual keyboard
Behavior Widgets	
Scatter	Used to build interactive widgets that can be translated, rotated, and scaled
StencilView	Limits the drawing of child widgets to its bounding box
Screen Management	
ScreenManager	Manages multiple screens and their transitions

Table 12-1 SciPy subpackages and their primary functions

Subpackage	Description and User Guide
cluster	Clustering algorithms
constants	Physical and mathematical constants
differentiate	Finite difference differentiation tools
fft	Fourier Transforms (scipy.fft)
fftpack	Fast Fourier Transform routines (legacy)
integrate	Integration (scipy.integrate)
interpolate	Interpolation (scipy.interpolate)
io	File IO (scipy.io)
linalg	Linear algebra (scipy.linalg)
ndimage	Multidimensional Image Processing (scipy.ndimage)
odr	Orthogonal distance regression
optimize	Optimization (scipy.optimize)
signal	Signal processing (scipy.signal)
sparse	Sparse arrays (scipy.sparse)
spatial	Spatial data structures and algorithms (scipy.spatial)
special	Special functions (scipy.special)
stats	Statistics (scipy.stats)

Table 12-4 Attributes of the RichResult object returned by SciPy's differentiation functions

Attribute	Type	Description
success	bool array	True where the algorithm terminated successfully (status 0) and False otherwise.
status	int array	An integer representing the exit status of the algorithm: *0*: The algorithm converged to the specified tolerances. *–1*: The error estimate increased, so iteration was terminated. *–2*: The maximum number of iterations was reached. *–3*: A non-finite value was encountered. *–4*: Iteration was terminated by callback. *1*: The algorithm is proceeding normally (in callback only).
df	float array	The derivative of f at x, if the algorithm terminated successfully for derivative. The Jacobian of f at x, if the algorithm terminated successfully for jacobian.
ddf	float array	The Hessian of f at x, if the algorithm terminated successfully for hessian.
error	float array	An estimate of the error: the magnitude of the difference between the current estimate of the derivative/Jacobian/Hessian and the estimate in the previous iteration.
nit	int array	The number of iterations of the algorithm that were performed.
nfev	int array	The number of points at which f was evaluated.
x	float array	The value at which the derivative of f was evaluated, after broadcasting with args and step_direction for derivative.

Table 12-60 Parameters for the scipy.signal.tf2zpk function

Parameter	Description	Type
b	Numerator polynomial coefficients	array_like
a	Denominator polynomial coefficients	array_like

Characteristic	Root Locus Method	Pole Placement	Frequency Response Analysis
Domain	s-Plane (Time)	s-Plane (Time)	Frequency
Graphical tool	Root locus plot	s-Plane (only for choosing poles)	Bode plot
Ideal application	PID, lead/lag compensators	Controllers with a defined structure (PID), state feedback	Lead/lag compensators, PID
Mathematical model	Transfer function (G(s))	Transfer function (G(s)), state space for Ackermann's formula	Transfer function (G(s))

```
from kivy.app import App
from kivy.lang import Builder
# We define the UI and widgets in one string
kv_code = """
AnchorLayout:
  anchor_x: 'right'
  anchor_y: 'bottom'
  Button:
    text: 'Anchored'
    size_hint: None, None
    size: '100dp', '36dp'
"""
class EngApp(App):
  def build(self):
    return Builder.load_string(kv_code)
if __name__ == '__main__':
  EngApp().run()
```

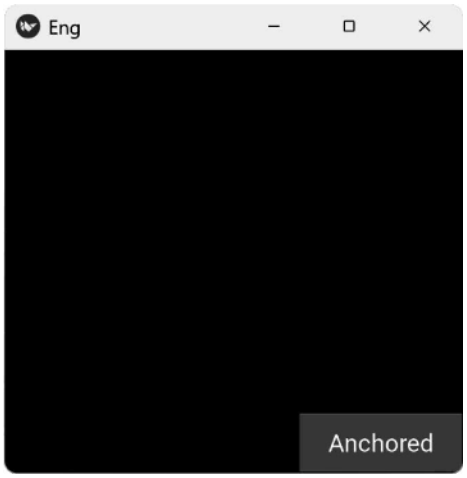


Figure 13-9 Positioning a button at the bottom-right corner using AnchorLayout

Table 12-21 Key functions for numerical integration and solving ODEs in scipy.integrate

Function	Description
quad	Compute a definite integral.
quad_vec	Adaptive integration of a vector-valued function.
cubature	Adaptive cubature of a multidimensional array–valued function.
dblquad	Compute a double integral.
tplquad	Compute a triple (definite) integral.
nquad	Integration over multiple variables.
tanhsinh	Evaluate a convergent integral numerically using tanh-sinh quadrature.
fixed_quad	Compute a definite integral using fixed-order Gaussian quadrature.
newton_cotes	Return weights and error coefficient for Newton–Cotes integration.
lebedev_rule	Lebedev quadrature.
qmc_quad	Compute an integral in N dimensions using Quasi-Monte Carlo quadrature.
trapezoid	Integrate along the given axis using the composite trapezoidal rule.
cumulative_trapezoid	Cumulatively integrate y(x) using the composite trapezoidal rule.
simpson	Integrate y(x) using samples along the given axis and the composite Simpson's rule.
cumulative_simpson	Cumulatively integrate y(x) using the composite Simpson's 1/3 rule.
romb	Romberg integration using samples of a function.
nsum	Evaluate a convergent finite or infinite series.
solve_ivp	Solve an initial value problem for a system of ODEs.
RK23, RK45, DOP853, Radau, BDF, LSODA	Various ODE solvers.
solve_bvp	Solve a boundary value problem for a system of ODEs.

Table 12-33 Return values for the make_splprep function

Return Value	Description
spl	A BSpline instance representing the smoothed approximation to x
u	The values of the parameters

Table 3-4 Common error messages for list methods

Method	Error Type	Description
user_list.append(x)	TypeError	Raised when append() is called with an invalid number of arguments, for example, append(1, 2).
user_list.extend(iterable)	TypeError	Triggered when a non-iterable object is passed as an argument.
user_list.insert(i, x)	TypeError	Raised if the index is not an integer or the method is called with an invalid number of arguments. Out-of-range integer indexes are clamped.
user_list.remove(x)	ValueError	Occurs when the list doesn't contain the element to be removed.
user_list.pop([i])	IndexError	Raised when the index is out of range or there's nothing to pop from an empty list.
user_list.index(x, start, end)	ValueError	Raised when the element is not found within the specified range.
user_list.sort()	TypeError	Occurs when trying to sort a list containing elements of incompatible types.

Table 12-28 Comprehensive list of functions and classes available in scipy.interpolate

Function/Class	Description
Univariate Interpolation	
BarycentricInterpolator()	Interpolating polynomial for a set of points.
KroghInterpolator()	Interpolating polynomial for a set of points.
barycentric_interpolate()	Convenience function for polynomial interpolation.
krogh_interpolate()	Convenience function for polynomial interpolation.
PchipInterpolator()	Convenience function for Pchip interpolation.
CubicHermiteSpline()	Piecewise cubic interpolator matching values and first derivatives.
PchipInterpolator()	Pchip 1D monotonic cubic interpolation.
Akima1DInterpolator()	Akima interpolator.
CubicSpline()	Cubic spline data interpolator.
PPoly()	Piecewise polynomial in terms of coefficients and breakpoints.
BPoly()	Piecewise polynomial in terms of coefficients and breakpoints.
FloaterHormannInterpolator()	Floater–Hormann barycentric rational interpolation.
Multivariate Interpolation	
griddata()	Interpolate unstructured N-D data.
LinearNDInterpolator()	Piecewise linear interpolator in N > 1 dimensions.
NearestNDInterpolator()	Nearest-neighbor interpolator in N dimensions.
CloughTocher2DInterpolator()	Piecewise cubic, C1 smooth, curvature-minimizing interpolator in 2D.
RBFInterpolator()	Radial basis function (RBF) interpolation in N dimensions.
RegularGridInterpolator()	Interpolator on a regular or rectilinear grid in arbitrary dimensions.
RectBivariateSpline()	Bivariate spline approximation over a rectangular mesh.
1D Splines	
BSpline()	Univariate spline in the B-spline basis.
make_interp_spline()	Compute interpolating B-spline coefficients.
make_lsq_spline()	Compute LSQ-based fitting B-spline coefficients.

Function/Class	Description
make_smoothing_spline()	Compute smoothing cubic spline coefficients.
generate_knots()	Replicate FITPACK's knot vector construction.
make_splrep()	Find B-spline representation of a 1D function.
make_splprep()	Find smoothed B-spline representation of a parametric N-D curve.
2D Splines	
RectBivariateSpline()	Bivariate spline approximation over a rectangular mesh.
RectSphereBivariateSpline()	Bivariate spline approximation over a rectangular mesh on a sphere.
BivariateSpline()	Base class for bivariate splines.
SmoothBivariateSpline()	Smooth bivariate spline approximation.
SmoothSphereBivariateSpline()	Smooth bivariate spline approximation in spherical coordinates.
LSQBivariateSpline()	Weighted least-squares bivariate spline approximation.
LSQSphereBivariateSpline()	Weighted least-squares bivariate spline approximation in spherical coordinates.
bisplrep()	Find bivariate B-spline representation of a surface.
bisplev()	Evaluate a bivariate B-spline and its derivatives.
Rational Approximation	
pade()	Return Pade approximation to a polynomial as the ratio of two polynomials.
AAA()	AAA (Antoulas–Anderson algorithm) real or complex rational approximation.
Additional Tools	
lagrange()	Return a Lagrange interpolating polynomial.
approximate_taylor_polynomial()	Estimate the Taylor polynomial of f at a given point x0.

Table 13-4 Kivy widget position properties and coordinate values

Property	Description	Range
pos	A ReferenceListProperty of (x, y). Sets the bottom-left corner of the widget.	Any pair of numeric pixel values, for example, (100, 50). Visibility is within the parent's (width, height).
x, y	NumericProperty values for the individual components of pos. Changing x or y will update pos and vice versa.	Any numeric pixel value. Visibility is typically between 0 and the parent's dimension (width or height).
center	A ReferenceListProperty of (center_x, center_y). Sets the center of the widget.	Any pair of numeric pixel values.
center_x, center_y	AliasProperty values for the center coordinates. Setting these will adjust x and y accordingly.	Any numeric pixel value.
right	An AliasProperty representing the x coordinate of the widget's right edge (x + width). Setting it moves the widget.	
top	An AliasProperty representing the y coordinate of the widget's top edge (y + height). Setting it moves the widget.	

Table 12-54 Return values and optional outputs for the scipy.optimize.curve_fit function

Return Value	Type	Description
If full_output=False (Default)		
popt	array	Optimal values for the parameters
pcov	2D array	Estimated covariance of the parameters
If full_output=True		
infodict	dictionary	Dictionary with optional outputs: *nfev*: The number of function calls *fvec*: The residual values evaluated at the solution *fjac*, *ipvt*, *qtf*: Information about the Jacobian matrix and its factorization (for method 'lm' only)
mesg	str	String message about the solution
ier	int	Integer flag indicating solution status

Table 3-5 List comprehension filtering example

Fruit	Contains “i”?	Added to newlist?
apple	No	No
banana	No	No
cherry	No	No
kiwi	Yes	Yes
mango	No	No

Table 6-10 A trace showing how the break statement exits a for loop prematurely

Iteration	Value of "num"	Condition (num == 3)	Action	Output
1	1	False	Print num.	1
2	2	False	Print num.	2
3	3	True	Print "Found 3!" and break.	Found 3!

Table 7-11 Special methods for object construction and lifecycle management

Method	Purpose
__init__	Sets up the initial state of the object upon instantiation
__new__	Creates and returns a new instance of the class
__del__	Performs cleanup tasks when the object is about to be deleted
__call__	Enables the object to be invoked as if it were a function
__init_subclass__	Modifies the behavior when a new subclass is created

Table 10-1 Common graphs used across various engineering disciplines

Discipline	Graphs
Electrical and electronics	Bode plot, Nyquist plot, Nichols plot, root locus plot, time–current curve, current–voltage curve, Smith chart, eye diagram, constellation diagram
Mechanical and aerospace	Stress–strain curve, S–N curve, Moody chart, psychrometric chart, Campbell diagram, pump and system curves
Chemical engineering	Phase diagram, McCabe–Thiele diagram, T-xy/P-xy diagram, Arrhenius plot
Civil and geotechnical	Particle size distribution, Mohr's circle, influence line diagram, interaction diagram
General	Contour/vector plots, surface plot, Pareto chart

```
from kivy.app import App
from kivy.lang import Builder
# We define the UI and widgets in one string
kv_code = """
StackLayout:
 orientation: 'lr-tb'
 spacing: 5
 padding: 5
 Button:
  text: '1'
  size_hint: None, None
  size: '100dp', '36dp'
 Button:
  text: '2'
  size_hint: None, None
  size: '100dp', '36dp'
 Button:
  text: '3'
  size_hint: None, None
  size: '100dp', '36dp'
 Button:
  text: '4'
  size_hint: None, None
  size: '100dp', '36dp'
"""
class EngApp(App):
 def build(self):
  return Builder.load_string(kv_code)
if __name__ == '__main__':
 EngApp().run()
```

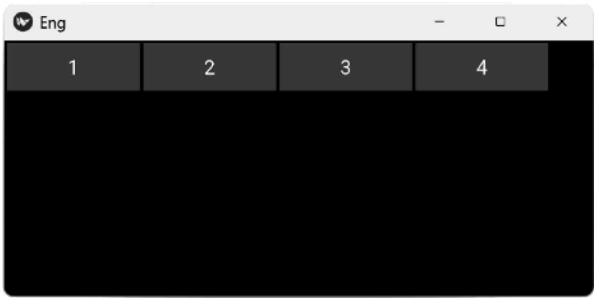


Figure 13-10 Arrangement of four buttons using StackLayout

Table 11-9 Detailed summary of operation families and representative functions in NumPy

Category	Types of Operations
Arithmetic operations	Elementwise addition, subtraction, multiplication, division, remainder, and powers. Examples: a + b, a - b, a * b, a / b, a ** 2, np.add, np.subtract, np.multiply, np.divide, np.floor_divide, np.mod, np.power.
Aggregation and reduction	Summarize arrays by combining elements along axes. Examples: np.sum, np.prod, np.mean, np.std, np.var, np.min, np.max, np.argmin, np.argmax, np.cumsum, np.cumprod, np.diff, np.trace.
Trigonometric and hyperbolic functions	Elementwise trigonometric and hyperbolic functions used in waves, rotations, and periodic models. Examples: np.sin, np.cos, np.tan, np.arcsin, np.arccos, np.arctan, np.sinh, np.cosh, np.tanh, np.arcsinh, np.arctanh.
Exponential, logarithmic, and power	Nonlinear transformations for growth models, scaling, and normalization. Examples: np.exp, np.expm1, np.log, np.log10, np.log2, np.log1p, np.sqrt, np.cbrt, np.square.
Logical, boolean, and comparison	Elementwise comparisons and logical operations, boolean masks, and conditional selection. Examples: a > 0, a == b, np.greater, np.less, np.equal, np.logical_and, np.logical_or, np.logical_not, np.logical_xor, np.where, np.all, np.any.
Sorting, searching, and set operations	Sorting, ranking, searching, and set-like operations on arrays. Examples: np.sort, np.argsort, np.lexsort, np.searchsorted, np.partition, np.argpartition, np.unique, np.intersect1d, np.union1d, np.setdiff1d, np.isin.
Linear algebra operations	Matrix and vector operations (some in numpy.linalg). Examples: np.dot, a @ b, np.vdot, np.inner, np.outer, np.matmul, np.linalg.inv, np.linalg.solve, np.linalg.eig, np.linalg.svd, np.linalg.norm, np.linalg.det.
Fourier Transform operations	Fast Fourier Transforms for converting between time and frequency domains (in numpy.fft). Examples: np.fft.fft, np.fft.ifft, np.fft.fftn, np.fft.ifftn, np.fft.rfft, np.fft.irfft, np.fft.fftshift, np.fft.ifftshift.
Complex number operations	Operations specific to complex-valued arrays. Examples: np.real, np.imag, np.conj, np.angle, np.abs (magnitude of complex numbers).
Array creation	Functions for constructing new arrays for computation and initialization. Examples: np.array, np.zeros, np.ones, np.full, np.empty, np.eye, np.identity, np.arange, np.linspace, np.logspace, np.meshgrid.
Array manipulation	Reshaping and reorganizing arrays. Examples: np.reshape, np.ravel, np.transpose, np.swapaxes, np.moveaxis, np.expand_dims, np.squeeze, np.stack, np.vstack, np.hstack, np.concatenate, np.split, np.repeat, np.tile.
Random number generation	Random sampling and pseudo-random streams (in numpy.random). Examples: np.random.default_rng, rng.random, rng.integers, rng.normal, rng.uniform, rng.poisson, rng.choice, rng.permutation, rng.shuffle.

Category	Types of Operations
NaN/Inf handling and special values	Utilities to handle missing or non-finite values. Examples: np.isnan, np.isfinite, np.isinf, np.nan_to_num, np.nanmean, np.nanstd, np.nanmin, np.nanmax.
Rounding and approximation	Rounding and discrete approximations. Examples: np.round, np.rint, np.floor, np.ceil, np.trunc, np.fix.
Polynomial and numerical tools	Basic polynomial evaluation and numerical helpers (some in numpy.polynomial). Examples: np.polyval, np.polyfit, and the numpy.polynomial submodule (Chebyshev, Legendre, etc.).
Histograms and binning	Histogram computation and bin edges. Examples: np.histogram, np.histogram_bin_edges, np.bincount, np.digitize.
String and text operations	Vectorized string operations (mostly via np.char). Examples: np.char.add, np.char.lower, np.char.upper, np.char.find, np.char.replace, np.char.strip.
Masked arrays	Operations on arrays with missing/invalid entries (in numpy.ma). Examples: np.ma.array, np.ma.masked, np.ma.masked_invalid, np.ma.filled, and masked versions of sum, mean, etc.
Testing and comparison utilities	Tools for validating numerical results (in numpy.testing). Examples: np.testing.assert_allclose, np.testing.assert_array_equal, np.allclose, np.array_equal.

Table 12-5 Summary of Fast Fourier Transform (FFT) functions in scipy.fft categorized by dimensionality

Topic	Function	Description
1D Fourier Transforms	fft(x)	Computes the 1D DFT (Discrete Fourier Transform) of a signal
	ifft(X)	Reconstructs the original signal from its DFT
	rfft(x)	Computes the 1D DFT for real input
	irfft(X)	Reconstructs the original real-valued signal
	hfft(x)	Computes the FFT of a Hermitian-symmetric signal
	ihfft(X)	Reconstructs the original signal from its Hermitian-symmetric FFT
2D Fourier Transforms	fft2(x)	Computes the 2D DFT of a 2D array
	ifft2(X)	Reconstructs the original 2D signal from its 2D DFT
	rfft2(x)	Computes the 2D DFT for real 2D input
	irfft2(X)	Reconstructs the original real-valued 2D signal
	hfft2(x)	Computes the 2D FFT of a Hermitian-symmetric complex array
	ihfft2(X)	Reconstructs the original 2D signal from its Hermitian-symmetric 2D FFT
N-D Fourier Transforms	fftn(x)	Computes the N-dimensional DFT of an array
	ifftn(X)	Reconstructs the original N-dimensional signal from its DFT
	rfftn(x)	Computes the N-dimensional DFT for real input
	irfftn(X)	Reconstructs the original real-valued N-dimensional signal
	hfftn(x)	Computes the N-D FFT of a Hermitian-symmetric complex input
	ihfftn(X)	Reconstructs the original N-dimensional signal from its Hermitian-symmetric FFT

Table 6-4 A trace of a for loop iterating over a list of numbers

Iteration Number	Current Value of "number"	Output
1	2	2
2	4	4
3	6	6

Table 9-5 Attributes of the decimal arithmetic context

Attribute	Description
prec	*Precision*: An integer for the total number of significant digits to store in a calculation's result.
rounding	*Rounding Mode*: The rule for how to round numbers when they exceed the set precision.
traps	*Error Traps*: A dictionary where you can set certain conditions like DivisionByZero or Overflow, to raise an exception instead of just returning a special value like Infinity.
flags	*Signal Flags*: A dictionary that records whether a condition occurred during a calculation, even when it was not trapped (e.g., whether the result was Inexact).
Emin	The minimum possible exponent. Controls how small a number can be.
Emax	The maximum possible exponent. Controls how large a number can be.
capitals	An integer (1 or 0). If 1, scientific notation uses a capital "E" (e.g., 1.2E+3). If 0, it uses a lowercase "e" (1.2e+3).

Table 10-4 The six basic steps to create a plot with Matplotlib

Stage	Action
1. Load plotting tools.	Import Matplotlib and any numerical library required by the calculation.
2. Obtain the engineering data.	Define measured values, simulation results, or calculated variables.
3. Create the figure.	Initialize the figure and obtain the axes that will receive the data.
4. Represent the results.	Select a plot type that is appropriate for the physical meaning of the variables.
5. Add technical information.	Identify quantities, units, operating conditions, and relevant reference values.
6. Inspect or export the figure.	Display the result for analysis or prepare the figure for a report.

Table 6-12 A trace illustrating the behavior of the pass statement within a for loop

Iteration	Value of "i"	Condition (i == 3)	Action	Output
1	1	False	Print "i" value.	1
2	2	False	Print "i" value.	2
3	3	True	Pass.	None
4	4	False	Print "i" value.	4
5	5	False	Print "i" value.	5

Table 12-53 Parameters for the scipy.optimize.curve_fit function

Parameter	Description	Default Value
f	The model function to fit	Required
xdata	The independent variable data	Required
ydata	The dependent variable data	Required
p0	Initial guess for the parameters	None (uses 1 for all parameters)
sigma	Uncertainty in ydata	None (uses 1D sigma filled with ones)
absolute_sigma	Whether sigma is absolute or relative	False
check_finite	Whether to check for finite values in input arrays	True (if nan_policy is not specified)
bounds	Lower and upper bounds on parameters	(–inf, inf) (no bounds)
method	Method to use for optimization	'lm' (for unconstrained) or 'trf' (for bounded)
jac	Function to compute the Jacobian matrix	None (uses numerical approximation)
full_output	Whether to return additional output information	False
nan_policy	How to handle NaN values in input arrays	None (no special handling)
****kwargs**	Additional keyword arguments for leastsq or least_squares	None

Table 12-50 A selection of commonly used optimization methods available in scipy.optimize.minimize

Method	Category	Application
Nelder–Mead	Don't require derivatives	General-purpose optimization, robust but slow
L-BFGS-B	Bound constraints	Bound-constrained optimization, efficient for large problems
BFGS	Quasi-Newton	Quasi-Newton method, efficient for smooth functions
SLSQP	Constrained optimization	Sequential least-squares programming, efficient for nonlinear constraints

Table 7-6 Comparison between protected and private attributes

Aspect	Protected Attributes	Private Attributes
Prefix	Starts with a single underscore like _attribute_name.	Starts with a double underscore like __attribute_name.
Access	Accessible within the class and subclasses.	Intended for use within the class; still accessible through name mangling.
Enforcement	Not strictly enforced, convention only. It is possible to use the decorator @property to control access.	
Purpose	For attributes intended for internal use but accessible to subclasses.	For attributes intended to avoid accidental external access.
Name mangling	No name mangling.	Name mangling changes the attribute name to _ClassName__attribute_name.
Subclass access	Accessible in subclasses.	Not directly accessible in subclasses.
External access	Can be accessed directly (not recommended).	Can be accessed through the mangled name (not recommended).

Table 3-10 Set-like operations between dictionaries

Operation	Op.	Description
Union	|	Combines both dictionaries; values from the second replace duplicate keys (Python 3.9+).
Intersection	& keys	Builds a dictionary from keys common to both key views.
Difference	- keys	Builds a dictionary from keys present only in the first key view.
Symmetric difference	^ keys	Builds a dictionary from keys present in only one key view.

Table 12-11 Parameters and descriptions for the scipy.fft.fht function

Parameter	Type	Description
a	array_like (…, n)	Real periodic input array, uniformly logarithmically spaced. For multidimensional input, the transform is performed over the last axis.
dln	float	Uniform logarithmic spacing of the input array.
mu	float	Order of the Hankel transform, any positive or negative real number.
offset	float (optional)	Offset of the uniform logarithmic spacing of the output array.
bias	float (optional)	Exponent of power-law bias, any positive or negative real number.

Table 11-14 Operations on complex-valued arrays

Example	Meaning
a + 5	Adds 5 to every element
a * [1, 2, 3]	Broadcasts row/column depending on shape
a ** 2	Squares the array elementwise

Table 4-5 Advanced and special-case import methods

Method	Description	Example
Importing third-party modules	Imports modules installed via a package manager like "pip"	`import numpy as np` `print(np.array([1, 2, 3]))`

Conditional imports	Imports modules only when a specific condition is met	`if some_condition:`
		`import math`
		`print(math.sqrt(16))`

Lazy imports	Defers the import of a module until it is first used	`import importlib`
		`math = importlib.import_module('math')`
		`print(math.sqrt(16))`

Importing with "__import__"	Dynamically imports a module using the "__import__" function	`module = __import__('math')` `print(module.sqrt(16))`
Importing packages	Imports an entire package or specific modules within it	`import my_package` `from my_package import module_a`

Circular imports	Occurs when two or more modules depend on each other. Should be avoided	`# module_a.py`
		`from module_b import func_b`
		`# module_b.py`
		`from module_a import func_a`

Table 7-10 Dunder methods in Python, categorized by functionality

Category	Method	Description
Object lifecycle	__new__	Creates a new instance of the class
	__init__	Initializes a new instance of the class
	__del__	Destructor method, called when the instance is about to be destroyed
	__repr__	Returns a string representation of the object, used for debugging
	__str__	Returns a human-readable string representation of the object
	__format__	Returns a formatted string representation of the object
	__hash__	Returns the hash value of the object
	__bool__	Returns the boolean value of the object
	__init_subclass__	Called when a class is subclassed
Attribute access	__getattr__	Called when an attribute lookup has not found the attribute
	__getattribute__	Called unconditionally to implement attribute accesses
	__setattr__	Called when an attribute assignment is attempted
	__delattr__	Called when an attribute deletion is attempted
	__dir__	Called when dir() is called on the object
Container methods	__len__	Returns the length of the container
	__getitem__	Returns the item at the specified key
	__setitem__	Sets the item at the specified key
	__delitem__	Deletes the item at the specified key
	__iter__	Returns an iterator object
	__reversed__	Returns a reverse iterator
	__contains__	Implements membership test operators (in and not in)
	__missing__	Called when a key is not found in a dictionary
Numeric operations	__matmul__	Defines the matrix multiplication operator (@)
	__rmatmul__	Defines the reflected matrix multiplication operator
	__imatmul__	Defines the in-place matrix multiplication operator (@=)
	__add__	Defines the addition operator (+)

Category	Method	Description
	__sub__	Defines the subtraction operator (-)
	__mul__	Defines the multiplication operator (*)
	__truediv__	Defines the division operator (/)
	__floordiv__	Defines the floor division operator (//)
	__mod__	Defines the modulo operator (%)
	__neg__	Defines the unary negation operator (-)
	__pos__	Defines the unary positive operator (+)
	__abs__	Defines the absolute value function (abs())
	__pow__	Defines the power operator (**)
	__lshift__	Defines the left shift operator (<<)
	__rshift__	Defines the right shift operator (>>)
	__and__	Defines the bitwise AND operator (&)
	__xor__	Defines the bitwise XOR operator (^)
	__or__	Defines the bitwise OR operator (|)
	__radd__	Defines the reflected addition operator
	__rsub__	Defines the reflected subtraction operator
	__rmul__	Defines the reflected multiplication operator
	__rtruediv__	Defines the reflected division operator
	__rfloordiv__	Defines the reflected floor division operator
	__rmod__	Defines the reflected modulo operator
	__rpow__	Defines the reflected power operator
	__rlshift__	Defines the reflected left shift operator
	__rrshift__	Defines the reflected right shift operator
	__rand__	Defines the reflected bitwise AND operator
	__rxor__	Defines the reflected bitwise XOR operator
	__ror__	Defines the reflected bitwise OR operator
	__iadd__	Defines the in-place addition operator (+=)
	__isub__	Defines the in-place subtraction operator (-=)
	__imul__	Defines the in-place multiplication operator (*=)
	__itruediv__	Defines the in-place division operator (/=)
	__ifloordiv__	Defines the in-place floor division operator (//=)

Category	Method	Description
	__imod__	Defines the in-place modulo operator (%=)
	__ipow__	Defines the in-place power operator (**=)
	__ilshift__	Defines the in-place left shift operator (<<=)
	__irshift__	Defines the in-place right shift operator (>>=)
	__iand__	Defines the in-place bitwise AND operator (&=)
	__ixor__	Defines the in-place bitwise XOR operator (^=)
	__ior__	Defines the in-place bitwise OR operator (|=)
	__invert__	Defines the bitwise inversion operator (~)
	__complex__	Defines the conversion to a complex number
	__int__	Defines the conversion to an integer
	__float__	Defines the conversion to a float
	__round__	Defines the rounding function (round())
	__index__	Defines the conversion to an integer for use in slicing
Comparison operators	__lt__	Defines the less than operator (<)
	__le__	Defines the less than or equal to operator (<=)
	__eq__	Defines the equality operator (==)
	__ne__	Defines the inequality operator (!=)
	__gt__	Defines the greater than operator (>)
	__ge__	Defines the greater than or equal to operator (>=)
Callable function	__call__	Allows an instance of the class to be called as a function
Context management	__enter__	Enters the runtime context related to this object
	__exit__	Exits the runtime context related to this object
Descriptors	__get__	Called to get the attribute of the owner class (usually a property)
	__set__	Called to set the attribute of the owner class
	__delete__	Called to delete the attribute of the owner class
Pickling	__getnewargs__	Returns a tuple of arguments to be passed to __new__ when unpickling
	__getstate__	Returns the state of the object for pickling
	__setstate__	Sets the state of the object when unpickling
	__reduce__	Returns a tuple to be used for pickling

Category	Method	Description
	__reduce_ex__	Returns a tuple to be used for pickling with a specific protocol
Miscellaneous	__subclasses__	Returns a list of immediate subclasses
	__copy__	Returns a shallow copy of the object
	__deepcopy__	Returns a deep copy of the object
	__bytes__	Returns a bytes representation of the object
	__sizeof__	Returns the size of the object in memory

Table 8-13 A summary of commands used for common .xlsx file operations with the openpyxl library

Number	Operation	Used Command
1	Read an .xlsx file.	load_workbook(), sheet.iter_rows()
2	Read all sheets.	workbook.sheetnames, sheet.iter_rows()
3	Read a specific sheet.	Workbook[], sheet.iter_rows()
4	Read a specific row.	sheet[]
5	Read a specific column.	sheet[]
6	Read non-empty rows or columns.	sheet.iter_rows(), any()
7	Read a range of cells.	sheet[], cell.value
8	Read a specific cell.	sheet[].value or sheet.cell().value
9	Write to an .xlsx file.	Workbook(), workbook.active, sheet[], workbook.save()
10	Append to an .xlsx file.	load_workbook(), sheet.append([]), workbook.save()
11	Get workbook information.	len(), sheet.max_row, sheet.max_column
12	Count non-empty rows and columns.	sheet.iter_rows(), sheet.iter_cols()

Table 2-6 Type conversion methods in Python

Conversion Method	Description	Example Code	Output	Resulting Type
int(x)	Converts x to an integer (truncates decimals if x is float)	int(3.14)	3	<class 'int'>
float(x)	Converts x to a floating-point number	float(42)	42.0	<class 'float'>
str(x)	Converts x to a string	str(42)	"42"	<class 'str'>
bool(x)	Converts x to a boolean. Nonzero/non-empty = True	bool(0)	False	<class 'bool'>
list(x)	Converts x to a list	list("abc")	['a', 'b', 'c']	<class 'list'>
tuple(x)	Converts x to a tuple	tuple([1, 2, 3])	(1, 2, 3)	<class 'tuple'>
set(x)	Converts x to a set (removes duplicates)	set([1, 2, 2, 3])	{1, 2, 3}	<class 'set'>
dict(x)	Converts x to a dictionary (must be key–value pairs)	dict([('a', 1), ('b', 2)])	{'a':1,'b': 2}	<class 'dict'>
bin(x)	Converts integer x to a binary string prefixed with 0b	bin(42)	'0b101010'	<class 'str'>
oct(x)	Converts integer x to an octal string prefixed with 0o	oct(42)	'0o52'	<class 'str'>
hex(x)	Converts integer x to a hexadecimal string prefixed with 0x	hex(42)	'0x2a'	<class 'str'>
ord(x)	Converts a single character x to its Unicode code point	ord('A')	65	<class 'int'>
chr(x)	Converts an integer x to its corresponding Unicode character	chr(65)	'A'	<class 'str'>
Complex (x, y)	Converts x and y to a complex number (x + yj)	complex(2, 3)	(2+3j)	<class 'complex'>

```
from kivy.app import App
from kivy.lang import Builder
kv_code = """
ScatterLayout:
 Button:
  text: 'drag, pinch, rotate'
  size_hint: None, None
  size: '100dp', '36dp'
  pos_hint: {'center_x': 0.5,
'center_y': 0.5}
"""
class EngApp(App):
 def build(self):
  return Builder.load_string(kv_code)
if __name__ == '__main__':
 EngApp().run()
```

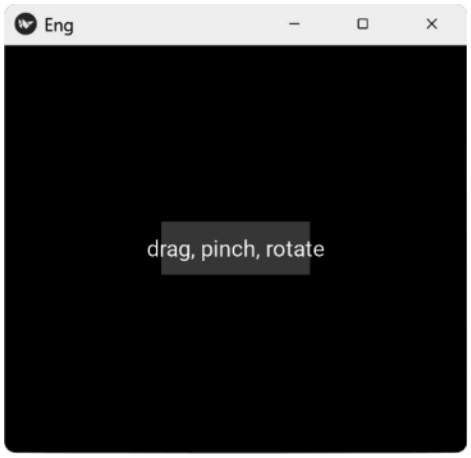


Figure 13-13 Interactive translation, scaling, and rotation using ScatterLayout

Table 2-4 Method for string operations

Category	Methods	Description
1. Converting to uppercase or lowercase	upper(), lower(), capitalize(), title(), swapcase()	These methods are used to change the case of a string, either entirely or by capitalizing specific words or characters.
2. Finding and replacing	find(), replace()	These methods are used to find a substring's index or replace part of the string with another.
3. Checking string properties	startswith(), endswith(), isalpha(), isdigit(), isalnum(), isspace(), islower(), isupper()	These methods check specific properties of a string, such as whether it starts/ends with a substring or if all characters are alphanumeric.
4. Splitting and joining strings	split(), join()	These methods allow splitting a string into a list based on a delimiter and joining a list into a string.
5. Stripping whitespace and characters	strip(), lstrip(), rstrip()	These methods remove leading and trailing whitespace or specific characters from the string.
6. Padding and justifying	zfill(), ljust(), rjust(), center()	These methods adjust the width of a string by adding padding to the left, right, or center.
7. Working with multi-line strings	splitlines()	This method splits a multi-line string into a list, with each line as a separate list element.
8. Formatting strings	format()	This method allows for string formatting, inserting variables or expressions into a string.
9. Checking for substrings	find()	This method checks if a substring is present and returns the index of its first occurrence or –1 if not found.

Table 8-11 A summary of commands used for various .csv file operations

	Operation	Used Command
1	Read an entire CSV file as a string.	file.read()
2	Read all lines.	file.readlines()
3	Read the first line.	file.readline()
4	Read a specific line.	file.readlines()[index]
5	Read a specific cell.	list(csv.reader(file))[row][col]
6	Iterate through lines.	csv.reader(file)
7	Read a specific row.	list(csv.reader(file))[row]
8	Convert .csv to a list.	list(csv.reader(file))
9	Append a new row.	csv.writer(file).writerow()
10	Modify a specific row.	writerows()
11	Insert a row at a specific position.	insert(), writerows()
12	Delete a specific row.	del, writerows()
13	Write to a specific cell.	writerows()

Table 10-5 Common customization options for Matplotlib chart elements

Chart Element	Customization Options	Example Code
Text and Labels		
Title	Set the main title for a subplot. Control font size, font weight, and color.	ax.set_title("My Chart Title", fontsize=16)
Axis labels	Set the text for the x- and y-axes. Control padding with labelpad.	ax.set_xlabel("X-Axis Label")
Annotations	Add descriptive text to any point on the chart, with optional arrows.	ax.text(x, y, "An annotation")
Legends		
Legend	Add a box that identifies multiple plotted datasets. Requires a label on each plot.	ax.plot(x, y, label="Sine") ax.legend()
Position and style	Control the location (loc), number of columns (ncol), and removal of the frame.	ax.legend(loc='upper left', ncol=2, frameon=False)
Grid and Ticks		
Grid lines	Display a grid to help read data points. Control color, line style, and line width.	ax.grid(True, color='gray', linestyle='--', linewidth=0.5)
Tick locations	Manually set the positions of the tick marks on an axis.	ax.set_xticks([0, 1, 2, 3])
Tick labels	Manually set the text labels for the tick marks.	ax.set_xticklabels(['A', 'B', 'C', 'D'])
Tick parameters	Customize the appearance of ticks, labels, and gridlines (e.g., color, size).	ax.tick_params(axis='y', labelcolor='blue')
Lines and Spines		
Line styles	Change the color, width (linewidth), and style (linestyle) of plotted lines.	ax.plot(x, y, color='red', linewidth=2, linestyle=':')
Markers	Add markers to data points and control their style (marker), size, and color.	ax.plot(x, y, marker='o', markersize=5)
Spines	The boundary lines of the plot. Can be hidden or have their color changed.	ax.spines['right'].set_visible(False)

Table 7-2 Key concepts and applications of common programming paradigms

Paradigm Name	Key Concepts	Common Applications
Object-oriented programming (OOP)	Classes, encapsulation, inheritance, polymorphism, abstraction	GUI development, games, simulation, large-scale software systems
Procedural programming	Functions, sequences, loops, conditionals	System programming, embedded systems, scientific computing
Functional programming	Pure functions, immutability, higher-order functions, recursion	Data processing, parallel programming, mathematical computations
Multi-paradigm programming	Supports multiple paradigms	Web development, data science, general-purpose programming
Imperative programming	Assignment, control flow, loops, conditionals	Operating systems, device drivers, real-time systems
Declarative programming	Declarative statements, constraints, rules	Database queries, user interface design, configuration management

Table 12-55 Main categories of functions available in the scipy.signal subpackage

Category	Description	Example Functions
Convolution and correlation	Perform convolution and correlation on N-dimensional arrays.	convolve, correlate, fftconvolve, oaconvolve
B-splines	Compute B-spline coefficients and evaluate splines.	gauss_spline, cspline1d, qspline1d, cspline2d
Filtering	Apply median, Wiener, FIR, and IIR filters.	medfilt, medfilt2d, wiener, lfilter, sosfilt, filtfilt
Filter design	Design filters with specific properties.	firwin, firwin2, iirdesign, iirfilter, butter, cheby1
Continuous-time linear systems	Simulate and analyze continuous-time systems.	lsim, impulse, step, freqresp, bode
Discrete-time linear systems	Simulate and analyze discrete-time systems.	dlsim, dimpulse, dstep, dfreqresp, dbode
LTI representations	Convert between different LTI system representations.	tf2zpk, tf2ss, zpk2tf, ss2tf, ss2zpk
Waveforms	Generate waveforms like chirps, sawtooth waves, and square waves.	chirp, gausspulse, sawtooth, square, unit_impulse
Window functions	Obtain window functions for signal processing.	get_window
Peak finding	Identify and analyze peaks in signals.	find_peaks, peak_prominences, peak_widths, argrelmin
Spectral analysis	Estimate power spectral densities and analyze signals in frequency domain.	periodogram, welch, csd, coherence, spectrogram
Short-Time Fourier Transform	Compute Short-Time Fourier Transforms and their inverses.	stft, istft, ShortTimeFFT
Chirp Z-transform and Zoom FFT	Compute frequency responses around a spiral in the Z plane.	czt, zoom_fft, CZT, ZoomFFT

Table 9-1 Python's native math operations

Op./Fun.	Description	Example
+	Addition	Adds two operands. 5 + 3 results in 8.
-	Subtraction	Subtracts the right operand from the left. 5 - 3 results in 2.
*****	Multiplication	Multiplies two operands. 5 * 3 results in 15.
/	Division	Divides the left operand by the right. With built-in int and float operands, the result is a float; 5 / 3 results in 1.666 ….
//	Floor division	Returns the floor of the quotient. With integers, 5 // 3 results in 1; with a float operand, 5.0 // 3 results in 1.0.
%	Modulus	Returns the remainder of a division. 5 % 3 results in 2.
**	Exponentiation	Raises the left operand to the power of the right operand. 5 ** 3 results in 125.
-	Unary negation	Negates the operand. –5 results in –5.
+	Unary plus	Returns the operand as is. +5 results in 5.
+=	Addition assignment	Adds the right operand to the left and assigns the result to the left operand. x = 5; x += 3 results in x being 8.
-=	Subtraction assignment	Subtracts the right operand from the left and assigns the result to the left operand. x = 5; x -= 3 results in x being 2.
***=**	Multiplication assignment	Multiplies the right operand with the left and assigns the result to the left operand. x = 5; x *= 3 results in x being 15.
/=	Division assignment	Divides the left operand by the right and assigns the result to the left operand. x = 5; x /= 3 results in x being 1.666 ….
//=	Floor division assignment	Performs floor division and assigns the result to the left operand. The result type follows the operand types.
%=	Modulus assignment	Takes the modulus of the two operands and assigns the result to the left operand. x = 5; x %= 3 results in x being 2.
****=**	Exponentiation assignment	Raises the left operand to the power of the right and assigns the result to the left operand. x = 5; x **= 3 results in x being 125.
abs()	Absolute value	Returns the absolute value of a number. abs(-5) results in 5.
round()	Rounding	Rounds a number to a given precision in decimal digits. round(3.14159, 2) results in 3.14.

Op./Fun.	Description	Example
pow()	Power	Returns the first argument raised to the power of the second argument. pow(5, 3) results in 125.
divmod()	Divmod	Takes two numbers as arguments and returns a pair of numbers consisting of their quotient and remainder. divmod(5, 3) results in (1, 2).
max()	Maximum	Returns the largest of the given arguments or the largest item in an iterable. max(5, 10, 3) results in 10.
min()	Minimum	Returns the smallest of the given arguments or the smallest item in an iterable. min(5, 10, 3) results in 3.
sum()	Summation	Sums the items of an iterable. sum([1, 2, 3, 4, 5]) results in 15.

Table 8-6 Common commands for positioning the cursor within a file

Command	Description	Example
seek()	Moves the file pointer to a specified position	file.seek(10)
tell()	Returns the current file pointer position	file.tell()
truncate()	Resizes the file to a specified size (in bytes)	file.truncate(100)

Table 12-34 Parameters for the RectBivariateSpline class

Param.	Description	Default Value
x, y	1D arrays of coordinates in strictly ascending order	–
z	2D array of data with shape (x.size, y.size)	–
bbox	Sequence of length 4 specifying the boundary of the rectangular approximation domain	[min(x), max(x), min(y), max(y)]
kx, ky	Degrees of the bivariate spline	3
s	Positive smoothing factor for estimation condition	0 (interpolation)

```
from kivy.app import App
from kivy.lang import Builder
# We define the UI and widgets in one string
kv_code = """
FloatLayout:
 Button:
  text: "Fixed Position"
  size_hint: None, None
  size: '100dp', '36dp'
  pos: 100, 100
 Button:
  text: "Relative Position"
  size_hint: None, None
  size: '100dp', '36dp'
  pos_hint: {'right': 1, 'top': 1}
"""
class EngApp(App):
 def build(self):
  return Builder.load_string(kv_code)
if __name__ == '__main__':
 EngApp().run()
```

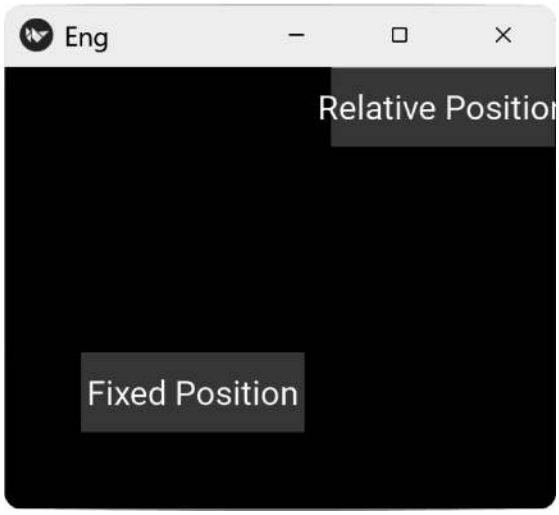


Figure 13-7 Fixed and proportional positioning of buttons using FloatLayout

Table 7-7 Comparison between dynamic and static attributes

Aspect	Dynamic Attributes	Static Attributes (Class Attributes)
Definition	Added to an object at runtime	Defined in the class body
Scope	Specific to the instance they are added to. Not consistent across all instances of the class	Shared across instances of the class
Creation	Created dynamically by assigning a value to an object	Created during class definition

Table 7-4 Types of attributes in Python classes

Attribute Type	Definition	Scope
Class attributes	Defined at the class level; shared by all instances. Useful for constants or data common to all instances	Shared by all instances
Instance attributes	Defined in __init__ or instance methods; unique to each instance	Specific to an instance
Properties	Implemented using @property; allow controlled access via getter/setter/deleter	Specific to an instance
Conventions (private/protected)	Naming conventions (_attribute or __attribute) to indicate intended access levels; not enforced by Python	Class or subclass visibility

Table 12-18 Common applications of Fast Fourier Transform operations

Operation	Application
Fast Fourier Transform (FFT)	Decompose sequences into constituent frequencies.
Frequency spectrum analysis	Examine distribution of power across different frequencies.
Signal filtering	Remove noise, extract specific frequency bands, or perform other filtering.
Modulation and demodulation	Perform amplitude modulation, frequency modulation, and other types.
Image filtering and analysis	Filter images, remove noise, or perform other image analysis.
Time-series data analysis	Examine frequency content of time-series data.
Spectral analysis	Examine distribution of power across different frequencies.
Algorithm implementation	Implement algorithms for convolution, deconvolution, and more.

Table 11-20 Array reshaping, reorganization, and combination operations in NumPy

Operation Type	Command	Description
Reshape	np.reshape(a, new_shape)	Returns a reshaped view if possible
Flatten (copy)	a.flatten()	Returns a 1D copy of the array
Flatten (view if possible)	a.ravel()	Returns a 1D view, when possible
Transpose	np.transpose(a)	Permutes axes of an array
Swap axes	np.swapaxes(a, a1, a2)	Switches two dimensions
Move axis	np.moveaxis(a, src, dst)	Moves an axis to a new position
Expand dimensions	np.expand_dims(a, axis)	Inserts a new axis
Add dimension (None)	a[:, None]	Adds a new axis (same as reshape)
Squeeze	np.squeeze(a)	Removes axes of size 1
Stack arrays	np.stack([a, b])	Stacks arrays along new axis
Vertical stack	np.vstack([a, b])	Stacks row-wise
Horizontal stack	np.hstack([a, b])	Stacks column-wise
Concatenate	np.concatenate([a, b], axis)	Joins arrays along an existing axis
Split	np.split(a, n)	Splits into equal parts
Array repeat	np.repeat(a, n)	Repeats each element n times
Tile	np.tile(a, reps)	Repeats the whole array
Array split	np.array_split(a, n)	Splits into nearly equal parts when an exact division is not possible
Flip	np.flip(a, axis=None)	Reverses element order along one or more axes
Roll	np.roll(a, shift, axis=None)	Circularly shifts elements along an axis
Broadcast to	np.broadcast_to(a, shape)	Returns a read-only broadcasted view with the requested compatible shape

Table 12-23 Return values for the scipy.integrate.quad function

Return Value	Data Type	Description
y	float	The integral of func from a to b
abserr	float	An estimate of the absolute error in the result
infodict	dict	A dictionary containing additional information
message	str	A convergence message
explain	str	Explanation of the codes in infodict['ierlst'] (only for 'cos' or 'sin' weighting and infinite integration limits)

Table 12-6 Parameters and descriptions for the scipy.fft.fft function

Attribute	Type	Description	Default Value
x	array_like	Input array (can be complex).	–
n	int, optional	Length of the transformed axis of the output. If n is smaller than the length of the input, the input is cropped. If larger, the input is padded with zeros. If not given, the input length along the axis is used.	None
axis	int, optional	Axis over which to compute the FFT. If not given, the last axis is used.	–1
norm	{"backward", "ortho", "forward"}, optional	Normalization mode. Default is "backward" (no normalization on forward transforms, scaling by 1/n on ifft). "forward" applies 1/n on forward transform. "ortho" scales both directions by 1/sqrt(n).	"backward"
overwrite_x	bool, optional	If True, the contents of x can be destroyed. Default is False.	False
workers	int, optional	Maximum number of workers for parallel computation. If negative, wraps around from os.cpu_count().	None
plan	object, optional	Reserved for precomputed plans from downstream FFT vendors. Not currently used in SciPy.	None

Table 11-21 Parameters of the np.memmap constructor

Parameter	Description
filename	Name of the file to be memory-mapped
dtype	Data type of the array
mode	Mode to open the file: "r" (read-only), "r+" (read–write existing file), "w+" (create or overwrite), or "c" (copy-on-write)
shape	Shape of the array
offset	Offset into the file where the data starts

Table 3-3 Set-like operations between lists

Operation	Description
Union (|)	Convert lists to sets, perform a union, and convert back to a list.
Intersection (&)	Convert lists to sets, perform an intersection, and convert back to a list.
Difference (-)	Convert lists to sets, perform a difference, and convert back to a list.
Symmetric difference (^)	Convert lists to sets, perform a symmetric difference, and convert back to a list.

Table 12-10 Functions for the Fast Hankel Transform (FHT) and its inverse

Function	Purpose
fht(a)	Fast Hankel Transform
ifht(A)	Inverse Fast Hankel Transform

Table 11-5 Summary of array dimensions, usage purposes, and code examples

Dimension	Purpose	Example
0D (scalar)	Holds a single value	np.array(42)
1D (vector)	List of values	np.array([1, 2, 3, 4])
2D (matrix)	Rows × columns	np.array([[1,2,3],[4,5,6]])
3D (tensor)	Stack of 2D matrices	np.array([[[1,2],[3,4]], [[5,6],[7,8]]])
N-D (higher-D)	Any number of dimensions > 3	np.random.rand(2,3,4,5)

Table 4-3 Basic module import methods

Method	Description	Example
Import the entire module.	Imports the entire module, accessed with the module name as a prefix.	`import math` `print(math.sqrt(16))`

Import specific components.	Imports specific functions, classes, or variables from a module.	`from math import sqrt, pi`
		`print(sqrt(16))`
		`print(pi)`

Import with an alias.	Assigns an alias to the module for easier reference.	`import numpy as np` `print(np.array([1, 2, 3]))`
Import specific components with an alias.	Assigns an alias to specific components of a module.	`from math import sqrt as square_root` `print(square_root(16))`

Import all components.	Imports all components of a module into the current namespace. This is called wildcard.	`from math import *`
		`print(sqrt(16))`
		`print(pi)`

Table 9-7 Preconfigured decimal.BasicContext

***Context*: decimal.BasicContext.**
This context is designed primarily for debugging.
***Precision*: Nine digits.**
***Rounding*: ROUND_HALF_UP.**
***Traps*: Most signals (InvalidOperation, DivisionByZero, Overflow) are enabled as "traps." This means that when one of these conditions occurs, Python will throw a Python exception instead of silently continuing. This is very useful during development to immediately detect potential problems in your calculations.**

Table 3-8 Example of dictionary key–value pairs

Key	Value
part_number	MX-2001
voltage	220
rpm	1500
power_rating_kw	1.5

Table 13-7 Configurable properties of common Kivy widgets

Widget	Property Category	Property	Description
Button	Text properties	text	The text displayed on the button.
		font_size	Font size of the text.
		color	Text color in (r, g, b, a) format.
	Background properties	background_color	Acts as a multiplier for the background texture color.
		background_normal	Image for the button's normal (not pressed) state.
		background_down	Image for the button's pressed state.
		background_disabled_normal	Image for the button's disabled state.
		background_disabled_down	Image for the button's disabled and pressed state.
	Layout properties	border	Border used for the background image, in (bottom, right, top, left) format.
	Behavior properties	state	The current state, either 'normal' or 'down'.
		always_release	If True, on_release fires even if the touch ends outside the widget.
CheckBox	State properties	active	Boolean indicating if the checkbox is checked ('down' state).
		group	If set, the CheckBox becomes a radio button. Only one button in a group can be active.
		allow_no_selection	If True, allows all radio buttons in a group to be deselected.
	Appearance properties	color	Color used for tinting the graphical representation.
Slider	Value properties	min	Minimum value of the slider. Defaults to 0.
		max	Maximum value of the slider. Defaults to 100.
		value	Current value of the slider.

Widget	Property Category	Property	Description
		value_normalized	Value normalized between 0 and 1.
	Appearance properties	orientation	'horizontal' or 'vertical'.
		padding	Padding around the slider, used for graphical representation and interaction.
		value_track	Boolean to show a line indicating the current value.
		value_track_color	Color of the value track line.
	Behavior properties	step	Step size of the slider. A value of 0 allows continuous movement.
		sensitivity	Can be 'all' or 'handle', determining if the whole widget or just the handle is sensitive to touch.
ProgressBar	Value properties	max	Maximum value of the progress bar. Defaults to 100.
		value	Current value of the progress bar.
		value_normalized	Value normalized between 0 and 1.
Label	Text properties	text	The text to display.
		font_size	Font size in pixels.
		color	Text color in (r, g, b, a) format.
		bold, italic, underline, strikethrough	Boolean properties for text style.
		line_height	Spacing between lines of text.
		outline_color	Color of the text outline.
		outline_width	Width of the text outline.
	Layout properties	text_size	Constrains the text to a bounding box, enabling alignment and wrapping.
		halign, valign	Horizontal and vertical alignment of text within the text_size box.
		shorten	If True, shortens the text with an ellipsis (…) to fit.
		padding	Padding around the text.

Widget	Property Category	Property	Description
	Markup properties	markup	If True, allows for Text Markup styling (e.g., [b]bold[/b]).
TextInput	Text properties	text	The editable text of the widget.
		hint_text	Hint text shown when the input is empty.
	Behavior properties	multiline	If False, the 'enter' key validates input instead of adding a new line.
		password	If True, masks the input characters.
		readonly	If True, the content cannot be changed by the user.
		input_filter	Can be 'int', 'float', or a callable to filter characters.
		focus	Boolean indicating if the widget currently has keyboard focus.
		use_bubble	If True, displays a cut/copy/paste bubble on mobile.
		use_handles	If True, displays selection handles on mobile.
		write_tab	If True, the Tab key inserts a tab character instead of changing focus.
		tab_width	The width of a tab character in spaces.
		auto_indent	If True, automatically indents new lines.
Image	Source properties	source	Filename or source of the image.
		texture	The texture object of the image.
	Appearance properties	fit_mode	Determines how the image is resized to fit the widget box ('scale-down', 'fill', 'contain', 'cover').
		color	Tint color for the image in (r, g, b, a) format.
		mipmap	If True, enables OpenGL mipmapping for better rendering at smaller sizes.
		nocache	If True, the image will not be stored in the cache.

Widget	Property Category	Property	Description
	Animation properties	anim_delay	Delay between frames for animated images (e.g., GIFs).
		anim_loop	Number of times to loop an animation; 0 means loop indefinitely.
Switch	State properties	active	Boolean property indicating whether the switch is active (on) or inactive (off).
Popup	Content properties	title	String for the title of the popup.
		title_align	Horizontal alignment of the title.
		title_color	Color of the title text.
		title_font	Font name for the title.
		title_size	Font size for the title.
		content	The main widget to display within the popup.
	Behavior properties	auto_dismiss	If True, the popup is dismissed when clicking outside it.
	Layout properties	size_hint, size	Controls the size of the popup window.
		separator_color, separator_height	Style the separator line between the title and content.
Video	Source properties	source	Filename or URI of the video to play.
	Playback control	state	The current state of playback ('play', 'pause', 'stop').
		play	Deprecated since Kivy 1.4.0; use the state property instead.
		volume	The volume of the audio, from 0 (mute) to 1 (full).
		position	The current playback position in seconds.
		duration	The total duration of the video in seconds (read-only).
		eos	Boolean indicating whether playback has reached the end of the stream.

Widget	Property Category	Property	Description
	Appearance	preview	An image to show before the video starts playing.
Spinner	Data properties	text	The currently selected value, displayed on the spinner.
		values	A list of strings that populates the dropdown options.
	Appearance	sync_height	If True, the dropdown's height will sync with the available height.
		option_cls	The class used to display the options in the dropdown (defaults to a specialized Button).
	Behavior	is_open	Boolean property that is True when the dropdown list is open.

Table 11-19 Linear algebra operations and functions in NumPy

Operation Type	Command	Description
Matrix multiplication	a @ b	Performs matrix multiplication
Matrix multiplication (explicit)	np.matmul(a, b)	Equivalent to a @ b
Dot product	np.dot(a, b)	Inner product or matrix product depending on shapes
Inner product	np.inner(a, b)	Sum of elementwise products
Outer product	np.outer(a, b)	Outer product of two vectors
Vector dot product	np.vdot(a, b)	Dot product treating vectors as 1D
Norm	np.linalg.norm(a)	Vector or matrix norm
Determinant	np.linalg.det(A)	Matrix determinant
Inverse matrix	np.linalg.inv(A)	Computes the inverse of a matrix
Solve Ax = b	np.linalg.solve(A, b)	Solves a linear system.
Eigenvalues/vectors	np.linalg.eig(A)	Computes eigenvalues and eigenvectors
Singular Value Decomposition	np.linalg.svd(A)	Computes U, S, V^T decomposition
Rank	np.linalg.matrix_rank(A)	Returns matrix rank
Trace	np.trace(A)	Sum of diagonal elements
QR decomposition	np.linalg.qr(A)	Computes Q and R matrices
Cholesky decomposition	np.linalg.cholesky(A)	Lower-triangular Cholesky factorization

	Operation	Command
1	Creating a Path object	Path()
2	Joining paths	/ operator
3	Getting the parent directory	.parent
4	Getting path components	.name, .stem, .suffix
5	Checking if a path exists	.exists()
6	Checking if a path is a file/directory	.is_file(), .is_dir()
7	Creating directories	.mkdir()
8	Reading and writing files	.read_text(), .write_text()
9	Renaming/moving files	.rename()
10	Deleting files and directories	.unlink(), .rmdir()
11	Getting file size	.stat().st_size
12	Getting the last modified time	.stat().st_mtime
13	Finding files with glob	.glob()
14	Recursive glob	.rglob()
15	Getting an absolute path	.resolve()
16	Getting a relative path	.relative_to()

Table 13-8 Events and methods of common Kivy widgets

Widget	Type	Name	Description
Button	Event	on_press	Fired when the button is pressed.
		on_release	Fired when the button is released.
	Method	trigger_action(duration=0.1)	Programmatically triggers the on_press and on_release events.
Label	Event	on_ref_press	Fired when the user clicks text marked with a [ref] tag while markup is enabled.
	Method	texture_update()	Forces texture recreation with the current Label properties.
TextInput	Event	on_text_validate	Fired when the user presses "Enter" in single-line mode.
		on_double_tap/on_triple_tap	Fired on a double or triple tap, selecting the word or line by default.
	Method	insert_text(substring)	Inserts new text at the current cursor position; can be overridden to filter input.
		copy(), cut(), paste()	Perform clipboard operations on the selected text.
		select_all(), select_text(start, end)	Programmatically select text.
		delete_selection()	Deletes the currently selected text.
Image	Method	reload()	Reloads the image from the disk.
		remove_from_cache()	Removes the image from Kivy's cache, forcing a reload on next use.
Popup	Event	on_open	Fired when the Popup is opened.
		on_dismiss	Fired when the Popup is closed. Returning True from the callback cancels the dismissal.
	Method	open()	Displays the popup.
		dismiss()	Closes the popup.
Carousel	Method	load_next(), load_previous()	Animate to the next or previous slide.
		load_slide(slide)	Animates to a specific slide widget.

Widget	Type	Name	Description
Screen	Event	on_pre_enter, on_enter	Fired before and after the screen is shown.
		on_pre_leave, on_leave	Fired before and after the screen is hidden.
ScreenManager	Method	switch_to(screen, **options)	Adds a new screen and switches to it, with optional transition settings. For a screen already added to the manager, set the current property instead.
		get_screen(name)	Returns the screen widget associated with a given name.
AccordionItem	Property	collapse	Boolean indicating whether the accordion item is collapsed; bind to this property to detect expansion or collapse.
		collapse_alpha	Value from 0 (selected/open) to 1 (collapsed), mainly used during animation.
DropDown	Event	on_select	Fired when an item in the list is selected.
	Event	on_dismiss	Fired when the dropdown is dismissed.
	Method	open(widget)	Opens the dropdown and attaches it to the specified widget.
		dismiss()	Closes the dropdown.
		select(data)	Triggers the on_select event and passes data as the selected value; data may be any object.
Widget	Event	on_touch_down	Fired for a new touch event.
		on_touch_move	Fired when an existing touch moves.
		on_touch_up	Fired when an existing touch is released.
	Method	collide_point(x, y)	Checks if a point (in parent's coordinates) is within the widget's bounding box.
		collide_widget(wid)	Checks if another widget's bounding box collides with this one.
		add_widget(widget)	Adds a new widget as a child.

Widget	Type	Name	Description
		remove_widget(widget)	Removes a child widget.
		clear_widgets()	Removes all children widgets.

Table 7-8 Types of methods in Python classes

Method Type	First Parameter	Decorator	Use Case
Instance method	self	None	Most common method type, used for instance-specific operations
Class method	cls	@classmethod	Used for alternative constructors or class-level operations
Static method	None	@staticmethod	Utility functions that don't need instance or class access
Abstract method	self	@abstractmethod	Declared in abstract base classes, must be implemented by subclasses
Magic/dunder method	Varies	None	Implements/customizes special behaviors (e.g., __init__, __str__)
Property method	self	@property (getter) @<attribute>.setter (setter)	Create read-only or read–write attributes with getter/setter methods

Table 12-58 Parameters for the scipy.signal.periodogram function

Parameter	Description	Values/Default
x	Time series of measurement values	array_like
fs	Sampling frequency	float, optional (default: 1.0)
window	Desired window to use	str, tuple, or array_like, optional (default: 'boxcar')
nfft	Length of the FFT used	int, optional (default: None)
detrend	Specifies how to detrend each segment	str, function, or False, optional (default: 'constant')
return_onesided	Return a one-sided spectrum for real data	bool, optional (default: True)
scaling	Selects between computing power spectral density or spectrum	{'density', 'spectrum'}, optional (default: 'density')
axis	Axis along which the periodogram is computed	int, optional (default: -1)

Table 12-16 Parameters for the scipy.fft.set_backend context manager

Parameter	Data Type	Description
backend	object, 'scipy'	The backend to use. Can be a string or an object implementing the uarray protocol
coerce	bool	Whether to allow expensive conversions for the x parameter
only	bool	If True, raises BackendNotImplemented error if backend returns NotImplemented, ignoring lower-priority backends

Table 6-9 An iteration trace demonstrating how the continue statement skips the current iteration in a while loop

Iteration	Value of "i"	Condition ("i % 2 == 0")	Action
1	1	False	Print 1, and increment i to 2.
2	2	True	Skip (continue), and increment i to 3.
3	3	False	Print 3, and increment i to 4.
4	4	True	Skip (continue), and increment i to 5.
5	5	False	Print 5, and increment i to 6.
–	–	–	The loop ends.

Table 8-1 A summary of common file types, their descriptions, and the primary Python libraries or frameworks used to handle them

File Type	Description	Library/Framework	File Extensions
Text files	Reading, writing, and processing plain-text files	open(), io	.txt
Spreadsheets	Manipulating Excel files	pandas, openpyxl, xlrd	.xls, .xlsx, .ods
JSON files	Reading and writing JSON data	json	.json
PDF files	Extracting, merging, splitting, or creating PDFs	PyPDF2, pdfplumber, ReportLab	.pdf
CSV files	Reading and writing CSV files	open(), csv, pandas	.csv
XML (eXtensible Markup Language) files	Parsing and generating XML data	xml.etree.ElementTree, lxml	.xml
Images	Processing image files (PNG, JPEG)	Pillow, OpenCV	.png, .jpg, .jpeg, .bmp, .gif
Audio files	Reading and manipulating audio files	pydub, wave	.mp3, .wav, .flac, .aac
Video files	Processing video files	moviepy, OpenCV	.mp4, .avi, .mkv, .mov
Databases	Interacting with database files or external databases	sqlite3, SQLAlchemy, pymysql	.db, .sqlite, .sql
ZIP/compressed files	Extracting and creating compressed files	zipfile, tarfile	.zip, .tar, .gz, .7z
Markdown files	Reading and converting Markdown files	markdown, mistune	.md
YAML files	Parsing and writing YAML files	PyYAML	.yaml, .yml

File Type	Description	Library/Framework	File Extensions
HTML files	Parsing and scraping HTML files	Beautiful Soup, lxml	.html, .htm
Log files	Analyzing and processing log files	loguru, re	.log
Binary files	Reading and writing binary data	struct, open() (binary mode)	.bin, .dat
HDF5 files	Working with hierarchical data	h5py, pytables	.h5, .hdf5
GIS files	Processing geospatial data	geopandas, shapefile	.shp, .geojson, .kml
Parquet files	Handling columnar data formats	pandas, pyarrow	.parquet
Pickle files	Serializing and deserializing Python objects	pickle	.pkl
INI files	Reading and writing configuration files	configparser	.ini
DOCX files	Creating and editing Word documents	python-docx	.docx
Email files	Parsing and sending emails	email, extract-msg, smtplib	.eml, .msg
MAT files	Working with MATLAB .mat files	scipy.io	.mat
NumPy files	Handling .npy files for numerical data	numpy	.npy, .npz
2D drawings formats	Handling 2D CAD formats	ezdxf, pyautocad; ODA Drawings SDK (commercial C++ SDK with Python support)	.dxf, .dwg
3D drawings formats	Handling 3D CAD formats	cadquery, trimesh, pyvista	.stl, .obj, .step, .3ds

Table 4-4 Additional module import methods

Method	Description	Example
Import submodules explicitly.	Imports a specific submodule of a module	`import os.path` `print(os.path.basename('/path/to/file'))`
Import submodules using "from".	Imports specific submodules using the "from" keyword	`from os import path` `print(path.basename('/path/to/file'))`

Dynamic imports using "importlib".	Dynamically imports a module at runtime using "importlib"	`import importlib`
		`math = importlib.import_module('math')`
		`print(math.sqrt(16))`

Relative imports.	Imports modules relative to the current module's location within a package	`from .. import module_a` `from .submodule import function_name`
Absolute imports.	Imports modules using the full path from the project's root or top-level package	`from my_package import module_a` `from my_package.sub_package import module_b`
Import built-in modules.	Imports modules from Python's standard library	`import os` `print(os.name)`

Import from ZIP files.	Imports modules directly from ZIP files	`import sys`
		`sys.path.append('path/to/your.zip')`
		`import your_module`

Import with "sys.path	Modifies "sys.path" to include custom directories for imports	`import sys`
		`sys.path.append('/path/to/custom/modules')`
		`import custom_module`

Table 12-17 Summary of functions used for managing FFT computation backends

Function	Description
set_backend(backend)	Context manager to set the backend within a fixed scope
skip_backend(backend)	Context manager to skip a backend within a fixed scope
set_global_backend(backend)	Sets the global FFT backend
register_backend(backend)	Registers a backend for permanent use

Table 12-61 Return values for the scipy.signal.tf2zpk function

Return Value	Description	Type
z	Zeros of the transfer function	ndarray
p	Poles of the transfer function	ndarray
k	System gain	float

Table 12-36 Summary of fundamental functions in the Basics group of scipy.linalg

Function	Description
inv()	Compute the inverse of a matrix.
solve()	Solve a system of linear equations.
solve_banded()	Solve a system of linear equations with a banded matrix.
solveh_banded()	Solve a system of linear equations with a Hermitian banded matrix.
solve_circulant()	Solve a system of linear equations with a circulant matrix.
solve_triangular()	Solve a system of linear equations with a triangular matrix.
solve_toeplitz()	Solve a Toeplitz system using Levinson Recursion.
matmul_toeplitz()	Efficient Toeplitz Matrix–Matrix Multiplication using FFT.
det()	Compute the determinant of a matrix.
norm()	Compute the matrix or vector norm.
lstsq()	Compute the least-squares solution to a system of linear equations.
pinv()	Compute the Moore–Penrose pseudo-inverse of a matrix.
pinvh()	Compute the Moore–Penrose pseudo-inverse of a Hermitian matrix.
kron()	Compute the Kronecker product of two matrices
khatri_rao()	Compute the Khatri–Rao product of two matrices.
orthogonal_procrustes()	Compute the matrix solution of the orthogonal Procrustes problem.
matrix_balance()	Compute a diagonal similarity transformation for row/column balancing.
subspace_angles()	Compute the subspace angles between two matrices.
bandwidth()	Return the lower and upper bandwidth of a 2D numeric array.
issymmetric()	Check if a square 2D array is symmetric.
ishermitian()	Check if a square 2D array is Hermitian.

Table 1-3 List of essential folders used by Python

Folder Name	Primary Content	Main Purpose	Typical Use/Scenario
src/, app/, or project/	.py source code files	Core application or library logic	All project types
tests/	test_*.py scripts	Verifies correctness and prevents regressions	All project types
docs/	Markdown, reStructuredText, HTML (HyperText Markup Language), Sphinx config	User guides, API references, developer docs	All project types
.venv/ or venv/	Virtual environment files	Isolates project dependencies	All project types
data/	CSV (comma-separated values), JSON, raw/processed datasets	Stores datasets used or generated by the project	Data science, ML, analytics
scripts/ or bin/	Standalone .py or .sh files	Helper scripts for automation, setup, deployment	All project types, DevOps, automation
.github/	YAML workflows, templates	Configures GitHub Actions CI/CD pipelines	All project types with GitHub CI/CD
notebooks/	Jupyter .ipynb files	Data exploration, prototyping, reporting	Data science, ML, research, experimentation
examples/	Example .py files or sample projects	Demonstrates usage of the package/library	All project types
configs/	.yaml, .toml, .json	Application configs (env settings, pipelines)	All project types, web apps, automation
migrations/	Migration scripts (SQL, Alembic, Django)	Tracks and applies database schema changes	Web apps, database-driven apps
templates/	.html, .jinja, etc.	Stores frontend templates for web frameworks	Web applications
static/	CSS, JS, images	Static files for web apps	Web applications
logs/	.log files	Runtime logs for debugging/monitoring	All project types, especially servers and automation

Folder Name	Primary Content	Main Purpose	Typical Use/Scenario
build/ or dist/	Wheels, build artifacts	Stores compiled distribution packages	Packaging, library distribution, production apps
tmp/ or var/	Temp files, caches	Temporary runtime data	All project types
assets/	Images, fonts, audio, video	Project-specific media and resources	GUI apps, games, web apps
fixtures/	JSON, YAML, or CSV	Predefined data for testing or seeding databases	Web apps, database-heavy apps, testing
locales/	Translation files (.po, .mo)	Internationalization and localization	Web apps, GUI apps, internationalization
models/	Python classes, ML models, database models	Defines domain logic, database tables, or machine learning models	ML projects, web apps, data-driven apps

Table 2-2 Mathematical and utility operations

Operator/Function	Description	Example	Result
Arithmetic Operations (x=42, y=7)			
+	Addition	x + y	49
-	Subtraction	x - y	35
*	Multiplication	x * y	294
//	Floor division (returns the integer part)	x // y	6
%	Modulus (returns the remainder)	x % y	0
**	Exponentiation (to the power of)	x ** 2	1764
Bitwise Operations (x=42, y=7)			
&	Bitwise AND	x & y	2
|	Bitwise OR	x | y	47
^	Bitwise XOR (exclusive OR)	x ^ y	45
<<	Left shift	x << 1	84
>>	Right shift	x >> 1	21
Conversion Methods (x=42, f=3.14)			
.to_bytes()	Converts an integer to a byte object	x.to_bytes(2, 'big')	b'\x00*'
int.from_bytes()	Converts a byte object to an integer	int.from_bytes(b'\x00*', 'big')	42
.as_integer_ratio()	Represents a float as a fraction	f.as_integer_ratio()	(7070651414971679, 2251799813685248)
Representation Methods (f=3.14159)			
.hex()	Converts a float to its hexadecimal string representation	f.hex()	0x1.921f9f01b866ep+1'
float.fromhex()	Converts a hexadecimal string back to a float	float.fromhex ('0x1.921f9f01b866ep+1')	3.14159

Operator/Function	Description	Example	Result
Utility Methods and Functions (x=42, f=3.14159)			
abs()	Returns the absolute value of a number	abs(-x)	42
.bit_length()	Returns the number of bits to represent an integer in binary	x.bit_length()	6
.is_integer()	Checks if a float has a whole number value	f.is_integer()	FALSE
Built-In Functions (x=42, y=7, f=3.14159)			
round()	Rounds a float to a specified number of decimal places	round(f, 2)	3.14
divmod()	Returns a tuple with the quotient and remainder	divmod(x, y)	(6, 0)
pow(x, y)	Raises x to the power of y	pow(x, y)	515377520732
pow(x, y, z)	Calculates (x ** y) % z efficiently	pow(3, 4, 5)	1

Table 1-1 Small example of community-defined PEPs

PEP	Purpose and Impact	Date
PEP 8	Defines a coding style for Python projects	**2001**
PEP 257	Standardizes docstring conventions	**2001**
PEP 484	Adds optional static typing to Python	**2014**
PEP 703	Makes the GIL optional in CPython builds to support free-threaded execution	**2023**
PEP 711	Proposes the draft PyBI format for distributing prebuilt Python interpreters	2023

Table 3-1 Methods and operations for Python lists.Source: Adapted from the Python Software Foundation, Python Tutorial, section 5.1, "More on Lists" (*https://docs.python.org/3/tutorial/datastructures.html*)

Method	Description	Example Usage/Operator
len()	Returns the number of elements in the list.	len(user_list)
append(x)	Adds an item to the end of the list. Equivalent to "a[len(a):] = [x]".	user_list.append(10)
extend(iterable)	Extends the list by appending all items from the given iterable. Equivalent to "a[len(a):] = iterable".	user_list.extend([1, 2, 3])
insert(i, x)	Inserts an item at the given position "i". If "i" is 0, inserts at the front. If "i" is "len(a)", behaves like "append".	user_list.insert(1, "item")
remove(x)	Removes the first occurrence of the item with value "x". Raises "ValueError" if "x" is not in the list.	user_list.remove(5)
pop([i])	Removes and returns the item at the given index "i". If no index is specified, removes and returns the last item. Raises "IndexError" if the index is out of range.	user_list.pop()/ user_list.pop(2)
clear()	Removes all items from the list. Equivalent to "del a[:]".	user_list.clear()
index(x, start, end)	Returns the index of the first occurrence of "x" in the list. Optional "start" and "end" arguments limit the search to a subsequence. Raises "ValueError" if "x" is not found.	user_list.index(3)/ user_list.index(3, 1, 5)
count(x)	Returns the number of times "x" appears in the list.	user_list.count(4)
sort(*, key=None, reverse=False)	Sorts the items of the list in place. Optional "key" and "reverse" arguments customize the sorting behavior.	user_list.sort()/ user_list.sort(reverse=True)
reverse()	Reverses the elements of the list in place.	user_list.reverse()
copy()	Returns a shallow copy of the list. Equivalent to "a[:]".	copied_list = user_list.copy()
max(), min()	Finds the smallest or largest element in a list.	max(user_list) min(user_list)
sum()	Calculates the sum of all numeric elements in a list.	sum(user_list)

Method	Description	Example Usage/Operator
Membership	Checks if an element exists within the list, returning True or False.	in, not in
Concatenation	Combines two lists to create a new, single list.	+
Repetition	Creates a new list by repeating its elements a specified number of times.	*
Slicing	Creates a new list containing a range of elements.	[:]
Equality	Compares two lists to check if they have identical elements in the same order.	==, !=

Table 11-17 Trigonometric, hyperbolic, exponential, and logarithmic functions in NumPy

Operation Type	Command	Description
Sine	np.sin(a)	Computes the sine of each element
Cosine	np.cos(a)	Computes the cosine of each element
Tangent	np.tan(a)	Computes the tangent of each element
Inverse sine	np.arcsin(a)	Computes arcsin elementwise
Inverse cosine	np.arccos(a)	Computes arccos elementwise
Inverse tangent	np.arctan(a)	Computes arctan elementwise
Hyperbolic sine	np.sinh(a)	Hyperbolic sine elementwise
Hyperbolic cosine	np.cosh(a)	Hyperbolic cosine elementwise
Hyperbolic tangent	np.tanh(a)	Hyperbolic tangent elementwise
Exponential	np.exp(a)	Computes e**a elementwise
exp(x) - 1 (stable)	np.expm1(a)	Stable version of exp(a)-1 for small a
Natural logarithm	np.log(a)	Computes ln(a)
log10	np.log10(a)	Log base 10
log2	np.log2(a)	Log base 2
log(1+x) (stable)	np.log1p(a)	Stable version of log(1+a)
Square root	np.sqrt(a)	Computes $\sqrt{a}$
Cube root	np.cbrt(a)	Computes a^(1/3)
Square (power 2)	np.square(a)	Elementwise a^2
General power	np.power(a, b)	Raises each element of *a* to power *b*

Table 12-48 Overview of topics and function families within the scipy.optimize subpackage

Topics	Description
Optimization	
Scalar function optimization	Minimization of functions with one variable.
Local (multivariate) optimization	Minimization of functions with multiple variables.
Global optimization	Finding the global minimum of a function.
Least Squares and Curve Fitting	
Nonlinear least squares	Solve nonlinear least-squares problems.
Linear least squares	Solve linear least-squares problems.
Curve fitting	Fit a function to data using nonlinear least squares.
Root Finding	
Scalar functions	Find roots of scalar functions.
Multidimensional	Find roots of vector functions.
Elementwise scalar optimization	Elementwise root finding and minimization.
Linear Programming/MILP	
Mixed-integer linear programming	Solve MILP problems.
Linear programming	Minimize a linear objective function subject to constraints.
Assignment Problems	
Linear sum assignment	Solve the linear sum assignment problem.
Quadratic assignment	Approximate solution to the quadratic assignment problem.
Utilities	
Finite-difference approximation	Approximate derivatives using finite differences.
Line search	Find alpha that satisfies strong Wolfe conditions.
Hessian approximation	Approximate the Hessian matrix.
Benchmark problems	Test functions for evaluating optimization algorithms.

Control Area	Main Focus	Design Methods	Python Tools
Classical control	Stability and performance for simple systems	• PID tuning (Ziegler–Nichols and others) • Lead–lag compensator design • Root locus shaping	• control • scipy.signal • NumPy/Matplotlib
Modern control	Full control of the system's internal states	• Pole placement • State feedback Controller design • Observer/estimator design	• control • NumPy • SciPy
Nonlinear control	Handling the actual (nonlinear) dynamics of systems	• Sliding mode control (SMC) • Backstepping design • Feedback linearization	• SciPy.integrate.solve_ivp • SymPy • NumPy • control: NonlinearIOSystem
Predictive control (MPC)	Future optimization while respecting constraints	• Model-based predictive control formulation • Constraint and cost function definition	• GEKKO • CasADi • Pyomo • CVXPY
Robust control (LMI)	Guaranteeing stability and performance with model uncertainties	• H∞ loop shaping • LMI-based controller synthesis • μ-Synthesis	• control • CVXPY/PICOS • NumPy
Optimal control	Minimizing a cost function (error and energy usage)	• LQR controller design • Solving Hamilton–Jacobi–Bellman equations • Calculus of variations	• control • GEKKO/CasADi • SciPy.optimize • CVXPY

Control Area	Main Focus	Design Methods	Python Tools
Intelligent control	Handling complex systems or those without a precise mathematical model	• Fuzzy rule-based design • Neural network training for control • Reinforcement learning policy optimization	• scikit-fuzzy • PyTorch/TensorFlow • Stable-Baselines3 • Gymnasium (OpenAI Gym)

Table 11-7 Key ndarray properties and structural memory attributes

Property	Description
ndarray.shape	Dimensions of the array
ndarray.ndim	Number of dimensions
ndarray.size	Total number of elements
ndarray.dtype	Data type of elements
ndarray.itemsize	Size (bytes) of each element
ndarray.nbytes	Total memory used by the array
ndarray.strides	Step in bytes to move along dimensions
ndarray.flat	1D iterator over elements
ndarray.T	Transposed array
ndarray.real	Real part
ndarray.imag	Imaginary part
ndarray.flags	Memory layout information

Table 10-7 Function prototypes and key parameters for Matplotlib scales

Scale Name	Function Prototype	Key Parameters and Explanation
Linear	ax.set_xscale('linear')	Default scale with no special parameters
Log	ax.set_xscale('log', base=10, nonpositive='clip')	base: Logarithm base nonpositive: Handles zero/negative values ('clip' or 'mask')
Symlog	ax.set_xscale('symlog', linthresh=1.0, base=10, linscale=1.0)	linthresh: Linear range around zero base: Logarithm base linscale: Adjusts linear section space
Logit	ax.set_xscale('logit', nonpositive='clip')	nonpositive: Handles values outside (0, 1) range
Asinh	ax.set_xscale('asinh', linear_width=1.0, base=10)	linear_width: Effective width of the quasi-linear region around zero base: Number base used for tick-location rounding
Function	ax.set_xscale('function', functions= (forward_func, inverse_func))	Applying a fully custom, non-standard scale

Table 12-26 Return values for the scipy.integrate.dblquad function

Return Value	Data Type	Description
y	float	The computed integral result
abserr	float	Estimate of the absolute error

Table 11-10 Elementwise arithmetic operations, operators, and corresponding ufuncs

Operation Type	Command	Description
Addition	a + b	Elementwise addition (broadcasting supported)
	np.add(a, b)	Ufunc version of addition
Subtraction	a - b	Elementwise subtraction
	np.subtract(a, b)	Ufunc version of subtraction
Multiplication	a * b	Elementwise multiplication
	np.multiply(a, b)	Ufunc version of multiplication
Division	a / b	Elementwise true division (float output)
	np.divide(a, b)	Ufunc version of division
Floor division	a // b	Quotient rounded down to nearest integer
	np.floor_divide(a, b)	Ufunc version of floor division
Remainder (Modulo)	a % b	Elementwise remainder
	np.mod(a, b) or np.remainder(a, b)	Ufunc version of modulo (handles negatives)
Power/exponentiation	a ** b	Raises each element of a to the power b
	np.power(a, b)	Ufunc version of exponentiation

Table 12-47 Summary of special matrix functions and low-level routines in scipy.linalg

Function	Description
Sketches and Random Projections	
clarkson_woodruff_transform	Applies the Clarkson–Woodruff transform/sketch.
Special Matrices	
block_diag	Create block diagonal matrix.
circulant	Construct circulant matrix.
companion	Create companion matrix.
convolution_matrix	Construct convolution matrix.
dft	Discrete Fourier Transform matrix.
fiedler	Symmetric Fiedler matrix.
fiedler_companion	Fiedler companion matrix.
hadamard	Construct Hadamard matrix.
hankel	Construct Hankel matrix.
helmert	Create Helmert matrix.
hilbert	Create Hilbert matrix.
invhilbert	Inverse Hilbert matrix.
leslie	Create Leslie matrix.
pascal	Pascal matrix.
invpascal	Inverse Pascal matrix.
toeplitz	Construct Toeplitz matrix.
Low-Level Routines	
get_blas_funcs	Return BLAS function objects.
get_lapack_funcs	Return LAPACK function objects.
find_best_blas_type	Find best-matching BLAS/LAPACK type.

Table 11-18 Comparison, logical, sorting, searching, and set operations in NumPy

Operation Type	Command	Description
Elementwise comparison	a == b, a != b, a > b, a < b	Compares arrays elementwise and returns boolean arrays
Greater than	np.greater(a, b)	Equivalent to a > b
Less than	np.less(a, b)	Equivalent to a < b
Equal	np.equal(a, b)	Tests equality elementwise
Not equal	np.not_equal(a, b)	Tests inequality elementwise
Logical AND	np.logical_and(a, b)	Boolean AND elementwise
Logical OR	np.logical_or(a, b)	Boolean OR elementwise
Logical NOT	np.logical_not(a)	Boolean negation
Logical XOR	np.logical_xor(a, b)	Exclusive OR
Where (conditional)	np.where(cond, x, y)	Selects values based on a boolean condition
All true?	np.all(a)	Returns True if all elements are true/non-zero
Any true?	np.any(a)	Returns True if any element is true/non-zero
Sort	np.sort(a)	Returns sorted copy of array
Argsort	np.argsort(a)	Returns indexes that would sort the array
Partition	np.partition(a, k)	Partially sorts array (useful for top-k)
Searchsorted	np.searchsorted(a, v)	Finds index where value should be inserted
Unique	np.unique(a)	Returns sorted unique values
Set intersection	np.intersect1d(a, b)	Elements common to both arrays
Set union	np.union1d(a, b)	Unique elements from both arrays
Set difference	np.setdiff1d(a, b)	Elements in a not in b
Membership test	np.isin(a, b)	Boolean mask for elements of a in b

Table 12-62 Parameters for the scipy.signal.tf2ss function

Parameter	Description	Type
num	Numerator polynomial coefficients	array_like
den	Denominator polynomial coefficients	array_like

Table 9-2 Main built-in modules for mathematical operations

Module	Purpose	Capabilities
math	Common mathematical functions and constants	Number theory, floating-point manipulation, powers and logarithms, combinatorics, trigonometric and hyperbolic functions, special functions, and constants
cmath	Mathematical functions for complex numbers	Complex-aware versions of math functions, conversions to/from polar coordinates, power/logarithmic functions, trigonometric functions, hyperbolic functions, classification functions, constants (pi, e, tau, inf, nan, infj)
decimal	Exact decimal number representation and arithmetic	Arbitrary-precision arithmetic, useful for financial calculations or precise decimal arithmetic
fractions	Rational number arithmetic	Exact arithmetic with fractions
statistics	Mathematical statistics of numerical data	Averages (mean, median, mode), measures of spread (variance, standard deviation), other statistical calculations
random	Pseudo-random number generation	Random number generation for simulations, modeling, and statistical analysis

Table 12-20 Overview of common digital signal processing (DSP) operations and their corresponding Python libraries

Category	Operations	Libraries/Functions
Arithmetic operations	Addition, subtraction, multiplication, division, square root, exponentiation	numpy, math
Signal processing operations	Convolution	numpy.convolve, scipy.signal.convolve
	Correlation	numpy.correlate, scipy.signal.correlate
	Discrete Fourier Transform (DFT)	numpy.fft.fft, scipy.fft.fft
	Fast Fourier Transform (FFT)	numpy.fft.fft, scipy.fft.fft
	Inverse FFT (IFFT)	numpy.fft.ifft, scipy.fft.ifft
	Z-Transform	Needs custom implementation
	Laplace Transform	sympy.laplace_transform (symbolic)
	Wavelet Transform	pywavelets (PyWavelets package)
Filtering operations	FIR, IIR filtering	scipy.signal.firwin, scipy.signal.lfilter, scipy.signal.butter
	Low-pass, high-pass, band-pass filters	scipy.signal.butter, scipy.signal.filtfilt
	Adaptive filtering	padasip (for LMS, RLS algorithms)
	Decimation and interpolation	scipy.signal.decimate, scipy.signal.resample
Statistical and probabilistic operations	Mean, variance, standard deviation	numpy.mean, numpy.var, numpy.std
	Auto-correlation, cross-correlation	numpy.correlate, scipy.signal.correlate
	Power spectral density (PSD)	scipy.signal.welch, matplotlib.mlab.psd
	Entropy calculations	scipy.stats.entropy
Modulation and demodulation	AM, FM, PM	scipy.signal.hilbert

Category	Operations	Libraries/Functions
	QAM, PSK, FSK	scipy.signal or commpy (Communications package)
Sampling and reconstruction	Sampling	numpy.linspace, numpy.arange
	Quantization	numpy.round, custom quantization functions
	Interpolation	scipy.interpolate.interp1d, numpy.interp
	Decimation	scipy.signal.decimate
Control and optimization	PID (Proportional–Integral–Derivative) control	control package, scipy.optimize
	Kalman filtering	pykalman or filterpy
	Adaptive equalization	padasip, commpy
Logical and bitwise operations	AND, OR, XOR, NOT	numpy.bitwise_and, numpy.bitwise_or
	Bit shifting (left/right)	numpy.left_shift, numpy.right_shift
	Rounding, Truncation	numpy.round, numpy.trunc
Machine learning and AI in DSP	Feature extraction (MFCC, STFT, spectrogram)	librosa, scipy.signal.stft
	Neural network–based filtering	tensorflow, pytorch
	Speech and image recognition	speechbrain, torchvision

Table 12-37 Parameters for the scipy.linalg.inv function

Param.	Type	Description	Default
a	array_like	Square matrix to be inverted.	–
overwrite_a (optional)	bool	Discard data in a (may improve performance).	False
check_finite (optional)	bool	Check if input matrix contains only finite numbers.	Implied True

Table 12-57 Parameters for the scipy.signal.butter function

Param.	Description	Values/Default
N	Filter order	integer
Wn	Critical frequency	scalar (lowpass/highpass) or length-2 sequence (bandpass/bandstop)
btype	Filter type	'lowpass', 'highpass', 'bandpass', 'bandstop' (default: 'lowpass')
analog	Filter type (analog/digital)	True (analog), False (digital) (default: False)
output	Output format	'ba', 'zpk', 'sos' (default: 'ba')
fs	Sampling frequency	float (optional, default: None)

Table 7-5 Comparison between class and instance attributes

Aspect	Class Attributes	Instance Attributes
Definition	Defined at the class level.	Defined in the constructor method or instance methods.
Scope	Common to all instances of the class.	Specific to each instance of the class.
Access	Accessed using the class name or instances.	Accessed only through the instance.
Modification	Alterations impact all instances.	Alterations only affect the particular instance.
Application	For data common to all instances.	For data unique to each instance.

Table 8-12 A summary of commands used for common .docx file operations

	Operation	Used Command
1	Read a .docx file.	Document(); for paragraphs in doc.paragraphs
2	Write in a .docx file.	Document(); doc.add_heading(), doc.add_paragraph(), doc.save()
3	Append to a .docx file.	Document(); doc.add_paragraph(), doc.save()
4	Get font format and size.	Document(); for loop
5	Translate text.	Document(); await translator.translate(), translated_doc.add_paragraph()
6	Read a specific paragraph from a .docx file.	Document(); doc.paragraphs[], .text
7	Replace a specific paragraph in a .docx file.	Document(); doc.paragraphs[], .clear(), .add_run(), .save()
8	Append text to a specific paragraph.	Document(); doc.paragraphs[], .add_run(), .save()
9	Count paragraphs, tables, and images.	Document(); len(), sum()
10	Extract all paragraphs.	Document(); enumerate()
11	Identify heading paragraphs.	Document(); startswith()

Table 8-10 An index of common operations for .csv files

1	**Read an entire CSV file as a string.**	**8**	**Convert a CSV file to a list.**
2	Read all lines into a list.	9	Append a new row.
3	Read the first line.	10	Modify a specific row.
4	Read a specific line.	11	Insert a row at a specific position.
5	Read a specific cell.	12	Delete a specific row.
6	Iterate through rows.		
7	Read a specific row.		

Table 8-4 Common commands for reading files

Command	Description	Example
read()	Reads the entire file	file.read()
readline()	Reads a single line	file.readline()
readlines()	Reads all lines into a list	file.readlines()
readable()	Checks if the file is readable	file.readable()
read(n)	Reads n characters from the file	file.read(10)
__iter__()	Iterates over lines in the file (used in loops)	for line in file:

Table 12-41 Functions for solving standard and specialized eigenvalue problems in scipy.linalg

Function	Description
Eigenvalue Solvers	
eig()	Eigenvalue decomposition general
eigvals()	Compute eigenvalues general
eigh()	Eigenvalue decomposition Hermitian/symmetric
eigvalsh()	Compute eigenvalues Hermitian/symmetric
Specialized Solvers	
eig_banded()	Eigenvalue solver for banded matrices
eigvals_banded()	Compute eigenvalues for banded matrices
eigh_tridiagonal()	Eigenvalue solver for tridiagonal matrices
eigvalsh_tridiagonal()	Compute eigenvalues for tridiagonal matrices

Table 12-39 Supported matrix structures for the assume_a parameter in scipy.linalg.solve

diagonal	**'diagonal'**
tridiagonal	'tridiagonal'
banded	'banded'
upper triangular	'upper triangular'
lower triangular	'lower triangular'
symmetric	'symmetric' (or 'sym')
hermitian	'hermitian' (or 'her')
positive definite	'positive definite' (or 'pos')
general	'general' (or 'gen')

Table 11-1 Overview of main operation categories in NumPy

Category	Types of Operations
Arithmetic operations	Perform basic element-by-element math on arrays: addition, subtraction, multiplication, division, exponentiation, floor division, and modulo operations. Examples: a + b, a * b, np.add(a, b).
Aggregation and reduction	Summarize arrays: sums, products, means, variances, standard deviations, cumulative sums, differences, and matrix traces. Examples: np.sum, np.mean, np.diff.
Trigonometric and hyperbolic functions	Calculate sin, cos, tan, arcsin, arccos, sinh, cosh, and tanh elementwise, useful for waves, rotations, and periodic models.
Exponential, logarithmic, and power functions	Apply exp, log, log10, log2, sqrt, cbrt, and power operations for modeling growth, scaling, and nonlinear transformations.
Logical, boolean, and comparison operations	Elementwise logical AND, OR, NOT; comparisons such as greater than/less than/equal to; boolean masking and conditional selection with np.where.
Sorting, searching, and set operations	Sort arrays, retrieve indexes, perform stable or lexicographical sorts, search values, find unique elements, and compute intersections/unions. Examples: np.sort, np.argsort, np.unique.
Linear algebra operations	Matrix operations: dot products, matrix multiplication, vector norms, matrix norms, inverses, determinants, eigenvalues, SVD (Singular Value Decomposition), QR (Orthogonal–Triangular), and solving systems. Examples: np.dot, np.linalg.inv, np.linalg.eig.
Fourier Transform operations	Transform signals between time and frequency domains using FFTs: np.fft.fft, np.fft.ifft, np.fft.fftn.
Complex number operations	Handle complex values: real/imag parts, magnitude, angle, conjugation. Examples: np.real, np.imag, np.angle, np.conj.
Array creation	Create arrays with zeros, ones, constants, sequences (arange, linspace), identity matrices, or functions. Examples: np.zeros, np.linspace, np.eye.
Array manipulation	Reshape, transpose, concatenate, split, repeat, broadcast, expand, or squeeze dimensions. Examples: reshape, transpose, stack, split.
Random number generation	Generate random samples from uniform, normal, Poisson, binomial, or integer distributions; shuffle and permute arrays. Examples: np.random.rand, np.random.normal.

Category	Types of Operations
Miscellaneous mathematical functions	Rounding, clipping, NaN/Inf checks, polynomial fitting, correlations, grid generation, histograms, and applying functions along axes.
String and text operations	Vectorized UTF-8 string handling: concatenation, comparison, case conversion, substring search, missing-data handling. Examples: np.char.add, np.char.find.

Table 12-25 Parameters for the scipy.integrate.dblquad function

Parameter	Data Type	Description	Default
func	callable	Function of at least two variables y, x	–
a, b	float	Limits of integration for x (where a < b)	–
gfun	callable or float	Lower boundary curve in y, can be a function of x or a constant	–
hfun	callable or float	Upper boundary curve in y, similar to gfun	–
args	tuple, optional	Extra arguments to pass to func	()
epsabs	float, optional	Absolute error tolerance	1.49e-08
epsrel	float, optional	Relative error tolerance	1.49e-08

Table 12-45 Return values for the scipy.linalg.svd function

Return	Description	Shape
U	Unitary matrix with left singular vectors	(M, M) or (M, K)
s	Array of singular values	(K,)
Vh	Unitary matrix with right singular vectors	(N, N) or (K, N)

Table 12-46 Summary of matrix functions and equation solvers in scipy.linalg

Function	Description
expm()	Matrix exponential
logm()	Matrix logarithm
cosm(), sinm(), tanm()	Matrix trigonometric functions
coshm(), sinhm(), tanhm()	Matrix hyperbolic functions
signm()	Matrix sign function
sqrtm()	Matrix square root
solve_sylvester()	Solves Sylvester equation
solve_continuous_are(), solve_discrete_are()	Solves Riccati equations
solve_continuous_lyapunov(), solve_discrete_lyapunov()	Solves Lyapunov equations
clarkson_woodruff_transform()	Randomized dimensionality reduction

Table 6-2 A trace of a while loop with an else block, showing when the else statement is executed

Iteration	Value of "i"	Condition (i <= 5)	Action	Output
1	1	True	Print "i" value.	1
2	2	True	Print "i" value.	2
3	3	True	Print "i" value.	3
4	4	True	Print "i" value.	4
5	5	True	Print "i" value.	5
6	6	False	Exit loop.	Loop finished!

Table 9-6 Common rounding modes in the decimal module

Constant Name	Rule Description	Decimal ("3.14")
ROUND_HALF_EVEN	Rounds to the nearest value. Ties go to the nearest even integer.	3.1
ROUND_HALF_UP	Rounds to the nearest value. Ties go away from zero.	3.1
ROUND_HALF_DOWN	Rounds to the nearest value. Ties go toward zero.	3.1
ROUND_UP	Always rounds away from zero.	3.2
ROUND_DOWN	Always rounds toward zero (truncates the value).	3.1
ROUND_CEILING	Rounds toward positive infinity (up).	3.2
ROUND_FLOOR	Rounds toward negative infinity (down).	3.1

Table 6-6 An iteration trace for a for loop that uses range(5) to generate a sequence of numbers

Iteration Number	Current Value of "i"	Output
1	0	0
2	1	1
3	2	2
4	3	3
5	4	4

Table 12-30 Parameters for the CubicSpline class

Parameter	Description	Default Value
x	1D array of independent variable values (must be real, finite, and strictly increasing)	–
y	Array of dependent variable values (can have arbitrary dimensions, but length along axis must match length of x)	–
axis	Axis along which y is assumed to be varying and use int data	0
bc_type	Boundary condition type (string or 2-tuple)	'not-a-knot'
	not-a-knot	
	periodic	
	clamped	
	natural	
	order, deriv_values	
extrapolate	Whether to extrapolate to out-of-bound points (bool, 'periodic', or None)	None (set to 'periodic' for bc_type="periodic", True otherwise)

Table 4-1 Examples of Python packages by application

Category	Package	Application
Data science and machine learning	NumPy	For numerical computations and array manipulation
	Pandas	For data manipulation and analysis
	Matplotlib	For data visualization
	Seaborn	For statistical data visualization
	Scikit-learn	For machine learning algorithms
	TensorFlow	For deep learning and neural networks
	PyTorch	Another popular deep learning framework
	Statsmodels	For statistical modeling and hypothesis testing
Web development	Flask	A lightweight web framework
	Django	A full-stack web framework
	FastAPI	For building APIs quickly and efficiently
	Requests	For making HTTP requests
Automation and scripting	Beautiful Soup	For web scraping
	Selenium	For browser automation
	PyAutoGUI	For GUI automation
	Schedule	For task scheduling
Game development	Pygame	For creating 2D games
	Godot-Python	For scripting in the Godot game engine
Scientific computing	SciPy	For scientific computing and technical computing
	SymPy	For symbolic mathematics
	OpenCV	For computer vision tasks
Networking and security	Socket	For low-level networking
	Paramiko	For SSH and SFTP
	Scapy	For packet manipulation and network analysis
Miscellaneous	Pillow	For image processing
	PyPDF2	For working with PDF files
	SQLAlchemy	For database management
	pytest	For testing Python code

Category	Package	Application
Engineering calculations and simulations	CoolProp	For thermophysical property calculations (e.g., fluids, gases)
	Cantera	For chemical kinetics, thermodynamics, and combustion modeling
	Pyomo	For optimization modeling in engineering and operations research
	OpenMDAO	For multidisciplinary design, analysis, and optimization
	FEniCS	For solving partial differential equations (PDEs)
	SimPy	For process-based discrete-event simulation
	Pint	Unit conversion and handling physical quantities
	Uncertainties	Propagation of uncertainties in calculations
	ANSYS-PyMAPDL	For scripting and automating ANSYS Mechanical simulations
	Abaqus-Py	For scripting and automating Abaqus simulations
	PyBullet	For physics-based simulations and robotics
	OpenFOAM-Py	For interfacing with OpenFOAM, a CFD (computational fluid dynamics) tool
	SALOME-Meca	For pre- and post-processing in finite element analysis (FEA)
	PySPH	For Smoothed Particle Hydrodynamics (SPH) simulations
	NetworkX	Graph and network analysis for engineering systems
Control systems	Control	For modeling, analysis, and design of control systems
	Python-Control	Similar to "control," used for linear and nonlinear control systems
	SymPy	Symbolic mathematics, useful for deriving transfer functions and state-space models
	SciPy-Signal	Signal processing and control system tools (e.g., Bode plots, transfer functions)
	Matplotlib	Visualization of control system responses (e.g., step response, Nyquist plots)
Signal processing	SciPy-Signal	Tools for filtering, Fourier Transforms, and signal analysis

Category	Package	Application
	NumPy	Numerical operations, including FFTs (Fast Fourier Transforms) and array manipulations
	PyWavelets	Wavelet transforms for signal processing
	Librosa	Audio and time-series signal processing
Numerical analysis	NumPy	Core library for numerical computations
	SciPy	Advanced numerical methods, including optimization, integration, and interpolation
	SymPy	Symbolic computation for solving equations and performing algebraic manipulations
	mpmath	Arbitrary-precision arithmetic for numerical analysis
Optimization	SciPy-Optimize	Optimization algorithms for engineering problems
	CVXPY	Convex optimization for control and engineering design
	Pyomo	Optimization modeling for linear and nonlinear problems
	PuLP	Linear programming and optimization
Finite element analysis	FEniCS	Open source library for solving partial differential equations using finite element methods
	PyCalculix	Finite element analysis for mechanical engineering problems
	SfePy	Simple finite element analysis in Python
Robotics	Robotics-Toolbox-Python	Tools for modeling and simulating robotic systems
	PyBullet	Physics simulation for robotics and control
	ROS (Robot Operating System)	Framework for robotics development (Python bindings available)
Data analysis	Pandas	Data manipulation and analysis
	Matplotlib	Visualization of engineering data
	Seaborn	Statistical data visualization
Machine learning	Scikit-learn	Machine learning algorithms for engineering applications
	TensorFlow	Deep learning framework for advanced engineering problems

Category	Package	Application
	PyTorch	Another deep learning framework, widely used in research and industry
Electrical engineering	scikit-rf (skrf)	Tools for RF and microwave engineering
	PyLab	MATLAB-like environment for numerical and signal processing
	LTSpice	Python interface for LTSpice circuit simulation
Mechanical engineering	PyBullet	Physics simulation for mechanical systems
	Matplotlib	Visualization of mechanical system simulations
	PyCalculix	Finite element analysis for mechanical engineering
Thermodynamics	CoolProp	Thermophysical property calculations for fluids
	Thermo	Chemical engineering thermodynamics and transport properties
Chemical engineering	Cantera	Chemical kinetics, thermodynamics, and transport processes
	Chemicals	Chemical properties and process engineering calculations
	Thermo	Thermodynamics and chemical engineering calculations
Structural engineering	PyNite	Structural analysis and finite element analysis for beams, trusses, and frames
	OpenSeesPy	Python interface for OpenSees, a structural analysis software
Fluid dynamics	OpenFOAM-Py	Open source CFD software with Python bindings
	PyFLUENT	Python interface for ANSYS Fluent
	FEniCS	Solving fluid dynamics problems using finite element methods

Table 6-3 A detailed trace of a while loop containing a nested if–elif–else structure

Iteration	Value of i	Condition (i <= 5)	Action	Output
1	1	True	Print "1 is less than 3".	1 is less than 3
2	2	True	Print "2 is less than 3".	2 is less than 3
3	3	True	Print "3 is equal to 3".	3 is equal to 3
4	4	True	Print "4 is greater than 3".	4 is greater than 3
5	5	True	Print "5 is greater than 3".	5 is greater than 3
6	6	False	Exit loop.	Loop finished!

Table 12-35 Summary of function groups within the scipy.linalg subpackage

Group	Description
Basics	Fundamental linear algebra functions, matrix inversion, solving systems of linear equations
Eigenvalue Problems	Finding eigenvalues and eigenvectors of matrices
Decompositions	Matrix decompositions, Lower–Upper decomposition, Singular Value Decomposition, Orthogonal–Triangular decomposition
Matrix Functions	Computing matrix functions, exponentials, trigonometric functions
Matrix Equation Solvers	Solving specific types of matrix equations, Sylvester, Riccati
Sketches and Random Projections	Generating random projections for dimensionality reduction and data compression
Special Matrices	Creating special matrices, block diagonal, circulant
Low-Level Routines	Access to underlying BLAS and LAPACK libraries

Table 8-5 Common commands for writing to files

Command	Description	Example
write()	Writes a string to the file	file.write('Hello')
writelines()	Writes a list of strings to the file	file.writelines(['Line1', 'Line2'])
writable()	Checks if the file is writable	file.writable()
flush()	Flushes the internal buffer, writing data to disk immediately	file.flush()

Table 9-11 Most frequently used functions of the random module

Function	Description	Common Use Cases
random.randint(a, b)	Returns a random integer N where a <= N <= b	Simulating dice rolls, generating random indexes, creating simple games
random.choice(seq)	Returns a single random element from a sequence	Picking a random winner, selecting a random item from a list of options
random.random()	Returns a random float X in the range 0.0 <= X < 1.0	The base for many other random functions; used when a fractional number is needed
random.shuffle(x)	Shuffles the elements of a sequence in place	Randomizing the order of items, like shuffling a deck of cards or a playlist
random.sample(population, k)	Returns k unique elements from a population	Drawing lottery numbers, selecting a random subset of data for testing
random.uniform(a, b)	Returns a random float N where a <= N <= b	Generating random values within a specific range, like a temperature or a price
random.seed(a)	Initializes the random number generator	Used for creating reproducible results, essential for testing and debugging

```
from kivy.app import App
from kivy.lang import Builder
# We define the UI and widgets in one string
kv_code = """
RelativeLayout:
 Button:
  text: 'Relative Button'
  size_hint: None, None
  size: '100dp', '36dp'
  pos: 50, 50
"""
class EngApp(App):
 def build(self):
  return Builder.load_string(kv_code)
if __name__ == '__main__':
 EngApp().run()
```

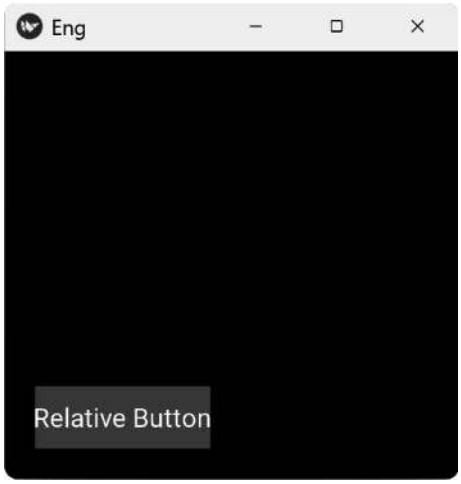


Figure 13-8 Positioning a button relative to its parent using RelativeLayout

Table 9-8 Preconfigured decimal.ExtendedContext

***Context*: decimal.ExtendedContext.**
This context is designed for applications that prioritize completing an execution even when they encounter exceptional conditions.
***Precision*: Nine digits.**
***Rounding*: ROUND_HALF_EVEN.**
***Traps*: No traps are enabled. When an operation such as division by zero or an invalid operation occurs, it does not throw an exception. Instead, it returns a special value such as Infinity or NaN (Not a Number). This allows a program to continue processing a large set of data, for example, and handle the exceptional results later.**

Table 3-2 List operators

Operator	Description
in, not in	Membership
+	Concatenation
*****	Repetition
[:]	Slicing
==, !=	Equality

Table 12-51 Parameters for the scipy.optimize.differential_evolution function

Parameter	Description	Default Value
func	Objective function to be minimized	–
bounds	Bounds for variables	–
args	Additional fixed parameters for the objective function	()
strategy	Differential evolution strategy	'best1bin'
maxiter	Maximum number of generations	1000
popsize	Population size multiplier	15
tol	Relative tolerance for convergence	0.01
mutation	Mutation constant	(0.5, 1)
recombination	Recombination constant	0.7
rng	Random number generator	None
callback	Callable function to be executed after each iteration	None
disp	Flag to print the evaluated function at every iteration	False
polish	Flag to perform polishing using a local optimization algorithm	True
init	Population initialization method	'latinhypercube'
atol	Absolute tolerance for convergence	0
updating	Update strategy for the best solution vector	'immediate'
workers	Number of workers for parallelization	1
constraints	Constraints on the solver	()
x0	Initial guess for the minimization	None
integrality	Boolean array indicating whether each decision variable is integer-valued	None
vectorized	Flag to indicate whether the objective function is vectorized	False

Table 1-5 Comparing Unicode characters

Format	Example	Output	Usage
\u + 4 hex digits	"\u03A9"	Ω	For characters in the Basic Multilingual Plane (BMP), which includes most European scripts, symbols, and special characters
\U + 8 hex digits	"\U0001F600"		For characters outside the BMP, such as emojis and characters from ancient or rare scripts
\N{name}	"\N{GREEK CAPITAL LETTER OMEGA}"	Ω	Uses the official Unicode name of the character, making code more descriptive and easier to read

Category	Command	Description
Project setup	uv init <name>	Creates a new project folder with basic files and prepares it for uv management
Dependency management	uv add <package>	Adds a package to your project, installs it into .venv, and updates the lockfile
	uv add --dev <package>	Adds a dependency used only for development
	uv remove <package>	Uninstalls a package and removes it from project configuration
Running code/environment handling	uv run <command>	Runs commands inside the project's isolated environment, creating or updating .venv if needed
	uv sync	Rebuilds .venv so it matches the lockfile exactly
	uv venv	Creates a standalone virtual environment manually (optional)
Lockfile	uv lock	Regenerates the lockfile with the exact versions your project uses
	uv cache prune	Removes unused cached packages to free space
Python versions	uv python install <version>	Installs a specific Python version managed by uv
	uv python pin <version>	Locks the project to a specific Python version
	uv python list	Shows all Python versions installed by uv
Tool/CLI handling	uv tool install <package>	Installs a Python-based command-line tool globally in an isolated environment
	uv tool list	Lists all globally installed tools
	uv tool run <tool>	Runs a tool without installing it permanently
On-demand tools	uvx <tool> [args]	Runs a command-line tool instantly without installation
Package utilities	uv pip install <package>	Installs packages using uv's fast pip replacement
	uv pip list	Shows packages installed in the current environment
	uv add --upgrade <package>	Updates a dependency to a newer compatible version
Self-maintenance	uv self update	Updates uv to the latest release

Table 12-13 Summary of helper functions in scipy.fft for optimizing and interpreting transform results

Function	Description
fftshift(x)	Shifts the zero-frequency component to the center of the spectrum
ifftshift(x)	The inverse of fftshift
fftfreq(n)	Returns the Discrete Fourier Transform sample frequencies
rfftfreq(n)	Returns the Discrete Fourier Transform sample frequencies (for usage with rfft, irfft)
fhtoffset(dln, mu)	Returns the optimal offset for a Fast Hankel Transform
next_fast_len(target)	Finds the next fast size of input data to fft, for zero-padding and others
prev_fast_len(target)	Finds the previous fast size of input data to fft
set_workers(workers)	Context manager for the default number of workers used in scipy.fft
get_workers()	Returns the default number of workers within the current context

Table 12-27 Summary of interpolation categories in scipy.interpolate

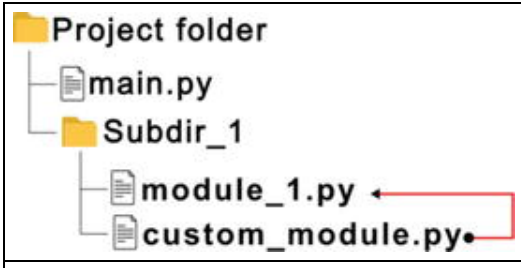
Topic	Description
Univariate interpolation	Methods for estimating values from one-dimensional data, such as single-variable functions, based on known x and y pairs
Multivariate interpolation	Techniques for estimating values from data in two or more dimensions, such as 2D or 3D, whether the data is arranged in a grid or scattered
1D splines	Smooth curve-fitting methods for one-dimensional data using splines, including B-splines, allowing control over the level of smoothness
2D splines	Spline-based methods for creating smooth surfaces from two-dimensional data, working for both regularly spaced gridded and irregular scattered data points
Rational approximation	Approximates functions using fractions of polynomials, such as Pade approximation or barycentric rational interpolation, to avoid problems like oscillations in high-degree polynomial fits
Additional tools	Extra interpolation options, including Lagrange polynomials and Taylor polynomial approximations, useful in special cases or for educational purposes

Table 1-7 List of useful commands for Windows, Linux, and macOS

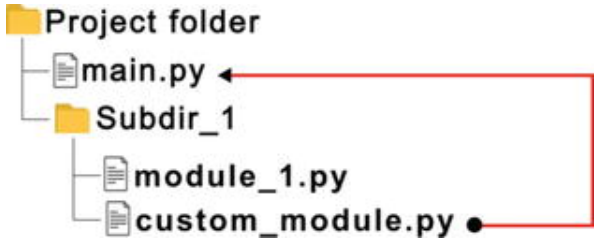
Category	Action	Command on Windows	Command on macOS/Linux
Version management	Check the active Python version.	python --version / python -V or py --version / py -V	python3 --version or python3 -V
	List all installed Python versions.	py -0	pyenv versions
	Run a script with a specific version.	py -3.9 my_script.py	python3.9 my_script.py
	Find the path of the Python executable.	where python	which python3
Virtual environments	Create a virtual environment.	python -m venv venv	python3 -m venv venv
	Activate the virtual environment.	.\venv\Scripts\activate	source venv/bin/activate
	Deactivate the virtual environment.	deactivate	deactivate
Package management with pip	List installed packages.	pip list	pip3 list
	Install a new package.	pip install <package>	pip3 install <package>
	Upgrade a package.	pip install --upgrade <package>	pip3 install --upgrade <package>
	Uninstall a package.	pip uninstall <package>	pip3 uninstall <package>
	Show details of a package.	pip show <package>	pip3 show <package>
	Create a dependency file.	pip freeze > requirements.txt	pip3 freeze > requirements.txt
	Install from a dependency file.	pip install -r requirements.txt	pip3 install -r requirements.txt
Code execution	Start the interactive console.	python or py	python3
	Run a script file.	python my_script.py	python3 my_script.py
Directory navigation	Change directory.	cd folder_name	cd folder_name
	Go up one directory.	cd ..	cd ..
	Go to the home directory.	cd %HOMEPATH%	cd ~

Category	Action	Command on Windows	Command on macOS/Linux
	Show current directory.	cd	pwd
	Create a new directory.	mkdir folder_name	mkdir folder_name
	Remove an empty directory.	rmdir folder_name	rmdir folder_name or rm -d folder_name
	Remove a directory with files.	rmdir /s folder_name	rm -rf folder_name
File management	Show contents of a text file.	type file.txt	cat file.txt
	Copy a file.	copy file.txt C:\backup\	cp file.txt ~/backup/
	Move or rename a file.	move file.txt C:\backup\	mv file.txt ~/backup/
	Delete a file.	del file.txt	rm file.txt
Terminal utilities	Clear the terminal screen.	cls	clear
	List files and folders.	dir	ls
	List with details.	dir	ls -l
	List including hidden files.	*Windows shows by default*	ls -a
	Display the PATH environment variable.	echo %PATH%	echo $PATH
	Check system IP address.	ipconfig	ifconfig or ip addr

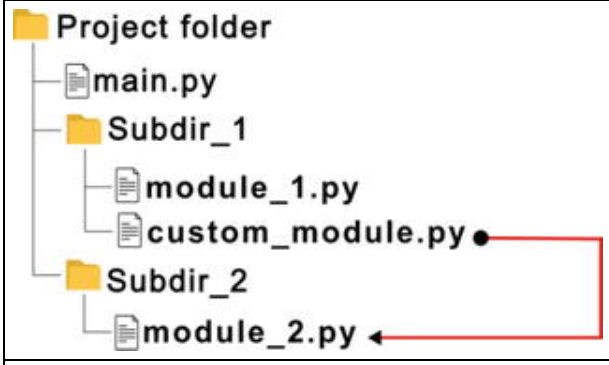
Table 4-6 Examples of relative imports



`# Importing all functions and variables.`
`from .module_1 import *`
`# Calling the function from module_1.py.`
`x = sum(2, 3)`
`print(x)  # Output: 5`

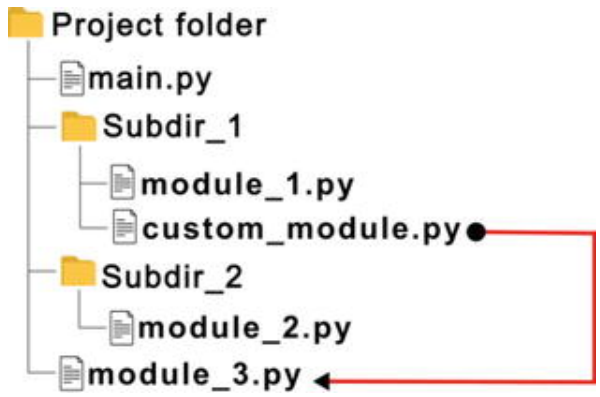


`# Importing all functions and variables.`
`from ..main import *`
`# Calling the function from main.py.`
`x = sum(2, 3)`
`print(x)  # Output: 5`

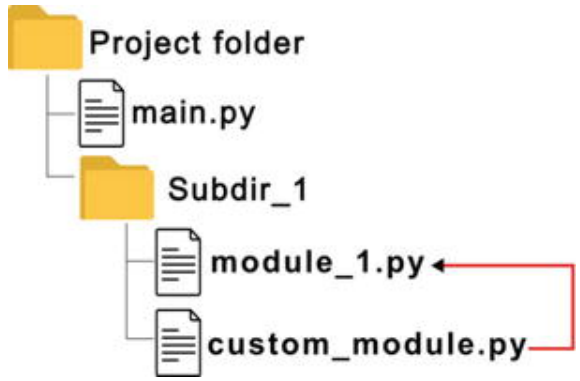
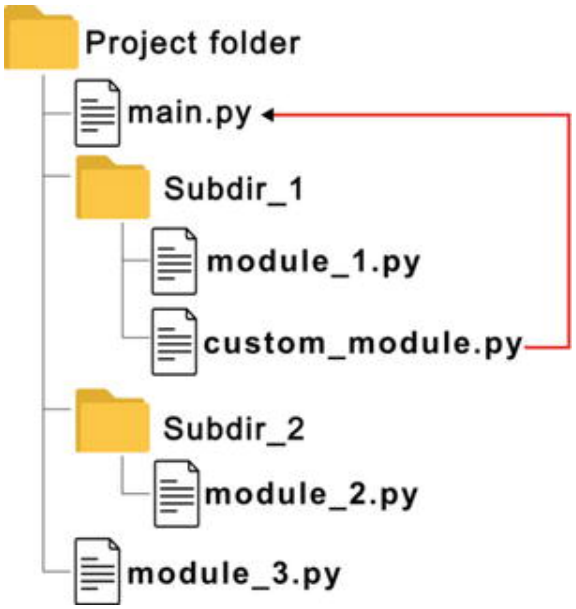


`# Importing all functions and variables.`
`from ..Subdir_2.module_2 import *`
`# Calling the function from module_2.py.`
`x = sum(2, 3)`

```
print(x)  # Output: 5
```




```
# Importing all functions and variables.
from ..module_3 import *
# Calling the function from module_1.py.
x = sum(2, 3)
print(x)  # Output: 5
```

Position of main.py relative to custom_module.py	Example of code in main.py to call custom_module.py
	
# Importing all functions and variables. **from .Subdir_1.Subdir_2.custom_module import ***	# Importing all functions and variables. from .Subdir_1.Subdir_2.custom_module import *

# Calling the 'sum' function from 'custom_module' to add two numbers. 	
x = sum(2, 3)
print(x) # Output: 5
	# Calling the 'sum' function from 'custom_module' to add two numbers.
	x = sum(2, 3)
	print(x) # Output: 5

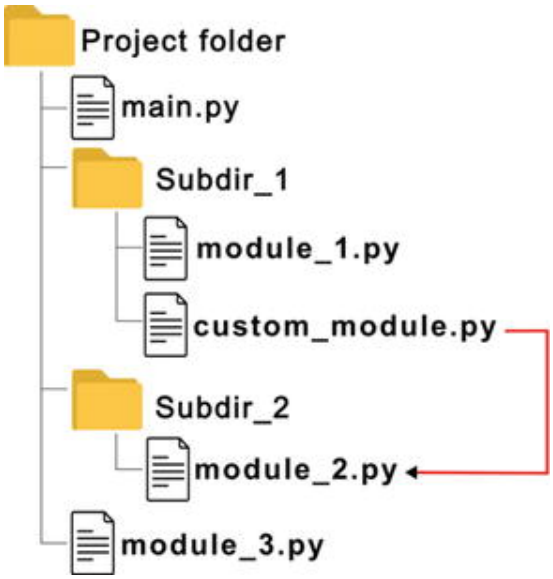
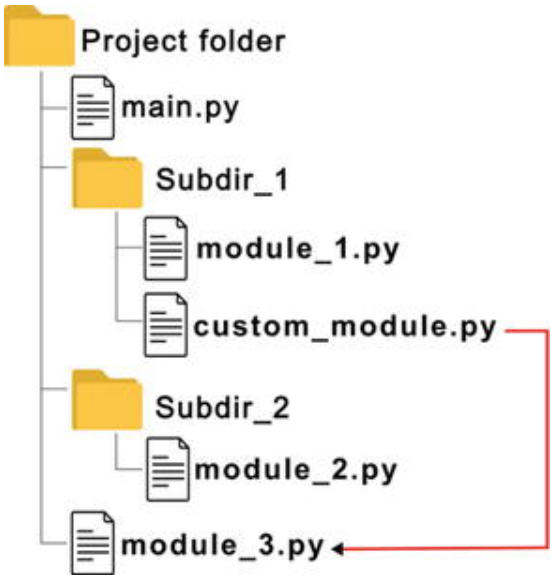
	

Table 6-11 A trace showing how the continue statement skips specific iterations in a for loop

Iteration	Value of "i"	Condition (i % 2 == 0)	Action	Output
1	1	False	Print "i" value.	1
2	2	True	Skip using "continue".	–
3	3	False	Print "i" value.	3
4	4	True	Skip using "continue".	–
5	5	False	Print "i" value.	5

Table 12-52 A selection of commonly used strategies for the differential_evolution function

Strategy	Description	Best Application
'best1bin'	Default strategy, uses binomial crossover with one difference vector	General-purpose optimization, suitable for most problems
'rand1bin'	Uses binomial crossover with one difference vector and random base vector	Multi-modal problems, where random base vector can help explore diverse solutions
'best1exp'	Uses exponential crossover with one difference vector	Problems with fewer local optima, where exponential crossover can help converge faster
'rand1exp'	Uses exponential crossover with one difference vector and random base vector	Problems with high dimensionality, where random base vector and exponential crossover can aid exploration

Table 12-63 Return values for the scipy.signal.tf2ss function

Return Value	Description	Type
A	State matrix	ndarray
B	Input matrix	ndarray
C	Output matrix	ndarray
D	Feedthrough matrix	ndarray

Table 4-7 Examples of absolute imports

Directory Structure	Example of Code in main.py to Call custom_module.py

	`# Importing all functions and variables.`
	`from custom_module import *`
	`# Calling the 'sum' function from 'custom_module' to add two numbers.`
	`x = sum(2, 3)`
	`print(x)  # Output: 5`

	`# Importing all functions and variables.`
	`from Subdir.custom_module import *`
	`# Calling the 'sum' function from 'custom_module' to add two numbers.`
	`x = sum(2, 3)`
	`print(x)  # Output: 5`

	`# Importing all functions and variables.`
	`from Subdir_1.Subdir_2.custom_module import *`
	`# Calling the 'sum' function from 'custom_module' to add two numbers.`
	`x = sum(2, 3)`
	`print(x)  # Output: 5`

Table 11-4 Main data types (dtypes), descriptions and value ranges supported by NumPy

Category	Dtype Name	Description	Value Range
Boolean	bool_	Boolean (True or False), stored as 1 byte.	{False, True}
Signed integers	int8	Signed 8-bit integer.	−128 to 127
	int16	Signed 16-bit integer.	−32,768 to 32,767
	int32	Signed 32-bit integer.	−2,147,483,648 to 2,147,483,647
	int64	Signed 64-bit integer.	−9.22×10^18 to 9.22×10^18
Unsigned integers	uint8	Unsigned 8-bit integer.	0 to 255
	uint16	Unsigned 16-bit integer.	0 to 65,535
	uint32	Unsigned 32-bit integer.	0 to 4,294,967,295
	uint64	Unsigned 64-bit integer.	0 to 1.84×10^19
Floating-point numbers	float16	Half-precision floating-point.	≈ −6.55×10^4 to 6.55×10^4
	float32	Single-precision floating-point.	≈ −3.4×10^38 to 3.4×10^38
	float64	Double-precision floating-point (default float).	≈ −1.8×10^308 to 1.8×10^308
	float128*	Extended-precision floating point (platform-dependent; commonly unavailable on Windows).	Platform-dependent; often 80-bit extended precision on x86-64 Linux
Complex numbers	complex64	Complex number using 2 × float32.	Real/imag range = float32
	complex128	Complex number using 2 × float64.	Real/imag range = float64
	complex256*	Complex number using the platform's extended-precision floating type (when available).	Platform-dependent
String types	str_	Fixed-length Unicode strings.	Depends on length
Bytes/ASCII strings	bytes_	Fixed-length raw bytes.	Depends on byte size
Object type	object_	Generic Python object stored in an array.	N/A

Category	Dtype Name	Description	Value Range
Datetime types	datetime64	Date/time representation with units (ns, ms, s, etc.).	Depends on unit (ns → ~1678 to 2262)
Timedelta types	timedelta64	Differences between datetime values.	Depends on unit (ns → ±292 years)
Structured/record types	custom dtype	Allows arrays of records with named fields (e.g., `dtype([('x', int), ('y', float)])`).	Depends on fields
Void type	void	Raw chunk of memory; used internally for structured arrays.	N/A
Flexible scalars	np.bytes_, np.str_	Fixed-width scalar string types; the width is set when the array is created.	Depends on length

Table 5-2 Common methods and use cases for lambda expressions in Python, such as their use with built-in functions and in sorting operations

Methods to Use Lambda Expressions		Description
1	Assigning a lambda to a variable	Assign a lambda expression to a variable and use it like a regular function.
2	Using a lambda inside built-in functions	Use a lambda with map(), filter(), zip(), and reduce() for data transformations.
3	Using a lambda in sorting	Lambda expressions serve as the key argument in sorting functions like sorted().
4	Using a lambda in list comprehensions	Lambda expressions are useful for performing inline operations within list comprehensions.
5	Using a lambda in conditional statements	Utilize a lambda function to implement conditional logic in an inline if–else statement.
6	Using a lambda in dictionaries	Use lambdas as values in dictionaries for dynamic behavior.
7	Using a lambda in regular functions	Return a lambda from a function for dynamic behavior.
8	Using any() and all() functions	Use generator expressions with any() and all() to evaluate conditions on iterables.
9	Using lambda in sorted() with multiple keys	Lambda expressions are useful for sorting by multiple criteria with sorted().

Table 12-49 Parameters and descriptions for the scipy.optimize.minimize function

Parameter	Description	Type	Default
fun	Function to minimize	Function	Required
x0	Initial guess	Array of numbers	Required
args	Extra arguments for fun parameter	Tuple of values	An empty tuple ()
method	Minimization algorithm	String of the custom function	None
jac	Gradient function or estimation	Function or string	None
hess	Hessian function or estimation	Function or string	None
hessp	Hessian times vector function	Function	None
bounds	Variable bounds	List of tuples	None
constraints	Constraints on variables	Dictionary or list of dictionaries	An empty tuple ()
tol	Tolerance for termination	Number	None
callback	Function called after each iteration	A Python function	None
options	Solver options	Dictionary	An empty dictionary {}

```
from kivy.app import App
from kivy.lang import Builder
# We define the UI and widgets in one string
kv_code = """
PageLayout:
  Button:
    text: 'Page 1'
  Button:
    text: 'Page 2'
"""
class EngApp(App):
  def build(self):
    return Builder.load_string(kv_code)
if __name__ == '__main__':
  EngApp().run()
```

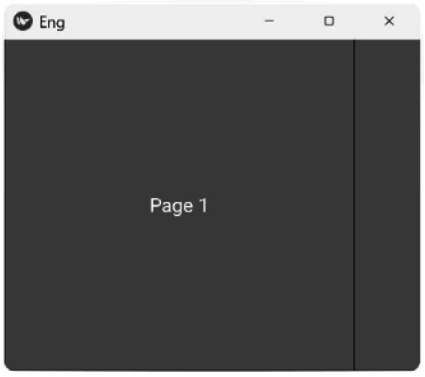


Figure 13-11 (upper screenshot). First page displayed by a PageLayout

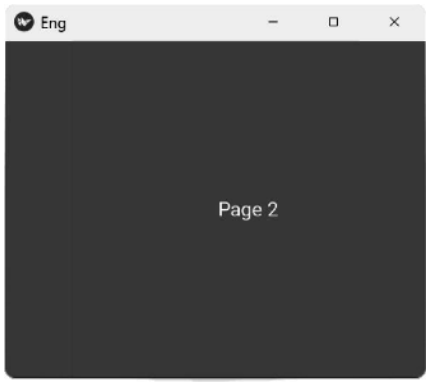


Figure 13-12 (lower screenshot). Second page displayed by a PageLayout

Table 10-2 Popular Python libraries for data visualization

Library	Use Case	Characteristics
Matplotlib	General-purpose and scientific plotting	Foundation of many other libraries. Offers immense, low-level control over every visual element. Excellent for publication-quality static plots. Interactivity is limited to basic panning and zooming. Requires more code, giving it a medium learning curve.
Seaborn	Statistical data visualization	Built on Matplotlib. Provides a high-level, beautiful default style for common statistical plots. Tightly integrated with Pandas DataFrames, making it easy to use. Interactivity is limited.
Plotly	Interactive and web-based dashboards	Creates fully interactive HTML-based visualizations. Excellent for web applications and 3D plots. Its high-level Plotly Express API makes it very easy to use. Provides high interactivity (hover, zoom, pan, select).
Bokeh	Interactive web-based dashboards and applications	Creates highly interactive plots and applications for web browsers. Optimized for large or streaming datasets. Can build complex dashboards with Python. Offers high interactivity but has a medium learning curve due to its powerful capabilities.
Altair	Declarative statistical visualization	It uses a simple, declarative grammar: you can describe what you want, not how to draw it. This makes it easy to use to create visually consistent, highly interactive graphics with minimal, readable code.
Folium	Interactive maps	Renders data on an interactive Leaflet.js map. Ideal for creating various map types like choropleth and markers for web pages. It is easy to use and produces highly interactive maps.

Table 13-1 Comparison of Python GUI libraries and mobile-platform support

Library	Description	Runs on Mobile (Android/iOS)
Tkinter	The standard GUI library for Python, included with most Python distributions. A popular choice for simple desktop applications.	No
PyGUI	A high-level, cross-platform GUI library designed to follow Python's idioms. It is now a legacy project with limited recent activity, so its maintenance status should be checked before use in a new application.	No
Kivy	An open source library for rapid development of applications with innovative user interfaces, such as multi-touch apps.	Yes
Flet	Enables building real-time web, mobile, and desktop apps. It uses Flutter for its UI, allowing for a modern look and feel.	Yes
Toga	Part of the BeeWare suite, Toga is a Python-native, OS-native, cross-platform GUI toolkit aiming for a "write once, run everywhere" experience.	Yes
PySide (Qt for Python)	The official Python bindings for the Qt framework. Qt has robust support for mobile development.	Yes
PyQt	A popular and mature set of Python bindings for the Qt framework, which supports mobile development.	Yes
wxPython	A wrapper for the popular cross-platform GUI toolkit, wxWidgets, which creates native-looking user interfaces.	No
Dear PyGui	A fast and powerful GUI framework that uses the immediate mode paradigm, ideal for dynamic UIs.	No
PySimpleGUI	A wrapper that simplifies GUI development with Tkinter and other backends. The project is open source under LGPLv3 again; version 5 previously used a commercial licensing model.	Partial (via web backend)

Table 1-6 Unicode examples

Symbol	Name/Description	Unicode	Symbol	Name/Description	Unicode
Mathematics			Technical		
≈	Almost equal to	U+2248	⌀	Diameter	U+2300
≠	Not equal to	U+2260	∠	Angle	U+2220
≤	Less than or equal to	U+2264	⊥	Perpendicular	U+22A5
≥	Greater than or equal to	U+2265	⏚	Ground symbol	U+23DA
±	Plus–minus	U+00B1	°	Degree	U+00B0
×	Multiplication sign	U+00D7	′	Prime/minute	U+2032
·	Multiplication dot	U+22C5	″	Double prime/second	U+2033
÷	Division sign	U+00F7	→	Rightward arrow	U+2192
√	Square root	U+221A	⇌	Equilibrium arrows	U+21CC
∫	Integral	U+222B	2	Superscript two	U+00B2
∑	Summation	U+2211	3	Superscript three	U+00B3
∞	Infinity	U+221E	**Emojis**		
∇	Nabla/gradient	U+2207	☑	Green check mark	U+2705
∂	Partial derivative	U+2202	❌	Red X	U+274C
Greek Letters			⚠	Warning	U+26A0
α	Alpha (lowercase)	U+03B1	🚀	Rocket	U+1F680
β	Beta (lowercase)	U+03B2	🚗	Car	U+1F697
Δ	Delta (uppercase)	U+0394	💡	Light bulb	U+1F4A1
δ	Delta (lowercase)	U+03B4	🔧	Wrench	U+1F527
ε	Epsilon (lowercase)	U+03B5	⚙	Gear	U+2699
θ	Theta (lowercase)	U+03B8	📈	Chart increasing	U+1F4C8
λ	Lambda (lowercase)	U+03BB	🔋	Battery	U+1F50B
μ	Micro/mu	U+00B5	📐	Triangular ruler	U+1F4D0
π	Pi (lowercase)	U+03C0	📏	Straight ruler	U+1F4CF
ρ	Rho (lowercase)	U+03C1	💻	Laptop computer	U+1F4BB
σ	Sigma (lowercase)	U+03C3	📊	Bar chart	U+1F4CA
τ	Tau (lowercase)	U+03C4	🔒	Locked	U+1F512

Symbol	Name/Description	Unicode	Symbol	Name/Description	Unicode
φ	Phi (lowercase)	U+03C6		Unlocked	U+1F513
ω	Omega (lowercase)	U+03C9		Link/chain	U+1F517
Ω	Omega (uppercase)	U+03A9		Construction barrier	U+1F6A7

Table 12-59 Parameters for the scipy.signal.firwin function

Param.	Description	Values/Default
numtaps	Length of the filter (number of coefficients)	int
cutoff	Cutoff frequency(ies) of filter	float or 1D array_like
width	Approximate width of the transition region	float or None, optional (default: None)
window	Desired window to use	string or tuple, optional (default: 'hamming')
pass_zero	Filter type (gain at frequency 0)	{True, False, 'bandpass', 'lowpass', 'highpass', 'bandstop'}, optional (default: True)
scale	Scale coefficients for unity gain	bool, optional (default: True)
fs	Sampling frequency of the signal	float, optional (default: 2)

```
from kivy.app import App
from kivy.lang import Builder
#Define the UI and widgets in one string
kv_code = """
BoxLayout:
 orientation: 'horizontal'
 spacing: 10
 padding: 10
 Button:
  text: 'Hello'
  size_hint: None, None
  size: '100dp', '36dp'
 Button:
  text: 'World'
  size_hint: None, None
  size: '100dp', '36dp'
"""
class EngApp(App):
 def build(self):
  return Builder.load_string(kv_code)
if __name__ == '__main__':
 EngApp().run()
```

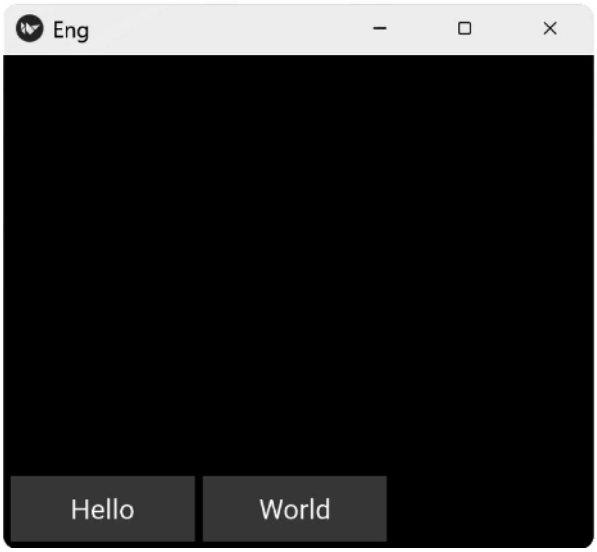


Figure 13-5 Horizontal arrangement of two buttons using BoxLayout