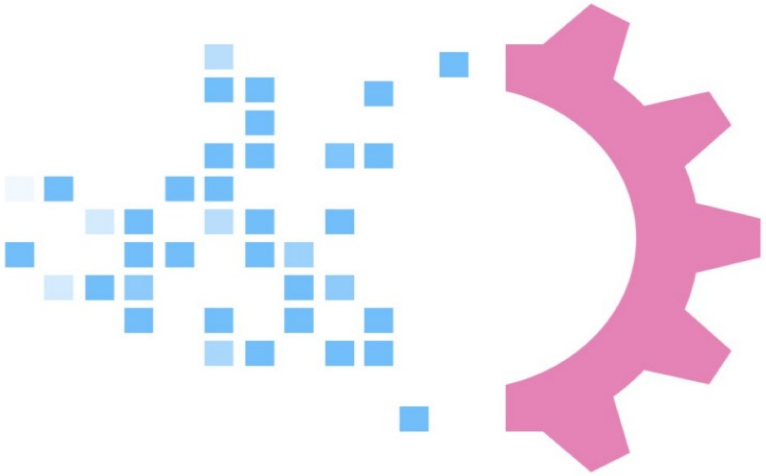
PRODUCTION READY DATA SCIENCE



FROM PROTOTYPING TO PRODUCTION WITH PYTHON

KHUYEN TRAN

CodeCut

Production Ready Data Science: From Prototyping to Production with Python

Khuyen Tran

Table of contents

Preface

Motivation

Have you ever encountered these situations in your data science projects?

- Your Jupyter Notebook starts simple but becomes a mess as the project grows
- Debugging takes forever because code is scattered and poorly organized
- Package installations break your environment and waste hours troubleshooting
- Code is difficult to adapt to new datasets or requirements
- Code fails to run consistently across different environments
- Changes are hard to track and rollback to previous working versions
- Previously written code is challenging to reuse and extend
- Critical bugs surface late in development
- Adding new features feels risky due to potential regressions

These challenges arise from the gap between exploratory data analysis and production-grade software engineering practices. This book aims to bridge this gap.

The book covers a wide range of essential topics for building production-ready data science applications. Here's an overview of what you'll learn:

1. **Version Control for Code:** Explore version control systems like Git and learn how to apply version control practices to your code, enabling you to track changes, collaborate with others, and manage your codebase effectively.
2. **Dependency Management:** Learn how to handle Python package dependencies using tools like pip or poetry, ensuring consistent and reproducible environments for your projects.
3. **Python Modules and Packages:** Master the creation, organization, and use of Python modules and packages to structure your code efficiently and promote reusability.
4. **Python Variables, Functions, and Classes:** Learn techniques for writing clean and modular code using variables, functions, and classes, enabling better code organization and reusability.
5. **Unit Testing:** Learn how to write effective unit tests using frameworks like pytest, enabling you to catch bugs early, improve code quality, and facilitate future code changes.
6. **Project Configuration:** Learn how to separate configuration parameters from code logic, allowing for easier customization and deployment across different environments.
7. **Logging and Exception Handling:** Learn how to generate informative log messages that aid debugging, troubleshooting, and monitoring application behavior.
8. **Data Validation:** Discover techniques for validating data types, ranges, formats, and consistency, enabling you to build more reliable and robust data science pipelines.
9. **Version Control for Data:** Learn strategies and tools for versioning your data, ensuring reproducibility and traceability in your data science projects.
10. **Packaging Projects:** Discover how to structure your project for distribution, create setup files, and publish your package to PyPI, making it easy for others to install and use your code.
11. **Building a CI Pipeline:** Learn how to set up a Continuous Integration (CI) to automate code testing and documentation generation, ensuring code quality and facilitating collaborative development.

12. **Jupyter Notebook Best Practices:** Master techniques for creating well-structured, reproducible, and shareable Jupyter notebooks, including cell organization, markdown usage, and version control integration.

Audience

The primary audience for this book includes:

1. **Data Scientists**: Professionals who are skilled in data analysis, machine learning, and statistical modeling, but may lack experience in software engineering practices necessary for production environments.
2. **Data Analysts**: Those who work with data and create analyses but want to improve the scalability and maintainability of their projects.
3. **Machine Learning Engineers**: Professionals who are looking to bridge the gap between creating models and deploying them in production environments.
4. **Data Science Students**: Advanced students or recent graduates who want to learn practical skills for transitioning from academic projects to industry-standard practices.
5. **Research Scientists**: Those in academia or research institutions who want to make their work more reproducible and easier to collaborate on.
6. **Data Science Team Leads**: Professionals responsible for improving their team's workflow and code quality.

Prerequisites

- Familiarity with fundamental Python concepts, syntax, and data structures.

- A foundational understanding of basic data science concepts, such as data processing and model training.
- Basic knowledge of using the command-line interface for tasks like navigating directories and running scripts.
- Basic familiarity with popular data science tools like pandas, NumPy, and matplotlib would be beneficial but not mandatory.

What Makes This Book Different

1. **Simplified Language:** The book materials are presented in a manner that is easy to understand, making complex concepts more accessible to learners.
2. **Visual Support:** Clear and visually appealing graphs and examples accompany each concept and topic, enhancing understanding and providing visual aids for better retention.
3. **Practical Examples:** The examples provided are directly related to data science projects, offering practical applications for the concepts discussed.

Hands-On Examples

This book is accompanied by a comprehensive GitHub repository containing practical implementations of every concept discussed:

https://github.com/khuyentran1401/production-ready-data-science-code

Each directory contains runnable examples, sample data, and detailed README files to help you practice the concepts immediately.

About the Author

Khuyen Tran transforms how data scientists learn and work. She has written over 180 articles as a top writer on Towards Data Science, helping data professionals bridge the gap between prototyping and production.

As founder of CodeCut, she publishes daily Python tips in her newsletter that reach over 10,000 views per month and has built a

community of 110,000 LinkedIn followers.

Previously an MLOps Engineer and Senior Data Engineer at Accenture, she built enterprise data solutions for clients worldwide.

Copyright

Production Ready Data Science: From Prototyping to Production with Python

First Edition

Published: January 2025

Published by: CodeCut Technologies LLC

Author: Khuyen Tran

Contact: khuyentran@codecut.ai or visit codecut.ai

Disclaimer

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or

damage caused or alleged to be caused directly or indirectly by the information contained in this book.

The code examples and techniques presented in this book are for educational purposes. Readers should exercise caution and best practices when implementing these techniques in production environments.

Trademarks

All trademarks mentioned in this book are the property of their respective owners.

1 Version Control

1.1 What Is Version Control?

Version control is a system that tracks changes to files and enables software developers to collaborate in a safe, organized, and effective way. Version control allows teams to manage their codebase efficiently, revert changes when needed, and safely experiment with code changes.

1.2 Why Is Version Control Essential?

Version control is especially important in a data science project for several key reasons.

1.2.1 Track Changes and Revert Easily

Version control provides safety and efficiency by tracking every code change, allowing quick recovery when problems occur, and maintaining a complete project history.

Consider developing a machine learning model for customer churn prediction without version control. After making significant changes to your `model.py` file, testing shows degraded performance. Without an accurate record of changes, you spend hours manually trying to undo modifications, risking new errors in the process.

Version control solves this by letting you commit changes regularly during development. When testing reveals performance drops, you can review commit history, identify the problematic change, and revert to the previous working state, as shown in Figure 1.1.

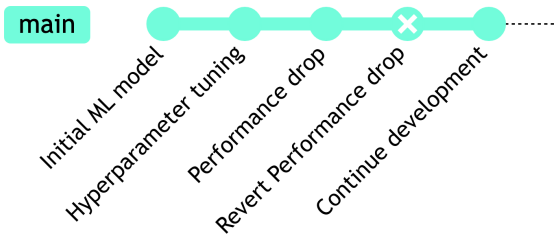


Figure 1.1: Version control workflow for machine learning model development

1.2.2 Collaborate Effectively

Version control transforms team collaboration by eliminating file conflicts, tracking contributor changes, and enabling organized project coordination.

Consider a data analysis project with multiple team members where each person saves work on their local machine or shared drive. Combining everyone's work creates conflicting file versions and overwritten changes. You spend hours manually merging different code versions, trying to reconcile discrepancies and ensure nothing is lost.

Version control provides a shared repository where each team member works on their own branch without affecting the main codebase. Changes are tracked with contributor information and timestamps, enabling safe merging back into the main branch, as illustrated in Figure 1.2.

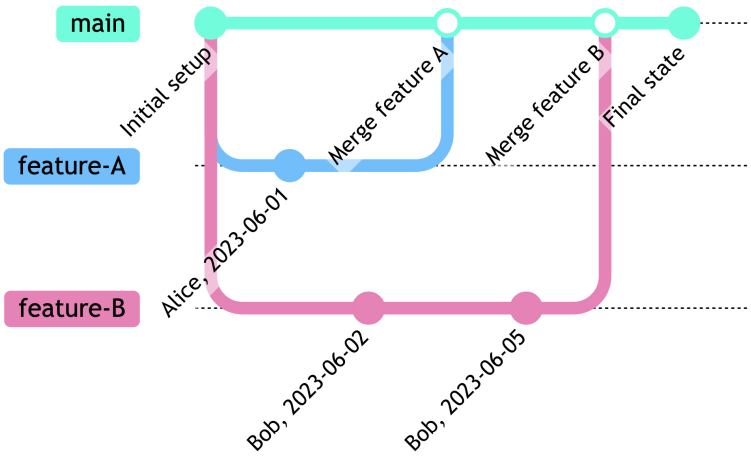


Figure 1.2: Branching and merging workflow in version control

1.2.3 Reproduce Results Reliably

Version control delivers reliable research reproduction by tracking exact code versions and removing guesswork about which files generated specific results.

Consider this scenario: you publish a machine learning model, then need to reproduce results six months later. Without version control, you find multiple script copies with slight variations but can't identify which version created the published results.

Version control eliminates this uncertainty by letting you tag the exact code version used for publication. When you need to reproduce results, you simply checkout the tagged version to recreate the analysis, as shown in Figure 1.3.

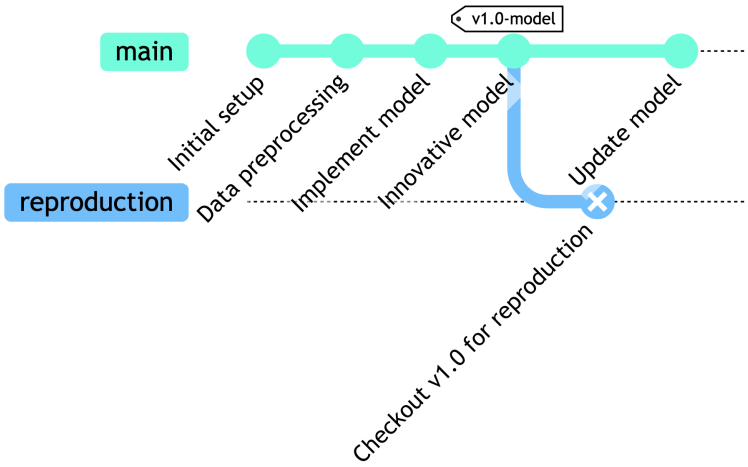


Figure 1.3: Reproducing analysis using a tagged version of code

1.2.4 Experiment Safely

Version control eliminates the fear of experimentation by providing a safety net for code changes. Instead of risking production systems with direct modifications, you can test ideas in isolation, maintain system stability, and recover quickly from failed experiments.

Consider this scenario: you're working on a production data workflow that processes customer data daily and want to test an optimization. Without version control, you make changes directly to production code. Your experiment fails, breaking the system and disrupting daily data flow while you struggle to revert changes under pressure.

With version control, you create a new branch called `feature/new-processing` and freely experiment with your new ideas where changes won't affect the main production code. After thoroughly testing your experiment, you create a pull request to merge your changes into the `main` branch.

If your changes don't work, you simply discard the experimental branch without affecting the production code, as shown in Figure 1.4, allowing you to innovate without fear of breaking the existing system.

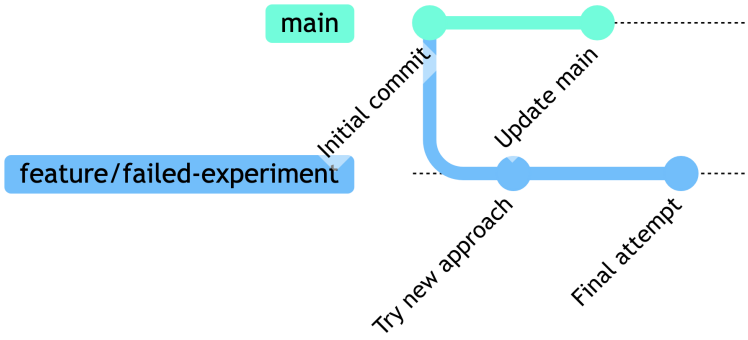


Figure 1.4: Discarding failed experimental branch

1.2.5 Backup Your Project Securely

Version control protects your work by creating automatic backups and preserving your project history. You can recover from computer crashes, accidental deletions, and other disasters without losing progress.

Imagine working on a data science project for weeks, making steady progress on your code and files. Your computer crashes, and you realize you don't have a backup of your project. You've lost all your hard work and must start from scratch. This becomes a frustrating and time-consuming process that sets your project back significantly.

Version control solves this by letting you create a repository for your project and commit changes regularly. The repository serves as both project history and backup, keeping your code safe. If your computer crashes or you accidentally delete a file, you simply restore the project from the repository, as illustrated in Figure 1.5.

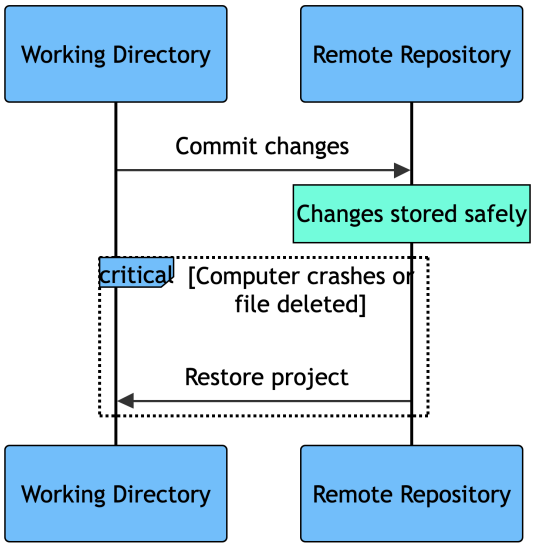


Figure 1.5: Restoring a project from a remote repository

1.3 Use Git for Version Control

To implement effective version control and reap its benefits, developers need a robust tool. This is where Git comes into play. Git is a free open-source version control tool that's ubiquitous and trusted by developers worldwide.

1.3.1 Key Git Concepts

Before diving into Git usage, let's understand some key terminology:

- **Working Directory**: This is the directory on your computer where you're actively working on your project files.
- **Local Repository**: This is a hidden `.git` folder in your working directory that contains the complete history of your project. When you commit changes, they are stored here.

- **Remote Repository**: This is a version of your project hosted on a server (like GitHub or GitLab). The Remote Repository allows you to back up your code and collaborate with others. You can push changes to and pull updates from your repository.

Figure 1.6 illustrates the components of Git version control.

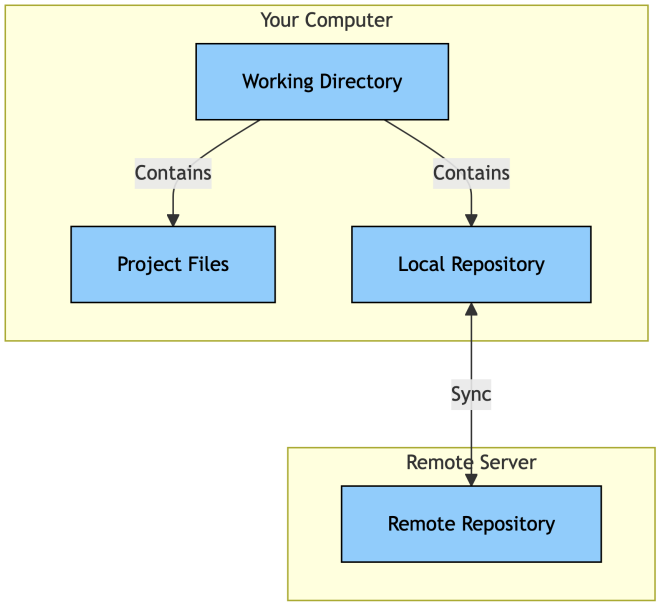


Figure 1.6: Components of Git Version Control

Let's explore how we can effectively use Git in different scenarios.

1.3.2 Scenario 1: Starting a New Project

When beginning a new data science project, establishing version control from the start creates a solid foundation for development. This scenario covers the complete workflow from initializing Git in your project directory to connecting with a remote repository for backup and collaboration.

1.3.2.1 Overview

This workflow involves three main phases:

- Creating the local repository
- Making your first commit
- Connecting to a remote repository for backup

1.3.2.2 Step-by-Step Process

Phase 1: Initialize Local Repository

1. Initialize a new Git repository in your working directory:

```
git init
```

Phase 2: Create First Commit

2. Stage the changes or new files in your Git repository:

```
# Add all changes and new files
git add .
```

3. Review the list of changes to be committed:

```
git status

Changes to be committed:

      new file:   .gitignore
      new file:   .pre-commit-config.yaml
      ...
```

4. Save the staged changes permanently in your local repository's history along with a commit message:

```
git commit -m 'init commit'
```

Phase 3: Connect to Remote Repository

5. Create a repository on GitHub/GitLab and add the remote connection. If you're using GitHub as the remote repository,

create a new repository on GitHub and copy its URL. Then, add the URL to your local Git repository with the name `origin`:

```
git remote add origin <repository URL>
```

6. Push your initial commit to establish the remote backup:

```
# Push to the main branch on the origin repository
git push origin main
```

Figure 1.7 illustrates this workflow.

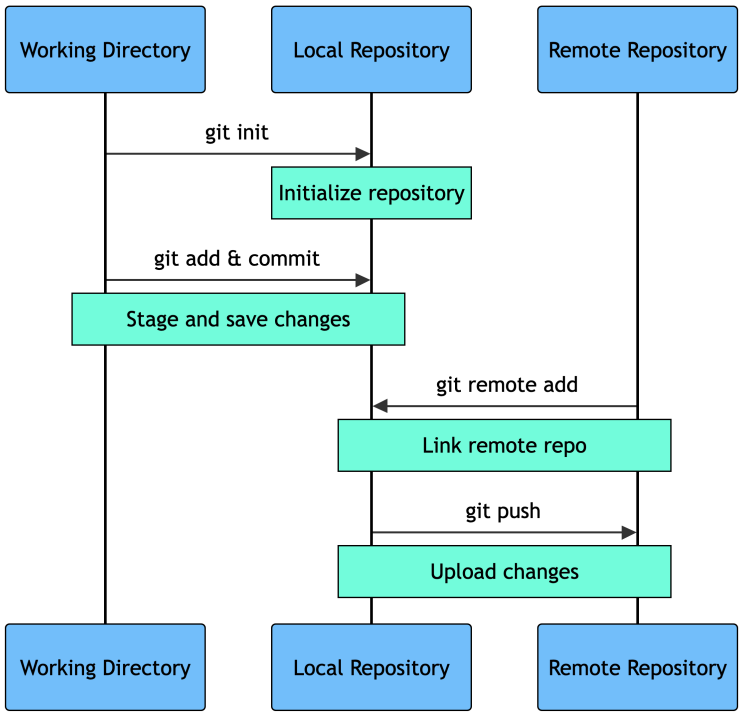


Figure 1.7: Initializing a Git repository and uploading the project to a remote repository

1.3.3 Scenario 2: Contributing to an Existing Project

When you want to contribute to an existing data science project, whether it's an open-source library or your team's codebase, you need to safely integrate your changes without disrupting the main project. This scenario covers the complete workflow from forking and cloning to submitting your contributions through pull requests.

1.3.3.1 Overview

This workflow involves four main phases:

- Getting access to the project code
- Setting up your local development environment
- Making and testing your changes
- Submitting your contributions for review

1.3.3.2 Step-by-Step Process

Phase 1: Get Access to the Project Code

1. Fork the repository on GitHub if you don't have write access to the main repository.

2. Use `git clone` to create a local copy of the remote repository on your machine.

   ```
   git clone https://github.com/username/project-name.git
   ```

Phase 2: Set Up Your Development Environment

3. Navigate to the project directory:

   ```
   cd project-name
   ```

4. Create and switch to a new branch to safely develop your changes without affecting the main codebase:

   ```
   git checkout -b <branch-name>
   ```

Phase 3: Implement Your Changes

5. Make your code modifications in the new branch.

6. Stage, commit, and push your changes:

```
git add .
git commit -m "Descriptive message about your changes"
git push origin <branch-name>
```

Phase 4: Submit Your Contribution

7. Create a pull request on GitHub to propose merging your changes. This enables project maintainers to review your contributions before integrating them into the main project.

Figure 1.8 illustrates this process.

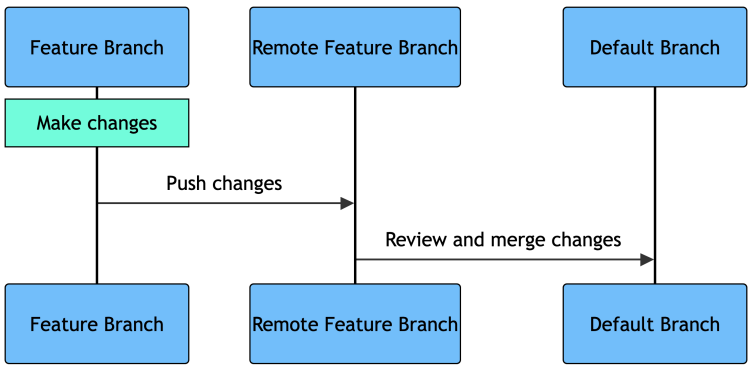


Figure 1.8: Pull request workflow

1.3.4 Scenario 3: Staying Synchronized

When working on a team project or contributing to an active repository, the main branch often receives updates while you're developing your features. This scenario covers how to keep your local

work synchronized with remote changes to avoid conflicts and maintain a current codebase.

1.3.4.1 Overview

This workflow involves two main phases:

- Securing your current work
- Integrating remote updates

1.3.4.2 Step-by-Step Process

Phase 1: Secure Your Current Work

1. Ensure your local work is saved by staging and committing your local changes. This prevents losing your progress:

```
git add .
git commit -m 'commit-2'
```

Phase 2: Integrate Remote Updates

2. Pull changes from the remote main branch with `git pull`, which creates a merge commit combining your work with the latest updates:

```
git pull origin main
```

Figure 1.9 illustrates this process.

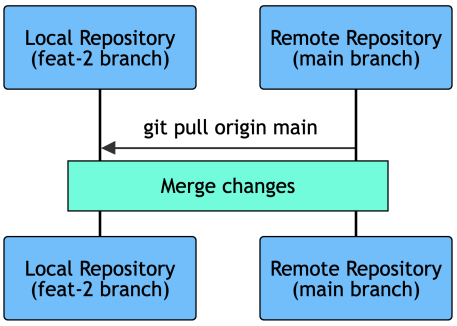


Figure 1.9: Merging remote changes from the main branch into the local feat-2 branch

1.4 Best Practices in Version Control

1.4.1 Error Recovery and History Management

Have you ever pushed a commit and immediately realized it contained a bug? When you need to undo changes in a shared repository, choosing the right recovery method prevents disrupting team workflows and maintains project integrity.

Git provides two main approaches for handling these situations:

- **Safe recovery with `git revert`**: Creates new commits that undo changes, preserving complete history
- **History rewriting with `git reset`**: Moves branch pointer to different commits, effectively rewriting history

1.4.1.1 When You Need to Preserve History

Use `git revert` when:

- Commits have been pushed to shared repositories

- Working in team environments where history preservation is important
- You want to maintain a complete audit trail of changes

To revert a specific commit, first identify the commit hash:

```
git log
```

```
commit 0b9bee172936b45c3007b6bf6fa387ac51bdeb8c
    commit-2

commit 992601c3fb66bf1a39cec566bb88a832305d705f
    commit-1
```

Then use `git revert` with the commit hash:

```
git revert 992601c3fb66bf1a39cec566bb88a832305d705f
```

Figure 1.10 illustrates the git revert process.

1.4.1.2 When You Need to Remove Commits

Use `git reset` when:

- Commits exist only in your local repository
- You need to completely remove commits from history
- Working on private feature branches before sharing

To reset commits, identify the target commit hash with `git log`, then choose your reset type:

```
# Soft reset: Keep changes staged
git reset --soft <commit-hash>

# Mixed reset: Keep changes unstaged (default)
git reset <commit-hash>

# Hard reset: Discard all changes
git reset --hard <commit-hash>
```

Figure 1.11 illustrates the git reset process.

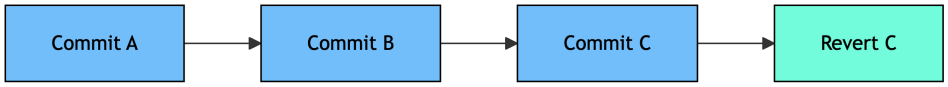


Figure 1.10: Git revert creates new commit to undo changes

Figure 1.11: Git reset removes commits from history

Warning

Warning: Unlike `git revert`, `git reset` rewrites commit history. Never use `git reset` on commits that have been pushed to a shared repository, as this can cause problems for other team members.

1.4.2 Managing Uncommitted Work

Have you ever been deep in coding when you suddenly need to pull updates from the remote repository, but your changes aren't ready to commit? Properly managing work-in-progress prevents lost changes and maintains clean development workflows.

Git stash provides a solution for temporarily storing uncommitted changes:

- **Temporary storage**: Save current changes without creating commits
- **Clean workspace**: Switch branches or pull updates safely
- **Easy restoration**: Reapply stashed changes when ready to continue

For example, when you have uncommitted changes but need to pull updates:

```
git status
```

```
On branch feat-2
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working
directory)
        modified:   file1.txt
        modified:   file2.txt
```

Use `git stash` to temporarily save your changes:

```
git stash
```

Now your working directory is clean:

```
git status
```

```
On branch feat-2
nothing to commit, working tree clean
```

You can safely pull updates:

```
git pull origin feat-2
```

After pulling, reapply your stashed changes:

```
git stash pop
```

```
On branch feat-2
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working
directory)
        modified:   file1.txt
        modified:   file2.txt
Dropped refs/stash@{0} (1234abcd5678efgh)
```

Figure 1.12 illustrates this process.

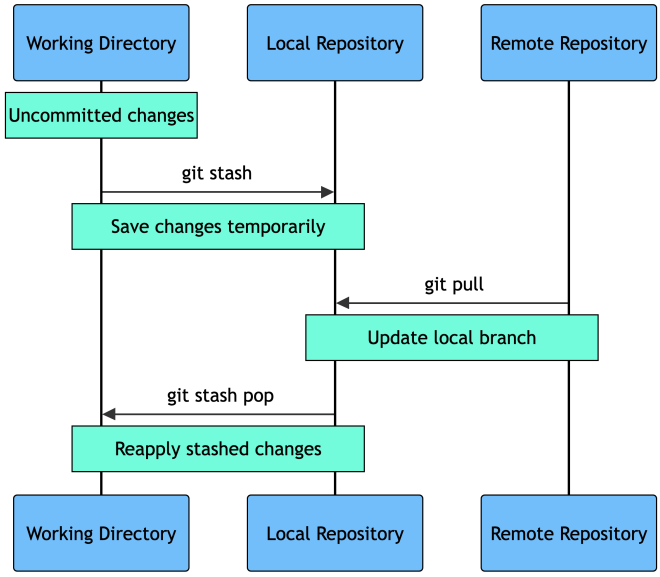


Figure 1.12: Stashing and reapplying changes during a pull operation

1.4.3 Ignore Large and Private Files

Have you ever tried to clone a repository only to wait ages for a massive download to complete? When developers include large datasets or confidential credentials in their Git repository, it creates bloated repositories that are slow, insecure, and difficult to share.

Git's ignore functionality solves this by letting you specify which files to exclude from version control. This helps:

- Keep repositories small and efficient
- Protect sensitive information
- Reduce unnecessary version tracking of large binary files

Create a `.gitignore` file in your project's root directory to specify which files and directories Git should ignore (shown in Example 11.2).

Example 1.1: .gitignore

```
# Ignore large data files
*.csv
*.parquet
*.feather
*.h5

# Ignore model files
*.pkl
*.joblib
*.pt

# Ignore sensitive information
.env

# Ignore Jupyter notebook checkpoints
.ipynb_checkpoints/

# Ignore virtual environment
venv/
env/

# Ignore IDE files
.vscode/
```

1.4.4 Commit Often and Logically

Have you ever struggled to understand what changed in a massive commit? Large commits mixing unrelated changes make it difficult to review, understand, and selectively revert specific modifications.

When commit focuses on a specific aspect of the project, it becomes easier to:

- Track changes and their impact
- Review code modifications
- Revert specific changes if needed
- Understand the project's evolution
- Communicate what each commit does to other team members

Here are some examples of small commits with clear, descriptive messages:

```
# Commit 1: Data preprocessing
git commit -m "Add data cleaning and preprocessing steps"

# Commit 2: Feature engineering
git commit -m "Create new features for customer churn prediction"

# Commit 3: Model training
git commit -m "Train initial random forest model"

# Commit 4: Model evaluation
git commit -m "Add detailed model evaluation"

# Commit 5: Model improvement
git commit -m "Optimize model hyperparameters"
```

1.4.5 Fetch Before Merge

Have you ever run `git pull` only to find unexpected merge conflicts or mysterious code changes in your working directory? Using `git pull` automatically merges remote changes without review, causing unexpected conflicts and unintentional changes.

Instead, use `git fetch` followed by `git merge` to examine incoming changes before merging. You can review new commits, check for conflicts, and decide when and how to integrate updates into your work.

Here are the steps to fetch and merge remote changes:

1. Fetch the latest changes from the remote repository without modifying your working directory.

   ```
   git fetch origin
   ```

2. Review the changes:

```
# Check differences between current branch and remote main
git log ..origin/main

e4f5g6h Update data preprocessing script
d7e8f9a Add new feature extraction function
```

3. Once you're satisfied with the changes, merge them:

```
git merge origin/main

Updating a1b2c3d..e4f5g6h
Fast-forward
preprocessing.py | 15 ++++++++++++---
feature_extraction.py | 25 +++++++++++++++++++++++++
2 files changed, 37 insertions(+), 3 deletions(-)
```

Figure 1.13 demonstrates this process.

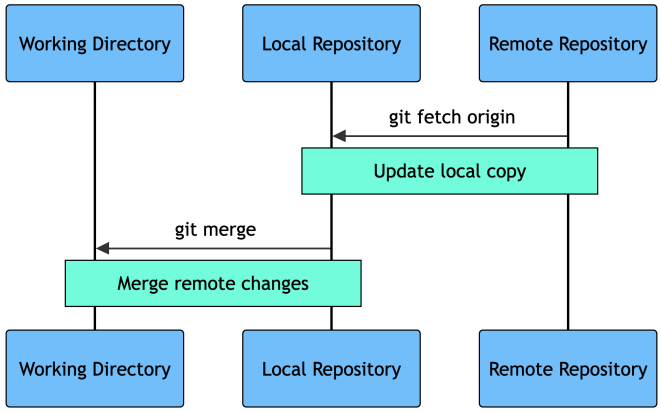


Figure 1.13: Fetching and merging remote changes

1.5 Key Takeaways

Version control is an essential tool for data scientists, enabling efficient collaboration, experimentation, and project management. Here are the key takeaways from this chapter:

1. Core benefits of version control:
    - Track changes and revert to previous versions when needed
    - Collaborate effectively with team members
    - Reproduce results reliably
    - Experiment safely without affecting production code
    - Backup your project safely and securely
2. Git fundamentals:
    - Working Directory: Where you make changes to files
    - Local Repository: Stores complete project history
    - Remote Repository: Hosts code for backup and collaboration
    - Commits: Snapshots of changes with descriptive messages
    - Branches: Isolated environments for feature development
3. Essential Git commands:
    - `git init`: Start a new repository
    - `git add` & `git commit`: Save changes
    - `git push` & `git pull`: Sync with remote repository
    - `git branch` & `git checkout`: Manage different versions
    - `git merge`: Combine changes from different branches
    - `git revert` & `git reset`: Undo changes when needed
4. Best practices:
    - Use `.gitignore` to exclude large files and sensitive data
    - Make small, focused commits with clear messages
    - Fetch before merging to review changes
    - Create feature branches for new development
    - Use `git revert` for shared repositories, `git reset` only locally
    - Use `git stash` for work-in-progress storage
    - Regularly sync with the remote repository

By following these practices and understanding these concepts, you can effectively manage your code, collaborate with others, and maintain a clean, organized project history.

2 Dependency Management

2.1 What Is Dependency Management?

2.1.1 Dependencies

Dependencies are the code libraries or packages that a project depends on. For example, in a data science project, you might use the following libraries:

- pandas for data manipulation and analysis
- scikit-learn for machine learning algorithms
- matplotlib and seaborn for data visualization
- requests for making HTTP requests
- pytest for testing code

External packages provide functionality that you can use in your project without having to write everything from scratch.

2.1.2 Dependency Management Tool

A dependency management tool is software that helps automate and streamline the process of managing project dependencies. Dependency management tools handle tasks such as:

- Installing packages and their dependencies
- Resolving version conflicts between packages

- Creating isolated environments for different projects
- Generating dependency specifications
- Updating packages to the newest version
- Ensuring reproducible environments across different machines

Python has multiple dependency management tools available, each with specific strengths and weaknesses:

pip: Simple but limited

- Pros: Minimal learning curve, established ecosystem
- Cons: No environment management, slow resolution

Conda: Scientific computing focus

- Pros: Built-in environments, fast resolution
- Cons: Limited packaging, moderate learning curve

uv: Modern and fast

- Pros: Fastest resolution, comprehensive packaging
- Cons: Growing ecosystem, moderate learning curve

The next section will cover best practices that apply to dependency management regardless of which tool you choose.

2.2 Best Practices for Dependency Management

To ensure efficient and effective dependency management in your Python data science projects, it is essential to follow a set of best practices:

2.2.1 Use Virtual Environments

Data scientists frequently need to manage multiple projects simultaneously, each with unique package version requirements. Installing dependencies globally can lead to conflicts between different projects.

For example, one project might require pandas 1.0 to maintain compatibility with legacy code, while another needs pandas 2.0 to utilize newer features. Installing both versions globally could lead to compatibility issues since Python can only load one version of a package at a time, as shown in Figure 2.1.

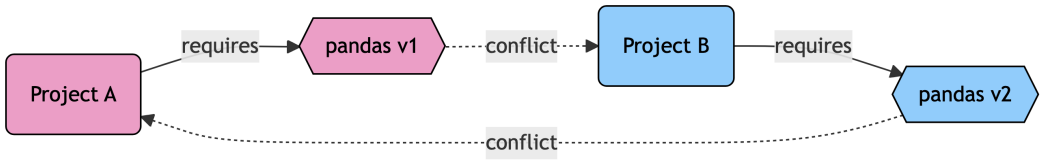


Figure 2.1: Conflicting package version requirements between two projects

To solve this issue, you can create separate virtual environments for each project. A virtual environment is an isolated Python environment. When you activate a virtual environment, any packages you install will be installed only in that environment, not globally on your system.

As shown in Figure 2.2, project A's environment contains pandas version 1 while project B's environment contains pandas version 2, preventing any version conflicts between the projects.

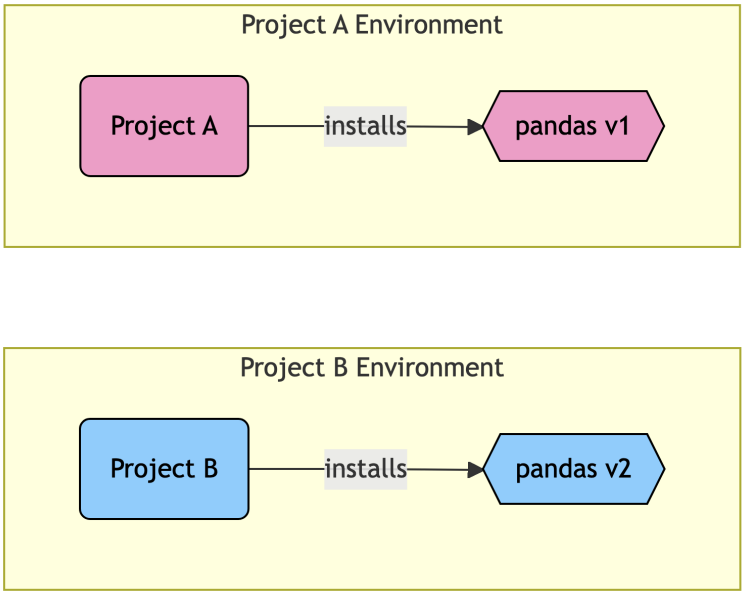


Figure 2.2: Using virtual environments to solve conflicting package version requirements

2.2.2 Avoid Specifying Exact Versions for Dependencies

Specifying exact versions or including all nested dependencies in your requirements file can lead to rigid setups that block beneficial updates.

For example, pinning pandas to `1.3.3` in Example 2.1 prevents you from getting bug fixes in `1.3.4`. However, unrestricted versions risk breaking changes when major versions jump from `1.3.3` to `2.0.0`.

Example 2.1: requirements.txt

```
pandas==1.3.3
matplotlib==3.4.3
scikit-learn==0.24.2
```

Example 2.2: requirements.txt (Continued)

```
numpy==1.21.2
python-dateutil==2.8.2
pytz==2021.1
six==1.16.0
cycler==0.10.0
kiwisolver==1.3.1
pillow==8.3.2
pyparsing==2.4.7
scipy==1.7.1
joblib==1.0.1
threadpoolctl==2.2.0
...
(potentially dozens more sub-dependencies)
```

To solve this issue, use version ranges to automatically receive safe updates while preventing breaking changes. For example, the range `pandas>=1.3.3,<1.4.0` in Example 2.3 gets you pandas 1.3.4's improvements while avoiding potential pandas 1.4.0 incompatibilities.

Example 2.3: requirements.txt

```
pandas>=1.3.3,<1.4.0
```

2.2.3 Separate Development and Production Dependencies

Mixing development and production dependencies creates bloated deployments and potential conflicts.

Example 2.4 combines development tools (pytest, pre-commit) with production libraries (scikit-learn, pandas), forcing production environments to install unnecessary tools.

Example 2.4: requirements.txt

```
pytest # development
pre-commit # development
scikit-learn # development and production
pandas # development and production
```

Instead, separate dependencies into two files: `requirements.txt` for production and `requirements-dev.txt` for development. Install the appropriate file based on your environment to deploy only necessary packages efficiently.

Example 2.5: requirements.txt

```
pandas==1.3.3
matplotlib==3.4.3
scikit-learn==0.24.2
```

Example 2.6: requirements-dev.txt

```
-r requirements.txt
pytest==6.2.5
black==21.9b0
flake8==3.9.2
```

Install production dependencies:

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

Install development dependencies:

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

2.3 Use uv to Manage Dependencies

uv is a modern Python package installer and resolver written in Rust. While there are many libraries to manage Python dependencies, uv stands out for its exceptional speed and reliability.

uv's implementation in Rust brings several key performance and reliability advantages:

- Installs packages 10-20x faster than pip, making it ideal for large projects

- Uses significantly less memory than other package managers when handling complex dependency trees

uv also acts as a drop-in replacement for pip, providing faster package installation and dependency resolution through its optimized Rust implementation. Below is a quick look at the tools uv consolidates into one streamlined interface:

uv Functionality	Replaces Tool(s)
Dependency management	pip, pip-tools, Poetry
Virtual environment creation	virtualenv, venv, Poetry
CLI tool execution	pipx
Python version management	pyenv
Project management	Poetry

You can install uv as a standalone executable or as a Python package. Follow the installation guide to get started.

Let's explore uv's functionalities.

2.3.1 Initialize a Project with Ease

uv simplifies project initialization by handling all the setup complexity for you. Instead of manually creating configuration files and setting up project structure, uv generates everything needed for a modern Python project. Create a new project with the `uv init` command:

```
uv init
```

After running the command, the following files will be created:

- `.python-version`: The Python version used in the project.
- `README.md`: The README for the project.

- `main.py`: The main entry point for the project.
- `pyproject.toml`: The project metadata and dependencies (shown in Example 2.7).

Example 2.7: pyproject.toml

```
[project]
name = "test-uv"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.11"
dependencies = []
```

2.3.2 Run a Python Script

uv simplifies script execution by automatically managing the environment for you. Instead of manually activating virtual environments, use `uv run` to execute scripts with the correct dependencies:

```
uv run main.py
```

2.3.3 Add Dependencies

uv simplifies dependency management by combining package installation with project configuration updates. Rather than separately installing packages and updating requirement files, uv automatically maintains your `pyproject.toml` and lock files.

To add new packages, use the `uv add` command:

```
uv add pandas scikit-learn
```

Running this command automatically updates your `pyproject.toml` file to with the new dependencies:

Example 2.8: pyproject.toml

```
[project]
name = "test-uv"
version = "0.1.0"
description = "Add your description here"
readme = "README.md"
requires-python = ">=3.11"
dependencies = [
    "pandas>=2.2.3",
    "scikit-learn>=1.6.1",
    "seaborn>=0.13.2",
]
```

The `add` command also creates a `uv.lock` file containing the exact versions of all installed packages and their dependencies.

Example 2.9: uv.lock

```
[[package]]
name = "pandas"
version = "2.2.3"
```

Example 2.10: uv.lock (Continued)

```
dependencies = [
    { name = "numpy" },
    { name = "python-dateutil" },
    { name = "pytz" },
    { name = "tzdata" },
]

[[package]]
name = "numpy"
version = "2.2.6"
...

[[package]]
name = "python-dateutil"
version = "2.9.0.post0"
...
```

2.3.4 Uninstall Packages

When removing packages, uv also automatically uninstalls dependent packages, thereby freeing up storage space and minimizing potential conflicts.

Let's test this out by first installing pandas and scikit-learn:

```
uv add pandas scikit-learn
```

The dependency tree reveals the shared dependency: both pandas and scikit-learn depend on NumPy.

```
uv tree
```

```
test-uv v0.1.0
├── pandas v2.2.3
|   ├── numpy v2.2.6
|   └── python-dateutil v2.9.0.post0
├── scikit-learn v1.6.1
|   ├── joblib v1.5.1
|   └── scipy v1.15.3
...
```

Now, let's remove pandas:

```
uv remove pandas
```

```
Resolved 20 packages in 47ms
Uninstalled 5 packages in 236ms
 - pandas==2.2.3
 - python-dateutil==2.9.0.post0
 - pytz==2025.2
 - six==1.17.0
 - tzdata==2025.2
```

The tree output confirms uv removed pandas and its exclusive dependencies (python-dateutil, pytz, tzdata) while preserving NumPy, which scikit-learn still requires.

```
test-uv v0.1.0
├── scikit-learn v1.6.1
|   ├── joblib v1.5.1
|   ├── numpy v2.2.6
```

```
|   ├── scipy v1.15.3
|   |   └── numpy v2.2.6
|   └── threadpoolctl v3.6.0
...
```

After running the `remove` command, the `pyproject.toml` file will be updated to reflect the changes:

Example 2.11: pyproject.toml

```
...
dependencies = [
    "scikit-learn>=1.6.1",
]
```

2.3.5 Reproduce an Environment

Lock files enable consistent development environments across different machines and team members by capturing exact dependency versions.

When a team member clones your project containing the `pyproject.toml` and `uv.lock` files, they can easily reproduce your exact environment by running the following command:

```
uv sync
```

```
Resolved 25 packages in 16ms
Installed 23 packages in 142ms
```

2.3.6 Separate Dependencies for Different Purposes

Dependency groups offer a structured approach to managing project requirements by categorizing packages based on their purpose. uv provides a convenient way to separate dependencies for different purposes while keeping track of all dependencies in a single place.

For example, you can add dependencies for a production environment with the following command:

```
uv add numpy pandas
```

Running this command will add dependencies under `dependencies` in the `pyproject.toml` file.

Example 2.12: pyproject.toml

```
dependencies = [
    "pandas>=2.2.3",
    "scikit-learn>=1.6.1",
]
```

Later, when you need to add dependencies for development and testing, you can use the `--dev` flag to specify the dependencies for the `dev` group:

```
uv add pytest pre-commit --dev
```

This will add the development dependencies to your `pyproject.toml` file:

Example 2.13: pyproject.toml

```
...
dependencies = [
    "pandas>=2.2.3",
    "scikit-learn>=1.6.1",
]
```

Example 2.14: pyproject.toml (Continued)

```
[dependency-groups]
dev = [
    "pre-commit>=4.2.0",
    "pytest>=8.3.5",
]
```

To install both the development and production dependencies, you can run the following command:

```
uv sync
```

To exclude the development dependencies, you can use the `--no-dev` flag:

```
uv sync --no-dev
```

2.3.7 Update Python Version

Consider a project that requires Python 3.8, but you need to upgrade to Python 3.11. Traditional Python version upgrades require downloading installers, configuring PATH variables, and recreating virtual environments.

With uv, you can complete this entire process using just two commands.

```
uv python install 3.11.2
uv python pin 3.11.2
```

2.3.8 Manage Dependencies for Single-File Scripts

Sometimes, you just want to run a script without installing anything globally, like when exploring data with matplotlib or seaborn for a quick one-off task without.

uv makes this effortless by allowing you to declare dependencies inline and automatically manage an isolated environment tied to the script itself.

For example, you can run the `main.py` script with seaborn in an isolated environment without installing anything globally.

Example 2.15: main.py

```
import seaborn as sns
import matplotlib.pyplot as plt


# Sample data
data = sns.load_dataset("penguins").dropna()
```

Example 2.16: main.py (Continued)

```
# Plot using seaborn only
sns.scatterplot(
    data=data, x="flipper_length_mm", y="body_mass_g"
)
plt.title("Flipper Length vs Body Mass by Species")
plt.show()
```

```
uv run --with seaborn main.py
```

2.3.9 Execute and Install CLI Tools

Tools like ruff, black, and isort are often used globally across projects. Installing them in the base environment can cause version conflicts or unnecessary clutter.

uvx is a command-line tool that comes with uv for running Python CLI tools in isolated environments. With uvx, you can run CLI tools on demand in isolated environments without needing to install them first or worry about version conflicts.

To run ruff without installing it:

```
uvx ruff check main.py
```

```
All checks passed!
```

2.3.10 Replace Pip, pip-tools, and Virtualenv Seamlessly

If you're installing packages with pip, freezing requirements, or managing environments with virtualenv, you can adopt uv without changing your existing workflow.

uv uses familiar commands but runs them much faster and more cleanly. Let's see how to use uv to create a virtual environment, install packages, and run CLI tools.

Create a virtual environment:

```
uv venv
```

Activate the virtual environment:

```
# Unix/macOS
source .venv/bin/activate

# Windows
.venv\Scripts\activate
```

Install packages:

```
uv pip install pandas scikit-learn
```

Deactivate the virtual environment:

```
deactivate
```

2.4 Key Takeaways

Dependency management is a crucial aspect of Python development, especially in data science projects where reproducibility and collaboration are essential. Here are the key takeaways from this chapter:

1. Best practices:
    - Use virtual environments to isolate dependencies

    - Declare your project's dependencies with flexible version ranges
    - Separate development and production dependencies using dependency groups
2. Key features of uv:
    - Fast package installation and dependency resolution
    - Intelligent dependency cleanup that only removes unused packages
    - Easy environment reproduction with `uv sync`
    - Support for running scripts and CLI tools in isolated environments
    - Drop-in replacement for familiar tools like pip and virtualenv

These practices and uv's capabilities ensure clean, reproducible, and efficient Python environments.

3 Python Modules and Packages

3.1 What Are Python Modules and Packages?

3.1.1 Python Modules

Python modules are `.py` files containing reusable code elements like variables, functions, and classes.

For example, we can create two interconnected modules:

- The `utils` module includes both the configuration and CSV saving functionality (shown in Example 3.1).
- The `process_data` module imports and uses the `save_to_csv` function and `config` variable from `utils` (shown in Example 3.2).

Example 3.1: utils.py

```
# Configuration dictionary
config = {'data_path': 'data', 'model_path': 'model'}

# CSV saving function
def save_to_csv(file_name, result):
    result.to_csv(file_name, index=False)
    print(f"Data is saved to {file_name}")
```

Example 3.2: process_data.py

```
from utils import save_to_csv, config
import pandas as pd
from pathlib import Path
```

Example 3.3: process_data.py (Continued)

```
# Create sample data
X = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]})

# Use imported config
data_path = config['data_path']
Path(data_path).mkdir(exist_ok=True)

# Use imported function
save_to_csv(f'{data_path}/mydata.csv', X)
```

3.1.2 Python Packages

Packages are essentially folders that contain one or more related code modules.

In the following directory structure, `src` is a package that contains three distinct modules: `process_data`, `train_model`, and `utils`.

```
src
├── process_data.py
├── train_model.py
└── utils.py
```

3.2 Project Organization Best Practices

3.2.1 Break Down Large Files into Smaller Modules

Consider Example 3.4, which contains multiple distinct responsibilities:

- Data loading
- Preprocessing
- Model training
- Model evaluation

Example 3.4: main.py

```
def load_data(filepath):
    ...

def preprocess_data(df):
    ...

def train_model(X, y):
    ...
```

Example 3.5: main.py (Continued)

```
def evaluate_model(model, X_test, y_test):
    ...

# Main execution
df = load_data('data.csv')
df = preprocess_data(df)
X, y = df.drop('target', axis=1), df['target']
model, X_test, y_test = train_model(X, y)
accuracy = evaluate_model(model, X_test, y_test)
```

Writing a large file with multiple responsibilities can lead to several issues:

- The file can become long and difficult to manage.
- It's unclear which parts of the code are responsible for which functionality.
- Adding new features or modifying existing ones becomes risky, because changes might affect other parts of the code.

Instead, it's better to break down large files into smaller modules in order to make your code easier to read and manage. For example, we can break down the Example 3.4 to multiple modules:

- `process.py` handles data loading and preprocessing (shown in Example 3.6).
- `train_model.py` focuses on model training and evaluation (shown in Example 3.7).
- `main.py` coordinates the execution flow (shown in Example 3.8).

Example 3.6: process.py

```
def load_data(filepath):
    ...

def preprocess_data(df):
    ...
```

Example 3.7: train_model.py

```
def train_model(X, y):
    ...

def evaluate_model(model, X_test, y_test):
    ...
```

Example 3.8: main.py

```
from data_processing import load_data, preprocess_data
from model import train_model, evaluate_model



df = load_data('data.csv')
df = preprocess_data(df)
X = df.drop('target', axis=1)
y = df['target']
model, X_test, y_test = train_model(X, y)
accuracy = evaluate_model(model, X_test, y_test)
```

As the project grows, you can expand Example 3.7 to Example 3.9 without affecting other modules:

Example 3.9: train_model.py

```
def train_model(X, y):
    ...

def evaluate_model(model, X_test, y_test):
    ...

# Add a new function
def cross_validate(model, X, y, cv=5):
    ...
```

3.2.2 Have a Standardized Project Structure

Without a standardized project structure, data science projects can quickly become chaotic and hard to maintain. For example, the following directory structure places all files in the root directory without categorization, making it difficult to locate data, models, and other components.

```
project_root/
├── data.csv
├── analysis.ipynb
├── model.py
├── utils.py
├── results.txt
├── requirements.txt
└── README.md
```

A standardized project structure provides a better approach. It helps team members locate project components quickly, improving efficiency and consistency. New developers can also orient themselves to your code more easily.

The following directory structure shows a well-organized data science project structure:

```
project_root/
|
├── config/                 # Configuration files
├── data/                   # Data storage
|   ├── processed/          # Processed data
|   └── raw/                # Original data
├── docs/                   # Project documentation
├── models/                 # Trained models and artifacts
├── notebooks/              # Jupyter notebooks
├── src/                    # Source code
|   ├── __init__.py
|   ├── process_data.py
|   ├── train_model.py
|   └── utils.py
├── tests/                  # Test scripts
|   ├── __init__.py
|   ├── test_process_data.py
|   └── test_train_model.py
├── README.md               # Project overview
└── pyproject.toml          # Dependencies
```

Let's go over the folders in this structure:

- `config/`: Contains configuration files (YAML, JSON) for project settings, hyperparameters, etc.
- `data/`: Stores all data used in the project.
- `docs/`: Holds project documentation, which may include project overview, methodologies, results, and any other relevant information.
- `models/`: Stores trained models and related artifacts (e.g., serialized model files, model checkpoints).
- `notebooks/`: Contains Jupyter notebooks used for exploratory data analysis, prototyping, and visualizations.
- `src/`: Contains the main source code for the project.
- `tests/`: Contains test scripts to ensure the reliability of the code.

3.3 Import Best Practices

3.3.1 Avoid Importing Everything

Avoid using wildcard imports or importing entire modules since it can lead to confusion and potential errors in your code.

Consider the `processors.py` module that contains a custom `SimpleImputer` class. The `process_data.py` module uses both this custom `SimpleImputer` and `KNNImputer` from scikit-learn with wildcard imports.

Example 3.10: processors.py

```
class SimpleImputer:
    ...
```

Example 3.11: process_data.py

```
from processors import *
from sklearn.impute import *

simple_imputer = SimpleImputer() # imported from sklearn.impute
iterative_imputer = KNNImputer()
```

Using wildcard imports can be problematic for several reasons:

- It's not immediately clear which module `SimpleImputer` and `KNNImputer` come from without knowing the contents of each module.
- Our intention is to use the `SimpleImputer` class from the `processors` module, but the `SimpleImputer` class in `process_data` is from `sklearn.impute` instead. This is because Python will use the last imported definition when there are naming conflicts.

Use explicit imports instead to import only necessary classes or functions from each module. This makes code easier to maintain and debug by clearly showing which `SimpleImputer` version is used:

Example 3.12: process_data.py

```
from processors import SimpleImputer
from sklearn.impute import KNNImputer

simple_imputer = SimpleImputer() # imported from processors
iterative_imputer = KNNImputer()
```

3.3.2 Use Absolute Imports

Python supports both relative and absolute imports for accessing modules. Absolute imports specify the complete module path from the project root, while relative imports use dots to navigate directory hierarchies.

Avoid relative imports since they can cause confusion, potential errors, and issues when restructuring your project.

Consider a data science project with the following directory structure:

```
.
└── project/                  # parent directory of src/
    ├── src/                  # parent directory of train_model.py
    |   ├── process_data.py
    |   └── train_model.py    # you are here
    └── utils/
        └── helpers.py
```

In the `train_model.py` module, you can use relative imports:

Example 3.13: project/src/train_model.py

```
from . import process_data # import from src/
from .. import utils       # import from project/
```

While this works initially, if you move `train_model.py` into a new `models/` subdirectory:

```
project/
├── src/                      # parent directory of models/
|   ├── process_data.py
```

```
|   └── models/             # parent directory of train_model.py
|       └── train_model.py  # you are here
└── utils/
    └── helpers.py
```

The relative import in `train_model.py` will break because `.` now refers to `src/models/` instead of `src/`:

Example 3.14: project/src/train_model.py

```
from . import process_data # import from src/models/

# ModuleNotFoundError: No module named 'process_data'
```

With absolute imports, relocating `train_model.py` won't cause the error:

Example 3.15: project/src/train_model.py

```
from project.src import process_data
from project import utils
```

3.3.3 Use Main Block

The main block, denoted by `if __name__ == "__main__":`, prevents code from executing during imports and separates reusable functions from script-specific code.

Without a main block, code executes immediately when you import a module, causing unintended behavior. Consider this example:

- `process_data.py` with no main block (shown in Example 3.16)
- `main.py` that calls the function in `process_data.py` (shown in Example 3.17)

Example 3.16: process_data.py

```
def process_data(data: list):
    return [num + 1 for num in data]
```

```
print(f"Process data from {__name__}")
process_data([1, 2, 3])
```

Example 3.17: main.py

```
from process_data import process_data

print(f"Process data from {__name__}")
process_data([1, 2, 3])
```

Running `main.py` executes the `process_data` function twice due to calls in both modules. The execution sequence is:

- Importing `process_data.py` executes the module code, printing `Process data from process_data`
- Calling the function in `main.py` prints `Process data from __main__`

```
python main.py
```

```
Process data from process_data
Process data from __main__
```

Figure 3.1 illustrates the flow of execution.

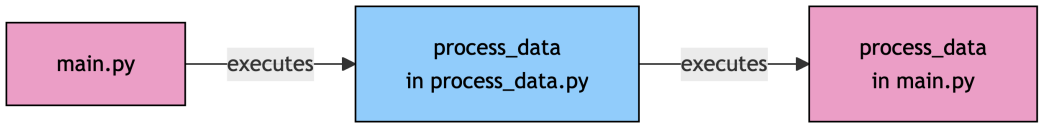


Figure 3.1: Without a main block, modules can be executed unintentionally during import

Instead, add a main block to both modules, as shown in Example 3.18 and Example 3.19. With the main block in place:

- Importing `process_data.py` defines only the function without executing the main block code
- `main.py` executes its main block, calling `process_data([1, 2, 3])` once and printing `"Process data from __main__"`

Example 3.18: process_data.py

```python
def process_data(data: list):
    return [num + 1 for num in data]

if __name__ == "__main__":
    # This code is only called when process_data is run directly
    print(f"Process data from {__name__}")
    process_data([1, 2, 3])
```

Example 3.19: main.py

```python
from process_data import process_data

if __name__ == "__main__":
    # This code is only called when main is run directly
    print(f"Process data from {__name__}")
    process_data([1, 2, 3])
```

Now, running `main.py` produces:

```
python main.py
```

```
Process data from __main__
```

Figure 3.2 shows the improved flow of execution when using a main block.

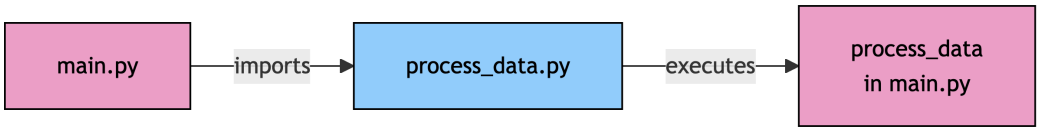


Figure 3.2: Using a main block to control code execution when importing modules

3.3.4 Group Imports Logically

Standardize the order of imports to make your code more readable and easier to navigate. You can group imports into three categories: standard library, third-party, and local.

The following code demonstrates this well-organized import structure:

```
# Standard library imports
import datetime
import json
import os
import sys

# Third-party imports
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Local application imports
from utils import helpers
```

3.3.5 Avoid Circular Imports

Circular imports occur when two or more modules import each other, creating a dependency loop that can cause the program to fail to start or behave unexpectedly.

In the following example, `data_loader.py` and `data_processor.py` both import from each other, creating a circular import. This circular import can lead to `ImportError` or other unexpected behavior, making debugging difficult.

Example 3.20: data_loader.py

```
from data_processor import process_data

def load_data():
    # Load data from a CSV file
```

```
    data = pd.read_csv("dataset.csv")
    return process_data(data)
```

Example 3.21: data_processor.py

```
from data_loader import load_data

def process_data(data):
    # Process the data
    processed = data.dropna()
    return processed

def main():
    data = load_data()
    # Further analysis...
```

Instead, restructure your code to break circular imports. `main.py` coordinates `data_loader.py` and `data_processor.py`, eliminating the circular dependencies.

Example 3.22: main.py

```
from data_loader import load_data
from data_processor import process_data

def main():
    raw_data = load_data()
    processed_data = process_data(raw_data)
    # Further analysis...

if __name__ == "__main__":
    main()
```

Example 3.23: data_loader.py

```
import pandas as pd

def load_data():
    # Load data from a CSV file
    return pd.read_csv("dataset.csv")
```

Example 3.24: data_processor.py

```
def process_data(data):
    # Process the data
    return data.dropna()
```

3.4 Key Takeaways

Python modules and packages are essential tools for organizing and structuring your code effectively. Here are the key takeaways from this chapter:

1. Understanding modules and packages:
    - Modules are `.py` files containing reusable code (variables, functions, classes)
    - Packages are directories containing related modules
2. Project organization best practices:
    - Break down large files into smaller, focused modules
    - Use a standardized project structure (`src/`, `tests/`, `data/`, etc.)
    - Keep related functionality together in the same module
    - Use clear, descriptive names for modules and packages
3. Import best practices:
    - Avoid wildcard imports (`from module import *`)
    - Use explicit imports to prevent naming conflicts
    - Group imports logically (standard library, third-party, local)
    - Use absolute imports instead of relative imports
    - Avoid circular dependencies between modules
4. Code execution control:
    - Use `if __name__ == "__main__":` to control script execution
    - Separate reusable code from script-specific code
    - Prevent unintended code execution during imports
    - Make modules both importable and executable

By following these practices, you can create well-organized, maintainable Python projects that are easier to understand, test, and

collaborate on.

4 Python Variables

4.1 What Are Variables?

Variables are used to store data in a program. To define a variable in Python, use the assignment operator = with the variable name on the left and the value on the right.

Here are examples of different types of values that variables can store:

```
# Numbers
age = 25

# Strings
name = "Alice"

# Lists
fruits = ["apple", "banana", "orange"]

# Dictionaries
student = {"name": "Bob", "age": 20, "grades": [85, 90, 88]}

# Sets
unique_numbers = {1, 2, 3, 4, 5}

# Tuples
coordinates = (10, 20)
```

4.2 Choose the Right Python Collection

Python collections are built-in data structures that allow you to store multiple items in a single variable. Python provides several collection types, each with unique characteristics that make each one suitable for different coding scenarios. Understanding the differences between each type of collection helps you choose the most appropriate collection for your specific needs.

4.2.1 Lists: Ordered, Mutable Sequences

Use lists when you need:

- An ordered sequence of items
- To modify the collection (add, remove, or change items)
- To access elements by their position
- To allow duplicate elements

Tracking a sequence of data points in a time series is a good use case for a list because it is ordered, can be modified, and allows duplicate elements.

```
# List of daily temperatures
temperatures = [72.5, 73.1, 71.8, 74.2, 73.5]

# Add a new temperature
temperatures.append(72.9)

# Access temperature by position
print(f"Temperature on day 3: {temperatures[2]}°F")

# Modify a temperature
temperatures[0] = 73.0
```

```
Temperature on day 3: 71.8°F
```

4.2.2 Tuples: Ordered, Immutable Sequences

Use tuples when you need:

- An ordered sequence that cannot be changed
- To ensure data integrity
- To use the collection as a dictionary key
- To allow duplicate elements

Storing coordinates is a good use case for a tuple because it is ordered, cannot be changed, and might be used as a dictionary key.

```
# Tuple of coordinates
point = (10, 20)

# Can't modify coordinates
try:
    point[0] = 15  # This will raise an error
except TypeError as e:
    print(f"Error: {e}")

# Can be used as a dictionary key
points = {
    (10, 20): "Point A",
    (30, 40): "Point B",
}
```

```
Error: 'tuple' object does not support item assignment
```

4.2.3 Sets: Unordered, Mutable Collections of Unique Elements

Use sets when you need:

- To store unique elements
- Fast membership testing

- To perform set operations (union, intersection, difference)
- To remove duplicates from a sequence

Tracking unique visitors to a website is a good use case for a set because it is unordered, can be modified, and does not allow duplicate elements.

```
# Set of unique visitors
visitors = {"user1", "user2", "user3"}

# Add a new visitor
visitors.add("user4")

# Check if a user has visited
print(f"Has user1 visited?: {'user1' in visitors}")

# Remove duplicates from a list
all_visits = ["user1", "user2", "user1", "user3", "user2"]
unique_visitors = set(all_visits)
print(f"Unique visitors: {unique_visitors}")
```

```
Has user1 visited?: True
Unique visitors: {'user2', 'user3', 'user1'}
```

4.2.4 Dictionaries: Key-Value Pairs

Use dictionaries when you need:

- To associate values with unique keys
- Fast lookups by key
- To store related data together
- To count occurrences of items

Example: Storing user information:

```
# Dictionary of user information
user = {
    "name": "John",
    "age": 30,
```

```
    "email": "john@example.com"
}

# Access information by key
print(f"User's name: {user['name']}")

# Update information
user["age"] = 31

# Add new information
user["location"] = "New York"
```

```
User's name: John
```

Storing user information is a good use case for a dictionary because the data is unordered, can be modified, and allows lookup by key.

This table summarizes the key characteristics of each collection type:

Collection	Ordered	Mutable	Duplicates	Key-Value
List	Yes	Yes	Yes	No
Tuple	Yes	No	Yes	No
Set	No	Yes	No	No
Dictionary	No	Yes	No (keys)	Yes

4.3 Best Practices for Python Variables

4.3.1 Use Descriptive Variable Names

Use descriptive variable names that clearly indicate the purpose or content of each variable. Descriptive names make your code self-documenting and reduce the need for extensive comments.

This example from the Kaggle notebook *How I made top 0.3% on a Kaggle competition* demonstrates the problem: variables `ls`, `m`, and `l` require extensive code analysis to understand their meaning.

```
def squares(df, ls):
    m = df.shape[1]
    for l in ls:
        df = df.assign(newcol=pd.Series(df[l]*df[l]).values)
        df.columns.values[m] = l + '_sq'
        m += 1
    return df

squared_features = ['LotFrontage_log',  'GrLivArea_log']

df = squares(df, squared_features)
```

To improve readability, use descriptive variable names:

```
def calculate_squared_features(df, feature_list):
    column_count = df.shape[1]
    for feature_name in feature_list:
        squared_feature = df[feature_name]**2
        df = df.assign(newcol=squared_feature)
        df.columns.values[column_count] = feature_name + '_sq'
        column_count += 1
    return df

features_to_square = ['LotFrontage_log', 'GrLivArea_log']

squared_df = calculate_squared_features(df, features_to_square)
```

Line 1
`ls` becomes `feature_list` to indicate the list of features to be squared.
Line 2
`m` becomes `column_count` to indicate the number of columns in the DataFrame.
Line 3
`l` becomes `feature_name` to indicate the name of the feature to be squared.
Line 4

`df[feature_name]**2` is assigned to `squared_feature` to make the code more readable.
Line 12
The return value is assigned to `squared_df` to indicate the name of the DataFrame that contains the squared features.

Why use `df` as the name of a DataFrame?

While we generally encourage descriptive variable names, using `df` is considered good practice because it is a well-established convention that most data scientists recognize.

If you need to work with multiple DataFrames, you can add descriptive prefixes or suffixes (e.g., `raw_df`, `cleaned_df`, `sales_df`) to distinguish between them.

4.3.2 Avoid Reserved Keywords

Python has a set of reserved keywords that have special meaning in the language and cannot be used as variable or function names. These include:

- Control flow keywords: `if`, `else`, `elif`, `for`, `while`, `break`, `continue`, `return`
- Definition keywords: `def`, `class`, `lambda`
- Logical keywords: `and`, `or`, `not`, `is`, `in`
- Value keywords: `True`, `False`, `None`
- Other keywords: `import`, `from`, `as`, `try`, `except`, `finally`, `raise`, `with`, `global`, `nonlocal`, `assert`, `del`, `pass`, `yield`

Avoid using Python's reserved keywords as variable names to prevent syntax errors.

In the following example, `class` is a reserved keyword in Python. Using it as variable name causes syntax errors.

```
import pandas as pd
```

```
# Problematic use of reserved keywords
class = pd.read_csv('data/classes.csv')
```

```
SyntaxError: invalid syntax
```

To fix these issues, use descriptive names that are not reserved keywords:

```
import pandas as pd

class_df = pd.read_csv('data/classes.csv')
```

4.3.3 Use Uppercase for Constants

A constant is like a variable, but is used to store values that don't change while a program runs. For example, `PI = 3.14159` represents a mathematical constant that never changes during program execution.

Use uppercase names for constants to distinguish them from regular variables and improve code maintainability. This naming convention signals that values shouldn't change during program execution.

The following example demonstrates why hard-coded values like `0.01`, `1000000`, and `30` cause maintenance issues:

- The values provide no context about their purpose.
- Modifying the interest rate from `0.01` to `0.02` forces you to locate and replace every instance manually. This is error-prone and time-consuming.

```
def calculate_loan_payment(principal, years):
    num_payments = years * 12
```

```
    if principal > 1000000:
        return "Loan amount too high"
    if years > 30:
        return "Term too long"
    return (
```

```
        principal
        * (0.01 * (1 + 0.01) ** num_payments)
        / ((1 + 0.01) ** num_payments - 1)
    )
```

To improve readability and maintainability, use meaningful uppercase names instead of hard-coded values:

```
MONTHLY_INTEREST_RATE = 0.01
GROWTH_FACTOR = 1 + MONTHLY_INTEREST_RATE
MAX_LOAN_AMOUNT = 1000000
MAX_LOAN_TERM = 30

def calculate_loan_payment(principal, years):
    num_payments = years * 12
    if principal > MAX_LOAN_AMOUNT:
        return "Loan amount too high"
    if years > MAX_LOAN_TERM:
        return "Term too long"
    return (
        principal
        * (MONTHLY_INTEREST_RATE * GROWTH_FACTOR**num_payments)
        / (GROWTH_FACTOR**num_payments - 1)
    )
```

Can You Modify Constants?

While constants should not change during program execution, they can be modified **before** the program starts running.

For example, you can change the values of `MAX_LOAN_AMOUNT` before the program starts running:

```
MAX_LOAN_AMOUNT = 2_000_000 # Changed from 1_000_000
```

4.3.4 Use Plural Nouns for Collections

Use plural nouns for collections to clearly indicate they contain multiple items.

This example shows how singular naming confuses developers. The variable `city` contains multiple cities, but its name implies a single value, resulting in the flawed logic `city == 'New York'` that compares a list to a string.

```
# Problematic use of singular nouns for collections
city = ['New York', 'London', 'Tokyo', 'Paris', 'Sydney']

if city == 'New York':
    # This won't work as expected if city is a list
    print("Big Apple!")
```

Replace singular names with plural forms to eliminate confusion. The name `cities` signals multiple values, making the logic `'New York' in cities` self-explanatory.

```
# Improved code using plural nouns for collections
cities = ['New York', 'London', 'Tokyo', 'Paris', 'Sydney']

if 'New York' in cities:
    print("Big Apple!")
```

```
Big Apple!
```

4.3.5 Name Slice Indices

Use descriptive names for slice indices to make your code more readable and eliminate confusion about what specific index ranges represent.

The slice indices `4` and `4:` in this price analysis example lack context, making it difficult to understand what data ranges they represent. d

```
prices = [5, 3, 5, 4, 5, 3, 3.5, 3]

price_diff = sum(prices[:4]) - sum(prices[4:])
print(price_diff)
```

```
2.5
```

To improve readability, replace numeric indices with descriptive slice objects using the `slice` function. This example creates `JANUARY` and `FEBRUARY` variables to represent specific data ranges.

```
prices = [5, 3, 5, 4, 5, 3, 3.5, 3]

# Create slice objects to represent specific ranges
JANUARY = slice(0, 4)  # First 4 elements
FEBRUARY = slice(4, len(prices))  # Remaining elements

price_diff = sum(prices[JANUARY]) - sum(prices[FEBRUARY])
print(f"Price difference between January and February:
{price_diff}")
```

```
Price difference between January and February: 2.5
```

4.3.6 Use Underscore for Throwaway Variables

Use underscores for unused variables to keep code clean and signal you're ignoring them.

In the following example, `filename` is a throwaway variable that is not used.

```
import os

full_path = '/home/user/data/project/data.csv'

# Split the path into directory and filename
directory, filename = os.path.split(full_path)

# Add a new file to directory
new_file_name = 'new_data.csv'
new_file_path = os.path.join(directory, new_file_name)

print(new_file_path)
```

```
/home/user/data/project/new_data.csv
```

To signal that this value is intentionally ignored, use the underscore (_) instead of `filename`.

```
import os

full_path = '/home/user/data/project/data.csv'

# Split the path into directory and filename
directory, _ = os.path.split(full_path)

# Add a new file to directory
new_file_name = 'new_data.csv'
new_file_path = os.path.join(directory, new_file_name)

print(new_file_path)
```

```
/home/user/data/project/new_data.csv
```

4.3.7 Signal Private Variables with Underscores

Use underscore prefixes to mark variables as private and prevent external access. This Python convention tells other developers the variable is for internal use only.

This bank account example shows the problem: without privacy signals, other developers can directly modify the balance, creating security risks.

```
class Bank:
    def __init__(self, account_number, initial_balance):
        self.account_number = account_number
        self.balance = initial_balance

# Bad: Accessing the balance directly from outside the class
bank_account = Bank("123456789", 200)
print(f"Initial balance: {bank_account.balance}")

# Bad: Modifying the balance directly from outside the class
```

```
bank_account.balance += 500
print(f"Balance after deposit: {bank_account.balance}")
```

```
Initial balance: 200
Balance after deposit: 700
```

Use underscore prefix (_) to mark the balance as private and provide access through methods instead of direct access.

```
class Bank:
    def __init__(self, account_number, initial_balance):
        self.account_number = account_number
        self._balance = initial_balance  # Private variable

    def get_balance(self):
        return self._balance


bank_account = Bank("123456789", 200)
print(f"Initial balance: {bank_account.get_balance()}")
```

```
Initial balance: 200
```

4.3.8 Avoid Variable Repurposing

Use distinct variable names for different data states to prevent confusion. Repurposing variables causes these issues:

1. **Loss of History**: You can't easily see what the data looked like at each step
2. **Debugging Difficulty**: If something goes wrong, it's hard to identify which transformation caused the issue
3. **Code Readability**: The meaning of the variable changes throughout the code, making it harder to understand
4. **Accidental Modifications**: You might accidentally use the wrong version of the variable

This example shows how repurposing `df` creates confusion about what data the variable contains at each transformation step.

```
import pandas as pd

# Problematic code with variable repurposing
df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df = df.assign(c=lambda x: x["a"] + x["b"])  # has a new column
df = df[df["c"] > 5]  # df now has filtered rows
df = df.drop("b", axis=1)  # df now has different columns

# What's in df now? It's hard to tell without checking each step
```

To clearly show what data each step produces, use distinct variables for each transformation.

```
import pandas as pd

# Improved code with distinct variables
original_df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df_with_sum = original_df.assign(c=lambda x: x["a"] + x["b"])
filtered_df = df_with_sum[df_with_sum["c"] > 5]
final_df = filtered_df.drop("b", axis=1)
```

4.4 Key Takeaways

1. Choose the right collection type:
    - Use lists for ordered, mutable sequences
    - Use tuples for immutable sequences
    - Use sets for unique elements and fast membership testing
    - Use dictionaries for key-value pairs
2. Best practices:
    - Use descriptive names that clearly indicate purpose
    - Avoid reserved keywords
    - Use uppercase for constants
    - Use plural nouns for collections
    - Use underscore for throwaway variables
    - Prefix private variables with underscore

- Use distinct variables for different states

5 Python Functions

5.1 What Are Python Functions?

Functions are reusable blocks of code that perform specific tasks. They help you organize your code, make it more readable, and promote code reuse.

To define a function in Python, use the `def` keyword followed by the function name and parentheses. The function body is indented below.

Here's an example of a function commonly used in data science:

```
def standardize_features(X):
    """Standardize the features in the input data."""
    X_standardized = (X - X.mean()) / X.std()
    return X_standardized
```

5.2 Why Are Python Functions Essential?

5.2.1

Functions streamline development by turning repetitive tasks into single, reusable calls. Reusable functions cut coding time, reduce copy-paste errors, and centralize logic for easier updates.

In the following example, the same preprocessing logic is repeated for both the training and test sets, forcing you to update the same logic in multiple places whenever you modify the preprocessing steps.

```
import numpy as np

X_train = np.array([5, 10, 15, 20, 25])
X_test = np.array([8, 12, 18, 22, 28])

X_train_standardized = (X_train - X_train.mean()) / X_train.std()
X_test_standardized = (X_test - X_train.mean()) / X_train.std()
```

To avoid this problem, encapsulate the data preprocessing logic in a function and reuse it across your data science project:

```
def standardize_features(X):
    return (X - X.mean()) / X.std()

X_train = np.array([5, 10, 15, 20, 25])
X_test = np.array([8, 12, 18, 22, 28])

X_train_standardized = standardize_features(X_train)
X_test_standardized = standardize_features(X_test)
```

5.2.2 Improve Code Readability

Functions improve code readability by replacing complex inline operations with descriptive function names.

The following example demonstrates the readability problem: complex data processing steps buried under verbose comments that make the core logic harder to follow.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestRegressor

# Load the data from a CSV file
```

```
data = pd.read_csv("data/dataset.csv")

# Handle missing values by filling them with 0
data = data.fillna(0)

# Encode the categorical variable 'category'
data["category"] = data["category"].map(
    {"A": 0, "B": 1, "C": 2}
)

# Using the StandardScaler to standardize the data
scaler = StandardScaler()
data["feature1"] = scaler.fit_transform(
    data["feature1"].values.reshape(-1, 1)
)
```

By breaking data processing into named functions, each step becomes self-documenting with a clear purpose, eliminating verbose comments and making the workflow immediately understandable.

```
def load_data():
    ...
    return data

def handle_missing_values(data):
    ...
    return data

def encode_categorical_variables(data):
    ...
    return data

def scale_numeric_features(data, features_to_scale):
    ...
    return data

def process_data(data, categorical_columns, numeric_columns):
    data = handle_missing_values(data)
    data = encode_categorical_variables(data)
    data = scale_numeric_features(data, numeric_columns)
    return data
```

5.2.3 Hide Implementation Details

Functions hide implementation details, keeping your code focused on high-level logic rather than technical specifics.

The following code exposes database implementation details, making readers examine connection strings and INSERT statements to understand the function's purpose.

```
import sqlite3

# Save a new user
username = "john_doe"
email = "john@example.com"

with sqlite3.connect("data/users.db") as conn:
    try:
        cursor = conn.cursor()

        cursor.execute(
            "INSERT INTO users (username, email) VALUES (?, ?)",
            (username, email),
        )
        conn.commit()
        print("User saved successfully")
    except sqlite3.Error:
        print("Failed to save user")
```

Use a function like `save_user` to encapsulate database operations, letting users save data without knowing how the database connection works.

```
import sqlite3



def save_user(username, email):
    with sqlite3.connect("data/users.db") as conn:
        try:
            cursor = conn.cursor()
            cursor.execute(
```

```
                "INSERT INTO users (username, email) VALUES (?,
?)",
                (username, email),
            )
            conn.commit()
            print("User saved successfully")
        except sqlite3.Error:
            print("Failed to save user")

# Users can call this function without understanding database
details
save_user("john_doe", "john@example.com")
```

```
User saved successfully
```

5.3 Best Practices for Python Functions

5.3.1 Use Descriptive Verb-Based Names

Apply snake_case, descriptive, verb-based function names that reflect their purpose to enhance code clarity and standardize your codebase. Snake_case is a naming convention in which spaces are replaced with underscores and all letters are lowercase.

In the following code, function names are inconsistent and don't clearly indicate their actions.

```
def data_clean(df):
    return df.dropna()



def transform(s):
    return np.log(s)



def above_mean(df, column):
    return df[df[column] > df[column].mean()]
```

To improve these unclear function names, use descriptive, verb-based function names in snake_case that reflect their purpose. That approach makes the code more self-documenting and easier to maintain.

```
def remove_missing_values(df):
    return df.dropna()


def apply_log_transformation(s):
    return np.log(s)


def filter_values_above_mean(df, column):
    return df[df[column] > df[column].mean()]
```

5.3.2 Keep Functions Focused

Use small, single-purpose functions to make debugging easier, reduce code duplication, and enhance reusability while simplifying modifications to your codebase.

Consider this example of a large, multi-purpose function that performs multiple operations, making it hard to isolate bugs and modify individual steps.

```
from sklearn.preprocessing import StandardScaler


def process_sales_data(df):
    # Remove missing values
    df = df.dropna()

    # Log transform sales
    df["log_sales"] = np.log1p(df["Sales"])

    # Encode categorical variables
    df = pd.get_dummies(df, columns=["Category", "Region"])
```

```
    # Normalize numeric features
    scaler = StandardScaler()
    num_columns = ["Sales", "Quantity"]
    df[num_columns] = scaler.fit_transform(df[num_columns])

    return df
```

To solve these problems, break the function into smaller, focused functions:

```
def remove_missing_values(df):
    return df.dropna()



def log_transform_sales(df):
    return df.assign(
        log_sales=lambda x: np.log1p(x["Sales"])
    )



def encode_categorical_variables(df, cat_columns):
    return pd.get_dummies(df, columns=cat_columns)



def normalize_numeric_features(df, num_columns):
    scaler = StandardScaler()
    df = df.copy()
    df[num_columns] = scaler.fit_transform(df[num_columns])
    return df
```

Then chain these functions using the `pipe` method in `process_sales_data`. Each function handles one operation while the main function coordinates the workflow.

```
def process_sales_data(df):
    return (
        df.pipe(remove_missing_values)
        .pipe(log_transform_sales)
        .pipe(
            encode_categorical_variables,
```

```
            cat_columns=["Category", "Region"],
        )
        .pipe(
            normalize_numeric_features,
            num_columns=["Sales", "Quantity"],
        )
    )
```

5.3.3 Use Type Hints

Enhance your Python functions with type hints to prevent runtime errors and enable better IDE support with autocomplete and error detection.

In the following example, the `calculate_average_rating` function lacks type hints, making it unclear what types the parameters expect or what the function returns:

```
def calculate_average_rating(ratings, product_id):
    product_ratings = [
        r for r in ratings if r["product_id"] == product_id
    ]
    if not product_ratings:
        return None
    total_score = sum(r["score"] for r in product_ratings)
    return total_score / len(product_ratings)
```

Improve the function by adding type hints that specify:

- `ratings` as a list of dictionaries with integer values
- `product_id` as an integer
- the return type as either a float or None

```
def calculate_average_rating(
    ratings: list[dict[str, int]], product_id: int
) -> float | None:


   ...
```

5.3.4 Write Clear and Helpful Docstrings

Docstrings are a simple way to add helpful documentation to your Python code. Docstrings are valuable for documenting complex behavior, parameters, return values, and examples that aren't immediately obvious from the function signature alone.

Consider this function without a docstring. The function has a descriptive name and type hints, but critical details remain unclear from the function signature:

- The expected format of the `text` parameter (pipe-delimited key-value pairs)
- The structure of the returned dictionary (keys from even indices, values from odd indices)

```
def parse_pipe_delimited_text(text: str) -> dict:
    parts = text.split("|")
    if len(parts) % 2 != 0:
        raise ValueError(
            "Input string must have an even number of parts"
        )
    return {parts[i]: parts[i + 1] for i in range(0, len(parts),
2)}
```

Add a docstring to explain how the function works. The docstring provides a clear description of the function's purpose, parameter types, return value, potential exceptions, and usage example.

```
def parse_pipe_delimited_text(text: str) -> dict:
    """
    Parse a pipe-delimited string into a dictionary.

    Parameters
    ----------
    text: str
        A pipe-delimited string to parse

    Returns
    -------
```

```
    dict
        Dictionary with even indices as keys, odd indices as values

    Raises
    ------
    ValueError: If the input string has an odd number of parts

    Examples
    --------
    >>> parse_pipe_delimited_text("name|John|age|30")
    {'name': 'John', 'age': '30'}
    """
    parts = text.split("|")
    if len(parts) % 2 != 0:
        raise ValueError(
            "Input string must have an even number of parts"
        )
    return {parts[i]: parts[i + 1] for i in range(0, len(parts),
2)}
```

Don't Overuse Docstrings

For simple functions with clear names and type hints, extensive docstrings can be unnecessary and even distracting. For example, the `square` function below is a simple function with a clear name and type hint. Thus, the docstring is unnecessary.

```
def square(x: float) -> float:
    return x * x
```

Documentation isn't a substitute for writing unclear code. Focus on writing clear, self-documenting code first, then add docstrings only when they provide additional value.

5.3.5 Use Function Parameters Instead of Global Variables

Use function parameters to pass necessary data instead of relying on global variables. Global variables make your code harder to read and can produce unexpected results.

Consider this example that relies on global variables taken from the Kaggle notebook *How I made top 0.3% on a Kaggle competition*:

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score
import numpy as np

X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
train_labels = np.array([2, 4, 5, 4, 5])

kf = KFold(n_splits=5, random_state=42, shuffle=True)

model = LinearRegression()

def cv_rmse(model, X):
    return np.sqrt(
        -cross_val_score(
            model, X, train_labels,
            scoring="neg_mean_squared_error", cv=kf
        )
    )

# Calculate RMSE scores
scores = cv_rmse(model=model, X=X)
print(f"Mean RMSE: {scores.mean():.3f}")
```

```
Mean RMSE: 1.093
```

Using global variables like this can lead to several issues:

1. **Unpredictable Behavior**: The function's output depends on an external state that can be modified anywhere in the code, making it hard to predict the function's behavior.

    ```
    # Change the global variable
    kf = KFold(n_splits=2, random_state=42, shuffle=True)

    # The function's output will change
    scores = cv_rmse(model=model, X=X)
    print(f"Mean RMSE: {scores.mean():.3f}")
    ```

    ```
    Mean RMSE: 1.393
    ```

2. **Testing Difficulties**: Tests must set up global state first and can interfere with each other.

3. **Reusability**: The function is hard to reuse in different contexts because it is tightly coupled to a global variable

4. **Maintainability Issues**: Changes to global variables require carefully checking all dependent code to avoid breaking functionality

5. **Poor Readability**: Functions using global state fail to clearly communicate their dependencies through their interfaces

    ```
    def cv_rmse(model, X):
        rmse = np.sqrt(
            -cross_val_score(
                model, X,
                train_labels, # What is this?
                scoring="neg_mean_squared_error",
                cv=kf, # What is this?
            )
        )
    return rmse
    ```

6. **Debugging Challenges**: When bugs occur, tracking down which part of the code modified the global state becomes time-consuming

To avoid these issues, pass global variables as parameters instead:

```
def cv_rmse(model, X, train_labels, kf):
    rmse = np.sqrt(
        -cross_val_score(
            model, X, train_labels,
            scoring="neg_mean_squared_error", cv=kf
        )
    )
    return rmse
```

5.3.6 Avoid Modifying Input Parameters

Modifying input parameters directly can corrupt the original data and cause unexpected behavior in other parts of your code.

Consider the following example where we modify a DataFrame in-place by overwriting the specified columns with normalized values:

```
def normalize_data(df: pd.DataFrame, columns: list) ->
pd.DataFrame:
    df[columns] = (
        df[columns] - df[columns].mean()
    ) / df[columns].std()
    return df
```

Calling `normalize_data` overwrites the original DataFrame values, making the original data unavailable for subsequent analysis steps.

```
data = pd.DataFrame(
    {
        "temperature": [25.5, 27.8, 23.2],
        "humidity": [60.0, 55.5, 62.3],
        "pressure": [1013.2, 1015.7, 1012.8],
    }
)

normalized_data = normalize_data(data, columns=["humidity"])
print(f"Original data:\n{data.head()}")
```

```
Original data:
   temperature  humidity  pressure
0         25.5  0.212019    1013.2
1         27.8 -1.089008    1015.7
2         23.2  0.876989    1012.8
```

Instead, create a copy of the input data with `df.copy()` and return the new DataFrame to avoid modifying the original data:

```
def normalize_data(df: pd.DataFrame, columns: list) ->
pd.DataFrame:
    df = df.copy()
    df[columns] = (
        df[columns] - df[columns].mean()
```

```
    ) / df[columns].std()
    return df

data = pd.DataFrame(
    {
        "temperature": [25.5, 27.8, 23.2],
        "humidity": [60.0, 55.5, 62.3],
        "pressure": [1013.2, 1015.7, 1012.8],
    }
)
normalized_data = normalize_data(data, columns=["humidity"])
print(f"Original data:\n{data}")
```

```
Original data:
   temperature  humidity  pressure
0         25.5      60.0    1013.2
1         27.8      55.5    1015.7
2         23.2      62.3    1012.8
```

5.3.7 Avoid Using Flags As Parameters

Create separate, purpose-specific functions instead of using boolean flags as parameters. Boolean flags increase complexity and make it difficult to understand what the function does without examining the implementation.

Consider this example of a data preprocessing function with boolean flags:

```
def preprocess_data(
    df: pd.DataFrame,
    fill_missing: bool = False,
    normalize: bool = False,
) -> pd.DataFrame:

    if fill_missing:
        df = df.fillna(df.mean())

    if normalize:
        df = (df - df.mean()) / df.std()
```

```
    return df
```

To fill missing values only, you can set `fill_missing` to `True` and `normalize` to `False`.

```
df = pd.read_csv("data/sample.csv")
cleaned_df = preprocess_data(df, fill_missing=True,
normalize=False)
```

This function may seem convenient at first, but a function like this quickly becomes harder to maintain as you add more functionality. For instance, introducing a new preprocessing step, like removing outliers, would require adding yet another boolean flag.

```
def preprocess_data(
    df: pd.DataFrame,
    fill_missing: bool = False,
    normalize: bool = False,
    remove_outliers: bool = False  # New flag
) -> pd.DataFrame:
    if fill_missing:
        df = df.fillna(df.mean())
    if normalize:
        df = (df - df.mean()) / df.std()
    if remove_outliers:  # New condition
        df = df[df.apply(lambda x: abs(x - x.mean()) <= 3 *
x.std())]
    return df
```

Instead, create separate functions for each preprocessing task and compose them as needed. Start with splitting the `preprocess_data` function into smaller functions.

```
import pandas as pd
import numpy as np

def fill_missing_values(df: pd.DataFrame) -> pd.DataFrame:
    return df.fillna(df.mean())

def normalize_data(df: pd.DataFrame) -> pd.DataFrame:
    return (df - df.mean()) / df.std()
```

```
def preprocess_data(df: pd.DataFrame, steps: list) -> pd.DataFrame:
    for step in steps:
        df = step(df)
    return df
```

Now, you can create a pipeline by passing functions as a list to `preprocess_data`.

```
df = pd.read_csv("data/sample.csv")
cleaning_steps = [normalize_data, fill_missing_values]
cleaned_df = preprocess_data(df, cleaning_steps)
```

5.3.8 Extract Common Logic Into Utilities

Create reusable utility functions to eliminate code duplication and ensure consistent behavior when you need to modify shared logic.

The following functions repeat the same text cleaning steps: converting to lowercase, removing special characters, and trimming whitespace:

```
def clean_text_data(df: pd.DataFrame) -> pd.DataFrame:
    df["text"] = df["text"].str.lower()
    df["text"] = df["text"].str.replace(
        "[^a-zA-Z\s]", "", regex=True
    )
    df["text"] = df["text"].str.strip()
    return df

def preprocess_user_input(text: str) -> str:
    text = text.lower()
    text = "".join(
        char for char in text if char.isalnum() or char.isspace()
    )
    text = text.strip()
    return text
```

Although `clean_text_data` processes DataFrames and `preprocess_user_input` handles strings, both functions duplicate the

same cleaning logic. Modifying the cleaning approach requires updating both functions, risking inconsistencies if one gets missed.

```
df = pd.read_csv("data/comments.csv")
cleaned_df = clean_text_data(df)
user_input = "Hello, World! 123"
cleaned_input = preprocess_user_input(user_input)
```

To eliminate code duplication, create a reusable `clean_text` utility function that both `clean_text_data` and `preprocess_user_input` can use.

```
def clean_text(text: str) -> str:
    text = text.lower()
    text = "".join(
        char for char in text if char.isalnum() or char.isspace()
    )
    return text.strip()


def clean_text_data(df: pd.DataFrame) -> pd.DataFrame:
    df["text"] = df["text"].apply(clean_text)
    return df


def preprocess_user_input(text: str) -> str:
    return clean_text(text)
```

5.4 Advanced Function Toolkit

5.4.1 Lambda Functions

Lambda functions in Python are small, anonymous functions defined using the `lambda` keyword, allowing for the creation of one-line function objects.

Use `lambda` functions when you need a simple function for a short period, particularly in higher-order functions like `map()`, `filter()`, or

as arguments to other functions.

Here are some examples of using `lambda` functions:

1. Using `lambda` with `map()` to square numbers in a list:

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, numbers))
print(squared)
```

```
[1, 4, 9, 16, 25]
```

2. Using `lambda` with `filter()` to get even numbers from a list:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

```
[2, 4, 6, 8, 10]
```

3. Using `lambda` in sorting a list of tuples based on the second element:

```
data = [('Alice', 25), ('Bob', 30), ('Charlie', 22)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
```

```
[('Charlie', 22), ('Alice', 25), ('Bob', 30)]
```

5.4.2 Partial Functions

A partial function is a technique from `functools.partial` that allows you to create a new function by fixing some arguments of an existing function. Use partial functions when you want to reduce the number of required arguments for function calls.

For example, if you frequently use the `cv_rmse` function with the same dataset and cross-validation strategy, you can create a partial

function called `cv_rmse_with_data` that fixes the dataset and cross-validation strategy.

```
from functools import partial

# Create sample data
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
train_labels = np.array([2, 4, 5, 4, 5])
kf = KFold(n_splits=5, random_state=42, shuffle=True)

# Create a partial function with fixed X, train_labels, and kf
cv_rmse_with_data = partial(
    cv_rmse, X=X, train_labels=train_labels, kf=kf
)
```

You can now easily apply consistent cross-validation across different models without repeating the same parameters.

```
# Call the partial function with Linear Regression
linear_scores = cv_rmse_with_data(model=LinearRegression())

# Call the partial function with Ridge Regression
ridge_scores = cv_rmse_with_data(model=Ridge(alpha=0.5))
```

5.4.3 `*args` and `**kwargs`

`*args` and `**kwargs` are special syntax in Python used to pass a variable number of arguments to a function.

Use `*args` to pass a variable number of non-keyword arguments to a function. In the example below, `*transformers` allows `transform_pipeline` to accept multiple transformation functions.

```
import numpy as np
from typing import Callable



def transform_pipeline(
    data: np.ndarray, *transformers: Callable
```

```
) -> np.ndarray:
    for transformer in transformers:
        data = transformer(data)
    return data
```

The flexible design lets you add `log_transform`, `standardize`, or any number of transformation functions without changing `transform_pipeline`.

```
def log_transform(data: np.ndarray) -> np.ndarray:
    return np.log1p(data)



def standardize(data: np.ndarray) -> np.ndarray:
    return (data - data.mean()) / data.std()



raw_data = np.random.rand(100, 5) * 100

transformed_data = transform_pipeline(
    raw_data, log_transform, standardize
)
```

Use `**kwargs` to pass a variable number of keyword arguments to a function. In the example below, `**transformers` allows `transform_pipeline` to accept any number of named transformation functions.

```
import numpy as np
from typing import Callable



def transform_pipeline(
    data: np.ndarray, **transformers: dict[str, Callable]
) -> np.ndarray:
    for transformer_name, transformer_func in transformers.items():
        if not callable(transformer_func):
            raise ValueError(
                f"{transformer_name} is not callable"
            )
```

```
        data = transformer_func(data)
    return data


def log_transform(data: np.ndarray) -> np.ndarray:
    return np.log1p(data)


def standardize(data: np.ndarray) -> np.ndarray:
    return (data - data.mean()) / data.std()
```

This design lets you pass transformation functions by name without changing the function signature.

```
raw_data = np.random.rand(100, 5) * 100

transformed_data = transform_pipeline(
    raw_data,
    log_transform=log_transform,
    standardize=standardize,
)
```

Be Careful with `*args` and `**kwargs`

While `*args` and `**kwargs` provide flexibility, they can also make your code harder to understand and maintain:

- **Type Safety**: Type hints become less specific, making it harder to catch type-related errors
- **Documentation**: It's harder to document all possible arguments and their types
- **IDE Support**: Code completion and documentation features may not work as well
- **Debugging**: Errors can be harder to track down when arguments are passed through multiple layers

Use them sparingly and only when:

- You need to create wrapper functions that pass through arguments
- You're building flexible APIs that need to handle varying numbers of arguments
- You're implementing decorators or other metaprogramming features

For most cases, explicitly declared parameters are clearer and safer.

5.4.4 Python Decorators in Data Science

A decorator is a design pattern in Python that allows you to add new functionality to an existing object without modifying its structure.

Use decorators when you want to extend or modify the behavior of functions or methods without changing their source code. For example, you can create a decorator to add timing functionality to a function:

```
import time
from typing import Callable, List, Union
import numpy as np



def timer_decorator(func: Callable) -> Callable:
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(
            f"Function {func.__name__} took "
            f"{end_time - start_time:.2f} seconds to execute."
        )
        return result

    return wrapper
```

To apply the decorator to a function, simply place the decorator above the function definition.

```
@timer_decorator
def train_model(X: np.ndarray, y: np.ndarray | list[float]) ->
None:
    """Simulating a time-consuming model training process"""
    time.sleep(2)



if __name__ == "__main__":
    X = np.random.rand(1000, 10)
```

```
    y = np.random.rand(1000)
    train_model(X, y)
```

```
Function train_model took 2.01 seconds to execute.
```

In this example, the `timer_decorator` wraps the original function, adds timing functionality, and prints the execution time. When `train_model` is called, it automatically includes the timing feature without modifying the original function's code.

Notice that when executed, this code replaces `train_model` function's name with "wrapper". It also substitutes `train_model`'s docstring and annotations with `wrapper`'s empty attributes.

```
print(f"name: {train_model.__name__}")
print(f"doc: {train_model.__doc__}")
print(f"annotations: {train_model.__annotations__}")
```

```
name: wrapper
doc: None
annotations: {}
```

To maintain essential metadata from the original function when creating a decorator, use `functools.wraps`:

```
from functools import wraps


def timer_decorator(func: Callable) -> Callable:
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(
            f"Function {func.__name__} took "
            f"{end_time - start_time:.2f} seconds to execute."
        )
        return result

    return wrapper
```

Let's apply the decorator to a function and test the timing functionality.

```
@timer_decorator
def train_model(X: np.ndarray, y: np.ndarray | list[float]) ->
None:
    """Simulating a time-consuming model training process"""
    time.sleep(2)

if __name__ == "__main__":
    X = np.random.rand(1000, 10)
    y = np.random.rand(1000)
    train_model(X, y)

    print(f"name: {train_model.__name__}")
    print(f"doc: {train_model.__doc__}")
    print(f"annotations: {train_model.__annotations__}")
```

```
Function train_model took 2.01 seconds to execute.
name: train_model
doc: Simulating a time-consuming model training process
annotations: {'X': <class 'numpy.ndarray'>,
              'y': numpy.ndarray | list[float],
              'return': None}
```

We can see that the wrapper function now takes on the identity of the wrapped function such as `name`, `doc`, and `annotations`.

5.5 Key Takeaways

1. Function design:

- Use descriptive, verb-based names in snake_case
- Keep functions focused on a single task
- Use type hints to improve code clarity and catch errors early
- Write clear docstrings for complex functions
- Avoid modifying input parameters directly
- Use function parameters instead of global variables
- Avoid boolean flags as parameters

2. Code organization:

- Create reusable utility functions to avoid code duplication
- Compose functions to build complex pipelines

3. Advanced features:

- Use decorators to add functionality without modifying source code
- Use lambda functions for simple, one-line operations
- Use partial functions to create specialized versions of general functions
- Use `*args` and `**kwargs` sparingly and only when necessary

6 Python Classes

6.1 What Are Python Classes?

Classes are blueprints for creating objects in Python. They encapsulate data (attributes) and behavior (methods) that define what an object can store and do. Unlike functions, which perform specific tasks on data that is passed to the function, classes contain data as well as functions that perform operations on that data.

For example, a `IceCream` class might have attributes like `flavor`, and methods like `eat()`. Each ice cream object created from this class would have its own values for these attributes but share the same methods:

```
class IceCream:
    def __init__(self, flavor: str):
        self.flavor = flavor

    def eat(self):
        print(
            f"Eating the {self.flavor} ice cream"
        )



chocolate = IceCream("chocolate")
vanilla = IceCream("vanilla")

chocolate.eat()
vanilla.eat()
```

```
Eating the chocolate ice cream
Eating the vanilla ice cream
```

6.2 Best Practices for Python Classes

6.2.1 Use Descriptive Class Names

Use specific class names to clearly communicate the purpose and functionality of your code. Follow these naming conventions:

- Use PascalCase (capitalize first letter of each word)
- Use nouns or noun phrases
- Be descriptive but concise
- Consider adding a suffix that describes the type (e.g., `Manager`, `Trainer`, `Processor`)

For example:

- Good: `DataProcessor`, `FeatureExtractor`, `ModelTrainer`
- Bad: `process_data`, `extract_features`, `train_model`

6.2.2 Hide Implementation Details

Exposing implementation details like helper methods and internal state as public attributes creates maintenance problems. Users can directly modify these attributes or call internal methods, breaking the class's intended behavior.

For example, the `Standardizer` class below exposes attributes like `mean`, `std`, and helper methods like `calculate_mean()` that users shouldn't access directly.

```
import numpy as np
```

```
class Standardizer:
    def __init__(self, data: np.ndarray) -> None:
        self.data = data
        self.mean = 0
        self.std = 1
        self.is_fitted = False

    def calculate_mean(self) -> None:
        self.mean = np.mean(self.data)

    def calculate_std(self) -> None:
        self.std = np.std(self.data)

    def transform(self) -> np.ndarray:
        return (self.data - self.mean) / self.std
```

The public attributes and helper methods lead to several issues, including:

- Users must know to call `calculate_mean()` and `calculate_std()` before `transform()`
- Internal attributes like `mean` and `std` can be modified directly
- Input `data` can be changed unexpectedly

```
s = Standardizer(np.array([1, 2, 3]))

# Users shouldn't need to call these methods
s.calculate_mean()
s.mean = 10
s.calculate_std()

# Calling transform() will use the wrong mean and std
result = s.transform()
print(f"Unexpected result: {result}")
```

```
Unexpected result: [-11.02270384  -9.79795897  -8.5732141 ]
```

Instead, hide implementation details by prefixing internal attributes and methods with a single underscore (`_data`, `_calculate_mean`).

```
class Standardizer:
    def __init__(self, data: np.ndarray) -> None:
        self._data = data

    def _calculate_mean(self) -> None:
        return np.mean(self._data)

    def _calculate_std(self) -> None:
        return np.std(self._data)

    def transform(self) -> np.ndarray:
        mean_ = self._calculate_mean()
        std = self._calculate_std()
        return (self._data - mean_) / std
```

The private attribute pattern exposes only the essential `transform()` method while protecting internal logic, creating a cleaner API.

```
s = Standardizer(np.array([1, 2, 3]))
result = s.transform()  # Only expose what users need
print(f"Expected result: {result}")
```

```
Expected result: [-1.22474487  0.          1.22474487]
```

6.2.3 Use Abstract Base Classes for Consistent Interfaces

Combine inheritance with abstract base classes for consistent, extensible interfaces. Inheritance enables child classes to inherit attributes and methods from parent classes, while abstract base classes (ABCs) define required blueprints that subclasses must implement.

In the example below, the `MissingValueHandler` and `DuplicateHandler` classes use inconsistent method names (`fill_nulls` and `process_dupes`) for similar data cleaning operations.

```
import pandas as pd
```

```
class MissingValueHandler:
    def fill_nulls(self, data: pd.DataFrame) -> pd.DataFrame:
        return data.fillna(data.mean())


class DuplicateHandler:
    def process_dupes(self, data: pd.DataFrame) -> pd.DataFrame:
        return data.drop_duplicates()
```

This inconsistency forces the `clean_dataset` function to handle each class differently through type checking and if-else statements.

```
def clean_dataset(
    data: pd.DataFrame, cleaners: list
) -> pd.DataFrame:
    for cleaner in cleaners:
        if isinstance(cleaner, MissingValueHandler):
            data = cleaner.fill_nulls(data)
        elif isinstance(cleaner, DuplicateHandler):
            data = cleaner.process_dupes(data)
    return data
```

If a new data cleaning class is added, the `clean_dataset` function needs to be modified to handle it.

```
class OutlierHandler:
    def process_outliers(
        self, data: pd.DataFrame
    ) -> pd.DataFrame:
        mean = data.mean()
        std = data.std()
        z_scores = (data - mean) / std
        return data[z_scores.abs() <= 3]


def clean_dataset(
    data: pd.DataFrame, cleaners: list
) -> pd.DataFrame:
    for cleaner in cleaners:
        if isinstance(cleaner, MissingValueHandler):
            data = cleaner.fill_nulls(data)
```

```
        elif isinstance(cleaner, DuplicateHandler):
            data = cleaner.process_dupes(data)
        # Add new cleaner types here
        elif isinstance(cleaner, OutlierHandler):
            data = cleaner.process_outliers(data)
    return data
```

To fix the inconsistent interface problem above, create a `DataTransformer` abstract base class that defines the `transform` method. Concrete classes like `MissingValueHandler` inherit from `DataTransformer` and implement the required method.

```
from abc import ABC, abstractmethod
import pandas as pd
from typing import List


class DataTransformer(ABC):
    @abstractmethod
    def transform(self, data: pd.DataFrame) -> pd.DataFrame:
        """Transform the input data"""
        pass


class MissingValueHandler(DataTransformer):
    def transform(self, data: pd.DataFrame) -> pd.DataFrame:
        return data.fillna(data.mean())


class DuplicateRemover(DataTransformer):
    def transform(self, data: pd.DataFrame) -> pd.DataFrame:
        return data.drop_duplicates()
```

The `clean_dataset` function processes data by applying a list of transformers in sequence. This design eliminates type checking because each transformer follows the `DataTransformer` interface contract.

```
def clean_dataset(
    data: pd.DataFrame, transformers: List[DataTransformer]
) -> pd.DataFrame:
```

```
    for transformer in transformers:
        data = transformer.transform(data)
    return data


if __name__ == "__main__":
    df = pd.DataFrame({"values": [1, 2, None, 2]})
    transformers = [MissingValueHandler(), DuplicateRemover()]
    clean_df = clean_dataset(df, transformers)
```

6.2.4 Choose Composition Over Inheritance

While inheritance can be useful for modeling “is-a” relationships and sharing common functionality, inherited classes can quickly become complex and difficult to maintain, especially when multiple inheritance is involved.

Let’s look at an example of using inheritance to create a data processing pipeline:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler


class MissingValueHandler:
    def process(self, df: pd.DataFrame) -> pd.DataFrame:
        print("Handle missing values")
        return df.fillna(0)


class FeatureScaler(MissingValueHandler):
    def process(self, df: pd.DataFrame) -> pd.DataFrame:
        df = super().process(df)
        print("Scale numeric features")
        scaler = StandardScaler().set_output(transform="pandas")
        return scaler.fit_transform(df)
```

```
class NumericDataProcessor(FeatureScaler):
    def process(self, df: pd.DataFrame) -> pd.DataFrame:
        df = super().process(df)
        print("Remove duplicates")
        return df.drop_duplicates()
```

Using `NumericDataProcessor` triggers a chain of `super().process()` calls that processes data through the inheritance hierarchy:

- `NumericDataProcessor` inherits from `FeatureScaler`
- `FeatureScaler` inherits from `MissingValueHandler`

```
df = pd.DataFrame(
    {
        "feature1": [10.5, np.nan, 10.5],
        "feature2": [100.0, 200.0, 100.0],
    }
)

processor = NumericDataProcessor()
result = processor.process(df)
print("Result:\n", result)
```

```
Handle missing values
Scale numeric features
Remove duplicates
Result:
    feature1  feature2
0  0.707107 -0.707107
1 -1.414214  1.414214
```

Problems with this implementation:

- The processing steps are locked in a fixed order through the inheritance chain, making it impossible to reorder without changing the class hierarchy.
- Adding a new step in the middle of the process requires restructuring the entire inheritance hierarchy.
- The code becomes harder to test because each class depends on its parent's implementation.

Instead, use composition to create flexible data preprocessing pipelines that can be easily modified and maintained.

To implement the composition pattern, we can define a `DataPipeline` class that takes a list of processing steps and applies them sequentially to transform the input data.

```
import pandas as pd
from sklearn.preprocessing import StandardScaler
import numpy as np
from typing import List



class DataPipeline:
    def __init__(self, steps: List[callable]):
        self.steps = steps

    def process(self, df: pd.DataFrame) -> pd.DataFrame:
        for step in self.steps:
            df = step(df)
        return df
```

Define processing functions as pipeline steps. Each function takes and returns a DataFrame.

```
def handle_missing_values(df: pd.DataFrame) -> pd.DataFrame:
    print("Handle missing values")
    return df.fillna(0)



def scale_features(df: pd.DataFrame) -> pd.DataFrame:
    print("Scale numeric features")
    scaler = StandardScaler().set_output(transform="pandas")
    return scaler.fit_transform(df)



def remove_duplicates(df: pd.DataFrame) -> pd.DataFrame:
    print("Remove duplicates")
    return df.drop_duplicates()
```

Modular functions enable flexible pipeline composition. Create targeted pipelines for specific needs: missing value handling, feature scaling, or full preprocessing workflows.

```
df = pd.DataFrame(
    {
        "feature1": [10.5, np.nan, 10.5],
        "feature2": [100.0, 200.0, 100.0],
    }
)



# Pipeline without scaling
pipeline = DataPipeline(
    [handle_missing_values, remove_duplicates]
)
result = pipeline.process(df)
```

```
Handle missing values
Remove duplicates
```

6.3 Advanced Class Toolkit

6.3.1 `__str__` and `__repr__` Methods

The `__str__` and `__repr__` special methods control how Python displays your objects as strings. Use `__str__` to create readable output for end users, and `__repr__` to create detailed output for debugging that shows the object's exact state.

The example below shows how implementing both methods gives the `ModelMetrics` class readable user output and precise debugging information.

```
class ModelMetrics:
    def __init__(self, model_name: str):
        self.model_name = model_name
```

```
    def __str__(self) -> str:
        return f"{self.model_name} Performance"

    def __repr__(self) -> str:
        return f"ModelMetrics(model_name='{self.model_name}')"


rf_metrics = ModelMetrics("Random Forest")
print(rf_metrics)
print(repr(rf_metrics))
```

```
Random Forest Performance
ModelMetrics(model_name='Random Forest')
```

When `rf_metrics` is the last line in a Jupyter notebook cell, it automatically calls `__repr__` for each object, providing a complete representation of the object's state.

```
rf_metrics
```

```
ModelMetrics(model_name='Random Forest')
```

6.3.2 `__eq__` and `__add__` Methods

`__eq__` and `__add__` are special methods that customize equality comparison and addition operations between objects. Use `__eq__` to define when two objects are considered equal, and `__add__` to define how objects should be combined.

In the `ExperimentResults` class, `__eq__` compares experiments based on validation loss and hyperparameters, while `__add__` averages metrics to combine experiment results.

```
class ExperimentResults:
    def __init__(self, learning_rate, val_loss):
        self.learning_rate = learning_rate
```

```
        self.val_loss = val_loss

    def __eq__(self, other):
        """Check if experiments are similar"""
        return (
            abs(self.val_loss - other.val_loss) < 0.01
            and abs(self.learning_rate - other.learning_rate)
            < 1e-4
        )

    def __add__(self, other):
        """Average results of multiple experiment runs"""
        return ExperimentResults(
            (self.learning_rate + other.learning_rate) / 2,
            (self.val_loss + other.val_loss) / 2,
        )
```

Try out the `ExperimentResults` class with different experiments:

```
exp1 = ExperimentResults(learning_rate=0.001, val_loss=0.245)
exp2 = ExperimentResults(learning_rate=0.001, val_loss=0.248)

print("Comparisons:")
print(f"exp1 == exp2: {exp1 == exp2}")

# Average experiments
avg_exp = exp1 + exp2
print(f"\nAverage loss: {avg_exp.val_loss:.3f}")
print(f"LR: {avg_exp.learning_rate}")
```

```
Comparisons:
exp1 == exp2: True

Average loss: 0.246
LR: 0.001
```

6.3.3 Data Classes

Dataclasses automatically generate special methods like `__init__`, `__repr__`, and `__eq__` for classes that primarily store data, reducing boilerplate code and improving maintainability.

Use dataclasses for simple data containers. The `ModelMetrics` dataclass below demonstrates automatic method generation and type enforcement.

```
from dataclasses import dataclass



@dataclass
class ModelMetrics:
    model_name: str
    accuracy: str



rf_metrics = ModelMetrics("Random Forest", 0.945)
print(rf_metrics)
```

```
ModelMetrics(model_name='Random Forest', accuracy=0.945)
```

Dataclasses handle mutable default values using `default_factory`. The `Student` class below uses `default_factory=list` to ensure each instance gets its own separate grades list, preventing shared mutable state issues.

```
from dataclasses import dataclass, field
from typing import List



@dataclass
class Student:
    name: str
    grades: List[int] = field(default_factory=list)
```

Creating two `Student` instances demonstrates that `student1.grades.append(90)` doesn't affect `student2.grades`.

```
student1 = Student("John")
student2 = Student("Jane")

# Appending grade to student1
student1.grades.append(90)
```

```
print(student1)

# doesn't affect the grades of student2
print(student2)
```

```
Student(name='John', grades=[90])
Student(name='Jane', grades=[])
```

6.3.4 Pydantic

Pydantic is a data validation library that uses Python type annotations to enforce data types and validate complex data structures.

Use Pydantic for data validation, type checking, and ensuring consistency in data science pipelines where data quality is critical.

The `DatasetConfig` class below validates that `train_split` is between 0 and 1, features are strings, and all required fields are present.

```
from pydantic import BaseModel, Field, ValidationError
from typing import List


class DatasetConfig(BaseModel):
    dataset_name: str
    features: List[str]
    target_column: str
    train_split: float = Field(gt=0, lt=1)


# Using the model in a machine learning pipeline
config = DatasetConfig(
    dataset_name="housing_prices",
    features=["sqft", "bedrooms", "location"],
    target_column="price",
    train_split=0.8,
)
```

If any validation fails, Pydantic raises clear error messages, helping catch configuration issues early in the data science workflow.

```
# This will raise a validation error
try:
    invalid_config = DatasetConfig(
        dataset_name="housing_prices",
        features=["sqft", "bedrooms", "location"],
        target_column="price",
        train_split=1.5,
    )
except ValidationError as e:
    print("ValidationError:", e)
```

```
ValidationError: 1 validation error for DatasetConfig
train_split
  ensure this value is less than 1 [type=less_than,
  input_value=1.5, input_type=float]
```

6.3.5 Classmethod in Python

Classmethod is a decorator that defines a method that operates on the class rather than instances. Use classmethods when you need alternative constructors or methods that don't require access to instance-specific data.

For example, we can create the classmethod `from_csv` in the `Dataset` class to create a new instance from a CSV file.

```
import pandas as pd
from typing import List



class Dataset:
    def __init__(
        self, data: pd.DataFrame, name: str, features: List[str]
    ):
        self.data = data
        self.name = name
        self.features = features
```

```
    def __str__(self) -> str:
        return (
            f"Dataset '{self.name}' with {len(self.features)} "
            f"features and {len(self.data)} samples"
        )

    @classmethod
    def from_csv(cls, filepath: str) -> "Dataset":
        data = pd.read_csv(filepath)
        name = filepath.split("/")[-1].replace(".csv", "")
        features = list(data.columns)
        return cls(data, name, features)
```

The `from_csv` classmethod lets you create Dataset instances from CSV files without specifying `data`, `name`, and `features` parameters.

```
housing_data = Dataset.from_csv("data/housing.csv")
print(housing_data)
```

```
Dataset 'housing' with 3 features and 3 samples
```

6.3.6 Staticmethod

Staticmethod defines methods independent of class or instance state. Use staticmethods for utility functions logically related to the class but requiring no class-specific data.

For example, we can create a static method `is_valid_probability` in the `ModelEvaluator` class to check if predictions are valid probabilities.

```
import numpy as np


class ModelEvaluator:
    def __init__(self, predictions: np.ndarray, actuals:
np.ndarray):
        self.predictions = predictions
        self.actuals = actuals
```

```
    @staticmethod
    def is_valid_probability(predictions: np.ndarray) -> bool:
        """Check if predictions are valid probabilities"""
        return all(0 <= p <= 1 for p in predictions)

    def calculate_metrics(self) -> dict:
        """Instance method using static methods"""
        if not self.is_valid_probability(self.predictions):
            raise ValueError("Invalid prediction probabilities")
        squared_errors = (self.predictions - self.actuals) ** 2
        rmse = np.sqrt(np.mean(squared_errors))
        return {"rmse": round(rmse, 3)}
```

You can call the `is_valid_probability` static method in two ways:

Direct class call:

```
# Using static methods directly without instance
predictions = np.array([0.1, 0.8, 0.3])
actuals = np.array([0, 1, 0])

is_valid_probabilities = ModelEvaluator.is_valid_probability(
    predictions
)
print(f"Valid probabilities: {is_valid_probabilities}")
```

```
Valid probabilities: True
```

Instance method call:

```
self.is_valid_probability(self.predictions)
```

6.3.7 Property Decorator

Property decorator provides a way to customize getter, setter, and deleter methods for class attributes. Use it when you need to add validation, computation, or control access to class attributes.

The `@property` decorator creates a `sample_size` property in the `DatasetProfile` class that:

- Enables clean, attribute-like syntax (`profile.sample_size = 2`) while maintaining data integrity
- Adds validation rules in the setter to ensure sample sizes are valid integers within bounds

```
import pandas as pd
from typing import Optional



class DatasetProfile:
    def __init__(self, data: pd.DataFrame):
        self._data = data
        self._sample_size: Optional[int] = None

    @property
    def sample_size(self) -> Optional[int]:
        """Getter for sample size"""
        return self._sample_size

    @sample_size.setter
    def sample_size(self, value: int) -> None:
        """Setter with validation"""
        if not isinstance(value, int):
            raise TypeError("Sample size must be an integer")
        if value <= 0 or value > len(self._data):
            raise ValueError("Invalid sample size")
        self._sample_size = value
```

Let's try out the `DatasetProfile` class with a sample dataset:

```
df = pd.DataFrame({"A": [1, 2, None, 4], "B": [5, None, 7, 8]})

profile = DatasetProfile(df)

# Using setter with validation
try:
    profile.sample_size = 2
    print(f"Sample size set to: {profile.sample_size}")

    # This will raise an error
    profile.sample_size = -1
```

```
except ValueError as e:
    print(f"ValueError: {e}")
```

```
Sample size set to: 2
ValueError: Invalid sample size
```

6.3.8 Slots in Python Classes

The `__slots__` attribute defines a fixed set of instance attributes and reduces memory usage. Apply slots when you want to prevent dynamic attribute addition and optimize memory consumption.

To demonstrate the benefits of using slots, let's create two classes:

- `StandardFeature` class without slots
- `OptimizedFeature` class with slots

```
from typing import List, Optional
import sys

class StandardFeature:
    def __init__(self, name: str, values: List[float]):
        self.name = name
        self.values = values



class OptimizedFeature:
    __slots__ = ["name", "values"]

    def __init__(self, name: str, values: List[float]):
        self.name = name
        self.values = values
```

The `OptimizedFeature` class below uses slots to block accidental attribute assignment, preventing bugs and reducing memory overhead.

```
values = [1, 2, 3, 4, 5]
```

```
# Standard class (without slots)
std_feature = StandardFeature("age", values)

# Dynamic attribute creation works
std_feature.new_attr = "allowed"

# Optimized class (with slots)
opt_feature = OptimizedFeature("age", values)

# Dynamic attribute creation is not allowed
try:
    opt_feature.new_attr = "not allowed"
except AttributeError as e:
    print(
        f"AttributeError: Cannot add new attributes to slotted
class"
    )
```

```
AttributeError: Cannot add new attributes to slotted class
```

When comparing the memory usage of `StandardFeature` and `OptimizedFeature`, slots reduce memory usage by 8 bytes per instance.

```
# Memory comparison
print(
    f"Memory without slots: {sys.getsizeof(std_feature)} bytes"
)
print(f"Memory with slots: {sys.getsizeof(opt_feature)} bytes")
```

```
Memory without slots: 56 bytes
Memory with slots: 48 bytes
```

6.3.9 Scikit-Learn Compatible Class

A scikit-learn compatible class must implement `fit` and `transform` (or `predict`) methods. Use this pattern when creating custom transformers or estimators that need to work within scikit-learn's ecosystem, particularly in pipelines.

The `OutlierCapper` class demonstrates scikit-learn compatibility by:

- Inheriting from `BaseEstimator` and `TransformerMixin`
- Implementing required `fit` and `transform` methods for pipeline integration

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
import numpy as np
from typing import Optional



class OutlierCapper(BaseEstimator, TransformerMixin):
    def __init__(self, percentile: float = 95):
        self.percentile = percentile
        self.threshold_: Optional[np.ndarray] = None

    def fit(self, X: np.ndarray, y: Optional[np.ndarray] = None):
        self.threshold_ = np.percentile(X, self.percentile, axis=0)
        return self

    def transform(self, X: np.ndarray) -> np.ndarray:
        if self.threshold_ is None:
            raise ValueError("Fit the transformer first")
        return np.minimum(X, self.threshold_)
```

Let's try out the `OutlierCapper` class in a scikit-learn pipeline:

```
X = np.array([[1, 10], [2, 20], [100, 1000]])

pipeline = Pipeline(
    [
        ("capper", OutlierCapper(percentile=75)),
        ("scaler", StandardScaler()),
    ]
)

X_transformed = pipeline.fit_transform(X)
print("Transformed data:\n", X_transformed)
```

```
Transformed data:
 [[-0.72842279 -0.72842279]
```

```
[-0.68557439 -0.68557439]
[ 1.41399717  1.41399717]]
```

6.4 Key Takeaways

1. Class design principles:
    - Use descriptive class names in PascalCase that clearly communicate purpose
    - Avoid redundant parameter names that repeat the class name
    - Hide implementation details using private attributes (single underscore prefix)
    - Keep classes focused on single responsibilities
    - Expose only necessary methods to users
2. Data transformation patterns:
    - Use abstract base classes to define standard interfaces for data transformers
    - Implement consistent method names across related classes
    - Use composition to create flexible processing pipelines
3. Inheritance vs Composition:
    - Use inheritance for “is-a” relationships and sharing common functionality
    - Use composition for combining independent behaviors
4. Advanced class features:
    - Use `__str__` for user-friendly output and `__repr__` for unambiguous representation
    - Implement `__eq__` and `__add__` for custom equality and addition operations
    - Use dataclasses for simple data containers
    - Use Pydantic for runtime type checking and validation
    - Use `@classmethod` for alternative constructors
    - Use `@staticmethod` for utility functions
    - Use `@property` for controlled attribute access
    - Use `__slots__` for memory optimization
5. Scikit-learn integration:

- Inherit from `BaseEstimator` and `TransformerMixin` for compatibility
- Implement `fit` and `transform` methods

7 Unit Testing

7.1 What Is Unit Testing?

Unit testing is a software testing method where individual components or units of code are tested to verify that they work as expected.

Key aspects of unit testing:

1. **Isolation**: Each test focuses on a single unit of code, independent of external dependencies
2. **Automation**: Tests can be run automatically and repeatedly
3. **Fast execution**: Unit tests should run quickly since they test small pieces of code
4. **Predictable results**: Same input should always produce same output

7.2 Why Is Unit Testing Essential?

7.2.1 Ensure That Your Code Works as Intended

Automated testing reveals bugs that visual code review cannot detect reliably. Consider code that appears correct but contains subtle logic errors that only surface under specific conditions:

```
import pandas as pd

df = pd.DataFrame(
    {"num_apples": [1, 2, 3, 0], "num_oranges": [4, 5, 0, 6]}
)



def create_booleans(feature):
    return (feature == 0) * 1



df["has_apples"] = create_booleans(df["num_apples"])
df["has_oranges"] = create_booleans(df["num_oranges"])
df
```

	num_apples	num_oranges	has_apples	has_orange
0	1	4	0	0
1	2	5	0	0
2	3	0	0	1
3	0	6	1	0

At first glance, the output appears to be correct, with 0s and 1s present in the `has_apples` and `has_oranges` columns. However, the `create_booleans()` function is actually producing incorrect results: non-zero values should be converted to 1 and zero values should remain as 0.

Through observation alone, this bug could go unnoticed, leading to potentially incorrect downstream analyses or decisions.

Unit tests verify function behavior through systematic automated testing. The example below tests whether `create_booleans()` correctly converts boolean DataFrame values to 0 and 1 format.

```
import pytest
import pandas as pd
from pandas.testing import assert_frame_equal



def create_booleans(feature):
    return (feature == 0) * 1



def test_create_booleans():
    df = pd.DataFrame(
        {"num_apples": [1, 2, 3, 0], "num_oranges": [4, 5, 0, 6]}
    )
    expected_df = pd.DataFrame(
        {"has_apples": [1, 1, 1, 0], "has_oranges": [1, 1, 0, 1]}
    )
    df["has_apples"] = create_booleans(df["num_apples"])
    df["has_oranges"] = create_booleans(df["num_oranges"])
    assert_frame_equal(
        df[["has_apples", "has_oranges"]], expected_df
    )
```

From the test result, we not only see that the test failed but also the reason why it failed.

```
DataFrame.iloc[:, 0] (column name="has_apples") values are
different
[index]: [0, 1, 2, 3]
[left]:  [0, 0, 0, 1]
[right]: [1, 1, 1, 0]
```

7.2.2 Unit Tests Help Identify Edge Cases

Unit tests systematically verify edge cases that normal usage patterns don't reveal.

In the following example, the `calculate_average` function works correctly for most common input cases but fails when given unexpected data like an empty list.

```
def calculate_average(nums: list) -> float:
    return sum(nums) / len(nums)

print(calculate_average([]))
```

```
      1 # Edge case: List with non-numeric values
----> 2 print(calculate_average([]))

      1 def calculate_average(nums: list) -> float:
----> 2     return sum(nums) / len(nums)

ZeroDivisionError: division by zero
```

The `test_calculate_average_empty_list()` function reveals that the function fails with empty lists, enabling you to add proper error handling.

```
import pytest

def calculate_average(nums: list) -> float:
    return sum(nums) / len(nums)

def test_calculate_average_positive_numbers():
    assert calculate_average([1, 2, 3, 4, 5]) == 3

def test_calculate_average_empty_list():
    assert calculate_average([]) == 0
```

```
.F                                                                  [100%]
============================= FAILURES ==============================
_________________ test_calculate_average_empty_list __________________

    def test_calculate_average_empty_list():
>       assert calculate_average([]) == 0
```

7.2.3 Enable Safe Replacement of Existing Code

Unit tests enable safe code refactoring by verifying that changes don't break existing functionality. This confidence allows you to

improve code without fear of introducing bugs.

For example, you might want to replace the `/` operator with `np.divide()` to improve performance. First, create a test that verifies `calculate_ratio()` works correctly with the current implementation.

```
import numpy as np
from pandas.testing import assert_series_equal


def calculate_ratio(
    df: pd.DataFrame, col1: str, col2: str
) -> pd.Series:
    # You want to change from this:
    return df[col1] / df[col2]
    # To this:
    # return np.divide(df[col1], df[col2])

def test_calculate_ratio():
    data = pd.DataFrame({"sales": [100, 200], "cost": [50, 100]})
    expected = pd.Series([2.0, 2.0])
    output = calculate_ratio(data, "sales", "cost")
    assert_series_equal(output, expected)
```

Now replace `calculate_ratio()` with the `np.divide()` version and run the test. If the test passes, the refactoring preserves the original functionality.

```
# Can safely replace with vectorized version
def calculate_ratio(
    df: pd.DataFrame, col1: str, col2: str
) -> pd.Series:
    return np.divide(df[col1], df[col2])


def test_calculate_ratio():
    data = pd.DataFrame({"sales": [100, 200], "cost": [50, 100]})
    expected = pd.Series([2.0, 2.0])
    output = calculate_ratio(data, "sales", "cost")
    assert_series_equal(output, expected)
```

```
1 passed in 0.03s
```

7.2.4 Provide Documentation Through Tests

Unit tests serve as living documentation that shows exactly how functions should be used. Unlike comments that can become outdated, tests demonstrate actual usage patterns and expected behavior through working examples.

For example, consider a function that calculates the distance between two points in 3D space. Without looking at the implementation, it might not be immediately clear what coordinate system this function uses or whether it handles negative coordinates.

```
import math


def calculate_distance(
    x1: float, y1: float, z1: float, x2: float, y2: float, z2:
float
) -> float:
    return math.sqrt(
        (x1 - x2) ** 2 + (y1 - y2) ** 2 + (z1 - z2) ** 2
    )
```

Reading the test cases for `calculate_distance()` reveals key aspects of the function's behavior:

1. It takes six float arguments representing the x, y, and z coordinates of two 3D points.
2. It calculates the Euclidean distance between the two points and returns the result as a float.
3. It can handle positive, negative, and zero coordinate values.
4. It expects the input arguments to be of the correct data type (float), and will raise a `TypeError` if given invalid inputs.

```
import pytest
```

```
def test_calculate_distance_positive_coordinates():
    assert calculate_distance(1, 2, 3, 4, 5, 6) == pytest.approx(
        5.19, abs=1e-2
    )


def test_calculate_distance_negative_coordinates():
    assert calculate_distance(
        -1, -2, -3, -4, -5, -6
    ) == pytest.approx(5.19, abs=1e-2)


def test_calculate_distance_zero_coordinates():
    assert calculate_distance(0, 0, 0, 0, 0, 0) == 0.0


def test_calculate_distance_invalid_inputs():
    with pytest.raises(TypeError):
        calculate_distance("1", 2, 3, 4, 5, 6)
```

7.3 Use Pytest for Unit Testing

When it comes to testing frameworks in Python, pytest stands out as a user-friendly choice. It simplifies the process of writing concise tests, making it an excellent tool for developers new to testing.

To install pytest, execute the following command:

```
pip install pytest
```

7.3.1 Get Started

Let's consider an example where we want to test the `extract_sentiment` function. In pytest, test functions should start with the prefix `test_`. Here's an example implementation:

Example 7.1: test_sentiment.py

```
from textblob import TextBlob


def extract_sentiment(text: str):
    '''Extract sentiment using textblob.
    Polarity is within range [-1, 1]'''
    text = TextBlob(text)
    return text.sentiment.polarity

def test_extract_sentiment():
    text = "I think today will be a great day"
    sentiment = extract_sentiment(text)
    assert sentiment > 0
```

With the test function in place, we can run the test using the following command:

```
pytest test_sentiment.py
```

pytest automatically detects and runs all Python files beginning with `test` and functions prefixed with `test` in the current working directory. After running the test, the output will resemble the following:

```
test_sentiment.py .                                                  [100%]
======================== 1 passed in 0.68s ========================
```

When tests fail, pytest displays the exact test that failed and the expected versus actual values.

```
def test_extract_sentiment():
    text = "I think today will be a great day"
    sentiment = extract_sentiment(text)
    assert sentiment < 0
```

```
========================== FAILURES ===============================
____________________ test_extract_sentiment ____________________

    def test_extract_sentiment():
        text = "I think today will be a great day"
        sentiment = extract_sentiment(text)
```

```
>       assert sentiment < 0
E       assert 0.8 < 0
```

7.3.2 Multiple Tests for the Same Function

pytest allows us to create multiple tests for the same function, enabling comprehensive testing scenarios.

Figure 7.1 illustrates this concept, where two separate tests (Test 1 and Test 2) are used to thoroughly validate the behavior of a single function.

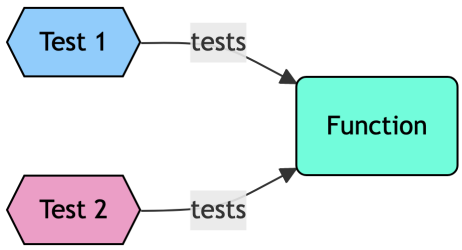


Figure 7.1: Multiple tests for a single function

Here's an example of how to write multiple tests for the same function:

```python
def test_extract_sentiment_positive():
    text = "I think today will be a great day"
    sentiment = extract_sentiment(text)
    assert sentiment > 0

def test_extract_sentiment_negative():
    text = "I do not think this will turn out well"
    sentiment = extract_sentiment(text)
    assert sentiment < 0
```

The test summary shows that the `test_extract_sentiment_negative` test failed while the `test_extract_sentiment_positive` test passed.

```
=================== short test summary info ===================
FAILED sentiment.py::test_extract_sentiment_negative - assert 0.0 <
0
================= 1 failed, 1 passed in 0.78s =================
```

7.3.3 Parametrization: Combining Tests

When multiple test functions test the same functionality, combine them into a single test function with parameterization using `pytest.mark.parametrize()`.

Figure 7.2 illustrates this approach, where a single test function utilizes different parameters (Param 1 and Param 2) to test the same function.

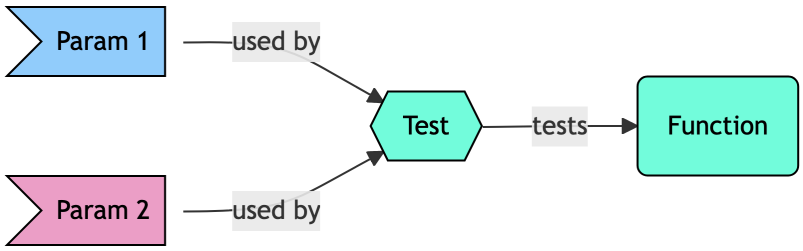


Figure 7.2: Parameterized test for a single function

Let's explore two approaches to parametrizing tests:

- Using a list of input samples to test the same condition
- Using pairs of inputs and expected outputs to verify specific results

7.3.3.1 Parametrize With a List of Samples

The `@pytest.mark.parametrize` decorator enables testing multiple inputs with identical logic.

In the example below, it automatically executes `test_extract_sentiment` for each string in `testdata`, validating positive sentiment across all samples.

```
from textblob import TextBlob
import pytest



def extract_sentiment(text: str):
    text = TextBlob(text)
    return text.sentiment.polarity



testdata = [
    "I think today will be a great day",
    "I do not think this will turn out well",
]



@pytest.mark.parametrize("sample", testdata)
def test_extract_sentiment(sample):
    sentiment = extract_sentiment(sample)
    assert sentiment > 0
```

The results show that the positive sentiment test passed but the negative sentiment test failed:

```
============================ FAILURES ============================
sample = 'I do not think this will turn out well'

    @pytest.mark.parametrize('sample', testdata)
    def test_extract_sentiment(sample):
        sentiment = extract_sentiment(sample)
>       assert sentiment > 0
E       assert 0.0 > 0

sentiment.py:19: AssertionError
```

```
=================== short test summary info ====================
================== 1 failed, 1 passed in 0.80s ==================
```

7.3.3.2 Parametrize With a List of Examples and Expected Outputs

Apply `pytest.mark.parametrize` with a list of tuples to test a function against multiple input-output combinations.

```
import pytest

def sentence_contain_word(word: str, sentence: str):
    return word in sentence



testdata = [
    ("There is a duck", True), ("There is nothing here", False)
]



@pytest.mark.parametrize("sample, expected_output", testdata)
def test_sentence_contain_word(sample, expected_output):
    word = "duck"
    assert sentence_contain_word(word, sample) == expected_output
```

The test shows that both tests passed.

```
test_sentence_contain_word[There is a duck-True] PASSED
test_sentence_contain_word[There is nothing here-False] PASSED
====================== 2 passed in 0.04s =======================
```

7.3.4 Test One Function at a Time

Testing one function at a time improves test execution times, enabling faster feedback loops during development.

By default, running `pytest test_file.py` executes all the tests in that file. However, if you want to test a specific function, you can run

`pytest test_file.py::test_function_name`.

Figure 7.3 illustrates this concept, where running the pytest command with the specific test function name results in only that test being executed, while the other tests in the file are skipped.

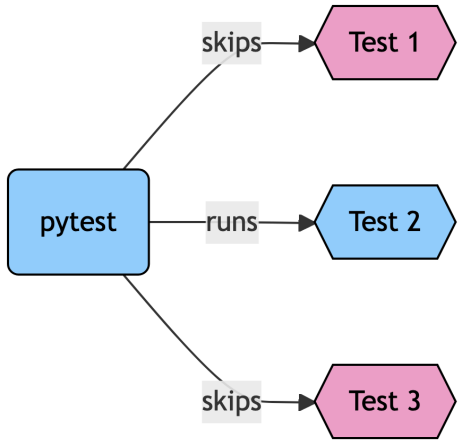


Figure 7.3: Selectively running a single test

Let's consider a file called `sentiment_two_tests.py` that contains two test functions:

Example 7.2: sentiment_two_tests.py

```
from textblob import TextBlob

def extract_sentiment(text: str):
    text = TextBlob(text)
    return text.sentiment.polarity

def test_extract_sentiment_positive():
    text = "I think today will be a great day"
    sentiment = extract_sentiment(text)
    assert sentiment > 0

def test_extract_sentiment_negative():
    text = "I do not think this will turn out well"
```

```
    sentiment = extract_sentiment(text)
    assert sentiment < 0
```

To run only `test_sentence_contain_word`, type:

```
pytest sentiment_two_tests.py::test_extract_sentiment_positive

sentiment_two_tests.py::test_extract_sentiment_positive PASSED

======================== 1 passed in 0.73s ========================
```

7.3.5 Fixtures: Use the Same Data to Test Different Functions

pytest fixtures allow you to use the same data across multiple test functions, ensuring consistency and reducing duplication.

Figure 7.4 illustrates this concept, where a common data source (Data) is used by multiple test functions (Test 1 and Test 2) to validate the behavior of a single function.

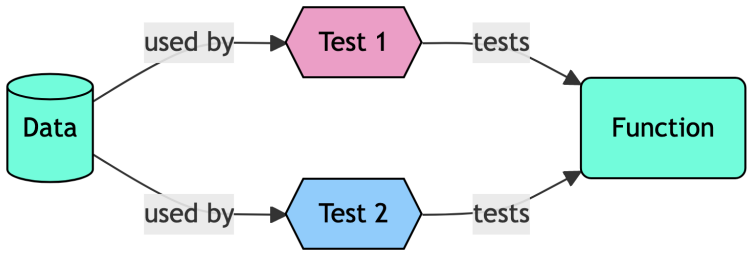


Figure 7.4: Reusing test data with fixtures

The following example demonstrates how to use a fixture to share test data across multiple functions. The `example_data` fixture:

- Returns the sentence “Today I found a duck and I am happy”
- Supplies input data to two different test functions

```
@pytest.fixture
def example_data():
    return 'Today I found a duck and I am happy'

def test_extract_sentiment(example_data):
    sentiment = extract_sentiment(example_data)
    assert sentiment > 0

def test_sentence_contain_word(example_data):
    word = 'duck'
    assert sentence_contain_word(word, example_data) == True
```

```
..
[100%]
2 passed in 0.01s
```

7.4 Best Practices for Unit Testing

7.4.1 Use Descriptive Test Names in Unit Tests

Use descriptive test names that clearly communicate the test's purpose and expected behavior.

In the example below, generic names like `test_sentiment_1` fail to describe what's being tested, making error messages unclear.

```
from textblob import TextBlob

def extract_sentiment(text: str):
    """Extract sentiment using textblob.
    Polarity is within range [-1, 1]"""
    text = TextBlob(text)
    return text.sentiment.polarity

def test_sentiment_1():
    text = "I think today will be a great day"
    sentiment = extract_sentiment(text)
```

```
    assert sentiment > 0

def test_sentiment_2():
    text = "I am very sad today"
    sentiment = extract_sentiment(text)
    assert sentiment < 0

def test_sentiment_3():
    text = "The weather is neither good nor bad"
    sentiment = extract_sentiment(text)
    assert -0.1 < sentiment < 0.1
```

Instead, use descriptive test names to clearly communicate the test's purpose, expected behavior, and conditions being tested.

```
def test_extract_sentiment_positive():
    text = "I think today will be a great day"
    sentiment = extract_sentiment(text)
    assert sentiment > 0


def test_extract_sentiment_negative():
    text = "I am very sad today"
    sentiment = extract_sentiment(text)
    assert sentiment < 0


def test_extract_sentiment_neutral():
    text = "The weather is neither good nor bad"
    sentiment = extract_sentiment(text)
    assert -0.1 < sentiment < 0.1
```

7.4.2 Use Logical Test Directory Structure

Use a clear directory structure and naming conventions to make your tests organized, discoverable, and maintainable.

The following directory structure shows a common approach where each `test_*.py` file in the `tests/` directory corresponds to a `.py` file in the `src/` directory:

```
project/
│
├── src/
│   ├── __init__.py
│   ├── data_processor.py
│   └── model.py
└── tests/
    ├── __init__.py
    ├── conftest.py           # Shared fixtures
    ├── test_data_processor.py
    └── test_model.py
```

The `conftest.py` file contains shared fixtures that can be used by all tests.

Example 7.3: tests/conftest.py

```
import pytest
import pandas as pd



@pytest.fixture
def sample_df():
    return pd.DataFrame({"feature": [1, 2, 3], "target": [0, 1,
0]})
```

7.4.3 Test One Thing at a Time

Write separate tests for each function behavior to:

- Provide clear failure messages that identify exactly which behavior failed
- Ensure all test cases execute independently

The example below shows a test with multiple assertions. When the test fails, developers must manually investigate which assertion broke and what input caused the failure.

```
def test_sentiment_analysis_multiple_cases():
    # Test positive sentiment
```

```
    text1 = "I am very happy today!"
    sentiment1 = extract_sentiment(text1)
    assert sentiment1 == 0

    # Test negative sentiment
    text2 = "I am very sad today"
    sentiment2 = extract_sentiment(text2)
    assert sentiment2 < 0

    # Test empty string handling
    text3 = ""
    sentiment3 = extract_sentiment(text3)
    assert sentiment3 == 0
```

```
F
[100%]
============================== FAILURES
==============================
_____________ test_sentiment_analysis_multiple_cases
_____________

    def test_sentiment_analysis_multiple_cases():
        # Test positive sentiment
        text1 = "I am very happy today!"
        sentiment1 = extract_sentiment(text1)
>       assert sentiment1 == 0
E       assert 1.0 == 0
```

Instead, write focused tests that verify one specific behavior. Each test should make a single assertion with a descriptive name that explains the expected outcome.

```
def test_positive_text_returns_positive():
    text = "I am very happy today!"
    sentiment = extract_sentiment(text)
    assert sentiment == 0

def test_negative_text_returns_negative():
    text = "I am very sad today"
    sentiment = extract_sentiment(text)
    assert sentiment < 0

def test_empty_text_returns_zero():
```

```
    text = ""
    sentiment = extract_sentiment(text)
    assert sentiment == 0
```

By splitting the test into three separate tests, we can immediately see that the `test_positive_text_returns_positive` test failed.

```
    def test_positive_text_returns_positive():
        text = "I am very happy today!"
        sentiment = extract_sentiment(text)
>       assert sentiment == 0
E       assert 1.0 == 0
```

7.4.4 Test Both Expected And Unexpected Cases

Test both normal operations and edge cases to ensure your code handles all scenarios robustly. Here's an example of tests for the `extract_sentiment()` function.

Test basic functionality:

```
class TestBasicSentiment:
    def test_positive_sentiment(self):
        text = "I am happy today"
        sentiment = extract_sentiment(text)
        assert sentiment > 0

    def test_negative_sentiment(self):
        text = "I am sad today"
        sentiment = extract_sentiment(text)
        assert sentiment < 0

    def test_neutral_sentiment(self):
        text = "The sky is blue"
        sentiment = extract_sentiment(text)
        assert -0.1 <= sentiment <= 0.1
```

Test boundary conditions:

```
class TestBoundaryConditions:
    def test_empty_string(self):
        text = ""
        sentiment = extract_sentiment(text)
        assert sentiment == 0

    def test_single_character(self):
        text = "."
        sentiment = extract_sentiment(text)
        assert isinstance(sentiment, float)

    def test_very_long_text(self):
        text = "I am happy " * 1000
        sentiment = extract_sentiment(text)
        assert isinstance(sentiment, float)
        assert sentiment > 0
```

Test text formatting:

```
class TestTextFormatting:
    def test_multiple_whitespace(self):
        text = "   I   am   happy   "
        sentiment = extract_sentiment(text)
        assert sentiment > 0

    def test_newlines_and_tabs(self):
        text = "I am\nhappy\ttoday"
        sentiment = extract_sentiment(text)
        assert sentiment > 0

    def test_special_characters(self):
        text = "I am happy! 😊 #blessed"
        sentiment = extract_sentiment(text)
        assert sentiment > 0
```

Test invalid inputs:

```
class TestInvalidInputs:
    @pytest.mark.parametrize(
        "invalid_input", [None, 123, ["text"], {"text": "happy"}]
    )
    def test_invalid_input_types(self, invalid_input):
```

```
        with pytest.raises(TypeError):
            extract_sentiment(invalid_input)
```

7.4.5 Centralize Test Data Preparation

Extract common test data creation into reusable fixtures to improve maintainability and reduce repetitive setup code.

In the following example, we repeat creating the same DataFrame in each test, creating unnecessary code duplication.

```
import pandas as pd



def analyze_sales(df: pd.DataFrame) -> dict:
    return {
        "total": df["amount"].sum(), "avg": df["amount"].mean()
    }



def test_total_sales():
    df = pd.DataFrame(
        {
            "amount": [100, 200, 300],
            "date": ["2023-01-01", "2023-01-02", "2023-01-03"],
        }
    )
    result = analyze_sales(df)
    assert result["total"] == 600



def test_average_sales():
    df = pd.DataFrame(
        {
            "amount": [100, 200, 300],
            "date": ["2023-01-01", "2023-01-02", "2023-01-03"],
        }
    )
    result = analyze_sales(df)
    assert result["avg"] == 200
```

We can simplify the data setup by using a pytest fixture to define a single test data source:

```
@pytest.fixture
def sample_df():
    return pd.DataFrame(
        {
            "amount": [100, 200, 300],
            "date": ["2023-01-01", "2023-01-02", "2023-01-03"],
        }
    )


def test_total_sales(sample_df):
    result = analyze_sales(sample_df)
    assert result["total"] == 600


def test_average_sales(sample_df):
    result = analyze_sales(sample_df)
    assert result["avg"] == 200
```

Alternatively, use a setup method to set up the data once and reuse it in each test:

```
class TestSalesAnalysis:
    def setup_method(self):
        self.df = pd.DataFrame(
            {
                "amount": [100, 200, 300],
                "date": ["2023-01-01", "2023-01-02", "2023-01-03"],
            }
        )

    def test_total_sales(self):
        result = analyze_sales(self.df)
        assert result["total"] == 600

    def test_average_sales(self):
        result = analyze_sales(self.df)
        assert result["avg"] == 200
```

By using a single test data source, you make your tests more consistent, easier to update, and cleaner to read.

7.4.6 Use Synthetic Test Data

Generate test data within your tests rather than depending on external files for better reliability and control. This approach eliminates file dependency issues and simplifies edge case testing.

The example below shows a test that depends on `data.csv`. When the file changes or goes missing, the test will break.

```
def drop_outliers(df: pd.DataFrame, col: str, threshold: float):
    df = df[df[col] < threshold]
    return df


def test_drop_outliers():
    data = pd.read_csv("data/sample.csv") # can change or go
missing
    result = drop_outliers(data, "age", 100)
    assert (result["age"] < 100).all()
```

Instead, create a DataFrame with specific age values to test `drop_outliers`. This synthetic data provides reliable, predictable test conditions.

```
def test_drop_outliers():
    # Using synthetic data with known values
    data = pd.DataFrame({"age": [10, 50, 90, 110, 20]})
    result = drop_outliers(data, "age", 100)
    assert list(result["age"]) == [10, 50, 90, 20]
```

7.4.7 Use Mocking in Unit Tests

Mock external dependencies like database connections to create isolated, reliable tests. Mocking replaces real external services with

fake objects that simulate their behavior without actual network calls or database queries.

The example below shows a test that depends on a real database connection, which creates several issues:

- It requires a live database connection, making tests slow to run
- Results can be inconsistent between test runs due to changing database state
- Tests may fail if database is unavailable or credentials expire

```
import pandas as pd

def get_active_users() -> pd.DataFrame:
    # Direct database dependency
    df = pd.read_sql(
        "SELECT user_id FROM users WHERE status = 'active'",
        "postgresql://user:pass@localhost:5432/prod"
    )
    return df.head(10)

def test_get_active_users():
    # Test requires actual database
    result = get_active_users()
    assert len(result) <= 10
    assert 'user_id' in result.columns
```

Instead of using a live database, mock the `pd.read_sql` function to return controlled test data. Key components include:

- `@patch` specifies which function to mock (`pandas.read_sql`)
- Mock objects are injected automatically as test function arguments (`mock_read_sql`)
- `return_value` controls what the mock returns when called
- `assert_called_once()` verifies the mock was called exactly once

```
@patch("pandas.read_sql")
def test_get_active_users(mock_read_sql):
    # Step 1: Set up the mock response
    mock_df = pd.DataFrame({"user_id": [1, 2, 3]})
```

```
    # Tell the mock what to return when called
    mock_read_sql.return_value = mock_df

    # Step 2: Call the function under test
    result = get_active_users()

    # Step 3: Check the results
    assert len(result) == 3
    assert list(result.columns) == ["user_id"]

    # Step 4: Verify the mock was called correctly
    mock_read_sql.assert_called_once()
```

You can also use `side_effect` to simulate database connection failures in the mocked `pd.read_sql` function.

```
@patch("pandas.read_sql")
def test_get_active_users_error(mock_read_sql):
    # Make the mock raise an error
    mock_read_sql.side_effect = ConnectionError("DB Error")

    # Test error handling
    try:
        result = get_active_users()
    except ConnectionError as e:
        assert str(e) == "DB Error"
```

7.4.8 Use Simple Test Examples

Using simple, focused tests makes the expected outcomes clear and enables faster test execution.

The test below is overly complex, including unnecessary data fields like `timestamp`, `category`, and `metadata`. Its detailed assertions and complex test data make it difficult to understand the expected behavior.

```
import pandas as pd
import numpy as np
```

```
def calculate_metrics(df: pd.DataFrame, col: str) -> dict:
    return {"mean": df[col].mean(), "std": df[col].std()}


def test_calculate_metrics():
    # Complex test data with unnecessary values
    data = pd.DataFrame(
        {
            "value": [
                10.5, 20.3, 15.7, 18.9, 22.1,
                19.5, 17.8, 16.4, 21.2, 23.4,
                12.3, 14.7, 19.8, 22.5, 16.9
            ],
            "timestamp": pd.date_range("2023-01-01", periods=15),
            "category": ["A", "B", "C"] * 5,
            "metadata": [f"info_{i}" for i in range(15)],
        }
    )

    result = calculate_metrics(data, "value")

    # Overly precise assertions
    assert round(result["mean"], 3) == 18.133
    assert round(result["std"], 3) == 3.747
```

Create simple, focused tests with clear intentions instead. The following test eliminates unnecessary complexity by using basic data and simple assertions:

```
def test_calculate_metrics():
    # Simple test data with round numbers
    data = pd.DataFrame({"value": [10, 20, 30]})

    result = calculate_metrics(data, "value")

    assert result["mean"] == 20
    assert result["std"] == 10
```

7.5 Key Takeaways

1. Benefits of unit testing:
    - Verifies code works as intended
    - Identifies edge cases
    - Enables safe code refactoring
    - Serves as living documentation
    - Improves code quality and maintainability
2. Pytest best practices:
    - Use descriptive test names that explain the test's purpose
    - Test one function at a time for faster feedback
    - Use fixtures to share test data between tests
    - Use parametrization to test multiple scenarios
    - Follow a logical test directory structure
3. Writing effective tests:
    - Test both normal cases and edge cases
    - Keep tests simple and focused
    - Keep tests independent and isolated using synthetic data or mocking

8 Configuration Management

8.1 What Is Configuration Management?

Configuration management involves organizing and controlling application settings, parameters, and environment variables in a structured way. For example, in a data science project, you might need to manage:

- Data paths and sources (e.g., CSV files, databases, S3 buckets)
- Model hyperparameters (e.g., learning rate, number of epochs, batch size)
- Feature engineering settings (e.g., columns to drop, scaling parameters)
- Training configurations (e.g., train/test split ratio, random seed)
- Environment-specific settings (e.g., development vs production databases)

8.2 Why Is Configuration Management Essential?

8.2.1 Cleaner Codebase

Separating configuration from code helps keep your codebase cleaner and more organized.

The following code mixes core logic with hard-coded values like file paths, test size, and random state, making it difficult to understand the core logic.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
import joblib

data = pd.read_csv("data/raw/winequality-red.csv")

data = data.drop(columns=["free sulfur dioxide"])

X = data.drop(columns="quality")
y = data["quality"]
```

To keep the codebase focused on the logic, store the configuration in a separate YAML file.

main.yaml

```
data_path:
  raw: data/raw/winequality-red.csv
  intermediate: data/intermediate

model_path: models/model.pkl

columns:
  to_drop:
    - free sulfur dioxide
  target: quality
```

Then, load the configuration in the Python script using the yaml library.

Example 8.1: main.py

```
import yaml
```

```
# Load config
with open("main.yaml", "r") as f:
    config = yaml.safe_load(f)

data = pd.read_csv(config["data_path"]["raw"])

data = data.drop(columns=config["columns"]["to_drop"])

X = data.drop(columns=config["columns"]["target"])
y = data[config["columns"]["target"]]
```

8.2.2 Flexible Experimentation

Configuration files enable you to experiment with different parameters, such as hyperparameters, without modifying the core code.

You can create separate config files for each experiment and switch between them to test different approaches .

Experiment 1:

Example 8.2: config/svm_experiment.yaml

```
split:
  test_size: 0.2
  random_state: 42

train:
  model: SVM
  hyperparameters:
    kernel: rbf
    C: 1.0
    gamma: 0.001
```

Experiment 2:

Example 8.3: config/random_forest_experiment.yaml

```yaml
split:
  test_size: 0.2
  random_state: 42
```

Example 8.4: config/random_forest_experiment.yaml (Continued)

```yaml
train:
  model: RandomForest
  hyperparameters:
    n_estimators: 200
    max_depth: 15
    min_samples_split: 5
```

8.2.3 Environment-Specific Configurations

In data science projects, you often need different configurations for development, staging, and production environments. For example:

- Development: Using local CSV files and small datasets for rapid iteration
- Staging: Testing with cloud storage (e.g., S3) and medium-sized datasets
- Production: Running on cloud infrastructure with full datasets and strict security

Configuration files let you seamlessly switch between these environments by changing a single environment variable, without modifying any code. This separation of configuration from code is crucial for maintaining a reliable deployment pipeline.

For example, you can create separate configuration files for each environment.

Development:

Example 8.5: config/development.yaml

```
data_source: csv
file_path: data/raw/winequality-red.csv
logging_level: DEBUG
```

Staging:

Example 8.6: config/staging.yaml

```
data_source: s3
bucket: staging-ml-bucket
file_path: datasets/winequality-red.csv
logging_level: INFO
```

Production:

Example 8.7: config/production.yaml

```
data_source: s3
bucket: production-ml-bucket
file_path: datasets/winequality-red.csv
logging_level: WARNING
```

Then, load the appropriate configuration file based on the environment variable.

Example 8.8: main.py

```
import yaml
import os

# Load environment-specific config
env = os.getenv("ENVIRONMENT", "development")
with open(f"config/{env}.yaml", "r") as file:
    config = yaml.safe_load(file)

print(f"Using {env} environment")
print(f"Data source: {config['data_source']}")
```

```
Using development environment
Data source: csv
```

8.3 Use Hydra to Manage Configurations

8.3.1 Introduction to Hydra

Hydra is a powerful framework that simplifies the configuration and management of complex applications.

It helps you keep your code clean, flexible, and scalable by supporting:

- Intuitive access to parameters via dot notation
- Quick overrides from the command line for fast iteration
- Logical grouping of configs to manage complexity
- Multi-run execution to automate combinations of configurations

Let's dig deeper into each of these features.

8.3.2 Convenient Parameter Access

Hydra enables convenient parameter access through dot notation by decorating your main function with the `@hydra.main` decorator.

Suppose all configuration files are stored under the `conf` folder and all Python scripts are stored under the `src` folder.

```
.
├── conf/
|   └── main.yaml
└── src/
    └── process.py
```

Configuration file:

`conf/main.yaml`

```
process:
cols_to_drop:
- free sulfur dioxide
feature: quality
test_size: 0.2
```

Use the `@hydra.main` decorator to tell Hydra where to find and apply the configuration. The `config` parameter is a `DictConfig` object that supports both dot notation (`config.key`) and dictionary-style access (`config['key']`).

Example 8.9: src/process.py

```
import hydra
from omegaconf import DictConfig


@hydra.main(
    config_path="../conf", config_name="main", version_base=None
)
def process_data(config: DictConfig):
    # You can use a bracket notation
    print(config["process"]["cols_to_drop"])

    # or use a dot notation
    print(config.process.cols_to_drop)


if __name__ == "__main__":
    process_data()
```

```
['free sulfur dioxide']
['free sulfur dioxide']
```

8.3.3 Command-Line Help Feature

Hydra's help menu allows you to easily explore your script's configuration options without needing to review code or configuration files.

To access the help menu, simply run your script with the `--help` flag.

```
python src/process.py --help
```

```
process is powered by Hydra.

== Config ==
Override anything in the config (foo.bar=value)

process:
  cols_to_drop:
  - free sulfur dioxide
  feature: quality
  test_size: 0.2
```

8.3.4 Command-Line Configuration Override

Command-line configuration overrides let you test different parameters without modifying configuration files.

For example, when experimenting with different `test_size` values, editing the configuration file repeatedly is time-consuming. Instead, use Hydra's command-line syntax to override values directly:

```
python src/process.py process.test_size=0.3
```

8.3.5 Multi-run

Hydra's multirun feature allows simultaneous execution with different configurations, eliminating the need to run experiments sequentially.

For example, instead of running experiments with different `test_size` values of 0.2 and 0.3 sequentially, use multirun to execute them simultaneously:

```
python src/process.py --multirun process.test_size=0.2,0.3
```

8.3.6 Interpolation

Hydra's interpolation feature lets you reference config values using `${...}` syntax. This helps reduce duplicates and makes updates easier.

The following configuration file repeats `data/customer_segmentation/v1` in multiple paths.

conf/main.yaml

```yaml
project:
  name: customer_segmentation
  version: v1

paths:
  base: data/customer_segmentation/v1
  raw: data/customer_segmentation/v1/raw
  processed: data/customer_segmentation/v1/processed
  reports: data/customer_segmentation/v1/reports
```

With interpolation, you can simplify the file by:

- Setting `base` to reference `project.name` and `project.version`
- Setting other paths to reference `base`

conf/main.yaml

```yaml
project:
  name: customer_segmentation
  version: v1

paths:
  base: data/${project.name}/${project.version}
  raw: ${paths.base}/raw
  processed: ${paths.base}/processed
  reports: ${paths.base}/reports
```

8.3.7 Grouping config files

Hydra supports organizing related configurations into logical groups, making it easier to switch between variations of preprocessing steps, models, or training strategies.

Here's how to set up and use a config group for processing options:

Update your project structure to organize different processing strategies under a `process/` config group.

```
.
└── conf/
    ├── main.yaml
    └── process/
        ├── drop_missing.yaml
        └── impute.yaml
```

Create a configuration file for each processing strategy.

conf/process/drop_missing.yaml

```yaml
strategy: drop_missing
cols_to_drop: ["id", "timestamp", "customer_id"]
impute_strategy: null
feature: quality
test_size: 0.2
```

conf/process/impute.yaml

```yaml
strategy: impute
cols_to_drop: []
impute_strategy: mean
feature: quality
test_size: 0.2
```

Update the `main.yaml` file to reference the `process` group.

conf/main.yaml

```yaml
defaults:
  - process: drop_missing
  - _self_
```

```yaml
data:
  raw: data/raw/winequality-red.csv
  intermediate: data/intermediate
```

To switch between groups, simply use command-line overrides.

```
python src/process.py process=impute
```

Apply the same grouping approach to training strategies. Update `main.yaml` to reference the `train` group.

Updated directory structure:

```
conf/
├── main.yaml
├── process/
|   ├── drop_missing.yaml
|   └── impute.yaml
└── train/
    ├── basic.yaml
    └── advanced.yaml
```

Updated `main.yaml` file:

conf/main.yaml

```yaml
defaults:
  - process: drop_missing
  - train: basic
  - _self_
```

Mix and match configurations with a single command:

```
python src/process.py process=impute train=advanced
```

8.4 Best Practices for Configuration Management

8.4.1 Use Meaningful YAML Configuration Structure

Use hierarchical structure in YAML configuration files to organize data science project settings logically and make them easier to maintain:

- Developers can quickly find relevant sections without scanning the entire file
- Adding new parameters is straightforward, reducing errors and clutter

Here is an example of a well-structured YAML configuration file:

```yaml
paths:
  data:
    raw: data/raw/winequality-red.csv
    intermediate: data/intermediate
  model: models/model.pkl

preprocessing:
  columns_to_drop:
    - free sulfur dioxide
  target: quality
  split:
    test_size: 0.2
    random_state: 42
```

8.4.2 Never Include Sensitive Data in Code or Config Files

Never include sensitive data like API keys, passwords, or access tokens in code or configuration files. These files are tracked in version control so anyone with repository access can view past commits and obtain the sensitive data.

```yaml
api:
  key: sk_live_12345abcdef
  url: https://api.alphavantage.co/query
```

Instead, store only non-sensitive configuration in config files and put sensitive data in `.env` files.

Config file:

`main.yaml`

```yaml
api:
  url: https://api.alphavantage.co/query
```

Environment file:

`.env`

```
ALPHA_VANTAGE_KEY=sk_live_12345abcdef
```

Add the `.env` file to `.gitignore` to prevent commits:

```
# .gitignore
.env
```

To load the sensitive information from the `.env` file, use the python-dotenv library:

```python
import os
from dotenv import load_dotenv
import yaml
import pandas as pd

# Load environment variables
load_dotenv(".env")

# Get API key from environment
api_key = os.getenv("ALPHA_VANTAGE_KEY")
if not api_key:
    raise ValueError("API key not found in environment variables")
```

8.5 Key Takeaways

1. Benefits of configuration management:
    - Separate configuration from code for better maintainability
    - Keep code clean and focused on core logic
    - Enable flexible experimentation with different parameters
2. Hydra framework benefits:
    - Provides intuitive parameter access via dot notation
    - Enables quick configuration overrides from the command line
    - Supports logical grouping of related configurations
    - Allows multi-run execution for testing multiple configurations
    - Offers interpolation to reduce configuration duplication
3. Best practices:
    - Use a hierarchical structure in configuration files
    - Never include sensitive data in code or configuration files

9 Logging and Exception Handling

9.1 What Is Logging?

If you're familiar with using `print()` statements to track what's happening in your code, logging is like `print()` but with superpowers.

Think of logging as a more sophisticated way to monitor your code's execution, especially useful when your data science projects grow beyond simple notebooks.

9.2 Why Should You Use Logging Instead of Print?

As your data science projects evolve from notebooks to production-ready pipelines, `print()` becomes harder to manage. Logging offers structured execution tracking that scales with your codebase.

For example, consider the following data science project where you want to track different stages like data loading, preprocessing, model training, and error handling:

```
print("Loaded 1000 rows")
print("Training RandomForest model")
```

```
print("Missing values detected")
print("Model training failed")
```

```
Loaded 1000 rows
Training RandomForest model
Missing values detected
Model training failed
```

This works fine locally, but in a production environment:

- There's no record of when these events occurred.
- There's no way to save that record to a file for later inspection.
- There's no indication of the severity of each message, making it hard to distinguish between general informational messages and serious runtime errors.

Unlike `print`, the `logging` module supports log levels, output formatting, and saving to a file. Here's an example:

Example 9.1: train.py

```
import logging

logging.basicConfig(
    level=logging.DEBUG,
    format="%(asctime)s | %(levelname)s | "
           "%(module)s:%(funcName)s:%(lineno)d - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)

def main():
    logging.debug("Loaded 1000 rows")
    logging.info("Training RandomForest model")
    logging.warning("Missing values detected")
    logging.error("Model training failed")

if __name__ == "__main__":
    main()
```

```
2025-06-04 13:24:03 | DEBUG | train:main:11 - Loaded 1000 rows
2025-06-04 13:24:03 | INFO | train:main:12 - Training RandomForest
model
```

```
2025-06-04 13:24:03 | WARNING | train:main:13 - Missing values
detected
2025-06-04 13:24:03 | ERROR | train:main:14 - Model training failed
```

In this output, we can see the following:

- The timestamp of the event (`2025-06-04 13:24:03`)
- The log level (`DEBUG`, `INFO`, `WARNING`, `ERROR`)
- The module and function name (`train:main`)
- The line number (`11`, `12`, `13`, `14`)
- The message (`Loaded 1000 rows`)

You can hide debug logs and focus only on more critical messages by changing the log level to `INFO`:

Example 9.2: train.py

```
logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | "
           "%(module)s:%(funcName)s:%(lineno)d - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
```

The output shows only info, warning, and error logs. Debug logs are now hidden.

```
2025-06-04 13:24:03 | INFO | train:main:12 - Training RandomForest
model
2025-06-04 13:24:03 | WARNING | train:main:13 - Missing values
detected
2025-06-04 13:24:03 | ERROR | train:main:14 - Model training failed
```

9.3 Use Loguru for Python Logging

While logging offers many benefits, data scientists often prefer print statements due to their simplicity and minimal setup requirements. For small scripts and one-time tasks, the extra overhead of configuring a logging framework may seem unnecessary.

This is where Loguru comes in handy. It's a library that combines the power of logging with the ease of using print statements.

To install Loguru, run the following command:

```
pip install loguru
```

In the following sections, we'll explore how Loguru simplifies the logging process.

9.3.1 Elegant Out-of-the-Box Functionality

Let's see how Loguru compares to logging in terms of out-of-the-box functionality.

9.3.2 With logging

By default, the logging library produces bland and less informative logs:

```
import logging

logging.basicConfig(level=logging.DEBUG)



def main():
    logging.debug("This is a debug message")
    logging.info("This is an info message")
    logging.warning("This is a warning message")
    logging.error("This is an error message")



if __name__ == "__main__":
    main()
```

```
DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
```

9.3.3 With Loguru

In contrast, Loguru generates colorful and informative logs by default:

```python
from loguru import logger

def main():
    logger.debug("This is a debug message")
    logger.info("This is an info message")
    logger.warning("This is a warning message")
    logger.error("This is an error message")

if __name__ == "__main__":
    main()
```

```
2025-05-03 15:55:18 | DEBUG   | __main__:<main>:4 - This is a debug message
2025-05-03 15:55:18 | INFO    | __main__:<main>:5 - This is an info message
2025-05-03 15:55:18 | WARNING | __main__:<main>:6 - This is a warning message
2025-05-03 15:55:18 | ERROR   | __main__:<main>:7 - This is an error message
```

9.3.4 Format Logs Easily

Formatting logs allows you to add useful information to logs such as timestamps, log levels, module names, function names, and line numbers. Here's how to do it with both logging and Loguru:

9.3.4.1 With logging

The traditional logging approach uses the `%` formatting, which is not intuitive to use and maintain:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | "
           "%(module)s:%(funcName)s:%(lineno)d - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
)
```

9.3.4.2 With Loguru

In contrast, Loguru uses the `{}` formatting, which is much more readable and easy to use:

```
import sys
from loguru import logger

# Remove the default handler
logger.remove()

# Add a stream handler
logger.add(
    sys.stdout,
    format="{time:YYYY-MM-DD HH:mm:ss} | {level} | "
           "{module}:{function}:{line} - {message}",
    level="INFO",
)
```

In the code above:

- `logger.remove()` clears the default Loguru handler so that only your custom configuration is active.
- `logger.add(sys.stdout, ...)` explicitly adds a stream handler that logs to the terminal using your specified format and log level.

Other common options for time formatting:

Category	Token	Output Example
Year	YYYY	2025

Category	Token	Output Example
Month	MM	01 … 12
Day	DD	01 … 31
Day of Week	ddd	Mon, Tue, Wed
Hour (24h)	HH	00 … 23
Hour (12h)	hh	01 … 12
Minute	mm	00 … 59
Second	ss	00 … 59
Microsecond	SSSSSS	000000 … 999999
AM/PM	A	AM, PM
Timezone	Z	+00:00, -07:00

9.3.5 Save Logs to File

Saving logs to a file can help preserve important information over time and aid debugging. Here's how to do it with both logging and Loguru:

9.3.5.1 With logging

Saving logs to both a file and the terminal using the logging module requires setting up separate handlers:

- `FileHandler`: writes log messages to a specified file so that they can be reviewed later
- `StreamHandler`: sends log messages to the console (stdout), allowing you to see logs in real time during execution

```
import logging

logging.basicConfig(
```

```
    level=logging.DEBUG,
    format="%(asctime)s | %(levelname)s | "
           "%(module)s:%(funcName)s:%(lineno)d - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    handlers=[
        logging.FileHandler(filename="info.log"),
        logging.StreamHandler(),
    ],
)
```

9.3.5.2 With Loguru

Loguru makes it easy to log to both a file and the terminal simultaneously. Just call the `add()` method with the file path, format, and log level. Since Loguru logs to the terminal by default, you only need to add the file handler:

```
from loguru import logger

logger.add(
    "info.log",
    format="{time:YYYY-MM-DD HH:mm:ss} | {level} | "
           "{module}:{function}:{line} - {message}",
    level="INFO",
)
```

9.3.6 Rotate and Retain Logs

Without log rotation, long-running processes like ETL jobs or model training can generate massive log files that waste disk space and are hard to manage. Automatic rotation keeps logs compact and readable.

Here's how to do it with both logging and Loguru:

9.3.6.1 With logging

To automatically rotate the log file using the logging module, you need to use `TimedRotatingFileHandler`, which has the following key parameters:

- `filename`: the file where logs are written.
- `when`: the time interval for creating new log files. Use `'S'` for seconds, `'M'` for minutes, `'H'` for hours, `'D'` for days, `'W0'` – `'W6'` for weekdays, or `'midnight'` for daily rotation at midnight.
- `interval`: how often rotation should happen based on the unit provided in `when`.
- `backupCount`: how many rotated log files to keep before old ones are deleted.

This setup gives you finer control, but requires more manual configuration than Loguru.

```
import logging
from logging.handlers import TimedRotatingFileHandler

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

handler = TimedRotatingFileHandler(
    "debug.log", when="W0", interval=1, backupCount=4
)
handler.setLevel(logging.INFO)
handler.setFormatter(
    logging.Formatter(
        "%(asctime)s | %(levelname)s | "
        "%(module)s:%(funcName)s:%(lineno)d - %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
    )
)
logger.addHandler(handler)
```

9.3.6.2 With Loguru

With Loguru, you can rotate and retain logs in a single line using the `rotation` and `retention` parameters in `add()`:

- `rotation`: when to create a new log file (e.g., size or time)
- `retention`: how long to keep old log files

```
from loguru import logger

logger.add(
    "debug.log", level="INFO",
    rotation="1 week", retention="4 weeks",
)
```

You can also customize log rotation and retention rules in Loguru using different triggers and strategies:

```
logger.add("logs/file_1.log", rotation="500 MB")
logger.add("logs/file_2.log", rotation="12:00")
logger.add("logs/file_3.log", rotation="1 week")

logger.add("logs/file_X.log", retention="10 days")

logger.add("logs/file_Y.log", compression="zip")
```

Line 1
Automatically rotate if the file exceeds 500 MB
Line 2
Create a new log file daily at noon
Line 3
Rotate weekly
Line 5
Keep logs for 10 days, then delete old ones
Line 7
Compress rotated logs to save space

9.3.7 Filter Logs by Content

Filtering log messages helps you capture only the information you care about, such as messages containing specific keywords or values. Here's how to do it with both logging and Loguru:

9.3.7.1 With logging

To filter log messages based on custom content using the built-in logging module, you need to define and attach a custom `Filter` class to the logger:

Example 9.3: main.py

```
import logging

logging.basicConfig(
    filename="hello.log", level=logging.INFO,
    format="%(asctime)s | %(levelname)s | "
    "%(module)s:%(funcName)s:%(lineno)d - %(message)s",
)
```

Example 9.4: main.py (Continued)

```
class CustomFilter(logging.Filter):
    def filter(self, record):
        return "Hello" in record.msg

# Get the root logger and add the custom filter to it
logger = logging.getLogger()
logger.addFilter(CustomFilter())

def main():
    logger.info("Hello World")
    logger.info("Bye World")

if __name__ == "__main__":
    main()
```

```
2025-07-08 16:01:08,230 | INFO | main:main:22 - Hello World
```

9.3.7.2 With Loguru

With Loguru, filtering log messages is simple: just pass a `filter` function to the `add()` method, no need to define a separate filter class.

Example 9.5: main.py

```
from loguru import logger

logger.remove()
logger.add(
    "hello.log",
    filter=lambda record: "Hello" in record["message"],
)
```

```
2025-05-03 15:12:00.180 | INFO     | __main__:main:8 - Hello World
```

9.3.8 Better Exception Logging

When exceptions occur, logging can help you understand not only what went wrong, but also where and why. Here's how traditional logging compares with Loguru when it comes to capturing exception details:

9.3.8.1 With logging

To catch and log exceptions using the built-in logging module, you typically wrap your code in a try-except block and call `logging.exception()` to capture the traceback:

Example 9.6: main.py

```
import logging

def divide(a, b):
    return a / b

def main():
    try:
        divide(1, 0)
    except ZeroDivisionError:
        logging.exception("Division by zero")

main()
```

This traceback doesn't show the values of `a` and `b`, leaving you to guess what inputs triggered the error.

```
Traceback (most recent call last):
  File ".../main.py", line 16, in nested
    division(1, c)
  File ".../main.py", line 11, in division
    return a / b
           ~~^~~
ZeroDivisionError: division by zero
```

9.3.8.2 With Loguru

Loguru improves debugging by capturing the full stack trace and the state of local variables at each level.

Example 9.7: main.py

```
from loguru import logger

def division(a, b):
    return a / b
```

Example 9.8: main.py (Continued)

```
def nested(c):
    try:
        division(1, c)
    except ZeroDivisionError:
        logger.exception("ZeroDivisionError")

if __name__ == "__main__":
    nested(0)
```

In the traceback, Loguru shows that `a` is 1 and `b` is 0, making it immediately clear what inputs caused the failure.

```
> File ".../main.py", line 14, in <module>
    nested(0)
    └ <function nested at 0x106492520>

  File ".../main.py", line 10, in nested
```

```
    division(1, c)
    |           └ 0
    └ <function division at 0x105051800>

  File ".../main.py", line 5, in division
    return a / b
           |   └ 0
           └ 1

ZeroDivisionError: division by zero
```

You can also capture and display full tracebacks in any function simply by adding the `@logger.catch` decorator.

Example 9.9: main.py

```python
from loguru import logger

def divide(a, b):
    return a / b

@logger.catch
def main():
    divide(1, 0)

main()
```

```
> File ".../main.py", line 14, in <module>
    nested(0)
    └ <function nested at 0x100a5df80>

  File ".../main.py", line 10, in nested
    division(1, c)
    |           └ 0
    └ <function division at 0x1003b5b20>

  File ".../main.py", line 5, in division
    return a / b
           |   └ 0
           └ 1

ZeroDivisionError: division by zero
```

9.3.9 Pretty Logging with Colors

9.3.9.1 With logging

Traditional logging does not support color formatting out of the box. You would need to install and configure a third-party library like colorlog to manually define colorized output formats.

```
import logging
from colorlog import ColoredFormatter

formatter = ColoredFormatter(
    "%(log_color)s%(asctime)s | %(levelname)s | %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    log_colors={
        'DEBUG':    'cyan',
        'INFO':     'green',
        'WARNING':  'yellow',
        'ERROR':    'red',
        'CRITICAL': 'bold_red',
    }
)

handler = logging.StreamHandler()
handler.setFormatter(formatter)

logger = logging.getLogger(__name__)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

logger.info("Colorized info message")
```

9.3.9.2 With Loguru

By default, Loguru outputs logs with colorized formatting in the terminal. You can also customize the color for each log level using the `colorize` option and the `{level.color}` formatting token:

```
from loguru import logger
import sys
```

```python
logger.remove()
logger.add(
    sys.stdout,
    colorize=True,
    format="<green>{time:YYYY-MM-DD HH:mm:ss}</green> | "
           "<level>{level}</level> | <cyan>{message}</cyan>",
)

if __name__ == "__main__":
    logger.debug("This is a debug message")
    logger.info("This is an info message")
    logger.warning("This is a warning message")
    logger.error("This is an error message")
```

```
2025-05-03 15:55:18 | DEBUG   | This is a debug message
2025-05-03 15:55:18 | INFO    | This is an info message
2025-05-03 15:55:18 | WARNING | This is a warning message
2025-05-03 15:55:18 | ERROR   | This is an error message
```

Here's a quick reference of available color and style tags you can use in your format strings:

Color	Abbreviation	Style	Abbreviation
Black	k	Bold	b
Blue	e	Dim	d
Cyan	c	Normal	n
Green	g	Italic	i
Magenta	m	Underline	u
Red	r	Strike	s
White	w		
Yellow	y		

You can also combine colors and styles in your format string by nesting tags. For example: `<red><bold>{message}</bold></red>` will show the message in bold red text.

9.4 Best Practices For Exception Handling

9.4.1 Use Specific Exceptions

Avoid using generic exceptions like `Exception` because they don't specify what went wrong, making debugging and error handling more difficult.

In the example below, the generic exception hides whether the error stems from a missing file or empty data.

```
import pandas as pd
import numpy as np

def process_sales_data(filepath: str) -> pd.DataFrame:
    try:
        df = pd.read_csv(filepath)
        total_sales = df['sales'].sum()
        daily_average = df.groupby('date')['sales'].mean()
        return df
    except Exception as e:
        print(f"Error processing data: {e}")

if __name__ == "__main__":
    result = process_sales_data("data/sales_data.csv")
```

```
Error processing data: 'sales'
```

Use specific exceptions instead of generic ones to get clearer error information. The code below catches specific exceptions

(`FileNotFoundError`, `EmptyDataError`, `KeyError`) with informative messages, using a generic `Exception` handler only as a fallback.

```
def process_sales_data(filepath: str) -> pd.DataFrame:
    try:
        df = pd.read_csv(filepath)
        total_sales = df["sales"].sum()
        return df
    except FileNotFoundError:
        logging.error(f"File '{filepath}' not found")
    except pd.errors.EmptyDataError:
        logging.error(f"File '{filepath}' is empty")
    except KeyError:
        logging.error(f"Column 'sales' not found")
    except Exception as e:
        logging.error(f"Unexpected error: {e}")

if __name__ == "__main__":
    result = process_sales_data("data/sales_data.csv")
```

```
2025-07-29 15:42:16,410 - ERROR - Column 'sales' not found
```

9.4.2 Use Else Outside of the Try Block

Avoid mixing operations that might fail with operations that are guaranteed to work in the same block.

The code below mixes operations that might fail (`sum(nums)`) with operations that are guaranteed to work (division and printing) in the `try` block, making it harder to identify the source of a potential `TypeError`.

```
nums = [1, 2, "3"]
try:
    sum_nums = sum(nums)
    mean_nums = sum_nums / len(nums)
    print(f"The mean of the numbers is {mean_nums}.")
except TypeError as e:
    raise TypeError("Items in the list must be numbers") from e
```

```
TypeError
      2 try:
----> 3     sum_nums = sum(nums)
      4     mean_nums = sum_nums / len(nums)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Instead, use the `else` clause to separate operations that might fail from those guaranteed to work. This approach:

- Makes it easier to identify the source of a `TypeError`
- Prevents mean calculation if the sum operation fails
- Creates cleaner, more maintainable code structure

```
nums = [1, 2, "3"]
try:
    sum_nums = sum(nums)
except TypeError as e:
    raise TypeError("Items in the list must be numbers") from e
else:
    mean_nums = sum_nums / len(nums)
    print(f"The mean of the numbers is {mean_nums}.")
```

```
TypeError
      2 try:
----> 3     sum_nums = sum(nums)
      4 except TypeError as e:

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

9.5 Key Takeaways

1. Logging vs print:
    - Logging provides structured, timestamped output with severity levels
    - Logs can be saved to files and filtered by level
    - Logging includes metadata like module name, function, and line number
2. Loguru benefits:

    - Provides beautiful, colorful output out of the box
    - Simpler syntax compared to traditional logging
    - Better exception tracking with variable values
    - Easy configuration for file rotation and retention
    - Built-in support for filtering and formatting
3. Exception handling best practices:
    - Use specific exceptions instead of catching all exceptions
    - Use the else clause to separate error-prone code from safe operations.

10 Data Validation

10.1 What Is Data Validation?

Data validation is the process of ensuring that data meets specified criteria, formatting rules, and quality standards before processing. Data validation helps maintain data integrity and prevents errors in downstream operations.

10.2 Why Is Data Validation Essential?

10.2.1 Validate Data Schema

Schema validation identifies data type mismatches and missing fields before processing begins, ensuring your pipeline receives correctly formatted data.

Consider a scenario where you have a dataset with an “age” column that’s expected to be an integer, but some values are stored as strings:

```
import pandas as pd
import numpy as np

# Create sample data with mixed age types
df = pd.DataFrame(
    {
```

```
        "customer_id": [1, 2, 3, 4, 5],
        "age": [25, "30", 35, "40", 45], # Some values are strings
        "transaction_amount": [100.00, 50.00, 75.00, 125.00,
200.00],
    }
)

print(df)
try:
    young_customers = df[df["age"] < 35]
except TypeError as e:
    print("TypeError:", e)
```

```
   customer_id age  transaction_amount
0            1  25               100.0
1            2  30                50.0
2            3  35                75.0
3            4  40               125.0
4            5  45               200.0
TypeError: '<' not supported between instances of 'str' and 'int'
```

With schema validation, we can confirm the “age” column contains integers. Non-integer values flag the data as invalid, requiring investigation (Figure 10.1).

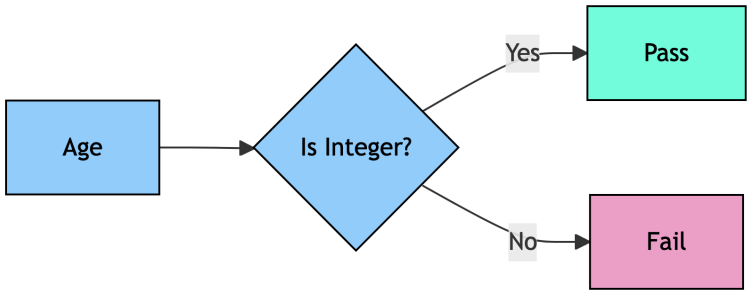


Figure 10.1: Data schema validation

10.2.2 Ensure Consistent Data Relationships

Cross-column validation catches calculation errors and data inconsistencies, preventing flawed analysis and incorrect business decisions.

Consider a dataset tracking employee salaries and bonuses with a calculation error:

- Employee `E002` shows `total_compensation` of 72000
- The correct value should be 72500 (65000 salary + 7500 bonus)

employee_id	salary	bonus	total_compensation
E001	50000	5000	55000
E002	65000	7500	72000
E003	45000	4000	49000

With schema validation, we can check that `salary` and `bonus` columns sum to `total_compensation` for each employee (Figure 10.2). Mismatched calculations flag the data as invalid for investigation.

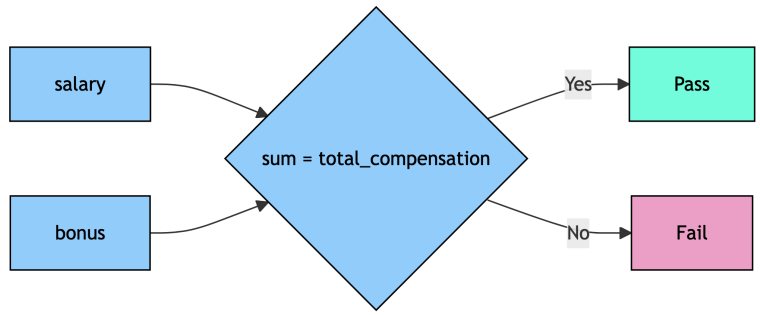


Figure 10.2: Data consistent data relationships validation

10.2.3 Detect Outliers

Establishing baseline patterns and ranges in your data helps identify anomalies that may indicate data quality issues or interesting

insights. Data outliers aren't always erroneous, but you need to rule out that possibility.

Consider analyzing customer transaction data with unusual spending patterns. Figure 10.3 shows suspicious transactions (pink stars) with significantly higher amounts than normal transactions (blue circles).

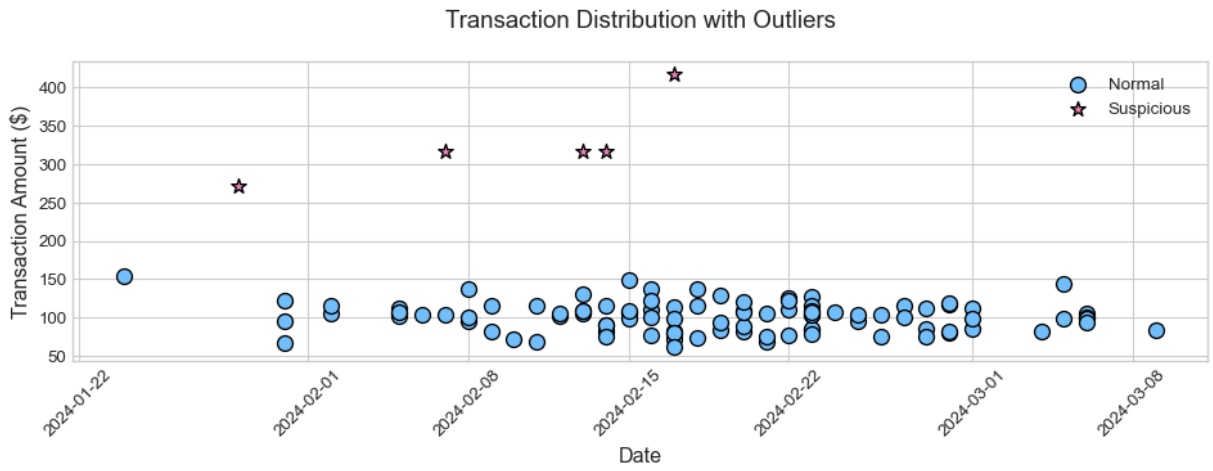


Figure 10.3: Transaction distribution showing normal and suspicious transactions

With schema validation, we can establish rules that flag transaction amounts outside $40-$160 as outliers requiring manual investigation (Figure 10.4).

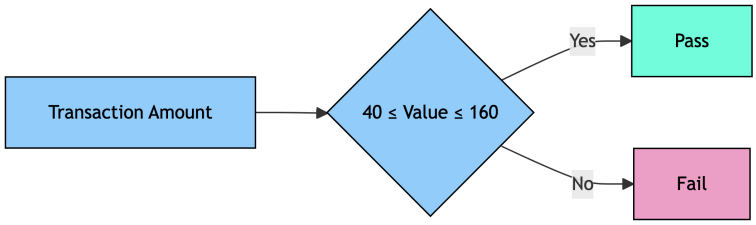


Figure 10.4: Data distribution validation

10.2.4 Validate Data Freshness

Working with outdated data causes serious problems in data science projects:

- Inaccurate insights and predictions that don't reflect current reality
- Flawed business decisions based on stale information
- Poor model performance due to training on obsolete patterns
- Wasted time and resources analyzing irrelevant historical data

Data freshness validation prevents these issues by ensuring you work with current, relevant information.

Consider this example: a recommender system built on 2019 transaction data fails when shopping patterns shift due to events like COVID-19. Figure 10.5 demonstrates how the pandemic's online shopping surge (pink line) exceeded pre-pandemic forecasts (blue dashed line).

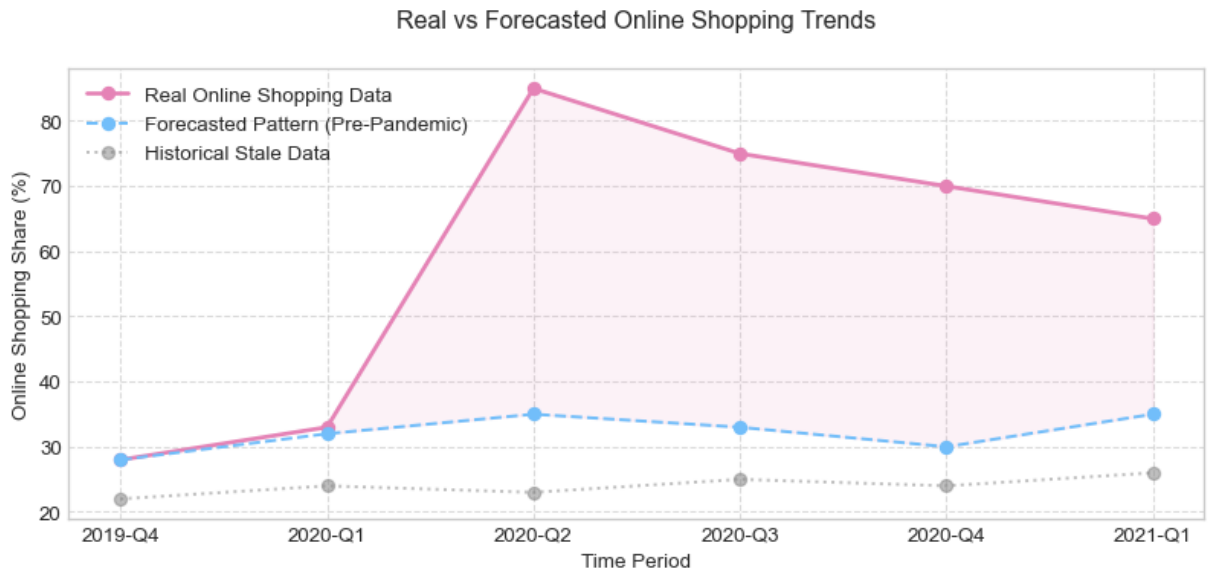


Figure 10.5: Real vs forecasted online shopping trends during COVID-19

With schema validation, we can establish rules that flag transaction data from before 2022 as stale requiring further investigation (Figure 10.6).

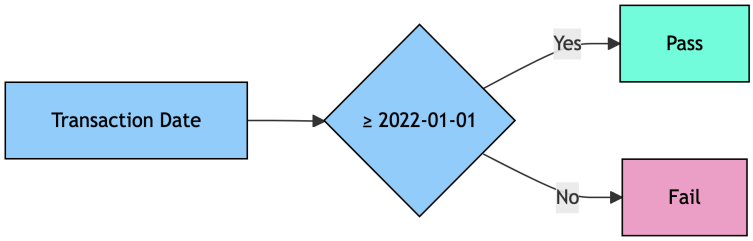


Figure 10.6: Data freshness validation

10.2.5 Detect Missing Values

Missing value validation catches data gaps early, ensuring your analyses and models use complete, reliable data for accurate insights.

Consider sales data with missing values for certain product categories or regions. As shown in Figure 10.7, missing values (blue bars with “Missing Data” labels) create gaps in regional analysis that could lead to incorrect business decisions.

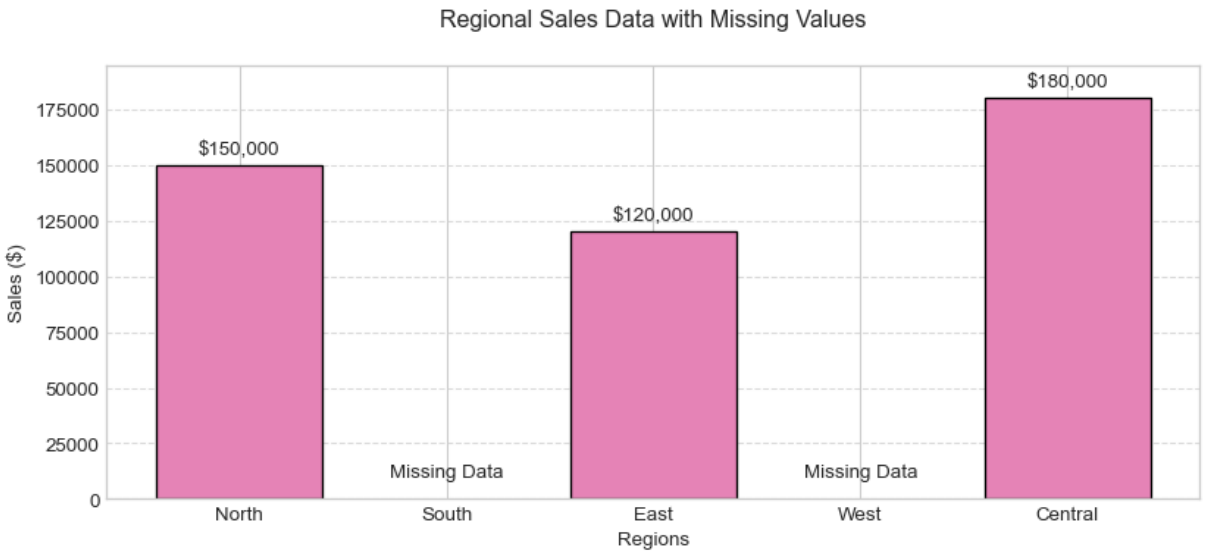


Figure 10.7: Regional sales data with missing values highlighted

With schema validation, we can establish rules that flag sales data with missing values as invalid requiring further investigation (Figure 10.8).

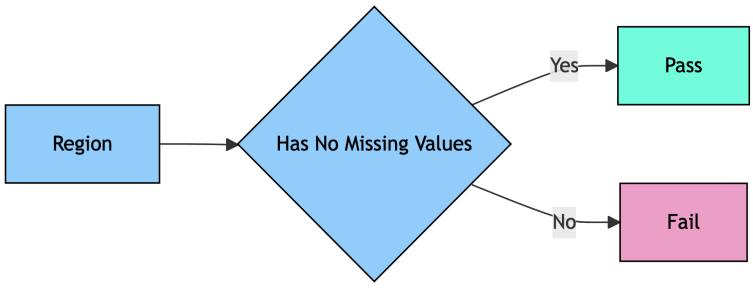


Figure 10.8: Data missing values validation

10.2.6 Validate Data Uniqueness

Data uniqueness validation ensures your analyses and models are based on clean, non-redundant information, helping you draw accurate insights and make well-informed decisions.

Consider a customer database with 10,000 records containing 800 duplicate entries, leaving only 9,200 unique customers. As shown in Figure 10.9, duplicates significantly impact metrics:

- With duplicates: 1,500 churned customers out of 10,000 total records = 15% churn rate
- Without duplicates: 1,500 churned customers out of 9,200 unique customers = 16.3% churn rate

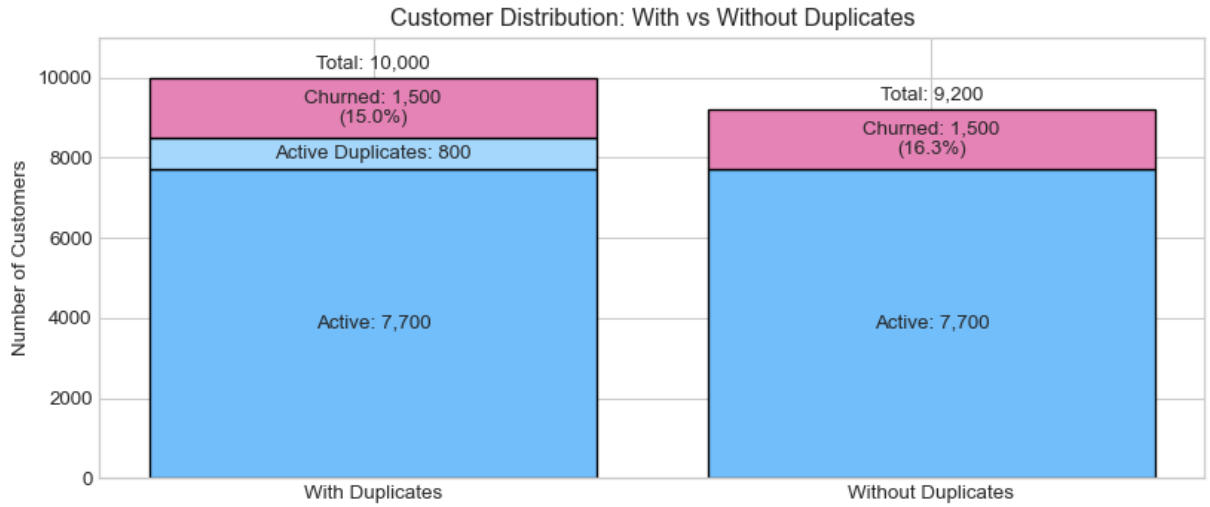


Figure 10.9: Customer distribution comparison with and without duplicates

With schema validation, we can establish rules that flag customer data with duplicates as invalid requiring further investigation (Figure 10.10).

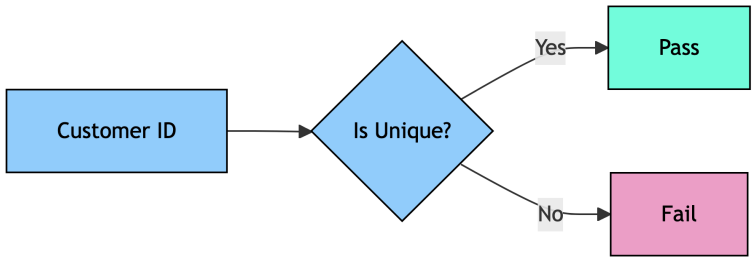


Figure 10.10: Data uniqueness validation

10.2.7 Validate Data Volume

Data volume validation ensures you have enough information to develop reliable forecasting models and make well-informed decisions.

Consider predicting future market trends based on historical data. As shown in Figure 10.11, small datasets (pink dots) produce forecasting models (dashed pink line) that fail to capture underlying patterns (black line). Larger datasets (blue dots) generate forecasts (solid blue line) that closely follow true market trends.

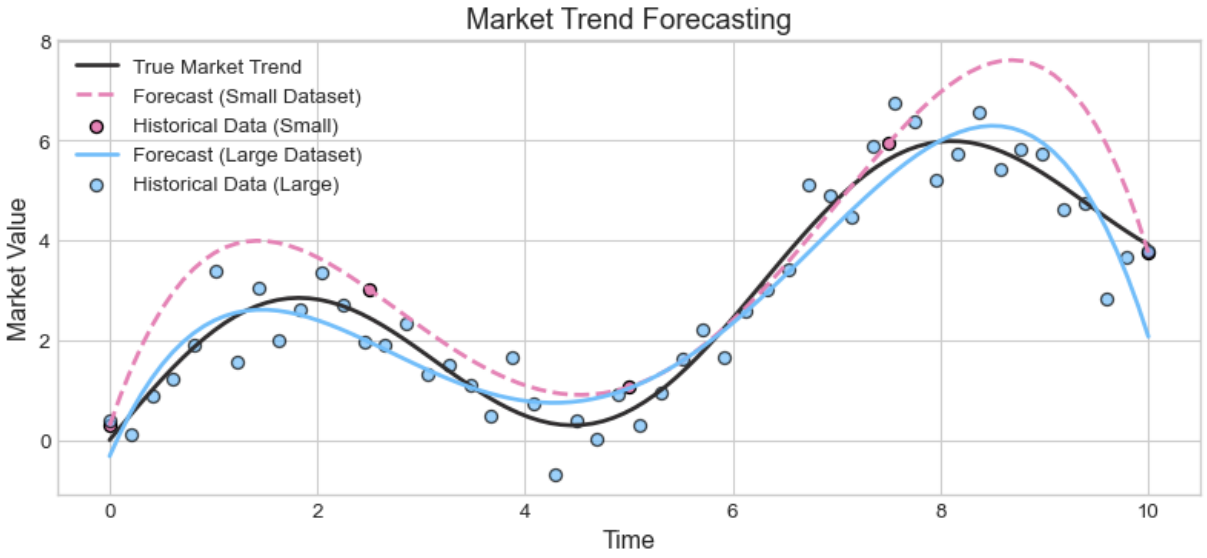


Figure 10.11: Market trend forecasting comparison between small and large datasets

With schema validation, we can establish rules that flag historical data with less than 1000 rows as invalid requiring further investigation (Figure 10.12).

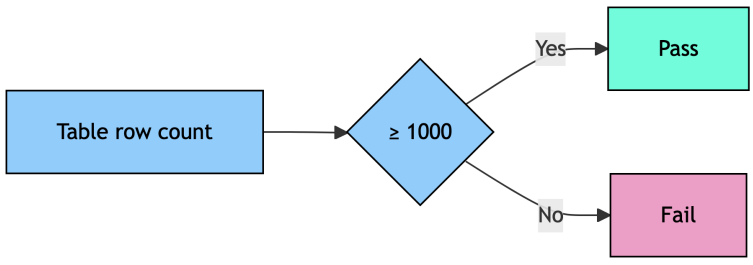


Figure 10.12: Data volume validation

10.3 Data Validation Made Easy with Pandera

Now that we've discussed the importance of data validation, let's dive deeper into how to validate your data effectively with Pandera.

Pandera is an open-source data validation library that provides a simple and expressive way to define data validation rules. Pandera supports a wide range of data formats, including CSV, JSON, and SQL, and can be easily integrated into your data pipeline.

To install Pandera, type:

```
pip install pandera
```

10.3.1 Basic Building Blocks

To learn how Pandera works, let's start with creating a bank dataset:

```
import pandas as pd

# Create sample data with mixed age types
df = pd.DataFrame(
    {
        "customer_id": [1, 2, 3, 4, 5],
        "age": [25, 30, 35, 40, 45],
        "transaction_amount": [100.0, 50.0, 75.0, 125.0, 200.0],
    }
)
```

We want to ensure the following conditions are met:

- `customer_id` is an integer, greater than 1, and is unique.
- `age` is an integer, greater than 0 and less than 120.
- `transaction_amount` is a float and greater than 0.

To validate the data against these conditions, we need to define a schema using Pandera. Here are the three main classes that Pandera provides for schema definition:

- A `DataFrameSchema` defines the structure and constraints of a Pandas DataFrame.
- A `Column` represents a single column in a DataFrame. A `Column` is used to define the properties and constraints of a column, such as its data type, allowed values, and relationships to other columns.
- A `Check` represents a single constraint or validation rule for a column or DataFrame.

Here's an example:

```
import pandera.pandas as pa



# Define the schema
schema = pa.DataFrameSchema(
    {
        "customer_id": pa.Column(
            int, checks=pa.Check.ge(1), unique=True
        ),
        "age": pa.Column(
            int, checks=pa.Check.between(0, 120)
        ),
        "transaction_amount": pa.Column(
            float, checks=pa.Check.ge(0)
        ),
    }
)
# Validate the DataFrame
validated_df = schema.validate(df)
print(validated_df)
```

Line 9
Checks that `customer_id` is an integer, greater than or equal to 1, and is unique.
Line 12

Checks that `age` is an integer, greater than 0 and less than 120.
Line 15
Checks that `transaction_amount` is a float and greater than 0.
Line 19
Validates a DataFrame `df` against this schema.

If all validations pass, the `DataFrameSchema` object returns the validated DataFrame.

If any of these conditions are not met, Pandera will raise a `SchemaError`.

```
# Example of validation failure
invalid_df = pd.DataFrame(
    {
        "customer_id": [1, 2, 2, 4, 5],  # Duplicate ID
        "age": [25, 150, -5, 40, 45],  # Invalid ages
        "transaction_amount": [100.00, 50.00, 75.00, 125.00,
200.00],
    }
)

# This will raise SchemaError
try:
    schema.validate(invalid_df)
except pa.errors.SchemaError as err:
    print('SchemaError:', err)
```

```
SchemaError: series 'customer_id' contains duplicate values:
1    2
2    2
Name: customer_id, dtype: int64
```

10.3.2 Checks

`Check` objects accept a function as a required argument, which is expected to take a `pa.Series` input and output a `boolean` or a `Series` of boolean values. For the check to pass, all of the elements in the boolean series must evaluate to `True`.

The following code shows how to create a `Check` that validates even numbers:

```
check_is_even = pa.Check(lambda s: s % 2 == 0)

schema = pa.DataFrameSchema(
    {"column1": pa.Column(int, check_is_even)}
)
schema.validate(pd.DataFrame({"column1": [2, 4, 6, 8]}))
```

	column1
0	2
1	4
2	6
3	8

Figure 10.13 shows how the `Check` function evaluates each value and determines pass/fail status.

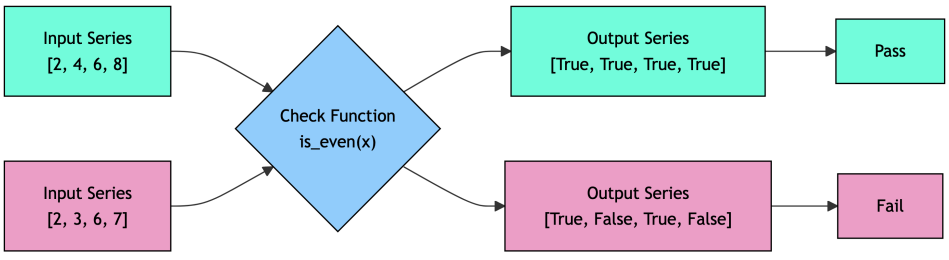


Figure 10.13: Data validation check example

10.3.2.1 Built-in Checks

Pandera provides a variety of built-in checks, making it easier to check data. Let's create a schema with some built-in checks:

```
from datetime import datetime

customer_schema = pa.DataFrameSchema(
    {
        "customer_id": pa.Column(
            str, checks=pa.Check.str_length(min_value=5)
        ),
        "email": pa.Column(str, checks=pa.Check.str_contains("@")),
        "signup_date": pa.Column(
            datetime, checks=pa.Check.le(datetime.now())
        ),  # Date not in future
    }
)
```

Use the `validate` method to validate the data against the schema:

```
customer = pd.DataFrame(
    {
        "customer_id": ["CUST01", "CUST02", "CUST03"],
        "email": ["john@mail.com", "jane@mail.com",
"bob@mail.com"],
        "signup_date": ["2023-01-01", "2023-02-15", "2023-03-30"],
    }
)
customer["signup_date"] = pd.to_datetime(customer["signup_date"])

# Validate data
validated_df = customer_schema.validate(customer)
print("Validation passed!")
print(validated_df)
```

```
Validation passed!
  customer_id          email signup_date
0      CUST01  john@mail.com  2023-01-01
1      CUST02  jane@mail.com  2023-02-15
2      CUST03   bob@mail.com  2023-03-30
```

See the Check API reference for a complete list of built-in checks.

10.3.2.2 Column Check Groups

Column checks support grouping by a different column so that you can make assertions about subsets of the column of interest.

In the following example, the `groupby="store"` parameter groups profit data by store location, then compares mean values between CA and NY stores.

```
# Create sample sales data
df = pd.DataFrame(
    {
        "store": ["NY", "CA", "NY", "CA"],
        "profit": [200.0, 300.0, 300.0, 400.0],
    }
)

# Define schema with wide check using groupby
schema = pa.DataFrameSchema(
    {
        "store": pa.Column(str),
        "profit": pa.Column(
            float,
            # Check CA stores have higher average profit than NY
            pa.Check(
                lambda g: g["CA"].mean() > g["NY"].mean(),
                groupby="store",
            ),
        ),
    }
)

# Validate the DataFrame
validated_df = schema.validate(df)
print("Validation passed!")
```

```
Validation passed!
```

Figure 10.14 illustrates this example.

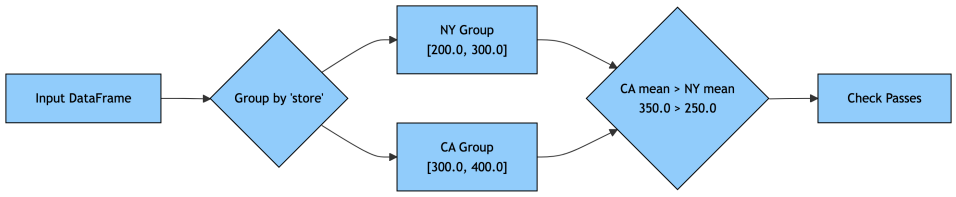


Figure 10.14: Data validation grouped validation

10.3.2.3 Wide Checks

Pandera also supports validating relationships between columns using the `checks` parameter at the `DataFrameSchema` level.

In the following example, the wide check applies a row-wise calculation to verify the profit formula across the entire DataFrame.

```
# Create sample sales data
df = pd.DataFrame({
    "revenue": [1000.0, 1500.0, 1200.0],
    "expenses": [800.0, 1200.0, 900.0],
    "profit": [200.0, 300.0, 300.0],
})

# Define schema with wide check
schema = pa.DataFrameSchema(
    columns={
        "revenue": pa.Column(float),
        "expenses": pa.Column(float),
        "profit": pa.Column(float),
    },
    checks=pa.Check(
        lambda df: df["profit"] == df["revenue"] - df["expenses"]
    ),
)

validated_df = schema.validate(df)
print("Validation passed!")
```

```
Validation passed!
```

10.3.3 Validation Decorator

10.3.3.1 Check Input

To validate the inputs of a function before calling that function, use the `check_input` decorator. Let's create a schema for the inputs of the `calculate_grade` function:

```
from pandera import check_input



input_schema = pa.DataFrameSchema(
    {
        "name": pa.Column(str),
        "age": pa.Column(int, pa.Check.between(0, 120)),
        "score": pa.Column(float, pa.Check.between(0, 100)),
    }
)



@check_input(input_schema)
def calculate_grade(data: pd.DataFrame):
    data["grade"] = pd.cut(
        data["score"],
        bins=[0, 70, 80, 90, 100],
        labels=["F", "C", "B", "A"],
        include_lowest=True,
    )
    return data
```

If the input data conforms to the schema, the function will return the data with the `grade` column added:

```
df = pd.DataFrame(
    {
        "name": ["John", "Jane", "Bob"],
        "age": [25, 30, 35],
        "score": [95.5, 88.3, 92.7],
    }
)
```

```
result = calculate_grade(df)
print(result)
```

```
   name  age  score grade
0  John   25   95.5     A
1  Jane   30   88.3     B
2   Bob   35   92.7     A
```

If the input data does not conform to the schema, an error will be raised, preventing the function from executing with invalid data.

```
invalid_df = pd.DataFrame(
    {
        "name": ["John", "Jane", "Bob"],
        "age": [25, 30, 35],
        "score": [95.5, 88.3, 120.0],
    }
)

try:
    result = calculate_grade(invalid_df)
except pa.errors.SchemaError as err:
    print("SchemaError:", err)
```

```
SchemaError: error in check_input decorator of function
'calculate_grade': Column 'score' failed element-wise
validator number 0: in_range(0, 100) failure cases: 120.0
```

10.3.3.2 Check Output

To validate the output of a function, use the `check_output` decorator. Let's create an output schema for the `calculate_grade` function:

```
from pandera import check_output

output_schema = pa.DataFrameSchema(
    {
        "name": pa.Column(str),
        "age": pa.Column(int, pa.Check.between(0, 120)),
        "score": pa.Column(float, pa.Check.between(0, 100)),
        "grade": pa.Column(
            str, pa.Check(lambda x: x.isin(["A", "B", "C", "F"]))
```

```
        ),
    }
)



@check_input(input_schema)
@check_output(output_schema)
def calculate_grade(data: pd.DataFrame):
    data["grade"] = pd.cut(
        data["score"],
        bins=[0, 70, 80, 90, 100],
        labels=["F", "C", "B", "A"],
        include_lowest=True,
    )
    return data
```

If the output data does not conform to the schema, an error will be raised:

```
@check_output(output_schema)
def calculate_grade(data: pd.DataFrame):
    data["grade"] = pd.cut(
        data["score"],
        bins=[0, 70, 80, 90, 100],
        labels=["F", "C", "B", "X"],
        include_lowest=True,
    )
    return data



df = pd.DataFrame(
    {
        "name": ["John", "Jane", "Bob"],
        "age": [25, 30, 35],
        "score": [95.5, 88.3, 92.7],
    }
)

try:
    result = calculate_grade(df)
except pa.errors.SchemaError as err:
    print("SchemaError:", err)
```

```
SchemaError: error in check_output decorator of function
'calculate_grade':
Column 'grade' failed element-wise validator number 0:
<Check <lambda>> failure cases: X, X
```

10.3.3.3 Check Both Inputs and Outputs

To check both inputs and outputs, use the `check_io` decorator in Pandera. Let's create a schema for the inputs and outputs of the `calculate_grade` function:

```
from pandera import check_io



@check_io(data=input_schema, out=output_schema)
def calculate_grade(data: pd.DataFrame):
    data["grade"] = pd.cut(
        data["score"],
        bins=[0, 70, 80, 90, 100],
        labels=["F", "C", "B", "A"],
        include_lowest=True,
    )
    return data
```

If the input and output data conform to the schema, the function will return the data with the `grade` column added:

```
df = pd.DataFrame(
    {
        "name": ["John", "Jane", "Bob"],
        "age": [25, 30, 35],
        "score": [95.5, 88.3, 92.7],
    }
)
result = calculate_grade(df)
print(result)
```

```
   name  age  score grade
0  John   25   95.5     A
1  Jane   30   88.3     B
2   Bob   35   92.7     A
```

If either the inputs or the output do not conform to their respective schemas, Pandera will raise an error:

```
invalid_df = pd.DataFrame(
    {
        "name": ["John", "Jane", "Bob"],
        "age": [25, 30, 35],
        "score": [95.5, 88.3, 120],
    }
)

try:
    result = calculate_grade(invalid_df)
except pa.errors.SchemaError as err:
    print("SchemaError:", err)
```

```
SchemaError: Column 'score' failed element-wise validator number 0:
in_range(0, 100) failure cases: 120.0
```

10.3.4 Other Arguments for Column Validation

10.3.4.1 Deal with Null Values

Pandera raises errors for null values by default. Use `nullable=True` to allow missing values in a column.

In the following example, setting `nullable=True` allows the schema to accept missing values in the `name` and `age` columns.

```
schema = pa.DataFrameSchema(
    {
        "id": pa.Column(int),  # Does not allow nulls
        "name": pa.Column(str, nullable=True),  # Allows nulls
        "age": pa.Column(float, nullable=True),  # Allows nulls
    }
)

df = pd.DataFrame(
```

```
    {
        "id": [1, 2, 3],
        "name": ["John", None, "Mary"],
        "age": [25.0, 30.0, None],
    }
)

validated_df = schema.validate(df)
print("Validation passed!")
```

```
Validation passed!
```

10.3.4.2 Deal with Duplicates

Pandera allows duplicates by default. Use `unique=True` to reject duplicate values in a column.

In the following example, setting `unique=True` causes Pandera to raise a `SchemaError` when the `id` column contains duplicate values.

```
# Define schema with unique constraint
schema = pa.DataFrameSchema(
    {
        "id": pa.Column(int, unique=True),  # Must be unique
        "name": pa.Column(str),  # Duplicates allowed
    }
)



df = pd.DataFrame(
    {"id": [1, 1, 2], "name": ["John", "Jane", "Mary"]}
)

try:
    validated_df = schema.validate(df)
except pa.errors.SchemaError as e:
    print("SchemaError:", e)
```

```
SchemaError: series 'id' contains duplicate values:
0    1
1    1
Name: id, dtype: int64
```

10.3.4.2.1 Required Columns

Pandera requires all schema columns to be present by default. Use `required=False` to make columns optional.

In the following example, setting `required=False` allows the schema to validate successfully even when the `age` column is missing.

```
# Define schema with required columns
schema = pa.DataFrameSchema(
    {
        "id": pa.Column(int),  # Required column
        "name": pa.Column(str),  # Required column
        "age": pa.Column(int, required=False),  # Optional column
    }
)



df = pd.DataFrame(
    {"id": [1, 2, 3], "name": ["John", "Jane", "Mary"]}
)

validated_df = schema.validate(df)
print("Validation passed!")
```

```
Validation passed!
```

10.3.4.3 Convert Data Types

Pandera disables type conversion by default. Use `coerce=True` to automatically convert column data types, or raise an error if conversion fails.

In the following example, since coercion from string to integer in the `id` column is not possible, Pandera raises an error.

```
schema = pa.DataFrameSchema(
    {
        "id": pa.Column(int, coerce=True),
        "price": pa.Column(float, coerce=True),
```

```
    }
)

df = pd.DataFrame(
    {
        "id": ["12.34", "2", "3"],  # Will fail coercion to int
        "price": ["10.99", "20.50", "15.75"],
    }
)

try:
    validated_df = schema.validate(df)
except pa.errors.SchemaError as e:
    print("SchemaError:", e)
```

```
SchemaError: Error while coercing 'id' to type int64:
Could not coerce <class 'pandas.core.series.Series'> into type
int64:
   index failure_case
0      0        12.34
```

10.3.4.4 Match Patterns

To validate multiple columns with shared patterns, set `regex=True` instead of defining each column individually.

In the following example, `regex=True` applies the same validation rules to all columns matching the `score_.*` pattern.

```
# Define schema using regex to match column patterns
schema = pa.DataFrameSchema({
    # Match any column starting with 'score_'
    'score_.*': pa.Column(float, regex=True, nullable=True),
    # Regular columns without regex
    'student_id': pa.Column(int),
    'name': pa.Column(str)
})

df = pd.DataFrame({
    'student_id': [1, 2, 3],
    'name': ['John', 'Mary', 'Bob'],
    'score_math': [85.5, 90.0, None],
```

```
    'score_science': [88.0, None, 92.5],
    'score_history': [78.5, 88.5, 95.0],
})

validated_df = schema.validate(df)
print("Validation passed!")
```

```
Validation passed!
```

10.3.5 Schema Model

Pandera also supports creating models using similar syntax to Pydantic models.

To demonstrate how to create a `DataFrameModel`, let's consider the following schema with `DataFrameSchema`:

```
# Define the schema using DataFrameSchema
customer_schema = pa.DataFrameSchema(
    {
        "customer_id": pa.Column(
            str,
            checks=pa.Check.str_length(
                min_value=5, max_value=10
            ),
        ),
        "email": pa.Column(
            str, checks=pa.Check.str_contains("@")
        ),
        "signup_date": pa.Column(
            str,
            checks=pa.Check(
                lambda s: pd.to_datetime(s) <= pd.Timestamp.now()
            ),
        ),
    }
)
```

Transform this schema into a `DataFrameModel` using three key changes:

- Apply `Series[str]` type annotations instead of `Column(str)`
- Apply `@pa.check` decorators instead of lambda functions
- Organize validations within a reusable class structure

The `DataFrameModel` approach provides IDE support, static type checking, and explicit validation logic.

```
from pandera.typing import Series


class CustomerSchema(pa.DataFrameModel):
    customer_id: Series[str] = pa.Field(
        str_length={"min_value": 5, "max_value": 10}
    )
    email: Series[str] = pa.Field(str_contains="@")
    signup_date: Series[str]

    @pa.check("signup_date")
    def check_date_not_in_future(
        cls, signup_date: Series[str]
    ) -> Series[bool]:
        return pd.to_datetime(signup_date) < pd.Timestamp.now()
```

Validate the data against the schema using `DataFrameModel` is similar to using `DataFrameSchema`:

```
# Example data
customer_data = pd.DataFrame(
    {
        "customer_id": ["CUST01", "CUST02"],
        "email": ["john@mail.com", "jane@mail.com"],
        "signup_date": ["2023-01-01", "2023-02-15"],
    }
)

# Validate data
validated_df = customer_schema.validate(customer_data)
print("Validation passed!")
```

```
Validation passed!
```

You can also use `DataFrameModel` with the `@pa.check_types` decorator to validate function inputs and outputs against the schema defined in the function signature.

For example, you can set up the schema for the function inputs and outputs.

```
from pandera.typing import Series, DataFrame
import hashlib



class CustomerSchema(pa.DataFrameModel):
    customer_id: Series[str] = pa.Field(
        str_length={"min_value": 5, "max_value": 10}
    )
    email: Series[str] = pa.Field(str_contains="@")



class AnonymizedCustomerSchema(pa.DataFrameModel):
    customer_id: Series[str] = pa.Field(
        str_length={"min_value": 5, "max_value": 10}
    )
    anonymized_email: Series[str] = pa.Field(
        str_length={"min_value": 32, "max_value": 32}
    )
```

Or use the schema to validate the function inputs and outputs.

```
@pa.check_types
def anonymize_customer_data(
    df: DataFrame[CustomerSchema],
) -> DataFrame[AnonymizedCustomerSchema]:
    """
    Returns a DataFrame with hashed emails for data privacy
    """
    df = df.copy()

    # Hash email addresses
    df["anonymized_email"] = df["email"].apply(
        lambda x: hashlib.md5(x.encode()).hexdigest()
    )
```

```
    # Drop original email column
    df = df.drop("email", axis=1)

    return df
```

10.3.6 Export and Load From a YAML File

10.3.6.1 Export to YAML

Pandera can export Python schemas to YAML format. This enables sharing validation requirements with team members who don't use Python.

The following example exports a schema to YAML using the `to_yaml()` method:

```
from pathlib import Path



# Define the schema
schema = pa.DataFrameSchema(
    {
        "customer_id": pa.Column(
            int, checks=pa.Check.ge(1), unique=True
        )
    }
)

# Get a YAML object
yaml_schema = schema.to_yaml()

# Save to a file
f = Path("data/schema.yml")
f.touch()
f.write_text(yaml_schema)
```

The resulting `schema.yml` file will have the following structure:

```
# schema.yml
schema_type: dataframe
version: 0.17.2
columns:
  customer_id:
    title: null
    description: null
    dtype: int64
    nullable: false
    checks:
      ge: 1
    unique: true
    coerce: false
    required: true
    regex: false
checks: null
index: null
```

10.3.6.2 Load from YAML

Use `pa.io.from_yaml(yaml_schema)` to load a schema from a YAML file into your Python code.

```
from pathlib import Path

f = Path("data/schema.yml")

with f.open() as file:
    yaml_schema = file.read()

schema = pa.io.from_yaml(yaml_schema)
```

10.4 Best Practices for Data Validation

10.4.1 Validate Data at the Point of Entry

Validating data at entry points prevents wasted computation and provides immediate feedback on data quality issues.

In the following example, data type errors are discovered late during computation, after processing has already begun.

```
def analyze_sales_data(sales_df: pd.DataFrame) -> dict:
    # Problems only discovered during processing
    revenue = sales_df["price"] * sales_df["quantity"]

    return {
        "total_revenue": revenue.sum(),
        "max_sale": sales_df["quantity"].max(),
    }



if __name__ == "__main__":
    # Data with issues
    data = pd.DataFrame(
        {
            "price": [50, 100, "invalid", 75],
            "quantity": [5, 3, 2, "error"],
        }
    )
    try:
        results = analyze_sales_data(data)
        print(results)
    except Exception as e:
        print(f"Error during analysis: {e}")
```

```
Error during analysis: unsupported operand type(s) for +: 'int' and
'str'
```

Instead, validate data immediately when it enters the system using schema validation before any processing begins. Here's an example:

```
# Define schema for sales DataFrame
sales_schema = pa.DataFrameSchema(
    {
        "price": pa.Column(float, checks=[pa.Check.ge(0)]),
        "quantity": pa.Column(int, checks=[pa.Check.ge(0)]),
    }
)
```

```
@check_input(sales_schema)
def analyze_sales_data(sales_df: pd.DataFrame) -> dict:
    revenue = sales_df["price"] * sales_df["quantity"]

    return {
        "total_revenue": revenue.sum(),
        "max_sale": sales_df["quantity"].max(),
    }
```

Run the `analyze_sales_data` function with a DataFrame containing mixed data types:

```
data = pd.DataFrame(
    {
        "price": [50, 100, "invalid", 75],
        "quantity": [5, 3, 2, "error"],
    }
)
try:
    results = analyze_sales_data(data)
    print(results)
except pa.errors.SchemaError as e:
    print("SchemaError:", e)
```

The validation decorator prevents processing and reports the type mismatch immediately.

```
SchemaError: error in check_input decorator of function
'analyze_sales_data':
expected series 'price' to have type float64, got object
```

10.4.2 Validate Only Critical Columns

Avoid validating every column in large datasets. Focus validation on essential columns that directly impact your analysis to reduce processing overhead.

In the following code, we only create a schema for the `amount` and `store` columns, which are used in the calculation:

```
# Only validate columns used in the calculation
schema = pa.DataFrameSchema(
    {
        "amount": pa.Column(float, checks=pa.Check.gt(0)),
        "store": pa.Column(
            str, checks=pa.Check.isin(["A", "B"])
        ),
    }
)



@pa.check_input(schema)
def get_amount_by_store(df):
    return df.groupby("store")["amount"].sum()
```

Even if the DataFrame contains additional columns, only the `amount` and `store` columns are validated:

```
df = pd.DataFrame(
    {
        "customer_id": [1, 2, 3],
        "amount": [100.0, 200.0, 300.0],
        "date": ["2023-01-01", "2023-01-02", "2023-01-03"],
        "store": ["A", "B", "A"],
    }
)
amount_by_store = get_amount_by_store(df)
```

10.5 Key Takeaways

1. Benefits of data validation:
    - Prevents misleading analyses and incorrect conclusions by ensuring data quality
    - Prevents cascading negative effects on downstream systems and models
    - Saves computational resources by catching issues early before processing
    - Ensures data freshness and relevance for accurate insights
    - Helps detect and handle outliers and anomalies

    - Identifies missing values that could impact analysis
2. Features of Pandera:
    - Schema definition using DataFrameSchema or DataFrameModel
    - Built-in checks for common validation scenarios
    - Support for custom validation functions
    - Decorator-based validation for function inputs and outputs
    - YAML export/import for sharing validation rules
3. Best practices:
    - Validate data at the point of entry to catch issues early
    - Focus validation on critical columns that impact your analysis

11 Data Version Control

11.1 What Is Data Version Control?

Data Version Control is the practice of tracking and managing changes to data files and datasets over time, in a similar way to how traditional version control systems handle code. Data version control helps maintain data lineage, reproducibility, and collaboration in data science projects.

11.2 Why Is Data Version Control Essential?

11.2.1 Replicate Experiments

Comprehensive versioning helps maintain consistency and reproducibility, even as your datasets and codebase evolve over time.

Imagine you've developed a machine learning model that achieves excellent performance on a specific dataset. After several months, you receive an updated version of the dataset with new records and corrections. When you retrain your model on this new dataset, you discover the model performs poorly compared to the original version.

By tracking both your data and models, you can easily roll back to previous versions of your data and code, as shown in Figure 11.1.

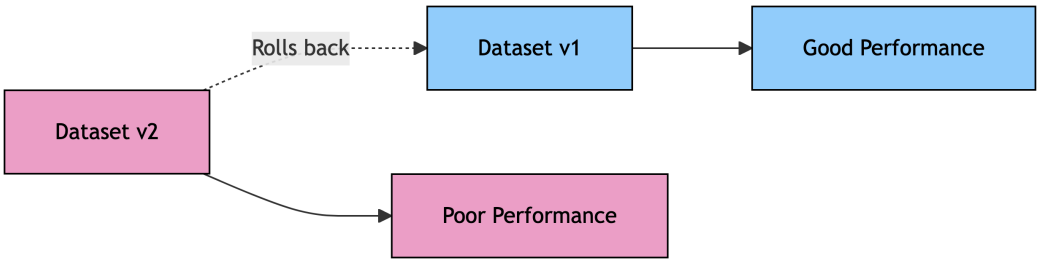


Figure 11.1: Dataset versioning with rollback capability

11.2.2 Version Control Large Datasets

Data version control keeps your Git repository lean while tracking dataset and model changes effectively.

Storing large datasets directly in Git repositories presents several challenges:

- Git repositories become bloated when storing large binary files.
- Common Git operations (cloning, fetching, pushing) slow down significantly.
- Tracking changes and understanding dataset evolution over time is difficult.

Data version control addresses these issues by storing datasets remotely (Amazon S3, Google Cloud Storage, or local servers) and maintaining only lightweight references in Git. This separation ensures efficient version control for both code and data.

Figure 11.2 illustrates this workflow.

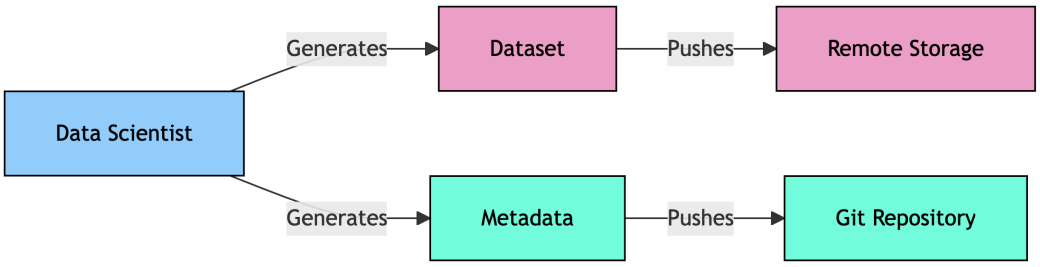


Figure 11.2: Dataset versioning with remote storage

11.2.3 Share and Sync Datasets

Data version control enables team members to share, sync, and collaborate on datasets automatically. This ensures everyone works with identical, consistent data across the entire project.

Teams face two critical problems when manually managing datasets:

- **Inconsistent Versions**: Team members work with different dataset versions, producing conflicting analyses and unreliable results.
- **Inefficient Workflow**: Large dataset transfers create bottlenecks that slow progress and fragment team coordination.

These issues reduce productivity and prevent effective collaboration, as shown in Figure 11.3.

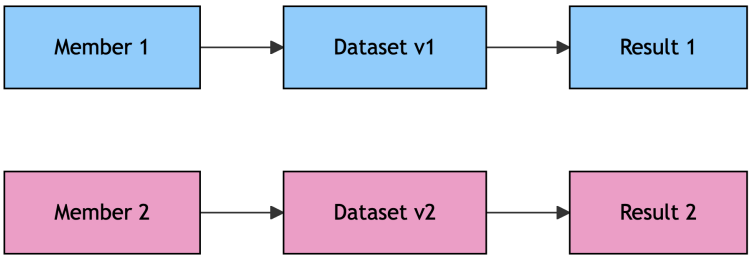


Figure 11.3: The problem of manually sharing and syncing datasets

Data version control transforms team collaboration by automatically synchronizing datasets across all members. Teams can share large datasets efficiently without manual file transfers or version mismatches, as demonstrated in Figure 11.4.

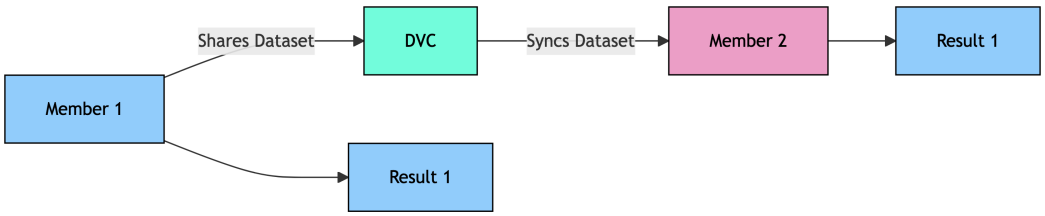


Figure 11.4: Dataset collaboration with data version control

11.3 Use DVC for Data Version Control

DVC (Data Version Control) is an open-source version control system designed specifically for machine learning projects. DVC helps track large files, datasets, and machine learning models while working alongside Git for code version control.

Main components:

- **Tracking System**: Manages data versions using metadata and hash values
- **Storage Backend**: Supports various remote storage options (S3, GCP, etc.)
- **Pipeline Management**: Defines and tracks data processing workflows
- **Git Integration**: Works with Git to version control both code and data

To install DVC, use pip:

```
pip install dvc
```

11.3.1 Get Started

Initialize DVC inside an existing Git repository:

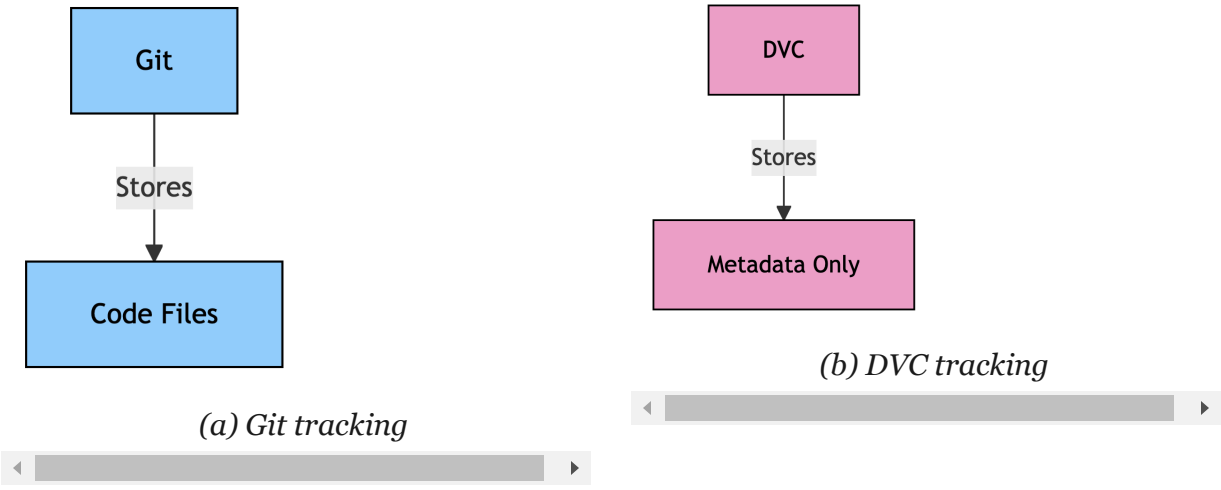
```
dvc init
```

After running `dvc init`, DVC sets up the project with the necessary configuration to start tracking data. Your directory structure will look like this:

```
.
├── .dvc/                  # DVC config and internal files
├── .dvcignore             # Like .gitignore but for DVC operations
├── .git/                  # Git repository
└── (your project files)
```

11.3.2 Track Data

Git stores code files directly in its repository (Figure 11.5 (a)). DVC stores only metadata, keeping actual data files separate (Figure 11.5 (b)).



(a) Git tracking

(b) DVC tracking

Figure 11.5: Git vs. DVC tracking approaches

For example, if you have a `data/` directory with your raw files, you can use `dvc add` to start tracking it:

```
dvc add data/
```

This creates a `data.dvc` metadata file. DVC stores three key pieces of information in this file:

- `outs`: Lists the tracked outputs
- `md5`: A unique hash that identifies the data version
- `path`: Location of the tracked data

Example 11.1: data.dvc

```
outs:
- md5: 86451bd526f5f95760f0b7a412508746.dir
  path: data
```

Since this file is lightweight, it can be easily committed to Git:

```
git add data.dvc .gitignore
git commit -m "Track dataset with DVC"
```

When you run `dvc add data/`, DVC also adds `data/` to the `.gitignore` file to prevent Git from tracking the actual `data/` directory.

Example 11.2: .gitignore

```
data/
```

The directory structure after running `dvc add data/` looks like this:

```
.
├── data/        # Contains your actual dataset
├── data.dvc     # Metadata file tracked by Git
└── .gitignore   # Contains an entry to ignore /data/
```

11.3.3 Store Data Remotely

While the metadata file is lightweight, the actual data files can be large. To store the actual data files in a remote location, you can use DVC's remote storage feature.

DVC supports many storage backends like S3, GCS, Azure, SSH, and Google Drive. In this section, we'll use Amazon S3 as an example.

11.3.3.1 Prerequisites

Configure AWS credentials using the AWS CLI:

```
aws configure
```

Create an S3 bucket (or use an existing one). Let's use `my-dvc-bucket` as an example.

11.3.3.2 Configuration Steps

To configure the remote storage, run:

```
dvc remote add -d myremote s3://my-dvc-bucket/path/to/data
```

This command:

- Creates a new remote named `myremote`
- Sets it as the default (`-d` flag) by default
- Points to your S3 bucket path `s3://my-dvc-bucket/path/to/data`

The command creates/updates `.dvc/config`:

Example 11.3: .dvc/config

```
[core]
    remote = myremote

['remote "myremote"']
    url = s3://my-dvc-bucket/path/to/data
```

Commit the configuration to Git:

```
git add .dvc/config
git commit -m "Configure S3 remote for DVC"
```

Upload your tracked data to S3:

```
dvc push
```

11.3.4 Retrieve Data

Suppose you just joined a project that uses DVC to manage datasets and model files. After cloning the Git repository, you might only see `.dvc` files and pipeline definitions, but not the actual data content.

For example:

```
.
└── data/
    └── raw.dvc
```

The `.dvc` file contains metadata pointing to the data stored in a remote location. To download and restore the full dataset locally, simply run:

```
dvc pull
```

This command downloads the required files from the configured remote storage and rebuilds the full directory structure:

```
.
└── data/
    ├── final/
    |   └── segmented.csv
    ├── intermediate/
    |   └── scale_features.csv
    ├── raw/
    |   └── marketing_campaign.csv
    └── raw.dvc
```

11.3.5 Switch Between Versions

Without a reliable workflow, it's easy to accidentally pair the wrong version of code with the wrong version of data, leading to results you can't reproduce or trust.

The `dvc checkout` command makes it easy to switch between data and model versions tied to specific Git commits or branches.

To demonstrate, let's track and switch between two dataset versions.

Suppose you have a `data.csv` file with the following content:

feature	target
1	0
2	1
3	0

From the terminal, track the dataset with DVC:

```
dvc add data.csv
git add data.csv.dvc .gitignore
git commit -m "Version 1 of data"
```

Next, make changes to the `data.csv` file:

feature	target
10	1
20	0
30	1

Track the updated dataset:

```
dvc add data.csv
git add data.csv.dvc
git commit -m "Version 2 of data"
```

Now switch back to version 1:

```
git checkout HEAD~1
dvc checkout
```

This command restores `data.csv` to version 1, keeping data synchronized with the corresponding code version.

Feature	Target
1	0
2	1
3	0

11.3.6 Build a DVC Pipeline

Beyond tracking data, DVC allows you to create reproducible machine learning pipelines that connect stages like preprocessing and training.

The `dvc.yaml` file in Example 11.4 defines a two-stage pipeline (`process_data` and `train`) that DVC executes and tracks. Each stage contains:

- `stages`: Top-level section containing all pipeline stages
- `process_data`, `train`: Individual stage names mapping to pipeline steps
- `cmd`: Command DVC executes for each stage
- `deps`: Stage dependencies including data files, Python scripts, or configuration files
- `outs`: Stage outputs that DVC versions and manages automatically

Example 11.4: dvc.yaml

```
stages:
  process_data:
```

```
    cmd: python src/process_data.py
    deps:
    - data/raw
    - src/process_data.py
    - config
    outs:
    - data/intermediate
  train:
    cmd: python src/segment.py
    deps:
    - data/intermediate
    - src/segment.py
    - config
    outs:
    - data/final
    - model/cluster.pkl
```

To execute the pipeline, use `dvc repro`. DVC runs only modified stages. This approach eliminates unnecessary recomputation and ensures reproducible results across pipeline runs.

For example, modifying `src/segment.py` executes only the affected `train` stage.

Example 11.5: src/segment.py

```
def get_pca_model(data: pd.DataFrame) -> PCA:
    pca = PCA(n_components=4)  # changed from 3 to 4
    pca.fit(data)
    return pca
```

```
dvc repro
```

```
'data/raw.dvc' didn't change, skipping
Stage 'process_data' didn't change, skipping
Running stage 'train':
> python src/segment.py
```

11.4 Key Takeaways

1. Benefits of Data Version Control

    - Enables experiment replication by tracking exact data versions and pipeline steps
    - Provides version control for large datasets without bloating Git repositories
    - Facilitates team collaboration through shared data versioning and synchronization
2. DVC core features
    - Tracks large datasets without bloating Git repositories
    - Supports multiple remote storage backends (S3, GCS, Azure, etc.)
    - Creates reproducible machine learning pipelines
    - Integrates seamlessly with Git for version control
3. Best practices
    - Always initialize DVC in a Git repository
    - Use remote storage for large datasets
    - Commit `.dvc` files to Git, not the actual data

12 Continuous Integration

12.1 What Is Continuous Integration?

Continuous Integration (CI) is a software development practice where team members frequently integrate their code changes into a shared repository. Each integration is automatically verified by running tests and other quality checks. By frequently integrating code changes to a common repository, a team can ensure that new changes don't break existing functionality.

12.2 Why Is Continuous Integration Important?

CI offers several key benefits for data science projects:

12.2.1 Early Bug Detection

CI automatically catches bugs and issues early in the development process, preventing them from reaching production and saving significant time and resources that would otherwise be spent investigating and fixing problems later.

Consider this common scenario in machine learning development:

- You modify the feature engineering function and push untested changes to GitHub.
- Later, you discover bugs, but your team members have already pulled your changes.
- The bugs affect the whole team's work, degrading the quality of the ML model.

This scenario is illustrated in Figure 12.1.

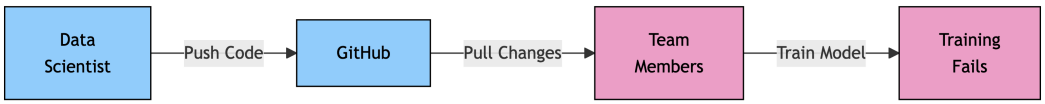


Figure 12.1: Consequences of Pushing Untested Code

By implementing CI, tests run automatically whenever code changes are made, preventing these project setbacks that affect everyone. Figure 12.2 illustrates how CI transforms this workflow.

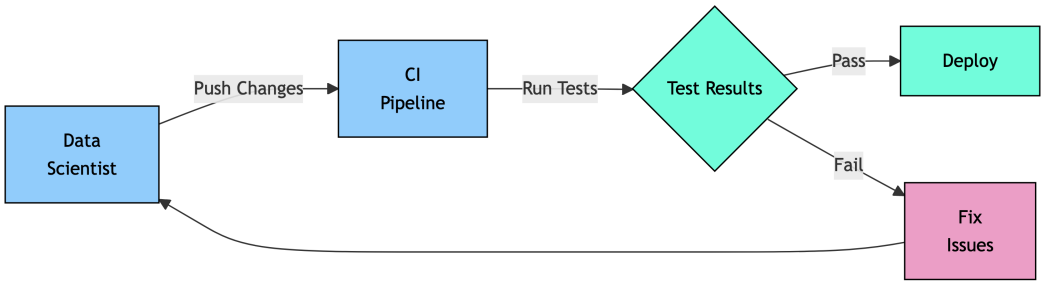


Figure 12.2: CI Pipeline

12.2.2 Improved Code Quality

CI pipelines maintain high code quality by automatically checking contributions against established standards.

For example, when a team member submits a PR with a machine learning model, the CI system verifies that the code follows PEP 8 standards and includes proper docstrings for all functions. Figure 12.3 illustrates this quality gate.

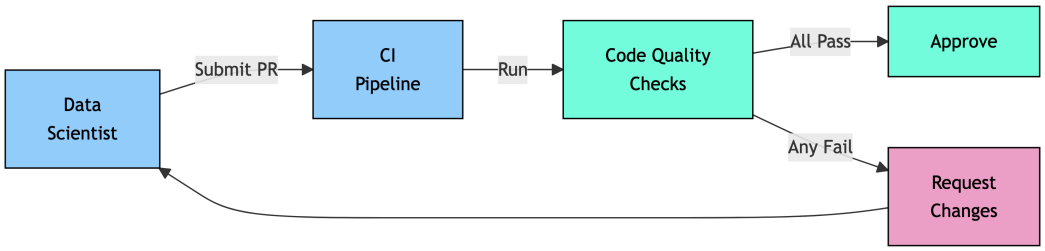


Figure 12.3: Code Quality Checks in CI

12.2.3 Documentation Maintenance

CI eliminates outdated documentation by automatically updating it whenever code changes.

For example, when a data scientist updates a machine learning model's hyperparameters, CI automatically regenerates the model's documentation with the new parameter values, performance metrics, and example usage. This ensures that team members always have access to accurate, up-to-date information about the model. Figure 12.4 illustrates this process.

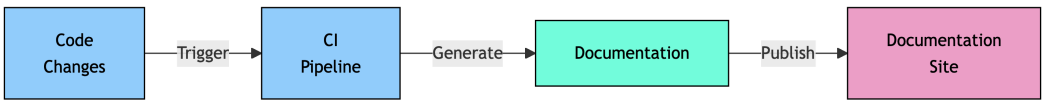


Figure 12.4: Documentation Maintenance with CI

12.3 Use GitHub Actions for Continuous Integration

12.3.1 What Is GitHub Actions?

GitHub Actions is a powerful tool that allows you to automate your workflows and tasks. GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that enables you to build, test, and deploy your code with ease.

GitHub Actions is built on top of the GitHub platform, and is tightly integrated with the GitHub repository. The tool allows you to create workflows that are triggered by events in your repository, such as when a pull request is created, a push is made to a branch, or a new issue is opened.

There are four main components in GitHub Actions:

- Workflows
- Events
- Jobs
- Steps

The relationship between these components is illustrated in Figure 12.5.

Figure 12.5: Components of GitHub Actions

Let's go through each of these components in detail.

12.3.2 Workflows

In GitHub Actions, *workflows* are sets of automated rules and actions that are triggered by specific *events* in a GitHub repository.

Workflows are defined in the `.github/workflows` directory in a repository, and a repository can have multiple workflows, each of which can perform a different set of tasks. For example, you can have one workflow to run tests and another workflow to create documentation.

The workflow files are stored in a `.github/workflows` directory at the root of your repository. The directory structure looks like this:

```
.
└──.github/
    └── workflows/
        ├── create_documentation.yaml
        └── run_tests.yaml
```

12.3.3 Event

Events in GitHub Actions are specific activities that occur in a repository and can trigger the execution of a workflow.
The `on` keyword is used to define event triggers in a workflow file.

Here are some examples of events:

1. Triggering on any pull request activity:

```yaml
on:
  pull_request
```

2. Triggering when a pull request is opened or updated in the `main` branch.

```yaml
on:
  pull_request:
```

```
    branches:
      - main
```

3. Triggering on pull request with path filters:

```
on:
  pull_request:
    branches:
      - main
    paths:
      - src/**
      - tests/**
      - config/**
```

By combining different event types and using filters like `branches` and `paths`, you can precisely control when your workflows should be triggered based on specific activities or changes in your repository.

12.3.4 Jobs

Jobs are a set of tasks that are executed in response to an event trigger in a GitHub Actions workflow. A job:

- Contains a sequence of steps that are executed in order
- Can run independently and in parallel with other jobs by default
- Runs in its own virtual environment with specified dependencies
- Can depend on other jobs using the `needs` keyword

Let's explore examples of jobs running in parallel and jobs with dependencies.

12.3.4.1 Jobs running in parallel

Running jobs in parallel reduces overall workflow execution time by allowing independent tasks to execute simultaneously.

The following example defines two jobs: `evaluate_model` and `get_predictions`. Each job is configured to run on the latest Ubuntu environment. Since these jobs have no dependencies specified between them, they will execute in parallel.

```
jobs:
  evaluate_model:
    name: Evaluate model
    runs-on: ubuntu-latest
  get_predictions:
    name: Get predictions
    runs-on: ubuntu-latest
```

12.3.4.2 Jobs with dependencies

Job dependencies ensure that tasks execute in the correct order, preventing failures that occur when later steps run before their required inputs are ready.

In the following example, we define four jobs that represent a typical machine learning workflow:

- `data_preprocessing` handles data cleaning and feature engineering
- `model_training` trains the machine learning model using the preprocessed data
- `evaluate_model` calculates model performance metrics
- `get_predictions` generates predictions on new data

Each job has dependencies that enforce the correct execution order:

- `model_training` requires `data_preprocessing` to complete first, since it needs cleaned data.
- Both `evaluate_model` and `get_predictions` wait for `model_training` to finish, since they need the trained model.

```
jobs:
  data_preprocessing:
    name: Process data
```

```
    runs-on: ubuntu-latest
  model_training:
    name: Train model
    runs-on: ubuntu-latest
    needs: data_preprocessing
  evaluate_model:
    name: Evaluate model
    runs-on: ubuntu-latest
    needs: model_training
  get_predictions:
    name: Get predictions
    runs-on: ubuntu-latest
    needs: model_training
```

The dependencies are illustrated in Figure 12.6.

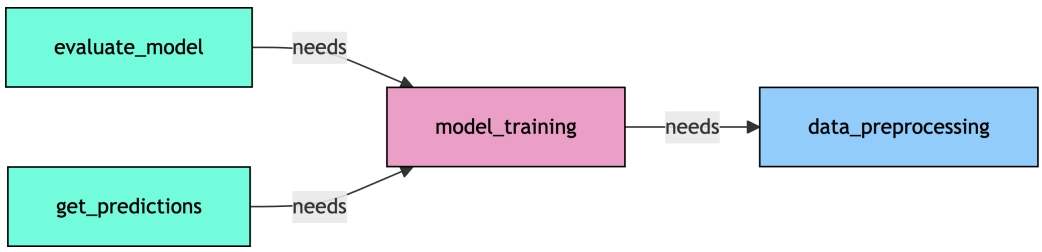


Figure 12.6: Job Dependencies in Machine Learning Workflow

12.3.5 Steps

Steps in a job are executed sequentially, and the output of each step is used by the subsequent step.

Steps are defined under the `steps` key within each job in the workflow file. Here is an example of a job called "Run unit tests" that contains multiple steps:

```
jobs:
  run_tests:
    name: Run unit tests
    runs-on: ubuntu-latest
```

```
    steps:
      - name: Checkout
        run: git clone https://github.com/example/my-project

      - name: Environment setup
        run: |
          sudo apt-get update
          sudo apt-get install python3.8 -y

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest

      - name: Run tests
        run: pytest tests
```

Line 6
Checkout: Clones the Git repository
Line 9
Environment setup: Installs Python 3.8 on the runner
Line 14
Install dependencies: Installs project requirements and pytest
Line 19
Run tests: Executes tests with pytest

12.3.6 Actions

Many steps, like checkout and environment setup, are commonly needed across different projects. Rather than rewriting these steps for each project, GitHub Actions provides reusable components called *actions*.

Actions are pre-packaged, reusable steps that can be shared across workflows. Actions encapsulate common functionality into standardized units that can be imported from GitHub's Marketplace, making workflows simpler and more maintainable.

Here is the rewritten YAML file of the `run_tests` job that uses pre-defined actions (`actions/checkout@v2` and `actions/setup-python@v2`).

```yaml
jobs:
  run_tests:
    name: Run unit tests
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Environment setup
        uses: actions/setup-python@v2
        with:
          python-version: 3.8

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest

      - name: Run tests
        run: pytest tests
```

Line 7
`actions/checkout@v2`: Clones the Git repository
Line 10
`actions/setup-python@v2`: Sets up the Python environment

12.4 Common Data Science Workflows

Now that we've learned the basics of GitHub Actions, let's explore some common data science workflows that can be automated with GitHub Actions.

12.4.1 Create Documentation

Automated documentation generation keeps your project documentation synchronized with code changes, eliminating the manual effort of updating docs.

Here's how to set up GitHub Actions to create documentation when a pull request modifies files in the `src` directory:

```
name: Create documentation
on:
  pull_request:
    paths:
      - src/**

jobs:
  create_documentation:
    name: Create documentation
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        id: checkout
        uses: actions/checkout@v2

      - name: Environment setup
        uses: actions/setup-python@v2
        with:
          python-version: 3.8

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pdoc3

      - name: Create documentation
        run: pdoc --html src -o docs --force

      - name: Create artifact
        uses: actions/upload-artifact@v2
        with:
          name: documentation
          path: docs
```

Line 21

Installs the pdoc3 package, an API documentation generator for Python.
Line 26
Executes the `pdoc` command to generate HTML documentation for the `src` directory. The documentation is outputted to the `docs` directory.
Line 29
Uploads the `docs` directory as an artifact named “documentation”, making it available for further use or distribution.

Whenever changes are made to files within the `src` directory in a pull request, this workflow will be triggered. The workflow generates HTML documentation for the code in `src` and creates an artifact containing the documentation.

Figure 12.7 illustrates the workflow.

Figure 12.7: Create Documentation Workflow

12.4.2 Run Data Pipeline on Data Changes

GitHub Actions can automatically trigger your data pipeline whenever data changes, keeping your processed outputs synchronized with source data.

Here’s how to set up GitHub Actions to execute the data pipeline when changes are made to files in the `data` directory:

```
name: Data Pipeline Workflow

on:
  push:
```

```
    paths:
      - data/**

jobs:
  run_pipeline:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        id: checkout
        uses: actions/checkout@v2

      - name: Environment setup
        uses: actions/setup-python@v2
        with:
          python-version: 3.8

      - name: Install DVC
        run: pip install dvc

      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY
}}
          aws-region: us-east-1

      - name: Configure DVC Remote
        run: dvc remote modify my_remote s3://my-bucket/data

      - name: Pull Latest Data
        run: dvc pull

      - name: Execute Data Pipeline
        run: dvc repro
```

Line 4
Triggers the workflow when changes are made to files within the `data` directory.
Line 25

Configures AWS credentials. The credentials are retrieved from the repository's secrets.
Line 32
Configures the DVC remote storage to use an S3 bucket named "my-bucket" and a directory named "data".
Line 35
Pulls the latest data from the configured DVC remote storage using the `dvc pull` command.

With this workflow, your data pipeline will be automatically executed whenever changes are made to files in the `data` directory, as illustrated in Figure 12.8.

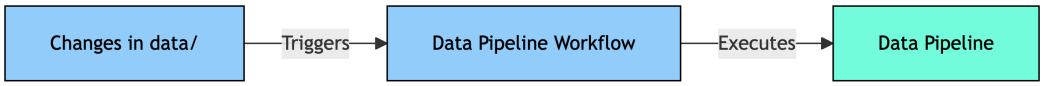


Figure 12.8: Data Change Workflow

12.4.3 Generate Report

Automated report generation ensures stakeholders always receive the latest analysis results without manual intervention.

Use the following YAML file to set up this workflow:

```yaml
name: Report Generation Workflow

on:
  push:
    paths:
      - analysis/*.py

jobs:
  generate_report:
    runs-on: ubuntu-latest
```

```
    steps:
      - name: Checkout
        id: checkout
        uses: actions/checkout@v2

      - name: Environment setup
        uses: actions/setup-python@v2
        with:
          python-version: 3.8

      - name: Install Dependencies
        run: pip install -r requirements.txt

      - name: Generate Report
        run: python analysis/generate_report.py

      - name: Upload Report
        uses: actions/upload-artifact@v2
        with:
          name: generated-report
          path: analysis/report.pdf
```

Line 6
Triggers the workflow when changes are made to Python analysis scripts within the `analysis` directory.
Line 25
Executes the `generate_report.py` script, generating the desired report or visualization.
Line 28
Uploads the generated report as an artifact named “generated-report”.

The generated report can then be accessed as an artifact, ready to be shared with others, as illustrated in Figure 12.9.

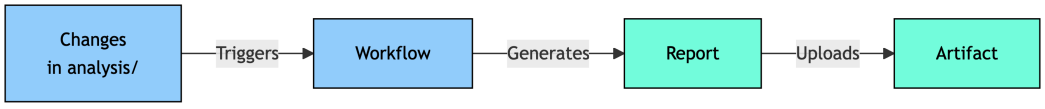


Figure 12.9: Generate Report Workflow

12.5 Key Takeaways

1. Benefits of Continuous Integration:

- CI automates testing and documentation tasks triggered by PR submissions
- Helps catch bugs early before they affect the entire team
- Enables faster development cycles and better collaboration
- Increases confidence in code quality through automated validation

2. GitHub Actions components:

- **Workflows**: Automated rules triggered by repository events
- **Events**: Activities that trigger workflows (e.g., PR creation, push)
- **Jobs**: Sets of tasks that execute in response to events
- **Steps**: Sequential tasks within a job
- **Actions**: Reusable components for common tasks

13 Package Your Project

13.1 What Is Packaging?

Packaging is the process of organizing your code into a structured format that makes your project easy to distribute and install.

13.2 Why Is Packaging Essential?

13.2.1 Easy Distribution and Sharing

Packaging your project enables easy distribution and sharing, allowing others to install your utilities with a single command while automatically handling dependencies and version management.

Consider this scenario: You've developed useful data processing utilities that your colleagues want to use as part of a different project. While they could get the code directly from your repository, this approach has several drawbacks:

- They would need to manually integrate your code into their projects, which can be labor-intensive and error-prone.
- They would need to identify and install all the required dependencies themselves.
- When you update your utilities with bug fixes or improvements, they would need to manually sync their copy.

- Different colleagues might end up with different versions of your code, leading to inconsistencies.

Packaging solves these issues by enabling upload to PyPI, where others can install with `pip install your-package` and automatically receive dependencies and the correct version, as shown in Figure 13.1.

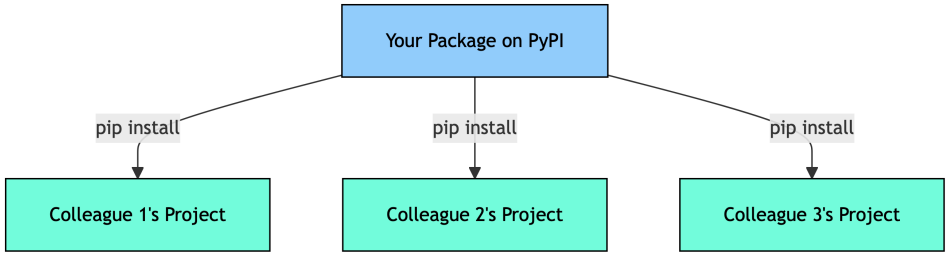


Figure 13.1: Packaging Your Project

13.2.2 Version Control and Dependency Management

Packaging provides version control capabilities that solve compatibility conflicts by allowing multiple versions to coexist across different projects.

Different colleagues may have varying requirements or constraints that make certain versions of your library more suitable for their specific needs. For example:

- One colleague might need an older version that maintains compatibility with their legacy systems
- Another might want the latest version with new features

By packaging your library, you can define version numbers, specify dependencies, and include other crucial metadata. This enables

different users to simultaneously use different versions of your package without conflicts, as shown in Figure 13.2.

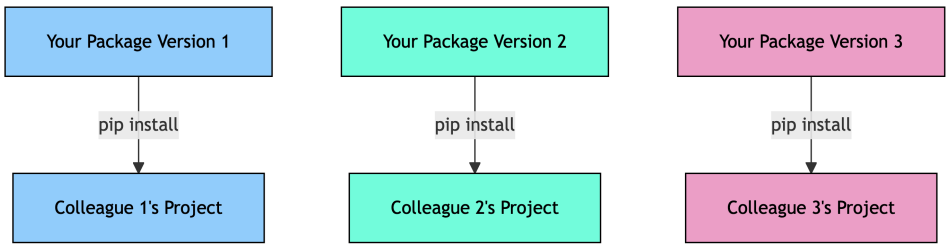


Figure 13.2: Packaging Your Project Version Control

13.3 Use uv for Packaging

Several tools are available for packaging Python projects, including setuptools, Poetry, and uv. In this chapter, we'll focus on using uv since it provides a modern, streamlined approach to both dependency management and packaging.

As discussed in Dependency Management, uv is a fast and reliable tool that handles dependency management. Beyond managing dependencies, uv also includes robust packaging capabilities, making it an excellent all-in-one solution. For projects already using uv for dependency management, the packaging process integrates seamlessly with your existing workflow.

13.3.1 Prepare the Project Structure

To prepare your project for packaging, organize your code so the source directory matches your package name, enabling direct imports and clear module access. For a project called `pandas_processors`, the source directory must also be named `pandas_processors`:

```
pandas_processors
├── pandas_processors/
|   ├── __init__.py
|   ├── create.py
|   ├── impute.py
|   └── normalize.py
├── tests/
├── pyproject.toml
├── README.md
└── ...
```

This directory structure allows users to import your package directly using:

```
import pandas_processors
```

If the `impute` module includes a class named `MeanMedianImputer`, users can access it with:

```
from pandas_processors.create import MeanMedianImputer
```

13.3.2 Add Metadata

Configure your project's packaging metadata and build system in `pyproject.toml` to define how your package will be built and distributed.

Start by adding metadata such as the package name, version, author, description, and README file.

Example 13.1: pyproject.toml

```
[tool.uv]
name = "pandas-processors"
version = "0.1.0"
description = "Utilities for pandas DataFrame processing."
authors = ["khuyentran1401 <khuyentran1476@gmail.com>"]
readme = "README.md"
```

Next, add the `[build-system]` section to the `pyproject.toml` file, which specifies the build backend to use. We will use hatchling, a modern, fast alternative to setuptools, as our build backend.

Example 13.2: pyproject.toml

```toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

13.3.3 Build the Package

Building your package creates distribution files that can be uploaded to PyPI and installed by users on different systems and platforms.

Execute the following command to start the build process:

```
uv build
```

```
Building source distribution...
Building wheel from source distribution...
Successfully built dist/pandas_processors-0.1.0.tar.gz
Successfully built dist/pandas_processors-0.1.0-py3-none-any.whl
```

Running this command will generate the `dist/` directory with the following files:

```
dist/
├──.gitignore
├── pandas_processors-0.1.0.tar.gz
└── pandas_processors-0.1.0-py3-none-any.whl
```

These files represent two different ways to distribute your Python package:

1. **Source Distribution** (`.tar.gz`):
    - A compressed archive containing your package's source code.
    - Platform-independent and can be installed on any system.

    - Requires the user to have a Python environment with build tools.
2. **Wheel Distribution** (`.whl`):
    - A pre-built, binary package format that installs faster than source distributions
    - Cross-platform compatible with Python 3 (filename shows `py3-none-any`)
    - The preferred format for most installations

When users install your package using `pip install pandas-processors`, pip will automatically choose the appropriate distribution format based on their system. On most systems, pip will prefer the wheel file for faster installation.

13.3.4 Publish the Package

The final step in packaging is uploading your built distribution files to PyPI, where they become available for installation worldwide.

PyPI (Python Package Index) is the official repository for Python packages. Publishing to PyPI makes your package:

1. **Publicly Available**: Anyone can install it using pip/uv
2. **Discoverable**: Listed in PyPI's searchable index
3. **Version Controlled**: Supports multiple versions
4. **Dependency Managed**: PyPI handles dependencies automatically

To publish the package to PyPI, run the following command:

```
uv publish
```

```
Publishing pandas-processors (0.1.0) to PyPI
 - Uploading pandas_processors-0.1.0-py3-none-any.whl 100%
 - Uploading pandas_processors-0.1.0.tar.gz 100%
```

After publishing, your package will be accessible on PyPI. You can install the package using:

```
pip install pandas-processors
```

13.4 Manage Package Versions

When maintaining a Python package, proper version management is crucial for both users and developers. Here's how to effectively manage your package versions:

13.4.1 Semantic Versioning

Follow the Semantic Versioning (SemVer) specification, which uses a three-part version number: `MAJOR.MINOR.PATCH`:

- **MAJOR** version: Increment when making incompatible API changes
- **MINOR** version: Increment when adding functionality in a backward-compatible manner
- **PATCH** version: Increment when making backward-compatible bug fixes

For example:

- `1.0.0`: Initial release
- `1.0.1`: Bug fix
- `1.1.0`: New feature
- `2.0.0`: Breaking changes

13.4.2 Updating Package Version

To update the package version, follow the following steps:

1. Update the version in the `pyproject.toml` file:

```
# pyproject.toml
[tool.uv]
name = "pandas-processors"
version = "1.0.1"  # Increment as needed
```

2. Create a Git tag to mark the release version:

```
git tag -a v1.0.1 -m "Version 1.0.1"
git push origin v1.0.1
```

3. Build and publish the package:

```
uv build
uv publish
```

13.4.3 Version Constraints

Users can install specific versions using:

```
# Install exact version
pip install pandas-processors==1.0.1

# Install latest minor version
pip install pandas-processors~=1.0.0

# Install latest major version
pip install pandas-processors>=1.0.0
```

13.5 Add a Documentation Page

To enhance developer adoption of your package, create API documentation and host it for easy access. This process involves two main steps:

1. Create an API documentation page
2. Enable GitHub Pages to host the documentation

13.5.1 Create an API Documentation Page

API documentation generation transforms your code comments and docstrings into professional, browsable documentation that helps developers understand your package's functionality.

We will use pdoc3 for this purpose, a Python tool that automatically extracts and formats documentation from your code.

Start with installing pdoc3 as a development dependency using uv, since it's not needed for end users.

```
uv add pdoc3 --dev
```

From your project's root directory, run the following command to generate the HTML documentation:

```
pdoc --html pandas_processors -o docs
```

This command creates a `docs` directory with the following structure:

```
pandas_processors/
└── docs/
    └── pandas_processors/
        ├── create.html
        ├── impute.html
        ├── index.html
        └── normalize.html
```

Add and commit the `docs` directory to your GitHub repository:

```
git add docs
git commit -m "Add API documentation generated by pdoc3"
git push origin main
```

13.5.2 Enable GitHub Pages

To make your API documentation accessible to developers worldwide, you have several hosting options. GitHub Pages is one

popular choice that provides free web hosting and integrates seamlessly with your repository.

GitHub Pages can be configured in two ways:

- **Branch deployment**: Publish directly from a repository branch and folder
- **GitHub Actions workflow**: Use automated deployment for more control

Both approaches make your documentation accessible via a public URL that you can link from your PyPI package page.

For detailed setup instructions, refer to GitHub's documentation on setting up GitHub Pages.

13.5.3 Build and Publish the Updated Package

Complete the documentation setup by updating your package configuration with the documentation link and publishing a new version.

Example 13.3: pyproject.toml

```
[tool.uv]
name = "pandas-processors"
version = "0.1.2"
description = "Utilities for pandas DataFrame processing."
authors = ["khuyentran1401 <khuyentran1476@gmail.com>"]
repository = "https://github.com/khuyentran1401/pandas-
processors.git"
readme = "README.md"
documentation = "https://khuyentran1401.github.io/pandas-
processors/"
```

Build the updated package:

```
uv build
```

Publish the updated package:

```
uv publish
```

After publishing, the documentation will appear under the “Project links” section of your PyPI repository.

13.6 Key Takeaways

1. Project organization and structure:
    - Use a nested directory structure (`package_name/package_name/`) for clean imports
    - Include essential files like `pyproject.toml`, `README.md`, and `.gitignore`
2. Build system and distribution:
    - Use modern tools like uv and hatchling for building packages
    - Configure build settings in `pyproject.toml`
    - Publish to PyPI for public distribution
3. Version management:
    - Follow semantic versioning (MAJOR.MINOR.PATCH)
    - Use Git tags to mark releases
4. Documentation and visibility:
    - Generate API documentation using pdoc3
    - Host documentation on GitHub Pages

14 Notebooks in Production

14.1 Notebook Production Challenges

While Jupyter Notebooks excel at exploratory data analysis and prototyping with their interactive code execution and visualization capabilities, they face significant challenges when used in production environments.

Production workflows, like data pipelines, model training, and deployment, require robust, maintainable, and reproducible code. Let's explore why Jupyter Notebooks can be problematic in production and see what alternatives exist.

14.1.1 Hidden State Issues

Jupyter notebooks maintain a hidden execution state that can diverge from what's visible in the code cells, creating unpredictable behavior and making code difficult to reproduce reliably.

Consider this scenario: you assign `x = 5` in a cell, then later delete that cell. The variable `x` disappears from your visible code but remains in memory, creating invisible dependencies that can break your notebook's reproducibility.

Example 14.1: notebook.ipynb

```
x = 5
```

This mismatch between visible code and execution state, illustrated in Figure 14.1 and Figure 14.2, makes it impossible to reliably reproduce results by running cells from top to bottom.

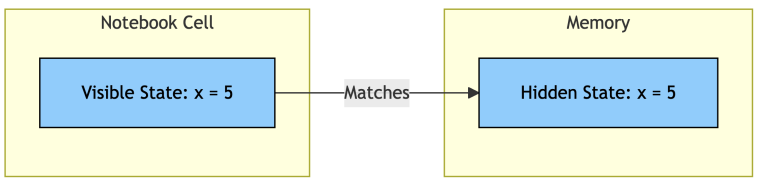


Figure 14.1: Matching Hidden State

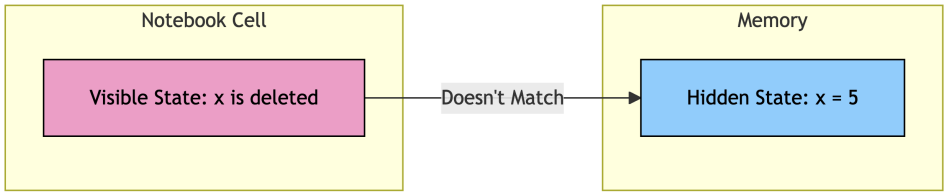


Figure 14.2: Mismatching Hidden State

14.1.2 Variable Redefinition Issues

Variable redefinition occurs when a variable is assigned a new value, overwriting its previous value. While redefinition is possible in any Python script, it's particularly common in Jupyter notebooks because cells can be executed in any order and multiple times.

Consider this data analysis notebook where the `date` column undergoes three transformations:

Example 14.2: Cell 1

```
# Initial data loading
df = pd.read_csv('data.csv')
df['date'] = pd.to_datetime(df['date'])  # datetime object
```

Example 14.3: Cell 2

```
# Convert to date only
df['date'] = df['date'].dt.date  # date object
```

Example 14.4: Cell 3

```
# Convert to string for categorical encoding
df['date'] = df['date'].astype(str)  # string
```

This pattern creates three critical problems:

- **State tracking confusion**: You can't easily tell what format `date` is in
- **Debugging complexity**: Tracking down errors becomes difficult because you need to determine which transformation step broke your code
- **Broken dependencies**: Other cells will fail unexpectedly if they assume the wrong date format based on execution order

14.1.3 Cell Execution Dependency Issues

Jupyter notebooks allow arbitrary cell execution order, but when cells depend on previous steps, this flexibility becomes a liability. Running cells out of order can produce incorrect results or errors, making notebooks difficult to maintain and unreliable when revisited later.

Consider this example of a data preprocessing pipeline where each cell depends on the previous one completing successfully:

Example 14.5: Cell 1

```
# Load and clean data
df = pd.read_csv('data/sales_data.csv')
```

Example 14.6: Cell 2

```
# Feature engineering
df['profit_margin'] = df['revenue'] / df['cost']
df['profit_margin'] = df['profit_margin'].round(2)
```

Example 14.7: Cell 3

```
# Remove outliers
df = df[df['profit_margin'] < 10]
```

Example 14.8: Cell 4

```
# Calculate statistics
mean_profit = df['profit_margin'].mean()
```

```
print(f"Average profit margin: {mean_profit:.2f}%")
```

When executed in the correct sequence, this code produces the expected results. However, executing cells in a different order can lead to incorrect calculations or errors.

For example, if you run Cell 4 before Cell 3, you'll get statistics that include outliers.

Example 14.9: Cell 4

```
# Calculate statistics
mean_profit = df['profit_margin'].mean()
print(f"Average profit margin: {mean_profit:.2f}%")
```

Example 14.10: Cell 3

```
# Remove outliers
df = df[df['profit_margin'] < 10]
```

If you run Cell 3 before Cell 2, you'll get a `KeyError` because `profit_margin` hasn't been created yet.

Example 14.11: Cell 3

```
# Remove outliers
df = df[df['profit_margin'] < 10]
```

```
KeyError: 'profit_margin'
```

Example 14.12: Cell 2

```
# Feature engineering
df['profit_margin'] = df['revenue'] / df['cost']
df['profit_margin'] = df['profit_margin'].round(2)
```

14.1.4 Version Control Issues

Jupyter notebooks use a JSON file format to store code and output together. This single-file approach makes version control challenging

and creates several problems:

- **Unreadable diffs**: When you make changes to your notebook, Git shows the raw JSON, making it hard to understand what actually changed in your code.
- **Large file sizes**: Notebooks can grow very large when they contain plot outputs, large data frames, and model training results.
- **Merge conflicts**: When multiple people edit the same notebook, the JSON format makes it difficult to resolve conflicts.
- **GitHub rendering**: Large output files can cause GitHub to fail to render notebook diffs properly, making code reviews impossible.

Let's say we have the following notebook cells and their outputs:

Example 14.13: Cell 1

```
data = [1, 2, 3]
```

Example 14.14: Cell 2

```
print(f'Sum: {sum(data)}')
```

The underlying JSON structure contains metadata, execution counts, and outputs that make tracking actual code changes difficult:

```
{
  "cells": [
    {
      "cell_type": "code",
      "execution_count": 1,
      "outputs": [],
      "source": ["data = [1, 2, 3]"]
    },
    {
      "cell_type": "code",
      "execution_count": 2,
```

```
      "outputs": [
        {
          "name": "stdout",
          "output_type": "stream",
          "text": ["Sum: 6\n"]
        }
      ],
      "source": ["print(f'Sum: {sum(data)}')"]
    }
  ]
}
```

14.1.5 Testing Issues

Testing code in Jupyter notebooks is challenging because notebooks lack built-in testing capabilities. You need external tools like ipytest.

This example shows the extra setup required, with Cell 2 using `%%ipytest -qq` to test the function defined in Cell 1:

Example 14.15: Cell 1

```
# Define function that uses global state
def calculate_profit_margin(df):
    return (df['profit'].sum() / df['sales'].sum()) * 100
```

Example 14.16: Cell 2

```
# Test the function
%%ipytest -qq
def test_profit_margin():
    df = pd.DataFrame({
        'sales': [100, 200, 300, 400, 500],
        'cost': [50, 100, 150, 200, 250]
    })

    df['profit'] = df['sales'] - df['cost']

    margin = calculate_profit_margin(df)
    assert margin == 50.0  # Expected margin
```

14.2 Best Practices for Jupyter Notebooks

Here are some best practices for using Jupyter notebooks to avoid the issues we've discussed:

1. **Use Functions or Classes:** Encapsulate code in functions or classes with clear inputs and outputs. This reduces hidden state dependencies and makes code easier to understand and maintain.
2. **Avoid Variable Redefinition:** Use new variable names when modifying data instead of redefining variables across cells. For example, use `filtered_df` rather than reassigning to `df`.
3. **Clean Environment and Outputs:** Clear outputs and restart the kernel regularly, especially before sharing notebooks. Use "Cell" > "All Output" > "Clear" and "Cell" > "Run All" to ensure clean execution.

While these best practices can help, they're difficult to maintain consistently in an environment designed for flexible execution. Manually tracking clean environments, cleared outputs, and variable names creates extra work during development.

What if there was a modern notebook that could enforce these best practices automatically instead of relying on your memory?

marimo provides this solution by enforcing reproducibility through its design.

14.3 Use marimo for Reproducible Data Science

marimo is a next-generation Python notebook that revolutionizes the data science workflow by combining interactivity with reproducibility. Its modern design and seamless integration make it

an ideal choice for data scientists who want to create both exploratory analyses and production-ready applications.

14.3.1 Why marimo?

marimo solves three major issues common in traditional notebooks:

- **Hidden state and out-of-order execution**: marimo enforces top-to-bottom execution with clear cell dependencies.
- **Version control and diffing**: marimo notebooks are stored as plain Python scripts, making them easy to version control with Git, review changes through diffs, and collaborate with teammates through pull requests.
- **Reusability and sharing**: You can export a marimo notebook as a web app or module, making your analysis immediately interactive and reusable.

14.3.2 Getting Started

To install marimo, run:

```
pip install marimo
```

To start a new notebook, run:

```
marimo edit my_notebook.py
```

Your notebook opens in a browser, but stays a clean `.py` file under the hood.

14.3.3 Auto-Update Dependent Cells

A major strength of marimo is automatic dependency tracking. When you modify a cell, marimo automatically detects which other cells depend on its outputs and re-executes them in the correct order.

For example, when you create a data filtering workflow with two cells:

Example 14.17: Cell 1

```
# Define threshold
threshold = 30
```

Example 14.18: Cell 2

```
# Filter data using threshold
data = [20, 40, 60, 80]
filtered = [x for x in data if x > threshold]
print(filtered)
```

```
[40, 60, 80]
```

If you change `threshold` to `50` in Cell 1, marimo **automatically** detects the dependency and reruns Cell 2 with the new value:

Example 14.19: Cell 1

```
# Updated threshold
threshold = 50
```

Example 14.20: Cell 2

```
# Automatically rerun with new threshold
data = [20, 40, 60, 80]
filtered = [x for x in data if x > threshold]
print(filtered)
```

```
[60, 80]
```

14.3.4 Prevent Variable Redefinition

marimo prevents you from redefining variables across different cells. This eliminates bugs caused by naming collisions and makes notebook logic more predictable.

For example, when you accidentally reuse a variable name across cells, marimo immediately catches the error:

Example 14.21: Cell 1

```
# Initial data
data = [1, 2, 3]
```

Example 14.22: Cell 2

```
# Attempt to redefine data
data = [4, 5, 6]
```

```
This cell wasn't run because it has errors

This cell redefines variables from other cells.
'data' was also defined by:
• cell-1
```

To fix this error, use a unique variable name in Cell 2:

Example 14.23: Cell 1

```
# Initial data
data = [1, 2, 3]
```

Example 14.24: Cell 2

```
# New variable with different name
new_data = [4, 5, 6]
```

Alternatively, use a private variable with an underscore prefix that stays local to the cell:

Example 14.25: Cell 2

```
# Private variable (not shared between cells)
_data = [4, 5, 6]
```

14.3.5 Enable Clean Version Control

Since marimo notebooks are plain Python files, they can be versioned just like any other source code.

Consider this interactive notebook workflow:

Example 14.26: Cell 1

```
# Define data
data = [1, 2, 3]
```

Example 14.27: Cell 2

```
# Process data
summary = sum(data)
print("Sum:", summary)
```

Example 14.28: Cell 3

```
# Create new dataset
data_1 = [10, 20, 30]
```

Behind the scenes, marimo saves this as a clean Python script that works seamlessly with Git and code review tools:

Example 14.29: my_notebook.py

```
import marimo

__generated_with = "0.13.0"
app = marimo.App()

@app.cell
def _():
    data = [1, 2, 3]
    return (data,)

@app.cell
def _(data):
    summary = sum(data)
    print("Sum:", summary)
    return
```

```python
@app.cell
def _():
    data_1 = [10, 20, 30]
    return

if __name__ == "__main__":
    app.run()
```

14.3.6 Add Lightweight Unit Testing

Because marimo notebooks are Python scripts, you can run pytest directly on the notebook file from the terminal. This enables seamless CI pipeline testing using standard pytest commands.

```
pytest test_example.py
```

```
test_example.py ....                                    [100%]

=================== 4 passed in 0.22s ===================
```

14.3.7 Export in Multiple Reusable Formats

marimo notebooks offer flexible export capabilities for various deployment scenarios. These formats support the complete data science lifecycle from exploration to production.

Each export mode serves a specific purpose:

- **HTML**: Great for sharing static visualizations and reports with stakeholders.
- **HTML-WASM**: Perfect for publishing interactive dashboards online without needing a backend server.
- **IPYNB**: Enables compatibility with traditional Jupyter workflows.

- **Markdown**: Ideal for documentation, blogs, or version-controlled notebooks in prose form.
- **Script**: Useful for integrating analytical logic into larger Python projects or production pipelines.

```
marimo export html my_notebook.py
marimo export html-wasm my_notebook.py
marimo export ipynb my_notebook.py
marimo export md my_notebook.py
marimo export script my_notebook.py
```

14.4 Key Takeaways

1. Traditional Jupyter Notebooks are not ideal for production because:
    - Hidden state and out-of-order execution can lead to unpredictable results
    - Variable redefinition creates confusion and makes code harder to maintain
    - Version control is challenging due to JSON format
    - Testing requires external tools and additional setup
2. Best practices for using Jupyter Notebooks:
    - Encapsulate code in functions or classes with explicit inputs/outputs
    - Avoid variable redefinition across cells
    - Regularly clear outputs and restart the kernel
    - Use descriptive variable names to track data transformations
3. marimo offers a modern alternative by:
    - Storing notebooks as plain Python scripts for better version control
    - Automatically tracking and updating cell dependencies
    - Preventing variable redefinition across cells
    - Supporting multiple export formats for different use cases
    - Enabling easy testing through standard Python testing tools