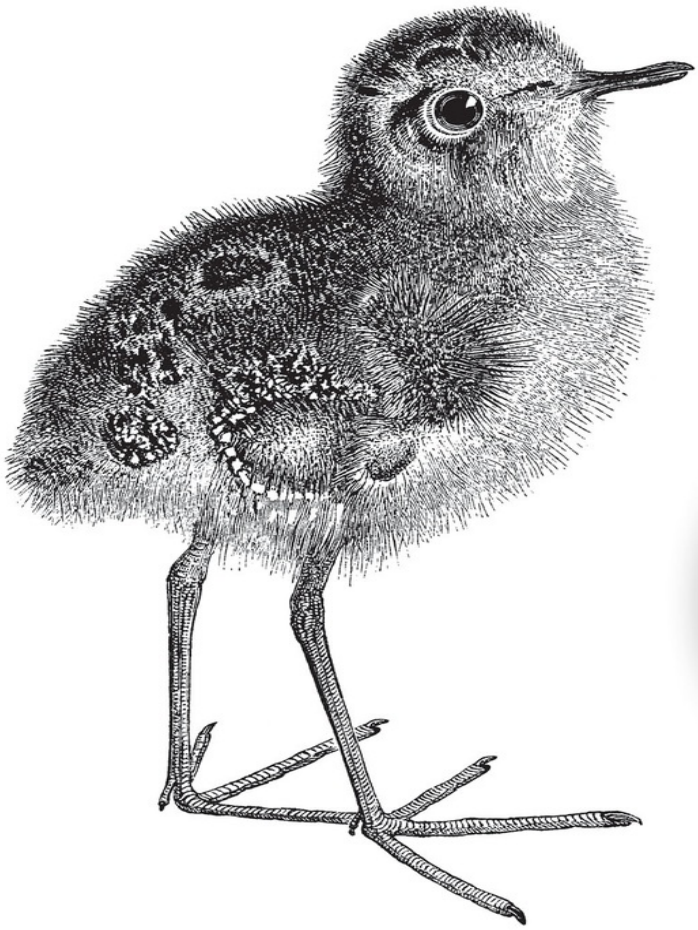
O'REILLY®

Fourth Edition

Python in a Nutshell

A Desktop Quick Reference



Early Release

RAW & UNEDITED

Alex Martelli, Anna Martelli Ravenscroft,
Steve Holden & Paul McGuire

Python in a Nutshell

A Desktop Quick Reference

FOURTH EDITION

Alex Martelli, Anna Martelli Ravenscroft, Steve Holden, and Paul McGuire

Python in a Nutshell

by Alex Martelli, Anna Martelli Ravenscroft, Steve Holden, and Paul McGuire

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editor: Angela Rufino
- Production Editor: Christopher Faucher
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- April 2017: Third Edition
- December 2022: Fourth Edition

Revision History for the Early Release

- 2022-01-28: First Release
- 2022-03-18: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781449392925 for release details.

O’Reilly Media, Inc.

978-1-098-11349-0

Chapter 1. The Python Interpreter

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

To develop software systems in Python, you write text files that contain Python source code. Use any text editor, including those in Integrated Development Environments (IDEs). Then, process the source files with the Python compiler and interpreter. You can do this directly, or within an IDE, or via another program that embeds Python. The Python interpreter also lets you execute Python code interactively, as do IDEs.

The Python Program

The Python interpreter program is run as `python` (it's named *python.exe* on Windows). `python` includes both the interpreter itself and the Python compiler, which is implicitly invoked, as and if needed, on imported modules. Depending on your system, the program may typically have to be in a directory listed in your PATH environment variable. Alternatively, as with any other program, you can give a complete pathname to it at a command (shell) prompt, or in the shell script (or shortcut target, etc.) that runs it.[1]

PEP 397

On Windows, since PEP 397, *py.exe*, the launcher, installs in the system area, meaning it is sure—barring further manipulation on your part—to be on the PATH.

On Windows, press the Windows key and start typing `python`: "Python 3.x (command-line)" appears, along with other choices, such as "IDLE (Python GUI)." If you have the *py.exe* launcher installed (which is the normal case), at any command prompt, typing `py` launches Python.

Environment Variables

Besides PATH, other environment variables affect the `python` program. Some environment variables have the same effects as options passed to `python` on the command line, as we show in the next section. Several environment variables provide settings not available via command-line options. The following list covers just the basics of a few frequently used ones; for all details, see the online docs.

`PYTHONHOME`

> The Python installation directory. A *lib* subdirectory, containing the standard Python library, must exist under this directory. On Unix-like systems, the standard library modules should be in *lib/python-3.x* for Python 3.x, where *x* is the minor Python version. If not set, Python uses some heuristics to locate the installation directory.

`PYTHONPATH`

> A list of directories, separated by colons on Unix-like systems, and by semicolons on Windows. Python can import modules from these directories. This list extends the initial value for Python's sys.path variable. We cover modules, importing, and sys.path in Chapter "Modules."

`PYTHONSTARTUP`

> The name of a Python source file to run each time an interactive interpreter session starts. No such file runs if you don't set this variable, or set it to the path of a file that is not found. The PYTHONSTARTUP file does not run when you run a Python script; it runs only when you start an interactive session.

How to set and examine environment variables depends on your operating system. In Unix, use shell commands, often within startup shell scripts. On Windows, press the Windows key and start typing `environment var` and a couple of shortcuts appear: one for user env vars, the other for system ones. On a Mac, you can work like in other Unix-like systems, but you have options, including a MacPython-specific IDE. For more information about Python on the Mac, see Using Python on a Mac.

Command-Line Syntax and Options

The Python interpreter command-line syntax can be summarized as follows:

```
[path]python {options} [-c command | -m module | file | -] {args}
```

Brackets ([]) enclose what's optional, braces ({}) enclose items of which zero or more may be present, and bars (|) mean a choice among alternatives. Python uses a slash (/) for file paths, as in Unix.

Running a Python script at a command line can be as simple as:

```
$ python hello.py
Hello World
```

You can also explicitly provide the path to the script:

```
$ python ./hello/hello.py
Hello World
```

The filename of the script can be any absolute or relative file path, and need not have any specific extension (though it is conventional to use a *.py* extension). Each operating system has its own way to make the Python scripts themselves executable, but we do not cover those details here.

Options are case-sensitive short strings, starting with a hyphen, that ask `python` for non-default behavior. `python` accepts only options that start with a hyphen (-). The most frequently used options are in Table 2-1. Each option's description gives the environment variable (if any) that, when set, requests that behavior. Many options have longer versions, starting with two hyphens, as shown by `python -h`. For all details, see the online docs.

Table 2-1. Python frequently used command-line options

Option	Meaning (and environment variable, if any)
-B	Don't save compiled bytecode files to disk (PYTHONDONTWRITEBYTECODE)
-c	Gives Python statements within the command line
-E	Ignores all environment variables
-h	Show then full list of options, then terminate
-i	Runs an interactive session after the file or command runs (PYTHONINSPECT)
-m	Specifies a Python module to run as the main script
-O	Optimizes bytecode (PYTHONOPTIMIZE)—note that this is an uppercase letter O, not the digit 0
-OO	Like `-O`, but also removes docstrings from the bytecode

-S	Omits the implicit import site on startup (covered in “The site and sitecustomize Modules”)
-t, -tt	Issues warnings about inconsistent tab usage (`-tt` instead issues errors for the same issues)
-u	Uses unbuffered binary files for standard output and standard error (PYTHONUNBUFFERED)
-v	Verbosely traces module import and cleanup actions (PYTHONVERBOSE)
-V	Prints the Python version number, then terminates
-W arg	Adds an entry to the warnings filter (see “The warnings Module”)
-x	Excludes (skips) the first line of the script’s source

Use `-i` when you want to get an interactive session immediately after running some script, with top-level variables still intact and available for inspection. You do not need `-i` for normal interactive sessions, though it does no harm.

`-O` and `-OO` yield small savings of time and space in bytecode generated for modules you import, turning assert statements into no-operations, as covered in “The assert Statement.” `-OO` also discards documentation strings.[2]

After the options, if any, tell Python which script to run. A file path means a Python source or bytecode file to run; on any platform, you may use a slash (/) to separate components in this path. On Windows only, you may alternatively use a backslash (\). Instead of a file path, you can use `-c command` to execute a Python code string `command`. `command` normally contains spaces, so you need quotes around it to satisfy your operating

system's shell or command-line processor. Some shells (e.g., `bash`) let you enter multiple lines as a single argument, so that `command` can be a series of Python statements. Other shells (e.g., Windows shells) limit you to a single line; `command` can then be one or more simple statements separated by semicolons (;), as we discuss in "Statements."

Another way to specify which Python script to run is `-m module`. This option tells Python to load and run a module named `module` (or the *__main__.py* member of a package or ZIP file named `module`) from some directory that is part of Python's sys.path; this is useful with several modules from Python's standard library. For example, as covered in "The timeit module," `-m timeit` is often the best way to perform micro-benchmarking of Python statements.

A hyphen, or the lack of any token in this position, tells the interpreter to read program source from standard input—normally, an interactive session. You need a hyphen only if arguments follow. `args` are arbitrary strings; the Python you run can access these strings as items of the list sys.argv.

For example, on a standard Windows installation, you can enter the following at a command prompt to have Python print the current date and time:

```
C:\> py -c "import time; print(time.asctime())"
```

On Cygwin, Linux, OpenBSD, macOS, and other Unix-like systems, with a default installation of Python from sources, enter the following at a shell prompt to start an interactive session with verbose tracing of module import and cleanup:

```
$ /usr/local/bin/python -v
```

You can start the command with just `python` (you do not have to specify the full path to Python) if the directory of the Python executable is in your PATH environment variable. (If you have multiple versions of Python installed, you can specify the version, with, for example, `python3`, or

`python3.10`, as appropriate; then, the version used if you just say `python` is the one you installed most recently.)

Interactive Sessions

When you run `python` without a script argument, Python starts an interactive session and prompts you to enter Python statements or expressions. Interactive sessions are useful to explore, to check things out, and to use Python as a powerful, extensible interactive calculator. (IPython, mentioned in "IPython," is like "Python on steroids" specifically for interactive-session usage.)

When you enter a complete statement, Python executes it. When you enter a complete expression, Python evaluates it. If the expression has a result, Python outputs a string representing the result and also assigns the result to the variable named _ (a single underscore) so that you can immediately use that result in another expression. The prompt string is >>> when Python expects a statement or expression and ... when a statement or expression has been started but not completed. In particular, Python prompts you with ... when you have opened a parenthesis (or other matched delimiter) on a previous line and have not closed it yet.

There are several ways you can end an interactive session. The most common are:

- Enter the end-of-file keystroke for your operating system (Ctrl-Z on Windows, Ctrl-D on Unix-like systems).

- Execute either of the built-in functions quit or exit, using the form quit() or exit(). (Omitting the trailing ()'s will display a message like "Use quit() or Ctrl-D (i.e. EOF) to exit," but will still leave you in the interpreter.)

- Execute the statement raise SystemExit, or call sys.exit(), either interactively or in running code (we cover SystemExit and raise in Chapter "Exceptions").

NOTE

Use the Python Interactive Interpreter for Simple Experimenting

Trying out Python statements in the interactive interpreter is a quick way to experiment with Python and immediately see the results. For example, here is a simple test of the built-in enumerate function:

```
>>> print(list(enumerate("abc")))
            (0, 'a'), (1, 'b'), (2, 'c')]
```

The interactive interpreter is a great introductory platform for learning basic Python syntax and features. (Even experienced Python developers will often open a Python interpreter to quickly check out an infrequently-used command or function.)

Line-editing and history facilities depend in part on how Python was built: if the readline module was included, all features of the GNU readline library are available. Windows has a simple but usable history facility for interactive textmode programs like *python*. To use other line-editing and history facilities, install pyreadline on Windows, or pyrepl for Unix.

In addition to the built-in Python interactive environment, and those offered as part of richer development environments covered in the next section, you can freely download other alternative, powerful interactive environments. The most popular one is IPython, covered in “IPython,” which offers a dazzling wealth of features. A simpler, lighter-weight, but still quite handy alternative read-line interpreter is bpython.

Python Development Environments

The Python interpreter’s built-in interactive mode is the simplest development environment for Python. It is primitive, but is lightweight, has a small footprint, and starts fast. Together with a good text editor (as discussed in “Free Text Editors with Python Support”), and line-editing and history facilities, the interactive interpreter (or, alternatively, the much more powerful IPython/Jupyter command-line interpreter) is a usable

development environment. However, there are several other development environments you can use.

IDLE

Python's Integrated DeveLopment Environment (IDLE) comes with standard Python distributions on most platforms. IDLE is a cross-platform, 100% pure Python application based on the Tkinter GUI. IDLE offers a Python shell similar to the interactive Python interpreter, but richer. It also includes a text editor optimized to edit Python source code, an integrated interactive debugger, and several specialized browsers/viewers.

For more functionality in IDLE, install IdleX, a substantial collection of free third-party extensions to it.

To install and use IDLE in macOS, follow these specific instructions.

Other Python IDEs

IDLE is mature, stable, easy, fairly rich, and extensible. There are, however, many other IDEs—cross-platform and platform-specific, free and commercial (including commercial IDEs with free offerings, especially if you're developing open source software), standalone and add-ons to other IDEs.

Some of these IDEs sport features such as static analysis, GUI builders, debuggers, and so on. Python's IDE wiki page lists over 30, and points to many other URLs with reviews and comparisons. If you're an IDE collector, happy hunting!

We can't do justice to even a tiny subset of those IDEs, but it's worth singling out the popular cross-platform, cross-language modular IDE Eclipse: the free third-party plug-in PyDev for Eclipse has excellent Python support. Steve is a long-time user of Wing IDE by Archaeopteryx, the most venerable Python-specific IDE. Paul's IDE of choice, and perhaps the single most popular third-party Python IDE today may be PyCharm. And, not to be overlooked, Microsoft's Visual Studio Code (also referred to as Visual Studio, or VSCode) is an excellent cross-platform IDE, with support

for a number of languages, including Python. If you use Visual Studio, check out PTVS, an open source plug-in that's particularly good at allowing mixed-language debugging in Python and C as and when needed.

Free Text Editors with Python Support

You can edit Python source code with any text editor, even simplistic ones such as Notepad on Windows or *ed* on Linux. Powerful free editors support Python with extra features such as syntax-based colorization and automatic indentation. Cross-platform editors let you work in uniform ways on different platforms. Good text editors also let you run, from within the editor, tools of your choice on the source code you're editing. An up-to-date list of editors for Python can be found on the Python wiki, which lists dozens of them.

The very best for sheer editing power may be classic Emacs (see the Python wiki for Python-specific add-ons). Emacs is not easy to learn, nor is it lightweight. Alex's personal favorite is another classic, vim, Bram Moolenaar's improved version of traditional Unix editor *vi*: perhaps not *quite* as powerful as Emacs, but still well worth considering—fast, lightweight, Python-programmable, runs everywhere in both text-mode and GUI versions, and excellently taught in O'Reilly's book "Learning the vi and vim editors," now in its 8th edition. See the Python wiki for Python-specific tips and add-ons. Steve and Anna also use *vim*. Where it's available, Steve also uses the commercial editor *Sublime Text 2*, with good syntax coloring and enough integration to run your programs from inside the editor. For quick editing and executing of short Python scripts (and as a fast and lightweight general text editor, even for multi-megabyte text files), SciTE is Paul's go-to editor.

Tools for Checking Python Programs

The Python compiler does not check programs and modules thoroughly: the compiler checks only the code's syntax. If you want more thorough checking of your Python code, download and install third-party tools for the purpose. Pyflakes is a very fast, lightweight checker: it's not thorough, but

does not import the modules it's checking, which makes using it safer. At the other end of the spectrum, PyLint is very powerful and highly configurable. PyLint is not lightweight, but repays that by being able to check many style details in a highly configurable way based on customizable configuration files.

For more thorough checking of Python code for proper variable type usages, tools like mypy are used; see more discussion in "Type Annotations."

Running Python Programs

Whatever tools you use to produce your Python application, you can see your application as a set of Python source files, which are normal text files. A *script* is a file that you can run directly. A *module* is a file that you can import (as covered in Chapter "Modules") to provide functionality to other files or interactive sessions. A Python file can be *both* a module (providing functionality when imported) *and* a script (OK to run directly). A useful and widespread convention is that Python files that are primarily intended to be imported as modules, when run directly, should execute some self-test operations, as covered in "Testing."

The Python interpreter automatically compiles Python source files as needed. Python source files normally have the extension *.py*. Python saves the compiled bytecode in subdirectory *__pycache__* of the directory with the module's source, with a version-specific extension, and annotated to denote optimization level.

Run Python with option `-B` to avoid saving compiled bytecode to disk, which can be handy when you import modules from a read-only disk. Also, Python does not save the compiled bytecode form of a script when you run the script directly; rather, Python recompiles the script each time you run it. Python saves bytecode files only for modules you import. It automatically rebuilds each module's bytecode file whenever necessary—for example, when you edit the module's source. Eventually, for deployment, you may

package Python modules using tools covered in Chapter "Distributing Extensions and Programs."

You can run Python code with the Python interpreter or an IDE[3]. Normally, you start execution by running a top-level script. To run a script, give its path as an argument to `python`, as covered earlier in "The python Program." Depending on your operating system, you can invoke `python` directly from a shell script or command file. On Unix-like systems, you can make a Python script directly executable by setting the file's permission bits x and r and beginning the script with a *shebang* line, a line such as:

```
#!/usr/bin/env python
```

or some other line starting with #! followed by a path to the *python* interpreter program, in which case you can optionally add a single word of options, for example:

```
#!/usr/bin/python -O
```

On Windows, you can use the same style #! line, in accordance with PEP 397, to specify a particular version of Python, so your scripts can be cross-platform between Unix-like and Windows systems. You can also run Python scripts with the usual Windows mechanisms, such as double-clicking their icons. When you run a Python script by double-clicking the script's icon, Windows automatically closes the text-mode console associated with the script as soon as the script terminates. If you want the console to linger (to allow the user to read the script's output on the screen), ensure the script doesn't terminate too soon. For example, use, as the script's last statement:

```
input('Press Enter to terminate')
```

This is not necessary when you run the script from a command prompt.

On Windows, you can also use extension *.pyw* and interpreter program *pythonw.exe* instead of *.py* and *python.exe*. The *w* variants run Python

without a text-mode console, and thus without standard input and output. This is good for scripts that rely on GUIs, or run invisibly in the background. Use them only when a program is fully debugged, to keep standard output and error available for information, warnings, and error messages during development. On a Mac, use interpreter program `pythonw`, rather than `python`, when you want to run a script that needs to access any GUI toolkit, rather than just text-mode interaction.

Applications coded in other languages may embed Python, controlling the execution of Python for their own purposes. We examine this briefly in "Embedding Python."

The PyPy Interpreter

PyPy may be run similarly to *python*:

```
[path]pypy {options} [-c command | file | - ] {args}
```

See the PyPy homepage for complete, up-to-date information.

---

1 This may involve using quotes if the pathname contains spaces—again, this depends on your operating system.

2 This may affect code that parses docstrings for meaningful purposes; we suggest you avoid writing such code.

3 or, online: one of the authors, for example, maintains an online list of online Python interpreters.

Chapter 2. The Python Language

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

This chapter is a guide to the Python language. To learn Python from scratch, we suggest you start with the appropriate links from the online docs. If you already know at least one other programming language well, and just want to learn specifics about Python, this chapter is for you. However, we're not trying to teach Python: we cover a lot of ground at a pretty fast pace. We focus on the rules, and only secondarily point out best practices and style; as your Python style guide, use PEP 8 (optionally augmented by extra guidelines such as The Hitchhiker's Guide, CKAN's, and Google's).

Lexical Structure

The *lexical structure* of a programming language is the set of basic rules that govern how you write programs in that language. It is the lowest-level syntax of the language, specifying such things as what variable names look like and how to denote comments. Each Python source file, like any other text file, is a sequence of characters. You can also usefully consider it a sequence of lines, tokens, or statements. These different lexical views

complement each other. Python is very particular about program layout, especially lines and indentation: pay attention to this information if you are coming to Python from another language.

Lines and Indentation

A Python program is a sequence of *logical lines*, each made up of one or more *physical lines*. Each physical line may end with a comment. A hash sign `#` that is not inside a string literal starts a comment. All characters after the `#`, up to but excluding the line end, are the comment: Python ignores them. A line containing only whitespace, possibly with a comment, is a *blank line*: Python ignores it. In an interactive interpreter session, you must enter an empty physical line (without any whitespace or comment) to terminate a multiline statement.

In Python, the end of a physical line marks the end of most statements. Unlike in other languages, you don't normally terminate Python statements with a delimiter, such as a semicolon (`;`). When a statement is too long to fit on a physical line, you can join two adjacent physical lines into a logical line by ensuring that the first physical line has no comment and ends with a backslash (`\`). However, Python also automatically joins adjacent physical lines into one logical line if an open parenthesis (`(`), bracket (`[`), or brace (`{`) has not yet been closed: take advantage of this mechanism to produce more readable code than you'd get with backslashes at line ends. Triple-quoted string literals can also span physical lines. Physical lines after the first one in a logical line are known as *continuation lines*. Indentation rules apply to the first physical line of each logical line, not to continuation lines.

Python uses indentation to express the block structure of a program. Python does not use braces, or other begin/end delimiters, around blocks of statements; indentation is the only way to denote blocks. Each logical line in a Python program is *indented* by the whitespace on its left. A *block* is a contiguous sequence of logical lines, all indented by the same amount; a logical line with less indentation ends the block. All statements in a block must have the same indentation, as must all clauses in a compound statement. The first statement in a source file must have no indentation (i.e.,

must not begin with any whitespace). Statements that you type at the interactive interpreter primary prompt >>> (covered in "Interactive Sessions") must also have no indentation.

Python treats each tab as if it was up to eight spaces, so that the next character after the tab falls into logical column 9, 17, 25, and so on. Standard Python style is to use four spaces (*never* tabs) per indentation level.

If you must use tabs, Python does not allow mixing tabs and spaces for indentation.

Use Spaces, Not Tabs

Configure your favorite editor to expand a Tab keypress into four spaces, so that all Python source code you write contains just spaces, not tabs. This way, all tools, including Python itself, are consistent in handling indentation in your Python source files. Optimal Python style is to indent blocks by exactly four spaces: use no tab characters.

Character Sets

A Python source file can use any Unicode character, encoded by default as UTF-8. (Characters with codes between 0 and 127, AKA *ASCII characters*, encode in UTF-8 into the respective single bytes, so an ASCII text file is a fine Python source file, too.)

You may choose to tell Python that a certain source file is written in a different encoding. In this case, Python uses that encoding to read the file. To let Python know that a source file is written with a nonstandard encoding, start your source file with a comment whose form must be, for example:

```
# coding: iso-8859-1
```

After `coding:`, write the name of an ASCII-compatible codec from the `codecs` module, such as `utf-8` or `iso-8859-1`. Note that this *coding directive* comment (also known as an *encoding declaration*) is taken as such only if it is at the start of a source file (possibly after the "shebang line" covered in "Running Python Programs"). Best practice is to use `utf-8` for all of your text files, including Python source files.

Tokens

Python breaks each logical line into a sequence of elementary lexical components known as *tokens*. Each token corresponds to a substring of the logical line. The normal token types are *identifiers, keywords, operators, delimiters*, and *literals*, which we cover in the following sections. You may freely use whitespace between tokens to separate them. Some whitespace separation is necessary between logically adjacent identifiers or keywords; otherwise, Python would parse them as a single, longer identifier. For example, `ifx` is a single identifier; to write the keyword `if` followed by the identifier `x`, you need to insert some whitespace (typically only one space character, i.e., you write `if x`).

Identifiers

An *identifier* is a name used to specify a variable, function, class, module, or other object. An identifier starts with a letter (any character that Unicode classifies as a letter) or an underscore (_), followed by zero or more letters, underscores, digits or other characters that Unicode classifies as digits or combining marks (as defined in Unicode® Standard Annex #31).

For example, in the 8-bit ASCII-plus character range, the valid leading characters for an identifier are:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
ªµºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþÿ
```

After the leading character, the valid identifier body characters are:

```
0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_abcdefghijklmnopqrstuvwxyz
ªµ·ºÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖØÙÚÛÜÝÞßàáâãäåæçèéêëìíîïðñòóôõöøùúûüýþ
ÿ
```

Case is significant: lowercase and uppercase letters are distinct. Punctuation characters such as `@`, `$`, and `!` are not allowed in identifiers.

Beware of Using Unicode Characters that are Homoglyphs

Some Unicode characters look very similar to, if not indistinguishable from, other characters - such character pairs are called *homoglyphs*. For instance, compare capital letter ‘A’ and capital Greek letter alpha 'Α'. These are actually two different letters that just look very similar in most fonts. In Python, they define two different variables:

```
>>> A = 100
>>> Α = 200  # this variable is GREEK CAPITAL LETTER ALPHA
>>> print(A, Α)
100 200
```

If you want to make your Python code widely usable, we recommend a policy that all identifiers, comments, and documentation are written in English, avoiding, in particular, non-English homoglyph characters. For more information, see PEP 3131.

Unicode normalization strategies add further complexities (Python uses NFKC normalization when parsing identifiers containing Unicode characters). See Jukka K. Korpela’s “Unicode Explained” and other technical information at https://unicode.org, particularly all books in the list they recommend.

Unicode Normalization can Create Unintended Overlap Between Variables

Python may create an unintended alias between variables when one contains certain Unicode characters. This normalization internally converts the name as shown in the Python script to one using normalized characters. For example, the letters “ª” and “º” normalize to the ASCII lowercase letters “a” and “o”. See how variables using these letters could clash with other variables:

```
>>> a, o = 100, 101
>>> ª, º = 200, 201
>>> print(a, o, ª, º)
200 201 200 201  # not "100 101 200 201"
```

It is best to avoid using normalizable Unicode characters in your Python identifiers.

Normal Python style is to start class names with an uppercase letter, and other identifiers with a lowercase letter. Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be private. Starting an identifier with two leading underscores indicates a *strongly private* identifier; if the identifier also ends with two trailing underscores, however, this means that the identifier is a language-defined special name. Identifiers composed of multiple words should be all lowercase with underscores between words, sometimes referred to as “snake case,” as in `login_password`.

Single Underscore _ in the Interactive Interpreter

The identifier _ (a single underscore) is special in interactive interpreter sessions: the interpreter binds _ to the result of the last expression statement it has evaluated interactively, if any.

Keywords

Python has 35 keywords, which are identifiers that Python reserves for special syntactic uses. Like identifiers, keywords are case-sensitive. You cannot use keywords as regular identifiers (thus, they're sometimes known as "reserved words"). Some keywords begin simple statements or clauses of compound statements, while other keywords are operators. We cover all the keywords in detail in this book, either in this chapter or in Chapters "Object-Oriented Python", "Exceptions", and "Modules". The keywords in Python are:

```
and       break    elif     from    is       pass     with

as        class    else     global  lambda   raise    yield

assert    continue except   if      nonlocal return   False

```

async	def	finally	import	not	try	None
await	del	for	in	or	while	True

You can list them by importing the `keyword` module and printing `keyword.kwlist`.

||**3.9**++|| In addition, Python 3.9 introduced the concept of *soft keywords*. Soft keywords are keywords that are context-sensitive. They are language keywords for some specific syntax constructs, but outside of those constructs they may be used as variable or function names, so they are *not* reserved words. No soft keywords were defined in Python 3.9, but Python 3.10 introduced the following soft keywords:

_	case	match

You can list them from the `keyword` module by printing `keyword.softkwlist`.

Operators

Python uses non-alphanumeric characters and character combinations as operators. Python recognizes the following operators, which are covered in detail in “Expressions and Operators”:

```
+   -   *   /   %    **   //   <<   >>   &    @
|   ^   ~   <   <=   >    >=   <>   !=   ==   @=   :=
```

You can use `@` as an operator (in matrix multiplication, covered in Chapter 15), although the character is technically a delimiter.

Delimiters

Python uses the following characters and combinations as delimiters in expressions, list, dictionary, and set literals and comprehensions, and various statements, among other purposes:

```
(     )     [     ]     {     }
,     :     .     `     =     ;     @
+=    -=    *=    /=    //=   %=
&=    |=    ^=    >>=   <<=   **=
```

The period (`.`) can also appear in floating-point literals (e.g., `2.3`) and imaginary literals (e.g., `2.3j`). The last two rows are the augmented assignment operators, which are delimiters, but also perform operations. We discuss the syntax for the various delimiters when we introduce the objects or statements using them.

The following characters have special meanings as part of other tokens:

```
'   "   #   \
```

`'` and `"` surround string literals. `#` outside of a string starts a comment, which ends at the end of the current line. `\` at the end of a physical line joins the following physical line into one logical line; `\` is also an escape character in strings. The characters `$` and `?`, and all control characters[1] except whitespace, can never be part of the text of a Python program, except in comments or string literals.

Literals

A *literal* is the direct denotation in a program of a data value (a number, string, or container). The following are number and string literals in Python:

```
42                      # Integer literal
3.14                    # Floating-point literal
1.0j                    # Imaginary literal
'hello'                 # String literal
"world"                 # Another string literal
"""Good
night"""                # Triple-quoted string literal, spanning
2 lines
```

Combining number and string literals with the appropriate delimiters, you can build literals that directly denote data values of container types:

```
[42, 3.14, 'hello']     # List
[]                      # Empty list
100, 200, 300           # Tuple
()                      # Empty tuple
{'x':42, 'y':3.14}      # Dictionary
{}                      # Empty dictionary
{1, 2, 4, 8, 'string'}  # Set

# There is no literal to denote an empty set; use set() instead
```

We cover the syntax for literals in detail in “Data Types”, when we discuss the various data types Python supports.

Statements

You can look at a Python source file as a sequence of simple and compound statements.

Simple statements

A *simple statement* is one that contains no other statements. A simple statement lies entirely within a logical line. As in many other languages, you may place more than one simple statement on a single logical line, with a semicolon (`;`) as the separator. However, one statement per line is the usual and recommended Python style, and makes programs more readable.

Any *expression* can stand on its own as a simple statement (we discuss expressions in "Expressions and Operators"). When working interactively, the interpreter shows the result of an expression statement you enter at the prompt (>>>) and binds the result to a global variable named _ (underscore). Apart from interactive sessions, expression statements are useful only to call functions (and other *callables*) that have side effects (e.g., perform output, change global variables, or raise exceptions).

An *assignment* is a simple statement that assigns values to variables, as we discuss in "Assignment Statements". An assignment in Python using the = operator is a statement and can never be part of an expression. To perform an assignment as part of an expression, you must use the `:=` (jokingly known as the "walrus") operator.

Compound statements

A *compound statement* contains one or more other statements and controls their execution. A compound statement has one or more *clauses*, aligned at the same indentation. Each clause has a *header* starting with a keyword and ending with a colon (`:`), followed by a *body*, which is a sequence of one or more statements. Normally, these statements, also known as a *block*, are on separate logical lines after the header line, indented four spaces rightward. The block lexically ends when the indentation returns to that of the clause header (or further left from there, to the indentation of some enclosing compound statement). Alternatively, the body can be a single simple statement, following the `:` on the same logical line as the header. The body may also consist of several simple statements on the same line with semicolons between them, but, as we've already mentioned, this is not good Python style.

Data Types

The operation of a Python program hinges on the data it handles. Data values in Python are known as *objects*; each object, AKA *value*, has a *type*. An object's type determines which operations the object supports (in other

words, which operations you can perform on the value). The type also determines the object's *attributes* and *items* (if any) and whether the object can be altered. An object that can be altered is known as a *mutable object*, while one that cannot be altered is an *immutable object*. We cover object attributes and items in "Object attributes and items".

The built-in `type`(*obj*) accepts any object as its argument and returns the type object that is the type of *obj*. The built-in function `isinstance`(*obj, type*) returns `True` when object *obj* has type *type* (or any subclass thereof); otherwise, it returns `False`.

Python has built-in types for fundamental data types such as numbers, strings, tuples, lists, dictionaries, and sets, as covered in the following sections. You can also create user-defined types, known as *classes*, as discussed in "Classes and Instances".

Numbers

The built-in numeric types in Python include integers, floating-point numbers, and complex numbers. The standard library also offers decimal floating-point numbers, covered in "The decimal Module", and fractions, covered in "The fractions Module". All numbers in Python are immutable objects; therefore, when you perform an operation on a number object, you produce a new number object. We cover operations on numbers, also known as arithmetic operations, in "Numeric Operations".

Numeric literals do not include a sign: a leading `+` or `-`, if present, is a separate operator, as discussed in "Arithmetic Operations".

Integer numbers

Integer literals can be decimal, binary, octal, or hexadecimal. A decimal literal is a sequence of digits in which the first digit is nonzero. A binary literal is `0b` followed by a sequence of binary digits (`0` or `1`). An octal literal is `0o` followed by a sequence of octal digits (`0` to `7`). A hexadecimal literal is `0x` followed by a sequence of hexadecimal digits (`0` to `9` and `A` to `F`, in either upper- or lowercase). For example:

```
1, 23, 3493                 # Decimal integer literals
0b010101, 0b110010          # Binary integer literals
0o1, 0o27, 0o6645           # Octal integer literals
0x1, 0x17, 0xDA5, 0xda5     # Hexadecimal integer literals
```

Integer literals have no defined upper bound.

An `int` object *i* supports the following methods:

as_integer_ratio	`||3.8++|| i.as_integer_ratio()` Returns a tuple of 2 `int`s, whose exact ratio is the original integer value. (Since *i* is always `int`, the tuple is always `(i, 1)`; compare with `float.`*as_integer_ratio*.)
bit_count	`||3.10++|| i.bit_count()` Returns the number of ones in a binary representation of `abs`*(i)*.
bit_length	*i*`.bit_length()` Returns the minimum number of bits needed to represent *i*. Equivalent to the length of the binary representation of `abs`*(i)*, after removing `'b'` and all leading zeros. *(0).bit_length()* returns 0.
to_bytes	*i*`.to_bytes(`*length*`, `*byteorder*`, `*signed*`=False)` Returns a `bytes` value *length* bytes in size representing the binary value of *i*. byteorder must be the str value `'big'` or `'little'`, indicating whether the return value should be big-endian (most-significant byte first) or little-endian (least-significant byte first). For example, `(258).to_bytes(2, 'big')` returns `b'\x01\x02'`, and `(258).to_bytes(2, 'little')` returns `b'\x02\x01'`. When *i* `< 0` and *signed* is `True`, `to_bytes` returns the bytes of *i* represented in 2's complement. If *i* `< 0` and *signed* is `False`, `to_bytes` raises `OverflowError`.
from_bytes	`int.from_bytes(`*bytes_value*`,` byteorder`,` *signed*`=False)` Returns an `int` from the bytes in *bytes_value* following the same argument usage as in `to_bytes`. (Note that `from_bytes` is a classmethod of `int`.)

Floating-point numbers

A floating-point literal is a sequence of decimal digits that includes a decimal point (`.`), an exponent suffix (`e` or `E`, optionally followed by `+` or `-`, followed by one or more digits), or both. The leading character of a floating-point literal cannot be `e` or `E`; it may be any digit or a period (`.`). For example:

```
0., 0.0, .0, 1., 1.0, 1e0, 1.e0, 1.0e0   # Floating-point
literals
```

A Python floating-point value corresponds to a C `double` and shares its limits of range and precision, typically 53 bits of precision on modern platforms. (For the exact range and precision of floating-point values on the current platform, and many other details, see `sys.float_info`: we do not cover that in this book—see the online docs.)

A float object *f* supports the following methods:

as_integer_ratio	*f*`.as_integer_ratio()`
	Returns a tuple of 2 `int`s, a numerator and a denominator, whose exact ratio is the original float value, *f*. Example:
	>>> *f*=2.5
	>>> *f*.as_integer_ratio()
	(5, 2)

is_integer	*f*`.is_integer()` Returns a bool value indicating if *f* is an integer value. Equivalent to `int`(*f*) == *f*.
hex	`f.hex()` Returns a hexadecimal representation of *f*, with leading `0x` and trailing `p` and exponent. For example, `(99.0).hex()` returns `'0x1.8c00000000000p+6'`.
from_hex	`float.from_hex(s)` Returns a `float` value from the hexadecimal `str` value *s*. *s* can be of the form returned by `f.hex()`, or simply a string of hexadecimal digits. When the latter is the case, `from_hex` returns `float(int(`*s*`,16))`.

Complex numbers

A complex number is made up of two floating-point values, one each for the real and imaginary parts. You can access the parts of a complex object `z` as read-only attributes `z.real` and `z.imag`. You can specify an imaginary literal as any floating-point or integer decimal literal followed by a `j` or `J`:

```
0j, 0.j, 0.0j, .0j, 1j, 1.j, 1.0j, 1e0j, 1.e0j, 1.0e0j
```

The `j` at the end of the literal indicates the square root of `-1`, as commonly used in electrical engineering (some other disciplines use `i` for this purpose, but Python has chosen `j`). There are no other complex literals. To denote any constant complex number, add or subtract a floating-point (or integer) literal and an imaginary one. For example, to denote the complex number that equals one, use expressions like `1+0j` or `1.0+0.0j`. Python performs the addition or subtraction at compile time, so, no need to worry about overhead.

A complex object *c* supports the following method:

conjugate	*c*`.conjugate()` Returns a new complex number `complex(`*c*`.imag,`*c*`.real)` (i.e., the return value has *c*'s `real` and `imag` attributes exchanged).

See "The math and cmath Modules" for several other functions that use floats and complex numbers.

Underscores in numeric literals

To assist visual assessment of the magnitude of a number, numeric literals can include single underscore (_) characters between digits or after any base specifier. It's not only decimal numeric constants that can benefit from this notational freedom:

```
>>> 100_000.000_0001, 0x_FF_FF, 0o7_777, 0b_1010_1010
(100000.0000001, 65535, 4095, 170)
```

There is no enforcement of location of the underscores (except that two may not occur consecutively), so `123_456` and `12_34_56` both represent the same `int` as `123456`.

Sequences

A *sequence* is an ordered container of items, indexed by integers. Python has built-in sequence types known as strings (bytes and Unicode), tuples, and lists. Library and extension modules provide other sequence types, and

you can write yet others yourself (as discussed in “Sequences”). You can manipulate sequences in a variety of ways, as discussed in “Sequence Operations”.

Iterables

A Python concept that generalizes the idea of “sequence” is that of *iterables*, covered in “The for Statement,” “Iterators,” and “Iterables vs. Iterators”. All sequences are iterable: whenever we say you can use an iterable, you can, in particular, use a sequence (for example, a list).

Also, when we say that you can use an iterable, we mean, usually, a *bounded* iterable: an iterable that eventually stops yielding items. All sequences are bounded. Iterables, in general, can be unbounded, but, if you try to use an unbounded iterable without special precautions, you could produce a program that never terminates, or one that exhausts all available memory.

Strings

Python has two built-in string types, *str* and *bytes*[2]. A `str` object is a sequence of characters used to store and represent text-based information. A `bytes` object stores and represents arbitrary sequences of binary bytes. Strings of both types in Python are *immutable*: when you perform an operation on strings, you always produce a new string object of the same type, rather than mutating an existing string. String objects provide many methods, as discussed in detail in “Methods of String and Bytes Objects”.

A string literal can be quoted or triple-quoted. A quoted string is a sequence of zero or more characters within matching quotes, single (`'`) or double (`"`). For example:

```
'This is a literal string'
"This is another string"
```

The two different kinds of quotes function identically; having both lets you include one kind of quote inside of a string specified with the other kind, with no need to escape quote characters with the backslash character (`\`):

```
'I\'m a Python fanatic'          # a quote can be escaped
"I'm a Python fanatic"           # this way may be more readable
```

Most (but not all) style guides that pronounce on the subject suggest that you use single quotes when the choice is otherwise indifferent.

To have a string literal span multiple physical lines, you can use a `\` as the last character of a line to indicate that the next line is a continuation:

```
'A not very long string \
that spans two lines'       # comment not allowed on previous
line
```

You can embed a newline in the string to make it print over two lines rather than just one:

```
'A not very long string\n\
that prints on two lines'   # comment not allowed on previous
line
```

A better approach is to use a triple-quoted string, enclosed by matching triplets of quote characters (`'''`, or better, as mandated by PEP 8, `"""`). In a triple-quoted string literal, line breaks in the literal remain as newline characters in the resulting string object:

```
"""An even bigger
string that spans
three lines"""              # comments not allowed on previous
lines
```

You can start a triple-quoted literal with an escaped newline, to avoid having the first line of the literal string's content at a different indentation level from the rest. For example:

```
the_text = """\
First line
Second line
"""  # the same as "First line\nSecond line\n" but more readable
```

The only character that cannot be part of a triple-quoted string literal is an unescaped backslash, while a single-quoted string literal cannot contain unescaped backslashes, nor line ends, nor the quote character that encloses it. The backslash character starts an *escape sequence*, which lets you introduce any character in either kind of string literal. See all of Python's string escape sequences in Table 3-1.

Table 2-1. String escape sequences

Sequence	Meaning	ASCII/ISO code
`\<newline>`	Ignore end of line	None
`\\`	Backslash	0x5c
`\'`	Single quote	0x27
`\"`	Double quote	0x22
`\a`	Bell	0x07
`\b`	Backspace	0x08
`\f`	Form feed	0x0c
`\n`	Newline	0x0a
`\r`	Carriage return	0x0d

`\t`	Tab	0x09
`\v`	Vertical tab	0x0b
\ *DDD*	Octal value *DDD*	As given
`\x` *XX*	Hexadecimal value *XX*	As given
\N{Unicode char name}	Unicode character	As given
\ *other*	Any other character: a two-character string	0x5c + as given

A variant of a string literal is a *raw string literal*. The syntax is the same as for quoted or triple-quoted string literals, except that an `r` or `R` immediately precedes the leading quote. In raw string literals, escape sequences are not interpreted as in Table 3-1, but are literally copied into the string, including backslashes and newline characters. Raw string literal syntax is handy for strings that include many backslashes, especially regular expression patterns (see "Pattern-String Syntax") and Windows absolute filenames (which use backslashes as directory separators). A raw string literal cannot end with an odd number of backslashes: the last one would be taken as escaping the terminating quote.

Raw and Triple-Quoted String Literals are Different Source Code Representations, Not Different Types

Raw and triple-quotes string literals are *not* different types from other strings; they are just alternative syntaxes for literals of the usual two string types, `bytes` and `str`.

In `str` literals, you can use `\u` followed by four hex digits, or `\U` followed by eight hex digits, to denote Unicode characters; you can also include the escape sequences listed in Table 3-1. `str` literals can also include Unicode characters using the escape sequence `\N{`*name*`}`, where *name* is a standard

Unicode name. For example, `\N{Copyright Sign}` indicates a Unicode copyright sign character (©).

Formatted string literals let you inject formatted expressions into your string "literals", which are therefore no longer constant, but rather are subject to evaluation at execution time. The formatting process is described in "String Formatting." From a syntactic point of view, these new literals can be regarded just as another kind of string literal.

Multiple string literals of any kind—quoted, triple-quoted, raw, bytes, formatted—can be adjacent, with optional whitespace in between (as long as you do not mix text and bytes strings). The compiler concatenates such adjacent string literals into a single string object. Writing a long string literal in this way lets you present it readably across multiple physical lines and gives you an opportunity to insert comments about parts of the string. For example:

```
marypop = ('supercalifragilistic' # Open paren->logical line
continues
           'expialidocious')      # Indentation ignored in
continuation
```

The string assigned to `marypop` is a single word of 34 characters.

Bytes

A *bytes* object is a sequence of `int`s from 0 to 255. Bytes objects are usually encountered when reading data from or writing data to a binary source (e.g, a file, a socket, or a network resource).

A `bytes` object can be initialized from a list of `int`s or from a string of characters. A `bytes` literal has the same syntax as a `str` literal, prefixed with `'b'`:

```
b'abc'
bytes([97, 98, 99])         # same as the previous line
rb'\ = solidus'             # a raw bytes literal, containing a
'\'
```

To convert a `bytes` object to a `str`, use the `bytes.decode()` method. To convert a `str` object to a `bytes`, use the `str.encode()` method, as described in detail in Chapter “Strings and Things”.

Bytearray

A *bytearray* is a **mutable** ordered sequence of `int`s from 0 to 255; like `bytes`, you can construct it from a sequence of `int`s or characters. Apart from mutability, it is just like a `bytes` object. As they are mutable, `bytearray` objects support methods and operators that modify elements within the array of byte values.

```
ba = bytearray([97, 98, 99]) # like bytes, can construct from a
sequence of ints
ba[1] = 97                   # unlike bytes, contents can be
modified
print(ba.decode())           # prints 'aac'
```

Chapter “Strings and Things” has additional material on creating and working with `bytearray` objects.

Tuples

A *tuple* is an immutable ordered sequence of items. The items of a tuple are arbitrary objects and may be of different types. You can use mutable objects (such as lists) as tuple items; however, best practice is to avoid tuples with mutable items.

To denote a tuple, use a series of expressions (the *items* of the tuple) separated by commas (`,`); if every item is a literal, the whole construct is a *tuple literal*. You may optionally place a redundant comma after the last item. You may group tuple items within parentheses, but the parentheses are necessary only where the commas would otherwise have another meaning (e.g., in function calls), or to denote empty or nested tuples. A tuple with exactly two items is also known as a *pair*. To create a tuple of one item, add a comma to the end of the expression. To denote an empty tuple, use an empty pair of parentheses. Here are some tuple literals, all with the optional parentheses (the parentheses are not optional in the last case):

```
(100, 200, 300)             # Tuple with three items
(3.14,)                     # Tuple with 1 item needs trailing
comma
()                          # Empty tuple (parentheses NOT
optional)
```

You can also call the built-in type `tuple` to create a tuple. For example:

```
tuple('wow')
```

This builds a tuple equal to that denoted by the tuple literal:

```
('w', 'o', 'w')
```

`tuple()` without arguments creates and returns an empty tuple, like `()`. When *x* is iterable, `tuple(`*x*`)` returns a tuple whose items are the same as those in *x*.

Lists

A *list* is a mutable ordered sequence of items. The items of a list are arbitrary objects and may be of different types. To denote a list, use a series of expressions (the *items* of the list) separated by commas (`,`), within brackets (`[]`); if every item is a literal, the whole construct is a *list literal*. You may optionally place a redundant comma after the last item. To denote an empty list, use an empty pair of brackets. Here are some examples of list literals:

`list()` without arguments creates and returns an empty list, like `[]`. When *x* is iterable, `list(`*x*`)` returns a list whose items are the same as those in *x*.

```
[42, 3.14, 'hello']         # List with three items
[100]                       # List with one item
[]                          # Empty list
```

You can also call the built-in type `list` to create a list. For example:

```
list('wow')
```

This builds a list equal to that denoted by the list literal:

```
['w', 'o', 'w']
```

You can also build lists with list comprehensions, covered in “List comprehensions”.

Sets

Python has two built-in set types, `set` and `frozenset`, to represent arbitrarily ordered collections of unique items. Items in a set may be of different types, but they must be *hashable* (see `hash` in Table 7-2). Instances of type `set` are mutable, and thus, not hashable; instances of type `frozenset` are immutable and hashable. You can’t have a set whose items are sets, but you can have a set (or frozenset) whose items are frozensets. Sets and frozensets are *not* ordered.

To create a set, you can call the built-in type `set` with no argument (this means an empty set) or one argument that is iterable (this means a set whose items are those of the iterable). You can similarly build a frozenset by calling `frozenset`.

Alternatively, to denote a (non-frozen, non-empty) set, use a series of expressions (the items of the set) separated by commas (`,`) within braces (`{}`); if every item is a literal, the whole assembly is a *set literal*. You may optionally place a redundant comma after the last item. Some example sets (two literals, one not):

```
{42, 3.14, 'hello'}      # Literal for a set with three items
{100}                    # Literal for a set with one item
set()                    # Empty set (can't use {}--empty dict!)
```

You can also build non-frozen sets with set comprehensions, as discussed in “Set comprehensions”.

Dictionaries

A *mapping* is an arbitrary collection of objects indexed by nearly[3] arbitrary values called *keys*. Mappings are mutable and, like sets but unlike sequences, are ***not*** (necessarily) ordered.

Python provides a single built-in mapping type: the dictionary type. Library and extension modules provide other mapping types, and you can write others yourself (as discussed in "Mappings"). Keys in a dictionary may be of different types, but they must be *hashable* (see `hash` in Table 7-2). Values in a dictionary are arbitrary objects and may be of any type. An *item* in a dictionary is a key/value pair. You can think of a dictionary as an associative array (known in some other languages as a "map," "hash table," or "hash").

To denote a dictionary, you can use a series of colon-separated pairs of expressions (the pairs are the items of the dictionary) separated by commas (`,`) within braces (`{}`); if every expression is a literal, the whole construct is a *dict literal*. You may optionally place a redundant comma after the last item. Each item in a dictionary is written as *key*`:` *value*, where *key* is an expression giving the item's key and *value* is an expression giving the item's value. If a key's value appears more than once in a dictionary expression, only an arbitrary one of the items with that key is kept in the resulting dictionary object—dictionaries do not allow duplicate keys. To denote an empty dictionary, use an empty pair of braces.

Here are some dictionary literals:

```
{'x':42, 'y':3.14, 'z':7}    # Dictionary with three items, str
keys
{1:2, 3:4}                   # Dictionary with two items, int
keys
{1:'za', 'br':23}            # Dictionary with mixed key types
{}                           # Empty dictionary
```

You can also call the built-in type `dict` to create a dictionary in a way that, while usually less concise, can sometimes be more readable. For example, the dictionaries in the preceding snippet can equivalently be written as:

```
dict(x=42, y=3.14, z=7)      # Dictionary with three items, str
keys
```

```
dict([(1, 2), (3, 4)])       # Dictionary with two items, int
keys
dict([(1,'za'), ('br',23)])  # Dictionary with mixed key types
dict()                       # Empty dictionary
```

`dict()` without arguments creates and returns an empty dictionary, like `{}`. When the argument *x* to `dict` is a mapping, `dict` returns a new dictionary object with the same keys and values as *x*. When *x* is iterable, the items in *x* must be pairs, and `dict(`*x*`)` returns a dictionary whose items (key/value pairs) are the same as the items in *x*. If a key value appears more than once in *x*, only the *last* item from *x* with that key value is kept in the resulting dictionary.

When you call `dict`, in addition to, or instead of, the positional argument *x*, you may pass *named arguments*, each with the syntax *name=value*, where *name* is an identifier to use as an item's key and *value* is an expression giving the item's value. When you call `dict` and pass both a positional argument and one or more named arguments, if a key appears both in the positional argument and as a named argument, Python associates to that key the value given with the named argument (i.e., the named argument "wins").

You can also create a dictionary by calling `dict.fromkeys`. The first argument is an iterable whose items become the keys of the dictionary; the second argument is the value that corresponds to each and every key (all keys initially map to the same value). If you omit the second argument, it defaults to `None`. For example:

```
dict.fromkeys('hello', 2)   # same as {'h':2, 'e':2, 'l':2,
'o':2}
dict.fromkeys([1, 2, 3])    # same as {1:None, 2:None, 3:None}
```

You can also build dicts with dict comprehensions, as discussed in "Dict comprehensions".

None

The built-in `None` denotes a null object. `None` has no methods or other attributes. You can use `None` as a placeholder when you need a reference but you don't care what object you refer to, or when you need to indicate that no object is there. Functions return `None` as their result unless they have specific `return` statements coded to return other values. `None` is hashable and can be used as a dict key.

Ellipsis (...)

The Ellipsis, written as three periods with no intervening spaces `"..."`, is a special object in Python used in numerical applications, or as an alternative to `None` when `None` is a valid entry. For instance, to initialize a dict that may take `None` as a legitimate value, you can initialize it with `...` as an indicator of "no value supplied, not even None". Ellipsis is hashable and can be used as a dict key.

```
# use None for "None of the above", ... for "no entry"
votes_tally = dict.fromkeys(['Candidate A', 'Candidate B', None,
...], 0)
```

Callables

In Python, callable types are those whose instances support the function call operation (see "Calling Functions"). Functions are callable. Python provides several built-in functions (see "Built-in Functions") and supports user-defined functions (see "The def Statement"). Generators are also callable (see "Generators").

Types are also callable, as we already saw for the `dict`, `list`, `set`, and `tuple` built-in types. (See "Built-in Types" for a complete list of built-in types.) As we discuss in "Python Classes", `class` objects (user-defined types) are also callable. Calling a type normally creates and returns a new instance of that type.

Other callables include *methods*, which are functions bound as class attributes, and instances of classes that supply a special method named `__call__`.

Boolean Values

Any data value in Python can be used as a truth value: true or false. Any nonzero number or nonempty container (e.g., string, tuple, list, set, or dictionary) is true. 0 (of any numeric type), `None`, and empty containers are false.

Beware Using a float as a Truth Value

Be careful about using a floating-point number as a truth value: that's like comparing the number for exact equality with zero, and floating-point numbers should almost never be compared for exact equality.

The built-in type `bool` is a subclass of `int`. The only two values of type `bool` are `True` and `False`, which have string representations of `'True'` and `'False'`, but also numerical values of `1` and `0`, respectively. Several built-in functions return `bool` results, as do comparison operators.

You can call `bool(`*x*`)` with any *x* as the argument. The result is `True` when *x* is true and `False` when *x* is false. Good Python style is not to use such calls when they are redundant, as they most often are: *always* write `if` *x*`:`, *never* any of `if bool(`*x*`):`, `if` *x* `is True:`, `if` *x*`==True:`, `if bool(`*x*`)==True:`. However, you *can* use `bool(`*x*`)` to count the number of true items in a sequence. For example:

```
def count_trues(seq):
    return sum(bool(x) for x in seq)
```

In this example, the `bool` call ensures each item of *seq* is counted as `0` (if false) or `1` (if true), so `count_trues` is more general than `sum(`*seq*`)` would be.

When we say "`expression` is true" we mean that `bool(expression)` would return `True`. This is also known as

"`expression` being *truthy"* (the other possibility is that "`expression` is *falsy"*).

Variables and Other References

A Python program accesses data values through *references*. A *reference* is a "name" that refers to a value (object). References take the form of variables, attributes, and items. In Python, a variable or other reference has no intrinsic type. The object to which a reference is bound at a given time always has a type, but a given reference may be bound to objects of various types in the course of the program's execution.

Variables

In Python, there are no "declarations." The existence of a variable begins with a statement that *binds* the variable (in other words, sets a name to hold a reference to some object). You can also *unbind* a variable, resetting the name so it no longer holds a reference. Assignment statements are the usual way to bind variables and other references. The (rarely used) `del` statement unbinds references.

Binding a reference that was already bound is also known as *rebinding* it. Whenever we mention binding, we implicitly include rebinding (except where we explicitly exclude it). Rebinding or unbinding a reference has no effect on the object to which the reference was bound, except that an object goes away when nothing refers to it. The cleanup of objects with no references is known as *garbage collection*.

You can name a variable with any identifier except the 30-plus reserved as Python's keywords (see "Keywords"). A variable can be global or local. A *global variable* is an attribute of a module object (see Chapter "Modules"). A *local variable* lives in a function's local namespace (see "Namespaces").

Object attributes and items

The main distinction between the attributes and items of an object is in the syntax you use to access them. To denote an *attribute* of an object, use a reference to the object, followed by a period (`.`), followed by an identifier known as the *attribute name*. For example, *x.y* refers to one of the attributes of the object bound to name *x*, specifically that attribute whose name is *'y'*.

To denote an *item* of an object, use a reference to the object, followed by an expression within brackets (`[]`). The expression in brackets is known as the item's *index* or *key*, and the object is known as the item's *container*. For example, *x*`[`*y*`]` refers to the item at the key or index bound to name *y*, within the container object bound to name *x*.

Attributes that are callable are also known as *methods*. Python draws no strong distinctions between callable and noncallable attributes, as some other languages do. All general rules about attributes also apply to callable attributes (methods).

Accessing nonexistent references

A common programming error is to access a reference that does not exist. For example, a variable may be unbound, or an attribute name or item index may not be valid for the object to which you apply it. The Python compiler, when it analyzes and compiles source code, diagnoses only syntax errors. Compilation does not diagnose semantic errors, such as trying to access an unbound attribute, item, or variable. Python diagnoses semantic errors only when the errant code executes—that is, *at runtime*. When an operation is a Python semantic error, attempting it raises an exception (see Chapter "Exceptions"). Accessing a nonexistent variable, attribute, or item—just like any other semantic error—raises an exception.

Assignment Statements

Assignment statements can be plain or augmented. Plain assignment to a variable (e.g., *name=value*) is how you create a new variable or rebind an existing variable to a new value. Plain assignment to an object attribute (e.g., *x.attr=value*) is a request to object *x* to create or rebind the attribute named *'attr'*. Plain assignment to an item in a container (e.g., *x*`[`*k*`]` =

value) is a request to container *x* to create or rebind the item with index or key *k*.

Augmented assignment (e.g., *name*+=*value*) cannot, per se, create new references. Augmented assignment can rebind a variable, ask an object to rebind one of its existing attributes or items, or request the target object to modify itself. When you make any kind of request to an object, it is up to the object to decide whether and how to honor the request, and whether to raise an exception.

Plain assignment

A plain assignment statement in the simplest form has the syntax:

```
target = expression
```

The target is known as the left-hand side (LHS), and the expression is the right-hand side (RHS). When the assignment executes, Python evaluates the RHS expression, then binds the expression's value to the LHS target. The binding never depends on the type of the value. In particular, Python draws no strong distinction between callable and noncallable objects, as some other languages do, so you can bind functions, methods, types, and other callables to variables, just as you can numbers, strings, lists, and so on. This is part of functions and other callables being *first-class objects*.

Details of the binding do depend on the kind of target. The target in an assignment may be an identifier, an attribute reference, an indexing, or a slicing:

An identifier

> Is a variable name. Assigning to an identifier binds the variable with this name.

An attribute reference

> Has the syntax *obj.name*, where *obj* is an arbitrary expression, and *name* is an identifier, known as an *attribute name* of the object. Assigning to an attribute reference asks object *obj* to bind its attribute named *'name'*.

An indexing

> Has the syntax *obj*`[`*expr*`]`. *obj* and *expr* are arbitrary expressions. Assigning to an indexing asks container *obj* to bind its item indicated by the value of *expr*, also known as the index or key of the item in the container.

A slicing

> Has the syntax *obj*`[`*start*`:`*stop*`]` or *obj*`[`*start*`:`*stop*`:`*stride*`]`. *obj*, *start*, *stop*, and *stride* are arbitrary expressions. *start*, *stop*, and *stride* are all optional (i.e., *obj*`[:`*stop*`:]` and *obj*`[:`*stop*`]` are also syntactically correct slicings, each being equivalent to *obj*`[None:`*stop*`:None]`). Assigning to a slicing asks container *obj* to bind or unbind some of its items. Assigning to a slicing such as *obj*`[`*start*`:`*stop*`:`*stride*`]` is equivalent to assigning to the indexing *obj*`[slice(`*start*`, `*stop*`, `*stride*`)]`. See Python's built-in type `slice` in (Table 7-1), whose instances represent slices.

We'll get back to indexing and slicing targets when we discuss operations on lists, in "Modifying a list", and on dictionaries, in "Indexing a Dictionary".

When the target of the assignment is an identifier, the assignment statement specifies the binding of a variable. This is *never* disallowed: when you request it, it takes place. In all other cases, the assignment statement denotes a request to an object to bind one or more of its attributes or items. An object may refuse to create or rebind some (or all) attributes or items, raising an exception if you attempt a disallowed creation or rebinding (see also `__setattr__` in Table 4-1 and `__setitem__` in "Container methods").

A plain assignment can use multiple targets and equals signs (=). For example:

```
a = b = c = 0
```

binds variables `a`, `b`, and `c` to the same value, `0`. Each time the statement executes, the RHS expression evaluates just once, no matter how many targets are in the statement. Each target, left to right, is bound to the one object returned by the expression, just as if several simple assignments executed one after the other.

The target in a plain assignment can list two or more references separated by commas, optionally enclosed in parentheses or brackets. For example:

```
a, b, c = x
```

This statement requires `x` to be an iterable with exactly three items, and binds `a` to the first item, `b` to the second, and `c` to the third. This kind of assignment is known as an *unpacking assignment*. The RHS expression must be an iterable with exactly as many items as there are references in the target; otherwise, Python raises an exception. Each reference in the target gets bound to the corresponding item in the RHS. An unpacking assignment can be used, for example, to swap references:

```
a, b = b, a
```

This assignment statement rebinds name *a* to what name *b* was bound to, and vice versa. Exactly one of the multiple targets of an unpacking assignment may be preceded by `*`. That *starred* target, if present, is bound to a list of all items, if any, that were not assigned to other targets. For example:

```
first, *middle, last = x
```

when *x* is a list, is the same as (but more concise, clearer, more general, and faster than):

```
first, middle, last = x[0], x[1:-1], x[-1]
```

Each of these forms requires *x* to have at least two items. This feature is known as *extended unpacking*.

Augmented assignment

An *augmented assignment* (sometimes also known as an *in-place assignment*) differs from a plain assignment in that, instead of an equals sign (=) between the target and the expression, it uses an *augmented operator*, which is a binary operator followed by =. The augmented operators are `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `|=`, `>>=`, `<<=`, `&=`, `^=`, and `@=`. An augmented assignment can have only one target on the LHS; augmented assignment does not support multiple targets.

In an augmented assignment, like in a plain one, Python first evaluates the RHS expression. Then, when the LHS refers to an object that has a special method for the appropriate *in-place* version of the operator, Python calls the method with the RHS value as its argument (it is up to the method to modify the LHS object appropriately and return the modified object; "Special Methods" covers special methods). When the LHS object has no applicable in-place special method, Python uses the corresponding binary operator on the LHS and RHS objects, then rebinds the target to the result. E.g.: *x*+=*y* is like *x*=*x*`.__iadd__(`*y*`)` when *x* has special method `__iadd__`, "in-place addition". Otherwise, *x*+=*y* is like *x*=*x*+*y*.

Augmented assignment never creates its target reference; the target must already be bound when augmented assignment executes. Augmented assignment can rebind the target reference to a new object, or modify the same object to which the target reference was already bound. Plain assignment, in contrast, can create or rebind the LHS target reference, but it never modifies the object, if any, to which the target reference was previously bound. The distinction between objects and references to objects is crucial here. For example, *x*=*x*+*y* never modifies the object to which name *x* was originally bound, if any. Rather, it rebinds name *x* to refer to a new object. *x*+=*y*, in contrast, modifies the object to which the name *x* is bound, when that object has special method `__iadd__`; otherwise, *x*+=*y* rebinds the name *x* to a new object, just like *x*=*x*+*y*.

del Statements

Despite its name, a `del` statement *unbinds references*—it does *not*, per se, *delete* objects. Object deletion may automatically follow, by garbage collection, when no more references to an object exist.

A `del` statement consists of the keyword `del`, followed by one or more target references separated by commas (`,`). Each target can be a variable, attribute reference, indexing, or slicing, just like for assignment statements, and must be bound at the time `del` executes. When a `del` target is an identifier, the `del` statement means to unbind the variable. If the identifier was bound, unbinding it is never disallowed; when requested, it takes place.

In all other cases, the `del` statement specifies a request to an object to unbind one or more of its attributes or items. An object may refuse to unbind some (or all) attributes or items, raising an exception if you attempt a disallowed unbinding (see also `__delattr__` in "General-Purpose Special Methods" and `__delitem__` in "Container methods"). Unbinding a slicing normally has the same effect as assigning an empty sequence to that slicing, but it is up to the container object to implement this equivalence.

Containers are also allowed to have `del` cause side effects. For example, assuming `del` *C*`[2]` succeeds, when *C* is a dict, this makes future references to *C*`[2]` invalid (raising `KeyError`) until and unless you assign to *C*`[2]` again; but when *C* is a list, `del` *C*`[2]` implies that every following item of *C* "shifts left by one"—so, if *C* is long enough, future references to *C*`[2]` are still valid, but denote a different item than they did before the `del` (generally, what you'd have used *C*`[3]` to refer to, before the `del` statement).

Expressions and Operators

An expression is a "phrase" of code, which Python evaluates to produce a value. The simplest expressions are literals and identifiers. You build other expressions by joining subexpressions with the operators and/or delimiters

listed in Table 3-2. This table lists operators in decreasing order of precedence, higher precedence before lower. Operators listed together have the same precedence. The third column lists the associativity of the operator: L (left-to-right), R (right-to-left), or NA (non-associative).

Table 2-2.
Operator precedence in expressions

Operator	Description	Associativity
`{ key : expr ,...}`	Dictionary creation	NA
`{ expr ,...}`	Set creation	NA
`[ expr ,...]`	List creation	NA
`( expr ,...)`	Tuple creation (parentheses recommended, but not always required; 1+ commas required), or just parentheses	NA
`f ( expr ,...)`	Function call	L
`x [ index : index ]`	Slicing	L
`x [ index ]`	Indexing	L
`x . attr`	Attribute reference	L
	Exponentiation (x to the yth power)	R

`x ** y`		
`~ x`	Bitwise NOT	NA
`+x, -x`	Unary plus and minus	NA
`x*y, x/y, x//y, x%y`	Multiplication, division, floor division, remainder	L
`x+y, x-y`	Addition, subtraction	L
`x<<y, x>>y`	Left-shift, right-shift	L
`x & y`	Bitwise AND	L
`x ^ y`	Bitwise XOR	L
`x | y`	Bitwise OR	L
`x<y, x<=y, x>y, x>=y,x!=y, x==y`	Comparisons (less than, less than or equal, greater than, greater than or equal, inequality, equality)	NA
`x is y, x is not y`	Identity tests	NA
`x in y, x not in y`	Membership tests	NA
`not x`	Boolean NOT	NA

`x and y`	Boolean AND	L
`x or y`	Boolean OR	L
`x if expr else y`	Ternary operator	NA
`lambda arg,...: expr`	Anonymous simple function	NA
`(id := expr)`	Assignment expression (parentheses recommended, but not always required)	NA

In Table 3-2, `expr`, `key`, `f`, `index`, `x`, and `y` mean any expression, while `attr`, `arg` and `id` mean any identifier. The notation `,...` means commas join zero or more repetitions; in such cases, a trailing comma is optional and innocuous.

Comparison Chaining

You can *chain* comparisons, implying a logical `and`. For example:

```
a < b <= c < d
```

where `a`, `b`, `c`, and `d` are arbitrary expressions, has (in the absence of evaluation side-effects) the same value as:

```
a < b and b <= c and c < d
```

The chained form is more readable, and evaluates each subexpression at most once.

Short-Circuiting Operators

The `and` and `or` operators *short-circuit* their operands' evaluation: the right hand operand evaluates only when its value is needed to get the truth value of the entire `and` or `or` operation.

In other words, `x and y` first evaluates `x`. When `x` is false, the result is `x`; otherwise, the result is `y`. Similarly, `x or y` first evaluates `x`. When `x` is true, the result is `x`; otherwise, the result is `y`.

`and` and `or` don't force their results to be `True` or `False`, but rather return one or the other of their operands. This lets you use these operators more generally, not just in Boolean contexts. `and` and `or`, because of their short-circuiting semantics, differ from other operators, which fully evaluate all operands before performing the operation. `and` and `or` let the left operand act as a *guard* for the right operand.

The ternary operator

Another short-circuiting operator is the ternary operator `if`/`else`:

```
whentrue if condition else whenfalse
```

Each of `whentrue`, `whenfalse`, and `condition` is an arbitrary expression. `condition` evaluates first. When `condition` is true, the result is `whentrue`; otherwise, the result is `whenfalse`. Only one of the subexpressions `whentrue` and `whenfalse` evaluates, depending on the truth value of `condition`.

The order of the subexpressions in this ternary operator may be a bit confusing. The recommended style is to always place parentheses around the whole expression.

Numeric Operations

Python offers the usual numeric operations, as we've just seen in Table 3-2. Numbers are immutable objects: when you perform operations on number

objects, you produce new number objects, never modify existing ones. You can access the parts of a complex object `z` as read-only attributes `z.real` and `z.imag`. Trying to rebind these attributes raises an exception.

A number's optional + or − sign, and the + or − that joins a floating-point literal to an imaginary one to make a complex number, are not part of the literals' syntax. They are ordinary operators, subject to normal operator precedence rules (see Table 3-2). For example, `-2**2` evaluates to `-4`: exponentiation has higher precedence than unary minus, so the whole expression parses as `-(2**2)`, not as `(-2)**2`. (Again, parentheses are recommended, to avoid confusing a reader of the code).

Numeric Conversions

You can perform arithmetic operations and comparisons between any two numbers of Python built-in types. If the operands' types differ, Python converts the operand with the "smaller" type to the "larger" type[4]. Builtin number types, in order from smallest to largest, are integers, floating-point numbers, and complex numbers. You can request an explicit conversion by passing a non-complex numeric argument to any of the built-in number types: `int`, `float`, and `complex`. `int` drops its argument's fractional part, if any (e.g., `int(9.8)` is `9`). You can also call `complex` with two numeric arguments, giving real and imaginary parts. You cannot convert a `complex` to another numeric type in this way, because there is no single unambiguous way to convert a complex number into, for example, a `float`.

You can also call each built-in numeric type with a string argument with the syntax of an appropriate numeric literal, with small extensions: the argument string may have leading and/or trailing whitespace, may start with a sign, and—for complex numbers—may sum or subtract a real part and an imaginary one. `int` can also be called with two arguments: the first one a string to convert, and the second the *radix*, an integer between 2 and 36 to use as the base for the conversion (e.g., `int('101', 2)` returns `5`, the value of `'101'` in base `2`). For radices larger than 10, the appropriate

subset of ASCII letters from the start of the alphabet (in either lower- or uppercase) are the extra needed "digits."

Arithmetic Operations

Python arithmetic operations behave in rather obvious ways, with the possible exception of division and exponentiation.

Division

When the right operand of `/`, `//`, or `%` is `0`, Python raises an exception at runtime. Otherwise, the `//` operator performs *floor* division, which means it returns an integer result (converted to the same type as the wider operand) and ignores the remainder, if any. The `/` operator performs *true* division, returning a floating-point result (or a complex result if either operand is a complex number). The `%` operator returns the *remainder* of the (floor) division.

The built-in `divmod` function takes two numeric arguments and returns a pair whose items are the quotient and remainder, so you don't have to use both `//` for the quotient and `%` for the remainder[5].

Exponentiation

The exponentiation ("raise to power") operation, when `a` is less than zero and `b` is a floating-point value with a nonzero fractional part, returns a complex number. The built-in `pow(a, b)` function returns the same result as `a**b`. With three arguments, `pow(a, b, c)` returns the same result as `(a**b)%c` but is faster. Note that, unlike other arithmetic operations, exponentiation evaluates right-to-left: in other words, `a**b**c` evaluates as `a**(b**c)`.

Comparisons

All objects, including numbers, can be compared for equality (==) and inequality (!=). Comparisons requiring order (<, <=, >, >=) may be used between any two numbers, unless either operand is complex, in which case

they raise exceptions at runtime. All these operators return Boolean values (`True` or `False`). Beware comparing floating-point numbers for equality, as the online tutorial explains.

Bitwise Operations on Integers

Integers can be interpreted as strings of bits and used with the bitwise operations shown in [Link to Come]. Bitwise operators have lower priority than arithmetic operators. Positive integers are conceptually extended by an unbounded string of `0` bits on the left. Negative integers, since they're held in two's complement representation, are conceptually extended by an unbounded string of `1` bits on the left.

Sequence Operations

Python supports a variety of operations applicable to all sequences, including strings, lists, and tuples. Some sequence operations apply to all containers (including sets and dictionaries, which are not sequences); some apply to all iterables (meaning "any object over which you can loop," as covered in [Link to Come]; all containers, be they sequences or not, are iterable, and so are many objects that are not containers, such as files, covered in [Link to Come], and generators, covered in "Generators"). In the following we use the terms *sequence*, *container*, and *iterable* quite precisely, to indicate exactly which operations apply to each category.

Sequences in General

Sequences are ordered containers with items that are accessible by indexing and slicing .

The built-in `len` function takes any container as an argument and returns the number of items in the container.

The built-in `min` and `max` functions take one argument, an iterable whose items are comparable, and return the smallest and largest items, respectively. You can also call `min` and `max` with multiple arguments, in which case they return the smallest and largest arguments, respectively.

`min` and `max` also accept two keyword-only optional arguments: *key*, a callable to apply to each item (the comparisons are then performed on the callable's results rather than on the items themselves); and *default*, the value to return when the iterable is empty (when the iterable is empty and you supply no *default* argument, the function raises `ValueError`). For example, `max('who', 'why', 'what', key=len)` returns 'what'.

The built-in `sum` function takes one argument, an iterable whose items are numbers, and returns the sum of the numbers.

Sequence conversions

There is no implicit conversion between different sequence types. You can call the built-ins `tuple` and `list` with a single argument (any iterable) to get a new instance of the type you're calling, with the same items, in the same order, as in the argument.

Concatenation and repetition

You can concatenate sequences of the same type with the `+` operator. You can multiply a sequence `S` by an integer `n` with the `*` operator. `S*n` is the concatenation of `n` copies of `S`. When `n<=0`, `S*n` is an empty sequence of the same type as `S`.

Membership testing

The `x in S` operator tests to check whether object `x` equals any item in the sequence (or other kind of container or iterable) `S`. It returns `True` when it does and `False` when it doesn't. The `x not in S` operator is equivalent to `not (x in S)`. For dictionaries, `x in S` tests for the presence of `x` as a key. In the specific case of strings, `x in S` is more widely applicable; in this case, `x in S` tests whether `x` equals any *substring* of `S`, not just any single *character*.

Indexing a sequence

To denote the `n`th item of a sequence `S`, use *indexing*: `S[n]`. Indexing is zero-based (`S`'s first item is `S[0]`). If `S` has `L` items, the index `n` may be `0`,

1…up to and including L-1, but no larger. n may also be -1, -2…down to and including -L, but no smaller. A negative n (e.g., -1) denotes the same item in S as L+n (e.g., L + -1) does. In other words, S[-1], like S[L-1], is the last element of S, S[-2] is the next-to-last one, and so on. For example:

```
x = [1, 2, 3, 4]
x[1]                    # 2
x[-1]                   # 4
```

Using an index >=L or <-L raises an exception. Assigning to an item with an invalid index also raises an exception. You can add elements to a list, but to do so you assign to a slice, not to an item, as we'll discuss shortly.

Slicing a sequence

To indicate a subsequence of S, you can use *slicing*, with the syntax S[i:j], where i and j are integers. S[i:j] is the subsequence of S from the ith item, included, to the jth item, excluded. In Python, ranges always include the lower bound and exclude the upper bound. A slice is an empty subsequence when j is less than or equal to i, or when i is greater than or equal to L, the length of S. You can omit i when it is equal to 0, so that the slice begins from the start of S. You can omit j when it is greater than or equal to L, so that the slice extends all the way to the end of S. You can even omit both indices, to mean a shallow copy of the entire sequence: S[:]. Either or both indices may be less than 0. Here are some examples:

```
x = [1, 2, 3, 4]
x[1:3]                  # [2, 3]
x[1:]                   # [2, 3, 4]
x[:2]                   # [1, 2]
```

A negative index n in slicing indicates the same spot in S as L+n, just like it does in indexing. An index greater than or equal to L means the end of S, while a negative index less than or equal to -L means the start of S.

Slicing can use the *extended* syntax `S[i:j:k]`. `k` is the *stride* of the slice, meaning the distance between successive indices. `S[i:j]` is equivalent to `S[i:j:1]`, `S[::2]` is the subsequence of `S` that includes all items that have an even index in `S`, and `S[::-1]` is a slicing, also whimsically known as "the Martian smiley," with the same items as `S` but in reverse order. With a negative stride, in order to have a nonempty slice, the second ("stop") index needs to be *smaller* than the first ("start") one—the reverse of the condition that must hold when the stride is positive. A stride of `0` raises an exception.

```
y = list(range(10))
y[-5:]           #  last five items
[5, 6, 7, 8, 9]
y[::2]           #  every other item
[0, 2, 4, 6, 8]
y[10:0:-2]       #  every other item, in reverse order
[9, 7, 5, 3, 1]
y[:0:-2]         #  every other item, in reverse order (simpler)
[9, 7, 5, 3, 1]
y[::-2]          #  every other item, in reverse order (best)
[9, 7, 5, 3, 1]
```

Strings

String objects (byte strings, as well as text, AKA Unicode, ones) are immutable: attempting to rebind or delete an item or slice of a string raises an exception. (For bytes, though not for text, there's a mutable, but otherwise equivalent, built-in type, bytearray). The items of a text string (each of the characters in the string) are themselves text strings, each of length `1`—Python has no special data type for "single characters" (the items of a bytes or bytearray object are ints). All slices of a string are strings of the same kind. String objects have many methods, covered in [Link to Come].

Tuples

Tuple objects are immutable: therefore, attempting to rebind or delete an item or slice of a tuple raises an exception. The items of a tuple are arbitrary

objects and may be of different types; tuple items may be mutable, but we recommend not mutating them, as doing so can be confusing. The slices of a tuple are also tuples. Tuples have no normal (nonspecial) methods, except count and index, with the same meanings as for lists; they do have many of the special methods covered in [Link to Come].

Lists

List objects are mutable: you may rebind or delete items and slices of a list. Items of a list are arbitrary objects and may be of different types. Slices of a list are lists.

Modifying a list

You can modify a single item in a list by assigning to an indexing. For instance:

```
x = [1, 2, 3, 4]
x[1] = 42                # x is now [1, 42, 3, 4]
```

Another way to modify a list object `L` is to use a slice of `L` as the target (LHS) of an assignment statement. The RHS of the assignment must be an iterable. When the LHS slice is in extended form (i.e., the slicing specifies a stride `!=1`), then the RHS must have just as many items as the number of items in the LHS slice. When the LHS slicing does not specify a stride, or explicitly specifies a stride of `1`, the LHS slice and the RHS may each be of any length; assigning to such a slice of a list can make the list longer or shorter. For example:

```
x = [1, 2, 3, 4]
x[1:3] = [22, 33, 44]     # x is now [1, 22, 33, 44, 4]
x[1:4] = [8, 9]           # x is now [1, 8, 9, 4]
```

There are some important special cases of assignment to slices:

- Using the empty list `[]` as the RHS expression removes the target slice from `L`. In other words, L[i:j]=[] has the same effect as del L[i:j] (or the peculiar statement L[i:j]*=0).

Using an empty slice of `L` as the LHS target inserts the items of the RHS at the appropriate spot in `L`. For example, L[i:i]=['a','b'] inserts 'a' and `'b'` before the item that was at index `i` in `L` prior to the assignment.

- Using a slice that covers the entire list object, L[:], as the LHS target, totally replaces the contents of `L`.

You can delete an item or a slice from a list with del. For instance:

```
x = [1, 2, 3, 4, 5]
del x[1]                # x is now [1, 3, 4, 5]
del x[::2]              # x is now [3, 5]
```

In-place operations on a list

List objects define in-place versions of the `+` and `*` operators, which you can use via augmented assignment statements. The augmented assignment statement L+=L1 has the effect of adding the items of iterable `L1` to the end of `L`, just like L.extend(L1). L*=n has the effect of adding `n-1` copies of `L` to the end of `L`; if n<=0, L*=n makes `L` empty, like L[:]=[] or del L[:].

List methods

List objects provide several methods, as shown in Table 3-3. *Nonmutating methods* return a result without altering the object to which they apply, while *mutating methods* may alter the object to which they apply. Many of a list's mutating methods behave like assignments to appropriate slices of the list. In Table 3-3, `L` indicates any list object, `i` any valid index in `L`, `s` any iterable, and `x` any object.

Table 3-3. List object methods

Method	Description
Nonmutating	

`L.count(x)`	Returns the number of items of `L` that are equal to `x`.
`L.index(x)`	Returns the index of the first occurrence of an item in `L` that is equal to `x`, or raises an exception if `L` has no such item.

Mutating

`L.append(x)`	Appends item `x` to the end of `L`; like `L[len(L):]=[x]`.
`L.extend(s)`	Appends all the items of iterable `s` to the end of `L`; like `L[len(L):]=s` or `L += s`.
`L.insert(i, x)`	Inserts item `x` in `L` before the item at index `i`, moving following items of `L` (if any) "rightward" to make space (increases `len(L)` by one, does not replace any item, does not raise exceptions; acts just like `L[i:i]=[x]`).
`L.remove(x)`	Removes from `L` the first occurrence of an item in `L` that is equal to `x`, or raises an exception if `L` has no such item.
`L.pop(i=-1)`	Returns the value of the item at index `i` and removes it from `L`; when you omit `i`, removes and returns the last item; raises an exception if `L` is empty or `i` is an invalid index in `L`.
`L.reverse()`	Reverses, in place, the items of `L`.
`L.sort(key=None, reverse=False)`	Sorts, in-place, the items of `L` (in ascending order, by default; in descending order, if argument `reverse` is true) When argument `key` is not `None`, what gets compared for each item `x` is `key(x)`, not `x` itself. For more details, see "Sorting a list".

All mutating methods of list objects, except `pop`, return `None`.

Sorting a list

A list's method `sort` causes the list to be sorted in-place (reordering items to place them in increasing order) in a way that is guaranteed to be *stable* (elements that compare equal are not exchanged). In practice, `sort` is extremely fast, often *preternaturally* fast, as it can exploit any order or reverse order that may be present in any sublist (the advanced algorithm `sort` uses, known as *timsort* to honor its inventor, great Pythonista Tim Peters, is a "non-recursive adaptive stable natural mergesort/binary insertion sort hybrid"—now *there's* a mouthful for you!).

The `sort` method takes two optional arguments, which may be passed with either positional or named-argument syntax. The argument `key`, if not `None`, must be a function that can be called with any list item as its only argument. In this case, to compare any two items `x` and `y`, Python compares `key(x)` and `key(y)` rather than `x` and `y` (internally, Python implements this in the same way as the decorate-sort-undecorate idiom presented in [Link to Come], but much faster). The argument `reverse`, if true, causes the result of each comparison to be reversed; this is not exactly the same thing as reversing `L` after sorting, because the sort is stable (elements that compare equal are never exchanged) whether the argument `reverse` is true or false. In other words, Python sorts the list in ascending order by default, in descending order if `reverse` is true.

```
            mylist = ['alpha', 'Beta', 'GAMMA']
            mylist.sort() # ['Beta', 'GAMMA', 'alpha']
            mylist.sort(key=str.lower) # ['alpha', 'Beta',
'GAMMA']
```

Python also provides the built-in function `sorted` (covered in [Link to Come]) to produce a sorted list from any input iterable. `sorted`, after the first argument (which is the iterable supplying the items), accepts the same two optional arguments as a list's method `sort`.

The standard library module `operator` (covered in [Link to Come]) supplies higher-order functions `attrgetter`, `itemgetter`, and `methodcaller`, which produce functions particularly suitable for the

`key=` optional argument of lists' method `sort` and the built-in function `sorted`. The `key=` optional argument also exists, with exactly the same meaning, for built-in functions `min` and `max`, as well as for functions `nsmallest`, `nlargest`, and `merge` in standard library module `heapq`, covered in [Link to Come], and class `groupby` in standard library module `itertools`, covered in [Link to Come].

Set Operations

Python provides a variety of operations applicable to sets (both plain and frozen). Since sets are containers, the built-in `len` function can take a set as its single argument and return the number of items in the set. A set is iterable, so you can pass it to any function or method that takes an iterable argument. In this case, iteration yields the items of the set in some arbitrary order. For example, for any set `S`, `min(S)` returns the smallest item in `S`, since `min` with a single argument iterates on that argument (the order does not matter, because the implied comparisons are transitive).

Set Membership

The `k in S` operator checks whether object `k` equals one of the items of set `S`. It returns `True` when it does, `False` when it doesn't. `k not in S` is like `not (k in S)`.

Set Methods

Set objects provide several methods, as shown in Table 3-4. Nonmutating methods return a result without altering the object to which they apply, and can also be called on instances of `frozenset`; mutating methods may alter the object to which they apply, and can be called only on instances of `set`. In Table 3-4, `S` denotes any set object, `S1` any iterable with hashable items (often but not necessarily a `set` or `frozenset`), `x` any hashable object.

Table 3-4. Set object methods

Method	Description
Nonmutating	
`S .copy()`	Returns a shallow copy of `S` (a copy whose items are the same objects as `S`'s, not copies thereof), like `set(S)`
`S .difference( S1 )`	Returns the set of all items of `S` that aren't in `S1`
`S .intersection( S1 )`	Returns the set of all items of `S` that are also in `S1`
`S .issubset( S1 )`	Returns `True` when all items of `S` are also in `S1`; otherwise, returns `False`
`S .issuperset( S1 )`	Returns `True` when all items of `S1` are also in `S`; otherwise, returns `False` (like `S1.issubset(S)`)
`S .symmetric_difference( S1 )`	Returns the set of all items that are in either `S` or `S1`, but not both
`S .union( S1 )`	Returns the set of all items that are in `S`, `S1`, or both
Mutating	
`S .add( x )`	Adds `x` as an item to `S`; no effect if `x` was already an item in `S`
`S .clear()`	Removes all items from `S`, leaving `S` empty
`S .discard( x )`	Removes `x` as an item of `S`; no effect when `x` was not an item of `S`
`S .pop()`	Removes and returns an arbitrary item of `S`
`S .remove( x )`	Removes `x` as an item of `S`; raises a `KeyError` exception when `x` was not an item of `S`

All mutating methods of set objects, except `pop`, return `None`.

The `pop` method can be used for destructive iteration on a set, consuming little extra memory. The memory savings make `pop` usable for a loop on a huge set, when what you want is to "consume" the set in the course of the loop. Besides saving memory, a potential advantage of a destructive loop such as

```
while S:
                item = S.pop()
                ...handle item...
```

in comparison to a nondestructive loop such as

```
for item in S:
                ...handle item...
```

is that, in the body of the destructive loop, you're allowed to modify `S` (adding and/or removing items), which is not allowed in the nondestructive loop.

Sets also have mutating methods named `difference_update`, `intersection_update`, `symmetric_difference_update`, and `update` (corresponding to non-mutating method `union`). Each such mutating method performs the same operation as the corresponding nonmutating method, but it performs the operation in place, altering the set on which you call it, and returns `None`.

The four corresponding non-mutating methods are also accessible with operator syntax: where `S2` is a set or frozenset, respectively, `S-S2`, `S&S2`, `S^S2`, and `S|S2`; the mutating methods are accessible with augmented assignment syntax: respectively, `S-=S2`, `S&=S2`, `S^=S2`, and `S|=S2`. In addition, sets (and frozensets) also support comparison operators: == (the sets have the same items, AKA, they're "equal" sets), != (the reverse of ==), >= (`issuperset`), <= (`issubset`), < (`issubset` and not equal), > (`issuperset` and not equal).

When you use operator or augmented assignment syntax, both operands must be sets or frozensets; however, when you call named methods, argument `S1` can be any iterable with hashable items, and it works just as if the argument you passed was `set(S1)`.

Dictionary Operations

Python provides a variety of operations applicable to dictionaries. Since dictionaries are containers, the built-in `len` function can take a dictionary as its argument and return the number of items (key/value pairs) in the dictionary. A dictionary is iterable, so you can pass it to any function that takes an iterable argument. In this case, iteration yields only the keys of the dictionary, in insertion order. For example, for any dictionary `D`, `min(D)` returns the smallest key in `D`.

Dictionary Membership

The `k in D` operator checks whether object `k` is a key in dictionary `D`. It returns `True` when it is, `False` otherwise. `k not in D` is like `not (k in D)`.

Indexing a Dictionary

To denote the value in a dictionary `D` currently associated with key `k`, use an indexing: `D[k]`. Indexing with a key that is not present in the dictionary raises an exception. For example:

```
d = {'x':42, 'y':3.14, 'z':7}
          d['x'] # 42
          ['z'] # 7

          d['a'] # raises KeyError exception
```

Plain assignment to a dictionary indexed with a key that is not yet in the dictionary (e.g., `D[newkey]=value`) is a valid operation and adds the key and value as a new item in the dictionary. For instance:

```
d = {'x':42, 'y':3.14}
          d['a'] = 16 # d is now {'x':42, 'y':3.14, 'a':16}
```

The `del` statement, in the form `del D[k]`, removes from the dictionary the item whose key is `k`. When `k` is not a key in dictionary `D`, `del D[k]` raises a `KeyError` exception.

Dictionary Methods

Dictionary objects provide several methods, as shown in Table 3-5. Nonmutating methods return a result without altering the object to which they apply, while mutating methods may alter the object to which they apply. In Table 3-5, `D` and `D1` indicate any dictionary objects, `k` any hashable object, and `x` any object.

Table 2-3.
Dictionary object methods

Method	Description
Nonmutating	
`D.copy()`	Returns a shallow copy of the dictionary (a copy whose items are the same objects as `D`'s, not copies thereof), like `dict(D)`
`D.get(k[, x])`	Returns `D[k]` when `k` is a key in `D`; otherwise, returns `x` (or `None`, when `x` is not given)
`D.items()`	Returns an iterable "view" object whose items are all current items (key/value pairs) in `D`.

`D.keys()`	Returns an iterable "view" object whose items are all current keys in `D`
`D.values()`	Returns an iterable "view" object whose items are all current values in `D`
Mutating	
`D.clear()`	Removes all items from `D`, leaving `D` empty
`D.pop(k[, x])`	Removes and returns `D[k]` when `k` is a key in `D`; otherwise, returns `x` (or raises a `KeyError` exception when `x` is not given)
`D.popitem()`	Removes and returns an arbitrary item (key/value pair)
`D.setdefault(k[, x])`	Returns `D[k]` when `k` is a key in `D`; otherwise, sets `D[k]` equal to `x` (or `None`, when `x` is not given) and returns `x`
`D.update(D1)`	For each `k` in mapping `D1`, sets `D[k]` equal to `D1[k]`

The `items`, `keys`, and `values` methods return values known as *view* objects. If the underlying `dict` changes, the retrieved view changes also; and you are not allowed to alter the set of keys in the underlying `dict` while using a for loop on any of its view objects.

Iterating on any of the view objects yields values in insertion order. In particular, when you call more than one of these methods without any intervening change to the dict, the order of the results is the same for all.

Never modify a dict's set of keys while iterating on it

Never modify the set of keys in a dict (i.e., never add nor remove keys) while iterating over that dict, or any of the iterable views returned by its methods. If you need to avoid such constraints against mutation during iteration, iterate instead on a list explicitly built from the dict or view; for example, iterate on `list(D)`. Iterating directly on a dict `D` is exactly like iterating on `D.keys()`.

The return values of methods `items` and `keys` also implement `set` nonmutating methods and behave much like `frozensets`; the return value of method `values` doesn't, since, differently from the others (and from sets), it may contain duplicates.

The `popitem` method can be used for destructive iteration on a dictionary. Both `items` and `popitem` return dictionary items as key/value pairs. `popitem` is usable for a loop on a huge dictionary, when what you want is to "consume" the dictionary in the course of the loop.

`D.setdefault(k, x)` returns the same result as `D.get(k, x)`, but, when `k` is not a key in `D`, `setdefault` also has the side effect of binding `D[k]` to the value `x`. (In modern Python, `setdefault` is not often used, since type `collections.defaultdict`, covered in #defaultdict, often offers similar, faster, clearer functionality.)

The `pop` method returns the same result as `get`, but, when `k` is a key in `D`, `pop` also has the side effect of removing `D[k]` (when `x` is not specified, and `k` is not a key in `D`, `get` returns `None`, but `pop` raises an exception). `d.pop(key, None)` is a useful shortcut for removing a key from a dict without having to first check if the key is present, much as `s.discard(x)` (as opposed to `s.remove(x)`) when *s* is a set.

The `update` method is accessible with augmented assignment syntax: where `D2` is a dict, `D|=D2` is the same as `D.update(D2)`. Operator syntax, `D|D2`, mutates neither dictionary: rather, it returns a new dictionary result, such that `D3=D|D2` is equivalent to `D3=D.copy(); D3.update(D2)`.

The `update` method (but not the `|` and `|=` operators) can also accept an iterable of key/value pairs, as an alternative argument instead of a mapping, and can accept named arguments instead of—or in addition to—its positional argument; the semantics are the same as for passing such arguments when calling the built-in `dict` type, as covered in #dictionaries.

Control Flow Statements

A program's *control flow* regulates the order in which the program's code executes. The control flow of a Python program mostly depends on conditional statements, loops, and function calls. (This section covers the `if` conditional statement and `for` and `while` loops; we cover the `match` conditional statement in "The match statement", and functions in "Functions".) Raising and handling exceptions also affects control flow; we cover exceptions in Chapter "Exceptions".

The if Statement

Often, you need to execute some statements only when some condition holds, or choose statements to execute depending on mutually exclusive conditions. The compound statement `if`—comprising `if`, `elif`, and `else` clauses—lets you conditionally execute blocks of statements. The syntax for the `if` statement is:

```
if expression:
    statement(s)
elif expression:
    statement(s)
elif expression:
    statement(s)
...
else:
    statement(s)
```

The `elif` and `else` clauses are optional. Before the introduction of the `match` construct (see "The match statement"), `if`, `elif`, and `else` had to be used for all conditional processing.

Here's a typical `if` statement with all three kinds of clauses:

```
if x < 0:
    print('x is negative')
elif x % 2:
    print('x is positive and odd')
else:
    print('x is even and non-negative')
```

Each clause controls one or more statements (known as "a *block*"): place the block's statements on separate logical lines after the line containing the clause's keyword (known as the *header line* of the clause), indented 4 spaces from the header line. The block terminates when the indentation returns to that of the clause header (or further left from there). (This is the style mandated by *PEP 8*).

You can use any Python expression as the condition in an `if` or `elif` clause. Using an expression this way is known as using it *in a Boolean context.* In this context, any value is taken as either true or false. As mentioned earlier, any nonzero number or nonempty container (string, tuple, list, dictionary, set, ...) evaluates as true; zero (of any numeric type), `None`, and empty containers evaluate as false. To test a value *x* in a Boolean context, use the following coding style:

```
if x:
```

This is the clearest and most Pythonic form. Do *not* use any of the following:

```
if x is True:
if x == True:
if bool(x):
```

There is a crucial difference between saying that an expression *returns* `True` (meaning the expression returns the value `1` with the `bool` type) and saying that an expression *evaluates as* true (meaning the expression returns any result that is true in a Boolean context). When testing an expression, for example in an `if` clause, you only care about what it *evaluates as*, not

what, precisely, it *returns*. Informally, “evaluates as true” is often expressed as “is *truthy*”, and “evaluated as false” as “is *falsy*”.

When the `if` clause’s condition evaluates as true, the statements within the `if` clause execute, then the entire `if` statement ends. Otherwise, Python evaluates each `elif` clause’s condition, in order. The statements within the first `elif` clause whose condition evaluates as true, if any, execute, and the entire `if` statement ends. Otherwise, when an `else` clause exists, it executes. In any case, statements following the entire `if` construct, at the same level, execute next.

||3.10+|| The match statement

The `match` statement brings *structural pattern matching* to the Python language. You might think of this as doing for other Python types something similar to what the `re` module (see “Regular expressions and the re module”) does for strings: it allows easy testing of the structure and contents of Python objects[6]. Resist the temptation to use `match` unless there is a need to analyse the *structure* of an object.

The overall syntactic structure of the statement is the new (soft) keyword **match** followed by an expression whose value becomes the *matching subject*. This is followed by one or more indented **case** clauses, each of which controls the execution of the indented code block it contains.

```
match expression:
    case pattern [if guard]:
        statement(s)
        ...
```

In execution, Python first evaluates the *expression*, then tests the resulting *subject* value against the *pattern* in each `case` in turn, in order from first to last, until one matches: then, the block indented within the matching `case` clause evaluates. A pattern can do two things:

- verify that the subject is an object with a particular structure, and

- bind matched components to names for further use (usually within the associated `case` clause).

When a pattern matches the subject, the *guard* allows a final check before selection of the case for execution. All the pattern's name bindings have occurred and you can use them in the guard. When there is no guard, or when the guard evaluates as true, the case's indented code block executes, after which the `match` statement's execution is complete and no further cases are checked.

Unlike the `if` statement, there is no syntactic equivalent to the `else` clause. The `match` statement, per se, provides no default action. If one is needed, the last `case` clause must specify a *wildcard* case—one whose syntax ensures it matches any subject value. It is a `SyntaxError` to follow a `case` clause having such a wildcard pattern with any further `case` clauses.

Pattern elements cannot be created in advance, bound to variables and (for example) re-used in multiple places. Pattern syntax is only valid immediately following the (soft) keyword case, so there is no way to perform such an assignment. For each execution of a `match` statement, the interpreter is free to cache pattern expressions that repeat inside the cases, but the cache starts empty for each new execution.

We first describe the various types of pattern expressions, before discussing guards and providing some more complex examples.

Pattern Expressions

The syntax of pattern expressions can seem familiar, but their interpretation is sometimes quite different from their non-pattern interpretation, which could mislead readers unaware of the differences. Specific syntactic forms are used in the `case` clause to indicate matching of particular structures. To summarise all this syntax would require more than the simplified notation we use in this book[7]. We therefore prefer to explain this new feature in plain language, with examples. For more detailed examples, refer to the

Python documentation, which details the `match` statement features and the various pattern types.

Building Patterns

Patterns are expressions, though with a syntax specific to the case clause, so familiar grammatical rules apply, despite different interpretation of various features. They can be in parentheses, to let elements of a pattern be treated as a single expression unit. Like other expressions, patterns have a recursive syntax and can be combined to form more complex patterns. Let's start with the simplest patterns first.

Literal Patterns

Most literal values are valid patterns. Integer, float, complex number and string literals (but *not* formatted string literals) are all permissible, and all succeed in matching subjects of the same type and value.

```
>>> for subject in (42, 42.0, 42.1, 1+1j, b'abc', 'abc'):
...     print(subject, end=': ')
...     match subject:
...         case 42: print('integer')  # note this matches
42.0,too!
...         case 42.1: print('float')
...         case 1+1j: print('complex')
...         case b'abc': print('bytestring')
...         case 'abc': print('string')
42: integer
42.0: integer
42.1: float
(1+1j): complex
b'abc': bytestring
abc: string
```

For most matches, the interpreter checks for equality, without type checking, which is why 42.0 matches integer 42. If the distinction is important, consider using class matching (see "Class Patterns") rather than literal matching. `True`, `False`, and `None` being singleton objects, each matches itself.

The Wildcard Pattern

In pattern syntax, the underscore (_) plays the role of a wildcard expression. As the simplest wildcard pattern, _ matches any value at all:

```
>>> for subject in 42, 'string', ('tu', 'ple'), ['list'], object:
...     match subject:
...         case _: print('matched', subject)
...
matched 42
matched string
matched ('tu', 'ple')
matched ['list']
matched <class 'object'>
```

Capture Patterns

The use of unqualified names (names with no dots in them) is so different in patterns that we feel it necessary to begin this section with a warning note.

> **Simple Names Bind to Matched Elements Inside Patterns**
>
> Unqualified names—simple identifiers (e.g., color) rather than attribute references (e.g., name.attr)—do not necessarily have their usual meaning in pattern expressions. Some names, rather than being references to values, are instead bound to elements of the subject value during pattern matching.

Unqualified names, except _, are *capture patterns*—they're wildcards, matching anything, but with a side-effect: the name, in the current local namespace, gets bound to the object matched by the pattern. Bindings created by matching remain after the statement has executed, allowing the statements in the case clause and subsequent code to process extracted portions of the subject value.

The example below is similar to the preceding one, except that the name x, instead of the underscore, matches the subject. The absence of exceptions shows that the name captures the whole subject in all cases.

```
>>> for subject in 42, 'string', ('tu', 'ple'), ['list'], object:
...     match subject:
...         case x: assert x == subject
...
```

Value Patterns

This section, too, begins with a warning to remind readers that simple names can't be used to inject their bindings into pattern values to be matched.

> **Represent Variable Values in Patterns with Qualified Names**
>
> Because simple names capture values during pattern matching, you *must* use attribute references (qualified names like *name.attr*) to express values that may change between different executions of the same match statement.

Though this feature is useful, it means you can't reference values directly with simple names. Therefore, in patterns, values must be represented by qualified names, which are known as *value patterns*—they *represent* values, rather than *capturing* them as simple names do. While slightly inconvenient, the use of qualified names is easily accomplished by setting attribute values on an otherwise empty class[8]; for example:

```
>>> class m: v1 = "one"; v2 = 2; v3 = 2.56
...
>>> match ('one', 2, 2.56):
...     case (m.v1, m.v2, m.v3):  print('matched')
...
matched
```

It is also relatively easy to give yourself access to the current module's global namespace, like this:

```
>>> import sys
>>> g = sys.modules[__name__]
```

```
>>> v1 = "one"; v2 = 2; v3 = 2.56
>>> match ('one', 2, 2.56):
...     case (g.v1, g.v2, g.v3):  print('matched')
...
matched
```

OR Patterns

When *P1* and *P2* are patterns, the expression *P1* | *P2* is an *OR pattern*, matching anything that matches either *P1* or *P2*, as shown below. Any number of alternate patterns can be used, and matches are attempted from left to right.

```
>>> for subject in range(5):
...     match subject:
...         case 1 | 3: print('odd')
...         case 0 | 2 | 4: print('even')
even
odd
even
odd
even
```

It is a syntax error to follow a wildcard pattern with further alternatives, since they can never be activated. While our initial examples are simple, remember that the syntax is recursive, so patterns of arbitrary complexity can replace any of the sub-patterns in our examples.

Group Patterns

If *P1* is a pattern, then *(P1)* is also a pattern that matches the same values. This addition of "grouping" parentheses can be useful when patterns become complicated, just as it is with standard expressions. As in other expressions, take care to distinguish between *(P1)*, a simple grouped pattern matching *P1*, and *(P1,)*, a sequence pattern (see "Sequence Patterns") matching a sequence with a single element matching *P1*.

Sequence Patterns

A list or tuple of patterns, optionally with a single starred wildcard (*_) or starred capture pattern (**name*), is a *sequence pattern*. When the starred

pattern is absent, the pattern matches a fixed-length sequence of values of the same length as the pattern. Elements of the sequence are matched one at a time, until all elements have matched (then, matching succeeds), or, an element fails to match (then, matching fails).

When the sequence pattern includes a starred pattern, that sub-pattern matches a sequence of elements sufficiently long to allow the remaining unstarred patterns to match the final elements of the sequence. When the starred pattern is of the form **name*, *name* is bound to the (possibly empty) list of the elements in the middle that don't correspond to individual patterns at the beginning or end.

You can match a sequence with patterns that look like tuples or lists—it makes no difference to the matching process. The next example shows an unnecessarily complicated way to extract the first, middle, and last elements of a sequence.

```
>>> for sequence in (["one", "two", "three"], range(2),
range(6)):
...     match sequence:
...         case  (first, *vars, last): print(first, vars, last)
one ['two'] three
0 [] 1
0 [1, 2, 3, 4] 5
```

AS Patterns

You can use so-called *AS patterns* to capture values matched by more complex patterns, or components of a pattern, which simple capture patterns (see "Capture Patterns" above) cannot.

When *P1* is a pattern, then *P1* **as** *name* is also a pattern; when *P1* succeeds, Python binds the matched value to name *name* in the local namespace. The interpreter tries to ensure that, even with complicated patterns, the same bindings always take place when a match occurs.

```
>>> match subject:
...     case ((0 | 1) as x) | 2: print(x)
...
SyntaxError: alternative patterns bind different names
>>> match subject:
```

```
...     case (2 | x): print(x)
...
SyntaxError: alternative patterns bind different names
>>> match 42:
...     case (1 | 2 | 42) as x: print(x)
42
```

Mapping Patterns

Mapping patterns match mapping objects, usually dictionaries, which associate keys with values. The syntax of mapping patterns uses `key: pattern` pairs. The keys must be either literal or value patterns.

The interpreter iterates over the keys in the mapping pattern, processing each as follows.

- Python looks up the key in the subject mapping; a lookup failure causes immediate match failure.
- Python then matches the extracted value against the pattern associated with the key; if the value fails to match the pattern, then the whole match fails.

When all keys match, the whole match succeeds.

```
>>> match {1: "two", "two": 1}:
...     case {1: v1, "two": v2}: print(v1, v2)
...
two 1
```

You can also use a mapping pattern together with an `as` clause:

```
>>> match {1: "two", "two": 1}:
...     case {1: v1} as v2: print(v1, v2)
...
two {1: 'two', 'two': 1}
```

The `as` pattern in the second example binds *v2* to the whole subject dictionary, not just the matched keys.

The final element of the pattern may optionally be a double-starred capture pattern such as ***name*; when that is the case, Python binds *name* to a

possibly-empty dictionary whose items are the `(key, value)` pairs from the subject mapping whose keys were *not* present in the pattern.

```
>>> match {1: 'one', 2: 'two', 3: 'three'}:
...     case {2: middle, **others}: print(middle, others)
...
two {1: 'one', 3: 'three'}
```

Class Patterns

The final, and maybe the most versatile kind of pattern, is the *class pattern*, offering the ability to match instances of particular classes and their attributes.

A class pattern is of the general form

```
name_or_attr(patterns)
```

where *name_or_attr* is a simple or qualified name bound to a class – specifically, an instance of the built-in type *type* (or of a subclass thereof, but, no super-fancy metaclasses need apply!). *patterns* is a (possibly empty) comma-separated list of pattern specifications. When no pattern specifications are present in a class pattern, the match succeeds whenever the subject is an instance of the given class, so for example the pattern `int()` matches *any* integer.

Like function arguments and parameters, the pattern specifications can be positional (like `pattern`) or named (like *name*=`pattern`). If a class pattern has positional pattern specifications, they must all precede the first named pattern specification. User-defined classes cannot use positional patterns without setting the class's *__match_args__* attribute (see "Configuring Classes for Positional Matching.")

The built-in types `bool`, `bytearray`, `bytes`, `dict`, `float`, `frozenset`, `int`, `list`, `set`, `str`, and `tuple`, are all configured to take a single positional pattern, which is matched against the instance value. For example, the pattern `str(`*x*`)` matches any string and binds its value to

x by matching the string's value against the capture pattern—as does `str() as x`.

You may remember a literal pattern example we presented earlier, showing that literal matching could not discriminate between the integer 42 and the float 42.0 because 42 == 42.0. You can use class matching to overcome that issue:

```
>>> for subject in 42, 42.0:
...     match subject:
...         case int(x): print('integer', x)
...         case float(x): print('float', x)
...
integer 42
float 42.0
```

Once the type of the subject value has matched, for each of the named patterns *name*=`pattern`, Python retrieves the attribute *name* from the instance and matches its value against `pattern`. If all named pattern matches succeed, the whole match succeeds. Python handles positional patterns by converting them to named patterns, as we describe in "Configuring Classes for Positional Matching."

Guards

When a case clause's pattern succeeds, it is often convenient to determine on the basis of values extracted from the match whether this case should execute. When a guard is present, it executes after a successful match. If the guard expression evaluates as false, Python abandons the current case, despite the match, and moves to consider the next case. This example uses a guard to exclude odd integers by checking the value bound in the match.

```
>>> for subject in range(5):
...     match subject:
...         case int(i) if i % 2 == 0: print(i, "is even")
...
0 is even
2 is even
4 is even
```

Configuring Classes for Positional Matching

When you want your own classes to handle positional patterns in matching, you have to tell the interpreter which *attribute of the instance* (**not** "which argument to `__init__`") each positional pattern corresponds to. You do this by setting the class's *__match_args__* attribute to a sequence of names. The interpreter raises a `TypeError` exception if you attempt to use more positional patterns than you defined.

```
>>> class Color:
...     __match_args__ = ('red', 'green', 'blue')
...     def __init__(self, r, g, b, name='anonymous'):
...         self.name = name
...         self.red, self.green, self.blue = r, g, b
...
>>> red = Color(255,0,0, 'red')
>>> blue = Color(0, 0, 255)
>>> for subject in (42.0, red, blue):
...     match subject:
...         case float(x):
                print('float', x)
...         case Color(a, b, c, name='red'):
                print(type(subject).__name__, subject.name a, b,
c)
...         case Color(a, b, c=255) as blue:
...             print(type(blue).__name__, a, b, c, blue.name,)
…                   case _: print(type(subject), subject)
...
float 42.0
Color red 255 0 0
Color 0 0 255 anonymous
>>> match red:
...     case Color(1, 2, 3, 4): print("matched")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: Color() accepts 3 positional sub-patterns (4 given)
```

The while Statement

The `while` statement repeats execution of a statement, or block of statements, as long as a conditional expression evaluates as true. Here's the syntax of the `while` statement:

```
while expression:
    statement(s)
```

A `while` statement can also include an `else` clause, covered in "The else Clause on Loop Statements", and `break` and `continue` statements, covered in "The break Statement" and "The continue Statement".

Here's a typical `while` statement:

```
count = 0
while x > 0:
    x //= 2             # floor division
    count += 1
print('The approximate log2 is', count)
```

First, Python evaluates *expression*, which is known as the *loop condition*, in a Boolean context. When the condition is false, the `while` statement ends. When the loop condition evaluates as true, the statement or block of statements that make up the *loop body* execute. Once the loop body finishes executing, Python evaluates the loop condition again, to check whether another iteration should execute. This process continues until the loop condition evaluates as false, at which point the `while` statement ends.

The loop body should contain code that eventually makes the loop condition false, otherwise the loop never ends (unless the body raises an exception or executes a `break` statement). A loop within a function's body also ends if the loop body executes a `return` statement, since in this case the whole function ends.

The for Statement

The `for` statement repeats execution of a statement, or block of statements, controlled by an iterable expression. Here's the syntax of the `for` statement:

```
for target in iterable:
    statement(s)
```

The `in` keyword is part of the syntax of the `for` statement; its purpose here is distinct from the `in` operator, which tests membership.

Here's a rather typical `for` statement:

```
for letter in 'ciao':
    print(f'give me a {letter}...')
```

A `for` statement can also include an `else` clause, covered in "The else Clause on Loop Statements", and `break` and `continue` statements, covered in "The break Statement" and "The continue Statement". *iterable* may be any iterable Python expression (as explained in detail in the next section). In particular, any sequence is iterable. The interpreter calls the iterable's `__iter__` method to produce an *iterator* (see "Iterators"), which it then iterates over.

target is normally an identifier naming the *control variable* of the loop; the `for` statement successively rebinds this variable to each item of the iterator, in order. The statement or statements that make up the *loop body* execute once for each item in *iterable* (unless the loop ends because of an exception or a `break` or `return` statement). Since the loop body may terminate before the iterator is exhausted, this is one case in which you may use an *unbounded* iterable—one that, per se, would never cease yielding items.

You can also use a target with multiple identifiers, as in an unpacking assignment. In this case, the iterator's items must themselves be iterables, each with exactly as many items as there are identifiers in the target. Precisely *one* of the identifiers may be preceded by a star, in which case the starred identifier is bound to a list of all items not assigned to other targets. For example, when *d* is a dictionary, this is a typical way to loop on the items (key/value pairs) in *d*:

```
for key, value in d.items():
    if key and value:              # print only truish keys and
values
        print(key, value)
```

The `items` method returns another kind of iterable (a "view"), whose items are key/value pairs; so, we use a `for` loop with two identifiers in the target to unpack each item into `key` and `value`. Although components of a target are commonly identifiers, values can be bound to any acceptable LHS expression as covered in "Assignment Statements":

```
prototype = [1, 'placeholder', 3]
for prototype[1] in 'xyz':
    print(prototype)
# prints [1, 'x', 3], then [1, 'y', 3], then [1, 'z', 3]
```

Don't Alter Mutable Objects While Looping on Them

When an iterator has a mutable underlying iterable, don't alter that underlying object during a `for` loop on the iterable. For example, the preceding key/value printing example cannot alter *d*. The `items` method returns a "view" iterable whose underlying object is *d*, so the loop body cannot mutate the set of keys in *d* (*e.g.*, by executing `del` *d*`[`*key*`]`). To ensure that *d* is not the underlying object of the iterable, you may, for example, iterate over *list(d.items())* to allow the loop body to mutate *d*. Specifically:

- When looping on a list, do not insert, append, or delete items (rebinding an item at an existing index is OK) into that list.
- When looping on a dictionary, do not add or delete items (rebinding the value for an existing key is OK) into that dict.
- When looping on a set, do not add or delete items (no alteration permitted).

The loop body may rebind control target variable(s), but the next iteration of the loop will always rebind them again. If the iterator yields no items, the loop body does not execute at all. In this case, the `for` statement does not bind or rebind its control variable it in any way. However, if the iterator yields at least one item, then, when the loop statement ends, the control variable remains bound to the last value to which the loop statement bound it. The following code is therefore correct *only* when `someseq` is not empty:

```
for x in someseq:
    process(x)
print(f'Last item processed was {x}')  # potential NameError on
empty sequence
```

Iterators

An *iterator* is an object *i* such that you can call `next(`*i*`)`, which returns the next item of iterator *i* or, when exhausted, raises a `StopIteration` exception. Alternatively, you can call `next(`*i*, *default*`)`, in which case, when iterator *i* has no more items, the call returns *default*.

When you write a class (see "Classes and Instances"), you can let instances of the class be iterators by defining a special method `__next__` that takes no argument except `self`, and returns the next item or raises `StopIteration`. Most iterators are built by implicit or explicit calls to built-in function `iter`, covered in Table 7-2. Calling a generator also returns an iterator, as we discuss in "Generators".

As pointed out in "The for statement", the `for` statement implicitly calls `iter` on iits iterable to get an iterator. The statement:

```
for x in c:
    statement(s)
```

is exactly equivalent to:

```
_temporary_iterator = iter(c)
while True:
    try:
            x = next(_temporary_iterator)
    except StopIteration:
            break
    statement(s)
```

where *_temporary_iterator* is an arbitrary name not used elsewhere in the current scope.

Thus, when `iter(c)` returns an iterator *i* such that `next`*(i)* never raises `StopIteration` (an *unbounded iterator*), the loop `for` *x* `in` *c* continues indefinitely unless the loop body includes suitable `break` or `return`

statements, or raises or propagates exceptions. `iter(`*c*`)`, in turn, calls special method *c*`.__iter__()` to obtain and return an iterator on *c*. We'll talk more about the special method `__iter__` in "Container methods".

Many of the best ways to build and manipulate iterators are found in the standard library module `itertools`, covered in "The itertools Module".

Iterables vs. Iterators

Python's built-in sequences, like all iterables, implement an `__iter__` method, which the interpreter calls to produce an iterator over the iterable. Because each call to an iterable's `__iter__` method produces a new iterator, it is possible to nest multiple iterations over the same iterable.

```
>>> iterable = [1, 2]
>>> for i in iterable:
...     for j in iterable:
...         print(i, j)
...
1 1
1 2
2 1
2 2
```

Iterators also implement an `__iter__` method, but it always returns *self*, so nesting iterations over them doesn't work as expected.

```
>>> iterator = iter([1, 2])
>>> for i in iterator:
...     for j in iterator:
...         print(i, j)
...
1 2
```

Here both the inner and outer loops are iterating over the same iterator. By the time the inner loop first gets control, the first value from the iterator has already been consumed. The first iteration of the inner loop exhausts the iterator, which therefore terminates the loops when the next iteration is attempted.

range

Looping over a sequence of integers is a common task, so Python provides built-in function `range` to generate integer sequences. The simplest way to loop *n* times in Python is:

```
for i in range(n):
    statement(s)
```

`range(x)` generates the consecutive integers from `0` (included) up to *x* (excluded). `range(x, y)` generates a list whose items are consecutive integers from *x* (included) up to *y* (excluded). `range(x, y, stride)` generates a list of integers from *x* (included) up to *y* (excluded), such that the difference between each two adjacent items is *stride*. If *stride* is less than `0`, `range` counts down from *x* to *y*.

`range` generates an empty iterator when *x* is >= *y* and *stride* is > `0`, or when *x* is <= *y* and *stride* is < `0`. When *stride* == `0`, `range` raises an exception.

`range` returns a special-purpose object, intended just for use in iterations like the `for` statement shown previously. `range` returns an iterable, not an iterator; you can easily obtain such an iterator, should you need one, by calling `iter(range(...))`. The special-purpose object returned by `range` consumes less memory (for wide ranges, *much* less memory) than the equivalent list object would. If you *need* a `list` that's an arithmetic progression of `int`s, call `list(range(...))`. You will most often find that you don't, in fact, need such a complete list to be fully built in memory.

List comprehensions

A common use of a `for` loop is to inspect each item in an iterable and build a new list by appending the results of an expression computed on some or all of the items. The expression form known as a *list comprehension* or *listcomp* lets you code this common idiom concisely and directly. Since a list comprehension is an expression (rather than a block of statements), you can use it wherever you need an expression (e.g., as an argument in a

function call, in a `return` statement, or as a subexpression of some other expression).

A list comprehension has the following syntax:

```
[ expression for target in iterable lc-clauses ]
```

target and *iterable* are the same as in a regular `for` statement. When *expression* denotes a tuple, you must enclose it in parentheses.

lc-clauses is a series of zero or more clauses, each with one of the two forms:

```
for target in iterable
if expression
```

target and *iterable* in each `for` clause of a list comprehension have the same syntax and meaning as those in a regular `for` statement, and the *expression* in each `if` clause of a list comprehension has the same syntax and meaning as the *expression* in a regular `if` statement.

A list comprehension is equivalent to a `for` loop that builds the same list by repeated calls to the resulting list's `append` method. For example (assigning the list comprehension result to a variable for clarity):

```
result1 = [x+1 for x in some_sequence]
```

is (apart from the different variable name) the same as the `for` loop:

```
result2 = []
for x in some_sequence:
    result2.append(x+1)
```

Here's a list comprehension that uses an `if` clause:

```
result3 = [x+1 for x in some_sequence if x>23]
```

This list comprehension is the same as a `for` loop that contains an `if` statement:

```
result4 = []
for x in some_sequence:
    if x>23:
        result4.append(x+1)
```

Here's a list comprehension using a nested `for` clause to flatten a "list of lists" into a single list of items:

```
result5 = [x for sublist in listoflists for x in sublist]
```

This is the same as a `for` loop with another `for` loop nested inside:

```
result6 = []
for sublist in listoflists:
    for x in sublist:
        result6.append(x)I
```

As these examples show, the order of `for` and `if` in a list comprehension is the same as in the equivalent loop, but, in the list comprehension, the nesting remains implicit. If you remember "order `for` clauses as in a nested loop," that can help you correctly get the ordering of the list comprehension's clauses.

Don't Build A List Unless You Need To

If you are just going to loop over the items, rather than needing an actual, indexable, list, use a generator expression instead (covered in "Generator expressions"). This avoids list creation, and uses less memory. In particular, resist the temptation to use comprehensions as a "single-line loop" such as

```
[side_effects_but_no_return_value(x) for x in seq]
```

-- just use a normal for loop instead!

List Comprehensions And Variable Scope

A list comprehension expression evaluates in its own scope (as do set and dict comprehensions, described in the following sections, and generator expressions—see "Generator expressions"). When a *target* component in the for statement is a name, the name is defined solely within the expression scope and is not available outside it.

Set comprehensions

A *set comprehension* has exactly the same syntax and semantics as a list comprehension, except that you enclose it in braces (`{}`) rather than in brackets (`[]`). The result is a `set`; for example:

```
s = {n//2 for n in range(10)}
print(sorted(s))  # prints: [0, 1, 2, 3, 4]
```

A similar list comprehension would have each item repeated twice, but a `set` removes duplicates.

Dict comprehensions

A *dict comprehension* has the same syntax as a set comprehension, except that, instead of a single expression before the `for` clause, you use two expressions with a colon `:` between them—*key*`:`*value*. The result is a `dict`, which retains insertion ordering. For example:

```
d = {s: i for (i, s) in enumerate(['zero', 'one', 'two'])}
print(d)  # prints: {'zero': 0, 'one': 1, 'two': 2}
```

The break Statement

You can use a `break` statement **only** within a loop body. When `break` executes, the loop terminates *without executing any* `else` *clause the loop may have*. When loops are nested, a `break` terminates only the innermost nested loop. In practice, a `break` is typically within a clause of an `if` (or,

occasionally, a `match`) statement in the loop body, so that `break` executes conditionally.

One common use of `break` is to implement a loop that decides whether it should keep looping only in the middle of each loop iteration (what Donald Knuth called the "loop and a half" structure in his great 1974 paper "Structured Programming with go to Statements"[9]). For example:

```
while True:           # this loop can never terminate "naturally"
    x = get_next()
    y = preprocess(x)
    if not keep_looping(x, y):
        break
    process(x, y)
```

The continue Statement

The `continue` statement can exist only within a loop body. It causes the current iteration of the loop body to terminate, and execution continues with the next iteration of the loop. In practice, a `continue` is usually within a clause of an `if` (or, occasionally, a `match`) statement in the loop body, so that `continue` executes conditionally.

Sometimes, a `continue` statement can take the place of nested `if` statements within a loop. For example, here each *x* has to pass multiple tests before being completely processed:

```
for x in some_container:
    if seems_ok(x):
        lowbound, highbound = bounds_to_test()
        if lowbound <= x < highbound:
            pre_process(x)
            if final_check(x):
                do_processing(x)
```

Nesting increases with the number of conditions. Equivalent code with `continue` flattens the logic:

```
for x in some_container:
    if not seems_ok(x):
        continue
```

```
    lowbound, highbound = bounds_to_test()
    if x < lowbound or x >= highbound:
        continue
    pre_process(x)
    if final_check(x):
        do_processing(x)
```

Flat Is Better Than Nested

Both versions work the same way, so which one you use is a matter of personal preference and style. One of the principles of The Zen of Python (which you can see at any time by typing `import this` at an interactive Python interpreter prompt) is "Flat is better than nested." The `continue` statement is just one way Python helps you move toward the goal of a less-nested structure in a loop, if you so choose.

The else Clause on Loop Statements

`while` and `for` statements may optionally have a trailing `else` clause. The statement or block under that `else` executes when the loop terminates *naturally* (at the end of the `for` iterator, or when the `while` loop condition becomes false), but not when the loop terminates *prematurely* (via `break`, `return`, or an exception). When a loop contains one or more `break` statements, you often want to check whether the loop terminates naturally or prematurely. You can use an `else` clause on the loop for this purpose:

```
for x in some_container:
    if is_ok(x):
        break  # item x is satisfactory, terminate loop
else:
    print('Beware: no satisfactory item was found in container')
    x = None
```

The pass Statement

The body of a Python compound statement cannot be empty; it must always contain at least one statement. You can use a `pass` statement, which performs no action, as an explicit placeholder when a statement is

syntactically required but you have nothing to do. Here's an example of using `pass` in a conditional statement as a part of somewhat convoluted logic to test mutually exclusive conditions:

```
if condition1(x):
    process1(x)
elif x>23 or (x<5 and condition2(x)):
    pass  # nothing to be done in this case
elif condition3(x):
    process3(x)
else:
    process_default(x)
```

Empty `def` or `class` Statements: Use a Docstring, Not `pass`

As the body of an otherwise empty `def` or `class` statement, use a docstring, covered in "Docstrings"; when you do write a docstring, you do not need to also add a `pass` statement (you can do so if you wish, but it's not optimal Python style).

The try and raise Statements

Python supports exception handling with the `try` statement, which includes `try`, `except`, `finally`, and `else` clauses. Your code can explicitly raise an exception with the `raise` statement. All of this is discussed in detail in "Exception Propagation" in Chapter "Exceptions". When code raises an exception, normal control flow of the program stops, and Python looks for a suitable exception handler.

The with Statement

A `with` statement can often be a more readable, useful alternative to the `try/finally` statement. We discuss it in detail in "The with Statement and Context Managers" in Chapter "Exceptions". A good grasp of context

managers can often help you structure code more clearly without compromising efficiency.

Functions

Most statements in a typical Python program are part of some function. Code in a function body may be faster than at a module's top level, as covered in "Avoid exec and from ... import *", so there are excellent practical reasons to put most of your code into functions, and *no* disadvantages: clarity, readability and code reusability all improve when you avoid having any substantial chunks of module-level code.

A *function* is a group of statements that execute upon request. Python provides many built-in functions and lets programmers define their own functions. A request to execute a function is known as a *function call*. When you call a function, you can pass arguments that specify data upon which the function performs its computation. In Python, a function always returns a result value, either `None` or a value, the result of the computation. Functions defined within `class` statements are also known as *methods*. We cover issues specific to methods in "Bound and Unbound Methods"; the general coverage of functions in this section, however, also applies to methods.

Python is somewhat unusual in the flexibility it affords the programmer in defining and calling functions. This flexibility does mean that some constraints are not adequately expressed solely by the syntax. In Python, functions are objects (values), handled just like other objects. Thus, you can pass a function as an argument in a call to another function, and a function can return another function as the result of a call. A function, just like any other object, can be bound to a variable, can be an item in a container, and can be an attribute of an object. Functions can also be keys into a dictionary. The fact that functions are ordinary objects in Python is often expressed by saying that functions are *first-class* objects.

For example, given a dict keyed by functions, with values being each function's inverse, you could make the dictionary bidirectional by adding

the inverse values as keys, with their corresponding keys as values. Here's a small example of this idea, using some functions from module `math`, covered in "The math and cmath Modules", that takes a one-way mapping of inverse pairs and then adds the inverse of each entry to complete the mapping:

```
def add_inverses(i_dict):
    for f in list(i_dict):  #  iterates over keys while mutating
i_dict
        i_dict[i_dict[f]] = f
math_map = {sin:asin, cos:acos, tan:atan, log:exp}
add_inverses(math_map)
```

Note that in this case the function mutates its argument (whence its need to use a `list` call). In Python, the usual convention is for such functions not to return a value (see "The return statement").

Defining Functions: the def Statement

The `def` statement is the usual way to create a function. `def` is a single-clause compound statement with the following syntax:

```
def function_name(parameters):
   statement(s)
```

function_name is an identifier, and the non-empty indented *statement(s)* are the *function body*. When the interpreter meets a `def` statement, it compiles the function body, creating a function object, and binds (or rebinds, if there was an existing binding) *function_name* to the compiled function object in the containing namespace (typically the module namespace, or a class namespace when defining methods).

parameters is an optional list specifying the identifiers that will be bound to values that each function call provides. We distinguish between those identifiers, and the values provided for them in calls, by referring to the former as *parameters* and the latter as *arguments*.

In the simplest case, a function defines no parameters, meaning the function won't accept any arguments when you call it. In this case, the `def` statement has empty parentheses after *function_name*, as must all calls. Otherwise, *parameters* will be a list of specifications (see "Parameters" below). The function body does not execute when the `def` statement executes. Rather, Python compiles it into bytecode, saves it as the function object's `__code__` attribute, and executes it later on each call to the function. The function body can contain zero or more occurrences of the `return` statement, as we'll discuss shortly.

Each call to the function supplies argument expressions corresponding to the parameters defined in the function definition. The interpreter evaluates the argument expressions from left to right and creates a new namespace in which it binds the argument values to the parameter names as local variables of the function call (as we discuss later in "Calling functions"). Then, Python executes the function body, with the function-call namespace as the local namespace.

Here's a simple function that returns a value that is twice the value passed to it each time it's called:

```
def twice(x):
    return x*2
```

The argument can be anything that you can multiply by two, so you could call the function with a number, string, list, or tuple as an argument: each call returns a new value of the same type as the argument.

Function signatures

The number of parameters of a function, together with the parameters' names, the number of mandatory parameters, and the information on whether and where unmatched arguments should be collected, are a specification known as the function's *signature*. A signature defines how you can call the function.

Parameters

Parameters (pedantically, *formal parameters*) name the values passed into a function call, and may specify default values for them. Each time you call the function, the call binds each parameter name to the corresponding argument value in a new local namespace, which Python later destroys on function exit.

Besides letting you name individual arguments, Python also lets you collect argument values not matched by individual parameters, and lets you specifically require that some arguments be positional, or named.

Positional parameters

Each positional parameter is an identifier *name*, which names the parameter. You use these names inside the function body to access the argument values to the call. Callers can normally provide values for these parameters with either *positional* or *named* arguments (see "Matching arguments to parameters").

Named parameters

Each of these takes the form *name=expression*. They are also known as *default*, *optional*, and even, alas!—confusingly, since they do not involve any Python keyword—*keyword parameters*. Executing the `def` statement, the interpreter evaluates each such *expression* and saves the resulting value, known as the *default value* for the parameter, among the attributes of the function object. A function call need not provide an argument value for a named parameter: in that case, the call binds it to its default value. Unless a function's signature includes a positional argument collector (see below), the call may provide positional arguments as values for some named parameters (see "Matching arguments to parameters").

Python computes each default value **exactly once**, when the `def` statement executes, *not* each time you call the resulting function. In particular, this means that Python binds exactly the *same* object, the default value, to the named parameter, whenever the caller does not supply a corresponding argument.

Beware Using Mutable Default Values

A function can alter a mutable default value, such as a list, each time you call the function without an argument corresponding to the respective parameter. This is usually not the behavior you want; see all details under “Mutable default parameter values”.

Positional-only marker

||**3.8**++|| A function’s signature may contain a single *positional-only marker* (`/`) as a dummy parameter. The parameters preceding the marker are known as *positional-only parameters*, and *must* be provided as positional arguments, **not** named arguments, when calling the function. Using named arguments for these parameters raises a `TypeError` exception.

Positional argument collector

This can take one of two forms, either **name* or (||**3.8**++||) just `*`. In the former case, *name* is bound at call-time to a tuple of unmatched positional arguments (see “Matching arguments to parameters”—when all positional arguments are matched, the tuple is empty). In the latter case (the `*` is a dummy parameter), a call with unmatched positional arguments raises a TypeError exception.

When a function’s signature has a positional argument collector, no call can provide a positional argument for a named parameter: either the collector prohibits (in the `*` form), or provides a destination for (in the **name* form), positional arguments not corresponding to positional parameters.

Named argument collector

This final, optional parameter specification has the form ***name*. When the function is called, *name* is bound to a dictionary whose items are the (name, value) pairs of any unmatched named arguments (see “Matching arguments to parameters”), or an empty dictionary if there are no such arguments.

Parameter sequence

Generally speaking, positional parameters are followed by named parameters, with the positional and named argument collectors (if present) last. The positional-only marker, however, may appear at any position in the list of parameters.

Mutable default parameter values

When a named parameter's default value is a mutable object, things get tricky if the function body alters the parameter. For example:

```
def f(x, y=[]):
    y.append(x)
    return id(y), y
print(f(23))              # prints: (4302354376, [23])
print(f(42))              # prints: (4302354376, [23, 42])
```

The second `print` prints `[23, 42]` because the first call to `f` altered the default value of `y`, originally an empty list `[]`, by appending `23` to it. The `id` values confirm that both calls return the same object. If you want `y` to be a new, empty list object, each time you call `f` with a single argument (a far more frequent need!), use the following idiom instead:

```
def f(x, y=None):
    if y is None:
        y = []
    y.append(x)
    return id(y), y
print(f(23))              # prints: (4302354376, [23])
print(f(42))              # prints: (4302180040, [42])
```

Of course, there are cases in which you explicitly want to alter a parameter's default value, most often for caching purposes, as in the following example:

```
def cached_compute(x, _cache={}):
    if x not in _cache:
        _cache[x] = costly_computation(x)
    return _cache[x]
```

Such caching behavior (also known as *memoization*), however, is usually best obtained by decorating the underlying function with **functools.lru_cache**, covered in Table 7-4.

Argument collector parameters

The presence of argument collectors (the special forms *, **name* and ***name*) in a function's signature allows functions to prohibit (*) or collect positional (**name*) or named(***name*) arguments that do not match any parameters (see "Matching arguments to parameters"). There is no requirement to use particular names—you can use any identifier you want in each special form. *args* and *kwds* or *kwargs*, as well as *a* and *k*, are popular choices. We discuss positional and named *arguments* in "Calling Functions".

The presence of the special form * causes calls with unmatched positional arguments to raise a *TypeError* exception.

**args* specifies that any extra positional arguments to a call (*i.e.*, positional arguments not matching positional parameters in the signature, as we cover in "Function signatures") get collected into a (possibly empty) tuple, bound in the call's local namespace to the name *args*. Without a positional arguments collector, unmatched positional arguments raise a *TypeError* exception.

Similarly, ***kwds* specifies that any extra named arguments (i.e., those named arguments not explicitly specified in the signature, as we cover in "Function signatures") get collected into a (possibly empty) dictionary whose items are the names and values of those arguments, bound to the name *kwds* in the function call namespace. Without a named arguments collector, unmatched named arguments raise a *TypeError* exception.

For example, here's a function that accepts any number of positional arguments and returns their sum (and demonstrates the use of an identifier other than **args*):

```
def sum_sequence(*numbers):
    return sum(numbers)
```

```
print(sum_sequence(23, 42))         # prints: 65
```

Attributes of Function Objects

The `def` statement sets some attributes of a function object `f`. String attribute `f.__name__` is the identifier that `def` uses as the function's name. You may rebind `__name__` to any string value, but trying to unbind it raises a *TypeError* exception. `f.__defaults__`, which you may freely rebind or unbind, is the tuple of default values for named parameters (empty, if the function has no named parameters).

Docstrings

Another function attribute is the *documentation string*, also known as the *docstring*. You may use or rebind a function `f`'s docstring attribute as `f.__doc__`. When the first statement in the function body is a string literal, the compiler binds that string as the function's docstring attribute. A similar rule applies to classes (see "Class documentation strings") and modules (see "Module documentation strings"). Docstrings can span multiple physical lines, so it's best to specify them in triple-quoted string literal form. For example:

```
def sum_sequence(*numbers):
    """Return the sum of multiple numerical arguments.

    The arguments are zero or more numbers.
    The result is their sum.
    """

    return sum(numbers)
```

Documentation strings should be part of any Python code you write. They play a role similar to that of comments, but the're even more useful, since they remain available at runtime (unless you run your program with **python -OO**, as covered in "Command-Line Syntax and Options"). Python's `help` function (see "The help function"), development environments, and other tools, can use the docstrings from function, class, and module objects to remind the programmer how to use those objects. The `doctest` module

(covered in “The doctest Module”) makes it easy to check that sample code present in docstrings is accurate and correct, and remains so as the code and docs get edited and maintained.

To make your docstrings as useful as possible, respect a few simple conventions, as detailed in PEP 257. The first line of a docstring should be a concise summary of the function’s purpose, starting with an uppercase letter and ending with a period. It should not mention the function’s name, unless the name happens to be a natural-language word that comes naturally as part of a good, concise summary of the function’s operation. Use imperative rather than descriptive form: e.g., say “Return xyz…” rather than “Returns xyz…”. If the docstring is multiline, the second line should be empty, and the following lines should form one or more paragraphs, separated by empty lines, describing the function’s parameters, preconditions, return value, and side effects (if any). Further explanations, bibliographical references, and usage examples (which you should check with `doctest`) can optionally (and often very usefully!) follow, toward the end of the docstring.

Other attributes of function objects

In addition to its predefined attributes, a function object may have other arbitrary attributes. To create an attribute of a function object, bind a value to the appropriate attribute reference in an assignment statement after the `def` statement executes. For example, a function could count how many times it gets called:

```
def counter():
    counter.count += 1
    return counter.count
counter.count = 0
```

Note that this is *not* common usage. More often, when you want to group together some state (data) and some behavior (code), you should use the object-oriented mechanisms covered in Chapter “Object-oriented Python”. However, the ability to associate arbitrary attributes with a function can sometimes come in handy.

Function Annotations

Every parameter in a `def` clause can be *annotated* with an arbitrary expression—that is, wherever within the `def`'s parameter list you can use an identifier, you can alternatively use the form *identifier:expression*, and the expression's value becomes the *annotation* for that parameter.

You can also annotate the return value of the function, using the form `->` *expression* between the `)` of the `def` clause and the `:` that ends the `def` clause; the expression's value becomes the annotation for name `'return'`. For example:

```
>>> def f(a:'foo', b)->'bar': pass
...
>>> f.__annotations__{'a': 'foo', 'return': 'bar'}
```

As shown in this example, the `__annotations__` attribute of the function object is a `dict` mapping each annotated identifier to the respective annotation.

You can currently, in theory, use annotations for whatever purpose you wish: Python itself does nothing with them, except construct the `__annotations__` attribute. However, this is possibly due to change ||3.11++||, focusing annotation on "type-hinting" purposes only. For detailed information about annotations used for type hinting, see Chapter "Type Annotations".

The return Statement

You can use the `return` keyword in Python only inside a function body, and you can optionally follow it with an expression. When `return` executes, the function terminates, and the value of the expression is the function's result. A function returns `None` when it terminates by reaching the end of its body, or by executing a `return` statement with no expression (or by explicitly executing `return None`).

Good Style in return Statements

As a matter of good style, when some `return` statements in a function have an expression, then all `return` statements in the function should have an expression. `return None` should only ever be written explicitly to meet this style requirement. Never write a `return` statement without an expression at the end of a function body. Python does not enforce these stylistic conventions, but your code is clearer and more readable when you follow them.

Calling Functions

A function call is an expression with the following syntax:

```
function_object(arguments)
```

function_object may be any reference to a function (or other callable) object; most often, it's just the function's name. The parentheses denote the function-call operation itself. *arguments*, in the simplest case, is a series of zero or more expressions separated by commas (`,`), giving values for the function's corresponding parameters. When the function call executes, the parameters are bound to the argument values in a new namespace, the function body executes, and the value of the function-call expression is whatever the function returns. Objects created inside and returned by the function are liable to garbage-collection unless the caller retains a reference to them.

Don't Forget The Trailing () To Call A Function

Just mentioning a function (or other callable object) does not, per se, call it. To call a function (or other object) without arguments, you must use `()` after the function's name (or other reference to the callable object).

Positional and named arguments

Arguments can be of two types. *Positional* arguments are simple expressions; *named* (also known, alas!, as *keyword*[10]) *arguments* take the form

```
identifier=expression
```

It is a syntax error for named arguments to precede positional ones in a function call. Zero or more positional arguments may be followed by zero or more named arguments. Each positional argument supplies the value for the parameter that corresponds to it by position (order) in the function definition. There is no requirement for positional arguments to match positional parameters, or *vice versa*—if there are more positional arguments than positional parameters, the additional arguments are bound by position to named parameters, if any, for all parameters preceding an argument collector in the signature. For example:

```
def f(a, b, c=23, d=42, *x):
    print(a, b, c, d, x)
f(1,2,3,4,5,6)  # prints (1, 2, 3, 4, (5, 6))
```

Note that it matters where in the function signature the argument collector occurs—see "Matching arguments to parameters" for all the gory details!

```
def f(a, b, *x, c=23, d=42):
    print(a, b, x, c, d)
f(1,2,3,4,5,6)  # prints 1 2 (3, 4, 5, 6) 23 42
```

In the absence of any `**kwds` parameter, each argument's name must be one of the parameter names used in the function's signature[11]. The *expression* supplies the value for the parameter of that name. Many built-in functions do not accept named arguments: you must call such functions with positional arguments only. However, functions coded in Python usually accept named as well as positional arguments, so you may call them in different ways. Positional parameters can be matched by named arguments, in the absence of matching positional arguments.

A function call must supply, via a positional or a named argument, exactly one value for each mandatory parameter, and zero or one value for each optional parameter[12]. For example:

```
def divide(divisor, dividend=94):
    return dividend // divisor
print(divide(12))                           # prints: 7
print(divide(12, 94))                       # prints: 7
print(divide(dividend=94, divisor=12))      # prints: 7
print(divide(divisor=12))                   # prints: 7
```

As you can see, the four calls to `divide` are equivalent. You can pass named arguments for readability purposes whenever you think that identifying the role of each argument and controlling the order of arguments enhances your code's clarity.

A common use of named arguments is to bind some optional parameters to specific values, while letting other optional parameters take default values:

```
def f(middle, begin='init', end='finis'):
    return begin+middle+end
print(f('tini', end=''))                    # prints: inittini
```

With named argument `end=''`, the caller specifies a value (the empty string `''`) for `f`'s third parameter, `end`, and still lets `f`'s second parameter, `begin`, use its default value, the string `'init'`. You may pass the arguments as positional, even when parameters are named; for example, with the preceding function:

```
print(f('a','c','t'))                       # prints: cat
```

At the end of the arguments in a function call, you may optionally use either or both of the special forms **seq* and ***dct*. If both forms are present, the form with two asterisks must be last. **seq* passes the items of iterable *seq* to the function as positional arguments (after the normal positional arguments, if any, that the call gives with the usual syntax). *seq* may be any iterable. ***dct* passes the items of *dct* to the function as named arguments, where *dct*

must be a mapping whose keys are all strings. Each item's key is a parameter name, and the item's value is the argument's value.

You may want to pass an argument of the form **seq* or ***dct* when the parameters use similar forms, as covered earlier in "Parameters". For example, using the function `sum_sequence` defined in that section (and shown again here), you may want to print the sum of all the values in dictionary *d*. This is easy with **seq*:

```
def sum_sequence(*numbers):
    return sum(numbers)
print(sum_sequence(*d.values()))
```

(Of course, `print(sum(d.values()))` would be simpler and more direct.)

You may also pass arguments **seq* or ***dct* when calling a function that does not use the corresponding forms in its signature. In that case, you must ensure that iterable *seq* has the right number of items, or, respectively, that dictionary *dct* uses the right identifier strings as keys; otherwise, the call raises an exception. As noted in "'Keyword-only' Parameters", below, a positional argument *cannot* match a keyword-only parameter; only a named argument, explicit or passed via ***kwargs*, can do that.

A function call may have zero or more occurrences of **seq* and/or ***dct*, as specified in PEP 448.

"Keyword-only" parameters

Parameters after a positional argument collector (**name* or `*`) in the function's signature are known as *keyword-only parameters*: corresponding arguments, if any, *must* be named arguments. In the absence of any match by name, such a parameter is bound to its default value, as set when you defined the function.

Keyword-only parameters can be either positional or named. You *must* pass them as named arguments, not as positional ones. It's more usual and readable to have simple identifiers, if any, at the start of the keyword-only

parameter specifications, and *identifier=default* forms, if any, following them, though this is not a requirement of the Python language.

Functions requiring keyword-only parameter specifications *without* collecting positional arguments indicate the start of the keyword-only parameter specifications with a dummy parameter consisting solely of an asterisk (`*`), to which no argument corresponds. For example:

```
def f(a, *, b, c=56):    # b and c are keyword-only
    return a, b, c
f(12,b=34)  # returns (12, 34, 56) – c's optional, since it has a
default
f(12)       # raises a TypeError exception, since you didn’t pass
`b`:
# error message is: missing 1 required keyword-only argument: 'b'
```

If you also specify the special form ***kwds*, it must come at the end of the parameter list (after the keyword-only parameter specifications, if any). For example:

```
def g(x, *a, b=23, **k):   # b is keyword-only
    return x, a, b, k
g(1, 2, 3, c=99)  # returns (1, (2, 3), 23, {'c': 99})
```

Matching arguments to parameters

A call *must* provide an argument for all positional parameters, and *may* do so for named parameters.

The matching proceeds as follows.

1. Arguments of the form **expression* are internally replaced by a sequence of positional arguments obtained by iterating over *expression*.

2. Arguments of the form ***expression* are internally replaced by a sequence of keyword arguments whose names and values are obtained by iterating over *expression*’s *items()*.

3. Say that the function has N positional parameters and the call has M positional arguments:
   - When M≤N, bind all the positional arguments to the first M positional parameter names; remaining positional parameters, if any, *must* be matched by named arguments.
   - When M>N, bind remaining positional arguments to named parameters *in the order in which they appear in the signature*. This process terminates in one of three ways:

     — All positional arguments have been bound.

     — The next item in the signature is a * argument collector: the interpreter raises a *TypeError* exception.

     — The next item in the signature is a **name* argument collector: the remaining positional arguments are collected in a tuple that is then bound to *name* in the function call namespace.

4. The named arguments are then matched, in the order of the arguments' occurrences in the call, by name with the parameters—both positional and named. Attempts to rebind an already-bound parameter name raise a *TypeError* exception.

5. If unmatched named arguments remain at this stage:
   - When the function signature includes a **name collector, the interpreter creates a dictionary that keys the argument values with their names and binds it to name in the function call namespace.
   - In the absence of such an argument collector, Python raises a TypeError exception.

6. Any remaining unmatched named parameters are bound to their default values.

7. At this point, the function call namespace is fully populated, and the interpreter executes the function's body using that "call namespace" as the local namespace for the function.

The semantics of argument passing

In traditional terms, all argument passing in Python is *by value* (although, in modern terminology, to say that argument passing is *by object reference* is more precise and accurate; you may also check out the synonym *call by sharing*). When you pass a variable as an argument, Python passes to the function the object (value) to which the variable currently refers(not "the variable itself"!), binding this object to the parameter name in the function call namespace. Thus, a function *cannot* rebind the caller's variables. However, if you pass a mutable object as an argument, the function may make changes to that object, because Python passes a reference to the object itself, not a copy. Rebinding a variable and mutating an object are totally disjoint concepts. For example:

```
def f(x, y):
    x = 23
    y.append(42)
a = 77
b = [99]
f(a, b)
print(a, b)              # prints: 77 [99, 42]
```

`print` shows that *a* is still bound to `77`. Function `f`'s rebinding of its parameter `x` to `23` has no effect on `f`'s caller, nor, in particular, on the binding of the caller's variable that happened to be used to pass `77` as the parameter's value. However, `print` also shows that `b` is now bound to `[99, 42]`. `b` is still bound to the same list object as before the call, but `f` has appended `42` to that list object, mutating it. In neither case has `f` has altered the caller's bindings, nor can `f` alter the number `77`, since numbers are immutable. However, `f` can alter a list object, since list objects are mutable.

Namespaces

A function's parameters, plus any names that are bound (by assignment or by other binding statements, such as `def`) in the function body, make up the function's *local namespace*, also known as its *local scope*. Each of these variables is known as a *local variable* of the function.

Variables that are not local are known as *global variables* (in the absence of nested function definitions, which we'll discuss shortly). Global variables are attributes of the module object, as covered in "Attributes of module objects". Whenever a function's local variable has the same name as a global variable, that name, within the function body, refers to the local variable, not the global one. We express this by saying that the local variable *hides* the global variable of the same name throughout the function body.

The global statement

By default, any variable that is bound within a function body is a local variable of the function. If a function needs to bind or rebind some global variables (*not* a good practice!), the first statement of the function's body must be:

```
global identifiers
```

where *identifiers* is one or more identifiers separated by commas (`,`). The identifiers listed in a `global` statement refer to the global variables (i.e., attributes of the module object) that the function needs to bind or rebind. For example, the function `counter` that we saw in "Other attributes of function objects" could be implemented using `global` and a global variable, rather than an attribute of the function object:

```
_count = 0
def counter():
    global _count
    _count += 1
    return _count
```

Without the `global` statement, the `counter` function would raise an `UnboundLocalError` exception when called, because `_count` would then be an uninitialized (unbound) local variable. While the `global` statement enables this kind of programming, this style is inelegant and ill-advised. As we mentioned earlier, when you want to group together some state and some behavior, the object-oriented mechanisms covered in "Object-oriented Python" are usually best.

Eschew global

Never use `global` if the function body just *uses* a global variable (including mutating the object bound to that variable, when the object is mutable). Use a `global` statement only if the function body *rebinds* a global variable (generally by assigning to the variable's name). As a matter of style, don't use `global` unless it's strictly necessary, as its presence causes readers of your program to assume the statement is there for some useful purpose. Never use `global` except as the first statement in a function body.

Nested functions and nested scopes

A `def` statement within a function body defines a *nested function*, and the function whose body includes the `def` is known as an *outer function* to the nested one. Code in a nested function's body may access (but *not* rebind) local variables of an outer function, also known as *free variables* of the nested function.

The simplest way to let a nested function access a value is often not to rely on nested scopes, but rather to pass that value explicitly as one of the function's arguments. If need be, you can bind the argument's value at nested-function `def` time: just use the value as the default for an optional argument. For example:

```
def percent1(a, b, c):
    def pc(x, total=a+b+c):
        return (x*100.0) / total
    print('Percentages are:', pc(a), pc(b), pc(c))
```

Here's the same functionality using nested scopes:

```
def percent2(a, b, c):
    def pc(x):
        return (x*100.0) / (a+b+c)
    print('Percentages are:', pc(a), pc(b), pc(c))
```

In this specific case, `percent1` has one tiny advantage: the computation of *a*+*b*+*c* happens only once, while `percent2`'s inner function `pc` repeats the computation three times. However, when the outer function rebinds local variables between calls to the nested function, repeating the computation can be necessary: be aware of both approaches, and choose the appropriate one case by case.

A nested function that accesses values from outer local variables is also known as a *closure*. The following example shows how to build a closure:

```
def make_adder(augend):
    def add(addend):
        return addend+augend
    return add
```

Closures are sometimes an exception to the general rule that the object-oriented mechanisms covered in Chapter "Object-Oriented Python" are the best way to bundle together data and code. When you need specifically to construct callable objects, with some parameters fixed at object-construction time, closures can be simpler and more effective than classes. For example, the result of `make_adder(7)` is a function that accepts a single argument and returns `7` plus that argument. An outer function that returns a closure is a "factory" for members of a family of functions distinguished by some parameters, such as the value of argument *augend* in the previous example, and may often help you avoid code duplication.

The `nonlocal` keyword acts similarly to `global`, but it refers to a name in the namespace of some lexically surrounding function. When it occurs in a function definition nested several levels deep (a rarely-needed structure!), the compiler searches the namespace of the most deeply nested containing function, then the function containing that one, and so on, until the name is

found or there are no further containing functions, in which case the compiler raises an error.

Here's a nested-functions approach to the "counter" functionality we implemented in previous sections using a function attribute, then a global variable:

```
def make_counter():
    count = 0
    def counter():
        nonlocal count
        count += 1
        return count
    return counter
c1 = make_counter()
c2 = make_counter()
print(c1(), c1(), c1())     # prints: 1 2 3
print(c2(), c2())           # prints: 1 2
print(c1(), c2(), c1())     # prints: 4 3 5
```

A key advantage of this approach versus the previous ones is that these two nested functions, just like an OOP approach would, let you make independent counters, here *c1* and *c2*—each closure keeps its own state and doesn't interfere with the other one. This approach, and OOP, are both quite acceptable.

lambda expressions

If a function body is a single `return` *expression* statement, you may (**very** optionally!) choose to replace the function with the special `lambda` expression form:

```
lambda parameters: expression
```

A `lambda` expression is the anonymous equivalent of a normal function whose body is a single `return` statement. Note that the `lambda` syntax does not use the `return` keyword. You can use a `lambda` expression wherever you could use a reference to a function. `lambda` can sometimes be handy when you want to use an *extremely simple* function as an

argument or return value. Here's an example that uses a `lambda` expression as an argument to the built-in `filter` function (covered in Table 7-2):

```
a_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
low = 3
high = 7
list(filter(lambda x: low<=x<high, a_list))    # returns: [3, 4,
5, 6]
```

Alternatively, you can always use a local `def` statement to give the function object a name, then use this name as an argument or return value. Here's the same `filter` example using a local `def` statement:

```
a_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
def within_bounds(value, low=3, high=7):
    return low<=value><high
filter(within_bounds, a_list)           # returns: [3, 4, 5, 6]
```

While `lambda` can at times be handy, `def` is usually better: it's more general, and makes the code more readable, since you can choose a clear name for the function.

Generators

When the body of a function contains one or more occurrences of the keyword `yield`, the function is known as a *generator*, or more precisely a *generator function*. When you call a generator, the function body does not execute. Instead, the generator function returns a special iterator object, known as a *generator object* (sometimes, quite confusingly, also called just "a generator"), wrapping the function body, its local variables (including parameters), and the current point of execution, initially the start of the function.

When you (implicitly or explicitly) call `next` on a generator object, the function body executes from the current point up to the next `yield`, which takes the form:

```
yield expression
```

A bare `yield` without the expression is also legal, and equivalent to `yield None`. When `yield` executes, the function execution is "frozen," preserving current point of execution and local variables, and the expression following `yield` becomes the result of `next`. When you call `next` again, execution of the function body resumes where it left off, again up to the next `yield`. When the function body ends, or executes a `return` statement, the iterator raises a `StopIteration` exception to indicate that the iteration is finished. The expression after `return`, if any, is the argument to the `StopIteration`.

`yield` is an expression, not a statement. When you call *g*`.send(`*value*`)` on a generator object *g*, the value of the `yield` is *value*; when you call `next(`*g*`)`, the value of the `yield` is `None`. We cover this in "Generators as near-coroutines": it's the elementary building block to implement coroutines in Python.

A generator function is often a handy way to build an iterator. Since the most common way to use an iterator is to loop on it with a `for` statement, you typically call a generator like this (with the call to `next` being implicit in the `for` statement):

```
for avariable in somegenerator(arguments):
```

For example, say that you want a sequence of numbers counting up from `1` to *N* and then down to `1` again. A generator can help:

```
def updown(N):
    for x in range(1, N):
        yield x
    for x in range(N, 0, -1):
        yield x
for i in updown(3):
    print(i)              # prints: 1 2 3 2 1
```

Here is a generator that works somewhat like built-in `range`, but returns an iterator on floating-point values rather than on integers:

```
def frange(start, stop, stride=1.0):
    while start < stop:
        yield start
        start += stride
```

This `frange` example is only *somewhat* like `range` because, for simplicity, it makes arguments `start` and `stop` mandatory, and assumes that `stride` is positive.

Generators are more flexible than functions that return lists. A generator may return an *unbounded* iterator, meaning one that yields an infinite stream of results (to use only in loops that terminate by other means, e.g., via a conditionally-executed `break` statement). Further, a generator-object iterator performs *lazy evaluation*: the iterator can compute each successive item only when and if needed, "just in time", while the equivalent function does all computations in advance and may require large amounts of memory to hold the results list. Therefore, if all you need is the ability to iterate on a computed sequence, it is usually best to compute the sequence in a generator, rather than in a function returning a list. If the caller needs a list of all the items produced by some bounded generator *g*(*arguments*), the caller can simply use the following code to explicitly request Python to build a list:

```
resulting_list = list(g(arguments))
```

yield from

To improve execution efficiency and clarity when multiple levels of iteration are yielding values, you can use the form `yield from` *expression*, where *expression* is iterable. This yields the values from *expression* one at a time into the calling environment, avoiding the need to `yield` repeatedly. We can thus simplify the `updown` generator we defined earlier:

```
def updown(N):
    yield from range(1, N)
    yield from range(N, 0, -1)
```

```
for i in updown(3):
    print(i)                    # prints: 1 2 3 2 1
```

Moreover, using `yield from` lets you use generators as full-fledged *coroutines*, as covered in Chapter "Multitasking".

Generator expressions

Python offers an even simpler way to code particularly simple generators: *generator expressions*, commonly known as *genexps*. The syntax of a genexp is just like that of a list comprehension (as covered in "List comprehensions"), except that a genexp is within parentheses (`()`) instead of brackets (`[]`). The semantics of a genexp are the same as those of the corresponding list comprehension, except that a genexp produces an iterator yielding one item at a time, while a list comprehension produces a list of all results in memory (therefore, using a genexp, when appropriate, saves memory). For example, to sum the squares of all single-digit integers, you could code `sum([x*x for x in range(10)])`; however, you can express this better as `sum(x*x for x in range(10))` (just the same, but omitting the brackets): you get just the same result but consume less memory. The parentheses that indicate the function call also "do double duty" and enclose the genexp. Parentheses are, however, required when the genexp is not the sole argument. Additional parentheses don't hurt, but are usually best omitted, for clarity.

Generators as near-coroutines

Generators are further enhanced, with the possibility of receiving a value (or an exception) back from the caller as each `yield` executes. This lets generators implement coroutines, as explained in PEP 342. When a generator resumes (i.e., you call `next` on it), the corresponding `yield`'s value is `None`. To pass a value *x* into some generator *g* (so that *g* receives *x* as the value of the `yield` on which it's suspended), instead of calling `next(g)`, call *g*`.send(`*x*`)` (*g*`.send(None)` is just like `next(g)`).

Other enhancements to generators regard exceptions: we cover them in "Generators and Exceptions".

Recursion

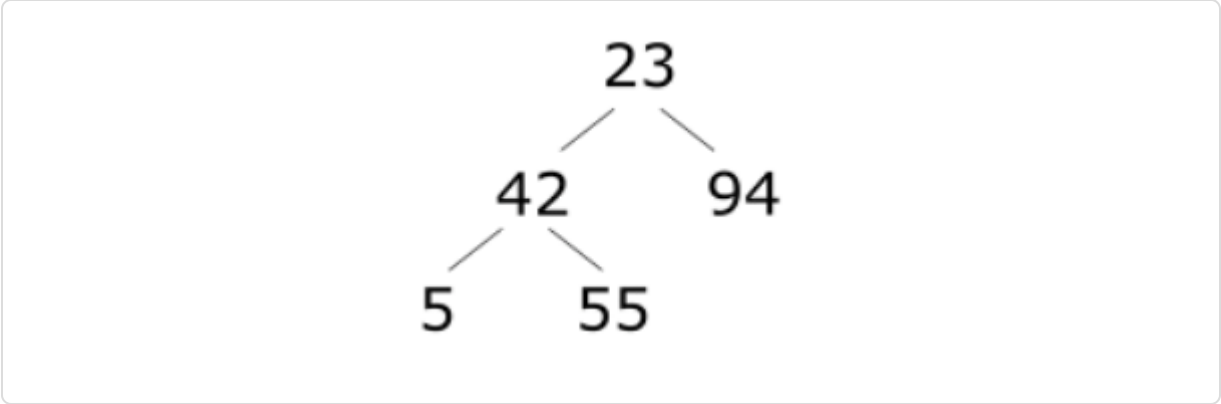
Python supports recursion (i.e., a Python function can call itself, directly or indirectly), but there is a limit to how deep the recursion can go. By default, Python interrupts recursion and raises a `RecursionLimitExceeded` exception (covered in "Standard Exception Classes") when it detects that recursion has exceeded a depth of 1,000. You can change this default recursion limit by calling `setrecursionlimit` in module `sys`, covered in Table 7-3.

However, changing the recursion limit does not give you unlimited recursion; the absolute maximum limit depends on the platform on which your program is running, particularly on the underlying operating system and C runtime library, but it's typically a few thousand levels. If recursive calls get too deep, your program crashes. Such runaway recursion, after a call to `setrecursionlimit` that exceeds the platform's capabilities, is one of the few ways a Python program can crash—really crash, hard, without the usual safety net of Python's exception mechanism. Therefore, beware "fixing" a program that is getting `RecursionLimitExceeded` exceptions by raising the recursion limit with `setrecursionlimit`. While it's a valid technique, most often you're better advised to look for ways to remove the recursion, unless you are confident you've been able to limit the depth of recursion that your program needs.

Readers who are familiar with Lisp, Scheme, or functional programming languages, must in particular be aware that Python does *not* implement the optimization of "tail-call elimination," which is so crucial in those languages. In Python, any call, recursive or not, has the same "cost" in terms of both time and memory space, dependent only on the number of arguments: the cost does not change, whether the call is a "tail-call" (meaning that the call is the last operation that the caller executes) or any other, non-tail call. This makes recursion removal even more important.

For example, consider a classic use for recursion: "walking a binary tree." Suppose you represent a binary tree structure as nodes, where each node is a three-element (payload, left, right) tuple where left and right are either similar tuples or None representing the left-side and right-side descendants

respectively. A simple example might be: (23, (42, (5, None, None), (55, None, None)), (94, None,None)) to represent the tree shown here.



To write a generator function that, given the root of such a tree, "walks the tree," yielding each payload in top-down order, the simplest approach is recursion:

```
def rec(t):
    yield t[0]
    for i in (1, 2):
        if t[i] is not None:
            yield from rec(t[i])
```

However, if a tree is very deep, recursion becomes a problem. To remove recursion, we can handle our own stack—a list used in last-in, first-out fashion, thanks to its `append` and `pop` methods. To wit:

```
def norec(t):
    stack = [t]
    while stack:
        t = stack.pop()
        yield t[0]
        for i in (2, 1):
            if t[i] is not None:
                stack.append(t[i])
```

The only small issue to be careful about, to keep exactly the same order of `yields` as `rec`, is switching the `(1, 2)` index order in which to examine descendants, to `(2, 1)`, adjusting to the "reversed" (last-in, first-out) behavior of *stack*.

1 Control characters include nonprinting characters such as \t (tab) and \n (newline), both of which count as whitespace; and others such as \a (alarm, AKA "beep") and \b (backspace), which are not whitespace.

2 Also see **bytearray**, covered later, for a **bytes**-like "string" which, however, is mutable.

3 Each specific mapping type may put constraints on the type of keys it accepts: in particular, dictionaries only accept hashable keys.

4 This is not, strictly speaking, the "coercion" you observe in other languages, but, among builtin number types, it produces pretty much the same effect.

5 Note that the second item of `divmod's` result, just like the result of `%`, is the *remainder*, **not** the *modulo*, despite the function's misleading name. The difference matters when the divisor is negative. In some other languages, such as C# and Javascript, the result of a `%` operator **is** the modulo; in others yet, such as C and C++, it's machine-dependent whether the result is the modulo or the remainder, when **either** operand is negative.

6 It is notable that the match statement specifically excludes matching values of type str, bytes, and bytearray with *sequence* patterns.

7 Indeed, the syntax notation used in the Python online documentation required, and got, updates to concisely describe some of Python's more recent syntax additions.

8 for this unique use-case, it's common to break the normal style conventions about making class names have an uppercase initial and avoiding semicolons to stash multiple assignments within one line, although the authors haven't yet found a style-guide that blesses this peculiar, rather-new usage.

9 in that paper, Knuth also first proposed using "devices like indentation, rather than delimiters" to express program structure—just as Python does!

10 "alas!" because they have nothing to do with Python keywords, so the terminology is confusing.

11 Python developers introduced positional-only arguments when they realised that parameters to many built-in functions effectively had no valid names as far as the interpreter was concerned.

12 an "optional parameter" being one for which the function's signature supplies a default value.

Chapter 3. Exceptions

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

Python uses exceptions to indicate errors and anomalies. An *exception* is an object that indicates an error or anomaly. When Python detects an error, it *raises* an exception—that is, Python signals the occurrence of an anomalous condition by passing an exception object to the exception-propagation mechanism. Your code can explicitly raise an exception by executing a `raise` statement.

Handling an exception means catching the exception object from the propagation mechanism and taking actions as needed to deal with the anomalous situation. If a program does not handle an exception, the program terminates with an error traceback message. However, a program can handle exceptions and keep running, despite errors or other anomalies, by using the `try` statement with `except` clauses.

Python also uses exceptions to indicate some situations that are not errors, and not even abnormal. For example, as covered in "Iterators", calling the `next` built-in on an iterator raises `StopIteration` when the iterator has no more items. This is not an error; it is not even an anomaly, since most iterators run out of items eventually. The optimal strategies for checking and handling errors and other special situations in Python are therefore different from other languages', and we cover that in "Error-Checking Strategies". This chapter shows how to use exceptions for errors and special

situations. It also covers the `logging` module of the standard library, in "Logging Errors", and the `assert` statement, in "The assert Statement".

The try Statement

The `try` statement is Python's core exception-handling mechanism. It's a compound statement with three kinds of optional clauses:

- it may have zero or more `except` clauses, defining how to handle particular classes of exceptions
- if it has `except` clauses, then it may also have, right afterwards, one `else` clause, executed only if the `try` suite raised no exceptions, and
- whether or not it has `except` clauses, it may have a single `finally` clause, unconditionally executed, with behavior covered in "The try/except/finally Statement".

Python's syntax requires the presence of at least one `except` clause or a `finally` clause, both of which might also be present in the same statement; `else` is only valid following one or more `except`s.

try/except

Here's the syntax for the `try/except` form of the `try` statement:

```
try:
    statement(s)
except [expression [as target]]:
    statement(s)
[else:
    statement(s)]
[finally:
    statement(s)]
```

This form of the `try` statement has one or more `except` clauses, as well as an optional `else` clause (and an optional `finally` clause, whose

meaning does not depend on whether `except` and `else` clauses are present: we cover it in the "try/finally" section below).

The body of each `except` clause is known as an *exception handler*. The code executes when the *expression* in the `except` clause matches an exception object propagating from the `try` clause. *expression* is a class (or tuple of classes, in parentheses), and matches any instance of one of those classes or their subclasses. The optional *target* is an identifier that names a variable that Python binds to the exception object just before the exception handler executes. A handler can also obtain the current exception object by calling the `exc_info` function of module `sys` (covered in Table 7-3).

Here is an example of the `try/except` form of the `try` statement:

```
try:
    1/0
    print('not executed')
except ZeroDivisionError:
    print('caught divide-by-0 attempt')
```

When an exception is raised, execution of the `try` suite immediately ceases. If a `try` statement has several `except` clauses, the exception-propagation mechanism checks the `except` clauses in order; the first `except` clause whose expression matches the exception object executes as the handler, and the exception-propagation mechanism checks no further except clauses after that.

Place Handlers For Specific Exceptions Before More General Ones

Place handlers for specific cases before handlers for more general cases: when you place a general case first, the more specific `except` clauses that follow never execute.

The last `except` clause need not specify an expression. An `except` clause without any expression handles any exception that reaches it during propagation. Such unconditional handling is rare, but it does occur, often in "wrapper" functions that must perform some extra task before reraising an exception, as we discuss in "The raise Statement" later in this chapter.

Avoid A "Bare Except" That Doesn't Re-raise The Exception

Beware of using a "bare `except`" (an `except` clause without an expression) unless you're re-raising the exception in it: such sloppy style can make bugs very hard to find, since the bare `except` is over-broad and can easily mask coding errors and other kinds of bugs by allowing execution to continue after an unanticipated exception.

New programmers who are "just trying to get things to work" may even write code like:

```
try:
    ...code that has a problem...
except:
    pass
```

This is a dangerous practice, since it catches important process-exiting exceptions such as `KeyboardInterrupt` or `SystemExit` - a loop with such an exception handler can't be exited with Ctrl-C, nor terminated with a system kill command. At the very least, such code should use `except Exception:`, which is still overly broad but at least does not catch the process-exiting exceptions.

Exception propagation terminates when it finds a handler whose expression matches the exception object. When a `try` statement is nested (lexically in the source code, or dynamically within function calls) in the `try` clause of another `try` statement, a handler established by the inner `try` is reached first on propagation, so it handles the exception when it matches it. This may not be what you want. For example:

```
try:
    try:
        1/0
```

```
    except:
        print('caught an exception')
except ZeroDivisionError:
    print('caught divide-by-0 attempt')
# prints: caught an exception
```

In this case, it does not matter that the handler established by the clause `except ZeroDivisionError:` in the outer `try` clause is more specific than the catch-all `except:` in the inner `try` clause. The outer `try` does not enter into the picture: the exception doesn't propagate out of the inner `try`. For more on exception propagation, see "Exception Propagation".

The optional `else` clause of `try/except` executes only when the `try` clause terminates normally. In other words, the `else` clause does not execute when an exception propagates from the `try` clause, or when the `try` clause exits with a `break`, `continue`, or `return`. Handlers established by `try/except` cover only the `try` clause, not the `else` clause. The `else` clause is useful to avoid accidentally handling unexpected exceptions. For example:

```
print(repr(value), 'is ', end=' ')
try:
    value + 0
except TypeError:
    # not a number, maybe a string...?
    try:
        value + ''
    except TypeError:
        print('neither a number nor a string')
    else:
        print('some kind of string')
else:
    print('some kind of number')
```

try/finally

Here's the syntax for the `try/finally` form of the `try` statement:

```
try:
    statement(s)
```

```
finally:
    statement(s)
```

This form has 1 `finally` clause (and no `else` clause—unless it also has 1+ `except` clauses, as covered in "The try/except/finally Statement").

The `finally` clause establishes what is known as a *clean-up handler*. The code always executes after the `try` clause terminates in any way. When an exception propagates from the `try` clause, the `try` clause terminates, the clean-up handler executes, and the exception keeps propagating. When no exception occurs, the cleanup handler executes anyway, regardless of whether the `try` clause reaches its end or exits by executing a `break`, `continue`, or `return` statement.

Clean-up handlers established with `try/finally` offer a robust and explicit way to specify finalization code that must always execute, no matter what, to ensure consistency of program state and/or external entities (e.g., files, databases, network connections); such assured finalization is nowadays usually best expressed via a *context manager* used in a `with` statement (see "The with Statement and Context Managers"). Here is an example of the `try/finally` form of the `try` statement:

```
f = open(some_file, 'w')
try:
    do_something_with_file(f)
finally:
    f.close()
```

and here is the corresponding, more concise and readable, example of using `with` for exactly the same purpose:

```
with open(some_file, 'w') as f:
    do_something_with_file(f)
```

Avoid break and return statements in a finally clause

A `finally` clause may contain one or more of the statements `continue` ||3.8++||, `break` , or `return`. Such usage may make your program less clear: exception propagation stops when such a statement executes, and most programmers would not expect propagation to be stopped within a `finally` clause. The usage may confuse people who are reading your code, so we recommend you avoid it.

The try/except/finally Statement

A `try/except/finally` statement, such as:

```
try:
    ...guarded clause…
except ...expression...:
    ...exception handler code…
finally:
    ...clean-up code...
```

is equivalent to the nested statement:

```
try:
    try:
        ...guarded clause...
    except ...expression...:
        ...exception handler code…
finally:
    ...clean-up code...
```

A `try` statement can have multiple `except` clauses, and optionally an `else` clause, before a terminating `finally` clause. In all variations, the effect is always as just shown—that is, just like nesting a `try/except` statement, with all the `except` clauses and the `else` clause if any, into a containing `try/finally` statement.

The with Statement and Context Managers

The `with` statement is a compound statement with the following syntax:

```
with expression [as varname] [, ...]:
    statement(s)
||3.10++||
with (expression [as varname], ...):
    statement(s)
```

The semantics of `with` are equivalent to:

```
_normal_exit = True
_manager = expression
varname = _manager.__enter__()
try:
    statement(s)
except:
    _normal_exit = False
    if not _manager.__exit_(*sys.exc_info()):
        raise
    # note that exception does not propagate if __exit__ returns
a true value
finally:
    if _normal_exit:
        _manager.__exit__(None, None, None)
```

where *_manager* and *_normal_exit* are arbitrary internal names that are not used elsewhere in the current scope. If you omit the optional `as` *varname* part of the `with` clause, Python still calls *_manager*`.__enter__()`, but doesn't bind the result to any name, and still calls *_manager*`.__exit_()` at block termination. The object returned by the *expression*, with methods `__enter__` and `__exit__`, is known as a *context manager*.

The `with` statement is the Python embodiment of the well-known C++ idiom "resource acquisition is initialization" (RAII): you need only write context manager classes—that is, classes with two special methods `__enter__` and `__exit__`. `__enter__` must be callable without arguments. `__exit__` must be callable with three arguments: all *None* when the body completes without propagating exceptions; otherwise, the type, value, and traceback of the exception. This provides the same guaranteed finalization behavior as typical *ctor/dtor* pairs have for `auto`

variables in C++, and `try/finally` statements have in Python or Java. In addition, they can finalize differently depending on what exception, if any, propagates, and optionally block a propagating exception by returning a true value from `__exit__`.

For example, here is a simple, purely illustrative way to ensure `<name>` and `</name>` tags are printed around some other output:

```
class tag(object):
    def __init__(self, tagname):
        self.tagname = tagname
    def __enter__(self):
        print(f'<{self.tagname}>', end='')
    def __exit__(self, etyp, einst, etb):
        print(f'</{self.tagname}>')
# to be used as:
tt = tag('sometag')
with tt:
    ...statements printing output to be enclosed in
       a matched open/close `sometag` pair
```

A simpler way to build context managers is to use the `contextmanager` decorator in the `contextlib` module of the standard Python library. This decorator turns a generator function into a factory of context manager objects.

The `contextlib` way to implement the `tag` context manager, having imported `contextlib` earlier, is:

```
@contextlib.contextmanager
def tag(tagname):
    print(f'<{tagname}>', end='')
    try:
        yield
    finally:
        print(f'</{tagname}>')
# to be used the same way as before
```

`contextlib` supplies, among others, the class and functions listed in Table 5-1.

Table 3-1. The contextlib module summarised

AbstractContextManager	`AbstractContextManager` Abstract base class with two overridable methods: `__enter__`, which defaults to `return self`, and `__exit__`, which defaults to `return None`.
contextmanager	`contextmanager` The above-described decorator, which you apply to a generator to make it into a context manager.
closing	`closing(`*something*`)` A context manager whose `__enter__` is `return` *something*, and whose `__exit__` calls *something*`.close()`.
nullcontext	`nullcontext(`*something*`)` A context manager whose `__enter__` is `return` *something*, and whose `__exit__` does nothing.
redirect_stderr	`redirect_stderr(`*destination*`)` A context manager which temporarily redirects, within the body of the `with` statement, `sys.stderr` to file or file-like object *destination*.
redirect_stdout	`redirect_stdout(`*destination*`)` A context manager which temporarily redirects, within the body of the `with` statement, `sys.stdout` to file or file-like object *destination*.
suppress	`suppress(`**exception_classes*`)` A context manager which silently suppresses exceptions, occurring in the body of the `with` statement, of any of the classes listed in *exception_classes*. Use sparingly, since silently suppressing exceptions is often bad practice.

For more details, examples, “recipes”, and even more (somewhat abstruse) classes, see Python’s online docs.

Generators and Exceptions

To help generators cooperate with exceptions, `yield` statements are allowed inside `try/finally` statements. Moreover, generator objects have two other relevant methods, `throw` and `close`. Given a generator object *g*, built by calling a generator function, the `throw` method's signature is:

```
g.throw(exc_value)
```

When the generator's caller calls *g*`.throw`, the effect is just as if a `raise` statement with the same argument executed at the spot of the `yield` at which generator *g* is suspended.

The generator method `close` has no arguments; when the generator's caller calls *g*`.close()`, the effect is like calling *g*`.throw(GeneratorExit())`[1]. `GeneratorExit` is a built-in exception class that inherits directly from `BaseException`. Generators also have a finalizer (special method `__del__`) which implicitly calls `close` when the generator object is garbage-collected.

If a generator raises or propagates `StopIteration`, Python turns the exception's type into `RuntimeError`.

Exception Propagation

When an exception is raised, the exception-propagation mechanism takes control. The normal control flow of the program stops, and Python looks for a suitable exception handler. Python's `try` statement establishes exception handlers via its `except` clauses. The handlers deal with exceptions raised in the body of the `try` clause, as well as exceptions propagating from functions called by that code, directly or indirectly. If an exception is raised within a `try` clause that has an applicable `except` handler, the `try` clause terminates and the handler executes. When the handler finishes, execution continues with the statement after the `try` statement (in the absence of any explicit change to the flow of control such as a `raise` or `return`).

If the statement raising the exception is not within a `try` clause that has an applicable handler, the function containing the statement terminates, and the exception propagates "upward" along the stack of function calls to the statement that called the function. If the call to the terminated function is within a `try` clause that has an applicable handler, that `try` clause terminates, and the handler executes. Otherwise, the function containing the call terminates, and the propagation process repeats, *unwinding* the stack of function calls until an applicable handler is found.

If Python cannot find any applicable handler, by default the program prints an error message to the standard error stream (file *sys.stderr*). The error message includes a traceback that gives details about functions terminated during propagation. You can change Python's default error-reporting behavior by setting `sys.excepthook` (covered in Table 7-3). After error reporting, Python goes back to the interactive session, if any, or terminates if execution was not interactive. When the exception type is `SystemExit`, termination is silent, and ends the interactive session, if any.

Here are some functions to show exception propagation at work:

```
def f():
    print('in f, before 1/0')
    1/0    # raises a ZeroDivisionError exception
    print('in f, after 1/0')
def g():
    print('in g, before f()')
    f()
    print('in g, after f()')
def h():
    print('in h, before g()')
    try:
        g()
        print('in h, after g()')
    except ZeroDivisionError:
        print('ZD exception caught')
    print('function h ends')
```

Calling the `h` function prints the following:

```
>>> h()
in h, before g()
```

```
in g, before f()
in f, before 1/0
ZD exception caught
function h ends
```

That is, none of the "after" `print` statements execute, since the flow of exception propagation "cuts them off."

The function *h* establishes a `try` statement and calls the function *g* within the `try` clause. *g*, in turn, calls *f*, which performs a division by `0`, raising an exception of type `ZeroDivisionError`. The exception propagates all the way back to the `except` clause in *h*. The functions *f* and *g* terminate during the exception-propagation phase, which is why neither of their "after" messages is printed. The execution of *h*'s `try` clause also terminates during the exception-propagation phase, so its "after" message isn't printed either. Execution continues after the handler, at the end of *h*'s `try/except` block.

The raise Statement

You can use the `raise` statement to raise an exception explicitly. `raise` is a simple statement with the following syntax:

```
raise [expression]
```

Only an exception handler (or a function that a handler calls, directly or indirectly) can use `raise` without any expression. A plain `raise` statement re-raises the same exception object that the handler received. The handler terminates, and the exception propagation mechanism keeps going up the call stack, searching for other applicable handlers. Using `raise` without any expression is useful when a handler discovers that it is unable to handle an exception it receives, or can handle the exception only partially, so the exception should keep propagating to allow handlers up the call stack to perform their own handling and cleanup.

When *expression* is present, it must be an instance of a class inheriting from the built-in class `BaseException`, and Python raises that instance.

Here's an example of a typical use of the `raise` statement:

```
def cross_product(seq1, seq2):
    if not seq1 or not seq2:
        raise ValueError('Sequence arguments must be non-empty')
    return [(x1, x2) for x1 in seq1 for x2 in seq2]
```

This *cross_product* example function returns a list of all pairs with one item from each of its sequence arguments, but first, it tests both arguments. If either argument is empty, the function raises `ValueError` rather than just returning an empty list as the list comprehension would normally do.

Check Only What You Need To

There is no need for *cross_product* to check whether *seq1* and *seq2* are iterable: if either isn't, the list comprehension itself raises the appropriate exception, presumably a `TypeError`.

Once an exception is raised, by Python itself or with an explicit `raise` statement in your code, it is up to the caller to either handle it (with a suitable `try/except` statement) or let it propagate further up the call stack.

Don't Use raise for Duplicate, Redundant Error Checks

Use the `raise` statement only to raise additional exceptions for cases that would normally be okay but that your specification defines to be errors. Do not use `raise` to duplicate the same error-checking that Python already, implicitly, does on your behalf.

Exception Objects

Exceptions are instances of `BaseException` (more specifically, instances of one of its subclasses). Any exception has attribute `args`, the tuple of arguments used to create the instance; this error-specific information is useful for diagnostic or recovery purposes. Some exception classes interpret `args` and set convenient named attributes on the classes' instances.

The Hierarchy of Standard Exceptions

Exceptions are instances of subclasses of `BaseException`.

The inheritance structure of exception classes is important, as it determines which `except` clauses handle which exceptions. Most exception classes extend the class `Exception`; however, the classes `KeyboardInterrupt`, `GeneratorExit`, and `SystemExit` inherit directly from `BaseException` and are not subclasses of `Exception`. Thus, a handler clause `except Exception as e:` does not catch `KeyboardInterrupt`, `GeneratorExit`, or `SystemExit` (we cover exception handlers in "try/except"). Instances of `SystemExit` are normally raised via the `exit` function in module `sys` (covered in Table 7-3). We cover `GeneratorExit` in "Generators and Exceptions". When the user hits Ctrl-C, Ctrl-Break, or other interrupting keys on their keyboard, that raises `KeyboardInterrupt`.

The hierarchy of built-in expression classes is, roughly:

```
BaseException
  Exception
    AssertionError, AttributeError, BufferError, EOFError,
    MemoryError, ReferenceError, OsError, StopAsyncIteration,
    StopIteration, SystemError, TypeError
    ArithmeticError
      OverflowError, ZeroDivisionError
    ImportError
      ModuleNotFoundError, ZipImportError
    LookupError
      IndexError, KeyError
```

```
    NameError
      UnboundLocalError
    OSError
      ...
    RuntimeError
      RecursionError
      NotImplementedError
    SyntaxError
      IndentationError
        TabError
    ValueError
      UnsupportedOperation
      UnicodeError
        UnicodeDecodeError, UnicodeEncodeError,
        UnicodeTranslateError
    Warning
      ...
  GeneratorExit
  KeyboardInterrupt
  SystemExit
```

There are other exception subclasses (in particular, `Warning` and `OSError` have many, summarized above with ellipses **...**), but this is the gist of the hierarchy. A more complete list is in Python's online docs.

Two subclasses of `Exception` are abstract ones, never instantiated directly. Their purpose is to make it easier for you to specify `except` clauses that handle a range of related errors. The two abstract subclasses of `Exception` are:

`ArithmeticError`

> The base class for exceptions due to arithmetic errors (i.e., `OverflowError`, `ZeroDivisionError`, and the currently-unused `FloatingPointError`)

`LookupError`

> The base class for exceptions that a container raises when it receives an invalid key or index (i.e., `IndexError`, `KeyError`)

Standard Exception Classes

Common runtime errors raise exceptions of the following classes:

`AssertionError`

An assert statement failed.

`AttributeError`

An attribute reference or assignment failed.

`ImportError`

An `import` or `from...import` statement (covered in "The import Statement") couldn't find the module to import (in this case, what Python raises is actually an instance of `ImportError`'s subclass `ModuleNotFoundError`), or couldn't find a name to be imported from the module.

`IndentationError`

The parser encountered a syntax error due to incorrect indentation. Extends `SyntaxError`.

`IndexError`

An integer used to index a sequence is out of range (using a noninteger as a sequence index raises `TypeError`). Extends `LookupError`.

`KeyError`

A key used to index a mapping is not in the mapping. Extends `LookupError`.

`KeyboardInterrupt`

The user pressed the interrupt key combination (Ctrl-C, Ctrl-Break, Delete, or others, depending on the platform's handling of the keyboard).

`MemoryError`

An operation ran out of memory.

`NameError`

A name was referenced, but it was not bound to any variable in the current scope.

`NotImplementedError`

Raised by abstract base classes to indicate that a concrete subclass must override a method.

`OSError`

Raised by functions in the module `os` (covered in "The os Module" and "Running Other Programs with the os Module") to indicate platform-dependent errors. It has many subclasses, covered at "OSError and subclasses".

`RecursionError`

Python detects that recursion depth has been exceeded. Extends `RuntimeError`.

`RuntimeError`

Raised for any error or anomaly not otherwise classified.

`SyntaxError`

Python's parser encounters a syntax error.

`SystemError`

Python has detected an error in its own code, or in an extension module. Please report this to the maintainers of your Python version, or of the

extension in question, including the error message, the exact Python version (`sys.version`), and, if possible, your program's source code.

`TypeError`

An operation or function was applied to an object of an inappropriate type.

`UnboundLocalError`

A reference was made to a local variable, but no value is currently bound to that local variable. Extends `NameError`.

`UnicodeError`

An error occurred while converting Unicode(i.e., an `str`) to a byte string, or vice versa. Extends `ValueError`.

`ValueError`

An operation or function was applied to an object that has a correct type but an inappropriate value, and nothing more specific (e.g., `KeyError`) applies.

`ZeroDivisionError`

A divisor (the righthand operand of a `/`, `//`, or `%` operator, or the second argument to the built-in function `divmod`) is `0`. Extends `ArithmeticError`.

OSError and subclasses

`OSError` represents errors detected by the operating system. To handle such errors much more elegantly, `OSError` has many subclasses, whose instances are what actually get raised—see Python's online docs.

For example, consider this task: try to read and return the contents of a certain file; return a default string if the file does not exist; propagate any

other exception that makes the file unreadable (except for the file not existing). Using an `OSError` subclass, you can accomplish the task quite simply:

```
def read_or_default(filepath, default):
    try:
        with open(filepath) as f:
            return f.read()
    except FileNotFoundError:
        return default
```

The `FileNotFoundError` subclass of `OSError` makes this kind of common task simple and direct to express in code.

Exceptions "wrapping" other exceptions or tracebacks

Sometimes, you cause an exception while trying to handle another. To let you clearly diagnose this issue, each exception instance holds its own traceback object; you can make another exception instance with a different traceback with the `with_traceback` method.

Moreover, Python automatically stores which exception it's handling as the `__context__` attribute of any further exception raised during the handling (unless you set the new exception's `__suppress_context__` attribute to true, which you do with the `raise...from` statement, which we cover shortly). If the new exception propagates, Python's error message uses that exception's `__context__` attribute to show details of the problem. For example, take the (deliberately!) broken code:

```
try: 1/0
except ZeroDivisionError:
    1+'x'
```

The error displayed is:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
During handling of the above exception, another exception
occurred:
```

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Thus, Python clearly displays both exceptions, the original and the intervening one.

To get more control over the error display, you can, if you wish, use the `raise...from` statement: when you execute `raise` *e* `from` *ex*, both *e* and *ex* are exception objects: *e* is the one that propagates, and *ex* is its "cause;" Python records *ex* as the value of *e.__cause__*, and sets *e.*`__suppress_context__` to true. (Alternatively, *ex* can be **None**: then, Python sets *e.__cause__* to **None**, but still sets *e.*`__suppress_context__` to true, and thus leaves *e.*`__context__` alone). For all details and motivations, see PEP 3134.

Custom Exception Classes

You can extend any of the standard exception classes in order to define your own exception class. Often, such a subclass adds nothing more than a docstring:

```
class InvalidAttribute(AttributeError):
    """Used to indicate attributes that could never be valid"""
```

An Empty Class Or Function Should Have A Docstring, Not `pass`

As covered in "The pass Statement", you don't need a `pass` statement to make up the body of a class. The docstring (which you should always write, to document the class's purpose if nothing else!) is enough to keep Python happy. Best practice for all "empty" classes (regardless of whether they are exception classes), just like for all "empty" functions, is to always have a docstring and no `pass` statement.

Given the semantics of `try/except`, raising a custom exception class such as `InvalidAttribute` is almost the same as raising its standard exception superclass, `AttributeError`. Any `except` clause that can handle `AttributeError` can handle `InvalidAttribute` just as well. In addition, client code that knows about your `InvalidAttribute` custom exception class can handle it specifically, without having to handle all other cases of `AttributeError` when it is not prepared for those. For example:

```
class SomeFunkyClass:
    """much hypothetical functionality snipped"""
    def __getattr__(self, name):
        """only clarifies the kind of attribute error"""
        if name.startswith('_'):
            raise InvalidAttribute(f'Unknown private attribute
{name!r}')
        else:
            raise AttributeError(f'Unknown attribute {name!r}')
```

Now, client code can, if it so chooses, be more selective in its handlers. For example:

```
s = SomeFunkyClass()
try:
    value = getattr(s, thename)
except InvalidAttribute as err:
    warnings.warn(str(err), stacklevel=2)
    value = None
# other cases of AttributeError just propagate, as they're
unexpected
```

Define And Raise Custom Exception Classes

It's an excellent idea to define, and raise, custom exception classes in your modules, rather than plain standard exceptions: by using custom exception classes which extend standard ones, you make it easier for callers of your module's code to handle exceptions that come from your module separately from others, if they choose to.

Custom Exceptions and Multiple Inheritance

An effective approach to custom exceptions is to multiply-inherit exception classes from your module's special custom exception class and a standard exception class, as in the following snippet:

```
class CustomAttributeError(CustomException, AttributeError):
    """An AttributeError which is ALSO a CustomException."""
```

Now, when your code raises an instance of `CustomAttributeError`, that exception can be caught by calling code that's designed to catch all cases of `AttributeError` as well as by code that's designed to catch all exceptions raised only, specifically, by your module.

Use Multiple Inheritance For Custom Exceptions

Whenever you must decide whether to raise a specific standard exception, such as `AttributeError`, or a custom exception class you define in your module, consider this multiple-inheritance approach, which gives you the best of both worlds in such cases. Make sure you clearly document this aspect of your module, because the technique, although handy, is not widely used. Users of your module may not expect it unless you clearly and explicitly document what you are doing.

Other Exceptions Used in the Standard Library

Many modules in Python's standard library define their own exception classes, which are equivalent to the custom exception classes that your own modules can define. Typically, all functions in such standard library modules may raise exceptions of such classes, in addition to exceptions in the standard hierarchy covered in "Standard Exception Classes". We cover the main cases of such exception classes throughout the rest of this book, in chapters covering the standard library modules that supply and may raise them.

Error-Checking Strategies

Most programming languages that support exceptions raise exceptions only in rare cases. Python's emphasis is different. Python deems exceptions appropriate whenever they make a program simpler and more robust, even if that makes exceptions rather frequent.

LBYL Versus EAFP

A common idiom in other languages, sometimes known as "Look Before You Leap" (LBYL), is to check in advance, before attempting an operation, for anything that might make the operation invalid. This approach is not ideal for several reasons:

- The checks may diminish the readability and clarity of the common, mainstream cases where everything is okay.
- The work needed for checking purposes may duplicate a substantial part of the work done in the operation itself.
- The programmer might easily err by omitting a needed check.
- The situation might change between the moment when you perform the checks, and the moment when, later (even by a tiny fraction of a second!), you attempt the operation.

The preferred idiom in Python is to attempt the operation in a `try` clause and handle the exceptions that may result in one or more `except` clauses. This idiom is known as "it's Easier to Ask Forgiveness than Permission" (EAFP), a motto widely credited to Rear Admiral Grace Murray Hopper, co-inventor of COBOL. EAFP shares none of the defects of LBYL. Here is a function using the LBYL idiom:

```
def safe_divide_1(x, y):
    if y==0:
        print('Divide-by-0 attempt detected')
        return None
```

```
    else:
        return x/y
```

With LBYL, the checks come first, and the mainstream case is somewhat hidden at the end of the function. Here is the equivalent function using the EAFP idiom:

```
def safe_divide_2(x, y):
    try:
        return x/y
    except ZeroDivisionError:
        print('Divide-by-0 attempt detected')
        return None
```

With EAFP, the mainstream case is upfront in a `try` clause, and the anomalies are handled in the following `except` clause, making the whole function easier to read and understand.

Proper usage of EAFP

EAFP is a good error-handling strategy, but it is not a panacea. In particular, never cast too wide a net, catching errors that you did not expect and therefore did not mean to catch. The following is a typical case of such a risk (we cover built-in function `getattr` in Table 7-2):

```
def trycalling(obj, attrib, default, *args, **kwds):
    try:
        return getattr(obj, attrib)(*args, **kwds)
    except AttributeError:
        return default
```

The intention of function *trycalling* is to try calling a method named *attrib* on object *obj*, but to return *default* if *obj* has no method thus named. However, the function as coded does not do *just* that: it also accidentally hides any error case where `AttributeError` is raised inside the sought-after method, silently returning `default` in those cases. This may easily hide bugs in other code. To do exactly what's intended, the function must take a little bit more care:

```
def trycalling(obj, attrib, default, *args, **kwds):
    try:
        method = getattr(obj, attrib)
    except AttributeError:
        return default
    else:
        return method(*args, **kwds)
```

This implementation of *trycalling* separates the `getattr` call, placed in the `try` clause and therefore guarded by the handler in the `except` clause, from the call of the method, placed in the `else` clause and therefore free to propagate any exception. The proper approach to EAFP involves frequent use of the `else` clause on `try/except` statements (which is more explicit, and thus better Python style, than just placing the nonguarded code after the whole `try/except` statement).

Handling Errors in Large Programs

In large programs, it is especially easy to err by making your `try/except` statements too wide, particularly once you have convinced yourself of the power of EAFP as a general error-checking strategy. A `try/except` combination is too wide when it catches too many different errors, or an error that can occur in too many different places. The latter is a problem when you need to distinguish exactly what went wrong and where, and the information in the traceback is not sufficient to pinpoint such details (or you discard some or all of the information in the traceback). For effective error handling, you have to keep a clear distinction between errors and anomalies that you expect (and thus know how to handle) and unexpected errors and anomalies that indicate a bug in your program.

Some errors and anomalies are not really erroneous, and perhaps not even all that anomalous: they are just special, "edge" cases, perhaps somewhat rare but nevertheless quite expected, which you choose to handle via EAFP rather than via LBYL to avoid LBYL's many intrinsic defects. In such cases, you should just handle the anomaly, often without even logging or reporting it.

Keep Your try/except Constructs Narrow

Be very careful to keep `try/except` constructs as narrow as feasible. Use a small try clause that contains a small amount of code that doesn't call too many other functions, and use very specific exception-class tuples in the `except` clauses; if need be, further analyze the details of the exception in your handler code, and `raise` again as soon as you know it's not a case this handler can deal with.

Errors and anomalies that depend on user input or other external conditions not under your control are always expected, precisely because you have no control over their underlying causes. In such cases, you should concentrate your effort on handling the anomaly gracefully, reporting and logging its exact nature and details, and keeping your program running with undamaged internal and persistent state. The breadth of `try/except` clauses under such circumstances should also be reasonably narrow, although this is not quite as crucial as when you use EAFP to structure your handling of not-really-erroneous special/edge cases.

Lastly, entirely unexpected errors and anomalies indicate bugs in your program's design or coding. In most cases, the best strategy regarding such errors is to avoid `try/except` and just let the program terminate with error and traceback messages. (You might want to log such information and/or display it more suitably with an application-specific hook in `sys.excepthook`, as we'll discuss shortly.) In the unlikely case that your program must keep running at all costs, even under dire circumstances, `try/except` statements that are quite wide may be appropriate, with the `try` clause guarding function calls that exercise vast swaths of program functionality, and broad `except` clauses.

In the case of a long-running program, make sure to log all details of the anomaly or error to some persistent place for later study (and also report to yourself some indication of the problem, so that you know such later study is necessary). The key is making sure that you can revert the program's persistent state to some undamaged, internally consistent point. The

techniques that enable long-running programs to survive some of their own bugs, as well as environmental adversities, are known as checkpointing (basically, periodically saving program state, and writing the program so it can reload the saved state and continue from there) and transaction processing; we do not cover them further in this book.

Logging Errors

When Python propagates an exception all the way to the top of the stack without finding an applicable handler, the interpreter normally prints an error traceback to the standard error stream of the process (`sys.stderr`) before terminating the program. You can rebind `sys.stderr` to any file-like object usable for output in order to divert this information to a destination more suitable for your purposes.

When you want to change the amount and kind of information output on such occasions, rebinding `sys.stderr` is not sufficient. In such cases, you can assign your own function to `sys.excepthook`: Python calls it when terminating the program due to an unhandled exception. In your exception-reporting function, output whatever information will help you diagnose and debug the problem and direct that information to whatever destinations you please. For example, you might use module `traceback` (covered in "The traceback Module") to format stack traces. When your exception-reporting function terminates, so does your program.

The logging package

The Python standard library offers the rich and powerful `logging` package to let you organize the logging of messages from your applications in systematic, flexible ways. Pushing things to the limit, you might write a whole hierarchy of `Logger` classes and subclasses; you might couple the loggers with instances of `Handler` (and subclasses thereof); you might also insert instances of class `Filter` to fine-tune criteria determining what messages get logged in which ways.

Messages are formatted by instances of the `Formatter` class—the messages themselves are instances of the `LogRecord` class. The `logging` package even includes a dynamic configuration facility, whereby you may dynamically set logging-configuration files by reading them from disk files, or even by receiving them on a dedicated socket in a specialized thread.

While the `logging` package sports a frighteningly complex and powerful architecture, suitable for implementing highly sophisticated logging strategies and policies that may be needed in vast and complicated software systems, in most applications you may get away with using a tiny subset of the package. First, `import logging`. Then, emit your message by passing it as a string to any of the module's functions `debug`, `info`, `warning`, `error`, or `critical`, in increasing order of severity. If the string you pass contains format specifiers such as `%s` (as covered in "Legacy String Formatting with %") then, after the string, pass as further arguments all the values to be formatted in that string. For example, don't call:

```
logging.debug('foo is %r' % foo)
```

which performs the formatting operation whether it's needed or not; rather, call:

```
logging.debug('foo is %r', foo)
```

which performs formatting if and only if needed (i.e., if and only if calling debug is going to result in logging output, depending on the current threshold level).

Unfortunately, the `logging` module does not support the more readable formatting approach covered in "String Formatting", but only the antiquated one covered in "Legacy String Formatting with %". Fortunately, it's very rare to need any formatting specifier beyond the simple `%s` and `%r`.

By default, the threshold level is `WARNING`: any of the functions `warning`, `error`, or `critical` results in logging output, but the functions `debug` and `info` do not. To change the threshold level at any time, call `logging.getLogger().setLevel`, passing as the only argument one of the corresponding constants supplied by module `logging`: `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`. For example, once you call:

```
logging.getLogger().setLevel(logging.DEBUG)
```

all of the logging functions from `debug` to `critical` result in logging output until you change level again; if later you call:

```
logging.getLogger().setLevel(logging.ERROR)
```

then only the functions `error` and `critical` result in logging output (`debug`, `info`, and `warning` won't result in logging output); this condition, too, persists until you change level again, and so forth.

By default, logging output goes to your process's standard error stream (`sys.stderr`, as covered in Table 7-3) and uses a rather simplistic format (for example, it does not include a timestamp on each line it outputs). You can control these settings by instantiating an appropriate handler instance, with a suitable formatter instance, and creating and setting a new logger instance to hold it. In the simple, common case in which you just want to set these logging parameters once and for all, after which they persist throughout the run of your program, the simplest approach is to call the `logging.basicConfig` function, which lets you set up things quite simply via named parameters. Only the very first call to `logging.basicConfig` has any effect, and only if you call it before any of the logging functions (`debug`, `info`, and so on). Therefore, the most common use is to call `logging.basicConfig` at the very start of your program. For example, a common idiom at the start of a program is something like:

```
import logging
logging.basicConfig(
    format='%(asctime)s %(levelname)8s %(message)s',
    filename='/tmp/logfile.txt', filemode='w')
```

This setting writes logging messages to a file, nicely formatted with a precise human-readable timestamp, followed by the severity level right-aligned in an eight-character field, followed by the message proper.

For excruciatingly large amounts of detailed information on the logging package, and all the wonders you can perform with it, be sure to consult Python's rich online information about it.

The assert Statement

The `assert` statement allows you to introduce "sanity checks" into a program. `assert` is a simple statement with the following syntax:

```
assert condition[, expression]
```

When you run Python with the optimize flag (**-O**, as covered in "Command-Line Syntax and Options"), `assert` is a null operation: the compiler generates no code for it. Otherwise, `assert` evaluates *condition*. When *condition* is satisfied, `assert` does nothing. When *condition* is not satisfied, `assert` instantiates `AssertionError` with *expression* as the argument (or without arguments, if there is no *expression*) and raises the resulting instance.[2]

`assert` statements can be an effective way to document your program. When you want to state that a significant, nonobvious condition *C* is known to hold at a certain point in a program's execution (known as an *invariant* of your program), `assert` *C* is often better than a comment that just states that *C* holds.

The advantage of `assert` is that, when *C* does *not* in fact hold, `assert` immediately alerts you to the problem by raising `AssertionError`, if the program is running without the **-O** flag. Once the code is thoroughly

debugged, run it with **-O**, turning `assert` into a null operation and incurring no overhead (the `assert` remains in your source code to document the invariant).

Don't Overuse assert

Never use `assert` for other purposes besides sanity-checking program invariants. A serious but very common mistake is to use `assert` about the values of inputs or arguments: checking for erroneous arguments or inputs is best done more explicitly, and in particular must not be turned into a null operation by a command-line flag.

The __debug__ Built-in Variable

When you run Python without option **-O**, the `__debug__` built-in variable is `True`. When you run Python with option **-O**, `__debug__` is `False`. Also, with option **-O**, the compiler generates no code for any `if` statement whose sole guard condition is `__debug__`.

To exploit this optimization, surround the definitions of functions that you call only in `assert` statements with `if __debug__:`. This technique makes compiled code smaller and faster when Python is run with **-O**, and enhances program clarity by showing that those functions exist only to perform sanity checks.

---

1 except that multiple calls to close are allowed and innocuous: all but the first one perform no operation.

2 Some third-party frameworks, such as pytest, materially improve the usefulness of the assert statement.

Chapter 4. Modules

A typical Python program is made up of several source files. Each source file is a *module*, grouping code and data for reuse. Modules are normally independent of each other, so that other programs can reuse the specific modules they need. Sometimes, to manage complexity, you group together related modules into a *package*—a hierarchical, tree-like structure.

A module explicitly establishes dependencies upon other modules by using `import` or `from` statements. In some programming languages, global variables provide a hidden conduit for coupling between modules. In Python, global variables are not global to all modules, but rather are attributes of a single module object. Thus, Python modules always communicate in explicit and maintainable ways, clarifying the couplings between them by making them explicit.

Python also supports *extension modules*—modules coded in other languages such as C, C++, Java, or C#—for use in Python. For the Python code importing a module, it does not matter whether the module is pure Python or an extension. You can always start by coding a module in Python. Later, should you need more speed, you can refactor and recode some parts of modules in lower-level languages, without changing the client code that uses those modules. Chapter "Extending and Embedding Classic Python" shows how to write extensions in C and Cython.

This chapter discusses module creation and loading. It also covers grouping modules into packages, using Python's distribution utilities (the older, deprecated distutils, and the currently-recommended setuptools) to install packages, and how to prepare packages for distribution; this latter subject is more thoroughly covered in Chapter "Distributing Extensions and Programs." This chapter closes with a discussion on how best to manage your Python environment(s).

Module Objects

A module is a Python object with arbitrarily named attributes that you can bind and reference. The Python code for a module named *aname* usually lives in a file named *aname.py*, as covered in "Module Loading."

In Python, modules are objects (values), handled like other objects. Thus, you can pass a module as an argument in a call to a function. Similarly, a function can return a module as the result of a call. A module, just like any other object, can be bound to a variable, an item in a container, or an attribute of an object. Modules can be keys or values in a dictionary, and can be members of a set. For example, the `sys.modules` dictionary, covered in "Module Loading," holds module objects as its values. The fact that modules can be treated like other values in Python is often expressed by saying that modules are *first-class* objects.

The import Statement

You can use any Python source file as a module by executing an `import` statement in another Python source file. `import` has the following syntax:

```
import modname [as varname][,...]
```

After the `import` keyword come one or more module specifiers separated by commas. In the simplest, most common case, a module specifier is just *modname*, an identifier—a variable that Python binds to the module object

when the `import` statement finishes. In this case, Python looks for the module of the same name to satisfy the `import` request. For example:

```
import mymodule
```

looks for the module named `mymodule` and binds the variable named *mymodule* in the current scope to the module object. *modname* can also be a sequence of identifiers separated by dots (`.`) to name a module contained in a package, as covered in "Packages."

When `as` *varname* is part of a module specifier, Python looks for a module named *modname* and binds the module object to the variable *varname*. For example:

```
import mymodule as alias
```

looks for the module named `mymodule` and binds the module object to variable *alias* in the current scope. *varname* must always be a simple identifier.

Module body

The body of a module is the sequence of statements in the module's source file. There is no special syntax required to indicate that a source file is a module; you can use any valid Python source file as a module. A module's body executes immediately the first time a given run of a program imports it. When the body starts executing, the module object has already been created, with an entry in `sys.modules` already bound to the module object. The module's (global) namespace is gradually populated as the module's body executes.

Attributes of module objects

An `import` statement creates a new namespace containing all the attributes of the module. To access an attribute in this namespace, use the name or alias of the module as a prefix:

```
import mymodule
a = mymodule.f()
```

or:

```
import mymodule as alias
a = alias.f()
```

Normally, it's statements in the module body that bind the attributes of a module object. When a statement in the module body binds a (global) variable, what gets bound is an attribute of the module object.

A Module Body Exists To Bind the Module's Attributes

The normal purpose of a module body is to create the module's attributes: `def` statements create and bind functions, `class` statements create and bind classes, and assignment statements can bind attributes of any type. For clarity and cleanliness of your code, be wary about doing anything else in the top logical level of the module's body *except* binding the module's attributes.

You can also bind module attributes in code outside the body (i.e., in other modules); just assign a value to the attribute reference syntax *M.name* (where *M* is any expression whose value is the module, and identifier *name* is the attribute name). For clarity, however, it's best to bind module attributes only in the module's own body.

The `import` statement binds some module attributes as soon as it creates the module object, before the module's body executes. The `__dict__` attribute is the `dict` object that the module uses as the namespace for its attributes. Unlike other attributes of the module, `__dict__` is not available to code in the module as a global variable. All other attributes in the module are items in `__dict__` and are available to code in the module as global variables. Attribute `__name__` is the module's name; attribute `__file__` is the filename from which the module was loaded; other

dunder-named attributes hold other module metadata. (Also see “Special Attributes of Package Objects” for attribute `__path__`, in packages only).

For any module object *M*, any object *x*, and any identifier string *S* (except `__dict__`), binding *M.S* = *x* is equivalent to binding *M.__dict__['S']* = *x*. An attribute reference such as *M.S* is also substantially equivalent to *M.__dict__['S']*. The only difference is that, when *S* is not a key in *M.__dict__*, accessing *M.__dict__['S']* raises `KeyError`, while accessing *M.S* raises `AttributeError`. Module attributes are also available to all code in the module’s body as global variables. In other words, within the module body, *S* used as a global variable is equivalent to *M.S* (i.e., *M.__dict__['S'])* for both binding and reference (when *S* is *not* a key in *M.__dict__*, however, referring to *S* as a global variable raises `NameError`).

Python built-ins

Python supplies several built-in objects (covered in Chapter “Core Built-ins and Standard Library Modules”). All built-in objects are attributes of a preloaded module named `builtins`. When Python loads a module, the module automatically gets an extra attribute named `__builtins__`, which refers either to the module `builtins` or to its dictionary. Python may choose either, so don’t rely on `__builtins__`. If you need to access the module `builtins` directly (a rare need), use an `import builtins` statement. When you access a variable found neither in the local namespace nor in the global namespace of the current module, Python looks for the identifier in the current module’s `__builtins__` before raising `NameError`.

The lookup is the only mechanism that Python uses to let your code access builtins. The built-ins’ names are not reserved, nor are they hardwired in Python itself. Your own code can use the access mechanism directly (do so in moderation, or your program’s clarity and simplicity will suffer). Since Python accesses built-ins only when it cannot resolve a name in the local or module namespace, it is usually sufficient to define a replacement in one of those namespaces. You can, however, add your own built-ins or substitute

your functions for the normal built-in ones, in which case all modules see the added or replaced one. The following toy example shows how you can wrap a built-in function with your own function, allowing `abs()` to take a string argument (and return a rather arbitrary mangling of the string):

```
# abs takes a numeric argument; let's make it accept a string as
well
import builtins
_abs = builtins.abs                          # save original
built-in
def abs(str_or_num):
    if isinstance(str_or_num, str):          # if arg is a string
        return ''.join(sorted(set(str_or_num)))  # get this
instead
    return _abs(str_or_num)                  # call real built-in
builtins.abs = abs                     # override built-in
w/wrapper
```

Module documentation string

If the first statement in the module body is a string literal, Python binds that string as the module's documentation string attribute, named `__doc__`. Documentation strings are also called *docstrings*; we cover them in "Docstrings."

Module-private variables

No variable of a module is truly private. However, by convention, every identifier starting with a single underscore (_), such as `_secret`, is *meant* to be private. In other words, the leading underscore communicates to client-code programmers that they should not access the identifier directly.

Development environments and other tools rely on the leading-underscore naming convention to discern which attributes of a module are public (i.e., part of the module's interface) and which are private (i.e., to be used only within the module).

Respect the "Leading Underscore Means Private" Convention

It's important to respect the "leading underscore means private" convention, particularly when you write client code that uses modules written by others. Avoid using any attributes in such modules whose names start with _. Future releases of the modules will strive to maintain their public interface, but are quite likely to change private implementation details: private attributes are meant exactly for such details.

The from Statement

Python's `from` statement lets you import specific attributes from a module into the current namespace. `from` has two syntax variants:

```
from modname import attrname [as varname][,...]
from modname import *
```

A `from` statement specifies a module name, followed by one or more attribute specifiers separated by commas. In the simplest and most common case, an attribute specifier is just an identifier *attrname*, which is a variable that Python binds to the attribute of the same name in the module named *modname*. For example:

```
from mymodule import f
```

modname can also be a sequence of identifiers separated by dots (`.`) to name a module within a package, as covered in "Packages."

When `as` *varname* is part of an attribute specifier, Python gets from the module the value of attribute *attrname* and binds it to variable *varname*. For example:

```
from mymodule import f as foo
```

attrname and *varname* are always simple identifiers.

You may optionally enclose in parentheses all the attribute specifiers that follow the keyword `import` in a `from` statement. This can be useful when you have many attribute specifiers, in order to split the single logical line of the `from` statement into multiple logical lines more elegantly than by using backslashes (`\`):

```
from some_module_with_a_long_name import (
    another_name, and_another as x, one_more, and_yet_another as
y)
```

from ... import *

Code that is directly inside a module body (not in the body of a function or class) may use an asterisk (`*`) in a `from` statement:

```
from mymodule import *
```

The `*` requests that "all" attributes of module *modname* be bound as global variables in the importing module. When module *modname* has an attribute named `__all__`, the attribute's value is the list of the attribute names that this type of `from` statement binds. Otherwise, this type of `from` statement binds all attributes of *modname* except those beginning with underscores.

Beware Using From M Import * in Your Code

Since from *M* `import *` may bind an arbitrary set of global variables, it can have unforeseen, undesired side effects, such as hiding built-ins and rebinding variables you still need. Use the `*` form of `from` very sparingly, if at all, and only to import modules that are explicitly documented as supporting such usage. Your code is most likely better off *never* using this form, which is meant mostly as a convenience for occasional use in interactive Python sessions.

from Versus import

The `import` statement is often a better choice than the `from` statement. When you always access module *M* with the statement `import` *M* and

always access *M*'s attributes with explicit syntax *M.A*, your code is slightly less concise but far clearer and more readable. One good use of `from` is to import specific modules from a package, as we discuss in "Packages." In most other cases, `import` is better style than `from`.

Module Loading

Module-loading operations rely on attributes of the built-in `sys` module (covered in "The sys Module") and are implemented in the built-in function `__import__`. Your code could call `__import__` directly, but this is strongly discouraged in modern Python; rather, `import importlib` and call `importlib.import_module` with the module name string as the argument. `import_module` returns the module object or, should the import fail, raises `ImportError`. However, it's best to have a clear understanding of the semantics of `__import__`, because `import_module` and `import` statements both depend on it.

To import a module named *M*, `__import__` first checks dictionary `sys.modules`, using string *M* as the key. When key *M* is in the dictionary, `__import__` returns the corresponding value as the requested module object. Otherwise, `__import__` binds `sys.modules[`*M*`]` to a new empty module object with a `__name__` of *M*, then looks for the right way to initialize (load) the module, as covered in "Searching the Filesystem for a Module."

Thanks to this mechanism, the relatively slow loading operation takes place only the first time a module is imported in a given run of the program. When a module is imported again, the module is not reloaded, since `__import__` rapidly finds and returns the module's entry in `sys.modules`. Thus, all imports of a given module after the first one are very fast: they're just dictionary lookups. (To *force* a reload, see "Reloading Modules.")

Built-in Modules

When a module is loaded, `__import__` first checks whether the module is built-in. The tuple `sys.builtin_module_names` names all built-in modules, but rebinding that tuple does not affect module loading. When Python loads a built-in module, as when it loads any other extension, Python calls the module's initialization function. The search for built-in modules also looks for modules in platform-specific locations, such as the Registry in Windows.

Searching the Filesystem for a Module

If module *M* is not built-in, `__import__` looks for *M*'s code as a file on the filesystem. `__import__` looks at the strings, which are the items of list `sys.path`, in order. Each item is the path of a directory, or the path of an archive file in the popular ZIP format. `sys.path` is initialized at program startup, using the environment variable `PYTHONPATH` (covered in "Environment Variables"), if present. The first item in `sys.path` is always the directory from which the main program is loaded. An empty string in `sys.path` indicates the current directory.

Your code can mutate or rebind `sys.path`, and such changes affect which directories and ZIP archives `__import__` searches to load modules. Changing `sys.path` does *not* affect modules that are already loaded (and thus already recorded in `sys.modules`) when you change `sys.path`.

If there is a text file with the extension *.pth* in the *PYTHONHOME* directory at startup, Python adds the file's contents to `sys.path`, one item per line. *.pth* files can contain blank lines and comment lines starting with the character `#`; Python ignores any such lines. *.pth* files can also contain `import` statements (which Python executes before your program starts to execute), but no other kinds of statements.

When looking for the file for module *M* in each directory and ZIP archive along `sys.path`, Python considers the following extensions in this order:

1. *.pyd* and *.dll* (Windows) or *.so* (most Unix-like platforms), which indicate Python extension modules. (Some Unix dialects use different extensions; e.g., *.sl* on HP-UX.) On most platforms, extensions cannot

be loaded from a ZIP archive—only source or bytecode-compiled Python modules can.

2. *.py*, which indicates Python source modules.

3. *.pyc*, which indicates bytecode-compiled Python modules.

4. When it finds a *.py* file, Python also looks for a directory called *__pycache__*; if it finds such a directory, Python looks in that directory for the extension .<*tag*>*.pyc*, where <*tag*> is a string specific to the version of Python that is looking for the module.

One last path in which Python looks for the file for module *M* is *M/__init__.py*: a file named *__init__.py* in a directory named *M*, as covered in "Packages."

Upon finding source file *M.py*, Python compiles it to *M.*<*tag*>*.pyc*, unless the bytecode file is already present, is newer than *M.py*, and was compiled by the same version of Python. If *M.py* is compiled from a writable directory, Python creates a *__pycache__* subdirectory if necessary and saves the bytecode file to the filesystem in that subdirectory so that future runs won't needlessly recompile. When the bytecode file is newer than the source file (based on an internal timestamp in the bytecode file, not on trusting the date as recorded in the filesystem), Python does not recompile the module.

Once Python has the bytecode, whether built anew by compilation or read from the filesystem, Python executes the module body to initialize the module object. If the module is an extension, Python calls the module's initialization function.

The Main Program

Execution of a Python application starts with a top-level script (known as the *main program*), as explained in "The python Program." The main program executes like any other module being loaded, except that Python keeps the bytecode in memory, not saving it to disk. The module name for

the main program is `'__main__'`, both as the `__name__` variable (module attribute) and as the key in `sys.modules`.

Don't Import the .py File You're Using as the Main Program

You should not import the same *.py* file that is the main program. If you do, Python loads the module again, and the body executes again in a separate module object with a different `__name__`.

Code in a Python module can test if the module is being used as the main program by checking if global variable `__name__` has the value `'__main__'`. The idiom:

```
if __name__ == '__main__':
```

is often used to guard some code so that it executes only when the module runs as the main program. If a module is meant only to be imported, it should normally execute unit tests when run as the main program, as covered in "Unit Testing and System Testing."

Reloading Modules

Python loads a module only the first time you import the module during a program run. When you develop interactively, you need to *reload* your modules after editing them (some development environments provide automatic reloading).

To reload a module, pass the module object (*not* the module name) as the only argument to the function `reload` from the `importlib` module. `importlib.reload(`*M*`)` ensures the reloaded version of *M* is used by client code that relies on `import` *M* and accesses attributes with the syntax *M.A*. However, `importlib.reload(`*M*`)` has no effect on other existing

references bound to previous values of *M*'s attributes (e.g., with a `from` statement). In other words, already-bound variables remain bound as they were, unaffected by `reload`. `reload`'s inability to rebind such variables is a further incentive to use `import` rather than `from`.

`reload` is not recursive: when you reload module *M*, this does not imply that other modules imported by *M* get reloaded in turn. You must reload, by explicit calls to `reload`, every module you have modified.

Circular Imports

Python lets you specify circular imports. For example, you can write a module *a.py* that contains `import b`, while module *b.py* contains `import a`.

If you decide to use a circular import for some reason, you need to understand how circular imports work in order to avoid errors in your code.

Avoid Circular Imports

In practice, you are nearly always better off avoiding circular imports, since circular dependencies are fragile and hard to manage.

Say that the main script executes `import a`. As discussed earlier, this `import` statement creates a new empty module object as `sys.modules['a']`, then the body of module `a` starts executing. When `a` executes `import b`, this creates a new empty module object as `sys.modules['b']`, and then the body of module `b` starts executing. `a`'s module body cannot proceed until `b`'s module body finishes.

Now, when `b` executes `import a`, the `import` statement finds `sys.modules['a']` already bound, and therefore binds global variable `a` in module `b` to the module object for module `a`. Since the execution of `a`'s module body is currently blocked, module `a` is usually only partly

populated at this time. Should the code in `b`'s module body try to access some attribute of module `a` that is not yet bound, an error results.

If you keep a circular import, you must carefully manage the order in which each module binds its own globals, imports other modules, and accesses globals of other modules. You get greater control over the sequence in which things happen by grouping your statements into functions, and calling those functions in a controlled order, rather than just relying on sequential execution of top-level statements in module bodies. Removing circular dependencies (for example, by moving an import away from module scope and into a referencing function) is easier than ensuring bomb-proof ordering to deal with circular dependencies.

sys.modules Entries

`__import__` never binds anything other than a module object as a value in `sys.modules`. However, if `__import__` finds an entry already in `sys.modules`, it returns that value, whatever type it may be. `import` and `from` statements rely on `__import__`, so they too can use objects that are not modules.

Custom Importers

Another advanced, rarely-needed functionality that Python offers is the ability to change the semantics of some or all `import` and `from` statements.

Rebinding __import__

You can rebind the `__import__` attribute of module `builtin` to your own custom importer function—for example, one using the generic built-in-wrapping technique shown in "Python built-ins." Such a rebinding affects all `import` and `from` statements that execute after the rebinding and thus can have an undesired global impact. A custom importer built by rebinding `__import__` must implement the same interface and semantics as the

built-in `__import__`, and, in particular, it is responsible for supporting the correct use of `sys.modules`.

Beware Rebinding Builtin `__import__`

While rebinding `__import__` may initially look like an attractive approach, in most cases where custom importers are necessary, you're better off implementing them via *import hooks*.

Import hooks

Python offers rich support for selectively changing the details of imports' behavior. Custom importers are an advanced and rarely-needed technique, yet some applications may need them for purposes such as importing code from archives other than ZIP files, databases, network servers, and so on.

The most suitable approach for such highly advanced needs is to record *importer factory* callables as items in the attributes `meta_path` and/or `path_hooks` of the module `sys`, as detailed in PEP 451. This is how Python hooks up the standard library module `zipimport` to allow seamless importing of modules from ZIP files, as previously mentioned. A full study of the details of PEP 451 is indispensable for any substantial use of `sys.path_hooks` and friends, but here's a toy-level example to help understand the possibilities, should you ever need them.

Suppose that, while developing the first outline of some program, you want to be able to use `import` statements for modules that you haven't written yet, getting just messages (and empty modules) as a consequence. You can obtain such functionality (leaving aside the complexities connected with packages, and dealing with simple modules only) by coding a custom importer module as follows:

```
import sys, types
class ImporterAndLoader(object):
    '''importer and loader can be a single class'''
    fake_path = '!dummy!'
```

```
        def __init__(self, path):
            # only handle our own fake-path marker
            if path != self.fake_path:
                raise ImportError
        def find_module(self, fullname):
            # don't even try to handle any qualified module name
            if '.' in fullname:
                return None
            return self
        def create_module(self, spec):
            # create module "the default way"
            return None
        def exec_module(self, mod):
            # populate the already-initialized module
            # just print a message in this toy example
            print(f'NOTE: module {mod!r} not yet written')
sys.path_hooks.append(ImporterAndLoader)
sys.path.append(ImporterAndLoader.fake_path)
if __name__ == '__main__':      # self-test when run as main
script
    import missing_module       # importing a simple *missing*
module
    print(missing_module)       # ...should succeed
    print(sys.modules.get('missing_module'))  # ...should also
succeed
```

We just write trivial versions of `create_module` (which in this case just returns `None`, asking the system to create the module object in the "default way") and `exec_module` (which receives the module object already initialized with dunder attributes, and whose task would normally be to populate it appropriately).

We could, alternatively, use the powerful new *module spec* concept as detailed in PEP 451. However, that requires the standard library module `importlib`; for this toy example, we don't need all that extra power. Therefore, we choose instead to implement the method `find_module`, which, although now deprecated, still works fine for backward compatibility.

Packages

A *package* is a module containing other modules. Some or all of the modules in a package may be *subpackages*, resulting in a hierarchical tree-like structure. A package named *P* typically resides in a subdirectory, also called *P*, of some directory in `sys.path`. Packages can also live in ZIP files; in this section, we explain the case in which the package lives on the filesystem, since the case in which a package is in a ZIP file is similar, relying on the hierarchical filesystem-like structure within the ZIP file.

The module body of *P* is in the file *P/__init__.py*. This file *must* exist (except for *namespace packages*, covered in "Namespace Packages"), even if it's empty (representing an empty module body), in order to tell Python that directory *P* is indeed a package. Python loads the module body of a package when you first import the package (or any of the package's modules), behaving just like for any other Python module. The other *.py* files in directory *P* are the modules of package *P*. Subdirectories of *P* containing *__init__.py* files are *subpackages* of *P*. Nesting can proceed to any depth.

You can import a module named *M* in package *P* as *P.M*. More dots let you navigate a hierarchical package structure. (A package's module body always loads *before* any module in the package.) If you use the syntax `import` *P.M*, the variable *P* is bound to the module object of package *P*, and the attribute *M* of object *P* is bound to the module *P.M*. If you use the syntax `import` *P.M* `as` *V*, the variable *V* is bound directly to the module *P.M*.

Using `from` *P* `import` *M* to import a specific module *M* from package *P* is a perfectly acceptable, indeed highly recommended practice: the `from` statement is specifically okay in this case. `from` *P* `import` *M* `as` *V* is also just fine, and exactly equivalent to `import` *P.M* `as` *V*. You can also use *relative* paths: that is, module *M* in package *P* can import its "sibling" module *X* (also in package *P*) with `from . import X`.

Sharing Objects Among Modules In A Package

The simplest, cleanest way to share objects (e.g., functions or constants) among modules in a package P is to group the shared objects in a module conventionally named P/common.py. That way, you can use `from . import common` in every module in the package that needs to access some of the common objects, and then refer to the objects as `common.f`, `common.K`, and so on.

Special Attributes of Package Objects

A package *P*'s `__file__` attribute is the string that is the path of *P*'s module body—that is, the path of the file *P/__init__.py*. *P*'s `__package__` attribute is the name of *P*'s package.

A package *P*'s module body—that is, the Python source that is in the file *P/__init__.py*—can optionally set a global variable named `__all__` (just like any other module can) to control what happens if some other Python code executes the statement `from` *P* `import *`. In particular, if `__all__` is not set, `from` *P* `import *` does not import *P*'s modules, but only names that are set in *P*'s module body and lack a leading _. In any case, this is *not* recommended usage.

A package *P*'s `__path__` attribute is the list of strings that are the paths to the directories from which *P*'s modules and subpackages are loaded. Initially, Python sets `__path__` to a list with a single element: the path of the directory containing the file *__init__.py* that is the module body of the package. Your code can modify this list to affect future searches for modules and subpackages of this package. This advanced technique is rarely necessary, but can be useful when you want to place a package's modules in various directories; a *namespace package*, as covered next, is however the usual way to accomplish this goal.

Namespace Packages

On `import foo`, when one or more directories that are immediate children of `sys.path` members are named *foo*, and none of them contains a file named *__init__.py*, Python deduces that `foo` is a *namespace package*. As a result, Python creates (and assigns to `sys.modules['foo']`) a package object `foo` without a `__file__` attribute; Python sets `foo.__path__` to the list of all the various directories that make up the package (like for any other package, your code may optionally choose to further alter it). This advanced approach is rarely needed.

Absolute Versus Relative Imports

As mentioned in "Packages," an `import` statement normally expects to find its target somewhere on `sys.path`, a behavior known as an *absolute* import. Alternatively, you can explicitly use a *relative* import, meaning an import of an object from within the current package. Relative imports use module or package names beginning with one or more dots, and are only available within the `from` statement. `from . import` *X* looks for the module or object named *X* in the current package; `from .`*X* `import` *y* looks in module or subpackage *X* within the current package for the module or object named *y*. If your package has subpackages, their code can access higher-up objects in the package by using multiple dots at the start of the module or subpackage name you place between `from` and `import`. Each additional dot ascends the directory hierarchy one level. Getting too fancy with this feature can easily damage your code's clarity, so use it with care, and only when necessary.

Distribution Utilities (distutils) and setuptools

Python modules, extensions, and applications can be packaged and distributed in several forms:

Compressed archive files

Generally *.zip* or *.tar.gz* (AKA *.tgz*) files—both forms are portable, and many other forms of compressed archives of trees of files and directories exist

Self-unpacking or self-installing executables

Normally *.exe* for Windows

Self-contained, ready-to-run executables that require no installation

For example, *.exe* for Windows, ZIP archives with a short script prefix on Unix, *.app* for the Mac, and so on

Platform-specific installers

For example, *.msi* on Windows, *.rpm* and *.srpm* on many Linux distributions, *.deb* on Debian GNU/Linux and Ubuntu, *.pkg* on macOS

Python Wheels

Popular third-party extensions, covered in "Python Wheels"

When you distribute a package as a self-installing executable or platform-specific installer, a user installs the package simply by running the installer. How to run such an installer program depends on the platform, but it doesn't matter which language the program was written in. We cover building self-contained, runnable executables for various platforms in Chapter "Distributing Extensions and Programs."

When you distribute a package as an archive file or as an executable that unpacks but does not install itself, it *does* matter that the package was coded in Python. In this case, the user must first unpack the archive file into some appropriate directory, say *C:\Temp\MyPack* on a Windows machine or *~/MyPack* on a Unix-like machine. Among the extracted files there should be a script, conventionally named *setup.py*, which uses the Python facility known as the *distribution utilities* (the now-deprecated, but still functioning, standard library package `distutils`[1]) or, better, the more popular,

modern, and powerful third-party package setuptools. The distributed package is then almost as easy to install as a self-installing executable. The user opens a command prompt window and changes to the directory into which the archive is unpacked. Then the user runs, for example:

```
C:\Temp\MyPack> python setup.py install
```

(`pip` is the preferred way to install packages nowadays, and is briefly discussed in "Python Environments.") The *setup.py* script, run with this **install** command, installs the package as a part of the user's Python installation, according to the options specified by the package's author in the setup script. Of course, the user needs appropriate permissions to write into the directories of the Python installation, so permission-raising commands such as *sudo* may also be needed; or, better yet, you can install into a *virtual environment*, covered in "Python Environments." `distutils` and `setuptools`, by default, print some information when the user runs *setup.py*. Option **--quiet**, right before the **install** command, hides most details (the user still sees error messages, if any). The following command gives detailed help on `distutils` or `setuptools`, depending on which toolset the package author used in their *setup.py*:

```
C:\Temp\MyPack> python setup.py --help
```

Recent versions of Python come with the excellent installer `pip` (a recursive acronym for "`pip` installs packages"), copiously documented online, yet very simple to use in most cases. `pip install` *package* finds the online version of *package* (usually on the huge PyPI repository, hosting more than 300,000 packages at the time of this writing), downloads it, and installs it for you (in a virtual environment, if one is active—see "Python Environments"). This books' authors have been using that simple, powerful approach for well over 90% of their installs for quite a while now.

Even if you have downloaded the package locally (say to */tmp/mypack*), for whatever reason (maybe it's not on PyPI, or you're trying out an experimental version that is not yet there), `pip` can still install it for you:

just run `pip install --no-index --find-links=/tmp/mypack` and `pip` does the rest.

Python Wheels

Python *wheels* are an archive format including structured metadata as well as Python code. Wheels offer an excellent way to package and distribute your Python packages, and `setuptools` (with the `wheel` extension, easily installed with `pip install wheel`) works seamlessly with them. Read all about them online and in Chapter "Distributing Extensions and Programs."

Python Environments

A typical Python programmer works on several projects concurrently, each with its own list of dependencies (typically, third-party libraries and data files). When the dependencies for all projects are installed into the same Python interpreter, it is very difficult to determine which projects use which dependencies, and impossible to handle projects with conflicting versions of certain dependencies.

Early Python interpreters were built on the assumption that each computer system would have "a Python interpreter" installed on it, to be used to run any Python program on that system. Operating system distributions started to include Python in their base installation, but, because Python has always been under active development, users often complained that they would like to use a version of the language more up-to-date than the one their operating system provided.

Techniques arose to let multiple versions of the language be installed on a system, but installation of third-party software remained nonstandard and intrusive. This problem was eased by the introduction of the *site-packages* directory as the repository for modules added to a Python installation, but it was still not possible to maintain projects with conflicting requirements using the same interpreter.

Programmers accustomed to command-line operations are familiar with the concept of a *shell environment*. A shell program running in a process has a current directory, variables that you can set with shell commands (very similar to a Python namespace), and various other pieces of process-specific state data. Python programs have access to the shell environment through `os.environ`.

Various aspects of the shell environment affect Python's operation, as mentioned in "Environment Variables." For example, the `PATH` environment variable determines which program, exactly, executes in response to **python** and other commands. You can think of those aspects of your shell environment that affect Python's operation as your *Python environment*. By modifying it you can determine which Python interpreter runs in response to the **python** command, which packages and modules are available under certain names, and so on.

Leave the System's Python to the System

We recommend taking control of your Python environment. In particular, do not build applications on top of a system-distributed Python. Instead, install another Python distribution independently, and adjust your shell environment so that the **python** command runs your locally installed Python rather than the system's Python.

Enter the Virtual Environment

The introduction of the `pip` utility created a simple way to install (and, for the first time, to uninstall) packages and modules in a Python environment. Modifying the system Python's *site-packages* still requires administrative privileges, and hence so does `pip` (although it can optionally install somewhere other than *site-packages*). Installed modules are still visible to all programs.

The missing piece is the ability to make controlled changes to the Python environment, to direct the use of a specific interpreter and a specific set of

Python libraries. That is just what *virtual environments (virtualenvs)* give you. Creating a virtualenv based on a specific Python interpreter copies or links to components from that interpreter's installation. Critically, though, each one has its own *site-packages* directory, into which you can install the Python resources of your choice.

Creating a virtualenv is *much* simpler than installing Python, and requires far less system resources (a typical newly created virtualenv takes less than 20 MB). You can easily create and activate them on demand, and deactivate and destroy them just as easily. You can activate and deactivate a virtualenv as many times as you like during its lifetime, and if necessary use `pip` to update the installed resources. When you are done with it, removing its directory tree reclaims all storage occupied by the virtualenv. A virtualenv's lifetime can be from minutes to months.

What Is a Virtual Environment?

A virtualenv is essentially a self-contained subset of your Python environment that you can switch in or out on demand. For a Python X.Y interpreter it includes, among other things, a *bin* directory containing a Python X.Y interpreter and a *lib/pythonX.Y/site-packages* directory containing pre-installed versions of `easy-install`, `pip`, `pkg_resources`, and `setuptools`. Maintaining separate copies of these important distribution-related resources lets you update them as necessary rather than forcing reliance on the base Python distribution.

A virtualenv has its own copies of (on Windows), or symbolic links to (on other platforms), Python distribution files. It adjusts the values of `sys.prefix` and `sys.exec_prefix`, from which the interpreter and various installation utilities determine the location of some libraries. This means that `pip` can install dependencies in isolation from other environments, in the virtualenv's *site-packages* directory. In effect, the virtualenv redefines which interpreter runs when you run the `python` command and which libraries are available to it, but leaves most aspects of your Python environment (such as the `PYTHONPATH` and `PYTHONHOME`

variables) alone. Since its changes affect your shell environment, they also affect any subshells in which you run commands.

With separate virtualenvs you can, for example, test two different versions of the same library with a project, or test your project with multiple versions of Python. You can also add dependencies to your Python projects without needing any special privileges, since you normally create your virtualenvs somewhere you have write permission.

The modern way to deal with virtualenvs is the `venv` module of the standard library: just run **python -m venv** *envpath*.

Creating and Deleting Virtual Environments

The command **python -m venv** *envpath* creates a virtual environment (in the *envpath* directory, which it also creates if necessary) based on the Python interpreter used to run the command. You can give multiple directory arguments to create, with a single command, several virtual environments (running the same Python interpreter); you can then install different sets of dependencies in each virtualenv. `venv` can take a number of options, as shown in Table 6-1.

Table 4-1. venv options

Option	Purpose
`--clear`	Removes any existing directory content before installing the virtual environment
`--copies`	Installs files by copying on the Unix-like platforms where using symbolic links is the default
`--help`	Prints out a command-line summary and a list of available options

`--help`	
`--system-site-packages`	Adds the standard system *site-packages* directory to the environment's search path, making modules already installed in the base Python available inside the environment
`--symlinks`	Installs files by using symbolic links on platforms where copying is the system default
`--upgrade`	Installs the running Python in the virtual environment, replacing whichever version had originally created the environment
`--without-pip`	Inhibits the usual behavior of calling `ensurepip` to bootstrap the `pip` installer utility into the environment

The following Unix terminal session shows the creation of a virtualenv and the structure of the directory tree created. The listing of the *bin* subdirectory shows that this particular user, by default, uses an interpreter installed in */usr/local/bin*.

```
machine:~ user$ python3 -m venv /tmp/tempenv
machine:~ user$ tree -dL 4 /tmp/tempenv
/tmp/tempenv
|--- bin
|--- include
|___ lib
     |___ python3.5
          |___ site-packages
               |--- __pycache__
               |--- pip
               |--- pip-8.1.1.dist-info
               |--- pkg_resources
               |--- setuptools
               |___ setuptools-20.10.1.dist-info
11 directories
machine:~ user$ ls -l /tmp/tempenv/bin/
total 80
-rw-r--r-- 1 sh wheel 2134 Oct 24 15:26 activate
```

```
-rw-r--r-- 1 sh wheel 1250 Oct 24 15:26 activate.csh
-rw-r--r-- 1 sh wheel 2388 Oct 24 15:26 activate.fish
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install
-rwxr-xr-x 1 sh wheel  249 Oct 24 15:26 easy_install-3.5
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3
-rwxr-xr-x 1 sh wheel  221 Oct 24 15:26 pip3.5
lrwxr-xr-x 1 sh wheel    7 Oct 24 15:26 python->python3
lrwxr-xr-x 1 sh wheel   22 Oct 24 15:26 python3-
>/usr/local/bin/python3
```

Deleting the virtualenv is as simple as removing the directory in which it resides (and all subdirectories and files in the tree: **rm -rf** *envpath* in Unix-like systems). Ease of removal is a helpful aspect of using virtualenvs.

The `venv` module includes features to help the programmed creation of tailored environments (e.g., by pre-installing certain modules in the environment or performing other post-creation steps). It is comprehensively documented online; we do not cover the API further in this book.

Working with Virtual Environments

To use a virtualenv you *activate* it from your normal shell environment. Only one virtualenv can be active at a time—activations don't "stack" like function calls. Activation tells your Python environment to use the virtualenv's Python interpreter and *site-packages* (along with the interpreter's full standard library). When you want to stop using those dependencies, deactivate the virtualenv and your standard Python environment is once again available. The virtualenv directory tree continues to exist until deleted, so you can activate and deactivate it at will.

Activating a virtualenv in Unix-based environments requires use of the `source` shell command so that the commands in the activation script make changes to the current shell environment. Simply running the script would mean its commands were executed in a subshell, and the changes would be lost when the subshell terminated. For bash and similar shells, you activate an environment located at path *envpath* with the command:

```
source envpath/bin/activate
```

Users of other shells are supported by scripts *activate.csh* and *activate.fish* located in the same directory. On Windows systems, use *activate.bat*:

```
envpath/Scripts/activate.bat
```

Activation does many things; most importantly:

- Adds the virtualenv's *bin* directory at the beginning of the shell's `PATH` environment variable, so its commands get run in preference to anything of the same name already on the `PATH`
- Defines a `deactivate` command to remove all effects of activation and return the Python environment to its former state
- Modifies the shell prompt to include the virtualenv's name at the start
- Defines a `VIRTUAL_ENV` environment variable as the path to the virtualenv's root directory (scripts can use this to introspect the virtualenv)

As a result of these actions, once a virtualenv is activated, the `python` command runs the interpreter associated with that virtualenv. The interpreter sees the libraries (modules and packages) installed in that environment, and `pip`—now the one from the virtualenv, since installing the module also installed the command in the virtualenv's *bin* directory—by default installs new packages and modules in the environment's *site-packages* directory.

Those new to virtualenvs should understand that a virtualenv is not tied to any project directory. It's perfectly possible to work on several projects, each with its own source tree, using the same virtualenv. Activate it, then move around your filesystem as necessary to accomplish your programming tasks, with the same libraries available (because the virtualenv determines the Python environment).

When you want to disable the virtualenv and stop using that set of resources, simply issue the command **deactivate**.

This undoes the changes made on activation, removing the virtualenv's *bin* directory from your `PATH`, so the `python` command once again runs your usual interpreter. As long as you don't delete it, the virtualenv remains available for future use by repeating the invocation to activate it.

Managing Dependency Requirements

Since virtualenvs were designed to complement installation with `pip`, it should come as no surprise that `pip` is the preferred way to maintain dependencies in a virtualenv. Because `pip` is already extensively documented, we mention only enough here to demonstrate its advantages in virtual environments. Having created a virtualenv, activated it, and installed dependencies, you can use the `pip freeze` command to learn the exact versions of those dependencies:

```
(tempenv) machine:- user$ pip freeze
appnope==0.1.0
decorator==4.0.10
ipython==5.1.0
ipython-genutils==0.1.0
pexpect==4.2.1
pickleshare==0.7.4
prompt-toolkit==1.0.8
ptyprocess==0.5.1
Pygments==2.1.3
requests==2.11.1
simplegeneric==0.8.1
six==1.10.0
traitlets==4.3.1
wcwidth==0.1.7
```

If you redirect the output of this command to a file called *filename*, you can recreate the same set of dependencies in a different virtualenv with the command `pip install -r` *filename*.

To distribute code for use by others, Python developers conventionally include a *requirements.txt* file listing the necessary dependencies. When you are installing software from the Python Package Index, `pip` installs the packages you request along with any indicated dependencies. When you're developing software it's convenient to have a requirements file, as you can

use it to add the necessary dependencies to the active virtualenv (unless they are already installed) with a simple `pip install -r requirements.txt`.

To maintain the same set of dependencies in several virtualenvs, use the same requirements file to add dependencies to each one. This is a convenient way to develop projects to run on multiple Python versions: create virtualenvs based on each of your required versions, then install from the same requirements file in each. While the preceding example uses exactly versioned dependency specifications as produced by `pip freeze`, in practice you can specify dependencies and version requirements in quite complex ways.

Other environment management solutions

Python virtual environments are focused on providing an isolated Python interpreter, into which you can install dependencies for one or more Python applications. The virtualenv package was the original way to create and manage virtualenvs. It has extensive facilities, including the ability to create environments from any available Python interpreter. Now maintained by the Python Packaging Authority team, a subset of its functionality has been extracted as the standard library `venv` module covered above, but `virtualenv` is worth learning about if you need more control.

The pipenv package is another dependency manager for Python environments. It maintains virtual environments whose contents are recorded in a file named *Pipfile*. Much in the manner of similar Javascript tools, it provides deterministic environments through the use of a *Pipfile.lock* file, allowing the exact same dependencies to be deployed as in the original installation.

The conda packages have a rather broader scope and can provide package, environment and dependency management for any language. An alternative `miniconda` package works exactly the same way but downloads only those packages it needs, while the full `anaconda` package pre-loads many hundreds of extension packages; the two are otherwise equivalent.

`conda` is written in Python, and installs its own Python interpreter in the base environment. Whereas a standard Python virtualenv normally uses the Python interpreter with which it was created, Python itself (when it is included in the environment) is simply another dependency. This makes it practical to update the version of Python used in the environment if necessary. You can also, if you wish, use `pip` to install packages in a Python-based `conda` environment. `conda` can dump an environment's contents as a YAML file, and you can use the file to replicate the environment elsewhere.

Because of its additional flexibility, coupled with comprehensive open source support led by its originators Anaconda, Inc. (formerly Continuum), `conda` is widely used in academic environments, particularly in data science and engineering, artificial intelligence, and financial analytics. It installs software from what it calls *channels*. The default channel maintained by Anaconda contains a wide range of packages, and third parties maintain specialised channels such as the *bioconda* channel for bioinformatics software. There is a community-based *conda-forge* channel, open to anyone who wants to join up and add software. Signing up for an account on the anaconda.org site lets you create your own channel, and also to distribute software through the conda-forge channel.

Best practices with virtualenvs

There is remarkably little advice on how best to manage your work with virtualenvs, though there are several sound tutorials: any good search engine gives you access to the most current ones. We can, however, offer a modest amount of advice that we hope will help you to get the most out of them.

When you are working with the same dependencies in multiple Python versions, it is useful to indicate the version in the environment name and use a common prefix. So for project *mutex* you might maintain environments called *mutex_39* and *mutex_310* for development under two different versions of Python. When it's obvious which Python is involved (remember, you see the environment name in your shell prompt), there's

less risk of testing with the wrong version. You maintain dependencies using common requirements to control resource installation in both.

Keep the requirements file(s) under source control, not the whole environment. Given the requirements file it's easy to re-create a virtualenv, which depends only on the Python release and the requirements. You distribute your project, and let your consumers decide which version(s) of Python to run it on and create the appropriate virtual environment(s).

Keep your virtualenvs outside your project directories. This avoids the need to explicitly force source code control systems to ignore them. It really doesn't matter where else you store them.

Your Python environment is independent of your projects' location in the filesystem. You can activate a virtual environment and then switch branches and move around a change-controlled source tree to use it wherever convenient.

To investigate a new module or package, create and activate a new virtualenv and then `pip install` the resources that interest you. You can play with this new environment to your heart's content, confident in the knowledge that you won't be installing unwanted dependencies into other projects.

You may find that experiments in a virtualenv require installation of resources that aren't currently project requirements. Rather than "pollute" your development environment, fork it: create a new virtualenv from the same requirements plus the testing functionality. Later, to make these changes permanent, use change control to merge your source and requirements changes back in from the fork.

If you are so inclined, you can create virtual environments based on debug builds of Python, giving you access to a wealth of instrumentation information about the performance of your Python code (and, of course, of the interpreter itself).

Developing your virtual environment itself requires change control, and the ease of virtualenv creation helps here too. Suppose that you recently released version 4.3 of a module, and you want to test your code with new

versions of two of its dependencies. You *could*, with sufficient skill, persuade **pip** to replace the existing copies of dependencies in your existing virtualenv.

It's much easier, though, to branch your project using source control tools, update the requirements, and create an entirely new virtual environment based on the updated requirements. You still have the original virtualenv intact, and you can switch between virtualenvs to investigate specific aspects of any migration issues that might arise. Once you have adjusted your code so that all tests pass with the updated dependencies, you check in your code *and* requirement changes, and merge into version 4.4 to complete the update, advising your colleagues that your code is now ready for the updated versions of the dependencies.

Virtual environments won't solve all of a Python programmer's problems. Tools can always be made more sophisticated, or more general. But, by golly, virtualenvs work, and we should take all the advantage of them that we can.

---

1 Planned to be deleted in Python 3.12.

Chapter 5. Strings and Things

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

Python's `str` type implements Unicode text strings with operators, built-in functions, methods, and dedicated modules. The somewhat similar `bytes` type represents arbitrary binary data as a sequence of bytes, also known as a *bytestring* or *byte string*. Many textual operations are possible on objects of either type: since these types are immutable, methods mostly create and return a new string unless returning the subject string unchanged. A mutable sequence of bytes can be represented as a *bytearray*, briefly introduced in "Built-In Types."

This chapter covers the methods available on these three types, in "Methods of String and Bytes Objects"; string formatting, in "String Formatting"; and the modules `string` (in "The string Module") and `pprint` (in "The pprint Module"). Issues related specifically to Unicode are covered in "Unicode". Formatted string literals are covered in "Formatted string literals".

Methods of String, Bytes and Bytearray Objects

`str`, `bytes` and `bytearray` objects are sequences, as covered in "Strings"; of these, only `bytearray` objects are mutable. All immutable-sequence operations (repetition, concatenation, indexing, and slicing) apply to instances of all three types, returning a new object of the same type. Unless otherwise specified in Table 8-1 methods are present on objects of all three types. Methods of `str`, `bytes` and `bytearray` objects return values of the same type.

Terms such as "letters," "whitespace," and so on, refer to the corresponding attributes of the `string` module, covered in "The string Module". Although `bytearray` objects are mutable, their methods returning a `bytearray` result do not mutate the object but instead return a newly-created `bytearray` even when the result is the same as the subject string.

For brevity the discussions below describe `bytes` and `bytearray` objects as `bytes`. Take care when mixing these types: while they are generally interoperable, the type of the result usually depends on the order of the operands.

> **NOTE**
>
> In Table 8-1, for conciseness, we use `sys.maxsize` for integer default values meaning, in practice, "any number, no matter how large."

Table 5-1. Significant string and bytes methods.

capitalize	*s*`.capitalize()` Returns a copy of *s* where the first character, if a letter, is uppercase, and all other letters, if any, are lowercase.
casefold	*s*`.casefold()` **str only**. Returns a string processed by the algorithm described in section 3.13 of the Unicode standard. This is similar to *s*`.lower` (described later in this list) but also takes into account equivalences such as that between the German `'ß'` and `'ss'`, and is thus better for case-insensitive matching.
	s`.center(`*n*, *fillchar*`=' ')` Returns a string of length `max(len(`*s*`),` *n*`)`, with a copy of *s* in the central part, surrounded by equal numbers of copies of character *fillchar* on both

center	sides (e.g., `'ciao'.center(2)` is `'ciao'` and `'x'.center(4,'_')` is `'_x__'`).
count	*s*`.count(`*sub*, *start*`=0`, *end*`=sys.maxsize)` Returns the number of nonoverlapping occurrences of substring *sub* in *s*`[`*start*`:`*end*`]`.
decode	*s*`.decode(`*encoding*`='utf-8'`, *errors*`='strict')` **Not str**. Returns a `str` object decoded from the bytes *s* according to the given encoding. *errors* specifies how to handle decoding errors: `'strict'` cause errors to raise `UnicodeError` exceptions, `'ignore'` ignores the malformed data, and `'replace'` replaces them with question marks; see "Unicode" for details. Other values can be registered via `codecs.register_error()`, covered in Table 8-7.
encode	*s*`.encode(`*encoding*`='utf-8'`, *errors*`='strict')` **str only**. Returns a `bytes` object obtained from *s* with the given encoding and error handling. See "Unicode" for more details.
endswith	*s*`.endswith(`*suffix*`,`*start*`=0,`*end*`=sys.maxsize)` Returns `True` when *s*`[`*start*`:`*end*`]` ends with string *suffix*; otherwise, `False`. *suffix* can be a tuple of strings, in which case `endswith` returns `True` when *s*`[`*start*`:`*end*`]` ends with any one of them.
expandtabs	*s*`.expandtabs(`*tabsize*`=8)` Returns a copy of *s* where each tab character is changed into one or more spaces, with tab stops every *tabsize* characters.
find	*s*`.find(`*sub*`,`*start*`=0`, *end*`=sys.maxsize)` Returns the lowest index in *s* where substring *sub* is found, such that *sub* is entirely contained in *s*`[`*start*`:`*end*`]`. For example, `'banana'.find('na')` is `2`, as is `'banana'.find('na',1)`, while `'banana'.find('na',3)` is `4`, as is `'banana'.find('na',-2)`. `find` returns `-1` when *sub* is not found.
format	*s*`.format(`**args*, ***kwargs*`)` `str` **only**. Formats the positional and named arguments according to formatting instructions contained in the string *s*. See "String Formatting" for further details.
format_map	*s*`.format_map(`*mapping*`)` **str only**. Formats the mapping argument according to formatting instructions contained in the string *s*. Equivalent to *s*`.format(**`*mapping*`)` but uses the mapping directly. See "String Formatting" for formatting details.
index	*s*`.index(`*sub*, *start*`=0`, *end*`=sys.maxsize)` Like `find`, but raises `ValueError` when *sub* is not found.
	s`.isalnum()` Returns `True` when `len(`*s*`)` is greater than `0` and all characters in *s* are

isalnum	letters or digits. When *s* is empty, or when at least one character of *s* is neither a letter nor a digit, `isalnum` returns `False`.
isalpha	*s*`.isalpha()` Returns `True` when `len(`*s*`)` is greater than `0` and all characters in *s* are letters. When *s* is empty, or when at least one character of *s* is not a letter, `isalpha` returns `False`.
isascii	Return `True` when the string is empty or all characters in the string are ASCII, `False` otherwise. ASCII characters have code points in the range U+0000-U+007F.
isdecimal	*s*`.isdecimal()` **str only**. Returns `True` when `len(`*s*`)` is greater than `0` and all characters in *s* can be used to form decimal-radix numbers. This includes Unicode characters defined as Arabic digits.[a]
isdigit	*s*`.isdigit()` Returns `True` when `len(`*s*`)` is greater than `0` and all characters in *s* are digits. When *s* is empty, or when at least one character of *s* is not a digit, `isdigit` returns `False`.
isidentifier	*s*`.isidentifier()` **str only**. Returns `True` when *s* is a valid identifier according to the Python language's definition; keywords also satisfy the definition, so, for example, `'class'.isidentifier()` returns `True`.
islower	*s*`.islower()` Returns `True` when all letters in *s* are lowercase. When *s* contains no letters, or when at least one letter of *s* is uppercase, `islower` returns `False`.
isnumeric	*s*`.isnumeric()` **str only**. Similar to *s*`.isdigit()`, but uses a broader definition of numeric characters that includes all characters defined as numeric in the Unicode standard (such as fractions).
isprintable	*s*`.isprintable()` **str only**. Returns `True` when all characters in *s* are spaces (`'\x20'`) or are defined in the Unicode standard as printable. Because the null string contains no unprintable characters, `''.isprintable()` returns `True`.
isspace	*s*`.isspace()` Returns `True` when `len(`*s*`)` is greater than `0` and all characters in *s* are whitespace. When *s* is empty, or when at least one character of *s* is not whitespace, `isspace` returns *False*.
istitle	*s*`.istitle()` Returns `True` when letters in *s* are *titlecase*: a capital letter at the start of each contiguous sequence of letters, all other letters lowercase (e.g., `'King Lear'.istitle()` is `True`). When *s* contains no letters, or when at least one letter of *s* violates the titlecase condition, `istitle` returns `False` (*e.g.*,

	`'1900'.istitle()` and `'Troilus and Cressida'.istitle()` return `False`).
isupper	*s*`.isupper()` Returns `True` when all letters in *s* are uppercase. When *s* contains no letters, or when at least one letter of *s* is lowercase, `isupper` returns `False`.
join	*s*`.join(`*seq*`)` Returns the string obtained by concatenating the items of *seq* separated by copies of *s* (*e.g.*, `''.join(str(x) for x in range(7))` is `'0123456'` and `'x'.join('aeiou')` is `'axexixoxu'`).
ljust	*s*`.ljust(`*n*, *fillchar*`=' ')` Returns a string of length `max(len(`*s*`),`*n*`)`, with a copy of *s* at the start, followed by zero or more trailing copies of character *fillchar*.
lower	*s*`.lower()` Returns a copy of *s* with all letters, if any, converted to lowercase.
lstrip	*s*`.lstrip(`*x*`=string.whitespace)` Returns a copy of *s* after removing any leading characters found in string *x*. For example, `'banana'.lstrip('ab')` returns `'nana'`.
removeprefix	||*3.9++*||*s.removeprefix(prefix)* When *s* begins with *prefix* returns the remainder of *s*, otherwise returns *s*
removesuffix	||*3.9++*||*s.removesuffix(suffix)* When *s* ends with *suffix* returns the rest of *s*, otherwise returns *s*
replace	*s*`.replace(`*old*,*new*,*count*`=sys.maxsize)` Returns a copy of *s* with the first *count* (or fewer, if there are fewer) nonoverlapping occurrences of substring *old* replaced by string *new* (e.g., `'banana'.replace('a', 'e', 2)` returns `'benena'`).
rfind	*s*`.rfind(`*sub*,*start*`=0`,*end*`=sys.maxsize)` Returns the highest index in *s* where substring *sub* is found, such that *sub* is entirely contained in *s*`[`*start*`:`*end*`]`. `rfind` returns `-1` if *sub* is not found.
rindex	*s*`.rindex(`*sub*,*start*`=0`,*end*`=sys.maxsize)` Like `rfind`, but raises `ValueError` if *sub* is not found.
rjust	*s*`.rjust(`*n*,*fillchar*`=' ')` Returns a string of length `max(len(`*s*`),`*n*`)`, with a copy of *s* at the end, preceded by zero or more leading copies of character *fillchar*.
rstrip	*s*`.rstrip(`*x*`=string.whitespace)` Returns a copy of *s*, removing trailing characters that are found in string *x*.

For example, `'banana'.rstrip('ab')` returns `'banan'`.

split	*s*`.split(`*sep*`=None,`*maxsplit*`=sys.maxsize)`
	Returns a list *L* of up to *maxsplit*`+1` strings. Each item of *L* is a "word" from *s*, where string *sep* separates words. When *s* has more than *maxsplit* words, the last item of *L* is the substring of *s* that follows the first *maxsplit* words. When *sep* is `None`, any string of whitespace separates words (e.g., `'four score and seven years'.split(None,3)` is `['four','score','and','seven years']`).
	Note the difference between splitting on `None` (any string of whitespace is a separator) and splitting on `' '` (each single space character, *not* other whitespace such as tabs and newlines, and *not* strings of spaces, is a separator). For example:
	`>>> x = 'a  b' # two spaces between a and b`
	`>>> x.split() # or x.split(None) ['a', 'b']`
	`>>> x.split(' ') ['a', '', 'b']`
	In the first case, the two-spaces string in the middle is a single separator; in the second case, each single space is a separator, so that there is an empty string between the two spaces.

splitlines	*s*`.splitlines(`*keepends*`=False)` Like *s*`.split('\n')`. When *keepends* is true, however, the trailing `'\n'` is included in each item of the resulting list (except the last one, if *s* does not end with `'\n'`).
startswith	*s*`.startswith(`*prefix*`,`*start*`=0,`*end*`=sys.maxsize)` Returns `True` when *s*`[`*start*`:`*end*`]` starts with string *prefix*; otherwise, `False`. *prefix* can be a tuple of strings, in which case `startswith` returns `True` when *s*`[`*start*`:`*end*`]` starts with any one of them.
strip	*s*`.strip(`*x*`=string.whitespace)` Returns a copy of *s*, removing both leading and trailing characters that are found in string *x*. For example, `'banana'.strip('ab')` is `'nan'`.
swapcase	*s*`.swapcase()` Returns a copy of *s* with all uppercase letters converted to lowercase and vice versa.
title	*s*`.title()` Returns a copy of *s* transformed to titlecase: a capital letter at the start of each contiguous sequence of letters, with all other letters (if any) lowercase.

translate	*s*.translate(*table,delete=b''*) Returns a copy of *s* where characters found in *table* are translated or deleted. When *s* is a str, you cannot pass argument *delete*; *table* is a dict whose keys are Unicode ordinals; values are Unicode ordinals, Unicode strings, or None (to delete the corresponding character)—for example: `tbl = {ord('a'):None, ord('n'):'ze'}` `print('banana'.translate(tbl))` *# prints: 'bzeze'* When the value of *s* is bytes, *table* is a bytes object of length 256; the result of *s*.translate(*t*, *d*) is a bytes object with each item *b* of *s* omitted if *b* is one of the items of *delete*, otherwise changed to *t*[ord(*b*)]. Each of bytes and str have a class method named maketrans which you can use to build tables suitable for the respective translate methods.
upper	*s*.upper() Returns a copy of *s* with all letters, if any, converted to uppercase.

a Note that this does not include the punctuation marks used as a radix, such as dot (.) and comma (,).

The string Module

The string module supplies several useful string attributes:

`ascii_letters`
The string `ascii_lowercase+ascii_uppercase`

`ascii_lowercase`
The string `'abcdefghijklmnopqrstuvwxyz'`

`ascii_uppercase`
The string `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`

`digits`
The string `'0123456789'`

`hexdigits`
The string `'0123456789abcdefABCDEF'`

`octdigits`
: The string `'01234567'`

`punctuation`
: The string `'!"#$%&\'()*+,-./:;<=>?@[\]^_'{|}~'` (i.e., all ASCII characters that are deemed punctuation characters in the `'C'` locale; does not depend on which locale is active)

`printable`
: The string of those ASCII characters that are deemed printable (i.e., digits, letters, punctuation, and whitespace)

`whitespace`
: A string containing all ASCII characters that are deemed whitespace: at least space, tab, linefeed, and carriage return, but more characters (e.g., certain control characters) may be present, depending on the active locale

You should not rebind these attributes; the effects of doing so are undefined, since other parts of the Python library may rely on them.

The module `string` also supplies the class `Formatter`, covered in "String Formatting".

String Formatting

Python provides a flexible mechanism for formatting strings (but *not* byte strings: for those, see "Legacy String Formatting with %"). A *format string* is simply a string containing *replacement fields* enclosed in braces ({}), made up of a *value part*, a *conversion part* and a *format specifier*.

```
{value-part!conversion-part:format-specifier}
```

The value part differs depending on the string type.

- For formatted string literals, the value part is evaluated as a Python expression (see “Formatted string literals”); expressions cannot end in an exclamation mark.

- For other strings the value part selects an argument or an element of an argument to the `format` method.

The optional conversion part is an exclamation mark (**!**) followed by one of the letters **s**, **r**, or **a**.

The optional format specifier begins with a colon (**:**) and determines how the converted value is rendered for interpolation in the format string in the place of the original replacement field.

Here’s a simple formatted string literal example. Notice that text surrounding the replacement fields is copied through literally into the result:

```
>>> n = 10; s = 'zero', 'one', 'two', 'three'; i=2
>>> f'start {"-"*n} : {s[i]} end'
'start ---------- : two end'
```

For other strings, the formatting operation is performed by a call to the string’s `format` method. In these cases the replacement field begins with a value part that selects an argument of the call. You can specify both positional and named arguments. A simple `format` method call is shown below:

```
>>> "This is a {0}, {1}, type of {type}".format("large", "green",
type="vase")
'This is a large, green type of vase'
```

For simplicity, none of the replacement fields above contain a conversion part or a format.

Values by expression evaluation

Because these expressions occur inside formatted string literals, take care to avoid syntax errors when attempting to use value part expressions that themselves contain string quotes. With four different string quotes plus the ability to use escape sequences most things are possible, though admittedly readability can suffer.

Values by argument lookup

The argument selection mechanism can handle positional and named arguments. The simplest replacement field is the empty pair of braces ({}), representing an *automatic* positional argument specifier. Each such replacement field automatically refers to the value of the next positional argument to `format`:

```
>>> 'First: {} second: {}'.format(1, 'two')
'First: 1 second: two'
```

To repeatedly select an argument, or use it out of order, use the argument's number to specify its position in the list of arguments (counting from zero):

```
>>> 'Second: {1}, first: {0}'.format(42, 'two')
'Second: two, first: 42'
```

You cannot mix automatic and numbered replacement fields: it's an either-or choice.

For named arguments, use argument names and if desired mix them with (automatic or numbered) positional arguments:

```
>>> 'a: {a}, 1st: {}, 2nd: {}, a again: {a}'.format(1, 'two',
a=3)
'a: 3, 1st: 1, 2nd: two, a again: 3'
>>> 'a: {a} first:{0} second: {1} first: {0}'.format(1, 'two',
a=3)
'a: 3 first:1 second: two first: 1'
```

If an argument is a sequence, you can use numeric indexes to select a specific element of the argument as the value to be formatted. This applies

to both positional (automatic or numbered) and named arguments:

```
>>> 'p0[1]: {[1]} p1[0]: {[0]}'.format(('zero', 'one'), ('two',
'three'))
'p0[1]: one p1[0]: two'
>>> 'p1[0]: {1[0]} p0[1]: {0[1]}'.format(('zero', 'one'), ('two',
'three'))
'p1[0]: two p0[1]: one'
>>> '{} {} {a[2]}'.format(1, 2, a=(5, 4, 3))'1 2 3'
```

If an argument is a composite object, you can select its individual attributes as values to be formatted by applying attribute-access dot notation to the argument selector. Here is an example using complex numbers, which have `real` and `imag` attributes that hold the real and imaginary parts, respectively:

```
>>> 'First r: {.real} Second i: {a.imag}'.format(1+2j, a=3+4j)
'First r: 1.0 Second i: 4.0'
```

Indexing and attribute-selection operations can be used multiple times, if required.

Value Conversion

You may apply a default conversion to the value via one of its methods. You indicate this by following any selector with `!s` to apply the object's `__str__` method, `!r` for its `__repr__` method, or `!a` for the `ascii` built-in.

In the presence of a conversion part the converted value replaces the original value in the remainder of the formatting process.

Value Formatting

The formatting of the value (if any further formatting is required) is determined by a final (optional) portion of the replacement field, following a colon (`:`), known as the *format specifier*. The absence of a colon in the replacement field means that the converted value is used with no further

formatting. Format specifiers may include one or more of the following: *fill, alignment, sign, radix indicator, width, comma separation, precision, type*.

Alignment, with optional (preceding) fill

If alignment is required the formatted value is filled to the correct field width. The default fill character is the space, but an alternative fill character (which may not be an opening or closing brace) can, if required, precede the alignment indicator. See Table 8-2.

Table 5-2. Alignment indicators

Character	Significance as alignment indicator
'<'	Align value on left of field
'>'	Align value on right of field
'^'	Align value in the center of the field
'='	Only for numeric types: add fill characters between the sign and the first digit of the numeric value

When no alignment is specified, most values are left-aligned, except that numeric values are right-aligned. Unless a field width is specified later in the format specifier, no fill characters are added, whatever the fill and alignment may be.

Optional sign indication

For numeric values only, you can indicate how positive and negative numbers are differentiated by optionally including a sign indicator. See Table 8-3.

Table 5-3. Sign indicators

Character	Significance as sign indicator
`'+'`	Insert `'+'` as sign for positive numbers; `'-'` as sign for negative numbers
`'-'`	Insert `'-'` as sign for negative numbers; do not insert any sign for positive numbers (default behavior if no sign indicator is included)
`' '`	Insert `' '` as sign for positive numbers; `'-'` as sign for negative numbers

Radix indicator

For numeric integer formats only, you can include a radix indicator, the `'#'` character. If present, this indicates that the digits of binary-formatted numbers are preceded by `'0b'`, those of octal-formatted numbers by `'0o'`, and those of hexadecimal-formatted numbers by `'0x'`. For example, `'{:x}'.format(23)` is `'17'`, while `'{:#x}'.format(23)` is `'0x17'`.

Field width

You can specify the width of the field to be printed. If the width specified is less than the length of the value, the length of the value is used (no truncation). If alignment is not specified, the value is left-justified (except numbers, which are right-justified):

```
>>> s = 'a string'
>>> '{^i2s}'.format(s)' a string '
>>> '{:.>12s}'.format(s)'....a string'
```

The field width can be a format argument too:

```
>>> '{:.>{}s}'.format(s, 20)
'............a string'
```

Digit grouping

For numeric values only in decimal (default) format type, you can insert either a comma (,) or an underscore (_) to request that each group of three digits in the result be separated by that character. For example:

```
print('{:,}'.format(12345678))# prints 12,345,678
```

This behavior ignores system locale; for a locale-aware use of appropriate digit grouping and decimal point character, see format type `'n'` in Table 8-4.

Precision specification

The precision (e.g., `.2`) has different meanings for different format types (see the following section), with `.6` as the default for most numeric formats. For the `f` and `F` format types, it specifies the number of *decimal* digits to which the value should be rounded in formatting; for the `g` and `G` format types, it specifies the number of *significant* digits to which the value should be rounded; for non-numeric values, it specifies *truncation* of the value to its leftmost characters before formatting.

```
>>> s = 'a string'
>>> x = 1.12345
>>> 'as f: {:.4f}'.format(x)
'as f: 1.1235'
>>> 'as g: {:.4g}'.format(x)
'as g: 1.123'
>>> 'as s: {:.6s}'.format(s)
'as s: a stri'
```

Format type

The format specification ends with an optional *format type*, which determines how the value gets represented in the given width and at the given precision. When the format type is omitted, the value being formatted applies a default format type.

The `s` format type is used to format Unicode strings.

Integer numbers have a range of acceptable format types, listed in Table 8-4.

Table 5-4. Table caption to come

Format type	Formatting description
`'b'`	Binary format—a series of ones and zeros
`'c'`	The Unicode character whose ordinal value is the formatted value
`'d'`	Decimal (the default format type)
`'o'`	Octal format—a series of octal digits
`'x'` or `'X'`	Hexadecimal format—a series of hexadecimal digits, with the letters, respectively, in lower- or uppercase
`'n'`	Decimal format, with locale-specific separators (commas in the UK and US) when system locale is set

Floating-point numbers have a different set of format types, shown in Table 8-5.

Table 5-5. Table caption to come

Format type	Formatting description
`'e'` or `'E'`	Exponential format—scientific notation, with an integer part between one and nine, using `'e'` or `'E'` just before the exponent
`'f'` or `'F'`	Fixed-point format with infinities (`'inf'`) and nonnumbers (`'nan'`) in

	lower- or uppercase
`'g'` or `'G'`	General format—uses a fixed-point format when possible, otherwise exponential format; uses lower- or uppercase representations for `'e'`, `'inf'`, and `'nan'`, depending on the case of the format type
`'n'`	Like general format, but uses locale-specific separators, when system locale is set, for groups of three digits and decimal points
`'%'`	Percentage format—multiplies the value by 100 and formats it as a fixed-point followed by `'%'`

When no format type is specified, a `float` uses the `'g'` format, with at least one digit after the decimal point and a default precision of 12.

```
>>> n = [3.1415, -42, 1024.0]
>>> for num in n:
...     '{:>+9,.2f}'.format(num)
...
'    +3.14'
'   -42.00'
'+1,024.00'
```

Nested format specifications

In some cases you want to include an argument to `format` to help determine the precise format of another argument: you can use nested formatting to achieve this. For example, to format a string in a field four characters wider than the string itself, you can pass a value for the width to `format`, as in:

```
>>> s = 'a string'
>>> '{0:>{1}s}'.format(s, len(s)+4)
'    a string'
>>> '{0:_^{1}s}'.format(s, len(s)+4)
'__a string__'
```

With some care, you can use width specification and nested formatting to print a sequence of tuples into well-aligned columns. For example:

```
def columnar_strings(str_seq, widths):
    for cols in str_seq:
        row = ['{c:{w}.{w}s}'.format(c=c, w=w)
               for c, w in zip(cols, widths)]
        print(' '.join(row))
```

(`'{c:{w}.{w}s}'.format(c=c, w=w)` can be simplified to `f'{c:{w}.{w}s}'`, as covered in "Formatted String Literals".) Given this function, the following code:

```
c = [
        'four score and'.split(),
        'seven years ago'.split(),
        'our forefathers brought'.split(),
        'forth on this'.split(),
    ]
print(columnar_strings(c, (8, 8, 8)))
```

prints:

```
four     score    and
seven    years    ago
our      forefathers brought
forth    on       this
```

Formatting of user-coded classes

Values are ultimately formatted by a call to their `__format__` method with the format specifier as an argument. Built-in types either implement their own method or inherit from `object`, whose `format` method only accepts an empty string as an argument.

```
>>> object().__format__('')
'<object object at 0x110045070>'
>>> math.pi.__format__('18.6')
'           3.14159'
```

You can use this knowledge to implement an entirely different formatting mini-language of your own, should you so choose. The following simple example demonstrates the passing of format specifications and the return of a (constant) formatted string result. The interpretation of the format

specification is under your control, and you may choose to implement whatever formatting notation you choose.

```
>>> class S:
...     def __init__(self, value):
...         self.value = value
...     def __format__(self, fstr):
...         match fstr:
...             case "U":
...                 return self.value.upper()
...             case 'L':
...                 return self.value.lower()
...             case 'T':
...                 return self.value.title()
...             case _:
...                 return ValueError(f'Unrecognised format code
{fstr!r}')
...
>>> my_s = S('random string')
>>> f'{my_s:L}, {my_s:U}, {my_s:T}'
'random string, RANDOM STRING, Random String'
```

The return value of the `__format__` method is substituted for the replacement field in the output of the call to `format`, allowing any desired interpretation of the format string.

To help you format your objects more easily, the `string` module provides a `Formatter` class with many helpful methods for handling formatting tasks. See the documentation for `Formatter` in the online docs.

Formatted String Literals

This feature helps use the formatting capabilities just described. It uses the same formatting syntax, but lets you specify expression values inline rather than through parameter substitution. Instead of argument specifiers, *f-strings* use expressions, evaluated and formatted as specified. For example, instead of:

```
>>> name = 'Dawn'
>>> print('{name!r} is {l} characters long'
          .format(name=name, l=len(name)))
'Dawn' is 4 characters long
```

you can use the more concise form:

```
>>> print(f'{name!r} is {len(name)} characters long')
'Dawn' is 4 characters long
```

You can use nested braces to specify components of formatting expressions:

```
>>> for width in 8, 11:
...     for precision in 2, 3, 4, 5:
...         print(f'{3.14159:{width}.{precision}}')
...
     3.1
    3.14
   3.142
  3.1416
        3.1
       3.14
      3.142
     3.1416
```

Do remember, though, that these string literals are *not* constants—they evaluate each time a statement containing them runs, causing runtime overhead.

Debug printing with formatted string literals

||3.8++|| As a convenience for debugging, the last non-blank character of the value expression in a formatted string literal can be followed by an equals sign (=), optionally surrounded by spaces. In this case the text of the expression itself and the equals sign, including any leading and trailing spaces, is output before the value. If no format is specified the interpreter uses the **repr()** of the value as output, otherwise the **str()** of the value is used unless a **!r** value conversion is specified.

```
>>> f'{a*s=}'
"a*s='*-*-*-*-*-*-*-*-*-*-'"
>>> f'{a*s = :30}'
'a*s = *-*-*-*-*-*-*-*-*-*-      '
```

Note that this form is *only* available in formatted string literals.

Legacy String Formatting with %

A legacy form of string formatting expression in Python has the syntax:

```
format % values
```

where *format* is a string, bytes or bytearray containing format specifiers and *values* are the values to format, usually as a tuple (in this book we cover only the subset of this legacy feature, the format specifier, that you must know to properly use the logging module, covered in "The logging package"). Unlike Python's newer formatting capabilities, you can use %-formatting with bytes and bytearray objects.

The equivalent use in `logging` would be, for example:

```
logging.info(format, *values)
```

with the *values* coming as positional arguments after the first, *format* one.

The legacy string-formatting approach has roughly the same set of features as the C language's `printf` and operates in a similar way. Each format specifier is a substring of *format* that starts with a percent sign (`%`) and ends with one of the conversion characters shown in Table 8-6.

Table 5-6. String-formatting conversion characters

Character	Output format	Notes
`d`, `i`	Signed decimal integer	Value must be a number.
`u`	Unsigned decimal integer	Value must be a number.
`o`	Unsigned octal integer	Value must be a number.

`x`	Unsigned hexadecimal integer (lowercase letters)	Value must be a number.
`X`	Unsigned hexadecimal integer (uppercase letters)	Value must be a number.
`e`	Floating-point value in exponential form (lowercase `e` for exponent)	Value must be a number.
`E`	Floating-point value in exponential form (uppercase `E` for exponent)	Value must be a number.
`f`, `F`	Floating-point value in decimal form	Value must be a number.
`g`, `G`	Like `e` or `E` when *exp* is `>=4` or < precision; otherwise, like `f` or `F`	*exp* is the exponent of the number being converted.
`a`	String	Converts any value with `ascii`.
`r`	String	Converts any value with `repr`.
`s`	String	Converts any value with `str`.
`%`	Literal `%` character	Consumes no value.

The `a`, `r` and `s` conversion characters are the ones most often used with the `logging` module. Between the `%` and the conversion character, you can specify a number of optional modifiers, as we'll discuss shortly.

What is logged with a formatting expression is *format*, where each format specifier is replaced by the corresponding item of *values* converted to a string according to the specifier. Here are some simple examples:

```
import logging
logging.getLogger().setLevel(logging.INFO)
x = 42
y = 3.14
z = 'george'
logging.info('result = %d', x)         # logs: result = 42
logging.info('answers: %d %f', x, y)   # logs: answers: 42
3.140000
logging.info('hello %s', z)            # logs: hello george
```

Format Specifier Syntax

A format specifier can include modifiers to control how the corresponding item in *values* is converted to a string. The components of a format specifier, in order, are:

- The mandatory leading `%` character that marks the start of the specifier
- Zero or more optional conversion flags:

  — `#` The conversion uses an alternate form (if any exists for its type).

  — `0` The conversion is zero-padded.

  — `-` The conversion is left-justified.

  — *A space* Negative numbers are signed, a space is placed before a positive number.

  — `+` A numeric sign (+ or −) is placed before any numeric conversion.
- An optional minimum width of the conversion: one or more digits, or an asterisk (`*`), meaning that the width is taken from the next item in *values*
- An optional precision for the conversion: a dot (`.`) followed by zero or more digits, or by a `*`, meaning that the precision is taken from the next item in *values*
- A mandatory conversion type from Table 8-6

Each format specifier corresponds to an item in *values* by position, and there must be exactly as many *values* as *format* has specifiers (plus one extra for each width or precision given by `*`). When a width or precision is given by `*`, the `*` consumes one item in *values*, which must be an integer and is taken as the number of characters to use as width or precision of that conversion.

When to use %r (or %a)

Most often, the format specifiers in your *format* string are all `%s`; occasionally, you'll want to ensure horizontal alignment on the output (for example, in a right-justified, maybs-truncated space of exactly 6 characters, in which case you'd use `%6.6s`). However, there is an important special case for `%r` or `%a`.

> **Always Use %r (or %a) to Log Possibly Erroneous Strings**
>
> When you're logging a string value that might be erroneous (for example, the name of a file that is not found), don't use `%s`: when the error is that the string has spurious leading or trailing spaces, or contains some nonprinting characters such as `\b`, `%s` can make this hard for you to spot by studying the logs. Use `%r` or `%a` instead, so that all characters are clearly shown, possibly via escape sequences.

Text Wrapping and Filling

The `textwrap` module supplies a class and a few functions to format a string by breaking it into lines of a given maximum length. To fine-tune the filling and wrapping, you can instantiate the `TextWrapper` class supplied by `textwrap` and apply detailed control. Most of the time, however, one of the these functions exposed by `textwrap` suffices:

`wrap(`*s*`,`*width*`=70)`

wrap Returns a list of strings (without terminating newlines), each no longer than

	width characters. *s*`.wrap` also supports other named arguments (equivalent to attributes of instances of class `TextWrapper`); for such advanced uses, see the online docs.
fill	`fill(`*s*`,`*width*`=70)` Returns a single multiline string equal to `'\n'.join(wrap(`*s*`,`*width*`))`.
dedent	`dedent(s)` Takes a multiline string and returns a copy in which all lines have had the same amount of leading whitespace removed, so that some lines have no leading whitespace.

The pprint Module

The `pprint` module pretty-prints complicated data structures, with formatting that strives to be more readable than that supplied by the built-in function `repr` (covered in Table 7-2). To fine-tune the formatting, you can instantiate the `PrettyPrinter` class supplied by `pprint` and apply detailed control, helped by auxiliary functions also supplied by `pprint`. Most of the time, however, one of two functions exposed by `pprint` suffices:

pformat	`pformat(`*obj*`)` Returns a string representing the pretty-printing of *obj*.

pprint	`pprint(`*obj*`, `*stream*`=sys.stdout)`
	Outputs the pretty-printing of *obj* to open-for-writing file object *stream*, with a terminating newline.
	The following statements do exactly the same thing:
	`print(pprint.pformat(x))` `pprint.pprint(x)`
	Either of these constructs is roughly the same as `print(`*x*`)` in many cases, for example for a container that can be displayed within a single line. However, with something like *x*`=list(range(30))`, `print(`*x*`)` displays *x* in two lines, breaking at an arbitrary point, while using the module `pprint` displays `x` over 30 lines, one line per item. Use `pprint` when you prefer the module's specific display effects to the ones of normal string representation.

The reprlib Module

The `reprlib` module supplies an alternative to the built-in function `repr` (covered in Table 7-2), with limits on length for the representation string. To fine-tune the length limits, you can instantiate or subclass the `Repr` class supplied by the module and apply detailed control. Most of the time, however, the function exposed by the module suffices.

`repr(`*obj*`)`

repr Returns a string representing *obj*, with sensible limits on length.

Unicode

To convert bytestrings into Unicode strings use the `decode` method of bytestrings. The conversion must always be explicit, and is performed using an auxiliary object known as a *codec* (short for *coder-decoder*). A codec can also convert Unicode strings to bytestrings using the `encode` method of strings. To identify a codec, pass the codec name to `decode`, or `encode`. When you pass no codec name Python uses a default encoding, normally `'utf8'`.

Every conversion has a parameter *errors*, a string specifying how conversion errors are to be handled. Sensibly, the default is `'strict'`, meaning any error raises an exception.

When *errors* is `'replace'`, the conversion replaces each character causing errors with `'?'` in a bytestring result, with `u'\ufffd'` in a Unicode result. When *errors* is `'ignore'`, the conversion silently skips characters causing errors. When *errors* is `'xmlcharrefreplace'`, the conversion replaces each character causing errors with the XML character reference representation of that character in the result. You may code your own function to implement a conversion error handling strategy and register it under an appropriate name by calling `codecs.register_error`, covered in Table 8-7 below.

The codecs Module

The mapping of codec names to codec objects is handled by the `codecs` module. This module also lets you develop your own codec objects and register them so that they can be looked up by name, just like built-in codecs. The `codecs` module also lets you look up any codec explicitly, obtaining the functions the codec uses for encoding and decoding, as well as factory functions to wrap file-like objects. Such advanced facilities are rarely used, and we do not cover them in this book.

The `codecs` module, together with the `encodings` package of the standard Python library, supplies built-in codecs useful to Python developers dealing with internationalization issues. Python comes with over 100 codecs; a list of these codecs, with a brief explanation of each, is in the online docs. It's *not* good practice to install a codec as the site-wide default in the module `sitecustomize`; rather, the preferred usage is to always specify the codec by name whenever converting between byte and Unicode strings. A popular codec in Western Europe is `'latin-1'`, a fast, built-in implementation of the ISO 8859-1 encoding that offers a one-byte-per-character encoding of special characters found in Western European languages; beware that it lacks the Euro currency character `'€'`-- if you need that, use `'iso8859-15'`.

The `codecs` module also supplies codecs implemented in Python for most ISO 8859 encodings, with codec names from `'iso8859-1'` to `'iso8859-15'`. On Windows systems only, the codec named `'mbcs'` wraps the platform's multibyte character set conversion procedures. The `codecs` module also supplies various code pages with names from `'cp037'` to `'cp1258'`, and Unicode standard encodings `'utf-8'` (likely to be most often the best choice, thus recommended, and the default) and `'utf-16'` (which has specific big-endian and little-endian variants: `'utf-16-be'` and `'utf-16-le'`). For use with UTF-16, `codecs` also supplies attributes `BOM_BE` and `BOM_LE`, byte-order marks for big-endian and little-endian machines, respectively, and `BOM`, the byte-order mark for the current platform.

The `codecs` module also supplies a function to let you register your own conversion-error-handling functions, as described in Table 8-7.

Table 5-7. Table caption to come

register_error	`register_error(`*name, func*`)` *name* must be a string. *func* must be callable with one argument *e* that's an instance of exception `UnicodeDecodeError`, and must return a tuple with two items: the Unicode string to insert in the converted-string result and the index from which to continue the conversion (the latter is normally *e*`.end`). The function's body can use *e*`.encoding`, the name of the codec of this conversion, and *e*`.object[`*e*`.start:`*e*`.end]`, the substring that caused the conversion error.

The unicodedata Module

The `unicodedata` module supplies easy access to the Unicode Character Database. Given any Unicode character, you can use functions supplied by `unicodedata` to obtain the character's Unicode category, official name (if any), and other relevant information. You can also look up the Unicode character (if any) that corresponds to a given official name.

```
>>> unicodedata.name("🌈" 'RAINBOW')
>>> unicodedata.name("VI")
'ROMAN NUMERAL SIX'
>>> int("VI")
ValueError: invalid literal for int() with base 10: 'VI'
>>> unicodedata.numeric("VI")  # use unicodedata to get the
numeric value
6.0
>>> unicodedata.lookup("MUSICAL SCORE")
§'🎼'
```

Chapter 6. Regular Expressions

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

Regular expressions (REs) let you specify pattern strings and perform searches and substitutions. Regular expressions are not easy to master, but they can be a powerful tool for processing text. Python offers rich regular expression functionality through the built-in `re` module. In this chapter, we thoroughly present all of Python's REs.

Regular Expressions and the re Module

A *regular expression* (RE) is built from a string that represents a pattern. With RE functionality, you can examine any string and check which parts of the string, if any, match the pattern.

The `re` module supplies Python's RE functionality. The `compile` function builds an RE object from a pattern string and optional flags. The methods of an RE object look for matches of the RE in a string or perform substitutions. The `re` module also exposes functions equivalent to an RE object's methods, but with the RE's pattern string as the first argument.

REs can be difficult to master, and this book does not purport to teach them; we cover only the ways in which you can use REs in Python. For general coverage of REs, we recommend the book *Mastering Regular Expressions*,

by Jeffrey Friedl (O'Reilly), offering thorough coverage of REs at both tutorial and advanced levels. Many tutorials and references on REs can also be found online, including an excellent, detailed tutorial in Python's online docs. Sites like Pythex and regex101 let you test your REs interactively.

REs and bytes Versus str

REs in Python work in two ways, depending on the type of the object being matched: when applied to `str` instances, an RE matches accordingly (for example, a Unicode character *c* is deemed to be "a letter" if `'LETTER' in unicodedata.name(`*c*`)`; when applied to `bytes` instances, an RE matches in terms of ASCII (for example, a byte *c* is deemed to be "a letter" if *c* `in string.ascii_letters`). For example:

```
import re
print(re.findall(r'\w+', 'cittá'))# prints ['cittá']
print(re.findall(rb'\w+', 'cittá'.encode())) # prints [b'citt']
```

Pattern-String Syntax

The pattern string representing a regular expression follows a specific syntax:

- Alphabetic and numeric characters stand for themselves. An RE whose pattern is a string of letters and digits matches the same string.
- Many alphanumeric characters acquire special meaning in a pattern when they are preceded by a backslash (\).
- Punctuation works the other way around: self-matching when escaped, special meaning when unescaped.
- The backslash character is matched by a repeated backslash (i.e., pattern \\).

Since RE patterns often contain backslashes, it's best to always specify them using raw string literal form (covered in "Strings"). Pattern elements

(such as `r'\t'`, equivalent to the string literal `'\\t'`) do match the corresponding special characters (in this case, the tab character `'\t'`); so, you can use a raw string literal even when you need a literal match for such special characters.

Table 9-1 lists the special elements in RE pattern syntax. The exact meanings of some pattern elements change when you use optional flags, together with the pattern string, to build the RE object. The optional flags are covered in “Optional Flags”.

Table 6-1. RE pattern syntax

Element	Meaning
`.`	Matches any single character except `\n` (if `DOTALL`, also matches `\n`)
`^`	Matches start of string (if `MULTILINE`, also matches right after `\n`)
`$`	Matches end of string (if `MULTILINE`, also matches right before `\n`)
`*`	Matches zero or more cases of the previous RE; greedy (match as many as possible)
`+`	Matches one or more cases of the previous RE; greedy (match as many as possible)
`?`	Matches zero or one case of the previous RE; greedy (match one if possible)
`*?`, `+?`, `??`	Non-greedy versions of `*`, `+`, and `?`, respectively (match as few as possible)
{m}	Matches *m* cases of the previous RE
{*m*, *n*}	Matches between *m* and *n* cases of the previous RE; *m* or *n* (or both) may be omitted, defaulting to *m=0* and *n=infinity* (greedy)
{*m*, *n*}?	Matches between *m* and *n* cases of the previous RE (non-greedy)
	Matches any one of a set of characters contained within the brackets

`[...]`	
`[^...]`	Matches one character *not* contained within the brackets after the caret `^`
`|`	Matches either the preceding RE or the following RE
`(...)`	Matches the RE within the parentheses and indicates a *group*
`(?aiLmsux)`	Alternate way to set optional flags; no effect on match[a]
`(?:...)`	Like `(...)` but does not capture the matched characters in a group
`(?P<`*id*`>...)`	Like `(...)` but the group also gets the name *id*
`(?P=`*id*`)`	Matches whatever was previously matched by group named *id*
`(?#...)`	Content of parentheses is just a comment; no effect on match
`(?=...)`	*Lookahead assertion*: matches if RE ... matches what comes next, but does not consume any part of the string
	Negative lookahead assertion: matches if RE ... does not match what comes

`(?!...)`	next, and does not consume any part of the string
`(?<=...)`	*Lookbehind assertion*: matches if there is a match ending at the current position for RE ... (... must match a fixed length)
`(?<!...)`	*Negative lookbehind assertion*: matches if there is no match ending at the current position for RE ... (... must match a fixed length)
\ *number*	Matches whatever was previously matched by group numbered *number* (groups are automatically numbered left to right, from 1 to 99)
`\A`	Matches an empty string, but only at the start of the whole string
`\b`	Matches an empty string, but only at the start or end of a word (a maximal sequence of alphanumeric characters; see also `\w`)
`\B`	Matches an empty string, but not at the start or end of a word
`\d`	Matches one digit, like the set `[0-9]` (in Unicode mode, many other Unicode characters also count as “digits” for `\d`, but not for `[0-9]`)
`\D`	Matches one non-digit, like the set `[^0-9]` (in Unicode mode, many other Unicode characters also count as “digits” for `\D`, but not for `[^0-9]`)
`\s`	Matches a whitespace character, like the set `[\t\n\r\f\v]`
`\S`	Matches a non-whitespace character, like the set `[^\t\n\r\f\v]`
`\w`	Matches one alphanumeric character; unless in Unicode mode, or `LOCALE` or `UNICODE` is set, `\w` is like `[a-zA-Z0-9_]`
`\W`	Matches one non-alphanumeric character, the reverse of `\w`
`\Z`	Matches an empty string, but only at the end of the whole string

\\	Matches one backslash character

a Always place the (?...) construct for setting flags, if any, at the start of the pattern, for readability; placing it elsewhere raises a DeprecationWarning.

Using a `'\'` character followed by an alphabetic character not listed here or in Table 3-1 raises a `re.error` exception.

Common Regular Expression Idioms

Always Use r'...' Syntax for RE Pattern Literals

Use raw string literals for all RE pattern literals, and for them only: this ensures you'll never forget to escape a backslash (\), and improves code readability as it makes your RE pattern literals stand out.

`.*` as a substring of a regular expression's pattern string means "any number of repetitions (zero or more) of any character." In other words, `.*` matches any substring of a target string, including the empty substring. `.+` is similar, but matches only a nonempty substring. For example:

```
r'pre.*post'
```

matches a string containing a substring `'pre'` followed by a later substring `'post'`, even if the latter is adjacent to the former (e.g., it matches both `'prepost'` and `'pre23post'`). On the other hand:

```
r'pre.+post'
```

matches only if `'pre'` and `'post'` are not adjacent (e.g., it matches `'pre23post'` but does not match `'prepost'`). Both patterns also

match strings that continue after the `'post'`. To constrain a pattern to match only strings that *end* with `'post'`, end the pattern with `\Z`. For example:

```
r'pre.*post\Z'
```

matches `'prepost'`, but not `'preposterous'`.

All of these examples are *greedy*, meaning that they match the substring beginning with the first occurrence of `'pre'` all the way to the *last* occurrence of `'post'`. When you care about what part of the string you match, you may want to specify *nongreedy* matching, meaning to match the substring beginning with the first occurrence of `'pre'` but only up to the *first* following occurrence of `'post'`.

For example, when the string is `'preposterous and post facto'`, the greedy RE pattern `r'pre.*post'` matches the substring `'preposterous and post'`; the nongreedy variant `r'pre.*?post'` matches just the substring `'prepost'`.

Another frequently used element in RE patterns is `\b`, which matches a word boundary. To match the word `'his'` only as a whole word and not its occurrences as a substring in such words as `'this'` and `'history'`, the RE pattern is:

```
r'\bhis\b'
```

with word boundaries both before and after. To match the beginning of any word starting with `'her'`, such as `'her'` itself and `'hermetic'`, but not words that just contain `'her'` elsewhere, such as `'ether'` or `'there'`, use:

```
r'\bher'
```

with a word boundary before, but not after, the relevant string. To match the end of any word ending with `'its'`, such as `'its'` itself and `'fits'`,

but not words that contain `'its'` elsewhere, such as `'itsy'` or `'jujitsu'`, use:

```
r'its\b'
```

with a word boundary after, but not before, the relevant string. To match whole words thus constrained, rather than just their beginning or end, add a pattern element `\w*` to match zero or more word characters. To match any full word starting with `'her'`, use:

```
r'\bher\w*'
```

To match just the first three letters of any word starting with `'her'`, but not the word `'her'` itself, use a negative word boundary `\B`:

```
r'\bher\B'
```

To match any full word ending with `'its'`, including `'its'` itself, use:

```
r'\w*its\b'
```

Sets of Characters

You denote sets of characters in a pattern by listing the characters within brackets (`[]`). In addition to listing characters, you can denote a range by giving the first and last characters of the range separated by a hyphen (`-`). The last character of the range is included in the set, differently from other Python ranges. Within a set, special characters stand for themselves, except `\`, `]`, and `-`, which you must escape (by preceding them with a backslash) when their position is such that, if not escaped, they would form part of the set's syntax. You can denote a class of characters within a set by escaped-letter notation, such as `\d` or `\S`. `\b` in a set means a backspace character (`chr(8)`), not a word boundary. If the first character in the set's pattern, right after the `[`, is a caret (`^`), the set is *complemented*: such a set matches any character *except* those that follow `^` in the set pattern notation.

A frequent use of character sets is to match “a word”, using a definition of which characters can make up a word that differs from `\w`’s default (letters and digits). To match a word of one or more characters, each of which can be an ASCII letter, an apostrophe, or a hyphen, but not a digit (e.g., `"Finnegan-O'Hara"`), use:

```
r"[a-zA-Z'\-]+"
```

Escape a Hyphen that's Part of an RE Character Set, for Readability

It's not strictly necessary to escape the hyphen with a backslash in this case, since its position at the end of the set makes the situation syntactically unambiguous. However, the backslash is advisable because it makes the pattern more readable, by visually distinguishing the hyphen that you want to have as a character in the set from those used to denote ranges.

Alternatives

A vertical bar (`|`) in a regular expression pattern, used to specify alternatives, has low syntactic precedence. Unless parentheses change the grouping, `|` applies to the whole pattern on either side, up to the start or end of the pattern, or to another `|`. A pattern can be made up of any number of subpatterns joined by `|`. It is important to note that an RE of subpatterns joined by | will match the *first* matching subpattern, not the longest. A pattern like `r'ab|abc'` will never match `'abc'`, because the `'ab'` match gets evaluated first.

Given a list *L* of words, an RE pattern that matches any one of the words is:

```
'|'.join(rf'\b{word}\b' for word in L)
```

Escaping Strings

If the items of *L* can be more general strings, not just words, you need to *escape* each of them with the function `re.escape` (covered in Table 9-3), and you may not want the `\b` word boundary markers on either side. In this case, you could use the following RE pattern (sorting the list in reverse order by length to avoid "masking" a longer word by a shorter one):

```
'|'.join(re.escape(s) for s in sorted(L, key=len, reverse=True))
```

Groups

A regular expression can contain any number of groups, from none to 99 (or even more, but only the first 99 groups are fully supported). Parentheses in a pattern string indicate a group. Element `(?P<`*id*`>...)` also indicates a group, and gives the group a name, *id*, that can be any Python identifier. All groups, named and unnamed, are numbered, left to right, 1 to 99; "group 0" means the string that the whole RE matches.

For any match of the RE with a string, each group matches a substring (possibly an empty one). When the RE uses `|`, some groups may not match any substring, although the RE as a whole does match the string. When a group doesn't match any substring, we say that the group does not *participate* in the match. An empty string (`''`) is used as the matching substring for any group that does not participate in a match, except where otherwise indicated later in this chapter. For example:

```
r'(.+)\1+\Z'
```

matches a string made up of two or more repetitions of any nonempty substring. The `(.+)` part of the pattern matches any nonempty substring (any character, one or more times) and defines a group, thanks to the parentheses. The `\1+` part of the pattern matches one or more repetitions of the group, and `\Z` anchors the match to the end of the string.

Optional Flags

A regular expression pattern element with one or more of the letters `aiLmsux` between `(?` and `)` lets you set RE options within the pattern, rather than by the *flags* argument to the `compile` function of the `re` module. Options apply to the whole RE, no matter where the options element occurs in the pattern.

Always Place Options at the Start of an RE's Pattern

In particular, placement at the start is mandatory if *x* is among the options, since *x* changes the way Python parses the pattern. Options not at the start of the pattern produce a deprecation warning.

Using the explicit *flags* argument is more readable than placing an options element within the pattern. The *flags* argument to the function `compile` is a coded integer built by bitwise ORing (with Python's bitwise OR operator, `|`) one or more of the following attributes of the module `re`. Each attribute has both a short name (one uppercase letter), for convenience, and a long name (an uppercase multi-letter identifier), which is more readable and thus normally preferable:

`A` *or* `ASCII`

Uses ASCII-only characters for \w, \W, \b, \B, \d and \D; overrides the default UNICODE flag

`I` *or* `IGNORECASE`

Makes matching case-insensitive

`L` *or* `LOCALE`

Uses the Python LOCALE setting to determine characters for \w, \W, \b, \B, \d and \D markers; can only be used with bytes patterns

`M` *or* `MULTILINE`

Makes the special characters `^` and `$` match at the start and end of each line (i.e., right after/before a newline), as well as at the start and end of the whole string (`\A` and `\Z` always match only the start and end of the whole string)

`S` *or* `DOTALL`

Causes the special character `.` to match any character, including a newline

`U` *or* `UNICODE`

Uses full Unicode to determine characters for \w, \W, \b, \B, \d and \D markers; although retained for backwards compatibility, this flag is now the default

`X` *or* `VERBOSE`

Causes whitespace in the pattern to be ignored, except when escaped or in a character set, and makes a non-escaped # character in the pattern begin a comment that lasts until the end of the line

For example, here are three ways to define equivalent REs with function `compile`, covered in Table 9-3. Each of these REs matches the word "hello" in any mix of upper- and lowercase letters:

```
import re
r1 = re.compile(r'(?i)hello')
r2 = re.compile(r'hello', re.I)
r3 = re.compile(r'hello', re.IGNORECASE)
```

The third approach is clearly the most readable, and thus the most maintainable, though slightly more verbose. The raw string form is not strictly necessary here, since the patterns do not include backslashes. However, using raw string literals does no harm, and we recommend you always do that for RE patterns, and only for RE patterns, to improve clarity and readability.

Option `re.VERBOSE` (or `re.X`) lets you make patterns more readable and understandable by appropriate use of whitespace and comments. Complicated and verbose RE patterns are generally best represented by strings that take up more than one line, and therefore you normally want to

use a triple-quoted raw string literal for such pattern strings. For example, to match a string representing an integer that may be in octal, hex, or decimal format:

```
repat_num1 = r'(0o[0-7]*|0x[\da-fA-F]+|[1-9]\d*)\Z'
repat_num2 = r'''(?x)             # (re.VERBOSE) pattern matching
int literals
               (  0o [0-7]*       # octal: leading 0o, 0+ octal
digits
                | 0x [\da-fA-F]+  # hex: 0x, then 1+ hex digits
                | [1-9] \d*       # decimal: leading non-0, 0+
digits
               )\Z                # end of string
               '''
```

The two patterns defined in this example are equivalent, but the second one is made more readable and understandable by the comments and the free use of whitespace to visually group portions of the pattern in logical ways.

Match Versus Search

So far, we've been using regular expressions to *match* strings. For example, the RE with pattern `r'box'` matches strings such as `'box'` and `'boxes'`, but not `'inbox'`. In other words, an RE match is implicitly anchored at the start of the target string, as if the RE's pattern started with `\A`.

Often, you're interested in locating possible matches for an RE anywhere in the string, without anchoring (e.g., find the `r'box'` match inside such strings as `'inbox'`, as well as in `'box'` and `'boxes'`). In this case, the Python term for the operation is a *search*, as opposed to a match. For such searches, use the `search` method of an RE object; the `match` method matches only from the start. For example:

```
import re
r1 = re.compile(r'box')
if r1.match('inbox'):
    print('match succeeds')
else:
    print('match fails')          # prints: match fails
```

```
if r1.search('inbox'):
    print('search succeeds')      # prints: search succeeds
else:
    print('search fails')
```

Anchoring at String Start and End

`\A` and `\Z` are the pattern elements ensuring that a regular expression match is anchored at the string's start or end. Elements `^` for start and `$` for end are also used in similar roles. For RE objects that are not flagged as MULTILINE, `^` is the same as `\A`, and `$` is the same as `\Z`. For a multiline RE, however, `^` can anchor at the start of the string or the start of any line (where "lines" are determined based on \n separator characters). Similarly, with a multiline RE, `$` can anchor at the end of the string or the end of any line. `\A` and `\Z` always anchor exclusively at the start and end of the string, whether the RE object is multiline or not. For example, here's a way to check whether a file has any lines that end with digits:

```
import re
digatend = re.compile(r'\d$', re.MULTILINE)
with open('afile.txt') as f:
    if digatend.search(f.read()):
        print('some lines end with digits')
    else:
        print('no line ends with digits')
```

A pattern of `r'\d\n'` is almost equivalent, but in that case the search fails if the very last character of the file is a digit not followed by an end-of-line character. With the preceding example, the search succeeds if a digit is at the very end of the file's contents, as well as in the more usual case where a digit is followed by an end-of-line character.

Regular Expression Objects

A regular expression object *r* has the following read-only attributes that detail how *r* was built (by the function `compile` of the module `re`, covered in Table 9-3):

`flags`

The *flags* argument passed to `compile`, or `re.UNICODE` when *flags* is omitted; also includes any flags specified in the pattern itself using a leading `(?...)` element

`groupindex`

A dictionary whose keys are group names as defined by elements `(?P<`*id*`>...)`; the corresponding values are the named groups' numbers

`pattern`

The pattern string from which *r* is compiled

These attributes make it easy to get back from a compiled RE object to its pattern string and flags, so you never have to store those separately.

An RE object *r* also supplies methods to locate matches for *r* within a string, as well as to perform substitutions on such matches (Table 9-2). Matches are generally represented by special objects, covered in "Match Objects".

Table 6-2. Methods of RE objects

findall

r`.findall(`*s*`)`
When *r* has no groups, `findall` returns a list of strings, each a substring of *s* that is a nonoverlapping match with *r*. For example, to print out all words in a file, one per line:

```
import re
reword = re.compile(r'\w+')
with open('afile.txt') as f:
    for aword in reword.findall(f.read()):
        print(aword)
```

When *r* has one group, `findall` also returns a list of strings, but each is the substring of *s* that matches *r*'s group. For example, to print only words that are followed by whitespace (not the ones followed by punctuation or end of

string), you need to change only one statement in the example:
reword = re.compile('(\w+)\s')
When *r* has *n* groups (with *n*>1), `findall` returns a list of tuples, one per non-overlapping match with *r*. Each tuple has *n* items, one per group of *r*, the substring of *s* matching the group. For example, to print the first and last word of each line that has at least two words:

```
import re
first_last =
re.compile(r'^\W*
(\w+)\b.*\b(\w+)\W*$',re.MULTILINE)
with open('afile.txt') as f:
    for first, last in
first_last.findall(f.read()):
        print(first, last)
```

finditer

r`.finditer(`*s*`)`
`finditer` is like `findall`, except that, instead of a list of strings or tuples, it returns an iterator whose items are match objects. In most cases, `finditer` is therefore more flexible and performs better than `findall`.

fullmatch

r`.fullmatch(`*s*`,` *start*`=0,` *end*`=sys.maxsize)`
Returns a match object when the complete substring *s*, starting at index *start* and ending at index *end*, matches *r.* Otherwise, `fullmatch` returns `None`.

match

r`.match(`*s*`,` *start*`=0,` *end*`=sys.maxsize)`
Returns an appropriate match object when a substring of *s*, starting at index *start* and not reaching as far as index *end*, matches *r*. Otherwise, `match` returns `None`. `match` is implicitly anchored at the starting position *start* in *s*. To search for a match with *r* at any point in *s* from *start* onward, call *r*`.search`, not *r*`.match`. For example, here one way to print all lines in a file that start with digits:

```
import re
digs = re.compile(r'\d')
with open('afile.txt') as f:
    for line in f:
        if digs.match(line):
            print(line, end='')
```

search

r`.search(`*s*, *start*`=0`, *end*`=sys.maxsize)`
Returns an appropriate match object for the leftmost substring of *s*, starting not before index *start* and not reaching as far as index *end*, that matches *r*. When no such substring exists, `search` returns `None`. For example, to print all lines containing digits, one simple approach is as follows:

```
import re
digs = re.compile(r'\d')
with open('afile.txt') as f:
    for line in f:
        if digs.search(line):
            print(line, end='')
```

r`.split(`*s*, *maxsplit*=`0)`

split Returns a list *L* of the *splits* of *s* by *r* (i.e., the substrings of *s* separated by nonoverlapping, nonempty matches with *r*). For example, here's one way to eliminate all occurrences of substring `'hello'` (in any mix of lowercase and uppercase) from a string:

```
import re
rehello = re.compile(r'hello', re.IGNORECASE)
astring = ''.join(rehello.split(astring))
```

When *r* has *n* groups, *n* more items are interleaved in *L* between each pair of splits. Each of the *n* extra items is the substring of *s* that matches *r*'s corresponding group in that match, or `None` if that group did not participate in the match. For example, here's one way to remove whitespace only when it occurs between a colon and a digit:

```
import re
re_col_ws_dig = re.compile(r'(:)\s+(\d)')
astring = ''.join(re_col_ws_dig.split(astring))
```

If *maxsplit* is greater than `0`, at most *maxsplit* splits are in *L*, each followed by *n* items as above, while the trailing substring of *s* after *maxsplit* matches of *r*, if any, is *L*'s last item. For example, to remove only the first occurrence of substring `'hello'` rather than all of them, change the last statement in the first example above to:
astring=''.join(rehello.split(astring, 1))

r`.sub(`*repl,s,count*=`0)`

sub Returns a copy of *s* where non-overlapping matches with *r* are replaced by *repl*, which can be either a string or a callable object, such as a function. An empty match is replaced only when not adjacent to the previous match. When *count* is greater than `0`, only the first *count* matches of *r* within *s* are replaced. When *count* equals `0`, all matches of *r* within *s* are replaced. For example, here's another, more natural way to remove only the first occurrence of substring `'hello'` in any mix of cases:

```
import re
rehello = re.compile(r'hello', re.IGNORECASE)
```

```
astring = rehello.sub('', astring, 1)
```

Without the final `1` argument to `sub`, the example removes all occurrences of `'hello'`.

When *repl* is a callable object, *repl* must accept one argument (a match object) and return a string (or `None`, which is equivalent to returning the empty string `''`) to use as the replacement for the match. In this case, `sub` calls *repl*, with a suitable match-object argument, for each match with *r* that `sub` is replacing. For example, here's one way to uppercase all occurrences of words starting with `'h'` and ending with `'o'` in any mix of cases:

```
import re
h_word = re.compile(r'\bh\w*o\b', re.IGNORECASE)
def up(mo):
    return mo.group(0).upper()
astring = h_word.sub(up, astring)
```

When *repl* is a string, `sub` uses *repl* itself as the replacement, except that it expands back references. A *back reference* is a substring of *repl* of the form `\g<`*id*`>`, where *id* is the name of a group in *r* (established by syntax `(?P<`*id*`>...`) in *r*'s pattern string) or `\dd`, where *dd* is one or two digits taken as a group number. Each back reference, named or numbered, is replaced with the substring of *s* that matches the group of *r* that the back reference indicates. For example, here's a way to enclose every word in braces:

```
import re
grouped_word = re.compile('(\w+)')
astring = grouped_word.sub(r'{\1}', astring)
```

subn

r`.subn(`*repl*`,`*s*`,`*count*`=0)`

`subn` is the same as `sub`, except that `subn` returns a pair (*new_string*, *n*), where *n* is the number of substitutions that `subn` has performed. For example, here's one way to count the number of occurrences of substring `'hello'` in any mix of cases:

```
import re
rehello = re.compile(r'hello', re.IGNORECASE)
_, count = rehello.subn('', astring)
print(f'Found {count} occurrences of "hello"')
```

Match Objects

Match objects are created and returned by the methods `match` and `search` of a regular expression object, and are the items of the iterator returned by the method `finditer`. They are also implicitly created by the methods `sub` and `subn` when the argument *repl* is callable, since in that case the appropriate match object is passed as the only argument on each call to *repl*. A match object *m* supplies the following read-only attributes that detail how a search or match created *m*:

`pos`

The *start* argument that was passed to `search` or `match` (i.e., the index into *s* where the search for a match began)

`endpos`

The *end* argument that was passed to `search` or `match` (i.e., the index into *s* before which the matching substring of *s* had to end)

`lastgroup`

The name of the last-matched group (`None` if the last-matched group has no name, or if no group participated in the match)

`lastindex`

The integer index (1 and up) of the last-matched group (`None` if no group participated in the match)

`re`

The RE object *r* whose method created *m*

`string`

The string *s* passed to `finditer`, `match`, `search`, `sub`, or `subn`

A match object *m* also supplies several methods, detailed in Table 9-3.

Table 6-3. Methods of match object

end, span, start	*m*`.end(`*groupid*`=0)` *m*`.span(`*groupid*`=0)` *m*`.start(`*groupid*`=0)` These methods return the limit indices, within *m*`.string`, of the substring that matches the group identified by *groupid* (a group number or name; "group 0", the default value for *groupid*, means "the whole RE"). When the matching substring is *m*`.string[`*i*`:`*j*`]`, *m*`.start` returns *i*, *m*`.end` returns *j*, and *m*`.span` returns (*i*, *j*). If the group did not participate in the match, *i* and *j* are `-1`.
expand	*m*`.expand(`*s*`)` Returns a copy of *s* where escape sequences and back references are replaced in the same way as for the method *r*`.sub`, covered in Table 9-2.
group	*m*`.group(`*groupid*`=0, *`*groupids*`)` When called with a single argument *groupid* (a group number or name), `group` returns the substring that matches the group identified by *groupid*, or `None` if that group did not participate in the match. The idiom *m*`.group()`, also spelled *m*`.group(0)`, returns the whole matched substring, since group number `0` means the whole RE. Groups can also be accessed using `m[index]` notation, as if called using `m.group(index)` (in either case, index may be an `int` or a `str`). When `group` is called with multiple arguments, each argument must be a group number or name. `group` then returns a tuple with one item per argument, the substring matching the corresponding group, or `None` if that group did not participate in the match.
groups	*m*`.groups(`*default*`=None)` Returns a tuple with one item per group in *r*. Each item is the substring that matches the corresponding group, or *default* if that group did not participate in the match. The tuple does not include the 0-group representing the full pattern match.
groupdict	*m*`.groupdict(`*default*`=None)` Returns a dictionary whose keys are the names of all named groups in *r*. The value for each name is the substring that matches the corresponding group, or *default* if that group did not participate in the match.

Functions of the re Module

The `re` module supplies the attributes listed in "Optional Flags". It also provides one function for each method of a regular expression object (`findall`, `finditer`, `fullmatch`, `match`, `search`, `split`, `sub`, and `subn`), each with an additional first argument, a pattern string that the function implicitly compiles into an RE object. It is usually better to

compile pattern strings into RE objects explicitly and call the RE object's methods, but sometimes, for a one-off use of an RE pattern, calling functions of the module `re` can be handier. For example, to count the number of occurrences of `'hello'` in any mix of cases, one concise, function-based way is:

```
import re
_, count = re.subn(r'hello', '', astring, flags=re.I)
print(f'Found {count} occurrences of "hello"')
```

The `re` module internally caches RE objects it creates from the patterns passed to functions; to purge the cache and reclaim some memory, call `re.purge()`.

The `re` module also supplies `error`, the class of exceptions raised upon errors (generally, errors in the syntax of a pattern string), and two more functions (Table 9-4):

Table 6-4. Add legend here.

compile	`compile(`*pattern*`,`*flags*`=0)` Creates and returns an RE object, parsing string *pattern* as per the syntax covered in "Pattern-String Syntax", and using integer *flags*, as covered in "Optional Flags".
escape	`escape(`*s*`)` Returns a copy of string *s* with each nonalphanumeric character escaped (i.e., preceded by a backslash \); useful to match string *s* literally as part of an RE pattern string.

REs and the := operator

The introduction of the `:=` operator in Python 3.8 established support for a successive-match idiom in Python similar to one that's common in Perl. In this idiom, a series of if-elif branches tests a string against different regular expressions. In Perl, the `if ($var =~ /regExpr/)` statement both evaluates the regular expression and saves the successful match in the variable `var`.[1]

```
if    ($statement =~ /I love (\w+)/) {
  print "He loves $1\n";
}
elsif ($statement =~ /Ich liebe (\w+)/) {
  print "Er liebt $1\n";
}
elsif ($statement =~ /Je t\'aime (\w+)/) {
  print "Il aime $1\n";
}
```

Prior to Python 3.8, this behavior of evaluate-and-store was not possible in a single `if-elif` statement; developers had to use a cumbersome cascade of nested `if-else` statements:

```
m = re.match('I love (\w+)', statement)
if m:
    print(f'He loves {m.group(1)}')
else:
    m = re.match('Ich liebe (\w+)', statement)
    if m:
        print(f'Er liebt {m.group(1)}')
    else:
            m = re.match('J'aime (\w+)', statement)
        if m:
                print(f'Il aime {m.group(1)}')
```

Using the `:=` operator, this code simplifies to:

```
if m := re.match(r'I love (\w+)', statement):
    print(f'He loves {m.group(1)}')
elif m := re.match(r'Ich liebe (\w+)', statement):
    print(f'Er liebt {m.group(1)}')
elif m := re.match(r'J'aime (\w+)', statement):
    print(f'Il aime {m.group(1)}')
```

The 3rd party regex module

In addition to Python's built-in `re` module, a popular package for regular expressions is the third-party regex module, by Matthew Barnett. `regex` has a compatible API with the `re` module and adds a number of extended features, including:

Recursive expressions

Applying some of the inline flags to only a part of the pattern

Define character sets by Unicode property/value

Overlapping matches

Fuzzy matching

Multithreading support – releases GIL during matching

Matching timeout

Unicode case folding in case-insensitive matches

Nested sets

---

1 Example taken from regex - Match groups in Python - Stack Overflow

Chapter 7. Time Operations

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

A Python program can handle time in several ways. Time *intervals* are floating point numbers in units of seconds (a fraction of a second is the fractional part of the interval): all standard library functions accepting an argument that expresses a time interval in seconds accept a float as the value of that argument. *Instants* in time are expressed in seconds since a reference instant, known as the *epoch*. (Midnight, UTC, of January 1, 1970, is a popular epoch used on both Unix and Windows platforms.) Time instants often also need to be expressed as a mixture of units of measurement (e.g., years, months, days, hours, minutes, and seconds), particularly for I/O purposes. I/O, of course, also requires the ability to format times and dates into human-readable strings, and parse them back from string formats.

The time Module

The `time` module is somewhat dependent on the underlying system's C library, which determines the range of dates that the `time` module can handle. On Unix systems, years 1970 and 2038 are typical cut-off points, a limitation that `datetime` avoids. Time instants are normally specified in UTC (Coordinated Universal Time, once known as GMT, or Greenwich

Mean Time). The `time` module also supports local time zones and daylight saving time (DST), but only to the extent the underlying C system library does.

As an alternative to seconds since the epoch, a time instant can be represented by a tuple of nine integers, called a *timetuple*. (Timetuples are covered in Table 12-1.) All items are integers: timetuples don't keep track of fractions of a second. A timetuple is an instance of `struct_time`. You can use it as a tuple, and you can also, more usefully, access the items as the read-only attributes *x*`.tm_year`, *x*`.tm_mon`, and so on, with the attribute names listed in Table 12-1. Wherever a function requires a timetuple argument, you can pass an instance of `struct_time` or any other sequence whose items are nine integers in the right ranges (all ranges in the table include both lower and upper bounds; in the table, upper bounds are included).

Table 7-1. Tuple form of time representation

Item	Meaning	Field name	Range	Notes
0	Year	tm_year	1970–2038	Wider on some platforms.
1	Month	tm_mon	1–12	1 is January; 12 is December.
2	Day	tm_mday	1–31	
3	Hour	tm_hour	0–23	0 is midnight; 12 is noon.
4	Minute		0–59	

		`tm_min`		
5	Second	`tm_sec`	0–61	`60` and `61` for leap seconds.
6	Weekday	`tm_wday`	0–6	`0` is Monday; `6` is Sunday.
7	Year day	`tm_yday`	1–366	Day number within the year.
8	DST flag	`tm_isdst`	−1 to 1	−1 means the library determines DST.

To translate a time instant from a "seconds since the epoch" floating-point value into a timetuple, pass the floating-point value to a function (e.g., `localtime`) that returns a timetuple with all nine items valid. When you convert in the other direction, `mktime` ignores redundant items six (`tm_wday`) and seven (`tm_yday`) of the tuple. In this case, you normally set item eight (`tm_isdst`) to −1 so that `mktime` itself determines whether to apply DST.

`time` supplies the functions and attributes listed in Table 12-2.

Table 7-2. Table caption to come

asctime	`asctime([`*tupletime*`])` Accepts a timetuple and returns a readable 24-character string such as `'Sun Jan 8 14:41:06 2017'`. `asctime()` without arguments is like `asctime(localtime(time()))` (formats current time in local time).
ctime	`ctime([`*secs*`])` Like `asctime(localtime(`*secs*`))`, accepts an instant expressed in seconds since the epoch and returns a readable 24-character string form of that instant, in local time. `ctime()` without arguments is like `asctime()` (formats current time in local time).
	`gmtime([`*secs*`])`

gmtime	Accepts an instant expressed in seconds since the epoch and returns a timetuple *t* with the UTC time (*t*.`tm_isdst` is always `0`). `gmtime()` without arguments is like `gmtime(time())` (returns the timetuple for the current time instant).
localtime	`localtime([`*secs*`])` Accepts an instant expressed in seconds since the epoch and returns a timetuple *t* with the local time (*t*.`tm_isdst` is `0` or `1`, depending on whether DST applies to instant *secs* by local rules). `localtime()` without arguments is like `localtime(time())` (returns the timetuple for the current time instant).
mktime	`mktime(`*tupletime*`)` Accepts an instant expressed as a timetuple in local time and returns a floating-point value with the instant expressed in seconds since the epoch.[a] DST, the last item in *tupletime*, is meaningful: set it to `0` to get solar time, to `1` to get DST, or to −1 to let `mktime` compute whether DST is in effect at the given instant.
monotonic	`monotonic()` Like `time()`, returns the current time instant, a `float` with seconds since the epoch. Guaranteed to never go backward between calls, even when the system clock is adjusted (e.g., due to leap seconds).
perf_counter	`perf_counter()` Returns the value in fractional seconds using the highest-resolution clock available to get accuracy for short durations. It is system-wide and *includes* time elapsed during `sleep`. Use only the difference between successive calls, as there is no defined reference point.
process_time	`process_time()` Returns the value in fractional seconds using the highest-resolution clock available to get accuracy for short durations. It is process-wide and *doesn't* include time elapsed during `sleep`. Use only the difference between successive calls, as there is no defined reference point.
sleep	`sleep(`*secs*`)` Suspends the calling thread for *secs* seconds. The calling thread may start executing again before *secs* seconds (when it's the main thread and some signal wakes it up) or after a longer suspension (depending on system scheduling of processes and threads). You can call sleep with `secs=0` to offer other threads a chance to run, incurring no significant delay if the current thread is the only one ready to run.
strftime	`strftime(`*fmt*`[, `*tupletime*`])` Accepts an instant expressed as a timetuple in local time and returns a string representing the instant as specified by string *fmt*. If you omit *tupletime*, `strftime` uses `localtime(time())` (formats the current time instant). The syntax of string *format* is similar to the one covered in "Legacy String Formatting with %." Conversion characters are different, as shown in Table 12-3. Refer to the time instant specified by *tupletime*; the format can't specify width and precision.

	For example, you can obtain dates just as formatted by `asctime` (e.g., `'Tue Dec 10 18:07:14 2002'`) with the format string: '%a %b %d %H:%M:%S %Y' You can obtain dates compliant with RFC 822 (e.g., `'Tue, 10 Dec 2002 18:07:14 EST'`) with the format string: '%a, %d %b %Y %H:%M:%S %Z'
strptime	`strptime(`*str*`,[`*fmt*`='%a %b %d %H:%M:%S %Y'])` Parses *str* according to format string *fmt* and returns the instant as a timetuple. The format string's syntax is as covered in `strftime` earlier.
time	`time()` Returns the current time instant, a `float` with seconds since the epoch. On some (mostly, older) platforms, the precision of this time is as low as one second. May return a lower value in a subsequent call if the system clock is adjusted backward between calls (e.g., due to leap seconds).
timezone	`timezone` The offset in seconds of the local time zone (without DST) from UTC (`>0` in the Americas; `<=0` in most of Europe, Asia, and Africa).
tzname	`tzname` A pair of locale-dependent strings, which are the names of the local time zone without and with DST, respectively.

a mktime's result's fractional part is always 0, since its timetuple argument does not account for fractions of a second.

Table 7-3. Conversion characters for strftime

Type char	Meaning	Special notes
`a`	Weekday name, abbreviated	Depends on locale
`A`	Weekday name, full	Depends on locale
`b`	Month name, abbreviated	Depends on locale
`B`	Month name, full	Depends on locale

c	Complete date and time representation	Depends on locale
d	Day of the month	Between 1 and 31
f	Microsecond as decimal, padded on left	1 to 6 digits
G	ISO 8601:2000 standard week-based year number	
H	Hour (24-hour clock)	Between 0 and 23
I	Hour (12-hour clock)	Between 1 and 12
j	Day of the year	Between 1 and 366
m	Month number	Between 1 and 12
M	Minute number	Between 0 and 59
p	A.M. or P.M. equivalent	Depends on locale
S	Second number	Between 0 and 61
u	day of week	Monday is 1, up to 7
U	Week number (Sunday first weekday)	Between 0 and 53
V	ISO 8601:2000 standard week-based week number	
	Weekday number	0 is Sunday, up to 6

`w`		
`W`	Week number (Monday first weekday)	Between 0 and 53
`x`	Complete date representation	Depends on locale
`X`	Complete time representation	Depends on locale
`y`	Year number within century	Between 0 and 99
`Y`	Year number	1970 to 2038, or wider
z	UTC offset as a string: ±HHMM[SS[.ffffff]]	
`Z`	Name of time zone	Empty if no time zone exists
`%`	A literal % character	Encoded as `%%`

The datetime Module

`datetime` provides classes for modeling date and time objects, which can be either *aware* of time zones or *naive* (the default). The class `tzinfo`, whose instances model a time zone, is abstract: module `datetime` supplies only one simple implementation, `datetime.timezone` (for all the gory details, see the online docs); module `zoneinfo`, covered in "The zoneinfo Module," offers a richer concrete implementation of `tzinfo`, which lets you easily create timezone-aware `datetime` objects. All types in `datetime` have immutable instances: attributes are read-only, instances can be keys in a `dict` or items in a `set`, and all functions and methods return new objects, never altering objects passed as arguments.

The date Class

Instances of the `date` class represent a date (no time of day in particular within that date) between `date.min<=d<=date.max`, are always naive, and assume the Gregorian calendar was always in effect. `date` instances have three read-only integer attributes: `year`, `month`, and `day`:

date	`date(`*year,month,day*`)` Returns a date object for the given *year*, *month*, and *day* arguments, in the valid ranges `1<=`*year*` <=9999`, `1<=`*month*`<=12`, and `1<=`*day*`<=` number of days for the given month and year. Raises *ValueError* if invalid values are given.
The `date` class also supplies these class methods usable as alternative constructors:	
fromordinal	`date.fromordinal(`*ordinal*`)` Returns a `date` object corresponding to the proleptic Gregorian ordinal *ordinal*, where a value of `1` corresponds to the first day of year `1` CE.
fromtimestamp	`date.fromtimestamp(`*timestamp*`)` Returns a `date` object corresponding to the instant *timestamp* expressed in seconds since the epoch.
today	`date.today()` Returns a `date` representing today's date.

Instances of the `date` class support some arithmetic: the difference between `date` instances is a `timedelta` instance; you can add or subtract a `timedelta` to/from a `date` instance to make another `date` instance. You can compare any two instances of the `date` class (the later one is greater).

An instance *d* of the class `date` supplies the following methods:

ctime	*d*`.ctime()` Returns a string representing the date *d* in the same 24-character format as `time.ctime` (with the time of day set to 00:00:00, midnight).
	d`.isocalendar()` Returns a tuple with three integers (ISO year, ISO week number, and ISO

isocalendar	weekday). See the ISO 8601 standard for more details about the ISO (International Standards Organization) calendar.
isoformat	*d*.isoformat() Returns a string representing date *d* in the format 'YYYY-MM-DD'; same as str(*d*).
isoweekday	*d*.isoweekday() Returns the day of the week of date *d* as an integer, 1 for Monday through 7 for Sunday; like d.weekday() + 1.
replace	*d*.replace(*year*=None,*month*=None,*day*=None) Returns a new date object, like *d* except for those attributes explicitly specified as arguments, which get replaced. For example: date(x,y,z).replace(month=m) == date(x,m,z)
strftime	*d*.strftime(*fmt*) Returns a string representing date *d* as specified by string *fmt*, like: time.strftime(*fmt*, *d*.timetuple())
timetuple	*d*.timetuple() Returns a time tuple corresponding to date *d* at time 00:00:00 (midnight).
toordinal	*d*.toordinal() Returns the proleptic Gregorian ordinal for date *d*. For example: date(1,1,1).toordinal() == 1
weekday	*d*.weekday() Returns the day of the week of date *d* as an integer, 0 for Monday through 6 for Sunday; like d.isoweekday() - 1.

The time Class

Instances of the time class represent a time of day (of no particular date), may be naive or aware regarding time zones, and always ignore leap seconds. They have five attributes: four read-only integers (hour, minute, second, and microsecond) and an optional read-only tzinfo (None for naive instances).

time	time(*hour*=0,*minute*=0,*second*=0,*microsecond*=0, *tzinfo*=None) Instances of the class time do not support arithmetic. You can compare two instances of time (the one that's later in the day is greater), but only if they are either both aware or both naive.

An instance *t* of the class `time` supplies the following methods:

isoformat	*t*`.isoformat()` Returns a string representing time *t* in the format `'HH:MM:SS'`; same as `str(`*t*`)`. If *t*`.microsecond!=0`, the resulting string is longer: `'HH:MM:SS.mmmmmm'`. If *t* is aware, six more characters, `'+HH:MM'`, are added at the end to represent the time zone's offset from UTC. In other words, this formatting operation follows the ISO 8601 standard.
replace	*t*`.replace(`*hour*`=None,` *minute*`=None,`*second*`=None,` *microsecond*`=None[,` *tzinfo*`])` Returns a new `time` object, like *t* except for those attributes explicitly specified as arguments, which get replaced. For example: time(x,y,z).replace(minute=m) == time(x,m,z)
strftime	*t*`.strftime(`*fmt*`)` Returns a string representing time *t* as specified by the string *fmt*.

An instance *t* of the class `time` also supplies methods `dst`, `tzname`, and `utcoffset`, which accept no arguments and delegate to *t*`.tzinfo`, returning `None` when *t*`.tzinfo` is `None`.

The datetime Class

Instances of the `datetime` class represent an instant (a date, with a specific time of day within that date), may be naive or aware of time zones, and always ignore leap seconds. `datetime` extends `date` and adds `time`'s attributes; its instances have read-only integers `year`, `month`, `day`, `hour`, `minute`, `second`, and `microsecond`, and an optional `tzinfo` (`None` for naive instances).

Instances of `datetime` support some arithmetic: the difference between `datetime` instances (both aware, or both naive) is a `timedelta` instance, and you can add or subtract a `timedelta` instance to/from a `datetime` instance to construct another `datetime` instance. You can compare two instances of the `datetime` class (the later one is greater) as long as they're both aware or both naive.

datetime	`datetime`(*year*,*month, day,hour*=`0`,*minute*=`0`, *second*=`0`, *microsecond*=`0`,*tzinfo*=`None`) Returns a `datetime` object following similar constraints as the `date` class constructor.
The class `datetime` also supplies some class methods usable as alternative constructors.	
combine	`datetime.combine`(*date*, *time*) Returns a `datetime` object with the date attributes taken from *date* and the time attributes (including `tzinfo`) taken from *time.* `datetime.combine`(*d, t*) is like: datetime(d.year, d.month, d.day, t.hour, t.minute, t.second, t.microsecond, t.tzinfo)
fromordinal	`datetime.fromordinal`(*ordinal*) Returns a `datetime` object for the date given proleptic Gregorian ordinal *ordinal*, where a value of `1` means the first day of year `1` CE, at midnight.
fromtimestamp	`datetime.fromtimestamp`(*timestamp*, *tz*=`None`) Returns a `datetime` object corresponding to the instant *timestamp* expressed in seconds since the epoch, in local time. When *tz* is not `None`, returns an aware `datetime` object with the given `tzinfo` instance *tz*.
now	`datetime.now`(*tz*=`None`) Returns a `datetime` object for the current local date and time. When *tz* is not `None`, returns an aware `datetime` object with the given `tzinfo` instance *tz*.
strptime	`datetime.strptime`(*str*, *fmt*) Returns a `datetime` representing *str* as specified by string *fmt*. When `%z` is present in *fmt*, the resulting `datetime` object is time zone-aware.
today	`datetime.today()` Returns a naive `datetime` object representing the current local date and time, same as the `now` class method (but not accepting optional argument *tz*).
utcfromtimestamp	`datetime.utcfromtimestamp`(*timestamp*) Returns a naive `datetime` object corresponding to the instant *timestamp* expressed in seconds since the epoch, in UTC.
utcnow	`datetime.utcnow()` Returns a naive `datetime` object representing the current date and time, in UTC.

An instance *d* of `datetime` also supplies the following methods:

astimezone	*d*`.astimezone(`*tz*`)` Returns a new aware `datetime` object, like *d* (which must also be aware), except that the time zone is converted to the one in `tzinfo` object *tz*. [a]
ctime	*d*`.ctime()` Returns a string representing date and time *d* in the same 24-character format as `time.ctime`.
date	*d*`.date()` Returns a date object representing the same date as *d*.
isocalendar	*d*`.isocalendar()` Returns a tuple with three integers (ISO year, ISO week number, and ISO weekday) for *d*'s date.
isoformat	*d*`.isoformat(`*sep*`='T')` Returns a string representing *d* in the format `'YYYY-MM-DDxHH:MM:SS'`, where *x* is the value of argument *sep* (must be a string of length `1`). If *d*`.microsecond!=0`, seven characters, `'.mmmmmm'`, are added after the `'SS'` part of the string. If *t* is aware, six more characters, `'+HH:MM'`, are added at the end to represent the time zone's offset from UTC. In other words, this formatting operation follows the ISO 8601 standard. `str(`*d*`)` is the same as *d*`.isoformat(' ')`.
isoweekday	*d*`.isoweekday()` Returns the day of the week of *d*'s date as an integer; `1` for Monday through `7` for Sunday.
replace	*d*`.replace(`*year*`=None,`*month*`=None,`*day*`=None,`*hour*`=None,`*minute*`=None,`*second*`=None,` *microsecond*`=None[,`*tzinfo*`])` Returns a new `datetime` object, like *d* except for those attributes specified as arguments, which get replaced (but does no timezone conversion, see footnote 2). For example: datetime(x,y,z).replace(month=m) == datetime(x,m,z)
strftime	*d*`.strftime(`*fmt*`)` Returns a string representing *d* as specified by the format string *fmt*.
time	*d*`.time()` Returns a naive time object representing the same time of day as *d*.
	d`.timestamp()`

timestamp	Returns a float with the seconds since the epoch. Naive instances are assumed to be in the local time zone.
timetz	*d*`.timetz()` Returns a `time` object representing the same time of day as *d*, with the same `tzinfo`.
timetuple	*d*`.timetuple()` Returns a timetuple corresponding to instant *d*.
toordinal	*d*`.toordinal()` Returns the proleptic Gregorian ordinal for *d*'s date. For example: datetime(1,1,1).toordinal() == 1
utctimetuple	*d*`.utctimetuple()` Returns a timetuple corresponding to instant *d*, normalized to UTC if *d* is aware.
weekday	*d*`.weekday()` Returns the day of the week of *d*'s date as an integer; `0` for Monday through `6` for Sunday.

a Note that *d*.astimezone(*tz*) is quite different from *d*.replace(tzinfo=*tz*): the latter does no time zone conversion, but rather just copies all of *d*'s attributes except for *d*.tzinfo.

An instance *d* of the class `datetime` also supplies the methods `dst`, `tzname`, and `utcoffset`, which accept no arguments and delegate to *d*`.tzinfo`, returning `None` when *d*`.tzinfo` is `None` (i.e., when *d* is naive).

The timedelta Class

Instances of the `timedelta` class represent time intervals with three read-only integer attributes: `days`, `seconds`, and `microseconds`.

timedelta	`timedelta(`*days*`=0,` *seconds*`=0,` *microseconds*`=0,` *milliseconds*`=0,` *minutes*`=0,` *hours*`=0,` *weeks*`=0)` Converts all units with the obvious factors (a week is 7 days, an hour is 3,600 seconds, and so on) and normalizes everything to the three integer attributes, ensuring that `0<=seconds<3600*24` and

`0<=microseconds<1000000`. For example:
print(repr(timedelta(minutes=0.5))
#prints: datetime.timedelta(seconds=30)
print(repr(timedelta(minutes=-0.5))) datetime.timedelta(days=-1, seconds=86370)
Instances of `timedelta` support arithmetic: `+` and `-` between themselves and with instances of the classes `date` and `datetime`; `*` with integers; `/` with integers and `timedelta` instances (floor division, true division, `divmod`, `%`); and comparisons between themselves.

An instance *td* of `timedelta` supplies the following method:

td`.total_seconds()`
total_seconds Returns the total seconds represented by a `timedelta` instance.

The zoneinfo Module

The `zoneinfo` module ||**3.9++**||[1] is a concrete implementation of timezones for use with datetime's `tzinfo`. `zoneinfo` uses the system's timezone data by default, with `tzdata` (available on PyPI) as a fallback.[2] `zoneinfo` provides one class: `ZoneInfo`, a concrete implementation of the `datetime.tzinfo` abstract class. You can assign it to `tzinfo` or `tz` during construction of an aware datetime instance, or use it with `datetime.replace` or `datetime.astimezone` methods. You can find a list of the time zones on Wikipedia. Here is an example of construction:

```
>>> from datetime import datetime
>>> from zoneinfo import ZoneInfo
>>> d=datetime.now(tz=ZoneInfo("America/Los_Angeles"))
>>> d
datetime.datetime(2021,10,21,16,32,23,96782,tzinfo=zoneinfo.ZoneI
nfo(key='America/Los_Angeles'))
```

Update the timezone to a different one without changing other attributes:

```
>>> dny=d.replace(tzinfo=ZoneInfo("America/New_York"))
>>> dny
datetime.datetime(2021,10,21,16,32,23,96782,tzinfo=zoneinfo.ZoneI
nfo(key='America/New_York'))
```

Convert a datetime instance to UTC:

```
>>> dutc=d.astimezone(tz=ZoneInfo("UTC"))
>>> dutc
datetime.datetime(2021,10,21,23,32,23,96782,tzinfo=zoneinfo.ZoneI
nfo(key='UTC'))
```

Convert the datetime instance into a different timezone:

```
>>>
dutc.astimezone(ZoneInfo("Europe/Rome"))datetime.datetime(2021,10
,22,1,32,23,96782,tzinfo=zoneinfo.ZoneInfo(key='Europe/Rome'))
```

Always Use The Utc Time Zone Internally

The best way to program around the traps and pitfalls of time zones is to always use the UTC time zone internally, converting from other time zones on input, and use datetime.astimezone only for display purposes.

The dateutil Module

The third-party package dateutil (which you can install with **pip install python-dateutil**) offers modules to manipulate dates in many ways: time deltas, recurrence, timezones, Easter dates, and fuzzy parsing. (See the package's website for complete documentation of its rich functionality.) In addition to timezone-related operations (now best performed with `zoneinfo`), `dateutil`'s main modules are:

easter

`easter.easter`(*year*)
Returns the `datetime.date` object for Easter of the given *year*. For example:
from dateutil import easter

```
print(easter.easter(2006))
```

prints **2006-04-16**

parser	`parser.parse(`*s*`)`
	Returns the `datetime.datetime` object denoted by string *s*, with very permissive (AKA "fuzzy") parsing rules. For example:
	from dateutil import parser
	
	`print(parser.parse('''Saturday, January 28, 2006, at 11:15pm'''))`
	
	prints **2006-01-28 23:15:00**

relativedelta	`relativedelta.relativedelta(...)` `relativedelta` allows, among other things, an easy way to find "next Monday," "last year," etc. `dateutil`'s docs offer detailed explanations of the rules defining the inevitably complicated behavior of `relativedelta` instances.
rrule	`rrule.rrule(`*freq*`, ...)` Module `rrule` implements RFC2445 (also known as the `iCalendar` RFC), in all the glory of its 140+ pages. `rrule` allows you to deal with recurring events, providing such methods as `after`, `before`, `between`, and `count`. See the dateutil docs for more information .

The sched Module

The `sched` module implements an event scheduler, letting you easily deal, along a single thread of execution or in multithreaded environments, with events that may be scheduled in either a "real" or a "simulated" time scale. `sched` supplies a `scheduler` class:

scheduler	`class scheduler([`*timefunc*`],[`*delayfunc*`])` The arguments *timefunc* and *delayfunc* are optional and default to `time.monotonic` and `time.sleep`, respectively. *timefunc* must be callable without arguments to get the current time instant (in any unit of measure); for example, you can pass `time.time` or `time.monotonic`. *delayfunc* is callable with one argument (a time duration, in the same units as *timefunc*) to delay the current thread for that time; for example, you can pass `time.sleep`. `scheduler` calls *delayfunc*`(0)` after each event to give other threads a chance; this is compatible with `time.sleep`. By taking functions as arguments, `scheduler` lets you use whatever "simulated time" or "pseudotime" fits your application's needs (a great example of the dependency injection design pattern for purposes not necessarily related to testing).

If monotonic time (time cannot go backward, even if the system clock is adjusted backward between calls, e.g., due to leap seconds) is important to your application, use `time.monotonic` for your scheduler. A `scheduler` instance *s* supplies the following methods:

cancel

s`.cancel(`*event_token*`)`
Removes an event from *s*'s queue. *event_token* must be the result of a previous call to *s*`.enter` or *s*`.enterabs`, and the event must not yet have happened; otherwise, `cancel` raises `RuntimeError`.

empty

s`.empty()`
Returns `True` when *s*'s queue is currently empty; otherwise, `False`.

enterabs

s`.enterabs(`*when,priority,func,args*`=()`,*kwargs*`={})`
Schedules a future event (a callback to *func(args, kwargs))* at time *when*. *when* is in the units used by the time functions of *s*. Should several events be scheduled for the same time, *s* executes them in increasing order of *priority*. `enterabs` returns an event token *t*, which you may later pass to *s*`.cancel` to cancel this event.

enter

s`.enter(`*delay,priority,func,args*`=()`,*kwargs*`={})`
Like `enterabs`, except that *delay* is a relative time (a positive difference forward from the current instant), while `enterabs`'s argument *when* is an absolute time (a future instant).
To schedule an event for *repeated* execution, use a little wrapper function; for example:

```
def enter_repeat(s, first_delay, period, priority,
func, args):
    def repeating_wrapper():
        s.enter(period, priority,
            repeating_wrapper, ())
        func(*args)
    s.enter(first_delay, priority,
        repeating_wrapper, args)
```

run

s`.run(`*blocking*`=True)`
Runs scheduled events. If *blocking* is true, *s*`.run` loops until *s*`.empty()`, using the *delayfunc* passed on *s*'s initialization to wait for each scheduled event. If *blocking* is false, executes any soon-to-expire events, then returns the next event's deadline (if any). When a callback *func* raises an exception, *s* propagates it, but *s* keeps its own state, removing the event from the schedule. If a callback *func* runs longer than the time available before the next scheduled event, *s* falls behind but keeps executing scheduled events in order, never dropping any. Call *s*`.cancel` to drop an event explicitly if that event is no longer of interest.

The calendar Module

The `calendar` module supplies calendar-related functions, including functions to print a text calendar for a given month or year. By default, `calendar` takes Monday as the first day of the week and Sunday as the last one. To change this, call `calendar.setfirstweekday`. `calendar` handles years in module `time`'s range, typically (at least) 1970 to 2038.

The `calendar` module supplies the following functions:

calendar	`calendar(`*year,w=2, l=1,c=6*`)` Returns a multiline string with a calendar for year *year* formatted into three columns separated by *c* spaces. *w* is the width in characters of each date; each line has length `21*`*w*`+18+2*`*c*. *l* is the number of lines for each week.
firstweekday	`firstweekday()` Returns the current setting for the weekday that starts each week. By default, when `calendar` is first imported, this is `0`, meaning Monday.
isleap	`isleap(`*year*`)` Returns `True` if *year* is a leap year; otherwise, `False`.
leapdays	`leapdays(`*y1,y2*`)` Returns the total number of leap days in the years within `range(`*y1,y2*`)` (remember, this means that *y2* is excluded).
month	`month(`*year,month,w=2,l=1*`)` Returns a multiline string with a calendar for month *month* of year *year*, one line per week plus two header lines. *w* is the width in characters of each date; each line has length `7*`*w*`+6`. *l* is the number of lines for each week.
monthcalendar	`monthcalendar(`*year,month*`)` Returns a list of lists of `int`s. Each sublist denotes a week. Days outside month *month* of year *year* are set to `0`; days within the month are set to their day-of-month, `1` and up.
monthrange	`monthrange`*(year,month)* Returns two integers. The first one is the code of the weekday for the first day of the month *month* in year *year*; the second one is the number of days in the month. Weekday codes are `0` (Monday) to `6` (Sunday); month numbers are `1` to `12`.

prcal	`prcal`*(year,w=2,l=1,c=6)* Like `print(calendar.calendar(`*year*, *w,l,c*`))`.
prmonth	`prmonth`*(year,month,w=2,l=1)* Like `print(calendar.month(`*year,month,w,l*`))`.
setfirstweekday	`setfirstweekday(`*weekday*`)` Sets the first day of each week to weekday code *weekday*. Weekday codes are `0` (Monday) to `6` (Sunday). `calendar` supplies the attributes `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY`, `SATURDAY`, and `SUNDAY`, whose values are the integers `0` to `6`. Use these attributes when you mean weekdays (e.g., `calendar.FRIDAY` instead of `4`) to make your code clearer and more readable.
timegm	`timegm(`*tupletime*`)` Just like `time.mktime`: accepts a time instant in timetuple form and returns that instant as a `float` num of seconds since the epoch.
weekday	`weekday(`*year,month,day*`)` Returns the weekday code for the given date. Weekday codes are `0` (Monday) to `6` (Sunday); month numbers are `1` (Jan) to `12` (Dec).

python -m calendar offers a useful command-line interface to the module's functionality: run **python -m calendar -h** to get a brief help message.

---

1 pre-3.9, use instead third-party module pytz

2 On some platforms, you may need to *pip install tzdata*; once installed, you don't import tzdata in your program -- rather, zoneinfo uses it automatically.

Chapter 8. Numeric Processing

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 15th chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

You can perform some numeric computations with operators (covered in "Numeric Operations") and built-in functions (covered in "Built-in Functions"). Python also provides modules that support additional numeric computations, covered in this chapter: `math` and `cmath` in "The math and cmath Modules", `statistics` in "The statistics Module", `operator` in "The operator Module", `random` and `secrets` in "The random Module", `fractions` in "The fractions Module", and `decimal` in "The decimal Module". Numeric processing often requires, more specifically, the processing of *arrays* of numbers, covered in "Array Processing", focusing on the standard library module `array` and popular third-party extension NumPy. Finally, "Additional Numeric Packages" lists several additional numeric processing packages produced by the Python community.

Floating-point Values

Python represents real numeric values (that is, those that are not integers) using variables of type `float`. Unlike integers, computers can rarely represent `floats` exactly, due to their internal implementation as a fixed-size binary integer *significand* (often incorrectly called "mantissa") and a fixed-size binary integer exponent. They are limited in terms of how many

decimal places they can represent, how large an integer they can accurately store, and how large an overall number they can store.

For most everyday applications, floats are sufficient for arithmetic, but they are limited in the number of decimal places they can represent.

```
>>> f = 1.1 + 2.2 - 3.3  # f should be equal to 0
>>> f
4.440892098500626e-16
```

They are also limited in the range of integer values they can accurately store ("accurately" meaning "can distinguish from next largest or smallest integer value").

```
>>> f = 2**53
>>> f
9007199254740992
>>> f + 1
9007199254740993    # integer arithmetic is not bounded
>>> f + 1.0
9007199254740992.0  # conversion to float loses integer precision
at 2**53
```

Always keep in mind that `floats` are not entirely precise, due to their internal representation in the computer. The same consideration applies to `complex` numbers.

DON'T USE == BETWEEN FLOATING-POINT OR COMPLEX NUMBERS

Given the approximate nature of floating-point arithmetic, it rarely makes sense to check if two floats *x* and *y* are equal. Tiny variations in how each was computed can easily result in unexpected differences. Instead, use function `isclose` exported by built-in module `math`. The following code illustrates why:

```
>>> import math
>>> f = 1.1 + 2.2 - 3.3 # f is intuitively equal to 0
>>> f==0
False
>>> f
4.440892098500626e-16
>>> # default tolerance is fine for this comparison
>>> math.isclose(-1, f-1)
True
```

For some values, you may have to set the tolerance value explicitly (it is always necessary when comparing with 0):

```
>>> # near-0 comparison with default tolerances
>>> math.isclose(0,f)
False
>>> # use abs_tol for near-0 comparison
>>> math.isclose(0,f,abs_tol=1e-15)
True
```

DON'T USE A FLOAT AS A LOOP CONTROL VARIABLE

A common error is to use a floating-point value as the control variable of a loop, assuming that it will eventually equal some ending value, such as 0. However, the following loop, expected to loop 5 times and then end, will loop forever:

```
>>> f = 1
>>> while f != 0:
... f -= 0.2 # even though f started as int, it's now a float
```

This code shows why:

```
>>> 1 - 0.2 - 0.2 - 0.2 - 0.2 - 0.2 # should be 0, but...
5.551115123125783e-17
```

Even using the inequality operator > results in incorrect behavior, looping 6 times instead of 5 (since the residual float value is still greater than 0):

```
>>> f = 1
>>> count = 0
>>> while f > 0:
... count += 1
... f -= 0.2
>>> print(count)
6 # 1 time too many!
```

Using `math.isclose` for comparing *f* with 0, the for loop now repeats the correct number of times:

```
>>> f = 1
>>> count = 0
>>> while not math.is_close(0,f,abs_tol=1e-15):
... count += 1
... f -= 0.2
>>> print(count)
5 # just right this time!
```

In general, it is better practice to use an `int` for a loop's control variable, rather than a `float`.

The math and cmath Modules

The `math` module supplies mathematical functions on floating-point numbers; the `cmath` module supplies equivalent functions on complex numbers. For example, `math.sqrt(-1)` raises an exception, but `cmath.sqrt(-1)` returns `1j`.

Just like for any other module, the cleanest, most readable way to use these is to have, for example, `import math` at the top of your code, and explicitly call, say, `math.sqrt` afterward. However, if your code includes a large number of calls to the modules' well-known mathematical functions, it is allowed (though it may lose some readability) to either use `from math import *`, or use `from math import sqrt`, and afterward just call `sqrt`.

Each module exposes three `float` attributes bound to the values of fundamental mathematical constants, `e`, `pi`, and tau, and a variety of functions, including those shown in Table 15-1.

The `math` and `cmath` modules are not fully symmetric. The following table lists the methods in these modules, and, for each method, indicates whether it is in `math`, `cmath`, or both.

Table 8-1. Methods in the math and cmath modules

		math	cmath
acos, asin, atan, cos, sin, tan	`acos(`*x*`)` Returns the arccosine, arcsine, arctangent, cosine, sine, or tangent of *x*, respectively, in radians.	✓	✓
acosh, asinh, atanh, cosh, sinh, tanh	`acosh(`*x*`)` Returns the arc hyperbolic cosine, arc hyperbolic sine, arc hyperbolic tangent, hyperbolic cosine, hyperbolic sine, or hyperbolic tangent of *x*, respectively, in radians.	✓	✓

atan2	`atan2(`*y*,*x*`)` Like `atan(`*y*/*x*`)`, except that `atan2` properly takes into account the signs of both arguments. For example:
	`>>> import math`
	`>>> math.atan(-1./-1.) 0.78539816339744828`
	`>>> math.atan2(-1., -1.) -2.3561944901923448`
	When *x* equals `0`, `atan2` returns `pi/2`, while dividing by *x* would raise `ZeroDivisionError`.
		✓	

ceil	`ceil(`*x*`)` Returns `float(`*i*`)`, where *i* is the lowest integer such that *i*>=*x*.	✓	
comb	‖**3.8++**‖ `comb(n, k)` Returns the number of *combinations* of `n` items taken `k` items at a time, regardless of order. When counting the number of combinations taken from 3 items A, B, and C, 2 at a time (`comb(3, 2)`), A-B and B-A are considered the same combination. Raises `ValueError` if *k* or *n* is negative; raises `TypeError` if *k* or *n* are not `int`.	✓	
copysign	`copysign(`*x*`, `*y*`)` Returns the absolute value of *x* with the sign of *y*.	✓	

degrees		`degrees(`*x*`)` Returns the degree measure of the angle *x* given in radians.	✓	
	dist	||**3.8++**|| `dist(`*pt0*`, `*pt1*`)` Returns the Euclidean distance between two n-dimensional points, where each point is represented as a sequence of values (coordinates). Raises `ValueError` if *pt0* and *pt1* are not the same length.	✓	
	e	The mathematical constant *e* (2.718281828459045).	✓	✓
	erf	`erf(`*x*`)` Returns the error function of *x* as used in statistical calculations.	✓	
	erfc	`erfc(`*x*`)` Returns the complementary error function at *x*, defined as `1.0 - erf(`*x*`)`.	✓	
	exp	`exp(`*x*`)` Returns `e**`*x*.	✓	✓
expm1		`expm1(`*x*`)` Returns `e**`*x* *- 1*. Inverse of `log1p`.	✓	
	fabs	`fabs(`*x*`)` Returns the absolute value of *x*.	✓	
factorial		`factorial(`*x*`)` Returns the factorial of *x*. Raises `ValueError` when *x* is negative and `TypeError` when *x* is not integral.	✓	
	floor	`floor(`*x*`)` Returns `float(`*i*`)`, where *i* is the greatest integer such that *i*<=*x*.	✓	
	fmod	`fmod(`*x,y*`)` Returns the float *r*, with the same sign as *x*, such that *r*==*x*-*n***y* for some integer *n*, and `abs(`*r*`)<abs(`*y*`)`. Like *x%y*, except that, when *x* and *y* differ in sign, *x%y* has the same sign as *y*, not the same sign as *x*.	✓	
	frexp	`frexp(`*x*`)` [a]Returns a pair (*m*, *e*) where *m* is a floating-point number, and *e* is an integer such that *x*==*m**`(2**e)` and `0.5<=abs(`*m*`)<1`, except that	✓	

		`frexp(0)` returns `(0.0,0)`.		
	fsum	`fsum(`*iterable*`)` Returns the floating-point sum of the values in *iterable* to greater precision than the `sum` built-in function.	✓	
gamma		gamma(x) Returns the Gamma function evaluated at `x`.	✓	
	gcd	`gcd(`*x,y*`)` Returns the Greatest Common Divisor of *x* and *y*. When *x* and *y* are both zero, returns `0`. (‖3.9++‖ `gcd` can accept any number of values)	✓	
	hypot	`hypot(`*x,y*`)` Returns `sqrt(`*x*x+y*y*`)`. (‖3.8++‖ can accept any number of values, to compute a hypotenuse length in n-dimensions)	✓	
	inf	`inf` A floating-point positive infinity, like `float('inf')`.	✓	✓
	infj	infj A complex imaginary infinity, equal to complex(0, float('inf'))		✓
isclose		`isclose(`*x*, *y*, `rel_tol=1e-09`, `abs_tol=0.0`) Returns `True` when *x* and *y* are approximately equal, within relative tolerance `rel_tol`, with minimum absolute tolerance of `abs_tol`; otherwise, returns `False`. Default is `rel_tol` within 9 decimal digits. `rel_tol` must be greater than `0`. `abs_tol` is used for comparisons near zero: it must be at least `0.0`. `NaN` is not considered close to any value (including `NaN` itself); each of `-inf` and `inf` is only considered close to itself. Except for behavior at +/- `inf`, `isclose` is like: `abs(x-y) <= max(rel_tol*max(abs(x), abs(y)),abs_tol)`	✓	✓
isfinite		`isfinite(`*x*`)` Returns `True` when *x* (in `cmath`, both the real and imaginary part of *x*) is neither infinity nor `NaN`; otherwise, returns `False`.	✓	✓
	isinf	`isinf(`*x*`)` Returns `True` when *x* (in `cmath`, either the real or imaginary part of *x*, or both) is positive or negative infinity; otherwise, returns `False`.	✓	✓
		`isnan(`*x*`)` Returns `True` when *x* (in `cmath`, either the real or imaginary part of	✓	✓

	isnan	Returns `True` when *x* (in `cmath`, either the real or imaginary part of *x*, or both) is `NaN`; otherwise, returns `False`.	
	isqrt	||**3.8++|| `isqrt(x)` Returns `int(sqrt(`*x*`))`.	✓
	lcm	||**3.9++|| `lcm(x, ...)` Returns the Least Common Multiple of the given ints. If all values are not ints, raises `TypeError`.	
	ldexp	`ldexp(`*x,i*`)` Returns *x*`* (2**`*i*`)` (*i* must be an `int`; when *i* is a `float`, `ldexp` raises `TypeError`). Inverse of `frexp`.	✓
lgamma		lgamma(x) Returns the natural log of the absolute value of the Gamma function evaluated at `x`.	✓
	log	`log(`*x*`)` Returns the natural logarithm of *x*.	✓ ✓
	log10	`log10(`*x*`)` Returns the base-10 logarithm of *x*.	✓ ✓
	log1p	`log1p(`*x*`)` Returns the natural log of `1+`*x*. Inverse of `expm1`.	✓
	log2	log2(x) Returns the base-2 logarithm of *x*.	✓
	modf	`modf(`*x*`)` Returns a pair *(f,i)* with fractional and integer parts of *x*, meaning two `floats` with the same sign as *x* such that *i*`==int(`*i*`)` and *x*==*f*+*i*.	✓
	nan	`nan` A floating-point “Not a Number” (`NaN`) value, like `float('nan')` or `complex('nan')`.	✓ ✓
	nanj	A complex number with a 0.0 real part and floating-point “Not a Number” (`NaN`) imaginary part.	✓
nextafter		||**3.9++|| `nextafter(`*a, b*`)` Returns the next higher or lower float value from *a* in the direction of *b*.	✓
		||**3.8++|| `perm(`*n, k*`)` Returns the number of *permutations* of *n* items taken *k* items at a time	✓

perm	Returns the number of *permutations* of *n* items taken *k* items at a time, where selections of the same items but in differing order are counted separately. When counting the number of permutations of 3 items A, B, and C, taken 2 at a time (`perm(3, 2)`), A-B and B-A are considered to be different permutations. Raises `ValueError` when *k* or *n* is negative; raises `TypeError` when *k* or *n* are not `int`.		
pi	The mathematical constant *π*, 3.141592653589793.	✓	✓
phase	`phase(x)` Returns the phase of *x*, as a float in the range *(-π, π)*. Like `math.atan2(x.imag, x.real)`. See "Conversions to and from polar coordinates" in the Python online docs.		✓
polar	`polar(x)` Returns the polar coordinate representation of *x*, as a pair (`r, phi`) where `r` is the modulus of *x* and `phi` is the phase of *x*. Like (`abs(x), cmath.phase(x)`). See "Conversions to and from polar coordinates" in the Python online docs.		✓
pow	`pow(x,y)` Returns *x****y*.	✓	
prod	‖**3.8++**‖ `prod(seq, start=1)` Returns the product of all values in the sequence, beginning with the given `start` value, which defaults to 1.	✓	
radians	`radians(x)` Returns the radian measure of the angle *x* given in degrees.	✓	
rect	`rect(r, phi)` Returns the complex value representing the polar coordinates `(r, phi)` converted to rectangular coordinates as `(x + yj)`.		✓
remainder	`remainder(x, y)` Returns the remainder from dividing *x* `/` *y*.	✓	
sqrt	`sqrt(x)` Returns the square root of *x*.	✓	✓
tau	The mathematical constant τ=2*π*, 6.283185307179586.	✓	✓
trunc	`trunc(x)`	✓	

	Returns *x* truncated to an `int`.	
ulp	||**3.9++|| `ulp(x)` Returns the least significant bit of floating-point value *x*. For positive values, equals `math.nextafter(x, x+1) - x`. For negative values, equals `ulp(-x)`. If *x* is NaN or inf, returns *x*. `ulp` stands for "Unit of Least Precision."	✓

a Formally, *m* is the mantissa or, rather, *significand*, and *e* is the exponent. Used to render a cross-platform portable representation of a floating-point value.

The statistics Module

The `statistics` module supplies functions to compute common statistics, and the class `NormalDist` to perform distribution analytics.

`harmonic_mean`	`median_high`	`pvariance`
`mean`	`median_low`	`stdev`
`median`	`mode`	`variance`
`median_grouped`	`pstdev`	

||3.8++||

`fmean`	`multimode`	`NormalDist`

```
    geometric_mean          quantiles
```

||**3.10++**||

```
        correlation
                            covariance          linear_regression
```

The Python online docs contain detailed information on the signatures and use of these functions.

The operator Module

The `operator` module supplies functions that are equivalent to Python's operators. These functions are handy in cases where callables must be stored, passed as arguments, or returned as function results. The functions in `operator` have the same names as the corresponding special methods (covered in "Special Methods"). Each function is available with two names, with and without "dunder" (leading and trailing double underscores): for example, both `operator.add(`*a*,*b*`)` and `operator.__add__(`*a*,*b*`)` return *a*+*b*.

Matrix multiplication support has been added for the infix operator `@`, but you must implement it by defining your own __matmul__, __rmatmul__, and/or __imatmul__; NumPy currently supports `@` (but, as of this writing, not yet `@=`) for matrix multiplication.

Table 15-2 lists some of the functions supplied by the `operator` module.

Table 8-2. Functions supplied by the operator module

Method	Signature	Behaves like
abs	*abs(a)*	abs(*a*)
add	*add(a, b)*	*a* + *b*
and_	*and_(a, b)*	*a* & *b*
concat	*concat (a, b)*	*a* + *b*
contains	*contains(a, b)*	*b* in *a*
countOf	*countOf (a, b)*	*a* .count (*b*)
delitem	*delitem (a, b)*	del *a*[*b*]
delslice	*delslice*	del *a*[*b*:*c*]

	(a, *b,* *c)*	
`div`	*div(a, b)*	*a / b*
`eq`	*eq(* *a, b* *)*	*a == b*
`floordiv`	*floordiv* *(a, b* *)*	*a // b*
`ge`	*ge* *(a, b* *)*	*a >= b*
`getitem`	*getitem* *(a, b* *)*	*a* `[` *b* `]`
`getslice`	*getslice* *(a, b, c* *)*	*a* `[` *b* `:` *c* `]`
`gt`	*gt* *(a, b* *)*	*a > b*

indexOf	*indexOf (a, b)*	*a* `.index(` *b* `)`
invert, inv	*invert(a), inv(a)*	*~a*
is_	*is_(a, b)*	*a* `is` *b*
is_not	*is_not (a, b)*	*a* `is not` *b*
le	*le(a, b)*	*a <= b*
lshift	*lshift (a, b)*	*a << b*
lt	*lt (a, b)*	*a < b*
matmul	*matmul (m1, m2)*	*m1 @ m2*
mod	*mod(a, b*	*a % b*

	)	
mul	*mul(* *a, b* *)*	*a * b*
ne	*ne(a, b* *)*	*a != b*
neg	*neg(* *a* *)*	*- a*
not_	*not_(a* *)*	`not` *a*
or_	*or_(a, b* *)*	*a | b*
pos	*pos(* *a* *)*	*+ a*
pow	*pow(a, b)*	a ** b
repeat	*repeat(a,* *b*	*a * b*

	)	
rshift	*rshift (a, b)*	*a >> b*
setitem	*setitem (a, b, c)*	*a [b] = c*
setslice	*setslice (a, b , c , d)*	*a [b : c] = d*
sub	*sub(a, b)*	*a - b*
truediv	*truediv (a, b*	*a/b* `# "true" div -> no truncation`

	)	
truth	*truth(* *a* *)*	bool(*a*), not not a
xor	*xor* *(a,* *b* *)*	*a ^ b*

The operator module also supplies additional higher-order functions. Three of these functions, attrgetter, itemgetter, and methodcaller, return functions suitable for passing as named argument *key*= to the sort method of lists, the sorted, min, and max built-in functions, and several functions in standard library modules such as heapq and itertools.

attrgetter	attrgetter(*attr*) Returns a callable *f* such that *f(o)* is the same as getattr(*o,attr*). The *attr* string can include dots (.), in which case the callable result of attrgetter calls getattr repeatedly. For example, operator.attrgetter('a.b') is equivalent to lambda *o*: getattr(getattr(*o*, 'a'), 'b'). attrgetter(**attrs*) When you call attrgetter with multiple arguments, the resulting callable extracts each attribute thus named and returns the resulting tuple of values.
itemgetter	itemgetter(*key*) Returns a callable *f* such that *f*(*o*) is the same as getitem(*o, key*). itemgetter(**keys*) When you call itemgetter with multiple arguments, the resulting callable extracts each item thus keyed and returns the resulting tuple of values. For example, say that L is a list of lists, with each sublist at least three items long: you want to sort L, in-place, based on the third item of each sublist, with sublists having equal third items sorted by their first items. The simplest way:

	`import operator L.sort(key=operator.itemgetter(2, 0))`
length_hint	length_hint(iterable, default=0) Used to try to pre-allocate storage for items in `iterable`. Calls object `iterable`'s `__len__` method to try to get an exact length. If `__len__` is not implemented, then Python tries calling `iterable`'s `__length_hint__` method. If also not implemented, `length_hint` returns the given `default`.
methodcaller	methodcaller(methodname, args...) Returns a callable *f* such that *f*(*o*) is the same as `o.methodname(args, ...)`. The optional `args` may be given as positional or named arguments.

Random and Pseudorandom Numbers

The `random` module of the standard library generates pseudorandom numbers with various distributions. The underlying uniform pseudorandom generator uses the popular Mersenne Twister algorithm, with a period of length `2**19937-1`.

The random Module

All functions of the `random` module are methods of one hidden global instance of the class `random.Random`. You can instantiate `Random` explicitly to get multiple generators that do not share state. Explicit instantiation is advisable if you require random numbers in multiple threads (threads are covered in Chapter "Threads and Processes"). Alternatively, instantiate `SystemRandom` if you require higher-quality random numbers. (See "Physically Random and Cryptographically Strong Random Numbers".) This section documents the most frequently used functions exposed by module `random`:

choice	`choice(`*seq*`)` Returns a random item from nonempty sequence *seq*.
	`choices(`*seq*`, *, `*weights*`, `*cum_weights*`, `*k=1*`)`

choices	Returns *k* elements from nonempty sequence *seq*, with replacement. If *weights* or *cum_weights* are given (as a list of floats or ints), then their respective choices are weighted by that amount during choosing. The *cum_weights* argument accepts a list of `floats` or `ints` as would be returned by `itertools.accumulate(`*weights*`)` ; e.g., if *weights* for a *seq* containing 3 items were `[1, 2, 1]`, then the corresponding *cum_weights* would be `[1, 3, 4]`. Only one of *weights* or *cum_weights* may be specified, and the one specified must be the same length as *seq*. If neither is specified, elements are chosen with equal probability. (If used, *cum_weights* and *k* must be given as named arguments.)
getrandbits	`getrandbits(`*k*`)` Returns an `int >=0` with *k* random bits, like `randrange(2**`*k*`)` (but faster, and with no problems for large *k*).
getstate	`getstate()` Returns a hashable and pickleable object *S* representing the current state of the generator. You can later pass *S* to function `setstate` to restore the generator's state.
jumpahead	`jumpahead(`*n*`)` Advances the generator state as if *n* random numbers had been generated. This is faster than generating and ignoring *n* random numbers.
randbytes	`randbytes(`*k*`)` **||3.9++||** Generates *k* random bytes. To generate bytes for secure or cryptographic applications, use `secrets.randbits(`*k*`*8)`, then unpack the `int` it returns into *k* bytes, using `int.to_bytes(`*k*`, 'big')`.
randint	`randint(`*start*, *stop*`)` Returns a random `int` *i* from a uniform distribution such that *start*<=*i*<=*stop*. Both endpoints are included: this is quite unnatural in Python, so you would normally prefer `randrange`.
random	`random()` Returns a random `float` *r* from a uniform distribution, `0`<=*r*<`1`.
randrange	`randrange([`*start*`,]`*stop*`[,`*step*`])` Like `choice(range(`*start*,*stop*,*step*`))`, but much faster.
sample	`sample(`*seq,k*`)` Returns a new list whose *k* items are unique items randomly drawn from *seq*. The list is in random order, so that any slice of it is an equally valid random sample. *seq* may contain duplicate items. In this case, each occurrence of an item is a candidate for selection in the sample, and the sample may also contain such duplicates.
	`seed(`*x*`=None)`

seed	Initializes the generator state. *x* can be any `int, float, str, bytes,` or `bytearray`. When *x* is `None`, and when the module `random` is first loaded, `seed` uses the current system time (or some platform-specific source of randomness, if any) to get a seed. *x* is normally an `int` up to `2**256,` a `float,` or a `str, bytes,` or `bytearray` up to 32 bytes in size. Larger *x* values are accepted, but may produce the same generator state as smaller ones.
setstate	`setstate(`*S*`)` Restores the generator state. *S* must be the result of a previous call to `getstate` (such a call may have occurred in another program, or in a previous run of this program, as long as object *S* has correctly been transmitted, or saved and restored).
shuffle	`shuffle(`*alist*`)` Shuffles, in place, mutable sequence *alist*.
uniform	`uniform(`*a,b*`)` Returns a random floating-point number *r* from a uniform distribution such that *a*<=*r*<*b*.

The `random` module also supplies several other functions that generate pseudo random floating-point numbers from other probability distributions (Beta, Gamma, exponential, Gauss, Pareto, etc.) by internally calling `random.random` as their source of randomness.

Physically and Cryptographically Strong Random Numbers: the secrets module

Pseudorandom numbers provided by the `random` module, while sufficient for simulation and modeling, are not of cryptographic quality. To get random numbers and sequences for use in security and cryptography applications, use the functions defined in the `secrets` module. Those functions use the `random.SystemRandom` class, which in turn calls `os.urandom`. `os.urandom` returns random bytes, read from physical sources of random bits such as */dev/urandom* on older Linux releases, or the `getrandom()` syscall on Linux 3.17 and above. On Windows, `os.urandom` uses cryptographical-strength sources such as the

`CryptGenRandom` API. If no suitable source exists on the current system, `os.urandom` raises `NotImplementedError`. Module `secrets` exports the following functions:

choice	choice(seq) Returns a randomly selected item from nonempty sequence `seq`.
randbelow	randbelow(n) Returns a random int `x` in the range `0 <= x < n`
randbits	randbits(k) Returns an int with `k` random bits.
token_bytes	token_bytes(n) Returns a bytes object of `n` random bytes. If `n` is omitted, a default value, such as 32, is used.
token_hex	token_hex(n) Returns a string of hexadecimal characters from `n` random bytes, with two characters per byte. If `n` is omitted, a default value, such as 32, is used.
token_urlsafe	token_urlsafe(n) Returns a string of base64-encoded characters from `n` random bytes; the resulting string's length is approx 1.3 times `n`. If `n` is omitted, a default value, such as 32, is used.

Additional recipes and best cryptographic practices are listed in Python's online documentation.

An alternative source of physically random numbers is online, from Fourmilab.

The fractions Module

The `fractions` module supplies a rational number class called `Fraction` whose instances can be constructed from a pair of integers, another rational number, or a string. You can pass a pair of (optionally

signed) `int`s: the *numerator* and *denominator*. When the denominator is `0`, a `ZeroDivisionError` is raised. A string can be of the form `'3.14'`, or can include an optionally signed numerator, a slash (`/`) , and a denominator, such as `'-22/7'`. `Fraction` also supports construction from `decimal.Decimal` instances, and from `float`s (although the latter may not provide the result you'd expect, given `float`s' bounded precision). `Fraction` class instances have the properties `numerator` and `denominator`.

Reduced to Lowest Terms

`Fraction` reduces the fraction to the lowest terms—for example, `f = Fraction(226, 452)` builds an instance `f` equal to one built by `Fraction(1, 2)`. The specific numerator and denominator originally passed to `Fraction` are not recoverable from the instance thus built.

```
>>> from fractions import Fraction
>>> Fraction(1,10)
Fraction(1, 10)
>>> Fraction(Decimal('0.1'))
Fraction(1, 10)
>>> Fraction('0.1')
Fraction(1, 10)
>>> Fraction('1/10')
Fraction(1, 10)
>>> Fraction(0.1)
Fraction(3602879701896397, 36028797018963968)
>>> Fraction(-1, 10)
Fraction(-1, 10)
>>> Fraction(-1,-10)
Fraction(1, 10)
```

`Fraction` supplies methods, including `limit_denominator`, which allows you to create a rational approximation of a `float`—for example, `Fraction(0.0999).limit_denominator(10)` returns `Fraction(1, 10)`. `Fraction` instances are immutable and can be keys in dictionaries and members of sets, as well as being used in

arithmetic operations with other numbers. See the `fractions` online docs for more complete coverage.

The `fractions` module also supplies a function called `gcd` that works just like `math.gcd`, covered in Table 15-1.

The decimal Module

A Python `float` is a binary floating-point number, normally according to the standard known as IEEE 754, implemented in hardware in modern computers. An excellent, concise, practical introduction to floating-point arithmetic and its issues can be found in David Goldberg's essay "What Every Computer Scientist Should Know about Floating-Point Arithmetic". A Python-focused essay on the same issues is part of the online tutorial; another excellent summary, not focused on Python, is also available online.

Often, particularly for money-related computations, you may prefer to use *decimal* floating-point numbers; Python supplies an implementation of the standard known as IEEE 854, for base 10, in the standard library module `decimal`. The module has excellent documentation: there, you can find complete reference documentation, pointers to the applicable standards, a tutorial, and advocacy for `decimal`. Here, we cover only a small subset of `decimal`'s functionality, the most frequently used parts of the module.

The `decimal` module supplies a `Decimal` class (whose immutable instances are decimal numbers), exception classes, and classes and functions to deal with the *arithmetic context*, which specifies such things as precision, rounding, and which computational anomalies (such as division by zero, overflow, underflow, and so on) raise exceptions when they occur. In the default context, precision is 28 decimal digits, rounding is "half-even" (round results to the closest representable decimal number; when a result is exactly halfway between two such numbers, round to the one whose last digit is even), and the anomalies that raise exceptions are: invalid operation, division by zero, and overflow.

To build a decimal number, call `Decimal` with one argument: an integer, float, string, or tuple. If you start with a `float`, it is converted losslessly to the exact decimal equivalent (which may require 53 digits or more of precision):

```
>>> from decimal import Decimal
>>> df = Decimal(0.1)
>>> df
Decimal('0.1000000000000000055511151231257827021181583404541015625')
```

If this is not the behavior you want, you can pass the float as a string; for example:

```
>>> ds = Decimal(str(0.1))  # or, directly, Decimal('0.1')
>>> ds
Decimal('0.1')
```

You can easily write a factory function for ease of interactive experimentation with `decimal`:

```
def dfs(x):
    return Decimal(str(x))
```

Now `dfs(0.1)` is just the same thing as `Decimal(str(0.1))`, or `Decimal('0.1')`, but more concise and handier to write.

Alternatively, you may use the `quantize` method of `Decimal` to construct a new decimal by rounding a float to the number of significant digits you specify:

```
>>> dq = Decimal(0.1).quantize(Decimal('.00'))
>>> dq
Decimal('0.10')
```

If you start with a tuple, you need to provide three arguments: the sign (0 for positive, 1 for negative), a tuple of digits, and the integer exponent:

```
>>> pidigits = (3, 1, 4, 1, 5)
>>> Decimal((1, pidigits, -4))
Decimal('-3.1415')
```

Once you have instances of `Decimal`, you can compare them, including comparison with `floats` (use `math.isclose` for this); `pickle` and `unpickle` them; and use them as keys in dictionaries and as members of sets. You may also perform arithmetic among them, and with integers, but not with `floats` (to avoid unexpected loss of precision in the results), as demonstrated here:

```
>>> a = 1.1
>>> d = Decimal('1.1')
>>> a == d
False
>>> math.isclose(a, d)
True
>>> a + d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
  'decimal.Decimal' and 'float'
>>> d + Decimal(a) # new decimal constructed from a
Decimal('2.200000000000000088817841970') # whoops
>>> d + Decimal(str(a)) # convert a to decimal with str(a)
Decimal('2.20')
```

The online docs include useful recipes for monetary formatting, some trigonometric functions, and a list of Frequently Asked Questions (FAQ).

Array Processing

You can represent arrays with lists (covered in “Lists”), as well as with the `array` standard library module (covered in “The array Module”). You can manipulate arrays with loops; indexing and slicing; list comprehensions; iterators; generators; genexps (all covered in Chapter “The Python Language”); built-ins such as `map`, `reduce`, and `filter` (all covered in “Built-in Functions”); and standard library modules such as `itertools` (covered in “The itertools Module”). If you only need a lightweight, one-

dimensional array, stick with `array`. However, to process large arrays of numbers, such functions may be slower and less convenient than third-party extensions such as NumPy and SciPy (covered in "Extensions for Numeric Array Computation"). When you're doing data analysis and modeling, pandas, which is built on top of NumPy, might be most suitable.

The array Module

The `array` module supplies a type, also called `array`, whose instances are mutable sequences, like lists. An `array` *a* is a one-dimensional sequence whose items can be only characters, or only numbers of one specific numeric type, fixed when you create *a*.

`array.array`'s advantage is that, compared to a list, it can save memory to hold objects all of the same (numeric or character) type. An `array` object *a* has a one-character, read-only attribute *a*`.typecode`, set on creation: the type code of *a*'s items. Table 15-3 shows the possible type codes for `array`.

Table 8-3. Type codes for the array module

typecode	C type	Python type	Minimum size
`'b'`	`char`	`int`	1 byte
`'B'`	`unsigned char`	`int`	1 byte
`'u'`	`unicode char`	`str (length 1)`	see note
`'h'`	`short`	`int`	2 bytes
			2 bytes

`'H'`	unsigned short	int	2 bytes
`'i'`	int	int	2 bytes
`'I'`	unsigned int	int	4 bytes
`'l'`	long	int	4 bytes
`'L'`	unsigned long	int	8 bytes
`'q'`	long long	int	8 bytes
`'Q'`	unsigned long long	int	4 bytes
`'f'`	float	float	8 bytes
`'d'`	double	float	

NOTE

Note: `'u'` has an item size of `2` on a few platforms (mostly, Windows) and `4` on just about every other platform. You can check the build type of a Python interpreter by using `array.array('u').itemsize`.

The size in bytes of each item may be larger than the minimum, depending on the machine's architecture, and is available as the read-only attribute *a*`.itemsize`.

The module `array` supplies just the type object called `array`:

array	`array(`*typecode,init*`='')` Creates and returns an `array` object *a* with the given *typecode. init* can be a string (a bytestring, except for *typecode* `'u'`) whose length is a multiple of `itemsize`: the string's bytes, interpreted as machine values, directly initialize *a*'s items. Alternatively, *init* can be an iterable (of chars when *typecode* is `'u'`, otherwise of numbers): each item of the iterable initializes one item of *a*. Array objects expose all methods and operations of mutable sequences (as covered in "Sequence Operations"), except `sort`. Concatenation with + or +=, and slice assignment, require both operands to be arrays with the same typecode; in contrast, the argument to *a*`.extend` can be any iterable with items acceptable to *a*.

In addition to the methods of mutable sequences (`append, extend, insert, pop`, etc.), an array object *a* exposes the following methods and properties.

buffer_info	*a*`.buffer_info()` Returns a 2-item tuple `(address, array_length)`, where `array_length` is the number of elements that can be stored in *a*. The size of *a* in bytes is *a*`.buffer_info()[1] * `*a*`.itemsize`.
byteswap	*a*`.byteswap()` Swaps the byte order of each item of *a*.
frombytes	*a*`.frombytes(`*s*`)` `frombytes` appends to *a* the bytes, interpreted as machine values, of bytes *s*. `len(`*s*`)` must be an exact multiple of *a*`.itemsize`.
fromfile	*a*`.fromfile(`*f, n*`)` Reads *n* items, taken as machine values, from file object *f* and appends the items to *a*. Note that *f* should be open for reading in binary mode—for example, with mode `'rb'`. When fewer than *n* items are available in *f*, `fromfile` raises `EOFError` after appending the items that are available.
fromlist	*a*`.fromlist(`*L*`)` Appends to *a* all items of list *L*.
fromunicode	*a*`.fromunicode(`*s*`)` Appends to *a* all characters from string *s*. *a* must have typecode `'u'`; otherwise, Python raises `ValueError`.
	a`.itemsize` Property that returns the size in bytes of an element in *a*.

itemsize	
tobytes	*a*`.tobytes()` `tobytes` returns the bytes representation of the items in a. For any *a*, `len(`*a*`.tobytes())== len(`*a*`)*`*a*`.itemsize`.*f*`.write(`*a*`.tobytes())` is the same as *a*`.tofile(`*f*`)`.
tofile	*a*`.tofile(`*f*`)` Writes all items of *a*, taken as machine values, to file object *f*. Note that *f* should be open for writing in binary mode—for example, with mode `'wb'`.
tolist	*a*`.tolist()` Creates and returns a list object with the same items as *a*, like `list(`*a*`)`.
tounicode	*a*`.tounicode()` Creates and returns a string with the same items as *a*, like `''.join(`*a*`)`. *a* must have typecode `'u'`; otherwise, Python raises `ValueError`.
typecode	*a*`.typecode` Property that returns the type code character used to create `a`.

Extensions for Numeric Array Computation

As you've seen, Python has great support for numeric processing. However, third-party library SciPy and packages such as NumPy, Matplotlib, Sympy, numba, pandas, and TensorFlow provide even more tools. We introduce NumPy here, then provide a brief description of SciPy and other packages, with pointers to their documentation.

NumPy

If you need a lightweight one-dimensional array of numbers, the standard library's `array` module may suffice. If you are handling scientific computing, image processing, multidimensional arrays, linear algebra, or other applications involving large amounts of data, the popular third-party NumPy package meets your needs. Extensive documentation is available online; a free PDF of Travis Oliphant's Guide to NumPy book is also available.

NumPy or numpy?

The docs variously refer to the package as NumPy or Numpy; however, in coding, the package is called `numpy`, and you usually import it with `import numpy as np`. In this section, we use all of these monikers.

`NumPy` provides class `ndarray`, which you can subclass to add functionality for your particular needs. An `ndarray` object has *n* dimensions of homogenous items (items can include containers of heterogenous types). An `ndarray` object *a* has a number of dimensions (AKA *axes*) known as its *rank*. A *scalar* (i.e., a single number) has rank `0`, a *vector* has rank `1`, a *matrix* has rank `2`, and so forth. An `ndarray` object also has a *shape*, which can be accessed as property `shape`. For example, for a matrix *m* with 2 columns and 3 rows, *m*`.shape` is `(3,2)`.

`NumPy` supports a wider range of numeric types (instances of `dtype`) than Python; the default numerical types are: `bool_`, one byte; `int_`, either *int64* or *int32* (depending on your platform); `float_`, short for *float64*; and `complex_`, short for *complex128*.

Creating a NumPy Array

There are several ways to create an array in `NumPy`; among the most common are:

- with the factory function `np.array`, from a sequence (often a nested one), with *type inference* or by explicitly specifying *dtype*
- with factory functions `zeros`, `ones`, `empty`, which default to *dtype* `float64`, and `indices`, which defaults to `int64`
- with factory function `arange` (with the usual *start, stop, stride*), or with factory function `linspace` (*start, stop, quantity*) for better floating-point behavior

- reading data from files with other `np` functions (e.g., CSV with `genfromtxt`)

Here are examples of creating an array, as just listed:

```
import numpy as np
np.array([1, 2, 3, 4])  # from a Python list
array([1, 2, 3, 4])

np.array(5, 6, 7)  # a common error: passing items separately
(they
                   # must be passed as a sequence, e.g. a list)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: only 2 non-keyword arguments accepted

s = 'alph', 'abet'  # a tuple of two strings
np.array(s)
array(['alph', 'abet'], dtype='<U4')

t = [(1,2), (3,4), (0,1)]  # a list of tuples
np.array(t, dtype='float64')  # explicit type designation
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 0.,  1.]]

x = np.array(1.2, dtype=np.float16)  # a scalar
x.shape
()
x.max()
1.2

np.zeros(3)  # shape defaults to a vector
array([ 0.,  0.,  0.])

np.ones((2,2))  # with shape specified
array([[ 1.,  1.],
       [ 1.,  1.]])

np.empty(9)  # arbitrary float64s
array([  4.94065646e-324,   9.88131292e-324,   1.48219694e-323,
         1.97626258e-323,   2.47032823e-323,   2.96439388e-323,
         3.45845952e-323,   3.95252517e-323,   4.44659081e-323])

np.indices((3,3))
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],
```

```
       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])

np.arange(0, 10, 2)  # upper bound excluded
array([0, 2, 4, 6, 8])

np.linspace(0, 1, 5)  # default: endpoint included
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  ])

np.linspace(0, 1, 5, endpoint=False)  # endpoint not included
array([ 0. ,  0.2,  0.4,  0.6,  0.8])

import io
np.genfromtxt(io.BytesIO(b'1 2 3\n4 5 6'))  # using a pseudo-file
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

with io.open('x.csv', 'wb') as f:
    f.write(b'2,4,6\n1,3,5')
np.genfromtxt('x.csv', delimiter=',')  # using an actual CSV file
array([[ 2.,  4.,  6.],
       [ 1.,  3.,  5.]])
```

Shape, Indexing, and Slicing

Each `ndarray` object *a* has an attribute *a*`.shape`, which is a tuple of `ints`. `len(`*a*`.shape)` is *a*'s rank; for example, a one-dimensional array of numbers (also known as a *vector*) has rank `1`, and *a*`.shape` has just one item. More generally, each item of *a*`.shape` is the length of the corresponding dimension of *a*. *a*'s number of elements, known as its *size*, is the product of all items of *a*`.shape` (also available as property *a*`.size`). Each dimension of *a* is also known as an *axis*. Axis indices are from `0` and up, as usual in Python. Negative axis indices are allowed and count from the right, so `-1` is the last (rightmost) axis.

Each array *a* (except a scalar, meaning an array of rank-`0`) is a Python sequence. Each item *a*`[`*i*`]` of *a* is a subarray of *a*, meaning it is an array with a rank one less than *a*'s: *a*`[`*i*`].shape==`*a*`.shape[1:]`. For example, if *a* is a two-dimensional matrix (*a* is of rank `2`), *a*`[`*i*`]`, for any valid index *i*, is a one-dimensional subarray of *a* that corresponds to a row

of the matrix. When *a*'s rank is `1` or `0`, *a*'s items are *a*'s elements (just one element, for rank-`0` arrays). Since *a* is a sequence, you can index *a* with normal indexing syntax to access or change *a*'s items. Note that *a*'s items are *a*'s subarrays; only for an array of rank `1` or `0` are the array's *items* the same thing as the array's *elements*.

As for any other sequence, you can also *slice a*: after *b*=*a*`[`*i*`:` *j*`]`, *b* has the same rank as *a*, and *b*`.shape` equals *a*`.shape` except that *b*`.shape[0]` is the length of the slice *i*`:` *j* (*j*- *i* when *a*`.shape[0]`>*j*>=*i*>=`0`, and so on).

Once you have an array *a*, you can call *a*`.reshape` (or, equivalently, `np.reshape` with *a* as the first argument). The resulting shape must match *a*`.size`: when *a*`.size` is `12`, you can call *a*`.reshape(3,4)` or *a*`.reshape(2,6)`, but *a*`.reshape(2,5)` raises `ValueError`. Note that `reshape` does not work in place: you must explicitly bind or rebind the array—that is, *a* = *a*`.reshape(`*i*, *j*`)` or *b* = *a*`.reshape(` *i*, *j*`)`.

You can also loop on (nonscalar) *a* in a `for`, just as you can with any other sequence. For example:

```
for x in a:
    process(x)
```

means the same thing as:

```
for _ in range(len(a)):
    x = a[_]
    process(x)
```

In these examples, each item *x* of *a* in the `for` loop is a subarray of *a*. For example, if *a* is a two-dimensional matrix, each *x* in either of these loops is a one-dimensional subarray of *a* that corresponds to a row of the matrix.

You can also index or slice *a* by a tuple. For example, when *a*'s rank is >=`2`, you can write *a*`[`*i*`][`*j*`]` as *a*`[`*i*,*j*`]`, for any valid *i* and *j*, for rebinding as well as for access; tuple indexing is faster and more convenient. Do not put parentheses inside the brackets to indicate that you are indexing *a* by a

tuple: just write the indices one after the other, separated by commas. *a*`[`*i,j*`]` means the same thing as *a*`[(`*i,j*`)]`, but the form without parentheses is more readable.

An indexing is a slicing when one or more of the tuple's items are slices, or (at most once per slicing) the special form `...` (the Python built-in `Ellipsis`). `...` expands into as many all-axis slices (`:`) as needed to "fill" the rank of the array you're slicing. For example, *a*`[1,...,2]` is like *a*`[1,:,:,2]` when *a*'s rank is 4, but like *a*`[1,:,:,:,:,2]` when *a*'s rank is 6.

The following snippets show looping, indexing, and slicing:

```
a = np.arange(8)
a
array([0, 1, 2, 3, 4, 5, 6, 7])
a = a.reshape(2,4)
a
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
print(a[1,2])
6
a[:,:2]
array([[0, 1],
       [4, 5]])
for row in a:
    print(row)
[0 1 2 3]
[4 5 6 7]
for row in a:
    for col in row[:2]:  # first two items in each row
        print(col)
0
1
4
5
```

Matrix Operations in NumPy

As mentioned in "The operator Module", `NumPy` implements the operator `@` for matrix multiplication of arrays. `a1 @ a2` is like `np.matmul(a1,a2)`. When both matrices are two-dimensional, they're treated as conventional matrices. When one argument is a vector, you

conceptually promote it to a two-dimensional array, as if by temporarily appending or prepending a 1, as needed, to its shape. Do not use `@` with a scalar; use `*` instead (see the following example). Matrices also allow addition (using +) with a scalar (see example), as well as with vectors and other matrices of compatible shapes. Dot product is also available for matrices, using `np.dot(`*a1, a2*`)`. A few simple examples of these operators follow:

```
a = np.arange(6).reshape(2,3)  # a 2-d matrix
b = np.arange(3)               # a vector
a
array([[0, 1, 2],
       [3, 4, 5]])
a + 1     # adding a scalar
array([[1, 2, 3],
       [4, 5, 6]])
a + b    # adding a vector
array([[0, 2, 4],
       [3, 5, 7]])
a * 2    # multiplying by a scalar
array([[ 0,  2,  4],
       [ 6,  8, 10]])
a * b    # multiplying by a vector
array([[ 0,  1,  4],
       [ 0,  4, 10]])
a @ b    # matrix-multiplying by vector
array([ 5, 14])
c = (a*2).reshape(3,2)   # using scalar multiplication to create
c                        # another matrix
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
a @ c    # matrix multiplying two 2-d matrices
array([[20, 26],
       [56, 80]])
```

NumPy is rich enough to warrant books of its own; we have only touched on a few details. See the NumPy documentation for extensive coverage of its many features.

SciPy

NumPy contains classes and methods for handling arrays; the SciPy library supports more advanced numeric computation. For example, while NumPy provides a few linear algebra methods, SciPy provides many more functions, including advanced decomposition methods, and also more advanced functions, such as allowing a second matrix argument for solving generalized eigenvalue problems. In general, when you are doing advanced numerical computation, it's a good idea to install both SciPy and NumPy.

SciPy.org also hosts docs for a number of other packages, which are integrated with SciPy and NumPy: Matplotlib, which provides 2D plotting support; Sympy, which supports symbolic mathematics; Jupyter/Notebook, a powerful interactive console shell and web-application kernel; and pandas, which supports data analysis and modeling.

Additional Numeric Packages

The Python community has produced many more packages in the field of numeric processing.

Anaconda - Anaconda is a consolidated environment that simplifies the installation of pandas, numpy, and many related numerical processing, analytical, and visualization packages, and provides package management via its own `conda` package installer.

gmpy2 - the gmpy2 module supports the GMP/MPIR, MPFR, and MPC libraries, to extend and accelerate Python's abilities for multiple-precision arithmetic.

numba - numba is a just-in-time compiler to convert numba-decorated Python functions and Numpy code to LLVM. Numba-compiled numerical algorithms in Python can approach the speeds of C or FORTRAN. ()

TensorFlow - TensorFlow is a comprehensive machine learning platform that operates at large scale and in mixed environments. It uses dataflow graphs to represent computation, shared state, and state manipulation operations. TensorFlow supports processing across multiple machines in a

cluster, and within-machine across multicore CPUs, GPUs, and custom-designed ASICs. TensorFlow's main and most popular API uses Python.

Chapter 9. Structured Text: HTML

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 22nd chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

Most documents on the web use HTML, the HyperText Markup Language. *Markup* is the insertion of special tokens, known as *tags*, in a text document, to structure the text. HTML is, in theory, an application of the large, general standard known as SGML, the Standard General Markup Language. In practice, many documents on the web use HTML in sloppy or incorrect ways.

HTML[1] is not suitable for much more than presenting documents on a browser. Complete, precise extraction of the information in the document, working backward from what most often amounts to the document's presentation, often turns out to be unfeasible. To tighten things up, HTML tried evolving into a more rigorous standard called XHTML. XHTML is similar to traditional HTML, but it is defined in terms of XML, and more precisely than HTML. You can handle well-formed XHTML with the tools covered in Chapter "Structured text: XML". However, as of this writing, XHTML does not appear to have enjoyed overwhelming success, getting scooped instead by the (non-XML) newest version, HTML5.

Despite the difficulties, it's often possible to extract at least some useful information from HTML documents (a task known as *screen-scraping*, or

just *scraping*). Python's standard library tries to help, supplying the `html` package for the task of parsing HTML documents, whether this parsing is for the purpose of presenting the documents, or, more typically, as part of an attempt to extract ("scrape") information. However, when you're dealing with somewhat-broken web pages (which is almost always!), the third-party module BeautifulSoup usually offers your last, best hope. In this book, we mostly cover `BeautifulSoup`, ignoring the standard library modules competing with it.

Generating HTML, and embedding Python in HTML, are also reasonably frequent tasks. The standard Python library doesn't support HTML generation or embedding, but you can use Python string formatting, and third-party modules can also help. `BeautifulSoup` lets you alter an HTML tree (so, in particular, you can build one up programmatically, even "from scratch"); an (often preferable) alternative approach is *templating*, supported, for example, by the third-party module jinja2, whose bare essentials we cover in "The `jinja2` Package."

The html.entities Module

The `html.entities` module in Python's standard library supplies a few attributes, all of them being mappings. They come in handy whatever general approach you're using to parse, edit, or generate HTML, including the BeautifulSoup package covered in "The BeautifulSoup Third-Party Package."

`codepoint2name`

> A mapping from Unicode codepoints to HTML entity names. For example, `entities.codepoint2name[228]` is `'auml'`, since Unicode character 228, `ä`, "lowercase a with diaeresis," is encoded in HTML as `'&auml;'`.

`entitydefs`

A mapping from HTML entity names to Unicode equivalent single-character strings. For example, `entities.entitydefs['auml']` is `'ä'`, and `entities.entitydefs['sigma']` is `'σ'`.

`html5`

`html5` is a mapping from HTML5 named character references to equivalent single-character strings. For example, `entities.html5['gt;']` is `'>'`. The trailing semicolon in the key *does* matter—a few, but far from all, HTML5 named character references can optionally be spelled without a trailing semicolon, and, in those cases, both keys (with and without the trailing semicolon) are present in `entities.html5`.

`name2codepoint`

A mapping from HTML entity names to Unicode codepoints. For example, `entities.name2codepoint['auml']` is `228`.

The BeautifulSoup Third-Party Package

BeautifulSoup lets you parse HTML even if it's rather badly formed—`BeautifulSoup` uses simple heuristics to compensate for typical HTML brokenness, and succeeds at this hard task with surprisingly good frequency. The current major version of BeautifulSoup is version 4, also known as `bs4`; in this book, we specifically cover version 4.10, the latest stable one as of this writing, of `bs4`.

Installing Versus Importing Beautifulsoup

You install the module, for example, by running, at a shell command prompt, pip install beautifulsoup4; but when you import it, in your Python code, use `import bs4`.

The BeautifulSoup Class

The `bs4` module supplies the `BeautifulSoup` class, which you instantiate by calling it with one or two arguments: first, *`htmltext`*—either a file-like object (which is read to get the HTML text to parse) or a string (which is the text to parse)—and next, an optional *`parser`* argument.

Which parser BeautifulSoup uses

If you don't pass a `parser` argument, `BeautifulSoup` "sniffs around" to pick the best parser (you may get a warning in this case). If you haven't installed any other parser, `BeautifulSoup` defaults to `html.parser` from the Python standard library (to specify that parser explicitly, use the string `'html.parser'`). To get more control – to avoid the differences between parsers mentioned in the `BeautifulSoup` documentation, pass the name of the parser library to use as the second argument as you instantiate `BeautifulSoup`. Unless specified otherwise, the following examples use the default Python `html.parser`.

For example, if you have installed the third-party package `html5lib` (to parse HTML in the same way as all major browsers do, albeit more slowly), you may call:

```
soup = bs4.BeautifulSoup(thedoc, 'html5lib')
```

When you pass `'xml'` as the second argument, you must have installed[2] the third-party package `lxml`, mentioned in "ElementTree," and `BeautifulSoup` parses the document as XML, rather than as HTML. In this case, the attribute `is_xml` of *`soup`* is `True`; otherwise, *`soup`*`.is_xml` is `False`. (If you have installed `lxml`, you can also use it to parse HTML, by passing as the second argument `'lxml'`).

```
>>> import bs4
>>> s = bs4.BeautifulSoup('<p>hello', 'html.parser')
>>> sx = bs4.BeautifulSoup('<p>hello', 'xml')
>>> sl = bs4.BeautifulSoup('<p>hello', 'lxml')
>>> s5 = bs4.BeautifulSoup('<p>hello', 'html5lib')
```

```
>>> print(s, s.is_xml)
<p>hello</p> False
>>> print(sx, sx.is_xml)
<?xml version="1.0" encoding="utf-8"?><p>hello</p> True
>>> print(sl, sl.is_xml)
<html><body><p>hello</p></body></html> False
>>> print(s5, s5.is_xml)<html><head></head><body><p>hello</p>
</body></html> False
```

Differences Between Parsers in Fixing Invalid HTML Input

In the example, `'html.parser'` just inserts end-tag `</p>`, missing from the input. As also shown, other parsers go further in repairing invalid HTML input, adding required tags such as `<body>` and `<html>`, to different extents depending on the parser.

BeautifulSoup, Unicode, and encoding

`BeautifulSoup` uses Unicode, deducing or guessing the encoding[3] when the input is a bytestring or binary file. For output, the `prettify` method returns an `str` (thus, Unicode) representation of the tree, including tags, with attributes, plus extra white-space and newlines to indent elements, to show the nesting structure; to have it instead return a `bytes` object (a byte string) in a given encoding, pass it the encoding name as an argument. If you don't want the result to be "prettified," use the `encode` method to get a bytestring, and the `decode` method to get a Unicode string. For example:

```
>>> s = bs4.BeautifulSoup('<p>hello', 'html.parser')
>>> print(s.prettify())
<p>
 hello
</p>
>>> print(s.decode())
<p>hello</p>
>>> print(s.encode())
b'<p>hello</p>'
```

The Navigable Classes of bs4

An instance *b* of class `BeautifulSoup` supplies attributes and methods to "navigate" the parsed HTML tree, returning instances of classes `Tag` and `NavigableString` (and subclasses of `NavigableString`: `CData, Comment, Declaration, Doctype`, and `ProcessingInstruction`—differing only in how they are emitted when you output them).

Navigable Classes Terminology

When we say "instances of `NavigableString`," we include instances of any of its subclasses; when we say "instances of `Tag`," we include instances of `BeautifulSoup`, since the latter is a subclass of `Tag`. Instances of navigable classes are also known as the *elements* or *nodes* of the tree.

Each instance of a "navigable class" lets you keep navigating, or dig for more information, with pretty much the same set of navigational attributes and search methods as *b* itself. There are differences: instances of `Tag` can have HTML attributes and children nodes in the HTML tree, while instances of `NavigableString` cannot (instances of `NavigableString` always have one text string, a parent `Tag`, and zero or more siblings, i.e., other children of the same parent tag).

All instances of navigable classes have attribute `name`: it's the tag string for `Tag` instances, `'[document]'` for `BeautifulSoup` instances, and `None` for instances of `NavigableString`.

Instances of `Tag` let you access their HTML attributes by indexing; or, you can get them all as a `dict` via the `.attrs` Python attribute of the instance.

Indexing instances of Tag

When *t* is an instance of `Tag`, a construct like *t*`['foo']` looks for an HTML attribute named `foo` within *t*'s HTML attributes, and returns the string for the `foo` HTML attribute. When *t* has no HTML attribute named `foo`, *t*`['foo']` raises a `KeyError` exception; just like on a `dict`, call *t*`.get( 'foo',` *default*`=None)` to get the value of the default argument, instead of an exception, when *t* has no HTML attribute named `foo`.

A few attributes, such as `class`, are defined in the HTML standard as being able to have multiple values (e.g., `<body class="foo bar">...</body>`); in these cases, the indexing returns a `list` of values—for example, `soup.body['class']` would be `['foo', 'bar']` (again, you get a `KeyError` exception when the attribute isn't present at all; use the `get` method, instead of indexing, to get a default value instead).

To get a `dict` that maps attribute names to values (or, in a few cases defined in the HTML standard, `lists` of values), use the attribute *t*`.attrs`:

```
>>> s = bs4.BeautifulSoup('<p foo="bar" class="ic">baz')
>>> s.get('foo')
>>> s.p.get('foo')
'bar'
>>> s.p.attrs
{'foo': 'bar', 'class': ['ic']}
```

How To Check if a Tag Instance Has a Certain Attribute

To check if a `Tag` instance *t*'s HTML attributes include one named `'foo'`, *don't* use `if 'foo' in` *t*:—the `in` operator on `Tag` instances looks among the `Tag`'s children, *not* its attributes. Rather, use `if 'foo' in` *t*`.attrs:` or `if` *t*`.has_attr('foo'):`.

When you have an instance of `NavigableString`, you often want to access the actual text string it contains; when you have an instance of `Tag`,

you may want to access the unique string it contains, or, should it contain more than one, all of them—perhaps with their text stripped of any whitespace surrounding it. Here's how you can best accomplish these tasks.

Getting an actual string

When you have a `NavigableString` instance *`s`* and you need to stash or process its text somewhere, without further navigation on it, call `str(`*`s`*`)`. Or, use *`s`*`.encode(`*`codec`*`='utf8')` to get a bytestring, and *`s`*`.decode()` to get a string (Unicode). These give you the actual string, without references to the `BeautifulSoup` tree impeding garbage collection (*`s`* supports all methods of Unicode strings, so call those directly if they do all you need).

Given an instance *`t`* of `Tag`, you can get its single contained `NavigableString` instance with *`t`*`.string` (so *`t`*`.string.decode()` could be the actual text you're looking for). *`t`*`.string` only works when *`t`* has a single child that's a `NavigableString`, or a single child that's a `Tag` whose only child is a `NavigableString`; otherwise, *`t`*`.string` is `None`.

As an iterator on *all* contained (navigable) strings, use *`t`*`.strings` (`''.join(`*`t`*`.strings)` could be the string you want). To ignore whitespace around each contained string, use the iterator *`t`*`.stripped_strings` (it also skips strings that are all-whitespace).

Alternatively, call *`t`*`.get_text()`—it returns a single (Unicode) string with all the text in *`t`*'s descendants, in tree order (equivalently, access the attribute *`t`*`.text`). You can optionally pass, as the only positional argument, a string to use as a separator (default is the empty string `''`); pass the named parameter `strip=True` to have each string stripped of whitespace around it, and all-whitespace strings skipped:

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> print(soup.p.string)
None
>>> print(soup.p.b.string)
bold
```

```
>>> print(soup.get_text())
Plain bold
>>> print(soup.text)
Plain bold
>>> print(soup.get_text(strip=True))
Plainbold
```

The simplest, most elegant way to navigate down an HTML tree or subtree in `bs4` is to use Python's attribute reference syntax (as long as each tag you name is unique, or you care only about the first tag so named at each level of descent).

Attribute references on instances of BeautifulSoup and Tag

Given any instance *`t`* of a `Tag`, a construct like *`t`*`.foo.bar` looks for the first tag `foo` within *`t`*'s descendants, gets a `Tag` instance *`ti`* for it, looks for the first tag `bar` within *`ti`*'s descendants, and returns a `Tag` instance for the `bar` tag.

It's a concise, elegant way to navigate down the tree, when you know there's a single occurrence of a certain tag within a navigable instance's descendants, or when the first occurrence of several is all you care about, but beware: if any level of look-up doesn't find the tag it's looking for, the attribute reference's value is `None`, and then any further attribute reference raises `AttributeError`.

Beware typos in attribute references on Tag instances

Due to this BeautifulSoup behavior, any typo you may make in an attribute reference on a `Tag` instance gives a value of `None`, not an `AttributeError` exception—so, be especially careful!

`bs4` also offers more general ways to navigate down, up, and sideways along the tree. In particular, each navigable class instance has attributes that identify a single "relative" or, in plural form, an iterator over all relatives of that ilk.

contents, children, descendants

Given an instance *t* of `Tag`, you can get a list of all of its children as *t*`.contents`, or an iterator on all children as *t*`.children`. For an iterator on all *descendants* (children, children of children, and so on), use *t*`.descendants`.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> list(t.name for t in soup.p.children)
[None, 'b']
>>> list(t.name for t in soup.p.descendants)
[None, 'b', None]
```

The `names` that are `None` correspond to the `NavigableString` nodes; only the first one of them is a *child* of the `p` tag, but both are *descendants* of that tag.

parent, parents

Given an instance *n* of any navigable class, its parent node is *n*`.parent`; an iterator on all ancestors, going upwards in the tree, is *n*`.parents`. This includes instances of `NavigableString`, since they have parents, too. An instance *b* of `BeautifulSoup` has *b*`.parent None`, and *b*`.parents` is an empty iterator.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.parent.name
'p'
```

next_sibling, previous_sibling, next_siblings, previous_siblings

Given an instance *n* of any navigable class, its sibling node to the immediate left is *n*`.previous_sibling`, and the one to the immediate right is *n*`.next_sibling`; either or both can be `None` if *n* has no such sibling. An iterator on all left siblings, going leftward in the tree, is *n*`.previous_siblings`; an iterator on all right siblings, going rightward in the tree, is *n*`.next_siblings` (either or both iterators can be empty). This includes instances of `NavigableString`, since they

have siblings, too. An instance *b* of `BeautifulSoup` has *b*`.previous_sibling` and *b*`.next_sibling` both `None`, and both of its sibling iterators are empty.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.previous_sibling, soup.b.next_sibling
('Plain ', None)
```

next_element, previous_element, next_elements, previous_elements

Given an instance *n* of any navigable class, the node parsed just before it is *n*`.previous_element`, and the one parsed just after it is *n*`.next_element`; either or both can be `None` when *n* is the first or last node parsed, respectively. An iterator on all previous elements, going backward in the tree, is *n*`.previous_elements`; an iterator on all following elements, going forward in the tree, is *n*`.next_elements` (either or both iterators can be empty). Instances of `NavigableString` have such attributes, too. An instance *b* of `BeautifulSoup` has *b*`.previous_element` and *b*`.next_element` both `None`, and both of its element iterators are empty.

```
>>> soup = bs4.BeautifulSoup('<p>Plain <b>bold</b></p>')
>>> soup.b.previous_element, soup.b.next_element
('Plain ', 'bold')
```

As shown in the previous example, the `b` tag has no `next_sibling` (since it's the last child of its parent); however, as shown here, it does have a `next_element`—the node parsed just after it (which in this case is the `'bold'` string it contains).

bs4 find... Methods ("Search Methods")

Each navigable class in `bs4` offers several methods whose names start with `find`, known as *search methods*, to locate tree nodes that satisfy conditions you specify.

Search methods come in pairs—one method of each pair walks all the relevant parts of the tree and returns a list of nodes satisfying the conditions, and the other one stops and returns a single node satisfying the conditions as soon as it finds it (or `None` when it finds no such node). So, calling the latter method is like calling the former one with argument `limit=1`, and indexing the resulting one-item list to get its single item, but a bit faster and more elegant.

So, for example, for any `Tag` instance *`t`* and any group of positional and named arguments represented by ..., the following equivalence always holds:

```
just_one = t.find(...)
other_way_list = t.find_all(..., limit=1)
other_way = other_way_list[0] if other_way_list else None
assert just_one == other_way
```

The method pairs are:

find, find_all	*b*`.find(...)` *b*`.find_all(...)` Searches the *descendants* of *b*, except that, if you pass named argument `recursive=False` (available only for these two methods, not for other search methods), it searches *b*'s *children* only. These methods are not available on `NavigableString` instances, since they have no descendants; all other search methods are available on `Tag` and `NavigableString` instances. Since `find_all` is frequently needed, `bs4` offers an elegant shortcut: calling a tag is like calling its `find_all` method. That is, *b*`(...)` is the same as *b*`.find_all(...)`. Another shortcut, already mentioned in "Attribute references on instances of BeautifulSoup and Tag," is that *b*`.foo.bar` is like *b*`.find('foo').find('bar')`.
find_next, find_all_next	*b*`.find_next(...)` *b*`.find_all_next(...)` Searches the `next_elements` of *b*.
find_next_sibling	*b*`.find_next_sibling(...)` *b*`.find_next_siblings(...)` Searches the `next_siblings` of *b*.

, **find_next_siblings**	
find_parent **,** **find_parents**	*b*.find_parent(...) *b*.find_parents(...) Searches the parents of *b*.
find_previous **,** **find_all_previous**	*b*.find_previous(...) *b*.find_all_previous(...) Searches the previous_elements of *b*.
find_previous_sibling **,** **find_previous_siblings**	*b*.find_previous_sibling(...) *b*.find_previous_siblings(...) Searches the previous_siblings of *b*.

Arguments of search methods

Each search method has three optional arguments: *name, attrs*, and *string. name* and *string* are *filters*, as described later in this section; *attrs* is a dict, also described later in this section. In addition, find and find_all only (not the other search methods) can optionally be called with the named argument *recursive*=False, to limit the search to children, rather than all descendants.

Any search method returning a list (i.e., one whose name is plural or starts with find_all) can optionally have the named argument limit, whose value, if passed, is an integer, putting an upper bound on the length of the list it returns.

After these optional arguments, each search method can optionally have any number of arbitrary named arguments, whose name can be any identifier (except the name of one of the search method's specific arguments), while the value is a filter.

Search method arguments: filters

A *filter* is applied against a *target* that can be a tag's name (when passed as the *`name`* argument); a `Tag`'s `string` or a `NavigableString`'s textual content (when passed as the *`string`* argument); or a `Tag`'s attribute (when passed as the value of a named argument, or in the *`attrs`* argument). Each filter can be:

A Unicode string

: The filter succeeds when the string exactly equals the target

A bytestring

: It's decoded to Unicode using `utf8`, and then the filter succeeds when the resulting Unicode string exactly equals the target

A regular expression object (AKA RE, as produced by `re.compile`, covered in "Regular Expressions and the re Module")

: The filter succeeds when the `search` method of the RE, called with the target as the argument, succeeds

A list of strings

: The filter succeeds if any of the strings exactly equals the target (if any of the strings are bytestrings, they're decoded to Unicode using `utf8`)

A function object

: The filter succeeds when the function, called with the `Tag` or `NavigableString` instance as the argument, returns `True`

`True`

The filter always succeeds

As a synonym of “the filter succeeds,” we also say, “the target matches the filter.”

Each search method finds the relevant nodes that match all of its filters (that is, it implicitly performs a logical `and` operation on its filters on each candidate node).

Search method arguments: name

To look for `Tags` whose `name` matches a filter, pass the filter as the first positional argument to the search method, or pass it as `name=`*`filter`*:

```
soup.find_all('b') # or soup.find_all(name='b')
# returns all instances of Tag 'b' in the document
soup.find_all(['b', 'bah'])
# returns all instances of Tags 'b' and 'bah' in the document
soup.find_all(re.compile(r'^b'))
# returns all instances of Tags starting with 'b' in the document
soup.find_all(re.compile(r'bah'))
# returns all instances of Tags including string 'bah' in the
document
def child_of_foo(tag):
  return tag.parent == 'foo'
soup.find_all(name=child_of_foo)
# returns all instances of Tags whose parent's name is 'foo'
```

Search method arguments: string

To look for `Tag` nodes whose `.string`’s text matches a filter, or `NavigableString` nodes whose text matches a filter, pass the filter as `string=`*`filter`*:

```
soup.find_all(string='foo')
# returns all instances of NavigableString whose text is 'foo'
soup.find_all('b', string='foo')
# returns all instances of Tag 'b' whose .string's text is 'foo'
```

Search method arguments: attrs

To look for tag nodes who have attributes whose values match filters, use a `dict` *`d`* with attribute names as keys, and filters as the corresponding values. Then, pass *`d`* as the second positional argument to the search method, or pass `attrs=`*`d`*.

As a special case, you can use, as a value in *`d`*, `None` instead of a filter; this matches nodes that *lack* the corresponding attribute.

As a separate special case, if the value *`f`* of `attrs` is not a `dict`, but a filter, that is equivalent to having an `attrs` of `{'class' : `*`f`*`}`. (This convenient shortcut helps because looking for tags with a certain CSS class is a frequent task.)

You cannot apply both special cases at once: to search for tags without any CSS class, you must explicitly pass `attrs={'class' : None}` (i.e., use the first special case, but not at the same time as the second one):

```
soup.find_all('b', {'foo': True, 'bar': None})
# returns all instances of Tag 'b' w/an attribute 'foo' and no
'bar'
```

Matching Tags with Multiple CSS Classes

Differently from most attributes, a tag can have multiple values for its attribute `'class'`. These are shown in HTML as a space-separated string (e.g., `'<p class='foo bar baz'>...'`), and in `bs4` as a list of strings (e.g., *`t`*`['class']` being `['foo', 'bar', 'baz']`).

When you filter by CSS class in any search method, the filter matches a tag if it matches any of the multiple CSS classes of such a tag.

To match tags by multiple CSS classes, you can write a custom function and pass it as the filter to the search method; or, if you don't need other added functionality of search methods, you can eschew search methods and instead use the method *`t`*`.select`, covered in "bs4 CSS Selectors," and go with the syntax of CSS selectors.

Search method arguments: other named arguments

Named arguments, beyond those whose names are known to the search method, are taken to augment the constraints, if any, specified in `attrs`. For example, calling a search method with *`foo=bar`* is like calling it with `attrs={'`*`foo`*`':`*`bar`*`}`.

bs4 CSS Selectors

`bs4` tags supply the methods `select` and `select_one`, roughly equivalent to `find_all` and `find` but accepting as the single argument a string that's a CSS selector and returning the list of tag nodes satisfying that selector, or, respectively, the first such tag node.

`bs4` supports only a subset of the rich CSS selector functionality, and we do not cover CSS selectors further in this book. (For complete coverage of CSS, we recommend the book *CSS: The Definitive Guide*, 4th Edition [O'Reilly].) In most cases, the search methods covered in "bs4 find... Methods ("Search Methods")" are better choices; however, in a few special cases, calling `select` can save you the (small) trouble of writing a custom filter function:

```
def foo_child_of_bar(t):
    return t.name=='foo' and t.parent and t.parent.name=='bar'
soup(foo_child_of_bar)
# returns tags with name 'foo' children of tags with name 'bar'
soup.select('foo < bar')
# exactly equivalent, with no custom filter function needed
```

An HTML Parsing Example with BeautifulSoup

The following example uses `bs4` to perform a typical task: fetch a page from the web, parse it, and output the HTTP hyperlinks in the page.

```
import urllib.request, urllib.parse, bs4
f = urllib.request.urlopen('http://www.python.org')
b = bs4.BeautifulSoup(f)
seen = set()
for anchor in b('a'):
    url = anchor.get('href')
    if url is None or url in seen:
        continue
```

```
        seen.add(url)
        pieces = urllib.parse.urlparse(url)
        if pieces[0]=='http':
            print(urllib.parse.urlunparse(pieces))
```

The example calls the instance of class `bs4.BeautifulSoup` (equivalent to calling its `find_all` method) to obtain all instances of a certain tag (here, tag `'<a>'`), then the `get` method of instances of the tag in question to obtain the value of an attribute (here, `'href'`), or `None` when that attribute is missing.

Generating HTML

Python does not come with tools specifically meant to generate HTML, nor with ones that let you embed Python code directly within HTML pages. Development and maintenance are eased by separating logic and presentation issues through *templating*, covered in "Templating." An alternative is to use `bs4` to create HTML documents, in your Python code, by gradually altering very minimal initial documents. Since these alterations rely on `bs4` *parsing* some HTML, using different parsers affects the output, as covered in "Which parser BeautifulSoup uses."

Editing and Creating HTML with bs4

You can alter the tag name of an instance `t` of `Tag` by assigning to `t.name`; you can alter `t`'s attributes by treating `t` as a mapping: assign to an indexing to add or change an attribute, or delete the indexing—for example, `del t['foo']` removes the attribute `foo`. If you assign some `str` to `t.string`, all previous `t.contents` (`Tag`s and/or strings—the whole subtree of `t`'s descendants) are discarded and replaced with a new `NavigableString` instance with that `str` as its textual content.

Given an instance `s` of `NavigableString`, you can replace its textual content: calling `s.replace_with('other')` replaces `s`'s text with `'other'`.

Building and adding new nodes

Altering existing nodes is important, but creating new ones and adding them to the tree is crucial for building an HTML document from scratch.

To create a new `NavigableString` instance, just call the class, with the textual content as the single argument:

```
s = bs4.NavigableString(' some text ')
```

To create a new `Tag` instance, call the `new_tag` method of a `BeautifulSoup` instance, with the tag name as the single positional argument, and optionally named arguments for attributes:

```
t = soup.new_tag('foo', bar='baz')
print(t)
<foo bar="baz"></foo>
```

To add a node to the children of a `Tag`, you can use the `Tag`'s `append` method to add the node at the end of the existing children, if any:

```
t.append(s)
print(t)
<foo bar="baz"> some text </foo>
```

If you want the new node to go elsewhere than at the end, at a certain index among *`t`*'s children, call *`t`*`.insert(`*`n, s`*`)` to put *`s`* at index *`n`* in *`t`*`.contents` (*`t`*`.append` and *`t`*`.insert` work as if *`t`* was a list of its children).

If you have a navigable element *`b`* and want to add a new node *`x`* as *`b`*'s `previous_sibling`, call *`b`*`.insert_before(`*`x`*`)`. If instead you want *`x`* to become *`b`*'s `next_sibling`, call *`b`*`.insert_after(`*`x`*`)`.

If you want to wrap a new parent node *`t`* around *`b`*, call *`b`*`.wrap(`*`t`*`)` (which also returns the newly wrapped tag). For example:

```
print(t.string.wrap(soup.new_tag('moo', zip='zaap')))
<moo zip="zaap"> some text </moo>
```

```
print(t)
<foo bar="baz"><moo zip="zaap"> some text </moo></foo>
```

Replacing and removing nodes

You can call `t.replace_with` on any tag `t`: the call replaces `t`, and all its previous contents, with the argument, and returns `t` with its original contents. For example:

```
soup = bs4.BeautifulSoup(
       '<p>first <b>second</b> <i>third</i></p>', 'lxml')
i = soup.i.replace_with('last')
soup.b.append(i)
print(soup)
<html><body><p>first <b>second<i>third</i></b> last</p></body>
</html>
```

You can call `t.unwrap()` on any tag `t`: the call replaces `t` with its contents, and returns `t` "emptied," that is, without contents. For example:

```
empty_i = soup.i.unwrap()
print(soup.b.wrap(empty_i))
<i><b>secondthird</b></i>
print(soup)
<html><body><p>first <i><b>secondthird</b></i> last</p></body>
</html>
```

`t.clear()` removes `t`'s contents, destroys them, and leaves `t` empty (but still in its original place in the tree). `t.decompose()` removes and destroys both `t` itself, and its contents. For example:

```
soup.i.clear()
print(soup)
<html><body><p>first <i></i> last</p></body></html>
soup.p.decompose()
print(soup)
<html><body></body></html>
```

Lastly, `t.extract()` removes `t` and its contents, but—doing no actual destruction—returns `t` with its original contents.

Building HTML with bs4

Here's an example of how to use `bs4`'s methods to generate HTML. Specifically, the following function takes a sequence of "rows" (sequences) and returns a string that's an HTML table to display their values:

```
def mktable_with_bs4(seq_of_rows):
  tabsoup = bs4.BeautifulSoup('<table>', 'html.parser')
  tab = tabsoup.table
  for row in seq_of_rows:
    tr = tabsoup.new_tag('tr')
    tab.append(tr)
    for item in row:
      td = tabsoup.new_tag('td')
      tr.append(td)
      td.string = str(item)
  return tab
```

Here is an example using the function we just defined:

```
example = (
  ('foo', 'g>h', 'g&h'),
  ('zip', 'zap', 'zop'),
)
print(mktable_with_bs4(example))
# prints:
<table><tr><td>foo</td><td>g&gt;h</td><td>g&amp;h</td></tr><tr>
<td>zip</td><td>zap</td><td>zop</td></tr></table>
```

Note that `bs4` automatically "escapes" strings containing mark-up characters such as <, >, and `&`; for example, `'g>h'` renders as `'g>h'`.

Templating

To generate HTML, the best approach is often *templating*. Start with a *template*, a text string (often read from a file, database, etc.) that is almost valid HTML, but includes markers, known as *placeholders*, where dynamically generated text must be inserted. Your program generates the needed text and substitutes it into the template.

In the simplest case, you can use markers of the form `{`*`name`*`}`. Set the dynamically generated text as the value for key `'`*`name`*`'` in some dictionary

d. The Python string formatting method `.format` (covered in "String Formatting") lets you do the rest: when *t* is the template string, `t.format(d)` is a copy of the template with all values properly substituted.

In general, beyond substituting placeholders, you also want to use conditionals, perform loops, and deal with other advanced formatting and presentation tasks; in the spirit of separating "business logic" from "presentation issues," you'd prefer it if all of the latter were part of your templating. This is where dedicated third-party templating packages come in. There are many of them, but all of this book's authors, having used and authored some in the past, currently prefer jinja2, covered next.

The jinja2 Package

For serious templating tasks, we recommend jinja2 (available on PyPI, like other third-party Python packages, so, easily installable with **pip install jinja2**).

The jinja2 docs are excellent and thorough, covering the templating language itself (conceptually modeled on Python, but with many differences to support embedding it in HTML, and the peculiar needs specific to presentation issues); the API your Python code uses to connect to jinja2, and expand or extend it if necessary; as well as other issues, from installation to internationalization, from sandboxing code to porting from other templating engines—not to mention, precious tips and tricks.

In this section, we cover only a tiny subset of `jinja2`'s power, just what you need to get started after installing it: we earnestly recommend studying `jinja2`'s docs to get the huge amount of extra, useful information they effectively convey.

The jinja2.Environment Class

When you use `jinja2`, there's always an `Environment` instance involved—in a few cases you could let it default to a generic "shared environment," but that's not recommended. Only in very advanced usage,

when you're getting templates from different sources (or with different templating language syntax), would you ever define multiple environments—usually, you instantiate a single `Environment` instance *`env`*, good for all the templates you need to render.

You can customize *`env`* in many ways as you build it, by passing named arguments to its constructor (including altering crucial aspects of templating language syntax, such as which delimiters start and end blocks, variables, comments, etc.), but the one named argument you'll almost always pass in real-life use is `loader=...`.

An environment's `loader` specifies where to load templates from, on request—usually some directory in a filesystem, or perhaps some database (you'd have to code a custom subclass of `jinja2.Loader` for the latter purpose), but there are other possibilities. You need a loader to let templates enjoy some of `jinja2`'s most powerful features, such as *template inheritance*.

You can equip *`env`*, as you instantiate it, with custom filters, tests, extensions, and so on (each of those can also be added later).

In the following sections' examples, we assume *`env`* was instantiated with nothing but `loader=jinja2.FileSystemLoader('`*`/path/to/templates`*`')`, and not further enriched—in fact, for simplicity, we won't even make use of the loader. In real life, however, the loader is almost invariably set; other options, seldom.

`env``.get_template(`*`name`*`)` fetches, compiles, and returns an instance of `jinja2.Template` based on what `env.loader(`*`name`*`)` returns. In the following examples, for simplicity, we'll instead use the rarely-warranted `env.from_string(`*`s`*`)` to build an instance of `jinja2.Template` from string *`s`*.

The jinja2.Template Class

An instance *t* of `jinja2.Template` has many attributes and methods, but the one you'll be using almost exclusively in real life is:

render	*t*`.render(...`*context*`...)` The *context* argument(s) are the same you might pass to a `dict` constructor—a mapping instance, and/or named arguments enriching and potentially overriding the mapping's key-to-value connections. *t*`.render(`*context*`)` returns a (Unicode) string resulting from the *context* arguments applied to the template *t*.

Building HTML with jinja2

Here's an example of how to use a jinja2 template to generate HTML. Specifically, just like previously in "Building HTML with bs4," the following function takes a sequence of "rows" (sequences) and returns an HTML table to display their values:

```
TABLE_TEMPLATE = '''\
<table>
{% for s in s_of_s %}
  <tr>
  {% for item in s %}
    <td>{{item}}</td>
  {% endfor %}
  </tr>
{% endfor %}
</table>'''
def mktable_with_jinja2(s_of_s):
    env = jinja2.Environment(
        trim_blocks=True,
        lstrip_blocks=True,
        autoescape=True)
    t = env.from_string(TABLE_TEMPLATE)
    return t.render(s_of_s=s_of_s)
```

The function builds the environment with option `autoescape=True`, to automatically "escape" strings containing mark-up characters such as <, >, and `&`; for example, with `autoescape=True`, `'g>h'` renders as `'g>h'`.

The options `trim_blocks=True` and `lstrip_blocks=True` are purely cosmetic, just to ensure that both the template string and the

rendered HTML string can be nicely formatted; of course, when a browser renders HTML, it does not matter whether the HTML itself is nicely formatted.

Normally, you would always build the environment with option `loader=...`, and have it load templates from files or other storage with method calls such as *`t`* = *`env`*`.get_template(`*`template_name`*`)`. In this example, just in order to present everything in one place, we omit the loader and build the template from a string by calling method *`env`*`.from_string` instead. Note that `jinja2` is not HTML- or XML-specific, so its use alone does not guarantee the validity of the generated content, which you should carefully check if standards conformance is a requirement.

The example uses only the two most common features out of the many dozens that the `jinja2` templating language offers: *loops* (that is, blocks enclosed in `{% for ... %}` and `{% endfor %}`) and *parameter substitution* (inline expressions enclosed in `{{ and }}`).

Here is an example use of the function we just defined:

```
example = (
  ('foo', 'g>h', 'g&h'),
  ('zip', 'zap', 'zop'),
)
print(mktable_with_jinja2(example))
# prints:
<table>
  <tr>
    <td>foo</td>
    <td>g&gt;h</td>
    <td>g&amp;h</td>
  </tr>
  <tr>
    <td>zip</td>
    <td>zap</td>
    <td>zop</td>
  </tr>
</table>
```

---

1 Except perhaps for its latest version (HTML5), when properly applied

2 The BeautifulSoup documentation provides detailed information about installing various parsers.

3 As explained in the BeautifulSoup documentation, which also shows various ways to guide or override BeautifulSoup's guesses.

Chapter 10. Structured Text: XML

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 23rd chapter of the final book. Please note that example code will be hosted at *https://github.com/holdenweb/pynut4*.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at *pynut4@gmail.com*.

XML, the *eXtensible Markup Language*, is a widely used data-interchange format. On top of XML itself, the XML community (in good part within the World Wide Web Consortium [W3C]) has standardized many other technologies, such as schema languages, namespaces, XPath, XLink, XPointer, and XSLT.

Industry consortia have defined industry-specific markup languages on top of XML for data exchange among applications in their respective fields. XML, XML-based markup languages, and other XML-related technologies are often used for inter-application, cross-language, cross-platform data interchange in specific industries.

Python's standard library, for historical reasons, has multiple modules supporting XML under the `xml` package, with overlapping functionality; this book does not cover them all--see the online documentation.

This book (and, specifically, this chapter) covers only the most Pythonic approach to XML processing: `ElementTree`, whose elegance, speed, generality, multiple implementations, and Pythonic architecture make it the package of choice for Python XML applications. For complete tutorials and

all details on the `xml.etree.ElementTree` module, see the online docs) and the website of `ElementTree`'s creator, deeply-missed Fredrik Lundh, best known as "the effbot."[1]

This book takes for granted some elementary knowledge of XML itself; if you need to learn more about XML, we recommend the book *XML in a Nutshell* (O'Reilly).

Parsing XML from untrusted sources puts your application at risk for many possible attacks; this book does not cover this issue specifically–see the online documentation, which recommends third-party modules to help safeguard your application if you do have to parse XML from sources you can't fully trust. In particular, if you need an `ElementTree` implementation with safeguards against parsing untrusted sources, consider defusedxml.ElementTree.

ElementTree

Python and third-party add-ons offer several alternative implementations of the `ElementTree` functionality; the one you can always rely on in the standard library is the module `xml.etree.ElementTree`. Just importing `xml.etree.ElementTree` gets you the fastest implementation available in your Python installation's standard library. The third-party package `defusedxml`, mentioned in the previous section of this chapter, offers slightly slower but safer implementations if you ever need to parse XML from untrusted sources; another third-party package, lxml, gets you faster performance, and some extra functionality, via lxml.etree.

Traditionally, you get whatever available implementation of `ElementTree` you prefer, by a `from...import...as` statement such as:

```
from xml.etree import ElementTree as et
```

(or more than one such statement, with `try...except ImportError:` guards to discover what's the best implementation available), then use `et` (some prefer the uppercase variant, `ET`) as the module's name in the rest of your code.

`ElementTree` supplies one fundamental class representing a *node* within the *tree* that naturally maps an XML document, the class `Element`. `ElementTree` also supplies other important classes, chiefly the one representing the whole tree, with methods for input and output and many convenience classes equivalent to ones on its `Element` *root*—that's the class `ElementTree`. In addition, the `ElementTree` module supplies several utility functions, and auxiliary classes of lesser importance.

The Element Class

The `Element` class represents a node in the tree that maps an XML document, and it's the core of the whole `ElementTree` ecosystem. Each element is a bit like a mapping, with *attributes* that are a mapping from string keys to string values, and a bit like a sequence, with *children* that are other elements (sometimes referred to as the element's "subelements"). In addition, each element offers a few extra attributes and methods. Each `Element` instance *e* has four data attributes or properties:

attrib	A `dict` containing all of the XML node's attributes, with strings, the attributes' names, as its keys (and, usually, strings as corresponding values as well). For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *e* whose *e*`.attrib` is `{'x': 'y'}`.
tag	The XML tag of the node, a string, sometimes also known as "the element's *type*." For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *e* with *e*`.tag` set to `'a'`.
tail	Arbitrary data (a string) immediately "following" the element. For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *e* with *e*`.tail` set to `'c'`.
text	Arbitrary data (a string) directly "within" the element. For example, parsing the XML fragment `<a x="y">b</a>c`, you get an *e* with *e*`.text` set to `'b'`.

Avoid Accessing Attrib On Element Instances, If Feasible

It's normally best to avoid accessing *e*`.attrib` when possible, because the implementation might need to build it on the fly when you access it. *e* itself, as covered later in this section, offers some typical mapping methods that you might otherwise want to call on *e*`.attrib`; going through *e*'s own methods allows a smart implementation to optimize things for you, compared to the performance you'd get via the actual `dict` *e*`.attrib`.

e has some methods that are mapping-like and avoid the need to explicitly ask for the *e*`.attrib dict`:

clear	*e*`.clear()` *e*`.clear()` leaves *e* "empty," except for its `tag`, removing all attributes and children, and setting `text` and `tail` to `None`.
get	*e*`.get(`*key*, *default*=`None)` Like *e*`.attrib.get(`*key*, *default*`)`, but potentially much faster. You cannot use *e*`[`*key*`]`, since indexing on *e* is used to access children, not attributes.
items	*e*`.items()` Returns the list of `(`*name, value*`)` tuples for all attributes, in arbitrary order.
keys	*e*`.keys()` Returns the list of all attribute names, in arbitrary order.
set	*e*`.set(`*key, value*`)` Sets the value of the attribute named *key* to *value*.

The other methods of *e* (including indexing with the *e*`[i]` syntax, and length as in `len(`*e*`)`) deal with all *e*'s children as a sequence, or in some cases—indicated in the rest of this section—with all descendants (elements in the subtree rooted at *e*, also known as *subelements* of *e*).

Don’t Rely On Implicit Bool Conversion Of An Element

In all versions up through Python 3.10, an `Element` instance *e* evaluates as false if *e* has no children, following the normal rule for Python containers’ implicit `bool` conversion. However, it is documented that this behavior may change in some future version of Python. For future compatibility, if you want to check whether *e* has no children, explicitly check `if len(`*e*`) == 0:`—don’t use the normal Python idiom `if not` *e*`:`.

The named methods of *e* dealing with children or descendants are the following (we do not cover XPath in this book: see the online docs). Many of the following methods take an optional argument `namespaces`, defaulting to `None`. When present, `namespaces` is a mapping with XML namespace prefixes as keys and corresponding XML namespace full names as values.

append	*e*`.append(`*se*`)` Adds subelement *se* (which must be an `Element`) at the end of *e*’s children.
extend	*e*`.extend(`*ses*`)` Adds each item of iterable *ses* (every item must be an `Element`) at the end of *e*’s children.
find	*e*`.find(`*match*, *namespaces*`=None)` Returns the first descendant matching *match*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. Returns `None` if no descendant matches *match.*
findall	*e*`.findall(`*match*, *namespaces*`=None)` Returns the list of all descendants matching *match*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. Returns `[]` if no descendants match *match.*
findtext	*e*`.findtext(`*match*, *default*`=None`, *namespaces*`=None)` Returns the `text` of the first descendant matching *match*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. The result may be an empty string `''` if the first descendant matching *match* has no `text`. Returns *default* if no descendant matches *match.*

insert	*e*.`insert`(*index, se*) Adds subelement *se* (which must be an `Element`) at index *index* within the sequence of *e*'s children.
iter	*e*.`iter`(*tag*=`'*'`) Returns an iterator walking in depth-first order over all of *e*'s descendants. When *tag* is not `'*'`, only yields subelements whose `tag` equals *tag*. Don't modify the subtree rooted at *e* while you're looping on *e*.`iter`.
iterfind	*e*.`iterfind`(*match*, *namespaces*=`None`) Returns an iterator over all descendants, in depth-first order, matching *match*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. The resulting iterator is empty when no descendants match *match*.
itertext	*e*.`itertext`(*match*, *namespaces*=`None`) Returns an iterator over the `text` (not the `tail`) attribute of all descendants, in depth-first order, matching *match*, which may be a tag name or an XPath expression within the subset supported by the current implementation of `ElementTree`. The resulting iterator is empty when no descendants match *match*.
remove	*e*.`remove`(*se*) Removes the descendant that `is` element *se* (as covered in Identity tests, in Table 3-2).

The ElementTree Class

The `ElementTree` class represents a tree that maps an XML document. The core added value of an instance *et* of `ElementTree` is to have methods for wholesale parsing (input) and writing (output) of a whole tree (Table 23-1), namely:

Table 10-1. ElementTree instance parsing and writing methods

parse	*et*.`parse`(*source,parser*=`None`) *source* can be a file open for reading, or the name of a file to open and read (to parse a string, wrap it in `io.StringIO`, covered in "In-Memory "Files": io.StringIO and io.BytesIO"), containing XML text. *et*.`parse` parses that text, builds its tree of `Elements` as the new content of *et* (discarding the previous content of *et*, if any), and returns the root element of the tree. *parser* is an optional parser instance; by default, *et*.`parse` uses an instance of class `XMLParser` supplied by the `ElementTree` module (this book does not cover `XMLParser`; see the online docs).
	et.`write`(*file, encoding*=`'us-ascii'`, *xml_declaration*=`None`,

write	*default_namespace*=None, *method*='xml', *short_empty_elements*=True) *file* can be a file open for writing, or the name of a file to open and write (to write into a string, pass as *file* an instance of `io.StringIO`, covered in "In-Memory "Files": io.StringIO and io.BytesIO"). *et*`.write` writes into that file the text representing the XML document for the tree that's the content of *et*. *encoding* should be spelled according to the standard—for example, `'iso-8859-1'`, not `'latin-1'`, even though Python itself accepts both spellings for this encoding. You can also pass *encoding* as `'unicode'`; this outputs text (Unicode) strings, when *file*`.write` accepts such strings; otherwise, *file*`.write` must accept bytestrings, and that is the type of strings *et*`.write` outputs, using XML character references for characters not in the encoding—for example, with the default ASCII encoding, "e with an acute accent", é, is output as `&#233;`. You can pass *xml_declaration* as `False` to not have the declaration in the resulting text, as `True` to have it; the default is to have the declaration in the result only when *encoding* is not one of `'us-ascii'`, `'utf-8'`, or `'unicode'`. You can optionally pass *default_namespace* to set the default namespace for `xmlns` constructs. You can pass *method* as `'text'` to output only the `text` and `tail` of each node (no tags). You can pass *method* as `'html'` to output the document in HTML format (which, for example, omits end tags not needed in HTML, such as </br>). The default is `'xml'`, to output in XML format. You can optionally (only by name, not positionally) pass *short_empty_elements* as `False` to always use explicit start and end tags, even for elements that have no text or subelements; the default is to use the XML short form for such empty elements. For example, an empty element with tag a is output as <a/> by default, as <a> if you pass *short_empty_elements* as `False`.

In addition, an instance *et* of `ElementTree` supplies the method `getroot`—*et*`.getroot()` returns the root of the tree—and the convenience methods `find`, `findall`, `findtext`, `iter`, and `iterfind`, each exactly equivalent to calling the same method on the root of the tree—that is, on the result of *et*`.getroot()`.

Functions in the ElementTree Module

The `ElementTree` module also supplies several functions, described in Table 23-2.

Table 10-2. Caption to come

Comment	`Comment(`*text*`=None)` Returns an `Element` that, once inserted as a node in an `ElementTree`, will be output as an XML comment with the given *text* string enclosed between `'<!--'` and `'-->'`. `XMLParser` skips XML comments in any document it parses, so this function is the only way to get comment nodes.
ProcessingInstruction	`ProcessingInstruction(`*target,text*`=None)` Returns an `Element` that, once inserted as a node in an `ElementTree`, will be output as an XML processing instruction with the given *target* and *text* strings enclosed between `'<?'` and `'?>'`. `XMLParser` skips XML processing instructions in any document it parses, so this function is the only way to get processing instruction nodes.
SubElement	`SubElement(`*parent, tag, attrib*`={}, **`*extra*`)` Creates an `Element` with the given *tag*, attributes from `dict` *attrib* and others passed as named arguments in *extra*, and appends it as the rightmost child of `Element` *parent*; returns the `Element` it has created.
XML	`XML(`*text,parser*`=None)` Parses XML from the *text* string and returns an `Element`. *parser* is an optional parser instance; by default, `XML` uses an instance of the class `XMLParser` supplied by the `ElementTree` module (this book does not cover class `XMLParser`; see the online docs).
XMLID	`XMLID(`*text,parser*`=None)` Parses XML from the *text* string and returns a tuple with two items: an `Element` and a `dict` mapping `id` attributes to the only `Element` having each (XML forbids duplicate `id`s). *parser* is an optional parser instance; by default, `XMLID` uses an instance of the class `XMLParser` supplied by the `ElementTree` module (this book does not cover the `XMLParser` class; see the online docs).
dump	`dump(`*e*`)` Writes `e`, which can be an `Element` or an `ElementTree`, as XML to `sys.stdout`; it is meant only for debugging purposes.
fromstring	`fromstring(`*text,parser*`=None)` Parses XML from the *text* string and returns an `Element`, just like the XML function just covered.
fromstringlist	`fromstringlist(`*sequence,parser=None*`)` Just like `fromstring(''.join(`*sequence*`))`, but can be a bit faster by avoiding the `join`.
iselement	`iselement(`*e*`)` Returns `True` if *e* is an `Element`.
	`iterparse(`*source,events*`=['end'],`*parser*`=None)`

iterparse	*source* can be a file open for reading, or the name of a file to open and read, containing an XML document as text. `iterparse` returns an iterator yielding tuples `(`*event, element*`)`, where *event* is one of the strings listed in argument *events* (each string must be `'start'`, `'end'`, `'start-ns'`, or `'end-ns'`), as the parsing progresses and `iterparse` incrementally builds the corresponding `ElementTree`. *element* is an `Element` for events `'start'` and `'end'`, `None` for event `'end-ns'`, and a tuple of two strings `(`*namespace_prefix, namespace_uri*`)` for event `'start-ns'`. `parser` is an optional parser instance; by default, `iterparse` uses an instance of the class `XMLParser` supplied by the `ElementTree` module (this book does not cover class `XMLParser`; see the online docs). The purpose of `iterparse` is to let you iteratively parse a large XML document, without holding all of the resulting `ElementTree` in memory at once, whenever feasible. We cover `iterparse` in more detail in "Parsing XML Iteratively".
parse	`parse(`*source,parser*`=None)` Just like the `parse` method of `ElementTree`, covered in Table 23-1, except that it returns the `ElementTree` instance it creates.
register_namespace	`register_namespace(`*prefix,uri*`)` Registers the string *prefix* as the namespace prefix for the string *uri*; elements in the namespace get serialized with this prefix.
tostring	`tostring(`*e,encoding*`='us-ascii',`*method*`='xml',` *short_empty_elements*`=True)` Returns a string with the XML representation of the subtree rooted at `Element` *e*. Arguments have the same meaning as for the `write` method of `ElementTree`, covered in Table 23-1.
tostringlist	`tostringlist(`*e,encoding*`='us-ascii',`*method*`='xml',`*short_empty_elements*`=True)` Returns a list of strings with the XML representation of the subtree rooted at `Element` *e*. Arguments have the same meaning as for the `write` method of `ElementTree`, covered in Table 23-1.

The `ElementTree` module also supplies the classes `QName`, `TreeBuilder`, and `XMLParser`, which we do not cover in this book, and the class `XMLPullParser`, covered in "Parsing XML Iteratively."

Parsing XML with ElementTree.parse

In everyday use, the most common way to make an `ElementTree` instance is by parsing it from a file or file-like object, usually with the

module function `parse` or with the method `parse` of instances of the class `ElementTree`.

For the examples in this chapter, we use the simple XML file found at *http://www.w3schools.com/xml/simple.xml*; its root tag is `'breakfast_menu'`, and the root's children are elements with the tag `'food'`. Each `'food'` element has a child with the tag `'name'`, whose text is the food's name, and a child with the tag `'calories'`, whose text is the string representation of the integer number of calories in a portion of that food. In other words, a simplified representation of that XML file's content of interest to the examples is:

```
<breakfast_menu>
  <food>
    <name>Belgian Waffles</name>
    <calories>650</calories>
  </food>
  <food>
    <name>Strawberry Belgian Waffles</name>
    <calories>900</calories>
  </food>
  <food>
    <name>Berry-Berry Belgian Waffles</name>
    <calories>900</calories>
  </food>
  <food>
    <name>French Toast</name>
    <calories>600</calories>
  </food>
  <food>
    <name>Homestyle Breakfast</name>
    <calories>950</calories>
  </food>
</breakfast_menu>
```

Since the XML document lives at a WWW URL, you start by obtaining a file-like object with that content, and passing it to `parse`; the simplest way uses the `urllib.request` module:

```
from urllib import request
from xml.etree import ElementTree as et
content =
```

```
request.urlopen('http://www.w3schools.com/xml/simple.xml')
tree = et.parse(content)
```

Selecting Elements from an ElementTree

Let's say that we want to print on standard output the calories and names of the various foods, in order of increasing calories, with ties broken alphabetically. The code for this task:

```
def bycal_and_name(e):
    return int(e.find('calories').text), e.find('name').text
for e in sorted(tree.findall('food'), key=bycal_and_name):
    print(f"{e.find('calories').text} {e.find('name').text}")
```

When run, this prints:

```
600 French Toast
650 Belgian Waffles
900 Berry-Berry Belgian Waffles
900 Strawberry Belgian Waffles
950 Homestyle Breakfast
```

Editing an ElementTree

Once an `ElementTree` is built (be that via parsing, or otherwise), it can be "edited"—inserting, deleting, and/or altering nodes (elements)—via various methods of `ElementTree` and `Element` classes, and module functions. For example, suppose our program is reliably informed that a new food has been added to the menu—buttered toast, two slices of white bread toasted and buttered, 180 calories—while any food whose name contains "berry," case-insensitive, has been removed. The "editing the tree" part for these specs can be coded as follows:

```
# add Buttered Toast to the menu
menu = tree.getroot()
toast = et.SubElement(menu, 'food')
tcals = et.SubElement(toast, 'calories')
tcals.text = '180'
tname = et.SubElement(toast, 'name')
tname.text = 'Buttered Toast'
# remove anything related to 'berry' from the menu
for e in menu.findall('food'):
```

```
    name = e.find('name').text
    if 'berry' in name.lower():
        menu.remove(e)
```

Once we insert these "editing" steps between the code parsing the tree and the code selectively printing from it, the latter prints:

```
180 Buttered Toast
600 French Toast
650 Belgian Waffles
950 Homestyle Breakfast
```

The ease of "editing" an `ElementTree` can sometimes be a crucial consideration, making it worth your while to keep it all in memory.

Building an ElementTree from Scratch

Sometimes, your task doesn't start from an existing XML document: rather, you need to make an XML document from data your code gets from a different source, such as a CSV document or some kind of database.

The code for such tasks is similar to the one we showed for editing an existing `ElementTree`—just add a little snippet to build an initially empty tree.

For example, suppose you have a CSV file, *menu.csv*, whose two comma-separated columns are the calories and name of various foods, one food per row. Your task is to build an XML file, *menu.xml*, similar to the one we parsed in previous examples. Here's one way you could do that:

```
import csv
from xml.etree import ElementTree as et
menu = et.Element('menu')
tree = et.ElementTree(menu)
with open('menu.csv') as f:
    r = csv.reader(f)
    for calories, namestr in r:
        food = et.SubElement(menu, 'food')
        cals = et.SubElement(food, 'calories')
        cals.text = calories
        name = et.SubElement(food, 'name')
```

```
        name.text = namestr
tree.write('menu.xml')
```

Parsing XML Iteratively

For tasks focused on selecting elements from an existing XML document, sometimes you don't need to build the whole `ElementTree` in memory—a consideration that's particularly important if the XML document is very large (not the case for the tiny example document we've been dealing with, but stretch your imagination and visualize a similar menu-focused document that lists millions of different foods).

So, again, what we want to do is print on standard output the calories and names of foods, this time only the 10 lowest-calorie foods, in order of increasing calories, with ties broken alphabetically; and *menu.xml*, which for simplicity's sake we now suppose is a local file, lists millions of foods, so we'd rather not keep it all in memory at once, since, obviously, we don't need complete access to all of it at once.

Here's some code that one might think would let us ace this task:

```
import heapq
from xml.etree import ElementTree as et
def cals_and_name():
    # generator for (calories, name) pairs
    for _, elem in et.iterparse('menu.xml'):
        if elem.tag != 'food':
            continue
        # just finished parsing a food, get calories and name
        cals = int(elem.find('calories').text)
        name = elem.find('name').text
        yield (cals, name)
lowest10 = heapq.nsmallest(10, cals_and_name)
for cals, name in heap:
    print(cals, name)
```

Simple But Memory-Intensive Approach

This approach does indeed work, but it consumes just about as much memory as an approach based on a full `et.parse` would!

Why does the simple approach still eat up memory? Because `iterparse`, as it runs, builds up a whole `ElementTree` in memory, incrementally, even though it only communicates back events such as (by default) just `'end'`, meaning "I just finished parsing this element."

To actually save memory, we can at least toss all contents of each element as soon as we're done processing it—that is, right after the `yield`, add `elem.clear()` to make the just-processed element empty.

This approach would indeed save some memory—but not all of it, because the tree's root would end up with a huge list of empty children nodes. To be really frugal in memory consumption, we need to get `'start'` events as well, so we can get hold of the root of the `ElementTree` being built, and remove each element from it as it's used, rather than just clearing the element—that is, change the generator into:

```
def cals_and_name():
    # memory-thrifty generator for (calories, name) pairs
    root = None
    for event, elem in et.iterparse('menu.xml', ['start',
'end']):
        if event == 'start':
            if root is not None:
                root = elem
            continue
        if elem.tag != 'food':
            continue
        # just finished parsing a food, get calories and name
        cals = int(elem.find('calories').text)
        name = elem.find('name').text
        yield (cals, name)
        root.remove(elem)
```

This approach saves as much memory as feasible, and still gets the task done!

Parsing XML within an asynchronous loop

While `iterparse`, used correctly, can save memory, it's still not good enough to use within an asynchronous loop. That's because `iterparse` makes blocking `read` calls to the file object passed as its first argument: such blocking calls are a no-no in async processing.

`ElementTree` offers the class `XMLPullParser` to help with this issue. See the ElementTree docs for the class's usage pattern.

---

1 Alex is far too modest to mention it, but from around 1995 to 2005 both he and Fredrik were, along with Tim Peters, the Python bots. Known as such for their encyclopedic and detailed knowledge of the language, the effbot, the martellibot, and the timbot have created software and documentation that are of immense value to millions of people.

About the Authors

Alex Martelli has been programming for 40 years, mainly in Python for the recent half of that time. He wrote the first two editions of Python in a Nutshell, and coauthored the first two editions of the *Python Cookbook* and the third edition of *Python in a Nutshell*. He is a PSF Fellow and Core Committer (emeritus), and won the 2002 Activators' Choice Award and the 2006 Frank Willison Memorial Award for contributions to the Python community. He is active on Stack Overflow and a frequent speaker at technical conferences. He's been living in Silicon Valley with his wife Anna for 17 years, and working at Google throughout this time, currently as Senior Staff Engineer in Google Cloud Tech Support.

Anna Martelli Ravenscroft is a PSF Fellow and winner of the 2013 Frank Willison Memorial Award for contributions to the Python community. She co-authored the second edition of the Python Cookbook and 3rd edition of *Python in a Nutshell*. She has been a technical reviewer for many Python books and is a regular speaker and track chair at technical conferences. Anna lives in Silicon Valley with her husband Alex, two dogs, one cat, and several chickens.

Passionate about programming and community, **Steve Holden** has worked with computers since 1967 and started using Python at version 1.4 in 1995. He has since written about Python, created instructor-led training, delivered it to an international audience built 40 hours of video training for "reluctant Python users." An Emeritus Fellow of the Python Software Foundation, Steve served as a director of the Foundation for eight years and as its chairman for three; he created PyCon, the Python community's international conference series and was presented with the Simon Willison Award for services to the Python community. He lives in Hastings, England and works as Technical Architect for the UK Department for International Trade, where he is responsible for the systems that maintain and regulate the trading environment.

Paul McGuire has been programming for 40+ years, in languages ranging from FORTRAN to Pascal, PL/I, COBOL, Smalltalk, Java, C/C++/C#, and Tcl, settling on Python as his language-of-choice in 2001. He is the author

and maintainer of the popular pyparsing module, as well as littletable and plusminus. Paul authored the O'Reilly Short Cut Getting Started with Pyparsing, and has written and edited articles for Python Magazine. He has also spoken at PyCon and at the Austin Python Users' Group, and is active on StackOverflow. Paul now lives in Austin, Texas with his wife and dog, and works for Indeed as a Site Reliability Engineer, helping people get jobs!