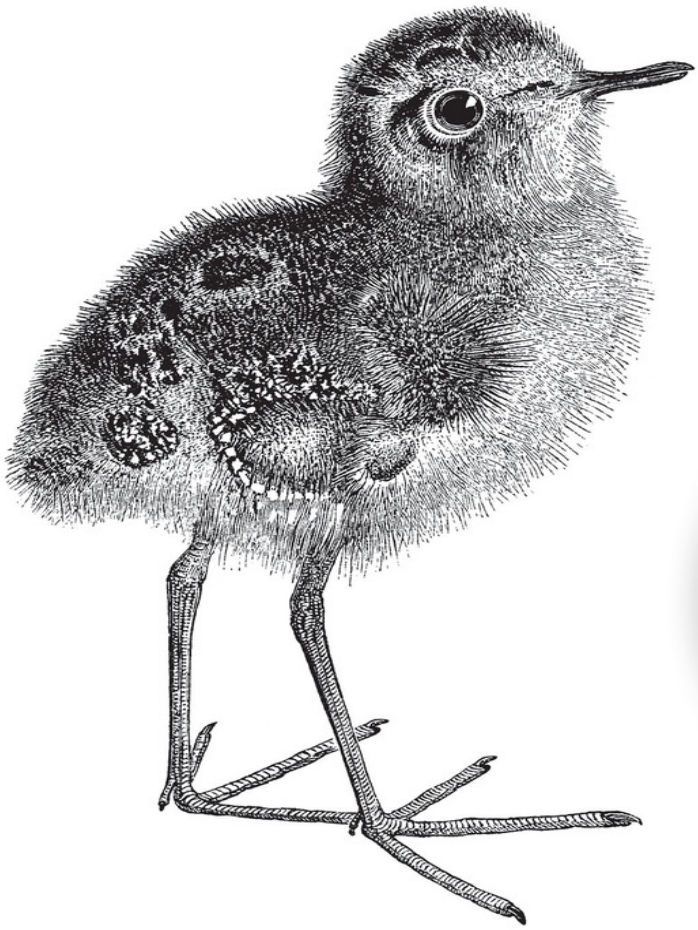
O'REILLY®

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
- 2022-04-26: Third Release
- 2022-06-13: Fourth Release

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

978-1-098-11349-0

Chapter 1. Introduction to Python

> **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 1st 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 is a well-established general-purpose programming language. Guido van Rossum, Python's creator, started developing Python back in 1990. This stable and mature language is high-level, dynamic, object-oriented, and cross-platform—all very attractive characteristics. Python runs on all major hardware platforms and operating systems, so it doesn't constrain your choices.

Python offers high productivity for all phases of the software life cycle: analysis, design, prototyping, coding, testing, debugging, tuning, documentation, deployment, and, of course, maintenance. Python's popularity has seen steadily increasing growth for many years, becoming the TIOBE Index leader in October 2021. Today, familiarity with Python is a plus for every programmer, as Python has snuck into most niches, with useful roles to play in any software solution.

Python provides a unique mix of elegance, simplicity, practicality, and sheer power. You'll quickly become productive with Python, thanks to its consistency and regularity, its rich standard library, and the many third-party packages and tools that are readily available for it. Python is easy to

learn, so it is quite suitable if you are new to programming, yet is also powerful enough for the most sophisticated expert.

The Python Language

The Python language, while not minimalist, is spare, for good pragmatic reasons. Once a language offers one good way to express a design, adding other ways has, at best, modest benefits; the cost of language complexity, though, grows more than linearly with the number of features. A complicated language is harder to learn and master (and to implement efficiently and without bugs) than a simpler one. Complications and quirks in a language hamper productivity in software development, particularly in large projects, where many developers cooperate and, often, maintain code originally written by others.

Python is simple, but not simplistic. It adheres to the idea that, if a language behaves a certain way in some contexts, it should ideally work similarly in all contexts. Python follows the principle that a language should not have "convenient" shortcuts, special cases, ad-hoc exceptions, overly-subtle distinctions, or mysterious and tricky under-the-covers optimizations. A good language, like any other well-designed artifact, must balance general principles with taste, common sense, and a lot of practicality.

Python is a general-purpose programming language: Python's traits are useful in almost any area of software development. There is no area where Python cannot be part of a solution. "Part" is important here; while many developers find that Python fills all of their needs, it does not have to stand alone: Python programs can cooperate with a variety of other software components, making it the right language for gluing together components in other languages. A design goal of the language is, and has long been, to "play well with others."

Python is a very high-level language (VHLL). This means that it uses a higher level of abstraction, conceptually further away from the underlying machine than classic compiled languages such as C, C++, and Java, traditionally called "high-level languages." Python is simpler, faster to

process (both for humans and for tools), and more regular, than classic high-level languages. This affords high programmer productivity, making Python a strong development tool. Good compilers for classic compiled languages can generate binary code that runs faster than Python. In most cases, however, the performance of Python-coded applications is sufficient. When it isn't, apply the optimization techniques covered in Chapter 16 to improve your program's performance while keeping the benefit of high productivity.

In terms of language level, Python is comparable to other powerful VHLLs like JavaScript, Ruby, and Perl. The advantages of simplicity and regularity, however, remain on Python's side.

Python is an object-oriented programming language, but lets you program in both object-oriented and procedural styles, with a touch of functional programming, too, mixing and matching as your application requires. Python's object-oriented features are conceptually similar to those of C++ but simpler to use.

The Python Standard Library and Extension Modules

There is more to Python programming than just the language: the standard library, and other extension modules are almost as important for Python use as the language itself. The Python standard library supplies many well-designed, solid, pure Python modules for convenient reuse. It includes modules for such tasks as representing data, processing text, interacting with the operating system and filesystem, and web programming. Since these modules are written in Python, they work on all platforms supported by Python.

Extension modules, from the standard library or from elsewhere, let Python code access functionality supplied by the underlying operating system or other software components, such as graphical user interfaces (GUIs), databases, and networks. Extensions also afford great speed in computationally intensive tasks such as XML parsing and numeric array

computations. Extension modules that are not coded in Python, however, do not necessarily enjoy the same cross-platform portability as pure Python code.

You can write extension modules in lower-level languages to optimize performance for small, computationally intensive parts that you originally prototyped in Python. You can also use tools such as Cython and CFFI to wrap existing C/C++ libraries into Python extension modules, as covered in "Extending Python Without Python's C API". You can also embed Python in applications coded in other languages, exposing application functionality to Python via app-specific Python extension modules.

This book documents many modules, from the standard library and other sources, for client- and server-side network programming, databases, processing text and binary files, and interacting with operating systems.

Python Implementations

At the time of this writing, Python has two full-production-quality implementations (CPython and PyPy) and several newer, high-performance ones in somewhat-earlier stages of development, such as Nuitka, GraalVM Python, and Pyston, which we do not cover further. In a later section, we also mention some other, even-earlier-stage implementations.

This book primarily addresses CPython, the most widely used implementation, which we often call just "Python" for simplicity. However, the distinction between a language and its implementations is important!

CPython

Classic Python (AKA CPython, often just called Python) is the most up-to-date, solid, and complete production-quality implementation of Python. It is the "reference implementation" of the language. CPython is a compiler, interpreter, and set of built-in and optional modules, all coded in standard C.

CPython can be used on any platform where the C compiler complies with the ISO/IEC 9899:1990 standard[1] (i.e., all modern, popular platforms). In "Installation", we explain how to download and install CPython. All of this book, except a few sections explicitly marked otherwise, applies to CPython. As of this writing, CPython's current release is 3.10 (work has started on 3.11, which should be ready by the end of 2022).

PyPy

PyPy is a fast and flexible implementation of Python, coded in a subset of Python itself, able to target several lower-level languages and virtual machines using advanced techniques such as type inferencing. PyPy's greatest strength is its ability to generate native machine code "just in time" as it runs your Python program. PyPy currently implements 3.7. PyPy has substantial advantages in execution speed.

Choosing Between CPython, PyPy, And Other Implementations

If your platform, as most are, is able to run all of CPython, PyPy, and several of the other Python implementations we mention, how do you choose among them? First of all, don't choose prematurely: download and install them all. They coexist without problems, and they're all free (some of them also offer commercial versions with added value such as tech support, but the respective free versions are fine, too). Having them all on your development machine costs only some download time and a little extra disk space, and lets you compare them directly.

If you need a custom version of Python, or high performance for long-running programs, consider PyPy (or, if you're OK with versions that are not quite production-ready yet, the other ones we mention).

To work mostly in a traditional environment, CPython is an excellent fit. If you don't have a strong alternative preference, start with the standard CPython reference implementation, which is most widely supported by third-party add-ons and extensions, and offers the most up-to-date version.

In other words, to experiment, learn, and try things out, use CPython. To develop and deploy, your best choice depends on the extension modules you want to use and how you want to distribute your programs. CPython, by definition, supports all Python extensions; however, PyPy supports most such extensions, and can often be even faster for long-running programs, thanks to just-in-time compilation to machine code--to check on that, benchmark your CPython code against PyPy (and, to be sure, other implementations as well).

CPython is most mature: it has been around longer, while PyPy (and the others) are newer and less proven in the field. The development of CPython versions proceeds ahead of that of other implementations.

PyPy, CPython, and other implementations we mentioned, are all good, faithful implementations of Python, reasonably close to each other in terms of usability and performance. It is wise to become familiar with the strengths and weaknesses of each, and then choose optimally for each development task.

Other Developments, Implementations, and Distributions

Python has become so popular that several groups and individuals have taken an interest in its development and provided features and implementations outside the core development team's focus.

Nowadays, most Unix-based systems include Python, most of them version 3.*x* for some value of *x*, as the "system Python." (To get Python on Windows, you usually download and run an installer). If you are serious about software development in Python, the first thing you should do is *leave your system-installed Python alone!* Quite apart from anything else, Python is increasingly used by some parts of the operating system itself, so tweaking the Python installation could lead to trouble.

Thus, even if your system comes with a "system Python", consider installing one or more Python implementations to freely use for your development convenience, safe in the knowledge that nothing you do affects the operating system. We also strongly recommend the use of *virtual*

environments (see “Python Environments”) to isolate projects from each other, letting them have what might otherwise be conflicting dependencies (e.g., if two of your projects require different versions of the same third-party module).

Python’s popularity has led to the creation of many active communities, and the language’s ecosystem is very active. The following sections outline some of the more interesting developments, but our failure to include a project here reflects limitations of space and time rather than implying any disapproval!

Jython and IronPython

Jython, supporting Python on top of a JVM, and IronPython, supporting Python on top of .NET, are open-source projects which, while offering production-level quality for the Python versions they support, appear to be “stalled” at the time of this writing, since the latest versions they support are quite substantially behind CPython’s. Any “stalled” open-source project could, potentially, come back to life again: all it takes is one or more enthusiastic, committed developers to devote themselves to “reviving” it. As an alternative to Jython for the JVM, you might also consider GraalVM Python, mentioned earlier.

Numba

Numba is an open-source just-in-time (JIT) compiler that translates a subset of Python and Numpy; given its strong focus on numeric processing, we mention it in chapter “Numeric Processing”, just after the coverage of NumPy itself.

Pyjion

Pyjion is an open-source project, originally started by Microsoft, with the key goal of adding an API to CPython to manage JIT compilers. Secondary goals include offering a JIT compiler for Microsoft’s open-source CLR environment (which is part of .NET), and a framework to develop JIT compilers. Pyjion does not **replace** CPython: rather, it is a module that you import from CPython (currently requires 3.10), which lets you translate

CPython's bytecode, "just in time", into machine code for several different environments. Integration of Pyjion with CPython is enabled by PEP 523 ||**3.6**++||; however, since building Pyjion requires several tools in addition to just a C compiler (which is all it takes to build CPython), the PSF will likely never bundle Pyjion into the CPython releases it distributes.

IPython

IPython enhances CPython's interactive interpreter to make it more powerful and convenient. IPython allows abbreviated function call syntax, and extensible functionality known as *magics* introduced by the percent (`%`) character. It also provides shell escapes, allowing a Python variable to receive the result of a shell command. You can use a question mark to query an object's documentation (two question marks for extended documentation); all the standard features of the Python interactive interpreter are also available.

IPython has made particular strides in the scientific and data-focused world, and has slowly morphed (through the development of IPython Notebook, now refactored and renamed Jupyter Notebook and shown in the Figure) into an interactive programming environment that, among snippets of code[2], also lets you embed commentary in literate programming style (including mathematical notation) and show the output of executing code, optionally with advanced graphics produced by such subsystems as `matplotlib` and `bokeh`. An example of `matplotlib` graphics is shown in the bottom half of the figure. Jupyter/IPython is one of Python's prominent success stories.

MicroPython

The continued trend in miniaturization has brought Python well within the range of the hobbyist. Single-board computers like Raspberry Pi and Beagle Board let you run Python in a full Linux environment. Below this level, there is a class of devices known as *microcontrollers*, programmable chips

with configurable hardware, which extend the scope of hobby and professional projects—for example, by making analog and digital sensing easy, enabling such applications as light and temperature measurements with little additional hardware.

Both hobbyists and professional engineers are making increasing use of these devices, which appear (and sometimes disappear) all the time. Thanks to the MicroPython project the rich functionality of many such devices (Arduino, pyboard, Lego Mindstorms EV3, …) can now be programmed in (a limited dialect of) Python. Of note at the time of writing is the introduction of the Raspberry Pi Pico. Given the success of the Raspberry Pi in the education world, and Pico's ability to run MicroPython, it seems that Python is consolidating its position as the programming language with the broadest range of applications.

MicroPython is a Python 3.4 implementation that can produce bytecode or executable machine code (though many users will be happily unaware of the latter fact). It implements Python 3.4's syntax, but lacks most of the standard library. Special hardware driver modules allow configuration and control of the various pieces of built-in configurable hardware, and access to Python's socket library lets devices interact with network services. External devices and timer events can trigger code. Thanks to MicroPython, the Python language can be a full player in the Internet of Things.

A device typically offers interpreter access through a USB serial port, or through a browser using the WebREPL protocol (we aren't aware of any `ssh` implementations yet, though, so take care to firewall these devices properly: *they should not be directly accessible across the Internet without proper, strong precautions!*). You can program the device's power-on bootstrap sequence in Python by creating a *boot.py* file in the device's memory, and this file can execute arbitrary MicroPython code of any complexity.

Anaconda and Miniconda

One of the more successful Python distributions[3] in recent years is Anaconda. This open-source package comes with an enormous number[4] of

preconfigured and tested extension modules in addition to the standard library. In many cases, you might find that it contains all the necessary dependencies for your work. In case your dependencies aren't supported, you can also install modules with `pip`. On Unix-based systems, it installs very simply in a single directory: to activate it, just add the Anaconda *bin* subdirectory at the front of your shell `PATH`.

Anaconda is based on a packaging technology called `conda`. A sister implementation, Miniconda, gives access to the same extensions but does not come with them preloaded, instead downloading them as required, making it a better choice for creating tailored environments. Conda does not use the standard virtual environments, but contains equivalent facilities to allow separation of the dependencies for multiple projects.

Pyenv: Simple Support for Multiple Versions

The basic purpose of pyenv is to make it possible to use as many different versions of Python as you need. It does so by installing so-called *shim* scripts for each executable, which dynamically compute the version required by looking at various sources of information in the following order.

1. The `PYENV_VERSION` environment variable (if set).

2. The `.pyenv_version` file in the current directory (if present)—you can set this with the `pyenv local` command..

3. The first `.pyenv_version` file found ascending the directory tree (if one is found).

4. The `version` file in the pyenv installation root directory—you can set this with the `pyenv global` command.

Pyenv installs its Python interpreters underneath its home directory (normally ~/.pyenv) , and once available a specific interpreter can be installed as the default Python in any project directory. Alternatively (e.g., when testing code under multiple versions) you can use scripting to change

the interpreter dynamically as the script proceeds, allowing easy use of multiple versions.

Thh `pyenv install -list` command shows an impressive list of over 500 supported distributions including Pypy, miniconda, micropython and several others, plus every official CPython implementation from 2.1.3 to (at the time of writing) 3.11a0.

Transcrypt: Convert your Python to JavaScript

Many attempts have been made to make Python into a browser-based language, but JavaScript's hold has been tenacious. The Transcrypt system is a pip-installable Python package to convert Python code (currently, up to version 3.9) into browser-executable JavaScript. You have full access to the browser's DOM, allowing your code to dynamically manipulate window content and use JavaScript libraries.

Although it creates minified code, Transcrypt provides full sourcemaps that allow you to debug with reference to the Python source rather than the generated JavaScript. You can write browser event handlers in Python, mixing it freely with HTML and JavaScript. Python may never replace Javascript as the embedded browser language, but Transcrypt means you might no longer need to worry about that.

Another very active project to let you script your web pages with Python (up to 3.10) is Brython, and there are others yet: Skulpt, not quite up to Python 3 yet but moving in that direction; PyPy.js, ditto; Pyodide, currently supporting Python 3.9 and many scientific extensions, and centered on Wasm.

Licensing and Price Issues

CPython is covered by the Python Software Foundation License Version 2, which is GNU Public License (GPL) compatible, but lets you use Python for any proprietary, free, or other open-source software development, similar to BSD/Apache/MIT licenses. Licenses for PyPy and other implementations are similarly liberal. Anything you download from the

main Python and PyPy sites won't cost you a penny. These licenses do not constrain what licensing and pricing conditions you can use for software you develop using the tools, libraries, and documentation they cover.

However, not everything Python-related is free from licensing costs or hassles. Many third-party Python sources, tools, and extension modules that you can freely download have liberal licenses, similar to that of Python itself. Others are covered by the GPL or Lesser GPL (LGPL), constraining the licensing conditions you can place on derived works. Some commercially developed modules and tools may require you to pay a fee, either unconditionally, or if you use them for profit[5].

There is no substitute for careful examination of licensing conditions and prices. Before you invest time and energy into any software tool or component, check that you can live with its license. Often, especially in a corporate environment, such legal matters may involve consulting lawyers. Modules and tools covered in this book, unless we explicitly say otherwise, can be taken to be, at the time of this writing, freely downloadable, open-source, and covered by a liberal license akin to Python's. However, we claim no legal expertise, and licenses can change over time, so doublechecking is always prudent.

Python Development and Versions

Python is developed, maintained, and released by a team of core developers led by Guido van Rossum, Python's inventor, architect, and now "ex" Benevolent Dictator For Life (BDFL). This title meant that Guido had the final say on what became part of the Python language and standard library. Once Guido decided to retire as BDFL, his decision-making role was taken over by a small "Steering Council", elected for yearly terms by PSF members.

Python's intellectual property is vested in the Python Software Foundation (PSF), a nonprofit corporation devoted to promoting Python, described in "Python Software Foundation". Many PSF Fellows and members have commit privileges to Python's reference source repositories (currently git),

as documented in the Python Developer's Guide, and most Python committers are members or Fellows of the PSF.

Proposed changes to Python are detailed in public docs called Python Enhancement Proposals (PEPs), debated (sometimes an advisory vote is taken) by Python developers and the wider Python community, and finally approved or rejected by the Steering Council, who take debates and votes into account but are not bound by them. Hundreds of people contribute to Python development through PEPs, discussion, bug reports, and patches to Python sources, libraries, and docs.

The Python core team releases minor versions of Python (*3.x* for growing values of *x*), also known as "feature releases," currently at a pace of once a year.

Each minor release (as opposed to bug-fix point releases) adds features that make Python more powerful and simpler to use, but also takes care to maintain backward compatibility. Python 3.0, which was allowed to break backward compatibility in order to remove redundant "legacy" features and simplify the language, was first released in December 2008. Python 3.10 (the most recent stable version at the time of writing) was first released in October 2021; 3.11 is scheduled for final release in October 2022.

Each minor release 3.*x* is first made available in alpha releases, tagged as 3.*x* a0, 3.*x* a1, and so on. After the alphas comes at least one beta release, 3.*x* b1, and after the betas, at least one release candidate, 3.*x* rc1. By the time the final release of 3.*x* (3.*x*.0) comes out, it is solid, reliable, and tested on all major platforms. Any Python programmer can help ensure this by downloading alphas, betas, and release candidates, trying them out, and filing bug reports for any problems that emerge.

Once a minor release is out, part of the attention of the core team switches to the next minor release. However, a minor release normally gets successive point releases (i.e., 3.*x*.1, 3.*x*.2, and so on), one every two months, that add no functionality, but can fix errors, address security issues, port Python to new platforms, enhance documentation, and add tools and (100%-backwards-compatible!) optimizations.

Python's backward compatibility is quite good within major releases. You can find code and documentation online for all old releases of Python.

Python Resources

The richest Python resource is the web. The best starting point is Python's homepage, full of links to explore. PyPy is documented at the PyPy site.

Documentation

Both CPython and PyPy come with good documentation. The manuals are available in many formats, suitable for viewing, searching, and printing. You can read CPython's manuals online (we often refer to these as "the online docs"). You can also find various downloadable formats. The Python documentation page has links to a large variety of documents. The IronPython Documentation page has links to IronPython-specific documents as well as general Python ones; there is also a documentation page for PyPy. You can also find online Frequently Asked Questions (FAQ) documents for Python and PyPy.

Python documentation for nonprogrammers

Most Python documentation (including this book) assumes some software-development knowledge. However, Python is quite suitable for first-time programmers, so there are exceptions to this rule. Good introductory, free online texts for nonprogrammers include:

- Josh Cogliati's "Non-Programmers Tutorial For Python 3" (currently centered on Python 3.9)
- Alan Gauld's "Learning to Program" (currently centered on Python 3.6)
- Allen Downey's "Think Python, 2nd edition" (centered on an unspecified version of Python 3.*x*)

An excellent resource for learning Python (for nonprogrammers, and for less experienced programmers too) is the Beginners' Guide wiki, which includes a wealth of links and advice. It's community-curated, so it will stay up-to-date as available books, courses, tools, and so on keep evolving and improving.

Extension modules and Python sources

A good starting point to explore Python extension binaries and sources is the Python Package Index (still fondly known to a few of us old-timers as "The Cheese Shop," but generally referred to now as PyPI), which at the time of this writing offers more than 360,000 packages, each with descriptions and pointers.

The standard Python source distribution contains excellent Python source code in the standard library and in the Demos and Tools directories, as well as C source for the many built-in extension modules. Even if you have no interest in building Python from source, we suggest you download and unpack the Python source distribution (e.g., the latest stable release of Python 3.10) for the sole purpose of studying it; or, if you so choose, peruse the current bleeding-edge version of Python's standard library online.

Many Python modules and tools covered in this book also have dedicated sites. We include references to such sites in the appropriate chapters in this book.

Books

Although the web is a rich source of information, books still have their place (if you and we didn't agree on this, we wouldn't have written this book, and you wouldn't be reading it). Books about Python are numerous. Here are a few we recommend (some cover older Python 3 versions, rather than current ones):

- If you know some programming, but are just starting to learn Python, and like graphical approaches to instruction, Head First Python, 2nd edition, by Paul Barry (O'Reilly), may serve you well. Like all Head First series books, it uses graphics and humor to teach its subject.

- Dive Into Python 3, by Mark Pilgrim (APress), teaches by example, in a fast-paced and thorough way that is quite suitable for people who are already expert programmers in other languages.

- Beginning Python: From Novice to Professional, by Magnus Lie Hetland (APress), teaches both via thorough explanations and by fully developing complete programs in various application areas.

Community

One of the greatest strengths of Python is its robust, friendly, welcoming community. Python programmers and contributors meet at conferences, "hackathons" (often known as *sprints* in the Python community), and local user groups; actively discuss shared interests; and help each other on mailing lists and social media. For a complete list of ways to connect, visit https://www.python.org/community/.

Python Software Foundation

Besides holding the intellectual property rights for the Python programming language, the Python Software Foundation (PSF) promotes the Python community. The PSF sponsors user groups, conferences, and sprints, and provides grants for development, outreach, and education, among other activities. The PSF has dozens of Fellows (nominated for their contributions to Python, including all of the Python core team, as well as three of the authors of this book); hundreds of members who contribute time, work, and money (including many who've earned Community Service Awards); and dozens of corporate sponsors. Anyone who uses and supports Python can become a member of the PSF. Check out the membership page for information on the various membership levels, and on how to become a member of the PSF. If you're interested in contributing to Python itself, see the Python Developer's Guide.

Workgroups

Workgroups are committees established by the PSF to do specific, important projects for Python. Among the several active workgroups, as of

this writing, are:

- the Python Packaging Authority improves and maintains the Python packaging ecosystem and publishes the Python Packaging Users Guide.
- the Python Education workgroup promotes education and learning with Python.
- the Diversity and Inclusion workgroup supports and facilitates the growth of a diverse and international community of Python programmers.

Python conferences

There are lots of Python conferences worldwide. General Python conferences include international and regional ones, such as PyCon and EuroPython, and other more local ones such as PyOhio and Pycon Italia. Topical conferences include SciPy and PyData. Conferences are often followed by coding sprints, where Python contributors get together for several days of coding focused on particular open-source projects and on abundant camaraderie. You can find a listing of conferences on the Community Workshops page. More than 17,000 videos of talks about Python, from more than 400 conferences, are available at the pyvideo site.

User groups and organizations

The Python community has local user groups in every continent except Antarctica,[6] more than 1500 of them, as listed at the Local User Groups wiki. There are thousands of Python Meetups in over 70 countries. PyLadies is an international mentorship group, with local chapters, to promote women in Python; anyone with an interest in Python is welcome. NumFOCUS, a nonprofit charity promoting open practices in research, data, and scientific computing, sponsors the PyData conference and other projects.

Mailing lists

The Community mailing lists page has links to Python-related mailing lists (and some Usenet groups, for those of us old enough to remember Usenet!). Alternatively, search Mailman to find active mailing lists covering a wide variety of interests. Python-related official announcements are posted to python announce To ask for individual help with specific problems, write to *help@python.org*. For help learning or teaching Python, write to *tutor@python.org* or, better yet, join the list. For a useful weekly round-up of Python-related news and articles, subscribe to *Python Weekly*.

Social media

For an RSS feed of Python-related blogs, see planetpython. If you're interested in tracking language developments, check out discuss.python.org —it sends useful summaries if you don't visit regularly. On Twitter, follow @ThePSF. Libera.Chat on IRC hosts several Python-related channels: the main one is `#python`. LinkedIn has many Python groups, including Python Web Developers. On Slack, join pyslackers.com. On Discord, check out pythondiscord.com. Technical questions and answers about Python programming can also be found and followed on stackoverflow.com under a variety of tags, including [python]. Python is currently the most active programming language on Stack Overflow (according to reports from StackOverflow), and many useful answers with illuminating discussions can be found there.

Installation

You can install Python, in classic (CPython) and PyPy versions, on most platforms. With a suitable development system (C, for CPython; PyPy, coded in Python itself, only needs CPython installed first), you can install Python versions from the respective source code distributions. On popular platforms, you also have the recommended alternative of installing prebuilt binary distributions.

Installing Python If It Comes Preinstalled

If your platform comes with a preinstalled version of Python, you're still best advised to install another, separate, updated version, for your own code development. When you do, do not remove or overwrite your platform's original version: rather, install the other version "side by side" with the first one. This way, you won't disturb any other software that is part of your platform: such software might rely on the specific Python version that came with the platform itself.

Installing CPython from a binary distribution is faster, saves you substantial work on some platforms, and is the only possibility if you have no suitable C compiler. Installing from source code gives you more control and flexibility, and is a must if you can't find a suitable pre-built binary distribution for your platform. Even if you install from binaries, it's best to also download the source distribution "on the side", since it includes examples, demos, and tools that may be missing from prebuilt binaries.

Installing Python from Binaries

If your platform is popular and current, you'll easily find prebuilt, packaged binary versions of Python ready for installation. Binary packages are typically self-installing, either directly as executable programs, or via appropriate system tools, such as the RedHat Package Manager (RPM) on some versions of Linux and the Microsoft Installer (MSI) on Windows. After downloading a package, install it by running the program and choosing installation parameters, such as the directory where Python is to be installed. In Windows, select the option labeled "Add Python 3.10 to PATH" to have the installer add the install location into the PATH in order to easily use Python at a command prompt (see "The python Program").

To download "official" Python binaries, visit the Python site, click Downloads, and, on the next page, click the button labeled Python 3.10.x to download the most-recent binary suitable for the browser's platform.

Many third parties supply free binary Python installers for other platforms. Installers exist for Linux distributions, whether your distribution is RPM-based (RedHat, Fedora, Mandriva, SUSE, etc.), or Debian-based, including Ubuntu, probably the most popular Linux distribution at the time of this writing. The Other Downloads page provides links to binary distributions for now-somewhat-exotic platforms such as AIX, OS/2, RISC OS, IBM AS/400, Solaris, HP-UX, and so forth (often not the latest Python versions, given the "now-quaint" nature of such platforms) as well as one for the very-current platform iOS, the operating system of the popular iPhone and iPad devices.

Anaconda, previously mentioned in this chapter, is a binary distribution including Python, plus the conda package manager, plus hundreds of third-party extensions, particularely for science, math, engineering, and data analysis. It's available for Linux, Windows, and macOS. The same package, but without all of those extensions (you can selectively install subsets of them with `conda`), is also available and is known as Miniconda, as also mentioned earlier in this chapter.

macOS

Apple's macOS still comes with a command-line program called `python`, which, however, is (at the time of this writing: this is due to be removed in the next version of macOS) a legacy Python 2.7 version, included only for backward compatibility, and due to soon be removed. There's also a `python3` program (currently 3.8); nevertheless, we recommend you install the latest version and enhancements by following the instructions and links on the Python site under Downloads; due to Apple's release cycles, the Python version included with your version of macOS may be out of date or incomplete. Python's latest version installs in addition to, not instead of, Apple's supplied one; Apple uses its own version of Python and proprietary extensions to implement some of the software it distributes as a part of macOS, so it's unwise to risk disturbing that version in any way.

The popular third-party macOS open-source package manager Homebrew offers, among many other open-source packages, excellent versions of

Python.

Installing Python from Source Code

To install CPython from source code, you need a platform with an ISO-compliant C compiler and tools such as `make`. On Windows, the normal way to build Python is with Visual Studio (ideally VS 2022, currently available to developers for free).

To download Python source code, visit the Python site, click Downloads, then pick the version. For more options, hover on Downloads, and use the menu that shows.

The *.tgz* file extension under the link labeled "Gzipped source tarball" is equivalent to *.tar.gz* (i.e., a *tar* archive of files, compressed by the popular `gzip` compressor). Alternatively, use the link labeled "XZ compressed source tarball" to get a version with an extension of *.tar.xz* instead of *.tgz*, compressed with the even more powerful `xz` compressor, if you have all the needed tools to deal with XZ compression.

Microsoft Windows

On Windows, installing Python from source code can be a chore unless you are familiar with Visual Studio and used to working in the text-oriented window known as the "command prompt"—most Windows users prefer to simply download the pre-built Python from the Microsoft store.

If the following instructions give you any trouble, stick with "Installing Python from Binaries". It's best to do a separate installation from binaries anyway, even if you also install from source. If you notice anything strange while using the version you installed from source, double-check with the installation from binaries. If the strangeness goes away, it must be due to some quirk in your installation from source, so you know you must double-check the details of how you chose to build the latter.

In the following sections, for clarity, we assume you have made a new folder called *%USERPROFILE%\py*, (e.g., *c:\users\tim\py*). Open this

folder by typing, for example, **`%USERPROFILE%`** in the address bar of a Windows Explorer window and going from there. Download the source *tgz* file—for example, *Python-3.10.1.tgz* (or other Python version of your choice) to that folder. Of course, name and place the folder as it best suits you: our name choice is just for expository purposes.

Uncompressing and unpacking the Python source code

You can uncompress and unpack a *.tgz* or *.tar.xz* file, for example, with the free program 7-Zip. Download and install 7-Zip from the download page, and run it on the sources' *tgz* file (e.g., *c:\users\alex\py\Python-3.10.1.tgz*) you downloaded. (The *tgz* file may have downloaded to *%USERPROFILE%\downloads*, in which case you need to move it to your *\py* folder (which you may also need to create, for example by typing the `mkdir` command in any command window) before unpacking it with 7-Zip.) You now have a folder *%USERPROFILE%\py\Python-3.10.1* (or other version originally downloaded), the root of a tree that contains the entire standard Python distribution in source form.

Building the Python source code

Open the text file *%USERPROFILE%\py\Python-3.10.1\PCBuild\readme.txt* (or whichever version of Python you downloaded) with any text editor, and follow the detailed instructions found there.

Unix-Like Platforms

On Unix-like platforms, installing Python from source code is generally simple[7]. In the following sections, for clarity, we assume you have created a new directory named *~/Py* and downloaded the sources' *tgz* file—for example, *Python-3.10.1.tgz* (or other Python version of your choice)—to that directory. Of course, name and place the directory as it best suits you: our name choice is just for expository purposes.

Uncompressing and unpacking the Python source code

You can uncompress and unpack a *.tgz* or *.tar.xz* file with the popular GNU version of *tar*. Just type the following at a shell prompt:

```
$ cd ~/Py & tar xzf Python-3.10.1.tgz
```

You now have a directory *~/Py/Python-3.10.1*, the root of a tree that contains the entire standard Python distribution in source form.

Configuring, building, and testing

Detailed notes are in *~/Py/Python-3.10.1/README* under the heading "Build instructions," and we recommend you study those notes. In the simplest case, however, all you need may be to give the following commands at a shell prompt:

```
$ cd ~/Py/Python-3.10.1
$ ./configure
    [configure writes much information - snipped here]
$ make
    [make takes quite a while, and emits much information]
```

If you run **`make`** without first running **`./configure`**, **`make`** implicitly runs **`./configure`**. When **`make`** finishes, check that the Python you have just built works as expected:

```
$ make test
    [takes quite a while, emits much information]
```

Usually, **`make test`** confirms that your build is working, but also informs you that some tests have been skipped because optional modules were missing.

Some of the modules are platform-specific (e.g., some may work only on machines running SGI's ancient Irix operating system); if so, don't worry about them. However, other modules are skipped because they depend on other open source packages that are not installed on your machine. For example, on Unix, module `_tkinter`—needed to run the Tkinter GUI package and the IDLE integrated development environment, which come

with Python—can be built only if **`./configure`** can find an installation of Tcl/Tk 8.0 or later on your machine. See *~/Py/Python-3.10.1/README* for more details and specific caveats about different Unix and Unix-like platforms.

Building from source code lets you tweak your configuration in several ways. For example, you can build Python in a special way that helps you debug memory leaks when you develop C-coded Python extensions, covered in "Building and Installing C-Coded Python Extensions". **`./configure --help`** is a good source of information about the configuration options you can use.

Installing after the build

By default, **`./configure`** prepares Python for installation in */usr/local/bin* and */usr/local/lib*. You can change these settings by running **`./configure`** with option **`--prefix`** before running **`make`**. For example, if you want a private installation of Python in subdirectory *py310* of your home directory, run:

```
$ cd ~/Py/Python-3.10.1
$ ./configure --prefix=~/py310
```

and continue with **`make`** as in the previous section. Once you're done building and testing Python, to perform the actual installation of all files, run:

```
$ make install
```

The user running **`make install`**[8] must have write permissions on the target directories. Depending on your choice of target directories, and permissions on those directories, you may need to **`su`** to *root, bin*, or some other user when you run **`make install`**. The common idiom for this purpose is **`sudo make install`**: if **`sudo`** prompts for a password, enter your current user's password, not root's. An alternative, and recommended,

approach is to install into a virtual environment, as covered in “Python Environments”.

Installing PyPy

To install PyPy from sources, download your chosen version of PyPy’s sources, unpack them, then follow the instructions included in your download. Alternatively (and, usually, preferably, for simplicity!), download a binary installer for your platform (if supported), uncompress it from its *.zip* or *.tar.bz2* format, and then run the resulting executable file.

---

1 It seems likely at the time of writing that 3.11 and later will use “C11 without optional features,” and specify that “the public API should be compatible with C++.”

2 which can be in many programming languages, not **just** Python

3 In fact conda’s capabilities extend to other languages, and Python is simply another dependency.

4 250+ automatically installed with Anaconda, 7500+ explicitly installable with conda install

5 a popular business model is *freemium*: releasing both a free version, and a commercial “premium” one with tech support and, perhaps, extra features.

6 we need to mobilize to get more penguins interested in our language!

7 Most problems with source installations concern the absence of various supporting libraries, whose absence may cause some features to be missing from the built interpreter. The Python Developers’ Guide explains how to handle dependencies on various platforms.

8 or make altinstall, if you want to avoid creating links to the Python executable and manual pages.

Chapter 2. 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").

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 3. 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 3-1. 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 3-2.
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 y th 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 "Special Methods".

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 "Searching and sorting", 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 3-3.
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

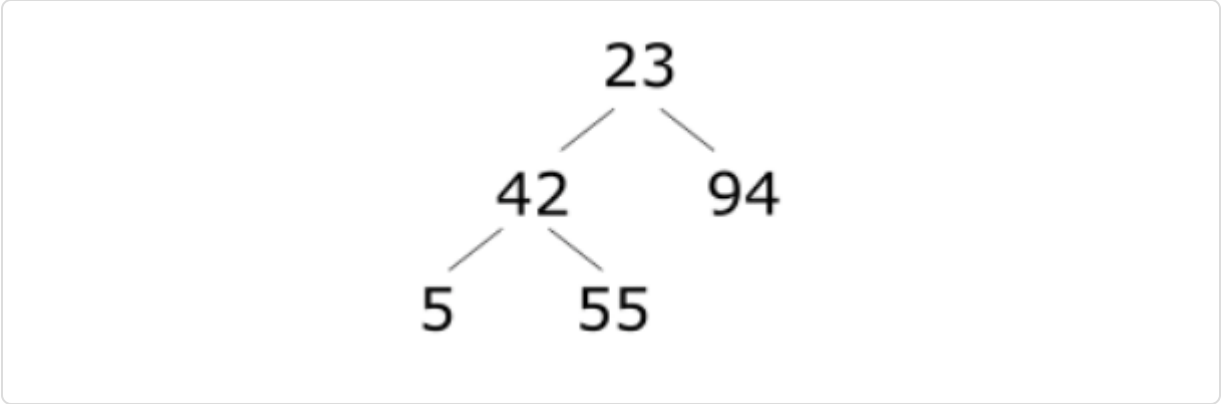
Python supports recursion (i.e., a Python function can call itself, directly or indirectly), but there is a limit to how deep the recursion can go. By default, Python interrupts recursion and raises a `RecursionLimitExceeded` exception (covered in "Standard Exception Classes") when it detects that recursion 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 4. Object-Oriented Python

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 4th 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 is an object-oriented (OO) programming language. Unlike some other object-oriented languages, Python doesn't force you to use the object-oriented paradigm exclusively: it also supports procedural programming with modules and functions, so that you can select the best paradigm for each part of your program. The object-oriented paradigm helps you group state (data) and behavior (code) together in handy packets of functionality. Moreover, it offers some useful specialized mechanisms covered in this chapter, like *inheritance* and *special methods*. The simpler procedural approach, based on modules and functions, may be more suitable when you don't need the pluses of object-oriented programming. With Python, you can mix and match paradigms.

This chapter also covers *special methods*, in "Special Methods," and advanced concepts known as *abstract base classes*, in "Abstract Base Classes"; *decorators*, in "Decorators"; and *metaclasses*, in "Metaclasses".

Classes and Instances

If you're familiar with object-oriented programming in other OO languages such as C++ or Java, you probably have a good grasp of classes and instances: a *class* is a user-defined type, which you *instantiate* to build *instances*, i.e., objects of that type. Python supports this through its class and instance objects.

Python Classes

A *class* is a Python object with several characteristics:

- You can call a class object like you'd call a function. The call, known as *instantiation*, returns an object known as an *instance* of the class; the class is called the instance's *type*.
- A class has arbitrarily named attributes that you can bind and reference.
- The values of class attributes can be *descriptors* (including functions), covered in "Descriptors," or ordinary data objects.
- Class attributes bound to functions are also known as *methods* of the class.
- A method can have any one of many Python-defined names with two leading and two trailing underscores (known as *dunder names*, short for "double-underscore names"—the name __init__, for example, is pronounced "dunder init"). Python implicitly calls such *special methods*, if a class supplies them, when various kinds of operations occur on instances of that class.
- A class can *inherit* from one or more classes, meaning it delegates to other class objects the lookup of attributes (including regular and dunder methods) that are not in the class itself.

An instance of a class is a Python object with arbitrarily named attributes that you can bind and reference. Every instance object delegates attribute lookup of attributes to its class for any attribute not found in the instance

itself. The class, in turn, may delegate the lookup to classes from which it inherits, if any.

In Python, classes are objects (values) handled just like other objects. You can pass a class as an argument in a call to a function; a function can return a class as the result of a call. You can bind a class to a variable, an item in a container, an attribute of an object. Classes can be keys into a dictionary. Since classes are ordinary objects in Python, we often say that classes are *first-class* objects.

The class Statement

The `class` statement is the usual way you create a class object. `class` is a single-clause compound statement with the following syntax:

```
class classname(base-classes):
    statement(s)
```

`classname` is an identifier: a variable that the class statement, when finished, binds (or rebinds) to the just-created class object.

`base-classes` is a comma-delimited series of expressions whose values are class objects. Various programming languages use different names for these classes: you can call them the *bases, superclasses*, or *parents* of the class. You can say the class created *inherits* from, *derives* from, *extends*, or *subclasses* its base classes; in this book, we generally use *extend*. This class is a *direct subclass* or *descendant* of its base classes. *`base-classes`* can include a named argument `metaclass=` to establish the class's *metaclass*[1], as covered in "How Python Determines a Class's Metaclass".

Syntactically, *`base-classes`* are optional: to indicate that you're creating a class without bases, just omit *`base-classes`* (and, optionally, also omit the parentheses around it)—place the colon right after the classname. Every class inherits from `object`, whether you specify explicit bases or not.

The subclass relationship between classes is transitive: if *C1* extends *C2*, and *C2* extends *C3*, then *C1* extends *C3*. Built-in function `issubclass(`*C1*`, `*C2*`)` accepts two class objects: it returns `True` when *C1* extends *C2*; otherwise, it returns `False`. Any class is a subclass of itself; therefore, `issubclass(`*C*`, `*C*`)` returns `True` for any class *C*. We cover how base classes affect a class's functionality in "Inheritance."

The nonempty sequence of indented statements that follows the `class` statement is the *class body*. A class body executes immediately as part of the `class` statement's execution. Until the body finishes executing, the new class object does not yet exist, and the *classname* identifier is not yet bound (or rebound). "How a Metaclass Creates a Class" provides more details about what happens when a `class` statement executes.

Note that the `class` statement does not immediately create any instance of the new class but rather defines the set of attributes shared by all instances when you later create instances by calling the class.

The Class Body

The body of a class is where you normally specify class attributes; these attributes can be descriptor objects (including functions) or ordinary data objects of any type. An attribute of a class can be another class—so, for example, you can have a `class` statement "nested" inside another `class` statement.

Attributes of class objects

You usually specify an attribute of a class object by binding a value to an identifier within the class body. For example:

```
class C1:
    x = 23
print(C1.x)                             # prints: 23
```

The class object *C1* has an attribute named *x*, bound to the value `23`, and *C1*`.`*x* refers to that attribute.

You can also bind or unbind class attributes outside the class body. For example:

```
class C2:
    pass
C2.x = 23
print(C2.x)                             # prints: 23
```

Your program is usually more readable if you bind class attributes only with statements inside the class body. However, rebinding them elsewhere may be necessary if you want to carry state information at a class, rather than instance, level; Python lets you do that, if you wish. There is no difference between a class attribute bound in the class body and one bound or rebound outside the body by assigning to an attribute.

As we'll discuss shortly, all class instances share all of the class's attributes.

The `class` statement implicitly sets some class attributes. Attribute `__name__` is the *`classname`* identifier string used in the `class` statement. Attribute `__bases__` is the tuple of class objects given (or implied) as the base classes in the `class` statement. For example, using the class *`C1`* we just created:

```
print(C1.__name__, C1.__bases__)  # prints: C1 (<class
'object'>,)
```

A class also has an attribute `__dict__`, the read-only mapping that the class uses to hold other attributes (AKA, informally, the class's *namespace*).

In statements directly in a class's body, references to attributes of the class must use a simple name, not a fully qualified name. For example:

```
class C3:
    x = 23
    y = x + 22                           # must use just x, not
C3.x
```

However, in statements within *methods* defined in a class body, references to class attributes must use a fully qualified name, not a simple name. For

example:

```
class C4:
    x = 23
    def amethod(self):
        print(C4.x)  # must use C4.x or self.x, not just x!
```

Attribute references (i.e., expressions like *`C.s`*) have semantics richer than attribute bindings. We cover such references in detail in "Attribute Reference Basics."

Function definitions in a class body

Most class bodies include `def` statements, since functions (known as *methods* in this context) are important attributes for most class objects. A `def` statement in a class body obeys the rules covered in "Functions." In addition, a method defined in a class body has a mandatory first parameter, conventionally always named `self`, that refers to the instance on which you call the method. The `self` parameter plays a special role in method calls, as covered in "Bound and Unbound Methods."

Here's an example of a class that includes a method definition:

```
class C5:
    def hello(self):
        print('Hello')
```

A class can define a variety of special methods (methods with names that have two leading and two trailing underscores)[2] relating to specific operations on its instances. We discuss special methods in detail in "Special Methods."

Class-private variables

When a statement in a class body (or in a method in the body) uses an identifier starting with two underscores (but not *ending* with underscores), such as `__`*`ident`*, Python implicitly changes the identifier to *`_classname__ident`*, where *`classname`* is the name of the class. This implicit change lets a class use "private" names for attributes,

methods, global variables, and other purposes, reducing the risk of accidentally duplicating names used elsewhere, particularly in subclasses.

By convention, identifiers starting with a *single* underscore are private to the scope that binds them, whether that scope is or isn't a class. The Python compiler does not enforce this privacy convention: it's up to programmers to respect it.

Class documentation strings

If the first statement in the class body is a string literal, the compiler binds that string as the *documentation string* for the class. This attribute, named `__doc__`, is the *docstring* of the class (`None` if the first statement in the body is *not* a string literal). See "Docstrings" for more information on docstrings.

Descriptors

A *descriptor* is an object whose class supplies a special method named `__get__`. Descriptors that are class attributes control the semantics of accessing and setting attributes on instances of that class. Roughly speaking: when you access an instance attribute, Python gets the attribute's value by calling `__get__` on the corresponding descriptor, if any. For example:

```
class Const:                # an overriding descriptor, see later
    def __init__(self, value):
        self.value = value
    def __set__(self, *_):  # silently ignore any attempt at
setting
        pass
    def __get__(self, *_):  # always return the constant value
        return self.value
class X:
    c = Const(23)
x=X()
print(x.c)  # prints: 23
x.c = 42    # silently ignored
print(x.c)  # prints: 23
```

For more details, see “Attribute Reference Basics”.

Overriding and non-overriding descriptors

When a descriptor’s class supplies a special method named `__set__`, the descriptor is known as an *overriding descriptor* (or, by an older, confusing terminology, a *data descriptor*); when the descriptor’s class supplies `__get__` and not `__set__`, the descriptor is known as a *non-overriding descriptor*.

For example, the class of function objects supplies `__get__`, but not `__set__`; therefore, function objects are non-overriding descriptors. Roughly speaking: when you assign a value to an instance attribute with a corresponding descriptor that is overriding, Python sets the attribute value by calling `__set__` on the descriptor. For more details, see “Attributes of instance objects.”

Instances

To create an instance of a class, call the class object as if it were a function. Each call returns a new instance whose type is that class:

```
an_instance = C5()
```

Built-in function `isinstance(`*`i, C`*`)`, with a class as argument *`C`*, returns `True` when *`i`* is an instance of class *`C`* or any subclass of *`C`*. Otherwise, `isinstance` returns `False`.

__init__

When a class defines or inherits a method named `__init__`, calling the class object executes `__init__` on the new instance to perform per-instance initialization. Arguments passed in the call must correspond to `__init__`‘s parameters, except for parameter `self`. For example, consider:

```
class C6:
    def __init__(self, n):
```

```
        self.x = n
```

Here's how you can create an instance of the *C6* class:

```
another_instance = C6(42)
```

As shown in the *C6* class, the `__init__` method typically contains statements that bind instance attributes. An `__init__` method must not return a value other than `None`; if it does, Python raises a `TypeError` exception.

The main purpose of `__init__` is to bind, and thus create, the attributes of a newly created instance. You may also bind, rebind, or unbind instance attributes outside `__init__`. However, your code is more readable when you initially bind all class instance attributes in the `__init__` method.

When `__init__` is absent (and not inherited from any base), you must call the class without arguments, and the new instance has no instance-specific attributes.

Attributes of instance objects

Once you have created an instance, you can access its attributes (data and methods) using the dot (`.`) operator. For example:

```
an_instance.hello()                         # prints: Hello
print(another_instance.x)                   # prints: 42
```

Attribute references such as these have fairly rich semantics in Python; we cover them in detail in "Attribute Reference Basics."

You can give an instance object an attribute by binding a value to an attribute reference. For example:

```
class C7:
    pass
z = C7()
z.x = 23
print(z.x)                                  # prints: 23
```

Instance object *z* now has an attribute named *x*, bound to the value 23, and *z.x* refers to that attribute. The __setattr__ special method, if present, intercepts every attempt to bind an attribute. (We cover __setattr__ in Table 4-1.)

When you attempt to bind on an instance an attribute whose name corresponds to an overriding descriptor in the class, the descriptor's __set__ method intercepts the attempt: if *C7.x* were an overriding descriptor, *z.x* = 23 would execute type(*z*).*x*.__set__(*z*, 23).

Creating an instance sets two instance attributes. For any instance *z*, *z*.__class__ is the class object to which *z* belongs, and *z*.__dict__ is the mapping *z* uses to hold its other attributes. For example, for the instance *z* we just created:

```
print(z.__class__.__name__, z.__dict__)    # prints: C7 {'x':23}
```

You may rebind (but not unbind) either or both of these attributes, but this is rarely necessary.

For any instance *z*, any object *x*, and any identifier *S* (except __class__ and __dict__), *z.S*=*x* is equivalent to *z*.__dict__['S']=*x* (unless a __setattr__ special method, or an overriding descriptor's __set__ special method, intercepts the binding attempt). For example, again referring to the *z* we just created:

```
z.y = 45
z.__dict__['z'] = 67
print(z.x, z.y, z.z)                       # prints: 23 45 67
```

There is no difference between instance attributes created by assigning to attributes and those created by explicitly binding an entry in *z*.__dict__.

The factory-function idiom

It's often necessary to create instances of different classes depending on some condition, or avoid creating a new instance if an existing one is available for reuse. A common misconception is that such needs might be

met by having `__init__` return a particular object. However, this approach is unfeasible: Python raises an exception if `__init__` returns any value other than `None`. The best way to implement flexible object creation is to use a function rather than calling the class object directly. A function used this way is known as a *factory function*.

Calling a factory function is a flexible approach: a function may return an existing reusable instance or create a new instance by calling whatever class is appropriate. Say you have two almost interchangeable classes (*`SpecialCase`* and *`NormalCase`*) and want to flexibly generate instances of either one of them, depending on an argument. The following *`appropriate_case`* factory function, as a "toy" example, allows you to do just that (we cover the role of the `self` parameter in "Bound and Unbound Methods"):

```
class SpecialCase:
    def amethod(self):
        print('special')
class NormalCase:
    def amethod(self):
        print('normal')
def appropriate_case(isnormal=True):
    if isnormal:
        return NormalCase()
    else:
        return SpecialCase()
aninstance = appropriate_case(isnormal=False)
aninstance.amethod()                  # prints: special
```

`__new__`

Every class has (or inherits) a class method named `__new__` (we cover class methods in "Class methods"). When you call *`C(*args,**kwds)`* to create a new instance of class *`C`*, Python first calls *`C.__new__(C,*args,**kwds)`*. Python uses `__new__`'s return value *`x`* as the newly created instance. Then, Python calls *`C.__init__(x,*args,**kwds)`*, but only when *`x`* is indeed an instance of *`C`* or any of its subclasses (otherwise, *`x`*'s state remains as

__new__ had left it). Thus, for example, the statement *x*=*C*(23) is equivalent to:

```
x = C.__new__(C, 23)
if isinstance(x, C):
    type(x).__init__(x, 23)
```

object.__new__ creates a new, uninitialized instance of the class it receives as its first argument. It ignores other arguments when that class has an __init__ method, but it raises an exception when it receives other arguments beyond the first, and the class that's the first argument does not have an __init__ method. When you override __new__ within a class body, you do not need to add __new__=classmethod(__new__), nor use an @classmethod decorator, as you normally would: Python recognizes the name __new__ and treats it as special in this context. In those sporadic cases in which you rebind *C*.__new__ later, outside the body of class *C*, you do need to use *C*.__new__=classmethod(*whatever*).

__new__ has most of the flexibility of a factory function, as covered in "The factory-function idiom." __new__ may choose to return an existing instance or make a new one, as appropriate. When __new__ does create a new instance, it usually delegates creation to object.__new__ or the __new__ method of another superclass of *C*. The following example shows how to override class method __new__ in order to implement a version of the Singleton design pattern:

```
class Singleton:
    _singletons = {}
    def __new__(cls, *args, **kwds):
        if cls not in cls._singletons:
            cls._singletons[cls] = super().__new__(cls)
        return cls._singletons[cls]
```

(We cover built-in super in "Cooperative superclass method calling".)

Any subclass of *Singleton* (that does not further override __new__) has exactly one instance. When the subclass defines __init__, it must

ensure `__init__` is safe to call repeatedly (at each call of the subclass) on the subclass's only instance[3].

Attribute Reference Basics

An *attribute reference* is an expression of the form *`x.name`*, where *`x`* is any expression and *`name`* is an identifier called the *attribute name*. Many Python objects have attributes, but an attribute reference has special, rich semantics when *`x`* refers to a class or instance. Remember that methods are attributes, too, so everything we say about attributes in general also applies to callable attributes (i.e., methods).

Say that *`x`* is an instance of class *`C`*, which inherits from base class *`B`*. Both classes and the instance have several attributes (data and methods), as follows:

```
class B:
    a = 23
    b = 45
    def f(self):
        print('method f in class B')
    def g(self):
        print('method g in class B')
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self):
        print('method g in class C')
    def h(self):
        print('method h in class C')
x = C()
x.d = 77
x.e = 88
```

A few attribute dunder-names are special. *`C.`*`__name__` is the string `'C'`, the class's name. *`C.`*`__bases__` is the tuple `(B,)`, the tuple of *`C`*'s base classes. *`x.`*`__class__` is class *`C`*, the class to which *`x`* belongs. When you refer to an attribute with one of these special names, the attribute reference looks directly into a dedicated slot in the class or instance object and fetches the value it finds there. You cannot unbind these attributes. You may rebind

them on the fly, changing the name or base classes of a class or the class of an instance, but this advanced technique is rarely necessary.

Class `C` and instance `x` each have one other special attribute: a mapping named `__dict__` (typically mutable for `x`, but not for `C`). All other attributes of a class or instance, except the few special ones, are held as items in the `__dict__` attribute of the class or instance.

Getting an attribute from a class

When you use syntax `C.name` to refer to an attribute on a class object `C`, lookup proceeds in 2 steps:

1. When `'name'` is a key in `C.__dict__`, `C.name` fetches the value `v` from `C.__dict__['name']`. Then, when `v` is a descriptor (i.e., `type(v)` supplies a method named `__get__`), the value of `C.name` is the result of calling `type(v).__get__(v, None, C)`. When `v` is not a descriptor, the value of `C.name` is `v`.

2. When `'name'` is *not* a key in `C.__dict__`, `C.name` delegates the lookup to `C`'s base classes, meaning it loops on `C`'s ancestor classes and tries the `name` lookup on each (in *method resolution order*, as covered in "Method resolution order").

Getting an attribute from an instance

When you use syntax `x.name` to refer to an attribute of instance `x` of class `C`, lookup proceeds in 3 steps:

1. When `'name'` is in `C` (or in one of `C`'s ancestor classes) as the name of an overriding descriptor `v` (i.e., `type(v)` supplies methods `__get__` and `__set__`)

   The value of `x.name` is the result of `type(v).__get__(v, x, C)`

2. Otherwise, when `'name'` is a key in `x.__dict__`

   `x.name` fetches and returns the value at `x.__dict__['name']`

3. Otherwise, *x.name* delegates the lookup to *x*'s class (according to the same two-step lookup used for *C.name*, as just detailed)

   When this finds a descriptor *v*, the overall result of the attribute lookup is, again, `type(v).__get__(v, x, C)`

   When this finds a non-descriptor value *v*, the overall result of the attribute lookup is just *v*.

When these lookup steps do not find an attribute, Python raises an `AttributeError` exception. However, for lookups of *x.name*, when *C* defines or inherits the special method `__getattr__`, Python calls *C*`.__getattr__(`*x*`, '`*name*`')` rather than raising the exception. It's then up to `__getattr__` to return a suitable value or raise the appropriate exception, normally `AttributeError`.

Consider the following attribute references, defined previously:

```
print(x.e, x.d, x.c, x.b, x.a)              # prints: 88 77 89 67
23
```

x.e and *x.d* succeed in step 2 of the instance lookup process, since no descriptors are involved, and `'e'` and `'d'` are both keys in *x*`.__dict__`. Therefore, the lookups go no further but rather return `88` and `77`. The other three references must proceed to step 3 of the instance process and look in *x*`.__class__` (i.e., *C*). *x.c* and *x.b* succeed in step 1 of the class lookup process, since `'c'` and `'b'` are both keys in *C*`.__dict__`. Therefore, the lookups go no further but rather return `89` and `67`. *x.a* gets all the way to step 2 of the class process, looking in *C*`.__bases__[0]` (i.e., *B*). `'a'` is a key in *B*`.__dict__`; therefore, *x.a* finally succeeds and returns `23`.

Setting an attribute

Note that the attribute lookup steps happen as just described only when you *refer* to an attribute, not when you *bind* an attribute. When you bind (on either a class or an instance) an attribute whose name is not special (unless

a `__setattr__` method, or the `__set__` method of an overriding descriptor, intercepts the binding of an instance attribute), you affect only the `__dict__` entry for the attribute (in the class or instance, respectively). In other words, for attribute binding, there is no lookup procedure involved, except for the check for overriding descriptors.

Bound and Unbound Methods

The method `__get__` of a function object can return the function object itself, or a *bound method object* that wraps the function; a bound method is associated with the specific instance it's obtained from.

In the code in the previous section, attributes *`f`*, *`g`*, and *`h`* are functions; therefore, an attribute reference to any one of them returns a method object that wraps the respective function. Consider the following:

```
print(x.h, x.g, x.f, C.h, C.g, C.f)
```

This statement outputs three bound methods, represented by strings like

`<bound method C.h of <__main__.C object at 0x8156d5c>>`

and then, three function objects, represented by strings like

`<function C.h at 0x102cabae8>`

Bound Methods vs. Function Objects

We get bound methods when the attribute reference is on instance *`x`*, and function objects when the attribute reference is on class *`C`*.

Because a bound method is already associated with a specific instance, call the method as follows:

```
x.h()                    # prints: method h in class C
```

The key thing to notice here is that you don't pass the method's first argument, `self`, by the usual argument-passing syntax. Rather, a bound method of instance *`x`* implicitly binds the `self` parameter to object *`x`*. Thus, the method's body can access the instance's attributes as attributes of `self`, even though we don't pass an explicit argument to the method.

Bound method details

When an attribute reference on an instance, in the course of the lookup, finds a function object that's an attribute in the instance's class, the lookup calls the function's `__get__` method to get the attribute's value. The call, in this case, creates and returns a *bound method* that wraps the function.

Note that when the attribute reference's lookup finds a function object in *`x`*`.__dict__`, the attribute reference operation does *not* create a bound method: in such cases, Python does not treat the function as a descriptor and does not call the function's `__get__` method; rather, the function object itself is the attribute's value. Similarly, Python creates no bound method for callables that are not ordinary functions, such as built-in (as opposed to Python-coded) functions, since such callables are not descriptors.

A bound method has three read-only attributes in addition to those of the function object it wraps. `im_class` is the class object that supplies the method; `im_func` is the wrapped function; `im_self` refers to *`x`*, the instance from which you got the method.

You use a bound method just like its `im_func` function, but calls to a bound method do not explicitly supply an argument corresponding to the first formal parameter (conventionally named `self`). When you call a bound method, the bound method passes `im_self` as the first argument to `im_func` before other arguments (if any) given at the point of call.

Let's follow in excruciating low-level detail the conceptual steps involved in a method call with the normal syntax *`x.name`*`(`*`arg`*`)`. In the following context:

```
def f(a, b): ...                # a function f with two arguments
class C(object):
    name = f
x = C()
```

x is an instance object of class *C*, `name` is an identifier that names a method of *x*'s (an attribute of *C* whose value is a function, in this case function *f*), and *arg* is any expression. Python first checks if `'`*name*`'` is the attribute name in *C* of an overriding descriptor, but it isn't—functions are descriptors because their type defines method `__get__`, but not overriding ones, because their type does not define method `__set__`. Python next checks if `'name'` is a key in *x*`.__dict__`, but it isn't. So Python finds `name` in *C* (everything would work just the same if `name` were found, by inheritance, in one of *C*'s `__bases__`). Python notices that the attribute's value, function object *f*, is a descriptor. Therefore, Python calls *f*`.__get__(`*x*`,` *C*`)`, which creates a bound method object with `im_func` set to *f*, `im_class` set to *C*, and `im_self` set to *x*. Then Python calls this bound method object, with *arg* as the only argument. The bound method inserts `im_self` (i.e., *x*) as the first argument, and *arg* becomes the second one in a call to the bound method's `im_func` (i.e., function `f`). The overall effect is just like calling:

```
x.__class__.__dict__['name'](x, arg)
```

When a bound method's function body executes, it has no special namespace relationship to either its `self` object or any class. Variables referenced are local or global, just like any other function, as covered in "Namespaces." Variables do not implicitly indicate attributes in `self`, nor do they indicate attributes in any class object. When the method needs to refer to, bind, or unbind an attribute of its `self` object, it does so by standard attribute-reference syntax (e.g., `self.`*name*)[4]. The lack of implicit scoping may take some getting used to (simply because Python differs in this respect from many, though far from all, other object-oriented languages), but it results in clarity, simplicity, and the removal of potential ambiguities.

Bound method objects are first-class objects: you can use them wherever you can use a callable object. Since a bound method holds references to both the function it wraps and the `self` object on which it executes, it's a powerful and flexible alternative to a closure (covered in "Nested functions and nested scopes"). An instance object whose class supplies the special method `__call__` (covered in Table 4-1) offers another viable alternative. These constructs let you bundle some behavior (code) and some state (data) into a single callable object. Closures are simplest, but they are limited in their applicability. Here's the closure from "Nested functions and nested scopes":

```
def make_adder_as_closure(augend):
    def add(addend, _augend=augend):
        return addend+_augend
    return add
```

Bound methods and callable instances are richer and more flexible than closures. Here's how to implement the same functionality with a bound method:

```
def make_adder_as_bound_method(augend):
    class Adder(object):
        def __init__(self, augend):
            self.augend = augend
        def add(self, addend):
            return addend+self.augend
    return Adder(augend).add
```

And here's how to implement it with a callable instance (an instance whose class supplies the special method `__call__`):

```
def make_adder_as_callable_instance(augend):
    class Adder(object):
        def __init__(self, augend):
            self.augend = augend
        def __call__(self, addend):
            return addend+self.augend
    return Adder(augend)
```

From the viewpoint of the code that calls the functions, all of these factory functions are interchangeable, since all of them return callable objects that are polymorphic (i.e., usable in the same ways). In terms of implementation, the closure is simplest; the bound method and the callable instance use more flexible, general, and powerful mechanisms, but there is no need for that extra power in this simple example.

Inheritance

When you use an attribute reference `C.name` on a class object `C`, and `'name'` is not a key in `C.__dict__`, the lookup implicitly proceeds on each class object that is in `C.__bases__` in a specific order (which for historical reasons is known as the *method resolution order*, or MRO, but, in fact, applies to all attributes, not just methods). `C`'s base classes may in turn have their own bases. The lookup checks direct and indirect ancestors, one by one, in MRO, stopping when `'name'` is found.

Method resolution order

The lookup of an attribute name in a class essentially occurs by visiting ancestor classes in left-to-right, depth-first order. However, in the presence of multiple inheritance (which makes the inheritance graph a general Directed Acyclic Graph rather than specifically a tree), this simple approach might lead to some ancestor class being visited twice. In such cases, the resolution order leaves in the lookup sequence only the *rightmost* occurrence of any given class.

Each class and built-in type has a special read-only class attribute called `__mro__`, which is the tuple of types used for method resolution, in order. You can reference `__mro__` only on classes, not on instances, and, since `__mro__` is a read-only attribute, you cannot rebind or unbind it. For a detailed and highly technical explanation of all aspects of Python's MRO, you may want to study an online essay by Michele Simionato, The Python 2.3 Method Resolution Order, and GvR's history note at the Python History site.

Overriding attributes

As we've just seen, the search for an attribute proceeds along the MRO (typically, up the inheritance tree) and stops as soon as the attribute is found. Descendant classes are always examined before their ancestors, so that, when a subclass defines an attribute with the same name as one in a superclass, the search finds the definition in the subclass and stops there. This is known as the subclass *overriding* the definition in the superclass. Consider:

```
class B:
    a = 23
    b = 45
    def f(self):
        print('method f in class B')
    def g(self):
        print('method g in class B')
class C(B):
    b = 67
    c = 89
    d = 123
    def g(self):
        print('method g in class C')
    def h(self):
        print('method h in class C')
```

In this code, class `C` overrides attributes `b` and `g` of its superclass `B`. Note that, unlike in some other languages, in Python you may override data attributes just as easily as callable attributes (methods).

Delegating to superclass methods

When a subclass `C` overrides a method `f` of its superclass `B`, the body of `C.f` often wants to delegate some part of its operation to the superclass's implementation of the method. This can sometimes be done using a function object, as follows:

```
class Base:
    def greet(self, name):
        print('Welcome', name)
class Sub(Base):
    def greet(self, name):
```

```
        print('Well Met and', end=' ')
        Base.greet(self, name)
x = Sub()
x.greet('Alex')
```

The delegation to the superclass, in the body of *`Sub.greet`*, uses a function object obtained by attribute reference *`Base.greet`* on the superclass, and therefore passes all arguments normally, including `self`. Delegating to a superclass implementation is a frequent use of such function objects.

One common use of delegation occurs with special method `__init__`. When Python creates an instance, it does not automatically call the `__init__` methods of base classes, as in some other object-oriented languages. Thus, it is up to a subclass to properly initialize superclasses, using delegation if necessary. For example:

```
class Base:
    def __init__(self):
        self.anattribute = 23
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.anotherattribute = 45
```

If the `__init__` method of class *`Derived`* didn't explicitly call that of class *`Base`*, instances of *`Derived`* would miss that portion of their initialization. Thus, such instances would lack attribute *`anattribute`*. This issue does *not* arise if a subclass does not define `__init__` since, in that case, it inherits it from the superclass. So there is *never* any reason to code:

```
class Derived(Base):
    def __init__(self):
        Base.__init__(self)
```

Never Code a Method That Just Delegates to the Superclass

Never define a semantically empty `__init__` (i.e., one that just delegates to the superclass). Rather, just inherit `__init__` from the superclass. This advice applies to all methods, special or not, but, for some reason, the bad habit of coding such semantically empty methods occurs most often for `__init__`.

The above code illustrates the concept of delegation to an object's superclass, but it is actually a poor practice to code these superclasses explicitly by name. If the base class is renamed, all the call sites to it must be updated. Or worse, if refactoring the class hierarchy introduces a new layer between the *`Derived`* and *`Base`* class, the newly-inserted class's method will be silently skipped.

The recommended approach is to call methods defined in a superclass using the *`super`* built-in type. To invoke methods up the inheritance chain, just call *`super()`*, without arguments.

```
class Derived(Base):
    def __init__(self):
        super().__init__()
        self.anotherattribute = 45
```

Cooperative superclass method calling

Explicitly calling the superclass's version of a method with the above syntax is also quite problematic in cases of multiple inheritance with "diamond-shaped" graphs. Consider the following definitions:

```
class A:
    def met(self):
        print('A.met')
class B(A):
    def met(self):
        print('B.met')
        A.met(self)
class C(A):
```

```
    def met(self):
        print('C.met')
        A.met(self)
class D(B,C):
    def met(self):
        print('D.met')
        B.met(self)
        C.met(self)
```

When we call *D().met(), A.met* ends up being called twice. How can we ensure that each ancestor's implementation of the method is called once and only once? Again, the solution is to use *super*:

```
class A:
    def met(self):
        print('A.met')
class B(A):
    def met(self):
        print('B.met')
        super().met()
class C(A):
    def met(self):
        print('C.met')
        super().met()
class D(B,C):
    def met(self):
        print('D.met')
        super().met()
```

Now, *D().met()* results in exactly one call to each class's version of *met*. If you get into the good habit of always coding superclass calls with `super`, your classes fit smoothly even in complicated inheritance structures. There are no ill effects if the inheritance structure instead turns out to be simple.

The only situation in which you may prefer to use the rougher approach of calling superclass methods through the explicit syntax is when various classes have different and incompatible signatures for the same method—an unpleasant situation in many respects; if you do have to deal with it, the explicit syntax may sometimes be the least of evils. Proper use of multiple inheritance is seriously hampered—but then, even the most fundamental properties of OOP, such as polymorphism between base and subclass

instances, are impaired when you give methods of the same name different signatures in superclass and subclass.

“Deleting” class attributes

Inheritance and overriding provide a simple and effective way to add or modify (override) class attributes (such as methods) noninvasively (i.e., without modifying the base class defining the attributes) by adding or overriding the attributes in subclasses. However, inheritance does not offer a way to delete (hide) base classes’ attributes noninvasively. If the subclass simply fails to define (override) an attribute, Python finds the base class’s definition. If you need to perform such deletion, possibilities include:

- Override the method and raise an exception in the method’s body.
- Eschew inheritance, hold the attributes elsewhere than in the subclass’s `__dict__`, and define `__getattr__` for selective delegation.
- Override `__getattribute__` to similar effect.

The last of these techniques is shown in “__getattribute__”.

The Built-in object Type

The built-in `object` type is the ancestor of all built-in types and classes. The `object` type defines some special methods (documented in “Special Methods”) that implement the default semantics of objects:

`__new__, __init__`

> You can create a direct instance of `object` by calling `object()` without any arguments. The call uses `object.__new__` and `object.__init__` to make and return an instance object without attributes (and without even a `__dict__` in which to hold attributes). Such instance objects may be useful as “sentinels,” guaranteed to compare unequal to any other distinct object.

`__delattr__, __getattribute__, __setattr__`

By default, any object handles attribute references (as covered in "Attribute Reference Basics") using these methods of `object`.

`__hash__, __repr__, __str__`

Any object can be passed to the functions `hash` and `repr` and to the type `str`.

A subclass of `object` (i.e., any class) may (often will!) override any of these methods and/or add others.

Class-Level Methods

Python supplies two built-in nonoverriding descriptor types, which give a class two distinct kinds of "class-level methods": *static methods* and *class methods*.

Static methods

A *static method* is a method that you can call on a class, or on any instance of the class, without the special behavior and constraints of ordinary methods regarding the first parameter. A static method may have any signature; it may have no parameters; the first parameter, if any, plays no special role. You can think of a static method as an ordinary function that you're able to call normally, despite the fact that it happens to be bound to a class attribute.

While it is never necessary to define static methods (you can always choose to instead define a normal function, outside the class), some programmers consider them to be an elegant syntax alternative when a function's purpose is tightly bound to some specific class.

To build a static method, call the built-in type `staticmethod` and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may also choose to perform it elsewhere. The only argument to `staticmethod` is the function to call

when Python calls the static method. The following example shows one way to define and call a static method:

```
class AClass(object):
    def astatic():
        print('a static method')
    astatic = staticmethod(astatic)
an_instance = AClass()
print(AClass.astatic())             # prints: a static method
print(an_instance.astatic())        # prints: a static method
```

This example uses the same name for the function passed to `staticmethod` and for the attribute bound to `staticmethod`'s result. This naming is not mandatory, but it's a good idea, and we recommend you always use it. Python offers a special, simplified syntax to support this style, covered in "Decorators."

Class methods

A *class method* is a method you can call on a class or on any instance of the class. Python binds the method's first parameter to the class on which you call the method, or the class of the instance on which you call the method; it does not bind it to the instance, as for normal bound methods. The first parameter of a class method is conventionally named *`cls`*.

While it is never *necessary* to define class methods (you can always choose to define a normal function, outside the class, that takes the class object as its first parameter), class methods are an elegant alternative to such functions (particularly since they can usefully be overridden in subclasses, when that is necessary).

To build a class method, call the built-in type `classmethod` and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may choose to perform it elsewhere. The only argument to `classmethod` is the function to call when Python calls the class method. Here's one way you can define and call a class method:

```
class ABase(object):
    def aclassmet(cls):
        print('a class method for', cls.__name__)
    aclassmet = classmethod(aclassmet)
class ADeriv(ABase):
    pass
b_instance = ABase()
d_instance = ADeriv()
print(ABase.aclassmet())        # prints: a class method for
ABase
print(b_instance.aclassmet())   # prints: a class method for
ABase
print(ADeriv.aclassmet())       # prints: a class method for
ADeriv
print(d_instance.aclassmet())   # prints: a class method for
ADeriv
```

This example uses the same name for the function passed to `classmethod` and for the attribute bound to `classmethod`'s result. This naming is not mandatory, but it's a good idea, and we recommend that you always use it. Python offers a special, simplified syntax to support this style, covered in "Decorators."

Properties

Python supplies a built-in overriding descriptor type, usable to give a class's instances *properties*.

A *property* is an instance attribute with special functionality. You reference, bind, or unbind the attribute with the normal syntax (e.g., `print(`*`x.prop`*`), x.`*`prop`*`=23, del `*`x.prop`*`)`. However, rather than following the usual semantics for attribute reference, binding, and unbinding, these accesses call on instance *`x`* the methods that you specify as arguments to the built-in type `property`. Here's one way to define a read-only property:

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def get_area(self):
```

```
        return self.width * self.height
    area = property(get_area, doc='area of the rectangle')
```

Each instance *r* of class *Rectangle* has a synthetic read-only attribute *r.area*, computed on the fly in method *r.get_area()* by multiplying the sides. The docstring *Rectangle.area*.__doc__ is 'area of the rectangle'. Attribute *r.area* is read-only (attempts to rebind or unbind it fail) because we specify only a get method in the call to property, no set or del methods.

Properties perform tasks similar to those of special methods __getattr__, __setattr__, and __delattr__ (covered in "General-Purpose Special Methods"), but properties are faster and simpler. To build a property, call the built-in type property and bind its result to a class attribute. Like all binding of class attributes, this is normally done in the body of the class, but you may choose to do it elsewhere. Within the body of a class *C*, you can use the following syntax:

```
attrib = property(fget=None, fset=None, fdel=None, doc=None)
```

When *x* is an instance of *C* and you reference *x.attrib*, Python calls on *x* the method you passed as argument *fget* to the property constructor, without arguments. When you assign *x.attrib* = *value*, Python calls the method you passed as argument *fset*, with *value* as the only argument. When you execute del *x.attrib*, Python calls the method you passed as argument *fdel*, without arguments. Python uses the argument you passed as *doc* as the docstring of the attribute. All parameters to property are optional. When an argument is missing, Python raises an exception when some code attempts that operation. For example, in the *Rectangle* example, we made property *area* read-only, because we passed an argument only for parameter *fget*, and not for parameters *fset* and *fdel*.

An elegant syntax to create properties in a class is to use property as a *decorator* (see "Decorators"):

```
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    @property
    def area(self):
        '''area of the rectangle'''
        return self.width * self.height
```

To use this syntax, you must give the getter method the same name as you want the property to have; the method's docstring becomes the docstring of the property. If you want to add a setter and/or a deleter as well, use decorators named (in this example) *`area.setter`* and *`area.deleter`*, and name the methods thus decorated the same as the property, too. For example:

```
import math
class Rectangle(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    @property
    def area(self):
        '''area of the rectangle'''
        return self.width * self.height
    @area.setter
    def area(self, value):
        scale = math.sqrt(value/self.area)
        self.width *= scale
        self.height *= scale
```

Why properties are important

The crucial importance of properties is that their existence makes it perfectly safe (and indeed advisable) for you to expose public data attributes as part of your class's public interface. Should it ever become necessary, in future versions of your class or other classes that need to be polymorphic to it, to have some code execute when the attribute is referenced, rebound, or unbound, you will be able to change the plain attribute into a property and get the desired effect without any impact on any code that uses your class (AKA "client code"). So you avoid goofy

idioms, such as *accessor* and *mutator* methods, required by OO languages lacking properties. For example, client code can use natural idioms such as:

```
some_instance.widget_count += 1
```

rather than being forced into contorted nests of accessors and mutators such as:

```
some_instance.set_widget_count(some_instance.get_widget_count() +
1)
```

If you're ever tempted to code methods whose natural names are something like *`get_this`* or *`set_that`*, wrap those methods into properties instead, for clarity.

Properties and inheritance

Inheritance of properties is just like for any other attribute. However, there's a little trap for the unwary: *the methods called upon to access a property are those defined in the class in which the property itself is defined*, without intrinsic use of further overriding that may happen in subclasses. For example:

```
class B(object):
    def f(self):
        return 23
    g = property(f)
class C(B):
    def f(self):
        return 42
c = C()
print(c.g)              # prints: 23, not 42
```

Accessing property *`c.g`* calls *`B.f`*, not *`C.f`* as you might expect. The reason is quite simple: the property constructor receives (directly or via the decorator syntax) the *function object `f`* (and that happens at the time the `class` statement for `B` executes, so the function object in question is the one also known as *`B.f`*). The fact that the subclass *`C`* later redefines *name `f`* is therefore irrelevant, since the property performs no lookup for that name,

but rather uses the function object it received at creation time. If you need to work around this issue, you can always do it by adding the extra level of lookup indirection yourself:

```
class B(object):
    def f(self):
        return 23
    def _f_getter(self):
        return self.f()
    g = property(_f_getter)
class C(B):
    def f(self):
        return 42
c = C()
print(c.g)                # prints: 42, as expected
```

Here, the function object held by the property is *`B._f_getter`*, which in turn does perform a lookup for name *`f`* (since it calls `self.f()`); therefore, the overriding of *`f`* has the expected effect. As David Wheeler famously put it, "All problems in computer science can be solved by another level of indirection."[5]

__slots__

Normally, each instance object *`x`* of any class *`C`* has a dictionary *`x`*`.__dict__` that Python uses to let you bind arbitrary attributes on *`x`*. To save a little memory (at the cost of letting *`x`* have only a predefined set of attribute names), you can define in class *`C`* a class attribute named `__slots__`, a sequence (normally a tuple) of strings (normally identifiers). When class *`C`* has `__slots__`, instance *`x`* of class *`C`* has no `__dict__`: trying to bind on *`x`* an attribute whose name is not in *`C`*`.__slots__` raises an exception.

Using `__slots__` lets you reduce memory consumption for small instance objects that can do without the powerful and convenient ability to have arbitrarily named attributes. `__slots__` is worth adding only to classes that can have so many instances that saving a few tens of bytes per instance is important—typically classes that could have millions, not mere

thousands, of instances alive at the same time. Unlike most other class attributes, `__slots__` works as we've just described only if an assignment in the class body binds it as a class attribute. Any later alteration, rebinding, or unbinding of `__slots__` has no effect, nor does inheriting `__slots__` from a base class. Here's how to add `__slots__` to the *`Rectangle`* class defined earlier to get smaller (though less flexible) instances:

```
class OptimizedRectangle(Rectangle):
    __slots__ = 'width', 'height'
```

No need to define a slot for the *`area`* property: `__slots__` does not constrain properties, only ordinary *instance* attributes, which would reside in the instance's `__dict__` if `__slots__` wasn't defined.

||**3.8++**|| `__slots__` attributes can also be defined using a **dict** with attribute names for the keys and docstrings for the values. *`OptimizedRectangle`* could be declared more fully as:

```
class OptimizedRectangle(Rectangle):
    __slots__ = {'width': 'rectangle width in pixels',
                 'height': 'rectangle height in pixels'}
```

`__getattribute__`

All references to instance attributes go through special method `__getattribute__`. This method comes from `object`, where it implements attribute reference semantics as documented in "Attribute Reference Basics". You may override `__getattribute__` for purposes such as hiding inherited class attributes for a subclass's instances. The following example shows one way to implement a list without `append`:

```
class listNoAppend(list):
    def __getattribute__(self, name):
        if name == 'append':
            raise AttributeError(name)
        return list.__getattribute__(self, name)
```

An instance *x* of class *listNoAppend* is almost indistinguishable from a built-in list object, except that performance is substantially worse, and any reference to *x*.append raises an exception.

Per-Instance Methods

An instance can have instance-specific bindings for all attributes, including callable attributes (methods). For a method, just like for any other attribute (except those bound to overriding descriptors), an instance-specific binding hides a class-level binding: attribute lookup does not consider the class when it finds a binding directly in the instance. An instance-specific binding for a callable attribute does not perform any of the transformations detailed in "Bound and Unbound Methods": the attribute reference returns exactly the same callable object that was earlier bound directly to the instance attribute.

However, this does not work as you might expect for per-instance bindings of the special methods that Python calls implicitly as a result of various operations, as covered in "Special Methods". Such implicit uses of special methods always rely on the *class-level* binding of the special method, if any. For example:

```
def fake_get_item(idx):
    return idx
class MyClass(object):
    pass
n = MyClass()
n.__getitem__ = fake_get_item
print(n[23])                        # results in:
# Traceback (most recent call last):
#   File "<stdin>", line 1, in ?
# TypeError: unindexable object
```

Inheritance from Built-in Types

A class can inherit from a built-in type. However, a class may directly or indirectly extend multiple built-in types only if those types are specifically designed to allow this level of mutual compatibility. Python does not support unconstrained inheritance from multiple arbitrary built-in types.

Normally, a new-style class only extends at most one substantial built-in type—for example:

```
class noway(dict, list):
    pass
```

raises a `TypeError` exception, with a detailed explanation of "multiple bases have instance lay-out conflict." If you ever see such error messages, it means that you're trying to inherit, directly or indirectly, from multiple built-in types that are not specifically designed to cooperate at such a deep level.

Special Methods

A class may define or inherit special methods (i.e., methods whose names begin and end with double underscores, AKA "dunder" or "magic" methods). Each special method relates to a specific operation. Python implicitly calls a special method whenever you perform the related operation on an instance object. In most cases, the method's return value is the operation's result, and attempting an operation when its related method is not present raises an exception.

Throughout this section, we point out the cases in which these general rules do not apply. In the following, *`x`* is the instance of class *`C`* on which you perform the operation, and *`y`* is the other operand, if any. The parameter `self` of each method also refers to the instance object *`x`*. In the following sections, whenever we mention calls to *`x.__whatever__(...)`*, keep in mind that the exact call happening is rather, pedantically speaking, *`x.__class__.__whatever__(x, ...)`*.

General-Purpose Special Methods

Some special methods relate to general-purpose operations. A class that defines or inherits these methods allows its instances to control such operations. These operations can be divided into categories:

Initialization and finalization

A class can control its instances' initialization (a very common requirement) via the special methods `__new__` and `__init__`, and/or their finalization (a rare requirement) via `__del__`.

Representation as string

A class can control how Python renders its instances as strings via special methods `__repr__`, `__str__`, `__format__`, `__bytes__`.

Comparison, hashing, and use in a Boolean context

A class can control how its instances compare with other objects (methods `__lt__`, `__le__`, `__gt__`, `__ge__`, `__eq__`, `__ne__`), how dictionaries use them as keys and sets use them as members (`__hash__`), and whether they evaluate to true or false in Boolean contexts (`__bool__`).

Attribute reference, binding, and unbinding

A class can control access to its instances' attributes (reference, binding, unbinding) via special methods `__getattribute__`, `__getattr__`, `__setattr__`, and `__delattr__`.

Callable instances

An instance is callable, just like a function object, if its class has the special method `__call__`.

Table 4-1 documents the general-purpose special methods.

Table 4-1. General-purpose special methods

`__bool__`	`__bool__(self)` When evaluating *x* as true or false (see "Boolean Values")—for example, on a call to `bool(`*x*`)`—Python calls *x*`.__bool__()`, which should return `True` or `False`. When `__bool__` is not present, Python calls `__len__`,

	and takes *x* as false when `x.__len__()` returns `0` (to check if a container is nonempty, avoid coding `if len(`*container*`)>0:`; use `if` *container*: instead). When neither `__bool__` nor `__len__` is present, Python considers *x* true.
__bytes__	`__bytes__(self)` Calling `bytes(x)` calls `x.__bytes__()`, if present. If a class supplies special methods `__bytes__` and `__str__`, they should return equivalent strings resp. of bytes and str type.
__call__	`__call__(self[,`*args*`...])` When you call `x([`*args*`...])`, Python translates the operation into a call to `x.__call__([`*args*`...])`. The arguments for the call operation correspond to the parameters for the `__call__` method, minus the first. The first parameter, conventionally called `self`, refers to *x*, and Python supplies it implicitly and automatically, just as in any other call to a bound method.
__dir__	`__dir__(self)` When you call `dir(x)`, Python translates the operation into a call to `x.__dir__()`, which must return a sorted list of *x*'s attributes. When *x*'s class has no `__dir__`, `dir(x)` performs introspection to return a list of *x*'s attributes, striving to produce relevant, rather than complete, information.
__del__	`__del__(self)` Just before *x* disappears via garbage collection, Python calls `x.__del__()` to let *x* finalize itself. If `__del__` is absent, Python does no special finalization on garbage-collecting *x* (this is the most common case: very few classes need to define `__del__`). Python ignores the return value of `__del__` and doesn't implicitly call `__del__` methods of class *C*'s superclasses. *C*.`__del__` must explicitly perform any needed finalization, including, if need be, by delegation. When class *C* has base classes to finalize, *C*.`__del__` must call `super().__del__()`. The `__del__` method has no connection with the `del` statement, covered in "del Statements". `__del__` is generally not the best approach when you need timely and guaranteed finalization. For such needs, use the `try/finally` statement covered in "try/finally" (or, even better, the `with` statement, covered in "The with Statement"). Instances of classes defining `__del__` don't participate in cyclic-garbage collection, covered in "Garbage Collection". Be careful to avoid reference loops involving such instances: define `__del__` only when there is no feasible alternative.
__delattr__	`__delattr__(self,` *name*`)` At every request to unbind attribute `x.y` (typically, a `del` statement `del x.y`), Python calls `x.__delattr__('y')`. All the considerations discussed later for `__setattr__` also apply to `__delattr__`. Python ignores the return value of `__delattr__`. Absent `__delattr__`, Python turns `del x.y` into `del x.__dict__['y']`.
__eq__,__ge__,	`__eq__(self,` *other*`)` `__ge__(self,` *other*`)` `__gt__(self,` *other*`)` `__le__(self,` *other*`)`

__gt__,__le__, __lt__,__ne__	`__lt__(self, other)` `__ne__(self, other)` The comparisons `x==y`, `x>=y`, `x>y`, `x<=y`, `x<y`, and `x!=y`, respectively, call the special methods listed here, which should return `False` or `True`. Each method may return `NotImplemented` to tell Python to handle the comparison in alternative ways (e.g., Python may then try `y>x` in lieu of `x<y`). Best practice is to define only one inequality comparison method (normally `__lt__`) plus `__eq__`, and decorate the class with `functools.total_ordering` (covered in Table 7-4) to avoid boilerplate, and any risk of logical contradictions in your comparisons.
__format__	`__format__(self, format_string='')` Calling `format(x)` calls `x.__format__('')`, and calling `format(x, format_string)` calls `x.__format__(format_string)`. The class is responsible for interpreting the format string (each class may define its own small "language" of format specifications, inspired by those implemented by built-in types as covered in "String Formatting"). If `__format__` is inherited from `object`, it delegates to `__str__` and does not accept a nonempty format string.
__getattr__	`__getattr__(self, name)` When `x.y` can't be found by the usual steps (i.e., when `AttributeError` would usually be raised), Python calls `x.__getattr__('y')`. Python does not call `__getattr__` for attributes found by normal means as keys in `x.__dict__`, or via `x.__class__`). If you want Python to call `__getattr__` for *every* attribute, keep the attributes elsewhere (e.g., in another dict referenced by an attribute with a private name), or override `__getattribute__` instead. `__getattr__` should raise `AttributeError` if it can't find `y`.
__getattribute__	`__getattribute_(self, name)` At every request to access attribute `x.y`, Python calls `x.__getattribute__('y')`, which must get and return the attribute value or else raise `AttributeError`. The usual semantics of attribute access (`x.__dict__`, `C.__slots__`, `C`'s class attributes, `x.__getattr__`) are all due to `object.__getattribute__`. When class `C` overrides `__getattribute__`, it must implement all of the attribute semantics it wants to offer. The typical way to implement attribute access is by delegating (e.g., call `object.__getattribute__(self, ...)` as part of the operation of your override of `__getattribute__`).
__hash__	`__hash__(self)` Calling `hash(x)` calls `x.__hash__()` (and so do other contexts that need to know `x`'s hash value, namely, using `x` as a dictionary key, such as `D[x]` where `D` is a dictionary, or using `x` as a set member). `__hash__` must return an `int` such that `x==y` implies `hash(x)==hash(y)`, and must always return the same value for a given object. When `__hash__` is absent, calling `hash(x)` calls `id(x)` instead, as long as `__eq__` is also absent. Other contexts that need to know `x`'s hash value behave the same way.

	Any *x* such that `hash(x)` returns a result, rather than raising an exception, is known as a *hashable object*. When `__hash__` is absent, but `__eq__` is present, calling `hash(x)` raises an exception (and so do other contexts that need to know *x*'s hash value). In this case, *x* is not hashable and therefore cannot be a dictionary key or set member. You normally define `__hash__` only for immutable objects that also define `__eq__`. Note that if there exists any *y* such that *x*==*y*, even if *y* is of a different type, and both *x* and *y* are hashable, you *must* ensure that `hash(x)==hash(y)`.
__init__	`__init__(self[,args...])` When a call `C([args...])` creates instance *x* of class *C*, Python calls `x.__init__([args...])` to let *x* initialize itself. If `__init__` is absent (i.e., it's inherited from `object`), you must call *C* without arguments, `C()`, and *x* has no instance-specific attributes on creation. Python performs no implicit call to `__init__` methods of class *C*'s superclasses. `C.__init__` must explicitly perform any initialization, including, if need be, by delegation. For example, when class *C* has a base class *B* to initialize without arguments, the code in `C.__init__` must explicitly call `super().__init__()`. `__init__`'s inheritance works just like for any other method or attribute: if *C* itself does not override `__init__`, it inherits it from the first superclass in its `__mro__` to override `__init__`, like every other attribute. `__init__` must return `None`; otherwise, calling the class raises a `TypeError`.
__new__	`__new__(cls[,args...])` When you call `C([args...])`, Python gets the new instance *x* that you are creating by invoking `C.__new__(C,[args...])`. Every class has the class method `__new__` (usually, it just inherits it from `object`), which can return any value *x*. In other words, `__new__` need not return a new instance of *C*, although it's expected to do so. If the value *x* that `__new__` returns is an instance of *C* or of any subclass of *C* (whether a new or previously existing one), Python continues by implicitly calling `__init__` on *x* (with the same `[args...]` that were originally passed to `__new__`).

Initialize Immutables in `__new__`, All Others in `__init__`

You can perform most kinds of initialization of new instances in either `__init__` or `__new__`, so you may wonder where best to place them. Put the initialization in `__init__` only, unless you have a reason to put it in `__new__`. (When a type is immutable, `__init__` cannot changed its instances: in this case, `__new__` has to perform all initialization.)

	__repr__	`__repr__(self)` Calling `repr(x)` (which happens implicitly in the interactive interpreter when *x* is the result of an expression statement) calls `x.__repr__()` to get and return a complete string representation of *x*. If `__repr__` is absent, Python uses a default string representation. `__repr__` should return a string with unambiguous information on *x*. When feasible, try to ensure that `eval(repr(x))==x` (but, don't go crazy aiming for that goal!).
setattr	**__setattr__**	`__setattr__(self, name, value)` At any request to bind attribute `x.y` (usually, an assignment statement `x.y=value`, but also, e.g., `setattr(x, 'y', value)`), Python calls `x.__setattr__('y', value)`. Python always calls `__setattr__` for *any* attribute binding on *x*—a major difference from `__getattr__` (`__setattr__` is closer to `__getattribute__`!). To avoid recursion, when `x.__setattr__` binds *x*'s attributes, it must modify `x.__dict__` directly (e.g., via `x.__dict__[name]=value`); or better, `__setattr__` can delegate to the superclass (call `super().__setattr__('y', value)`). Python ignores the return value of `__setattr__`. If `__setattr__` is absent (i.e., inherited from object), and `C.y` is not an overriding descriptor, Python usually translates `x.y=z` into `x.__dict__['y']=z`.
__str__		`__str__(self)` A `str(x)` call, and `print(x)`, call `x.__str__()` to get an informal, concise string representation of *x*. If `__str__` is absent, Python calls `x.__repr__`. `__str__` should return a convenient human-readable string, even when that entails some approximation.

Special Methods for Containers

An instance can be a *container* (a sequence, mapping, or set—mutually exclusive concepts[6]). For maximum usefulness, containers should provide special methods `__getitem__`, `__contains__`, and `__iter__` (and, if mutable, also `__setitem__` and `__delitem__`), plus nonspecial methods discussed in the following sections. In many cases, suitable implementations of the nonspecial methods can be had by extending the appropriate *abstract base class*, from module `collections`, such as `Sequence`, `MutableSequence`, and so on, as covered in "Abstract Base Classes".

Sequences

In each item-access special method, a sequence that has `L` items should accept any integer `key` such that `-L<=key<L`.[7] For compatibility with built-in sequences, a negative index `key`, `0>key>=-L`, should be equivalent to `key+L`. When `key` has an invalid type, indexing should raise `TypeError`. When `key` is a value of a valid type but out of range, indexing should raise `IndexError`. For sequence classes that do not define `__iter__`, the `for` statement relies on these requirements, as do built-in functions that take iterable arguments. Every item-access special method of a sequence should also, if at all practical, accept as its index argument an instance of the built-in type `slice` whose `start`, `step`, and `stop` attributes are `int`s or `None`; the *slicing* syntax relies on this requirement, as covered in "Container slicing".

A sequence should also allow concatenation (with another sequence of the same type) by `+`, and repetition by `*` (multiplication by an integer). A sequence should therefore have special methods `__add__`, `__mul__`, `__radd__`, and `__rmul__`, covered in "Special Methods for Numeric Objects"; *mutable* sequences should also have equivalent in-place methods `__iadd__` and `__imul__`. A sequence should be meaningfully comparable to another sequence of the same type, implementing *lexicographic* comparison like lists and tuples do. (Inheriting from ABCs `Sequence` or `MutableSequence`, alas, does not suffice to fulfill all of these requirements; such inheritance, at most, only supplies `__iadd__`.)

Every sequence should have the nonspecial methods covered in "List methods": `count` and `index` in any case, and, if mutable, then also `append`, `insert`, `extend`, `pop`, `remove`, `reverse`, and `sort`, with the same signatures and semantics as the corresponding methods of lists. (Inheriting from ABCs `Sequence` or `MutableSequence` does suffice to fulfill these requirements, except for `sort`.)

An immutable sequence should be hashable if, and only if, all of its items are. A sequence type may constrain its items in some ways (for example, accepting only string items), but that is not mandatory.

Mappings

A mapping's item-access special methods should raise `KeyError`, rather than `IndexError`, when they receive an invalid *`key`* argument value of a valid type. Any mapping should define the nonspecial methods covered in "Dictionary Methods": `copy`, `get`, `items`, `keys`, `values`. A mutable mapping should also define methods `clear`, `pop`, `popitem`, `setdefault`, and `update`. (Inheriting from ABCs `Mapping` or `MutableMapping` does fulfill these requirements, except for `copy`.)

An immutable mapping should be hashable if all of its items are. A mapping type may constrain its keys in some ways (for example, accepting only hashable keys, or, even more specifically, accepting, say, only string keys), but that is not mandatory. Any mapping should be meaningfully comparable to another mapping of the same type (at least for equality and inequality; not necessarily for ordering comparisons).

Sets

Sets are a peculiar kind of container—containers that are neither sequences nor mappings, and cannot be indexed, but do have a length (number of elements) and are iterable. Sets also support many operators (`&`, `|`, `^`, `-`, as well as membership tests and comparisons) and equivalent nonspecial methods (`intersection`, `union`, and so on). If you implement a set-like container, it should be polymorphic to Python built-in sets, covered in "Sets". (Inheriting from ABCs `Set` or `MutableSet` does fulfill these requirements.)

An immutable set-like type should be hashable if all of its elements are. A set-like type may constrain its elements in some ways (for example, accepting only hashable elements, or, even more specifically, accepting, say, only integer elements), but that is not mandatory.

Container slicing

When you reference, bind, or unbind a slicing such as *`x`*`[`*`i`*`:`*`j`*`]` or *`x`*`[`*`i`*`:`*`j`*`:`*`k`*`]` on a container *`x`* (in practice, this is only used with sequences), Python calls *`x`*'s applicable item-access special method, passing as *`key`* an object of a built-in type called a *slice object*. A slice object has the attributes

`start`, `stop`, and `step`. Each attribute is `None` if you omit the corresponding value in the slice syntax. For example, `del x[:3]` calls `x.__delitem__(y)`, where `y` is a slice object such that `y.stop` is `3`, `y.start` is `None`, and `y.step` is `None`. It is up to container object `x` to appropriately interpret slice object arguments passed to `x`'s special methods. The method `indices` of slice objects can help: call it with your container's length as its only argument, and it returns a tuple of three nonnegative indices suitable as `start`, `stop`, and `step` for a loop indexing each item in the slice. A common idiom in a sequence class's `__getitem__` special method, to fully support slicing, is, for example:

```
def __getitem__(self, index):
    # Recursively specialcase slicing
    if isinstance(index, slice):
        return self.__class__(self[x]
                              for x in
range(*self.indices(len(self))))
    # Check index, and deal with a negative and/or out-of-bounds
index
    if not isinstance(index, numbers.Integral):
        raise TypeError
    if index < 0:
        index += len(self)
    if not (0 <= index < len(self)):
        raise IndexError
    # Index is now a correct integral number, within
range(len(self))
    ...rest of __getitem__, dealing with single-item access...
```

This idiom uses generator-expression (genexp) syntax and assumes that your class's `__init__` method can be called with an iterable argument to create a suitable new instance of the class.

Container methods

The special methods `__getitem__`, `__setitem__`, `__delitem__`, `__iter__`, `__len__` and `__contains__` expose container functionality (see Table 4-2).

Table 4-2. Container methods

__contains__	`__contains__(self, item)`
	The Boolean test `y in x` calls `x.__contains__(y)`. When `x` is a sequence, or set-like, `__contains__` should return `True` when `y` equals the value of an item in `x`. When `x` is a mapping, `__contains__` should return `True` when `y` equals the value of a key in `x`. Otherwise, `__contains__` should return `False`. When `__contains__` is absent, Python performs `y in x` as follows, taking time proportional to `len(x)`:
	`for z in x:`
	`  if y==z:`
	`  return True`
	`return False`

__delitem__	`__delitem__(self, key)` For a request to unbind an item or slice of `x` (typically **del** `x[key]`), Python calls `x.__delitem__(key)`. A container `x` should have `__delitem__` if `x` is mutable and items (and possibly slices) can be removed.
__getitem__	`__getitem__(self, key)` When you access `x[key]` (i.e., when you index or slice container `x`), Python calls `x.__getitem__(key)`. All (non-set-like) containers should have `__getitem__`.
__iter__	`__iter__(self)` For a request to loop on all items of `x` (typically **for** `item` **in** `x`), Python calls `x.__iter__()` to get an iterator on `x`. The built-in function `iter(x)` also calls `x.__iter__()`. When `__iter__` is absent, `iter(x)` synthesizes and returns an iterator object that wraps `x` and yields `x[0]`, `x[1]`, and so on, until one of these indexings raises `IndexError` to indicate the end of the container. However, it is best to ensure that all of the container classes you code have `__iter__`.
__len__	`__len__(self)` Calling `len(x)` calls `x.__len__()` (and so do other built-in functions that need to know how many items are in container `x`). `__len__` should return an `int`, the number of items in `x`. Python also calls `x._len__()` to evaluate `x` in a Boolean context, when `__bool__` is absent; in this case, a container is taken as false if and only if the container is empty (i.e., the container's length is `0`). All containers should have `__len__`, unless it's just too expensive for the container to determine how many items it contains.
__setitem__	`__setitem__(self,key, value)` For a request to bind an item or slice of `x` (typically an assignment `x[key]=value`), Python calls `x.__setitem__(key, value)`. A

`__`	container *x* should have `__setitem__` if *x* is mutable, so items, and maybe slices, can be added or rebound.

Abstract Base Classes

Abstract base classes (ABCs) are an important pattern in object-oriented (OO) design: they're classes that cannot be directly instantiated, but exist to be extended by concrete classes (the more usual kind of classes, the ones that can be instantiated).

One recommended approach to OO design is to never extend a concrete class[8]: if two concrete classes have so much in common that you're tempted to have one of them inherit from the other, proceed instead by making an *abstract* base class that subsumes all they do have in common, and have each concrete class extend that ABC. This approach avoids many of the subtle traps and pitfalls of inheritance.

Python offers rich support for ABCs, enough to make them a first-class part of Python's object model.

abc

The standard library module `abc` supplies metaclass `ABCMeta` and class `ABC` (subclassing `abc.ABC` makes `abc.ABCMeta` the metaclass, and has no other effect).

When you use `abc.ABCMeta` as the metaclass for any class *C*, this makes *C* an ABC, and supplies the class method `C.register`, callable with a single argument: that single argument can be any existing class (or built-in type) *X*.

Calling `C.register(X)` makes *X* a *virtual* subclass of *C*, meaning that `issubclass(X, C)` returns `True`, but *C* does not appear in *X*`.__mro__`, nor does *X* inherit any of *C*'s methods or other attributes.

Of course, it's also possible to have a new class *Y* inherit from *C* in the normal way, in which case *C* does appear in *Y*`.__mro__`, and *Y* inherits all of *C*'s methods, as usual in subclassing.

An ABC *C* can also optionally override class method `__subclasshook__`, which `issubclass(X, C)` calls with the single argument *X*, *X* being any class or type. When `C.__subclasshook__(X)` returns `True`, then so does `issubclass(X, C)`; when `C.__subclasshook__(X)` returns `False`, then so does `issubclass(X, C)`; when `C.__subclasshook__(X)` returns `NotImplemented`, then `issubclass(X,C)` proceeds in the usual way.

The module `abc` also supplies the decorator `abstractmethod` (and `abstractproperty`, but the latter is deprecated: just apply both the `abstractmethod` and `property` decorators to get the same effect). Abstract methods and properties can have implementations (available to subclasses via the `super` built-in)—however, the point of making methods and properties abstract is that you can instantiate any nonvirtual subclass *X* of an ABC *C* only if *X* overrides every abstract property and method of *C*.

ABCs in the collections module

`collections` supplies many ABCs. The ABCs are in `collections.abc` (for backward compatibility, they were also accessible in the `collections` module, but these compatibility imports were deprecated in Python 3.3; in Python 3.10, they are now removed).

Some ABCs characterize any class defining or inheriting a specific abstract method, as listed in Table 4-3:

Table 4-3.

Callable	Any class with `__call__`
Container	Any class with `__contains__`
Hashable	Any class with `__hash__`
	Any class with `__iter__`

Iterable	
Sized	Any class with `__len__`

The other ABCs in `collections.abc` extend one or more of the preceding ones, add more abstract methods, and supply *mixin* methods implemented in terms of the abstract methods (when you extend any ABC in a concrete class, you must override the abstract methods; you can optionally override some or all of the mixin methods, if that helps improve performance, but you don't have to—you can just inherit them, if this results in performance that's sufficient for your purposes).

Here is the set of ABCs directly extending the preceding ones:

ABC	Extends	Abstract methods	Mixin methods
Iterator	`Iterable`	`__next__`	`__iter__`
Mapping	`Container`, `Iterable`, `Sized`	`__getitem__`, `__iter__`, `__len__`	`__contains__`, `__eq__`, `__ne__`, `get`, `items`, `keys`, `values`
MappingView	`Sized`		`__len__`
Sequence	`Container`, `Iterable`, `Sized`	`__getitem__`, `__len__`	`__contains__`, `__iter__`, `__reversed__`, `count`, `index`
Set		`__contains__`,	`__and__`, `__eq__`, `__ge__`, `__gt__`, `__le__`, `__lt__`, `__ne__`, `__or__`,

	Container, Iterable, Sized	__iter, __len__	__sub__, __xor__, isdisjoint

And lastly, the set of ABCs further extending the previous ones:

ABC	Extends	Abstract methods	Mixin methods
ItemsView	MappingView, Set		__contains__, __iter__
KeysView	MappingView, Set		__contains__, __iter__
MutableMapping	Mapping	__delitem__, __getitem__, __iter__, __len_, __setitem__	Mapping's methods, plus clear, pop, popitem, setdefault, update
MutableSequence	Sequence	__delitem__, __getitem__, __len__, __setitem__, insert	Sequence's methods, plus __iadd__, append, extend, pop, remove, reverse
MutableSet	Set	__contains__, __iter, __len__, add, discard	Set's methods, plus __iand__, __ior__, __isub__, __ixor__, clear, pop, remove
ValuesView	MappingView		__contains__, __iter__

See the online docs for further details and usage examples.

The numbers module

`numbers` supplies a hierarchy (also known as a *tower*) of ABCs representing various kinds of numbers. `numbers` supplies the following ABCs:

Number	The root of the hierarchy: numbers of *any* kind (need not support any given operation)
Complex	Extends `Number`; must support (via special methods) conversions to `complex` and `bool`, `+`, `-`, `*`, `/`, `==`, `!=`, `abs()`; and, directly, the method `conjugate()` and properties `real` and `imag`
Real	Extends `Complex`; additionally, must support (via special methods) conversion to `float`, `math.trunc()`, `round()`, `math.floor()`, `math.ceil()`, `divmod()`, `//`, `%`, `<`, `<=`, `>`, `>=`
Rational	Extends `Real`; additionally, must support the properties `numerator` and `denominator`
Integral	Extends `Rational`; additionally, must support (via special methods) conversion to `int`, `**`, and bitwise operations `<<`, `>>`, `&`, `^`, `|`, `~`

See the online docs for notes on implementing your own numeric types.

Special Methods for Numeric Objects

An instance may support numeric operations by means of many special methods. Some classes that are not numbers also support some of the special methods in Table 4-4 in order to overload operators such as `+` and `*`. In particular, sequences should have special methods `__add__`, `__mul__`, `__radd__`, and `__rmul__`, as mentioned in “Sequences”.

Table 4-4.

__abs__, __invert__, __neg__, __pos__	`__abs__(self)` `__invert__(self)` `__neg__(self)` `__pos__(self)` The unary operators `abs(x)`, `~x`, `-x`, and `+x`, respectively, call these methods.

__add__, __mod__, __mul___, __sub__	`__add__ (self, other) __mod__(self,other) __mul__(self, other) __sub__(self,other)` The operators `x+y`, `x%y`, `x*y`, and `x-y`, and `x/y`, respectively, call these methods, usually for arithmetic computations.
__floordiv__, __truediv__	`__floordiv__(self,other)` `__truediv__(self,other)` The operators `x/y` and `x//y` call these methods, usually for arithmetic divisions.
__ matmul __	`__matmul__(self,other)` The operator `x@y` calls this method, usually for matrix multiplication.
__and__, __lshift__, __or__, __rshift__, __xor__	`__and__(self,other) __lshift__(self,other) __or__(self,other) __rshift_(self,other) __xor__(self,other)` The operators `x&y, x<<y, x\y, x>>y`, and `x^y`, respectively, call these methods, usually for bitwise operations.
__complex__, __float__, __int__	`__complex__(self) __float__(self) __int__(self)` The built-in types `complex(x), float(x), int(x)` respectively, call these methods.
__ divmod __	**__divmod__(self**,`other`) The built-in function `divmod(x,y)` calls `x.__divmod__(y)`. `__divmod__` should return a pair `(quotient,remainder)` equal to `(x//y,x%y)`.
__iadd__, __ifloordiv__, __imod__, __imul__, __isub__, __itruediv__, __imatmul__	`__iadd__(self,other) __ifloordiv__(self,other) __imod__(self,other) __imul__(self,other) __isub__(self,other) __itruediv__(self,other)` __imatmul__(self, other) The augmented assignments `x+=y`, `x//=y`, `x%=y`, `x*=y`, `x-=y`, `x/=y`, and `x@=y`, respectively, call these methods. Each method should modify `x` in place and return `self`. Define these methods when `x` is mutable (i.e., when `x` can change in place).
__iand__, __ilshift__, __ior__, __irshift__, __ixor__	`__iand_(self,other) __ilshift_(self,other) __ior__(self,other) __irshift__(self,other) __ixor__(self,other)` The augmented assignments `x&=y`, `x<<=y`, `x\=y`, `x>>=y`, and `x^=y`, respectively, call these methods. Each method should modify `x` in place and return `self`.

__index__	`__index__(self)` Like `__int__`, but meant to be supplied only by types that are alternative implementations of integers (in other words, all of the type's instances can be exactly mapped into integers). For example, out of all built-in types, only `int` supplies `__index__`; `float` and `str` don't, although they do supply `__int__`. Sequence indexing and slicing internally use `__index__` to get the needed integer indices.
__ipow__	`__ipow__(self,`*`other`*`)` The augmented assignment *`x**=y`* calls *`x`*`.__ipow__(`*`y`*`).__ipow__` should modify *`x`* in place and return `self`.
__pow__	`__pow__(self,`*`other`*`[,`*`modulo`*`])` *`x**y`* and `pow(`*`x`*`,`*`y`*`)` both call *`x`*`.__pow__(`*`y`*`)`, while `pow(`*`x`*`,`*`y`*`,`*`z`*`)` calls *`x`*`.__pow__(`*`y`*`,`*`z`*`)`. *`x`*`.__pow__(`*`y`*`,`*`z`*`)` should return a value equal to the expression *`x`*`.__pow__(`*`y`*`)%`*`z`*.
__radd__, **__rmod__**, **__rmul__**, **__rsub__**, **__rmatmul__**	`__radd__(self,`*`other`*`)` `__rmod__(self,`*`other`*`)` `__rmul__(self,`*`other`*`)` `__rsub__(self,`*`other`*`)` __rmatmul__(self, other) The operators *`y+x`*, *`y/x`*, *`y%x`*, *`y*x`*, *`y-x`*, and *`y@x`*, respectively, call these methods on *`x`* when *`y`* doesn't have a needed method `__add__`, `__truediv__`, and so on, or when that method returns `NotImplemented`.
__rand__, **__rlshift__**, **__ror__**, **__rrshift__**, **__rxor__**	`__rand__(self,`*`other`*`)` `__rlshift__(self,`*`other`*`)` `__ror__(self,`*`other`*`)` `__rrshift__ (self,`*`other`*`)` `__rxor__(self,`*`other`*`)` The operators *`y&x`*, *`y<<x`*, *`y | x`*, *`y>>x`*, and *`x^y`*, respectively, call these methods on *`x`* when *`y`* doesn't have a needed method `__and__`, `__lshift__`, and so on, or when that method returns `NotImplemented`.
__rdivmod__	`__rdivmod_(self,`*`other`*`)` The built-in function `divmod(`*`y`*`,`*`x`*`)` calls *`x`*`.__rdivmod__(`*`y`*`)` when *`y`* doesn't have `__divmod__`, or when that method returns `NotImplemented`. `__rdivmod__` should return a pair (*`remainder`*, *`quotient`*).
__rpow__	`__rpow__(self,`*`other`*`)` *`y**x`* and `pow(`*`y`*`,`*`x`*`)` call *`x`*`.__rpow__(`*`y`*`)` when *`y`* doesn't have `__pow__`, or when that method returns `NotImplemented`. There is no three-argument form in this case.

Decorators

In Python, you often use so-called *higher-order functions*: callables that accept a function as an argument and return a function as their result. For example, descriptor types such as `staticmethod` and `classmethod`, covered in "Class-Level Methods", can be used, within class bodies, as:

```
def f(cls, ...):
  ...definition of f snipped...
f = classmethod(f)
```

However, having the call to `classmethod` textually *after* the `def` statement hurts code readability: while reading *`f`*'s definition, the reader of the code is not yet aware that *`f`* is going to become a class method rather than an instance method. The code is more readable if the mention of `classmethod` comes *before*, not *after*, the `def`. For this purpose, use the syntax form known as *decoration*:

```
@classmethod
def f(cls, ...):
  ...definition of f snipped...
```

The decorator must be immediately followed by a `def` statement and means that *`f`* `= classmethod(`*`f`*`)` executes right after the `def` statement (for whatever name *`f`* the `def` defines). More generally, `@`*`expression`* evaluates the expression (which must be a name, possibly qualified, or a call) and binds the result to an internal temporary name (say, `__`*`aux`*); any decorator must be immediately followed by a `def` (or `class`) statement, and means that *`f`* `= __`*`aux`*`(`*`f`*`)` executes right after the `def` or `class` (for whatever name *`f`* the `def` or `class` defines). The object bound to `__`*`aux`* is known as a *decorator*, and it's said to *decorate* function or class *`f`*.

Decorators are a handy shorthand for some higher-order functions. You may apply decorators to any `def` or `class` statement, not just in class bodies. You may code custom decorators, which are just higher-order functions

accepting a function or class object as an argument and returning a function or class object as the result. For example, here is a simple example decorator that does not modify the function it decorates, but rather prints the function's docstring to standard output at function-definition time:

```
def showdoc(f):
    if f.__doc__:
        print(f'{f.__name__}: {f.__doc__}')
    else:
        print(f'{f.__name__}: No docstring!')
    return f
@showdoc
def f1():
    """a docstring"""  # prints: f1: a docstring
@showdoc
def f2():
    pass  # prints: f2: No docstring!
```

The standard library module `functools` offers a handy decorator, `wraps`, to enhance decorators built by the common "wrapping" idiom:

```
import functools
def announce(f):
    @functools.wraps(f)
    def wrap(*a, **k):
        print(f'Calling {f.__name__}')
        return f(*a, **k)
    return wrap
```

Decorating a function *`f`* with `@`*`announce`* causes a line announcing the call to be printed before each call to *`f`*. Thanks to the `functools.wraps(`*`f`*`)` decorator, the wrapper adopts the name and docstring of the wrappee: this is useful, for example, when calling the built-in `help` on such a decorated function.

Metaclasses

Any object, even a class object, has a type. In Python, types and classes are also first-class objects. The type of a class object is also known as the class's *metaclass*.[9] An object's behavior is mostly determined by the type of

the object. This also holds for classes: a class's behavior is mostly determined by the class's metaclass. Metaclasses are an advanced subject, and you may want to skip the rest of this section. However, fully grasping metaclasses can lead you to a deeper understanding of Python; very occasionally, it can be useful to define your own custom metaclasses.

Alternatives to Custom Metaclasses for Simple Class Customization

While a custom metaclass lets you tweak classes' behaviors in pretty much any way you want, it's often possible to achieve some customizations more simply than by coding a custom metaclass.

When a class *C* has or inherits a class method `__init_subclass__`, Python calls that method whenever you subclass *C*, passing the newly-built subclass as the only positional argument. `__init_subclass__` can also have named parameters, in which case Python passes corresponding named arguments found on the class statement that performs the subclassing. As a purely-illustrative example:

```
>>> class C:
...     def __init_subclass__(cls, foo=None, **kw):
...         print(cls, kw)
...         cls.say_foo = staticmethod(lambda: f'*{foo}*')
...         super().__init_subclass__(**kw)
...
>>> class D(C, foo='bar'):
...     pass
...
<class '__main__.D'> {}
>>> D.say_foo()
'*bar*'
```

The code in `__init_subclass__` can alter *cls* in any applicable, post-class-creation way; essentially, it works like a class decorator which Python automatically applies to any subclass of *C*.

Another special method used for customization is **`__set_name__`**, which lets you ensure that instances of descriptors added as class attributes know

what class you're adding them to, and under what name. Specifically, when the class of a class attribute *ca* has method **__set_name__**, at the end of the class statement adding *ca* to class *C* with name *n*, Python calls *ca*.**__set_name__**(*C*, *n*). For example:

```
>>> class Attrib:
...     def __set_name__(self, cls, name):
...         print(f'Attribute {name!r} added to {cls}')
...
>>> class AClass:
...     some_name = Attrib()
...
Attribute 'some_name' added to <class '__main__.AClass'>
>>>
```

How Python Determines a Class's Metaclass

The `class` statement accepts optional named arguments (after the bases, if any). The most important named argument is `metaclass`, which, if present, identifies the new class's metaclass. Other named arguments are allowed only if a non-`type` metaclass is present, and then they are passed on to the `__prepare__` method of the metaclass (it's entirely up to said method `__prepare__` to make use of such named arguments)[10]. When the named argument `metaclass` is absent, Python determines the metaclass by inheritance; for classes with no explicitly specified bases, the metaclass defaults to `type`.

A metaclass has an optional method `__prepare__`, which Python calls as soon as it determines the metaclass, as follows:

```
class MC:
    def __prepare__(classname, *classbases, **kwargs):
        return {}
  ...rest of MC snipped...
class X(onebase, another, metaclass=MC, foo='bar'):
  ...body of X snipped...
```

Here, the call is equivalent to *MC*.`__prepare__('X'`, *onebase*, *another*, *foo*=`'bar'`). `__prepare__`, if present, must return a

mapping (usually just a dictionary), which Python uses as the *d* in which it executes the class body. If `__prepare__` is absent, Python uses a dictionary as *d*.

How a Metaclass Creates a Class

Having determined *M*, Python calls *M* with three arguments: the class name (a string), the tuple of base classes *t*, and the dictionary (or other mapping resulting from `__prepare__`) *d* in which the class body just finished executing. The call returns the class object *C*, which Python then binds to the class name, completing the execution of the `class` statement. Note that this is in fact an instantiation of type *M*, so the call to *M* executes *M*`.__init__(`C, *namestring*, *t*, *d*`)`, where *C* is the return value of *M*`.__new__(`*M, namestring, t, d*`)`, just as in any other instantiation.

After Python creates class object *C*, the relationship between class *C* and its type (`type(`*C*`)`, normally *M*) is the same as that between any object and its type. For example, when you call the class object *C* (to create an instance of *C*), *M*`.__call__` executes with class object *C* as the first argument.

Note the benefit, in this context, of the approach described in "Per-Instance Methods", whereby special methods are looked up only on the class, not on the instance. Calling *C* to instantiate it must execute the metaclass's *M*`.__call__`, whether or not *C* has a per-instance attribute (method) `__call__` (i.e., independently of whether *instances* of *C* are or aren't callable). The Python object model avoids having to make the relationship between a class and its metaclass an ad-hoc special case. Avoiding ad-hoc special cases is a key to Python's power: Python has few, simple, general rules, and applies them consistently.

Defining and using your own metaclasses

It's easy to define custom metaclasses: inherit from `type` and override some of its methods. You can also perform most of these tasks with `__new__`, `__init__`, `__getattribute__`, and so on, without

involving metaclasses. However, a custom metaclass can be faster, since special processing is done only at class creation time, which is a rare operation. A custom metaclass lets you define a whole category of classes in a framework that magically acquire whatever interesting behavior you've coded, quite independently of what special methods the classes themselves may choose to define.

A good alternative, to alter a specific class in an explicit way, is often to use a class decorator, as mentioned in "Decorators". However, decorators are not inherited, so the decorator must be explicitly applied to each class of interest[11]. Metaclasses, on the other hand, *are* inherited; in fact, when you define a custom metaclass *M*, it's usual to also define an otherwise-empty class *C* with metaclass *M*, so that other classes requiring metaclass *M* can just inherit from *C*.

Some behavior of class objects can be customized only in metaclasses. The following example shows how to use a metaclass to change the string format of class objects:

```
class MyMeta(type):
    def __str_(cls):
        return f'Beautiful class {cls.__name__!r}'
class MyClass(metaclass=MyMeta):
    pass
x = MyClass()
print(type(x))      # prints: Beautiful class 'MyClass'
```

A substantial custom metaclass example

Suppose that, programming in Python, we miss *C*'s `struct` type: an object that is just a bunch of data attributes, in order, with fixed names (data classes, covered in "Data Classes", fully address this requirement, which makes this example a purely illustrative one). Python lets us easily define a generic *Bunch* class, apart from the fixed order and names:

```
class SimpleBunch(object):
    def __init__(self, **fields):
        self.__dict__ = fields
p = SimpleBunch(x=2.3, y=4.5)
```

```
print(p)        # prints: <__main__.SimpleBunch object at
0x00AE8B10>
```

A custom metaclass can exploit the fact that attribute names are fixed at class creation time. The code shown in Example 4-1 defines a metaclass, *MetaBunch*, and a class, *Bunch*, to let us write code like:

```
class Point(Bunch):
    """ A Point has x and y coordinates, defaulting to 0.0,
        and a color, defaulting to 'gray'-and nothing more,
        except what Python and the metaclass conspire to add,
        such as __init__ and __repr__
    """
    x = 0.0
    y = 0.0
    color = 'gray'
# example uses of class Point
q = Point()
print(q)                    # prints: Point()
p = Point(x=1.2, y=3.4)
print(p)                    # prints: Point(x=1.2, y=3.4)
```

In this code, the `print` calls emit readable string representations of our *Point* instances. *Point* instances are quite memory-lean, and their performance is basically the same as for instances of the simple class *SimpleBunch* in the previous example (there is no extra overhead due to implicit calls to special methods). Example 4-1 is quite substantial, and following all its details requires understanding aspects of Python covered later in this book, such as strings (Chapter "Strings and Things") and module warnings ("The warnings Module"). The identifier *mcl* used in Example 4-1 stands for "metaclass," clearer in this special advanced case than the habitual case of *cls* standing for "class."

Example 4-1. The MetaBunch metaclass

```
import collections
import warnings
class MetaBunch(type):
    """
    Metaclass for new and improved "Bunch": implicitly defines
    __slots__, __init__ and __repr__ from variables bound in
    class scope.
    A class statement for an instance of MetaBunch (i.e., for a
```

```
    class whose metaclass is MetaBunch) must define only
    class-scope data attributes (and possibly special methods, but
    NOT __init__ and __repr__).  MetaBunch removes the data
    attributes from class scope, snuggles them instead as items in
    a class-scope dict named __dflts__, and puts in the class a
    __slots__ with those attributes' names, an __init__ that takes
    as optional named arguments each of them (using the values in
    __dflts__ as defaults for missing ones), and a __repr__ that
    shows the repr of each attribute that differs from its default
    value (the output of __repr__ can be passed to __eval__ to make
    an equal instance, as per usual convention in the matter, if
    each non-default-valued attribute respects the convention too).
    The order of data attributes remains the same as in the class
body.
    """
    def __new__(mcl, classname, bases, classdict):
        """ Everything needs to be done in __new__, since
            type.__new__ is where __slots__ are taken into account.
        """
        # define as local functions the __init__ and __repr__ that
        # we'll use in the new class
        def __init__(self, **kw):
            """ __init__ is simple : first, set attributes without
                explicit values to their defaults; then, set those
                explicitly passed in kw.
            """
            for k in self.__dflts__:
                if not k in kw:
                    setattr(self, k, self.__dflts__[k])
            for k in kw:
                setattr(self, k, kw[k])
        def __repr__(self):
            """ __repr__ is "clever": shows only attributes that
                differ from default values, for compactness.
            """
            rep = [f'{k}={getattr(self, k)!r}'
                   for k in self.__dflts__
                   if getattr(self, k) != self.__dflts__[k]
                  ]
            return f'{classname}({', '.join(rep)})'
        # build the newdict that we'll use as class-dict for the
        # new class
        newdict = { '__slots__':[],
            '__dflts__': {},
            '__init__' :__init__, '__repr__' :__repr__, }
        for k in classdict:
            if k.startswith('__') and k.endswith('__'):
                # dunder methods: copy to newdict, or warn
                # about conflicts
                if k in newdict:
```

```
                    warnings.warn(f'Cannot set attr {k!r} in bunch-
class {classname!r}')
                else:
                    newdict[k] = classdict[k]
            else:
                # class variables, store name in __slots__, and
                # name and value as an item in __dflts__
                newdict['__slots__'].append(k)
                newdict['__dflts__'][k] = classdict[k]
        # finally delegate the rest of the work to type.__new__
        return super().__new__(mcl, classname, bases, newdict)

class Bunch(metaclass=MetaBunch):
    """ For convenience: inheriting from Bunch can be used to get
        the new metaclass (same as defining metaclass= yourself).
    """
    pass
```

Data Classes

As the previous *Bunch* class exemplified, a class whose instances are just a bunch of named data items is a great convenience. Python's standard library covers that with the `dataclasses` module.

The main feature of the `dataclasses` module you'll be using is the `dataclass` function: a decorator you apply to any class whose instances you want to be just such a bunch of named data items. As a typical example, consider:

```
import dataclasses
@dataclasses.dataclass
class Point:
    x: float
    y: float
```

Now you can call, say, *`pt = Point(0.5, 0.5)`* and get a variable with attributes *`pt`*`.x` and *`pt`*`.y`, each equal to `0.5`. By default, the `dataclass` decorator has imbued **class Point** with an `__init__` method accepting initial floating-point values for attributes `x` and `y`, and a `__repr__` method ready to appropriately display any instance of the class:

```
>>> pt
Point(x=0.5, y=0.5)
>>>
```

Function `dataclass` takes many optional named parameters to let you tweak details of the class it decorates. The parameters you may be explicitly using most often are:

Parameter name	Default value and resulting behavior
init	True When `True`, it generates an `__init__` method (unless the class defines one).
repr	True When `True`, it generates a `__repr__` method (unless the class defines one).
eq	True When `True`, it generates an `__eq__` method (unless the class defines one).
order	False When `True`, it generates order-comparison special methods (`__le__`, `__lt__`, and so on) unles the class defines them.
frozen	False When `True`, it makes each instance of the class read-only (not allowing rebinding or deletion of attributes).
kw_only ||**3.10**++||	False When `True`, it forces arguments to `__init__` to be named, not positional ones.
slots ||**3.10**++||	False When `True`, it adds the appropriate `__slots__` attribute to the class (saving some amount of memory for each instance, but disallowing the addition of other, arbitrary attributes to class instances).

The decorator also adds to the class a `__hash__` method (allowing instances to be keys in a dict and members of a set) when that is safe (typically, when you set `frozen` to `True`). You may force the addition of

`__hash__` even when that's not necessarily safe, but we earnestly recommend that you don't; if you insist, check the online docs for all details of how to do so.

If you need to tweak each instance of a dataclass after the automatically-generated `__init__` method has done the core work of assigning each instance attribute, define a method called `__post_init__`, and the decorator will ensure it is called right after `__init__` is done.

Say you wish to add an attribute to `Point` to capture the time when the point was created. This could be added as an attribute assigned in `__post_init__`. Add the attribute *`create_time`* to the members defined for `Point`, as type `float` with a default value of 0, and then add an implementation for `__post_init__`:

```
def __post_init__(self):
    self.create_time = time.time()
```

Now if you create the variable *`pt`* `= Point(0.5, 0.5)`, printing it out will display with the creation timestamp, similar to the following:

```
>>> pt
Point(x=0.5, y=0.5, create_time=1645122864.3553088)
```

Like regular classes, dataclasses can also support additional methods and properties, such as this method that computes distance between two `Points`, and a property to return distance from a `Point` at the origin:

```
def distance_from(self, other):
    dx, dy = self.x - other.x, self.y - other.y
    return math.hypot(dx, dy)
@property
def distance_from_origin(self):
    return self.distance_from(Point(0, 0))
>>> pt.distance_from(Point(-1, -1))
2.1213203435596424
>>> pt.distance_from_origin
0.7071067811865476
```

The `dataclasses` module also supplies functions `asdict` and `astuple`, each taking a `dataclass` instance as the first argument and returning, respectively, a `dict` and a `tuple` with the class's fields. Furthermore, the module supplies a function `field` which you may use to customize the treatment of some of a dataclass's fields (i.e., instance attributes), and several other specialized functions and classes needed only for very advanced, esoteric purposes; to learn all about them, check the online docs.

Enumerated types (Enums)

When programming, you often want to create a set of related values that catalog or *enumerate* the possible values for a particular property or program setting[12]. These values could be a list of terminal colors, logging levels, process states, playing card suits, clothing sizes... An *enumerated type* (which we call just "an enum" in the following, just as is done in typical programmer jargon) is a type that defines a group of such values, with symbolic names that you can use as typed global constants. Python provides the `Enum` class and related subclasses in the `enum` module for defining enumerated types.

The reason to define an enum is to give your code a set of symbolic constants that represent the values in the enumeration. In the absence of enums, constants might be defined as `int`s, as in this code:

```
# colors
RED = 1
GREEN = 2
BLUE = 3
# sizes
XS = 1
S = 2
M = 3
L = 4
XL = 5
```

However, in this design, there is no mechanism to warn against nonsense expressions like `RED > XS` or `L*BLUE`, since they are all just `int`s.

There is also no logical grouping of the colors or sizes.

Instead use an `Enum` subclass to define these values:

```
from enum import Enum, auto
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

class Size(Enum):
    XS = auto()
    S = auto()
    M = auto()
    L = auto()
    XL = auto()
```

Now code like `Color.RED > Size.S` is visually incorrect, and, at run time, raises a Python `TypeError`. Using `auto()` auto-assigns incrementing `int` values - in most cases, the actual values assigned to enum members are not meaningful, as long as they are distinct.

Calling `Enum()` Creates a Class, Not an Instance

Surprisingly, when you call `enum.Enum()`, it doesn't return a newly built *instance*, but rather a newly built *subclass*. So, the above snippet is equivalent to:

```
from enum import Enum

Color = Enum('Color', ('RED', 'GREEN', 'BLUE'))

Size = Enum('Size', 'XS S M L XL')
```

When you *call* `Enum` (rather than explicitly subclassing it in a class statement), the first argument is the name of the subclass you're building; the second argument gives all the names of that subclass's members, either as a sequence of strings or as a single whitespace-separated (or comma-separated) string.

We recommend that you define **Enum** subclasses using class inheritance instead of this abbreviated form. The `class` form is more visually explicit, so it is easier to see if a member is missing, misspelled, or later added.

The values within an `Enum` are called its *members*. It is conventional to use all uppercase characters to name `Enum` members, treating them much as though they were manifest constants. Typical uses of the members of an `Enum` are assignment and identity checking.

```
while process_state is ProcessState.RUNNING:
    # running process code goes here
    if processing_completed():
        process_state = ProcessState.IDLE
```

You can obtain all members of an `Enum` by iterating over the `Enum` class itself, or from the class's `__members__` attribute. `Enum` members are all global singletons; so, comparison with `is` and `is not` is preferred over == or !=.

The `enum` module contains several classes[13] that support different forms of enums.

Enum	Basic enumeration class; member values can be any Python object, typically `int`s or `str`s, but do not support `int` or `str` methods. Useful for defining enumerated types whose members are an unordered group.
Flag	Enumerations that can be combined using `|`, `&`, `^`, and ~ operators; member values should be defined as `int`s to support these bitwise operations, but no ordering is assumed. `Flag` members with a `0` value are considered false; non-zero value members are true. Useful when values are created or tested using bitwise operations, such as file permissions. To support bitwise operations, values are usually given as powers of 2 (1, 2, 4, 8, etc.).
IntEnum	Equivalent to `class IntEnum(int, Enum)`; member values are `int`s, and support all `int` operations, including ordering. Useful when order among values is significant, such as when defining logging levels.
IntFlag	Equivalent to `class IntFlag(int, Flag)`; member values are `int`s, and support all `int` operations, including order comparisons.

The `enum` module also defines support functions, particularly:

auto	Used to auto-increment member values to use when defining members. Typically, starts at 1 and increments by 1; for `Flag`, increments are in powers of 2.
unique	Class decorator used to enforce that the values defined for members are all different from each other.

The following example shows how to define a `Flag` to work with the file permissions of the `st_mode` attribute returned from calling `os.stat()` or `Path.stat()`.

```
from enum import Flag
import stat
class Permission(Flag):
    EXEC_OTH = stat.S_IXOTH
    WRITE_OTH = stat.S_IWOTH
    READ_OTH = stat.S_IROTH
    EXEC_GRP = stat.S_IXGRP
    WRITE_GRP = stat.S_IWGRP
    READ_GRP = stat.S_IRGRP
    EXEC_USR = stat.S_IXUSR
```

```
    WRITE_USR = stat.S_IWUSR
    READ_USR = stat.S_IRUSR
    @classmethod
    def from_mode(cls, mode):
        return cls(mode & 0o777)
from pathlib import Path
cur_dir = Path.cwd()
dir_perm = Permission.from_mode(cur_dir.stat().st_mode)
if dir_perm & Permission.READ_OTH:
    # directory is readable
# raises TypeError
print(Permission.READ_USR > Permission.READ_OTH)
```

Enums can add readability and type integrity to your code, in place of using arbitrary ints or strings. More details on the classes and methods of the `enum` module can be found in the Python docs.

---

1 When that's the case, it's also OK to have other named arguments after metaclass= – such arguments, if any, are passed on to the metaclass.

2 AKA "magic methods", or "dunder (**d**ouble **under**score) methods," e.g., "dunder init" for __init__

3 That need arises because __init__, on any subclass of *Singleton* that defines it, repeatedly executes, each time you instantiate the subclass, on the only instance that exists for each subclass of *Singleton*.

4 other OO languages, like Modula-3, similarly require explicit mentions of self

5 To complete the usually-truncated famous quote: "except of course for the problem of too many indirections."

6 Third-party extensions can also define types of containers that are not sequences, not mappings, and not sets.

7 Lower-bound included; upper-bound excluded—as always, the norm for Python.

8 See, for example, this essay.

9 Strictly speaking, the type of a class C could be said to be the metaclass only of instances of C rather than of C itself, but this subtle distinction is rarely, if ever, observed in practice.

10 or when a base class has __init_subclass__, in which case the named arguments are passed to that method, as covered in "Alternatives to custom metaclasses for simple class customization".

11 __init_subclass__, covered in "Alternatives to custom metaclasses for simple class customization", works much like an "inherited decorator", so it's often an alternative to a custom metaclass.

12 don't confuse this concept with the unrelated `enumerate` built-in function, covered in Chapter "Core Built-ins and Standard Library Modules," which generates (number, item) pairs from an iterable.

13 `enum` specialized metaclass behaves so differently from the usual `type` metaclass that it's worth pointing out all the differences between `enum.Enum` and ordinary classes, as section How are Enums different? of Python's online documentation does.

Chapter 5. 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 5-1. 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 6. 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 6-1. 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 7. Core Built-ins and Standard Library Modules

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 7th 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*.

The term *built-in* has more than one meaning in Python. In most contexts, *built-in* means an object directly accessible to Python code without an `import` statement. Section "Python built-ins" shows Python's mechanism to allow this direct access. Built-in types in Python include numbers, sequences, dictionaries, sets, functions (all covered in Chapter 3), classes (covered in "Python Classes"), standard exception classes (covered in "Exception Objects"), and modules (covered in "Module Objects"). "The io Module" covers the `file` type, and "Internal Types" covers some other built-in types intrinsic to Python's internal operation. This chapter provides additional coverage of built-in core types (in "Built-in Types") and covers built-in functions available in the module `builtins` in "Built-in Functions".

Some modules are "built-in" because they're in the Python standard library (though it takes an `import` statement to use them). This differs from optional add-on modules, also known as Python *extensions*.

This chapter covers some *built-in core* modules: namely, the modules named `sys`, `copy`, `collections`, `functools`, `heapq`, `argparse`,

and `itertools`. For each of these modules' name *x*, you'll find it covered in the section titled "The *x* module."

Chapter 8 "Strings and Things" covers some string-related core built-in modules (`string`, `codecs`, and `unicodedata`), with the same section-name convention. Chapter 9 covers `re` in "Regular Expressions and the `re` Module".

Built-in Types

Table 7-1 covers Python's core built-in types, such as `int`, `float`, `dict`, and many others. More details about many of these types, and about operations on their instances, are found throughout Chapter 3. In this section, by "number" we mean, specifically, "non-complex number."

Table 7-1. Core built-in types

bool	`bool(`*x*`=False)` Returns `False` if *x* evaluates as false; returns `True` if *x* evaluates as true. (See "Boolean Values".) `bool` extends `int`: built-in names `False` and `True` refer to the only two instances of `bool`. These instances are also `ints`, equal to `0` and `1`, respectively, but `str(True)` is `'True'`, `str(False)` is `'False'`.
bytearray	`bytearray(`*x*`=b''[, `*codec*`[, `*errors*`]])` A mutable sequence of *bytes* (ints with values from `0` to `255`), supporting the usual methods of mutable sequences, plus the methods of `str` . When *x* is a `str`, you must also pass `codec` and may pass `errors`; the result is just like calling `bytearray(`*x*`.encode(codec, errors))`. When *x* is an `int`, it must be `>=0`: the resulting instance has a length of *x*, each item initialized to `0`. When *x* conforms to the buffer protocol, the read-only buffer of bytes from *x* initializes the instance. Otherwise, *x* must be an iterable yielding `ints >=0` and `<256`; e.g., `bytearray([1,2,3,4])==bytearray(b'\x01\x02\x03\x04')`.
bytes	`bytes(`*x*`=b''[, `*codec*`[, `*errors*`]])` An immutable sequence of *bytes*, with the same non-mutating methods and the same initialization behavior as `bytearray`.
complex	`complex(`*real*`=0,`*imag*`=0)` Converts any number, or a suitable string, to a complex number. *imag* may be present only when *real* is a number, and in that case, it is the imaginary part of the resulting complex number. See also "Complex numbers".

dict	`dict(`*x*`={})`
	Returns a new dictionary with the same items as *x*. (We cover dictionaries in "Dictionaries".) When `x` is a `dict`, `dict(`*x*`)` returns a shallow copy of *x*, like *x*`.copy()`. Alternatively, *x* can be an iterable whose items are pairs (iterables with two items each). In this case, `dict(`*x*`)` returns a dictionary whose keys are the first items of each pair in *x*, and whose values are the corresponding second items. In other words, when *x* is iterable, *c* = `dict(`*x*`)` is equivalent to:
	`c = {}`
	`for key, value in x:`
	`  c[key] = value`
	You can call `dict` with named arguments, in addition to, or instead of, positional argument *x*. Each named argument becomes an item in the dictionary, with the name as the key: it might overwrite an item from *x*.

float	`float(`*x*`=0.0)` Converts any number, or a suitable string, to a floating-point number. See "Floating-point numbers".
frozenset	`frozenset(`*seq*`=())` Returns a new frozen (i.e., immutable) set object with the same items as iterable *seq*. When *seq* is a frozen set, `frozenset(`*seq*`)` returns *seq* itself, just like *seq*`.copy()` does. See "Set Operations".
int	`int(`*x*`=0,`*radix*`=10)` Converts any number, or a suitable string, to an `int`. When *x* is a number, `int` truncates toward `0`, dropping any fractional part. *radix* may be present only when *x* is a string: then, *radix* is the conversion base, between `2` and `36`, with `10` as the default. *radix* can be explicitly passed as `0`: the base is then `2`, `8`, `10`, or `16`, depending on the form of string *x*, just like for integer literals, as covered in "Integer numbers".
list	`list(`*seq*`=())` Returns a new list object with the same items as iterable *seq*, in the same order. When *seq* is a list, `list(`*seq*`)` returns a shallow copy of *seq*, like *seq*`[:]`. See "Lists".
memoryview	`memoryview(`*x*`)` *x* must be an object supporting the buffer protocol (for example, an instance of `bytes`, `bytearray`, or `array`). `memoryview` returns an object *m* "viewing" exactly the same underlying memory as *x*, with items of *m*`.itemsize` bytes each; in the normal case in which *m* is "one-dimensional" (we don't cover "multi-dimensional" `memoryview` in this book), `len(`*m*`)` is the number of items. You can index *m* (returning `int`) or slice it (returning an instance of `memoryview` "viewing" the appropriate subset of the same underlying memory). *m* is mutable if *x* is (but you can't change *m*'s size, so, when you assign to a slice, it must be from a sequence of the same length as the slice). *m* is a sequence, thus iterable, and is hashable when *x* is and

		m.itemsize is one byte. *m* supplies several read-only attributes and methods; see the online docs for details. Two useful methods are *m*.tobytes() (returns *m*'s data as an instance of bytes) and *m*.tolist() (returns *m*'s data as a list of ints).
object		object() Returns a new instance of object, the most fundamental type in Python. Instances of type object have no functionality: the only use of such instances is as "sentinels"—i.e., objects comparing != to any distinct object.
	set	set(*seq*=()) Returns a new mutable set object with the same items as iterable object *seq*. When *seq* is a set, set(*seq*) returns a shallow copy of *seq*, like *seq*.copy(). See "Sets".
	slice	slice([*start*,]*stop*[, *step*]) Returns a slice object with the read-only attributes start, stop, and step bound to the respective argument values, each defaulting to None when missing. For positive indices, such a slice signifies the same indices as range(*start*, *stop*, *step*). Slicing syntax, *obj*[*start*:*stop*:*step*], passes a slice object as the argument to the __getitem__, __setitem__, or __delitem__ method of object *obj*. It is up to *obj*'s class to interpret the slice objects that its methods receive. See also "Container slicing".
	str	str(*obj*='') Returns a concise, readable string representation of *obj*. If *obj* is a string, str returns *obj*. See also repr in Table 7-2 and __str__ in Table 4-1.
	super	super() super(*cls*, *obj*) Returns a super-object of object *obj* (which must be an instance of class *cls* or of any subclass of *cls*), suitable for calling superclass methods. Instantiate this built-in type only within a method's code. See "Cooperative superclass method calling". You usually call super() without arguments, within a method, and Python determines the *cls* and *obj* by introspection.
	tuple	tuple(*seq*=()) Returns a tuple with the same items as iterable *seq*, in order. When *seq* is a tuple, tuple returns *seq* itself, like *seq*[:]. See "Tuples".
	type	type(*obj*) Returns the type object that is the type of *obj* (i.e., the most-derived, AKA *leafmost*, type of which *obj* is an instance). type(*x*) is the same as *x*.__class__ for any *x*. Avoid checking equality or identity of types: see "Type-Equality Checking: Avoid It!" below.

Type-Equality Checking: Avoid It!

Use `isinstance` (covered in Table 7-2), *not* equality comparison of types, to check whether an instance belongs to a particular class, in order to properly support inheritance. Checking `type(x)` for equality or identity to some other type object is known as *type equality checking*. Type-equality checking is inappropriate in production Python code, as it interferes with polymorphism. Typically, just try to use *x as if* it were of the type you expect, handling any problems with a `try/except` statement, as discussed in "Error-Checking Strategies"; this is known as *duck typing* (one of this book's authors is often credited with an early use of this term).

When you just *have* to type-check, usually for debugging purposes, use `isinstance` instead. In a broader sense, `isinstance(x, atype)` is also a form of type checking, but it is a lesser evil than `` `type(x) is atype` ``. `isinstance` accepts an *x* that is an instance of any subclass of *atype*, or an object that implements protocol *atype*, not just a **direct** instance of *atype* itself. In particular, `isinstance` is fine when you're checking for an ABC (abstract base class: see "Abstract Base Classes"); this newer idiom is also sometimes known as *goose typing*.

Built-in Functions

Table 7-2 covers Python functions (and some types that, in practice, are only used as if they were functions) in the module `builtins`, in alphabetical order. Built-ins' names are *not* keywords. You **can** bind, in local or global scope, an identifier that's a built-in name (although we recommend avoiding it; see the following warning!). Names bound in local or global scope override names bound in built-in scope: local and global names *hide* built-in ones. You can also rebind names in built-in scope, as covered in "Python built-ins".

Don’t Hide Built-ins

Avoid accidentally hiding built-ins: your code might need them later. It’s tempting to use, for your own variables, natural names such as `input`, `list`, or `filter`, but don’t do it: these are names of built-in Python types or functions. Unless you get into the habit of never hiding built-ins’ names with your own, sooner or later you’ll get mysterious bugs in your code caused by just such hiding occurring accidentally.

Most Built-ins Don’t Accept Named Arguments

Many built-in functions and types cannot be called with named arguments, only with positional ones. In Table 7.2, we mention cases in which this limitation does not hold.

__import__	`__import__(`*`module_name`*`[,`*`globals`*`[,`*`locals`*`[,`*`fromlist`*`]]])` Deprecated in modern Python; use, instead, `importlib.import_module`, covered in “Module Loading”.
abs	`abs(`*`x`*`)` Returns the absolute value of number *`x`*. When *`x`* is complex, `abs` returns the square root of *`x`*`.imag**2+`*`x`*`.real**2` (also known as the *magnitude* of the complex number). Otherwise, `abs` returns `-`*`x`* when *`x`* is `<0`, *`x`* when *`x`* is `>=0`. See also `__abs__`, `__invert__`, `__neg__`, `__pos__` in Table 4-4.

all	`all(`*`seq`*`)`
	`seq` is an iterable (often a *generator expression*; see “Generator expressions”). `all` returns `False` when any item of *`seq`* is false; otherwise , `all` returns `True`. Like operators `and` and `or`, covered in “Short-Circuiting Operators”, `all` stops evaluating, and returns a result, as soon as it knows the answer; in the case of `all`, this means it stops as soon as a false item is reached, but proceeds throughout *`seq`* if all of *`seq`*’s items are true. Here is a typical toy example of the use of `all`:
	`if all(x>0 for x in the_numbers):`
	`  print('all of the numbers are positive'`
	`else:`
	`  print('some of the numbers are not positive')`

When *seq* is empty, `all` returns `True`.

any

`any(`*seq*`)`
seq is an iterable (often a *generator expression*; see "Generator expressions"). `any` returns `True` if any item of *seq* is true; otherwise, `any` returns `False`. Like operators `and` and `or`, covered in "Short-Circuiting Operators", `any` stops evaluating, and returns a result, as soon as it knows the answer; in the case of `any`, this means it stops as soon as a true item is reached, but proceeds throughout *seq* if all of *seq*'s items are false. Here is a typical toy example of the use of `any`:

```
if any(x<0 for x in the_numbers):
  print('some of the numbers are negative'
else:
  print('none of the numbers are negative')
```

When *seq* is empty, `any` returns `False`.

ascii

`ascii(`*x*`)`
Like `repr`, but escapes non-ASCII characters in the string it returns; the result is usually similar to that of `repr`.

bin

`bin(`*x*`)`
Returns a binary string representation of integer *x*.

breakpoint

breakpoint()
Invokes the `pdb` Python debugger. Set `sys.breakpointhook` to a callable function if you want to invoke an alternate debugger.

callable

`callable(`*obj*`)`
Returns `True` when *obj* can be called; otherwise, `False`. An object can be called if it is a function, method, class, type, or an instance of a class with a `__call__` method. See also `__call__` in Table 4-1.

chr

`chr(`*code*`)`
Returns a string of length `1`, a single character corresponding to integer *code* in Unicode. See also `ord` in Table 7-2.

compile

`compile(`*string, filename, kind*`)`
Compiles a string and returns a code object usable by `exec` or `eval`. `compile` raises `SyntaxError` when *string* is not syntactically valid Python. When *string* is a multiline compound statement, the last character must be `'\n'`. *kind* must be `'eval'` when *string* is an expression and the result is meant for `eval`; otherwise, *kind* must be `'exec'`. *filename* must be a string, used only in error messages (if an error occurs). See also `eval` in Table 7-2, and "Compile and Code Objects".

`delattr(`*obj,name*`)`

delattr	Removes the attribute *name* from *obj*. `delattr(obj,'ident')` is like `del obj.ident`. If *obj* has an attribute named *name* just because its class has it (as is normally the case, for example, for *methods* of *obj*), you cannot delete that attribute from *obj* itself. You may be able to delete that attribute from the *class*, if the metaclass lets you. If you can delete the class attribute, *obj* ceases to have the attribute, and so does every other instance of that class.
dir	`dir([obj])` Called without arguments, `dir` returns a sorted list of all variable names that are bound in the current scope. `dir(obj)` returns a sorted list of names of attributes of *obj*, including ones coming from *obj*'s type or by inheritance. See also `vars` in Table 7-2.
divmod	`divmod(dividend, divisor)` Divides two numbers and returns a pair whose items are the quotient and remainder. See also `__divmod__` in Table 4-4.

enumerate	`enumerate(iterable,start=0)`
	Returns a new iterator whose items are pairs. For each such pair, the second item is the corresponding item in *iterable*, while the first item is an integer: *start, start+1, start+2...* For example, the following snippet loops on a list `L` of integers, changing `L` in place by halving every even value:
	for i, num **in** enumerate(L):
	`  if num % 2 == 0:`
	`  L[i] = num // 2`

eval	`eval(expr,[globals[,locals]])` Returns the result of an expression. *expr* may be a code object ready for evaluation, or a string; if a string, `eval` gets a code object by internally calling `compile(expr, '<string>', 'eval')`. `eval` evaluates the code object as an expression, using the *globals* and *locals* dictionaries as namespaces (when they're missing, `eval` uses the current namespace). `eval` doesn't execute statements: it only evaluates expressions. Nevertheless, `eval` is dangerous unless you know and trust that *expr* comes from a source that you are certain is safe. See also "Expressions", `ast.literal_eval` (covered in "Standard Input"), and "Dynamic Execution".
exec	`exec(statement,[globals[,locals]])` Like `eval`, but applies to any statement and returns `None`. `exec` is dangerous unless you know and trust that *statement* comes from a source that you are certain is safe. See also "Statements" and "Dynamic Execution".
filter	`filter(func,seq)` Returns an iterator of those items of *seq* for which *func* is true. *func* can be any callable object accepting a single argument, or `None`. *seq* can be any

	iterable. When *func* is callable, `filter` calls *func* on each item of *seq*, just like the following generator expression: (*item* **for** *item* **in** *seq* **if** *func*(*item*)) When *func* is `None`, `filter` tests for true items, just like: (*item* **for** *item* **in** *seq* **if** *item*)
format	`format(`*x*`,`*format_spec*`='')` Returns *x*`.__format__(`*format_spec*`)`. See Table 4-1.
getattr	`getattr(`*obj*`,`*name*`[,`*default*`])` Returns *obj*'s attribute named by string *name*. `getattr(`*obj*`, '`*ident*`')` is like *obj*`.`*ident*`.` When *default* is present and *name* is not found in *obj*, `getattr` returns *default* instead of raising `AttributeError`. See also "Object attributes and items" and "Attribute Reference Basics".
globals	`globals()` Returns the `__dict__` of the calling module (i.e., the dictionary used as the global namespace at the point of call). See also `locals` in Table 7-2.
hasattr	`hasattr(`*obj*`,`*name*`)` Returns `False` when *obj* has no attribute *name* (i.e., when `getattr(`*obj*`,`*name*`)` would raise `AttributeError`). Otherwise, `hasattr` returns `True`. See also "Attribute Reference Basics".
hash	`hash(`*obj*`)` Returns the hash value for *obj*. *obj* can be a dictionary key, or an item in a set, only if *obj* can be hashed. All objects that compare equal must have the same hash value, even if they are of different types. If the type of *obj* does not define equality comparison, `hash(`*obj*`)` normally returns `id(`*obj*`)`. See `id` below, and `__hash__` in Table 4-1.
hex	`hex(`*x*`)` Returns a hexadecimal string representation of integer *x*. See also `__hex__` in Table 4-4.
id	`id(`*obj*`)` Returns the integer value that denotes the identity of *obj*. The `id` of *obj* is unique and constant during *obj*'s lifetime (but may be reused at any later time after *obj* is garbage-collected, so don't rely on storing or checking `id` values). When a type or class does not define equality comparison, Python uses `id` to compare and hash instances. For any objects *x* and *y*, identity check *x* `is` *y* is the same as `id(`*x*`)==id(`*y*`)`, but more readable and better-performing.
input	`input(`*prompt*`='')` Writes *prompt* to standard output, reads a line from standard input, and returns the line (without `\n`) as a `str`. At end-of-file, `input` raises `EOFError`.

isinstance

`isinstance(obj,cls)`
Returns `True` when *`obj`* is an instance of class *`cls`* (or any subclass of *`cls`*, or implements protocol *`cls`*); otherwise, it returns `False`. *`cls`* can be a tuple whose items are classes: in this case, `isinstance` returns `True` if *`obj`* is an instance of any of the items of *`cls`*; otherwise, it returns `False`. See also "Abstract Base Classes".

issubclass

`issubclass(cls1,cls2)`
Returns `True` when *`cls1`* is a direct or indirect subclass of *`cls2`*, or defines all the elements of protocol *`cls2`*; otherwise, it returns `False`. *`cls1`* and *`cls2`* must be classes. *`cls2`* can also be a tuple whose items are classes. In this case, `issubclass` returns `True` when *`cls1`* is a direct or indirect subclass of any of the items of *`cls2`*, otherwise, it returns `False`. For any class *`C`*, `issubclass(C,C)` returns `True`.

iter

`iter(obj)`
`iter(func, sentinel)`
Creates and returns an *iterator*, an object that you can repeatedly pass to the `next` built-in function to get one item at a time (see "Iterators"). When called with one argument, `iter(obj)` normally returns `obj.__iter__()`. When *`obj`* is a sequence without a special method `__iter__`, `iter(obj)` is equivalent to the generator:
def iter_sequence(obj):

```
i = 0
while True:
  try:
    yield obj[i]
  except IndexError:
    raise StopIteration
  i += 1
```

See also "Sequences", and `__iter__` in Table 4-2.
When called with two arguments, the first argument must be callable without arguments, and `iter(func, sentinel)` is equivalent to the generator:
def iter_sentinel(func, sentinel):

```
while True:
  item = func()
  if item == sentinel:
    raise StopIteration
  yield item
```

`iter` is *idempotent*. In other words, when *`x`* is an iterator, `iter(x)` is *`x`*, as long as *`x`*'s class supplies an `__iter__` method whose body is just `return self`, as an iterator's class should.

Don’t Call iter in a for Clause

As discussed in “The `for` Statement”, the statement `for x in obj` is exactly equivalent to `for x in iter(obj)`; therefore, do not explicitly call `iter` in such a `for` statement: that would be redundant, and, therefore, bad Python style, slower and less readable.

len	`len(container)` Returns the number of items in *container*, which may be a sequence, a mapping, or a set. See also `__len__` in “Container methods”.
locals	`locals()` Returns a dictionary that represents the current local namespace. Treat the returned dictionary as read-only; trying to modify it may or may not affect the values of local variables, and might raise an exception. See also `globals` and `vars` in Table 7-2.

map	`map(func,seq)`
	`map(func,*seqs)`
	`map` calls *func* on every item of iterable *seq* and returns an iterator of the results. When you call `map` with multiple *seqs* iterables, *func* must be a callable object that accepts *n* arguments (where *n* is the number of *seqs* arguments). `map` repeatedly calls *func* with *n* arguments, one corresponding item from each iterable.
	For example, `map(func, seq)` is just like the generator expression `(func(item) for item in seq)`. `map(func, seq1, seq2)` is just like the generator expression `(func(a, b) for a, b in zip(seq1, seq2))`. When `map`’s iterable arguments have different lengths, `map` acts as if the longer ones were truncated.

max	`max(seq,key=None[,default=...])`
	`max(*args,key=None[,default=...])`
	Returns the largest item in the iterable argument *seq*, or the largest one of multiple arguments *args*. You can pass a `key=` argument, with the same semantics covered in “Sorting a list”. You can also pass a `default=` argument, the value to return if *seq* is empty; when you don’t pass `default`, and *seq* is empty, `max` raises `ValueError`. (If given, `key` and `default` must be passed as named arguments.)

min	`min(seq,key=None[,default=...])`
	`min(*args,key=None[,default=...])`
	Returns the smallest item in the iterable argument *seq* or the smallest one of multiple arguments *args*. You can pass a `key=` argument, with the same semantics covered in “Sorting a list”. You can also pass a `default=` argument, the value to return if *seq* is empty; when you don’t pass

	`default`, and *`seq`* is empty, `min` raises `ValueError`. (If given, `key` and `default` must be passed as named arguments.)
next	`next(`*`it`*`[, `*`default`*`])` Returns the next item from iterator *`it`*, which advances to the next item. When *`it`* has no more items, `next` returns *`default`*, or, when you don't pass *`default`*, raises `StopIteration`.
oct	`oct(`*`x`*`)` Converts integer *`x`* to an octal string representation. See also `__oct__` in Table 4-4.
open	`open(`*`filename`*`,`*`mode`*`='r',`*`bufsize`*`=-1)` Opens or creates a file and returns a new file object. Open accepts many optional parameters; see "The io Module".
ord	`ord(`*`ch`*`)` Returns an `int` between `0` and `sys.maxunicode` (inclusive), corresponding to single-character `str` argument *`ch`*. See also `chr` in Table 7-2.
pow	`pow(`*`x`*`,`*`y`*`[,`*`z`*`])` When *`z`* is present, `pow(`*`x`*`,`*`y`*`,`*`z`*`)` returns *`x**y%z`*. When *`z`* is missing, `pow(`*`x`*`,`*`y`*`)` returns *`x**y`*. See also `__pow__` in Table 4-4.
print	`print(`*`value`*`, ..., sep=' ', end='\n', file=sys.stdout, flush=False)` Formats with `str`, and emits to stream `file`, each *`value`*, separated by `sep`, with `end` after all of them (then flushes the stream if `flush` is true).
range	`range([`*`start`*`,]`*`stop`*`[,`*`step`*`=1])` Returns an iterator of integers in arithmetic progression: *start*, *start+step*, *start+2*step*, ... When *`start`* is missing, it defaults to `0`. When *`step`* is missing, it defaults to `1`. When *`step`* is `0`, `range` raises `ValueError`. When *`step`* is `>0`, the last item is the largest *`start+i*step`* strictly less than *`stop`*. When *`step`* is `<0`, the last item is the smallest *`start+i*step`* strictly greater than *`stop`*. The iterator is empty when *`start`* is greater than or equal to *`stop`* and *`step`* is greater than `0`, or when *`start`* is less than or equal to *`stop`* and *`step`* is less than `0`. Otherwise, the first item of the iterator is always *`start`*. If you need a `list` in arithmetic progression, call `list(range(...))`.
repr	`repr(`*`obj`*`)` Returns a complete and unambiguous string representation of *`obj`*. When feasible, `repr` returns a string that you can pass to `eval` in order to create a new object with the same value as *`obj`*. See also `str` in Table 7-1 and `__repr__` in Table 4-1.
reversed	`reversed(`*`seq`*`)` Returns a new iterator object that yields the items of *`seq`* (which must be specifically a sequence, not just any iterable) in reverse order.

round

`round(x,n=0)`
Returns a `float` whose value is number *x* rounded to *n* digits after the decimal point (i.e., the multiple of `10**-` *n* that is closest to *x*). When two such multiples are equally close to *x*, `round` returns the *even* multiple. Since today's computers represent floating-point numbers in binary, not in decimal, most of `round`'s results are not exact, as the online tutorial explains in detail. See also "The decimal Module" and this famous, language-independent article.

setattr

`setattr(obj, name, value)`
Binds *obj*'s attribute *name* to *value*. `setattr(obj,'ident', val)` is like *obj.ident=val*. See also built-in `getattr` covered in Table 7-2, "Object attributes and items", and "Setting an attribute".

sorted

`sorted(seq,key=None,reverse=False)`
Returns a list with the same items as iterable *seq*, in sorted order. Same as:
def sorted(*seq,key*=**None**,*reverse*=**False**):

```
    result = list(seq)
    result.sort(key, reverse)
    return result
```

See "Sorting a list" for the meaning of the arguments. If you want to pass `key=` and/or `reverse=`, you **must** pass them by name.

sum

`sum(seq,start=0)`
Returns the sum of the items of iterable *seq* (which should be numbers, and, in particular, cannot be strings) plus the value of *start*. When *seq* is empty, returns *start*. To "sum" (concatenate) an iterable of strings, in order, use `''.join(iterofstrs)`, as covered in Table 8-1 and "Building up a string from pieces".

vars

`vars([obj])`
When called with no argument, `vars` returns a dictionary with all variables that are bound in the current scope (like `locals`, covered in Table 7-2). Treat this dictionary as read-only. `vars(obj)` returns a dictionary with all attributes currently bound in *obj*, as covered in `dir` in Table 7-2. This dictionary may be modifiable, depending on the type of *obj*.

zip

`zip(seq,*seqs,strict=False)`
Returns an iterator of tuples, where the *n*th tuple contains the *n*th element from each of the argument sequences. You must call `zip` with at least one argument, and all arguments must be iterable. `zip` returns an iterator with as many items as the shortest iterable, ignoring trailing items in the other iterable objects. ||**3.10**++|| When the iterables have different lengths and *strict* is true, `zip` raises `ValueError` once it reaches the end of the shortest iterable. See also `map` in Table 7-2 and `izip_longest` in Table 7-5.

Dynamic Execution

Python's `exec` built-in function can execute code that you read, generate, or otherwise obtain during a program's run. `exec` dynamically executes a statement or a suite of statements. `exec` is a built-in function with the syntax:

```
exec(code, globals=None, locals=None)
```

`code` can be a string, bytes, or code object. *`globals`* is a `dict`; *`locals`*, any mapping.

If both *`globals`* are *`locals`* are present, they are the global and local namespaces in which *`code`* runs. If only *`globals`* is present, `exec` uses *`globals`* as both namespaces. If neither is present, *`code`* runs in the current scope.

Never Run exec in the Current Scope

Running `exec` in the current scope is a particularly bad idea: it can bind, rebind, or unbind any global name. To keep things under control, use `exec`, if at all, only with specific, explicit dictionaries.

Avoiding exec

A frequently asked question about Python is "How do I set a variable whose name I just read or built?" Literally, for a *global* variable, `exec` allows this, but it's a very bad idea to use it for this purpose. For example, if the name of the variable is in *`varname`*, you might think to use:

```
exec(varname + ' = 23')
```

Don't do this. An `exec` like this in current scope makes you lose control of your namespace, leading to bugs that are extremely hard to find, and

making your program unfathomably difficult to understand. Keep the "variables" that you need to set with dynamically-found names, not as actual variables, but as entries in a dictionary, say `mydict`. You could then use:

```
exec(varname+'=23', mydict)
```

While this is not quite as terrible as the previous example, it is *still* a bad idea. Keeping such "variables" as dictionary entries means that you don't have any need to use `exec` to set them! Just code:

```
mydict[varname] = 23
```

This way, your program is clearer, direct, elegant, and faster. There *are* some valid uses of `exec`, but they are extremely rare: just use explicit dictionaries instead.

Strive to Avoid exec

Use `exec` only when it's really indispensable, which is *extremely* rare. Most often, it's best to avoid `exec` and choose more specific, well-controlled mechanisms: `exec` weakens your control of your code's namespace, can damage your program's performance, and exposes you to numerous hard-to-find bugs and huge security risks.

Expressions

`exec` can execute an expression, because any expression is also a valid statement (called an *expression statement*). However, Python ignores the value returned by an expression statement. To evaluate an expression and obtain the expression's value, see the built-in function `eval`, covered in Table XYZ. (Note, however, that most of the same caveats as for `exec` also apply to `eval`).

Compile and Code Objects

To make a code object to use with `exec`, call the built-in function `compile` with the last argument set to `'exec'` (as covered in Table XYZ).

A code object *c* exposes many interesting read-only attributes whose names all start with `'co_'`, such as:

`co_argcount`

Number of parameters of the function of which *c* is the code (`0` when *c* is not the code object of a function, but rather is built directly by `compile`)

`co_code`

A bytestring with *c*'s bytecode

`co_consts`

The tuple of constants used in *c*

`co_filename`

The name of the file *c* was compiled from (the string that is the second argument to `compile`, when *c* was built that way)

`co_firstlineno`

The initial line number (within the file named by `co_filename`) of the source code that was compiled to produce *c*, if *c* was built by compiling from a file

`co_name`

The name of the function of which *c* is the code (`'<module>'` when *c* is not the code object of a function but rather is built directly by `compile`)

`co_names`

The tuple of all identifiers used within *c*

`co_varnames`

The tuple of local variables' identifiers in *c*, starting with parameter names

Most of these attributes are useful only for debugging purposes, but some may help advanced introspection, as exemplified later in this section.

If you start with a string holding one or more statements, first use `compile` on the string, then call `exec` on the resulting code object—that's better than giving `exec` the string to compile and execute. This separation lets you check for syntax errors separately from execution-time errors. You can often arrange things so that the string is compiled once and the code object executes repeatedly, which speeds things up. `eval` can also benefit from such separation. Moreover, the `compile` step is intrinsically safe (both `exec` and `eval` are extremely risky if you execute them on code that you don't 100%-trust), and you may be able to check the code object, before it executes, to lessen the risk (though never truly down to zero).

A code object has a read-only attribute `co_names`, which is the tuple of the names used in the code. For example, say that you want the user to enter an expression that contains only literal constants and operators—no function calls or other names. Before evaluating the expression, you can check that the string the user entered satisfies these constraints:

```
def safer_eval(s):
    code = compile(s, '<user-entered string>', 'eval')
    if code.co_names:
        raise ValueError(f'No names {code.co_names!r} allowed in
expression {s!r}')
    return eval(code)
```

This function `safer_eval` evaluates the expression passed in as argument *s* only when the string is a syntactically valid expression

(otherwise, `compile` raises `SyntaxError`) and contains no names at all (otherwise, `safer_eval` explicitly raises `ValueError`). (This is similar to the standard library function `ast.literal_eval`, covered in "Standard Input", but a bit more powerful, since it does allow the use of operators.)

Knowing what names the code is about to access may sometimes help you optimize the preparation of the dictionary that you need to pass to `exec` or `eval` as the namespace. Since you need to provide values only for those names, you may save work by not preparing other entries. For example, say that your application dynamically accepts code from the user, with the convention that variable names starting with *`data_`* refer to files residing in the subdirectory *data* that user-written code doesn't need to read explicitly. User-written code may, in turn, compute and leave results in global variables with names starting with *`result_`*, which your application writes back as files in subdirectory *data*. Thanks to this convention, you may later move the data elsewhere (e.g., to BLOBs in a database instead of files in a subdirectory), and user-written code won't be affected. Here's how you might implement these conventions efficiently:

```
def exec_with_data(user_code_string):
    user_code = compile(user_code_string, '<user code>', 'exec')
    datadict = {}
    for name in user_code.co_names:
        if name.startswith('data_'):
            with open(f'data/{name[5:]}', 'rb') as datafile:
                datadict[name] = datafile.read()
        elif name.startswith('result_'):
            pass  # user code can assign to variables named
`result_…`
        else:
            raise ValueError(f'invalid variable name {name!r}')
    exec(user_code, datadict)
    for name in datadict:
        if name.startswith('result_'):
            with open('data/{}'.format(name[7:]), 'wb') as
datafile:
                datafile.write(datadict[name])
```

Never exec or eval Untrusted Code

Some old versions of Python tried to supply tools to ameliorate the risks of using `exec` and `eval`, under the heading of "restricted execution," but those tools were never entirely secure against the ingenuity of able hackers, and recent versions of Python have dropped them. If you need to ward against such attacks, take advantage of your operating system's protection mechanisms: run untrusted code in a separate process, with privileges as restricted as you can possibly make them (study the mechanisms that your OS supplies for the purpose, such as `chroot`, `setuid`, and `jail`; in Windows, you might try third-party, commercial add-on WinJail, or run untrusted code in a separate, highly constrained virtual machine (or container, if you're an expert on how to securitize containers). To guard against "denial of service" attacks, have the main process monitor the separate one and terminate the latter if and when resource consumption becomes excessive. Processes are covered in "Running Other Programs".

`exec` and `eval` Are Unsafe With Untrusted Code

The function `exec_with_data` is not at all safe against untrusted code: if you pass to it, as the argument *`user_code_string`*, some string obtained in a way that you cannot *entirely* trust, there is essentially no limit to the amount of damage it might do. This is unfortunately true of just about any use of both `exec` and `eval`, except for those rare cases in which you can set very strict and checkable limits on the code to execute or evaluate, as was the case for the function `safer_eval`.

The sys Module

The attributes of the `sys` module are bound to data and functions that provide information on the state of the Python interpreter or affect the interpreter directly. Table 7-3 covers the most frequently used attributes of `sys`, in alphabetical order. Most `sys` attributes we don't cover are meant specifically for use in debuggers, profilers, and integrated development environments; see the online docs for more information.

Platform-specific information is best accessed using the `platform` module, covered online; we do not cover the `platform` module in this book.

argv	The list of command-line arguments passed to the main script. `argv[0]` is the name or full path of the main script, or `'-c'` if the command line used the **-c** option. See "The argparse Module" for one good way to use `sys.argv`.
audit	`audit(`*`event, *args`*`)` Raise an *audit event* whose name is `str` *`event`* and whose arguments are *`args`*. The rationale for Python's audit system is laid out in exhaustive detail at PEP 578; Python itself raises the large variety of events listed online. To *listen* for events, call `sys.addaudithook(`*`hook`*`)`, where *`hook`* is a callable whose arguments are a `str`, the event's name, followed by arbitrary positional arguments. For more details, see the online docs.
builtin_module_names	A tuple of strings, the name of all the modules compiled into this Python interpreter.

displayhook	`displayhook(`*`value`*`)` In interactive sessions, the Python interpreter calls `displayhook`, passing it the result of each expression statement you enter. The default `displayhook` does nothing if *`value`* is `None`; otherwise, it preserves *`value`* (in the built-in variable _) and displays it via `repr`:
	`def _default_sys_displayhook(value):`
	`  if value is not None:`
	`  __builtins__._ = value`
	`  print(repr(value))`
	You can rebind `sys.displayhook` in order to change interactive behavior. The original value is available as `sys.__displayhook__`.

dont_write_bytecode	When true, Python does not write a bytecode file (with extension *`.pyc`*) to disk, when it imports a source file (with the extension *`.py`*).
excepthook	`excepthook(`*`type, value, traceback`*`)` When an exception is not caught by any handler, propagating all the way up the call stack, Python calls `excepthook`, passing it the exception class, object, and traceback, as covered in "Exception Propagation". The default `excepthook` displays the error and traceback. You can rebind

	`sys.excepthook` to change how uncaught exceptions (just before Python returns to the interactive loop or terminates) get displayed and/or logged. The original value is available as `sys.__excepthook__`.
exc_info	`exc_info()` If the current thread is handling an exception, `exc_info` returns a tuple with three items: the class, object, and traceback for the exception. If the current thread is not handling an exception, `exc_info` returns (`None, None, None`). To display information from a traceback, see "The traceback Module".

Holding on to a Traceback Object Can Make Some Garbage Uncollectable

A traceback object indirectly holds references to all variables on the call stack; if you hold a reference to the traceback (e.g., indirectly, by binding a variable to the tuple that `exc_info` returns), Python must keep in memory data that might otherwise be garbage-collected. Make sure that any binding to the traceback object is of short duration, for example with a `try/finally` statement (discussed in "try/finally"). If you must hold a reference to an exception `e`, clear its traceback: `e.__traceback__=None`.

exit	`exit(`*`arg`*`=0)` Raises a `SystemExit` exception, which normally terminates execution after executing cleanup handlers installed by `try/finally` statements, `with` statements, and the `atexit` module. When *`arg`* is an `int`, Python uses *`arg`* as the program's exit code: `0` indicates successful termination, any other value indicates unsuccessful termination of the program. Most platforms require exit codes to be between `0` and `127`. When *`arg`* is not an `int`, Python prints *`arg`* to `sys.stderr`, and the exit code of the program is `1` (a generic "unsuccessful termination" code).
float_info	A read-only object whose attributes hold low-level details about the implementation of the `float` type in this Python interpreter. See the online docs for details.
getrefcount	`getrefcount(`*`object`*`)` Returns the reference count of *`object`*. Reference counts are covered in "Garbage Collection".
getrecursionli	`getrecursionlimit()` Returns the current limit on the depth of Python's call stack. See also "Recursion" and `setrecursionlimit` in Table 7-3.

mit	
getsizeof	`getsizeof(`*`obj`*`,[`*`default`*`])` Returns the size in bytes of *`obj`* (not counting any items or attributes *`obj`* may refer to), or *`default`* when *`obj`* does not provide a way to retrieve its size (in the latter case, when *`default`* is absent, `getsizeof` raises `TypeError`).
maxsize	Maximum number of bytes in an object in this version of Python (at least `2**31-1`, that is, `2147483647`).
maxunicode	The largest codepoint for a Unicode character in this version of Python; currently, always `1114111` (`0x10FFFF`).
modules	A dictionary whose items are the names and module objects for all loaded modules. See “Module Loading” for more information on `sys.modules`.
path	A list of strings that specifies the directories and `ZIP` files that Python searches when looking for a module to load. See “Searching the Filesystem for a Module” for more information on `sys.path`.
platform	A string that names the platform on which this program is running. Typical values are brief operating system names, such as `'darwin'`, `'linux2'`, and `'win32'`. For Linux, check `sys.platform.startswith('linux')`, for portability among Linux versions. See also the online docs for module platform, which we don’t cover in this book.
ps1, ps2	`ps1` and `ps2` specify the primary and secondary interpreter prompt strings, initially `'>>> '` and `'... '`, respectively. These `sys` attributes exist only in interactive interpreter sessions. If you bind either attribute to a non-`str` object *`x`*, Python prompts by calling `str(`*`x`*`)` on the object each time a prompt is output. This feature allows dynamic prompting: code a class that defines `__str__`, then assign an instance of that class to `sys.ps1` and/or `sys.ps2`. For example, to get numbered prompts:

```
>>> import sys
>>> class Ps1(object):
... def init (self):
... self.p = 0
... def __str__ (self):
... self.p += 1
... return f'[{self.p}]>>> '
...
>>> class Ps2(object):
```

```
... def __str__(self):
... return f'[{sys.ps1.p}]... '
...
>>> sys.ps1 = Ps1(); sys.ps2 = Ps2()
[1]>>> (2 +
[1]... 2)
4
[2]>>>
```

setrecursionlimit	`setrecursionlimit(`*`limit`*`)` Sets the limit on the depth of Python's call stack (the default is `1000`). The limit prevents runaway recursion from crashing Python. Raising the limit may be necessary for programs that rely on deep recursion, but most platforms cannot support very large limits on call-stack depth. More usefully, *lowering* the limit helps you check, during testing and debugging, that your program degrades gracefully, rather than abruptly crashing with a `RecursionError`, under situations of almost-runaway recursion. See also "Recursion" and `getrecursionlimit` in Table 7-3.
stdin, stdout, stderr	`stdin`, `stdout`, and `stderr` are predefined file-like objects that correspond to Python's standard input, output, and error streams. You can rebind `stdout` and `stderr` to file-like objects open for writing (objects that supply a `write` method accepting a string argument) to redirect the destination of output and error messages. You can rebind `stdin` to a file-like object open for reading (one that supplies a `readline` method returning a string) to redirect the source from which built-in function `input` reads. The original values are available as `__stdin__`, `__stdout__`, and `__stderr__`. File objects are covered in "The io Module".
tracebacklimit	The maximum number of levels of traceback displayed for unhandled exceptions. By default, this attribute is not set (i.e., there is no limit). When `sys.tracebacklimit` is `<=0`, Python prints only the exception type and value, without traceback.
version	A string that describes the Python version, build number and date, and `C` compiler used. Use `sys.version` only for logging or interactive output; to perform version comparisons, use `sys.version_info`.
version_info	A namedtuple of the `major`, `minor`, `micro`, `releaselevel`, and `serial` fields of the running Python version. For example, in the first post-beta release of Python 3.10, `sys.version_info` was `sys.version_info(major=3, minor=10, micro=0, releaselevel='final', serial=0)`, equivalent to the tuple `(3, 10, 0, 'final', 0)`. This form is defined to be directly comparable between versions; to see if the current version running is greater than or equal to, say 3.8, you can test `sys.version_info[:3] >= (3, 8, 0)`.

(Do *not* do string comparisons of the string `sys.version`, since the string "3.10" would compare as less than "3.9"!)

The copy Module

As discussed in "Assignment Statements", assignments in Python do not *copy* the righthand-side object being assigned. Rather, assignments *add references* to the righthand-side object. When you want a *copy* of object `x`, ask `x` for a copy of itself, or ask `x`'s type to make a new instance copied from `x`. If `x` is a list, `list(x)` returns a copy of `x`, as does `x[:]`. If `x` is a dictionary, `dict(x)` and `x.copy()` return a copy of `x`. If `x` is a set, `set(x)` and `x.copy()` return a copy of `x`. In each case, we prefer the uniform and readable idiom of calling the type, but there is no consensus on this style issue in the Python community.

The `copy` module supplies a `copy` function to create and return a copy of many types of objects. Normal copies, such as `list(x)` for a list `x` and `copy.copy(x)`, are known as *shallow* copies: when `x` has references to other objects (either as items or as attributes), a normal (shallow) copy of `x` has distinct references to the same objects. Sometimes, however, you need a *deep* copy, where referenced objects are deep-copied recursively; fortunately, this need is rare, since a deep copy can take a lot of memory and time. The `copy` module supplies a `deepcopy` function to create and return a deep copy.

copy

`copy(x)`
Creates and returns a shallow copy of `x`, for `x` of many types (copies of several types, such as modules, classes, files, frames, and other internal types, are, however, not supported). If `x` is immutable, `copy.copy(x)` may return `x` itself as an optimization. A class can customize the way `copy.copy` copies its instances by having a special method `__copy__(self)` that returns a new object, a shallow copy of `self`.

deepcopy

`deepcopy(x,[memo])`
Makes a deep copy of `x` and returns it. Deep copying implies a recursive walk over a directed (not necessarily acyclic) graph of references. A special precaution is needed to reproduce the graph's exact shape: when references to the same object are met more than once during the walk, distinct copies must *not* be made. Rather, references to the same copied object must be used. Consider the following simple example:

```
sublist = [1,2]
original = [sublist, sublist]
thecopy = copy.deepcopy(original)
```

`original[0]` is `original[1]` is `True` (i.e., the two items of `original` refer to the same object). This is an important property of `original` and anything claiming to "be a copy" must preserve it. The semantics of `copy.deepcopy` ensure that `thecopy[0] is thecopy[1]` is also `True`: the graphs of references of `original` and `thecopy` have the same shape. Avoiding repeated copying has an important beneficial side effect: it prevents infinite loops that would otherwise occur when the graph of references has cycles.

`copy.deepcopy` accepts a second, optional argument *`memo`*, `a dict` that maps the `id` of each object already copied to the new object that is its copy. *`memo`* is passed by all recursive calls of `deepcopy` to itself; you may also explicitly pass it (normally as an originally empty `dict`) if you need to maintain a correspondence map between the identities of originals and copies.

A class can customize the way `copy.deepcopy` copies its instances by having a special method `__deepcopy__(self, memo)` that returns a new object, a deep copy of `self`. When `__deepcopy__` needs to deep copy some referenced object *`subobject`*, it must do so by calling `copy.deepcopy(subobject, memo)`. When a class has no special method `__deepcopy__`, `copy.deepcopy` on an instance of that class also tries calling the special methods `__getinitargs__`, `__getnewargs__`, `__getstate__`, and `__setstate__`, covered in "Pickling instances".

The collections Module

The `collections` module supplies useful types that are collections (i.e., containers), as well as the *abstract base classes* (ABCs) covered in "Abstract Base Classes". Since Python 3.4, the ABCs are in `collections.abc` (but, for backward compatibility, can still be accessed directly in `collections` itself: the latter access is deprecated and will cease working in some future release).

ChainMap

`ChainMap` "chains" multiple mappings together; given a `ChainMap` instance *`c`*, accessing *`c[key]`* returns the value in the first of the mappings

that has that key, while *all* changes to `c` only affect the very first mapping in `c`. Just to further explain, you could approximate this as follows:

```
class ChainMap(collections.MutableMapping):
    def __init__(self, *maps):
        self.maps = list(maps)
        self._keys = set()
        for m in self.maps:
            self._keys.update(m)
    def __len__(self): return len(self._keys)
    def __iter__(self): return iter(self._keys)
    def __getitem__(self, key):
        if key not in self._keys: raise KeyError(key)
        for m in self.maps:
            try: return m[key]
            except KeyError: pass
    def __setitem__(self, key, value)
        self.maps[0][key] = value
        self._keys.add(key)
    def __delitem__(self, key)
        del self.maps[0][key]
        self._keys = set()
        for m in self.maps:
            self._keys.update(m)
```

Other methods could be defined for efficiency, but this is the minimum set that `MutableMapping` requires. See the `ChainMap` documentation in the online Python docs for more details and a collection of recipes on how to use `ChainMap`.

Counter

`Counter` is a subclass of `dict` with `int` values that are meant to *count* how many times the key has been seen (although values are allowed to be `<=0`); it roughly corresponds to types that other languages call "bag" or "multi-set." A `Counter` instance is normally built from an iterable whose items are hashable: `c = collections.Counter(iterable)`. Then, you can index `c` with any of `iterable`'s items, to get the number of times that item appeared. When you index `c` with any missing key, the result is `0` (to remove an entry in `c`, use `del c[entry]`; setting `c[entry] = 0` leaves `entry` in `c`, just with a corresponding value of `0`).

c supports all methods of `dict`; in particular, `c.update(`*`otheriterable`*`)` updates all the counts, incrementing them according to occurrences in *`otheriterable`*. So, for example:

```
c = collections.Counter('moo')
c.update('foo')
```

leaves `c['o']` giving 4, and `c['f']` and `c['m']` each giving `1`.

In addition to `dict` methods, *c* supports the following extra methods:

elements	`c.elements()` Yields, in arbitrary order, keys in *c* with `c[`*`key`*`]>0`, yielding each key as many times as its count.
most_common	`c.most_comon([`*`n`*`])` Returns a list of pairs for the *n* keys in *c* with the highest counts (all of them, if you omit *n*) in order of decreasing count ("ties" between keys with the same count are resolved arbitrarily); each pair is of the form `(`*`k`*`, c[`*`k`*`])` where *k* is one of the *n* most common keys in *c*.
subtract	`c.subtract(`*`iterable`*`)` Like `c.update(`*`iterable`*`)` "in reverse"—that is, *subtracting* counts rather than *adding* them. Resulting counts in *c* can be `<=0`.
total	`c.total()` Returns the sum of all the individual counts. Equivalent to `sum(c.values())`.

`Counter` objects support common arithmetic operators, such as `+`, `-`, `&`, and | for addition, subtraction, union, and intersection. See the `Counter` documentation in the online Python docs for more details and a collection of useful recipes on how to use `Counter`.

OrderedDict

`OrderedDict` is a subclass of `dict` that provides additional methods for accessing and manipulating items with respect to their insertion order. `o.popitem()` removes and returns the item at the key most recently inserted; `o.move_to_end(`*`key`*`, last=True)` moves the item with

key *key* to the end of the order (when `last` is `True`, the default) or to the start of the order (when `last` is `False`). Equality tests between two instances of `OrderedDict` are order-sensitive; equality tests between an instance of `OrderedDict` and a `dict` or other mapping are not. See the `OrderedDict` documentation in the online Python docs for more details and a collection of recipes on how to use `OrderedDict`.

defaultdict

`defaultdict` extends `dict` and adds one per-instance attribute, named `default_factory`. When an instance *d* of `defaultdict` has `None` as the value of *d*`.default_factory`, *d* behaves exactly like a `dict`. Otherwise, *d*`.default_factory` must be callable without arguments, and *d* behaves just like a `dict` except when you access *d* with a key *k* that is not `in` *d*. In this specific case, the indexing *d*`[`*k*`]` calls *d*`.default_factory()`, assigns the result as the value of *d*`[`*k*`]`, and returns the result. In other words, the type `defaultdict` behaves much like the following Python-coded class:

```
class defaultdict(dict):
    def __init__(self, default_factory=None, *a, **k):
        super().__init__(*a, **k)
        self.default_factory = default_factory
    def __getitem__(self, key):
        if key not in self and self.default_factory is not None:
            self[key] = self.default_factory()
        return dict.__getitem__(self, key)
```

As this Python equivalent implies, to instantiate `defaultdict` you usually pass it an extra first argument (before any other arguments, positional and/or named, if any, to pass on to plain `dict`). The extra first argument becomes the initial value of `default_factory`; you can also access and rebind `default_factory` later (though doing so is infrequent in normal Python code).

All behavior of `defaultdict` is essentially as implied by this Python equivalent (except `str` and `repr`, which return strings different from those

they would return for a `dict`). Named methods, such as `get` and `pop`, are not affected. All behavior related to keys (method `keys`, iteration, membership test via operator `in`, etc.) reflects exactly the keys that are currently in the container (whether you put them there explicitly, or implicitly via an indexing that called `default_factory`).

A typical use of `defaultdict` is, for example, to set `default_factory` to `list`, to make a mapping from keys to lists of values:

```
def make_multi_dict(items):
    d = collections.defaultdict(list)
    for key, value in items:
        d[key].append(value)
    return d
```

Called with any iterable whose items are pairs of the form `(`*`key`*`, `*`value`*`)`, with all keys being hashable, this *`make_multi_dict`* function returns a mapping that associates each key to the lists of one or more values that accompanied it in the iterable (if you want a pure `dict` result, change the last statement into `return dict(d)` — this is rarely necessary).

If you don't want duplicates in the result, and every *`value`* is hashable, use a `collections.defaultdict(set)`, and `add` rather than `append` in the loop.[1]

keydefaultdict

A variation on `defaultdict` that is not found in the `collections` module is a `defaultdict` whose `default_factory` takes the key as an initialization argument. This example shows how you can implement this for yourself:

```
class keydefaultdict(dict):
    def __init__(self, default_factory=None, *a, **k):
        super().__init__(*a, **k)
        self.default_factory = default_factory
    def __missing__(self, key):
        if self.default_factory is None:
```

```
            raise KeyError(key)
        self[key] = self.default_factory(key)
        return self[key]
```

The `dict` class supports the `__missing__` method for subclasses to implement custom behavior when a key is accessed that is not yet in the `dict`. In this example, we implement `__missing__` to call the default factory method with the new key, and add it to the dict.

deque

`deque` is a sequence type whose instances are "double-ended queues" (additions and removals at either end are fast). A `deque` instance *`d`* is a mutable sequence, with an optional maximum length, and can be indexed and iterated on (however, *`d`* cannot be sliced - it can only be indexed one item at a time, whether for access, rebinding, or deletion). If a deque instance *`d`* has a maximum length, when items are added to either side of *`d`* so that *`d`*'s length exceeds that maximum, items are silently dropped from the other side.

`deque` is especially useful for implementing first-in-first-out (FIFO) queues and last-in-first-out (LIFO) queues, or *stacks*. `deque` is also good for maintaining "the latest *N* things seen," also known in some other languages as a *ring buffer*.

`deque` supplies the following methods:

deque	`deque(`*`seq`*`=(),`*`maxlen`*`=None)` The initial items of *`d`* are those of *`seq`*, in the same order. *`d`*`.maxlen` is a read-only attribute: when `None`, *`d`* has no maximum length; when an `int`, it must be `>=0`: *`d`*'s maximum length is *`d`*`.maxlen`.
append	*`d`*`.append(`*`item`*`)` Appends *`item`* at the right (end) of *`d`*.
appendleft	*`d`*`.appendleft(`*`item`*`)` Appends *`item`* at the left (start) of *`d`*.
	`d``.clear()`

clear	Removes all items from `d`, leaving it empty.
extend	`d.extend(iterable)` Appends all items of `iterable` at the right (end) of `d`.
extendleft	`d.extendleft(iterable)` Appends all items of `iterable` at the left (start) of `d` in reverse order.
pop	`d.pop()` Removes and returns the last (rightmost) item from `d`. If `d` is empty, raises `IndexError`.
popleft	`d.popleft()` Removes and returns the first (leftmost) item from `d`. If `d` is empty, raises `IndexError`.
rotate	`d.rotate(n=1)` Rotates `d` `n` steps to the right (if `n<0`, rotates left).

The functools Module

The `functools` module supplies functions and types supporting functional programming in Python, listed in Table 7-4.

cached_property	`cached_property(func)` **||3.8++||** A caching version of the `property` decorator. When the property is evaluated its first time, the returned value is cached, so that subsequent calls can return the cached value instead of repeating the property calculation. `cached_property` uses a threading lock to ensure that the property calculation is performed only once, even in a multithreaded environment.[a]
lru_cache, **||3.9++|| cache**	`lru_cache(max_size=128, typed=False)` A *memoizing* decorator suitable for decorating a function whose arguments are all hashable, adding to the function a cache storing the last `max_size` results (`max_size` should be a power of 2, or `None` to have the cache keep all previous results); when the decorated function is called again with arguments that are in the cache, it immediately returns the previously cached result, bypassing the underlying function's body code. When `typed` is true, arguments that compare equal but have different types, such as 23 and 23.0,

	are cached separately. ||**3.9++**|| If setting maxsize to `None`, use `cache()` instead. For more details and examples, see the online docs.
partial	`partial(`*`func`*`, *`*`a`*`,**`*`k`*`)` *`func`* is any callable. `partial` returns another callable *`p`* that is just like *`func`*, but with some positional and/or named parameters already bound to the values given in *`a`* and *`k`*. In other words, *`p`* is a *partial application* of *`func`*, often also known (with debatable correctness, but colorfully) as a *currying* of *`func`* to the given arguments (named in honor of mathematician Haskell Curry). For example, say that we have a list of numbers `L` and want to clip the negative ones to `0`; one way to do it is: L = map(functools.partial(max, 0), L) as an alternative to the lambda-using snippet: L = map(**lambda** x: max(0, x), L) and to the most concise approach, a list comprehension: L = [max(0, x) **for** x **in** L] `functools.partial` comes into its own in situations that demand callbacks, such as event-driven programming for some GUIs and networking applications. `partial` returns a callable with the attributes `func` (the wrapped function), `args` (the `tuple` of prebound positional arguments), and `keywords` (the `dict` of prebound named arguments, or `None`).

reduce	`reduce(`*`func`*`,`*`seq`*`[,`*`init`*`])`
	Applies *`func`* to the items of *`seq`*, from left to right, to reduce the iterable to a single value. *`func`* must be callable with two arguments. `reduce` calls *`func`* on the first two items of *`seq`*, then on the result of the first call and the third item, and so on. `reduce` returns the result of the last such call. When *`init`* is present, it is used before *`seq`*'s first item, if any. When *`init`* is missing, *`seq`* must be nonempty. When *`init`* is missing and *`seq`* has only one item, `reduce` returns *`seq`*`[0]`. Similarly, when *`init`* is present and *`seq`* is empty, `reduce` returns *`init`*`.` `reduce` is roughly equivalent to:
	
	`def reduce_equiv(`*`func`*`,`*`seq`*`,`*`init`*`=None):`
	`  seq = iter(seq)`
	`  if `*`init`*` is None:`
	`  `*`init`*` = next(`*`seq`*`)`
	`  for `*`item`*` in `*`seq`*`:`
	`  `*`init`*` = `*`func`*`(`*`init`*`,`*`item`*`)`
	`  return `*`init`*
	
	An example use of `reduce` is to compute the product of a sequence of numbers:
	`prod=reduce(operator.mul,seq,1)`

	A function decorator to support multiple implementations of a method with differing types for their first argument. See a detailed description in the online

singledispatch, **singledispatchmethod**	docs.
total_ordering	A class decorator suitable for decorating classes that supply at least one inequality comparison method, such as `__lt__`, and also supply `__eq__`. Based on the class's existing methods, the class decorator `total_ordering` adds to the class all other inequality comparison methods, removing the need for you to add boilerplate code for them.
wraps	`wraps(`*`wrapped`*`)` A decorator suitable for decorating functions that wrap another function *`wrapped`* (often nested functions within another decorator). `wraps` copies the `__name__`, `__doc__`, and `__module__` attributes of *`wrapped`* on the decorated function, thus improving the behavior of the built-in function `help`, and of doctests, covered in "The doctest Module".

a In Python versions 3.8-3.10, cached_property is implemented using a class-level lock. As such, it synchronizes for all instances of the class or any subclass, not just the current instance. For this reason, cached_property can significantly reduce performance in a multithreaded environment, and is not recommended.

The heapq Module

The `heapq` module uses *min-heap* algorithms to keep a list in "nearly sorted" order as items are inserted and extracted. `heapq`'s operation is faster than calling a list's `sort` method after each insertion, and much faster than `bisect` (covered in the online docs). For many purposes, such as implementing "priority queues," the nearly-sorted order supported by `heapq` is just as good as a fully sorted order, and faster to establish and maintain. The `heapq` module supplies the following functions:

heapify	`heapify(`*`alist`*`)` Permutes `list` *`alist`* as needed to make it satisfy the (min) *`heap`* condition: ▪ for any *`i`*`>=0` — *`alist`*`[`*`i`*`]<=`*`alist`*`[2*`*`i`*`+1]` and — *`alist`*`[`*`i`*`]<=`*`alist`*`[2*`*`i`*`+2]` ▪ as long as all the indices in question are `<` `len(`*`alist`*`)`

	If a `list` satisfies the (min) heap condition, the list's first item is the smallest (or equal-smallest) one. A sorted `list` satisfies the heap condition, but many other permutations of a list also satisfy the heap condition, without requiring the list to be fully sorted. `heapify` runs in `O(len(`*`alist`*`))` time.
heappop	`heappop(`*`alist`*`)` Removes and returns the smallest (first) item of *`alist`*, a `list` that satisfies the heap condition, and permutes some of the remaining items of *`alist`* to ensure the heap condition is still satisfied after the removal. `heappop` runs in `O(log(len(`*`alist`*`)))` time.
heappush	`heappush(`*`alist`*`, `*`item`*`)` Inserts the *`item`* in *`alist`*, a `list` that satisfies the heap condition, and permutes some items of *`alist`* to ensure the heap condition is still satisfied after the insertion. `heappush` runs in `O(log(len(`*`alist`*`)))` time.

heappushpop	`heappushpop(`*`alist`*`,`*`item`*`)`
	Logically equivalent to `heappush` followed by `heappop`, similarly to:
	`def heappushpop(alist, item):`
	`  heappush(alist, item)`
	`  return heappop(alist)`
	`heappushpop` runs in `O(log(len(`*`alist`*`)))` time and is generally faster than the logically equivalent function just shown. `heappushpop` can be called on an empty *`alist`*: in that case, it returns the *`item`* argument, as it does when *`item`* is smaller than any existing item of *`alist`*.

heapreplace	`heapreplace(`*`alist`*`, `*`item`*`)`
	Logically equivalent to `heappop` followed by `heappush`, similar to:
	`def heapreplace(alist, item):`
	`  try: return heappop(alist)`
	`  finally: heappush(alist, item)`
	`heapreplace` runs in `O(log(len(`*`alist`*`)))` time and is generally faster than the logically equivalent function just shown. `heapreplace` cannot be called on an empty *`alist`*: `heapreplace` always returns an item that was already in *`alist`*, never the *`item`* just being pushed onto it.

merge	`merge(`*`*iterables`*`)` Returns an iterator yielding, in sorted order (smallest to largest), the items of the *`iterables`*, each of which must be smallest-to-largest sorted.
	`nlargest(`*`n`*`,`*`seq`*`,`*`key`*`=None)`

nlargest	Returns a reverse-sorted `list` with the *`n`* largest items of iterable *`seq`* (less than *`n`* if *`seq`* has fewer than *`n`* items); like `sorted(`*`seq`*`, reverse=True)[:`*`n`*`]`, but faster for small values of *`n`*. You may also specify a `key=` argument, like you can for `sorted`.
nsmallest	`nsmallest(`*`n`*`,`*`seq`*`,`*`key`*`=None)` Returns a sorted `list` with the *`n`* smallest items of iterable *`seq`* (less than *`n`* if *`seq`* has fewer than *`n`* items); like `sorted(`*`seq`*`)[:`*`n`*`]`, but faster for small values of *`n`*. You may also specify a `key=` argument, like you can for `sorted`.

The Decorate-Sort-Undecorate Idiom

Several functions in the `heapq` module, although they perform comparisons, do not accept a `key=` argument to customize the comparisons. This is inevitable, since the functions operate in-place on a plain `list` of the items: they have nowhere to "stash away" custom comparison keys computed once and for all.

When you need both heap functionality and custom comparisons, you can apply the good old *decorate-sort-undecorate (DSU)* idiom (which used to be crucial to optimize sorting in ancient versions of Python, before the `key=` functionality was introduced).

The DSU idiom, as applied to `heapq`, has the following components:

Decorate

> Build an auxiliary list *`A`* where each item is a tuple starting with the sort key and ending with the item of the original list *`L`*.

Sort[2]

> Call `heapq` functions on *`A`*, typically starting with `heapq.heapify(`*`A`*`)`.

Undecorate

> When you extract an item from *`A`*, typically by calling `heapq.heappop(`*`A`*`)`, return just the last item of the resulting tuple (which corresponds to an item of the original `list` *`L`*).

When you add an item to *A* by calling `heapq.heappush(`*A*, *item*`)`, decorate the actual item you're inserting into a tuple starting with the sort key.

This sequencing is best wrapped up in a class, as in this example:

```
import heapq
class KeyHeap(object):
    def __init__(self, alist, key):
        self.heap = [
            (key(o), i, o)
            for i, o in enumerate(alist)]
        heapq.heapify(self.heap)
        self.key = key
        if alist:
            self.nexti = self.heap[-1][1] + 1
        else:
            self.nexti = 0
    def __len__(self):
        return len(self.heap)
    def push(self, o):
        heapq.heappush(
            self.heap,
            (self.key(o), self.nexti, o))
        self.nexti += 1
    def pop(self):
        return heapq.heappop(self.heap)[-1]
```

In this example, we use an increasing number in the middle of the decorated tuple (after the sort key, before the actual item) to ensure that actual items are never compared directly, even if their sort keys are equal (this semantic guarantee is an important aspect of the *key=* argument's functionality to `sort` and the like).

The argparse Module

When you write a Python program meant to be run from the command line (or from a "shell script" in Unix-like systems, or a "batch file" in Windows), you often want to let the user pass to the program, on the command line, *command-line arguments* (including *command-line options*, which by convention are usually arguments starting with one or two dash

characters). In Python, you can access the arguments as `sys.argv`, an attribute of module `sys` holding those arguments as a list of strings (`sys.argv[0]` is the name by which the user started your program; the arguments are in sublist `sys.argv[1:]`). The Python standard library offers three modules to process those arguments; we only cover the newest and most powerful one, `argparse`, and we only cover a small, *core* subset of `argparse`'s rich functionality. See the online reference and tutorial for much, much more.

ArgumentParser	`ArgumentParser(**kwarps)` `ArgumentParser` is the class whose instances perform argument parsing. It accepts many named arguments, mostly meant to improve the help message that your program displays if command-line arguments include `-h` or `--help`. One named argument you should pass is `description=`, a string summarizing the purpose of your program.

Given an instance *`ap`* of `ArgumentParser`, prepare it by one or more calls to *`ap`*`.add_argument`; then, use it by calling *`ap`*`.parse_args()` without arguments (so it parses `sys.argv`): the call returns an instance of `argparse.Namespace`, with your program's args and options as attributes.

`add_argument` has a mandatory first argument: an identifier string, for positional command-line arguments, or a flag name, for command-line options. In the latter case, pass one or more flag names; an option can have both a short name (dash, then a character) and a long name (two dashes, then an identifier).

After the positional arguments, pass to `add_argument` zero or more named arguments to control its behavior. Here are the most commonly-used ones:

action	What the parser does with this argument. Default: `'store'`, store the argument's value in the namespace (at the name given by `dest`). Also useful: `'store_true'` and `'store_false'`, making an option into a `bool` one (defaulting to the opposite `bool` if the option is not present); and `'append'`, appending argument values to a list (and thus allowing an option to be repeated).

choices	A set of values allowed for the argument (parsing the argument raises an exception if the value is not among these); default, no constraints.
default	Value if the argument is not present; default, `None`.
dest	Name of the attribute to use for this argument; default, same as the first positional argument stripped of dashes.
help	A `str` about the argument, for help messages.
nargs	Number of command-line arguments used by this logical argument (default `1`, stored in the namespace). Can be an `int>0` (uses that many arguments, stores them as a list), `'?'` (one, or none in which case `default` is used), `'*'` (`0` or more, stored as a list), `'+'` (`1` or more, stored as a list), or `argparse.REMAINDER` (all remaining arguments, stored as a list).
type	A callable accepting a string, often a type such as `int`; used to transform values from strings to something else. Can be an instance of `argparse.FileType` to open the string as a filename (for reading if `FileType('r')`, for writing if `FileType('w')`, and so on).

Here's a simple example of `argparse`—save this code in a file called *greet.py*:

```
import argparse
ap = argparse.ArgumentParser(description='Just an example')
ap.add_argument('who', nargs='?', default='World')
ap.add_argument('--formal', action='store_true')
ns = ap.parse_args()if ns.formal:
    greet = 'Most felicitous salutations, o {}.'else:
    greet = 'Hello, {}!'
print(greet.format(ns.who))
```

Now, **`python greet.py`** prints `Hello, World!`, while **`python greet.py --formal Cornelia`** prints `Most felicitous salutations, o Cornelia.`

The itertools Module

The `itertools` module offers high-performance building blocks to build and manipulate iterators. To handle long processions of items, iterators are usually better than lists, thanks to iterators' intrinsic "lazy evaluation" approach: an iterator produces items one at a time, as needed, while all items of a list (or other sequence) must be in memory at the same time. This approach even makes it feasible to build and use unbounded iterators, while lists must always have finite numbers of items (since any machine has a finite amount of memory).

Table 7-5 covers the most frequently used attributes of `itertools`; each of them is an iterator type, which you call to get an instance of the type in question, or a factory function behaving similarly. See the itertools documentation in the online Python docs for more `itertools` attributes, including *combinatorial* generators for permutations, combinations, and Cartesian products, as well as a useful taxonomy of `itertools` attributes.

The online docs also offer *recipes*, ways to combine and use `itertools` attributes. The recipes assume you have `from itertools import *` at the top of your module; this is *not* recommended use, just an assumption to make the recipes' code more compact. It's best to `import itertools as it`, then use references such as `it.`*`something`* rather than the more verbose `itertools.`*`something`*.

accumulate	`accumulate(`*`seq`*`,`*`func`*`[,`*`initial=init`*`])` Similar to `functools.reduce(`*`func`*`,`*`seq`*`)`, but returns an iterator of all the intermediate computed values, not just the final value. ||**3.8**++|| You can also pass an initial value *`init`*, as described in `functools.reduce`.
chain	`chain(*`*`iterables`*`)` Yields items from the first argument, then items from the second argument, and so on until the end of the last argument: that's just like the generator expression: (*it* **for** *itble* **in** *iterables* **for** *it* **in** *itble*)
chain.from_it	`chain.from_iterable(`*`iterables`*`)` Yields items from the iterables in the argument, in order, just like the genexp: (*it* **for** *itble* **in** *iterables* **for** *it* **in** *itble*)

erable

compress	`compress(`*`data`*`,` *`conditions`*`)` Yields each item from *`data`* corresponding to a true item in *`conditions`*, just like the generator expression: (*it* **for** *it*, **cond in** zip(*data*, *conditions*) **if** *cond*)

count	`count(`*`start`*`=0,` *`step`*`=1)`
	Yields consecutive integers starting from *`start`*, just like the generator:
	
	`def count(start=0, step=1):`
	`while True:`
	`  yield start`
	`  start += step`

cycle	`cycle(`*`iterable`*`)`
	Yields each item of *`iterable`*, endlessly repeating items from the beginning each time it reaches the end, just like the generator:
	
	`def cycle(iterable):`
	`saved = []`
	`for item in iterable:`
	`  yield item`
	`  saved.append(item)`
	`while saved:`
	`  for item in saved: yield item`

dropwhile	`dropwhile(`*`func`*`,` *`iterable`*`)`
	Drops the `0+` leading items of *`iterable`* for which *`func`* is true, then yields each other item, just like the generator:
	
	`def dropwhile(func, iterable):`
	`iterator = iter(iterable)`
	`for item in iterator:`
	`  if not func(item):`
	`  yield item`
	`  break`
	`for item in iterator:`
	`  yield item`

filterfalse	`filterfalse(`*`func`*`,` *`iterable`*`)` Yields those items of *`iterable`* for which *`func`* is false, just like the genexp: (*it* **for** *it* **in** *iterable* **if not** *func(it)*) *`func`* can be any callable accepting a single argument, or `None`. When

func is None, `filterfalse` yields false items, just like the genexp: (*it* **for** *it* **in** *iterable* **if not** *it*)

groupby

`groupby(`*`iterable`*`,`*`key`*`=None)`

`iterable` normally needs to be already sorted according to *`key`* (`None`, as usual, standing for the identity function). `groupby` yields pairs *`(k,g)`*, each pair representing a *group* of adjacent items from *`iterable`* having the same value *`k`* for *`key(item)`*; each *`g`* is an iterator yielding the items in the group. When the `groupby` object advances, previous iterators *`g`* become invalid (so, if a group of items needs to be processed later, you'd better store somewhere a `list` "snapshot" of it, `list(`*`g`*`)`).

Another way of looking at the groups `groupby` yields is that each terminates as soon as *`key(item)`* changes (which is why you normally call `groupby` only on an *`iterable`* that's already sorted by *`key`*).

For example, suppose that, given a `set` of lowercase words, we want a `dict` that maps each initial to the longest word having that initial (with "ties" broken arbitrarily):

```
import itertools as it
import operator
def set2dict(aset):
  first = operator.itemgetter(0)
  words = sorted(aset, key=first)
  adict = {}
  for initial, group in it.groupby(
    words, key=first)):
      adict[initial] = max(
    group, key=len)
  return adict
```

islice

`islice(`*`iterable`*`[, `*`start`*`], `*`stop`*`[,`*`step`*`])`

Yields items of *`iterable`*, skipping the first *`start`* ones (default `0`), until the *`stop`*th one excluded, advancing by steps of *`step`* (default `1`) at a time. All arguments must be nonnegative integers (or `None`), and *`step`* must be `>0`. Apart from checks and optional arguments, it's like the generator:

```
def islice(iterable,start,stop,step=1):
  en = enumerate(iterable)
  n = stop
  for n, item in en:
    if n>=start:
      break
  while n<stop:
    yield item
    for x in range(step):
      n, item = next(en)
```

pairwise	`pairwise(`*`seq`*`)` **‖3.10++‖** Yields pairs of items in *`seq`*, with overlap (for example, `pairwise('ABCD')` will yield `'AB'`, `'BC'` and `'CD'`). Equivalent to the iterator returned from `zip(`*`seq`*`, `*`seq`*`[1:])`.

repeat	`repeat(`*`item`*`[, `*`times`*`])`
	Repeatedly yields *`item`*, just like the genexp:
	(item **for** x **in** range(times))
	When *`times`* is absent, the iterator is unbounded, yielding a potentially infinite number of items, which are all the object *`item`*. Just like the generator:
	`def repeat_unbounded(item):`
	`  while True:`
	`  yield item`

starmap	`starmap(`*`func`*`, `*`iterable`*`)`
	Yields `func(*`*`item`*`)` for each *`item`* in *`iterable`* (each *`item`* must be an iterable, normally a tuple), just like the generator:
	`def starmap(func, iterable):`
	`  for item in iterable:`
	`  yield func(*item)`

takewhile	`takewhile(`*`func`*`, `*`iterable`*`)`
	Yields items from *`iterable`* as long as `func(`*`item`*`)` is true, then finishes, just like the generator:
	`def takewhile(func, iterable):`
	`  for item in iterable:`
	`  if func(item):`
	`  yield item`
	`  else:`
	`  break`

tee	`tee(`*`iterable`*`,`*`n`*`=2)` Returns a tuple of *`n`* independent iterators, each yielding items that are the same as those of *`iterable`*. The returned iterators are independent from each other, but they are *not* independent from *`iterable`*; avoid altering the object *`iterable`* in any way, as long as you're still using any of the returned iterators.
zip_longest	`zip_longest(*`*`iterables`*`,`*`fillvalue`*`=None)` Yields tuples with one corresponding item from each of the *`iterables`*; stops when the longest of the *`iterables`* is exhausted, behaving as if each

of the others was “padded” to that same length with references to *`fillvalue`*.

We have shown equivalent generators and genexps for many attributes of `itertools`, but it’s important to remember the sheer speed of `itertools`! As a trivial example, consider repeating some action 10 times:

```
for _ in itertools.repeat(None, 10): pass
```

This turns out to be about 10 to 20 percent faster, depending on Python release and platform, than the straightforward alternative:

```
for _ in range(10): pass
```

---

1 When first introduced, defaultdict(int) was also useful to maintain counts of items. Since Counter is now also in the collections module, prefer Counter to defaultdict(int) for the specific task of counting items.

2 this step is not *quite* a full “sort”, but it looks close enough to name it as such, at least if you squint.

Chapter 8. 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.

In Table 8-1, for conciseness, we use `sys.maxsize` for integer default values meaning, in practice, "any number, no matter how large."

Table 8-1. 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.
center	*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 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.
isalnum	*s*.isalnum() Returns True when len(*s*) is greater than 0 and all characters in *s* are 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.
	s.isalpha() Returns True when len(*s*) is greater than 0 and all characters in *s* are

isalpha	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`).
	s`.isupper()` Returns `True` when all letters in *s* are uppercase. When *s* contains no letters,

isupper	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 8-2. 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 8-3. Sign indicators

	Significance as sign indicator

Character	
`'+'`	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 8-4. 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 8-5. 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 8-6. 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 8-7. 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 9. 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 9-1. 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 9-2. 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 9-3. 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 9-4. 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 10. Persistence and Databases

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 11th 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 supports several ways of persisting data. One way, *serialization*, views data as a collection of Python objects. These objects can be *serialized* (saved) to a byte stream, and later *deserialized* (loaded and re-created) back from the byte stream. *Object persistence* lives on top of serialization, adding features such as object naming. This chapter covers the Python modules that support serialization and object persistence.

Another way to make data persistent is to store it in a database (DB). One simple category of DBs are file formats using *keyed access* to enable selective reading and updating of parts of the data. This chapter covers Python standard library modules that support several variations of such a file format, known as *DBM*.

A *relational DB management system* (RDBMS), such as PostgreSQL or Oracle, offers a more powerful approach to storing, searching, and retrieving persistent data. Relational DBs rely on dialects of *Structured Query Language* (SQL) to create and alter a DB's schema, insert and update data in the DB, and query the DB with search criteria. (This book does not provide reference material on SQL. We recommend *SQL in a Nutshell*, by

Kevin Kline [O'Reilly].) Unfortunately, despite the existence of SQL standards, no two RDBMSes implement exactly the same SQL dialect.

The Python standard library does not come with an RDBMS interface. However, many third-party modules let your Python programs access a specific RDBMS. Such modules mostly follow the Python Database API 2.0 standard, also known as the *DBAPI*. This chapter covers the DBAPI standard and mentions a few of the most popular third-party modules that implement it.

A DBAPI module that is particularly handy—because it comes with every standard installation of Python—is sqlite3, which wraps SQLite, "a self-contained, server-less, zero-configuration, transactional SQL DB engine," which is the most widely deployed relational DB engine in the world. We cover `sqlite3` in "SQLite".

Besides relational DBs, and the simpler approaches covered in this chapter, there exist Python-specific object DBs such as ZODB, as well as many NoSQL DBs, each with Python interfaces. We do not cover advanced nonrelational DBs in this book.

Serialization

Python supplies several modules to *serialize* (save) Python objects to various kinds of byte streams and *deserialize* (load and re-create) Python objects back from streams. Serialization is also known as *marshaling*, or as "formatting for *data interchange*"

Serialization approaches span a vast range: all the way from (by now) language-independent JSON to (low-level, Python-version-specific) `marshal`, both limited to elementary data types; through richer but Python-specific `pickle`; to pretty rich cross-language formats such as XML, YAML, protocol buffers, and MessagePack.

In this section, we cover `csv`, JSON, and `pickle`. We cover XML in Chapter 23. `marshal` is too low-level to use in applications; should you need to maintain old code using it, refer to the online docs. As for protocol

buffers, MessagePack, YAML, and other data-interchange/serialization approaches (each with specific advantages and weaknesses), we cannot cover everything in this book; we recommend studying them via the resources available on the web.

The csv Module

While the CSV (standing for *comma-separated values*[1]) format isn't usually considered as a serialization: it's a widely-used and convenient interchange format for tabular data. Since much data is tabular, CSV data is used a lot despite some lack of agreement on exactly how it should be represented in files. In order to overcome this issue, `csv` provides a number of *dialects* (specifications of the way particular sources encode CSV data) and lets you define your own dialects. You can register additional dialects, and list the available dialects by calling the `csv.list_dialects` function. For further information on dialects, consult the module's documentation.

The `csv` module provides two kinds of readers and writers, to let you handle CSV data rows in Python as either lists or dictionaries.

Functions and Classes of csv

The `csv` module exposes the following functions and classes:

reader	`reader(csvfile, dialect='excel', **kw)` Creates and returns a reader object *r*. *csvfile* can be any iterable object yielding text rows as `str`s (usually a list of lines or a file opened with *newline*=''); *dialect* is the name of a registered dialect. To modify the dialect, add named arguments: their values override dialect fields of the same name. Iterating over *r* yields a sequence of lists, each containing the elements from one row of *csvfile*.
writer	`writer(csvfile, dialect='excel', **kw)` Creates and returns a writer object *w*. *csvfile* is an object with a `write` method (if a file, open it with *newline*=''); *dialect* is the name of a registered dialect. To modify the dialect, add named arguments: their values override dialect fields of the same name. *w.writerow* accepts a sequence of values and writes their CSV representation as a row to *csvfile*. *w.writerows* accepts an iterable of such lists and calls *w.writerow* on each. You are responsible for closing *csvfile*.

DictReader	`DictReader(csvfile, fieldnames=None,restkey=None,restval=None, dialect='excel', *args,**kw)` Creates and returns an object *`r`* that iterates over *`csvfile`* to generate an iterable of dictionaries (‖3.8--‖ `OrderedDicts`), one for each row. When the *`fieldnames`* argument is given, it is used to name the fields in *`csvfile`*; otherwise, the field names are taken from the first row of *`csvfile`*. If a row contains more columns than field names, the extra values are saved as a list with the key *`restkey`*. If there are insufficient values in any row then those column values will be set to *`restval`*. *`dialect`*, *`kw`*, and *`args`* are passed to the underlying reader object.
DictWriter	`DictWriter(csvfile, fieldnames,restval='', extrasaction='raise', dialect='excel', *args, **kwds)` Creates and returns an object *`w`* whose `writerow` and `writerows` methods take a dictionary or iterable of dictionaries and write them using the *`csvfile`*'s `write` method. *`fieldnames`* is a sequence of str, the keys to the dictionaries. *`restval`* is the value used to fill up a dictionary that's missing some keys. *`extrasaction`* specifies what to do when a dictionary has extra keys not listed in *`fieldnames`*: when `'raise'`, the default, the function raises a `ValueError` in such cases; when `'ignore'`, the function ignores such errors. *`dialect`*, *`kw`*, and *`args`* are passed to the underlying reader object. You are responsible for closing *`csvfile`* (usually a file opened with *`newline`*`=''`)

A CSV example

Here is a simple example using `csv` to read color data from a list of strings:

```
import csv
color_data = '''\
color,r,g,b
red,255,0,0
green,0,255,0
blue,0,0,255
cyan,0,255,255
magenta,255,0,255
yellow,255,255,0
'''.splitlines()
colors = {row['color']: row
```

```
        for row in csv.DictReader(color_data)}
print(colors['red']) # prints {'color': 'red', 'r': '255', 'g':
'0', 'b': '0'}
```

Note that the integer values are read as strings. `csv` does not do any data conversion; that needs to be done by your program code with the `dicts` returned from `DictReader`.

The json Module

The standard library's `json` module supplies four key functions:

dump

`dump(`*`value`*`,`*`fileobj`*`,`*`skipkeys`*`=False,`*`ensure_ascii`*`=True, `*`check_circular`*`=True,`*`allow_nan`*`=True,`*`cls`*`=JSONEncoder,`*`indent`*`=None, `*`separators`*`=(', ', ': '),`*`default`*`=None, `*`sort_keys`*`=False,**kw)`

`dump` writes the JSON serialization of object *`value`* to file-like object *`fileobj`*, which must be opened for writing in text mode, via calls to *`fileobj`*`.write`. Each call to *`fileobj`*`.write` passes as argument a text (Unicode) string.

When *`skipkeys`* is `True` (by default, it's `False`), `dict` keys that are not scalars (i.e., are not of types `bool`, `float`, `int`, `str`, or `None` raise an exception. In any case, keys that *are* scalars are turned into strings (e.g., `None` becomes `'null'`): JSON only allows strings as keys in its mappings.

When *`ensure_ascii`* is `True` (as it is by default), all non-ASCII characters in the output are escaped; when it's `False`, they're output as-is.

When *`check_circular`* is `True` (as it is by default), containers in *`value`* are checked for circular references, raising a `ValueError` exception if circular references are found; when it's `False`, the check is skipped, and many different exceptions can get raised as a result (even a crash is possible).

When *`allow_nan`* is `True` (as it is by default), float scalars `nan`, `inf`, and `-inf` are output as their respective JavaScript equivalents, `NaN`, `Infinity`, `-Infinity`; when it's `False`, the presence of such scalars raises a `ValueError`.

You can optionally pass *`cls`* in order to use a customized subclass of `JSONEncoder` (such advanced customization is rarely needed and we don't cover it in this book); in this case, *`**kw`* gets passed in the call to *`cls`* which instantiates it. By default, encoding uses the `JSONEncoder` class directly.

When *`indent`* is an `int>0`, `dump` "pretty-prints" the output by prepending that many spaces to each array element and object member; when an `int<=0`, `dump` just inserts `\n` characters; when `None`, which is the default, `dump` uses the most compact representation. *`indent`* can also be a `str`—for example, `'\t'`—and in that case `dump` uses that string for indenting.

`separators` must be a tuple with two items, respectively the strings used to separate items, and keys from values. You can explicitly pass

`separators=(',',':')` to ensure `dump` inserts no whitespace.
You can optionally pass *`default`* in order to transform some otherwise non-serializable objects into serializable ones; *`default`* is a function, called with a single argument that's a non-serializable object, and must return a serializable object or else raise `ValueError` (by default, the presence of non-serializable objects raises `ValueError`).
When *`sort_keys`* is `True` (by default, it's `False`), mappings are output in sorted order of their keys; when it's `False`, they're output in whatever is their natural order of iteration (nowadays, for most mappings, insertion order).

dumps

`dumps(`*`value`*`,`*`skipkeys`*`=False,`*`ensure_ascii`*`=True,`*`check_circular`*`=True,`
`allow_nan``=True,`*`cls`*`=JSONEncoder,`*`indent`*`=None,`
`separators``=(', ', ': '),`
`default``=None,`*`sort_keys`*`=False,`*`**kw`*`)`
`dumps` returns the string that's the JSON serialization of object *`value`*—that is, the string that `dump` would write to its file-object argument. All arguments to `dumps` have exactly the same meaning as the arguments to `dump`.

JSON Serializes Just One Object Per File

JSON is not what is known as a framed format: this means it is not possible to call `dump` more than once in order to serialize multiple objects into the same file, nor later call `load` more than once to deserialize the objects, as would be possible, for example, with `pickle`. So, technically, JSON serializes just one object per file. However, you can make that one object be a `list`, or `dict`, which in turn can contain as many items as you wish.

load

`load(`*`fileobj`*`,`*`encoding`*`='utf-8',`*`cls`*`=JSONDecoder,`*`object_hook`*`=None,`
`parse_float``=float,`*`parse_int`*`=int,`*`parse_constant`*`=None,`
`object_pairs_hook``=None,`*`** kw`*`)`

`load` creates and returns the object *`v`* previously serialized into file-like object *`fileobj`*, which must be opened for reading in text mode, getting *`fileobj`*'s contents via a call to *`fileobj`*`.read`. The call to *`fileobj`*`.read` must return a text (Unicode) string.
The functions `load` and `dump` are complementary. In other words, a single call to `load(`*`f`*`)` deserializes the same value previously serialized when *`f`*'s

contents were created by a single call to `dump(v,f)` (possibly with some alterations: e.g., all dictionary keys are turned into strings).
You can optionally pass *`cls`* in order to use a customized subclass of `JSONDecoder` (such advanced customization is rarely needed and we don't cover it in this book); in this case, *`**kw`* gets passed in the call to *`cls`*, which instantiates it. By default, decoding uses the `JSONDecoder` class directly.
You can optionally pass *`object_hook`* or *`object_pairs_hook`* (if you pass both, *`object_hook`* is ignored and only *`object_pairs_hook`* is used), a function that lets you implement custom decoders. When you pass *`object_hook`* but not *`object_pairs_hook`*, then, each time an object is decoded into a `dict`, `load` calls *`object_hook`* with the `dict` as the only argument, and uses *`object_hook`*'s return value instead of that `dict`. When you pass *`object_pairs_hook`*, then, each time an object is decoded, `load` calls *`object_pairs_hook`* with, as the only argument, a list of the pairs of *`(key, value)`* items of the object, in the order in which they are present in the input, and uses *`object_pairs_hooks`*' return value; this lets you perform specialized decoding that potentially depends on the order of *`(key, value)`* pairs in the input.
`parse_float, parse_int`, and *`parse_constant`* are functions called with a single argument that's a `str` representing a `float`, an `int`, or one of the three special constants `'NaN'`, `'Infinity'`, `'-Infinity'`; `load` calls the appropriate function each time it identifies in the input a `str` representing a number, and uses the function's return value. By default, *`parse_float`* is `float`, *`parse_int`* is `int`, and *`parse_constant`* returns one of the three special float scalars `nan`, `inf`, `-inf`, as appropriate. For example, you could pass `parse_float=decimal.Decimal` to ensure that all numbers in the result that would normally be `float`s are instead decimals (covered in "The decimal Module").

loads

```
loads(s,cls=JSONDecoder,object_hook=None,parse_float=float,
parse_int=int,parse_constant=None,object_pairs_hook=None,**kw)
```

`loads` creates and returns the object *`v`* previously serialized into the string *`s`*. All arguments to `loads` have exactly the same meaning as the arguments to `load`.

A JSON example

Say you need to read several text files, whose names are given as your program's arguments, recording where each distinct word appears in the files. What you need to record for each word is a list of *`(filename, linenumber)`* pairs. The following example uses the `fileinput`

module to iterate through all the files given as program arguments, and `json` to encode the lists of *`(filename, linenumber)`* pairs as strings and store them in a DBM-like file (as covered in "DBM Modules"). Since these lists contain tuples, each containing a string and a number, they are within `json`'s abilities to serialize.

```
import collections, fileinput, json, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        word_pos[word].append(pos)
with dbm.open('indexfilem', 'n') as dbm_out:
    for word, word_positions in word_pos.items():
        dbm_out[word] = json.dumps(word_positions)
```

We need `json` to deserialize the data stored in the DBM-like file *indexfilem*, as in the following example:

```
import sys, json, dbm, linecache
with dbm.open('indexfilem') as dbm_in:
    for word in sys.argv[1:]:
        if word not in dbm_in:
             print(f'Word {word!r} not found in index
file',file=sys.stderr)
             continue
        places = json.loads(dbm_in[word])
        for fname, lineno in places:
            print(f'Word {word!r} occurs in line {lineno} of file
{fname!r}:')
            print(linecache.getline(fname, lineno), end='')
```

The pickle Module

The `pickle` module supplies factory functions, named `Pickler` and `Unpickler`, to generate objects that wrap file-like objects and supply Python-specific serialization mechanisms. Serializing and deserializing via these modules is also known as *pickling* and *unpickling*. `Pickler` and `Unpickler` are factory functions that generate instances of non-subclassable types, not classes.

Serialization shares some of the issues of deep copying, covered in “The copy Module”. The `pickle` module deals with these issues in much the same way as the `copy` module does. Serialization, like deep copying, implies a recursive walk over a directed graph of references. `pickle` preserves the graph’s shape: when the same object is encountered more than once, the object is serialized only the first time, and other occurrences of the same object serialize references to that single value. `pickle` also correctly serializes graphs with reference cycles. However, this means that if a mutable object `o` is serialized more than once to the same `Pickler` instance `p`, any changes to `o` after the first serialization of `o` to `p` are not saved.

Don’t Alter Objects While Their Serialization Is In Progress

For clarity, correctness, and simplicity, don’t alter objects that are being serialized while serialization to a `Pickler` instance is in progress.

`pickle` can serialize with a legacy ASCII protocol or with one of a few compact binary ones (better ones are sometimes added in newer versions of Python).

Protocol	Format	Python2 compatible	Description
0	ASCII	Y	Human-readable format, slow to serialize/deserialize.
1	binary	Y	Early binary format, superseded by protocol 2.
2	binary	Y	Improved support for later Python2 features.
3	binary	N	(default) Added specific support for `bytes` objects.
4	bin	N	(‖3.8++‖ default) includes support for very large objects

	ary		
5	bin ary	N	||3.8++|| Adds features to support pickling as serialization for transport between processes, per PEP 574.

Always pickle With Protocol 2 or Higher

Always use at least protocol `2`. The size and speed savings can be substantial, and binary format has basically no downside except loss of compatibility of resulting pickles with truly ancient versions of Python.

When you reload objects, `pickle` transparently recognizes and uses any protocol that the Python version you're currently using supports.

`pickle` serializes classes and functions by name, not by value[2]. `pickle` can therefore deserialize a class or function only by importing it from the same module where the class or function was found when `pickle` serialized it. In particular, `pickle` can normally serialize and deserialize classes and functions only if they are top-level names for their module (i.e., attributes of their module). For example, consider the following:

```
def adder(augend):
    def inner(addend, augend=augend):
        return addend+augend
    return inner
plus5 = adder(5)
```

This code binds a closure to name `plus5` (as covered in "Nested functions and nested scopes")—a nested function `inner` plus an appropriate outer scope. Therefore, trying to pickle `plus5` raises an `AttributeError`: a function can be pickled only when it is top-level, and the function `inner`, whose closure is bound to the name `plus5` in this code, is not top-level but rather nested inside the function `adder`. Similar issues apply to pickling nested functions and nested classes (i.e., classes not at top-level).

Functions and classes of pickle

The `pickle` module exposes the following functions and classes:

dump, dumps	`dump(value, fileobj,protocol=None,bin=None)`
	`dumps(value, protocol=None, bin=None)`
	
	`dumps` returns a bytestring representing object *`value`*. `dump` writes the same string to the file-like object *`fileobj`*, which must be opened for writing. `dump(v,f)` is like `f.write(dumps(v))`. Do not pass the *`bin`* parameter, which exists only for compatibility with old versions of Python. The *`protocol`* parameter can be `0` (ASCII output, slowest and bulkiest), or a larger `int` for various kinds of binary output. Unless *`protocol`* is `0` (ASCII output), the *`fileobj`* parameter to dump must be open for binary writing.

load, loads	`load(fileobj)`
	`loads(s,*,fix_imports=True, encoding="ASCII", errors="strict")`
	The functions `load` and `dump` are complementary. In other words, a sequence of calls to `load(f)` deserializes the same values previously serialized when *`f`*'s contents were created by a sequence of calls to `dump(v,f)`. `load` reads the right number of bytes from file-like object *`fileobj`* and creates and returns the object *`v`* represented by those bytes. `load` and `loads` transparently support pickles performed in any binary or ASCII protocol. If data is pickled in any binary format, the file must be open as binary for both `dump` and `load`. `load(f)` is like `Unpickler(f).load()`.
	`loads` creates and returns the object *`v`* represented by byte string *`s`*, so that for any object *`v`* of a supported type, *`v`*`==loads(dumps(v))`. If s is longer than `dumps(v)`, `loads` ignores the extra bytes. Optional arguments `fix_imports`, `encoding`, and `errors`, are provided for handling streams generated by Python 2 code. See the `pickle.loads` documentation.

Never Unpickle Untrusted Data

Unpickling from an untrusted data source is a security risk; an attacker could exploit this vulnerability to execute arbitrary code.

Pickler	`Pickler(`*`fileobj`*`,`*`protocol`*`=None,`*`bin`*`=None)` Creates and returns an object *`p`* such that calling *`p`*`.dump` is equivalent to calling the function `dump` with the *`fileobj, protocol`*, and *`bin`* arguments passed to `Pickler`. To serialize many objects to a file, `Pickler` is more convenient and faster than repeated calls to `dump`. You can subclass `pickle.Pickler` to override `Pickler` methods (particularly the method `persistent_id`) and create your own persistence framework. However, this is an advanced issue and is not covered further in this book.
Unpickler	`Unpickler(`*`fileobj`*`)` Creates and returns an object *`u`* such that calling the *`u`*`.load` is equivalent to calling function `load` with the *`fileobj`* argument passed to `Unpickler`. To deserialize many objects from a file, `Unpickler` is more convenient and faster than repeated calls to the function `load`. You can subclass `pickle.Unpickler` to override `Unpickler` methods (particularly method `persistent_load`) and create your own persistence framework. However, this is an advanced issue and is not covered further in this book.

A pickling example

The following example handles the same task as the `json` example shown earlier, but uses `pickle` instead of `json` to serialize lists of *`(filename, linenumber)`* pairs as strings:

```
import collections, fileinput, pickle, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        word_pos[word].append(pos)
with dbm.open('indexfilep', 'n') as dbm_out:
    for word, word_positions in word_pos.items():
        dbm_out[word] = pickle.dumps(word_positions, protocol=2)
```

We can then use `pickle` to read back the data stored to the DBM-like file *indexfilep*, as shown in the following example:

```
import sys, pickle, dbm, linecache
with dbm.open('indexfilep') as dbm_in:
    for word in sys.argv[1:]:
        if word not in dbm_in:
            print(f'Word {word!r} not found in index
file',file=sys.stderr)
            continue
        places = pickle.loads(dbm_in[word])
```

```
        for fname, lineno in places:
            print(f'Word {word!r} occurs in line {lineno} of file
{fname!r}:')
            print(linecache.getline(fname, lineno), end='')
```

Pickling instances

In order for `pickle` to reload an instance *`x`*, `pickle` must be able to import *`x`*'s class from the same module in which the class was defined when `pickle` saved the instance. Here is how `pickle` saves the state of instance object *`x`* of class *`T`* and later reloads the saved state into a new instance *`y`* of *`T`* (the first step of the reloading is always to make a new empty instance *`y`* of *`T`*, except where we explicitly say otherwise in the following):

- When *`T`* supplies the method `__getstate__`, `pickle` saves the result *`d`* of calling *`T`*`.__getstate__(`*`x`*`)`.
- When *`T`* supplies the method `__setstate__`, *`d`* can be of any type, and `pickle` reloads the saved state by calling *`T`*`.__setstate__(`*`y`*`,` *`d`*`)`.
- Otherwise, *`d`* must be a dictionary, and `pickle` just sets *`y`*`.__dict__ =` *`d`*.
- Otherwise, when *`T`* supplies the method `__getnewargs__`, and `pickle` is pickling with protocol 2 or higher, `pickle` saves the result *`t`* of calling *`T`*`.__getnewargs__(`*`x`*`)`; *`t`* must be a tuple.
- `pickle`, in this one case, does not start with an empty *`y`*, but rather creates *`y`* by executing *`y`* = *`T`*`.__new__(`*`T`*`, *`*`t`*`)`, which concludes the reloading.
- Otherwise, by default, `pickle` saves as *`d`* the dictionary *`x`*`.__dict__`.

- When *T* supplies the method `__setstate__`, `pickle` reloads the saved state by calling *T*`.__setstate__(`*y*`, `*d*`)`.
- Otherwise, `pickle` just sets *y*`.__dict__ = `*d*.

All the items in the *d* or *t* object that `pickle` saves and reloads (normally a dictionary or tuple) must, in turn, be instances of types suitable for pickling and unpickling (AKA *pickleable* objects), and the procedure just outlined may be repeated recursively, if necessary, until `pickle` reaches primitive pickleable built-in types (dictionaries, tuples, lists, sets, numbers, strings, etc.).

As mentioned in "The copy Module", the special methods `__getnewargs__`, `__getstate__`, and `__setstate__` also control the way instance objects are copied and deep-copied. If a class defines `__slots__`, and therefore its instances do not have a `__dict__`, `pickle` does its best to save and restore a dictionary equivalent to the names and values of the slots. However, such a class should define `__getstate__` and `__setstate__`; otherwise, its instances may not be correctly pickleable and copy-able through such best-effort endeavors.

Pickling customization with the copy_reg module

You can control how `pickle` serializes and deserializes objects of an arbitrary type by registering factory and reduction functions with the module `copy_reg`. This is particularly, though not exclusively, useful when you define a type in a C-coded Python extension. The `copy_reg` module supplies the following functions:

constructor	`constructor(`*fcon*`)` Adds *fcon* to the table of constructors, which lists all factory functions that `pickle` may call. *fcon* must be callable and is normally a function.
pickle	`pickle(`*type*`,`*fred*`,`*fcon*`=None)` Registers function *fred* as the *reduction function* for type *type*, where *type* must be a type object. To save an object *o* of type *type*, the module `pickle` calls *fred(o)* and saves the result. *fred(o)* must return a tuple *(fcon, t)* or *(fcon, t, d)*, where *fcon* is a constructor and *t* is a

tuple. To reload *o*, `pickle` uses *`o=fcon(*t)`*. Then, when *`fred`* also returned a *d*, `pickle` uses *d* to restore *o*'s state (when *o* supplies `__setstate__`, *`o.__setstate__(d)`*; otherwise, *`o.__dict__.update(d)`*), as in "Pickling instances". If *`fcon`* is not `None`, `pickle` also calls `constructor(`*`fcon`*`)` to register *`fcon`* as a constructor.

`pickle` does not support pickling of code objects, but `marshal` does. Here's how you could customize pickling to support code objects by delegating the work to `marshal` thanks to `copy_reg`:

```
>>> import pickle, copy_reg, marshal
>>> def marsh(x):
    return marshal.loads, (marshal.dumps(x),)
...
>>> c=compile('2+2','','eval')
>>> copy_reg.pickle(type(c), marsh)
>>> s=pickle.dumps(c, 2)
>>> cc=pickle.loads(s)
>>> print(eval(cc))
4
```

WARNING

Using `marshal` makes your code Python-version-dependent

Beware using `marshal` in your code, as the preceding example does. `marshal`'s serialization isn't guaranteed to be stable across versions, so using `marshal` means you may be unable to load objects, serialized with one Python version, with other versions.

The shelve Module

The `shelve` module orchestrates the modules `pickle`, `io`, and `dbm` (and its underlying modules for access to DBM-like archive files, as discussed in "DBM Modules") in order to provide a simple, lightweight persistence mechanism.

`shelve` supplies a function `open` that is polymorphic to `dbm.open`. The mapping *s* returned by `shelve.open` is less limited than the mapping *a* returned by `dbm.open`. *a*'s keys and values must be strings. *s*'s keys must also be strings, but *s*'s values may be of any pickleable types. `pickle` customizations (`copy_reg`, `__getnewargs__`, `__getstate__`, and

`__setstate__`) also apply to `shelve`, as `shelve` delegates serialization to `pickle`. Keys and values are stored as bytes. When strings are used they are implicitly converted to the default encoding before being stored.

Beware of a subtle trap when you use `shelve` with mutable objects: when you operate on a mutable object held in a shelf, the changes aren't stored back unless you assign the changed object back to the same index. For example:

```
import shelve
s = shelve.open('data')
s['akey'] = list(range(4))
print(s['akey'])           # prints: [0, 1, 2, 3]
s['akey'].append(9)        # trying direct mutation
print(s['akey'])           # doesn't "take"; prints: [0, 1, 2, 3]
x = s['akey']              # fetch the object
x.append(9)                # perform mutation
s['akey'] = x              # key step: store the object back!
print(s['akey'])           # now it "takes", prints: [0, 1, 2, 3,
9]
```

You can finesse this issue by passing the named argument `writeback=True` when you call `shelve.open`, but this can seriously impair the performance of your program.

A shelving example

The following example handles the same task as the earlier `json` and `pickle` examples, but uses `shelve` to persist lists of *`(filename, linenumber)`* pairs:

```
import collections, fileinput, shelve
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = fileinput.filename(), fileinput.filelineno()
    for word in line.split():
        word_pos[word].append(pos)
with shelve.open('indexfiles','n') as sh_out:
    sh_out.update(word_pos)
```

We must use `shelve` to read back the data stored to the DBM-like file *indexfiles*, as shown in the following example:

```
import sys, shelve, linecache
with shelve.open('indexfiles') as sh_in:
    for word in sys.argv[1:]:
        if word not in sh_in:
            print(f'Word {word!r} not found in index
file',file=sys.stderr)
            continue
        places = sh_in[word]
        for fname, lineno in places:
            print(f'Word {word!r} occurs in line {lineno} of file
{fname!r}:')
            print(linecache.getline(fname, lineno), end='')
```

These two examples are the simplest and most direct of the various equivalent pairs of examples shown throughout this section. This reflects the fact that the module `shelve` is higher-level than the modules used in previous examples.

DBM Modules

DBM, a long-time Unix tradition, is a family of libraries supporting data files with pairs of strings *`(key, data)`*, with fast fetching and storing of the data given a key, a usage pattern known as *keyed access*. Keyed access, while nowhere as powerful as the data-access functionality of relational DBs, imposes less overhead, yet may suffice for a given program's needs. If DBM-like files are sufficient, you may end up with a program that is smaller and faster than one using a relational DB.

DBM support in Python's standard library is organized in a clean and elegant way: the package `dbm` exposes two general functions, and within the same package live other modules supplying specific implementations.

The dbm Package

The `dbm` package supplies the following two top-level functions:

open

`open(`*`filepath`*`,`*`flag`*`='r',`*`mode`*`=0o666)`

Opens or creates the DBM file named by *`filepath`* (any path to a file) and returns a mapping object corresponding to the DBM file. When the DBM file already exists, `open` uses the function `whichdb` to determine which DBM submodule can handle the file. When `open` creates a new DBM file, it chooses the first available `dbm` submodule in the following order of preference: `gnu`, `ndbm`, `dumb`.

`flag` is a one-character string that tells `open` how to open the file and whether to create it, as shown in Table 11-1. *`mode`* is an integer that `open` uses as the file's permission bits if `open` creates the file, as covered in "Creating a "file" Object with `io.open`". Flag values for *dbm.open*:

```
Flag   Read-only?   If file exists           If
file does not exsit
`r`    Yes          open opens the file.     open
raises error.
`w`    No           open opens the file.     open
raises error.
`c`    No           open opens the file.     open
creates the file.
`n`    No           open truncates the file. open
creates the file.
```

`dbm.open` returns a mapping object *`m`* with a subset of the functionality of dictionaries (covered in "Dictionary Operations"). *`m`* only accepts strings as keys and values, and the only nonspecial mapping methods *`m`* supplies are *`m`*`.get`, *`m`*`.keys`, and *`m`*`.setdefault`. You can bind, rebind, access, and unbind items in *`m`* with the same indexing syntax *`m`*`[`*`key`*`]` that you would use if *`m`* were a dictionary. If *`flag`* is `'r'`, *`m`* is read-only, so that you can only access *`m`*'s items, not bind, rebind, or unbind them. You can check if a string *`s`* is a key in *`m`* with the usual expression *`s`* **in** *`m`*; you cannot iterate directly on *`m`*, but you can, equivalently, iterate on *`m`*`.keys()`.

One extra method that *`m`* supplies is *`m`*`.close`, with the same semantics as the `close` method of a "file" object. Just like for "file" objects, you should ensure *`m`*`.close()` is called when you're done using *`m`*. The `try/finally` statement (covered in "try/finally") is one way to ensure finalization, but the `with` statement, covered in "The with Statement", is even better (you can use `with`, since *`m`* is a context manager).

whichdb

`whichdb(`*`filepath`*`)`

Opens and reads the file specified by *`filepath`* to discover which `dbm` submodule created the file. `whichdb` returns `None` when the file does not exist or cannot be opened and read. `whichdb` returns `''` when the file exists and can be opened and read, but it is not possible to determine which `dbm` submodule created the file (typically, this means that the file is not a DBM

file). If it can find out which module can read the DBM-like file, `whichdb` returns a string that names a `dbm` submodule, such as `'dbm.ndbm'`, `'dbm.dumb'`, or `'dbm.gdbm'`.

In addition to these two top-level functions, the `dbm` package contains specific modules, such as `ndbm`, `gnu`, and `dumb`, which provide various implementations of DBM functionality, which you normally access only via the two top-level functions of the package `dbm`. Third-party packages can install further implementation modules in `dbm`.

The only implementation module of the `dbm` package that's guaranteed to exist on all platforms is `dumb`. The `dumb` submodule of `dbm` has minimal DBM functionality and mediocre performance. `dumb`'s only advantage is that you can use it anywhere, since `dumb` does not rely on any library. You don't normally `import dbm.dumb`: rather, `import dbm`, and let `dbm.open` supply the best DBM module available, defaulting to `dumb` if no better sub-module is available on the current Python installation. The only case in which you import `dumb` directly is the rare one in which you need to create a DBM-like file that must be readable in any Python installation. The `dumb` module supplies an `open` function polymorphic to `dbm`'s.

Examples of DBM-Like File Use

DBM's keyed access is suitable when your program needs to record persistently the equivalent of a Python dictionary, with strings as both keys and values. For example, suppose you need to analyze several text files, whose names are given as your program's arguments, and record where each word appears in those files. In this case, the keys are words and, therefore, intrinsically strings. The data you need to record for each word is a list of *`(filename, linenumber)`* pairs. However, you can encode the data as a string in several ways—for example, by exploiting the fact that the path separator string `os.pathsep` (covered in "Path-String Attributes of the os Module") does not normally appear in filenames. (More general approaches to the issue of encoding data as strings are covered with the

same example in “Serialization”.) With this simplification, the program that records word positions in files might be as follows:

```
import collections, fileinput, os, dbm
word_pos = collections.defaultdict(list)
for line in fileinput.input():
    pos = f'{fileinput.filename()}{os.pathsep}
{fileinput.filelineno()}'
    for word in line.split():
        word_pos[word].append(pos)
sep2 = os.pathsep * 2
with dbm.open('indexfile','n') as dbm_out:
    for word in word_pos:
        dbm_out[word.encode('utf-8')] =
sep2.join(word_pos[word]).encode('utf-8')
```

DBM Databases Are Bytes-oriented

Note that the input file is text, but DBM databases require both keys and values to be bytes values, so the input is explicitly encoded in UTF-8 before storage. Similarly, the inverse decoding must be performed when reading back the values.

We can read back the data stored to the DBM-like file *indexfile* in several ways. The following example accepts words as command-line arguments and prints the lines where the requested words appear:

```
import sys, os, dbm, linecache

sep = os.pathsep
sep2 = sep * 2
with dbm.open('indexfile') as dbm_in:
    for word in sys.argv[1:]:
        e_word = word.encode('utf-8')
        if e_word not in dbm_in:
            print(f'Word {word!r} not found in index file',
                  file=sys.stderr)
            continue
        places = dbm_in[e_word].decode('utf-8').split(sep2)
        for place in places:
            fname, lineno = place.split(sep)
            print(f'Word {word!r} occurs in line {lineno} of file
```

```
    {filename}:')
                print(linecache.getline(fname, int(lineno)), end='')
```

Berkeley DB Interfacing

The `bsddb` is no longer in the Python standard library, so we do not recommend it. If you do need to interface to a BSD DB archive, we recommend instead the excellent third-party package bsddb3.

The Python Database API (DBAPI) 2.0

As we mentioned earlier, the Python standard library does not come with an RDBMS interface (except for `sqlite3`, covered in "SQLite", which is a rich implementation, not just an interface). Many third-party modules let your Python programs access specific DBs. Such modules mostly follow the Python Database API 2.0 standard, also known as the DBAPI, as specified in PEP 249.

After importing any DBAPI-compliant module, call the module's `connect` function with DB-specific parameters. `connect` returns *x*, an instance of `Connection`, which represents a connection to the DB. *x* supplies `commit` and `rollback` methods to deal with transactions, a `close` method to call as soon as you're done with the DB, and a `cursor` method to return *c*, an instance of `Cursor`. *c* supplies the methods and attributes used for DB operations. A DBAPI-compliant module also supplies exception classes, descriptive attributes, factory functions, and type-description attributes.

Exception Classes

A DBAPI-compliant module supplies the exception classes `Warning`, `Error`, and several subclasses of `Error`. `Warning` indicates anomalies such as data truncation on insertion. `Error`'s subclasses indicate various kinds of errors that your program can encounter when dealing with the DB

and the DBAPI-compliant module that interfaces to it. Generally, your code uses a statement of the form:

```
try:
    ...
except module.Error as err:
    ...
```

to trap all DB-related errors that you need to handle without terminating.

Thread Safety

When a DBAPI-compliant module has a `threadsafety` attribute greater than `0`, the module is asserting some level of thread safety for DB interfacing. Rather than relying on this, it's usually safer, and always more portable, to ensure that a single thread has exclusive access to any given external resource, such as a DB, as outlined in "Threaded Program Architecture".

Parameter Style

A DBAPI-compliant module has an attribute called `paramstyle` to identify the style of markers used as placeholders for parameters. Insert such markers in SQL statement strings that you pass to methods of `Cursor` instances, such as the method `execute`, to use runtime-determined parameter values. Say, for example, that you need to fetch the rows of DB table *`ATABLE`* where field *`AFIELD`* equals the current value of Python variable *`x`*. Assuming the cursor instance is named *`c`*, you *could* theoretically (but very ill-advisedly!) perform this task with Python's string formatting:

```
c.execute(f'SELECT * FROM ATABLE WHERE AFIELD={x!r}')
```

Avoid SQL Query String Formatting: Use Parameter Substitution

String formatting is not the recommended approach. It generates a different string for each value of *x*, requiring statements to be parsed and prepared anew each time; it also opens up the possibility of security weaknesses such as SQL injection vulnerabilities. With parameter substitution, you pass to `execute` a single statement string, with a placeholder instead of the parameter value. This lets `execute` parse and prepare the statement just once, for better performance; more importantly, parameter substitution improves solidity and security, hampering SQL injection attacks.

For example, when a module's `paramstyle` attribute (described next) is `'qmark'`, express the preceding query as:

```
c.execute('SELECT * FROM ATABLE WHERE AFIELD=?', (some_value,))
```

The read-only string attribute `paramstyle` tells your program how it should use parameter substitution with that module. The possible values of `paramstyle` are shown in Table 11-2:

Paramstyle	
`format`	The marker is %s, as in old-style string formatting (always with s: never use other type indicator letters, whatever the data's type). A query looks like: `c.execute('SELECT * FROM ATABLE WHERE AFIELD=%s', (some_value,))`
`named`	The marker is :name, and parameters are named. A query looks like: `c.execute('SELECT * FROM ATABLE WHERE AFIELD=:x', {'x':some_value})`
	The marker is :n, giving the parameter's number, 1 and up. A query looks like:

`numeric`	`c.execute('SELECT * FROM ATABLE WHERE AFIELD=:1', (some_value,))`
`pyformat`	The marker is %(name)s, and parameters are named. Always use s: never use other type indicator letters, whatever the data's type. A query looks like: `c.execute('SELECT * FROM ATABLE WHERE AFIELD=%(x)s', {'x':some_value})`
`qmark`	The marker is ?. A query looks like: `c.execute('SELECT * FROM ATABLE WHERE AFIELD=?', (x,))`

When parameters are named (i.e., when paramstyle is `'pyformat'` or `'named'`), the second argument of the execute method is a mapping. Otherwise, the second argument is a sequence.

`format` and `pyformat` Only Accept Type Indicator `s`

The *only* valid type indicator letter for `format` or `pyformat` is `s`; neither accepts any other type indicator—for example, never use `%d` nor `%(name)d`. Use `%s` or `%(name)s` for all parameter substitutions, regardless of the type of the data.

Factory Functions

Parameters passed to the DB via placeholders must typically be of the right type: this means Python numbers (integers or floating-point values), strings (bytes or Unicode), and `None` to represent SQL `NULL`. There is no type universally used to represent dates, times, and binary large objects (BLOBs). A DBAPI-compliant module supplies factory functions to build such objects. The types used for this purpose by most DBAPI-compliant modules are those supplied by the modules `datetime` or `mxDateTime`

(covered in “Time Operations”), and strings or buffer types for BLOBs. The factory functions specified by the DBAPI are as follows:

Binary	`Binary(`*`string`*`)` Returns an object representing the given *`string`* of bytes as a BLOB.
Date	`Date(`*`year, month, day`*`)` Returns an object representing the specified date.
DateFromTicks	`DateFromTicks(`*`s`*`)` Returns an object representing the date *`s`* seconds after the epoch of module `time`, covered in Chapter “Time Operations”. For example, `DateFromTicks(time.time())` means “today.”
Time	`Time(`*`hour`*`,`*`minute, second`*`)` Returns an object representing the specified time.
TimeFromTicks	`TimeFromTicks(`*`s`*`)` Returns an object representing the time *`s`* seconds after the epoch of module `time`, covered in Chapter “Time Operations”. For example, `TimeFromTicks(time.time())` means “the current time of day.”
Timestamp	`Timestamp(`*`year`*`,`*`month`*`,`*`day`*`,`*`hour`*`,`*`minute`*`,`*`second`*`)` Returns an object representing the specified date and time.
TimestampFromTicks	`TimestampFromTicks(`*`s`*`)` Returns an object representing the date and time *`s`* seconds after the epoch of module `time`, covered in Chapter “Time Operations”. For example, `TimestampFromTicks(time.time())` is the current date and time.

Type Description Attributes

A `Cursor` instance’s attribute `description` describes the types and other characteristics of each column of the `SELECT` query you last `executed` on that cursor. Each column’s *type* (the second item of the tuple describing the column) equals one of the following attributes of the DBAPI-compliant module:

`BINARY`	Describes columns containing BLOBs
`DATETIME`	Describes columns containing dates, times, or both
`NUMBER`	Describes columns containing numbers of any kind
`ROWID`	Describes columns containing a row-identification number
`STRING`	Describes columns containing text of any kind

A cursor's description, and in particular each column's type, is mostly useful for introspection about the DB your program is working with. Such introspection can help you write general modules and work with tables using different schemas, including schemas that may not be known at the time you are writing your code.

The connect Function

A DBAPI-compliant module's `connect` function accepts arguments that depend on the kind of DB and the specific module involved. The DBAPI standard recommends that `connect` accept named arguments. In particular, `connect` should at least accept optional arguments with the following names:

database	Name of the specific database to connect
dsn	Data-source name to use for the connection
host	Hostname on which the database is running

password	Password to use for the connection
user	Username for the connection

Connection Objects

A DBAPI-compliant module's `connect` function returns an object *x* that is an instance of the class `Connection`. *x* supplies the following methods:

close	*x*`.close()` Terminates the DB connection and releases all related resources. Call `close` as soon as you're done with the DB. Keeping DB connections open needlessly can be a serious resource drain on the system.
commit	*x*`.commit()` Commits the current transaction in the DB. If the DB does not support transactions, *x*`.commit()` is an innocuous no-op.
cursor	*x*`.cursor()` Returns a new instance of the class `Cursor`, covered in "Cursor Objects".
rollback	*x*`.rollback()` Rolls back the current transaction in the DB. If the DB does not support transactions, *x*`.rollback()` raises an exception. The DBAPI recommends that, for DBs that do not support transactions, the class `Connection` supplies no `rollback` method, so that *x*`.rollback()` raises `AttributeError`: you can test whether transactions are supported with `hasattr(`*x*`, 'rollback')`.

Cursor Objects

A Connection instance provides a `cursor` method that returns an object *c* that is an instance of the class `Cursor`. A SQL cursor represents the set of results of a query and lets you work with the records in that set, in sequence, one at a time. A cursor as modeled by the DBAPI is a richer concept, since it's the only way your program executes SQL queries in the first place. On the other hand, a DBAPI cursor allows you only to advance

in the sequence of results (some relational DBs, but not all, also provide higher-functionality cursors that are able to go backward as well as forward), and does not support the SQL clause `WHERE CURRENT OF CURSOR`. These limitations of DBAPI cursors enable DBAPI-compliant modules to provide DBAPI cursors even on RDBMSes that supply no real SQL cursors at all. An instance of the class `Cursor c` supplies many attributes and methods; the most frequently used ones are:

close	*c*`.close()` Closes the cursor and releases all related resources.
description	A read-only attribute that is a sequence of seven-item tuples, one per column in the last query executed: name, typecode, displaysize, internalsize, precision, scale, nullable *c*`.description` is `None` if the last operation on *c* was not a `SELECT` query or returned no usable description of the columns involved. A cursor's description is mostly useful for introspection about the DB your program is working with. Such introspection can help you write general modules that are able to work with tables using different schemas, including schemas that may not be fully known at the time you are writing your code.
execute	*c*`.execute(`*statement*`,`*parameters*`=None)` Executes a SQL *statement* string on the DB with the given *parameters*. *parameters* is a sequence when the module's `paramstyle` is `'format'`, `'numeric'`, or `'qmark'`, and a mapping when `paramstyle` is `'named'` or `'pyformat'`. Some DBAPI modules require the sequences to be specifically tuples.
executemany	*c*`.executemany(`*statement*`,`*parameters*`)` Executes a SQL *statement* on the DB, once for each item of the given *parameters*. *parameters* is a sequence of sequences when the module's `paramstyle` is `'format'`, `'numeric'`, or `'qmark'`, and a sequence of mappings when `paramstyle` is `'named'` or `'pyformat'`. For example, the statement: c.executemany('UPDATE atable SET x=?' 'WHERE y=?',(12,23),(23,34)) when `paramstyle` is `'qmark'`, is equivalent to—but faster than—the two statements: c.execute('UPDATE atable SET x=12 WHERE y=23') c.execute('UPDATE atable SET x=23 WHERE y=34')
fetchall	`c.fetchall()` Returns all remaining rows from the last query as a sequence of tuples. Raises an exception if the last operation was not a `SELECT`.

fetchmany	`c.fetchmany(n)` Returns up to `n` remaining rows from the last query as a sequence of tuples. Raises an exception if the last operation was not a `SELECT`.
fetchone	`c.fetchone()` Returns the next row from the last query as a tuple. Raises an exception if the last operation was not a `SELECT`.
rowcount	A read-only attribute that specifies the number of rows fetched or affected by the last operation, or `-1` if the module is unable to determine this value.

DBAPI-Compliant Modules

Whatever relational DB you want to use, there's at least one (often more than one) Python DBAPI-compliant module downloadable from the Internet. There are so many DBs and modules, and the set of possibilities is so constantly shifting, that we couldn't possibly list them all, nor (importantly) could we maintain the list over time. Rather, we recommend you start from the community-maintained wiki page, which has at least a fighting chance to be complete and up-to-date at any time.

What follows is therefore only a very short, time-specific list of a very few DBAPI-compliant modules that, at the time of writing, are very popular themselves, and interface to very popular open-source DBs.

ODBC

> Open DataBase Connectivity (ODBC) is a standard way to connect to many different DBs, including a few not supported by other DBAPI-compliant modules. For an ODBC-compliant DBAPI-compliant module with a liberal open source license, use pyodbc; for a commercially supported one, mxODBC.

MySQL

> MySQL is a popular open-source RDBMS, currently owned by Oracle. Oracle's own "official" DBAPI-compliant interface to it is MySQL Connector/Python.

PostgreSQL

PostgreSQL is an open-source RDBMS. A popular DBAPI-compliant interface to it is psycopg2.

SQLite

SQLite is "a self-contained, server-less, zero-configuration, transactional SQL database engine," which is the most widely deployed DB engine in the world—it's a C-coded library that implements a DB within a single file, or even in memory for sufficiently small and transient cases. Python's standard library supplies the package `sqlite3`, which is a DBAPI-compliant interface to `SQLite`.

`SQLite` has rich advanced functionality, with many options you can choose; `sqlite3` offers access to much of that functionality, plus further possibilities to make interoperation between your Python code and the underlying DB smoother and more natural. In this book, we don't cover every nook and cranny of these two powerful software systems; we focus on the subset that is most commonly used and most useful. For a great level of detail, including examples and tips about best practices, see SQLite's documentation and `sqlite3`'s online documentation. As a book on the subset, we recommend O'Reilly's *Using SQLite*.

Package `sqlite3` supplies the following functions:

connect

`connect(`*`filepath`*`, `*`timeout`*`=5.0, `*`detect_types`*`=0, `*`isolation_level`*`='', `*`check_same_thread`*`=True, `*`factory`*`=Connection, `*`cached_statements`*`=100, `*`uri`*`=False)`

`connect` connects to the `SQLite` DB in the file named by *`filepath`* (creating it if necessary) and returns an instance of the `Connection` class (or subclass thereof passed as *`factory`*). To create an in-memory DB, pass `':memory:'` as the first argument, *`filepath`*.

If `True`, the *`uri`* argument activates SQLite's URI functionality, allowing a few extra options to be passed, along with the file path, via the *`filepath`* argument.

`timeout` is the number of seconds to wait, before raising an exception, if another connection is keeping the DB locked in a transaction.

`sqlite3` directly supports only the SQLite native types: `BLOB`, `INTEGER`, `NULL`, `REAL`, and `TEXT` (any other type name is treated as `TEXT` unless properly detected and passed through a converter registered with the function

register_converter, covered later in this section), converting as follows:

```
SQLite type     Python type
BLOB            bytes
INTEGER         int
NULL            None
REAL            float
TEXT            depends on the text_factory
attribute
                of the Connection instance,
coverered
                later in this section; by default,
str
```

To allow type name detection, pass as *detect_types* either of the constants PARSE_COLNAMES and PARSE_DECLTYPES, supplied by the sqlite3 package (or both, joining them with the | bitwise-or operator).

connect (continued)

When you pass detect_types=sqlite3.PARSE_COLNAMES, the type name is taken from the name of the column in the SELECT SQL statement that retrieves the column; for example, a column retrieved as *foo* AS [*foo* CHAR(10)] has a type name of CHAR.

When you pass detect_types=sqlite3.PARSE_DECLTYPES, the type name is taken from the declaration of the column in the original CREATE TABLE or ALTER TABLE SQL statement that added the column; for example, a column declared as *foo* CHAR(10) has a type name of CHAR.

When you pass detect_types=sqlite3.PARSE_COLNAMES|sqlite3.PARSE_DECLTYPES, both mechanisms are used, with precedence given to the column name when it has at least two words (the second word gives the type name in this case), falling back to the type that was given for that column at declaration (the first word of the declaration type gives the type name in this case).

isolation_level lets you exercise some control over how SQLite processes transactions; it can be '' (the default), None (to use *autocommit* mode), or one of the three strings 'DEFERRED', 'EXCLUSIVE', and 'IMMEDIATE'. The SQLite online docs cover the details of types of transactions and their relation to the various levels of file locking that SQLite intrinsically performs.

By default, a connection object can be used only in the Python thread that created it, to avoid accidents that could easily corrupt the DB due to minor bugs in your program; minor bugs are, alas, common in multithreaded programming. If you're entirely confident about your threads' use of locks and other synchronization mechanisms, and do need to reuse a connection

	object among multiple threads, you can pass `check_same_thread=False`: then, `sqlite3` performs no checks, trusting your assertion that you know what you're doing and that your multithreading architecture is 100% bug-free—good luck! *`cached_statements`* is the number of SQL statements that `sqlite3` caches in a parsed and prepared state, to avoid the overhead of parsing them repeatedly. You can pass in a value lower than the default `100` to save a little memory, or a larger one if your application uses a dazzling variety of SQL statements.
register_adapter	`register_adapter(`*`type, callable`*`)` `register_adapter` registers *`callable`* as the adapter, from any object of Python type *`type`*, to a corresponding value of one of the few Python types that `sqlite3` handles directly—`int`, `float`, `str`, and `bytes`. *`callable`* must accept a single argument, the value to adapt, and return a value of a type that `sqlite3` handles directly.
register_converter	`register_converter(`*`typename, callable`*`)` `register_converter` registers *`callable`* as the converter, from any value identified in SQL as being of a type named *`typename`* (see parameter *`detect_types`* to function `connect` for an explanation of how the type name is identified), to a corresponding Python object. *`callable`* must accept a single argument, the string form of the value obtained from SQL, and return the corresponding Python object. The *`typename`* matching is case-sensitive.

In addition, `sqlite3` supplies the classes `Connection`, `Cursor`, and `Row`. Each can be subclassed for further customization; however, this is an advanced issue that we do not cover further in this book. The `Cursor` class is a standard DBAPI cursor class, except for an extra convenience method `executescript` accepting a single argument, a string of multiple statements separated by `;` (no parameters). The other two classes are covered in the following sections.

class sqlite3.Connection

In addition to the methods common to all `Connection` classes of DBAPI-compliant modules, covered in "Connection Objects", `sqlite3.Connection` supplies the following methods and other attributes:

`create_aggregate(`*`name,num_params,aggregate_class`*`)`
`aggregate_class` must be a class supplying two instance methods:

create_aggregate	`step`, accepting exactly *`num_param`* arguments, and *`finalize`*, accepting no arguments and returning the final result of the aggregate, a value of a type natively supported by `sqlite3`. The aggregate function can be used in SQL statements by the given *`name`*.
create_collation	`create_collation(`*`name, callable`*`)` *`callable`* must accept two bytestring arguments (encoded in `'utf-8'`), and return `-1` if the first must be considered "less than" the second, `1` if it must be considered "greater than," and `0` if it must be considered "equal," for the purposes of this comparison. Such a collation can be named by the given *`name`* in a SQL `ORDER BY` clause in a `SELECT` statement.
create_function	`create_function(`*`name,num_params,func`*`)` *`func`* must accept exactly *`num_params`* arguments and return a value of a type natively supported by `sqlite3`; such a user-defined function can be used in SQL statements by the given *`name`*.
interrupt	`interrupt()` Call from any other thread to abort all queries executing on this connection (raising an exception in the thread using the connection).
isolation_level	A read-only attribute that's the value given as *`isolation_level`* parameter to the `connect` function.
iterdump	`iterdump()` Returns an iterator that yields strings: the SQL statements that build the current DB from scratch, including both schema and contents. Useful, for example, to persist an in-memory DB to one on disk for future reuse.
row_factory	A callable that accepts the cursor and the original row as a tuple, and returns an object to use as the real result row. A common idiom is *`x`*`.row_factory=sqlite3.Row`, to use the highly optimized Row class covered in "class sqlite3.Row", supplying both index-based and case-insensitive name-based access to columns with negligible overhead.
text_factory	A callable that accepts a single bytestring parameter and returns the object to use for that `TEXT` column value—by default, `str`, but you can set it to any similar callable.
total_changes	The total number of rows that have been modified, inserted, or deleted since the connection was created.

`Connection` objects can also be used as a context manager, to automatically commit database updates or rollback if an exception occurs. Using `Connection` as a context manager does not close the connection, so `Connection.close()` must always be called explicitly.

class sqlite3.Row

`sqlite3` supplies the class `Row`, which is mostly like a `tuple` but also supplies the method `keys()`, returning a list of column names, and supports indexing by a column name as an extra alternative to indexing by column number.

A sqlite3 example

The following example handles the same task as the examples shown earlier in the chapter, but uses `sqlite3` for persistence, without creating the index in memory:

```
import fileinput, sqlite3
connect = sqlite3.connect('database.db')
cursor = connect.cursor()
with connect:
    cursor.execute('CREATE TABLE IF NOT EXISTS Words '
                   '(Word TEXT, File TEXT, Line INT)')
    for line in fileinput.input():
        f, l = fileinput.filename(), fileinput.filelineno()
        cursor.executemany('INSERT INTO Words VALUES (:w, :f,
:l)',
            [{'w':w, 'f':f, 'l':l} for w in line.split()])
connect.close()
```

We can then use `sqlite3` to read back the data stored in the DB file *database.db*, as shown in the following example:

```
import sys, sqlite3, linecache
connect = sqlite3.connect('database.db')
cursor = connect.cursor()
for word in sys.argv[1:]:
    cursor.execute('SELECT File, Line FROM Words '
                   'WHERE Word=?', [word])
    places = cursor.fetchall()
    if not places:
```

```
            print(f'Word {word!r} not found in index file',
                  file=sys.stderr)
            continue
        for fname, lineno in places:
            print(f'Word {word!r} occurs in line {lineno} of file
{fname!r}:')
            print(linecache.getline(fname, lineno), end='')
connect.close()
```

---

1 In fact “CSV” is something of a misnomer, since some dialects use tabs or other characters rather than commas as the field separator. It might be easier to think of them as “delimiter-separated values.”

2 consider third-party package dill if you need to extend pickle in this and other aspects

Chapter 11. 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 11-1. 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 11-2. 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 11-3. 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
easterprint(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 12. Controlling Execution

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 13th 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 directly exposes, supports, and documents many of its internal mechanisms. This may help you understand Python at an advanced level, and lets you hook your own code into such Python mechanisms, controlling them to some extent. For example, "Python built-ins" covers the way Python arranges for built-ins to be visible. This chapter covers some other advanced Python techniques; Chapter 16 covers issues specific to testing, debugging, and profiling. Other issues related to controlling execution are about using multiple threads and processes, covered in Chapter 14.

Per-Site Customization

Python provides a specific "hook" to let each site customize some aspects of Python's behavior at the start of each run.

We strongly recommend that you avoid altering the `site.py` file that performs the base customization, because it might cause Python to behave differently on your system than elsewhere, and, in any case, the file would be overwritten each and every time you update your Python installation.

If, as a system administrator (or in an equivalent role, such as a user who did a Python install in their home directory for their sole use) you think you absolutely need some customization, perform it in a new file you will name `sitecustomize.py` (create it in the same directory where `site.py` lives).

The site and sitecustomize Modules

Python loads the standard module `site` just before the main script. If Python is run with option **`-S`**, it does not load site. **`-S`** allows faster startup but saddles the main script with initialization chores.

`site`'s tasks are, chiefly, to put `sys.path` in standard form (absolute paths, no duplicates), including as directed by environment variables, by virtual environments, and by each *.pth* file found in a directory in `sys.path`.

Secondarily, if the session starting is an interactive one, `site` adds several handy built-ins, such as `exit`, `copyright`, etc., and, if readline is enabled, configure completion as the function of the `tab` key.

In any normal Python installation, the installation process sets everything up to ensure that `site`'s work is sufficient to let Python programs and interactive sessions run "normally", i.e., as they would on any other system with that version of Python installed. In exceptional cases, as the system admin or in an equivalent role, you're sure you need further tweaks, write a `sitecustomize.py` file in the same directory where `site.py` is found. In the rare cases where `sitecustomize.py` is present, what it typically does is add yet more dictionaries to `sys.path`--the best way to perform this task is for `sitecustomize.py` to import site and then to call `site.addsitedir(`*`path_to_a_dir`*`)`.

Termination Functions

The `atexit` module lets you register termination functions (i.e., functions to be called at program termination, "last in, first out"). Termination

functions are similar to clean-up handlers established by `try/finally` or `with`. However, termination functions are globally registered and get called at the end of the whole program, while clean-up handlers are established lexically and get called at the end of a specific `try` clause or `with` statement. Termination functions and clean-up handlers are called whether the program terminates normally or abnormally, but not when the program ends by calling `os._exit` (so you normally call `sys.exit` instead). The `atexit` module supplies a function called `register`:

register `register(`*`func`*`, *`*`args`*`, **`*`kwds`*`)`

Ensures that *`func`*`(*`*`args`*`,**`*`kwds`*`)` is called at program termination time.

Dynamic Execution and exec

Python's `exec` built-in function can execute code that you read, generate, or otherwise obtain during a program's run. `exec` dynamically executes a statement or a suite of statements. `exec` is a built-in function with the syntax:

```
exec(code, globals=None, locals=None)
```

`code` can be a string, bytes, or code object. *`globals`* is a dict; *`locals`*, any mapping.

If both *`globals`* are *`locals`* are present, they are the global and local namespaces in which *`code`* runs. If only *`globals`* is present, exec uses *`globals`* as both namespaces. If neither is present, *`code`* runs in the current scope.

Never run exec in the current scope

Running exec in the current scope is a very bad idea: it can bind, rebind, or unbind any global name. To keep things under control, use exec, if at all, only with specific, explicit dictionaries.

Avoiding exec

A frequently asked question about Python is "How do I set a variable whose name I just read or built?" Literally, for a *global* variable, `exec` allows this, but it's a bad idea. For example, if the name of the variable is in *`varname`*, you might think to use:

```
exec(varname + ' = 23')
```

Don't do this. An `exec` like this in current scope makes you lose control of your namespace, leading to bugs that are extremely hard to find, and making your program unfathomably difficult to understand. Keep the "variables" that you need to set with dynamically-found names, not as variables, but as entries in a dictionary, say *`mydict`*. You could then use:

```
exec(varname+'=23', mydict)
```

While this is not quite as terrible as the previous example, it is *still* a bad idea. Keeping such "variables" as dictionary entries means that you don't have any need to use `exec` to set them. Just code:

```
mydict[varname] = 23
```

This way, your program is clearer, direct, elegant, and faster. There *are* some valid uses of `exec`, but they are extremely rare: just use explicit dictionaries instead.

Strive to avoid exec

Use `exec` only when it's really indispensable, which is *extremely* rare. Most often, it's best to avoid `exec` and choose more specific, well-controlled mechanisms: `exec` weakens your control of your code's namespace, can damage your program's performance, and exposes you to numerous hard-to-find bugs and huge security risks.

Expressions

`exec` can execute an expression, because any expression is also a valid statement (called an *expression statement*). However, Python ignores the value returned by an expression statement. To evaluate an expression and obtain the expression's value, see the built-in function `eval`, covered in Table 7-2. (Note, however, that most of the same caveats as for `exec` also apply to `eval`).

Compile and Code Objects

To make a code object to use with `exec`, call the built-in function `compile` with the last argument set to 'exec' (as covered in Table 7-2).

A code object *c* exposes many interesting read-only attributes whose names all start with `'co_'`, such as:

`co_argcount`

Number of parameters of the function of which *c* is the code (0 when *c* is not the code object of a function, but rather is built directly by compile)

`co_code`

A bytestring with *c*'s bytecode

`co_consts`

The tuple of constants used in *c*

`co_filename`

The name of the file *c* was compiled from (the string that is the second argument to compile, when *c* was built that way)

`co_firstlineno`

The initial line number (within the file named by co_filename) of the source code that was compiled to produce *c*, if *c* was built by compiling from a file

`co_name`

The name of the function of which *c* is the code ('<module>' when *c* is not the code object of a function but rather is built directly by compile)

`co_names`

The tuple of all identifiers used within *c*

`co_varnames`

The tuple of local variables' identifiers in *c*, starting with parameter names

Most of these attributes are useful only for debugging purposes, but some may help advanced introspection, as exemplified later in this section.

If you start with a string that holds one or more statements, first use `compile` on the string, then call `exec` on the resulting code object—that's better than giving `exec` the string to compile and execute. This separation lets you check for syntax errors separately from execution-time errors. You can often arrange things so that the string is compiled once and the code object executes repeatedly, which speeds things up. `eval` can also benefit from such separation. Moreover, the `compile` step is intrinsically safe

(both `exec` and `eval` are extremely risky if you execute them on code that you don't 100%-trust), and you may be able to perform some checks on the code object, before it executes, to lessen the risk (though never truly down to zero).

A code object has a read-only attribute `co_names`, which is the tuple of the names used in the code. For example, say that you want the user to enter an expression that contains only literal constants and operators—no function calls or other names. Before evaluating the expression, you can check that the string the user entered satisfies these constraints:

```
def safer_eval(s):
    code = compile(s, '<user-entered string>', 'eval')
    if code.co_names:
        raise ValueError('No names {!r} allowed in expression
{!r}'
                         .format(code.co_names, s))
    return eval(code)
```

This function `safer_eval` evaluates the expression passed in as argument *`s`* only when the string is a syntactically valid expression (otherwise, `compile` raises `SyntaxError`) and contains no names at all (otherwise, `safer_eval` explicitly raises `ValueError`). (This is similar to the standard library function `ast.literal_eval`, covered in "Standard Input", but a bit more powerful, since it does allow the use of operators.)

Knowing what names the code is about to access may sometimes help you optimize the preparation of the dictionary that you need to pass to `exec` or `eval` as the namespace. Since you need to provide values only for those names, you may save work by not preparing other entries. For example, say that your application dynamically accepts code from the user with the convention that variable names starting with *`data_`* refer to files residing in the subdirectory *data* that user-written code doesn't need to read explicitly. User-written code may in turn compute and leave results in global variables with names starting with *`result_`*, which your application writes back as files in subdirectory *data*. Thanks to this

convention, you may later move the data elsewhere (e.g., to BLOBs in a database instead of files in a subdirectory), and user-written code won't be affected. Here's how you might implement these conventions efficiently (in v3; in v2, use `exec user_code in datadict` instead of `exec(user_code, datadict)`):

```
def exec_with_data(user_code_string):
    user_code = compile(user_code_string, '<user code>', 'exec')
    datadict = {}
    for name in user_code.co_names:
        if name.startswith('data_'):
            with open('data/{}'.format(name[5:]), 'rb') as
datafile:
                datadict[name] = datafile.read()
        elif name.startswith('result_'):
            pass  # user code can assign to variables named
`result_…`
        else:
            raise ValueError(f'invalid variable name {name!r}')
    exec(user_code, datadict)
    for name in datadict:
        if name.startswith('result_'):
            with open('data/{}'.format(name[7:]), 'wb') as
datafile:
                datafile.write(datadict[name])
```

Never exec or eval Untrusted Code

Old versions of Python tried to supply tools to ameliorate the risks of using `exec` and `eval`, under the heading of "restricted execution," but those tools were never entirely secure against the ingenuity of able hackers, and recent versions of Python have therefore dropped them. If you need to ward against such attacks, take advantage of your operating system's protection mechanisms: run untrusted code in a separate process, with privileges as restricted as you can possibly make them (study the mechanisms that your OS supplies for the purpose, such as `chroot`, `setuid`, and `jail`; in Windows, you might purchase third-party, commercial add-on WinJail, or run untrusted code in a separate, highly constrained virtual machine (or container, if you're an expert on how to securitize containers). To guard against "denial of service" attacks, have the main process monitor the

separate one and terminate the latter if and when resource consumption becomes excessive. Processes are covered in “Running Other Programs”.

`exec` and `eval` are unsafe with untrusted code

The function `exec_with_data` is not at all safe against untrusted code: if you pass it, as the argument *`user_code_string`*, some string obtained in a way that you cannot *entirely* trust, there is essentially no limit to the amount of damage it might do. This is unfortunately true of just about any use of both exec and eval, except for those rare cases in which you can set very strict and checkable limits on the code to execute or evaluate, as was the case for the function `safer_eval`.

Internal Types

Some of the internal Python objects in this section are hard to use, and indeed **not** meant for use most of the time. Using such objects correctly and to good effect requires some study of your Python implementation’s own C (or Java, or C#) sources. Such black magic is rarely needed, except to build general-purpose development tools, and similar wizardly tasks. Once you do understand things in depth, Python empowers you to exert control if and when needed. Since Python exposes many kinds of internal objects to your Python code, you can exert that control by coding in Python, even when an understanding of C (or Java, or C#) is needed to read Python’s sources to understand what’s going on.

Type Objects

The built-in type named `type` acts as a callable factory, returning objects that are types. Type objects don’t have to support any special operations except equality comparison and representation as strings. However, most type objects are callable and return new instances of the type when called. In particular, built-in types such as `int`, `float`, `list`, `str`, `tuple`, `set`, and `dict` all work this way; specifically, when called without

arguments, they return a new empty instance, or, for numbers, one that equals `0`. The attributes of the `types` module are the built-in types, each with one or more names, but only for built-in types that don't already have a built-in name, as covered in Chapter 7. Besides being callable to generate instances, many type objects are also useful because you can inherit from them, as covered in "Classes and Instances".

The Code Object Type

Besides using the built-in function `compile`, you can get a code object via the `__code__` attribute of a function or method object. (For the attributes of code objects, see "Compile and Code Objects".) Code objects are not callable, but you can rebind the `__code__` attribute of a function object with the right number of parameters in order to wrap a code object into callable form. For example:

```
def g(x): print('g', x)
code_object = g.__code__
def f(x): pass
f.__code__ = code_object
f(23)     # prints: g 23
```

Code objects that have no parameters can also be used with `exec` or `eval`. To create a new object, call the type object you want to instantiate. However, directly creating code objects requires many parameters; see Stack Overflow's nonofficial docs (http://stackoverflow.com/questions/16064409/how-to-create-a-code-object-in-python) on how to do it (almost always, you're better off calling `compile` instead).

The frame Type

The function `_getframe` in the module `sys` returns a frame object from Python's call stack. A frame object has attributes giving information about the code executing in the frame and the execution state. The modules

`traceback` and `inspect` help you access and display such information, particularly when an exception is being handled. Chapter 16 provides more information about frames and tracebacks, and covers the module `inspect`, which is the best way to perform such introspection. As the leading underscore in the name `_getframe` hints, the function is "somewhat private", meant for use only by tools such as debuggers, ones which inevitably require deep introspection into Python's internals.

Garbage Collection

Python's garbage collection normally proceeds transparently and automatically, but you can choose to exert some direct control. The general principle is that Python collects each object *x* at some time after *x* becomes unreachable—that is, when no chain of references can reach *x* by starting from a local variable of a function instance that is executing, nor from a global variable of a loaded module. Normally, an object *x* becomes unreachable when there are no references at all to *x*. In addition, a group of objects can be unreachable when they reference each other but no global or local variables reference any of them, even indirectly (such a situation is known as a *mutual reference loop*).

Classic Python keeps with each object *x* a count, known as a *reference count*, of how many references to *x* are outstanding. When *x*'s reference count drops to `0`, CPython immediately collects *x*. The function `getrefcount` of the module `sys` accepts any object and returns its reference count (at least `1`, since `getrefcount` itself has a reference to the object it's examining). Other versions of Python, such as Jython or IronPython, rely on other garbage-collection mechanisms supplied by the platform they run on (e.g., the JVM or the MSCLR). The modules `gc` and `weakref` therefore apply only to CPython.

When Python garbage-collects *x* and there are no references at all to *x*, Python then finalizes *x* (i.e., calls `x.__del__()`) and makes the memory that *x* occupied available for other uses. If *x* held any references to other

objects, Python removes the references, which in turn may make other objects collectable by leaving them unreachable.

The gc Module

The `gc` module exposes the functionality of Python's garbage collector. `gc` deals only with unreachable objects that are part of mutual reference loops. In such a loop, each object in the loop refers to others, keeping the reference counts of all objects positive. However, no outside references to any one of the set of mutually referencing objects exist any longer. Therefore, the whole group, also known as *cyclic garbage*, is unreachable, and therefore garbage-collectable. Looking for such cyclic garbage loops takes time, which is why the module `gc` exists: to help you control whether and when your program spends that time. The functionality of "cyclic garbage collection," by default, is enabled with some reasonable default parameters: however, by importing the `gc` module and calling its functions, you may choose to disable the functionality, change its parameters, and/or find out exactly what's going on in this respect.

`gc` exposes functions you can use to help you keep cyclic garbage-collection times under control. These functions can sometimes let you track down a memory leak—objects that are not getting collected even though there *should* be no more references to them—by helping you discover what other objects are in fact holding on to references to them:

collect	`collect()` Forces a full cyclic garbage collection run to happen immediately.
disable	`disable()` Suspends automatic, periodic cyclic garbage collection.
enable	`enable()`

	Reenables periodic cyclic garbage collection previously suspended with `disable` .
garbage	A read-only attribute that lists the unreachable but uncollectable objects. This happens when any object in a cyclic garbage loop has a `__del__` special method, as there may be no demonstrably safe order for Python to finalize such objects.
get_debug	`get_debug()` Returns an int bit string, the garbage-collection debug flags set with `set_debug`.
get_objects	`get_objects()` Returns a list of all objects currently tracked by the cyclic garbage collector.
get_referrers	`get_referrers(*objs)` Returns a list of all container objects, currently tracked by the cyclic garbage collector, that refer to any one or more of the arguments.
get_threshold	`get_threshold()` Returns a three-item tuple *`(threshe, thresh1, thresh2)`*, the garbage-collection thresholds set with `set_threshold`.
isenabled	`isenabled()` Returns `True` when cyclic garbage collection is currently enabled. Returns `False` when collection is currently disabled.

set_debug	`set_debug(flags)`
	
	Sets debugging flags for garbage collection. *`flags`* is an `int`, interpreted as a bit string, built by ORing (with the bitwise-OR operator `|`) zero or more constants supplied by the module `gc`:
	
	`DEBUG_COLLECTABLE`
	
	Prints information on collectable objects found during collection

	`DEBUG_LEAK`
	
	The set of debugging flags that make the garbage collector print all information that can help you diagnose memory leaks; same as the bitwise-OR of all other constants (except `DEBUG_STATS`, which serves a different purpose)
	
	`DEBUG_SAVEALL`
	
	Saves all collectable objects to the list `gc.garbage` (where uncollectable ones are also always saved) to help you diagnose leaks
	
	`DEBUG_STATS`
	
	Prints statistics during collection to help you tune the thresholds
	
	`DEBUG_UNCOLLECTABLE`
	
	Prints information on uncollectable objects found during collection

set_threshold	`set_threshold(`*`thresh0`*`[, `*`thresh1`*`[, `*`thresh2`*`]])` Sets thresholds that control how often cyclic garbage-collection cycles run. A *`thresh0`* of `0` disables garbage collection. Garbage collection is an advanced, specialized topic, and the details of the generational garbage-collection approach used in Python (and consequently the detailed meanings of these thresholds) are beyond the scope of this book; see the online docs for some details.

When you know there are no cyclic garbage loops in your program, or when you can't afford the delay of cyclic garbage collection at some crucial time, suspend automatic garbage collection by calling `gc.disable()`. You can enable collection again later by calling `gc.enable()`. You can test whether automatic collection is currently enabled by calling

`gc.isenabled()`, which returns `True` or `False`. To control *when* time is spent collecting, you can call `gc.collect()` to force a full cyclic collection run to happen immediately. To wrap some time-critical code:

```
import gc
gc_was_enabled = gc.isenabled()
if gc_was_enabled:
    gc.collect()
    gc.disable()
# insert some time-critical code here
if gc_was_enabled:
    gc.enable()
```

Other functionality in the module `gc` is more advanced and rarely used, and can be grouped into two areas. The functions `get_threshold` and `set_threshold` and debug flag `DEBUG_STATS` help you fine-tune garbage collection to optimize your program's performance. The rest of `gc`'s functionality can help you diagnose memory leaks in your program. While `gc` itself can automatically fix many leaks (as long as you avoid defining `__del__` in your classes, since the existence of `__del__` can block cyclic garbage collection), your program runs faster if it avoids creating cyclic garbage in the first place.

Instrumenting garbage collection

`gc.callbacks` is an initially empty list to which you can add functions `f(phase, info)` which Python is to call upon garbage collection. When Python calls each such function, `phase` is `'start'` or `'stop'` to mark the beginning or end of a collection, and `info` is a dictionary containing information about the generational collection used by CPython. You can add functions to this list, for example to gather statistics about garbage collection. See the documentation for more details.

The weakref Module

Careful design can often avoid reference loops. However, at times you need objects to know about each other, and avoiding mutual references would

distort and complicate your design. For example, a container has references to its items, yet it can often be useful for an object to know about a container holding it. The result is a reference loop: due to the mutual references, the container and items keep each other alive, even when all other objects forget about them. Weak references solve this problem by allowing objects to reference others without keeping them alive.

A *weak reference* is a special object `w` that refers to some other object `x` without incrementing `x`'s reference count. When `x`'s reference count goes down to `0`, Python finalizes and collects `x`, then informs `w` of `x`'s demise. Weak reference `w` can now either disappear or get marked as invalid in a controlled way. At any time, a given `w` refers to either the same object `x` as when `w` was created, or to nothing at all; a weak reference is never retargeted. Not all types of objects support being the target `x` of a weak reference `w`, but classes, instances, and functions do.

The weakref module exposes functions and types to create and manage weak references:

getweakrefcount	`getweakrefcount(x)` Returns len(getweakrefs(*x*)).
getweakrefs	`getweakrefs(x)` Returns a list of all weak references and proxies whose target is `x`.
proxy	`proxy(x[,f])` Returns a weak proxy `p` of type `ProxyType` (`CallableProxyType` when `x` is callable) with object `x` as the target. In most contexts, using `p` is just like using `x`, except that, when you use `p` after `x` has been deleted, Python raises `ReferenceError`. `p` is not hashable (thus, you cannot use `p` as a dictionary key), even when `x` is. When `f` is present, it must be callable with one argument and is the finalization callback for `p` (i.e., right before finalizing `x`, Python calls `f(p)`). `f` executes right *after* `x` is no longer reachable from `p`.
ref	`ref(x[,f])` Returns a weak reference `w` of type `ReferenceType` with object `x` as the target. `w` is callable without arguments: calling `w`() returns `x` when `x` is still alive; otherwise, `w`() returns None. `w` is hashable when `x` is

	hashable. You can compare weak references for equality (==, !=), but not for order (<, >, <=, >=). Two weak references *x* and *y* are equal when their targets are alive and equal, or when *x* is *y*. When *f* is present, it must be callable with one argument and is the finalization callback for *w* (i.e., right before finalizing *x*, Python calls *f*(*w*)). *f* executes right *after* *x* is no longer reachable from *w*.
WeakKeyDictionary	class WeakKeyDictionary(*adict*={}) A WeakKeyDictionary *d* is a mapping weakly referencing its keys. When the reference count of a key *k* in *d* goes to 0, item *d*[*k*] disappears. *adict* is used to initialize the mapping.
WeakValueDictionary	class WeakValueDictionary(*adict*={}) A WeakValueDictionary *d* is a mapping weakly referencing its values. When the reference count of a value *v* in *d* goes to 0, all items of *d* such that *d*[*k*] is *v* disappear. *adict* is used to initialize the mapping.

`WeakKeyDictionary` lets you noninvasively associate additional data with some hashable objects, with no change to the objects. `WeakValueDictionary` lets you noninvasively record transient associations between objects, and build caches. In each case, use a weak mapping, rather than a `dict`, to ensure that an object that is otherwise garbage-collectable is not kept alive just by being used in a mapping.

A typical example is a class that keeps track of its instances, but does not keep them alive just in order to keep track of them:

```
import weakref
class Tracking(object):
    _instances_dict = weakref.WeakValueDictionary()
    def __init__(self):
        Tracking._instances_dict[id(self)] = self
    @classmethod
    def instances(cls):
       return cls._instances_dict.values()
```

Chapter 13. 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 13-1. 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(`*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`;	✓	✓

isclose	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	`isnan(x)` 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*.	✓	
perm	‖**3.8++**‖ `perm(`*n*`, `*k*`)` 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* `+` *y*`j)`.		✓
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 13-2. 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,*	*a* + *b*

	b)	
contains	*contains(a,* *b)*	*b* in *a*
countOf	*countOf (a,* *b)*	*a* .count (*b*)
delitem	*delitem (a,* *b)*	del *a*[*b*]
delslice	*delslice (a,* *b,* *c)*	del *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	`choices(`*seq*`, *, `*weights*`, `*cum_weights*`, `*k=1*`)` 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 `float`s or `int`s 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	`seed(`*x*`=None)` 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 13-3 shows the possible type codes for `array`.

Table 13-3. 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
'H'	unsigned short	int	2 bytes
'i'	int	int	2 bytes
'I'	unsigned int	int	2 bytes
'l'	long	int	4 bytes
'L'	unsigned long	int	4 bytes
'q'	long long	int	8 bytes
'Q'	unsigned long long	int	8 bytes
'f'	float	float	4 bytes

`'d'`	`double`	`float`	8 bytes

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*.
	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`.

frombytes	
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`.
itemsize	*a*.`itemsize` Property that returns the size in bytes of an element in *a*.
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 14. Testing, Debugging, and Optimizing

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 16th 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're not finished with a programming task when you're done writing the code; you're finished only when the code runs correctly and with acceptable performance. *Testing* (covered in "Testing") means verifying that code runs correctly, by automatically exercising the code under known conditions and checking that results are as expected. *Debugging* (covered in "Debugging") means discovering causes of incorrect behavior and repairing them (repair is often easy, once you figure out the causes).

Optimizing (covered in "Optimization") is often used as an umbrella term for activities meant to ensure acceptable performance. Optimizing breaks down into *benchmarking* (measuring performance for given tasks to check that it's within acceptable bounds), *profiling* (*instrumenting* the program with extra code to identify performance bottlenecks), and actual optimizing (removing bottlenecks to improve program performance). Clearly, you can't remove performance bottlenecks until you've found out where they are (via profiling), which in turn requires knowing that there *are* performance problems (via benchmarking).

This chapter covers the subjects in the natural order in which they occur in development: testing first and foremost, debugging next, and optimizing

last. Most programmers' enthusiasm focuses on optimization: testing and debugging are often (wrongly!) perceived as being chores, while optimization is seen as being fun. Were you to read only one section of the chapter, we might suggest that section be "Developing a Fast-Enough Python Application", which summarizes the Pythonic approach to optimization—close to Jackson's classic "Rules of Optimization: Rule 1: Don't do it. Rule 2 (for experts only): Don't do it *yet*."

All of these tasks are important; each could fill at least a book by itself. This chapter cannot even come close to exploring every related technique; rather, it focuses on Python-specific techniques, approaches, and tools. Often, for best results, you should approach the issue from the higher-level viewpoint of *system analysis and design*, rather than focusing only on implementation (in Python and/or any other mix of programming languages). Start by studying a good general book on the subject, such as System Analysis and Design (by Dennis, Wixom, and Roth; Wiley).

Testing

In this chapter, we distinguish between two different kinds of testing: *unit testing* and *system testing*. Testing is a rich, important field: many more distinctions could be drawn, but we focus on the issues that mostly matter to most software developers. Many developers are reluctant to spend time on testing, seeing it as time stolen from "real" development, but this is short-sighted: defects are easier to fix the earlier you find out about them—an extra hour spent developing tests will amply pay for itself as you find defects ASAP, saving you many hours of debugging that would otherwise have been needed in later phases of the software development cycle.[1]

Unit Testing and System Testing

Unit testing means writing and running tests to exercise a single module, or an even smaller unit, such as a class or function. *System testing* (also known as *functional* or *integration* or *end-to-end* testing) involves running an entire program with known inputs. Some classic books on testing also draw

the distinction between *white-box testing*, done with knowledge of a program's internals, and *black-box testing*, done without such knowledge. This classic viewpoint parallels, but does not exactly duplicate, the modern one of unit versus system testing.

Unit and system testing serve different goals. Unit testing proceeds apace with development; you can and should test each unit as you're developing it. One relatively modern approach (first proposed in 1971 in Weinberg's immortal classic *The Psychology of Computer Programming*) is known as *test-driven development* (TDD): for each feature that your program must have, you first write unit tests, and only then do you proceed to write code that implements the feature and makes the tests pass. TDD may seem upside-down, but it has advantages; for example, it ensures that you won't omit unit tests for some feature. Developing test-first is helpful because it urges you to focus first on exactly what tasks a certain function, class, or method should accomplish, dealing only afterward with *how* to implement that function, class, or method. An innovation along the lines of TDD is behavior-driven development.

In order to test a unit—which may depend on other units not yet fully developed—you often have to write *stubs*, also known as *mocks*[2]—fake implementations of various units' interfaces giving known, correct responses in cases needed to test other units. The `mock` module (part of Python's standard library in the package `unittest`) helps you implement such stubs.

System testing comes later, since it requires the system to exist, with at least some subset of system functionality believed (based on unit testing) to be working. System testing offers a sanity check: each module in the program works properly (passes unit tests), but does the *whole* program work? If each unit is okay but the system is not, there's a problem in the integration between units—the way the units cooperate. For this reason, system testing is also known as *integration* testing.

System testing is similar to running the system in production use, except that you fix inputs in advance so that any problems you may find are easy to reproduce. The cost of failures in system testing is lower than in production

use, since outputs from system testing are not used to make decisions, serve customers, control external systems, and so on. Rather, outputs from system testing are systematically compared with the outputs that the system *should* produce given the known inputs. The purpose is to find, in cheap and reproducible ways, discrepancies between what the program *should* do and what the program actually *does*.

Failures discovered by system testing (just like system failures in production use) may reveal some defects in unit tests, as well as defects in the code. Unit testing may have been insufficient: a module's unit tests may have failed to exercise all needed functionality of the module. In that case, the unit tests need to be beefed up. Do that *before* you change your code to fix the problem, then run the newly enhanced unit tests to confirm that they now show the problem. Then, fix the problem, and run unit tests again to confirm they show no problem anymore. Finally, rerun the system tests to confirm that the problem has indeed gone away.

Bug-fixing Best Practice

This best practice is a specific application of test-driven design that we recommend without reservation: never fix a bug before having added unit tests that would have revealed the bug (an excellent, cheap insurance against software regression bugs).

Often, failures in system testing reveal communication problems within the development team[3]: a module correctly implements a certain functionality, but another module expects different functionality. This kind of problem (an integration problem in the strict sense) is hard to pinpoint in unit testing. In good development practice, unit tests must run often, so it is crucial that they run fast. It's therefore essential, in the unit-testing phase, that each unit can assume other units are working correctly and as expected.

Unit tests run in reasonably late stages of development can reveal integration problems if the system architecture is hierarchical, a common and reasonable organization. In such an architecture, low-level modules

depend on no others (except library modules, which you can typically assume to be correct), so the unit tests of such low-level modules, if complete, suffice to assure correctness. High-level modules depend on low-level ones, and thus also depend on correct understanding about what functionality each module expects and supplies. Running complete unit tests on high-level modules (using true low-level modules, not stubs) exercises interfaces between modules, as well as the high-level modules' own code.

Unit tests for high-level modules are thus run in two ways. You run the tests with stubs for the low levels during the early stages of development, when the low-level modules are not yet ready or, later, when you only need to check the correctness of the high levels. During later stages of development, you also regularly run the high-level modules' unit tests using the true low-level modules. In this way, you check the correctness of the whole subsystem, from the high levels downward. Even in this favorable case, you *still* need to run system tests to ensure the system's functionality is exercised and checked, and no interface between modules is neglected.

System testing is similar to running the program in normal ways. You need special support only to ensure that known inputs are supplied and that outputs are captured for comparison with expected outputs. This is easy for programs that perform I/O (input/output) on files, and hard for programs whose I/O relies on a GUI, network, or other communication with external entities. To simulate such external entities and make them predictable and entirely observable, you generally need platform-dependent infrastructure. Another useful piece of supporting infrastructure for system testing is a *testing framework* to automate the running of system tests, including logging of successes and failures. Such a framework can also help testers prepare sets of known inputs and corresponding expected outputs.

Both free and commercial programs for these purposes exist, and usually do not depend on which programming languages are used in the system under test. System testing is a close kin to what was classically known as black-box testing: testing that is independent from the implementation of the system under test (and thus, in particular, independent from the

programming languages used for implementation). Instead, testing frameworks usually depend on the operating system platform on which they run, since the tasks they perform are platform-dependent: running programs with given inputs, capturing their outputs, simulating and capturing GUI, network, and other interprocess communication I/O. Since frameworks for system testing depend on the platform, not on programming languages, we do not cover them further in this book. For a thorough list of Python testing tools, see the Python wiki.

The doctest Module

The `doctest` module exists to let you create good examples in your code's docstrings, by checking that the examples do in fact produce the results that your docstrings show for them. `doctest` recognizes such examples by looking within the docstring for the interactive Python prompt `'>>> '`, followed on the same line by a Python statement, and the statement's expected output on the next line(s).

As you develop a module, keep the docstrings up to date and enrich them with examples. Each time a part of the module (e.g., a function) is ready, or partially ready, make it a habit to add examples to its docstring. Import the module into an interactive session, and use the parts you just developed in order to provide examples with a mix of typical cases, limit cases, and failing cases. For this specific purpose only, use `from` *`module`* `import *` so that your examples don't prefix *`module`*. to each name the module supplies. Copy and paste the interactive session into the docstring in an editor, adjust any glitches, and you're almost done.

Your documentation is now enriched with examples, and readers have an easier time following it, assuming you choose a good mix of examples, wisely seasoned with nonexample text. Make sure you have docstrings, with examples, for the module as a whole, and for each function, class, and method the module exports. You may choose to skip functions, classes, and methods whose names start with _, since (as their names indicate) they're meant to be private implementation details; `doctest` by default ignores them, and so should readers of your module.

Make Your Examples Match Reality

Examples that don't match the way your code works are worse than useless. Documentation and comments are useful only if they match reality; docs and comments that lie can be seriously damaging.

Docstrings and comments often get out of date as code changes, and thus become misinformation, hampering, rather than helping, any reader of the source. Better to have no comments and docstrings at all, poor as such a choice would be, than to have ones that lie. `doctest` can help you through the examples in your docstrings. A failing `doctest` run should prompt you to review the docstring that contains the failing examples, thus reminding you to keep the whole docstring updated.

At the end of your module's source, insert the following snippet:

```
if __name__ == '__main__' :
    import doctest
    doctest.testmod()
```

This code calls the function `testmod` of the module `doctest` when you run your module as the main program. `testmod` examines docstrings (the module docstring, and docstrings of all public functions, classes, and methods thereof). In each docstring, `testmod` finds all examples (by looking for occurrences of the interpreter prompt `'>>> '`, possibly preceded by whitespace) and runs each example. `testmod` checks that each example's results match the output given in the docstring right after the example. In case of exceptions, `testmod` ignores the traceback, and just checks that the expected and observed error messages are equal.

When everything goes right, `testmod` terminates silently. Otherwise, it outputs detailed messages about examples that failed, showing expected and actual output. Example 16-1 shows a typical example of `doctest` at work on a module *mod.py*.

Example 14-1. Using doctest

```
"""
This module supplies a single function reverse_words that reverses
a string by words.
>>> reverse_words('four score and seven years')
'years seven and score four'
>>> reverse_words('justoneword')
'justoneword'
>>> reverse_words('')
''
You must call reverse_words with a single argument, a string:
>>> reverse_words()
Traceback (most recent call last):
    ...
TypeError: reverse_words() missing 1 required positional argument:
'astring'
>>> reverse_words('one', 'another')
Traceback (most recent call last):
    ...
TypeError: reverse_words() takes 1 positional argument but 2 were
given
>>> reverse_words(1)
Traceback (most recent call last):
    ...
AttributeError: 'int' object has no attribute 'split'
>>> reverse_words('ڀنƲ熵嚞ڀ찝牝蕻ڀ□s all right too')
'too right all is ڀنƲ熵嚞ڀ찝牝蕻ڀ□As a side effect, reverse_words
eliminates any redundant spacing:
>>> reverse_words('with  redundant   spacing')
'spacing redundant with'
"""
def reverse_words(astring):
    words = astring.split()
    words.reverse()
    return ' '.join(words)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

In this module's docstring, we snipped the tracebacks from the docstring and replaced them with ellipses (`...`): this is good practice, since `doctest` ignores tracebacks, which add nothing to the explanatory value of a failing case. Apart from this snipping, the docstring is the copy and paste of an interactive session, plus some explanatory text and empty lines for readability. Save this source as *mod.py*, and then run it with **`python`**

`mod.py`. It produces no output, meaning that all examples work right. Try **`python mod.py -v`** to get an account of all tests tried and a verbose summary at the end. Finally, alter the example results in the module docstring, making them incorrect, to see the messages `doctest` provides for errant examples.

While `doctest` is not meant for general-purpose unit testing, it can be tempting to use it for that purpose. The recommended way to do unit testing in Python is with the module `unittest`, covered in "The unittest Module". However, unit testing with `doctest` can be easier and faster to set up, since that requires little more than copying and pasting from an interactive session. If you need to maintain a module that lacks unit tests, retrofitting such tests into the module with `doctest` is a reasonable compromise (although you should plan to eventually upgrade to full-fledged tests with `unittest`). It's better to have `doctest`-based unit tests than not to have any unit tests at all, as might otherwise happen should you decide that setting up tests properly with `unittest` from the start would take you too long.[4]

If you do decide to use `doctest` for unit testing, don't cram extra tests into your module's docstrings. This would damage the docstrings by making them too long and hard to read. Keep in the docstrings the right amount and kind of examples, strictly for explanatory purposes, just as if unit testing were not in the picture. Instead, put the extra tests into a global variable of your module, a dictionary named `__test__`. The keys in `__test__` are strings used as arbitrary test names, and the corresponding values are strings that `doctest` picks up and uses in just the same way as it uses docstrings. The values in `__test__` may also be function and class objects, in which case `doctest` examines their docstrings for tests to run. This latter feature is a convenient way to run `doctest` on objects with private names, which `doctest` skips by default.

The `doctest` module also supplies two functions that return instances of the `unittest.TestSuite` class based on doctests, so that you can

integrate such tests into testing frameworks based on `unittest`. Full documentation for this advanced functionality is online.

The unittest Module

The `unittest` module is the Python version of a unit-testing framework originally developed by Kent Beck for Smalltalk. Similar, widespread versions of the framework also exist for many other programming languages (e.g., the `JUnit` package for Java) and are often collectively referred to as `xUnit`.

To use `unittest`, don't put your testing code in the same source file as the tested module: rather, write a separate test module for each module to test. A popular convention is to name the test module like the module being tested, with a prefix such as `'test_'`, and put it in a subdirectory of the source's directory named *test*. For example, the test module for *mod.py* can be *test/test_mod.py*. A simple, consistent naming convention helps you write and maintain auxiliary scripts that find and run all unit tests for a package.

Separation between a module's source code and its unit-testing code lets you refactor the module more easily, including possibly recoding some functionality in C, without perturbing unit-testing code. Knowing that *test_mod.py* stays intact, whatever changes you make to *mod.py*, enhances your confidence that passing the tests in *test_mod.py* indicates that *mod.py* still works correctly after the changes.

A unit-testing module defines one or more subclasses of `unittest`'s `TestCase` class. Each subclass specifies one or more test cases by defining *test-case methods*, methods that are callable without arguments and whose names start with `test`.

The subclass usually overrides `setUp`, which the framework calls to prepare a new instance just before each test case, and, most often, also `tearDown`, which the framework calls to clean things up right after each test case; all of this setup-teardown arrangement is known as a *test fixture*.

Have `setUp` Use `addCleanup` When Needed

When `setUp` propagates an exception, `tearDown` does not execute. So, when `setUp` prepares several things needing eventual cleanup, and some preparation steps might cause uncaught exceptions, `setUp` must not rely on `tearDown` for the clean-up work; rather, right after each preparation step succeeds, call `self.addCleanup(`*f*, **a*, ***k*`)`, passing a clean-up callable *f* (and optionally positional and named arguments for *f*). In this case, *f*`(`**a*, ***k*`)` does get called after the test case (after `tearDown` when `setUp` propagates no exception, but, unconditionally, even when `setUp` does propagate exceptions), so the needed clean-up code always executes.

Each test case calls, on `self`, methods of the class `TestCase` whose names start with `assert` to express the conditions that the test must meet. `unittest` runs the testcase methods within a `TestCase` subclass in arbitrary order, each on a new instance of the subclass, running `setUp` just before each test case and `tearDown` just after each test case.

`unittest` provides other facilities, such as grouping test cases into test suites, per-class and per-module fixtures, test discovery, and other, even more advanced functionality. You do not need such extras unless you're building a custom unit-testing framework on top of `unittest`, or, at the very least, structuring complex testing procedures for equally complex packages. In most cases, the concepts and details covered in this section are enough to perform effective and systematic unit testing. Example 16-2 shows how to use `unittest` to provide unit tests for the module *mod.py* of Example 16-1.

This example uses `unittest` to perform exactly the same tests that Example 16-1 uses as examples in docstrings using `doctest`.

Example 14-2. Using unittest

```
""" This module tests function reverse_words
provided by module mod.py. """
import unittest
import mod

class ModTest(unittest.TestCase):
```

```
    def testNormalCaseWorks(self):
        self.assertEqual(
            mod.reverse_words('four score and seven years'),
            'years seven and score four')

    def testSingleWordIsNoop(self):
        self.assertEqual(
            mod.reverse_words('justoneword'),
            'justoneword')

    def testEmptyWorks(self):
        self.assertEqual(mod.reverse_words(''), '')

    def testRedundantSpacingGetsRemoved(self):
        self.assertEqual(
            mod.reverse_words('with   redundant   spacing'),
            'spacing redundant with')

    def testUnicodeWorks(self):
        self.assertEqual(
            mod.reverse_words('پఝ熵嘼ټ짭牝葺ټ□s all right too'),
            'too right all is پఝ熵嘼ټ짭牝葺ټ□

    def testExactlyOneArgumentIsEnforced(self):
        with self.assertRaises(TypeError):
            mod.reverse_words('one', 'another')

    def testArgumentMustBeString(self):
        with self.assertRaises((AttributeError,TypeError)):
            mod.reverse_words(1)

if __name__=='__main__':
    unittest.main()
```

Running this script with **`python test/test_mod.py`** (or, equivalently, **`python -m test.test_mod`**) is just a bit more verbose than using **`python mod.py`** to run `doctest`, as in Example 16-1. *test_mod.py* outputs a . (dot) for each test case it runs, then a separator line of dashes, and finally a summary line, such as "Ran 7 tests in 0.110s," and a final line of "OK" if every test passed.

Each test-case method makes one or more calls to methods whose names start with `assert`. Here, no method has more than one such call; in more complicated cases, however, multiple calls to assert methods from a single test-case method are quite common.

Even in a case as simple as this, one minor aspect shows that, for unit testing, `unittest` is more powerful and flexible than `doctest`. In the method `testArgumentMustBeString`, we pass as the argument to `assertRaises` a pair of exception classes, meaning we accept either kind of exception. *test_mod.py* therefore accepts as valid multiple implementations of *mod.py*. It accepts the implementation in Example 16-1, which tries calling the method `split` on its argument, and therefore raises `AttributeError` when called with an argument that is not a string. However, it also accepts a different hypothetical implementation, one that raises `TypeError` instead when called with an argument of the wrong type. It **is** possible to code such checks with `doctest`, but it would be awkward and nonobvious, while `unittest` makes it simple and natural.

This kind of flexibility is crucial for real-life unit tests, which to some extent are executable specifications for their modules. You could, pessimistically, view the need for test flexibility as meaning the interface of the code you're testing is not well defined. However, it's best to view the interface as being defined with a useful amount of flexibility for the implementer: under circumstance *X* (argument of invalid type passed to function `reverse_words`, in this example), either of two things (raising `AttributeError` or `TypeError`) is allowed to happen.

Thus, implementations with either of the two behaviors are correct, and the implementer can choose between them on the basis of such considerations as performance and clarity. By viewing unit tests as executable specifications for their modules (the modern view, and the basis of test-driven development), rather than as white-box tests strictly constrained to a specific implementation (as in some traditional taxonomies of testing), you'll find that the tests become an even more vital component of the software development process.

The TestCase class

With `unittest`, you write test cases by extending `TestCase`, adding methods, callable without arguments, whose names start with `test`. Such test-case methods, in turn, call methods that your class inherits from

`TestCase`, whose names start with `assert`, to indicate conditions that must hold for the test to succeed.

The `TestCase` class also defines two methods that your class can optionally override to group actions to perform right before and after each test-case method runs. This doesn't exhaust `TestCase`'s functionality, but you won't need the rest unless you're developing testing frameworks or performing other advanced tasks. The frequently called methods in a `TestCase` instance *`t`* are the following:

assertAlmostEqual	*`t`*`.assertAlmostEqual(`*`first`*`,`*`second`*`, `*`places`*`=7,`*`msg`*`=None)` Fails and outputs *`msg`* when *`first`*`!=`*`second`* to within *`places`* decimal digits; otherwise, does nothing. This method is better than `assertEqual` to compare `floats`, because, due to floating-point computation vagaries, equality in that realm is usually approximate, not 100% exact.
assertEqual	*`t`*`.assertEqual(`*`first`*`,`*`second`*`,`*`msg`*`=None)` Fails and outputs *`msg`* when *`first`*`!=`*`second`*; otherwise, does nothing.
assertFalse	*`t`*`.assertFalse(`*`condition`*`, `*`msg`*`=None)` Fails and outputs *`msg`* when *`condition`* is true; otherwise, does nothing.
assertNotAlmostEqual	*`t`*`.assertNotAlmostEqual(`*`first`*`,`*`second`*`,` *`places`*`=7,`*`msg`*`=None)` Fails and outputs *`msg`* when *`first`*`==`*`second`* to within *`places`* decimal digits; otherwise, does nothing.
assertNotEqual	*`t`*`.assertNotEqual(`*`first`*`,`*`second`*`,`*`msg`*`=None)` Fails and outputs *`msg`* when *`first`*`==`*`second`*; otherwise, does nothing.
assertRaises	*`t`*`.assertRaises(`*`exceptionSpec`*`,`*`callable`*`, `*`*args`*`, `*`**kwargs`*`)` Calls *`callable(*args,**kwargs)`*. Fails when the call doesn't raise any exception. When the call raises an exception that does not meet *`exceptionSpec`*, `assertRaises` propagates the exception. When the call raises an exception that meets *`exceptionSpec`*, `assertRaises` does nothing. *`exceptionSpec`* can be an exception class or a tuple of classes, just like the first argument of the `except` clause in a `try/except` statement. An alternative, and usually preferable, way to use `assertRaises` is as a *context manager*—that is, in a `with` statement:

	`with self.assertRaises(exceptionSpec):`
	`  ...a block of code...`
	
	Here, the "block of code" indented in the `with` statement executes, rather than just the *`callable`* being called with certain arguments. The expectation (which avoids the construct failing) is that the block of code raises an exception meeting the given exception specification (an exception class or a tuple of classes). This alternative approach is more general and readable.

assertTrue	*`t.assertTrue(condition,msg`*=None) Fails and outputs *`msg`* when *`condition`* is false; otherwise, does nothing. Do not use this method when you can use a more specific one, such as `assertEqual`: specific methods provide clearer messages.

fail	*`t`*`.fail(`*`msg`*=None)
	Fails unconditionally and outputs *`msg`*. An example snippet might be:
	
	`if not complex_check_if_its_ok(some, thing):`
	`  self.fail(`
	`  'Complex checks failed on'`
	`  f' {some}, {thing}'`
	`  )`

setUp	*`t.setUp()`* The framework calls *`t`*`.setUp()` just before calling each test-case method. `setUp` in `TestCase` does nothing. `setUp` exists only to let your class override the method when your class needs to perform some preparation for each test.
subTest	*t.subTest(msg=None, **k)* Returns a context manager that can define a portion of a test within a test method. Use `subTest` when a test method runs the same test multiple times with varying parameters. Enclosing these parameterized tests in `subTest` ensures that all the cases will be run, even if some of them fail or raise exceptions.
tearDown	*`t.tearDown()`* The framework calls *`t`*`.tearDown()` just after each test-case method. `tearDown` in `TestCase` does nothing. `tearDown` exists only to let your class override the method when your class needs to perform some cleanup after each test.

In addition, a `TestCase` instance maintains a last-in, first-out list (*LIFO* stack) of *clean-up functions*. When code in one of your tests (or in `setUp`) does something that requires cleanup, call `addCleanup`, passing a clean-up callable `f` and optionally positional and named arguments for `f`. To perform the stacked-up cleanups, you may call `doCleanups`; however, the framework itself calls `doCleanups` after `tearDown`. Here are the signatures of the two cleanup methods:

addCleanup	*t*.addCleanup(*func*, **a, **k*) Appends *(func, a, k)* at the end of the cleanups' list.

doCleanups	*t*.doCleanups()
	Perform all cleanups, if any is stacked. Substantially equivalent to:
	`while self.list_of_cleanups:`
	`  func, a, k = self.list_of_cleanups.pop()`
	`  func(*a, **k)`
	for a hypothetical stack *self.list_of_cleanups*, plus, of course, error-checking and reporting.

Unit tests dealing with large amounts of data

Unit tests must be fast: run them often as you develop. So, unit-test each aspect of your modules on small amounts of data, when feasible. This makes unit tests faster, and lets you embed the data in the test's source code. When you test a function that reads from or writes to a file object, use an instance of the class `io.TextIO` for a text—that is, Unicode—file (`io.BytesIO` for a byte, i.e., binary file, as covered in "In-Memory "Files": io.StringIO and io.BytesIO")—to get a "file" with the data in memory: faster than writing to disk, and no clean-up chore (removing disk files after the tests).

In rare cases, it may be impossible to exercise a module's functionality without supplying and/or comparing data in quantities larger than can be reasonably embedded in a test's source code. In such cases, your unit test

must rely on auxiliary, external data files to hold the data to supply to the module it tests and/or the data it needs to compare to the output. Even then, you're generally better off using instances of the above-mentioned `io` classes, rather than directing the tested module to perform actual disk I/O. Even more important, we strongly suggest that you generally use stubs to unit-test modules that interact with external entities, such as databases, GUIs, or other programs over a network. It's easier to control all aspects of the test when using stubs rather than real external entities. Also, to reiterate, the speed at which you can run unit tests is important, and it's faster to perform simulated operations in stubs, rather than real operations.

To Test, Make Randomness Reproducible By Supplying a Seed

If your code uses pseudorandom numbers (e.g., as covered in "The random Module"), you can make it easier to test by ensuring its "random" behavior is reproducible: specifically, ensure that it's easy for your tests to have `random.seed` called with a known argument, so that the ensuing pseudorandom numbers are fully predictable. This also applies when you use pseudorandom numbers to set up your tests by generating inputs: such generation should default to a known seed, to be used in most testing, keeping the flexibility of changing seeds only for specific techniques such as fuzzing.

Testing with pytest

The `pytest` module is a pip-installable third-party unit-testing framework that introspects a project's modules to find test cases in *test_*.py* or **_test.py* files, with method names starting with `"test"` at module level, or in classes with names starting with `"Test"`. Unlike the builtin `unittest` framework, `pytest` does not require that test cases extend any testing class hierarchy; it runs the discovered test methods, which use Python `assert` statements to determine the success or failure of each test. If a test raises any exception other than `AssertionError`, that indicates that there is an error in the test, rather than a simple test-failure.

In place of a hierarchy of test case classes, `pytest` provides a number of helper methods and decorators to simplify writing unit tests. The most common methods are listed below; consult the online docs for a more complete list of methods and optional arguments.

approx	*approx(float_value)*
	Used to support asserts that must compare floating-point values. *`float_value`* can be a single value or a sequence of values:
	`assert 0.1 + 0.2 == approx(0.3)`
	`assert ([0.1, 0.2, 0.1+0.2]) == approx([0.1, 0.2, 0.3])`

skip	*skip(skip_reason)* Forces skipping of the current test; use it, for example, when a test is dependent on a previous test that has already failed.
fail	*fail(failure_reason)* Forces failure of the current test. More explicit than injecting an `assert False` statement, but otherwise equivalent.
raises	*raises(expected_exception, match=regex_match)* A context manager that fails unless its context raises an exception *`exc`* such that `isinstance(`*`exc, expected_exception`*`)` is true. When *`regex_match`* is given, the test fails unless *`exc`*'s `str` representation also matches `re.search(`*`regex_match, str(exc)`*`)`.
warns	*warns(expected_warning, match=regex_match)* Similar to *`raises`*, to wrap code that tests that an expected warning is raised.

The `pytest.mark` subpackage includes decorators to "mark" test methods with additional test behavior. These include the following:

skip, skipif	*@skip(skip_reason)* *@skipif(condition, skip_reason)* Skips a test method, optionally based on some global condition.
parametrize	*@parametrize(args_string, arg_test_values)* Calls the decorated test method, setting the arguments named in the comma-separated list *`args_string`* to the values from each argument tuple in *`arg_test_values`*. The following code runs *`test_is_greater`* twice, once with `x=1, y=0,` and `expected=True`, and once with `x=0, y=1,` and `expected=False`.

```
@parametrize("x,y,expected",
  [(1, 0, True), (0, 1, False)])
def test_is_greater(x, y, expected):
  assert (x > y) == expected
```

Debugging

Since Python's development cycle is fast, the most effective way to debug is often to edit your code to output relevant information at key points. Python has many ways to let your code explore its own state in order to extract information that may be relevant for debugging. The `inspect` and `traceback` modules specifically support such exploration, which is also known as reflection or introspection.

Once you have debugging-relevant information, `print` is often the way to display it (`pprint`, covered in "The pprint Module", is also often a good choice). Better yet, log debugging information to files. Logging is useful for programs that run unattended (e.g., server programs). Displaying debugging information is just like displaying other information, as covered in Chapter 10. Logging such information is like writing to files (covered in Chapter 10); however, to help with the frequent task of logging, Python's standard library supplies a `logging` module, covered in "The logging package". As covered in Table 7-3, rebinding `excepthook` in the module `sys` lets your program log error info just before terminating with a propagating exception.

Python also offers hooks to enable interactive debugging. The `pdb` module supplies a simple text-mode interactive debugger. Other, powerful interactive debuggers for Python are part of integrated development environments (IDEs), such as IDLE and various commercial offerings, as mentioned in "Python Development Environments"; we do not cover these debuggers further.

Before You Debug

Before you embark on lengthy debugging explorations, make sure you have thoroughly checked your Python sources with the tools mentioned in Chapter 2. Such tools catch only a subset of the bugs in your code, but they're much faster than interactive debugging: their use amply repays itself.

Moreover, again before starting a debugging session, make sure that all the code involved is well covered by unit tests, covered in "The unittest Module". As mentioned earlier in the chapter, once you have found a bug, *before* you fix it, add to your suite of unit tests (or, if need be, to the suite of system tests) a test or two that would have found the bug had they been present from the start, and run the tests again to confirm that they now reveal and isolate the bug; only once that is done should you proceed to fix the bug. By regularly following this procedure, you get a much better suite of tests; learn to write better, more thorough tests; and gain much sounder assurance about the overall, *enduring* correctness of your code.

Remember, even with all the facilities offered by Python, its standard library, and whatever IDEs you fancy, debugging is still *hard*. Take this into account even before you start designing and coding: write and run plenty of unit tests, and keep your design and code *simple*, to reduce to the minimum the amount of debugging you will need! Classic advice about this, by Brian Kernighan: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as you can, you are, by definition, not smart enough to debug it." This is part of why "clever" is not a positive word when used to describe Python code, or a coder...

The inspect Module

The `inspect` module supplies functions to get information about all kinds of objects, including the Python call stack (which records all function calls currently executing) and source files. The most frequently used functions of `inspect` are as follows:

getargspec, formatargspec
`getargspec(f)`
||--*3.11*|| Deprecated in Python 3.5, planned to be removed in Python 3.11. The forward-compatible method to introspect callables is to use

`inspect.signature(f)` and the resulting instance of class `inspect.Signature`, covered in "Introspecting callables".

`f` is a function object. `getargspec` returns a named tuple with four items: `(args, varargs, keywords, defaults)`. `args` is the sequence of names of `f`'s parameters. `varargs` is the name of the special parameter of the form `*a`, or `None` when `f` has no such parameter. `keywords` is the name of the special parameter of the form `**k`, or `None` when `f` has no such parameter. `defaults` is the tuple of default values for `f`'s arguments. You can deduce other details of `f`'s signature from `getargspec`'s results: `f` has `len(args)-len(defaults)` mandatory positional arguments, and the names of `f`'s optional arguments are the strings that are the items of the list slice `args[-len(defaults):]`.

`formatargspec` accepts one to four arguments, same as the items of the named tuple that `getargspec` returns, and returns a string with this information. Thus, `formatargspec(*getargspec(f))` returns a string with `f`'s parameters (also known as `f`'s *signature*) in parentheses, as in the `def` statement that created `f`. For example:

```
import inspect
def f(a,b=23,**c): pass
print(inspect.formatargspec(*inspect.getargspec(f)
))
# prints: (a, b=23, **c)
```

getargvalues, formatargvalues

`getargvalues(f)`.

`f` is a frame object—for example, the result of a call to the function `_getframe` in module `sys` (covered in "The frame Type") or to function `currentframe` in module `inspect`. `getargvalues` returns a named-tuple with four items:

`(args, varargs, keywords, locals)`. `args` is the sequence of names of `f`'s function's parameters. `varargs` is the name of the special parameter of form `*a`, or `None` when `f`'s function has no such parameter. `keywords` is the name of the special parameter of form `**k`, or `None` when `f`'s function has no such parameter. `locals` is the dictionary of local variables for `f`. Since arguments, in particular, are local variables, the value of each argument can be obtained from `locals` by indexing the `locals` dictionary with the argument's corresponding parameter name.

`formatargvalues` accepts one to four arguments that are the same as the items of the named tuple that `getargvalues` returns, and returns a string with this information. `formatargvalues(*getargvalues(f))` returns a string with `f`'s arguments in parentheses, in named form, as used in the call statement that created `f`. For example:

```
def f(x=23):
  return inspect.currentframe()
print(inspect.formatargvalues(
```

```
  *inspect.getargvalues(f())))
# prints: (x=23)
```

currentframe	`currentframe()` Returns the frame object for the current function (the caller of `currentframe`). `formatargvalues(*getargvalues(currentframe()))`, for example, returns a string with the arguments of the calling function.
getdoc	`getdoc(`*`obj`*`)` Returns the docstring for *`obj`*, a multiline string with tabs expanded to spaces and redundant whitespace stripped from each line.
getfile, getsourcefile	`getfile(`*`obj`*`)` Returns the name of the file that defined *`obj`*; raises `TypeError` when unable to determine the file. For example, `getfile` raises `TypeError` when *`obj`* is built-in. `getfile` returns the name of a binary or source file. `getsourcefile` returns the name of a source file and raises `TypeError` when all it can find is a binary file, not the corresponding source file.

getmembers	`getmembers(`*`obj, filter`*`=None)`
	Returns all attributes (members), both data and methods (including special methods) of *`obj`*, a sorted list of *`(name, value)`* pairs. When *`filter`* is not `None`, returns only attributes for which callable *`filter`* is true when called on the attribute's *`value`*, like:
	`sorted((n, v) for n, v in getmembers(obj) if filter(v))`

getmodule	`getmodule(`*`obj`*`)` Returns the module object that defined *`obj`*, or `None` when it is unable to determine it.

getmro	`getmro(`*`c`*`)`
	Returns a tuple of bases and ancestors of class *`c`* in "method resolution order". *`c`* is the first item in the tuple. Each class appears only once in the tuple. For example:
	`class A: pass`
	`class B(A): pass`
	`class C(A): pass`
	`class D(B, C): pass`
	`for c in inspect.getmro(D):`
	`  print(c.__name__, end=' ')`
	`# prints: D B C A object`

getsource, getsourcelines	`getsource(obj)` Returns a multiline string that is the source code for *`obj`*; raises `IOError` if it is unable to determine or fetch it. `getsourcelines` returns a pair: the first item is the source code for *`obj`* (a list of lines), and the second item is the line number of the first line within its file.
isbuiltin, isclass, iscode, isframe, isfunction, ismethod, ismodule, isroutine	`isbuiltin(obj), etc` Each of these functions accepts a single argument *`obj`* and returns `True` when *`obj`* is of the kind indicated in the function name. Accepted objects are, respectively: built-in (C-coded) functions, class objects, code objects, frame objects, Python-coded functions (including `lambda` expressions), methods, modules, and—for `isroutine`—all methods or functions, either C-coded or Python-coded. These functions are often used as the *`filter`* argument to `getmembers`.
stack	`stack(context=1)` Returns a list of six-item tuples. The first tuple is about `stack`'s caller, the second about the caller's caller, and so on. Each tuple's items, are: frame object, filename, line number, function name, list of *`context`* source lines around the current line, index of current line within the list.

Introspecting callables

To introspect a callable's signature, call `inspect.signature(`*`f`*`)`, which returns an instance *`s`* of class `inspect.Signature`.

`s``.parameters` is a `dict` mapping parameter names to `inspect.Parameter` instances. Call *`s`*`.bind(*`*`a`*`, **`*`k`*`)` to bind all parameters to the given positional and named arguments, or *`s`*`.bind_partial(*`*`a`*`, **`*`k`*`)` to bind a subset of them: each returns an instance *`b`* of `inspect.BoundArguments`.

For detailed information and examples on how to introspect callables' signatures through these classes and their methods, see PEP 362.

An example of using inspect

Suppose that somewhere in your program you execute a statement such as:

```
x.f()
```

and unexpectedly receive an `AttributeError` informing you that object *`x`* has no attribute named *`f`*. This means that object *`x`* is not as you expected,

so you want to determine more about *x* as a preliminary to ascertaining why *x* is that way and what you should do about it. Change the statement to:

```
try:
    x.f()
except AttributeError:
    import sys, inspect
    print(f'x is type {type(x)}, ({x!r})', file=sys.stderr)
    print("x's methods are:", file=sys.stderr, end='')
    for n, v in inspect.getmembers(x, callable):
        print(n, file=sys.stderr, end=' ')
    print(file=sys.stderr)
    raise
```

This example uses `sys.stderr` (covered in Table 7-3), since it displays information related to an error, not program results. The function `getmembers` of the module `inspect` obtains the name of all the methods available on *x* in order to display them. If you need this kind of diagnostic functionality often, package it up into a separate function, such as:

```
import sys, inspect
def show_obj_methods(obj, name, show=sys.stderr.write):
    show(f'{name} is type {type(obj)}({obj!r})\n')
    show(f"{name}'s methods are: ")
    for n, v in inspect.getmembers(obj, callable):
       show(f'{n} ')
    show('\n')
```

And then the example becomes just:

```
try:
    x.f()
except AttributeError:
    show_obj_methods(x, 'x')
    raise
```

Good program structure and organization are just as necessary in code intended for diagnostic and debugging purposes as they are in code that implements your program's functionality. See also "The __debug__ Built-in

Variable" for a good technique to use when defining diagnostic and debugging functions.

The traceback Module

The `traceback` module lets you extract, format, and output information about tracebacks which uncaught exceptions normally produce. By default, the `traceback` module reproduces the formatting Python uses for tracebacks. However, the `traceback` module also lets you exert fine-grained control. The module supplies many functions, but in typical use you need only one of them:

print_exc	`print_exc(`*`limit`*`=None, `*`file`*`=sys.stderr)` Call `print_exc` from an exception handler, or a function called, directly or indirectly, by an exception handler. `print_exc` outputs to file-like object *`file`* the traceback that Python outputs to `stderr` for uncaught exceptions. When *`limit`* is an integer, `print_exc` outputs only *`limit`* traceback nesting levels. For example, when, in an exception handler, you want to cause a diagnostic message just as if the exception propagated, but stop the exception from propagating further (so that your program keeps running and no further handlers are involved), call `traceback.print_exc()`.

The pdb Module

The `pdb` module uses the Python interpreter's debugging and tracing hooks to implement a simple command-line interactive debugger. `pdb` lets you set breakpoints, single-step on source code, examine stack frames, and so on.

To run code under `pdb`'s control, import `pdb`, then call `pdb.run`, passing as the single argument a string of code to execute. To use `pdb` for post-mortem debugging (debugging of code that just terminated by propagating an exception at an interactive prompt), call `pdb.pm()` without arguments. To trigger `pdb` directly from your application code, use the builtin function `breakpoint()`.

When `pdb` starts, it first reads text files named *.pdbrc* in your home directory and in the current directory. Such files can contain any `pdb` commands, but most often you put `alias` commands in them in order to

define useful synonyms and abbreviations for other commands that you use often.

When `pdb` is in control, it prompts with the string `'(Pdb) '`, and you can enter `pdb` commands. The command `help` (which you can enter in the abbreviated form `h`) lists available commands. Call `help` with an argument (separated by a space) to get help about any specific command. You can abbreviate most commands to the first one or two letters, but you must always enter commands in lowercase: `pdb`, like Python itself, is case-sensitive. Entering an empty line repeats the previous command. The most frequently used `pdb` commands are listed in Table 16-1.

!	`! statement` Executes Python statement *`statement`* in the currently-debugged context.
alias, unalias	`alias [ name [ command] ]` Defines a short-form of a frequently used command. *`command`* is any `pdb` command, with arguments, and may contain `%1`, `%2`, and so on to refer to specific arguments passed to the new alias *`name`* being defined, or `%*` to refer to all such arguments. `alias` with no arguments lists currently defined aliases. `alias` *`name`* outputs the current definition of alias *`name`*. Command `unalias` *`name`* removes an alias.
args, a	`args` Lists all arguments passed to the function you are currently debugging.
break, b	`break [ location [ ,condition ] ]` `break` with no arguments lists currently defined breakpoints and the number of times each breakpoint has triggered. With an argument, `break` sets a breakpoint at the given *`location`*.*`location`* can be a line number or a function name, optionally preceded by *`filename`*`:` to set a breakpoint in a file that is not the current one or at the start of a function whose name is ambiguous (i.e., a function that exists in more than one file). When *`condition`* is present, it is an expression to evaluate (in the debugged context) each time the given line or function is about to execute; execution breaks only when the expression returns a true value. When setting a new breakpoint, `break` returns a breakpoint number, which you can then use to refer to the new breakpoint in any other breakpoint-related `pdb` command.
	`clear [ breakpoint-numbers ]` Clears (removes) one or more breakpoints. `clear` with no arguments

clear, cl	removes all breakpoints after asking for confirmation. To deactivate a breakpoint without removing it, see `disable`, covered below.
condition	`condition` *`breakpoint-number`* `[` *`expression`* `]` `condition` *`n expression`* sets or changes the condition on breakpoint *`n`*. `condition` *`n`*, without *`expression`*, makes breakpoint *`n`* unconditional.
continue, c, cont	`continue` Continues execution of the code being debugged, up to a breakpoint, if any.
disable	`disable [` *`breakpoint-numbers`* `]` Disables one or more breakpoints. `disable` without arguments disables all breakpoints (after asking for confirmation). This differs from `clear` in that the debugger remembers the breakpoint, and you can reactivate it via `enable`.
down, d	`down` Moves down one frame in the stack (i.e., toward the most recent function call). Normally, the current position in the stack is at the bottom (i.e., at the function that was called most recently and is now being debugged), so command `down` can't go further down. However, command `down` is useful if you have previously executed command `up`, which moves the current position upward.
enable	`enable [` *`breakpoint-numbers`* `]` Enables one or more breakpoints. `enable` without arguments enables all breakpoints after asking for confirmation.

ignore	`ignore` *`breakpoint-number`* `[` *`count`* `]`
	Sets the breakpoint's ignore count (to `0` if *`count`* is omitted). Triggering a breakpoint whose ignore count is greater than `0` just decrements the count. Execution stops, presenting you with an interactive `pdb` prompt, only when you trigger a breakpoint whose ignore count is `0`. For example, say that module *fob.py* contains the following code:
	
	`def f():`
	`  for i in range(1000):`
	`  g(i)`
	`  def g(i):`
	`  pass`
	
	Now consider the following interactive `pdb` session (minor formatting details may change depending on the Python version you're running):

```
>>> import pdb
>>> import fob
>>> pdb.run('fob.f()')
> <string>(1)?()
(Pdb) break fob.g
Breakpoint 1 at C:\mydir\fob.py:5
(Pdb) ignore 1 500
Will ignore next 500 crossings of breakpoint 1.
(Pdb) continue
> C:\mydir\fob.py(5)
g()-> pass
(Pdb) print(i)
500
```

The `ignore` command, as `pdb` says, tells `pdb` to ignore the next `500` hits on breakpoint `1`, which we set at *`fob.g`* in the previous `break` statement. Therefore, when execution finally stops, function g has already been called `500` times, as we show by printing its argument `i`, which indeed is now `500`. The ignore count of breakpoint `1` is now `0`; if we give another `continue` and `print i, i` shows as `501`. In other words, once the ignore count decrements to `0`, execution stops every time the breakpoint is hit. If we want to skip some more hits, we must give `pdb` another `ignore` command, setting the ignore count of breakpoint `1` at some value greater than `0` yet again.

list, l	`list [` *`first`* `[ , `*`last`*` ] ]` `list` without arguments lists `11` (eleven) lines centered on the current one, or the next `11` lines if the previous command was also a `list`. Arguments to the `list` command can optionally specify the first and last lines to list within the current file. The `list` command lists physical lines, including comments and empty lines, not logical lines.
next, n	`next` Executes the current line, without "stepping into" any function called from the current line. However, hitting breakpoints in functions called directly or indirectly from the current line does stop execution.
print, p	`p` *`expression`* or `print(`*`expression`*`)` Evaluates *`expression`* in the current context and displays the result.
quit, q	`quit` Immediately terminates both `pdb` and the program being debugged.
return, r	`return` Executes the rest of the current function, stopping only at breakpoints, if any.

step, s	`step` Executes the current line, stepping into any function called from the current line.
tbreak	`tbreak [` *`location`* `[ ,`*`condition`* `] ]` Like `break`, but the breakpoint is temporary (i.e., `pdb` automatically removes the breakpoint as soon as the breakpoint is triggered).
up, u	`up` Moves up one frame in the stack (i.e., away from the most recent function call and toward the calling function).
where, w	`where` Shows the stack of frames and indicates the current one (i.e., in which frame's context command `!` executes statements, command `args` shows arguments, command *`print`* evaluates expressions, etc.).

You can also enter a Python expression at the `(Pdb)` prompt and it will be evaluated and the result displayed, just as if you were at the Python interpreter prompt. However, if you enter an expression whose first term coincides with a `pdb` command, then the `pdb` command will be executed. This is especially problematic when debugging code with single letter variables like *`p`* and *`q`*. In these cases, you must begin the expression with `!` or precede it with the `print` or `p` command.

The warnings Module

Warnings are messages about errors or anomalies that aren't serious enough to disrupt the program's control flow (as would happen by raising an exception). The `warnings` module affords fine-grained control over which warnings are output and what happens to them. You can conditionally output a warning by calling the function `warn` in the `warnings` module. Other functions in the module let you control how warnings are formatted, set their destinations, and conditionally suppress some warnings or transform some warnings into exceptions.

Classes

Exception classes that represent warnings are not supplied by `warnings`: rather, they are built-ins. The class `Warning` subclasses `Exception` and is the base class for all warnings. You may define your own warning classes; they must subclass `Warning`, either directly or via one of its other existing subclasses, which include:

`DeprecationWarning`

Use of deprecated features supplied only for backward compatibility

`RuntimeWarning`

Use of features whose semantics are error-prone

`SyntaxWarning`

Use of features whose syntax is error-prone

`UserWarning`

Other user-defined warnings that don't fit any of the above cases

Objects

Python supplies no concrete warning objects. A warning is made up of a *`message`* (a string), a *`category`* (a subclass of `Warning`), and two pieces of information to identify where the warning was raised from: *`module`* (name of the module that raised the warning) and *`lineno`* (line number of the source code line raising the warning). Conceptually, you may think of these as attributes of a warning object *`w`*: we use attribute notation later for clarity, but no specific object *`w`* actually exists.

Filters

At any time, the `warnings` module keeps a list of active filters for warnings. When you import `warnings` for the first time in a run, the module examines `sys.warnoptions` to determine the initial set of

filters. You can run Python with the option **`-W`** to set `sys.warnoptions` for a given run. Do not rely on the initial set of filters being held specifically in `sys.warnoptions`, as this is an implementation detail that may change in future versions of Python.

As each warning *`w`* occurs, `warnings` tests *`w`* against each filter until a filter matches. The first matching filter determines what happens to *`w`*. Each filter is a tuple of five items. The first item, *`action`*, is a string that defines what happens on a match. The other four items, *`message`*, *`category`*, *`module`*, and *`lineno`*, control what it means for *`w`* to match the filter: for a match, all conditions must be satisfied. Here are the meanings of these items (using attribute notation to indicate conceptual attributes of *`w`*):

`message`

A regular expression pattern string; the match condition is `re.match(`*`message`*`,`*`w`*`.message, re.I)` (the match is case-insensitive).

`category`

`Warning` or a subclass of `Warning`; the match condition is `issubclass(`*`w`*`.category, `*`category`*`)`.

`module`

A regular expression pattern string; the match condition is `re.match(`*`module`*`,`*`w`*`.module)` (the match is case-sensitive).

`lineno`

An `int`; the match condition is *`lineno`* `in (`*`0`*`, `*`w`*`.lineno)`: that is, either *`lineno`* is `0`, meaning *`w`*`.lineno` does not matter, or *`w`*`.lineno` must exactly equal *`lineno`*.

Upon a match, the first field of the filter, the *`action`*, determines what happens:

`'always'`

w`.message` is output whether or not *w* has already occurred.

`'default'`

w`.message` is output if, and only if, this is the first time *w* occurs from this specific location (i.e., this specific *w*`.module`, *w*`.location` pair).

`'error'`

w`.category(`*w*`.message)` is raised as an exception.

`'ignore'`

w is ignored.

`'module'`

w`.message` is output if, and only if, this is the first time *w* occurs from *w*`.module`.

`'once'`

w`.message` is output if, and only if, this is the first time *w* occurs from any location.

__warningsgregistry__

When a module issues a warning, `warnings` adds to that module's global variables a `dict` named `__warningsgregistry__`, if that `dict` is not already present. Each key in the `dict` is a pair *`(message, category)`*, or a tuple with three items *`(message, category, lineno)`*; the corresponding value is `True` when further occurrences of that message are to be suppressed. Thus, for example, you can reset the suppression state of all warnings from a module *m* by executing

m.__warningsregistry__.clear(): when you do that, all messages get output again (once) even if, for example, they've previously triggered a filter with an *action* of 'module'.

Functions

The warnings module supplies the following functions:

filterwarnings	filterwarnings(*action*,*message*='.*',*category*=Warning, *module*='.*', *lineno*=0, *append*=False) Adds a filter to the list of active filters. When *append* is true, filterwarnings adds the filter after all other existing filters (i.e., appends the filter to the list of existing filters); otherwise, filterwarnings inserts the filter before any other existing filter. All components, save *action*, have default values that mean "match everything." As detailed above, *message* and *module* are pattern strings for regular expressions, *category* is some subclass of Warning, *lineno* is an integer, and *action* is a string that determines what happens when a message matches this filter.
formatwarning	formatwarning(*message*,*category*,*filename*, *lineno*) Returns a string that represents the given warning with standard formatting.
resetwarnings	resetwarnings() Removes all filters from the list of filters. resetwarnings also discards any filters originally added with the **-W** command-line option.
showwarning	showwarning(*message*,*category*, *filename*, *lineno*, *file*=sys.stderr) Outputs the given warning to the given file object. Filter actions that output warnings call showwarning, letting the argument *file* default to sys.stderr. To change what happens when filter actions output warnings, code your own function with this signature and bind it to warnings.showwarning, thus overriding the default implementation.

warn	warn(*message*,*category*=UserWarning,*stacklevel*=1)
	Sends a warning so that the filters examine and possibly output it. The location of the warning is the current function (caller of warn) if *stacklevel* is 1, or the caller of the current function if *stacklevel* is 2. Thus, passing 2 as the value of *stacklevel* lets you write functions that send warnings on their caller's behalf, such as:
	`def toUnicode(bytestr):`
	`  try:`
	`  return bytestr.decode()`

```
except UnicodeError:
  warnings.warn(
    f'Invalid characters in {bytestr!r}',
    stacklevel=2)
  return bytestr.decode(errors='ignore')
```

Thanks to the parameter `stacklevel=2`, the warning appears to come from the caller of `toUnicode`, rather than from `toUnicode` itself. This is very important when the `action` of the filter that matches this warning is `default` or `module`, since these actions output a warning only the first time the warning occurs from a given location or module.

Optimization

"First make it work. Then make it right. Then make it fast." This quotation, often with slight variations, is widely known as "the golden rule of programming." As far as we've been able to ascertain, the quotation is by Kent Beck, who credits his father with it. This principle is often quoted, but too rarely followed. A negative form, slightly exaggerated for emphasis, is in a quotation by Don Knuth (who credits Hoare with it): "Premature optimization is the root of all evil in programming."

Optimization is premature if your code is not working yet, or if you're not sure what, precisely, your code should be doing (since then you cannot be sure if it's working). First make it work: ensure that your code is correctly performing exactly the tasks it is *meant* to perform.

Optimization is also premature if your code is working but you are not satisfied with the overall architecture and design. Remedy structural flaws before worrying about optimization: first make it work, then make it right. These steps are not optional; working, well-architected code is *always* a must.

In contrast, you don't always need to make it fast. Benchmarks may show that your code's performance is already acceptable after the first two steps. When performance is not acceptable, profiling often shows that all performance issues are in a small part of the code, perhaps 10 to 20 percent of the code where your program spends 80 or 90 percent of the time. Such

performance-crucial regions of your code are known as *bottlenecks*, or *hot spots*. It's a waste of effort to optimize large portions of code that account for, say, 10 percent of your program's running time. Even if you made that part run 10 times as fast (a rare feat), your program's overall runtime would only decrease by 9 percent, a speedup no user would even notice. If optimization is needed, focus your efforts where they matter: on bottlenecks. You can optimize bottlenecks while keeping your code 100 percent pure Python, thus not preventing future porting to other Python implementations. In some cases, you can resort to recoding some computational bottlenecks as Python extensions (as covered in Chapter "Extending and Embedding Classic Python"), potentially gaining even better performance (possibly at the expense of some potential future portability).

Developing a Fast-Enough Python Application

Start by designing, coding, and testing your application in Python, using available extension modules if they save you work. This takes much less time than it would with a classic compiled language. Then benchmark the application to find out if the resulting code is fast enough. Often it is, and you're done—congratulations! Ship it!

Since much of Python itself is coded in highly optimized C (as are many of its standard library and extension modules), your application may even turn out to already be faster than typical C code. However, if the application is too slow, you need, first and foremost, to rethink your algorithms and data structures. Check for bottlenecks due to application architecture, network traffic, database access, and operating system interactions. For typical applications, each of these factors is more likely than language choice to cause slowdowns. Tinkering with large-scale architectural aspects can often dramatically speed up an application, and Python is an excellent medium for such experimentation.

If your program is still too slow, profile it to find out where the time is going. Applications often exhibit computational bottlenecks: small areas of the source code—often 20% or less of it—account for 80% or more of the

running time[5]. Optimize the bottlenecks, applying the techniques suggested in the rest of this chapter.

If normal Python-level optimizations still leave some outstanding computational bottlenecks, you can recode those as Python extension modules, as covered in Chapter “Extending and Embedding Classic Python”. In the end, your application runs at roughly the same speed as if you had coded it all in C, C++, or Fortran—or faster, when large-scale experimentation has let you find a better architecture. Your overall programming productivity with this process is not much less than if you coded everything in Python. Future changes and maintenance are easy, since you use Python to express the overall structure of the program, and lower-level, harder-to-maintain languages for only a few specific computational bottlenecks.

As you build applications in a given area following this process, you accumulate a library of reusable Python extension modules. You therefore become more and more productive at developing other fast-running Python applications in the same field.

Even if external constraints eventually force you to recode the whole application in a lower-level language, you’re still better off for having started in Python. Rapid prototyping has long been acknowledged as the best way to get software architecture just right. A working prototype lets you check that you have identified the right problems and taken a good path to their solution. A prototype also affords the kind of large-scale architectural experiments that can make a real difference in performance. Starting your prototype with Python allows a gradual migration to other languages by way of extension modules, if need be. The application remains fully functional and testable at each stage. This ensures against the risk of compromising a design’s architectural integrity in the coding stage. The resulting software is faster and more robust than if all of the coding had been lower-level from the start, and your productivity—while not quite as good as with a pure Python application—is still better than if you had been coding at a lower level throughout.

Benchmarking

Benchmarking (also known as *load testing*) is similar to system testing: both activities are much like running the program for production purposes. In both cases, you need to have at least some subset of the program's intended functionality working, and you need to use known, reproducible inputs. For benchmarking, you don't need to capture and check your program's output: since you make it work and make it right before you make it fast, you're already fully confident about your program's correctness by the time you load-test it. You do need inputs that are representative of typical system operations, ideally ones that may be most challenging for your program's performance. If your program performs several kinds of operations, make sure you run some benchmarks for each different kind of operation.

Elapsed time as measured by your wristwatch is probably precise enough to benchmark most programs. Programs with hard real-time constraints are another matter, but they have needs very different from those of normal programs in most respects. A 5 or 10 percent difference in performance, except in programs with very peculiar constraints, makes no practical difference to a program's real-life usability.

When you benchmark "toy" programs or snippets in order to help you choose an algorithm or data structure, you may need more precision: the `timeit` module of Python's standard library (covered in "The timeit module") is quite suitable for such tasks. The benchmarking discussed in this section is of a different kind: it is an approximation of real-life program operation for the sole purpose of checking whether the program's performance at each task is acceptable, before embarking on profiling and other optimization activities. For such "system" benchmarking, a situation that approximates the program's normal operating conditions is best, and high accuracy in timing is not all that important.

Large-Scale Optimization

The aspects of your program that are most important for performance are large-scale ones: your choice of overall architecture, algorithms, and data structures.

The performance issues that you must often take into account are those connected with the traditional big-O notation of computer science. Informally, if you call `N` the input size of an algorithm, big-O notation expresses algorithm performance, for large values of `N`, as proportional to some function of `N`. (In precise computer science lingo, this should be called big-Theta, but, in real life, programmers call this big-O, perhaps because an uppercase Theta looks a bit like an O with a dot in the center!)

An `O(1)` algorithm (also known as "constant time") is one that takes a time not growing with `N`. An `O(N)` algorithm (also known as "linear time") is one where, for large enough `N`, handling twice as much data takes about twice as much time, three times as much data three times as much time, and so on, proportionally to `N`. An $O(N^2)$ algorithm (also known as a "quadratic time" algorithm) is one where, for large enough `N`, handling twice as much data takes about four times as much time, three times as much data nine times as much time, and so on, growing proportionally to `N` squared. Identical concepts and notation are used to describe a program's consumption of memory ("space") rather than of time.

To find more information on big-O notation, and about algorithms and their complexity, any good book about algorithms and data structures can help; we recommend Magnus Lie Hetland's excellent book *Python Algorithms: Mastering Basic Algorithms in the Python Language*, 2nd edition (Apress, 2015).

To understand the practical importance of big-O considerations in your programs, consider two different ways to accept all items from an input iterable and accumulate them into a list in reverse order:

```
def slow(it):
    result = []
    for item in it:
        result.insert(0, item)
    return result
```

```
def fast(it):
    result = []
    for item in it:
        result.append(item)
    result.reverse()
    return result
```

We could express each of these functions more concisely, but the key difference is best appreciated by presenting the functions in these elementary terms. The function `slow` builds the result list by inserting each input item before all previously received ones. The function `fast` appends each input item after all previously received ones, then reverses the result list at the end. Intuitively, one might think that the final reversing represents extra work, and therefore `slow` should be faster than `fast`. But that's not the way things work out.

Each call to `result.append` takes roughly the same amount of time, independent of how many items are already in the list `result`, since there is (nearly) always a free slot for an extra item at the end of the list (in pedantic terms, `append` is *amortized* `O(1)`, but we don't cover amortization in this book). The `for` loop in the function `fast` executes `N` times to receive `N` items. Since each iteration of the loop takes a constant time, overall loop time is `O(N)`. `result.reverse` also takes time `O(N)`, as it is directly proportional to the total number of items. Thus, the total running time of `fast` is `O(N)`. (If you don't understand why a sum of two quantities, each `O(N)`, is also `O(N)`, consider that the sum of any two linear functions of `N` is also a linear function of `N`—and "being `O(N)`" has exactly the same meaning as "consuming an amount of time that is a linear function of `N`.")

On the other hand, each call to `result.insert` makes space at slot `0` for the new item to insert, moving all items that are already in list `result` forward one slot. This takes time proportional to the number of items already in the list. The overall amount of time to receive `N` items is therefore proportional to `1+2+3+...N-1`, a sum whose value is $O(N^2)$. Therefore, total running time of `slow` is $O(N^2)$.

It's almost always worth replacing an `O(N²)` solution with an `O(N)` one, unless you can somehow assign rigorous small limits to input size `N`. If `N` can grow without very strict bounds, the `O(N²)` solution turns out to be disastrously slower than the `O(N)` one for large values of `N`, no matter what the proportionality constants in each case may be (and no matter what profiling tells you). Unless you have other `O(N²)` or even worse bottlenecks elsewhere that you can't eliminate, a part of the program that is `O(N²)` turns into the program's bottleneck, dominating runtime for large values of `N`. Do yourself a favor and watch out for the big O: all other performance issues, in comparison, are usually almost insignificant.

Incidentally, you can make the function `fast` even faster by expressing it in more idiomatic Python. Just replace the first two lines with the following single statement:

```
result = list(it)
```

This change does not affect `fast`'s big-O character (`fast` is still `O(N)` after the change), but does speed things up by a large constant factor.

Simple is Better than Complex, and Usually Faster!

More often than not, in Python, the simplest, clearest, most direct and idiomatic way to express something is also the fastest.

Choosing algorithms with good big-O is roughly the same task in Python as in any other language. You just need a few hints about the big-O performance of Python's elementary building blocks, and we provide them in the following sections.

List operations

Python lists are internally implemented as *vectors* (also known as *dynamic arrays*), not as "linked lists." This implementation choice determines just

about all performance characteristics of Python lists, in big-O terms.

Chaining two lists `L1` and `L2`, of length `N1` and `N2` (i.e., `L1+L2`) is `O(N1+N2)`. Multiplying a list `L` of length `N` by integer `M` (i.e., `L*M`) is `O(N*M)`. Accessing or rebinding any list item is `O(1)`. `len()` on a list is also `O(1)`. Accessing any slice of length `M` is `O(M)`. Rebinding a slice of length `M` with one of identical length is also `O(M)`. Rebinding a slice of length `M1` with one of different length `M2` is `O(M1+M2+N1)`, where `N1` is the number of items *after* the slice in the target list (in other words, such length-changing slice rebindings are relatively cheap when they occur at the *end* of a list, more costly when they occur at the *beginning* or around the middle of a long list). If you need first-in, first-out (FIFO) operations, a list is probably not the fastest data structure for the purpose: instead, try the type `collections.deque`, covered in "deque".

Most list methods, as shown in Table 3-3, are equivalent to slice rebindings and have equivalent big-O performance. The methods `count`, `index`, `remove`, and `reverse`, and the operator `in`, are `O(N)`. The method `sort` is generally `O(N*log(N))`, but `sort` is highly optimized[6] to be `O(N)` in some important special cases, such as when the list is already sorted or reverse-sorted except for a few items. `range(a,b,c)` is `O(1)`, but looping on all items of the result is `O((b-a)//c)`.

String operations

Most methods on a string of length `N` (be it bytes or Unicode) are `O(N)`. `len(astring)` is `O(1)`. The fastest way to produce a copy of a string with transliterations and/or removal of specified characters is the string's method `translate`. The single most practically important big-O consideration involving strings is covered in "Building up a string from pieces".

Dictionary operations

Python `dicts` are implemented with hash tables. This implementation choice determines all performance characteristics of Python dictionaries, in big-O terms.

Accessing, rebinding, adding, or removing a dictionary item is `O(1)`, as are the methods `get`, `setdefault`, and `popitem`, and operator `in`. *d*`1.update(`*d*`2)` is `O(len(`*d*`2))`. `len(`*adict*`)` is `O(1)`. The methods `keys`, `items`, and `values` are `O(1)`, but looping on all items of the iterators those methods return is `O(N)`, as is looping directly on a `dict`.

When the keys in a dictionary are instances of classes that define `__hash__` and equality comparison methods, dictionary performance is of course affected by those methods. The performance indications presented in this section hold when hashing and equality comparison are `O(1)`.

Set operations

Python sets, like `dicts`, are implemented with hash tables. All performance characteristics of sets are, in big-O terms, the same as for dictionaries.

Adding or removing a set item is `O(1)`, as is operator `in`. `len(`*aset*`)` is `O(1)`. Looping on a set is `O(N)`. When the items in a set are instances of classes that define `__hash__` and equality comparison methods, set performance is of course affected by those methods. The performance hints presented in this section hold when hashing and equality comparison are `O(1)`.

Summary of big-O times for operations on Python built-in types

Let *L* be any list, *T* any string (str or bytes), *D* any dict, *S* any set, with (say) numbers as items (just for the purpose of ensuring `O(1)` hashing and comparison), and *x* any number (ditto):

`O(1)`

> `len(`*L*`), len(T), len(`*D*`), len(S),` *L*`[i],` *T*`[i],` *D*`[i], del D[i], if` *x* `in D, if` *x* `in` *S*`,` *S*`.add(`*x*`),` *S*`.remove(`*x*`),` appends or removals to/from the very right end of *L*

`O(N)`

Loops on *L*, *T*, *D*, *S*, general appends or removals to/from *L* (except at the very right end), all methods on *T*, `if` *x* `in` *L*, `if` *x* `in` *T*, most methods on *L*, all shallow copies

`O(N log N)`

L`.sort()`, mostly (but `O(N)` if *L* is already nearly sorted or reverse-sorted)

Profiling

Most programs have hot spots (i.e., relatively small regions of source code that account for most of the time elapsed during a program run). Don't try to guess where your program's hot spots are: a programmer's intuition is notoriously unreliable in this field. Instead, use the Python standard library module `profile` to collect profile data over one or more runs of your program, with known inputs. Then use the module `pstats` to collate, interpret, and display that profile data.

To gain accuracy, you can calibrate the Python profiler for your machine (i.e., determine what overhead profiling incurs on your machine). The `profile` module can then subtract this overhead from the times it measures, making profile data you collect closer to reality. The standard library module `cProfile` has similar functionality to `profile`; `cProfile` is preferable, since it's faster, which imposes less overhead.

There are also many third-party profiling tools worth considering, such as pyinstrument and Eliot; an excellent article by Itamar Turner-Trauring explains the basics and advantages of each of these tools.

The profile module

The `profile` module supplies one often-used function:

run

`run(`*code*`,`*filename*`=None)`
code is a string that is usable with `exec`, normally a call to the main function of the program you're profiling. *filename* is the path of a file that `run` creates or rewrites with profile data. Usually, you call `run` a few times, specifying different filenames, and different arguments to your program's

main function, in order to exercise various program parts in proportion to what you expect to be their use "in real life." Then, you use module `pstats` to display collated results across the various runs.

You may call `run` without a *`filename`* to get a summary report, similar to the one the `pstats` module could give you, on standard output. However, this approach gives you no control over the output format, nor any way to consolidate several runs into one report. In practice, you should rarely use this feature: it's best to collect profile data into files.

The `profile` module also supplies the class `Profile` (mentioned in the next section). By instantiating `Profile` directly, you can access advanced functionality, such as the ability to run a command in specified local and global dictionaries. We do not cover such advanced functionality of the class `profile.Profile` further in this book.

Calibration

To calibrate `profile` for your machine, you need to use the class `Profile`, which `profile` supplies and internally uses in the function `run`. An instance *`p`* of `Profile` supplies one method you use for calibration:

calibrate

`p``.calibrate(`*`N`*`)`

Loops *`N`* times, then returns a number that is the profiling overhead per call on your machine. *`N`* must be large if your machine is fast. Call *`p`*`.calibrate(10000)` a few times and check that the various numbers it returns are close to each other, then pick the smallest one of them. If the numbers vary a lot, try again with larger values of *`N`*.

The calibration procedure can be time-consuming. However, you need to perform it only once, repeating it only when you make changes that could alter your machine's characteristics, such as applying patches to your operating system, adding memory, or changing Python version. Once you know your machine's overhead, you can tell `profile` about it each time you import it, right before using `profile.run`. The simplest way to do this is as follows:

```
import profile
profile.Profile.bias = ...the overhead you
measured...
profile.run('main()', 'somefile')
```

The pstats module

The `pstats` module supplies a single class, `Stats`, to analyze, consolidate, and report on the profile data contained in one or more files written by the function `profile.run`:

Stats	`class Stats(`*`filename, *filenames`*`)` Instantiates `Stats` with one or more filenames of files of profile data written by function `profile.run`.

An instance *`s`* of the class `Stats` provides methods to add profile data and sort and output results. Each method returns *`s`*, so you can chain several calls in the same expression. *`s`*'s main methods are described in Table 16-2.

add	*`s`*`.add(`*`filename`*`)` Adds another file of profile data to the set that *`s`* is holding for analysis.
print_callees, print_callers	*`s`*`.print_callees(*`*`restrictions`*`)` Outputs the list of functions in *`s`*'s profile data, sorted according to the latest call to *`s`*`.sort_stats` and subject to given restrictions, if any. You can call each printing method with zero or more *`restrictions`*, to be applied one after the other, in order, to reduce the number of output lines. A restriction that is an `int` *`n`* limits the output to the first *`n`* lines. A restriction that is a `float` *`f`* between `0.0` and `1.0` limits the output to a fraction *`f`* of the lines. A restriction that is a string is compiled as a regular expression pattern (covered in "Regular Expressions and the re Module"); only lines that satisfy a `search` method call on the regular expression are output. Restrictions are cumulative. For example, *`s`*`.print_callees(10,0.5)` outputs the first 5 lines (half of 10). Restrictions apply only after the summary and header lines: the summary and header are output unconditionally. Each function *`f`* that is output is accompanied by the list of *`f`*'s callers (the functions that called *`f`*) or *`f`*'s callees (the functions that *`f`* called) according to the name of the method.
print_stats	*`s`*`.print_stats(*`*`restrictions`*`)` Outputs statistics about *`s`*'s profile data, sorted according to the latest call to *`s`*`.sort_stats` and subject to given restrictions, if any, as covered in **`print_callees, print_callers`**, above. After a few summary lines (date and time on which profile data was collected, number of function calls, and sort criteria used), the output—absent restrictions—is one line per function, with six fields per line, labeled in a header line. For each function *`f`*, `print_stats` outputs six fields: ▪ Total number of calls to *`f`*

- Total time spent in *f*, exclusive of other functions that *f* called
- Total time per call to *f* (i.e., field 2 divided by field 1)
- Cumulative time spent in *f*, and all functions directly or indirectly called from *f*
- Cumulative time per call to *f* (i.e., field 4 divided by field 1)
- The name of function *f*

sort_stats	*s*.sort_stats(*key*, **keys*) Gives one or more keys on which to sort future output, in priority order. Each key is a string. The sort is descending for keys that indicate times or numbers, and alphabetical for key 'nfl'. The most frequently used keys when calling sort_stats are: 'calls' Number of calls to the function (like field 1 covered in **print_stats**, above) 'cumulative' Cumulative time spent in the function and all functions it called (like field 4 covered in **print_stats**, above) 'nfl' Name of the function, its module, and the line number of the function in its file (like field 6 covered in **print_stats**, above) 'time' Total time spent in the function itself, exclusive of functions it called (like field 2 covered in **print_stats**, above)
strip_dirs	*s*.strip_dirs() Alters *s* by stripping directory names from all module names to make future output more compact. *s* is unsorted after *s*.strip_dirs(), and therefore you normally call *s*.sort_stats right after calling *s*.strip_dirs.

Small-Scale Optimization

Fine-tuning of program operations is rarely important. Tuning may make a small but meaningful difference in some particularly hot spot, but it is hardly ever a decisive factor. And yet, fine-tuning—in the pursuit of mostly irrelevant microefficiencies—is where a programmer's instincts are likely

to lead. It is in good part because of this that most optimization is premature and best avoided. The most that can be said in favor of fine-tuning is that, if one idiom is *always* speedier than another when the difference is measurable, then it's worth your while to get into the habit of always using the speedier way.[7]

Most often, in Python, if you do what comes naturally, choosing simplicity and elegance, you end up with code that has good performance as well as clarity and maintainability. In other words, "let Python do the work": when Python provides a simple, direct way to do a task, chances are that it's also the fastest way to perform that task. In a few cases, an approach that may not be intuitively preferable still offers performance advantages, as discussed in the rest of this section.

The simplest optimization is to run your Python programs using **`python -O`** or **`-OO`**. **`-OO`** makes little difference to performance compared to **`-O`**, but may save memory, as it removes docstrings from the bytecode, and memory is sometimes (indirectly) a performance bottleneck. The optimizer is not powerful in current releases of Python, but it may gain you performance advantages on the order of 5 percent, sometimes as large as 10 percent (potentially larger if you make use of `assert` statements and `if __debug__:` guards, as suggested in "The assert Statement"). The best aspect of **`-O`** is that it costs nothing—as long as your optimization isn't premature, of course (don't bother using **`-O`** on a program you're still developing).

The timeit module

The standard library module `timeit` is handy for measuring the precise performance of specific snippets of code. You can import `timeit` to use `timeit`'s functionality in your programs, but the simplest and most normal use is from the command line:

```
python -m timeit -s'setup statement(s)' 'statement(s) to be
timed'
```

The “setup statement” is executed only once, to set things up; the “statements to be timed” are executed repeatedly, to accurately measure the average time they take.

For example, say you’re wondering about the performance of `x=x+1` versus `x+=1`, where *x* is an `int`. At a command prompt, you can easily try:

```
$ python -m timeit -s 'x=0' 'x=x+1'
1000000 loops, best of 3: 0.0416 usec per loop
$ python -m timeit -s 'x=0' 'x+=1'
1000000 loops, best of 3: 0.0406 usec per loop
```

and find out that performance is, for all intents and purposes, the same in both cases (a difference of 2.5%, like this one, is best considered “noise”).

Memoizing

Memoizing is the technique of saving values returned from a function that is called repeatedly with the same argument values. When the function is called with arguments that have not been seen before, a memoizing function computes the results, and then saves the arguments used to call it and the corresponding result in a cache. When the function is called again later with the same arguments, the function just looks up the computed value in the cache instead of rerunning the function calculation logic. In this way, the calculation is performed just once for any particular argument or arguments.

Here is an example of a function for calculating the sin of a value given in degrees:

```
import math
def sin_degrees(x):
    return math.sin(math.radians(x))
```

If we determined that `sin_degrees` was a bottleneck, and was being repeatedly called with the same values for `x`, we could add a memoizing cache:

```
_cached_values = {}
def sin_degrees(x):
    if x not in _cached_values:
        _cached_values[x] = math.sin(math.radians(x))
    return _cached_values[x]
```

For functions that take multiple arguments, the tuple of argument values would be used for the cache key. (We define `_cached_values` outside the function, so that it's not reset each time we call the function.)

Caching is a classic approach in gaining performance at the expense of using memory (the *time-memory tradeoff*). The cache in the above example is unbounded, so, as `sin_degrees` is called with many different values of x, the cache will continue to grow, consuming more and more program memory. Caches are often configured with an *eviction policy*, which determines when values can be removed from the cache. Removing the oldest cached value is a common eviction policy. Since Python keeps dict entries in insertion order, the "oldest" key will be the first one found if we iterate over the dict:

```
_cached_values = {}
_maxsize = 64
def sin_degrees(x):
    if x not in _cached_values:
        _cached_values[x] = math.sin(math.radians(x))
        # remove oldest cache entry if exceed maxsize limit
        if len(_cached_values) > _maxsize:
            oldest_key = next(iter(_cached_values))
            del _cached_values[oldest_key]
    return _cached_values[x]
```

You can see that this starts to complicate the code, with the original logic for computing the sin of a value given in degrees hidden inside all the caching logic. The Python stdlib module `functools` includes caching decorators `lru_cache`, ||3.9++|| `cache`, and ||3.8++|| `cached_property` to perform memoization cleanly:

```
import functools
@functools.lru_cache(maxsize=64)
def sin_degrees(x):
    return math.sin(math.radians(x))
```

The signatures for these decorators are described in detail in "The functools module".

Building up a string from pieces

The single Python "anti-idiom" that's likeliest to damage your program's performance, to the point that you should *never* use it, is to build up a large string from pieces by looping on string concatenation statements such as `big_string+=piece`. Python strings are immutable, so each such concatenation means that Python must free the `M` bytes previously allocated for `big_string`, and allocate and fill `M+K` bytes for the new version. Doing this repeatedly in a loop, you end up with roughly $O(N^2)$ performance, where `N` is the total number of characters. More often than not, $O(N^2)$ performance where $O(N)$ is available is a disaster[8]—even though. On some platforms, things may be even bleaker due to memory fragmentation effects caused by freeing many areas of progressively larger sizes.

To achieve $O(N)$ performance, accumulate intermediate pieces in a list, rather than build up the string piece by piece. Lists, unlike strings, are mutable, so appending to a list is $O(1)$ (amortized). Change each occurrence of `big_string+=piece` into `temp_list.append(piece)`. Then, when you're done accumulating, use the following code to build your desired string result in $O(N)$ time:

```
big_string = ''.join(temp_list)
```

Using a list comprehension, generator expression, or other direct means (such as a call to `map`, or use of the standard library module `itertools`) to build `temp_list` may often offer further (substantial, but not big-O) optimization over repeated calls to `temp_list.append`. Other $O(N)$ ways to build up big strings, which a few Python programmers find more readable, are to concatenate the pieces to an instance of `array.array('u')` with the array's `extend` method, use a

`bytearray`, or write the pieces to an instance of `io.TextIO` or `io.BytesIO`.

In the special case where you want to output the resulting string, you may gain a further small slice of performance by using `writelines` on *`temp_list`* (never building *`big_string`* in memory). When feasible (i.e., when you have the output file object open and available in the loop, and the file is buffered), it's just as effective to perform a `write` call for each *`piece`*, without any accumulation.

Although not nearly as crucial as += on a big string in a loop, another case where removing string concatenation may give a slight performance improvement is when you're concatenating several values in an expression:

```
oneway = str(x) + ' eggs and ' + str(y) + ' slices of ' + k + '
ham'
another = '{} eggs and {} slices of {} ham'.format(x, y, k)
yetanother = f'{x} eggs and {y} slices of {k} ham'
```

Using the `format` method, or f-strings, to format strings, is often a good performance choice, as well as being more idiomatic and thereby clearer than concatenation approaches. On a sample run of the above example, the `format` approach is more than twice as fast as the (perhaps more intuitive) concatenation, and the f-string approach more than twice as fast as `format`.

Searching and sorting

The operator `in`, the most natural tool for searching, is `O(1)` when the righthand side operand is a `set` or `dict`, but `O(N)` when the righthand side operand is a string, list, or tuple. If you must perform many such checks on a container, you're **much** better off using a `set` or `dict`, rather than a list or tuple, as the container. Python `sets` and `dicts` are highly optimized for searching and fetching items by key. Building the `set` or `dict` from other containers, however, is `O(N)`, so, for this crucial optimization to be worthwhile, you must be able to hold on to the `set` or

`dict` over several searches, possibly altering it apace as the underlying sequence changes.

The method `sort` of Python lists is also a highly optimized and sophisticated tool. You can rely on `sort`'s performance. Most functions and methods in the stdlib that perform comparisons accept a `key=` argument to determine how, exactly, to compare items. You provide the `key` function, which computes a value for each element in the list. The list elements are sorted based on their corresponding values. For instance, you might write a key function for sorting objects based on an attribute *`attr`* as `lambda ob: ob.attr`, or sorting dicts by dict key `'attr'` as `lambda d: d['attr']`. (The `attrgetter` and `itemgetter` methods of the `operator` module are useful alternatives to these simple key functions, clearer and sharper than `lambda`, and offer performance gains as well.)

Older versions of Python used to use a `cmp` function, which would take list elements in pairs `(A,B)` and return a -1, 0, or 1 value depending if each pair should be considered `A<B`, `A==B`, or `A>B`. Sorting using a `cmp` function is very slow, as it may have to compare every element to every other element, potentially $O(N^2)$ performance. The `sort` function in current Python versions no longer accepts a `cmp` function argument. If you are migrating ancient code, and only have a function suitable as a `cmp` argument, you can use `functools.cmp_to_key`, covered in Table 7-4, to build from it a function suitable as the key argument, and pass the new function thus built as the `key=` argument, instead of passing the original function as the `cmp=` argument.

However, several functions in the module `heapq`, covered in "The heapq Module", do not accept a `key=` argument. In such cases, you can use the *decorate-sort-undecorate (DSU)* idiom, covered in "The Decorate-Sort-Undecorate Idiom". (Heaps are well worth keeping in mind, since in some cases they can save you from having to perform sorting on all of your data.)

Avoid exec and from ... import *

Code in a function runs faster than code at the top level in a module, because access to a function's local variables is very fast. If a function contains an `exec` without explicit dictionaries, however, the function slows down. The presence of such an `exec` forces the Python compiler to avoid the modest but important optimization it normally performs regarding access to local variables, since the `exec` might alter the function's namespace. A `from` statement of the form:

```
from MyModule import *
```

wastes performance, too, since it also can alter a function's namespace unpredictably, and therefore inhibits Python's local-variable optimizations.

`exec` itself is also quite slow, and even more so if you apply it to a string of source code rather than to a code object. By far the best approach—for performance, for correctness, and for clarity—is to avoid `exec` altogether. It's most often possible to find better (faster, more robust, and clearer) solutions. If you *must* use `exec`, *always* use it with explicit `dicts`. If you need to `exec` a dynamically obtained string more than once, `compile` the string just once and then repeatedly `exec` the resulting code object. But avoiding `exec` altogether is *far* better, if at all feasible.

`eval` works on expressions, not on statements; therefore, while still slow, it avoids some of the worst performance impacts of `exec`. With `eval`, too, you're best advised to use explicit `dicts`. If you need several evaluations of the same dynamically obtained string, `compile` the string once and then repeatedly `eval` the resulting code object. Avoiding `eval` altogether is even better.

See "Dynamic Execution and exec" for more details and advice about `exec`, `eval`, and `compile`.

Optimizing loops

Most of your program's bottlenecks will be in loops, particularly nested loops, because loop bodies execute repeatedly. Python does not implicitly perform any *code hoisting*: if you have any code inside a loop that you

could execute just once by hoisting it out of the loop, and the loop is a bottleneck, hoist the code out yourself. Sometimes the presence of code to hoist may not be immediately obvious:

```
def slower(anobject, ahugenumber):
    for i in range(ahugenumber):
        anobject.amethod(i)

def faster(anobject, ahugenumber):
    themethod = anobject.amethod
    for i in range(ahugenumber):
        themethod(i)
```

In this case, the code that `faster` hoists out of the loop is the attribute lookup `anobject.amethod`. `slower` repeats the lookup every time, while `faster` performs it just once. The two functions are not 100 percent equivalent: it is (barely) conceivable that executing `amethod` might cause such changes on `anobject` that the next lookup for the same named attribute fetches a different method object. This is part of why Python doesn't perform such optimizations itself. In practice, such subtle, obscure, and tricky cases happen very rarely; you're almost invariably safe in performing such optimizations yourself, to squeeze the last drop of performance out of some bottleneck.

Python is faster with local variables than with global ones. If a loop repeatedly accesses a global whose value does not change between iterations, "cache" the value in a local variable, and access the local instead. This also applies to built-ins:

```
def slightly_slower(asequence, adict):
    for x in asequence:
        adict[x] = hex(x)

def slightly_faster(asequence, adict):
    myhex = hex
    for x in asequence:
        adict[x] = myhex(x)
```

Here, the speedup is very modest, on the order of 5 percent or so.

Do not cache `None`, `True`, or `False`. Those global constants are keywords, so no further optimization is needed.

List comprehensions and generator expressions can be faster than loops, and, sometimes, so can `map` and `filter`. For optimization purposes, try changing loops into list comprehensions, generator expressions, or perhaps `map` and `filter` calls, where feasible. The performance advantage of `map` and `filter` is nullified, and worse, if you have to use a `lambda` or an extra level of function call. Only when you pass to `map` or `filter` a built-in function, or a function you'd have to call anyway even from an explicit loop, list comprehension, or generator expression, do you stand to gain some tiny speed-up.

The loops that you can replace most naturally with list comprehensions, or `map` and `filter` calls, are ones that build up a list by repeatedly calling `append` on the list. The following example shows this optimization in a micro-performance benchmark script (of course, we could use the module `timeit` instead of coding our own time measurement, but the example is meant to show how to do the latter):

```
import time, operator
def slow(asequence):
    result = []
    for x in asequence:
        result.append(-x)
    return result
def middling(asequence):
    return list(map(operator.neg, asequence))
def fast(asequence):
    return [-x for x in asequence]
biggie = range(500*1000)
n_times= [None]*50
def timit(afunc):
    lobi = biggie
    start = time.perf_counter()
    for x in n_times:
        afunc(lobi)
    stend = time.perf_counter()
    return f'{afunc.__name__:<10}: {stend-start:.2f}'
for afunc in slow, middling, fast, fast, middling, slow:
    print(timit(afunc))
```

Running this example on an old computer shows that `fast` takes about 1.88 seconds, `middling` 1.94 seconds, and `slow` 2.55 seconds. In other words, on that machine, `slow` (the loop of `append` method calls) is about 30 percent slower than `middling` (the single `map` call), and `middling`, in turn, is about 3 percent slower than `fast` (the list comprehension).

The list comprehension is the most direct way to express the task being micro-benchmarked in this example, so, not surprisingly, it's also fastest—significantly faster than the loop of `append` method calls.

Optimizing I/O

If your program does substantial amounts of I/O, it's quite likely that performance bottlenecks are due to I/O, rather than to computation. Such programs are said to be *I/O-bound*, rather than *CPU-bound*. Your operating system tries to optimize I/O performance, but you can help it in a couple of ways. One such way is to perform your I/O in chunks of a size that is optimal for performance, rather than simply convenient for your program's operations. Another way is to use threading. Often the very best way is to "go asynchronous" as described in "Multitasking".

From the point of view of a program's convenience and simplicity, the ideal amount of data to read or write at a time is often small (one character or one line) or very large (an entire file at a time). That's often okay: Python and your operating system work behind the scenes to let your program use convenient logical chunks for I/O, while arranging for physical I/O operations to use chunk sizes more attuned to performance. Reading and writing a whole file at a time is quite likely to be okay for performance as long as the file is not *very* large. Specifically, file-at-a-time I/O is fine as long as the file's data fits very comfortably in physical RAM, leaving ample memory available for your program and operating system to perform whatever other tasks they're doing at the same time. The hard problems of I/O-bound performance tend to come with huge files.

If performance is an issue, *never* use a file's `readline` method, which is limited in the amount of chunking and buffering it can perform. (Using `writelines`, on the other hand, gives no performance problem when that

method is convenient for your program.) When reading a text file, loop directly on the file object to get one line at a time with best performance. If the file isn't too huge, and so can conveniently fit in memory, time two versions of your program—one looping directly on the file object, the other reading the whole file into memory. Either may prove faster by a little.

For binary files, particularly large binary files whose contents you need just a part of on each given run of your program, the module `mmap` (covered in "The mmap Module") can sometimes give you both good performance and program simplicity.

Making an I/O-bound program multithreaded sometimes affords substantial performance gains, if you can arrange your architecture accordingly. Start a few worker threads devoted to I/O, have the computational threads request I/O operations from the I/O threads via `Queue` instances, and post the request for each input operation as soon as you know you'll eventually need that data. Performance increases only if there are other tasks your computational threads can perform while I/O threads are blocked waiting for data. You get better performance this way only if you can manage to overlap computation and waiting for data by having different threads do the computing and the waiting. (See "Threads in Python" for detailed coverage of Python threading and a suggested architecture.)

On the other hand, a possibly even faster and more scalable approach is to eschew threads in favor of asynchronous (event-driven) architectures, as discussed in "Multitasking".

---

1 This issue is related to "technical debt" and other aspects covered in "Good enough is good enough", a tech talk by one of this book's authors (that author's favorite tech talk out of the many he delivered!-), excellently-well summarized and discussed by Martin Michlmayr here.

2 Terminology in this area is confused and confusing: terms like dummies, fakes, spies, mocks, stubs, and "test doubles" are used by different authors with different distinctions. For an authoritative approach (though not the exact one we use), see Martin Fowler's essay.

3 That's partly because the structure of the system tends to mirror the structure of the organization, per Conway's Law.

4 However, be sure you know exactly what you're using doctest for in any given case: to quote Peter Norvig, writing precisely on this subject: "Know what you're aiming for; if you aim at

two targets at once you usually miss them both."

5 an application of Pareto's Principle.

6 using invented-for-Python adaptive sorting algorithm Timsort

7 a once-slower idiom may be optimized in some future version of Python, so it's worth redoing **timeit** measurements, just to check for this, when you upgrade to newer versions of Python.

8 current Python implementations bend over backward to help reducing the performance hit of this specific, terrible but common, anti-pattern, but they can't catch every occurrence, so, don't count on that!

Chapter 15. Networking Basics

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 17th 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*.

Connection-oriented protocols are like making a telephone call. You request a connection to a particular *network endpoint* (equivalent to dialing somebody's phone number), and your party either answers or doesn't. If they do, you can talk to them and hear them talking back (simultaneously, if necessary), and you know that nothing is getting lost. At the end of the conversation you both say goodbye and hang up, so it's obvious something has gone wrong if that finale doesn't occur (for example, if you just suddenly stop hearing the other party). TCP is the main connection-oriented transport protocol of the Internet, used by web browsers, secure shells, email, and many other applications.

Connectionless or *datagram* protocols are more like communicating by sending postcards. Mostly, the messages get through, but if anything goes wrong you have to be prepared to cope with the consequences—the protocol doesn't notify you whether your messages have been received, and messages can arrive out of order. To exchange short messages and get answers, datagram protocols have less overhead than connection-oriented ones, as long as the overall service can cope with occasional disruptions. A Domain Name Service (DNS) server may fail to respond, for example: most DNS communication was until recently connectionless. UDP is the main connectionless transport protocol for Internet communications.

Nowadays, security is increasingly important: understanding the underlying basis of secure communications helps you ensure that your communications are as secure as they need to be. If this summary dissuades you from trying to implement such technology yourself without a thorough understanding of the issues and risks, it will have served a worthwhile purpose.

All communications across network interfaces exchange strings of bytes. To communicate text or indeed most other information the sender must encode it as bytes, which the receiver must decode. We limit our discussion in this chapter to the case of a single sender and a single receiver.

The Berkeley Socket Interface

Most networking nowadays uses *sockets.* Sockets give access to pipelines between independent endpoints, using a *transport layer* to move information between those endpoints. The socket concept is general enough that the endpoints can be on the same computer, or on different computers networked together locally or via a wide-area network.

The most typical transport layers are UDP (the User Datagram Protocol, for connectionless networking) and TCP (the Transmission Control Protocol, for connection-oriented networking) carried over a common IP (Internet Protocol) network layer. These ubiquitous protocols, along with the many application protocols that run over them, are collectively known as *TCP/IP.* A good introduction appears in the (dated but still perfectly valid)) Gordon McMillan's *Socket Programming How-To* online overview.

The two most common socket *families* are *Internet sockets* based on TCP/IP communications (available in two flavors to accommodate the modern IPv6 and the more traditional IPv4), and *Unix sockets*, though other families are also available. Internet sockets allow communication between any two computers that can exchange IP datagrams; Unix sockets can only communicate between processes on the same Unix machine.

To support many concurrent Internet sockets, the TCP/IP protocol stack uses endpoints identified by an IP address, a *port number,* and a protocol. The port numbers allow the protocol-handling software to distinguish

between different endpoints at the same IP address using the same protocol, in the same way that PABX telephone systems allow direct dialing to individual members of a large organization. A connected socket is also associated with a *remote endpoint*, the *counter-party* socket to which it is connected and with which it can communicate.

Most Unix sockets have names in the Unix filesystem. On Linux platforms, sockets whose names begin with a zero byte live in a name pool maintained by the kernel. These are useful for communicating with a chroot-jail process, for example, where no filesystem is shared between two processes.

Both Internet and Unix sockets support connectionless and connection-oriented networking, so if you write your programs carefully they can work over either socket family. It is beyond the scope of this book to discuss other socket families, though we should mention that *raw sockets*, a subtype of the Internet socket family, let you send and receive link-layer packets (for example, Ethernet packets) directly. This is useful for some experimental applications.

After creating an Internet socket, you can associate *(bind)* a specific port number with the socket (as long as that port number is not in use by some other socket). This is the strategy many servers use, offering service on so-called *well-known port numbers* defined by Internet standards as being in the range 1-1,023. On Unix systems, root privileges are required to gain access to these ports. A typical client is unconcerned with the port number it uses, and so it typically requests an *ephemeral port*, assigned by the protocol driver and guaranteed to be unique on that host. There is no need to bind client ports.

Imagine two processes on the same computer, both acting as clients to the same remote server. The full association for their sockets has five components (`local_IP_address`, `local_port_number`, `protocol`, `remote_IP_address`, `remote_port_number`). When packets arrive at the server, the destination their source IP address, destination port number, protocol, and source IP address are the same for both clients. The guarantee of uniqueness for ephemeral port numbers makes it possible for the server to distinguish between traffic from the two

clients. This is how TCP/IP handles multiple conversations between the same two IP addresses.

Socket Addresses

The different types of sockets use different address formats.

Unix socket addresses are strings naming a node in the filesystem (on Linux platforms, strings starting with `b'\0'` correspond to names in a kernel table).

IPv4 socket addresses are pairs *`(address, port)`*. The first item is an IPv4 address, the second a port number in the range 1-65,535.

IPv6 socket addresses are four-item (*`address, port, flowinfo, scopeid`*) tuples. When providing an address as an argument, the *`flowinfo`* and *`scopeid`* items can generally be omitted without problems as long as the address scope is unimportant.

Client-Server Computing

The pattern we discuss hereafter is usually referred to as *client-server* networking, where a *server* listens for traffic on a specific endpoint from *clients* requiring the service. We do not cover *peer-to-peer* networking, which, lacking any central server, has to include the ability for peers to discover each other.

Most, though by no means all, network communication is performed using client/server techniques. The server listens for incoming traffic at a predetermined or advertised network endpoint. In the absence of such input, it does nothing, simply sitting there waiting for input from clients. Communication is somewhat different between connectionless and connection-oriented endpoints.

In connectionless networking such as UDP, requests arrive at a server randomly and are dealt with immediately: a response is dispatched to the requester without delay. Each message is handled on its own, usually without reference to any communications that may previously have

occurred between the two parties. Connectionless networking is thus well-suited to short-term, stateless interactions such as those required by DNS or network booting.

In connection-oriented networking, the client engages in an initial exchange with the server that effectively establishes a connection across a network pipeline between two processes (sometimes referred to as a *virtual circuit*), across which the processes can communicate until both have indicated their willingness to end the connection. Serving under these conditions requires the use of parallelism using a concurrency mechanism (such as threads, processes, and asynchronous programming – see Chapter "Multitasking") to handle each incoming connection asynchronously or simultaneously. Without such parallelism, the server would be unable to handle new incoming connections before earlier ones had terminated, since calls to socket methods normally *block* (meaning they pause the thread calling them until they terminate or time-out). Connections are the best way to handle lengthy interactions such as mail exchanges, command-line shell interactions, or the transmission of web content, and offer automatic error detection and correction when TCP is used.

Connectionless client and server structures

The broad logic flow of a connectionless server is as follows:

1. Create a socket of type `socket.SOCK_DGRAM` by calling `socket.socket`.

2. Associate the socket with the service endpoint by calling the socket's `bind` method.

3. Repeat the following steps *ad infinitum*:

    a. Request an incoming datagram from a client by calling the socket's `recvfrom` method; this call blocks until a datagram is received.

    b. Compute the result.

c. Send the result back to the client by calling the socket's `sendto` method.

The server spends most of its time in step 3a, awaiting input from clients.

A connectionless client's interaction with the server proceeds as follows:

1. Create a socket of type `socket.SOCK_DGRAM` by calling `socket.socket`.

2. Optionally, associate the socket with a specific endpoint by calling the socket's `bind` method.

3. Send a request to the server's endpoint by calling the socket's `sendto` method.

4. Await the server's reply by calling the socket's `recvfrom` method; this call blocks until the response is received. It is always necessary to apply a *timeout* to this call, to handle the case where a datagram goes missing, and either retry or abort the attempt: connectionless sockets do not guarantee delivery.

5. Use the result in the remainder of the client program's logic.

A single client program can perform several interactions with the same or multiple servers, depending on the services it needs to use. Many such interactions are hidden from the application programmer inside library code. A typical example is the resolution of a hostname to the appropriate network address, which commonly uses the `gethostbyname` library function (implemented in Python's `socket` module). Connectionless interactions normally involve sending a single packet to the server and receiving a single packet in response. The main exceptions involve *streaming* protocols, such as RTP[1], typically layered on top of UDP to minimize latency and delays: in streaming, many datagrams are sent and received.

Connection-oriented client and server structures

The broad flow of logic of a connection-oriented server is as follows:

1. Create a socket of type `socket.SOCK_STREAM` by calling `socket.socket`.

2. Associate the socket with the appropriate server endpoint by calling the socket's `bind` method.

3. Start the endpoint listening for connection requests by calling the socket's `listen` method.

4. Repeat the following steps *ad infinitum*:

    a. Await an incoming client connection by calling the socket's `accept` method; the server process blocks until an incoming connection request is received. When such a request arrives, a new socket object is created whose other endpoint is the client program.

    b. Create an asynchronous control thread to handle this specific connection, passing it the newly created socket; after which, the main thread continues by looping back to step 4a.

    c. In the new control thread, interact with the client using the new socket's `recv` and `send` methods, respectively, to read data from the client and send data to it. The `recv` method blocks until data is available from the client (or the client indicates it wishes to close the connection, in which case `recv` returns an empty result). The `send` method only blocks when the network software has so much data buffered that communication has to pause until the transport layer has emptied some of its buffer memory. When the server wishes to close the connection, it can do so by calling the socket's `close` method, optionally calling its `shutdown` method first.

The server spends most of its time in step 4a, awaiting connection requests from clients.

A connection-oriented client's overall logic is as follows:

1. Create a socket of type `socket.SOCK_STREAM` by calling `socket.socket`.

2. Optionally, associate the socket with a specific endpoint by calling the socket's `bind` method.

3. Establish a connection to the server by calling the socket's `connect` method.

4. Interact with the server using the socket's `recv` and `send` methods, respectively, to read data from the server and send data to it. The `recv` method blocks until data is available from the server (or the server indicates it wishes to close the connection, in which case the `recv` call returns an empty result). The `send` method only blocks when the network software has so much data buffered that communications have to pause until the transport layer has emptied some of its buffer memory. When the client wishes to close the connection, it can do so by calling the socket's `close` method, optionally calling its `shutdown` method first.

Connection-oriented interactions tend to be more complex than connectionless ones. Specifically, determining when to read and write data is more complicated, because inputs must be parsed to determine when a transmission from the other end of the socket is complete. The protocols used in connection-oriented networking have to accommodate this determination; sometimes this is done by indicating the data length as a part of the content, sometimes by more complex methods.

The socket Module

Python's `socket` module handles networking with the socket interface. There are minor differences between platforms, but the module hides most of them, making it relatively easy to write portable networking applications.

The module defines four exceptions: their base class `socket.error`, a (deprecated) alias for `exceptions.OSError`, and three exception subclasses as follows:

herror	`socket.herror` is raised for hostname-resolution errors—that is, when a name cannot be converted to a network address by the socket.gethostbyname function or no hostname can be found for a network address by the `socket.gethostbyaddr` function. The accompanying value is a two-element tuple *`(h_errno, string)`* where *`h_errno`* is the integer error number returned by the operating system and *`string`* is a description of the error.
gaierror	`socket.gaierror` is raised for addressing errors encountered in the `getaddrinfo` or `getnameinfo` functions.
timeout	`socket.timeout` is raised when an operation takes longer than the timeout limit (established by the module's `setdefaulttimeout` function, overridable on a per-socket basis).

The module also defines a large set of constants. The most important of these are the address families (`AF_*`) and the socket types (`SOCK_*`) listed next, members of `IntEnum` collections. The module defines many other constants, used to set socket options, but the documentation does not define them fully: to use them you must be familiar with documentation for the C sockets libraries and system calls.

AF_INET	Use to create sockets of the IPv4 address family.
AF_INET6	Use to create sockets of the IPv6 address family.
AF_UNIX	Use to create sockets of the Unix address family. This constant is only defined on platforms that make Unix sockets available.
AF_CAN	Use to create sockets for the Controller Area Network (CAN) address family, not covered in this book, but widely used in automation, automotive, and embedded device applications.
SOCK_STREAM	Use to create connection-oriented sockets, which provide full error detection and correction facilities.
SOCK_DGRAM	Use to create connectionless sockets, which provide best-effort message delivery without connection capabilities or error detection.

SOCK_RAW	Use to create sockets that give direct access to the link-layer drivers, typically used to implement lower-level network features outside the scope of this book.
SOCK_RDM	Use to create reliable connectionless message sockets used in the TIPC protocol, which is outside the scope of this book.
SOCK_SEQPACKET	Use to create reliable connection-oriented message sockets used in the TIPC protocol, outside the scope of this book.

The module defines a number of functions to create sockets, manipulate address information, and assist with representing data in a standard way. We do not cover them all in this book, as the socket module documentation is comprehensive; we deal with the ones that are essential in writing networked applications.

Miscellaneous socket module functions

The `socket` module contains many functions, but most of them are only useful in specific situations. When communication takes place between network endpoints, the computers at either end might have architectural differences and therefore represent the same data in different ways, and so there are functions to handle translation of a limited number of data types to and from a network-neutral form, for example. Here are a few of the more generally applicable functions:

getaddrinfo	`socket.getaddrinfo(`*`host, port, family=0, type=0, proto=0, flags=0`*`)` Takes a *`host`* and *`port`*, returns a list of five-item tuples of the form (*`family, type, proto, canonical_name, socket`*) that can be used to create a socket connection to a specific service. The *`canonical_name`* item is an empty string unless the *`socket.AI_CANONNAME`* bit is set in the *`flags`* argument. When you pass a hostname, rather than an IP address, the function returns a list of tuples, one for each IP address associated with the name.
getdefaulttimeout	`socket.getdefaulttimeout()` Returns the default timeout value in seconds for socket operations, or None if no value has been set. Some functions let you specify explicit timeouts.
getfqdn	`socket.getfqdn(`*`[host]`*`)`

	Returns the fully qualified domain name associated with a hostname or network address (by default, that of the computer on which you call it).
gethostbyaddr	`socket.gethostbyaddr(ip_address)` Takes a string containing an IPv4 or IPv6 address and returns a three-item tuple of the form (*`hostname, aliaslist, ipaddrlist`*). *`hostname`* is the canonical name for the IP, *`aliaslist`* a list of alternative names, and *`ipaddrlist`* a list of IPv4 and IPv6 addresses.
gethostbyname	`socket.gethostbyname(hostname)` Returns a string containing the IPv4 address associated with the given hostname. If called with an IP address, returns that address. This function does not support IPv6: use getaddrinfo for IPv6.
getnameinfo	`socket.getnameinfo(sock_addr, flags=0)` Takes a socket address and returns a (*`host, port`*) pair. Without *`flags, host`* is an IP address and *`port`* is an `int`.
setdefaulttimeout	`socket.setdefaulttimeout(timeout)` Sets sockets' timeout as a value in floating-point seconds. Newly created sockets operate in the mode determined by the *`timeout`* value, as discussed in the next section. Pass *`timeout`* as `None` to cancel the implicit use of timeouts on subsequently created sockets.

Socket Objects

The socket object is the primary means of network communication in Python. A new socket is also created when a `SOCK_STREAM` socket accepts a connection, each such socket being used to communicate with the relevant client.

Socket objects and `with` statements

Every socket object is a context manager, so you can use any socket object in a `with` statement's initial clause to ensure proper termination of the socket at exit from the `with` statement's body.

There are a number of ways you can create a socket, as detailed in the next section. The socket can operate in different modes, determined by its timeout value, established in one of three ways:

- By providing the timeout value on creation
- By calling the socket object's `settimeout` method
- According to the `socket` module's default timeout value as returned by the `socket.getdefaulttimeout` function

The timeout values to establish each mode are as follows:

`None`	Sets blocking mode. Each operation suspends the process (blocks) until the operation completes, unless the operating system raises an exception.
`0`	Sets nonblocking mode. Each operation raises an exception when it cannot be completed immediately, or when an error occurs. Use the `selectors` module, covered in "The selectors Module", to find out whether an operation can be completed immediately.
`>0.0`	Sets timeout mode. Each operation blocks until complete, or the timeout elapses (then, a `socket.timeout` exception is raised), or an error occurs.

Socket creation functions

Socket objects represent network endpoints. There are a number of different functions supplied by the `socket` module to create a socket:

create_connection `create_connection([`*`address`*`, [`*`timeout`*`, [`*`source_address`*`]]])`

Creates a socket connected to a TCP endpoint at an address (a `(`*`host`*`, `*`port`*`)` pair). *`host`* can either be a numeric network address or a DNS hostname; in the latter case, name resolution is attempted for both `AF_INET` and `AF_INET6`, and then a connection is attempted to each returned address in turn—a convenient way to create client programs using either IPv6 or IPv4 as appropriate.

The *`timeout`* argument, if given, specifies the connection timeout in seconds and thereby sets the socket's mode; when not present, the `socket.getdefaulttimeout` function is called to determine the value. The *`source_address`* argument, if given, must also be a pair (*`host`*`, `*`port`*) that the remote socket gets passed as the connecting endpoint. When *`host`* is '' or *`port`* is `0`, the default OS behavior is used.

socket	`socket(`*`family`*`=AF_INET, `*`type`*`=SOCK_STREAM, `*`proto`*`=0, `*`fileno`*`=None)` Creates and returns a socket of the appropriate address family and type (by default, a TCP socket on IPv4). The protocol number *`proto`* is only used with CAN sockets. When you pass the *`fileno`* argument, other arguments are ignored: the function returns the socket already associated with the given file descriptor. The socket does not get inherited by child processes.
socketpair	`socketpair([`*`family`*`[, `*`type`*`[, `*`proto`*`]]])` Returns a connected pair of sockets of the given address family, socket type, and (CAN sockets only) protocol. When *`family`* is not specified, the sockets are of family `AF_UNIX` on platforms where the family is available, and otherwise of family `AF_INET`. When *`type`* is not specified, it defaults to `SOCK_STREAM`.

A socket object *`s`* provides the following methods (out of which, those dealing with connections or requiring connected sockets work only for `SOCK_STREAM` sockets, while the others work with both `SOCK_STREAM` and `SOCK_DGRAM` sockets). In the following table, the exact set of flags available depends on your specific platform; the *`flags`* values available are documented on the appropriate Unix manual page for `recv(2)` or manual page for `send(2)`:

accept	`accept()` Blocks until a client establishes a connection to *`s`*, which must have been bound to an address (with a call to *`s`*.bind) and set to listening (with a call to *`s`*.listen). Returns a *new* socket object, which can be used to communicate with the other endpoint of the connection.
bind	`bind(`*`address`*`)` Binds *`s`* to a specific address. The form of the *`address`* argument depends on the socket's address family (see "Socket Addresses").
close	`close()` Marks the socket as closed. It does not necessarily close the connection immediately, depending on whether other references to the socket exist. If immediate closure is required, call the *`s`*`.shutdown` method first.

	The simplest way to ensure a socket is closed in a timely fashion is to use it in a `with` statement, since sockets are context managers.
connect	`connect(`*`address`*`)` Connects to a remote socket at *`address`*. The form of the *`address`* argument depends on the address family (see "Socket Addresses").
detach	`detach()` Puts the socket into closed mode, but allows the socket object to be reused for further connections.
dup	`dup()` Returns a duplicate of the socket, not inheritable by child processes.
fileno	`fileno()` Returns the socket's file descriptor.
get_inheritable	`get_inheritable()` Returns `True` when the socket is going to be inherited by child processes. Otherwise, returns `False`.
getpeername	`getpeername()` Returns the address of the remote endpoint to which this socket is connected.
getsockname	`getsockname()` Returns the address being used by this socket.
gettimeout	`gettimeout()` Returns the timeout associated with this socket.
listen	`listen([`*`backlog`*`])` Starts the socket listening for traffic on its associated endpoint. If given, the integer *`backlog`* argument determines how many unaccepted connections the operating system allows to queue up before starting to refuse connections.
makefile	`makefile(`*`mode`*`, `*`buffering=None`*`, *, encoding=None, `*`newline=None`*`)` Returns a file object allowing the socket to be used with file-like operations such as `read` and `write`. The mode can be `'r'` or `'w'`; `'b'` can be added for binary transmission. The socket must be in blocking

	mode; if a timeout value is set, unexpected results may be observed if a timeout occurs. The arguments are like for the built-in `open` function.
recv	`recv(`*`bufsiz, [flags]`*`)` Receive a maximum of *`bufsiz`* bytes of data on the socket. Returns the received data.
recvfrom	`recvfrom(`*`bufsiz, [flags]`*`)` Receive a maximum of *`bufsiz`* bytes of data from s. Returns a pair *`(bytes, address)`*: *`bytes`* is the received data, *`address`* the address of the counter-party socket that sent the data.
recvfrom_into	`recvfrom_into(`*`buffer, [nbytes, [flags]]`*`)` Receive a maximum of *`nbytes`* bytes of data from `s`, writing it into the given *`buffer`* object. Returns a pair *`(nbytes, address)`*: *`nbytes`* is the number of bytes received, *`address`* the address of the counter-party socket that sent the data.
recv_into	`recv_into(`*`buffer, [nbytes, [flags]]`*`)` Receive a maximum of *`nbytes`* bytes of data from s, writing it into the given *`buffer`* object. Returns the number of bytes received.
recvmsg	`recvmsg(`*`bufsiz, [ancbufsiz, [flags]]`*`)` Receive a maximum of *`bufsiz`* bytes of data on the socket and a maximum of *`ancbufsiz`* of ancillary ("out-of-band") data. Returns a four-item tuple *`(data, ancdata, msg_flags, address)`*, where *`bytes`* is the received data, *`ancdata`* is a list of three-item (*`cmsg_level, cmsg_type, cmsg_data`*) tuples representing the received ancillary data, *`msg_flags`* holds any flags received with the message, and *`address`* is the address of the counter-party socket that sent the data (if the socket is connected, this value is undefined, but the sender can be determined from the socket).
send	`send(`*`bytes, [flags]`*`)` Send the given data *`bytes`* over the socket, which must already be connected to a remote endpoint. Returns the number of bytes sent, which you should check: the call may not transmit all data, in which case transmission of the remainder will have to be separately requested.
sendall	`sendall(`*`bytes, [flags]`*`)` Send all the given data *`bytes`* over the socket, which must already be connected to a remote endpoint. The socket's timeout value applies to the transmission of all the data, even if multiple transmissions are needed.
sendto	`sendto(`*`bytes, address`*`)` or

	`sendto(`*`bytes, flags, address`*`)` Transmit the *`bytes`* (*`s`* must not be connected) to the given socket address.
sendmsg	`sendmsg(`*`buffers, [ancdata, [flags, [address]]]`*`)` Send normal and ancillary (out-of-band) data to the connected endpoint. *`buffers`* should be an iterable of bytes-like objects. The *`ancdata`* argument should be an iterable of (*`data, ancdata, msg_flags, address`*) tuples representing the ancillary data, and *`msg_flags`* are flags values documented on the Unix manual page for the `send(2)` system call. *address* should only be provided for an unconnected socket, and determines the endpoint to which the data is sent.
sendfile	`sendfile(`*`file, offset=0, count=None`*`)` Send the contents of file object *`file`* (which must be open in binary mode) to the connected endpoint. On platforms where `os.sendfile` is available, it's used; otherwise, the `send` call is used. *`offset`*, if any, determines the starting byte position in the file from which transmission begins; *`count`* sets maximum number of bytes to transmit. Returns the total number of bytes transmitted.
set_inheritable	`set_inheritable(`*`flag`*`)` Determines whether the socket gets inherited by child processes, according to the truth value of *`flag`*.
setblocking	`setblocking(`*`flag`*`)` Determines whether *`s`* operates in blocking mode (see "Socket Objects") according to the truth value of *`flag`*. `s.setblocking(True)` works like `s.settimeout(None)`; `s.set_blocking(False)` works like *s*.settimeout(0.0).
settimeout	`settimeout(`*`timeout`*`)` Establishes the mode of s (see "Socket Objects") according to the value of *`timeout`*.
shutdown	`shutdown(`*`how`*`)` Shuts down one or both halves of a socket connection according to the value of the *`how`* argument, as detailed here:
`socket.SHUT_RD`	No further receive operations can be performed on *`s`*.
`socket.SHUT_WR`	No further send operations can be performed on *`s`*.

`socket.SHUT_RDWR`	No further receive or send operations can be performed on *s*.

A socket object *s* also has the following attributes:

family	An attribute that is `s`'s socket family
type	An attribute that is `s`'s socket type

A Connectionless Socket Client

Consider a simplistic packet-echo service. Text encoded in UTF-8 (the assumed encoding of most Python source files, though here we have made it explicit) is sent to a server, which sends the same information back to the client originating it. In a connectionless service, all the client has to do is send each chunk of data to the defined server endpoint.

```
# coding: utf-8
import socket

UDP_IP = 'localhost'
UDP_PORT = 8883
MESSAGE = u"""\
This is a bunch of lines, each
of which will be sent in a single
UDP datagram. No error detection
or correction will occur.
Crazy bananas! £€ should go through."""

sock = socket.socket(socket.AF_INET,  # IP v4
                     socket.SOCK_DGRAM)  # UDP
server = UDP_IP, UDP_PORT
for line in MESSAGE.splitlines():
    data = line.encode('utf-8')
    sock.sendto(data, server)
    print('SENT', repr(data), 'to', server)
    response, address = sock.recvfrom(1024)  # buffer size: 1024
    print('RCVD', repr(response.decode('utf-8')), 'from',
address)
sock.close()
```

Note that the server is only expected to perform a byte-oriented echo function. The client, therefore, encodes its Unicode data into specific

bytestrings, and decodes the bytestring responses received from the server back into Unicode text using the same encoding.

A Connectionless Socket Server

A server for this service is also quite simple. It binds to its endpoint, receives packets (datagrams) at that endpoint, and returns a packet to the client sending each datagram, with exactly the same data. The server treats all clients equally and does not need to use any kind of concurrency (though this handy characteristic might not hold for a service where request handling takes more time).

The following server works, but offers no way to terminate the service other than by interrupting it (typically from the keyboard, with Ctrl-C or Ctrl-Break):

```
import socket
UDP_IP = 'localhost'
UDP_PORT = 8883
sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind((UDP_IP, UDP_PORT))
print('Serving at', UDP_IP, UDP_PORT)
while True:
    data, addr = sock.recvfrom(1024)  # buffer size is 1024 bytes
    print('RCVD', repr(data), 'from', addr)
    sock.sendto(data, addr)
    print('SENT', repr(data), 'to', addr)
```

Neither is there any mechanism to handle dropped packets and similar network problems. This is often acceptable in simple services.

The same programs will run using IPv6 by replacing the socket type, *AF.INET*, with *AF_INET6*.

A Connection-Oriented Socket Client

Consider a simplistic connection-oriented “echo-like” protocol: a server lets clients connect to its listening socket, receives arbitrary bytes from them,

and sends back to each client the same bytes that client sent to the server, until the client closes the connection. Here's an example of an elementary test client:[2]

```
# coding: utf-8
import socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect(('localhost', 8881))
    print('Connected to server')
    data = u"""\
A few lines of text
including non-ASCII characters: €£
to test the operation
of both server
and client."""
    for line in data.splitlines():
        sock.sendall(line.encode('utf-8'))
        print('Sent:', line)
        response = sock.recv(1024)
        print('Recv:', response.decode('utf-8'))
sock.close()
print('Disconnected from server')
```

Note that the data is text, so it must be encoded with a suitable representation, for which we chose the usual suspect—UTF-8. The server works in terms of bytes (since it is bytes, AKA octets, that travel on the network); the received bytes object gets decoded with UTF-8 back into Unicode text before printing. Any other suitable codec could be used: the key point is that text must be encoded before transmission and decoded after reception. The server, working in terms of bytes, does not even need to know which encoding is being used, except maybe for logging purposes.

A Connection-Oriented Socket Server

Here is a simplistic server corresponding to the testing client shown in "A Connection-Oriented Socket Client", using multithreading via concurrent.futures, covered in "The concurrent.futures Module":

```
from concurrent import futures
import socket
```

```
def handle(new_sock, address):
    print('Connected from', address)
    while True:
        received = new_sock.recv(1024)
        if not received:
            break
        s = received.decode('utf-8', errors='replace')
        print('Recv:', s)
        new_sock.sendall(received)
        print('Echo:', s)
    new_sock.close()
    print('Disconnected from', address)
servsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
servsock.bind(('localhost', 8881))
servsock.listen(5)
print('Serving at', servsock.getsockname())
with futures.ThreadPoolExecutor(20) as e:
    try:
        while True:
            new_sock, address = servsock.accept()
            e.submit(handle, new_sock, address)
    except KeyboardInterrupt:
        pass
    finally:
        servsock.close()
```

This server has its limits. In particular, it runs only 20 threads, so it cannot be simultaneously serving more than 20 clients; any further client trying to connect while 20 are being served waits in *servsock's* listening queue, and (should that queue fill up with 5 clients waiting to be accepted) further clients attempting connection get rejected outright. This server is intended just as an elementary example for demonstration purposes, not as a solid, scalable, or secure system.

As before, the same programs will run using IPv6 by replacing the socket type, *`AF.INET`*, with *`AF_INET6`*.

Transport Layer Security (TLS, AKA SSL)

The Transport Layer Security (TLS) is often also known as the Secure Sockets Layer (SSL), which was in fact the name of its predecessor protocol. TLS provides privacy and data integrity over TCP/IP, helping you

defend against server impersonation, eavesdropping on the bytes being exchanged, and malicious alteration of those bytes. For an introduction to TLS, we recommend the extensive Wikipedia entry.

In Python, you can use TLS via the `ssl` module of the standard library, documented in detail online. To use `ssl` well, you need its rich online docs, as well as a deep and broad understanding of TLS itself (the Wikipedia article, excellent and vast as it is, can only begin to cover this large, difficult subject). In particular, you must learn and thoroughly understand the security considerations section of the online docs, as well as all the materials found at the many links helpfully offered in that section.

If these considerations make it look like a perfect implementation of security precautions is a daunting task, that's because it *is.* In security, you're pitting wits and skills against those of sophisticated attackers who may be more familiar with the nooks and crannies of the problems involved, since they specialize in finding workarounds and breaking in, while (usually) your focus can't be exclusively on such issues—rather, you're trying to provide some useful services in your code. It's risky to see security as an afterthought or a secondary point—it *has* to be front-and-center throughout, to win said battle of skills and wits.

That said, we strongly recommend undertaking the above-outlined study of TLS to all readers—the better all developers understand security considerations, the better off we all are (except, we guess, for "black-hat" security-breaker wannabes).

Unless you have acquired a really deep and broad understanding of TLS and Python's `ssl` module (in which case, you'll know what exactly to do—better than we possibly could!), we recommend using an `SSLContext` instance to hold all details of your use of TLS. Build that instance with the `ssl.create_default_context` function, add your certificate if needed (it *is* needed if you're writing a secure server), then use the instance's `wrap_socket` method to wrap (almost[3]) every `socket.socket` instance you make into an instance of `ssl.SSLSocket`—behaving almost identically to the socket object it

wraps, but nearly transparently adding security checks and validation "on the side."

The default TLS contexts strike a good compromise between security and broad usability, and we recommend you stick by them (unless you're knowledgeable enough to fine-tune and tighten security for special needs). If you need to support outdated counterparts, those unable to use the most recent, most secure implementations of TLS, you may feel tempted to learn just enough to relax your security demands; do that at your own risk—we most definitely *don't* recommend wandering into such territory!

In the following sections, we cover the minimal subset of `ssl` you need if you just want to follow our recommendations. But, remember, even if that is the case, *please* also read up on TLS and `ssl`, just to gain some background knowledge about the intricate issues involved. It may stand you in good stead one day!

SSLContext

The `ssl` module supplies an `ssl.SSLContext` class, whose instances hold information about TLS configuration (including certificates and private keys) and offer many methods to set, change, check, and use that information. If you know exactly what you're doing, you can manually instantiate, set up, and use your own `SSLContext` instances for your own specialized purposes.

However, we recommend instead that you instantiate an `SSLContext` using the well-tuned function named `ssl.create_default_context` with a single argument: `ssl.Purpose.CLIENT_AUTH` if your code is a server (and thus may need to authenticate clients), or `ssl.Purpose.SERVER_AUTH` if your code is a client (and thus definitely needs to authenticate servers). If your code is both a client to some servers and a server to other clients (as, for example, some Internet *proxies* are), then you'll need two instances of `SSLContext`, one for each purpose.

For most client-side uses, your `SSLContext` is ready. If you're coding a server, or a client for one of the rare servers that require TLS authentication of the clients, you need to have a certificate file and a key file, and add them to the `SSLContext` instance (so that counter-parties can verify your identity) with code such as, for example:

```
ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ctx.load_cert_chain(certfile='mycert.pem', keyfile='mykey.key')
```

passing the paths to the certificate and key files to the `load_cert_chain` method. (See the online docs to learn how to obtain key and certificate files.)

Once your context instance *`ctx`* is ready, if you're coding a client, just call *`ctx`*`.wrap_socket` to wrap any socket you're about to connect to a server, and use the wrapped result (an instance of `ssl.SSLSocket`) instead of the socket you just wrapped. For example:

```
sock = socket.socket(socket.AF_INET)
sock = ctx.wrap_socket(sock, server_hostname='www.example.com')
sock.connect(('www.example.com', 443))# just use 'sock' normally
from here onwards
```

Note that, in the client case, you should also pass `wrap_socket` a `server_hostname` argument corresponding to the server you're about to connect to; this way, the connection can verify that the identity of the server you end up connecting to is indeed correct, one absolutely crucial step to any Internet security.

Server-side, *don't* wrap the socket that you are binding to an address, listening on, or accepting connections on; just bind the new socket to `accept` returns. For example:

```
sock = socket.socket(socket.AF_INET)
sock.bind(('www.example.com', 443))
sock.listen(5
```

```
while True:
    newsock, fromaddr = sock.accept()
    newsock = ctx.wrap_socket(newsock, server_side=True)
    # deal with 'newsock' as usual, shut down and close it when
done
```

In this case, what you need to pass to `wrap_socket` is an argument `server_side=True`, so it knows that you're on the server side of things.

Again, we recommend the online docs, particularly the examples, for better understanding even this simple subset of `ssl` operations.

---

1 and the relatively-newfangled multiplexed-connections transport protocol QUIC

2 This client example isn't secure; see "Transport Layer Security (TLS, AKA SSL)" for an introduction to making it secure.

3 We say "almost" because, when you code a server, you don't wrap the socket you bind, listen on, and accept connections from.

Chapter 16. Client-Side Network Protocol Modules

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 19th 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 standard library supplies several modules to simplify the use of Internet protocols, particularly on the client side (or for some simple servers). These days, the Python Package Index, best known as PyPI, offers many more such packages. These third-party packages support a wider array of protocols, and several offer better APIs than the standard library's equivalents. When you need to use a network protocol that's missing from the standard library, or covered by the standard library in a way you think is not satisfactory, be sure to search PyPI—you're likely to find better solutions there.

In this chapter, we cover some standard library packages that may prove satisfactory for some uses of network protocols, especially simple ones: when you can code without requiring third-party packages, your application or library is easier to install on other machines. We also mention a few third-party packages covering important network protocols not included in the standard library. We do not cover third-party packages using asynchronous programming.

For the very frequent use case of HTTP[1] clients and other network resources (such as anonymous FTP sites) best accessed via URLs,[2] the

standard library is seriously inferior to the splendid third-party package requests, so we cover and recommend that instead of standard library modules.

Email Protocols

Most email today is *sent* via servers implementing the Simple Mail Transport Protocol (SMTP) and *received* via servers and clients using Post Office Protocol version 3 (POP3) and/or IMAP4 (either in the original version, per RFC 1730, or the IMAP4rev1 one, per RFC 2060). Clients for these protocols are supported by the Python standard library modules `smtplib`, `poplib`, and `imaplib` respectively.

If you need to write a client that can connect via either POP3 or IMAP4, a standard recommendation would be to pick IMAP4, since it is more powerful, and—according to Python's own online docs—often more accurately implemented on the server side. Unfortunately `imaplib` is very complex, and far too vast to cover in this book. If you do choose to go that route, use the online docs, inevitably complemented by voluminous RFCs 1730 or 2060, and possibly other related RFCs, such as 5161 and 6855 for capabilities, and 2342 for namespaces. Using the RFCs, in addition to the online docs for the standard library module, can't be avoided: many of the arguments passed to `imaplib` functions and methods, and results from calling them, are strings with formats that are only documented in the RFCs, not in Python's own docs. We don't cover `imaplib` in this book; in the following sections, we only cover poplib and `smtplib`.

If you do want to use the rich and powerful IMAP4 protocol, we highly recommend that you do so, not directly via the standard library's `imaplib`, but rather by leveraging the simpler, higher-abstraction-level third-party package IMAPClient, available with a `pip install` and well-documented online.

The poplib Module

The `poplib` module supplies a class `POP3` to access a POP mailbox. The specifications of the POP protocol are in RFC 1939.

POP3	`class POP3(`*`host`*`,`*`port`*`=110)` Returns an instance *`p`* of class POP3 connected to *`host`* and *`port`*. Class `POP3_SSL` behaves just the same, but connects to the host (by default on port 995) over a secure TLS channel; it's needed to connect to email servers that demand some minimum security, such as pop.gmail.com.[a]

a To connect to a Gmail account, in particular, you need to configure that account to enable POP, "Allow less secure apps," and avoid 2-step verification—things that in general we don't recommend, as they weaken your email's security.

Instance *`p`* supplies many methods, of which the most frequently used are the following (in each case, *`msgnum`*, the identifying number of a message, can be a string or an `int`):

dele	*`p`*`.dele(`*`msgnum`*`)` Marks message *`msgnum`* for deletion and returns the server response string. The server queues such deletion requests, and executes them later when you terminate this connection by calling *`p`*`.quit`.
list	*`p`*`.list(`*`msgnum`*`=None)` Returns a tuple (*`response, messages, octets`*), where *`response`* is the server response string; *`messages`* a list of bytestrings, each of two words `b'`*`msgnum bytes`*`'`, message number and length in bytes of each message in the mailbox; *`octets`* is the length in bytes of the total response. When *`msgnum`* is not None, `list` returns a string, the response for the given *`msgnum`*, not a tuple.
pass_	*`p`*`.pass_(`*`password`*`)` Sends the password to the server. Must be called after *`p`*`.user`. The trailing underscore in the name is needed because `pass` is a Python keyword. Returns the server response string.
quit	*`p`*`.quit()` Ends the session and tells the server to perform deletions that were requested by calls to *`p`*`.dele`. Returns the server response string.
retr	*`p`*`.retr(`*`msgnum`*`)` Returns a three-item tuple (*`response, lines, bytes`*),

	where *response* is the server response string, *lines* the list of all lines in message *msgnum* as bytestrings, and *bytes* the total number of bytes in the message.
set_debuglevel	`p.set_debuglevel(debug_level)` Sets debug level to `int debug_level`: `0`, default, for no debugging; `1` for a modest amount of debugging output; and `2` or more for a complete output trace of all control information exchanged with the server.
stat	`p.stat()` Returns pair (*num_msgs,bytes*), where *num_msgs* is the number of messages in the mailbox, *bytes* the total number of bytes.
top	`p.top(msgnum,maxlines)` Like `retr`, but returns at most *maxlines* lines from the message's body (in addition to all the lines from the headers). Can be useful for peeking at the start of long messages.
user	`p.user(username)` Sends the server the username; invariably followed up by a call to `p.pass_`.

The smtplib Module

The `smtplib` module supplies a class SMTP to send mail via an SMTP server. The specifications of the SMTP protocol are in RFC 2821.

SMTP	`class SMTP([host,port=25])` Returns an instance *s* of the class `SMTP`. When *host* (and optionally `port`) is given, implicitly calls *s*`.connect(`*host*`,port)`. Class `SMTP_SSL` behaves just the same, but connects to the host (by default on port `465`) over a secure TLS channel; it's needed to connect to email servers that demand some minimum security, such as smtp.gmail.com.

The instance *s* supplies many methods, of which the most frequently used are the following:

connect	*s*`.connect(host=127.0.0.1,port=25)` Connects to an SMTP server on the given `host` (by default, the local host) and `port` (port `25` is the default port for the SMTP service; `465` is the default port for the more secure "SMTP over TLS").

login	`s.login(user, password)` Logs into the server with given `user` and `password`. Needed only if the SMTP server requires authentication (as just about all do).
quit	`s.quit()` Terminates the SMTP session.
sendmail	`s.sendmail(from_addr, to_addrs ,msg_string)` Sends mail message `msg_string` from the sender whose address is in string `from_addr` to each of the recipients whose addresses are the items of list `to_addrs`. `msg_string` must be a complete RFC 822 message in a single multiline bytestring: the headers, an empty line for separation, then the body. `from_addr` and `to_addrs` only direct the mail transport, and don't affect any header in `msg_string`. To prepare RFC 822-compliant messages, use the package `email`, covered in "MIME and Email Format Handling".

HTTP and URL Clients

Most of the time, your code uses HTTP and FTP protocols through the higher-abstraction URL layer, supported by the modules and packages covered in the following sections. Python's standard library also offers lower-level, protocol-specific modules that are less often used: for FTP clients, ftplib; for HTTP clients, `http.client` (we cover HTTP servers in Chapter 20). If you need to write an FTP server, look at the third-party module pyftpdlib. Implementations of the newer HTTP/2 may not be fully mature, but your best bet as of this writing is the third-party module HTTPX. We do not cover any of these lower-level modules in this book: we focus on higher-abstraction, URL-level access throughout the following sections.

URL Access

A URL is a type of URI (Uniform Resource Identifier). A URI is a string that *identifies* a resource (but does not necessarily *locate* it), while a URL *locates* a resource on the Internet. A URL is a string composed of several optional parts, called *components: scheme, location, path, query*, and

fragment. (The second component is sometimes also known as a *net location*, or *netloc* for short.) A URL with all parts looks like:

```
scheme://lo.ca.ti.on/pa/th?qu=ery#fragment
```

In *https://www.python.org/community/awards/psf-awards/#october-2016*, for example, the scheme is *http*, the location is *www.python.org*, the path is */community/awards/psf-awards/*, there is no query, and the fragment is *#october-2016.* (Most schemes default to a "*well-known port*" when the port is not explicitly specified; for example, 80 is the "well-known port" for the HTTP scheme.) Some punctuation is part of one of the components it separates; other punctuation characters are just separators, not part of any component. Omitting punctuation implies missing components. For example, in *mailto:me@you.com*, the scheme is *mailto*, the path is *me@you.com* (mailto:me@you.com), and there is no location, query, or fragment. No // means the URI has no location, no ? means it has no query, and no # means it has no fragment.

If the location ends with a colon followed by a number, this denotes a TCP port for the endpoint. Otherwise, the connection uses the "well-known port" associated with the scheme (e.g., port 80 for HTTP).

The urllib.parse module

The `urllib.parse` module supplies functions for analyzing and synthesizing URL strings. The most frequently used of these functions are `urljoin`, `urlsplit`, and `urlunsplit`:

urljoin

`urljoin(`*`base_url_string`*`, `*`relative_url_string`*`)`

Returns a URL string *`u`*, obtained by joining *`relative_url_string`*, which may be relative, with *`base_url_string`*. The joining procedure that urljoin performs to obtain its result *`u`* may be summarized as follows:

- When either of the argument strings is empty, *u* is the other argument.

- When *relative_url_string* explicitly specifies a scheme that is different from that of *base_url_string*, *u* is *relative_url_string*. Otherwise, *u*'s scheme is that of *base_url_string*.

- When the scheme does not allow relative URLs (e.g., `mailto`), or when `relative_url_string` explicitly specifies a location (even when it is the same as the location of *base_url_string*), all other components of *u* are those of *relative_url_string*. Otherwise, *u*'s location is that of *base_url_string*.

- *u*'s path is obtained by joining the paths of *base_url_string* and *relative_url_string* according to standard syntax for absolute and relative URL paths, as per RFC 1808. For example:

```
from urllib import parse as urlparse
urlparse.urljoin(
'http://host.com/some/path/here','../other/path
')# Result is:
# 'http://host.com/some/other/path'
```

urlsplit

`urlsplit(`*url_string*`,default_scheme='',allow_fragments=True)`

Analyzes *url_string* and returns a tuple (actually an instance of `SplitResult`, which you can treat as a tuple or use with named attributes) with five string items: *scheme*, *netloc*, *path*, *query*, and *fragment*. *default_scheme* is the first item when the *url_string* lacks an explicit scheme. When `allow_fragments` is `False`, the tuple's last item is always `''`, whether or not *url_string* has a fragment. Items corresponding to missing parts are ''. For example:

```
urlparse.urlsplit('http://www.python.org:80/faq.cgi?src=file')
# Result is:
#
('http','www.python.org:80','/faq.cgi','src=file','')
```

urlunsplit `urlunsplit(url_tuple)`

url_tuple is any iterable with exactly five items, all strings. Any return value from a urlsplit call is an acceptable argument for `urlunsplit`. `urlunsplit` returns a URL string with the given components and the needed separators, but with no redundant separators (e.g., there is no # in the result when the fragment, *url_tuple*'s last item, is `''`). For example:

```
urlparse.urlunsplit((
'http','www.python.org','/faq.cgi','src=fie',''))
# Result is:
# 'http://www.python.org/faq.cgi?src=fie'
```

`urlunsplit(urlsplit(x))` returns a normalized form of URL string *x*, which is not necessarily equal to *x* because *x* need not be normalized. For example:

```
urlparse.urlsplit('http://a.com/path/a?'))# Result is:
# 'http://a.com/path/a'
```

In this case, the normalization ensures that redundant separators, such as the trailing ? in the argument to `urlsplit`, are not present in the result.

The Third-Party requests Package

The third-party `requests` package (very well documented online) is how we recommend you access HTTP URLs. As usual for third-party packages, it's best installed with a simple pip install requests. In this section, we summarize how best to use it for reasonably simple cases.

Natively, requests only supports the HTTP transport protocol; to access URLs using other protocols, you need to install other third-party packages (known as *protocol adapters*), such as `requests-ftp` for FTP URLs, or others supplied as part of the rich `requests-toolbelt` package of `requests` utilities.

`requests`' functionality hinges mostly on three classes it supplies: `Request`, modeling an HTTP request to be sent to a server; `Response`, modeling a server's HTTP response to a request; and `Session`, offering continuity across a sequence of requests, also known as a *session.* For the common use case of a single request/response interaction, you don't need continuity, so you may often just ignore `Session`.

Sending requests

Most often, you don't need to explicitly consider the `Request` class: rather, you call the utility function `request`, which internally prepares and sends the `Request`, and returns the `Response` instance. `request` has two mandatory positional arguments, both `str`s: `method`, the HTTP method to use, and `url`, the URL to address; then, many optional named parameters may follow (in the next section, we cover the most commonly used named parameters to the `request` function).

For further convenience, the `requests` module also supplies functions whose names are the HTTP methods `delete`, `get`, `head`, `options`, `patch`, `post`, and `put`; each takes a single mandatory positional argument, `url`, then the same optional named arguments as the function `request`.

When you want some continuity across multiple requests, call `Session` to make an instance `s`, then use `s`'s methods `request`, `get`, `post`, and so on, which are just like the functions with the same names directly supplied

by the `requests` module (however, *`s`*'s methods merge *`s`*'s settings with the optional named parameters to prepare each request to send to the given `url`).

request's optional named parameters

The function `request` (just like the functions `get`, `post`, and so on—and methods with the same names on an instance *`s`* of class `Session`) accepts many optional named parameters—refer to the `requests` package's excellent online docs for the full set, if you need advanced functionality such as control over proxies, authentication, special treatment of redirection, streaming, cookies, and so on. The most frequently used named parameters are:

`data`

A `dict`, a sequence of key/value pairs, a bytestring, or a file-like object, to use as the body of the request

`headers`

A `dict` of HTTP headers to send in the request

`json`

Python data (usually a `dict`) to encode as JSON as the body of the request

`files`

A `dict` with names as keys, file-like objects, or *file tuples* as values, used with the POST method to specify a multipart-encoding file upload; we cover the format of values for `files=` in the next section

`params`

A `dict` of (name, value) items, or a bytestring to send as the query string with the request

`timeout`

> A `float` number of seconds, the maximum time to wait for the response before raising an exception

`data`, `json`, and `files` are mutually incompatible ways to specify a body for the request; use only one of them, and only for HTTP methods that do use a body, namely PATCH, POST, and PUT. The one exception is that you can have both `data=` passing a `dict`, and a `files=`, and that is very common usage: in this case, both the key/value pairs in the `dict`, and the files, form the body of the request as a single *multipart/form-data* whole, according to RFC 2388.

The files argument (and other ways to specify the request's body)

When you specify the request's body with `json=`, or `data=` passing a bytestring or a file-like object (which must be open for reading, usually in binary mode), the resulting bytes are directly used as the request's body; when you specify it with `data=` (passing a `dict`, or a sequence of key/value pairs), the body is built as a *form*, from the key/value pairs formatted in *application/x-www-form-urlencoded* format, according to the relevant web standard.

When you specify the request's body with `files=`, the body is also built as a form, in this case with the format set to *multipart/form-data* (the only way to upload files in a PATCH, POST, or PUT HTTP request). Each file you're uploading is formatted into its own part of the form; if, in addition, you want the form to give to the server further non-file parameters, then in addition to `files=` also pass a `data=` with a `dict` value (or a sequence of key/value pairs) for the further parameters—those parameters get encoded into a supplementary part of the multipart form.

For flexibility, the value of the `files=` argument can be a `dict` (its items are taken as a sequence of name/value pairs), or a sequence of name/value pairs (order is maintained in the resulting request body).

Either way, each value in a name/value pair can be a `str` (or, best,[3] a `bytes` or `byte array`) to be used directly as the uploaded file's contents; or, a file-like object open for reading (then, `requests` calls `.read()` on it, and uses the result as the uploaded file's contents; we strongly urge that, in such cases, you open the file in binary mode, to avoid any ambiguity regarding content-length). When any of these conditions apply, `requests` uses the *`name`* part of the pair (e.g., the key into the `dict`) as the file's name (unless it can improve on that because the open file object is able to reveal its underlying filename), takes its best guess at a content-type, and uses minimal headers for the file's form-part.

Alternatively, the value in each name/value pair can be a tuple with two to four items: *`fn`*, *`fp`*, `[`*`ft`*, `[`*`fh`*`]]` (using square brackets as meta-syntax to indicate optional parts). In this case, *`fn`* is the file's name, *`fp`* provides the contents (in just the same way as in the previous paragraph), optional *`ft`* provides the content type (if missing, `requests` guesses it, as in the previous paragraph), and the optional `dict` *`fh`* provides extra headers for the file's form-part.

How to study examples of requests

In practical applications, you don't usually need to consider the internal instance *r* of the class `requests.Request`, which functions like `requests.post` are building, preparing, and then sending on your behalf. However, to understand exactly what `requests` is doing, working at a lower level of abstraction (building, preparing, and examining *`r`*—no need to send it!) is instructive. For example:

```
import requests
r = requests.Request('GET', 'http://www.example.com',
    data={'foo': 'bar'}, params={'fie': 'foo'})
p = r.prepare()
print(p.url)
print(p.headers)
print(p.body)
```

prints out (splitting the *`p`*`.headers` `dict`'s printout for readability):

```
http://www.example.com/?fie=foo
{'Content-Length': '7',
 'Content-Type': 'application/x-www-form-urlencoded'}
foo=bar
```

Similarly, when `files=` is involved:

```
import requests
r = requests.Request('POST', 'http://www.example.com',
    data={'foo': 'bar'}, files={'fie': 'foo'})
p = r.prepare()
print(p.headers)
print(p.body)
```

prints out (with several lines split for readability):

```
{'Content-Length': '228',
 'Content-Type': 'multipart/form-data;
boundary=dfd600d8aa584962709b936134b1cfce'}
b'--dfd600d8aa584962709b936134b1cfce\r\nContent-Disposition:
form-data; name="foo"\r\n\r\nbar\r\n--
dfd600d8aa584962709b936134b1cfce\r\nContent-Disposition: form-
data; name="fie"; filename="fie"\r\n\r\nfoo\r\n--
dfd600d8aa584962709b936134b1cfce--\r\n'
```

Happy interactive exploring!

The Response class

The one class from the `requests` module that you always have to consider is `Response`: every request, once sent to the server (typically, that's done implicitly by methods such as `get`), returns an instance *`r`* of `requests.Response`.

The first thing you usually want to do is to check *`r`*`.status_code`, an `int` that tells you how the request went, in typical "HTTPese": 200 means "everything's fine," 404 means "not found," and so on. If you'd rather just get an exception for status codes indicating some kind of error, call *`r`*`.raise_for_status()`; that does nothing if the request went fine,

but raises a `requests.exceptions.HTTPError` otherwise. (Other exceptions, not corresponding to any specific HTTP status code, can and do get raised without requiring any such explicit call: e.g., `ConnectionError` for any kind of network problem, or `TimeoutError` for a timeout.)

Next, you may want to check the response's HTTP headers: for that, use *`r`*`.headers`, a `dict` (with the special feature of having case-insensitive string-only keys, indicating the header names as listed, e.g., in Wikipedia, per HTTP specs). Most headers can be safely ignored, but sometimes you'd rather check. For example, you may check whether the response specifies which natural language its body is written in, via *`r`*`.headers.get('content-language')`, to offer the user different presentation choices, such as the option to use some kind of language translation service to make the response more usable for the user.

You don't usually need to make specific status or header checks for redirects: by default, `requests` automatically follows redirects for all methods except HEAD (you can explicitly pass the `allow_redirection` named parameter in the request to alter that behavior). If you allow redirects, you may want to check *`r`*`.history`, a list of all `Response` instances accumulated along the way, oldest to newest, up to but excluding *`r`* itself (*`r`*`.history` is empty if there have been no redirects).

Most often, maybe after checking status and headers, you want to use the response's body. In simple cases, just access the response's body as a bytestring, *r*.content, or decode it as JSON (once you've checked that's how it's encoded, e.g., via *r*.headers.get('content-type')) by calling *r*.json().

Often, you'd rather access the response's body as (Unicode) text, with property *`r`*`.text`. The latter gets decoded (from the octets that actually make up the response's body) with the codec `requests` thinks is best, based on the content-type header and a cursory examination of the body itself. You can check what codec has been used (or is about to be used) via the attribute *`r`*`.encoding`, the name of a codec registered with the

`codecs` module, covered in "The codecs Module". You can even *override* the choice of codec to use by *assigning* to `r.encoding` the name of the codec you choose.

We do not cover other advanced issues, such as streaming, in this book; check `requests`' online docs.

The urllib Package

In addition to `urllib.parse`, covered in "The urllib.parse module", the `urllib` package supplies the module `urllib.robotparser` for the specific purpose of parsing a site's *robots.txt* file as per a well-known informal standard; the module `urllib.error`, containing all exception types raised by other `urllib` modules; and, mainly, the module `urllib.request`, for opening and reading URLs.

For full coverage of `urllib.request`, check the online docs, and Michael Foord's HOWTO.

Other Network Protocols

Many, *many* other network protocols are in use—a few are best supported by Python's standard library, but, for most of them, you'll be happier researching third-party modules on PyPI.

To connect as if you were logging into another machine (or, into a separate login session on your own node), you can use the secure SSH protocol, supported by the third-party module `paramiko`, or the higher abstraction layer wrapper around it, the third-party module `spur`. (You can also, with some likely security risks, still use classic telnet, supported by the standard library module `telnetlib`.)

Other network protocols include:

- NNTP, to access Usenet News servers, supported by the standard library module `nntplib`

- XML-RPC, for a rudimentary remote procedure call functionality, supported by `xmlrpc.client`
- gRPC, for a more modern remote procedure functionality, supported by third-party module `grpcio`
- NTP, to get precise time off the network, supported by third-party module `ntplib`
- SNMP, for network management, supported by third-party module `pysnmp`

...among many others. No single book (not even this one!) could possibly cover all these protocols and their supporting modules. Rather, our best suggestion in the matter is a strategic one: whenever you decide that your application needs to interact with some other system via a certain networking protocol, don't rush to implement your own modules to support that protocol. Instead, search and ask around, and you're likely to find excellent existing Python modules (third-party, or standard-library ones) supporting that protocol.[4]

Should you find some bug or missing feature in such modules, open a bug or feature request (and, ideally, supply a patch or pull request that would fix the problem and satisfy your application's needs). In other words, become an active member of the open-source community, rather than just a passive user: you will be welcome there, scratch your own itch, and help many others in the process. "Give forward," since you cannot "give back" to all the awesome people who contributed to give you most of the tools you're using!

---

1 HTTP, the Hypertext Transfer Protocol, is the core protocol of the World Wide Web: every web server and browser uses it, and it has become the dominant application-level protocol on the Internet today.

2 Uniform Resource Locators

3 As it gives you complete, explicit control of exactly what octets are uploaded.

4 Even more important: if you think you need to invent a brand-new protocol and implement it on top of sockets, think again, and search carefully: it's far more likely that one or more of the huge number of existing Internet protocols meets your needs just fine!

Chapter 17. Serving HTTP

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 20th 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*.

When a browser (or any other web client) requests a page from a server, the server may return either static or dynamic content. Serving dynamic content involves server-side web programs generating and delivering content on the fly, often based on information stored in a database.

In the prehistory of the web, the standard for server-side programming was *CGI* (the Common Gateway Interface), which required the server to run a separate program each time a client requested dynamic content. Process startup time, interpreter initialization, connection to databases, and script initialization add up to measurable overhead; CGI did not scale up well.

Nowadays, web servers support many server-specific ways to reduce overhead, serving dynamic content from processes that can serve for several hits rather than starting up a new process per hit. Therefore, we do not cover CGI in this book. To maintain existing CGI programs, or port them to more modern approaches, consult the online docs for the standard library modules `cgi` and `http.cookies`.[1]

HTTP has become even more fundamental to distributed systems design with the emergence of systems based on microservices, offering a convenient way to transport between processes the JSON content that is frequently used. There are thousands of publicly available HTTP data APIs

on the Internet. While HTTP's principles remain almost unchanged since its inception in the mid-1990s, it has been significantly enhanced over the years to extend its capabilities. For a more thorough grounding with excellent reference materials we recommend O'Reilly's *HTTP: The Definitive Guide*[2].

http.server

Python's standard library includes a module containing the server and handler classes to implement a simple HTTP server.

You can run this server from the command line by just entering:

```
python -m http.server port_number
```

By default, the server listens on all interfaces and provides access to the files in the current directory. One author uses this as a simple means for file transfer: start up a Python `http.server` in the file directory on the source system, and then copy files to the destination using a utility such as `wget` or `curl`.

> ### `http.server` has very limited security features
>
> For production use, use one of the frameworks mentioned in the following sections.

You can find further information on `http.server` in the online docs.

WSGI

Python's Web Server Gateway Interface (WSGI) is the standard way for all modern Python web development frameworks to interface with underlying

web servers or gateways. WSGI is not meant for direct use by your application programs; rather, you code your programs using any one of many higher-abstraction frameworks, and the framework, in turn, uses WSGI to talk to the web server.

You need to care about the details of WSGI only if you're implementing the WSGI interface for a web server that doesn't already provide it (should any such server exist), or if you're building a new Python web framework.[3] If so, study the WSGI PEP, the docs for the standard library package `wsgiref`, and the archive of wsgi.org.

A few WSGI *concepts* may be important to you if you use lightweight frameworks (i.e., ones that match WSGI closely). WSGI is an *interface*, and that interface has two sides: the *web-server/gateway* side, and the *application/framework* side.

The framework side's job is to provide a *WSGI application* object, a callable object (often the instance of a class with a `__call__` special method, but that's an implementation detail) respecting conventions in the PEP, and to connect the application object to the server by whatever means the specific server documents (often a few lines of code, or configuration files, or just a convention such as naming the WSGI application object `application` as a top-level attribute in a module). The server calls the application object for each incoming HTTP request, and the application object responds appropriately so that the server can form the outgoing HTTP response and send it on—all according to said conventions. A framework, even a lightweight one, shields you from such details (except that you may have to instantiate and connect the application object, depending on the specific server).

WSGI Servers

An extensive list of servers and adapters you can use to run WSGI frameworks and applications (for development and testing, in production web setups, or both) is available online—extensive, but just partial. For example, it does not mention that Google App Engine's Python runtime is

also a WSGI server, ready to dispatch WSGI apps as directed by the *app.yaml* configuration file.

If you're looking for a WSGI server to use for development, or to deploy in production behind, say, an Nginx-based load balancer, you should be happy, at least on Unix-like systems, with Green Unicorn: pure Python goodness, supporting nothing but WSGI, very lightweight. A worthy (also pure Python and WSGI only) alternative, currently with better Windows support, is Waitress. If you need richer features (such as support for Perl and Ruby as well as Python, and many other forms of extensibility), consider the bigger, more complex uWSGI.[4]

WSGI also has the concept of *middleware*—a subsystem that implements both the server and application sides of WSGI. A middleware object "wraps" a WSGI application; can selectively alter requests, environments, and responses; and presents itself to the server as "the application." Multiple layers of wrappers are allowed and common, forming a "stack" of middleware, offering services to the actual application-level code. If you want to write a cross-framework middleware component, then you may, indeed, need to become a WSGI expert.

ASGI

If you're into asynchronous Python (which we don't cover in this book), WSGI's "moral successor" for the role is ASGI – as is usually the case for asynchronous programs in a networking environment, it can offer greatly improved performance, albeit (arguably) at some conceptual cost in learning and practicing.

Python Web Frameworks

For a survey of most Python web frameworks, see the Python wiki page. It's authoritative, since it's on the official python.org website, and community-curated, so it stays up to date as time goes by. It lists and points to dozens of frameworks[5] that it identifies as "active," plus many more it identifies as "discontinued/inactive." In addition, it points to separate wiki

pages about Python content management systems, web servers, and web components and libraries thereof.

"Full-Stack" Versus "Lightweight" Frameworks

Roughly speaking, Python web frameworks can be classified as being either *full-stack* (trying to supply all the functionality you may need to build a web application) or *lightweight* (supplying just a handy interface to web serving itself, and letting you pick and choose your own favorite components for tasks such as interfacing to databases, covered in "The Python Database API (DBAPI) 2.0", and templating, covered in "Templating"—as well as other tasks covered by web components, as mentioned in the next section). Of course, like all taxonomies, this one is imprecise and incomplete, and requires value judgments; however, it's one way to start making sense of the many Python web frameworks.

In this book, we do not cover full-stack frameworks—each is far too complex. Nevertheless, one of them might be the best approach for your specific applications, so we mention a few of the most popular ones, and recommend that you check out their websites.

When you use lightweight frameworks

By definition of "lightweight", if you need any database, templating, or other functionality not strictly related to HTTP, you'll be picking and choosing separate components for that purpose (we cover aspects of databases and templating in other chapters). However, the lighter in weight your framework, the more components you will need to understand and integrate, for purposes such as authenticating a user or maintaining state across web requests by a given user. Many WSGI middleware packages can help you with such tasks. Some excellent ones are quite focused—for example, Oso for access control, Beaker for maintaining state in the form of lightweight sessions of any one of several kinds, and so forth.

However, when we (the authors of this book) require good WSGI middleware for just about any purpose, we almost invariably first check Werkzeug, a collection of such components that's amazing in breadth and

quality. We don't cover Werkzeug in this book (just as we don't cover other middleware), but we recommend it highly (Werkzeug is also the foundation on which is built Flask, our favorite lightweight framework, which we do cover later in this chapter).

A Few Popular Full-Stack Frameworks

By far the most popular full-stack framework is Django, which is vast and extensible. Django's so-called *applications* are in fact reusable subsystems, while what's normally called "an application" Django calls a *project*. Django requires its own unique mindset, but offers vast power and functionality in return.

An excellent full-stack alternative is web2py, just about as powerful, easier to learn, and well known for its dedication to backward compatibility (if it keeps up its great track record, any web2py application you code today will keep working indefinitely far in the future). Its documentation is outstanding.

A third worthy contender is TurboGears, which starts out as a lightweight framework, but achieves "full-stack" status by fully integrating other, independent third-party projects for the various other functionalities needed in most web apps, such as database interfacing and templating, rather than designing its own. Another, somewhat philosophically similar, "light but rich" framework is Pyramid.

Some Popular Lightweight Frameworks

As mentioned, Python has multiple frameworks, including many lightweight ones. We cover two of the latter: popular, general-purpose Flask, and API-centric FastAPI.

Lightweight frameworks: you must know what you're doing!

You may well notice that properly using lightweight frameworks requires you to understand HTTP (in other words, to know what you're doing), while a full-stack framework tries to lead you by the hand and have you do the right thing without really needing to understand how or why it is right—at the cost of time and resources, and of accepting the full-stack framework's conceptual map and mindset. The authors of this book are enthusiasts of the knowledge-heavy, resources-light approach of lightweight frameworks, but we acknowledge that there are many situations where the rich, heavy, all-embracing full-stack frameworks are more appropriate. To each their own!

Flask

The most popular Python lightweight framework is Flask. Although lightweight, it includes a development server and debugger, and explicitly relies on other, well-chosen packages such as Werkzeug for middleware and jinja for templating (both packages were originally authored by Armin Ronacher, the author of Flask).

In addition to the project website (which includes rich, detailed docs), look at the sources on GitHub and the PyPI entry. To run Flask on Google App Engine (locally on your computer, or on Google's servers at *appspot.com*), Dough Mahugh's Medium article can be quite handy.

We also highly recommend Miguel Greenberg's book Flask Web Development, 2nd edition (O'Reilly): although the 2nd edition is rather dated (almost 4 years old at the time of this writing), it still provides an excellent foundation, on top of which you'll have a far easier time learning the latest new additions.

The main class supplied by the `flask` package is named `Flask`. An instance of `flask.Flask`, besides being a WSGI application itself, also wraps a WSGI application as its `wsgi_app` property. When you need to further wrap the WSGI app in some WSGI middleware, use the idiom:

```
import flask
```

```
app = flask.Flask(__name__)
app.wsgi_app = some_middleware(app.wsgi_app)
```

When you instantiate `flask.Flask`, always pass it as the first argument the application name (often just the `__name__` special variable of the module where you instantiate it; if you instantiate it from within a package, usually in *__init__.py*, `__name__.partition('.')[0]` works). Optionally, you can also pass named parameters such as `static_folder` and `template_folder` to customize where static files and jinja templates are found; however, that's rarely needed—the default values (sub-folders named *static* and *templates*, respectively, located in the same folder as the Python script that instantiates `flask.Flask`) make perfect sense.

An instance *app* of `flask.Flask` supplies more than 100 methods and properties, many of them decorators to bind functions to *app* in various roles, such as *view functions* (serving HTTP verbs on a URL) or *hooks* (letting you alter a request before it's processed, or a response after it's built; handling errors; and so forth).

`flask.Flask` takes just a few parameters at instantiation (and the ones it takes are not ones that you usually need to compute in your code), and it supplies decorators you'll want to use as you define, for example, view functions. Thus, the normal pattern in `flask` is to instantiate *app* early in your main script, just as your application is starting up, so that the app's decorators, and other methods and properties, are available as you `def` view functions and so on.

Since there is a single global *app* object, you may wonder how thread-safe it can be to access, mutate, and rebind *app*'s properties and attributes. Not to worry: the names you see are actually just *proxies* to actual objects living in the *context* of a specific request, in a specific thread or greenlet. Never type-check those properties (their types are in fact obscure proxy types), and you'll be fine.

Flask also supplies many other utility functions and classes; often, the latter subclass or wrap classes from other packages, to add seamless, convenient Flask integration. For example, Flask's `Request` and `Response` classes

add just a little handy functionality by subclassing the corresponding Werkzeug classes.

Flask request objects

The class `flask.Request` supplies a large number of thoroughly documented properties. Table 20-1 lists the ones you'll be using most often.

Property	Content
`cookies`	A `dict` with the cookies from the request
`data`	A string, the request's body (for POST and PUT requests)
`files`	A `MultiDict` whose values are file-like objects, all the files uploaded in the request (for POST and PUT requests), the mapping's keys being the files' names
`headers`	A `MultiDict` with the request's headers
`values`	A `MultiDict` with the request's parameters (either from the query string or, for POST and PUT requests, from the form that's the request's body)

A `MultiDict` is like a `dict`, except that it can have multiple values for a key. Indexing, and get, on a `MultiDict` instance *m*, return an arbitrary one of the values; to get the list of values for a key (an empty list, if the key is not in *m*), call *`m`*`.getlist(`*`key`*`)`.

Flask response objects

Often, a Flask view function can just return a string (which becomes the response's body): Flask transparently wraps an instance *`r`* of `flask.Response` around the string, so you don't have to worry about the response class. However, sometimes you want to alter the response's headers; in this case, in the view function, call *`r`* `= flask.make_response(`*`astring`*`)`, alter `MultiDict` *`r`*`.headers` as you want, then `return` *`r`*. (To set a cookie, don't use *`r`*`.headers`; rather, call *`r`*`.set_cookie`, with arguments as mentioned in `set_cookie` in Table 20-1.)

Some of Flask's built-in integrations with other systems don't require subclassing: for example, the templating integration implicitly injects into

the jinja context the Flask globals `config`, `request`, `session`, and `g` (the latter being the handy "globals catch-all" object `flask.g`, a proxy in application context, on which your code can store whatever you want to "stash" for the duration of the request being served), and the functions `url_for` (to translate an endpoint to the corresponding URL, same as `flask.url_for`) and `get_flashed_messages` (to support *flashed messages*, which we do not cover in this book; same as `flask.get_flashed_messages`). Flask provides convenient ways for your code to inject more filters, functions, and values into the jinja context, without any subclassing.

Most of the officially recognized or approved Flask extensions (hundreds are available from PyPI at the time of this writing) adopt similar approaches, supplying classes and utility functions to seamlessly integrate other popular subsystems with your Flask applications.

Flask also introduces other features such as signals, to provide looser dynamic coupling in a "pub/sub" pattern, and blueprints, offering a substantial subset of a Flask application's functionality to ease refactoring large applications in highly modular, flexible ways. We do not cover these advanced concepts in this book.

Example 20-1 shows a simple Flask example.

Example 17-1. A Flask example

```
import datetime, flask
app = flask.Flask(__name__)
app.permanent_session_lifetime = datetime.timedelta(days=365)
app.secret_key = b'\xc5\x8f\xbc\xa2\x1d\xeb\xb3\x94;:d\x03'
@app.route('/')
def greet():
    lastvisit = flask.session.get('lastvisit')
    now = datetime.datetime.now()
    newvisit = now.ctime()
    template = '''
      <html><head><title>Hello, visitor!</title>
      </head><body>
      {% if lastvisit %}
        <p>Welcome back to this site!</p>
        <p>You last visited on {{lastvisit}} UTC</p>
        <p>This visit on {{newvisit}} UTC</p>
```

```
    {% else %}
      <p>Welcome to this site on your first visit!</p>
      <p>This visit on {{newvisit}} UTC</p>
      <p>Please Refresh the web page to proceed</p>
    {% endif %}
    </body></html>'''
  flask.session['lastvisit'] = newvisit
  return flask.render_template_string(
    template, newvisit=newvisit, lastvisit=lastvisit)
```

This Flask example just shows how to use a few of the many building blocks that `Flask` offers—the Flask class, a view function, and rendering the response (in this case, using `render_template_string` on a jinja template; in real life, templates are usually kept in separate files rendered with `render_template`). The example also shows how to maintain continuity of state among multiple interactions with the server from the same browser, with the handy `flask.session` variable. (The example might alternatively have put together the HTML response in Python code instead of using jinja, and used a cookie directly instead of the session; however, real-world Flask apps do tend to use jinja and sessions by preference.)

If this app had multiple view functions, it might want to set `lastvisit` in the session whatever URL the app was getting "visited" in. Here's how to code and decorate a hook function to execute after each request:

```
@app.after_request()
def set_lastvisit(response):
    now = datetime.datetime.now()
    flask.session['lastvisit'] = now.ctime()
    return response
```

You can now remove the `flask.session['lastvisit'] = newvisit` statement from the view function `greet`, and the app will keep working fine.

FastAPI

FastAPI is of more recent design than Flask or Django. While both of the latter have very usable extensions to provide API services, FastAPI aims

squarely at producing HTTP-based APIs, as its name suggests. It's also perfectly capable of producing dynamic web pages intended for browser consumption, making it a versatile server. FastAPI's home page provides simple, short examples showing how it works and highlighting the advantages, and is followed by very thorough and detailed reference documentation.

As type annotations (covered in Chapter "Type Annotations") entered the language, they found wider use than originally intended in tools like `pydantic`, which uses them to perform runtime data validation. The FastAPI server exploits this support for validated structures, demonstrating great potential to improve web coding productivity through built-in validation of inputs.

FastAPI also relies on Starlette, a high-performance asynchronous web framework, which in turn uses an ASGI server such as Uvicorn or Hypercorn. You don't need to use async directly to take advantage of FastAPI—you can write your application in more traditional Python style, though your application might perform even faster if you do switch to the async style.

FastAPI's ability to provide type-accurate APIs (and automatically generated documentation for them) aligned with the types indicated by your annotations means it can provide automatic validation of incoming data and conversion on both input and output of data.

Consider the sample code shown in Example 20-2, which defines a simple model for both pydantic and mongoengine. Each has four fields; `name` and `description` are strings, `price` and `tax` are decimal. Values are required for the name and price fields but description and tax are optional. Pydantic establishes a default value of **`None`** for the description and tax fields; mongoengine does not store a value for fields whose value is **`None`**.

Example 17-2. models.py: Pydantic and Mongoengine data models

```
from decimal import Decimal
from pydantic import BaseModel, Field
from mongoengine import Document, StringField, DecimalField
from typing import Optional
```

```
class PItem(BaseModel):
    "pydantic typed data class."
    name: str
    price: Decimal
    description: Optional[str] = None
    tax: Optional[Decimal] = None

class MItem(Document):
    "mongoengine document."
    name = StringField(primary_key=True)
    price = DecimalField()
    description = StringField(required=False)
    tax = DecimalField(required=False)
```

Suppose you wanted to accept such data through a web form or as JSON, and to be able to retrieve them as JSON or display them in HTML, the skeletal example 20-5 below (offering no facilities to maintain existing data) shows you how you might do this with FastAPI. This uses the `uvicorn` HTTP server, but makes no attempt to explicitly use Python's async features. As with Flask, the program begins by creating an application object `app`, whose `route` decorator is then used to determine which view function handles requests to which path.

The `home_page` function, which takes no arguments, simply renders a minimal HTML home page containing a form from the index.html file, shown in Example 20-3. The form submits to the */items/new/form/* endpoint, which triggers a call to the `create_item_from_form` function, which is declared in the routing decorator as producing an HTML response rather than the default JSON.

Example 17-3. The index.html file

```
<!DOCTYPE html>
<html lang="en">
  <body>
  <h2>FastAPI Demonstrator</h2>
  <form method="POST" action="/items/new/form/">
    <table>
    <tr><td>Name</td><td><input name="name"></td></tr>
    <tr><td>Price</td><td><input name="price"></td></tr>
    <tr><td>Description</td><td><input name="description"></td>
</tr>
    <tr><td>Tax</td><td><input name="tax"></td></tr>
```

```
    <tr><td></td><td><input type="submit"></td></tr>
    </table>
  </form>
  </body>
</html>
```

Figure 20-1 is handled by `create_item_from_form` function, whose signature takes an argument for each form field, with annotations defining each as a form field. Note that the signature defines its own default values for `description` and `tax`. The function creates an `MItem` object from the form data and tries to save it in the database. The `save` function forces insertions, inhibiting the update of an existing record, and reports failure by returning **`None`**; the return value is used to formulate a simple HTML reply. In a production application a templating engine such as `jinja2` would typically be used to render the response.

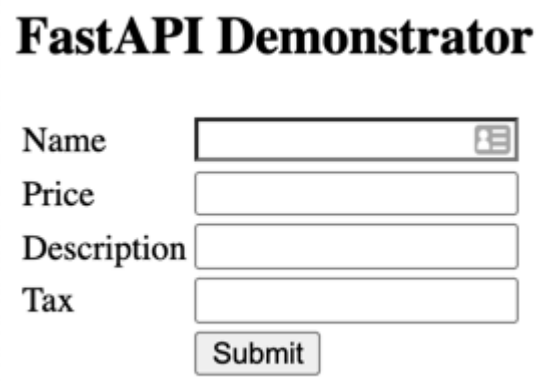


Figure 17-1. Please add a caption here.

Example 17-4. server.py: FastAPI sample code to accept and display item data

```
from decimal import Decimal
from fastapi import FastAPI, Form
from fastapi.responses import HTMLResponse, FileResponse
from mongoengine import connect
from mongoengine.errors import NotUniqueError
from typing import Optional
import json
```

```
import uvicorn
from models import PItem, MItem

DATABASE_URI = "mongodb://localhost:27017"
db=DATABASE_URI+"/mydatabase"
connect(host=db)
app = FastAPI()

def save(item):
    try:
        return item.save(force_insert=True)
    except NotUniqueError:
        return None

@app.get('/')
def home_page():
    "View function to display a simple form."
    return FileResponse("index.html")

@app.post("/items/new/form/", response_class=HTMLResponse)
def create_item_from_form(name: str=Form(...),
                          price: Decimal=Form(...),
                          description: Optional[str]=Form(""),
                          tax:
Optional[Decimal]=Form(Decimal("0.0"))):
    "View function to accept form data and create an item."
    mongoitem = MItem(name=name, price=price,
description=description, tax=tax)
    value = save(mongoitem)
    if value:
        body = f"Item({name!r}, {price!r}, {description!r},
{tax!r})"
    else:
        body = f"Item {name!r} already present."
    return f"""<html><body><h2>{body}</h2></body></html>"""

@app.post("/items/new/")
def create_item_from_json(item: PItem):
    "View function to accept JSON data and create an item."
    mongoitem = MItem(**item.dict())
    value = save(mongoitem)
    if not value:
        return f"Primary key {item.name!r} already present"
    return item.dict()

@app.get("/items/{name}/")
def retrieve_item(name: str):
    "View function to return the JSON contents of an item."
    m_item = MItem.objects(name=name).get()
    return json.loads(m_item.to_json())
```

```
if __name__ == "__main__":
    uvicorn.run("__main__:app", host="127.0.0.1"[6], port=8000,
reload=True)
```

The `create_item_from_json` function, routed from the */items/new/* endpoint, takes JSON input from a POST request. Its signature accepts a pydantic record, so in this case FastAPI will use pydantic's validation to determine whether the input is acceptable. The function returns a Python dictionary, which FastAPI automatically converts to a JSON response. This can easily be tested with a simple client, shown in Example 20-5.

Example 17-5. FastAPI test client

```
import requests, json

result = requests.post('http://localhost:8000/items/new/',
                       json={"name": "Item1",
                             "price": 12.34,
                             "description": "Rusty old bucket"})
print(result.status_code, result.json())
result = requests.get('http://localhost:8000/items/Item1/')
print(result.status_code, result.json())
result = requests.post('http://localhost:8000/items/new/',
                       json={"name": "Item2",
                             "price": "Not a number"})
print(result.status_code, result.json())
```

The results of running this program are shown in Example 20-6.

Example 17-6. FastAPI test client output

```
200 {'name': 'Item1', 'price': 12.34, 'description': 'Rusty old
bucket', 'tax': None}
200 {'_id': 'Item1', 'price': 12.34, 'description': 'Rusty old
bucket'}
422 {'detail': [{'loc': ['body', 'price'], 'msg': 'value is not a
valid decimal', 'type': 'type_error.decimal'}]}
```

The first post request to */items/new/* sees the server returning the same data it was presented with, confirming that it has been saved in the database. Note that the `tax` field was not supplied, so the pydantic default value is used here. The second line shows the output from retrieving the newly stored item (mongoengine identifies the primary key using the name `_id`).

The third line shows an error message, generated by the attempt to store a non-numeric value in the price field.

Finally, the `retrieve_item` view function, routed from URLs such as */items/Item1/*, extracts the key as the second path element and returns the JSON representation of the given item. It looks up the given key in mongoengine and converts the returned record to a dictionary that is rendered as JSON by FastAPI.

Summary

While this chapter can only give a flavor of the many options for serving web content, we hope this is enough to demonstrate Python's suitability for what is becoming an increasingly complex and common task. From lightweight to heavyweight, Python remains a popular language among web developers.

---

1 One historical legacy is that, in CGI, a server provided the CGI script with information about the HTTP request to be served mostly via the operating system's environment (in Python, that's `os.environ`); to this day, interfaces between web servers and application frameworks rely on "an environment" that's essentially a dictionary and generalizes and speeds up the same fundamental idea.

2 even more advanced versions of HTTP exist, but we do not cover them in this book.

3 Please don't. As Titus Brown once pointed out, Python is (in)famous for having more web frameworks than keywords. One of this book's authors once showed Guido how to easily fix that problem when he was first designing Python 3—just add a few hundred new keywords—but, for some reason, Guido was not very receptive to this suggestion.

4 Installing uWSGI on Windows currently requires compiling it with cygwin.

5 Since Python has barely more than 30 keywords, you can see why Titus Brown once pointed out that Python has more web frameworks than keywords.

6 Using "127.0.0.1" only allows local apps to access the web page; to allow apps on other hosts to access it you could use "0.0.0.0", but we do **not** recommend this, since you might incur security risks.

Chapter 18. Email, MIME, and Other Network Encodings

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 21st 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*.

What travels on a network are streams of bytes, also known in networking jargon as *octets*. Bytes can, of course, represent text, via any of several possible encodings. However, what you want to send over the network often has more structure than just a stream of text or bytes. The Multipurpose Internet Mail Extensions (MIME) and other encoding standards bridge the gap, by specifying how to represent structured data as bytes or text. While often originally designed for email, such encodings are also used on the web and in many other networked systems. Python supports such encodings through many library modules, such as `base64`, `quopri`, and `uu` (covered in "Encoding Binary Data as ASCII Bytes"), and the modules of the `email` package (covered in "MIME and Email Format Handling"). These encodings allow us to seamlessly create messages in one encoding containing attachments in another, for example, avoiding many awkward tasks along the way.

MIME and Email Format Handling

The `email` package handles parsing, generation, and manipulation of MIME files such as email messages, Network News (NNTP) posts, HTTP interactions, and so on. The Python standard library also contains other modules that handle some parts of these jobs. However, the `email` package offers a complete and systematic approach to these important tasks. We suggest you use `email`, not the older modules that partially overlap with parts of `email`'s functionality. `email`, despite its name, has nothing to do with receiving or sending email; for such tasks, see the modules `imaplib`, `poplib` and `smtplib`, covered in "Email Protocols". Rather, `email` deals with handling MIME messages (which may or may not be mail) after you receive them, or constructing them properly before you send them.

Functions in the email Package

The `email` package supplies two factory functions that return an instance *m* of the class `email.message.Message` from a text string or file. These functions rely on the class `email.parser.Parser`, but the factory functions are handier and simpler. Therefore, we do not cover the `email.parser` module further in this book.

message_from_string	`message_from_string(`*s*`)` Builds *m* by parsing string *s*.
message_from_file	`message_from_file(`*f*`)` Builds *m* by parsing the contents of text file-like object *f*, which must be open for reading.

Two similar factory functions build message objects from bytestrings and binary files:

message_from_bytes	`message_from_bytes(`*s*`)` Builds *m* by parsing bytestring *s*.
	`message_from_binary_file(`*f*`)`

message_from_binary_file	Builds *m* by parsing the contents of binary file-like object *f*, which must be open for reading.

The email.message Module

The `email.message` module supplies the class `Message`. All parts of the `email` package make, modify, or use instances of `Message`. An instance *m* of `Message` models a MIME message, including *headers* and a *payload* (data content). To create an initially empty *m*, call `Message` with no arguments. More often, you create *m* by parsing via the factory functions `message_from_string` and `message_from_file` of `email`, their bytes and binary equivalents, or other indirect means such as the classes covered in "Creating Messages". *m*'s payload can be a string, a single other instance of `Message`, or (for a multipart message) a list of other `Message` instances.

You can set arbitrary headers on email messages you're building. Several Internet RFCs specify headers for a wide variety of purposes. The main applicable RFC is RFC 2822; you can find a summary of many other RFCs about headers in non-normative RFC 2076. An instance *m* of the `Message` class holds headers as well as a payload. *m* is a mapping, with header names as keys and header value strings as values.

To make *m* more convenient, its semantics as a mapping are different from those of a `dict`. *m*'s keys are case-insensitive. *m* keeps headers in the order in which you add them, and the methods `keys`, `values`, and `items` return lists (not views!) of headers in that order. *m* can have more than one header named *key*: *m*`[`*key*`]` returns an arbitrary such header (or `None` when the header is missing), and `del` *m*`[`*key*`]` deletes all of them (it's not an error if the header is missing).

To get a list of all headers with a certain name, call *m*`.get_all(`*key*`)`. `len(`*m*`)` returns the total number of headers, counting duplicates, not just the number of distinct header names. When there is no header named *key*, *m*`[`*key*`]` returns `None` and does not raise `KeyError` (i.e., behaves like

m.get(*key*)): del *m*[*key*] does nothing in this case, and *m*.get_all(*key*) returns an empty list. You can loop directly on *m*: it's just like looping on *m*.keys() instead.

An instance *m* of Message supplies attributes and methods that deal with *m*'s headers and payload:

add_header	*m*.add_header(*_name*,*_value*,***_params*) Like *m*[*_name*]=*_value*, but you can also supply header parameters as named arguments. For each named argument *pname=pvalue*, add_header changes any underscores in *pname* to dashes, then appends to the header's value a string of the form: ; *pname*="*pvalue*" When *pvalue* is None, add_header appends only a string '; *pname*'. When a parameter's value contains non-ASCII characters, specify it as a tuple with three items, (*CHARSET*, *LANGUAGE*, *VALUE*). *CHARSET* names the encoding to use for the value, *LANGUAGE* is usually None or '' but can be set any language value per RFC 2231; *VALUE* is the string value containing non-ASCII characters.
as_string	*m*.as_string(*unixfrom*=False) Returns the entire message as a string. When *unixfrom* is true, also includes a first line, normally starting with 'From ', known as the *envelope header* of the message.
attach	*m*.attach(*payload*) Adds *payload*, a message, to *m*'s payload. When *m*'s payload was None, *m*'s payload is now the single-item list [*payload*]. When *m*'s payload was a list of messages, appends *payload* to the list. When *m*'s payload was anything else, *m*.attach(*payload*) raises MultipartConversionError.
epilogue	The attribute *m*.epilogue can be None, or a string that becomes part of the message's string-form after the last boundary line. Mail programs normally don't display this text. epilogue is a normal attribute of *m*: your program can access it when you're handling any *m*, and bind it when you're building or modifying *m*.
get_all	*m*.get_all(*name*, *default*=None) Returns a list with all values of headers named *name* in the order in which the headers were added to *m*. When *m* has no header named *name*, get_all returns *default*.
get_boundary	*m*.get_boundary(*default*=None) Returns the string value of the boundary parameter of *m*'s Content-Type header. When *m* has no Content-Type header, or the header has no boundary parameter, get_boundary returns *default*.
	m.get_charsets(*default*=None)

get_charsets	Returns the list *L* of string values of parameter `charset` of *m*'s Content-Type headers. When *m* is multipart, *L* has one item per part; otherwise, *L* has length `1`. For parts that have no Content-Type, no `charset` parameter, or a main type different from `'text'`, the corresponding item in *L* is *default*.
get_content_maintype	*m*`.get_content_maintype(`*default*`=None)` Returns *m*'s main content type: a lowercase string `'`*maintype*`'` taken from header Content-Type. For example, when Content-Type is `'Text/Html'`, `get_content_maintype` returns `'text'`. When *m* has no header Content-Type, `get_content_maintype` returns *default*.
get_content_subtype	*m*`.get_content_subtype(`*default*`=None)` Returns *m*'s content subtype: a lowercase string `'`*subtype*`'` taken from header Content-Type. For example, when Content-Type is `'Text/Html'`, `get_content_subtype` returns `'html'`. When *m* has no header Content-Type, `get_content_subtype` returns *default*.
get_content_type	*m*`.get_content_type(`*default*`=None)` Returns *m*'s content type: a lowercase string `'`*maintype*`/`*subtype*`'` taken from header Content-Type. For example, when Content-Type is `'Text/Html'`, `get_content_type` returns `'text/html'`. When *m* has no header Content-Type, `get_content_type` returns *default*.
get_filename	*m*`.get_filename(`*default*`=None)` Returns the string value of the `filename` parameter of *m*'s Content-Disposition header. When *m* has no Content-Disposition, or the header has no `filename` parameter, `get_filename` returns *default*.
get_param	*m*`.get_param(`*param*`,`*default*`=None,`*header*`='Content-Type')` Returns the string value of parameter *param* of *m*'s header *header*. Returns `''` for a parameter specified just by name (without a value). When *m* has no header *header*, or the header has no parameter named *param*, `get_param` returns *default*.
get_params	*m*`.get_params(`*default*`=None,`*header*`='Content-Type')` Returns the parameters of *m*'s header *header*, a list of pairs of strings that give each parameter's name and value. Uses `''` as the value for parameters specified just by name (without a value). When *m* has no header *header*, `get_params` returns *default*.
get_payload	*m*`.get_payload(`*i*`=None,`*decode*`=False)` Returns *m*'s payload. When *m*`.is_multipart()` is `False`, *i* must be `None`, and *m*`.get_payload()` returns *m*'s entire payload, a string or `Message` instance. If *decode* is true and the value of header Content-Transfer-Encoding is either `'quoted-printable'` or `'base64'`, *m*`.get_payload` also decodes the payload. If *decode* is false, or header Content-Transfer-Encoding is missing or has other values, *m*`.get_payload` returns the payload unchanged. When *m*`.is_multipart()` is `True`, *decode* must be false. When *i* is `None`, *m*`.get_payload()` returns *m*'s payload as a list. Otherwise,

	m.get_payload(*i*) returns the *i*th item of the payload, or raises TypeError if *i*<0 or *i* is too large.
get_unixfrom	*m*.get_unixfrom() Returns the envelope header string for *m*, or None when *m* has no envelope header.
is_multipart	*m*.is_multipart() Returns True when *m*'s payload is a list; otherwise, False.
preamble	Attribute *m*.preamble can be None, or a string that becomes part of the message's string form before the first boundary line. A mail program shows this text only if it doesn't support multipart messages, so you can use this attribute to alert the user that your message is multipart and a different mail program is needed to view it. preamble is a normal attribute of *m*: your program can access it when you're handling an *m* that is built by whatever means, and bind, rebind, or unbind it when you're building or modifying *m*.
set_boundary	*m*.set_boundary(*boundary*) Sets the boundary parameter of *m*'s Content-Type header to *boundary*. When *m* has no Content-Type header, raises HeaderParseError.
set_payload	*m*.set_payload(*payload*) Sets *m*'s payload to *payload*, which must be a string, or a list of Message instances, as appropriate to *m*'s Content-Type.
set_unixfrom	*m*.set_unixfrom(*unixfrom*) Sets the envelope header string for *m*. *unixfrom* is the entire envelope header line, including the leading 'From ' but *not* including the trailing '\n'.
walk	*m*.walk() Returns an iterator on all parts and subparts of *m* to walk the tree of parts, depth-first (see "Recursion").

The email.Generator Module

The email.Generator module supplies the class Generator, which you can use to generate the textual form of a message *m*. *m*.as_string() and str(*m*) may be enough, but Generator gives more flexibility. Instantiate the Generator class with a mandatory argument and two optional arguments:

Generator	class Generator(*outfp*,*mangle_from_*=False,*maxheaderlen*=78) *outfp* is a file or file-like object that supplies method write. When *mangle_from_* is true, *g* prepends '>' to any line in the payload that starts with 'From ', in order to make the message's textual form easier to parse. *g* wraps each header line, at semicolons, into physical lines of no more than *maxheaderlen* characters. To use *g*, call *g*.flatten: *g*.flatten(*m, unixfrom*=**False**) This emits *m* as text to *outfp*, like (but consuming less memory than) *outfp*.write(*m*.as_string(*unixfrom*)).

Creating Messages

Subpackage email.mime supplies modules, each with a subclass of Message named like the module. The modules' names are lowercase (e.g., email.mime.text), while the class names are in mixed case.

These classes help you create Message instances of various MIME types. The classes are as follows:

MIMEAudio	class MIMEAudio(*_audiodata*, *_subtype*=None, *_encoder*=None, ***_params*) *_audiodata* is a bytestring of audio data to pack in a message of MIME type 'audio/*_subtype*'. When *_subtype* is None, *_audiodata* must be parseable by standard Python library module sndhdr to determine the subtype; otherwise, MIMEAudio raises TypeError. When *_encoder* is None, MlMEAudio encodes data as Base64, which is usually optimal. Otherwise, *_encoder* must be callable with one parameter *m*, which is the message being constructed; *_encoder* must then call *m*.get_payload() to get the payload, encode the payload, put the encoded form back by calling *m*.set_payload, and set *m*'s 'Content-Transfer-Encoding' header. MIMEAudio passes the *_params* dictionary of named-argument names and values to *m*.add_header to construct *m*'s Content-Type.
MIMEBase	class MIMEBase(*_maintype*,*_subtype*,***_params*) Base class of all MIME classes, extends Message. Instantiating: *m* = MIMEBase(*main,sub*,***parms*) is equivalent to the longer and slightly less convenient idiom: *m* = Message() *m*.add_header('Content-Type',f'{*main*}/{*sub*}',***parms*)*m*.add_header('Mime-Version','1.0')
MIMEImage	class MIMEImage(*_imagedata*,*_subtype*=None,*_encoder*=None, ***_params*)

	Like `MIMEAudio`, but with main type `'image'`; uses standard Python module `imghdr` to determine the subtype, if needed.
MIMEMessage	`class MIMEMessage(`*`msg`*`,`*`_subtype`*`='rfc822')` Packs *`msg`*, which must be an instance of `Message` (or a subclass), as the payload of a message of MIME type `'message/`*`_subtype`*`'`.
MIMEText	`class MIMEText(`*`_text`*`,`*`_subtype`*`='plain',`*`_charset`*`='us-ascii',`*`_encoder`*`=None)` Packs text string *`_text`* as the payload of a message of MIME type `'text/`*`_subtype`*`'` with the given `charset`. When *`_encoder`* is `None`, `MIMEText` does not encode the text, which is generally the best choice. Otherwise, *`_encoder`* must be callable with one parameter *`m`*, which is the message being constructed; *`_encoder`* must then call *`m`*`.get_payload()` to get the payload, encode the payload, put the encoded form back by calling *`m`*`.set_payload`, and set *`m`*'s `'Content-Transfer-Encoding'` appropriately.

The email.encoders Module

The `email.encoders` module supplies functions that take a *non-multipart* message *`m`* as their only argument, encode *`m`*'s payload, and set *`m`*'s headers appropriately.

encode_base64	`encode_base64(`*`m`*`)` Uses Base64 encoding, usually optimal for arbitrary binary data.
encode_noop	`encode_noop(`*`m`*`)` Does nothing to *`m`*'s payload and headers.
encode_quopri	`encode_quopri(`*`m`*`)` Uses Quoted Printable encoding, usually optimal for text that is almost but not fully ASCII (see "The quopri Module").
encode_7or8bit	`encode_7or8bit(`*`m`*`)` Does nothing to *`m`*'s payload, and sets header Content-Transfer-Encoding to `'8bit'` when any byte of *`m`*'s payload has the high bit set; otherwise, to `'7bit'`.

The email.utils Module

The `email.utils` module supplies several functions for email processing.

formataddr	`formataddr(`*`pair`*`)` *`pair`* is a pair of strings `(`*`realname,email_address`*`)`. `formataddr` returns a string *`s`* with the address to insert in header fields such as `To` and `Cc`. When *`realname`* is false (e.g., the empty string, `''`), `formataddr` returns *`email_address`*.
formatdate	`formatdate(`*`timeval`*`=None, `*`localtime`*`=False)` *`timeval`* is a number of seconds since the epoch. When *`timeval`* is `None`, `formatdate` uses the current time. When *`localtime`* is true, `formatdate` uses the local time zone; otherwise, it uses UTC. `formatdate` returns a string with the time instant formatted as specified by RFC 2822.
getaddresses	`getaddresses(`*`L`*`)` Parses each item of *`L`*, a list of address strings as used in header fields such as `To` and `Cc`, and returns a list of pairs of strings *`(name,address)`*. When `getaddresses` cannot parse an item of *`L`* as an email address, it sets `('','')` as the corresponding item in the list.
mktime_tz	`mktime_tz(`*`t`*`)` *`t`* is a tuple with 10 items. The first nine items of *`t`* are in the same format used in the module `time`, covered in "The time Module". *`t`*`[-1]` is a time zone as an offset in seconds from UTC (with the opposite sign from `time.timezone`, as specified by RFC 2822). When *`t`*`[-1]` is `None`, `mktime_tz` uses the local time zone. `mktime_tz` returns a float with the number of seconds since the epoch, in UTC, corresponding to the instant that *`t`* denotes.
parseaddr	`parseaddr(`*`s`*`)` Parses string *`s`*, which contains an address as typically specified in header fields such as `To` and `Cc`, and returns a pair of strings `(`*`realname,address`*`)`. When `parseaddr` cannot parse *`s`* as an address, it returns `('','')`.
parsedate	`parsedate(`*`s`*`)` Parses string *`s`* as per the rules in RFC 2822 and returns a tuple *`t`* with nine items, as used in the module `time`, covered in "The time Module" (the items *`t`*`[-3:]` are not meaningful). `parsedate` also attempts to parse some erroneous variations on RFC 2822 that widespread mailers use. When `parsedate` cannot parse *`s`*, it returns `None`.
parsedate_tz	`parsedate_tz(`*`s`*`)` Like `parsedate`, but returns a tuple *`t`* with 10 items, where *`t`*`[-1]` is *`s`*'s time zone as an offset in seconds from UTC (with the opposite sign from `time.timezone`, as specified by RFC 2822), like in the argument that

	`mktime_tz` accepts. Items *t*`[-4:-1]` are not meaningful. When *s* has no time zone, *t*`[-1]` is `None`.
quote	`quote(`*s*`)` Returns a copy of string *s*, where each double quote (") becomes `'\"'` and each existing backslash is repeated.
unquote	`unquote(`*s*`)` Returns a copy of string *s* where leading and trailing double-quote characters (") and angle brackets (<>) are removed if they surround the rest of *s*.

Example Uses of the email Package

The `email` package helps you both in reading and composing email and email-like messages (but it's not involved in receiving and transmitting such messages: those tasks belong to different and separate modules covered in Chapter "Client-Side Network Protocol Modules"). Here is an example of how to use `email` to read a possibly multipart message and unpack each part into a file in a given directory:

```
import os, email
def unpack_mail(mail_file, dest_dir):
    ''' Given file object mail_file, open for reading, and
dest_dir, a
        string that is a path to an existing, writable directory,
        unpack each part of the mail message from mail_file to a
        file within dest_dir.
    '''
    with mail_file:
        msg = email.message_from_file(mail_file)
    for part_number, part in enumerate(msg.walk()):
        if part.get_content_maintype() == 'multipart':
            # we get each specific part later in the loop,
            # so, nothing to do for the 'multipart' itself
            continue
        dest = part.get_filename()
        if dest is None: dest = part.get_param('name')
        if dest is None: dest = 'part-{}'.format(part_number)
        # In real life, make sure that dest is a reasonable
filename
        # for your OS; otherwise, mangle that name until it is
        with open(os.path.join(dest_dir, dest), 'wb') as f:
            f.write(part.get_payload(decode=True))
```

And here is an example that performs roughly the reverse task, packaging all files that are directly under a given source directory into a single file suitable for mailing:

```
def pack_mail(source_dir, **headers):
    ''' Given source_dir, a string that is a path to an
existing,
        readable directory, and arbitrary header name/value
pairs
        passed in as named arguments, packs all the files
directly
        under source_dir (assumed to be plain text files) into a
        mail message returned as a MIME-formatted string.
    '''
    msg = email.Message.Message()
    for name, value in headers.items():
        msg[name] = value
    msg['Content-type'] = 'multipart/mixed'
    filenames = next(os.walk(source_dir))[-1]
    for filename in filenames:
        m = email.Message.Message()
        m.add_header('Content-type', 'text/plain',
name=filename)
        with open(os.path.join(source_dir, filename), 'r') as f:
            m.set_payload(f.read())
        msg.attach(m)
    return msg.as_string()
```

Encoding Binary Data as ASCII Text

Several kinds of media (e.g., email messages) can contain only ASCII text. When you want to transmit arbitrary binary data via such media, you need to encode the data as ASCII text strings. The Python standard library supplies modules that support the standard encodings known as Base64, Quoted Printable, and UU.

The base64 Module

The `base64` module supports the encodings specified in RFC 3548 as Base16, Base32, and Base64. Each of these encodings is a compact way to represent arbitrary binary data as ASCII text, without any attempt to

produce human-readable results. `base64` supplies 10 functions: 6 for Base64, plus 2 each for Base32 and Base16. The six Base64 functions are:

b64decode	`b64decode(`*`s`*`,` *`altchars`*`=None,` *`validate`*`=False)` Decodes B64-encoded bytestring *`s`*, and returns the decoded bytestring. *`altchars`*, if not `None`, must be a bytestring of at least two characters (extra characters are ignored) specifying the two nonstandard characters to use instead of + and / (potentially useful to decode URL-safe or filesystem-safe B64-encoded strings). When *`validate`* is `True`, when *`s`* contains any bytes that are not valid in B64-encoded strings, the call raises an exception (by default, such bytes are just ignored and skipped). Also raises an exception when *`s`* is improperly padded according to the Base64 standard.
b64encode	`b64encode(`*`s`*`, altchars=None)` Encodes bytestring *`s`* and returns the bytestring with the corresponding B64-encoded data. *`altchars`*, if not `None`, must be a bytestring of at least two characters (extra characters are ignored) specifying the two nonstandard characters to use instead of + and / (potentially useful to make URL-safe or filesystem-safe B64-encoded strings).
standard_b64decode	`standard_b64decode(`*`s`*`)` Like `b64decode(s)`.
standard_b64encode	`standard_b64encode(`*`s`*`)` Like `b64encode(`*`s`*`)`.
urlsafe_b64decode	`urlsafe_b64decode(`*`s`*`)` Like `b64decode(`*`s`*`, '-_')`.
urlsafe_b64encode	`urlsafe_b64encode(`*`s`*`)` Like `b64encode(`*`s`*`, '-_')`.

The four Base16 and Base32 functions are:

`b16decode(`*`s`*`,`*`casefold`*`=False)`
Decodes B16-encoded bytestring *`s`*, and returns the decoded bytestring. When *`casefold`* is `True`, lowercase characters in *`s`* are treated like their

b16decode	uppercase equivalents; by default, when lowercase characters are present, the call raises an exception.
b16encode	`b16encode(`*`s`*`)` Encodes bytestring *`s`* and returns the byte string with the corresponding B16-encoded data.
b32decode	`b32decode(`*`s`*`, `*`casefold`*`=False, `*`map01`*`=None)` Decodes B32-encoded bytestring *`s`*, and returns the decoded bytestring. When *`casefold`* is `True`, lowercase characters in *`s`* are treated like their uppercase equivalents; by default, when lowercase characters are present, the call raises an exception. When *`map01`* is `None`, characters `0` and `1` are not allowed in the input; when not `None`, it must be a single-character bytestring specifying what `1` is mapped to (lowercase `'l'` or uppercase `'L'`); `0` is then always mapped to uppercase `'O'`.
b32encode	`b32encode(`*`s`*`)` Encodes bytestring *`s`* and returns the bytestring with the corresponding B32-encoded data.

The module also supplies functions to encode and decode the nonstandard but popular encodings Base85 and Ascii85, which, while not codified in RFCs nor compatible with each other, can offer space savings of 15% by using larger alphabets for encoded bytestrings. See the online docs for details on those functions.

The quopri Module

The `quopri` module supports the encoding specified in RFC 1521 as Quoted Printable (QP). QP can represent any binary data as ASCII text, but it's mainly intended for data that is mostly text, with a small amount of characters with the high bit set (i.e., characters outside the ASCII range). For such data, QP produces results that are both compact and human-readable. The `quopri` module supplies four functions:

decode	`decode(`*`infile,outfile,header`*`=False)` Reads the binary file-like object *`infile`* by calling *`infile`*`.readline` until end of file (i.e., until a call to *`infile`*`.readline` returns an empty string), decodes the QP-encoded ASCII text thus read, and writes the results to binary file-like object *`outfile`*`.` When *`header`* is true, `decode` also turns _ (underscores) into spaces (per RFC 1522).

decodestring	`decodestring(`*`s`*`,`*`header`*`=False)` Decodes bytestring *`s`*, QP-encoded ASCII text, and returns the bytestring with the decoded data. When *`header`* is true, `decodestring` also turns _ (underscores) into spaces.
encode	`encode(`*`infile`*`,`*`outfile`*`,`*`quotetabs`*`,`*`header`*`=False)` Reads binary file-like object *`infile`* by calling *`infile`*`.readline` until end of file (i.e., until a call to *`infile`*`.readline` returns an empty string), encodes the data thus read in QP, and writes the encoded ASCII text to binary file-like object *`outfile`*`.` When *`quotetabs`* is true, `encode` also encodes spaces and tabs. When *`header`* is true, `encode` encodes spaces as _ (underscores).
encodestring	`encodestring(`*`s`*`, `*`quotetabs`*`=False,`*`header`*`=False)` Encodes bytestring *`s`*, which contains arbitrary bytes, and returns a bytestring with QP-encoded ASCII text. When *`quotetabs`* is true, `encodestring` also encodes spaces and tabs. When *`header`* is true, `encodestring` encodes spaces as _ (underscores).

The uu Module

The `uu` module supports the classic Unix-to-Unix (UU) encoding, as implemented by the Unix programs *uuencode* and *uudecode*. UU starts encoded data with a `begin` line, which includes the filename and permissions of the file being encoded, and ends it with an `end` line. Therefore, UU encoding lets you embed encoded data in otherwise unstructured text, while Base64 encoding relies on the existence of other indications of where the encoded data starts and finishes. The `uu` module supplies two functions:

decode	`decode(`*`infile`*`,`*`outfile`*`=None,`*`mode`*`=None)` Reads the file-like object *`infile`* by calling *`infile`*`.readline` until end of file (i.e., until a call to *`infile`*`.readline` returns an empty string) or until a terminator line (the string `'end'` surrounded by any amount of whitespace). `decode` decodes the UU-encoded text thus read and writes the decoded data to the file-like object *`outfile`*`.` When *`outfile`* is `None`, `decode` creates the file specified in the UU-format `begin` line, with the permission bits given by *`mode`* (the permission bits specified in the `begin` line, when *`mode`* is `None`). In this case, `decode` raises an exception if the file already exists.
encode	`encode(`*`infile`*`,`*`outfile`*`,`*`name`*`='-',`*`mode`*`=0o666)` Reads the file-like object *`infile`* by calling *`infile`*`.read` (45 bytes at a time, which is the amount of data that UU encodes into 60 characters in each output line) until end of file (i.e., until a call to *`infile`*`.read` returns an

	empty string). It encodes the data thus read in UU and writes the encoded text to file-like object *`outfile`*`.encode` also writes a UU `begin` line before the text and a UU `end` line after the text. In the `begin` line, `encode` specifies the filename as *`name`* and the mode as *`mode`*.

Chapter 19. 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 `'ä'`.

`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 20. 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 20-1. 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 `é`. 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 20-2. 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 Senior Site Reliability Engineer, helping people get jobs!