Learning Python 6th Edition PDF Download

Visit the link below to download the full version of the ebook

DOWNLOAD NOW

Scan to Download
or Type the Link

ebook.ac/learning6e

O'REILLY®

6th Edition

Learning Python

Powerful Object-Oriented Programming

Mark Lutz

Learning Python

SIXTH EDITION

Powerful Object-Oriented Programming

Mark Lutz

O'REILLY®

Learning Python

by Mark Lutz

Copyright © 2025 Mark Lutz. All rights reserved.

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*.

Acquisitions Editor: Louise Corrigan

Development Editor: Sara Hunter

Production Editor: Kristen Brown

Copyeditor: nSight, Inc.

Proofreader: Piper Content Partners

Indexer: nSight, Inc.

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

March 2025: Sixth Edition

Revision History for the Sixth Edition

- 2025-02-25: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098171308* for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Learning Python*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher’s views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-17130-8

[LSI]

[Dedication]

To Vera.

You are my life.

Preface

If you're browsing a bookstore and trying to make sense of this book, try this:

- *Python* is one of the most widely used programming languages in the world. It's part of nearly every role that computers play in our lives, and its relative ease of use makes it an ideal way to get started with programming.
- *This book* is a tutorial that teaches Python language fundamentals in depth. Its content is aimed at Python newcomers of all stripes, applies to every role that Python plays, and is based on decades of feedback from real learners like you.
- *This edition* updates this book for a decade of changes in Python and its world. It drops coverage of the now-defunct Python 2.X, explores new tools added to Python through version 3.12, and applies to other Pythons past and future.

The rest of this preface provides more background info on this book and its subject. It explains what's changed since the prior edition, debuts the book's examples package, and may help you get oriented before jumping into details.

Python

By most metrics you'll find on the web today, Python is now either the most-used programming language on the planet or very near the top of the list. The oft-cited TIOBE popularity index, for example, has ranked Python most popular for several years. As this edition is being written in 2024, it lists Python at #1 and well ahead of its nearest followers, C, C++, and Java.

Popularity ranks are prone to change, of course, and rely on usage metrics that are open to debate that we'll skip here. Based on every signal available, though, the Python revolution has clearly happened. It's now a ubiquitous, *go-to language* in web development, scientific programming, systems administration, AI, and nearly everything else computers do today. Thanks to its relative simplicity, Python is also commonly used to introduce newcomers to computer science across the education spectrum.

In fact, it's now fairly safe to say that Python played a pivotal role in *changing the world*. By spearheading a shift from statically typed compiled languages to dynamically typed scripting languages, Python ushered in changes that were at least as profound as those of the earlier transition from machine language to compiled languages. The scripting-language shift both enabled tasks formerly impractical, and opened the field to nonprofessional contributors. In the process, it propelled computers to a prominence that would have been unthinkable in decades past. The internet, for example, simply could not be what it is today without tools like Python. For better and worse, Python enables the new.

Lofty goals aside, all of this has two tangible implications for this book. First, because this is now a post-revolution Python world, this edition does much less *cheerleading* than its predecessors. There's no reason to waste your time promoting a tool that's already arrived. This edition still summarizes Python's value proposition in Chapter 1, but the domains and tools that you're likely to explore after learning the basics here are readily available on the web and change too regularly to cover in a fundamentals book like this in any event.

Second, Python's popularity means that by reading this book, you'll be adding a *valuable skill* to your toolset, which will help you in a wide variety of computer-software tasks. Learning Python will both make entire domains accessible to you and enable you to achieve programming goals that might otherwise be difficult or impossible. Python users now enjoy a wealth of prior art ready to be leveraged in a language that accelerates their work.

That said, Python isn't the only game out there, and you'd also be well served by learning computer science from the ground up—the *full stack* in developer speak. Studying lower-level languages like C and Java, for example, can still give you a much more complete perspective than scripting languages alone and help you solve complex problems as they arise. Python itself, after all, is just a C program in its most-used flavor.

Even so, Python is a great place to start, and enough for many a task. While it's not without warts and suffers from the thrashing that's endemic to software today, a multitude of developers still find Python a lot more fun to use than other tools. Java and C++, for example, seem languages designed for middle management: they hobble programmers with training wheels and bureaucratic hurdles that have little to do with your program's goals. Python, in sharp contrast, remains *more ally than obstacle*. That viewpoint is naturally subjective, but you've come to the right place to do the math on this yourself.

This Book

This book is a tutorial on the Python language and a classic in its domain. It's the product of *three decades* spent using, promoting, and teaching Python, and dates back to the mid-1990s, when Python was still at version 1.X, and the web was just something developers mused about over lunch. Although the focus here is firmly on the present, that legacy naturally adds some historical context that will help you understand Python more deeply. Despite what you may have heard, the past matters, especially in knowledge-based fields.

Just as importantly, this book has always been based on live-and-in-person *feedback* from Python beginners struggling to learn Python for the first time. This feedback mostly owes to Python training classes taught over a period of two decades. While these classes have now gone the way of the dodo and Yahoo, this book takes care to retain its learner-inspired material because that's much—if not most—of its value.

As a result, if you're like most of the thousands of learners whose experiences have been captured here, you'll probably find that this book works like a self-paced version of the Python training sessions from which it arose. You may sometimes even find that it answers your questions before they are asked because a host of learners before you have had the same queries. This isn't clairvoyance; it simply reflects the fact that learning resources do best when they listen to learners.

It's also worth noting up front that this book sometimes *critiques* Python changes while presenting them. Critical thinking is crucial in engineering domains—especially in one caught up in an arms race that convolutes tools used by millions of people. On some levels, Python remains a constantly morphing sandbox of ideas that too often prioritizes changer hubris over user need, and this book is not shy about calling this out. That said, the main goal here is to educate, not criticize, and opinions are always, well, opinionated. Although views here reflect decades of using and teaching Python, you should always judge the net worth of Python changes for yourself in whatever world you've been cast.

This Edition

This edition completely drops coverage of *Python 2.X*, the earlier version of the language, and adds new coverage of recent changes in *Python 3.X*, the newer and incompatible version. When the prior edition was published in 2013, Python 2.X was still widely used and probably even dominant. Because of that, the prior edition had to cover both the established 2.X and the new and upcoming 3.X, which at times made for a twisted tale indeed.

Over a decade later, 2.X has been officially sunsetted, and the Python world has adopted 3.X so fully that 2.X constitutes an unwarranted distraction to today's Python learners. Hence, after a decades-long tenure, 2.X-specific content has been cut here to make room for new 3.X topics and address book size in general. Formally, this edition has been updated to be current with *Python 3.12* and its era, though it also previews 3.13 mods, and its

focus on fundamentals makes it generally applicable to both older and newer Python versions.

Before the emails start flooding in, this book wants to make clear that it regrets the loss of historical context (and secretly pines for the simpler days of 2.X too). But 3.X is a substantial topic all by itself, without bifurcating the story and increasing the page count for a Python version that is now little used. So go well into that good Python night, 2.X, and long live 3.X. Unless stated otherwise, "Python" in this edition simply means the 3.X line in general and 3.12 and later in particular.

In terms of *3.X mods*, this edition newly covers `f'…'` f-string literals, `:=` named-assignment expressions, `match` statements, type hinting, `async` coroutines, star-unpacking proliferation, underscore digit separators, *__main__.py* package files, `__getattr__` module hooks, `except*` exception groups, dictionary-key insertion order, positional-only function arguments, hash-based bytecode files, and other additions, deprecations, and mutations that have cropped up over the last decade plus.

Among these, type hinting and `async` coroutines are not covered in depth—by design. The former is an optional and academic tool wholly unused by Python itself and at odds with its core principles. The latter is an advanced applications tool and has morphed constantly since its inception. And both quickly head over complexity cliffs that push them out of scope for Python beginners. When needed, supplemental info on such narrow topics is always just a search away on the web. Here, the goal is learning to walk well before trying to run.

Among other noteworthy changes this time around:

- The *Unicode* content in the advanced part's Chapter 37 is new and improved because this topic is now an essential in Python 3.X and the world at large.
- Usage coverage, including the new Appendix A, gives more focus to *macOS*, *Android*, *Linux*, and *iOS* because not all of this book's readers use Windows.

- Most code-file examples now have numbered *captions* because the extra formality distinguishes them better in the book, and it's worth the space.
- Some *redundancy* has been trimmed, but not all, because repetition is useful and even important in learning resources.
- The *size* of this book was reduced by the prior bullet, rewrites and flow mods, and the net of 2.X cuts and 3.X inserts because it's less to grok.
- The size of the *print* version of this book was further reduced by moving two advanced but optional chapters online (Chapters 38 and 39) because it's less to lug.
- Fictitious *names* in examples are more gender neutral: "Bob" is now an ambiguous "Pat" unless paired with "Sue" as before because it better defuses bias.
- The *Monty Python* references have been dropped because they can be confusing and might be divisive, and borrowing personality from media seems cheap.
- Both *first-person voice* and *personal anecdotes* have been globally sacked because you've bought this book to learn Python, not an author's life story.

About the last two: Python's namesake was funny stuff, to be sure, but compulsively aping the work of a nearly all-male comedy group can seem like the secret handshake of an exclusive boys' club in hindsight. And while an occasional "I" or "my" might add color or credibility, overuse tends to come off as narcissism. Hence, the former 1k "spam" are now symbols more inclusive, and this book's three-decade tenure will have to speak for itself.

Despite all the mods, this edition remains much more *technical novel* than reference manual, and meaty enough to be comparable to a *full-semester class* on Python and programming. It introduces topics and expands them in

later chapters as recurring themes, accumulating comprehensive coverage along the way. There are Python quick-reference resources at *python.org* and a multitude of blogs and videos that promise to teach Python rapidly. This book is for those who know that learning something well requires a bit more.

Media Choices

As of this writing, this book is destined to be available in three forms: *print* (i.e., paper), *ebook* (e.g., PDF, ePub, and Kindle), and *online* (a.k.a. web). The latter means the publisher's subscription service, currently branded as the O'Reilly learning platform (f.k.a. Safari). Naturally, each medium has valid uses that vary per reader. For instance, many prefer print for linear reads and electronic media for random searches and code copy/paste.

You're welcome to use the forms that work best for you, of course, but should carefully weigh the inherent *privacy trade-offs* of online media. By now, it should be abundantly clear that online anything comes with a potential for covert use and sale of customer information and access, which print and ebook media largely avoid. While monetization schemes vary, online users just might have a legion of salespeople peering over their shoulders as they use products for which they've already paid in full.

So please be careful out there. Unless you must use an employer's online subscription, this book suggests vetting your media options wisely and generally recommends its *print and ebook* forms to protect your privacy whenever possible. Your life really shouldn't be turned into a revenue stream unless you get reimbursed for it.

One last media tip: this book may also be fed by its publisher into *generative AI* models, the current hot topic in the tech press. Although this may prove useful for looking up isolated facts, it's not deep learning and isn't necessarily any more reliable than the least of the gossip it regurgitates. Until the world figures this out, please use wisely.

Updates and Examples

As most authors would attest, it's shockingly easy to miss typos in material you've read hundreds of times, and incompatible change is a norm in computer book topics. Hence, this edition, like all its predecessors, expects to be updated regularly after its publication.

This book's supplements, example files, and clarifications and corrections (a.k.a. errata) will all be maintained on the web. Here are the main coordinates for these online resources; as usual, consult your local search engine if these change over time:

Author website

> This site will be used to post general *notes and updates* related to this text or Python itself—a hedge against future changes and a sort of virtual appendix to this book.

Book examples

> This site will host this book's *examples package*, with both individual files to view and save online and a ZIP file of all the examples to download to your device.

Publisher website

> This site will maintain this edition's *errata list*, and chronicle patches applied to the text in reprints. It will also link to other formats, including ebooks.

The second of these is home to the examples' *source code*. Please see the usage notes there, as well as the examples' coverage in "Where to Run: Code Folders". There is no plan to host the examples on *GitHub* today, because that site's learning curve is a lot to ask of beginners, and its commercial agendas should be cause for concern.

At any of the preceding websites, all *error reports* and *suggestions* for this book are welcome, and this feedback is invaluable for book quality. But please keep it fact-based and civil. Posts on the errata list have been mostly constructive, but the list has limited utility, and has been known to attract the usual trolls. Such is life in the age of global conversation.

Conventions and Reuse

This book's mechanics will make more sense once you start reading it, but as a reference, this book uses the following typographical conventions:

Italic

Used for email addresses, URLs, filenames, pathnames, and emphasizing new terms when they are first introduced

`Constant width`

Used for program code, the contents of files and the output from commands, and to designate modules, methods, statements, and system commands

`Constant width bold`

Used in code sections to show commands or text that would be typed by the user, and, occasionally, to highlight portions of code

`Constant width italic`

Used for replaceables (content you must fill in) and some comments in code sections

> **NOTE**
>
> This element indicates a tip, suggestion, or general note relating to the nearby text. The icon may make more sense if you imagine a crow sounding an alarm.

Three more quick content notes here: first, you'll find occasional *sidebars* (delimited by boxes) and *footnotes* throughout, which are often optional reading but provide additional context on the topics being presented. For instance, the sidebars that begin with "Why You Will Care:" amplify language topics with real-world use cases.

Second, Python *error messages* are often shortened in this book to conserve space. Please run offending code on your own device for the full text. Some messages include stack traces, and some have sprouted location indicators and speculative "Did you..?" help in the latest Python that might be useful for beginners, though veterans' mileage may vary; either way, the extra text is excessive in a book tight on space.

Finally, the publisher maintains a standard statement about reusing code in this book, though the short, interactive code snippets used broadly here are hardly worth the legalese. Please see the book websites described earlier for the formal reuse story if you must care.

Acknowledgments

In keeping with the depersonalization goal discussed earlier, this edition will forego the usual lengthy acknowledgments of its predecessors. Instead, it extends simple gratitude to the scores of former students and readers, who largely shaped this book; the publishing company, which enabled this book to both reach learners and improve with time; the host of users, contributors, and promoters, who made Python what it is today; and the subject of this book's dedication page, who patiently tolerated yet another book project. This one might be the last, but you never know what a bored author might next do.

Part I. Getting Started

Chapter 1. A Python Q&A Session

If you're reading this book, you may already know what Python is and why it's an important tool to learn. If you don't, you probably won't be sold on Python until you've learned the language by reading the rest of this book and have done a project or two. But before we jump into details, this first chapter will briefly introduce some of the main reasons behind Python's popularity and begin sculpting a definition of the language. This takes the form of a question-and-answer session, which addresses some of the most common queries posed by beginners—like you.

Why Do People Use Python?

Because there are many programming languages to choose from, this is the usual first question of newcomers and a great place to start. Given that millions of people use Python today, there really is no way to answer this question with complete accuracy; the choice of development tools is often based on unique constraints or personal preference.

But after teaching Python to hundreds of groups and thousands of students, some common themes have emerged. The primary factors cited by Python users seem to be these:

Software quality

For many, Python's focus on readability, coherence, and software quality in general sets it apart from other tools in the programing world. Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity

of Python code makes it relatively easy to understand, even if you did not write it. In addition, Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and functional programming, that can further promote code quality.

Developer productivity

Python boosts developer productivity many times beyond compiled or statically typed languages. As one measure of this, Python code is typically *one-third to one-fifth* the size of equivalent C++ or Java code. That means there is less to type, less to debug, and less to maintain after the fact. Most Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

Program portability

Python programs generally run unchanged on *all major computer platforms*. Porting a Python program between Linux and Windows, for example, is often just a matter of copying its code between machines. Moreover, Python offers multiple options for coding portable graphical user interfaces, database access programs, web-based systems, and more. Even operating system interfaces as proprietary as program launches and directory processing are as portable in Python as they can possibly be.

Application support

Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both homegrown libraries and a vast collection of third-party software. As covered ahead, Python's *third-party domain* offers tools for website construction, numeric programming, AI, and much more. The NumPy extension, for instance, has elevated Python to a core tool in science, technology, engineering, and math (*STEM*), and Django and PyTorch have done similar for the web and AI.

Component integration

Python scripts can communicate with other parts of an application using a variety of integration mechanisms. Such integrations allow Python to be used as a product *customization and extension* tool. For instance, Python code can invoke compiled libraries and be run by compiled programs, interact with other components over networks, and use Android and iOS toolkits on smartphones. In fact, many Python core tools, including files, ultimately use precoded interfaces to system libraries; even if your program is all Python, it's not standalone.

Love of craft

> Because of Python’s ease of use and built-in toolset, it can make the act of programming *more pleasure than chore*. Although this benefit is intangible and subjective, its effect on productivity is an important asset. People do get paid for coding Python, but many use it just for *fun*—a testimonial rare in the software field.

Of these factors, the first two—quality and productivity—are probably the most compelling benefits to most Python users; let’s take a closer look at each.

Software Quality

By design, Python has a deliberately simple and readable syntax and a highly consistent programming model. As a slogan at an early Python conference attested, the net result is that Python seems to *fit your brain*—that is, features of the language interact in consistent and limited ways and follow naturally from a small set of core concepts. This makes the language easier to learn, understand, and remember. In practice, Python programmers do not need to constantly refer to manuals when reading or writing code; it generally just makes sense.

By philosophy, Python adopts a somewhat minimalist mindset. This means that although there are usually multiple ways to accomplish a coding task, there is usually just one obvious way, a few less obvious alternatives, and a small set of coherent interactions throughout. Moreover, Python usually doesn’t make arbitrary decisions for you; when interactions are ambiguous, explicit intervention is preferred over “magic.” In the Python way of thinking, explicit is better than implicit, and simple is better than complex.

Beyond such design themes, Python includes tools such as modules and object-oriented programming (OOP) that naturally promote code reusability in skilled hands of the sort you’re about to acquire. And because Python is

focused on quality, most Python programmers naturally are too; this can be a crucial advantage when it's time to use someone else's Python code.

Developer Productivity

If you've worked in the software field, you know that it can be a dynamic and bumpy ride. During the great internet boom of the mid-to-late 1990s, it was difficult to find enough programmers to implement software projects; developers were asked to implement systems as fast as the internet evolved. In later eras of economic recession and layoffs, the picture shifted; programming staffs were often asked to accomplish the same tasks with even fewer people.

In both of these scenarios, Python has shined as a tool that allows programmers to get more done with less effort. It is explicitly optimized for *speed of development*—its simple syntax, dynamic typing, lack of compile steps, and built-in toolset allow programmers to develop programs in a fraction of the time needed when using some other tools.

The net effect is that Python typically increases developer productivity many times beyond the levels supported by traditional languages and often makes the impossible possible. That's good news in both boom and bust times, and everywhere the software industry goes in between.

Is Python a "Scripting Language"?

Maybe. Python is often applied in scripting roles, but not always. It's regularly called an *object-oriented scripting language*—a definition that blends support for OOP with an orientation toward scripting contexts, but this may be too narrow and dismissive. If pressed for a one-liner, Python is probably better known as:

> *A general-purpose programming language that blends procedural, functional, and object-oriented paradigms and accelerates software development by reducing complexity.*

That may not fit on a t-shirt quite as well, but it captures both the richness and scope of today’s Python.

Nevertheless, the term *scripting* seems to have stuck to Python like glue, perhaps as a contrast with the larger efforts required by some other tools. For example, some people use the word “script” instead of “program” to describe a Python code file, because it seems simpler and less formal to code. More usefully, others reserve “script” for a top-level file, and “program” for a more sophisticated multifile application, both of which are common in Python.

Because the term *scripting language* has so many different meanings to different observers, though, some would prefer that it not be applied to Python at all. In fact, people tend to make three very different assumptions when they hear Python labeled as such, some of which are more useful than others:

Shell tools

> Sometimes when people hear Python described as a scripting language, they think it means that Python is a tool for coding operating-system-oriented scripts. Such programs are often launched from console command lines and perform tasks such as processing text files and launching other programs.
>
> As you’ll learn ahead, Python programs can and do serve such roles, but this is just one of dozens of common Python application domains. It is not just a better shell-script language.

Control language

> To others, scripting refers to a “glue” layer used to control and direct (i.e., script) other application components. As noted earlier, Python programs are indeed often deployed in the context of larger applications. For instance, to test hardware devices, Python programs may call out to

components that give low-level access to a device or port. Similarly, programs may run bits of Python code at strategic points to support end-user product customization without the need to ship and recompile the entire system's source code.

Python's simplicity makes it a naturally flexible control tool. Technically, though, this is also just a common Python role; many (and perhaps most) Python programmers code standalone scripts without ever using or knowing about any integrated components. It is not just a control language.

Ease of use

Probably the best use of the term *scripting language* is to refer to a relatively simple language used for coding tasks quickly. This is especially true when the term is applied to Python, which allows much faster program development than compiled languages like C++ or larger languages like Java. Its rapid development cycle fosters an exploratory, incremental mode of programming that has to be experienced to be appreciated.

Don't be fooled, though—Python is not just for simple tasks. Rather, it makes tasks simple by its ease of use and flexibility. Python has an approachable feature set, but it allows programs to scale up in sophistication as needed. Because of that, it is commonly used for both quick tactical tasks and longer-term strategic development.

So, is Python a scripting language or not? It depends on whom you ask (and perhaps when you ask them). In general, the term *scripting* is best reserved for the rapid and flexible mode of development that Python supports, rather than a particular application domain or limiting label.

On a related note, people also sometimes call Python an *interpreted language* to distinguish it from languages like C and C++. While this is also

sometimes meant to pigeonhole, it's also easier to dismiss: there are many implementations of Python today, spanning the spectrum from traditional interpreters to traditional compilers, so "interpreted" doesn't apply. The clearer distinction may be that Python is *dynamically typed*, not statically typed like languages that are normally compiled. As you'll soon learn, this accounts for much of the power that Python brings to development tasks.

OK, but What's the Downside?

The only universally recognized downside to Python that has emerged over its more than three-decade tenure may also be an inevitable trade-off for its ease of use: Python's *execution speed* may not always be as fast as that of fully compiled and lower-level languages such as C and C++. Though relatively rare today, you may still occasionally need to get "closer to the iron" for some tasks by using languages that are more directly mapped to the underlying hardware.

We'll talk about implementation concepts in Chapter 2, but in short, the most-used versions of Python today compile (i.e., translate) source code statements to an intermediate format known as *bytecode* and then interpret the bytecode. Bytecode provides portability, as it is a platform-independent format. However, because Python is not commonly compiled all the way down to binary *machine code* (e.g., instructions for an Intel or ARM chip in your PC or phone), some programs will run more slowly in Python than in a fully compiled language like C.

Whether you will ever *care* about this execution speed difference depends on what kinds of programs you write. Python is regularly optimized, and Python code runs fast enough by itself in most application domains. Furthermore, whenever you do something "real" in a Python script, like processing a file or constructing a graphical user interface (*GUI*), your program will actually run at C speed, because such tasks are immediately dispatched to compiled C code inside the Python interpreter.

And really, Python's speed is a bit of a red herring today: in truth, Python *is* commonly used in domains that require optimal execution speeds. Numeric

programming and computer games, for example, often need at least their core number-crunching components to run at C speed (or better), but are frequently coded in Python nevertheless.

There are multiple ways to achieve speed in Python when it counts. For instance, systems such as *PyPy* compile bytecode further as your program runs; *Cython* allows you to code in a C-and-Python hybrid that's compiled in full; and crucial code can be split off to *compiled extensions* linked into Python for use in scripts. As an example, the *NumPy* numeric-programming extension combines compiled and optimized libraries with the Python language, and in the process turns Python into a numeric programming tool that is simultaneously efficient and easy to use. Indeed, NumPy Python code regularly achieves speed parity with Fortran and C++, without their added complexities.

More fundamentally, though, execution speed is not the only priority in most software development. Python's *speed-of-development* gain is often far more important than any speed-of-execution loss, especially given modern computer speeds—and modern computer deadlines. Naturally, Python has other *potential* downsides, including its frequent changes and convolutions, but these are subjective calls best made after you've had a chance to vet them in this book.

Who Uses Python Today?

Because Python is a free and open source software (*FOSS*) tool, an accurate count of its user base is impossible—there are no license registrations to tally. Moreover, Python has been automatically included with countless Linux distributions, Macintosh computers, and a wide range of products and hardware, further clouding the user-base picture.

In general, though, Python enjoys a very large user base and an active developer community. It is generally considered to be among the *top 5* most widely used programming languages in the world today (and for true sports fans, often weighs in at #1). Because Python has been broadly used for *over three decades*, it's also very stable and robust.

Because of all this, Python is regularly applied in real revenue-generating products by real *companies*. While user lists are prone to change, a list of notable and known companies using Python today reads like a who's who of the software field: Google, Intel, Disney, YouTube, Industrial Light & Magic, Red Hat, NASA, Eve Online, Seagate, JPL, Hewlett-Packard, JP Morgan Chase, Dropbox, ESRI, Instagram, Spotify, Pinterest, Reddit, Microsoft, Netflix, and many more.

More broadly, Python is deployed by most organizations developing software today in either strategic or tactical roles and regularly serves as the tool of choice in computer science education. It's not just for one thing, it's for everything.

For a sampling of additional Python users and applications that's hopefully more up to date than a book can ever be, try the following pages at Python's site: Python Success Stories, Applications for Python, and Quotes about Python. As usual, you may also be able to uncover other Python roles of interest with a web search in your local browser or app.

What Can I Do with Python?

Commercial applications may be compelling, but people also use Python for all sorts of real-world, day-in/day-out tasks, whether for profit, need, hobby, or fun. In fact, as a general-purpose language, Python's roles are virtually unlimited: you can use it for everything from gaming and website development to robotics and content management.

That said, Python roles seem to fall into a few broad categories. The next few sections survey some of Python's most common application domains today, as well as prominent tools used in each domain.

Two notes up front: first, we won't be able to explore these tools in any depth either here or in this book at large. NumPy and Django, for example, are book-length topics on their own, and our goal in this book is to learn the *Python* code you will be writing to use systems like these. Second, apologies in advance to the many other tools omitted here only for space; if

you are interested in any of the following topics, please search online for more tools and resources.

Systems Programming

Python's built-in interfaces to operating-system services make it ideal for writing portable and maintainable system-administration utilities—sometimes called *shell* or *command-line* tools, though they may be used in numerous ways. Python programs can search files and directory trees, launch and configure other programs, do parallel processing with processes, threads, and coroutines, and more.

Python's standard library comes with Portable Operating System Interface (*POSIX)* bindings and support for all the usual OS tools, including environment variables, files, sockets, pipes, processes, threads, regular-expression pattern matching, command-line arguments, standard-stream interfaces, shell-command launchers, filename expansion, ZIP file utilities, and XML, JSON, and CSV parsers. In addition, the bulk of Python's system interfaces are designed to be portable; for example, a script that copies directory trees typically runs unchanged on all platforms that host Python.

GUIs and UIs

Python's simplicity and rapid turnaround also make it a good match for GUI programming on devices of all kinds. For instance, Python comes with a standard object-oriented interface to the Tk GUI toolkit called *Tkinter* (and `tkinter` in code), which allows Python programs to implement portable GUIs with a native look and feel. Python/Tkinter GUIs run unchanged on Windows, macOS, and Linux, and on Android with the help of a freeware app today (see Appendix A).

In addition, third-party tools offer other routes to portable UIs—including both traditional GUIs like *Kivy*, *BeeWare*'s *Toga*, *PyQt*, and *wxPython*; and web-browser based solutions like *Django*, *Flask*, and *WebAssembly*. If your app, like most, must interact with users, Python has multiple ways to write once and run everywhere.

Internet and Web Scripting

Python comes with standard internet modules that allow programs to perform a wide variety of networking tasks, in both client and server modes. Scripts can communicate over sockets; extract form information sent to server-side CGI web scripts; transfer files by FTP; parse and generate XML and JSON documents; compose and send, and receive and parse email; fetch web pages by URLs and parse their HTML; and more.

In addition, a large collection of third-party tools are available on the web for doing internet programming in Python. For example, web-development frameworks, such as *Django*, *Flask*, *TurboGears*, and *Zope*, support construction of full-featured and production-quality websites with Python. Many of these include tools like object-relational mappers, server-side scripting and templating, and AJAX support, to provide complete and enterprise-level web development solutions. The *Pyjamas* Python-to-JavaScript transpiler; the *Beautiful Soup* HTML content extractor; and the *WebAssembly*, *Pyodide*, *py2wasm*, and *PyScript* Python-in-the-browser enablers provide even more web possibilities.

Component Integration

We discussed the component integration role earlier. Python's ability to be extended by and embedded in systems coded in C, C++, and Java makes it useful as a flexible glue language for scripting the behavior of other software components. For instance, integrating a C library into Python allows Python to test and launch the library's tools, and embedding Python in a product enables on-site customizations to be coded without having to recompile the entire product (or ship its source code at all).

Tools such as *SWIG*, *Boost.Python*, *CFFI*, and *HPy* automate much of the work needed to link compiled components with Python scripts, and the *Cython* system allows coders to mix Python and C-like code to create compiled extensions. Other tools provide more ways to script components —including the *pyjnius* and *Chaquopy* Python/Java bridges; *pywin32*'s support for Windows Component Object Model (COM); the *REST*, *SOAP*,

and *XML-RPC* cross-network conduits; and the *Jython* Java and *IronPython* .NET implementations of Python. In short, most software components are in scope to Python code.

Database Access

For traditional database demands, there are Python interfaces to all commonly used relational database systems—Oracle, MySQL, PostgreSQL, Informix, ODBC, SQLite, and more. The Python world has also defined a *portable database API* for accessing SQL database systems from Python scripts, which looks the same on a variety of underlying database implementations. For basic program-storage needs, Python comes with built-in support for its own *pickle* objects, language-agnostic *JSON* documents, and the in-process *SQLite* embedded SQL database engine.

Also for Python scripts, *PyYAML* parses and emits YAML data; *ZODB* and *Durus* provide object-oriented databases; *SQLObject* and *SQLAlchemy* implement object relational mappers (ORMs), which graft Python classes onto relational tables; and *PyMongo* interfaces to MongoDB, a high-performance JSON-style document database. Python can also access cloud storage options in Google's *App Engine*, Microsoft's *Azure*, and Amazon's *AWS*.

Rapid Prototyping

To Python programs, components written in Python and C look the same. Because of this, it's possible to prototype systems in Python initially, and then move selected components to a fully compiled language such as C or C++ for delivery. Because Python doesn't require a complete rewrite once the prototype has solidified, the parts of the system that don't need the efficiency of a compiled language can remain coded in Python for ease of maintenance and use. But beware: given the many optimization routes you met earlier, your prototype may very well be your deliverable.

Numeric and Scientific Programming

Python is also widely used in numeric programming—a domain that would not traditionally have been considered to be in the scope of scripting languages but has grown to become one of Python's most compelling use cases. Prominent here, the *NumPy* high-performance numeric-programming extension for Python mentioned earlier includes such advanced tools as an array object, interfaces to standard mathematical libraries, and much more. By integrating Python with numeric routines coded in a compiled language for speed, NumPy turns Python into a sophisticated yet easy-to-use numeric-programming tool that can often replace code written in traditional languages like Fortran or C++.

Additional numeric tools often use NumPy as a core component, and add support for visualization, parallel processing, statistical analysis, and more. For example, *SciPy* provides libraries of scientific programming tools; *pandas* supports data analysis; *matplotlib* adds visualization tools; *Jupyter* notebooks are geared toward math workers' needs; *Numba* and *PyThran* offer just-in-time (JIT) and ahead-of-time (AOT) compilers for Python numeric code, respectively; and the *Cython* and *PyPy* systems noted earlier can optimize algorithmic code. For more basic needs, Python itself comes with a statistics module; complex, fixed-point, and rational math; and other tools you'll learn about in this book.

And More: AI, Games, Images, QA, Excel, Apps…

Python is commonly applied in far more domains and with far more tools than can possibly be covered here. For example, it's also used in:

- Artificial intelligence (see *PyTorch*, *TensorFlow*, and *Keras*)
- Game programming (see *pygame*, *Panda3D*, and *Kivy*)
- Image and graphics processing (see *Pillow*, *PyOpenGL*, and *OpenCV*)
- Quality assurance and testing (see *PyTest*, *unittest*, and *Selenium*)

- Excel spreadsheets (see *xlwings*, *PyXLL*, and *Excel*)
- Smartphone apps (see *Kivy*, *BeeWare*, and Appendix A)
- Microcontrollers and ports (see *MicroPython* and *PySerial*)
- And of course, much more

For links to resources in these and many other fields, try Wikipedia's Python software page; the PyPI website, which hosts extension packages installed by Python's *pip*; and the normal web searches.

Though application domains underscore Python's practical utility, keep in mind that many are largely just instances of Python's component *integration* role in action again. Adding Python as a frontend to libraries of components written in a compiled language like C makes Python useful for scripting in a wide variety of roles. As a general-purpose language that supports integration, Python is broadly, if not universally, applicable.

What Are Python's Technical Strengths?

Naturally, this is a developer's question. If you don't already have a programming background, the terminology in the next few sections may be a bit baffling—don't worry, we'll explore all of these topics in more detail as we proceed through this book. For both current and future developers, though, here is a quick rundown of Python's top technical features. Some have been touched on earlier, but this section fills in more of the story.

It's Object-Oriented and Functional

Python is an object-oriented language, from the ground up. As you'll find in this book, its *class model* supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet, in the context of Python's simple syntax and typing, OOP is remarkably easy to apply. In fact, if you don't understand these terms, you'll find they are much easier to learn with Python than with just about any other OOP language available.

Of equal significance, OOP is an *option* in Python; you can go far without having to become an object guru all at once. Much like C++, Python supports both procedural and object-oriented programming modes. Its object-oriented tools can be applied if and when constraints allow. This is especially useful in tactical development modes, which often preclude the design phases that best utilize OOP's benefits.

In addition to its original *procedural* (statement-based) and *object-oriented* (class-based) paradigms, Python today has built-in support for *functional* programming—a set that by most measures includes generators, comprehensions, closures, maps, decorators, anonymous-function lambdas, and first-class function objects. As you'll also learn in this book, these can serve as both complement and alternative to its OOP tools.

It's Free and Open

Python is completely free to use and distribute. As with other *open source software*, such as Linux and Apache, you can fetch the entire Python system's source code for free on the internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products.

But don't get the wrong idea: "free" doesn't mean "unsupported." On the contrary, Python users have access to numerous online resources that respond to queries and issues quicker than most commercial software. Moreover, because Python comes with complete source code, it empowers developers in ways that closed commercial software cannot. Although studying or changing a programming language's implementation code isn't everyone's idea of fun, it's comforting to know that you can. You're not dependent on the whims of a commercial vendor, because the ultimate documentation—*source code*—is at your disposal as a last resort.

It's Portable

We touched on portability earlier. The standard implementation of Python is written in portable ANSI C, and compiles and runs on virtually every major platform currently in use. For example, Python programs run today on

everything from smartphones to supercomputers. As a partial list, Python is available on Windows and macOS PCs, Linux and Unix workstations and servers, Android and iOS smartphones and tablets, real-time systems like VxWorks, Cray supercomputers, IBM mainframes, and more. Moreover, this list expands regularly; Android and iOS, for example, are gaining official *python.org* support at this writing.

Like the language itself, the *standard-library* modules that ship with Python are designed to be portable across platform boundaries; their file tools, for instance, remove many or most of the proprietary aspects of storage on some hosts. Furthermore, Python programs are automatically compiled to portable *bytecode*, which runs the same on any platform with a compatible version of Python installed (more on this in the next chapter), and there are multiple ways to code portable *user interfaces* in Python with traditional GUIs and web-based options described earlier.

This means that programs that use the Python language, its standard libraries, and portable extensions run the same on most systems that host a Python interpreter. Python also supports platform-specific extensions (e.g., *pywin32* on Windows, *PyObjC* on macOS, and *pyjnius* on Android), but Python itself works the same everywhere.

It’s Powerful

From a features perspective, Python is something of a hybrid. Its toolset places it between traditional scripting languages (such as Tcl, Scheme, and Perl) and systems development languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced software-engineering tools typically found in compiled languages. Unlike some scripting languages, this combination makes Python useful for large-scale development projects. As a preview, here are some of the main things you’ll find in Python’s toolbox:

Dynamic typing

Python keeps track of the kinds of objects your program uses when it runs, and doesn't require complicated type and size declarations in your code. In fact, as you'll see in Chapter 6, there is no such thing as a type or variable declaration anywhere in Python (apart from recent "hinting," which Python itself does not use). Because Python code does not constrain data types, it is also usually automatically applicable to a whole range of objects.

Automatic memory management

Python automatically allocates objects and reclaims ("garbage collects") them when they are no longer used. As you'll learn, Python keeps track of objects in use and the memory they hold, so you don't have to.

Programming-in-the-large support

For building larger systems, Python includes tools such as modules, classes, and exceptions. These tools allow you to organize systems into components, use OOP to reuse and customize code, and handle events and errors gracefully. Python's functional programming tools, described earlier, meet some of the same goals.

Built-in object types

Python provides commonly used data structures such as lists, dictionaries, and strings as intrinsic parts of the language. As you'll see, its built-in objects are both flexible

and easy to use. They can grow and shrink on demand, can be arbitrarily nested to represent complex information, and are immune to common memory errors.

Built-in tools

To process all those object types, Python comes with powerful and standard operations, including concatenation (joining collections), slicing (extracting sections), sorting, mapping, and more.

Library utilities

For more specific tasks, Python also comes with a large collection of precoded library tools that support everything from regular expression pattern matching to network servers. Once you learn the language itself, Python's library tools are where much of the application-level action occurs.

Third-party utilities

Because Python is open source, developers are encouraged to contribute precoded tools that support tasks beyond those supported by its built-ins. On the Web, you'll find free support for image processing, numeric programming, database access, website development, formal testing, and much more (see "What Can I Do with Python?").

Despite the array of tools in Python, it retains a noticeably simple syntax and design. The result is a powerful programming tool with all the usability of a scripting language.

It’s Mixable

As noted earlier, Python programs can easily be “glued” to components written in other languages in a variety of ways—both locally and across networks. That means you can add functionality to the Python system as needed and use Python programs within other environments or systems.

Mixing Python into systems coded in more demanding languages, for instance, makes it an easy-to-use frontend language for testing and customization. As also mentioned earlier, this makes Python good at rapid prototyping too—systems may be coded in Python first for development speed and later moved to extensions one piece at a time for execution speed if and when needed.

It’s Relatively Easy to Use

Compared to alternatives like C++, Java, and C#, Python programming seems astonishingly simple to most observers. To run Python code in most contexts, you simply type it and run it. There are no intermediate compile and link steps, like those typical for languages such as C or C++. Python executes programs immediately, which makes for an interactive programming experience and *rapid turnaround* after program changes—in many cases, you can witness the effect of a program change nearly as fast as you can type it.

Of course, development cycle turnaround is only one aspect of Python’s ease of use. It also provides a deliberately simple syntax and powerful built-in tools. In fact, some have gone so far as to call Python *executable pseudocode*. Because it eliminates much of the complexity of its contemporaries, Python programs are simpler, smaller, and more flexible than equivalent programs in other popular languages.

Which is not to say that Python makes programming a *no-brainer*. Python also has convolutions and dark corners that we’ll tackle head-on in this book, and some of its roles are more rapid than others. Python’s flavor of OOP inheritance, for example, is much more complicated than it once was, and building standalone apps or precompiled programs of the sort you’ll

meet in the next chapter can still be slow. Measured by its peers, though, Python is dramatically more coder friendly.

It’s Relatively Easy to Learn

Finally, this brings us to the point of this book: especially when compared to other widely used programming languages, the core Python language is remarkably easy to learn. In fact, if you’re already an experienced programmer, you can expect to be coding small-scale Python programs in a matter of days—though you shouldn’t expect to become an expert quite that fast, despite what you may have heard from marketing departments!

Naturally, mastering any topic as substantial as today’s Python is not trivial, and we’ll devote the rest of this book to this task. But the investment required to master Python is worthwhile: in the end, you’ll gain programming skills that apply to nearly every computer application domain. Moreover, most find Python’s learning curve to be much gentler than that of other programming tools, even if that curve is not quite as flat as some content publishers claim.

That’s good news for both professional developers seeking to add the language to their toolbox, and end users of systems that expose a Python layer for scripting roles. Today, many systems rely on the fact that people can learn enough Python to use the system with little or no support. Moreover, Python has spawned a legion of users who program for need or fun instead of career and may never require full-scale software development skills. Although Python has advanced tools you’ll meet in this book, its core fundamentals are accessible to beginners and gurus alike.

To see all this for yourself, let’s wrap up this overview and get started coding.

Chapter Summary

And that concludes the “hype” portion of this book. In this chapter, you’ve explored some of the reasons that people pick Python for their programming tasks. You’ve also seen how it is applied and looked at a representative sample of notable users today. This book’s goal is to teach Python, though, not to sell it. The best way to judge a language is to see it in action, so the rest of this book focuses entirely on the language details glossed over here.

The next two chapters begin your technical introduction to the language. In them, you’ll study ways to run Python programs, peek at Python’s execution model, and learn the basics of module files for saving code. The aim will be to give you just enough information to run the examples and exercises in the rest of the book. You won’t really start programming per se until Chapter 4, but make sure you have a handle on the startup details before moving on.

Test Your Knowledge: Quiz

In this book, we will be closing each chapter with a quick open-book quiz about the chapter’s coverage to help you review key concepts. The answers for these quizzes appear immediately after the questions, and you are encouraged to read the answers once you’ve taken a crack at the questions yourself, as they sometimes give useful summary context.

In addition to these end-of-chapter quizzes, you’ll find lab *exercises* at the end of each *part* of the book, designed to help you start coding Python on your own, with suggested answers available in an appendix. For now, here’s your first quiz. Good luck, and be sure to refer back to this chapter’s material as needed.

1. What are the six main reasons that people choose to use Python?

2. Name four notable companies or organizations using Python today.
3. Why might you *not* want to use Python in an application?
4. What can you do with Python?

Test Your Knowledge: Answers

How did you do? Here are the suggested answers, though there may be multiple solutions to some quiz questions. Again, even if you're sure of your answers, you're encouraged to look at the suggestions for additional context. See the chapter's coverage for more details if any of these responses don't make sense to you.

1. Software quality, developer productivity, program portability, support libraries, component integration, and simple enjoyment. Of these, the quality and productivity themes seem to be the main reasons that people choose to use Python, but enjoyment counts, too, in a field that can be as challenging as software.
2. Google, Industrial Light & Magic, JPL, ESRI, Instagram, and many more. Almost every organization doing software development uses Python in some fashion, whether for long-term strategic product development or for short-term tactical tasks such as testing and system administration.
3. Python's main downside is performance: in its currently most-common version, at least, it won't run as quickly as fully compiled languages like C and C++. On the other hand, it's quick enough for most applications, and typical Python code runs at close to C speed anyhow because it invokes linked-in C code in the interpreter. If speed is critical, compiled extensions and other tools like Cython are available to optimize the number-crunching parts of a Python application.
4. You can use Python for nearly anything you can do with a computer, from website development and gaming to AI and

spacecraft control. Numeric programming and web development may lead the pack today, though probably because those are some of the main things for which computers are used.

Chapter 2. How Python Runs Programs

This chapter and the next cover program execution—how you launch code and how Python runs it. In this chapter, we'll begin by studying how the Python interpreter executes programs in general, from an abstract vantage point. With that perspective in hand, Chapter 3 will dive into the nuts and bolts of getting your own programs up and running.

Startup details are inherently platform specific, and some of the material in these two chapters may not apply to the ways you'll be using Python, so you should feel free to skip parts not relevant to your goals. Likewise, readers who have used similar tools in the past and prefer to get to the meat of the language quickly may want to file some of these chapters away for future reference. For the rest of us, let's take a brief look at the way that Python will run our code, before we get into the details of writing or running it.

Introducing the Python Interpreter

So far, we've mostly been talking about Python as a programming language. In its most widely used form, though, it's also a software system called an *interpreter*. An interpreter is a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carries out the instructions it contains. In effect, the interpreter is a layer of logic between your code and the computer hardware on your machine.

When Python is installed, it generates a number of components—minimally, an interpreter and a support library. Depending on how you use it, the Python interpreter may take the form of an executable program, or a set of libraries linked into another program. Depending on which flavor of

Python you run, the interpreter itself may be implemented as a C program, a set of Java classes, or something else. Whatever form it takes, the Python code you write must always be run by this interpreter. And to enable that, you usually must install a Python interpreter on your computer.

You don't need to install Python at this point unless you want to work along with the sole trivial example coming up, and this book won't assume that you've got a Python ready to go until the next chapter. When you're ready, Python installation details vary by platform and are discussed briefly in Chapter 3, and covered in full in Appendix A.

Program Execution

What it means to run a Python script depends on whether you look at this task as a programmer, or as a Python interpreter. Both views offer important perspectives on Python programming.

The Programmer's View

In its simplest but most common form, a Python program is just a text file containing Python statements. For example, the file listed in Example 2-1, named *script0.py*, may be one of the most trivial Python scripts one could dream up, but it passes for a fully functional Python program.

Example 2-1. script0.py

```
print('hello world')
print(2 ** 100)
```

This file contains two Python `print` statements, which simply print a string (the text in quotes) and a numeric expression result (2 to the power 100) to the output stream (the GUI or window where the file is run, normally). Don't worry about the syntax of this code yet—for now, we're interested only in how it runs. This book will explain the `print` statement, and why you can raise 2 to the power 100 so easily in Python, in its later chapters.

You can create such a file of statements with any text editor you like; see Appendix A for suggestions. By convention, Python program files are given names that end in *.py*; technically, this naming scheme is required only for files that are "imported"—a term clarified in the next chapter—but most Python files have *.py* names for consistency.

After you've typed these statements into a text file, you must tell Python to *execute* the file—which simply means to run all the statements in the file from top to bottom, one after another. As you'll see in the next chapter, you can launch Python program files by typing command lines, by clicking their icons, from within coding GUIs, and with other techniques. If all goes well, when you execute the file, you'll see the results of the two `print` statements show up somewhere on your computer—usually and by default, in the same window you were in when you ran the program:

```
hello world
1267650600228229401496703205376
```

For example, here's what happened when this script was run in a Command Prompt window with a command line on a Windows laptop, to make sure it didn't have any silly typos:

```
C:\Users\me\code> py script0.py
hello world
1267650600228229401496703205376
```

See Chapter 3 for the full story on this process, especially if you're new to programming; we'll get into all the details of writing and launching programs there. For our purposes here, we've just run a Python script that prints a string and a number. We probably won't attract venture capital or go viral on GitHub with this code, but it's enough to capture the basics of program execution.

Python’s View

The brief description in the prior section is fairly standard for scripting languages, and it’s usually all that most new Python programmers need to know. You type code into text files, and you run those files through the interpreter. Under the hood, though, a bit more happens when you tell Python to “go.” Although knowledge of Python internals is not strictly required for Python programming, having a basic understanding of Python’s runtime structure up front can help you grasp how your code fits into the bigger picture of program execution.

When you instruct Python to run your script, there are a few steps that Python carries out before your code actually starts crunching away. Specifically, it’s first compiled to something called “bytecode” and then routed to something called a “virtual machine.” This holds true only for the most common version of Python, and you’ll meet variations on this model in a moment. Since the most common is, well, most common, let’s see how this works first.

Bytecode compilation

Internally, and almost completely hidden from you, when you execute a program Python first compiles your *source code* (the statements in your text file) into a format known as *bytecode*. Compilation is simply a translation step, and bytecode is a lower-level, platform-independent representation of your source code. Roughly, Python translates each of your source statements into a group of bytecode instructions by decomposing them into individual steps. This bytecode translation is performed to speed execution —bytecode can be run much more quickly than the original source code statements in your text file.

You’ll notice that the prior paragraph said that this is *almost* completely hidden from you. If the Python program has write access on your machine, it will save the bytecode of your programs in files that end with a *.pyc* extension (*.pyc* means *.py* source, compiled). Technically, this save doesn’t happen when running a single file as we did in the preceding section but

does for all but the topmost file in meatier multifile programs (more on this in a moment).

Python saves its bytecode files in a subdirectory named *__pycache__* located in the directory where your source files reside, and in files whose names identify the Python version that created them (e.g., *script0.cpython-312.pyc* for 3.12). The *__pycache__* subdirectory avoids clutter, and the naming convention for bytecode files prevents different Python versions installed on the same computer from overwriting each other's saved bytecode. We'll study this bytecode file model in more detail in Chapter 22, though it's automatic and irrelevant to most Python programs and is free to vary among the alternative Python implementations described ahead.

So why all the bother? In short, Python saves bytecode like this as a *startup-speed* optimization. The next time you run your program, Python will load the *.pyc* files and skip the compilation steps, as long as the bytecode is present, you haven't changed your source code since the bytecode was last saved, and you aren't running with a different Python than the one that created the bytecode. It works like this:

Source changes

> Python saves the last-modified timestamp and size (or hash value, optionally) of a source code file in its bytecode file, and compares this info to the source when the bytecode is loaded to know when it must recompile—if you edit and resave your source code, its bytecode is re-created the next time your program is run.

Python versions

> Python also adds a version-information suffix to bytecode filenames to know when it must recompile—if you run your

program on a different Python implementation or version, its bytecode is generated and saved for that Python too.

The result is that both source code changes and differing Python versions will trigger a new bytecode file automatically. If Python *cannot* write the bytecode files to your machine, your program still works—the bytecode is generated in memory and simply discarded on program exit.

Because *.pyc* files speed startup time, though, you'll want to make sure they are written for larger programs. Bytecode files are also one way to ship Python programs—Python is happy to run a program if all it can find are compatible *.pyc* files, even if the original *.py* source files are absent. This isn't as simple as deleting the *.py* files, though, and may require file moves and renames, or special techniques discussed later in Chapter 22. See Python's `compileall` module to force compiles when needed for packaging, and frozen binaries (see "Standalone Executables") for another shipping option.

Strictly speaking, bytecode is an *import* optimization. Bytecode is saved in *.pyc* files only for files that are *imported*, not for the top-level files of a program that are only run as scripts. Moreover, a given file is imported and possibly compiled only *once* per program run; later imports use what's already been loaded. We'll explore import basics in Chapter 3 and take a deeper look at imports in Part V. For now, keep in mind that bytecode is never saved for code typed at the *interactive prompt*—a programming mode you'll learn about in Chapter 3 and use early in this book.

The Python Virtual Machine (PVM)

Once your program has been compiled to bytecode (or the bytecode has been loaded from existing *.pyc* files), it is shipped off for execution to something generally known as the Python Virtual Machine (PVM, for acronym-inclined readers). The PVM sounds more impressive than it is; really, it's not a separate program, and it need not be installed by itself. In fact, the PVM is just a big code loop that iterates through your bytecode

instructions, one by one, to carry out their operations. That is, the PVM is the runtime engine in Python. It's always present as part of the Python system, is the component that truly runs your scripts, and is really just the last step of the "Python interpreter."

Figure 2-1 illustrates this runtime structure. Bear in mind that all of this complexity is deliberately hidden from Python programmers. Bytecode compilation is automatic, and the PVM is just part of the Python system that you have installed on your machine. Again, programmers simply code and run files of statements, and Python handles the logistics of running them behind the scenes.

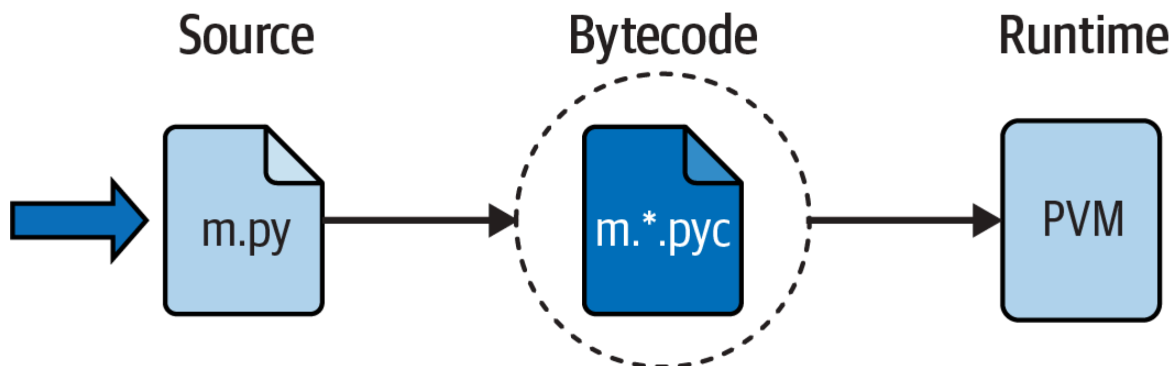

Figure 2-1. Python's traditional execution model: the PVM runs compiled bytecode

Performance implications

Readers with a background in fully compiled languages such as C and C++ might notice some glaring differences in the Python model. For one thing, there is usually no build or "make" step in Python work: code runs immediately after it is written. For another, Python bytecode is not binary *machine code* (e.g., instructions for an Intel or ARM chip, known as a *CPU*): it's a Python-specific format. There are exceptions to these rules (e.g., app builds for smartphones can take some time, and full compilers do exist, as you'll see ahead), but we're focusing on the common here first.

These differences explain why some Python code may not run as fast as C or C++ code, as described in Chapter 1—the PVM loop, not the CPU chip, still must interpret the bytecode, and bytecode instructions require more work than CPU instructions. On the other hand, unlike in classic

interpreters, there is still an internal compile step—Python does not need to reanalyze and reparse each source statement's text repeatedly. The net effect is that pure Python code runs at speeds somewhere between those of a traditional compiled language and a traditional interpreted language.

Development implications

Another ramification of Python's execution model is that there is really no distinction between the development and execution environments: the systems that compile and execute your source code are really one and the same. This similarity may have a bit more significance to readers with a background in traditional compiled languages, but in Python, the compiler is always present at runtime and is part of the system that runs programs.

This makes for a much more *rapid* development cycle. There is no need to precompile and link before execution can begin; simply type and run the code. This also adds a much more *dynamic* flavor to the language—it is possible, and often very convenient, for Python programs to construct and execute other Python programs at runtime. The `eval` and `exec` built-ins, for instance, accept and run strings containing Python program code. This structure is also why Python lends itself to product customization—because Python code can be changed on the fly, users can modify the Python parts of a system on-site without needing to compile or even possess the rest of the system's code.

At a more fundamental level, keep in mind that all we really have in Python is *runtime*—there is no initial compile-time phase at all, and everything happens as the program is running. This even includes operations such as the creation of functions and classes and the linkage of modules. Such events often occur before execution in more static languages, but happen during execution in Python. As you'll see, this makes for a much more dynamic programming experience than that to which some readers may be accustomed.

parsing

  slicing, Indexing and Slicing

  text, string methods, More String Methods: Parsing Text

pass statements, break, continue, pass, and the Loop else

pathnames, Filenames in open and Other Filename Tools-Filenames in open and Other Filename Tools

patterns

  factory functions, Closures and Factory Functions-Closures and Factory Functions

  re module, The re Pattern-Matching Module

  sequence assignments, Advanced sequence-assignment patterns

  structural pattern matching, match Statements

per-call scopes, Scopes Overview

performance, Performance implications

  list comprehensions, When to use list comprehensions: Speed, conciseness, etc.

permutation, sequences, Permutating Sequences-Why generators here: Space, time, and more

persistence, Stream Processors Revisited

pickle module, Test Your Knowledge: Answers, Pickles and Shelves, The pickle and json Serialization Modules-The pickle and json Serialization Modules

  object storage, Storing Objects with pickle-Storing Objects with pickle

pickle objects, Database Access, Stream Processors Revisited

- arbitrary expressions, Sequence Operations
- byte strings, Sequence Operations-Formatting
- indexing expressions, Sequence Operations
- list comprehensions, Conversions, methods, and immutability
- lists, Sequence Operations

sequence patterns, Advanced match Usage

sequence scans

- for loop, Sequence Scans: while, range, and for-Sequence Scans: while, range, and for
- range object, Sequence Scans: while, range, and for-Sequence Scans: while, range, and for
- while loop, Sequence Scans: while, range, and for-Sequence Scans: while, range, and for

sequences, Test Your Knowledge: Answers

- dictionaries, Dictionary Usage Tips
- for loops, Tuple (sequence) assignment in for loops-Tuple (sequence) assignment in for loops
- lists, Types Share Operation Sets by Categories
- permutation, Permutating Sequences-Why generators here: Space, time, and more
- reordering, Sequence Shufflers: range and len
- slicing, Scrambling Sequences
  - functions, Simple functions
  - generator expressions, Generator expressions