Powerful Object-Oriented Programming

5th Edition
Updated for 3.3 and 2.7

Learning

Python

O'REILLY®

Mark Lutz

FIFTH EDITION

Learning Python

Mark Lutz

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Learning Python, Fifth Edition
by Mark Lutz

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Rachel Roumeliotis
Production Editor: Christopher Hearse
Copyeditor: Rachel Monaghan
Proofreader: Julie Van Keuren

Indexer: Lucie Haskins
Cover Designer: Randy Comer
Interior Designer: David Futato
Illustrator: Rebecca Demarest

June 2013: Fifth Edition.

Revision History for the Fifth Edition:
2013-06-07 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449355739* for release details.

ISBN: 978-1-449-35573-9

[QG]

1370970520

To Vera.

You are my life.

Table of Contents

Part VI. Classes and OOP

Preface

If you're standing in a bookstore looking for the short story on this book, try this:

- *Python* is a powerful multiparadigm computer programming language, optimized for programmer productivity, code readability, and software quality.
- *This book* provides a comprehensive and in-depth introduction to the Python language itself. Its goal is to help you master Python fundamentals before moving on to apply them in your work. Like all its prior editions, this book is designed to serve as a single, all-inclusive learning resource for all Python newcomers, whether they will be using Python 2.X, Python 3.X, or both.
- *This edition* has been brought up to date with Python releases 3.3 and 2.7, and has been expanded substantially to reflect current practice in the Python world.

This preface describes this book's goals, scope, and structure in more detail. It's optional reading, but is designed to provide some orientation before you get started with the book at large.

This Book's "Ecosystem"

Python is a popular open source programming language used for both standalone programs and scripting applications in a wide variety of domains. It is free, portable, powerful, and is both relatively easy and remarkably fun to use. Programmers from every corner of the software industry have found Python's focus on developer productivity and software quality to be a strategic advantage in projects both large and small.

Whether you are new to programming or are a professional developer, this book is designed to bring you up to speed on the Python language in ways that more limited approaches cannot. After reading this book, you should know enough about Python to apply it in whatever application domains you choose to explore.

By design, this book is a tutorial that emphasizes the *core Python language* itself, rather than specific applications of it. As such, this book is intended to serve as the first in a two-volume set:

- *Learning Python*, this book, teaches Python itself, focusing on language fundamentals that span domains.
- *Programming Python*, among others, moves on to show what you can do with Python after you've learned it.

This division of labor is deliberate. While application goals can vary per reader, the need for useful language fundamentals coverage does not. Applications-focused books such as *Programming Python* pick up where this book leaves off, using realistically scaled examples to explore Python's role in common domains such as the Web, GUIs, systems, databases, and text. In addition, the book *Python Pocket Reference* provides reference materials not included here, and it is designed to supplement this book.

Because of this book's focus on foundations, though, it is able to present Python language fundamentals with more depth than many programmers see when first learning the language. Its bottom-up approach and self-contained didactic examples are designed to teach readers the entire language one step at a time.

The core language skills you'll gain in the process will apply to every Python software system you'll encounter—be it today's popular tools such as Django, NumPy, and App Engine, or others that may be a part of both Python's future and your programming career.

Because it's based upon a three-day Python training class with quizzes and exercises throughout, this book also serves as a self-paced introduction to the language. Although its format lacks the live interaction of a class, it compensates in the extra depth and flexibility that only a book can provide. Though there are many ways to use this book, linear readers will find it roughly equivalent to a semester-long Python class.

About This Fifth Edition

The prior *fourth edition* of this book published in 2009 covered Python versions 2.6 and 3.0.[1] It addressed the many and sometimes incompatible changes introduced in the Python 3.X line in general. It also introduced a new OOP tutorial, and new chapters on advanced topics such as Unicode text, decorators, and metaclasses, derived from both the live classes I teach and evolution in Python "best practice."

This *fifth edition* completed in 2013 is a revision of the prior, updated to cover both *Python 3.3 and 2.7*, the current latest releases in the 3.X and 2.X lines. It incorporates

1. And 2007's short-lived third edition covered Python 2.5, and its simpler—and *shorter*—single-line Python world. See *http://www.rmi.net/~lutz* for more on this book's history. Over the years, this book has grown in size and complexity in direct proportion to Python's own growth. Per Appendix C, Python 3.0 alone introduced 27 additions and 57 changes in the language that found their way into this book, and Python 3.3 continues this trend. Today's Python programmer faces two incompatible lines, three major paradigms, a plethora of advanced tools, and a blizzard of feature redundancy—most of which do not divide neatly between the 2.X and 3.X lines. That's not as daunting as it may sound (many tools are variations on a theme), but all are fair game in an inclusive, comprehensive Python text.

all language changes introduced in each line since the prior edition was published, and has been polished throughout to update and sharpen its presentation. Specifically:

- *Python 2.X* coverage here has been updated to include features such as dictionary and set comprehensions that were formerly for 3.X only, but have been back-ported for use in 2.7.
- *Python 3.X* coverage has been augmented for new `yield` and `raise` syntax; the *__pycache__* bytecode model; 3.3 namespace packages; PyDoc's all-browser mode; Unicode literal and storage changes; and the new Windows launcher shipped with 3.3.
- *Assorted* new or expanded coverage for JSON, `timeit`, PyPy, `os.popen`, generators, recursion, weak references, `__mro__`, `__iter__`, `super`, `__slots__`, metaclasses, descriptors, `random`, Sphinx, and more has been added, along with a general increase in 2.X compatibility in both examples and narrative.

This edition also adds a new *conclusion* as Chapter 41 (on Python's evolution), two new *appendixes* (on recent Python changes and the new Windows launcher), and one new *chapter* (on benchmarking: an expanded version of the former code timing example). See Appendix C for a concise summary of *Python changes* between the prior edition and this one, as well as links to their coverage in the book. This appendix also summarizes initial differences between 2.X and 3.X in general that were first addressed in the prior edition, though some, such as new-style classes, span versions and simply become mandated in 3.X (more on what the X's mean in a moment).

Per the last bullet in the preceding list, this edition has also experienced some growth because it gives fuller coverage to more *advanced language features*—which many of us have tried very hard to ignore as optional for the last decade, but which have now grown more common in Python code. As we'll see, these tools make Python more powerful, but also raise the bar for newcomers, and may shift Python's scope and definition. Because you might encounter any of these, this book covers them head-on, instead of pretending they do not exist.

Despite the updates, this edition retains most of the structure and content of the prior edition, and is still designed to be a comprehensive learning resource for both the 2.X and 3.X Python lines. While it is primarily focused on users of Python 3.3 and 2.7—the latest in the 3.X line and the likely last in the 2.X line—its historical perspective also makes it relevant to *older* Pythons that still see regular use today.

Though it's impossible to predict the future, this book stresses fundamentals that have been valid for nearly two decades, and will likely apply to *future* Pythons too. As usual, I'll be posting Python updates that impact this book at the book's website described ahead. The "What's New" documents in Python's manuals set can also serve to fill in the gaps as Python surely evolves after this book is published.

The Python 2.X and 3.X Lines

Because it bears heavily on this book's content, I need to say a few more words about the Python 2.X/3.X story up front. When the *fourth edition* of this book was written in 2009, Python had just become available in two flavors:

- Version 3.0 was the first in the line of an emerging and incompatible mutation of the language known generically as *3.X*.
- Version 2.6 retained backward compatibility with the vast body of existing Python code, and was the latest in the line known collectively as *2.X*.

While 3.X was largely the same language, it ran almost no code written for prior releases. It:

- Imposed a Unicode model with broad consequences for strings, files, and libraries
- Elevated iterators and generators to a more pervasive role, as part of fuller functional paradigm
- Mandated new-style classes, which merge with types, but grow more powerful and complex
- Changed many fundamental tools and libraries, and replaced or removed others entirely

The mutation of `print` from statement to function alone, aesthetically sound as it may be, broke nearly every Python program ever written. And strategic potential aside, 3.X's mandatory Unicode and class models and ubiquitous generators made for a different programming experience.

Although many viewed Python 3.X as both an improvement and the future of Python, Python 2.X was still very widely used and was to be supported in parallel with Python 3.X for years to come. The majority of Python code in use was 2.X, and migration to 3.X seemed to be shaping up to be a slow process.

The 2.X/3.X Story Today

As this *fifth edition* is being written in 2013, Python has moved on to versions 3.3 and 2.7, but this 2.X/3.X story is still largely *unchanged*. In fact, Python is now a dual-version world, with many users running *both* 2.X and 3.X according to their software goals and dependencies. And for many newcomers, the choice between 2.X and 3.X remains one of existing software versus the language's cutting edge. Although many major Python packages have been ported to 3.X, many others are still 2.X-only today.

To some observers, Python 3.X is now seen as a *sandbox* for exploring new ideas, while 2.X is viewed as the *tried-and-true* Python, which doesn't have all of 3.X's features but is still more pervasive. Others still see Python 3.X as the future, a view that seems supported by current core developer plans: Python 2.7 will continue to be supported but is to be the last 2.X, while 3.3 is the latest in the 3.X line's continuing evolution.

On the other hand, initiatives such as *PyPy*—today a still 2.X-only implementation of Python that offers stunning performance improvements—represent a 2.X future, if not an outright faction.

All opinions aside, almost five years after its release, 3.X has yet to supersede 2.X, or even match its user base. As one metric, 2.X is still downloaded more often than 3.X for Windows at python.org today, despite the fact that this measure would be naturally skewed to *new* users and the *most recent* release. Such statistics are prone to change, of course, but after five years are indicative of 3.X uptake nonetheless. The existing 2.X software base still trumps 3.X's language extensions for many. Moreover, being last in the 2.X line makes 2.7 a sort of *de facto standard*, immune to the constant pace of change in the 3.X line—a positive to those who seek a stable base, and a negative to those who seek growth and ongoing relevance.

Personally, I think today's Python world is large enough to accommodate *both* 3.X and 2.X; they seem to satisfy different goals and appeal to different camps, and there is precedence for this in other language families (C and C++, for example, have a long-standing coexistence, though they may differ more than Python 2.X and 3.X). Moreover, because they are so similar, the skills gained by learning either Python line transfer almost entirely to the other, especially if you're aided by dual-version resources like this book. In fact, as long as you understand how they diverge, it's often possible to write code that runs on both.

At the same time, this split presents a substantial *dilemma* for both programmers and book authors, which shows no signs of abating. While it would be easier for a book to pretend that Python 2.X never existed and cover 3.X only, this would not address the needs of the large Python user base that exists today. A vast amount of existing code was written for Python 2.X, and it won't be going away anytime soon. And while some newcomers to the language can and should focus on Python 3.X, anyone who must use code written in the past needs to keep one foot in the Python 2.X world today. Since it may still be years before many third-party libraries and extensions are ported to Python 3.X, this fork might not be entirely temporary.

Coverage for Both 3.X and 2.X

To address this dichotomy and to meet the needs of all potential readers, this book has been updated to cover *both* Python 3.3 and Python 2.7, and should apply to later releases in both the 3.X and 2.X lines. It's intended for programmers using Python 2.X, programmers using Python 3.X, and programmers stuck somewhere between the two.

That is, you can use this book to learn *either* Python line. Although 3.X is often emphasized, 2.X differences and tools are also noted along the way for programmers using older code. While the two versions are largely similar, they diverge in some important ways, and I'll point these out as they crop up.

For instance, I'll use 3.X `print` calls in most examples, but will also describe the 2.X `print` statement so you can make sense of earlier code, and will often use portable printing techniques that run on both lines. I'll also freely introduce new features, such as the `nonlocal` statement in 3.X and the string `format` method available as of 2.6 and 3.0, and will point out when such extensions are not present in older Pythons.

By proxy, this edition addresses other Python version 2.X and 3.X releases as well, though some older version 2.X code may not be able to run all the examples here. Although class decorators are available as of both Python 2.6 and 3.0, for example, you cannot use them in an older Python 2.X that did not yet have this feature. Again, see the change tables in Appendix C for summaries of recent 2.X and 3.X changes.

Which Python Should I Use?

Version choice may be mandated by your organization, but if you're new to Python and learning on your own, you may be wondering which version to install. The answer here depends on your goals. Here are a few suggestions on the choice.

When to choose 3.X: new features, evolution
: If you are learning Python for the first time and don't need to use any existing 2.X code, I encourage you to begin with Python 3.X. It cleans up some longstanding warts in the language and trims some dated cruft, while retaining all the original core ideas and adding some nice new tools. For example, 3.X's seamless Unicode model and broader use of generators and functional techniques are seen by many users as assets. Many popular Python libraries and tools are already available for Python 3.X, or will be by the time you read these words, especially given the continual improvements in the 3.X line. All new language evolution occurs in 3.X only, which adds features and keeps Python relevant, but also makes language definition a constantly moving target—a tradeoff inherent on the leading edge.

When to choose 2.X: existing code, stability
: If you'll be using a system based on Python 2.X, the 3.X line may not be an option for you today. However, you'll find that this book addresses your concerns, too, and will help if you migrate to 3.X in the future. You'll also find that you're in large company. Every group I taught in 2012 was using 2.X only, and I still regularly see useful Python software in 2.X-only form. Moreover, unlike 3.X, 2.X is no longer being changed—which is either an asset or liability, depending on whom you ask. There's nothing wrong with using and writing 2.X code, but you may wish to keep tabs on 3.X and its ongoing evolution as you do. Python's future remains to be written, and is largely up to its users, including you.

When to choose both: version-neutral code
: Probably the best news here is that Python's fundamentals are the same in both its lines—2.X and 3.X differ in ways that many users will find minor, and this book is designed to help you learn both. In fact, as long as you understand their differences, it's often straightforward to write version-neutral code that runs on both

Pythons, as we regularly will in this book. See Appendix C for pointers on 2.X/3.X migration and tips on writing code for both Python lines and audiences.

Regardless of which version or versions you choose to focus on first, your skills will transfer directly to wherever your Python work leads you.

About the Xs: Throughout this book, "3.X" and "2.X" are used to refer collectively to all releases in these two lines. For instance, *3.X* includes 3.0 through 3.3, and future 3.X releases; *2.X* means all from 2.0 through 2.7 (and presumably no others). More specific releases are mentioned when a topic applies to it only (e.g., 2.7's set literals and 3.3's launcher and namespace packages). This notation may occasionally be too broad —some features labeled 2.X here may not be present in early 2.X releases rarely used today—but it accommodates a 2.X line that has already spanned 13 years. The 3.X label is more easily and accurately applied to this younger five-year-old line.

This Book's Prerequisites and Effort

It's impossible to give absolute prerequisites for this book, because its utility and value can depend as much on reader motivation as on reader background. Both true beginners and crusty programming veterans have used this book successfully in the past. If you are motivated to learn Python, and willing to invest the time and focus it requires, this text will probably work for you.

Just how much time is required to learn Python? Although this will vary per learner, this book tends to work best when *read*. Some readers may use this book as an on-demand reference resource, but most people seeking Python mastery should expect to spend at least *weeks* and probably *months* going through the material here, depending on how closely they follow along with its examples. As mentioned, it's roughly equivalent to a full-semester course on the Python language itself.

That's the estimate for learning just Python itself and the software skills required to use it well. Though this book may suffice for basic scripting goals, readers hoping to pursue software development at large as a career should expect to devote additional time after this book to large-scale project experience, and possibly to follow-up texts such as *Programming Python*.[2]

2. The standard disclaimer: I wrote this and another book mentioned earlier, which work together as a set: *Learning Python* for language fundamentals, *Programming Python* for applications basics, and *Python Pocket Reference* as a companion to the other two. All three derive from 1995's original and broad *Programming Python*. I encourage you to explore the many Python books available today (I stopped counting at 200 at Amazon.com just now because there was no end in sight, and this didn't include related subjects like Django). My own publisher has recently produced Python-focused books on instrumentation, data mining, App Engine, numeric analysis, natural language processing, MongoDB, AWS, and more—specific domains you may wish to explore once you've mastered Python language fundamentals here. The Python story today is far too rich for any one book to address alone.

That may not be welcome news to people looking for instant proficiency, but programming is not a trivial skill (despite what you may have heard!). Today's Python, and software in general, are both challenging and rewarding enough to merit the effort implied by comprehensive books such as this. Here are a few pointers on using this book for readers on both sides of the experience spectrum:

To experienced programmers
: You have an initial advantage and can move quickly through some earlier chapters; but you shouldn't skip the core ideas, and may need to work at letting go of some baggage. In general terms, exposure to any programming or scripting before this book might be helpful because of the analogies it may provide. On the other hand, I've also found that prior programming experience can be a handicap due to expectations rooted in other languages (it's far too easy to spot the Java or C++ programmers in classes by the first Python code they write!). Using Python well requires adopting its mindset. By focusing on key core concepts, this book is designed to help you learn to code Python in Python.

To true beginners
: You can learn Python here too, as well as programming itself; but you may need to work a bit harder, and may wish to supplement this text with gentler introductions. If you don't consider yourself a programmer already, you will probably find this book useful too, but you'll want to be sure to proceed slowly and work through the examples and exercises along the way. Also keep in mind that this book will spend more time teaching Python itself than programming basics. If you find yourself lost here, I encourage you to explore an introduction to programming in general before tackling this book. Python's website has links to many helpful resources for beginners.

Formally, this book is designed to serve as *a first Python text for newcomers of all kinds*. It may not be an ideal resource for someone who has never touched a computer before (for instance, we're not going to spend any time exploring what a computer is), but I haven't made many assumptions about your programming background or education.

On the other hand, I won't insult readers by assuming they are "dummies," either, whatever that means—it's easy to do useful things in Python, and this book will show you how. The text occasionally contrasts Python with languages such as C, C++, Java, and others, but you can safely ignore these comparisons if you haven't used such languages in the past.

This Book's Structure

To help orient you, this section provides a quick rundown of the content and goals of the major parts of this book. If you're anxious to get to it, you should feel free to skip

this section (or browse the table of contents instead). To some readers, though, a book this large probably merits a brief roadmap up front.

By design, each *part* covers a major functional area of the language, and each part is composed of *chapters* focusing on a specific topic or aspect of the part's area. In addition, each chapter ends with *quizzes* and their answers, and each part ends with larger *exercises*, whose solutions show up in Appendix D.

Practice matters: I strongly recommend that readers work through the quizzes and exercises in this book, and work along with its examples in general if you can. In programming, there's no substitute for practicing what you've read. Whether you do it with this book or a project of your own, actual coding is crucial if you want the ideas presented here to stick.

Overall, this book's presentation is *bottom-up* because Python is too. The examples and topics grow more challenging as we move along. For instance, Python's classes are largely just packages of functions that process built-in types. Once you've mastered built-in types and functions, classes become a relatively minor intellectual leap. Because each part builds on those preceding it this way, most readers will find a *linear reading* makes the most sense. Here's a preview of the book's main parts you'll find along the way:

Part I
: We begin with a general overview of Python that answers commonly asked initial questions—why people use the language, what it's useful for, and so on. The first chapter introduces the major ideas underlying the technology to give you some background context. The rest of this part moves on to explore the ways that both Python and programmers run programs. The main goal here is to give you just enough information to be able to follow along with later examples and exercises.

Part II
: Next, we begin our tour of the Python language, studying Python's major built-in object types and what you can do with them in depth: numbers, lists, dictionaries, and so on. You can get a lot done with these tools alone, and they are at the heart of every Python script. This is the most substantial part of the book because we lay groundwork here for later chapters. We'll also explore dynamic typing and its references—keys to using Python well—in this part.

Part III
: The next part moves on to introduce Python's *statements*—the code you type to create and process objects in Python. It also presents Python's general syntax model. Although this part focuses on syntax, it also introduces some related tools (such as the PyDoc system), takes a first look at iteration concepts, and explores coding alternatives.

Part IV

This part begins our look at Python's higher-level program structure tools. *Functions* turn out to be a simple way to package code for reuse and avoid code redundancy. In this part, we will explore Python's scoping rules, argument-passing techniques, the sometimes-notorious lambda, and more. We'll also revisit iterators from a functional programming perspective, introduce user-defined generators, and learn how to time Python code to measure performance here.

Part V

Python *modules* let you organize statements and functions into larger components, and this part illustrates how to create, use, and reload modules. We'll also look at some more advanced topics here, such as module packages, module reloading, package-relative imports, 3.3's new namespace packages, and the `__name__` variable.

Part VI

Here, we explore Python's object-oriented programming tool, the *class*—an optional but powerful way to structure code for customization and reuse, which almost naturally minimizes redundancy. As you'll see, classes mostly reuse ideas we will have covered by this point in the book, and OOP in Python is mostly about looking up names in linked objects with a special first argument in functions. As you'll also see, OOP is optional in Python, but most find Python's OOP to be much simpler than others, and it can shave development time substantially, especially for long-term strategic project development.

Part VII

We conclude the language fundamentals coverage in this text with a look at Python's exception handling model and statements, plus a brief overview of development tools that will become more useful when you start writing larger programs (debugging and testing tools, for instance). Although exceptions are a fairly lightweight tool, this part appears after the discussion of classes because user-defined exceptions should now all be classes. We also cover some more advanced topics, such as context managers, here.

Part VIII

In the final part, we explore some advanced topics: Unicode and byte strings, managed attribute tools like properties and descriptors, function and class decorators, and metaclasses. These chapters are all optional reading, because not all programmers need to understand the subjects they address. On the other hand, readers who must process internationalized text or binary data, or are responsible for developing APIs for other programmers to use, should find something of interest in this part. The examples here are also larger than most of those in this book, and can serve as self-study material.

Part IX

The book wraps up with a set of four appendixes that give platform-specific tips for installing and using Python on various computers; present the new Windows

launcher that ships with Python 3.3; summarize changes in Python addressed by recent editions and give links to their coverage here; and provide solutions to the end-of-part exercises. Solutions to end-of-chapter quizzes appear in the chapters themselves.

See the table of contents for a finer-grained look at this book's components.

What This Book Is Not

Given its relatively large audience over the years, some have inevitably expected this book to serve a role outside its scope. So now that I've told you what this book is, I also want to be clear on what it isn't:

- This book is a tutorial, *not* a reference.
- This book covers the language itself, *not* applications, standard libraries, or third-party tools.
- This book is a comprehensive look at a substantial topic, *not* a watered-down overview.

Because these points are key to this book's content, I want to say a few more words about them up front.

It's Not a Reference or a Guide to Specific Applications

This book is a *language tutorial*, not a reference, and not an applications book. This is by design: *today's Python*—with its built-in types, generators, closures, comprehensions, Unicode, decorators, and blend of procedural, object-oriented, and functional programming paradigms—makes the core language a substantial topic all by itself, and a prerequisite to all your future Python work, in whatever domains you pursue. When you are ready for other resources, though, here are a few suggestions and reminders:

Reference resources
: As implied by the preceding structural description, you can use the index and table of contents to hunt for details, but there are no reference appendixes in this book. If you are looking for Python reference resources (and most readers probably will be very soon in their Python careers), I suggest the previously mentioned book that I also wrote as a companion to this one—*Python Pocket Reference*—as well as other reference books you'll find with a quick search, and the standard Python reference manuals maintained at *http://www.python.org*. The latter of these are free, always up to date, and available both on the Web and on your computer after a Windows install.

Applications and libraries
: As also discussed earlier, this book is not a guide to specific *applications* such as the Web, GUIs, or systems programming. By proxy, this includes the libraries and

tools used in applications work; although some *standard libraries* and tools are introduced here—including `timeit`, `shelve`, `pickle`, `struct`, `json`, `pdb`, `os`, `urllib`, `re`, `xml`, `random`, *PyDoc* and *IDLE*—they are not officially in this book's primary scope. If you're looking for more coverage on such topics and are already proficient with Python, I recommend the follow-up book *Programming Python*, among others. That book assumes this one as its prerequisite, though, so be sure you have a firm grasp of the core language first. Especially in an engineering domain like software, one must walk before one runs.

It's Not the Short Story for People in a Hurry

As you can tell from its size, this book also doesn't skimp on the details: it presents the *full Python language*, not a brief look at a simplified subset. Along the way it also covers *software principles* that are essential to writing good Python code. As mentioned, this is a multiple-week or -month book, designed to impart the skill level you'd acquire from a full-term class on Python.

This is also deliberate. Many of this book's readers don't need to acquire full-scale software development skills, of course, and some can absorb Python in a piecemeal fashion. At the same time, because *any* part of the language may be used in code you will encounter, no part is truly optional for most programmers. Moreover, even casual scripters and hobbyists need to know basic principles of software development in order to code well, and even to use precoded tools properly.

This book aims to address both of these needs—*language and principles*—in enough depth to be useful. In the end, though, you'll find that Python's more advanced tools, such as its object-oriented and functional programming support, are relatively easy to learn once you've mastered their prerequisites—and you will, if you work through this book one chapter at a time.

It's as Linear as Python Allows

Speaking of *reading order*, this edition also tries hard to minimize *forward references*, but Python 3.X's changes make this impossible in some cases (in fact, 3.X sometimes seems to assume you already know Python while you're learning it!). As a handful of representative examples:

- Printing, sorts, the string `format` method, and some `dict` calls rely on function *keyword* arguments.
- Dictionary key lists and tests, and the `list` calls used around many tools, imply *iteration* concepts.
- Using `exec` to run code now assumes knowledge of *file objects* and interfaces.
- Coding new *exceptions* requires *classes* and OOP fundamentals.

- And so on—even basic *inheritance* broaches advanced topics such as *metaclasses* and *descriptors*.

Python is still best learned as a progression from simple to advanced, and a *linear reading* here still makes the most sense. Still, some topics may require nonlinear jumps and random lookups. To minimize these, this book will point out forward dependencies when they occur, and will ease their impacts as much as possible.

But if your time is tight: Though depth is crucial to mastering Python, some readers may have limited time. If you are interested in starting out with a *quick Python tour*, I suggest Chapter 1, Chapter 4, Chapter 10, and Chapter 28 (and perhaps 26)—a short survey that will hopefully pique your interest in the more complete story told in the rest of the book, and which most readers will need in today's Python software world. In general, this book is intentionally *layered* this way to make its material easier to absorb—with introductions followed by details, so you can start with overviews, and dig deeper over time. You don't need to read this book all at once, but its gradual approach is designed to help you tackle its material eventually.

This Book's Programs

In general, this book has always strived to be agnostic about both Python versions and platforms. It's designed to be useful to all Python users. Nevertheless, because Python changes over time and platforms tend to differ in pragmatic ways, I need to describe the specific systems you'll see in action in most examples here.

Python Versions

This fifth edition of this book, and all the program examples in it, are based on Python versions *3.3 and 2.7*. In addition, many of its examples run under prior 3.X and 2.X releases, and notes about the history of language changes in earlier versions are mixed in along the way for users of older Pythons.

Because this text focuses on the core language, however, you can be fairly sure that most of what it has to say won't change very much in *future* releases of Python, as noted earlier. Most of this book applies to *earlier* Python versions, too, except when it does not; naturally, if you try using extensions added after a release you're using, all bets are off. As a rule of thumb, the latest Python is the best Python if you are able to upgrade.

Because this book focuses on the core language, most of it also applies to both *Jython* and *IronPython*, the Java- and .NET-based Python language implementations, as well as other Python implementations such as *Stackless* and *PyPy* (described in Chapter 2). Such alternatives differ mostly in usage details, not language.

Platforms

The examples in this book were run on a *Windows 7 and 8* ultrabook,[3] though Python's portability makes this mostly a moot point, especially in this fundamentals-focused book. You'll notice a few Windows-isms—including command-line prompts, a handful of screenshots, install pointers, and an appendix on the new Windows launcher in 3.3—but this reflects the fact that most Python newcomers will probably get started on this platform, and these can be safely ignored by users of other operating systems.

I also give a few launching details for other platforms like Linux, such as "#!" line use, but as we'll see in Chapter 3 and Appendix B, the 3.3 Windows launcher makes even this a more portable technique.

Fetching This Book's Code

Source code for the book's examples, as well as exercise solutions, can be fetched as a zip file from the book's website at the following address:

http://oreil.ly/LearningPython-5E

This site includes both all the code in this book as well as package usage instructions, so I'll defer to it for more details. Of course, the examples work best in the context of their appearance in this book, and you'll need some background knowledge on running Python programs in general to make use of them. We'll study startup details in Chapter 3, so please stay tuned for information on this front.

Using This Book's Code

The code in my Python books is designed to teach, and I'm glad when it assists readers in that capacity. O'Reilly itself has an official policy regarding reusing the book's examples in general, which I've pasted into the rest of this section for reference:

> This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

3. Mostly under Windows 7, but it's irrelevant to this book. At this writing, Python installs on Windows 8 and runs in its desktop mode, which is essentially the same as Windows 7 without a Start button as I write this (you may need to create shortcuts for former Start button menu items). Support for WinRT/Metro "apps" is still pending. See Appendix A for more details. Frankly, the future of Windows 8 is unclear as I type these words, so this book will be as version-neutral as possible.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Python*, Fifth Edition, by Mark Lutz. Copyright 2013 Mark Lutz, 978-1-4493-5573-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Font Conventions

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

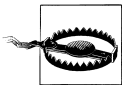
Italic
: Used for email addresses, URLs, filenames, pathnames, and emphasizing new terms when they are first introduced

`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 and some comments in code sections

Indicates a tip, suggestion, or general note relating to the nearby text.



Indicates a warning or caution relating to the nearby text.

You'll also find occasional *sidebars* (delimited by boxes) and *footnotes* (at page end) throughout, which are often optional reading, but provide additional context on the topics being presented. The sidebars in "Why You Will Care: Slices" on page 204, for example, often give example use cases for the subjects being explored.

Book Updates and Resources

Improvements happen (and so do mis^H^H^H typos). Updates, supplements, and corrections (a.k.a. *errata*) for this book will be maintained on the Web, and may be suggested at either the publisher's website or by email. Here are the main coordinates:

Publisher's site: http://oreil.ly/LearningPython-5E
: This site will maintain this edition's official list of book *errata*, and chronicle specific patches applied to the text in reprints. It's also the official site for the book's *examples* as described earlier.

Author's site: http://www.rmi.net/~lutz/about-lp5e.html
: This site will be used to post more *general updates* related to this text or Python itself—a hedge against future changes, which should be considered a sort of virtual appendix to this book.

My publisher also has an email address for comments and technical questions about this book:

bookquestions@oreilly.com

For more information about my publisher's books, conferences, Resource Centers, and the O'Reilly Network, see its general website:

http://www.oreilly.com

For more on my books, see my own book support site:

http://rmi.net/~lutz

Also be sure to search the Web if any of the preceding links become invalid over time; if I could become more clairvoyant, I would, but the Web changes faster than published books.

Acknowledgments

As I write this fifth edition of this book in 2013, it's difficult to not be somewhat retrospective. I have now been using and promoting Python for 21 years, writing books about it for 18, and teaching live classes on it for 16. Despite the passage of time, I'm still regularly amazed at how successful Python has been—in ways that most of us could not possibly have imagined in the early 1990s. So at the risk of sounding like a hopelessly self-absorbed author, I hope you'll pardon a few closing words of history and gratitude here.

The Backstory

My own Python history predates both Python 1.0 and the Web (and goes back to a time when an install meant fetching email messages, concatenating, decoding, and hoping it all somehow worked). When I first discovered Python as a frustrated C++ software developer in 1992, I had no idea what an impact it would have on the next two decades of my life. Two years after writing the first edition of *Programming Python* in 1995 for Python 1.3, I began traveling around the country and world teaching Python to beginners and experts. Since finishing the first edition of *Learning Python* in

1999, I've been an independent Python trainer and writer, thanks in part to Python's phenomenal growth in popularity.

Here's the damage so far. I've now written 13 Python books (5 of this, and 4 of two others), which have together sold some 400,000 units by my data. I've also been teaching Python for over a decade and a half; have taught some 260 Python training sessions in the U.S., Europe, Canada, and Mexico; and have met roughly 4,000 students along the way. Besides propelling me toward frequent flyer utopia, these classes helped me refine this text and my other Python books. Teaching honed the books, and vice versa, with the net result that my books closely parallel what happens in my classes, and can serve as a viable alternative to them.

As for Python itself, in recent years it has grown to become one of the top 5 to 10 most widely used programming languages in the world (depending on which source you cite and when you cite it). Because we'll be exploring Python's status in the first chapter of this book, I'll defer the rest of this story until then.

Python Thanks

Because teaching teaches teachers to teach, this book owes much to my live *classes*. I'd like to thank all the *students* who have participated in my courses during the last 16 years. Along with changes in Python itself, your feedback played a major role in shaping this text; there's nothing quite as instructive as watching 4,000 people repeat the same beginner mistakes live and in person! This book's recent editions owe their training-based changes primarily to recent classes, though every class held since 1997 has in some way helped refine this book. I'd like to thank clients who hosted classes in Dublin, Mexico City, Barcelona, London, Edmonton, and Puerto Rico; such experiences have been one of my career's most lasting rewards.

Because writing teaches writers to write, this book also owes much to its *audience*. I want to thank the countless *readers* who took time to offer suggestions over the last 18 years, both online and in person. Your feedback has also been vital to this book's evolution and a substantial factor in its success, a benefit that seems inherent in the open source world. Reader comments have run the gamut from "You should be banned from writing books" to "God bless you for writing this book"; if consensus is possible in such matters it probably lies somewhere between these two, though to borrow a line from Tolkien: the book is still too short.

I'd also like to express my gratitude to everyone who played a part in this book's *production*. To all those who have helped make this book a solid product over the years —including its editors, formatters, marketers, technical reviewers, and more. And to O'Reilly for giving me a chance to work on 13 book projects; it's been net fun (and only feels a little like the movie *Groundhog Day*).

Additional thanks is due to the entire *Python community*; like most open source systems, Python is the product of many unsung efforts. It's been my privilege to watch

Python grow from a new kid on the scripting languages block to a widely used tool, deployed in some fashion by almost every organization writing software. Technical disagreements aside, that's been an exciting endeavor to be a part of.

I also want to thank my original editor at O'Reilly, the late *Frank Willison*. This book was largely Frank's idea. He had a profound impact on both my career and the success of Python when it was new, a legacy that I remember each time I'm tempted to misuse the word "only."

Personal Thanks

Finally, a few more personal notes of thanks. To the late Carl Sagan, for inspiring an 18-year-old kid from Wisconsin. To my Mother, for courage. To my siblings, for the truths to be found in museum peanuts. To the book *The Shallows*, for a much-needed wakeup call.

To my son Michael and daughters Samantha and Roxanne, for who you are. I'm not quite sure when you grew up, but I'm proud of how you did, and look forward to seeing where life takes you next.

And to my wife Vera, for patience, proofing, Diet Cokes, and pretzels. I'm glad I finally found you. I don't know what the next 50 years hold, but I do know that I hope to spend all of them holding you.

—Mark Lutz, Amongst the Larch, Spring 2013

PART I

Getting Started

CHAPTER 1

A Python Q&A Session

If you've bought 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 of this book will briefly introduce some of the main reasons behind Python's popularity. To begin sculpting a definition of Python, this chapter takes the form of a question-and-answer session, which poses some of the most common questions asked by beginners.

Why Do People Use Python?

Because there are many programming languages available today, this is the usual first question of newcomers. Given that there are roughly 1 million Python users out there at the moment, there really is no way to answer this question with complete accuracy; the choice of development tools is sometimes based on unique constraints or personal preference.

But after teaching Python to roughly 260 groups and over 4,000 students during the last 16 years, I have seen some common themes emerge. 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 scripting 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 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 function programming.

Developer productivity
: Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. 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. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

Program portability
: Most Python programs run unchanged on *all major computer platforms*. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script's 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, including program launches and directory processing, are as portable in Python as they can possibly be.

Support libraries
: 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 application support software. Python's *third-party domain* offers tools for website construction, numeric programming, serial port access, game development, and much more (see ahead for a sampling). The NumPy extension, for instance, has been described as a free and more powerful equivalent to the Matlab numeric programming system.

Component integration
: Python scripts can easily 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. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA. It is not a standalone tool.

Enjoyment
: Because of Python's ease of use and built-in toolset, it can make the act of programming *more pleasure than chore*. Although this may be an intangible benefit, its effect on productivity is an important asset.

Of these factors, the first two (quality and productivity) are probably the most compelling benefits to most Python users, and merit a fuller description.

Software Quality

By design, Python implements a deliberately simple and readable syntax and a highly coherent programming model. As a slogan at a past Python conference attests, 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's a consistently designed system that many find yields surprisingly uniform code.

By philosophy, Python adopts a somewhat minimalist approach. 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 everywhere in the language. Moreover, Python 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.[1]

Beyond such design themes, Python includes tools such as modules and OOP that naturally promote code reusability. And because Python is focused on quality, so too, naturally, are Python programmers.

Developer Productivity

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 layoffs and economic recession, 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 deliberately 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 boosts developer productivity many times beyond the levels supported by traditional languages. That's good news in both boom and bust times, and everywhere the software industry goes in between.

Is Python a "Scripting Language"?

Python is a general-purpose programming language that is often applied in scripting roles. It is commonly defined as an *object-oriented scripting language*—a definition that blends support for OOP with an overall orientation toward scripting roles. If pressed for a one-liner, I'd say that Python is probably better known as a *general-purpose pro-*

1. For a more complete look at the Python philosophy, type the command `import this` at any Python interactive prompt (you'll see how in Chapter 3). This invokes an "Easter egg" hidden in Python—a collection of design principles underlying Python that permeate both the language and its user community. Among them, the acronym EIBTI is now fashionable jargon for the "explicit is better than implicit" rule. These principles are not religion, but are close enough to qualify as a Python motto and creed, which we'll be quoting from often in this book.

gramming language that blends procedural, functional, and object-oriented paradigms—a statement that captures the richness and scope of today's Python.

Still, the term "scripting" seems to have stuck to Python like glue, perhaps as a contrast with larger programming effort required by some other tools. For example, people often use the word "script" instead of "program" to describe a Python code file. In keeping with this tradition, this book uses the terms "script" and "program" interchangeably, with a slight preference for "script" to describe a simpler top-level file and "program" to refer to a more sophisticated multifile application.

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 associations, some of which are more useful than others, when they hear Python labeled as such:

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.

  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. 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. 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 (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 way to think of the term "scripting language" is that it refers to a simple language used for quickly coding tasks. This is especially true when the term is applied to Python, which allows much faster program development than compiled languages like C++. 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 a simple feature set, but it allows programs to scale up in sophistication as needed. Because of that, it is commonly used for quick tactical tasks and longer-term strategic development.

So, is Python a scripting language or not? It depends on whom you ask. In general, the term "scripting" is probably best used to describe the rapid and flexible mode of development that Python supports, rather than a particular application domain.

OK, but What's the Downside?

After using it for 21 years, writing about it for 18, and teaching it for 16, I've found that the only significant universal downside to Python is that, as currently implemented, its *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, for some tasks you may still occasionally need to get "closer to the iron" by using lower-level languages such as these that are more directly mapped to the underlying hardware architecture.

We'll talk about implementation concepts in detail later in this book. In short, the standard implementations of Python today compile (i.e., translate) source code statements to an intermediate format known as *byte code* and then interpret the byte code. Byte code provides portability, as it is a platform-independent format. However, because Python is not normally compiled all the way down to binary machine code (e.g., instructions for an Intel chip), some programs will run more slowly in Python than in a fully compiled language like C. The *PyPy* system discussed in the next chapter can achieve a 10X to 100X speedup on some code by compiling further as your program runs, but it's a separate, alternative implementation.

Whether you will ever *care* about the execution speed difference depends on what kinds of programs you write. Python has been optimized numerous times, 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, since such tasks are immediately dispatched to compiled C code inside the Python interpreter. More fundamentally, Python's speed-of-development gain is often far more important than any speed-of-execution loss, especially given modern computer speeds.

Even at today's CPU speeds, though, there still are some domains that do require optimal execution speeds. Numeric programming and animation, for example, often need at least their core number-crunching components to run at C speed (or better). If you work in such a domain, you can still use Python—simply split off the parts of the application that require optimal speed into *compiled extensions*, and link those into your system for use in Python scripts.

We won't talk about extensions much in this text, but this is really just an instance of the Python-as-control-language role we discussed earlier. A prime example of this dual language strategy is the *NumPy* numeric programming extension for Python; by combining compiled and optimized numeric extension libraries with the Python language, NumPy turns Python into a numeric programming tool that is simultaneously efficient and easy to use. When needed, such extensions provide a powerful optimization tool.

Other Python Tradeoffs: The Intangible Bits

I mentioned that execution speed is the only major downside to Python. That's indeed the case for most Python users, and especially for newcomers. Most people find Python to be easy to learn and fun to use, especially when compared with its contemporaries like Java, C#, and C++. In the interest of full disclosure, though, I should also note up front some more abstract tradeoffs I've observed in my two decades in the Python world—both as an educator and developer.

As an educator, I've sometimes found the *rate of change* in Python and its libraries to be a negative, and have on occasion lamented its *growth* over the years. This is partly because trainers and book authors live on the front lines of such things—it's been my job to teach the language despite its constant change, a task at times akin to chronicling the herding of cats! Still, it's a broadly shared concern. As we'll see in this book, Python's original "keep it simple" motif is today often subsumed by a trend toward more sophisticated solutions at the expense of the learning curve of newcomers. This book's size is indirect evidence of this trend.

On the other hand, by most measures Python is still much simpler than its alternatives, and perhaps only as complex as it needs to be given the many roles it serves today. Its overall coherence and open nature remain compelling features to most. Moreover, not everyone needs to stay up to date with the cutting edge—as Python 2.X's ongoing popularity clearly shows.

As a developer, I also at times question the tradeoffs inherent in Python's *"batteries included" approach* to development. Its emphasis on prebuilt tools can add dependencies (what if a battery you use is changed, broken, or deprecated?), and encourage special-case solutions over general principles that may serve users better in the long run (how can you evaluate or use a tool well if you don't understand its purpose?). We'll see examples of both of these concerns in this book.

For typical users, and especially for hobbyists and beginners, Python's toolset approach is a major asset. But you shouldn't be surprised when you outgrow precoded tools, and can benefit from the sorts of skills this book aims to impart. Or, to paraphrase a proverb: give people a tool, and they'll code for a day; teach them how to build tools, and they'll code for a lifetime. This book's job is more the latter than the former.

As mentioned elsewhere in this chapter, both Python and its toolbox model are also susceptible to downsides common to *open source* projects in general—the potential triumph of the *personal preference* of the few over common usage of the many, and the occasional appearance of *anarchy* and even *elitism*—though these tend to be most grievous on the leading edge of new releases.

We'll return to some of these tradeoffs at the end of the book, after you've learned Python well enough to draw your own conclusions. As an open source system, what Python "is" is up to its users to define. In the end, Python is more popular today than ever, and its growth shows no signs of abating. To some, that may be a more telling metric than individual opinions, both pro and con.

Who Uses Python Today?

At this writing, the best estimate anyone can seem to make of the size of the Python user base is that there are roughly 1 million Python users around the world today (plus or minus a few). This estimate is based on various statistics, like download rates, web statistics, and developer surveys. Because Python is open source, a more exact count is difficult—there are no license registrations to tally. Moreover, Python is automatically included with Linux distributions, Macintosh computers, and a wide range of products and hardware, further clouding the user-base picture.

In general, though, Python enjoys a large user base and a very active developer community. It is generally considered to be in *the top 5 or top 10* most widely used programming languages in the world today (its exact ranking varies per source and date). Because Python has been around for *over two decades* and has been widely used, it is also very stable and robust.

Besides being leveraged by individual users, Python is also being applied in real revenue-generating products by real companies. For instance, among the generally known Python user base:

- *Google* makes extensive use of Python in its web search systems.
- The popular *YouTube* video sharing service is largely written in Python.
- The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
- The *Raspberry Pi* single-board computer promotes Python as its educational language.
- *EVE Online*, a massively multiplayer online game (MMOG) by CCP Games, uses Python broadly.
- The widespread *BitTorrent* peer-to-peer file sharing system began its life as a Python program.
- *Industrial Light & Magic*, *Pixar*, and others use Python in the production of animated movies.
- *ESRI* uses Python as an end-user customization tool for its popular GIS mapping products.
- Google's *App Engine* web development framework uses Python as an application language.
- The *IronPort* email server product uses more than 1 million lines of Python code to do its job.
- *Maya*, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
- The *NSA* uses Python for cryptography and intelligence analysis.
- *iRobot* uses Python to develop commercial and military robotic devices.

- The *Civilization IV* game's customizable scripted events are written entirely in Python.
- The One Laptop Per Child (*OLPC*) project built its user interface and activity model in Python.
- *Netflix* and *Yelp* have both documented the role of Python in their software infrastructures.
- *Intel*, *Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm*, and *IBM* use Python for hardware testing.
- *JPMorgan Chase*, *UBS*, *Getco*, and *Citadel* apply Python to financial market forecasting.
- *NASA*, *Los Alamos*, *Fermilab*, *JPL*, and others use Python for scientific programming tasks.

And so on—though this list is representative, a full accounting is beyond this book's scope, and is almost guaranteed to change over time. For an up-to-date sampling of additional Python users, applications, and software, try the following pages currently at Python's site and Wikipedia, as well as a search in your favorite web browser:

- Success stories: *http://www.python.org/about/success*
- Application domains: *http://www.python.org/about/apps*
- User quotes: *http://www.python.org/about/quotes*
- Wikipedia page: *http://en.wikipedia.org/wiki/List_of_Python_software*

Probably the only common thread among the companies using Python today is that Python is used all over the map, in terms of application domains. Its general-purpose nature makes it applicable to almost all fields, not just one. In fact, it's safe to say that virtually every substantial organization writing software is using Python, whether for short-term tactical tasks, such as testing and administration, or for long-term strategic product development. Python has proven to work well in both modes.

What Can I Do with Python?

In addition to being a well-designed programming language, Python is useful for accomplishing real-world tasks—the sorts of things developers do day in and day out. It's commonly used in a variety of domains, as a tool for scripting other components and implementing standalone programs. In fact, as a general-purpose language, Python's roles are virtually unlimited: you can use it for everything from website development and gaming to robotics and spacecraft control.

However, the most common Python roles currently seem to fall into a few broad categories. The next few sections describe some of Python's most common applications today, as well as tools used in each domain. We won't be able to explore the tools

mentioned here in any depth—if you are interested in any of these topics, see the Python website or other resources for more details.

Systems Programming

Python's built-in interfaces to operating-system services make it ideal for writing portable, maintainable system-administration tools and utilities (sometimes called *shell tools*). Python programs can search files and directory trees, launch other programs, do parallel processing with processes and threads, and so on.

Python's standard library comes with POSIX bindings and support for all the usual OS tools: environment variables, files, sockets, pipes, processes, multiple threads, regular expression pattern matching, command-line arguments, standard stream interfaces, shell-command launchers, filename expansion, zip file utilities, XML and JSON parsers, CSV file handlers, and more. 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 major Python platforms. The *Stackless* Python implementation, described in Chapter 2 and used by *EVE Online*, also offers advanced solutions to multiprocessing requirements.

GUIs

Python's simplicity and rapid turnaround also make it a good match for graphical user interface programming on the desktop. Python comes with a standard object-oriented interface to the Tk GUI API called *tkinter* (*Tkinter* in 2.X) that allows Python programs to implement portable GUIs with a native look and feel. Python/tkinter GUIs run unchanged on Microsoft Windows, X Windows (on Unix and Linux), and the Mac OS (both Classic and OS X). A free extension package, *PMW*, adds advanced widgets to the tkinter toolkit. In addition, the *wxPython* GUI API, based on a C++ library, offers an alternative toolkit for constructing portable GUIs in Python.

Higher-level toolkits such as *Dabo* are built on top of base APIs such as wxPython and tkinter. With the proper library, you can also use GUI support in other toolkits in Python, such as *Qt* with PyQt, *GTK* with PyGTK, *MFC* with PyWin32, *.NET* with IronPython, and *Swing* with Jython (the Java version of Python, described in Chapter 2) or JPype. For applications that run in web browsers or have simple interface requirements, both Jython and Python web frameworks and server-side CGI scripts, described in the next section, provide additional user interface options.

Internet Scripting

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

email; fetch web pages by URLs; parse the HTML of fetched web pages; communicate over XML-RPC, SOAP, and Telnet; and more. Python's libraries make these tasks remarkably simple.

In addition, a large collection of third-party tools are available on the Web for doing Internet programming in Python. For instance, the *HTMLGen* system generates HTML files from Python class-based descriptions, the *mod_python* package runs Python efficiently within the Apache web server and supports server-side templating with its Python Server Pages, and the Jython system provides for seamless Python/Java integration and supports coding of server-side applets that run on clients.

In addition, full-blown web development framework packages for Python, such as *Django*, *TurboGears*, *web2py*, *Pylons*, *Zope*, and *WebWare*, support quick construction of full-featured and production-quality websites with Python. Many of these include features such as object-relational mappers, a Model/View/Controller architecture, server-side scripting and templating, and AJAX support, to provide complete and enterprise-level web development solutions.

More recently, Python has expanded into rich Internet applications (RIAs), with tools such as *Silverlight* in *IronPython*, and *pyjs* (a.k.a. *pyjamas*) and its Python-to-JavaScript compiler, AJAX framework, and widget set. Python also has moved into cloud computing, with *App Engine*, and others described in the database section ahead. Where the Web leads, Python quickly follows.

Component Integration

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

Tools such as the *SWIG* and *SIP* code generators can automate much of the work needed to link compiled components into Python for use in scripts, and the *Cython* system allows coders to mix Python and C-like code. Larger frameworks, such as Python's *COM* support on Windows, the Jython *Java*-based implementation, and the IronPython .NET-based implementation provide alternative ways to script components. On Windows, for example, Python scripts can use frameworks to script Word and Excel, access *Silverlight*, and much more.

Database Programming

For traditional database demands, there are Python interfaces to all commonly used relational database systems—Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL,

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 systems. For instance, because the vendor interfaces implement the portable API, a script written to work with the free MySQL system will work largely unchanged on other systems (such as Oracle); all you generally have to do is replace the underlying vendor interface. The in-process *SQLite* embedded SQL database engine is a standard part of Python itself since 2.5, supporting both prototyping and basic program storage needs.

In the non-SQL department, Python's standard `pickle` module provides a simple object persistence system—it allows programs to easily save and restore entire Python objects to files and file-like objects. On the Web, you'll also find third-party open source systems named *ZODB* and *Durus* that provide complete object-oriented database systems for Python scripts; others, such as *SQLObject* and *SQLAlchemy*, that implement object relational mappers (ORMs), which graft Python's class model onto relational tables; and *PyMongo*, an interface to *MongoDB*, a high-performance, non-SQL, open source *JSON*-style document database, which stores data in structures very similar to Python's own lists and dictionaries, and whose text may be parsed and created with Python's own standard library `json` module.

Still other systems offer more specialized ways to store data, including the datastore in Google's *App Engine*, which models data with Python classes and provides extensive scalability, as well as additional emerging cloud storage options such as *Azure*, *PiCloud*, *OpenStack*, and *Stackato*.

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 compiled language such as C or C++ for delivery. Unlike some prototyping tools, Python doesn't require a complete rewrite once the prototype has solidified. Parts of the system that don't require the efficiency of a language such as C++ can remain coded in Python for ease of maintenance and use.

Numeric and Scientific Programming

Python is also heavily 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 existing code written in traditional compiled languages such as FORTRAN or C++.

Additional numeric tools for Python support animation, 3D visualization, parallel processing, and so on. The popular *SciPy* and *ScientificPython* extensions, for example, provide additional libraries of scientific programming tools and use NumPy as a core component. The *PyPy* implementation of Python (discussed in Chapter 2) has also gained traction in the numeric domain, in part because heavily algorithmic code of the sort that's common in this domain can run dramatically faster in PyPy—often 10X to 100X quicker.

And More: Gaming, Images, Data Mining, Robots, Excel...

Python is commonly applied in more domains than can be covered here. For example, you'll find tools that allow you to use Python to do:

- Game programming and multimedia with *pygame*, *cgkit*, *pyglet*, *PySoy*, *Panda3D*, and others
- Serial port communication on Windows, Linux, and more with the *PySerial* extension
- Image processing with *PIL* and its newer *Pillow* fork, *PyOpenGL*, *Blender*, *Maya*, and more
- Robot control programming with the *PyRo* toolkit
- Natural language analysis with the *NLTK* package
- Instrumentation on the *Raspberry Pi* and *Arduino* boards
- Mobile computing with ports of Python to the Google *Android* and Apple *iOS* platforms
- Excel spreadsheet function and macro programming with the *PyXLL* or *DataNitro* add-ins
- Media file content and metadata tag processing with *PyMedia*, *ID3*, *PIL/Pillow*, and more
- Artificial intelligence with the *PyBrain* neural net library and the *Milk* machine learning toolkit
- Expert system programming with *PyCLIPS*, *Pyke*, *Pyrolog*, and *pyDatalog*
- Network monitoring with *zenoss*, written in and customized with Python
- Python-scripted design and modeling with *PythonCAD*, *PythonOCC*, *FreeCAD*, and others
- Document processing and generation with *ReportLab*, *Sphinx*, *Cheetah*, *PyPDF*, and so on
- Data visualization with *Mayavi, matplotlib*, *VTK*, *VPython*, and more
- XML parsing with the `xml` library package, the `xmlrpclib` module, and third-party extensions
- JSON and CSV file processing with the `json` and `csv` modules

- Data mining with the *Orange* framework, the *Pattern* bundle, *Scrapy*, and custom code

You can even play solitaire with the *PySolFC* program. And of course, you can always code custom Python scripts in less buzzword-laden domains to perform day-to-day system administration, process your email, manage your document and media libraries, and so on. You'll find links to the support in many fields at the PyPI website, and via web searches (search Google or *http://www.python.org* for links).

Though of broad practical use, many of these specific domains are largely just instances of Python's component integration role in action again. Adding it as a frontend to libraries of components written in a compiled language such as C makes Python useful for scripting in a wide variety of domains. As a general-purpose language that supports integration, Python is widely applicable.

How Is Python Developed and Supported?

As a popular open source system, Python enjoys a large and active development community that responds to issues and develops enhancements with a speed that many commercial software developers might find remarkable. Python developers coordinate work online with a source-control system. Changes are developed per a formal protocol, which includes writing a *PEP* (Python Enhancement Proposal) or other document, and extensions to Python's regression testing system. In fact, modifying Python today is roughly as involved as changing commercial software—a far cry from Python's early days, when an email to its creator would suffice, but a good thing given its large user base today.

The *PSF* (Python Software Foundation), a formal nonprofit group, organizes conferences and deals with intellectual property issues. Numerous Python conferences are held around the world; O'Reilly's *OSCON* and the PSF's *PyCon* are the largest. The former of these addresses multiple open source projects, and the latter is a Python-only event that has experienced strong growth in recent years. PyCon 2012 and 2013 reached *2,500 attendees* each; in fact, PyCon 2013 had to cap its limit at this level after a surprise sell-out in 2012 (and managed to grab wide attention on both technical and nontechnical grounds that I won't chronicle here). Earlier years often saw attendance double—from 586 attendees in 2007 to over 1,000 in 2008, for example—indicative of Python's growth in general, and impressive to those who remember early conferences whose attendees could largely be served around a single restaurant table.

Open Source Tradeoffs

Having said that, it's important to note that while Python enjoys a vigorous development community, this comes with inherent tradeoffs. Open source software can also appear chaotic and even resemble *anarchy* at times, and may not always be as smoothly implemented as the prior paragraphs might imply. Some changes may still manage to

defy official protocols, and as in all human endeavors, mistakes still happen despite the process controls (Python 3.2.0, for instance, came with a broken console `input` function on Windows).

Moreover, open source projects exchange commercial interests for the *personal preferences* of a current set of developers, which may or may not be the same as yours—you are not held hostage by a company, but you are at the mercy of those with spare time to change the system. The net effect is that open source software evolution is often driven by the few, but imposed on the many.

In practice, though, these tradeoffs impact those on the "bleeding" edge of new releases much more than those using established versions of the system, including prior releases in both Python 3.X and 2.X. If you kept using classic classes in Python 2.X, for example, you were largely immune to the *explosion* of class functionality and change in new-style classes that occurred in the early-to-mid 2000s. Though these become mandatory in 3.X (along with much more), many 2.X users today still happily skirt the issue.

What Are Python's Technical Strengths?

Naturally, this is a developer's question. If you don't already have a programming background, the language in the next few sections may be a bit baffling—don't worry, we'll explore all of these terms in more detail as we proceed through this book. For developers, though, here is a quick introduction to some of Python's top technical features.

It's Object-Oriented and Functional

Python is an object-oriented language, from the ground up. 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.

Besides serving as a powerful code structuring and reuse device, Python's OOP nature makes it ideal as a *scripting tool* for other object-oriented systems languages. For example, with the appropriate glue code, Python programs can subclass (specialize) classes implemented in C++, Java, and C#.

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 preclude design phases.

In addition to its original *procedural* (statement-based) and *object-oriented* (class-based) paradigms, Python in recent years has acquired 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. These can serve as both complement and alternative to its OOP tools.

It's Free

Python is completely free to use and distribute. As with other open source software, such as Tcl, Perl, 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. In fact, you can even sell Python's source code, if you are so inclined.

But don't get the wrong idea: "free" doesn't mean "unsupported." On the contrary, the Python online community responds to user queries with a speed that most commercial software help desks would do well to try to emulate. Moreover, because Python comes with complete source code, it empowers developers, leading to the creation of a large team of implementation experts. Although studying or changing a programming language's implementation isn't everyone's idea of fun, it's comforting to know that you can do so if you need to. 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.

As mentioned earlier, Python development is performed by a community that largely coordinates its efforts over the Internet. It consists of Python's original creator—*Guido van Rossum*, the officially anointed Benevolent Dictator for Life (*BDFL*) of Python—plus a supporting cast of thousands. Language changes must follow a formal enhancement procedure and be scrutinized by both other developers and the BDFL. This tends to make Python more conservative with changes than some other languages and systems. While the Python 3.X/2.X split broke with this tradition soundly and deliberately, it still holds generally true within each Python line.

It's Portable

The standard implementation of Python is written in portable ANSI C, and it compiles and runs on virtually every major platform currently in use. For example, Python programs run today on everything from PDAs to supercomputers. As a partial list, Python is available on:

- Linux and Unix systems
- Microsoft Windows (all modern flavors)
- Mac OS (both OS X and Classic)
- BeOS, OS/2, VMS, and QNX
- Real-time systems such as VxWorks
- Cray supercomputers and IBM mainframes

- PDAs running Palm OS, PocketPC, and Linux
- Cell phones running Symbian OS, and Windows Mobile
- Gaming consoles and iPods
- Tablets and smartphones running Google's Android and Apple's iOS
- And more

Like the language interpreter itself, the standard library modules that ship with Python are implemented to be as portable across platform boundaries as possible. Further, Python programs are automatically compiled to portable byte code, which runs the same on any platform with a compatible version of Python installed (more on this in the next chapter).

What that means is that Python programs using the core language and standard libraries run the same on Linux, Windows, and most other systems with a Python interpreter. Most Python ports also contain platform-specific extensions (e.g., COM support on Windows), but the core Python language and libraries work the same everywhere. As mentioned earlier, Python also includes an interface to the Tk GUI toolkit called tkinter (Tkinter in 2.X), which allows Python programs to implement full-featured graphical user interfaces that run on all major GUI desktop platforms without program changes.

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; it 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. 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, and most can grow and shrink on demand. As you'll learn, Python keeps track of low-level memory details 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, provide additional ways to meet many 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, they're both flexible and easy to use. For instance, built-in objects can grow and shrink on demand, can be arbitrarily nested to represent complex information, and more.

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 matching to networking. 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 COM, imaging, numeric programming, XML, database access, and much more.

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

It's Mixable

Python programs can easily be "glued" to components written in other languages in a variety of ways. For example, Python's C API lets C programs call and be called by Python programs flexibly. That means you can add functionality to the Python system as needed, and use Python programs within other environments or systems.

Mixing Python with libraries coded in languages such as C or C++, for instance, makes it an easy-to-use frontend language and customization tool. As mentioned earlier, this also makes Python good at rapid prototyping—systems may be implemented in Python first, to leverage its speed of development, and later moved to C for delivery, one piece at a time, according to performance demands.

It's Relatively Easy to Use

Compared to alternatives like C++, Java, and C#, Python programming seems astonishingly simple to most observers. To run a Python program, you simply type it and run it. There are no intermediate compile and link steps, like there are 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 in other tools, Python programs are simpler, smaller, and more flexible than equivalent programs in other popular languages.

It's Relatively Easy to Learn

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 an experienced programmer, you can expect to be coding small-scale Python programs in a matter of days, and may be able to pick up some limited portions of the language in just hours—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 true 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.

That's good news for professional developers seeking to learn the language to use on the job, as well as for end users of systems that expose a Python layer for customization or control. Today, many systems rely on the fact that end users can learn enough Python to tailor their Python customization code onsite, with little or no support. Moreover, Python has spawned a large group of users who program for fun instead of career, and may never need full-scale software development skills. Although Python does have advanced programming tools, its core language essentials will still seem relatively simple to beginners and gurus alike.

It's Named After Monty Python

OK, this isn't quite a technical strength, but it does seem to be a surprisingly well-kept secret in the Python world that I wish to expose up front. Despite all the reptiles on Python books and icons, the truth is that Python is named after the British comedy group *Monty Python*—makers of the 1970s BBC comedy series *Monty Python's Flying Circus* and a handful of later full-length films, including *Monty Python and the Holy Grail*, that are still widely popular today. Python's original creator was a fan of Monty Python, as are many software developers (indeed, there seems to be a sort of symmetry between the two fields...).

This legacy inevitably adds a humorous quality to Python code examples. For instance, the traditional "foo" and "bar" for generic variable names become "spam" and "eggs" in the Python world. The occasional "Brian," "ni," and "shrubbery" likewise owe their appearances to this namesake. It even impacts the Python community at large: some events at Python conferences are regularly billed as "The Spanish Inquisition."

All of this is, of course, very funny if you are familiar with the shows, but less so otherwise. You don't need to be familiar with Monty Python's work to make sense of examples that borrow references from it, including many you will see in this book, but at least you now know their root. (Hey—I've warned you.)

How Does Python Stack Up to Language X?

Finally, to place it in the context of what you may already know, people sometimes compare Python to languages such as Perl, Tcl, and Java. This section summarizes common consensus in this department.

I want to note up front that I'm not a fan of winning by disparaging the competition—it doesn't work in the long run, and that's not the goal here. Moreover, this is not a zero sum game—most programmers will use many languages over their careers. Nevertheless, programming tools present choices and tradeoffs that merit consideration. After all, if Python didn't offer something over its alternatives, it would never have been used in the first place.

We talked about performance tradeoffs earlier, so here we'll focus on functionality. While other languages are also useful tools to know and use, many people find that Python:

- Is more powerful than *Tcl*. Python's strong support for "programming in the large" makes it applicable to the development of larger systems, and its library of application tools is broader.
- Is more readable than *Perl*. Python has a clear syntax and a simple, coherent design. This in turn makes Python more reusable and maintainable, and helps reduce program bugs.
- Is simpler and easier to use than *Java* and *C#*. Python is a scripting language, but Java and C# both inherit much of the complexity and syntax of larger OOP systems languages like C++.
- Is simpler and easier to use than *C++*. Python code is simpler than the equivalent C++ and often one-third to one-fifth as large, though as a scripting language, Python sometimes serves different roles.
- Is simpler and higher-level than *C*. Python's detachment from underlying hardware architecture makes code less complex, better structured, and more approachable than C, C++'s progenitor.

- Is more powerful, general-purpose, and cross-platform than *Visual Basic*. Python is a richer language that is used more widely, and its open source nature means it is not controlled by a single company.
- Is more readable and general-purpose than *PHP*. Python is used to construct websites too, but it is also applied to nearly every other computer domain, from robotics to movie animation and gaming.
- Is more powerful and general-purpose than *JavaScript*. Python has a larger toolset, and is not as tightly bound to web development. It's also used for scientific modeling, instrumentation, and more.
- Is more readable and established than *Ruby*. Python syntax is less cluttered, especially in nontrivial code, and its OOP is fully optional for users and projects to which it may not apply.
- Is more mature and broadly focused than *Lua*. Python's larger feature set and more extensive library support give it a wider scope than Lua, an embedded "glue" language like Tcl.
- Is less esoteric than *Smalltalk*, *Lisp*, and *Prolog*. Python has the dynamic flavor of languages like these, but also has a traditional syntax accessible to both developers and end users of customizable systems.

Especially for programs that do more than scan text files, and that might have to be read in the future by others (or by you!), many people find that Python fits the bill better than any other scripting or programming language available today. Furthermore, unless your application requires peak performance, Python is often a viable alternative to systems development languages such as C, C++, and Java: Python code can often achieve the same goals, but will be much less difficult to write, debug, and maintain.

Of course, your author has been a card-carrying Python evangelist since 1992, so take these comments as you may (and other languages' advocates' mileage may vary arbitrarily). They do, however, reflect the common experience of many developers who have taken time to explore what Python has to offer.

Chapter Summary

And that concludes the "hype" portion of this book. In this chapter, we've explored some of the reasons that people pick Python for their programming tasks. We've also seen how it is applied and looked at a representative sample of who is using it today. My 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 we've glossed over here.

The next two chapters begin our technical introduction to the language. In them, we'll explore ways to run Python programs, peek at Python's byte code execution model, and introduce the basics of module files for saving code. The goal 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 edition of the book, we will be closing each chapter with a quick open-book quiz about the material presented herein to help you review the 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 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. 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?
5. What's the significance of the Python `import this` statement?
6. Why does "spam" show up in so many Python examples in books and on the Web?
7. What is your favorite color?

Test Your Knowledge: Answers

How did you do? Here are the answers I came up with, though there may be multiple solutions to some quiz questions. Again, even if you're sure of your answer, I encourage you to look at mine for additional context. See the chapter's text 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.
2. Google, Industrial Light & Magic, CCP Games, Jet Propulsion Labs, Maya, ESRI, 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: 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 are available for number-crunching parts of an application.

4. You can use Python for nearly anything you can do with a computer, from website development and gaming to robotics and spacecraft control.
5. This was mentioned in a footnote: `import this` triggers an Easter egg inside Python that displays some of the design philosophies underlying the language. You'll learn how to run this statement in the next chapter.
6. "Spam" is a reference from a famous Monty Python skit in which people trying to order food in a cafeteria are drowned out by a chorus of Vikings singing about spam. Oh, and it's also a common variable name in Python scripts...
7. Blue. No, yellow! (See the prior answer.)

Python Is Engineering, Not Art

When Python first emerged on the software scene in the early 1990s, it spawned what is now something of a classic conflict between its proponents and those of another popular scripting language, Perl. Personally, I think the debate is tired and unwarranted today—developers are smart enough to draw their own conclusions. Still, this is one of the most common topics I'm asked about on the training road, and underscores one of the main reasons people choose to use Python; it seems fitting to say a few brief words about it here.

The short story is this: *you can do everything in Python that you can in Perl, but you can read your code after you do it*. That's it—their domains largely overlap, but Python is more focused on producing readable code. For many, the enhanced readability of Python translates to better code reusability and maintainability, making Python a better choice for programs that will not be written once and thrown away. Perl code is easy to write, but can be difficult to read. Given that most software has a lifespan much longer than its initial creation, many see Python as the more effective tool.

The somewhat longer story reflects the backgrounds of the designers of the two languages. *Python* originated with a mathematician by training, who seems to have naturally produced an orthogonal language with a high degree of uniformity and coherence. *Perl* was spawned by a linguist, who created a programming tool closer to natural language, with its context sensitivities and wide variability. As a well-known Perl motto states, *there's more than one way to do it*. Given this mindset, both the Perl language and its user community have historically encouraged untethered freedom of expression when writing code. One person's Perl code can be radically different from another's. In fact, writing unique, tricky code is often a source of pride among Perl users.

But as anyone who has done any substantial code maintenance should be able to attest, *freedom of expression is great for art, but lousy for engineering*. In engineering, we need a minimal feature set and predictability. In engineering, freedom of expression can lead to maintenance nightmares. As more than one Perl user has confided to me, the result of too much freedom is often code that is much easier to rewrite from scratch than to modify. This is clearly less than ideal.

Consider this: when people create a painting or a sculpture, they do so largely for themselves; the prospect of someone else changing their work later doesn't enter into it. This is a critical difference between art and engineering. When people write *software*, they are not writing it for themselves. In fact, they are not even writing primarily for the computer. Rather, good programmers know that code is written for the next human being who has to read it in order to maintain or reuse it. If that person cannot understand the code, it's all but useless in a realistic development scenario. In other words, programming is not about being clever and obscure—*it's about how clearly your program communicates its purpose*.

This readability focus is where many people find that Python most clearly differentiates itself from other scripting languages. Because Python's syntax model almost *forces* the creation of readable code, Python programs lend themselves more directly to the full software development cycle. And because Python emphasizes ideas such as limited interactions, code uniformity, and feature consistency, it more directly fosters code that can be used long after it is first written.

In the long run, Python's focus on *code quality* in itself boosts programmer productivity, as well as programmer satisfaction. Python programmers can be wildly creative, too, of course, and as we'll see, the language does offer multiple solutions for some tasks—sometimes even more than it should today, an issue we'll confront head-on in this book too. In fact, this sidebar can also be read as a *cautionary tale*: quality turns out to be a fragile state, one that depends as much on *people* as on technology. Python has historically encouraged good engineering in ways that other scripting languages often did not, but the rest of the quality story is up to you.

At least, that's some of the common consensus among many people who have adopted Python. You should judge such claims for yourself, of course, by learning what Python has to offer. To help you get started, let's move on to the next chapter.

CHAPTER 2

How Python Runs Programs

This chapter and the next take a quick look at program execution—how you launch code, and how Python runs it. In this chapter, we'll study how the Python interpreter executes programs in general. Chapter 3 will then show you how to get 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 platform you work on, so more advanced readers should feel free to skip parts not relevant to their intended use. 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 as "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 learn how to write it.

Introducing the Python Interpreter

So far, I've mostly been talking about Python as a programming language. But, as currently implemented, it's also a software package 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 software logic between your code and the computer hardware on your machine.

When the Python package is installed on your machine, 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 must install a Python interpreter on your computer.

Python installation details vary by platform and are covered in more depth in Appendix A. In short:

- Windows users fetch and run a self-installing executable file that puts Python on their machines. Simply double-click and say Yes or Next at all prompts.
- Linux and Mac OS X users probably already have a usable Python preinstalled on their computers—it's a standard component on these platforms today.
- Some Linux and Mac OS X users (and most Unix users) compile Python from its full source code distribution package.
- Linux users can also find RPM files, and Mac OS X users can find various Mac-specific installation packages.
- Other platforms have installation techniques relevant to those platforms. For instance, Python is available on cell phones, tablets, game consoles, and iPods, but installation details vary widely.

Python itself may be fetched from the downloads page on its main website, *http://www.python.org*. It may also be found through various other distribution channels. Keep in mind that you should always check to see whether Python is already present before installing it. If you're working on Windows 7 and earlier, you'll usually find Python in the Start menu, as captured in Figure 2-1; we'll discuss the menu options shown here in the next chapter. On Unix and Linux, Python probably lives in your */usr* directory tree.

Because installation details are so platform-specific, we'll postpone the rest of this story here. For more details on the installation process, consult Appendix A. For the purposes of this chapter and the next, I'll assume that you've got Python ready to go.

Program Execution

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

The Programmer's View

In its simplest form, a Python program is just a text file containing Python statements. For example, the following file, named *script0.py*, is one of the simplest Python scripts I could dream up, but it passes for a fully functional Python program:

```
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. Don't worry about the syntax of this code yet—for this chapter, we're interested only

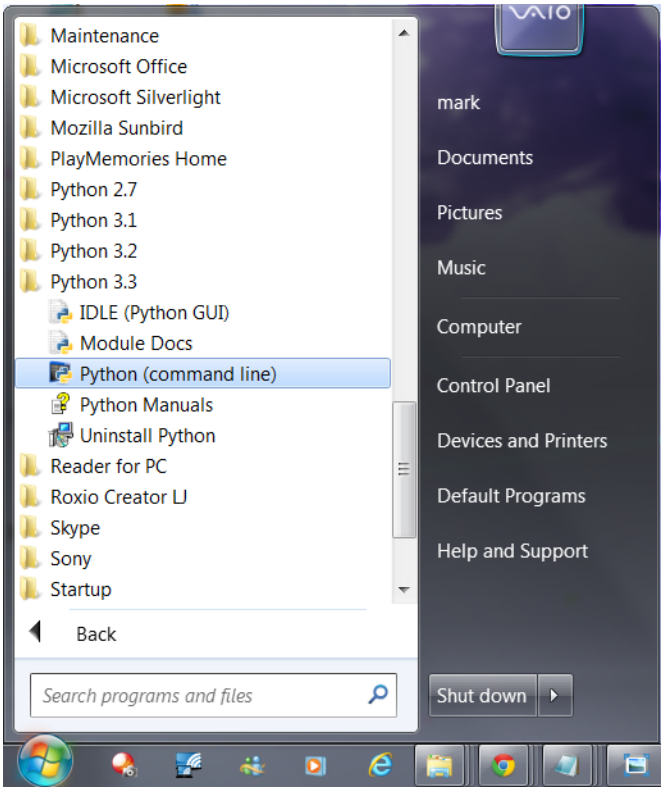


Figure 2-1. When installed on Windows 7 and earlier, this is how Python shows up in your Start button menu. This can vary across releases, but IDLE starts a development GUI, and Python starts a simple interactive session. Also here are the standard manuals and the PyDoc documentation engine (Module Docs). See Chapter 3 and Appendix A for pointers on Windows 8 and other platforms.

in getting it to run. I'll explain the `print` statement, and why you can raise 2 to the power 100 in Python without overflowing, in the next parts of this book.

You can create such a file of statements with any text editor you like. 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 shell command lines, by clicking their icons, from within IDEs, and with other standard 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—by default, usually in the same window you were in when you ran the program:

```
hello world
1267650600228229401496703205376
```

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

```
C:\code> python 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 gory 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 win any programming awards 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 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, a basic understanding of the runtime structure of Python can help you grasp 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 "byte code" and then routed to something called a "virtual machine."

Byte code compilation

Internally, and almost completely hidden from you, when you execute a program Python first compiles your *source code* (the statements in your file) into a format known as *byte code*. Compilation is simply a translation step, and byte code is a lower-level, platform-independent representation of your source code. Roughly, Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This byte code translation is performed to speed execution—byte code 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 process has write access on your machine, it will store the byte code of your programs in files that end with a *.pyc* extension (".pyc" means compiled ".py" source). Prior to Python 3.2, you will see these files show up on your computer after you've run a few programs alongside the corresponding source code files—that is, in the *same* directories. For instance, you'll notice a *script.pyc* after importing a *script.py*.

In 3.2 and later, Python instead saves its *.pyc* byte code 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., *script.cpython-33.pyc*). The new *__pycache__* subdirectory helps to avoid clutter, and the new naming convention for byte code files prevents different Python versions installed on the same computer from overwriting each other's saved byte code. We'll study these byte code file models in more detail in Chapter 22, though they are automatic and irrelevant to most Python programs, and are free to vary among the alternative Python implementations described ahead.

In both models, Python saves byte code like this as a startup speed optimization. The next time you run your program, Python will load the *.pyc* files and skip the compilation step, as long as you haven't changed your source code since the byte code was last saved, and aren't running with a different Python than the one that created the byte code. It works like this:

- *Source changes*: Python automatically checks the last-modified timestamps of source and byte code files to know when it must recompile—if you edit and resave your source code, byte code is automatically re-created the next time your program is run.
- *Python versions*: Imports also check to see if the file must be recompiled because it was created by a different Python version, using either a "magic" version number in the byte code file itself in 3.2 and earlier, or the information present in byte code filenames in 3.2 and later.

The result is that both source code changes and differing Python version numbers will trigger a new byte code file. If Python cannot write the byte code files to your machine, your program still works—the byte code is generated in memory and simply discarded on program exit. However, because *.pyc* files speed startup time, you'll want to make sure they are written for larger programs. Byte code files are also one way to ship Python programs—Python is happy to run a program if all it can find are *.pyc* files, even if the original *.py* source files are absent. (See "Frozen Binaries" on page 39 for another shipping option.)

Finally, keep in mind that byte code is saved in files only for files that are *imported*, not for the top-level files of a program that are only run as scripts (strictly speaking, it's an import optimization). We'll explore import basics in Chapter 3, and take a deeper look at imports in Part V. Moreover, a given file is only imported (and possibly compiled) *once* per program run, and byte code is also never saved for code typed at the *interactive prompt*—a programming mode we'll learn about in Chapter 3.

The Python Virtual Machine (PVM)

Once your program has been compiled to byte code (or the byte code has been loaded from existing *.pyc* files), it is shipped off for execution to something generally known as the Python Virtual Machine (PVM, for the more acronym-inclined among you). The

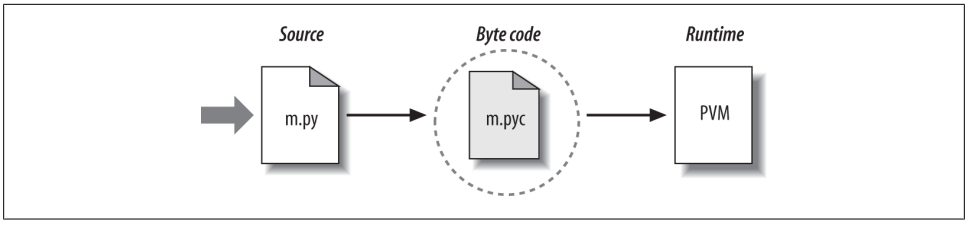


Figure 2-2. Python's traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

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 byte code instructions, one by one, to carry out their operations. The PVM is the runtime engine of Python; it's always present as part of the Python system, and it's the component that truly runs your scripts. Technically, it's just the last step of what is called the "Python interpreter."

Figure 2-2 illustrates the runtime structure described here. Keep in mind that all of this complexity is deliberately hidden from Python programmers. Byte code 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.

Performance implications

Readers with a background in fully compiled languages such as C and C++ might notice a few 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 byte code is not binary machine code (e.g., instructions for an Intel or ARM chip). Byte code is a Python-specific representation.

This is 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 byte code, and byte code 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. See Chapter 1 for more on Python performance tradeoffs.

Development implications

Another ramification of Python's execution model is that there is really no distinction between the development and execution environments. That is, 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 may 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 onsite without needing to have or compile the entire 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 occur before execution in more static languages, but happen as programs execute in Python. As we'll see, this makes for a much more dynamic programming experience than that to which some readers may be accustomed.

Execution Model Variations

Now that we've studied the internal execution flow described in the prior section, I should note that it reflects the standard implementation of Python today but is not really a requirement of the Python language itself. Because of that, the execution model is prone to changing with time. In fact, there are already a few systems that modify the picture in Figure 2-2 somewhat. Before moving on, let's briefly explore the most prominent of these variations.

Python Implementation Alternatives

Strictly speaking, as this book edition is being written, there are at least five implementations of the Python language—*CPython*, *Jython*, *IronPython*, *Stackless*, and *PyPy*. Although there is much cross-fertilization of ideas and work between these Pythons, each is a separately installed software system, with its own developers and user base. Other potential candidates here include the *Cython* and *Shed Skin* systems, but they are discussed later as optimization tools because they do not implement the standard Python language (the former is a Python/C mix, and the latter is implicitly statically typed).

In brief, *CPython* is the standard implementation, and the system that most readers will wish to use (if you're not sure, this probably includes you). This is also the version used in this book, though the core Python language presented here is almost entirely the same in the alternatives. All the other Python implementations have specific pur-

poses and roles, though they can often serve in most of CPython's capacities too. All implement the same Python language but execute programs in different ways.

For example, *PyPy* is a drop-in replacement for CPython, which can run most programs much quicker. Similarly, *Jython* and *IronPython* are completely independent implementations of Python that compile Python source for different runtime architectures, to provide direct access to Java and .NET components. It is also possible to access Java and .NET software from standard CPython programs—*JPype* and *Python for .NET* systems, for instance, allow standard CPython code to call out to Java and .NET components. Jython and IronPython offer more complete solutions, by providing full implementations of the Python language.

Here's a quick rundown on the most prominent Python implementations available today.

CPython: The standard

The original, and standard, implementation of Python is usually called CPython when you want to contrast it with the other options (and just plain "Python" otherwise). This name comes from the fact that it is coded in portable ANSI C language code. This is the Python that you fetch from *http://www.python.org*, get with the ActivePython and Enthought distributions, and have automatically on most Linux and Mac OS X machines. If you've found a preinstalled version of Python on your machine, it's probably CPython, unless your company or organization is using Python in more specialized ways.

Unless you want to script Java or .NET applications with Python or find the benefits of Stackless or PyPy compelling, you probably want to use the standard CPython system. Because it is the reference implementation of the language, it tends to run the fastest, be the most complete, and be more up-to-date and robust than the alternative systems. Figure 2-2 reflects CPython's runtime architecture.

Jython: Python for Java

The Jython system (originally known as JPython) is an alternative implementation of the Python language, targeted for integration with the Java programming language. Jython consists of Java classes that compile Python source code to Java byte code and then route the resulting byte code to the Java Virtual Machine (JVM). Programmers still code Python statements in *.py* text files as usual; the Jython system essentially just replaces the rightmost two bubbles in Figure 2-2 with Java-based equivalents.

Jython's goal is to allow Python code to script Java applications, much as CPython allows Python to script C and C++ components. Its integration with Java is remarkably seamless. Because Python code is translated to Java byte code, it looks and feels like a true Java program at runtime. Jython scripts can serve as web applets and servlets, build Java-based GUIs, and so on. Moreover, Jython includes integration support that allows Python code to import and use Java classes as though they were coded in Python, and

Java code to run Python code as an embedded language. Because Jython is slower and less robust than CPython, though, it is usually seen as a tool of interest primarily to Java developers looking for a scripting language to serve as a frontend to Java code. See Jython's website *http://jython.org* for more details.

IronPython: Python for .NET

A third implementation of Python, and newer than both CPython and Jython, IronPython is designed to allow Python programs to integrate with applications coded to work with Microsoft's .NET Framework for Windows, as well as the Mono open source equivalent for Linux. .NET and its C# programming language runtime system are designed to be a language-neutral object communication layer, in the spirit of Microsoft's earlier COM model. IronPython allows Python programs to act as both client and server components, gain accessibility both to and from other .NET languages, and leverage .NET technologies such as the *Silverlight* framework from their Python code.

By implementation, IronPython is very much like Jython (and, in fact, was developed by the same creator)—it replaces the last two bubbles in Figure 2-2 with equivalents for execution in the .NET environment. Also like Jython, IronPython has a special focus—it is primarily of interest to developers integrating Python with .NET components. Formerly developed by Microsoft and now an open source project, IronPython might also be able to take advantage of some important optimization tools for better performance. For more details, consult *http://ironpython.net* and other resources to be had with a web search.

Stackless: Python for concurrency

Still other schemes for running Python programs have more focused goals. For example, the *Stackless* Python system is an enhanced version and reimplementation of the standard CPython language oriented toward *concurrency*. Because it does not save state on the C language call stack, Stackless Python can make Python easier to port to small stack architectures, provides efficient multiprocessing options, and fosters novel programming structures such as coroutines.

Among other things, the *microthreads* that Stackless adds to Python are an efficient and lightweight alternative to Python's standard multitasking tools such as threads and processes, and promise better program structure, more readable code, and increased programmer productivity. CCP Games, the creator of *EVE Online*, is a well-known Stackless Python user, and a compelling Python user success story in general. Try *http://stackless.com* for more information.

PyPy: Python for speed

The *PyPy* system is another standard CPython reimplementation, focused on *performance*. It provides a fast Python implementation with a *JIT* (just-in-time) compiler, provides tools for a "sandbox" model that can run untrusted code in a secure environ-

ment, and by default includes support for the prior section's *Stackless* Python systems and its microthreads to support massive concurrency.

PyPy is the successor to the original *Psyco* JIT, described ahead, and subsumes it with a complete Python implementation built for speed. A JIT is really just an extension to the PVM—the rightmost bubble in Figure 2-2—that translates portions of your byte code all the way to binary machine code for faster execution. It does this as your program is *running*, not in a prerun compile step, and is able to created type-specific machine code for the dynamic Python language by keeping track of the *data types* of the objects your program processes. By replacing portions of your byte code this way, your program runs faster and faster as it is executing. In addition, some Python programs may also take up less memory under PyPy.

At this writing, PyPy supports Python 2.7 code (not yet 3.X) and runs on Intel x86 (IA-32) and x86_64 platforms (including Windows, Linux, and recent Macs), with ARM and PPC support under development. It runs most CPython code, though C extension modules must generally be recompiled, and PyPy has some minor but subtle language differences, including garbage collection semantics that obviate some common coding patterns. For instance, its non-reference-count scheme means that temporary files may not close and flush output buffers immediately, and may require manual close calls in some cases.

In return, your code may run much quicker. PyPy currently claims a *5.7X* speedup over CPython across a range of benchmark programs (per *http://speed.pypy.org/*). In some cases, its ability to take advantage of dynamic optimization opportunities can make Python code as quick as C code, and occasionally faster. This is especially true for heavily algorithmic or numeric programs, which might otherwise be recoded in C.

For instance, in one simple benchmark we'll see in Chapter 21, PyPy today clocks in at *10X* faster than CPython 2.7, and *100X* faster than CPython 3.X. Though other benchmarks will vary, such speedups may be a compelling advantage in many domains, perhaps even more so than leading-edge language features. Just as important, memory space is also optimized in PyPy—in the case of one posted benchmark, requiring 247 MB and completing in 10.3 seconds, compared to CPython's 684 MB and 89 seconds.

PyPy's tool chain is also general enough to support additional languages, including *Pyrolog*, a Prolog interpreter written in Python using the PyPy translator. Search for PyPy's website for more. PyPy currently lives at *http://pypy.org*, though the usual web search may also prove fruitful over time. For an overview of its current performance, also see *http://www.pypy.org/performance.html*.

Just after I wrote this, PyPy 2.0 was released in beta form, adding support for the ARM processor, and still a Python 2.X-only implementation. Per its 2.0 beta release notes:

"PyPy is a very compliant Python interpreter, almost a drop-in replacement for CPython 2.7.3. It's fast due to its integrated tracing JIT compiler. This release supports x86 machines running Linux 32/64, Mac OS X 64 or Windows 32. It also supports ARM machines running Linux."

The claims seem accurate. Using the timing tools we'll study in Chapter 21, PyPy is often an order of magnitude (factor of 10) faster than CPython 2.X and 3.X on tests I've run, and sometimes even better. This is despite the fact that PyPy is a 32-bit build on my Windows test machine, while CPython is a faster 64-bit compile.

Naturally the only benchmark that truly matters is your own code, and there are cases where CPython wins the race; PyPy's file iterators, for instance, may clock in slower today. Still, given PyPy's focus on performance over language mutation, and especially its support for the numeric domain, many today see PyPy as an important path for Python. If you write CPU-intensive code, PyPy deserves your attention.

Execution Optimization Tools

CPython and most of the alternatives of the prior section all implement the Python language in similar ways: by compiling source code to byte code and executing the byte code on an appropriate virtual machine. Some systems, such as the Cython hybrid, the Shed Skin C++ translator, and the just-in-time compilers in PyPy and Psyco instead attempt to optimize the basic execution model. These systems are not required knowledge at this point in your Python career, but a quick look at their place in the execution model might help demystify the model in general.

Cython: A Python/C hybrid

The *Cython* system (based on work done by the *Pyrex* project) is a hybrid language that combines Python code with the ability to call C functions and use C type declarations for variables, parameters, and class attributes. Cython code can be compiled to C code that uses the Python/C API, which may then be compiled completely. Though not completely compatible with standard Python, Cython can be useful both for wrapping external C libraries and for coding efficient C extensions for Python. See *http://cython.org* for current status and details.

Shed Skin: A Python-to-C++ translator

Shed Skin is an emerging system that takes a different approach to Python program execution—it attempts to translate Python source code to C++ code, which your computer's C++ compiler then compiles to machine code. As such, it represents a platform-neutral approach to running Python code.

Shed Skin is still being actively developed as I write these words. It currently supports Python 2.4 to 2.6 code, and it limits Python programs to an implicit statically typed constraint that is typical of most programs but is technically not normal Python, so we won't go into further detail here. Initial results, though, show that it has the potential to outperform both standard Python and Psyco-like extensions in terms of execution speed. Search the Web for details on the project's current status.

Psyco: The original just-in-time compiler

The Psyco system is not another Python implementation, but rather a component that extends the byte code execution model to make programs run faster. Today, Psyco is something of an *ex-project*: it is still available for separate download, but has fallen out of date with Python's evolution, and is no longer actively maintained. Instead, its ideas have been incorporated into the more complete *PyPy* system described earlier. Still, the ongoing importance of the ideas Psyco explored makes them worth a quick look.

In terms of Figure 2-2, Psyco is an enhancement to the PVM that collects and uses type information while the program runs to translate portions of the program's byte code all the way down to true binary machine code for faster execution. Psyco accomplishes this translation without requiring changes to the code or a separate compilation step during development.

Roughly, while your program runs, Psyco collects information about the kinds of objects being passed around; that information can be used to generate highly efficient machine code tailored for those object types. Once generated, the machine code then replaces the corresponding part of the original byte code to speed your program's overall execution. The result is that with Psyco, your program becomes quicker over time as it runs. In ideal cases, some Python code may become as fast as compiled C code under Psyco.

Because this translation from byte code happens at program runtime, Psyco is known as a *just-in-time* compiler. Psyco is different from the JIT compilers some readers may have seen for the Java language, though. Really, Psyco is a *specializing JIT compiler*—it generates machine code tailored to the data types that your program actually uses. For example, if a part of your program uses different data types at different times, Psyco may generate a different version of machine code to support each different type combination.

Psyco was shown to speed some Python code dramatically. According to its web page, Psyco provides "2X to 100X speed-ups, typically 4X, with an unmodified Python interpreter and unmodified source code, just a dynamically loadable C extension module." Of equal significance, the largest speedups are realized for algorithmic code written in pure Python—exactly the sort of code you might normally migrate to C to optimize. For more on Psyco, search the Web or see its successor—the PyPy project described previously.

Frozen Binaries

Sometimes when people ask for a "real" Python compiler, what they're really seeking is simply a way to generate standalone binary executables from their Python programs. This is more a packaging and shipping idea than an execution-flow concept, but it's somewhat related. With the help of third-party tools that you can fetch off the Web, it is possible to turn your Python programs into true executables, known as *frozen binaries* in the Python world. These programs can be run without requiring a Python installation.

Frozen binaries bundle together the byte code of your program files, along with the PVM (interpreter) and any Python support files your program needs, into a single package. There are some variations on this theme, but the end result can be a single binary executable program (e.g., an *.exe* file on Windows) that can easily be shipped to customers. In Figure 2-2, it is as though the two rightmost bubbles—byte code and PVM—are merged into a single component: a frozen binary file.

Today, a variety of systems are capable of generating frozen binaries, which vary in platforms and features: *py2exe* for Windows only, but with broad Windows support; *PyInstaller*, which is similar to py2exe but also works on Linux and Mac OS X and is capable of generating self-installing binaries; *py2app* for creating Mac OS X applications; *freeze*, the original; and *cx_freeze*, which offers both Python 3.X and cross-platform support. You may have to fetch these tools separately from Python itself, but they are freely available.

These tools are also constantly evolving, so consult *http://www.python.org* or your favorite web search engine for more details and status. To give you an idea of the scope of these systems, py2exe can freeze standalone programs that use the tkinter, PMW, wxPython, and PyGTK GUI libraries; programs that use the *pygame* game programming toolkit; win32com client programs; and more.

Frozen binaries are not the same as the output of a true compiler—they run byte code through a virtual machine. Hence, apart from a possible startup improvement, frozen binaries run at the same speed as the original source files. Frozen binaries are also not generally small (they contain a PVM), but by current standards they are not unusually large either. Because Python is embedded in the frozen binary, though, it does not have to be installed on the receiving end to run your program. Moreover, because your code is embedded in the frozen binary, it is more effectively hidden from recipients.

This single file-packaging scheme is especially appealing to developers of commercial software. For instance, a Python-coded user interface program based on the tkinter toolkit can be frozen into an executable file and shipped as a self-contained program on a CD or on the Web. End users do not need to install (or even have to know about) Python to run the shipped program.

Future Possibilities?

Finally, note that the runtime execution model sketched here is really an artifact of the current implementation of Python, not of the language itself. For instance, it's not impossible that a full, traditional compiler for translating Python source code to machine code may appear during the shelf life of this book (although the fact that one has not in over two decades makes this seem unlikely!).

New byte code formats and implementation variants may also be adopted in the future. For instance:

- The ongoing *Parrot* project aims to provide a common byte code format, virtual machine, and optimization techniques for a variety of programming languages, including Python. Python's own PVM runs Python code more efficiently than Parrot (as famously demonstrated by a pie challenge at a software conference—search the Web for details), but it's unclear how Parrot will evolve in relation to Python specifically. See *http://parrot.org* or the Web at large for details.
- The former *Unladen Swallow* project—an open source project developed by Google engineers—sought to make standard Python faster by a factor of at least 5, and fast enough to replace the C language in many contexts. This was an optimization branch of CPython (specifically Python 2.6), intended to be compatible yet faster by virtue of adding a JIT to standard Python. As I write this in 2012, this project seems to have drawn to a close (per its withdrawn Python PEP, it was "going the way of the Norwegian Blue"). Still, its lessons gained may be leveraged in other forms; search the Web for breaking developments.

Although future implementation schemes may alter the runtime structure of Python somewhat, it seems likely that the byte code compiler will still be the standard for some time to come. The portability and runtime flexibility of byte code are important features of many Python systems. Moreover, adding type constraint declarations to support static compilation would likely break much of the flexibility, conciseness, simplicity, and overall spirit of Python coding. Due to Python's highly dynamic nature, any future implementation will likely retain many artifacts of the current PVM.

Chapter Summary

This chapter introduced the execution model of Python—how Python runs your programs—and explored some common variations on that model: just-in-time compilers and the like. Although you don't really need to come to grips with Python internals to write Python scripts, a passing acquaintance with this chapter's topics will help you truly understand how your programs run once you start coding them. In the next chapter, you'll start actually running some code of your own. First, though, here's the usual chapter quiz.

Test Your Knowledge: Quiz

1. What is the Python interpreter?
2. What is source code?
3. What is byte code?
4. What is the PVM?
5. Name two or more variations on Python's standard execution model.
6. How are CPython, Jython, and IronPython different?
7. What are Stackless and PyPy?

Test Your Knowledge: Answers

1. The Python interpreter is a program that runs the Python programs you write.
2. Source code is the statements you write for your program—it consists of text in text files that normally end with a *.py* extension.
3. Byte code is the lower-level form of your program after Python compiles it. Python automatically stores byte code in files with a *.pyc* extension.
4. The PVM is the Python Virtual Machine—the runtime engine of Python that interprets your compiled byte code.
5. Psyco, Shed Skin, and frozen binaries are all variations on the execution model. In addition, the alternative implementations of Python named in the next two answers modify the model in some fashion as well—by replacing byte code and VMs, or by adding tools and JITs.
6. CPython is the standard implementation of the language. Jython and IronPython implement Python programs for use in Java and .NET environments, respectively; they are alternative compilers for Python.
7. Stackless is an enhanced version of Python aimed at concurrency, and PyPy is a reimplementation of Python targeted at speed. PyPy is also the successor to Psyco, and incorporates the JIT concepts that Psyco pioneered.

CHAPTER 3

How You Run Programs

OK, it's time to start running some code. Now that you have a handle on the program execution model, you're finally ready to start some real Python programming. At this point, I'll assume that you have Python installed on your computer; if you don't, see the start of the prior chapter and Appendix A for installation and configuration hints on various platforms. Our goal here is to learn how to run Python program code.

There are multiple ways to tell Python to execute the code you type. This chapter discusses all the program launching techniques in common use today. Along the way, you'll learn how to both type code *interactively*, and how to save it in *files* to be run as often as you like in a variety of ways: with system command lines, icon clicks, module imports, `exec` calls, menu options in the IDLE GUI, and more.

As for the previous chapter, if you have prior programming experience and are anxious to start digging into Python itself, you may want to skim this chapter and move on to Chapter 4. But don't skip this chapter's early coverage of preliminaries and conventions, its overview of debugging techniques, or its first look at module imports—a topic essential to understanding Python's program architecture, which we won't revisit until a later part. I also encourage you to see the sections on IDLE and other IDEs, so you'll know what tools are available when you start developing more sophisticated Python programs.

The Interactive Prompt

This section gets us started with interactive coding basics. Because it's our first look at running code, we also cover some preliminaries here, such as setting up a working directory and the system path, so be sure to read this section first if you're relatively new to programming. This section also explains some conventions used throughout the book, so most readers should probably take at least a quick look here.

Starting an Interactive Session

Perhaps the simplest way to run Python programs is to type them at Python's interactive command line, sometimes called the *interactive prompt*. There are a variety of ways to start this command line: in an IDE, from a system console, and so on. Assuming the interpreter is installed as an executable program on your system, the most platform-neutral way to start an interactive interpreter session is usually just to type **`python`** at your operating system's prompt, without any arguments. For example:

```
% python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

Typing the word "python" at your system shell prompt like this begins an interactive Python session; the "%" character at the start of this listing stands for a generic system prompt in this book—it's not input that you type yourself. On Windows, a *Ctrl-Z* gets you out of this session; on Unix, try *Ctrl-D* instead.

The notion of a *system shell prompt* is generic, but exactly how you access it varies by platform:

- On *Windows*, you can type **`python`** in a DOS console window—a program named `cmd.exe` and usually known as *Command Prompt*. For more details on starting this program, see this chapter's sidebar "Where Is Command Prompt on Windows?" on page 45.
- On *Mac OS X*, you can start a Python interactive interpreter by double-clicking on Applications→Utilities→Terminal, and then typing **`python`** in the window that opens up.
- On *Linux* (and other Unixes), you might type this command in a shell or terminal window (for instance, in an *xterm* or console running a shell such as *ksh* or *csh*).
- Other systems may use similar or platform-specific devices. On handheld devices, for example, you might click the Python icon in the home or application window to launch an interactive session.

On most platforms, you can start the interactive prompt in additional ways that don't require typing a command, but they vary per platform even more widely:

- On *Windows 7* and earlier, besides typing **`python`** in a shell window, you can also begin similar interactive sessions by starting the IDLE GUI (discussed later), or by selecting the "Python (command line)" menu option from the Start button menu for Python, as shown in Figure 2-1 in Chapter 2. Both spawn a Python interactive prompt with the same functionality obtained with a "python" command.
- On *Windows 8*, you don't have a Start button (at least as I write this), but there are other ways to get to the tools described in the prior bullet, including tiles, Search, File Explorer, and the "All apps" interface on the Start screen. See Appendix A for more pointers on this platform.

- Other platforms have similar ways to start a Python interactive session without typing commands, but they're too specific to get into here; see your system's documentation for details.

Anytime you see the >>> prompt, you're in an interactive Python interpreter session—you can type any Python statement or expression here and run it immediately. We will in a moment, but first we need to get a few startup details sorted out to make sure all readers are set to go.

Where Is Command Prompt on Windows?

So how do you start the command-line interface on Windows? Some Windows readers already know, but Unix developers and beginners may not; it's not as prominent as terminal or console windows on Unix systems. Here are some pointers on finding your Command Prompt, which vary slightly per Windows version.

On *Windows 7 and earlier*, this is usually found in the Accessories section of the Start→All Programs menu, or you can run it by typing **`cmd`** in the Start→Run... dialog box or the Start menu's search entry field. You can drag out a desktop shortcut to get to it quicker if desired.

On *Windows 8*, you can access Command Prompt in the menu opened by right-clicking on the preview in the screen's lower-left corner; in the Windows System section of the "All apps" display reached by right-clicking your Start screen; or by typing **`cmd`** or **`command prompt`** in the input field of the Search charm pulled down from the screen's upper-right corner. There are probably additional routes, and touch screens offer similar access. And if you want to forget all that, pin it to your desktop taskbar for easy access next time around.

These procedures are prone to vary over time, and possibly even per computer and user. I'm trying to avoid making this a book on Windows, though, so I'll cut this topic short here. When in doubt, try the system Help interface (whose usage may differ as much as the tools it provides help for!).

A note to any Unix users reading this sidebar who may be starting to feel like a fish out of water: you may also be interested in the *Cygwin* system, which brings a full Unix command prompt to Windows. See Appendix A for more pointers.

The System Path

When we typed **`python`** in the last section to start an interactive session, we relied on the fact that the system located the Python program for us on its program search path. Depending on your Python version and platform, if you have not set your system's `PATH` environment variable to include Python's install directory, you may need to replace the word "python" with the full path to the Python executable on your machine. On Unix, Linux, and similar, something like **`/usr/local/bin/python`** or **`/usr/bin/python3`** will often suffice. On Windows, try typing **`C:\Python33\python`** (for version 3.3):

```
c:\code> c:\python33\python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

Alternatively, you can run a "cd" change-directory command to go to Python's install directory before typing **python**—try the **cd c:\python33** command on Windows, for example:

```
c:\code> cd c:\python33
c:\Python33> python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

But you'll probably want to set your PATH eventually, so a simple "python" suffices. If you don't know what PATH is or how to set it, see Appendix A—it covers environment variables like this whose usage varies per platform, as well as Python command-line arguments we won't be using much in this book. The short story for Windows users: see the Advanced settings in the System entry of your Control Panel. If you're using Python 3.3 and later, this is now automatic on Windows, as the next section explains.

New Windows Options in 3.3: PATH, Launcher

The foregoing section and much of this chapter at large describe the generic state of play for all 2.X and 3.X Pythons prior to version 3.3. Starting with Python 3.3, the Windows installer has an option to *automatically* add Python 3.3's directory to your system PATH, if enabled in the installer's windows. If you use this option, you won't need to type a directory path or issue a "cd" to run python commands as in the prior section. Be sure to select this option during the install if you want it, as it's currently disabled by default.

More dramatically, Python 3.3 for Windows ships with and automatically installs the new *Windows launcher*—a system that comes with new executable programs, py with a console and pyw without, that are placed in directories on your system path, and so may be run out of the box without any PATH configurations, change-directory commands, or directory path prefixes:

```
c:\code> py
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z

c:\code> py -2
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z

c:\code> py -3.1
Python 3.1.4 (default, Jun 12 2011, 14:16:16) [MSC v.1500 64 bit (AMD64)] ...
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z
```

As shown in the last two commands here, these executables also accept Python version numbers on the command line (and in Unix-style `#!` lines at the top of scripts, as discussed later), and are associated to open Python files when clicked just like the original `python` executable—which is still available and works as before, but is somewhat superseded by the launcher's new programs.

The launcher is a standard part of Python 3.3, and is available standalone for use with other versions. We'll see more on this new launcher in this and later chapters, including a brief look at its `#!` line support here. However, because it is of interest only to Windows users, and even for this group is present only in 3.3 or where installed separately, I've collected almost all of the details about the launcher in Appendix B.

If you'll be working on Windows under Python 3.3 or later, I suggest taking a brief detour to that appendix now, as it provides an alternative, and in some ways better, way to run Python command lines and scripts. At a base level, launcher users can type **`py`** instead of **`python`** in most of the system commands shown in this book, and may avoid some configuration steps. Especially on computers with multiple Python versions, though, the new launcher gives you more explicit control over which Python runs your code.

Where to Run: Code Directories

Now that I've started showing you *how* to run code, I want to say a few words up front about *where* to run code. To keep things simple, in this chapter and book at large I'm going to be running code from a working directory (a.k.a. *folder*) I've created on my Windows computer called *C:\code*—a subdirectory at the top of my main drive. That's where I'll start most interactive sessions, and where I'll be both saving and running most script files. This also means the files that examples will create will mostly show up in this directory.

If you'll be working along, you should probably do something similar before we get started. Here are some pointers if you need help getting set up with a working directory on your computer:

- On *Windows*, you can make your working code directory in File Explorer or a Command Prompt window. In File Explorer, look for New Folder, see the File menu, or try a right-click. In Command Prompt, type and run a `mkdir` command, usually after you `cd` to your desired parent directory (e.g., `cd c\:` and `mkdir code`). Your working directory can be located wherever you like and called whatever you wish, and doesn't have to be *C:\code* (I chose this name because it's short in prompts). But running out of one directory will help you keep track of your work and simplify some tasks. For more Windows hints, see this chapter's sidebar on Command Prompt, as well as Appendix A.

- On *Unix*-based systems (including *Mac OS X* and *Linux*), your working directory might be in */usr/home* and be created by a `mkdir` command in a shell window or file explorer GUI specific to your platform, but the same concepts apply. The Cygwin Unix-like system for Windows is similar too, though your directory names may vary (*/home* and */cygdrive/c* are candidates).

You can store your code in Python's install directory too (e.g., *C:\Python33* on Windows) to simplify some command lines before setting `PATH`, but you probably shouldn't —this is for Python itself, and your files may not survive a move or uninstall.

Once you've made your working directory, always start there to work along with the examples in this book. The prompts in this book that show the directory that I'm running code in will reflect my Windows laptop's working directory; when you see `C:\code>` or `%`, think the location and name of your own directory.

What Not to Type: Prompts and Comments

Speaking of prompts, this book sometimes shows system prompts as a generic `%`, and sometimes in full `C:\code>` Windows form. The former is meant to be platform agnostic (and derives from earlier editions' use of Linux), and the latter is used in Windows-specific contexts. I also add a space after system prompts just for readability in this book. When used, the `%` character at the start of a system command line stands for the system's prompt, whatever that may be on your machine. For instance, on my machine `%` stands for `C:\code>` in Windows Command Prompt, and just `$` in my Cygwn install.

To beginners: don't type the `%` character (or the `C:\code` system prompt it sometimes stands for) you see in this book's interaction listings yourself—this is text the system prints. Type just the text *after* these system prompts. Similarly, do not type the `>>>` and `...` characters shown at the start of lines in interpreter interaction listings—these are prompts that Python displays automatically as visual guides for interactive code entry. Type just the text *after* these Python prompts. For instance, the `...` prompt is used for continuation lines in some shells, but doesn't appear in IDLE, and shows up in some but not all of this book's listings; don't type it yourself if it's absent in your interface.

To help you remember this, user inputs are shown in **`bold`** in this book, and prompts are not. In some systems these prompts may differ (for instance, the *PyPy* performance-focused implementation described in Chapter 2 uses four-character `>>>>` and `....`), but the same rules apply. Also keep in mind that commands typed after these system and Python prompts are meant to be run immediately, and are not generally to be saved in the source files we will be creating; we'll see why this distinction matters ahead.

In the same vein, you normally don't need to type text that starts with a `#` character in listings in this book—as you'll learn, these are *comments*, not executable code. Except when `#` is used to introduce a directive at the top of a script for Unix or the Python 3.3

Windows launcher, you can safely ignore the text that follows it (more on Unix and the launcher later in this chapter and in Appendix B).

If you're working along, interactive listings will drop most "..." continuation prompts as of Chapter 17 to aid cut-and-paste of larger code such as functions and classes from ebooks or other; until then, paste or type one line at a time and omit the prompts. At least initially, it's important to type code manually, to get a feel for syntax details and errors. Some examples will be listed either by themselves or in named files available in the book's examples package (per the preface), and we'll switch between listing formats often; when in doubt, if you see ">>>", it means the code is being typed interactively.

Running Code Interactively

With those preliminaries out of the way, let's move on to typing some actual code. However it's started, the Python interactive session begins by printing two lines of informational text giving the Python version number and a few hints shown earlier (which I'll omit from most of this book's examples to save space), then prompts for input with `>>>` when it's waiting for you to type a new Python statement or expression.

When working interactively, the results of your code are displayed below the `>>>` input lines after you press the Enter key. For instance, here are the results of two Python `print` statements (`print` is really a function call in Python 3.X, but not in 2.X, so the parentheses here are required in 3.X only):

```
% python
>>> print('Hello world!')
Hello world!
>>> print(2 ** 8)
256
```

There it is—we've just run some Python code (were you expecting the *Spanish Inquisition*?). Don't worry about the details of the `print` statements shown here yet; we'll start digging into syntax in the next chapter. In short, they print a Python string and an integer, as shown by the output lines that appear after each `>>>` input line (`2 ** 8` means 2 raised to the power 8 in Python).

When coding interactively like this, you can type as many Python commands as you like; each is run immediately after it's entered. Moreover, because the interactive session automatically prints the results of expressions you type, you don't usually need to say "print" explicitly at this prompt:

```
>>> lumberjack = 'okay'
>>> lumberjack
'okay'
>>> 2 ** 8
256
```

```
>>> ^Z                          # Use Ctrl-D (on Unix) or Ctrl-Z (on Windows) to exit
%
```

Here, the first line saves a value by assigning it to a *variable* (`lumberjack`), which is created by the assignment; and the last two lines typed are *expressions* (`lumberjack` and `2 ** 8`), whose results are displayed automatically. Again, to exit an interactive session like this and return to your system shell prompt, type Ctrl-D on Unix-like machines, and Ctrl-Z on Windows. In the IDLE GUI discussed later, either type Ctrl-D or simply close the window.

Notice the *italicized note* about this on the right side of this listing (staring with "#" here). I'll use these throughout to add remarks about what is being illustrated, but you don't need to type this text yourself. In fact, just like system and Python prompts, you shouldn't type this when it's on a system command line; the "#" part is taken as a comment by Python but may be an error at a system prompt.

Now, we didn't do much in this session's code—just typed some Python `print` and assignment statements, along with a few expressions, which we'll study in detail later. The main thing to notice is that the interpreter executes the code entered on each line immediately, when the Enter key is pressed.

For example, when we typed the first `print` statement at the `>>>` prompt, the output (a Python string) was echoed back right away. There was no need to create a source code file, and no need to run the code through a compiler and linker first, as you'd normally do when using a language such as C or C++. As you'll see in later chapters, you can also run multiline statements at the interactive prompt; such a statement runs immediately after you've entered all of its lines and pressed Enter twice to add a blank line.

Why the Interactive Prompt?

The interactive prompt runs code and echoes results as you go, but it doesn't save your code in a file. Although this means you won't do the bulk of your coding in interactive sessions, the interactive prompt turns out to be a great place to both *experiment* with the language and *test* program files on the fly.

Experimenting

Because code is executed immediately, the interactive prompt is a perfect place to experiment with the language and will be used often in this book to demonstrate smaller examples. In fact, this is the first rule of thumb to remember: if you're ever in doubt about how a piece of Python code works, fire up the interactive command line and try it out to see what happens.

For instance, suppose you're reading a Python program's code and you come across an expression like `'Spam!' * 8` whose meaning you don't understand. At this point, you can spend 10 minutes wading through manuals, books, and the Web to try to figure out what the code does, or you can simply run it interactively:

```
% python
>>> 'Spam!' * 8                                  # Learning by trying
'Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!'
```

The immediate feedback you receive at the interactive prompt is often the quickest way to deduce what a piece of code does. Here, it's clear that it does string repetition: in Python * means multiply for numbers, but repeat for strings—it's like concatenating a string to itself repeatedly (more on strings in Chapter 4).

Chances are good that you won't break anything by experimenting this way—at least, not yet. To do real damage, like deleting files and running shell commands, you must really try, by importing modules explicitly (you also need to know more about Python's system interfaces in general before you will become that dangerous!). Straight Python code is almost always safe to run.

For instance, watch what happens when you *make a mistake* at the interactive prompt:

```
>>> X                                            # Making mistakes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'X' is not defined
```

In Python, using a variable before it has been assigned a value is always an error—otherwise, if names were filled in with defaults, some errors might go undetected. This means you must initial counters to zero before you can add to them, must initial lists before extending them, and so on; you don't declare variables, but they must be assigned before you can fetch their values.

We'll learn more about that later; the important point here is that you don't crash Python or your computer when you make a mistake this way. Instead, you get a meaningful error message pointing out the mistake and the line of code that made it, and you can continue on in your session or script. In fact, once you get comfortable with Python, its error messages may often provide as much debugging support as you'll need (you'll learn more about debugging options in the sidebar "Debugging Python Code" on page 83).

Testing

Besides serving as a tool for experimenting while you're learning the language, the interactive interpreter is also an ideal place to test code you've written in files. You can import your module files interactively and run tests on the tools they define by typing calls at the interactive prompt on the fly.

For instance, the following tests a function in a precoded module that ships with Python in its standard library (it prints the name of the directory you're currently working in, with a doubled-up backslash that stands for just one), but you can do the same once you start writing module files of your own:

```
>>> import os
>>> os.getcwd()                                  # Testing on the fly
'c:\\code'
```

More generally, the interactive prompt is a place to test program components, regardless of their source—you can import and test functions and classes in your Python files, type calls to linked-in C functions, exercise Java classes under Jython, and more. Partly because of its interactive nature, Python supports an experimental and exploratory programming style you'll find convenient when getting started. Although Python programmers also test with in-file code (and we'll learn ways to make this simple later in the book), for many, the interactive prompt is still their first line of testing defense.

Usage Notes: The Interactive Prompt

Although the interactive prompt is simple to use, there are a few tips that beginners should keep in mind. I'm including lists of common mistakes like the following in this chapter for reference, but they might also spare you from a few headaches if you read them up front:

- **Type Python commands only**. First of all, remember that you can only type Python code at Python's >>> prompt, not system commands. There are ways to run system commands from within Python code (e.g., with `os.system`), but they are not as direct as simply typing the commands themselves.
- **`print` statements are required only in files**. Because the interactive interpreter automatically prints the results of expressions, you do not need to type complete `print` statements interactively. This is a nice feature, but it tends to confuse users when they move on to writing code in files: within a code file, you must use `print` statements to see your output because expression results are not automatically echoed. Remember, you must say `print` in files, but it's optional interactively.
- **Don't indent at the interactive prompt (yet)**. When typing Python programs, either interactively or into a text file, be sure to start all your unnested statements in column 1 (that is, all the way to the left). If you don't, Python may print a "SyntaxError" message, because blank space to the left of your code is taken to be indentation that groups nested statements. Until Chapter 10, all statements you write will be unnested, so this includes everything for now. Remember, a leading space generates an error message, so don't start with a space or tab at the interactive prompt unless it's nested code.
- **Watch out for prompt changes for compound statements**. We won't meet *compound* (multiline) statements until Chapter 4 and not in earnest until Chapter 10, but as a preview, you should know that when typing lines 2 and beyond of a compound statement interactively, the prompt may change. In the simple shell window interface, the interactive prompt changes to `...` instead of >>> for lines 2 and beyond; in the IDLE GUI interface, lines after the first are instead automatically indented.

  You'll see why this matters in Chapter 10. For now, if you happen to come across a `...` prompt or a blank line when entering your code, it probably means that you've somehow confused interactive Python into thinking you're typing a multiline

statement. Try hitting the Enter key or a Ctrl-C combination to get back to the main prompt. The >>> and ... prompt strings can also be changed (they are available in the built-in module sys), but I'll assume they have not been in the book's example listings.

- **Terminate compound statements at the interactive prompt with a blank line**. At the interactive prompt, inserting a blank line (by hitting the Enter key at the start of a line) is necessary to tell interactive Python that you're done typing the multiline statement. That is, you must press Enter twice to make a compound statement run. By contrast, blank lines are not required in files and are simply ignored if present. If you don't press Enter twice at the end of a compound statement when working interactively, you'll appear to be stuck in a limbo state, because the interactive interpreter will do nothing at all—it's waiting for you to press Enter again!
- **The interactive prompt runs one statement at a time**. At the interactive prompt, you must run one statement to completion before typing another. This is natural for simple statements, because pressing the Enter key runs the statement entered. For compound statements, though, remember that you must submit a blank line to terminate the statement and make it run before you can type the next statement.

Entering multiline statements

At the risk of repeating myself, I've received multiple emails from readers who'd gotten burned by the last two points, so they probably merit emphasis. I'll introduce multiline (a.k.a. compound) statements in the next chapter, and we'll explore their syntax more formally later in this book. Because their behavior differs slightly in files and at the interactive prompt, though, two cautions are in order here.

First, be sure to terminate multiline compound statements like `for` loops and `if` tests at the interactive prompt with a blank line. In other words, *you must press the Enter key twice*, to terminate the whole multiline statement and then make it run. For example (pun not intended):

```
>>> for x in 'spam':
...     print(x)                # Press Enter twice here to make this loop run
...
```

You don't need the blank line after compound statements in a script file, though; this is required *only* at the interactive prompt. In a file, blank lines are not required and are simply ignored when present; at the interactive prompt, they terminate multiline statements. Reminder: the ... continuation line prompt in the preceding is printed by Python automatically as a visual guide; it may not appear in your interface (e.g., IDLE), and is sometimes omitted by this book, but do not type it yourself if it's absent.

Also bear in mind that the interactive prompt runs just *one statement at a time*: you must press Enter twice to run a loop or other multiline statement before you can type the next statement:

```
>>> for x in 'spam':
...     print(x)                # Press Enter twice before a new statement
... print('done')
  File "<stdin>", line 3
    print('done')
        ^
SyntaxError: invalid syntax
```

This means you can't cut and paste multiple lines of code into the interactive prompt, unless the code includes blank lines after each compound statement. Such code is better run in a *file*—which brings us to the next section's topic.

System Command Lines and Files

Although the interactive prompt is great for experimenting and testing, it has one big disadvantage: programs you type there go away as soon as the Python interpreter executes them. Because the code you type interactively is never stored in a file, you can't run it again without retyping it from scratch. Cut-and-paste and command recall can help some here, but not much, especially when you start writing larger programs. To cut and paste code from an interactive session, you would have to edit out Python prompts, program outputs, and so on—not exactly a modern software development methodology!

To save programs permanently, you need to write your code in files, which are usually known as *modules*. Modules are simply text files containing Python statements. Once they are coded, you can ask the Python interpreter to execute the statements in such a file any number of times, and in a variety of ways—by system command lines, by file icon clicks, by options in the IDLE user interface, and more. Regardless of how it is run, Python executes all the code in a module file from top to bottom each time you run the file.

Terminology in this domain can vary somewhat. For instance, module files are often referred to as *programs* in Python—that is, a program is considered to be a series of precoded statements stored in a file for repeated execution. Module files that are run directly are also sometimes called *scripts*—an informal term usually meaning a top-level program file. Some reserve the term "module" for a file imported from another file, and "script" for the main file of a program; we generally will here, too (though you'll have to stay tuned for more on the meaning of "top-level," imports, and main files later in this chapter).

Whatever you call them, the next few sections explore ways to run code typed into module files. In this section, you'll learn how to run files in the most basic way: by listing their names in a `python` command line entered at your computer's system prompt. Though it might seem primitive to some—and can often be avoided altogether by using a GUI like IDLE, discussed later—for many programmers a system shell command-line window, together with a text editor window, constitutes as much of an

integrated development environment as they will ever need, and provides more direct control over programs.

A First Script

Let's get started. Open your favorite text editor (e.g., *vi*, Notepad, or the IDLE editor), type the following statements into a new text file named *script1.py*, and save it in your working code directory that you set up earlier:

```
# A first Python script
import sys                  # Load a library module
print(sys.platform)
print(2 ** 100)             # Raise 2 to a power
x = 'Spam!'
print(x * 8)                # String repetition
```

This file is our first official Python script (not counting the two-liner in Chapter 2). You shouldn't worry too much about this file's code, but as a brief description, this file:

- Imports a Python module (libraries of additional tools), to fetch the name of the platform
- Runs three `print` function calls, to display the script's results
- Uses a variable named `x`, created when it's assigned, to hold onto a string object
- Applies various object operations that we'll begin studying in the next chapter

The `sys.platform` here is just a string that identifies the kind of computer you're working on; it lives in a standard Python module called `sys`, which you must import to load (again, more on imports later).

For color, I've also added some formal Python *comments* here—the text after the `#` characters. I mentioned these earlier, but should be more formal now that they're showing up in scripts. Comments can show up on lines by themselves, or to the right of code on a line. The text after a `#` is simply ignored as a human-readable comment and is not considered part of the statement's syntax. If you're copying this code, you can ignore the comments; they are just informative. In this book, we usually use a different formatting style to make comments more visually distinctive, but they'll appear as normal text in your code.

Again, don't focus on the syntax of the code in this file for now; we'll learn about all of it later. The main point to notice is that you've typed this code into a file, rather than at the interactive prompt. In the process, you've coded a fully functional Python script.

Notice that the module file is called *script1.py*. As for all top-level files, it could also be called simply *script*, but files of code you want to *import* into a client have to end with a *.py* suffix. We'll study imports later in this chapter. Because you may want to import them in the future, it's a good idea to use *.py* suffixes for most Python files that you code. Also, some text editors detect Python files by their *.py* suffix; if the suffix is not present, you may not get features like syntax colorization and automatic indentation.