Chapter 1. A Python Q&A Session

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

Why Do People Use Python?

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

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

Software quality

For many, Python's focus on readability, coherence, and software quality in general sets it apart from other tools in the programing world. Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it relatively easy to understand, even if you did not write it. In addition, Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and functional programming, that can further promote code quality.

Developer productivity

Python boosts developer productivity many times beyond compiled or statically typed languages. As one measure of this, Python code is

typically *one-third to one-fifth* the size of equivalent C++ or Java code. That means there is less to type, less to debug, and less to maintain after the fact. Most Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed.

Program portability

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

Application support

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

Component integration

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

Love of craft

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

asset. People do get paid for coding Python, but many use it just for *fun*—a testimonial rare in the software field.

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

Software Quality

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

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

Beyond such design themes, Python includes tools such as modules and object-oriented programming (OOP) that naturally promote code reusability in skilled hands of the sort you're about to acquire. And because Python is focused on quality, most Python programmers naturally are too; this can be a crucial advantage when it's time to use someone else's Python code.

Developer Productivity

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

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

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

Is Python a "Scripting Language"?

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

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

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

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

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

Shell tools

Sometimes when people hear Python described as a scripting language, they think it means that Python is a tool for coding operating-system-oriented scripts. Such programs are often launched from console command lines and perform tasks such as processing text files and launching other programs.

As you'll learn ahead, Python programs can and do serve such roles, but this is just one of dozens of common Python application domains. It is not just a better shell-script language.

Control language

To others, scripting refers to a "glue" layer used to control and direct (i.e., script) other application components. As noted earlier, Python programs are indeed often deployed in the context of larger applications. For instance, to test hardware devices, Python programs may call out to components that give low-level access to a device or port. Similarly, programs may run bits of Python code at strategic points to support end-user product customization without the need to ship and recompile the entire system's source code.

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

Ease of use

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

Don't be fooled, though—Python is not just for simple tasks. Rather, it makes tasks simple by its ease of use and flexibility. Python has an approachable feature set, but it allows programs to scale up in

sophistication as needed. Because of that, it is commonly used for both quick tactical tasks and longer-term strategic development.

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

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

OK, but What's the Downside?

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

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

Whether you will ever *care* about this execution speed difference depends on what kinds of programs you write. Python is regularly optimized, and Python code runs fast enough by itself in most application domains. Furthermore,

whenever you do something “real” in a Python script, like processing a file or constructing a graphical user interface (*GUI*), your program will actually run at C speed, because such tasks are immediately dispatched to compiled C code inside the Python interpreter.

And really, Python’s speed is a bit of a red herring today: in truth, Python *is* commonly used in domains that require optimal execution speeds. Numeric programming and computer games, for example, often need at least their core number-crunching components to run at C speed (or better), but are frequently coded in Python nevertheless.

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

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

Who Uses Python Today?

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

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

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

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

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

What Can I Do with Python?

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

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

Two notes up front: first, we won't be able to explore these tools in any depth either here or in this book at large. NumPy and Django, for example, are

book-length topics on their own, and our goal in this book is to learn the *Python* code you will be writing to use systems like these. Second, apologies in advance to the many other tools omitted here only for space; if you are interested in any of the following topics, please search online for more tools and resources.

Systems Programming

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

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

GUIs and UIs

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

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

Internet and Web Scripting

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

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

Component Integration

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

Tools such as *SWIG*, *Boost.Python*, *CFFI*, and *HPy* automate much of the work needed to link compiled components with Python scripts, and the *Cython* system allows coders to mix Python and C-like code to create compiled extensions. Other tools provide more ways to script components—including the *pyjnius* and *Chaquopy* Python/Java bridges; *pywin32*'s support for Windows Component Object Model (COM); the *REST*, *SOAP*, and *XML-RPC* cross-network conduits; and the *Jython* Java and *IronPython* .NET implementations of Python. In short, most software components are in scope to Python code.

Database Access

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

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

Rapid Prototyping

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

Numeric and Scientific Programming

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

programming tool that can often replace code written in traditional languages like Fortran or C++.

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

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

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

- Artificial intelligence (see *PyTorch*, *TensorFlow*, and *Keras*)
- Game programming (see *pygame*, *Panda3D*, and *Kivy*)
- Image and graphics processing (see *Pillow*, *PyOpenGL*, and *OpenCV*)
- Quality assurance and testing (see *PyTest*, *unittest*, and *Selenium*)
- Excel spreadsheets (see *xlwings*, *PyXLL*, and *Excel*)
- Smartphone apps (see *Kivy*, *BeeWare*, and Appendix A)
- Microcontrollers and ports (see *MicroPython* and *PySerial*)
- And of course, much more

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

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

What Are Python's Technical Strengths?

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

It's Object-Oriented and Functional

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

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

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

It's Free and Open

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

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

It's Portable

We touched on portability earlier. The standard implementation of Python is written in portable ANSI C, and compiles and runs on virtually every major platform currently in use. For example, Python programs run today on everything from smartphones to supercomputers. As a partial list, Python is available on Windows and macOS PCs, Linux and Unix workstations and servers, Android and iOS smartphones and tablets, real-time systems like VxWorks, Cray supercomputers, IBM mainframes, and more. Moreover, this list expands regularly; Android and iOS, for example, are gaining official *python.org* support at this writing.

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

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

It's Powerful

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

Dynamic typing

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

Automatic memory management

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

Programming-in-the-large support

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

Built-in object types

Python provides commonly used data structures such as lists, dictionaries, and strings as intrinsic parts of the language. As you'll see, its built-in objects are both flexible and easy to use. They can grow and shrink on demand, can be arbitrarily nested to represent complex information, and are immune to common memory errors.

Built-in tools

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

Library utilities

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

Third-party utilities

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

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

It's Mixable

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

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

It's Relatively Easy to Use

Compared to alternatives like C++, Java, and C#, Python programming seems astonishingly simple to most observers. To run Python code in most contexts, you simply type it and run it. There are no intermediate compile and link

steps, like those typical for languages such as C or C++. Python executes programs immediately, which makes for an interactive programming experience and *rapid turnaround* after program changes—in many cases, you can witness the effect of a program change nearly as fast as you can type it.

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

Which is not to say that Python makes programming a *no-brainer*. Python also has convolutions and dark corners that we'll tackle head-on in this book, and some of its roles are more rapid than others. Python's flavor of OOP inheritance, for example, is much more complicated than it once was, and building standalone apps or precompiled programs of the sort you'll meet in the next chapter can still be slow. Measured by its peers, though, Python is dramatically more coder friendly.

It's Relatively Easy to Learn

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

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

That's good news for both professional developers seeking to add the language to their toolbox, and end users of systems that expose a Python layer for scripting roles. Today, many systems rely on the fact that people can learn enough Python to use the system with little or no support. Moreover, Python

has spawned a legion of users who program for need or fun instead of career and may never require full-scale software development skills. Although Python has advanced tools you'll meet in this book, its core fundamentals are accessible to beginners and gurus alike.

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

Chapter Summary

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

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

Test Your Knowledge: Quiz

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

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

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

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

Test Your Knowledge: Answers

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

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