

PART I

Getting Started

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, the first few pages 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 225 groups and over 3,000 students during the last 12 years, 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 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 programming (OOP).

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. 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, 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.

Software Quality

By design, Python implements a deliberately simple and readable syntax and a highly coherent programming model. As a slogan at a recent 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 regular-looking 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.*

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. Today, in an era of layoffs and economic recession, the picture has shifted. Programming staffs are often now 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. In fact, people often use the word “script” instead of “program” to describe a Python code file. In this book, the terms “script” and “program” are used interchangeably, with a slight

* 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 2](#)). This invokes an “Easter egg” hidden in Python—a collection of design principles underlying Python. The acronym EIBTI is now fashionable jargon for the “explicit is better than implicit” rule.

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, 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 17 years and teaching it for 12, the only downside to Python I've found is that, as currently implemented, its execution speed may not always be as fast as that of compiled languages such as C and C++.

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 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.

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 efficient and easy to use. You may never need to code such extensions in your own Python work, but they provide a powerful optimization mechanism if you ever do.

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 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 some products and hardware, further clouding the user-base picture.

In general, though, Python enjoys a large user base and a very active developer community. Because Python has been around for some 19 years and has been widely used, it is also very stable and robust. Besides being employed by individual users, Python is also being applied in real revenue-generating products by real companies. For instance:

- Google makes extensive use of Python in its web search systems, and employs Python's creator.
- The YouTube video sharing service is largely written in Python.
- The popular BitTorrent peer-to-peer file sharing system is a Python program.
- Google's popular App Engine web development framework uses Python as its application language.
- EVE Online, a Massively Multiplayer Online Game (MMOG), makes extensive use of Python.
- Maya, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM use Python for hardware testing.
- Industrial Light & Magic, Pixar, and others use Python in the production of animated movies.
- JPMorgan Chase, UBS, Getco, and Citadel apply Python for financial market forecasting.
- NASA, Los Alamos, Fermilab, JPL, and others use Python for scientific programming tasks.
- iRobot uses Python to develop commercial robotic devices.
- ESRI uses Python as an end-user customization tool for its popular GIS mapping products.
- The NSA uses Python for cryptography and intelligence analysis.
- The IronPort email server product uses more than 1 million lines of Python code to do its job.
- The One Laptop Per Child (OLPC) project builds its user interface and activity model in Python.

And so on. Probably the only common thread amongst 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.

For more details on companies using Python today, see Python’s website at <http://www.python.org>.

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, 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* system, 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. Python comes with a standard object-oriented interface to the Tk GUI API called *tkinter* (*Tkinter* in 2.6) 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 *PythonCard* and *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, generate, and analyze XML files; send, receive, compose, and parse email; fetch web pages by URLs; parse the HTML and XML 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.

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, the IronPython .NET-based implementation, and various CORBA toolkits for Python, provide alternative ways to script components. On Windows, for example, Python scripts can use frameworks to script Word and Excel.

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 have to do is replace the underlying vendor interface.

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 a third-party open source system named *ZODB* that provides a complete object-oriented database system for Python scripts, and others (such as *SQLObject* and *SQLAlchemy*) that map relational tables onto Python's class model. Furthermore, as of Python 2.5, the in-process *SQLite* embedded SQL database engine is a standard part of Python itself.

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

The *NumPy* 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 code.

Gaming, Images, Serial Ports, XML, Robots, and More

Python is commonly applied in more domains than can be mentioned here. For example, you can do:

- Game programming and multimedia in Python with the *pygame* system
- Serial port communication on Windows, Linux, and more with the *PySerial* extension
- Image processing with *PIL*, *PyOpenGL*, *Blender*, *Maya*, and others
- Robot control programming with the *PyRo* toolkit
- XML parsing with the *xml* library package, the *xmlrpclib* module, and third-party extensions
- Artificial intelligence programming with neural network simulators and expert system shells
- Natural language analysis with the *NLTK* package

You can even play solitaire with the *PySol* program. You'll find support for many such fields at the PyPI websites, and via web searches (search Google or <http://www.python.org> for links).

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 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 would find remarkable (if not downright shocking). Python developers coordinate work online with a source-control system. Changes follow a formal *PEP* (Python Enhancement Proposal) protocol and must be accompanied by extensions to Python's extensive 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 current large user base.

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. Attendance at PyCon 2008 nearly *doubled* from the prior year, growing from 586 attendees in 2007 to over 1,000 in 2008. This was on the heels of a 40% attendance increase in 2007, from 410 in 2006. PyCon 2009 had 943 attendees, a slight decrease from 2008, but a still very strong showing during a global recession.

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

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 object-oriented systems languages such as C++ and Java. 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.

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; the ultimate documentation source is at your disposal.

As mentioned earlier, Python development is performed by a community that largely coordinates its efforts over the Internet. It consists of Python's 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. Happily, this tends to make Python more conservative with changes than some other languages.

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 and DOS (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
- 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 2.6), which allows Python programs to implement full-featured graphical user interfaces that run on all major GUI 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.

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, CORBA ORBs, 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 Easy to Use

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 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 languages like C, C++, and Java.

It's Easy to Learn

This brings us to a key point of this book: compared to other programming languages, the core Python language is remarkably easy to learn. In fact, you can expect to be coding significant Python programs in a matter of days (or perhaps in just hours, if you're already an experienced programmer). 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 quickly learn enough Python to tailor their Python customizations' code onsite, with little or no support. Although Python does have advanced programming tools, its core language will still seem 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 that I wish to expose up front. Despite all the reptile icons in the Python world, the truth is that Python creator Guido van Rossum named it after the BBC comedy series *Monty Python's Flying Circus*. He is a big fan of Monty Python, as are many software developers (indeed, there seems to almost be a 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: talks at Python conferences are regularly billed as “The Spanish Inquisition.”

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

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. We talked about performance 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 support for “programming in the large” makes it applicable to the development of larger systems.
- Has a cleaner syntax and simpler design than Perl, which makes it more readable and maintainable and helps reduce program bugs.
- Is simpler and easier to use than Java. Python is a scripting language, but Java inherits much of the complexity and syntax of systems languages such as C++.
- Is simpler and easier to use than C++, but it doesn’t often compete with C++; as a scripting language, Python typically serves different roles.
- Is both more powerful and more cross-platform than Visual Basic. Its open source nature also means it is not controlled by a single company.
- Is more readable and general-purpose than PHP. Python is sometimes used to construct websites, but it’s also widely used in nearly every other computer domain, from robotics to movie animation.
- Is more mature and has a more readable syntax than Ruby. Unlike Ruby and Java, OOP is an option in Python—Python does not impose OOP on users or projects to which it may not apply.
- Has the dynamic flavor of languages like SmallTalk and Lisp, but also has a simple, traditional syntax accessible to developers as well as 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 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. 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 pop quiz about the material presented therein 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. 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 test. Good luck!

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 you got a question right, I encourage you to look at these answers 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, EVE Online, 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 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. `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!

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, so it seems fitting to say a few 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 difficult to read. Given that most software has a lifespan much longer than its initial creation, many see Python as a more effective tool.

The somewhat longer story reflects the backgrounds of the designers of the two languages and underscores some of the main reasons people choose to use Python. Python's creator is a mathematician by training; as such, he produced a language with a high degree of uniformity—its syntax and toolset are remarkably coherent. Moreover, like math, Python's design is orthogonal—most of the language follows from a small set of core concepts. For instance, once one grasps Python's flavor of polymorphism, the rest is largely just details.

By contrast, the creator of the Perl language is a linguist, and its design reflects this heritage. There are many ways to accomplish the same tasks in Perl, and language constructs interact in context-sensitive and sometimes quite subtle ways—much like natural language. As the well-known Perl motto states, "There's more than one way to do it." Given this design, both the Perl language and its user community have historically encouraged 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.

Consider this: when people create a painting or a sculpture, they do so for themselves for purely aesthetic purposes. The possibility of someone else having to change that painting or sculpture later does not 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.

This is where many people find that Python most clearly differentiates itself from scripting languages like Perl. Because Python's syntax model almost forces users to write 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 regularity, 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 creative, too, of course, and as we'll see, the language does offer multiple solutions for some tasks. At its core, though, Python encourages good engineering in ways that other scripting languages often do not.

At least, that's the common consensus among many people who have adopted Python. You should always 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.

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 the Python interpreter. [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 you should feel free to skip parts not relevant to your intended use. Likewise, more advanced 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 this chapter away as “for future reference.” For the rest of you, let’s learn how to run some code.

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, game consoles, and iPods, but installation details vary widely.

Python itself may be fetched from the downloads page on the 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, you'll usually find Python in the Start menu, as captured in [Figure 2-1](#) (these menu options are discussed 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 finesse 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 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.

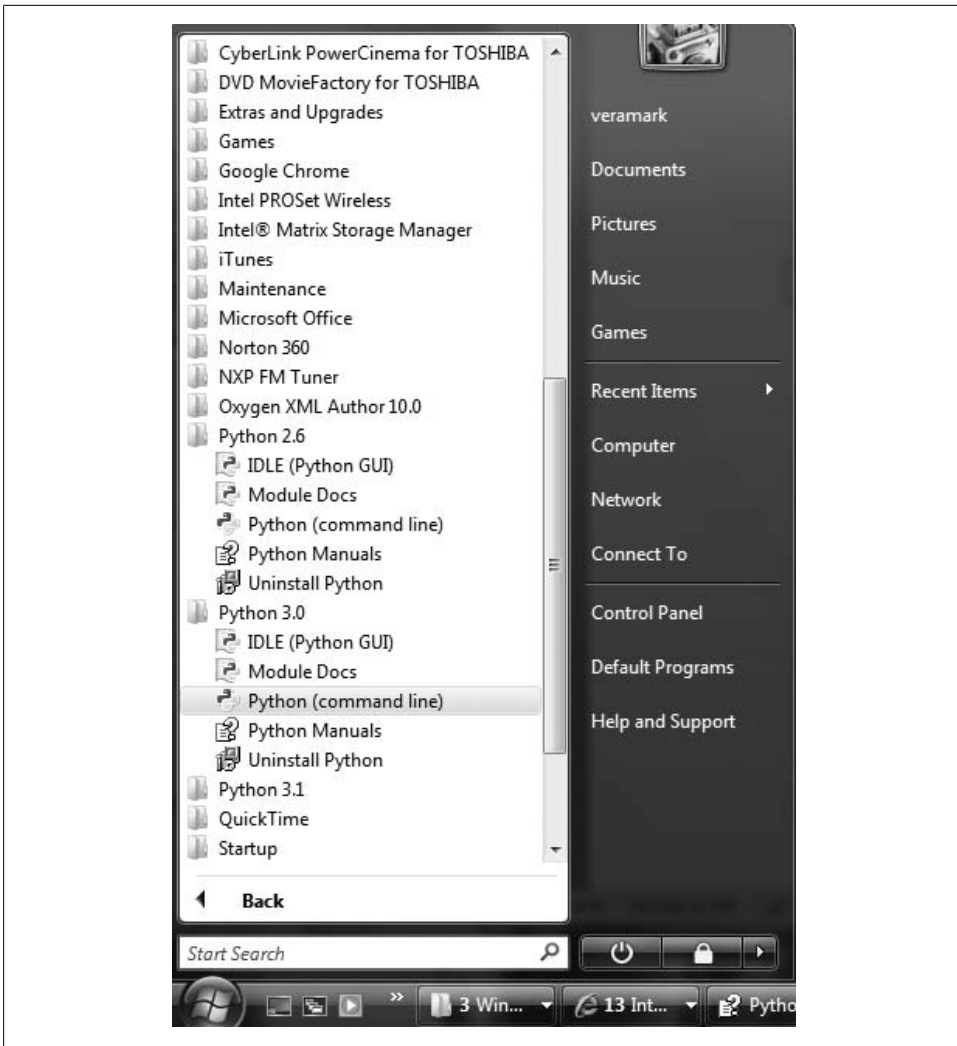


Figure 2-1. When installed on Windows, this is how Python shows up in your Start button menu. This can vary a bit from release to release, but IDLE starts a development GUI, and Python starts a simple interactive session. Also here are the standard manuals and the PyDoc documentation engine (Module Docs).

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,” as shown later in this book, 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 DOS command line on a Windows laptop (typically called a Command Prompt window, found in the Accessories program menu), to make sure it didn't have any silly typos:

```
C:\temp> python script0.py
hello world
1267650600228229401496703205376
```

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). 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).

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. Python automatically checks the timestamps of source and byte code files to know when it must recompile—if you resave your source code, byte code is automatically re-created the next time your program is run.

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 32](#) for another shipping option.)

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 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 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.

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 chip). Byte code is a Python-specific representation.

* And, strictly speaking, byte code is saved only for files that are imported, not for the top-level file of a program. We'll explore imports in [Chapter 3](#), and again in [Part V](#). Byte code is also never saved for code typed at the interactive prompt, which is described in [Chapter 3](#).

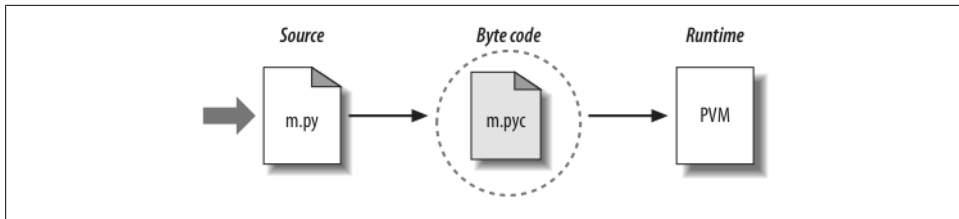


Figure 2-2. Python’s traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

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 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, the net effect makes for a much more dynamic programming experience than that to which some readers may be accustomed.

Execution Model Variations

Before moving on, I should point out that the internal execution flow described in the prior section 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. Let's take a few moments to explore the most prominent of these variations.

Python Implementation Alternatives

Really, as this book is being written, there are three primary implementations of the Python language—*CPython*, *Jython*, and *IronPython*—along with a handful of secondary implementations such as *Stackless Python*. In brief, CPython is the standard implementation; all the others have very specific purposes and roles. All implement the same Python language but execute programs in different ways.

CPython

The original, and standard, implementation of Python is usually called CPython, when you want to contrast it with the other two. Its 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 distribution, 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 is using Python in very specialized ways.

Unless you want to script Java or .NET applications with Python, 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 robust than the alternative systems. [Figure 2-2](#) reflects CPython's runtime architecture.

Jython

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. 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 be a frontend to Java code.

IronPython

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, accessible from other .NET languages.

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. Because it is being developed by Microsoft, though, IronPython might also be able to leverage some important optimization tools for better performance. IronPython's scope is still evolving as I write this; for more details, consult the Python online resources or search the Web.[†]

Execution Optimization Tools

CPython, Jython, and IronPython 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. Still other systems, including the Psyco just-in-time compiler and the Shedskin C++ translator, 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.

The Psyco 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. 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 real binary machine code for faster execution. Psyco accomplishes this

[†] Jython and IronPython are completely independent implementations of Python that compile Python source for different runtime architectures. It is also possible to access Java and .NET software from standard CPython programs: JType and Python for .NET systems, for example, allow CPython code to call out to Java and .NET components.

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 net effect is that, with Psyco, your program becomes much quicker over time and as it is running. 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 generally known as a *just-in-time* (JIT) compiler. Psyco is actually a bit 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 has been shown to speed 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. With Psyco, such migrations become even less important.

Psyco is not yet a standard part of Python; you will have to fetch and install it separately. It is also still something of a research project, so you'll have to track its evolution online. In fact, at this writing, although Psyco can still be fetched and installed by itself, it appears that much of the system may eventually be absorbed into the newer “PyPy” project—an attempt to reimplement Python's PVM in Python code, to better support optimizations like Psyco.

Perhaps the largest downside of Psyco is that it currently only generates machine code for Intel x86 architecture chips, though this includes Windows and Linux boxes and recent Macs. For more details on the Psyco extension, and other JIT efforts that may arise, consult <http://www.python.org>; you can also check out Psyco's home page, which currently resides at <http://psyco.sourceforge.net>.

The Shedskin C++ translator

Shedskin 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. Shedskin is still somewhat experimental as I write these words, and it limits Python programs to an implicit statically typed constraint that 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 the Psyco extension in terms of execution speed, and it is a promising project. Search the Web for details on the project's current status.

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.

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 byte code and PVM are merged into a single component—a frozen binary file.

Today, three primary systems are capable of generating frozen binaries: *py2exe* (for Windows), *PyInstaller* (which is similar to *py2exe* but also works on Linux and Unix and is capable of generating self-installing binaries), and *freeze* (the original). You may have to fetch these tools separately from Python itself, but they are available free of charge. They are also constantly evolving, so consult <http://www.python.org> or your favorite web search engine for more on these tools. 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 not 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.

Other Execution Options

Still other schemes for running Python programs have more focused goals:

- The *Stackless Python* system is a standard CPython implementation variant that does not save state on the C language call stack. This makes Python more easy to port to small stack architectures, provides efficient multiprocessing options, and fosters novel programming structures such as coroutines.
- 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.

For more details on these systems, search the Web for recent links.

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 one has not in nearly two decades!). New byte code formats and implementation variants may also be adopted in the future. For instance:

- The *Parrot* project aims to provide a common byte code format, virtual machine, and optimization techniques for a variety of programming languages (see <http://www.python.org>). Python's own PVM runs Python code more efficiently than Parrot, but it's unclear how Parrot will evolve.
- The *PyPy* project is an attempt to reimplement the PVM in Python itself to enable new implementation techniques. Its goal is to produce a fast and flexible implementation of Python.
- The Google-sponsored *Unladen Swallow* project aims to make standard Python faster by a factor of at least 5, and fast enough to replace the C language in many contexts. It is an optimization branch of CPython, intended to be fully compatible and significantly faster. This project also hopes to remove the Python multithreading Global Interpreter Lock (GIL), which prevents pure Python threads from truly overlapping in time. This is currently an emerging project being developed as open source by Google engineers; it is initially targeting Python 2.6, though 3.0 may acquire its changes too. Search Google for up-to-date details.

Although such 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 break 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 variations on Python's standard execution model.
6. How are CPython, Jython, and IronPython different?

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, Shedskin, and frozen binaries are all variations on the execution model.
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.

How You Run Programs

OK, it's time to start running some code. Now that you have a handle on program execution, 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 not, see the prior chapter and [Appendix A](#) for installation and configuration hints.

There are a variety of 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 type code *interactively* and how to save it in *files* to be run with system command lines, icon clicks, module imports and reloads, `exec` calls, menu options in GUIs such as IDLE, and more.

If you just want to find out how to run a Python program quickly, you may be tempted to read the parts of this chapter that pertain only to your platform and move on to [Chapter 4](#). But don't skip the material on module imports, as that's essential to understanding Python's program architecture. I also encourage you to at least skim the sections on IDLE and other IDEs, so you'll know what tools are available for when you start developing more sophisticated Python programs.

The Interactive Prompt

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 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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. 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.k.a. the Command Prompt, usually found in the Accessories section of the Start→Programs menu) or in the Start→Run... dialog box.
- On Unix, Linux, and Mac OS X, you might type this command in a shell or terminal window (e.g., 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 generally click the Python icon in the home or application window to launch an interactive session.

If you have not set your shell’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, **/usr/local/bin/python** or **/usr/bin/python** will often suffice. On Windows, try typing **C:\Python30\python** (for version 3.0):

```
C:\misc> c:\python30\python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Alternatively, you can run a change-directory command to go to Python’s install directory before typing “python”—try the **cd c:\python30** command on Windows, for example:

```
C:\misc> cd C:\Python30
C:\Python30> python
Python 3.0.1 (r301:69561, Feb 13 2009, 20:04:18) [MSC v.1500 32 bit (Intel)] ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

On Windows, besides typing **python** in a shell window, you can also begin similar interactive sessions by starting IDLE’s main window (discussed later) or by selecting the “Python (command line)” menu option from the Start button menu for Python, as shown in [Figure 2-1](#) back in [Chapter 2](#). Both spawn a Python interactive prompt with equivalent functionality; typing a shell command isn’t necessary.

Running Code Interactively

However it's started, the Python interactive session begins by printing two lines of informational text (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 after the `>>>` 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 3.0, but not in 2.6, so the parentheses here are required in 3.0 only):

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

Again, you don't need to 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
>>>
% <== 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, and the last two lines typed are expressions (`lumberjack` and `2 ** 8`)—their results are displayed automatically. To exit an interactive session like this one and return to your system shell prompt, type Ctrl-D on Unix-like machines; on MS-DOS and Windows systems, type Ctrl-Z to exit. In the IDLE GUI discussed later, either type Ctrl-D or simply close the window.

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 and books to try to figure out what the code does, or you can simply run it interactively:

```
>>> '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). 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 read more on debugging in the sidebar [“Debugging Python Code” on page 67](#)).

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.

For instance, of 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), but you can do the same once you start writing module files of your own:

```
>>> import os
>>> os.getcwd()                             <== Testing on the fly
'c:\\Python30'
```

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.

Using 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 this 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 the Python 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 not 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. This seems to be a recurring confusion in introductory Python classes. Remember, a leading space generates an error message.
- **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 interface, lines after the first are 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 received emails from readers who'd gotten burned by the last two points as I was updating this chapter, so it probably merits 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. *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.

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)  
...     print('done')  
...  
File "<stdin>", line 3  
    print('done')  
    ^  
SyntaxError: invalid syntax  
                                     <== Need to press Enter twice before a new statement
```

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—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 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. (More on the meaning of “top-level” and imports in a few moments.)

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, 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.

A First Script

Let’s get started. Open your favorite text editor (e.g., *vi*, Notepad, or the IDLE editor), and type the following statements into a new text file named *script1.py*:

```
# 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. 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 as well. 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.

Running Files with Command Lines

Once you’ve saved this text file, you can ask Python to run it by listing its full filename as the first argument to a `python` command, typed at the system shell prompt:

```
% python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

Again, you can type such a system shell command in whatever your system provides for command-line entry—a Windows Command Prompt window, an xterm window, or similar. Remember to replace “python” with a full directory path, as before, if your `PATH` setting is not configured.

If all works as planned, this shell command makes Python run the code in this file line by line, and you will see the output of the script’s three `print` statements—the name of the underlying platform, 2 raised to the power 100, and the result of the same string repetition expression we saw earlier (again, more on the last two of these in [Chapter 4](#)).

If all didn’t work as planned, you’ll get an error message—make sure you’ve entered the code in your file exactly as shown, and try again. We’ll talk about debugging options in the sidebar “[Debugging Python Code](#)” on page 67, but at this point in the book your best bet is probably rote imitation.

Because this scheme uses shell command lines to start Python programs, all the usual shell syntax applies. For instance, you can route the output of a Python script to a file to save it for later use or inspection by using special shell syntax:

```
% python script1.py > saveit.txt
```

In this case, the three output lines shown in the prior run are stored in the file *saveit.txt* instead of being printed. This is generally known as *stream redirection*; it works for input and output text and is available on Windows and Unix-like systems. It also has little to do with Python (Python simply supports it), so we will skip further details on shell redirection syntax here.

If you are working on a Windows platform, this example works the same, but the system prompt is normally different:

```
C:\Python30> python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

As usual, be sure to type the full path to Python if you haven't set your `PATH` environment variable to include this path or run a change-directory command to go to the path:

```
D:\temp> C:\python30\python script1.py
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

On all recent versions of Windows, you can also type just the name of your script, and omit the name of Python itself. Because newer Windows systems use the Windows Registry to find a program with which to run a file, you don't need to name "python" on the command line explicitly to run a *.py* file. The prior command, for example, could be simplified to this on most Windows machines:

```
D:\temp> script1.py
```

Finally, remember to give the full path to your script file if it lives in a different directory from the one in which you are working. For example, the following system command line, run from *D:\other*, assumes Python is in your system path but runs a file located elsewhere:

```
D:\other> python c:\code\otherscript.py
```

If your `PATH` doesn't include Python's directory, and neither Python nor your script file is in the directory you're working in, use full paths for both:

```
D:\other> C:\Python30\python c:\code\otherscript.py
```

Using Command Lines and Files

Running program files from system command lines is also a fairly straightforward launch option, especially if you are familiar with command lines in general from prior work. For newcomers, though, here are a few pointers about common beginner traps that might help you avoid some frustration:

- **Beware of automatic extensions on Windows.** If you use the Notepad program to code program files on Windows, be careful to pick the type All Files when it comes time to save your file, and give the file a `.py` suffix explicitly. Otherwise, Notepad will save your file with a `.txt` extension (e.g., as *script1.py.txt*), making it difficult to run in some launching schemes.

Worse, Windows hides file extensions by default, so unless you have changed your view options you may not even notice that you've coded a text file and not a Python file. The file's icon may give this away—if it doesn't have a snake on it, you may have trouble. Uncolored code in IDLE and files that open to edit instead of run when clicked are other symptoms of this problem.

Microsoft Word similarly adds a `.doc` extension by default; much worse, it adds formatting characters that are not legal Python syntax. As a rule of thumb, always pick All Files when saving under Windows, or use a more programmer-friendly text editor such as IDLE. IDLE does not even add a `.py` suffix automatically—a feature programmers tend to like, but users do not.

- **Use file extensions and directory paths at system prompts, but not for imports.** Don't forget to type the full name of your file in system command lines—that is, use `python script1.py` rather than `python script1`. By contrast, Python's `import` statements, which we'll meet later in this chapter, omit both the `.py` file suffix and the directory path (e.g., `import script1`). This may seem trivial, but confusing these two is a common mistake.

At the system prompt, you are in a system shell, not Python, so Python's module file search rules do not apply. Because of that, you must include both the `.py` extension and, if necessary, the full directory path leading to the file you wish to run. For instance, to run a file that resides in a different directory from the one in which you are working, you would typically list its full path (e.g., `python d:\tests\spam.py`). Within Python code, however, you can just say `import spam` and rely on the Python module search path to locate your file, as described later.

- **Use print statements in files.** Yes, we've already been over this, but it is such a common mistake that it's worth repeating at least once here. Unlike in interactive coding, you generally must use `print` statements to see output from program files. If you don't see any output, make sure you've said "print" in your file. Again, though, `print` statements are *not* required in an interactive session, since Python automatically echoes expression results; `prints` don't hurt here, but are superfluous extra typing.

Unix Executable Scripts (#!)

If you are going to use Python on a Unix, Linux, or Unix-like system, you can also turn files of Python code into executable programs, much as you would for programs coded in a shell language such as *bash* or *ksh*. Such files are usually called *executable scripts*. In simple terms, Unix-style executable scripts are just normal text files containing Python statements, but with two special properties:

- **Their first line is special.** Scripts usually start with a line that begins with the characters `#!` (often called “hash bang”), followed by the path to the Python interpreter on your machine.
- **They usually have executable privileges.** Script files are usually marked as executable to tell the operating system that they may be run as top-level programs. On Unix systems, a command such as `chmod +x file.py` usually does the trick.

Let’s look at an example for Unix-like systems. Use your text editor again to create a file of Python code called *brian*:

```
#!/usr/local/bin/python
print('The Bright Side ' + 'of Life...')           # + means concatenate for strings
```

The special line at the top of the file tells the system where the Python interpreter lives. Technically, the first line is a Python comment. As mentioned earlier, all comments in Python programs start with a `#` and span to the end of the line; they are a place to insert extra information for human readers of your code. But when a comment such as the first line in this file appears, it’s special because the operating system uses it to find an interpreter for running the program code in the rest of the file.

Also, note that this file is called simply *brian*, without the *.py* suffix used for the module file earlier. Adding a *.py* to the name wouldn’t hurt (and might help you remember that this is a Python program file), but because you don’t plan on letting other modules import the code in this file, the name of the file is irrelevant. If you give the file executable privileges with a `chmod +x brian` shell command, you can run it from the operating system shell as though it were a binary program:

```
% brian
The Bright Side of Life...
```

A note for Windows users: the method described here is a Unix trick, and it may not work on your platform. Not to worry; just use the basic command-line technique explored earlier. List the file’s name on an explicit `python` command line:

* As we discussed when exploring command lines, modern Windows versions also let you type just the name of a *.py* file at the system command line—they use the Registry to determine that the file should be opened with Python (e.g., typing `brian.py` is equivalent to typing `python brian.py`). This command-line mode is similar in spirit to the Unix `#!`, though it is system-wide on Windows, not per-file. Note that some programs may actually interpret and use a first `#!` line on Windows much like on Unix, but the DOS system shell on Windows simply ignores it.

```
C:\misc> python brian
The Bright Side of Life...
```

In this case, you don't need the special `#!` comment at the top (although Python just ignores it if it's present), and the file doesn't need to be given executable privileges. In fact, if you want to run files portably between Unix and Microsoft Windows, your life will probably be simpler if you always use the basic command-line approach, not Unix-style scripts, to launch programs.

The Unix `env` Lookup Trick

On some Unix systems, you can avoid hardcoding the path to the Python interpreter by writing the special first-line comment like this:

```
#!/usr/bin/env python
...script goes here...
```

When coded this way, the `env` program locates the Python interpreter according to your system search path settings (i.e., in most Unix shells, by looking in all the directories listed in the `PATH` environment variable). This scheme can be more portable, as you don't need to hardcode a Python install path in the first line of all your scripts.

Provided you have access to `env` everywhere, your scripts will run no matter where Python lives on your system—you need only change the `PATH` environment variable settings across platforms, not in the first line in all your scripts. Of course, this assumes that `env` lives in the same place everywhere (on some machines, it may be in `/sbin`, `/bin`, or elsewhere); if not, all portability bets are off!

Clicking File Icons

On Windows, the Registry makes opening files with icon clicks easy. Python automatically registers itself to be the program that opens Python program files when they are clicked. Because of that, it is possible to launch the Python programs you write by simply clicking (or double-clicking) on their file icons with your mouse cursor.

On non-Windows systems, you will probably be able to perform a similar trick, but the icons, file explorer, navigation schemes, and more may differ slightly. On some Unix systems, for instance, you may need to register the `.py` extension with your file explorer GUI, make your script executable using the `#!` trick discussed in the previous section, or associate the file MIME type with an application or command by editing files, installing programs, or using other tools. See your file explorer's documentation for more details if clicks do not work correctly right off the bat.

Clicking Icons on Windows

To illustrate, let's keep using the script we wrote earlier, `script1.py`, repeated here to minimize page flipping:

```

# 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

```

As we've seen, you can always run this file from a system command line:

```

C:\misc> c:\python30\python script1.py
win32
1267650600228229401496703205376

```

However, icon clicks allow you to run the file without any typing at all. If you find this file's icon—for instance, by selecting Computer (or My Computer in XP) in your Start menu and working your way down on the C drive on Windows—you will get the file explorer picture captured in [Figure 3-1](#) (Windows Vista is being used here). Python source files show up with white backgrounds on Windows, and byte code files show up with black backgrounds. You will normally want to click (or otherwise run) the source code file, in order to pick up your most recent changes. To launch the file here, simply click on the icon for *script1.py*.

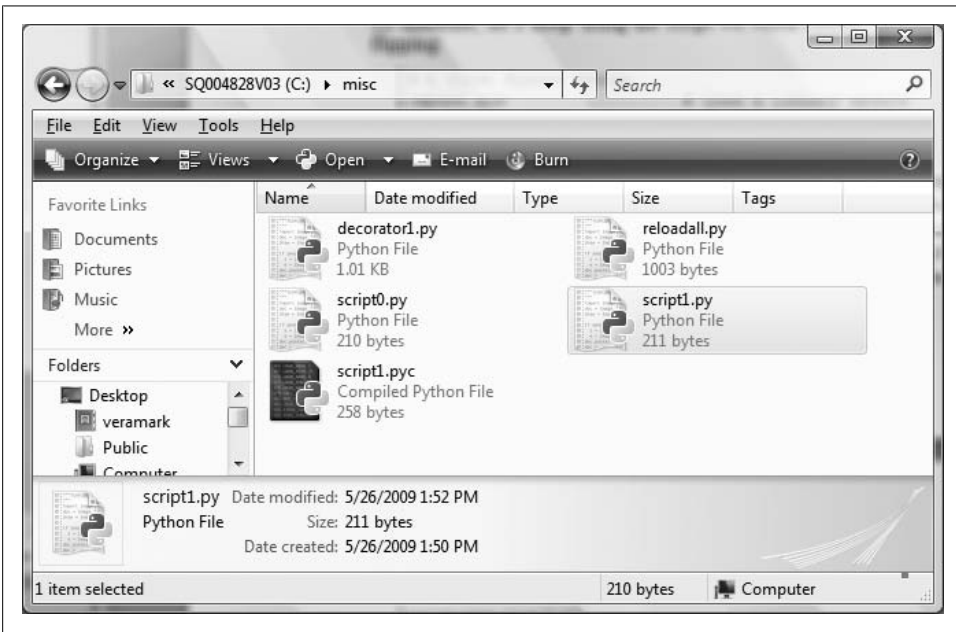


Figure 3-1. On Windows, Python program files show up as icons in file explorer windows and can automatically be run with a double-click of the mouse (though you might not see printed output or error messages this way).

The input Trick

Unfortunately, on Windows, the result of clicking on a file icon may not be incredibly satisfying. In fact, as it is, this example script generates a perplexing “flash” when clicked—not exactly the sort of feedback that budding Python programmers usually hope for! This is not a bug, but has to do with the way the Windows version of Python handles printed output.

By default, Python generates a pop-up black DOS console window to serve as a clicked file’s input and output. If a script just prints and exits, well, it just prints and exits—the console window appears, and text is printed there, but the console window closes and disappears on program exit. Unless you are very fast, or your machine is very slow, you won’t get to see your output at all. Although this is normal behavior, it’s probably not what you had in mind.

Luckily, it’s easy to work around this. If you need your script’s output to stick around when you launch it with an icon click, simply put a call to the built-in `input` function at the very bottom of the script (`raw_input` in 2.6: see the note ahead). For example:

```
# 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
input()                   # <== ADDED
```

In general, `input` reads the next line of standard input, waiting if there is none yet available. The net effect in this context will be to pause the script, thereby keeping the output window shown in [Figure 3-2](#) open until you press the Enter key.



Figure 3-2. When you click a program’s icon on Windows, you will be able to see its printed output if you include an `input` call at the very end of the script. But you only need to do so in this context!

Now that I've shown you this trick, keep in mind that it is usually only required for Windows, and then only if your script prints text and exits and only if you will launch the script by clicking its file icon. You should add this call to the bottom of your top-level files if and only if all of these three conditions apply. There is no reason to add this call in any other contexts (unless you're unreasonably fond of pressing your computer's Enter key!).[†] That may sound obvious, but it's another common mistake in live classes.

Before we move ahead, note that the `input` call applied here is the input counterpart of using the `print` statement for outputs. It is the simplest way to read user input, and it is more general than this example implies. For instance, `input`:

- Optionally accepts a string that will be printed as a prompt (e.g., `input('Press Enter to exit')`)
- Returns to your script a line of text read as a string (e.g., `nextinput = input()`)
- Supports input stream redirections at the system shell level (e.g., `python spam.py < input.txt`), just as the `print` statement does for output

We'll use `input` in more advanced ways later in this text; for instance, [Chapter 10](#) will apply it in an interactive loop.



Version skew note: If you are working in Python 2.6 or earlier, use `raw_input()` instead of `input()` in this code. The former was renamed to the latter in Python 3.0. Technically, 2.6 has an `input` too, but it also *evaluates* strings as though they are program code typed into a script, and so will not work in this context (an empty string is an error). Python 3.0's `input` (and 2.6's `raw_input`) simply returns the entered text as a string, unevaluated. To simulate 2.6's `input` in 3.0, use `eval(input())`.

Other Icon-Click Limitations

Even with the `input` trick, clicking file icons is not without its perils. You also may not get to see Python error messages. If your script generates an error, the error message text is written to the pop-up console window—which then immediately disappears! Worse, adding an `input` call to your file will not help this time because your script will likely abort long before it reaches this call. In other words, you won't be able to tell what went wrong.

[†] It is also possible to completely suppress the pop-up DOS console window for clicked files on Windows. Files whose names end in a `.pyw` extension will display only windows constructed by your script, not the default DOS console window. `.pyw` files are simply `.py` source files that have this special operational behavior on Windows. They are mostly used for Python-coded user interfaces that build windows of their own, often in conjunction with various techniques for saving printed output and errors to files.

Because of these limitations, it is probably best to view icon clicks as a way to launch programs after they have been debugged or have been instrumented to write their output to a file. Especially when starting out, use other techniques—such as system command lines and IDLE (discussed further in the section “[The IDLE User Interface](#)” on page 58)—so that you can see generated error messages and view your normal output without resorting to coding tricks. When we discuss exceptions later in this book, you’ll also learn that it is possible to intercept and recover from errors so that they do not terminate your programs. Watch for the discussion of the `try` statement later in this book for an alternative way to keep the console window from closing on errors.

Module Imports and Reloads

So far, I’ve been talking about “importing modules” without really explaining what this term means. We’ll study modules and larger program architecture in depth in [Part V](#), but because imports are also a way to launch programs, this section will introduce enough module basics to get you started.

In simple terms, every file of Python source code whose name ends in a `.py` extension is a module. Other files can access the items a module defines by *importing* that module; `import` operations essentially load another file and grant access to that file’s contents. The contents of a module are made available to the outside world through its attributes (a term I’ll define in the next section).

This module-based services model turns out to be the core idea behind program architecture in Python. Larger programs usually take the form of multiple module files, which import tools from other module files. One of the modules is designated as the main or *top-level* file, and this is the one launched to start the entire program.

We’ll delve into such architectural issues in more detail later in this book. This chapter is mostly interested in the fact that import operations *run* the code in a file that is being loaded as a final step. Because of this, importing a file is yet another way to launch it.

For instance, if you start an interactive session (from a system command line, from the Start menu, from IDLE, or otherwise), you can run the `script1.py` file you created earlier with a simple import (be sure to delete the `input` line you added in the prior section first, or you’ll need to press Enter for no reason):

```
C:\misc> c:\python30\python
>>> import script1
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

This works, but only once per session (really, process) by default. After the first import, later imports do nothing, even if you change and save the module's source file again in another window:

```
>>> import script1
>>> import script1
```

This is by design; imports are too expensive an operation to repeat more than once per file, per program run. As you'll learn in [Chapter 21](#), imports must find files, compile them to byte code, and run the code.

If you really want to force Python to run the file again in the same session without stopping and restarting the session, you need to instead call the `reload` function available in the `imp` standard library module (this function is also a simple built-in in Python 2.6, but not in 3.0):

```
>>> from imp import reload          # Must load from module in 3.0
>>> reload(script1)
win32
65536
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
<module 'script1' from 'script1.py'>
>>>
```

The `from` statement here simply copies a name out of a module (more on this soon). The `reload` function itself loads and runs the current version of your file's code, picking up changes if you've changed and saved it in another window.

This allows you to edit and pick up new code on the fly within the current Python interactive session. In this session, for example, the second `print` statement in `script1.py` was changed in another window to `print 2 ** 16` between the time of the first `import` and the `reload` call.

The `reload` function expects the name of an already loaded module object, so you have to have successfully imported a module once before you reload it. Notice that `reload` also expects parentheses around the module object name, whereas `import` does not. `reload` is a function that is *called*, and `import` is a statement.

That's why you must pass the module name to `reload` as an argument in parentheses, and that's why you get back an extra output line when reloading. The last output line is just the display representation of the `reload` call's return value, a Python module object. We'll learn more about using functions in general in [Chapter 16](#).



Version skew note: Python 3.0 moved the `reload` built-in function to the `imp` standard library module. It still reloads files as before, but you must import it in order to use it. In 3.0, run an `import imp` and use `imp.reload(M)`, or run a `from imp import reload` and use `reload(M)`, as shown here. We'll discuss `import` and `from` statements in the next section, and more formally later in this book.

If you are working in Python 2.6 (or 2.X in general), `reload` is available as a built-in function, so no import is required. In Python 2.6, `reload` is available in *both* forms—built-in and module function—to aid the transition to 3.0. In other words, reloading is still available in 3.0, but an extra line of code is required to fetch the `reload` call.

The move in 3.0 was likely motivated in part by some well-known issues involving `reload` and `from` statements that we'll encounter in the next section. In short, names loaded with a `from` are not directly updated by a `reload`, but names accessed with an `import` statement are. If your names don't seem to change after a `reload`, try using `import` and `module.attribute` name references instead.

The Grander Module Story: Attributes

Imports and reloads provide a natural program launch option because import operations execute files as a last step. In the broader scheme of things, though, modules serve the role of *libraries* of tools, as you'll learn in [Part V](#). More generally, a module is mostly just a package of variable names, known as a *namespace*. The names within that package are called *attributes*—an attribute is simply a variable name that is attached to a specific object (like a module).

In typical use, importers gain access to all the names assigned at the top level of a module's file. These names are usually assigned to tools exported by the module—functions, classes, variables, and so on—that are intended to be used in other files and other programs. Externally, a module file's names can be fetched with two Python statements, `import` and `from`, as well as the `reload` call.

To illustrate, use a text editor to create a one-line Python module file called *myfile.py* with the following contents:

```
title = "The Meaning of Life"
```

This may be one of the world's simplest Python modules (it contains a single assignment statement), but it's enough to illustrate the point. When this file is imported, its code is run to generate the module's attribute. The assignment statement creates a module attribute named `title`.

You can access this module's `title` attribute in other components in two different ways. First, you can load the module as a whole with an `import` statement, and then *qualify* the module name with the attribute name to fetch it:

```
% python                                # Start Python
>>> import myfile                       # Run file; load module as a whole
>>> print(myfile.title)                 # Use its attribute names: '.' to qualify
The Meaning of Life
```

In general, the dot expression syntax *object.attribute* lets you fetch any attribute attached to any object, and this is a very common operation in Python code. Here, we've used it to access the string variable `title` inside the module `myfile`—in other words, `myfile.title`.

Alternatively, you can fetch (really, copy) names out of a module with `from` statements:

```
% python                                # Start Python
>>> from myfile import title            # Run file; copy its names
>>> print(title)                       # Use name directly: no need to qualify
The Meaning of Life
```

As you'll see in more detail later, `from` is just like an `import`, with an extra assignment to names in the importing component. Technically, `from` copies a module's *attributes*, such that they become simple *variables* in the recipient—thus, you can simply refer to the imported string this time as `title` (a variable) instead of `myfile.title` (an attribute reference).‡

Whether you use `import` or `from` to invoke an import operation, the statements in the module file *myfile.py* are executed, and the importing component (here, the interactive prompt) gains access to names assigned at the top level of the file. There's only one such name in this simple example—the variable `title`, assigned to a string—but the concept will be more useful when you start defining objects such as functions and classes in your modules: such objects become reusable software components that can be accessed by name from one or more client modules.

In practice, module files usually define more than one name to be used in and outside the files. Here's an example that defines three:

```
a = 'dead'                             # Define three attributes
b = 'parrot'                           # Exported to other files
c = 'sketch'
print(a, b, c)                         # Also used in this file
```

This file, *threenames.py*, assigns three variables, and so generates three attributes for the outside world. It also uses its own three variables in a `print` statement, as we see when we run this as a top-level file:

‡ Notice that `import` and `from` both list the name of the module file as simply *myfile* without its *.py* suffix. As you'll learn in [Part V](#), when Python looks for the actual file, it knows to include the suffix in its search procedure. Again, you must include the *.py* suffix in system shell command lines, but not in `import` statements.

```
% python threenames.py
dead parrot sketch
```

All of this file’s code runs as usual the first time it is imported elsewhere (by either an `import` or `from`). Clients of this file that use `import` get a module with attributes, while clients that use `from` get copies of the file’s names:

```
% python
>>> import threenames                # Grab the whole module
dead parrot sketch
>>>
>>> threenames.b, threenames.c
('parrot', 'sketch')
>>>
>>> from threenames import a, b, c    # Copy multiple names
>>> b, c
('parrot', 'sketch')
```

The results here are printed in parentheses because they are really *tuples* (a kind of object covered in the next part of this book); you can safely ignore them for now.

Once you start coding modules with multiple names like this, the built-in `dir` function starts to come in handy—you can use it to fetch a list of the names available inside a module. The following returns a Python list of strings (we’ll start studying lists in the next chapter):

```
>>> dir(threenames)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'a', 'b', 'c']
```

I ran this on Python 3.0 and 2.6; older Pythons may return fewer names. When the `dir` function is called with the name of an imported module passed in parentheses like this, it returns all the attributes inside that module. Some of the names it returns are names you get “for free”: names with leading and trailing double underscores are built-in names that are always predefined by Python and that have special meaning to the interpreter. The variables our code defined by assignment—`a`, `b`, and `c`—show up last in the `dir` result.

Modules and namespaces

Module imports are a way to run files of code, but, as we’ll discuss later in the book, modules are also the largest program structure in Python programs.

In general, Python programs are composed of multiple module files, linked together by `import` statements. Each module file is a self-contained package of variables—that is, a namespace. One module file cannot see the names defined in another file unless it explicitly imports that other file, so modules serve to minimize name collisions in your code—because each file is a self-contained namespace, the names in one file cannot clash with those in another, even if they are spelled the same way.

In fact, as you'll see, modules are one of a handful of ways that Python goes to great lengths to package your variables into compartments to avoid name clashes. We'll discuss modules and other namespace constructs (including classes and function scopes) further later in the book. For now, modules will come in handy as a way to run your code many times without having to retype it.



import versus from: I should point out that the `from` statement in a sense defeats the namespace partitioning purpose of modules—because the `from` copies variables from one file to another, it can cause same-named variables in the importing file to be overwritten (and won't warn you if it does). This essentially collapses namespaces together, at least in terms of the copied variables.

Because of this, some recommend using `import` instead of `from`. I won't go that far, though; not only does `from` involve less typing, but its purported problem is rarely an issue in practice. Besides, this is something *you* control by listing the variables you want in the `from`; as long as you understand that they'll be assigned values, this is no more dangerous than coding assignment statements—another feature you'll probably want to use!

import and reload Usage Notes

For some reason, once people find out about running files using `import` and `reload`, many tend to focus on this alone and forget about other launch options that always run the current version of the code (e.g., icon clicks, IDLE menu options, and system command lines). This approach can quickly lead to confusion, though—you need to remember when you've imported to know if you can reload, you need to remember to use parentheses when you call `reload` (only), and you need to remember to use `reload` in the first place to get the current version of your code to run. Moreover, reloads aren't transitive—reloading a module reloads that module only, not any modules it may import—so you sometimes have to reload multiple files.

Because of these complications (and others we'll explore later, including the `reload/from` issue mentioned in a prior note in this chapter), it's generally a good idea to avoid the temptation to launch by imports and reloads for now. The IDLE Run→Run Module menu option described in the next section, for example, provides a simpler and less error-prone way to run your files, and always runs the current version of your code. System shell command lines offer similar benefits. You don't need to use `reload` if you use these techniques.

In addition, you may run into trouble if you use modules in unusual ways at this point in the book. For instance, if you want to import a module file that is stored in a directory other than the one you're working in, you'll have to skip ahead to [Chapter 21](#) and learn about the *module search path*.

For now, if you must import, try to keep all your files in the directory you are working in to avoid complications.[§]

That said, imports and reloads have proven to be a popular testing technique in Python classes, and you may prefer using this approach too. As usual, though, if you find yourself running into a wall, stop running into a wall!

Using `exec` to Run Module Files

In fact, there are more ways to run code stored in module files than have yet been exposed here. For instance, the `exec(open('module.py').read())` built-in function call is another way to launch files from the interactive prompt without having to import and later reload. Each `exec` runs the current version of the file, without requiring later reloads (*script1.py* is as we left it after a reload in the prior section):

```
C:\misc> c:\python30\python
>>> exec(open('script1.py').read())
win32
65536
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!

...change script1.py in a text edit window...

>>> exec(open('script1.py').read())
win32
4294967296
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
```

The `exec` call has an effect similar to an import, but it doesn't technically import the module—by default, each time you call `exec` this way it runs the file anew, as though you had pasted it in at the place where `exec` is called. Because of that, `exec` does not require module reloads after file changes—it skips the normal module import logic.

On the downside, because it works as if pasting code into the place where it is called, `exec`, like the `from` statement mentioned earlier, has the potential to silently overwrite variables you may currently be using. For example, our *script1.py* assigns to a variable named `x`. If that name is also being used in the place where `exec` is called, the name's value is replaced:

```
>>> x = 999
>>> exec(open('script1.py').read())    # Code run in this namespace by default
...same output...
>>> x                                  # Its assignments can overwrite names here
'Spam!'
```

[§] If you're burning with curiosity, the short story is that Python searches for imported modules in every directory listed in `sys.path`—a Python list of directory name strings in the `sys` module, which is initialized from a `PYTHONPATH` environment variable, plus a set of standard directories. If you want to import from a directory other than the one you are working in, that directory must generally be listed in your `PYTHONPATH` setting. For more details, see [Chapter 21](#).

By contrast, the basic `import` statement runs the file only once per process, and it makes the file a separate module namespace so that its assignments will not change variables in your scope. The price you pay for the namespace partitioning of modules is the need to reload after changes.



Version skew note: Python 2.6 also includes an `execfile('module.py')` built-in function, in addition to allowing the form `exec(open('module.py'))`, which both automatically read the file's content. Both of these are equivalent to the `exec(open('module.py').read())` form, which is more complex but runs in both 2.6 and 3.0.

Unfortunately, neither of these two simpler 2.6 forms is available in 3.0, which means you must understand both files and their read methods to fully understand this technique today (alas, this seems to be a case of aesthetics trouncing practicality in 3.0). In fact, the `exec` form in 3.0 involves so much typing that the best advice may simply be not to do it—it's usually best to launch files by typing system shell command lines or by using the IDLE menu options described in the next section. For more on the 3.0 `exec` form, see [Chapter 9](#).

The IDLE User Interface

So far, we've seen how to run Python code with the interactive prompt, system command lines, icon clicks, and module imports and `exec` calls. If you're looking for something a bit more visual, IDLE provides a graphical user interface for doing Python development, and it's a standard and free part of the Python system. It is usually referred to as an *integrated development environment* (IDE), because it binds together various development tasks into a single view.^{||}

In short, IDLE is a GUI that lets you edit, run, browse, and debug Python programs, all from a single interface. Moreover, because IDLE is a Python program that uses the *tkinter* GUI toolkit (known as Tkinter in 2.6), it runs portably on most Python platforms, including Microsoft Windows, X Windows (for Linux, Unix, and Unix-like platforms), and the Mac OS (both Classic and OS X). For many, IDLE represents an easy-to-use alternative to typing command lines, and a less problem-prone alternative to clicking on icons.

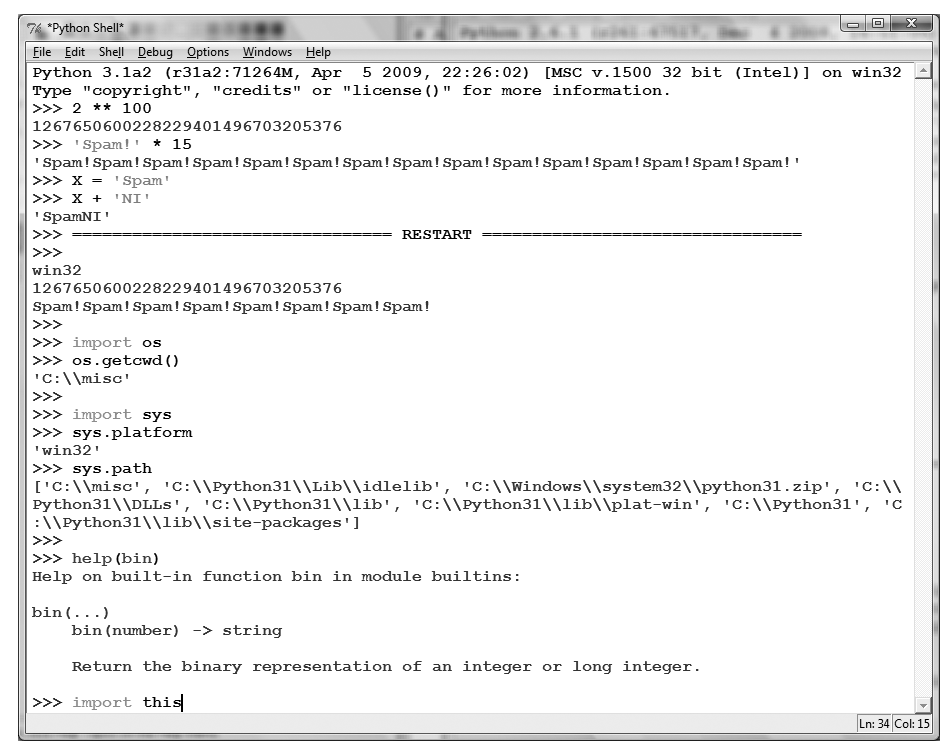
IDLE Basics

Let's jump right into an example. IDLE is easy to start under Windows—it has an entry in the Start button menu for Python (see [Figure 2-1](#), shown previously), and it can also be selected by right-clicking on a Python program icon. On some Unix-like systems,

^{||} IDLE is officially a corruption of IDE, but it's really named in honor of Monty Python member Eric Idle.

you may need to launch IDLE's top-level script from a command line, or by clicking on the icon for the *idle.pyw* or *idle.py* file located in the *idlelib* subdirectory of Python's *Lib* directory. On Windows, IDLE is a Python script that currently lives in *C:\Python30\Lib\idlelib* (or *C:\Python26\Lib\idlelib* in Python 2.6).#

Figure 3-3 shows the scene after starting IDLE on Windows. The Python shell window that opens initially is the main window, which runs an interactive session (notice the `>>>` prompt). This works like all interactive sessions—code you type here is run immediately after you type it—and serves as a testing tool.



```
Python 3.1a2 (r31a2:71264M, Apr 5 2009, 22:26:02) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 2 ** 100
1267650600228229401496703205376
>>> 'Spam!' * 15
'Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam! '
>>> X = 'Spam'
>>> X + 'NI'
'SpamNI'
>>> ===== RESTART =====
>>>
win32
1267650600228229401496703205376
Spam! Spam! Spam! Spam! Spam! Spam! Spam! Spam!
>>>
>>> import os
>>> os.getcwd()
'C:\\misc'
>>>
>>> import sys
>>> sys.platform
'win32'
>>> sys.path
['C:\\misc', 'C:\\Python31\\Lib\\idlelib', 'C:\\Windows\\system32\\python31.zip', 'C:\\Python31\\DLLs', 'C:\\Python31\\lib', 'C:\\Python31\\lib\\plat-win', 'C:\\Python31', 'C:\\Python31\\lib\\site-packages']
>>>
>>> help(bin)
Help on built-in function bin in module builtins:

bin(...)
    bin(number) -> string

    Return the binary representation of an integer or long integer.

>>> import this
```

Figure 3-3. The main Python shell window of the IDLE development GUI, shown here running on Windows. Use the File menu to begin (New Window) or change (Open...) a source file; use the text edit window's Run menu to run the code in that window (Run Module).

#IDLE is a Python program that uses the standard library's tkinter GUI toolkit (a.k.a. Tkinter in Python 2.6) to build the IDLE GUI. This makes IDLE portable, but it also means that you'll need to have tkinter support in your Python to use IDLE. The Windows version of Python has this by default, but some Linux and Unix users may need to install the appropriate tkinter support (a `yum tkinter` command may suffice on some Linux distributions, but see the installation hints in [Appendix A](#) for details). Mac OS X may have everything you need preinstalled, too; look for an `idle` command or script on your machine.

IDLE uses familiar menus with keyboard shortcuts for most of its operations. To make (or edit) a source code file under IDLE, open a text edit window: in the main window, select the File pull-down menu, and pick New Window (or Open... to open a text edit window displaying an existing file for editing).

Although it may not show up fully in this book's graphics, IDLE uses syntax-directed *colorization* for the code typed in both the main window and all text edit windows—keywords are one color, literals are another, and so on. This helps give you a better picture of the components in your code (and can even help you spot mistakes—run-on strings are all one color, for example).

To run a file of code that you are editing in IDLE, select the file's text edit window, open that window's Run pull-down menu, and choose the Run Module option listed there (or use the equivalent keyboard shortcut, given in the menu). Python will let you know that you need to save your file first if you've changed it since it was opened or last saved and forgot to save your changes—a common mistake when you're knee deep in coding.

When run this way, the output of your script and any error messages it may generate show up back in the main interactive window (the Python shell window). In [Figure 3-3](#), for example, the three lines after the “RESTART” line near the middle of the window reflect an execution of our *script1.py* file opened in a separate edit window. The “RESTART” message tells us that the user-code process was restarted to run the edited script and serves to separate script output (it does not appear if IDLE is started without a user-code subprocess—more on this mode in a moment).



IDLE hint of the day: If you want to repeat prior commands in IDLE's main interactive window, you can use the Alt-P key combination to scroll backward through the command history, and Alt-N to scroll forward (on some Macs, try Ctrl-P and Ctrl-N instead). Your prior commands will be recalled and displayed, and may be edited and rerun. You can also recall commands by positioning the cursor on them, or use cut-and-paste operations, but these techniques tend to involve more work. Outside IDLE, you may be able to recall commands in an interactive session with the arrow keys on Windows.

Using IDLE

IDLE is free, easy to use, portable, and automatically available on most platforms. I generally recommend it to Python newcomers because it sugarcoats some of the details and does not assume prior experience with system command lines. However, it is somewhat limited compared to more advanced commercial IDEs. To help you avoid some common pitfalls, here is a list of issues that IDLE beginners should bear in mind:

- **You must add “.py” explicitly when saving your files.** I mentioned this when talking about files in general, but it's a common IDLE stumbling block, especially

for Windows users. IDLE does not automatically add a `.py` extension to filenames when files are saved. Be careful to type the `.py` extension yourself when saving a file for the first time. If you don't, while you will be able to run your file from IDLE (and system command lines), you will not be able to import it either interactively or from other modules.

- **Run scripts by selecting Run→Run Module in text edit windows, not by interactive imports and reloads.** Earlier in this chapter, we saw that it's possible to run a file by importing it interactively. However, this scheme can grow complex because it requires you to manually reload files after changes. By contrast, using the Run→Run Module menu option in IDLE always runs the most current version of your file, just like running it using a system shell command line. IDLE also prompts you to save your file first, if needed (another common mistake outside IDLE).
- **You need to reload only modules being tested interactively.** Like system shell command lines, IDLE's Run→Run Module menu option always runs the current version of both the top-level file and any modules it imports. Because of this, Run→Run Module eliminates common confusions surrounding imports. You only need to reload modules that you are importing and testing interactively in IDLE. If you choose to use the import and reload technique instead of Run→Run Module, remember that you can use the Alt-P/Alt-N key combinations to recall prior commands.
- **You can customize IDLE.** To change the text fonts and colors in IDLE, select the Configure option in the Options menu of any IDLE window. You can also customize key combination actions, indentation settings, and more; see IDLE's Help pull-down menu for more hints.
- **There is currently no clear-screen option in IDLE.** This seems to be a frequent request (perhaps because it's an option available in similar IDEs), and it might be added eventually. Today, though, there is no way to clear the interactive window's text. If you want the window's text to go away, you can either press and hold the Enter key, or type a Python loop to print a series of blank lines (nobody really uses the latter technique, of course, but it sounds more high-tech than pressing the Enter key!).
- **tkinter GUI and threaded programs may not work well with IDLE.** Because IDLE is a Python/tkinter program, it can hang if you use it to run certain types of advanced Python/tkinter programs. This has become less of an issue in more recent versions of IDLE that run user code in one process and the IDLE GUI itself in another, but some programs (especially those that use multithreading) might still hang the GUI. Your code may not exhibit such problems, but as a rule of thumb, it's always safe to use IDLE to edit GUI programs but launch them using other options, such as icon clicks or system command lines. When in doubt, if your code fails in IDLE, try it outside the GUI.

- **If connection errors arise, try starting IDLE in single-process mode.** Because IDLE requires communication between its separate user and GUI processes, it can sometimes have trouble starting up on certain platforms (notably, it fails to start occasionally on some Windows machines, due to firewall software that blocks connections). If you run into such connection errors, it's always possible to start IDLE with a system command line that forces it to run in single-process mode without a user-code subprocess and therefore avoids communication issues: its `-n` command-line flag forces this mode. On Windows, for example, start a Command Prompt window and run the system command line `idle.py -n` from within the directory `C:\Python30\Lib\idlelib` (cd there first if needed).
- **Beware of some IDLE usability features.** IDLE does much to make life easier for beginners, but some of its tricks won't apply outside the IDLE GUI. For instance, IDLE runs your scripts in its own interactive namespace, so variables in your code show up automatically in the IDLE interactive session—you don't always need to run `import` commands to access names at the top level of files you've already run. This can be handy, but it can also be confusing, because outside the IDLE environment names must always be imported from files to be used.

IDLE also automatically changes both to the directory of a file just run and adds its directory to the module import search path—a handy feature that allows you to import files there without search path settings, but also something that won't work the same when you run files outside IDLE. It's OK to use such features, but don't forget that they are IDLE behavior, not Python behavior.

Advanced IDLE Tools

Besides the basic edit and run functions, IDLE provides more advanced features, including a point-and-click program debugger and an object browser. The IDLE debugger is enabled via the Debug menu and the object browser via the File menu. The browser allows you to navigate through the module search path to files and objects in files; clicking on a file or object opens the corresponding source in a text edit window.

IDLE debugging is initiated by selecting the Debug→Debugger menu option in the main window and then starting your script by selecting the Run→Run Module option in the text edit window; once the debugger is enabled, you can set breakpoints in your code that stop its execution by right-clicking on lines in the text edit windows, show variable values, and so on. You can also watch program execution when debugging—the current line of code is noted as you step through your code.

For simpler debugging operations, you can also right-click with your mouse on the text of an error message to quickly jump to the line of code where the error occurred—a trick that makes it simple and fast to repair and run again. In addition, IDLE's text editor offers a large collection of programmer-friendly tools, including automatic indentation, advanced text and file search operations, and more. Because IDLE uses

intuitive GUI interactions, you should experiment with the system live to get a feel for its other tools.

Other IDEs

Because IDLE is free, portable, and a standard part of Python, it's a nice first development tool to become familiar with if you want to use an IDE at all. Again, I recommend that you use IDLE for this book's exercises if you're just starting out, unless you are already familiar with and prefer a command-line-based development mode. There are, however, a handful of alternative IDEs for Python developers, some of which are substantially more powerful and robust than IDLE. Here are some of the most commonly used IDEs:

Eclipse and PyDev

Eclipse is an advanced open source IDE GUI. Originally developed as a Java IDE, Eclipse also supports Python development when you install the PyDev (or a similar) plug-in. Eclipse is a popular and powerful option for Python development, and it goes well beyond IDLE's feature set. It includes support for code completion, syntax highlighting, syntax analysis, refactoring, debugging, and more. Its downsides are that it is a large system to install and may require shareware extensions for some features (this may vary over time). Still, when you are ready to graduate from IDLE, the Eclipse/PyDev combination is worth your attention.

Komodo

A full-featured development environment GUI for Python (and other languages), Komodo includes standard syntax-coloring, text-editing, debugging, and other features. In addition, Komodo offers many advanced features that IDLE does not, including project files, source-control integration, regular-expression debugging, and a drag-and-drop GUI builder that generates Python/tkinter code to implement the GUIs you design interactively. At this writing, Komodo is not free; it is available at <http://www.activestate.com>.

NetBeans IDE for Python

NetBeans is a powerful open-source development environment GUI with support for many advanced features for Python developers: code completion, automatic indentation and code colorization, editor hints, code folding, refactoring, debugging, code coverage and testing, projects, and more. It may be used to develop both CPython and Jython code. Like Eclipse, NetBeans requires installation steps beyond those of the included IDLE GUI, but it is seen by many as more than worth the effort. Search the Web for the latest information and links.

PythonWin

PythonWin is a free Windows-only IDE for Python that ships as part of ActiveState's ActivePython distribution (and may also be fetched separately from <http://www.python.org> resources). It is roughly like IDLE, with a handful of useful Windows-specific extensions added; for example, PythonWin has support for

COM objects. Today, IDLE is probably more advanced than PythonWin (for instance, IDLE’s dual-process architecture often prevents it from hanging). However, PythonWin still offers tools for Windows developers that IDLE does not. See <http://www.activestate.com> for more information.

Others

There are roughly half a dozen other widely used IDEs that I’m aware of (including the commercial *Wing IDE* and *PythonCard*) but do not have space to do justice to here, and more will probably appear over time. In fact, almost every programmer-friendly text editor has some sort of support for Python development these days, whether it be preinstalled or fetched separately. Emacs and Vim, for instance, have substantial Python support.

I won’t try to document all such options here; for more information, see the resources available at <http://www.python.org> or search the Web for “Python IDE.” You might also try running a web search for “Python editors”—today, this leads you to a wiki page that maintains information about many IDE and text-editor options for Python programming.

Other Launch Options

At this point, we’ve seen how to run code typed interactively, and how to launch code saved in files in a variety of ways—system command lines, imports and execs, GUIs like IDLE, and more. That covers most of the cases you’ll see in this book. There are additional ways to run Python code, though, most of which have special or narrow roles. The next few sections take a quick look at some of these.

Embedding Calls

In some specialized domains, Python code may be run automatically by an enclosing system. In such cases, we say that the Python programs are *embedded* in (i.e., run by) another program. The Python code itself may be entered into a text file, stored in a database, fetched from an HTML page, parsed from an XML document, and so on. But from an operational perspective, another system—not you—may tell Python to run the code you’ve created.

Such an embedded execution mode is commonly used to support end-user customization—a game program, for instance, might allow for play modifications by running user-accessible embedded Python code at strategic points in time. Users can modify this type of system by providing or changing Python code. Because Python code is interpreted, there is no need to recompile the entire system to incorporate the change (see [Chapter 2](#) for more on how Python code is run).

In this mode, the enclosing system that runs your code might be written in C, C++, or even Java when the Jython system is used. As an example, it's possible to create and run strings of Python code from a C program by calling functions in the Python runtime API (a set of services exported by the libraries created when Python is compiled on your machine):

```
#include <Python.h>
...
Py_Initialize();
PyRun_SimpleString("x = 'brave ' + 'sir robin'");
```

// This is C, not Python
// But it runs Python code

In this C code snippet, a program coded in the C language embeds the Python interpreter by linking in its libraries, and passes it a Python assignment statement string to run. C programs may also gain access to Python modules and objects and process or execute them using other Python API tools.

This book isn't about Python/C integration, but you should be aware that, depending on how your organization plans to use Python, you may or may not be the one who actually starts the Python programs you create. Regardless, you can usually still use the interactive and file-based launching techniques described here to test code in isolation from those enclosing systems that may eventually use it.*

Frozen Binary Executables

Frozen binary executables, described in [Chapter 2](#), are packages that combine your program's byte code and the Python interpreter into a single executable program. This approach enables Python programs to be launched in the same ways that you would launch any other executable program (icon clicks, command lines, etc.). While this option works well for delivery of products, it is not really intended for use during program development; you normally freeze just before shipping (after development is finished). See the prior chapter for more on this option.

Text Editor Launch Options

As mentioned previously, although they're not full-blown IDE GUIs, most programmer-friendly text editors have support for editing, and possibly running, Python programs. Such support may be built in or fetchable on the Web. For instance, if you are familiar with the Emacs text editor, you can do all your Python editing and launching from inside that text editor. See the text editor resources page at <http://www.python.org/editors> for more details, or search the Web for the phrase "Python editors."

* See [Programming Python](#) (O'Reilly) for more details on embedding Python in C/C++. The embedding API can call Python functions directly, load modules, and more. Also, note that the Jython system allows Java programs to invoke Python code using a Java-based API (a Python interpreter class).

Still Other Launch Options

Depending on your platform, there may be additional ways that you can start Python programs. For instance, on some Macintosh systems you may be able to drag Python program file icons onto the Python interpreter icon to make them execute, and on Windows you can always start Python scripts with the Run... option in the Start menu. Additionally, the Python standard library has utilities that allow Python programs to be started by other Python programs in separate processes (e.g., `os.popen`, `os.system`), and Python scripts might also be spawned in larger contexts like the Web (for instance, a web page might invoke a script on a server); however, these are beyond the scope of the present chapter.

Future Possibilities?

This chapter reflects current practice, but much of the material is both platform- and time-specific. Indeed, many of the execution and launch details presented arose during the shelf life of this book's various editions. As with program execution options, it's not impossible that new program launch options may arise over time.

New operating systems, and new versions of existing systems, may also provide execution techniques beyond those outlined here. In general, because Python keeps pace with such changes, you should be able to launch Python programs in whatever way makes sense for the machines you use, both now and in the future—be that by drawing on tablet PCs or PDAs, grabbing icons in a virtual reality, or shouting a script's name over your coworkers' conversations.

Implementation changes may also impact launch schemes somewhat (e.g., a full compiler could produce normal executables that are launched much like frozen binaries today). If I knew what the future truly held, though, I would probably be talking to a stockbroker instead of writing these words!

Which Option Should I Use?

With all these options, one question naturally arises: which one is best for me? In general, you should give the IDLE interface a try if you are just getting started with Python. It provides a user-friendly GUI environment and hides some of the underlying configuration details. It also comes with a platform-neutral text editor for coding your scripts, and it's a standard and free part of the Python system.

If, on the other hand, you are an experienced programmer, you might be more comfortable with simply the text editor of your choice in one window, and another window for launching the programs you edit via system command lines and icon clicks (in fact, this is how I develop Python programs, but I have a Unix-biased past). Because the choice of development environments is very subjective, I can't offer much more in the

way of universal guidelines; in general, whatever environment you like to use will be the best for you to use.

Debugging Python Code

Naturally, none of my readers or students ever have bugs in their code (*insert smiley here*), but for less fortunate friends of yours who may, here's a quick look at the strategies commonly used by real-world Python programmers to debug code:

- **Do nothing.** By this, I don't mean that Python programmers don't debug their code—but when you make a mistake in a Python program, you get a very useful and readable error message (you'll get to see some soon, if you haven't already). If you already know Python, and especially for your own code, this is often enough—read the error message, and go fix the tagged line and file. For many, this is debugging in Python. It may not always be ideal for larger system you didn't write, though.
- **Insert print statements.** Probably the main way that Python programmers debug their code (and the way that I debug Python code) is to insert `print` statements and run again. Because Python runs immediately after changes, this is usually the quickest way to get more information than error messages provide. The `print` statements don't have to be sophisticated—a simple “I am here” or display of variable values is usually enough to provide the context you need. Just remember to delete or comment out (i.e., add a `#` before) the debugging `prints` before you ship your code!
- **Use IDE GUI debuggers.** For larger systems you didn't write, and for beginners who want to trace code in more detail, most Python development GUIs have some sort of point-and-click debugging support. IDLE has a debugger too, but it doesn't appear to be used very often in practice—perhaps because it has no command line, or perhaps because adding `print` statements is usually quicker than setting up a GUI debugging session. To learn more, see IDLE's Help, or simply try it on your own; its basic interface is described in the section “[Advanced IDLE Tools](#)” on page 62. Other IDEs, such as Eclipse, NetBeans, Komodo, and Wing IDE, offer advanced point-and-click debuggers as well; see their documentation if you use them.
- **Use the pdb command-line debugger.** For ultimate control, Python comes with a source-code debugger named `pdb`, available as a module in Python's standard library. In `pdb`, you type commands to step line by line, display variables, set and clear breakpoints, continue to a breakpoint or error, and so on. `pdb` can be launched interactively by importing it, or as a top-level script. Either way, because you can type commands to control the session, it provides a powerful debugging tool. `pdb` also includes a postmortem function you can run after an exception occurs, to get information from the time of the error. See the Python library manual and [Chapter 35](#) for more details on `pdb`.
- **Other options.** For more specific debugging requirements, you can find additional tools in the open source domain, including support for multithreaded programs, embedded code, and process attachment. The `Winpdb` system, for example, is a

standalone debugger with advanced debugging support and cross-platform GUI and console interfaces.

These options will become more important as we start writing larger scripts. Probably the best news on the debugging front, though, is that errors are detected and reported in Python, rather than passing silently or crashing the system altogether. In fact, errors themselves are a well-defined mechanism known as *exceptions*, which you can catch and process (more on exceptions in [Part VII](#)). Making mistakes is never fun, of course, but speaking as someone who recalls when debugging meant getting out a hex calculator and poring over piles of memory dump print-outs, Python's debugging support makes errors much less painful than they might otherwise be.

Chapter Summary

In this chapter, we've looked at common ways to launch Python programs: by running code typed interactively, and by running code stored in files with system command lines, file-icon clicks, module imports, `exec` calls, and IDE GUIs such as IDLE. We've covered a lot of pragmatic startup territory here. This chapter's goal was to equip you with enough information to enable you to start writing some code, which you'll do in the next part of the book. There, we will start exploring the Python language itself, beginning with its core data types.

First, though, take the usual chapter quiz to exercise what you've learned here. Because this is the last chapter in this part of the book, it's followed with a set of more complete exercises that test your mastery of this entire part's topics. For help with the latter set of problems, or just for a refresher, be sure to turn to [Appendix B](#) after you've given the exercises a try.

Test Your Knowledge: Quiz

1. How can you start an interactive interpreter session?
2. Where do you type a system command line to launch a script file?
3. Name four or more ways to run the code saved in a script file.
4. Name two pitfalls related to clicking file icons on Windows.
5. Why might you need to reload a module?
6. How do you run a script from within IDLE?
7. Name two pitfalls related to using IDLE.
8. What is a namespace, and how does it relate to module files?

Test Your Knowledge: Answers

1. You can start an interactive session on Windows by clicking your Start button, picking the All Programs option, clicking the Python entry, and selecting the “Python (command line)” menu option. You can also achieve the same effect on Windows and other platforms by typing **python** as a system command line in your system’s console window (a Command Prompt window on Windows). Another alternative is to launch IDLE, as its main Python shell window is an interactive session. If you have not set your system’s **PATH** variable to find Python, you may need to **cd** to where Python is installed, or type its full directory path instead of just **python** (e.g., **C:\Python30\python** on Windows).
2. You type system command lines in whatever your platform provides as a system console: a Command Prompt window on Windows; an xterm or terminal window on Unix, Linux, and Mac OS X; and so on.
3. Code in a script (really, module) file can be run with system command lines, file icon clicks, imports and reloads, the **exec** built-in function, and IDE GUI selections such as IDLE’s Run→Run Module menu option. On Unix, they can also be run as executables with the **#!** trick, and some platforms support more specialized launching techniques (e.g., drag-and-drop). In addition, some text editors have unique ways to run Python code, some Python programs are provided as standalone “frozen binary” executables, and some systems use Python code in embedded mode, where it is run automatically by an enclosing program written in a language like C, C++, or Java. The latter technique is usually done to provide a user customization layer.
4. Scripts that print and then exit cause the output file to disappear immediately, before you can view the output (which is why the **input** trick comes in handy); error messages generated by your script also appear in an output window that closes before you can examine its contents (which is one reason that system command lines and IDEs such as IDLE are better for most development).
5. Python only imports (loads) a module once per process, by default, so if you’ve changed its source code and want to run the new version without stopping and restarting Python, you’ll have to reload it. You must import a module at least once before you can reload it. Running files of code from a system shell command line, via an icon click, or via an IDE such as IDLE generally makes this a nonissue, as those launch schemes usually run the current version of the source code file each time.
6. Within the text edit window of the file you wish to run, select the window’s Run→Run Module menu option. This runs the window’s source code as a top-level script file and displays its output back in the interactive Python shell window.
7. IDLE can still be hung by some types of programs—especially GUI programs that perform multithreading (an advanced technique beyond this book’s scope). Also, IDLE has some usability features that can burn you once you leave the IDLE GUI:

a script's variables are automatically imported to the interactive scope in IDLE, for instance, but not by Python in general.

8. A namespace is just a package of variables (i.e., names). It takes the form of an object with attributes in Python. Each module file is automatically a namespace—that is, a package of variables reflecting the assignments made at the top level of the file. Namespaces help avoid name collisions in Python programs: because each module file is a self-contained namespace, files must explicitly import other files in order to use their names.

Test Your Knowledge: Part I Exercises

It's time to start doing a little coding on your own. This first exercise session is fairly simple, but a few of these questions hint at topics to come in later chapters. Be sure to check [“Part I, Getting Started” on page 1101](#) in the solutions appendix ([Appendix B](#)) for the answers; the exercises and their solutions sometimes contain supplemental information not discussed in the main text, so you should take a peek at the solutions even if you manage to answer all the questions on your own.

1. *Interaction.* Using a system command line, IDLE, or another method, start the Python interactive command line (`>>>` prompt), and type the expression `"Hello World!"` (including the quotes). The string should be echoed back to you. The purpose of this exercise is to get your environment configured to run Python. In some scenarios, you may need to first run a `cd` shell command, type the full path to the Python executable, or add its path to your `PATH` environment variable. If desired, you can set `PATH` in your `.cshrc` or `.kshrc` file to make Python permanently available on Unix systems; on Windows, use a `setup.bat`, `autoexec.bat`, or the environment variable GUI. See [Appendix A](#) for help with environment variable settings.
2. *Programs.* With the text editor of your choice, write a simple module file containing the single statement `print('Hello module world!')` and store it as `module1.py`. Now, run this file by using any launch option you like: running it in IDLE, clicking on its file icon, passing it to the Python interpreter on the system shell's command line (e.g., `python module1.py`), built-in `exec` calls, imports and reloads, and so on. In fact, experiment by running your file with as many of the launch techniques discussed in this chapter as you can. Which technique seems easiest? (There is no right answer to this, of course.)
3. *Modules.* Start the Python interactive command line (`>>>` prompt) and import the module you wrote in exercise 2. Try moving the file to a different directory and importing it again from its original directory (i.e., run Python in the original directory when you import). What happens? (Hint: is there still a `module1.pyc` byte code file in the original directory?)

4. *Scripts.* If your platform supports it, add the `#!` line to the top of your `module1.py` module file, give the file executable privileges, and run it directly as an executable. What does the first line need to contain? `#!` usually only has meaning on Unix, Linux, and Unix-like platforms such as Mac OS X; if you're working on Windows, instead try running your file by listing just its name in a DOS console window without the word "python" before it (this works on recent versions of Windows), or via the Start→Run... dialog box.
5. *Errors and debugging.* Experiment with typing mathematical expressions and assignments at the Python interactive command line. Along the way, type the expressions `2 ** 500` and `1 / 0`, and reference an undefined variable name as we did in this chapter. What happens?

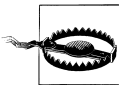
You may not know it yet, but when you make a mistake, you're doing exception processing (a topic we'll explore in depth in [Part VII](#)). As you'll learn there, you are technically triggering what's known as the *default exception handler*—logic that prints a standard error message. If you do not catch an error, the default handler does and prints the standard error message in response.

Exceptions are also bound up with the notion of *debugging* in Python. When you're first starting out, Python's default error messages on exceptions will probably provide as much error-handling support as you need—they give the cause of the error, as well as showing the lines in your code that were active when the error occurred. For more about debugging, see the sidebar "[Debugging Python Code](#)" on page 67.

6. *Breaks and cycles.* At the Python command line, type:

```
L = [1, 2]           # Make a 2-item list
L.append(L)          # Append L as a single item to itself
L                    # Print L
```

What happens? In all recent versions of Python, you'll see a strange output that we'll describe in the solutions appendix, and which will make more sense when we study references in the next part of the book. If you're using a Python version older than 1.5.1, a Ctrl-C key combination will probably help on most platforms. Why do you think your version of Python responds the way it does for this code?



If you do have a Python older than Release 1.5.1 (a hopefully rare scenario today!), make sure your machine can stop a program with a Ctrl-C key combination of some sort before running this test, or you may be waiting a long time.

7. *Documentation.* Spend at least 17 minutes browsing the Python library and language manuals before moving on to get a feel for the available tools in the standard library and the structure of the documentation set. It takes at least this long to become familiar with the locations of major topics in the manual set; once you've done this, it's easy to find what you need. You can find this manual via the Python

Start button entry on Windows, in the Python Docs option on the Help pull-down menu in IDLE, or online at <http://www.python.org/doc>. I'll also have a few more words to say about the manuals and other documentation sources available (including PyDoc and the `help` function) in [Chapter 15](#). If you still have time, go explore the Python website, as well as its PyPy third-party extension repository. Especially check out the [Python.org](#) documentation and search pages; they can be crucial resources.