

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](#).

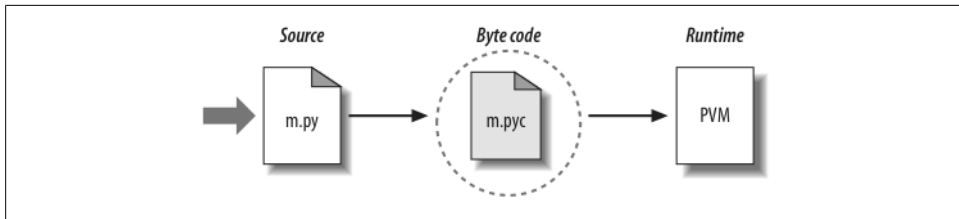


Figure 2-2. Python’s traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.

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!
```

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*.

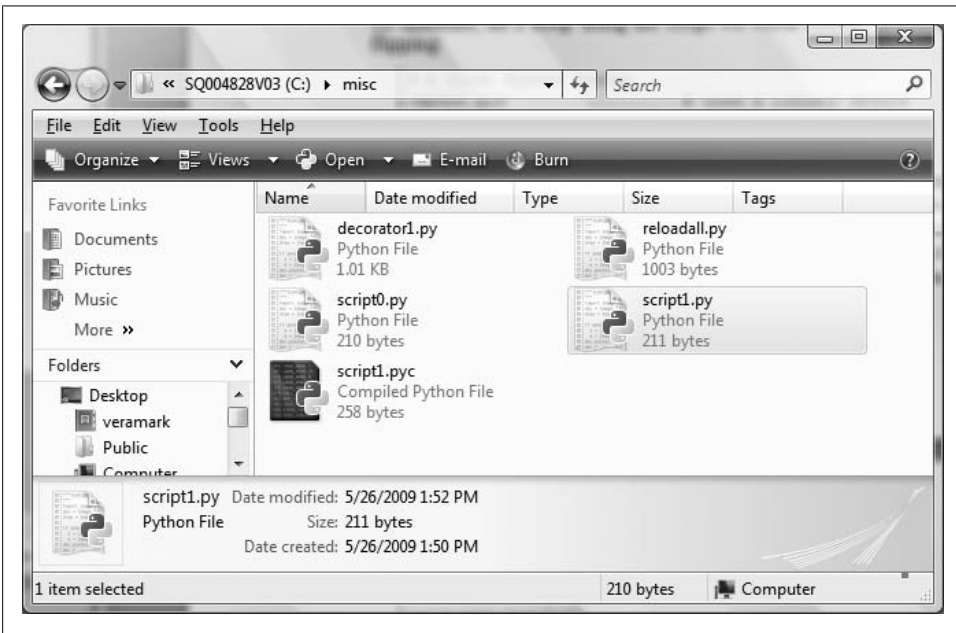


Figure 3-1. On Windows, Python program files show up as icons in file explorer windows and can automatically be run with a double-click of the mouse (though you might not see printed output or error messages this way).

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.

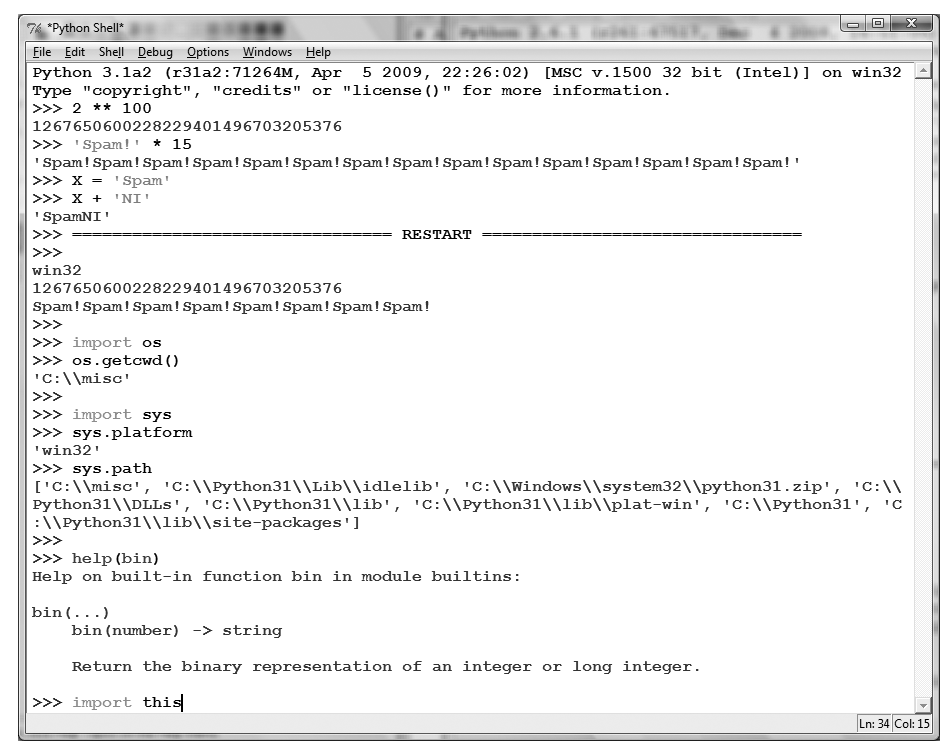
IDLE Basics

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?

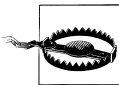
You may not know it yet, but when you make a mistake, you're doing exception processing (a topic we'll explore in depth in [Part VII](#)). As you'll learn there, you are technically triggering what's known as the *default exception handler*—logic that prints a standard error message. If you do not catch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

Types and Operations

Introducing Python Object Types

This chapter begins our tour of the Python language. In an informal sense, in Python, we do things with stuff. “Things” take the form of operations like addition and concatenation, and “stuff” refers to the objects on which we perform those operations. In this part of the book, our focus is on that stuff, and the things our programs can do with it.

Somewhat more formally, in Python, data takes the form of *objects*—either built-in objects that Python provides, or objects we create using Python or external language tools such as C extension libraries. Although we’ll firm up this definition later, objects are essentially just pieces of memory, with values and sets of associated operations.

Because objects are the most fundamental notion in Python programming, we’ll start this chapter with a survey of Python’s built-in object types.

By way of introduction, however, let’s first establish a clear picture of how this chapter fits into the overall Python picture. From a more concrete perspective, Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

1. Programs are composed of modules.
2. Modules contain statements.
3. Statements contain expressions.
4. *Expressions create and process objects.*

The discussion of modules in [Chapter 3](#) introduced the highest level of this hierarchy. This part’s chapters begin at the bottom, exploring both built-in objects and the expressions you can code to use them.

Why Use Built-in Types?

If you've used lower-level languages such as C or C++, you know that much of your work centers on implementing *objects*—also known as *data structures*—to represent the components in your application's domain. You need to lay out memory structures, manage memory allocation, implement search and access routines, and so on. These chores are about as tedious (and error-prone) as they sound, and they usually distract from your program's real goals.

In typical Python programs, most of this grunt work goes away. Because Python provides powerful object types as an intrinsic part of the language, there's usually no need to code object implementations before you start solving problems. In fact, unless you have a need for special processing that built-in types don't provide, you're almost always better off using a built-in object instead of implementing your own. Here are some reasons why:

- **Built-in objects make programs easy to write.** For simple tasks, built-in types are often all you need to represent the structure of problem domains. Because you get powerful tools such as collections (lists) and search tables (dictionaries) for free, you can use them immediately. You can get a lot of work done with Python's built-in object types alone.
- **Built-in objects are components of extensions.** For more complex tasks, you may need to provide your own objects using Python classes or C language interfaces. But as you'll see in later parts of this book, objects implemented manually are often built on top of built-in types such as lists and dictionaries. For instance, a stack data structure may be implemented as a class that manages or customizes a built-in list.
- **Built-in objects are often more efficient than custom data structures.** Python's built-in types employ already optimized data structure algorithms that are implemented in C for speed. Although you can write similar object types on your own, you'll usually be hard-pressed to get the level of performance built-in object types provide.
- **Built-in objects are a standard part of the language.** In some ways, Python borrows both from languages that rely on built-in tools (e.g., LISP) and languages that rely on the programmer to provide tool implementations or frameworks of their own (e.g., C++). Although you can implement unique object types in Python, you don't need to do so just to get started. Moreover, because Python's built-ins are standard, they're always the same; proprietary frameworks, on the other hand, tend to differ from site to site.

In other words, not only do built-in object types make programming easier, but they're also more powerful and efficient than most of what can be created from scratch. Regardless of whether you implement new object types, built-in objects form the core of every Python program.

Python’s Core Data Types

Table 4-1 previews Python’s built-in object types and some of the syntax used to code their *literals*—that is, the expressions that generate these objects.* Some of these types will probably seem familiar if you’ve used other languages; for instance, numbers and strings represent numeric and textual values, respectively, and files provide an interface for processing files stored on your computer.

Table 4-1. Built-in objects preview

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, Decimal, Fraction
Strings	'spam', "guido's", b'a\x01c'
Lists	[1, [2, 'three'], 4]
Dictionaries	{'food': 'spam', 'taste': 'yum'}
Tuples	(1, 'spam', 4, 'U')
Files	myfile = open('eggs', 'r')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes (Part IV, Part V, Part VI)
Implementation-related types	Compiled code, stack tracebacks (Part IV, Part VII)

Table 4-1 isn’t really complete, because *everything* we process in Python programs is a kind of object. For instance, when we perform text pattern matching in Python, we create pattern objects, and when we perform network scripting, we use socket objects. These other kinds of objects are generally created by importing and using modules and have behavior all their own.

As we’ll see in later parts of the book, *program units* such as functions, modules, and classes are objects in Python too—they are created with statements and expressions such as `def`, `class`, `import`, and `lambda` and may be passed around scripts freely, stored within other objects, and so on. Python also provides a set of *implementation-related* types such as compiled code objects, which are generally of interest to tool builders more than application developers; these are also discussed in later parts of this text.

We usually call the other object types in Table 4-1 *core* data types, though, because they are effectively built into the Python language—that is, there is specific expression syntax for generating most of them. For instance, when you run the following code:

```
>>> 'spam'
```

* In this book, the term *literal* simply means an expression whose syntax generates an object—sometimes also called a *constant*. Note that the term “constant” does not imply objects or variables that can never be changed (i.e., this term is unrelated to C++’s `const` or Python’s “immutable”—a topic explored in the section “Immutability” on page 82).

you are, technically speaking, running a literal expression that generates and returns a new string object. There is specific Python language syntax to make this object. Similarly, an expression wrapped in square brackets makes a list, one in curly braces makes a dictionary, and so on. Even though, as we'll see, there are no type declarations in Python, the syntax of the expressions you run determines the types of objects you create and use. In fact, object-generation expressions like those in [Table 4-1](#) are generally where types originate in the Python language.

Just as importantly, once you create an object, you bind its operation set for all time—you can perform only string operations on a string and list operations on a list. As you'll learn, Python is *dynamically typed* (it keeps track of types for you automatically instead of requiring declaration code), but it is also *strongly typed* (you can perform on an object only operations that are valid for its type).

Functionally, the object types in [Table 4-1](#) are more general and powerful than what you may be accustomed to. For instance, you'll find that lists and dictionaries alone are powerful data representation tools that obviate most of the work you do to support collections and searching in lower-level languages. In short, lists provide ordered collections of other objects, while dictionaries store objects by key; both lists and dictionaries may be nested, can grow and shrink on demand, and may contain objects of any type.

We'll study each of the object types in [Table 4-1](#) in detail in upcoming chapters. Before digging into the details, though, let's begin by taking a quick look at Python's core objects in action. The rest of this chapter provides a preview of the operations we'll explore in more depth in the chapters that follow. Don't expect to find the full story here—the goal of this chapter is just to whet your appetite and introduce some key ideas. Still, the best way to get started is to get started, so let's jump right into some real code.

Numbers

If you've done any programming or scripting in the past, some of the object types in [Table 4-1](#) will probably seem familiar. Even if you haven't, numbers are fairly straightforward. Python's core objects set includes the usual suspects: integers (numbers without a fractional part), floating-point numbers (roughly, numbers with a decimal point in them), and more exotic numeric types (complex numbers with imaginary parts, fixed-precision decimals, rational fractions with numerator and denominator, and full-featured sets).

Although it offers some fancier options, Python's basic number types are, well, basic. Numbers in Python support the normal mathematical operations. For instance, the plus sign (+) performs addition, a star (*) is used for multiplication, and two stars (**) are used for exponentiation:

```

>>> 123 + 222                                # Integer addition
345
>>> 1.5 * 4                                    # Floating-point multiplication
6.0
>>> 2 ** 100                                  # 2 to the power 100
1267650600228229401496703205376

```

Notice the last result here: Python 3.0's integer type automatically provides extra precision for large numbers like this when needed (in 2.6, a separate long integer type handles numbers too large for the normal integer type in similar ways). You can, for instance, compute 2 to the power 1,000,000 as an integer in Python, but you probably shouldn't try to print the result—with more than 300,000 digits, you may be waiting awhile!

```

>>> len(str(2 ** 1000000))                    # How many digits in a really BIG number?
301030

```

Once you start experimenting with floating-point numbers, you're likely to stumble across something that may look a bit odd on first glance:

```

>>> 3.1415 * 2                                # repr: as code
6.2830000000000004
>>> print(3.1415 * 2)                         # str: user-friendly
6.283

```

The first result isn't a bug; it's a display issue. It turns out that there are two ways to print every object: with full precision (as in the first result shown here), and in a user-friendly form (as in the second). Formally, the first form is known as an object's as-code `repr`, and the second is its user-friendly `str`. The difference can matter when we step up to using classes; for now, if something looks odd, try showing it with a `print` built-in call statement.

Besides expressions, there are a handful of useful numeric modules that ship with Python—*modules* are just packages of additional tools that we import to use:

```

>>> import math
>>> math.pi
3.1415926535897931
>>> math.sqrt(85)
9.2195444572928871

```

The `math` module contains more advanced numeric tools as functions, while the `random` module performs random number generation and random selections (here, from a Python list, introduced later in this chapter):

```

>>> import random
>>> random.random()
0.59268735266273953
>>> random.choice([1, 2, 3, 4])
1

```

Python also includes more exotic numeric objects—such as complex, fixed-precision, and rational numbers, as well as sets and Booleans—and the third-party open source

extension domain has even more (e.g., matrixes and vectors). We'll defer discussion of these types until later in the book.

So far, we've been using Python much like a simple calculator; to do better justice to its built-in types, let's move on to explore strings.

Strings

Strings are used to record textual information as well as arbitrary collections of bytes. They are our first example of what we call a *sequence* in Python—that is, a positionally ordered collection of other objects. Sequences maintain a left-to-right order among the items they contain: their items are stored and fetched by their relative position. Strictly speaking, strings are sequences of one-character strings; other types of sequences include lists and tuples, covered later.

Sequence Operations

As sequences, strings support operations that assume a positional ordering among items. For example, if we have a four-character string, we can verify its length with the built-in `len` function and fetch its components with *indexing* expressions:

```
>>> S = 'Spam'
>>> len(S)           # Length
4
>>> S[0]             # The first item in S, indexing by zero-based position
'S'
>>> S[1]             # The second item from the left
'p'
```

In Python, indexes are coded as offsets from the front, and so start from 0: the first item is at index 0, the second is at index 1, and so on.

Notice how we assign the string to a *variable* named `S` here. We'll go into detail on how this works later (especially in [Chapter 6](#)), but Python variables never need to be declared ahead of time. A variable is created when you assign it a value, may be assigned any type of object, and is replaced with its value when it shows up in an expression. It must also have been previously assigned by the time you use its value. For the purposes of this chapter, it's enough to know that we need to assign an object to a variable in order to save it for later use.

In Python, we can also index backward, from the end—positive indexes count from the left, and negative indexes count back from the right:

```
>>> S[-1]            # The last item from the end in S
'm'
>>> S[-2]            # The second to last item from the end
'a'
```

Formally, a negative index is simply added to the string's size, so the following two operations are equivalent (though the first is easier to code and less easy to get wrong):

```
>>> S[-1]                # The last item in S
'm'
>>> S[len(S)-1]          # Negative indexing, the hard way
'm'
```

Notice that we can use an arbitrary expression in the square brackets, not just a hard-coded number literal—anywhere that Python expects a value, we can use a literal, a variable, or any expression. Python's syntax is completely general this way.

In addition to simple positional indexing, sequences also support a more general form of indexing known as *slicing*, which is a way to extract an entire section (slice) in a single step. For example:

```
>>> S                    # A 4-character string
'Spam'
>>> S[1:3]               # Slice of S from offsets 1 through 2 (not 3)
'pa'
```

Probably the easiest way to think of slices is that they are a way to extract an entire *column* from a string in a single step. Their general form, `X[I:J]`, means “give me everything in `X` from offset `I` up to but not including offset `J`.” The result is returned in a new object. The second of the preceding operations, for instance, gives us all the characters in string `S` from offsets 1 through 2 (that is, $3 - 1$) as a new string. The effect is to slice or “parse out” the two characters in the middle.

In a slice, the left bound defaults to zero, and the right bound defaults to the length of the sequence being sliced. This leads to some common usage variations:

```
>>> S[1:]                # Everything past the first (1:len(S))
'pam'
>>> S                    # S itself hasn't changed
'Spam'
>>> S[0:3]               # Everything but the last
'Spa'
>>> S[:3]                # Same as S[0:3]
'Spa'
>>> S[:-1]               # Everything but the last again, but simpler (0:-1)
'Spa'
>>> S[:]                 # All of S as a top-level copy (0:len(S))
'Spam'
```

Note how negative offsets can be used to give bounds for slices, too, and how the last operation effectively copies the entire string. As you'll learn later, there is no reason to copy a string, but this form can be useful for sequences like lists.

Finally, as sequences, strings also support *concatenation* with the plus sign (joining two strings into a new string) and *repetition* (making a new string by repeating another):

```
>>> S
'Spam'
>>> S + 'xyz'            # Concatenation
```

```
'Spamxyz'
>>> S                               # S is unchanged
'Spam'
>>> S * 8                           # Repetition
'SpamSpamSpamSpamSpamSpamSpamSpam'
```

Notice that the plus sign (+) means different things for different objects: addition for numbers, and concatenation for strings. This is a general property of Python that we'll call *polymorphism* later in the book—in sum, the meaning of an operation depends on the objects being operated on. As you'll see when we study dynamic typing, this polymorphism property accounts for much of the conciseness and flexibility of Python code. Because types aren't constrained, a Python-coded operation can normally work on many different types of objects automatically, as long as they support a compatible interface (like the + operation here). This turns out to be a huge idea in Python; you'll learn more about it later on our tour.

Immutability

Notice that in the prior examples, we were not changing the original string with any of the operations we ran on it. Every string operation is defined to produce a new string as its result, because strings are *immutable* in Python—they cannot be changed in-place after they are created. For example, you can't change a string by assigning to one of its positions, but you can always build a new one and assign it to the same name. Because Python cleans up old objects as you go (as you'll see later), this isn't as inefficient as it may sound:

```
>>> S
'Spam'
>>> S[0] = 'z'                       # Immutable objects cannot be changed
...error text omitted...
TypeError: 'str' object does not support item assignment

>>> S = 'z' + S[1:]                 # But we can run expressions to make new objects
>>> S
'zspam'
```

Every object in Python is classified as either immutable (unchangeable) or not. In terms of the core types, numbers, strings, and tuples are immutable; lists and dictionaries are not (they can be changed in-place freely). Among other things, immutability can be used to guarantee that an object remains constant throughout your program.

Type-Specific Methods

Every string operation we've studied so far is really a sequence operation—that is, these operations will work on other sequences in Python as well, including lists and tuples. In addition to generic sequence operations, though, strings also have operations all their own, available as *methods*—functions attached to the object, which are triggered with a call expression.

For example, the string `find` method is the basic substring search operation (it returns the offset of the passed-in substring, or `-1` if it is not present), and the string `replace` method performs global searches and replacements:

```
>>> S.find('pa')           # Find the offset of a substring
1
>>> S
'Spam'
>>> S.replace('pa', 'XYZ') # Replace occurrences of a substring with another
'SXYZm'
>>> S
'Spam'
```

Again, despite the names of these string methods, we are not changing the original strings here, but creating new strings as the results—because strings are immutable, we have to do it this way. String methods are the first line of text-processing tools in Python. Other methods split a string into substrings on a delimiter (handy as a simple form of parsing), perform case conversions, test the content of the string (digits, letters, and so on), and strip whitespace characters off the ends of the string:

```
>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',')         # Split on a delimiter into a list of substrings
['aaa', 'bbb', 'ccccc', 'dd']
>>> S = 'spam'
>>> S.upper()              # Upper- and lowercase conversions
'SPAM'

>>> S.isalpha()            # Content tests: isalpha, isdigit, etc.
True

>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line = line.rstrip()    # Remove whitespace characters on the right side
>>> line
'aaa,bbb,ccccc,dd'
```

Strings also support an advanced substitution operation known as *formatting*, available as both an expression (the original) and a string method call (new in 2.6 and 3.0):

```
>>> '%s, eggs, and %s' % ('spam', 'SPAM!') # Formatting expression (all)
'spam, eggs, and SPAM!'

>>> '{0}, eggs, and {1}'.format('spam', 'SPAM!') # Formatting method (2.6, 3.0)
'spam, eggs, and SPAM!'
```

One note here: although sequence operations are generic, methods are not—although some types share some method names, string method operations generally work only on strings, and nothing else. As a rule of thumb, Python’s toolset is layered: generic operations that span multiple types show up as built-in functions or expressions (e.g., `len(X)`, `X[0]`), but type-specific operations are method calls (e.g., `aString.upper()`). Finding the tools you need among all these categories will become more natural as you use Python more, but the next section gives a few tips you can use right now.

Getting Help

The methods introduced in the prior section are a representative, but small, sample of what is available for string objects. In general, this book is not exhaustive in its look at object methods. For more details, you can always call the built-in `dir` function, which returns a list of all the attributes available for a given object. Because methods are function attributes, they will show up in this list. Assuming `S` is still the string, here are its attributes on Python 3.0 (Python 2.6 varies slightly):

```
>>> dir(S)
['_add_', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'_format_', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'_gt_', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'_mod_', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'_repr_', '__rmod_', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'_subclasshook_', '_formatter_field_name_split', '_formatter_parser',
'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier',
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

You probably won't care about the names with underscores in this list until later in the book, when we study operator overloading in classes—they represent the implementation of the string object and are available to support customization. In general, leading and trailing double underscores is the naming pattern Python uses for implementation details. The names without the underscores in this list are the callable methods on string objects.

The `dir` function simply gives the methods' names. To ask what they do, you can pass them to the `help` function:

```
>>> help(S.replace)
Help on built-in function replace:

replace(...)
    S.replace(old, new[, count]) -> str

    Return a copy of S with all occurrences of substring
    old replaced by new. If the optional argument count is
    given, only the first count occurrences are replaced.
```

`help` is one of a handful of interfaces to a system of code that ships with Python known as *PyDoc*—a tool for extracting documentation from objects. Later in the book, you'll see that *PyDoc* can also render its reports in HTML format.

You can also ask for help on an entire string (e.g., `help(S)`), but you may get more help than you want to see—i.e., information about every string method. It's generally better to ask about a specific method.

For more details, you can also consult Python’s standard library reference manual or commercially published reference books, but `dir` and `help` are the first line of documentation in Python.

Other Ways to Code Strings

So far, we’ve looked at the string object’s sequence operations and type-specific methods. Python also provides a variety of ways for us to code strings, which we’ll explore in greater depth later. For instance, special characters can be represented as backslash escape sequences:

```
>>> S = 'A\nB\tC'          # \n is end-of-line, \t is tab
>>> len(S)                  # Each stands for just one character
5

>>> ord('\n')               # \n is a byte with the binary value 10 in ASCII
10

>>> S = 'A\0B\0C'          # \0, a binary zero byte, does not terminate string
>>> len(S)
5
```

Python allows strings to be enclosed in single or double quote characters (they mean the same thing). It also allows multiline string literals enclosed in triple quotes (single or double)—when this form is used, all the lines are concatenated together, and end-of-line characters are added where line breaks appear. This is a minor syntactic convenience, but it’s useful for embedding things like HTML and XML code in a Python script:

```
>>> msg = """ aaaaaaaaaaaa
bbb' ' bbbbbbbbbbb' bbbbbbb' bbbb
cccccccccccccc"""
>>> msg
'\naaaaaaaaaaaa\nbbb'\ ' ' bbbbbbbbbbb' bbbbbbb' bbbb\ncccccccccccccc'
```

Python also supports a *raw* string literal that turns off the backslash escape mechanism (such string literals start with the letter *r*), as well as *Unicode* string support that supports internationalization. In 3.0, the basic `str` string type handles Unicode too (which makes sense, given that ASCII text is a simple kind of Unicode), and a `bytes` type represents raw byte strings; in 2.6, Unicode is a separate type, and `str` handles both 8-bit strings and binary data. Files are also changed in 3.0 to return and accept `str` for text and `bytes` for binary data. We’ll meet all these special string forms in later chapters.

Pattern Matching

One point worth noting before we move on is that none of the string object’s methods support pattern-based text processing. Text pattern matching is an advanced tool outside this book’s scope, but readers with backgrounds in other scripting languages may be interested to know that to do pattern matching in Python, we import a module called

`re`. This module has analogous calls for searching, splitting, and replacement, but because we can use patterns to specify substrings, we can be much more general:

```
>>> import re
>>> match = re.match('Hello[ \t]*(.*)world', 'Hello    Python world')
>>> match.group(1)
'Python '
```

This example searches for a substring that begins with the word “Hello,” followed by zero or more tabs or spaces, followed by arbitrary characters to be saved as a matched group, terminated by the word “world.” If such a substring is found, portions of the substring matched by parts of the pattern enclosed in parentheses are available as groups. The following pattern, for example, picks out three groups separated by slashes:

```
>>> match = re.match('/(.*)/(.*)/(.*)', '/usr/home/lumberjack')
>>> match.groups()
('usr', 'home', 'lumberjack')
```

Pattern matching is a fairly advanced text-processing tool by itself, but there is also support in Python for even more advanced language processing, including natural language processing. I’ve already said enough about strings for this tutorial, though, so let’s move on to the next type.

Lists

The Python list object is the most general sequence provided by the language. Lists are positionally ordered collections of arbitrarily typed objects, and they have no fixed size. They are also mutable—unlike strings, lists can be modified in-place by assignment to offsets as well as a variety of list method calls.

Sequence Operations

Because they are sequences, lists support all the sequence operations we discussed for strings; the only difference is that the results are usually lists instead of strings. For instance, given a three-item list:

```
>>> L = [123, 'spam', 1.23]           # A list of three different-type objects
>>> len(L)                           # Number of items in the list
3
```

we can index, slice, and so on, just as for strings:

```
>>> L[0]                             # Indexing by position
123

>>> L[: -1]                          # Slicing a list returns a new list
[123, 'spam']

>>> L + [4, 5, 6]                    # Concatenation makes a new list too
[123, 'spam', 1.23, 4, 5, 6]
```

```
>>> L                                     # We're not changing the original list
[123, 'spam', 1.23]
```

Type-Specific Operations

Python's lists are related to arrays in other languages, but they tend to be more powerful. For one thing, they have no fixed type constraint—the list we just looked at, for example, contains three objects of completely different types (an integer, a string, and a floating-point number). Further, lists have no fixed size. That is, they can grow and shrink on demand, in response to list-specific operations:

```
>>> L.append('NI')                        # Growing: add object at end of list
>>> L
[123, 'spam', 1.23, 'NI']

>>> L.pop(2)                             # Shrinking: delete an item in the middle
1.23

>>> L                                     # "del L[2]" deletes from a list too
[123, 'spam', 'NI']
```

Here, the list `append` method expands the list's size and inserts an item at the end; the `pop` method (or an equivalent `del` statement) then removes an item at a given offset, causing the list to shrink. Other list methods insert an item at an arbitrary position (`insert`), remove a given item by value (`remove`), and so on. Because lists are mutable, most list methods also change the list object in-place, instead of creating a new one:

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

The list `sort` method here, for example, orders the list in ascending fashion by default, and `reverse` reverses it—in both cases, the methods modify the list directly.

Bounds Checking

Although lists have no fixed size, Python still doesn't allow us to reference items that are not present. Indexing off the end of a list is always a mistake, but so is assigning off the end:

```
>>> L
[123, 'spam', 'NI']

>>> L[99]
...error text omitted...
IndexError: list index out of range
```

```
>>> L[99] = 1
...error text omitted...
IndexError: list assignment index out of range
```

This is intentional, as it’s usually an error to try to assign off the end of a list (and a particularly nasty one in the C language, which doesn’t do as much error checking as Python). Rather than silently growing the list in response, Python reports an error. To grow a list, we call list methods such as **append** instead.

Nesting

One nice feature of Python’s core data types is that they support arbitrary nesting—we can nest them in any combination, and as deeply as we like (for example, we can have a list that contains a dictionary, which contains another list, and so on). One immediate application of this feature is to represent matrixes, or “multidimensional arrays” in Python. A list with nested lists will do the job for basic applications:

```
>>> M = [[1, 2, 3],           # A 3 × 3 matrix, as nested lists
          [4, 5, 6],         # Code can span lines if bracketed
          [7, 8, 9]]

>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Here, we’ve coded a list that contains three other lists. The effect is to represent a 3×3 matrix of numbers. Such a structure can be accessed in a variety of ways:

```
>>> M[1]                     # Get row 2
[4, 5, 6]

>>> M[1][2]                 # Get row 2, then get item 3 within the row
6
```

The first operation here fetches the entire second row, and the second grabs the third item within that row. Stringing together index operations takes us deeper and deeper into our nested-object structure.[†]

Comprehensions

In addition to sequence operations and list methods, Python includes a more advanced operation known as a *list comprehension expression*, which turns out to be a powerful way to process structures like our matrix. Suppose, for instance, that we need to extract the second column of our sample matrix. It’s easy to grab rows by simple indexing

[†] This matrix structure works for small-scale tasks, but for more serious number crunching you will probably want to use one of the numeric extensions to Python, such as the open source *NumPy* system. Such tools can store and process large matrixes much more efficiently than our nested list structure. NumPy has been said to turn Python into the equivalent of a free and more powerful version of the Matlab system, and organizations such as NASA, Los Alamos, and JPMorgan Chase use this tool for scientific and financial tasks. Search the Web for more details.

because the matrix is stored by rows, but it's almost as easy to get a column with a list comprehension:

```
>>> col2 = [row[1] for row in M]          # Collect the items in column 2
>>> col2
[2, 5, 8]

>>> M                                     # The matrix is unchanged
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

List comprehensions derive from set notation; they are a way to build a new list by running an expression on each item in a sequence, one at a time, from left to right. List comprehensions are coded in square brackets (to tip you off to the fact that they make a list) and are composed of an expression and a looping construct that share a variable name (`row`, here). The preceding list comprehension means basically what it says: “Give me `row[1]` for each `row` in matrix `M`, in a new list.” The result is a new list containing column 2 of the matrix.

List comprehensions can be more complex in practice:

```
>>> [row[1] + 1 for row in M]             # Add 1 to each item in column 2
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0] # Filter out odd items
[2, 8]
```

The first operation here, for instance, adds 1 to each item as it is collected, and the second uses an `if` clause to filter odd numbers out of the result using the `%` modulus expression (remainder of division). List comprehensions make new lists of results, but they can be used to iterate over any iterable object. Here, for instance, we use list comprehensions to step over a hardcoded list of coordinates and a string:

```
>>> diag = [M[i][i] for i in [0, 1, 2]]   # Collect a diagonal from matrix
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'spam']      # Repeat characters in a string
>>> doubles
['ss', 'pp', 'aa', 'mm']
```

List comprehensions, and relatives like the `map` and `filter` built-in functions, are a bit too involved for me to say more about them here. The main point of this brief introduction is to illustrate that Python includes both simple and advanced tools in its arsenal. List comprehensions are an optional feature, but they tend to be handy in practice and often provide a substantial processing speed advantage. They also work on any type that is a sequence in Python, as well as some types that are not. You'll hear much more about them later in this book.

As a preview, though, you'll find that in recent Pythons, comprehension syntax in parentheses can also be used to create *generators* that produce results on demand (the `sum` built-in, for instance, sums items in a sequence):

```
>>> G = (sum(row) for row in M)           # Create a generator of row sums
>>> next(G)
6
>>> next(G)                               # Run the iteration protocol
15
```

The `map` built-in can do similar work, by generating the results of running items through a function. Wrapping it in `list` forces it to return all its values in Python 3.0:

```
>>> list(map(sum, M))                     # Map sum over items in M
[6, 15, 24]
```

In Python 3.0, comprehension syntax can also be used to create *sets* and *dictionaries*:

```
>>> {sum(row) for row in M}               # Create a set of row sums
{24, 6, 15}

>>> {i : sum(M[i]) for i in range(3)}     # Creates key/value table of row sums
{0: 6, 1: 15, 2: 24}
```

In fact, lists, sets, and dictionaries can all be built with comprehensions in 3.0:

```
>>> [ord(x) for x in 'spaam']             # List of character ordinals
[115, 112, 97, 97, 109]
>>> {ord(x) for x in 'spaam'}             # Sets remove duplicates
{112, 97, 115, 109}
>>> {x: ord(x) for x in 'spaam'}          # Dictionary keys are unique
{'a': 97, 'p': 112, 's': 115, 'm': 109}
```

To understand objects like generators, sets, and dictionaries, though, we must move ahead.

Dictionaries

Python dictionaries are something completely different (Monty Python reference intended)—they are not sequences at all, but are instead known as *mappings*. Mappings are also collections of other objects, but they store objects by key instead of by relative position. In fact, mappings don’t maintain any reliable left-to-right order; they simply map keys to associated values. Dictionaries, the only mapping type in Python’s core objects set, are also mutable: they may be changed in-place and can grow and shrink on demand, like lists.

Mapping Operations

When written as literals, dictionaries are coded in curly braces and consist of a series of “key: value” pairs. Dictionaries are useful anytime we need to associate a set of values with keys—to describe the properties of something, for instance. As an example, consider the following three-item dictionary (with keys “food,” “quantity,” and “color”):

```
>>> D = {'food': 'Spam', 'quantity': 4, 'color': 'pink'}
```


We can index this dictionary by key to fetch and change the keys' associated values. The dictionary index operation uses the same syntax as that used for sequences, but the item in the square brackets is a key, not a relative position:

```
>>> D['food']          # Fetch value of key 'food'
'Spam'

>>> D['quantity'] += 1  # Add 1 to 'quantity' value
>>> D
{'food': 'Spam', 'color': 'pink', 'quantity': 5}
```

Although the curly-braces literal form does see use, it is perhaps more common to see dictionaries built up in different ways. The following code, for example, starts with an empty dictionary and fills it out one key at a time. Unlike out-of-bounds assignments in lists, which are forbidden, assignments to new dictionary keys create those keys:

```
>>> D = {}
>>> D['name'] = 'Bob'      # Create keys by assignment
>>> D['job'] = 'dev'
>>> D['age'] = 40

>>> D
{'age': 40, 'job': 'dev', 'name': 'Bob'}

>>> print(D['name'])
Bob
```

Here, we're effectively using dictionary keys as field names in a record that describes someone. In other applications, dictionaries can also be used to replace searching operations—indexing a dictionary by key is often the fastest way to code a search in Python.

Nesting Revisited

In the prior example, we used a dictionary to describe a hypothetical person, with three keys. Suppose, though, that the information is more complex. Perhaps we need to record a first name and a last name, along with multiple job titles. This leads to another application of Python's object nesting in action. The following dictionary, coded all at once as a literal, captures more structured information:

```
>>> rec = {'name': {'first': 'Bob', 'last': 'Smith'},
           'job': ['dev', 'mgr'],
           'age': 40.5}
```

Here, we again have a three-key dictionary at the top (keys “name,” “job,” and “age”), but the values have become more complex: a nested dictionary for the name to support multiple parts, and a nested list for the job to support multiple roles and future expansion. We can access the components of this structure much as we did for our matrix earlier, but this time some of our indexes are dictionary keys, not list offsets:

```

>>> rec['name']                                # 'name' is a nested dictionary
{'last': 'Smith', 'first': 'Bob'}

>>> rec['name']['last']                        # Index the nested dictionary
'Smith'

>>> rec['job']                                  # 'job' is a nested list
['dev', 'mgr']
>>> rec['job'][-1]                             # Index the nested list
'mgr'

>>> rec['job'].append('janitor')                # Expand Bob's job description in-place
>>> rec
{'age': 40.5, 'job': ['dev', 'mgr', 'janitor'], 'name': {'last': 'Smith',
'first': 'Bob'}}

```

Notice how the last operation here expands the nested job list—because the job list is a separate piece of memory from the dictionary that contains it, it can grow and shrink freely (object memory layout will be discussed further later in this book).

The real reason for showing you this example is to demonstrate the *flexibility* of Python’s core data types. As you can see, nesting allows us to build up complex information structures directly and easily. Building a similar structure in a low-level language like C would be tedious and require much more code: we would have to lay out and declare structures and arrays, fill out values, link everything together, and so on. In Python, this is all automatic—running the expression creates the entire nested object structure for us. In fact, this is one of the main benefits of scripting languages like Python.

Just as importantly, in a lower-level language we would have to be careful to clean up all of the object’s space when we no longer need it. In Python, when we lose the last reference to the object—by assigning its variable to something else, for example—all of the memory space occupied by that object’s structure is automatically cleaned up for us:

```

>>> rec = 0                                    # Now the object's space is reclaimed

```

Technically speaking, Python has a feature known as *garbage collection* that cleans up unused memory as your program runs and frees you from having to manage such details in your code. In Python, the space is reclaimed immediately, as soon as the last reference to an object is removed. We’ll study how this works later in this book; for now, it’s enough to know that you can use objects freely, without worrying about creating their space or cleaning up as you go.[‡]

[‡] Keep in mind that the `rec` record we just created really could be a database record, when we employ Python’s *object persistence* system—an easy way to store native Python objects in files or access-by-key databases. We won’t go into details here, but watch for discussion of Python’s `pickle` and `shelve` modules later in this book.

Sorting Keys: for Loops

As mappings, as we’ve already seen, dictionaries only support accessing items by key. However, they also support type-specific operations with method calls that are useful in a variety of common use cases.

As mentioned earlier, because dictionaries are not sequences, they don’t maintain any dependable left-to-right order. This means that if we make a dictionary and print it back, its keys may come back in a different order than that in which we typed them:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

What do we do, though, if we do need to impose an ordering on a dictionary’s items? One common solution is to grab a list of keys with the dictionary `keys` method, sort that with the list `sort` method, and then step through the result with a Python `for` loop (be sure to press the Enter key twice after coding the `for` loop below—as explained in [Chapter 3](#), an empty line means “go” at the interactive prompt, and the prompt changes to “...” on some interfaces):

```
>>> Ks = list(D.keys())           # Unordered keys list
>>> Ks                           # A list in 2.6, "view" in 3.0: use list()
['a', 'c', 'b']

>>> Ks.sort()                   # Sorted keys list
>>> Ks
['a', 'b', 'c']

>>> for key in Ks:               # Iterate though sorted keys
    print(key, '=>', D[key])     # <== press Enter twice here

a => 1
b => 2
c => 3
```

This is a three-step process, although, as we’ll see in later chapters, in recent versions of Python it can be done in one step with the newer `sorted` built-in function. The `sorted` call returns the result and sorts a variety of object types, in this case sorting dictionary keys automatically:

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for key in sorted(D):
    print(key, '=>', D[key])

a => 1
b => 2
c => 3
```

Besides showcasing dictionaries, this use case serves to introduce the Python `for` loop. The `for` loop is a simple and efficient way to step through all the items in a sequence

and run a block of code for each item in turn. A user-defined loop variable (*key*, here) is used to reference the current item each time through. The net effect in our example is to print the unordered dictionary's keys and values, in sorted-key order.

The `for` loop, and its more general cousin the `while` loop, are the main ways we code repetitive tasks as statements in our scripts. Really, though, the `for` loop (like its relative the list comprehension, which we met earlier) is a sequence operation. It works on any object that is a sequence and, like the list comprehension, even on some things that are not. Here, for example, it is stepping across the characters in a string, printing the uppercase version of each as it goes:

```
>>> for c in 'spam':
        print(c.upper())

S
P
A
M
```

Python's `while` loop is a more general sort of looping tool, not limited to stepping across sequences:

```
>>> x = 4
>>> while x > 0:
        print('spam!' * x)
        x -= 1

spam!spam!spam!spam!
spam!spam!spam!
spam!spam!
spam!
```

We'll discuss looping statements, syntax, and tools in depth later in the book.

Iteration and Optimization

If the last section's `for` loop looks like the list comprehension expression introduced earlier, it should: both are really general iteration tools. In fact, both will work on any object that follows the *iteration protocol*—a pervasive idea in Python that essentially means a physically stored sequence in memory, or an object that generates one item at a time in the context of an iteration operation. An object falls into the latter category if it responds to the `iter` built-in with an object that advances in response to `next`. The *generator* comprehension expression we saw earlier is such an object.

I'll have more to say about the iteration protocol later in this book. For now, keep in mind that every Python tool that scans an object from left to right uses the iteration protocol. This is why the `sorted` call used in the prior section works on the dictionary directly—we don't have to call the `keys` method to get a sequence because dictionaries are iterable objects, with a `next` that returns successive keys.

This also means that any list comprehension expression, such as this one, which computes the squares of a list of numbers:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]
```

can always be coded as an equivalent `for` loop that builds the result list manually by appending as it goes:

```
>>> squares = []
>>> for x in [1, 2, 3, 4, 5]:           # This is what a list comprehension does
    squares.append(x ** 2)             # Both run the iteration protocol internally

>>> squares
[1, 4, 9, 16, 25]
```

The list comprehension, though, and related functional programming tools like `map` and `filter`, will generally run faster than a `for` loop today (perhaps even twice as fast)—a property that could matter in your programs for large data sets. Having said that, though, I should point out that performance measures are tricky business in Python because it optimizes so much, and performance can vary from release to release.

A major rule of thumb in Python is to code for simplicity and readability first and worry about performance later, after your program is working, and after you’ve proved that there is a genuine performance concern. More often than not, your code will be quick enough as it is. If you do need to tweak code for performance, though, Python includes tools to help you out, including the `time` and `timeit` modules and the `profile` module. You’ll find more on these later in this book, and in the Python manuals.

Missing Keys: if Tests

One other note about dictionaries before we move on. Although we can assign to a new key to expand a dictionary, fetching a nonexistent key is still a mistake:

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D['e'] = 99                      # Assigning new keys grows dictionaries
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                          # Referencing a nonexistent key is an error
...error text omitted...
KeyError: 'f'
```

This is what we want—it’s usually a programming error to fetch something that isn’t really there. But in some generic programs, we can’t always know what keys will be present when we write our code. How do we handle such cases and avoid errors? One trick is to test ahead of time. The dictionary `in` membership expression allows us to

query the existence of a key and branch on the result with a Python `if` statement (as with the `for`, be sure to press Enter twice to run the `if` interactively here):

```
>>> 'f' in D
False

>>> if not 'f' in D:
    print('missing')

missing
```

I'll have much more to say about the `if` statement and statement syntax in general later in this book, but the form we're using here is straightforward: it consists of the word `if`, followed by an expression that is interpreted as a true or false result, followed by a block of code to run if the test is true. In its full form, the `if` statement can also have an `else` clause for a default case, and one or more `elif` (else if) clauses for other tests. It's the main selection tool in Python, and it's the way we code logic in our scripts.

Still, there are other ways to create dictionaries and avoid accessing nonexistent keys: the `get` method (a conditional index with a default); the Python 2.X `has_key` method (which is no longer available in 3.0); the `try` statement (a tool we'll first meet in [Chapter 10](#) that catches and recovers from exceptions altogether); and the `if/else` expression (essentially, an `if` statement squeezed onto a single line). Here are a few examples:

```
>>> value = D.get('x', 0)                # Index but with a default
>>> value
0
>>> value = D['x'] if 'x' in D else 0     # if/else expression form
>>> value
0
```

We'll save the details on such alternatives until a later chapter. For now, let's move on to tuples.

Tuples

The tuple object (pronounced “toople” or “tuhple,” depending on who you ask) is roughly like a list that cannot be changed—tuples are sequences, like lists, but they are immutable, like strings. Syntactically, they are coded in parentheses instead of square brackets, and they support arbitrary types, arbitrary nesting, and the usual sequence operations:

```
>>> T = (1, 2, 3, 4)                    # A 4-item tuple
>>> len(T)                              # Length
4

>>> T + (5, 6)                          # Concatenation
(1, 2, 3, 4, 5, 6)

>>> T[0]                                # Indexing, slicing, and more
1
```

Tuples also have two type-specific callable methods in Python 3.0, but not nearly as many as lists:

```
>>> T.index(4)           # Tuple methods: 4 appears at offset 3
3
>>> T.count(4)          # 4 appears once
1
```

The primary distinction for tuples is that they cannot be changed once created. That is, they are immutable sequences:

```
>>> T[0] = 2             # Tuples are immutable
...error text omitted...
TypeError: 'tuple' object does not support item assignment
```

Like lists and dictionaries, tuples support mixed types and nesting, but they don't grow and shrink because they are immutable:

```
>>> T = ('spam', 3.0, [11, 22, 33])
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

Why Tuples?

So, why have a type that is like a list, but supports fewer operations? Frankly, tuples are not generally used as often as lists in practice, but their immutability is the whole point. If you pass a collection of objects around your program as a list, it can be changed anywhere; if you use a tuple, it cannot. That is, tuples provide a sort of integrity constraint that is convenient in programs larger than those we'll write here. We'll talk more about tuples later in the book. For now, though, let's jump ahead to our last major core type: the file.

Files

File objects are Python code's main interface to external files on your computer. Files are a core type, but they're something of an oddball—there is no specific literal syntax for creating them. Rather, to create a file object, you call the built-in `open` function, passing in an external filename and a processing mode as strings. For example, to create a text output file, you would pass in its name and the `'w'` processing mode string to write data:

```
>>> f = open('data.txt', 'w')   # Make a new file in output mode
>>> f.write('Hello\n')          # Write strings of bytes to it
6
>>> f.write('world\n')          # Returns number of bytes written in Python 3.0
6
>>> f.close()                  # Close to flush output buffers to disk
```

This creates a file in the current directory and writes text to it (the filename can be a full directory path if you need to access a file elsewhere on your computer). To read back what you just wrote, reopen the file in `'r'` processing mode, for reading text input—this is the default if you omit the mode in the call. Then read the file’s content into a string, and display it. A file’s contents are always a string in your script, regardless of the type of data the file contains:

```
>>> f = open('data.txt')           # 'r' is the default processing mode
>>> text = f.read()                # Read entire file into a string
>>> text
'Hello\nworld\n'

>>> print(text)                    # print interprets control characters
Hello
world

>>> text.split()                   # File content is always a string
['Hello', 'world']
```

Other file object methods support additional features we don’t have time to cover here. For instance, file objects provide more ways of reading and writing (`read` accepts an optional byte size, `readline` reads one line at a time, and so on), as well as other tools (`seek` moves to a new file position). As we’ll see later, though, the best way to read a file today is to *not read it at all*—files provide an *iterator* that automatically reads line by line in `for` loops and other contexts.

We’ll meet the full set of file methods later in this book, but if you want a quick preview now, run a `dir` call on any open file and a `help` on any of the method names that come back:

```
>>> dir(f)
[ ...many names omitted...
'buffer', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline',
'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write',
'writelines']

>>> help(f.seek)
...try it and see...
```

Later in the book, we’ll also see that files in Python 3.0 draw a sharp distinction between text and binary data. *Text files* represent content as strings and perform Unicode encoding and decoding automatically, while *binary files* represent content as a special `bytes` string type and allow you to access file content unaltered:

```
>>> data = open('data.bin', 'rb').read()   # Open binary file
>>> data                                    # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]
b'spam'
```


Although you won't generally need to care about this distinction if you deal only with ASCII text, Python 3.0's strings and files are an asset if you deal with internationalized applications or byte-oriented data.

Other File-Like Tools

The `open` function is the workhorse for most file processing you will do in Python. For more advanced tasks, though, Python comes with additional file-like tools: pipes, FIFOs, sockets, keyed-access files, persistent object shelves, descriptor-based files, relational and object-oriented database interfaces, and more. Descriptor files, for instance, support file locking and other low-level tools, and sockets provide an interface for networking and interprocess communication. We won't cover many of these topics in this book, but you'll find them useful once you start programming Python in earnest.

Other Core Types

Beyond the core types we've seen so far, there are others that may or may not qualify for membership in the set, depending on how broadly it is defined. *Sets*, for example, are a recent addition to the language that are neither mappings nor sequences; rather, they are unordered collections of unique and immutable objects. Sets are created by calling the built-in `set` function or using new set literals and expressions in 3.0, and they support the usual mathematical set operations (the choice of new `{...}` syntax for set literals in 3.0 makes sense, since sets are much like the keys of a valueless dictionary):

```
>>> X = set('spam')           # Make a set out of a sequence in 2.6 and 3.0
>>> Y = {'h', 'a', 'm'}       # Make a set with new 3.0 set literals
>>> X, Y
({'a', 'p', 's', 'm'}, {'a', 'h', 'm'})

>>> X & Y                       # Intersection
{'a', 'm'}

>>> X | Y                       # Union
{'a', 'p', 's', 'h', 'm'}

>>> X - Y                       # Difference
{'p', 's'}

>>> {x ** 2 for x in [1, 2, 3, 4]} # Set comprehensions in 3.0
{16, 1, 4, 9}
```

In addition, Python recently grew a few new numeric types: *decimal* numbers (fixed-precision floating-point numbers) and *fraction* numbers (rational numbers with both a numerator and a denominator). Both can be used to work around the limitations and inherent inaccuracies of floating-point math:

```
>>> 1 / 3                       # Floating-point (use .0 in Python 2.6)
0.3333333333333333
>>> (2/3) + (1/2)
```



```
>>> type(type(L))           # See Chapter 31 for more on class types
<class 'type'>
```

Besides allowing you to explore your objects interactively, the practical application of this is that it allows code to check the types of the objects it processes. In fact, there are at least three ways to do so in a Python script:

```
>>> if type(L) == type([]):   # Type testing, if you must...
    print('yes')
```

```
yes
>>> if type(L) == list:      # Using the type name
    print('yes')
```

```
yes
>>> if isinstance(L, list):   # Object-oriented tests
    print('yes')
```

```
yes
```

Now that I’ve shown you all these ways to do type testing, however, I am required by law to tell you that doing so is almost always the wrong thing to do in a Python program (and often a sign of an ex-C programmer first starting to use Python!). The reason why won’t become completely clear until later in the book, when we start writing larger code units such as functions, but it’s a (perhaps *the*) core Python concept. By checking for specific types in your code, you effectively break its flexibility—you limit it to working on just one type. Without such tests, your code may be able to work on a whole range of types.

This is related to the idea of polymorphism mentioned earlier, and it stems from Python’s lack of type declarations. As you’ll learn, in Python, we code to object *interfaces* (operations supported), not to types. Not caring about specific types means that code is automatically applicable to many of them—any object with a compatible interface will work, regardless of its specific type. Although type checking is supported—and even required, in some rare cases—you’ll see that it’s not usually the “Pythonic” way of thinking. In fact, you’ll find that polymorphism is probably the key idea behind using Python well.

User-Defined Classes

We’ll study *object-oriented programming* in Python—an optional but powerful feature of the language that cuts development time by supporting programming by customization—in depth later in this book. In abstract terms, though, classes define new types of objects that extend the core set, so they merit a passing glance here. Say, for example, that you wish to have a type of object that models employees. Although there is no such specific core type in Python, the following user-defined class might fit the bill:

```
>>> class Worker:
    def __init__(self, name, pay):    # Initialize when created
        self.name = name             # self is the new object
```

```

        self.pay = pay
    def lastName(self):
        return self.name.split()[-1]           # Split string on blanks
    def giveRaise(self, percent):
        self.pay *= (1.0 + percent)             # Update pay in-place

```

This class defines a new kind of object that will have **name** and **pay** attributes (sometimes called *state information*), as well as two bits of behavior coded as functions (normally called *methods*). Calling the class like a function generates instances of our new type, and the class’s methods automatically receive the instance being processed by a given method call (in the **self** argument):

```

>>> bob = Worker('Bob Smith', 50000)           # Make two instances
>>> sue = Worker('Sue Jones', 60000)           # Each has name and pay attrs
>>> bob.lastName()                             # Call method: bob is self
'Smith'
>>> sue.lastName()                             # sue is the self subject
'Jones'
>>> sue.giveRaise(.10)                         # Updates sue's pay
>>> sue.pay
66000.0

```

The implied “self” object is why we call this an object-oriented model: there is always an implied subject in functions within a class. In a sense, though, the class-based type simply builds on and uses core types—a user-defined **Worker** object here, for example, is just a collection of a string and a number (**name** and **pay**, respectively), plus functions for processing those two built-in objects.

The larger story of classes is that their inheritance mechanism supports software hierarchies that lend themselves to customization by extension. We extend software by writing new classes, not by changing what already works. You should also know that classes are an optional feature of Python, and simpler built-in types such as lists and dictionaries are often better tools than user-coded classes. This is all well beyond the bounds of our introductory object-type tutorial, though, so consider this just a preview; for full disclosure on user-defined types coded with classes, you’ll have to read on to [Part VI](#).

And Everything Else

As mentioned earlier, everything you can process in a Python script is a type of object, so our object type tour is necessarily incomplete. However, even though everything in Python is an “object,” only those types of objects we’ve met so far are considered part of Python’s core type set. Other types in Python either are objects related to program execution (like functions, modules, classes, and compiled code), which we will study later, or are implemented by imported module functions, not language syntax. The latter of these also tend to have application-specific roles—text patterns, database interfaces, network connections, and so on.

Moreover, keep in mind that the objects we’ve met here are objects, but not necessarily *object-oriented*—a concept that usually requires inheritance and the Python `class` statement, which we’ll meet again later in this book. Still, Python’s core objects are the workhorses of almost every Python script you’re likely to meet, and they usually are the basis of larger noncore types.

Chapter Summary

And that’s a wrap for our concise data type tour. This chapter has offered a brief introduction to Python’s core object types and the sorts of operations we can apply to them. We’ve studied generic operations that work on many object types (sequence operations such as indexing and slicing, for example), as well as type-specific operations available as method calls (for instance, string splits and list appends). We’ve also defined some key terms, such as immutability, sequences, and polymorphism.

Along the way, we’ve seen that Python’s core object types are more flexible and powerful than what is available in lower-level languages such as C. For instance, Python’s lists and dictionaries obviate most of the work you do to support collections and searching in lower-level languages. Lists are ordered collections of other objects, and dictionaries are collections of other objects that are indexed by key instead of by position. Both dictionaries and lists may be nested, can grow and shrink on demand, and may contain objects of any type. Moreover, their space is automatically cleaned up as you go.

I’ve skipped most of the details here in order to provide a quick tour, so you shouldn’t expect all of this chapter to have made sense yet. In the next few chapters, we’ll start to dig deeper, filling in details of Python’s core object types that were omitted here so you can gain a more complete understanding. We’ll start off in the next chapter with an in-depth look at Python numbers. First, though, another quiz to review.

Test Your Knowledge: Quiz

We’ll explore the concepts introduced in this chapter in more detail in upcoming chapters, so we’ll just cover the big ideas here:

1. Name four of Python’s core data types.
2. Why are they called “core” data types?
3. What does “immutable” mean, and which three of Python’s core types are considered immutable?
4. What does “sequence” mean, and which three types fall into that category?

5. What does “mapping” mean, and which core type is a mapping?
6. What is “polymorphism,” and why should you care?

Test Your Knowledge: Answers

1. Numbers, strings, lists, dictionaries, tuples, files, and sets are generally considered to be the core object (data) types. Types, `None`, and Booleans are sometimes classified this way as well. There are multiple number types (integer, floating point, complex, fraction, and decimal) and multiple string types (simple strings and Unicode strings in Python 2.X, and text strings and byte strings in Python 3.X).
2. They are known as “core” types because they are part of the Python language itself and are always available; to create other objects, you generally must call functions in imported modules. Most of the core types have specific syntax for generating the objects: `'spam'`, for example, is an expression that makes a string and determines the set of operations that can be applied to it. Because of this, core types are hardwired into Python’s syntax. In contrast, you must call the built-in `open` function to create a file object.
3. An “immutable” object is an object that cannot be changed after it is created. Numbers, strings, and tuples in Python fall into this category. While you cannot change an immutable object in-place, you can always make a new one by running an expression.
4. A “sequence” is a positionally ordered collection of objects. Strings, lists, and tuples are all sequences in Python. They share common sequence operations, such as indexing, concatenation, and slicing, but also have type-specific method calls.
5. The term “mapping” denotes an object that maps keys to associated values. Python’s dictionary is the only mapping type in the core type set. Mappings do not maintain any left-to-right positional ordering; they support access to data stored by key, plus type-specific method calls.
6. “Polymorphism” means that the meaning of an operation (like a `+`) depends on the objects being operated on. This turns out to be a key idea (perhaps *the* key idea) behind using Python well—not constraining code to specific types makes that code automatically applicable to many types.

Numeric Types

This chapter begins our in-depth tour of the Python language. In Python, data takes the form of *objects*—either built-in objects that Python provides, or objects we create using Python tools and other languages such as C. In fact, objects are the basis of every Python program you will ever write. Because they are the most fundamental notion in Python programming, objects are also our first focus in this book.

In the preceding chapter, we took a quick pass over Python’s core object types. Although essential terms were introduced in that chapter, we avoided covering too many specifics in the interest of space. Here, we’ll begin a more careful second look at data type concepts, to fill in details we glossed over earlier. Let’s get started by exploring our first data type category: Python’s numeric types.

Numeric Type Basics

Most of Python’s number types are fairly typical and will probably seem familiar if you’ve used almost any other programming language in the past. They can be used to keep track of your bank balance, the distance to Mars, the number of visitors to your website, and just about any other numeric quantity.

In Python, numbers are not really a single object type, but a category of similar types. Python supports the usual numeric types (integers and floating points), as well as literals for creating numbers and expressions for processing them. In addition, Python provides more advanced numeric programming support and objects for more advanced work. A complete inventory of Python’s numeric toolbox includes:

- Integers and floating-point numbers
- Complex numbers
- Fixed-precision decimal numbers

- Rational fraction numbers
- Sets
- Booleans
- Unlimited integer precision
- A variety of numeric built-ins and modules

This chapter starts with basic numbers and fundamentals, then moves on to explore the other tools in this list. Before we jump into code, though, the next few sections get us started with a brief overview of how we write and process numbers in our scripts.

Numeric Literals

Among its basic types, Python provides *integers* (positive and negative whole numbers) and *floating-point* numbers (numbers with a fractional part, sometimes called “floats” for economy). Python also allows us to write integers using hexadecimal, octal, and binary literals; offers a complex number type; and allows integers to have unlimited precision (they can grow to have as many digits as your memory space allows). [Table 5-1](#) shows what Python’s numeric types look like when written out in a program, as literals.

Table 5-1. Basic numeric literals

Literal	Interpretation
1234, -24, 0, 999999999999999	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0177, 0x9ff, 0b101010	Octal, hex, and binary literals in 2.6
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals in 3.0
3+4j, 3.0+4.0j, 3J	Complex number literals

In general, Python’s numeric type literals are straightforward to write, but a few coding concepts are worth highlighting here:

Integer and floating-point literals

Integers are written as strings of decimal digits. Floating-point numbers have a decimal point and/or an optional signed exponent introduced by an `e` or `E` and followed by an optional sign. If you write a number with a decimal point or exponent, Python makes it a floating-point object and uses floating-point (not integer) math when the object is used in an expression. Floating-point numbers are implemented as C “doubles,” and therefore get as much precision as the C compiler used to build the Python interpreter gives to doubles.

Integers in Python 2.6: normal and long

In Python 2.6 there are two integer types, normal (32 bits) and long (unlimited precision), and an integer may end in an `l` or `L` to force it to become a long integer. Because integers are automatically converted to long integers when their values overflow 32 bits, you never need to type the letter `L` yourself—Python automatically converts up to long integer when extra precision is needed.

Integers in Python 3.0: a single type

In Python 3.0, the normal and long integer types have been merged—there is only integer, which automatically supports the unlimited precision of Python 2.6’s separate long integer type. Because of this, integers can no longer be coded with a trailing `l` or `L`, and integers never print with this character either. Apart from this, most programs are unaffected by this change, unless they do type testing that checks for 2.6 long integers.

Hexadecimal, octal, and binary literals

Integers may be coded in decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2). Hexadecimals start with a leading `0x` or `0X`, followed by a string of hexadecimal digits (`0–9` and `A–F`). Hex digits may be coded in lower- or uppercase. Octal literals start with a leading `0o` or `0O` (zero and lower- or uppercase letter “o”), followed by a string of digits (`0–7`). In 2.6 and earlier, octal literals can also be coded with just a leading `0`, but not in 3.0 (this original octal form is too easily confused with decimal, and is replaced by the new `0o` format). Binary literals, new in 2.6 and 3.0, begin with a leading `0b` or `0B`, followed by binary digits (`0–1`).

Note that all of these literals produce integer objects in program code; they are just alternative syntaxes for specifying values. The built-in calls `hex(I)`, `oct(I)`, and `bin(I)` convert an integer to its representation string in these three bases, and `int(str, base)` converts a runtime string to an integer per a given base.

Complex numbers

Python complex literals are written as *realpart+imaginarypart*, where the *imaginarypart* is terminated with a `j` or `J`. The *realpart* is technically optional, so the *imaginarypart* may appear on its own. Internally, complex numbers are implemented as pairs of floating-point numbers, but all numeric operations perform complex math when applied to complex numbers. Complex numbers may also be created with the `complex(real, imag)` built-in call.

Coding other numeric types

As we’ll see later in this chapter, there are additional, more advanced number types not included in [Table 5-1](#). Some of these are created by calling functions in imported modules (e.g., decimals and fractions), and others have literal syntax all their own (e.g., sets).

Built-in Numeric Tools

Besides the built-in number literals shown in [Table 5-1](#), Python provides a set of tools for processing number objects:

Expression operators

`+`, `-`, `*`, `/`, `>>`, `**`, `&`, etc.

Built-in mathematical functions

`pow`, `abs`, `round`, `int`, `hex`, `bin`, etc.

Utility modules

`random`, `math`, etc.

We'll meet all of these as we go along.

Although numbers are primarily processed with expressions, built-ins, and modules, they also have a handful of type-specific *methods* today, which we'll meet in this chapter as well. Floating-point numbers, for example, have an `as_integer_ratio` method that is useful for the fraction number type, and an `is_integer` method to test if the number is an integer. Integers have various attributes, including a new `bit_length` method in the upcoming Python 3.1 release that gives the number of bits necessary to represent the object's value. Moreover, as part collection and part number, *sets* also support both methods and expressions.

Since expressions are the most essential tool for most number types, though, let's turn to them next.

Python Expression Operators

Perhaps the most fundamental tool that processes numbers is the *expression*: a combination of numbers (or other objects) and operators that computes a value when executed by Python. In Python, expressions are written using the usual mathematical notation and operator symbols. For instance, to add two numbers `X` and `Y` you would say `X + Y`, which tells Python to apply the `+` operator to the values named by `X` and `Y`. The result of the expression is the sum of `X` and `Y`, another number object.

[Table 5-2](#) lists all the operator expressions available in Python. Many are self-explanatory; for instance, the usual mathematical operators (`+`, `-`, `*`, `/`, and so on) are supported. A few will be familiar if you've used other languages in the past: `%` computes a division remainder, `<<` performs a bitwise left-shift, `&` computes a bitwise AND result, and so on. Others are more Python-specific, and not all are numeric in nature: for example, the `is` operator tests object identity (i.e., address in memory, a strict form of equality), and `lambda` creates unnamed functions.

Table 5-2. Python expression operators and precedence

Operators	Description
yield x	Generator function send protocol
lambda args: expression	Anonymous function generation
x if y else z	Ternary selection (x is evaluated only if y is true)
x or y	Logical OR (y is evaluated only if x is false)
x and y	Logical AND (y is evaluated only if x is true)
not x	Logical negation
x in y, x not in y	Membership (iterables, sets)
x is y, x is not y	Object identity tests
x < y, x <= y, x > y, x >= y	Magnitude comparison, set subset and superset;
x == y, x != y	Value equality operators
x y	Bitwise OR, set union
x ^ y	Bitwise XOR, set symmetric difference
x & y	Bitwise AND, set intersection
x << y, x >> y	Shift x left or right by y bits
x + y	Addition, concatenation;
x - y	Subtraction, set difference
x * y	Multiplication, repetition;
x % y	Remainder, format;
x / y, x // y	Division: true and floor
-x, +x	Negation, identity
~x	Bitwise NOT (inversion)
x ** y	Power (exponentiation)
x[i]	Indexing (sequence, mapping, others)
x[i:j:k]	Slicing
x(...)	Call (function, method, class, other callable)
x.attr	Attribute reference
(...)	Tuple, expression, generator expression
[...]	List, list comprehension
{...}	Dictionary, set, set and dictionary comprehensions

Since this book addresses both Python 2.6 and 3.0, here are some notes about version differences and recent additions related to the operators in [Table 5-2](#):

- In Python 2.6, value inequality can be written as either `X != Y` or `X <> Y`. In Python 3.0, the latter of these options is removed because it is redundant. In either version, best practice is to use `X != Y` for all value inequality tests.
- In Python 2.6, a backquoted expression ``X`` works the same as `repr(X)` and converts objects to display strings. Due to its obscurity, this expression is removed in Python 3.0; use the more readable `str` and `repr` built-in functions, described in [“Numeric Display Formats” on page 115](#).
- The `X // Y` floor division expression always truncates fractional remainders in both Python 2.6 and 3.0. The `X / Y` expression performs true division in 3.0 (retaining remainders) and classic division in 2.6 (truncating for integers). See [“Division: Classic, Floor, and True” on page 117](#).
- The syntax `[...]` is used for both list literals and list comprehension expressions. The latter of these performs an implied loop and collects expression results in a new list. See [Chapters 4, 14, and 20](#) for examples.
- The syntax `(...)` is used for tuples and expressions, as well as generator expressions—a form of list comprehension that produces results on demand, instead of building a result list. See [Chapters 4 and 20](#) for examples. The parentheses may sometimes be omitted in all three constructs.
- The syntax `{...}` is used for dictionary literals, and in Python 3.0 for set literals and both dictionary and set comprehensions. See the set coverage in this chapter and [Chapters 4, 8, 14, and 20](#) for examples.
- The `yield` and ternary `if/else` selection expressions are available in Python 2.5 and later. The former returns `send(...)` arguments in generators; the latter is shorthand for a multiline `if` statement. `yield` requires parentheses if not alone on the right side of an assignment statement.
- Comparison operators may be chained: `X < Y < Z` produces the same result as `X < Y` and `Y < Z`. See [“Comparisons: Normal and Chained” on page 116](#) for details.
- In recent Pythons, the slice expression `X[I:J:K]` is equivalent to indexing with a slice object: `X[slice(I, J, K)]`.
- In Python 2.X, magnitude comparisons of mixed types—converting numbers to a common type, and ordering other mixed types according to the type name—are allowed. In Python 3.0, nonnumeric mixed-type magnitude comparisons are not allowed and raise exceptions; this includes sorts by proxy.
- Magnitude comparisons for dictionaries are also no longer supported in Python 3.0 (though equality tests are); comparing `sorted(dict.items())` is one possible replacement.

We’ll see most of the operators in [Table 5-2](#) in action later; first, though, we need to take a quick look at the ways these operators may be combined in expressions.

Mixed operators follow operator precedence

As in most languages, in Python, more complex expressions are coded by stringing together the operator expressions in [Table 5-2](#). For instance, the sum of two multiplications might be written as a mix of variables and operators:

```
A * B + C * D
```

So, how does Python know which operation to perform first? The answer to this question lies in *operator precedence*. When you write an expression with more than one operator, Python groups its parts according to what are called *precedence rules*, and this grouping determines the order in which the expression's parts are computed. [Table 5-2](#) is ordered by operator precedence:

- Operators lower in the table have higher precedence, and so bind more tightly in mixed expressions.
- Operators in the same row in [Table 5-2](#) generally group from left to right when combined (except for exponentiation, which groups right to left, and comparisons, which chain left to right).

For example, if you write `X + Y * Z`, Python evaluates the multiplication first (`Y * Z`), then adds that result to `X` because `*` has higher precedence (is lower in the table) than `+`. Similarly, in this section's original example, both multiplications (`A * B` and `C * D`) will happen before their results are added.

Parentheses group subexpressions

You can forget about precedence completely if you're careful to group parts of expressions with parentheses. When you enclose subexpressions in parentheses, you override Python's precedence rules; Python always evaluates expressions in parentheses first before using their results in the enclosing expressions.

For instance, instead of coding `X + Y * Z`, you could write one of the following to force Python to evaluate the expression in the desired order:

```
(X + Y) * Z  
X + (Y * Z)
```

In the first case, `+` is applied to `X` and `Y` first, because this subexpression is wrapped in parentheses. In the second case, the `*` is performed first (just as if there were no parentheses at all). Generally speaking, adding parentheses in large expressions is a good idea—it not only forces the evaluation order you want, but also aids readability.

Mixed types are converted up

Besides mixing operators in expressions, you can also mix numeric types. For instance, you can add an integer to a floating-point number:

```
40 + 3.14
```

But this leads to another question: what type is the result—integer or floating-point? The answer is simple, especially if you’ve used almost any other language before: in mixed-type numeric expressions, Python first converts operands *up* to the type of the most complicated operand, and then performs the math on same-type operands. This behavior is similar to type conversions in the C language.

Python ranks the complexity of numeric types like so: integers are simpler than floating-point numbers, which are simpler than complex numbers. So, when an integer is mixed with a floating point, as in the preceding example, the integer is converted up to a floating-point value first, and floating-point math yields the floating-point result. Similarly, any mixed-type expression where one operand is a complex number results in the other operand being converted up to a complex number, and the expression yields a complex result. (In Python 2.6, normal integers are also converted to long integers whenever their values are too large to fit in a normal integer; in 3.0, integers subsume longs entirely.)

You can force the issue by calling built-in functions to convert types manually:

```
>>> int(3.1415)      # Truncates float to integer
3
>>> float(3)         # Converts integer to float
3.0
```

However, you won’t usually need to do this: because Python automatically converts up to the more complex type within an expression, the results are normally what you want.

Also, keep in mind that all these mixed-type conversions apply only when mixing *numeric* types (e.g., an integer and a floating-point) in an expression, including those using numeric and comparison operators. In general, Python does not convert across any other type boundaries automatically. Adding a string to an integer, for example, results in an error, unless you manually convert one or the other; watch for an example when we meet strings in [Chapter 7](#).



In Python 2.6, nonnumeric mixed types can be compared, but no conversions are performed (mixed types compare according to a fixed but arbitrary rule). In 3.0, nonnumeric mixed-type comparisons are not allowed and raise exceptions.

Preview: Operator overloading and polymorphism

Although we’re focusing on built-in numbers right now, all Python operators may be overloaded (i.e., implemented) by Python classes and C extension types to work on objects you create. For instance, you’ll see later that objects coded with classes may be added or concatenated with `+` expressions, indexed with `[i]` expressions, and so on.

Furthermore, Python itself automatically overloads some operators, such that they perform different actions depending on the type of built-in objects being processed.

For example, the `+` operator performs addition when applied to numbers but performs concatenation when applied to sequence objects such as strings and lists. In fact, `+` can mean anything at all when applied to objects you define with classes.

As we saw in the prior chapter, this property is usually called *polymorphism*—a term indicating that the meaning of an operation depends on the type of the objects being operated on. We’ll revisit this concept when we explore functions in [Chapter 16](#), because it becomes a much more obvious feature in that context.

Numbers in Action

On to the code! Probably the best way to understand numeric objects and expressions is to see them in action, so let’s start up the interactive command line and try some basic but illustrative operations (see [Chapter 3](#) for pointers if you need help starting an interactive session).

Variables and Basic Expressions

First of all, let’s exercise some basic math. In the following interaction, we first assign two *variables* (`a` and `b`) to integers so we can use them later in a larger expression. Variables are simply names—created by you or Python—that are used to keep track of information in your program. We’ll say more about this in the next chapter, but in Python:

- Variables are created when they are first assigned values.
- Variables are replaced with their values when used in expressions.
- Variables must be assigned before they can be used in expressions.
- Variables refer to objects and are never declared ahead of time.

In other words, these assignments cause the variables `a` and `b` to spring into existence automatically:

```
% python
>>> a = 3           # Name created
>>> b = 4
```

I’ve also used a *comment* here. Recall that in Python code, text after a `#` mark and continuing to the end of the line is considered to be a comment and is ignored. Comments are a way to write human-readable documentation for your code. Because code you type interactively is temporary, you won’t normally write comments in this context, but I’ve added them to some of this book’s examples to help explain the code.* In the next part of the book, we’ll meet a related feature—documentation strings—that attaches the text of your comments to objects.

* If you’re working along, you don’t need to type any of the comment text from the `#` through to the end of the line; comments are simply ignored by Python and not required parts of the statements we’re running.

Now, let's use our new integer objects in some expressions. At this point, the values of `a` and `b` are still 3 and 4, respectively. Variables like these are replaced with their values whenever they're used inside an expression, and the expression results are echoed back immediately when working interactively:

```
>>> a + 1, a - 1          # Addition (3 + 1), subtraction (3 - 1)
(4, 2)
>>> b * 3, b / 2          # Multiplication (4 * 3), division (4 / 2)
(12, 2.0)
>>> a % 2, b ** 2          # Modulus (remainder), power (4 ** 2)
(1, 16)
>>> 2 + 4.0, 2.0 ** b      # Mixed-type conversions
(6.0, 16.0)
```

Technically, the results being echoed back here are *tuples* of two values because the lines typed at the prompt contain two expressions separated by commas; that's why the results are displayed in parentheses (more on tuples later). Note that the expressions work because the variables `a` and `b` within them have been assigned values. If you use a different variable that has never been assigned, Python reports an error rather than filling in some default value:

```
>>> c * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'c' is not defined
```

You don't need to predeclare variables in Python, but they must have been assigned at least once before you can use them. In practice, this means you have to initialize counters to zero before you can add to them, initialize lists to an empty list before you can append to them, and so on.

Here are two slightly larger expressions to illustrate operator grouping and more about conversions:

```
>>> b / 2 + a              # Same as ((4 / 2) + 3)
5.0
>>> print(b / (2.0 + a))   # Same as 4 / (2.0 + 3))
0.8
```

In the first expression, there are no parentheses, so Python automatically groups the components according to its precedence rules—because `/` is lower in [Table 5-2](#) than `+`, it binds more tightly and so is evaluated first. The result is as if the expression had been organized with parentheses as shown in the comment to the right of the code.

Also, notice that all the numbers are integers in the first expression. Because of that, Python 2.6 performs integer division and addition and will give a result of 5, whereas Python 3.0 performs true division with remainders and gives the result shown. If you want integer division in 3.0, code this as `b // 2 + a` (more on division in a moment).

In the second expression, parentheses are added around the `+` part to force Python to evaluate it first (i.e., before the `/`). We also made one of the operands floating-point by adding a decimal point: `2.0`. Because of the mixed types, Python converts the integer

referenced by `a` to a floating-point value (`3.0`) before performing the `+`. If all the numbers in this expression were integers, integer division (`4 / 5`) would yield the truncated integer `0` in Python 2.6 but the floating-point `0.8` in Python 3.0 (again, stay tuned for division details).

Numeric Display Formats

Notice that we used a `print` operation in the last of the preceding examples. Without the `print`, you'll see something that may look a bit odd at first glance:

```
>>> b / (2.0 + a)          # Auto echo output: more digits
0.800000000000000004

>>> print(b / (2.0 + a))   # print rounds off digits
0.8
```

The full story behind this odd result has to do with the limitations of floating-point hardware and its inability to exactly represent some values in a limited number of bits. Because computer architecture is well beyond this book's scope, though, we'll finesse this by saying that all of the digits in the first output are really there in your computer's floating-point hardware—it's just that you're not accustomed to seeing them. In fact, this is really just a display issue—the interactive prompt's automatic result echo shows more digits than the `print` statement. If you don't want to see all the digits, use `print`; as the sidebar “[str and repr Display Formats](#)” on [page 116](#) will explain, you'll get a user-friendly display.

Note, however, that not all values have so many digits to display:

```
>>> 1 / 2.0
0.5
```

and that there are more ways to display the bits of a number inside your computer than using `print` and automatic echoes:

```
>>> num = 1 / 3.0
>>> num          # Echoes
0.3333333333333331
>>> print(num)   # print rounds
0.3333333333

>>> '%e' % num   # String formatting expression
'3.333333e-001'
>>> '%4.2f' % num # Alternative floating-point format
'0.33'
>>> '{0:4.2f}'.format(num) # String formatting method (Python 2.6 and 3.0)
'0.33'
```

The last three of these expressions employ *string formatting*, a tool that allows for format flexibility, which we will explore in the upcoming chapter on strings ([Chapter 7](#)). Its results are strings that are typically printed to displays or reports.

str and repr Display Formats

Technically, the difference between default interactive echoes and `print` corresponds to the difference between the built-in `repr` and `str` functions:

```
>>> num = 1 / 3
>>> repr(num)          # Used by echoes: as-code form
'0.33333333333333331'
>>> str(num)           # Used by print: user-friendly form
'0.333333333333'
```

Both of these convert arbitrary objects to their string representations: `repr` (and the default interactive echo) produces results that look as though they were code; `str` (and the `print` operation) converts to a typically more user-friendly format if available. Some objects have both—a `str` for general use, and a `repr` with extra details. This notion will resurface when we study both strings and operator overloading in classes, and you'll find more on these built-ins in general later in the book.

Besides providing print strings for arbitrary objects, the `str` built-in is also the name of the string data type and may be called with an encoding name to decode a Unicode string from a byte string. We'll study the latter advanced role in [Chapter 36](#) of this book.

Comparisons: Normal and Chained

So far, we've been dealing with standard numeric operations (addition and multiplication), but numbers can also be compared. Normal comparisons work for numbers exactly as you'd expect—they compare the relative magnitudes of their operands and return a Boolean result (which we would normally test in a larger statement):

```
>>> 1 < 2              # Less than
True
>>> 2.0 >= 1           # Greater than or equal: mixed-type 1 converted to 1.0
True
>>> 2.0 == 2.0         # Equal value
True
>>> 2.0 != 2.0         # Not equal value
False
```

Notice again how mixed types are allowed in numeric expressions (only); in the second test here, Python compares values in terms of the more complex type, float.

Interestingly, Python also allows us to *chain* multiple comparisons together to perform range tests. Chained comparisons are a sort of shorthand for larger Boolean expressions. In short, Python lets us string together magnitude comparison tests to code chained comparisons such as range tests. The expression `(A < B < C)`, for instance, tests whether `B` is between `A` and `C`; it is equivalent to the Boolean test `(A < B and B < C)` but is easier on the eyes (and the keyboard). For example, assume the following assignments:

```
>>> X = 2
>>> Y = 4
>>> Z = 6
```

The following two expressions have identical effects, but the first is shorter to type, and it may run slightly faster since Python needs to evaluate `Y` only once:

```
>>> X < Y < Z           # Chained comparisons: range tests
True
>>> X < Y and Y < Z
True
```

The same equivalence holds for false results, and arbitrary chain lengths are allowed:

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False

>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

You can use other comparisons in chained tests, but the resulting expressions can become nonintuitive unless you evaluate them the way Python does. The following, for instance, is `false` just because `1` is not equal to `2`:

```
>>> 1 == 2 < 3           # Same as: 1 == 2 and 2 < 3
False                    # Not same as: False < 3 (which means 0 < 3, which is true)
```

Python does not compare the `1 == 2` `False` result to `3`—this would technically mean the same as `0 < 3`, which would be `True` (as we’ll see later in this chapter, `True` and `False` are just customized `1` and `0`).

Division: Classic, Floor, and True

You’ve seen how division works in the previous sections, so you should know that it behaves slightly differently in Python 3.0 and 2.6. In fact, there are actually three flavors of division, and two different division operators, one of which changes in 3.0:

`X / Y`

Classic and *true* division. In Python 2.6 and earlier, this operator performs *classic* division, truncating results for integers and keeping remainders for floating-point numbers. In Python 3.0, it performs *true* division, always keeping remainders regardless of types.

`X // Y`

Floor division. Added in Python 2.2 and available in both Python 2.6 and 3.0, this operator always truncates fractional remainders down to their floor, regardless of types.

True division was added to address the fact that the results of the original classic division model are dependent on operand types, and so can be difficult to anticipate in a dynamically typed language like Python. Classic division was removed in 3.0 because of this constraint—the `/` and `//` operators implement true and floor division in 3.0.

In sum:

- In 3.0, the `/` now always performs *true* division, returning a float result that includes any remainder, regardless of operand types. The `//` performs *floor* division, which truncates the remainder and returns an integer for integer operands or a float if any operand is a float.
- In 2.6, the `/` does *classic* division, performing truncating integer division if both operands are integers and float division (keeping remainders) otherwise. The `//` does *floor* division and works as it does in 3.0, performing truncating division for integers and floor division for floats.

Here are the two operators at work in 3.0 and 2.6:

```
C:\misc> C:\Python30\python
>>>
>>> 10 / 4          # Differs in 3.0: keeps remainder
2.5
>>> 10 // 4         # Same in 3.0: truncates remainder
2
>>> 10 / 4.0        # Same in 3.0: keeps remainder
2.5
>>> 10 // 4.0       # Same in 3.0: truncates to floor
2.0

C:\misc> C:\Python26\python
>>>
>>> 10 / 4
2
>>> 10 // 4
2
>>> 10 / 4.0
2.5
>>> 10 // 4.0
2.0
```

Notice that the data type of the result for `//` is still dependent on the operand types in 3.0: if either is a float, the result is a float; otherwise, it is an integer. Although this may seem similar to the type-dependent behavior of `/` in 2.X that motivated its change in 3.0, the type of the return value is much less critical than differences in the return value itself. Moreover, because `//` was provided in part as a backward-compatibility tool for programs that rely on truncating integer division (and this is more common than you might expect), it must return integers for integers.

Supporting either Python

Although `/` behavior differs in 2.6 and 3.0, you can still support both versions in your code. If your programs depend on truncating integer division, use `//` in both 2.6 and 3.0. If your programs require floating-point results with remainders for integers, use `float` to guarantee that one operand is a float around a `/` when run in 2.6:

```
X = Y // Z          # Always truncates, always an int result for ints in 2.6 and 3.0
```

```
X = Y / float(Z)    # Guarantees float division with remainder in either 2.6 or 3.0
```

Alternatively, you can enable 3.0 `/` division in 2.6 with a `__future__` import, rather than forcing it with `float` conversions:

```
C:\misc> C:\Python26\python
>>> from __future__ import division          # Enable 3.0 "/" behavior
>>> 10 / 4
2.5
>>> 10 // 4
2
```

Floor versus truncation

One subtlety: the `//` operator is generally referred to as *truncating* division, but it's more accurate to refer to it as *floor* division—it truncates the result down to its floor, which means the closest whole number below the true result. The net effect is to round down, not strictly truncate, and this matters for negatives. You can see the difference for yourself with the Python `math` module (modules must be imported before you can use their contents; more on this later):

```
>>> import math
>>> math.floor(2.5)
2
>>> math.floor(-2.5)
-3
>>> math.trunc(2.5)
2
>>> math.trunc(-2.5)
-2
```

When running division operators, you only really truncate for positive results, since truncation is the same as floor; for negatives, it's a floor result (really, they are both floor, but floor is the same as truncation for positives). Here's the case for 3.0:

```
C:\misc> c:\python30\python
>>> 5 / 2, 5 / -2
(2.5, -2.5)

>>> 5 // 2, 5 // -2          # Truncates to floor: rounds to first lower integer
(2, -3)                     # 2.5 becomes 2, -2.5 becomes -3

>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)
```

```
>>> 5 // 2.0, 5 // -2.0      # Ditto for floats, though result is float too
(2.0, -3.0)
```

The 2.6 case is similar, but / results differ again:

```
C:\misc> c:\python26\python
>>> 5 / 2, 5 / -2           # Differs in 3.0
(2, -3)

>>> 5 // 2, 5 // -2        # This and the rest are the same in 2.6 and 3.0
(2, -3)

>>> 5 / 2.0, 5 / -2.0
(2.5, -2.5)

>>> 5 // 2.0, 5 // -2.0
(2.0, -3.0)
```

If you really want truncation regardless of sign, you can always run a float division result through `math.trunc`, regardless of Python version (also see the `round` built-in for related functionality):

```
C:\misc> c:\python30\python
>>> import math
>>> 5 / -2                  # Keep remainder
-2.5
>>> 5 // -2                 # Floor below result
-3
>>> math.trunc(5 / -2)      # Truncate instead of floor
-2

C:\misc> c:\python26\python
>>> import math
>>> 5 / float(-2)           # Remainder in 2.6
-2.5
>>> 5 / -2, 5 // -2         # Floor in 2.6
(-3, -3)
>>> math.trunc(5 / float(-2)) # Truncate in 2.6
-2
```

Why does truncation matter?

If you are using 3.0, here is the short story on division operators for reference:

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)      # 3.0 true division
(2.5, 2.5, -2.5, -2.5)

>>> (5 // 2), (5 // 2.0), (5 // -2.0), (5 // -2)  # 3.0 floor division
(2, 2.0, -3.0, -3)

>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)      # Both
(3.0, 3.0, 3, 3.0)
```

For 2.6 readers, division works as follows:

```
>>> (5 / 2), (5 / 2.0), (5 / -2.0), (5 / -2)      # 2.6 classic division
(2, 2.5, -2.5, -3)
```


Complex Numbers

Although less widely used than the types we've been exploring thus far, complex numbers are a distinct core object type in Python. If you know what they are, you know why they are useful; if not, consider this section optional reading.

Complex numbers are represented as two floating-point numbers—the real and imaginary parts—and are coded by adding a `j` or `J` suffix to the imaginary part. We can also write complex numbers with a nonzero real part by adding the two parts with a `+`. For example, the complex number with a real part of 2 and an imaginary part of `-3j` is written `2 + -3j`. Here are some examples of complex math at work:

```
>>> 1j * 1j
(-1+0j)
>>> 2 + 1j * 3
(2+3j)
>>> (2 + 1j) * 3
(6+3j)
```

Complex numbers also allow us to extract their parts as attributes, support all the usual mathematical expressions, and may be processed with tools in the standard `cmath` module (the complex version of the standard `math` module). Complex numbers typically find roles in engineering-oriented programs. Because they are advanced tools, check Python's language reference manual for additional details.

Hexadecimal, Octal, and Binary Notation

As described earlier in this chapter, Python integers can be coded in hexadecimal, octal, and binary notation, in addition to the normal base 10 decimal coding. The coding rules were laid out at the start of this chapter; let's look at some live examples here.

Keep in mind that these literals are simply an alternative syntax for specifying the value of an integer object. For example, the following literals coded in Python 3.0 or 2.6 produce normal integers with the specified values in all three bases:

```
>>> 0o1, 0o20, 0o377          # Octal literals
(1, 16, 255)
>>> 0x01, 0x10, 0xFF          # Hex literals
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111  # Binary literals
(1, 16, 255)
```

Here, the octal value `0o377`, the hex value `0xFF`, and the binary value `0b11111111` are all decimal 255. Python prints in decimal (base 10) by default but provides built-in functions that allow you to convert integers to other bases' digit strings:

```
>>> oct(64), hex(64), bin(64)
('0100', '0x40', '0b1000000')
```


The `oct` function converts decimal to octal, `hex` to hexadecimal, and `bin` to binary. To go the other way, the built-in `int` function converts a string of digits to an integer, and an optional second argument lets you specify the numeric base:

```
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)

>>> int('0x40', 16), int('0b1000000', 2)    # Literals okay too
(64, 64)
```

The `eval` function, which you'll meet later in this book, treats strings as though they were Python code. Therefore, it has a similar effect (but usually runs more slowly—it actually compiles and runs the string as a piece of a program, and it assumes you can trust the source of the string being run; a clever user might be able to submit a string that deletes files on your machine!):

```
>>> eval('64'), eval('0o100'), eval('0x40'), eval('0b1000000')
(64, 64, 64, 64)
```

Finally, you can also convert integers to octal and hexadecimal strings with *string formatting* method calls and expressions:

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64)
'100, 40, 1000000'

>>> '%o, %x, %X' % (64, 255, 255)
'100, ff, FF'
```

String formatting is covered in more detail in [Chapter 7](#).

Two notes before moving on. First, Python 2.6 users should remember that you can code octals with simply a leading zero, the original octal format in Python:

```
>>> 0o1, 0o20, 0o377    # New octal format in 2.6 (same as 3.0)
(1, 16, 255)
>>> 01, 020, 0377      # Old octal literals in 2.6 (and earlier)
(1, 16, 255)
```

In 3.0, the syntax in the second of these examples generates an error. Even though it's not an error in 2.6, be careful not to begin a string of digits with a leading zero unless you really mean to code an octal value. Python 2.6 will treat it as base 8, which may not work as you'd expect—`010` is always decimal 8 in 2.6, not decimal 10 (despite what you may or may not think!). This, along with symmetry with the hex and binary forms, is why the octal format was changed in 3.0—you must use `0o010` in 3.0, and probably should in 2.6.

Secondly, note that these literals can produce arbitrarily long integers. The following, for instance, creates an integer with hex notation and then displays it first in decimal and then in octal and binary with converters:

```
>>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF
>>> X
5192296858534827628530496329220095L
>>> oct(X)
```


We won't go into much more detail on "bit-twiddling" here. It's supported if you need it, and it comes in handy if your Python code must deal with things like network packets or packed binary data produced by a C program. Be aware, though, that bitwise operations are often not as important in a high-level language such as Python as they are in a low-level language such as C. As a rule of thumb, if you find yourself wanting to flip bits in Python, you should think about which language you're really coding. In general, there are often better ways to encode information in Python than bit strings.



In the upcoming Python 3.1 release, the integer `bit_length` method also allows you to query the number of bits required to represent a number's value in binary. The same effect can often be achieved by subtracting 2 from the length of the `bin` string using the `len` built-in function we met in [Chapter 4](#), though it may be less efficient:

```
>>> X = 99
>>> bin(X), X.bit_length()
('0b1100011', 7)
>>> bin(256), (256).bit_length()
('0b100000000', 9)
>>> len(bin(256)) - 2
9
```

Other Built-in Numeric Tools

In addition to its core object types, Python also provides both built-in functions and standard library modules for numeric processing. The `pow` and `abs` built-in functions, for instance, compute powers and absolute values, respectively. Here are some examples of the built-in `math` module (which contains most of the tools in the C language's math library) and a few built-in functions at work:

```
>>> import math
>>> math.pi, math.e                                     # Common constants
(3.1415926535897931, 2.7182818284590451)

>>> math.sin(2 * math.pi / 180)                         # Sine, tangent, cosine
0.034899496702500969

>>> math.sqrt(144), math.sqrt(2)                       # Square root
(12.0, 1.4142135623730951)

>>> pow(2, 4), 2 ** 4                                    # Exponentiation (power)
(16, 16)

>>> abs(-42.0), sum((1, 2, 3, 4))                      # Absolute value, summation
(42.0, 10)

>>> min(3, 1, 2, 4), max(3, 1, 2, 4)                   # Minimum, maximum
(1, 4)
```

The `sum` function shown here works on a sequence of numbers, and `min` and `max` accept either a sequence or individual arguments. There are a variety of ways to drop the

decimal digits of floating-point numbers. We met truncation and floor earlier; we can also round, both numerically and for display purposes:

```
>>> math.floor(2.567), math.floor(-2.567)      # Floor (next-lower integer)
(2, -3)

>>> math.trunc(2.567), math.trunc(-2.567)      # Truncate (drop decimal digits)
(2, -2)

>>> int(2.567), int(-2.567)                    # Truncate (integer conversion)
(2, -2)

>>> round(2.567), round(2.467), round(2.567, 2) # Round (Python 3.0 version)
(3, 2, 2.5699999999999998)

>>> '%.1f' % 2.567, '{0:.2f}'.format(2.567)     # Round for display (Chapter 7)
('2.6', '2.57')
```

As we saw earlier, the last of these produces strings that we would usually print and supports a variety of formatting options. As also described earlier, the second to last test here will output (3, 2, 2.57) if we wrap it in a `print` call to request a more user-friendly display. The last two lines still differ, though—`round` rounds a floating-point number but still yields a floating-point number in memory, whereas string formatting produces a string and doesn't yield a modified number:

```
>>> (1 / 3), round(1 / 3, 2), ('%.2f' % (1 / 3))
(0.33333333333333331, 0.33000000000000002, '0.33')
```

Interestingly, there are three ways to compute *square roots* in Python: using a module function, an expression, or a built-in function (if you're interested in performance, we will revisit these in an exercise and its solution at the end of [Part IV](#), to see which runs quicker):

```
>>> import math
>>> math.sqrt(144)                               # Module
12.0
>>> 144 ** .5                                     # Expression
12.0
>>> pow(144, .5)                                  # Built-in
12.0

>>> math.sqrt(1234567890)                         # Larger numbers
35136.418286444619
>>> 1234567890 ** .5
35136.418286444619
>>> pow(1234567890, .5)
35136.418286444619
```

Notice that standard library modules such as `math` must be imported, but built-in functions such as `abs` and `round` are always available without imports. In other words, modules are external components, but built-in functions live in an implied namespace that Python automatically searches to find names used in your program. This namespace corresponds to the module called `builtins` in Python 3.0 (`__builtin__` in 2.6). There

is much more about name resolution in the function and module parts of this book; for now, when you hear “module,” think “import.”

The standard library `random` module must be imported as well. This module provides tools for picking a random floating-point number between 0 and 1, selecting a random integer between two numbers, choosing an item at random from a sequence, and more:

```
>>> import random
>>> random.random()
0.44694718823781876
>>> random.random()
0.28970426439292829

>>> random.randint(1, 10)
5
>>> random.randint(1, 10)
4

>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Life of Brian'
>>> random.choice(['Life of Brian', 'Holy Grail', 'Meaning of Life'])
'Holy Grail'
```

The `random` module can be useful for shuffling cards in games, picking images at random in a slideshow GUI, performing statistical simulations, and much more. For more details, see Python’s library manual.

Other Numeric Types

So far in this chapter, we’ve been using Python’s core numeric types—integer, floating point, and complex. These will suffice for most of the number crunching that most programmers will ever need to do. Python comes with a handful of more exotic numeric types, though, that merit a quick look here.

Decimal Type

Python 2.4 introduced a new core numeric type: the decimal object, formally known as `Decimal`. Syntactically, decimals are created by calling a function within an imported module, rather than running a literal expression. Functionally, decimals are like floating-point numbers, but they have a fixed number of decimal points. Hence, decimals are *fixed-precision* floating-point values.

For example, with decimals, we can have a floating-point value that always retains just two decimal digits. Furthermore, we can specify how to round or truncate the extra decimal digits beyond the object’s cutoff. Although it generally incurs a small performance penalty compared to the normal floating-point type, the decimal type is well suited to representing fixed-precision quantities like sums of money and can help you achieve better numeric accuracy.

The basics

The last point merits elaboration. As you may or may not already know, floating-point math is less than exact, because of the limited space used to store values. For example, the following should yield zero, but it does not. The result is close to zero, but there are not enough bits to be precise here:

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
```

Printing the result to produce the user-friendly display format doesn't completely help either, because the hardware related to floating-point math is inherently limited in terms of accuracy:

```
>>> print(0.1 + 0.1 + 0.1 - 0.3)
5.55111512313e-17
```

However, with decimals, the result can be dead-on:

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

As shown here, we can make decimal objects by calling the `Decimal` constructor function in the `decimal` module and passing in strings that have the desired number of decimal digits for the resulting object (we can use the `str` function to convert floating-point values to strings if needed). When decimals of different precision are mixed in expressions, Python converts up to the largest number of decimal digits automatically:

```
>>> Decimal('0.1') + Decimal('0.10') + Decimal('0.10') - Decimal('0.30')
Decimal('0.00')
```



In Python 3.1 (to be released after this book's publication), it's also possible to create a decimal object from a floating-point object, with a call of the form `decimal.Decimal.from_float(1.25)`. The conversion is exact but can sometimes yield a large number of digits.

Setting precision globally

Other tools in the `decimal` module can be used to set the precision of all decimal numbers, set up error handling, and more. For instance, a context object in this module allows for specifying precision (number of decimal digits) and rounding modes (down, ceiling, etc.). The precision is applied globally for all decimals created in the calling thread:

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1428571428571428571428571429')

>>> decimal.getcontext().prec = 4
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
```

This is especially useful for monetary applications, where cents are represented as two decimal digits. Decimals are essentially an alternative to manual rounding and string formatting in this context:

```
>>> 1999 + 1.33
2000.3299999999999
>>>
>>> decimal.getcontext().prec = 2
>>> pay = decimal.Decimal(str(1999 + 1.33))
>>> pay
Decimal('2000.33')
```

Decimal context manager

In Python 2.6 and 3.0 (and later), it's also possible to reset precision temporarily by using the `with` context manager statement. The precision is reset to its original value on statement exit:

```
C:\misc> C:\Python30\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.333333333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
...     ctx.prec = 2
...     decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.333333333333333333333333333333')
```

Though useful, this statement requires much more background knowledge than you've obtained at this point; watch for coverage of the `with` statement in [Chapter 33](#).

Because use of the decimal type is still relatively rare in practice, I'll defer to Python's standard library manuals and interactive help for more details. And because decimals address some of the same floating-point accuracy issues as the fraction type, let's move on to the next section to see how the two compare.

Fraction Type

Python 2.6 and 3.0 debut a new numeric type, `Fraction`, which implements a *rational number* object. It essentially keeps both a numerator and a denominator explicitly, so as to avoid some of the inaccuracies and limitations of floating-point math.

The basics

`Fraction` is a sort of cousin to the existing `Decimal` fixed-precision type described in the prior section, as both can be used to control numerical accuracy by fixing decimal digits and specifying rounding or truncation policies. It's also used in similar ways—like

Decimal, Fraction resides in a module; import its constructor and pass in a numerator and a denominator to make one. The following interaction shows how:

```
>>> from fractions import Fraction
>>> x = Fraction(1, 3)           # Numerator, denominator
>>> y = Fraction(4, 6)          # Simplified to 2, 3 by gcd

>>> x
Fraction(1, 3)
>>> y
Fraction(2, 3)
>>> print(y)
2/3
```

Once created, Fractions can be used in mathematical expressions as usual:

```
>>> x + y
Fraction(1, 1)
>>> x - y
Fraction(-1, 3)
>>> x * y
Fraction(2, 9)
```

Results are exact: numerator, denominator

Fraction objects can also be created from floating-point number strings, much like decimals:

```
>>> Fraction('.25')
Fraction(1, 4)
>>> Fraction('1.25')
Fraction(5, 4)
>>>
>>> Fraction('.25') + Fraction('1.25')
Fraction(3, 2)
```

Numeric accuracy

Notice that this is different from floating-point-type math, which is constrained by the underlying limitations of floating-point hardware. To compare, here are the same operations run with floating-point objects, and notes on their limited accuracy:

```
>>> a = 1 / 3.0           # Only as accurate as floating-point hardware
>>> b = 4 / 6.0           # Can lose precision over calculations
>>> a
0.3333333333333333
>>> b
0.6666666666666666

>>> a + b
1.0
>>> a - b
-0.3333333333333333
>>> a * b
0.2222222222222222
```

This floating-point limitation is especially apparent for values that cannot be represented accurately given their limited number of bits in memory. Both Fraction and

Decimal provide ways to get exact results, albeit at the cost of some speed. For instance, in the following example (repeated from the prior section), floating-point numbers do not accurately give the zero answer expected, but both of the other types do:

```
>>> 0.1 + 0.1 + 0.1 - 0.3                # This should be zero (close, but not exact)
5.5511151231257827e-17

>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)

>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') - Decimal('0.3')
Decimal('0.0')
```

Moreover, fractions and decimals both allow more intuitive and accurate results than floating points sometimes can, in different ways (by using rational representation and by limiting precision):

```
>>> 1 / 3                                # Use 3.0 in Python 2.6 for true "/"
0.33333333333333331

>>> Fraction(1, 3)                       # Numeric accuracy
Fraction(1, 3)

>>> import decimal
>>> decimal.getcontext().prec = 2
>>> decimal.Decimal(1) / decimal.Decimal(3)
Decimal('0.33')
```

In fact, fractions both retain accuracy and automatically simplify results. Continuing the preceding interaction:

```
>>> (1 / 3) + (6 / 12)                   # Use ".0" in Python 2.6 for true "/"
0.83333333333333326

>>> Fraction(6, 12)                      # Automatically simplified
Fraction(1, 2)

>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)

>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
Decimal('0.83')

>>> 1000.0 / 1234567890
8.1000000737100011e-07
>>> Fraction(1000, 1234567890)
Fraction(100, 123456789)
```

Conversions and mixed types

To support fraction conversions, floating-point objects now have a method that yields their numerator and denominator ratio, fractions have a `from_float` method, and

`float` accepts a `Fraction` as an argument. Trace through the following interaction to see how this pans out (the `*` in the second test is special syntax that expands a tuple into individual arguments; more on this when we study function argument passing in [Chapter 18](#)):

```
>>> (2.5).as_integer_ratio()           # float object method
(5, 2)

>>> f = 2.5
>>> z = Fraction(*f.as_integer_ratio()) # Convert float -> fraction: two args
>>> z                                     # Same as Fraction(5, 2)
Fraction(5, 2)

>>> x                                     # x from prior interaction
Fraction(1, 3)
>>> x + z
Fraction(17, 6)                          # 5/2 + 1/3 = 15/6 + 2/6

>>> float(x)                             # Convert fraction -> float
0.3333333333333333
>>> float(z)
2.5
>>> float(x + z)
2.8333333333333335
>>> 17 / 6
2.8333333333333335

>>> Fraction.from_float(1.75)            # Convert float -> fraction: other way
Fraction(7, 4)
>>> Fraction(*(1.75).as_integer_ratio())
Fraction(7, 4)
```

Finally, some type mixing is allowed in expressions, though `Fraction` must sometimes be manually propagated to retain accuracy. Study the following interaction to see how this works:

```
>>> x
Fraction(1, 3)
>>> x + 2                                # Fraction + int -> Fraction
Fraction(7, 3)
>>> x + 2.0                              # Fraction + float -> float
2.3333333333333335
>>> x + (1./3)                           # Fraction + float -> float
0.6666666666666666

>>> x + (4./3)
1.6666666666666665
>>> x + Fraction(4, 3)                   # Fraction + Fraction -> Fraction
Fraction(5, 3)
```

Caveat: although you can convert from floating-point to fraction, in some cases there is an unavoidable precision loss when you do so, because the number is inaccurate in its original floating-point form. When needed, you can simplify such results by limiting the maximum denominator value:

```

>>> 4.0 / 3
1.3333333333333333
>>> (4.0 / 3).as_integer_ratio()           # Precision loss from float
(6004799503160661, 4503599627370496)

>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)

>>> 22517998136852479 / 13510798882111488.   # 5 / 3 (or close to it!)
1.6666666666666667

>>> a.limit_denominator(10)                 # Simplify to closest fraction
Fraction(5, 3)

```

For more details on the `Fraction` type, experiment further on your own and consult the Python 2.6 and 3.0 library manuals and other documentation.

Sets

Python 2.4 also introduced a new collection type, the *set*—an unordered collection of unique and immutable objects that supports operations corresponding to mathematical set theory. By definition, an item appears only once in a set, no matter how many times it is added. As such, sets have a variety of applications, especially in numeric and database-focused work.

Because sets are collections of other objects, they share some behavior with objects such as lists and dictionaries that are outside the scope of this chapter. For example, sets are iterable, can grow and shrink on demand, and may contain a variety of object types. As we'll see, a set acts much like the keys of a valueless dictionary, but it supports extra operations.

However, because sets are unordered and do not map keys to values, they are neither sequence nor mapping types; they are a type category unto themselves. Moreover, because sets are fundamentally mathematical in nature (and for many readers, may seem more academic and be used much less often than more pervasive objects like dictionaries), we'll explore the basic utility of Python's set objects here.

Set basics in Python 2.6

There are a few ways to make sets today, depending on whether you are using Python 2.6 or 3.0. Since this book covers both, let's begin with the 2.6 case, which also is available (and sometimes still required) in 3.0; we'll refine this for 3.0 extensions in a moment. To make a set object, pass in a sequence or other iterable object to the built-in `set` function:

```

>>> x = set('abcde')
>>> y = set('bdxyz')

```

You get back a set object, which contains all the items in the object passed in (notice that sets do not have a positional ordering, and so are not sequences):

```
>>> x
set(['a', 'c', 'b', 'e', 'd'])          # 2.6 display format
```

Sets made this way support the common mathematical set operations with *expression* operators. Note that we can't perform these expressions on plain sequences—we must create sets from them in order to apply these tools:

```
>>> 'e' in x                            # Membership
True

>>> x - y                               # Difference
set(['a', 'c', 'e'])

>>> x | y                               # Union
set(['a', 'c', 'b', 'e', 'd', 'y', 'x', 'z'])

>>> x & y                               # Intersection
set(['b', 'd'])

>>> x ^ y                               # Symmetric difference (XOR)
set(['a', 'c', 'e', 'y', 'x', 'z'])

>>> x > y, x < y                        # Superset, subset
(False, False)
```

In addition to expressions, the set object provides *methods* that correspond to these operations and more, and that support set changes—the set **add** method inserts one item, **update** is an in-place union, and **remove** deletes an item by value (run a **dir** call on any set instance or the **set** type name to see all the available methods). Assuming **x** and **y** are still as they were in the prior interaction:

```
>>> z = x.intersection(y)              # Same as x & y
>>> z
set(['b', 'd'])
>>> z.add('SPAM')                      # Insert one item
>>> z
set(['b', 'd', 'SPAM'])
>>> z.update(set(['X', 'Y']))          # Merge: in-place union
>>> z
set(['Y', 'X', 'b', 'd', 'SPAM'])
>>> z.remove('b')                     # Delete one item
>>> z
set(['Y', 'X', 'd', 'SPAM'])
```

As *iterable* containers, sets can also be used in operations such as **len**, **for** loops, and list comprehensions. Because they are unordered, though, they don't support sequence operations like indexing and slicing:

```
>>> for item in set('abc'): print(item * 3)
...
aaa
```

```
ccc  
bbb
```

Finally, although the set expressions shown earlier generally require two sets, their method-based counterparts can often work with *any iterable type* as well:

```
>>> S = set([1, 2, 3])  
  
>>> S | set([3, 4])          # Expressions require both to be sets  
set([1, 2, 3, 4])  
>>> S | [3, 4]  
TypeError: unsupported operand type(s) for |: 'set' and 'list'  
  
>>> S.union([3, 4])          # But their methods allow any iterable  
set([1, 2, 3, 4])  
>>> S.intersection([1, 3, 5])  
set([1, 3])  
>>> S.issubset(range(-5, 5))  
True
```

For more details on set operations, see Python’s library reference manual or a reference book. Although set operations can be coded manually in Python with other types, like lists and dictionaries (and often were in the past), Python’s built-in sets use efficient algorithms and implementation techniques to provide quick and standard operation.

Set literals in Python 3.0

If you think sets are “cool,” they recently became noticeably cooler. In Python 3.0 we can still use the `set` built-in to make set objects, but 3.0 also adds a new set literal form, using the curly braces formerly reserved for dictionaries. In 3.0, the following are equivalent:

```
set([1, 2, 3, 4])          # Built-in call  
{1, 2, 3, 4}              # 3.0 set literals
```

This syntax makes sense, given that sets are essentially like *valueless dictionaries*—because they are unordered, unique, and immutable, a set’s items behave much like a dictionary’s keys. This operational similarity is even more striking given that dictionary key lists in 3.0 are *view* objects, which support set-like behavior such as intersections and unions (see [Chapter 8](#) for more on dictionary view objects).

In fact, regardless of how a set is made, 3.0 displays it using the new literal format. The `set` built-in is still required in 3.0 to create empty sets and to build sets from existing iterable objects (short of using set comprehensions, discussed later in this chapter), but the new literal is convenient for initializing sets of known structure:

```
C:\Misc> c:\python30\python  
>>> set([1, 2, 3, 4])        # Built-in: same as in 2.6  
{1, 2, 3, 4}  
>>> set('spam')             # Add all items in an iterable  
{ 'a', 'p', 's', 'm' }  
  
>>> {1, 2, 3, 4}            # Set literals: new in 3.0
```

```

{1, 2, 3, 4}
>>> S = {'s', 'p', 'a', 'm'}
>>> S.add('alot')
>>> S
{'a', 'p', 's', 'm', 'alot'}

```

All the set processing operations discussed in the prior section work the same in 3.0, but the result sets print differently:

```

>>> S1 = {1, 2, 3, 4}
>>> S1 & {1, 3}           # Intersection
{1, 3}
>>> {1, 5, 3, 6} | S1    # Union
{1, 2, 3, 4, 5, 6}
>>> S1 - {1, 3, 4}       # Difference
{2}
>>> S1 > {1, 3}          # Superset
True

```

Note that `{}` is still a dictionary in Python. *Empty* sets must be created with the `set` built-in, and print the same way:

```

>>> S1 = {1, 2, 3, 4}    # Empty sets print differently
set()
>>> type({})             # Because {} is an empty dictionary
<class 'dict'>

>>> S = set()            # Initialize an empty set
>>> S.add(1.23)
>>> S
{1.23}

```

As in Python 2.6, sets created with 3.0 literals support the same methods, some of which allow general iterable operands that expressions do not:

```

>>> {1, 2, 3} | {3, 4}
{1, 2, 3, 4}
>>> {1, 2, 3} | [3, 4]
TypeError: unsupported operand type(s) for |: 'set' and 'list'

>>> {1, 2, 3}.union([3, 4])
{1, 2, 3, 4}
>>> {1, 2, 3}.union({3, 4})
{1, 2, 3, 4}
>>> {1, 2, 3}.union(set([3, 4]))
{1, 2, 3, 4}

>>> {1, 2, 3}.intersection((1, 3, 5))
{1, 3}
>>> {1, 2, 3}.issubset(range(-5, 5))
True

```

Immutable constraints and frozen sets

Sets are powerful and flexible objects, but they do have one constraint in both 3.0 and 2.6 that you should keep in mind—largely because of their implementation, sets can

only contain immutable (a.k.a “hashable”) object types. Hence, lists and dictionaries cannot be embedded in sets, but tuples can if you need to store compound values. Tuples compare by their full values when used in set operations:

```
>>> S
{1.23}
>>> S.add([1, 2, 3])
TypeError: unhashable type: 'list'
>>> S.add({'a':1})
TypeError: unhashable type: 'dict'
>>> S.add((1, 2, 3))
>>> S
{1.23, (1, 2, 3)}

>>> S | {(4, 5, 6), (1, 2, 3)}
{1.23, (4, 5, 6), (1, 2, 3)}
>>> (1, 2, 3) in S
True
>>> (1, 4, 3) in S
False
```

Only mutable objects work in a set

No list or dict, but tuple okay

Union: same as S.union(...)

Membership: by complete values

Tuples in a set, for instance, might be used to represent dates, records, IP addresses, and so on (more on tuples later in this part of the book). Sets themselves are mutable too, and so cannot be nested in other sets directly; if you need to store a set inside another set, the `frozenset` built-in call works just like `set` but creates an immutable set that cannot change and thus can be embedded in other sets.

Set comprehensions in Python 3.0

In addition to literals, 3.0 introduces a set comprehension construct; it is similar in form to the list comprehension we previewed in [Chapter 4](#), but is coded in curly braces instead of square brackets and run to make a set instead of a list. Set comprehensions run a loop and collect the result of an expression on each iteration; a loop variable gives access to the current iteration value for use in the collection expression. The result is a new set created by running the code, with all the normal set behavior:

```
>>> {x ** 2 for x in [1, 2, 3, 4]}
{16, 1, 4, 9}
```

3.0 set comprehension

In this expression, the loop is coded on the right, and the collection expression is coded on the left (`x ** 2`). As for list comprehensions, we get back pretty much what this expression says: “Give me a new set containing X squared, for every X in a list.” Comprehensions can also iterate across other kinds of objects, such as strings (the first of the following examples illustrates the comprehension-based way to make a set from an existing iterable):

```
>>> {x for x in 'spam'}
{'a', 'p', 's', 'm'}

>>> {c * 4 for c in 'spam'}
{'ssss', 'aaaa', 'pppp', 'mmmm'}
>>> {c * 4 for c in 'spamham'}
```

Same as: set('spam')

Set of collected expression results

```
{'ssss', 'aaaa', 'hhhh', 'pppp', 'mmmm'}

>>> S = {c * 4 for c in 'spam'}
>>> S | {'mmmm', 'xxxx'}
{'ssss', 'aaaa', 'pppp', 'mmmm', 'xxxx'}
>>> S & {'mmmm', 'xxxx'}
{'mmmm'}
```

Because the rest of the comprehensions story relies upon underlying concepts we're not yet prepared to address, we'll postpone further details until later in this book. In [Chapter 8](#), we'll meet a first cousin in 3.0, the dictionary comprehension, and I'll have much more to say about all comprehensions (list, set, dictionary, and generator) later, especially in [Chapters 14](#) and [20](#). As we'll learn later, all comprehensions, including sets, support additional syntax not shown here, including nested loops and `if` tests, which can be difficult to understand until you've had a chance to study larger statements.

Why sets?

Set operations have a variety of common uses, some more practical than mathematical. For example, because items are stored only once in a set, sets can be used to filter duplicates out of other collections. Simply convert the collection to a set, and then convert it back again (because sets are iterable, they work in the `list` call here):

```
>>> L = [1, 2, 1, 3, 2, 4, 5]
>>> set(L)
{1, 2, 3, 4, 5}
>>> L = list(set(L))                                # Remove duplicates
>>> L
[1, 2, 3, 4, 5]
```

Sets can also be used to keep track of where you've already been when traversing a graph or other cyclic structure. For example, the transitive module reloader and inheritance tree lister examples we'll study in [Chapters 24](#) and [30](#), respectively, must keep track of items visited to avoid loops. Although recording states visited as keys in a dictionary is efficient, sets offer an alternative that's essentially equivalent (and may be more or less intuitive, depending on who you ask).

Finally, sets are also convenient when dealing with large data sets (database query results, for example)—the intersection of two sets contains objects in common to both categories, and the union contains all items in either set. To illustrate, here's a somewhat more realistic example of set operations at work, applied to lists of people in a hypothetical company, using 3.0 set literals (use `set` in 2.6):

```
>>> engineers = {'bob', 'sue', 'ann', 'vic'}
>>> managers = {'tom', 'sue'}

>>> 'bob' in engineers                                # Is bob an engineer?
True

>>> engineers & managers                              # Who is both engineer and manager?
```



```

{'sue'}

>>> engineers | managers          # All people in either category
{'vic', 'sue', 'tom', 'bob', 'ann'}

>>> engineers - managers          # Engineers who are not managers
{'vic', 'bob', 'ann'}

>>> managers - engineers          # Managers who are not engineers
{'tom'}

>>> engineers > managers          # Are all managers engineers? (superset)
False

>>> {'bob', 'sue'} < engineers    # Are both engineers? (subset)
True

>>> (managers | engineers) > managers  # All people is a superset of managers
True

>>> managers ^ engineers          # Who is in one but not both?
{'vic', 'bob', 'ann', 'tom'}

>>> (managers | engineers) - (managers ^ engineers)  # Intersection!
{'sue'}

```

You can find more details on set operations in the Python library manual and some mathematical and relational database theory texts. Also stay tuned for [Chapter 8](#)'s revival of some of the set operations we've seen here, in the context of dictionary view objects in Python 3.0.

Booleans

Some argue that the Python Boolean type, `bool`, is numeric in nature because its two values, `True` and `False`, are just customized versions of the integers 1 and 0 that print themselves differently. Although that's all most programmers need to know, let's explore this type in a bit more detail.

More formally, Python today has an explicit Boolean data type called `bool`, with the values `True` and `False` available as new preassigned built-in names. Internally, the names `True` and `False` are instances of `bool`, which is in turn just a subclass (in the object-oriented sense) of the built-in integer type `int`. `True` and `False` behave exactly like the integers 1 and 0, except that they have customized printing logic—they print themselves as the words `True` and `False`, instead of the digits 1 and 0. `bool` accomplishes this by redefining `str` and `repr` string formats for its two objects.

Because of this customization, the output of Boolean expressions typed at the interactive prompt prints as the words `True` and `False` instead of the older and less obvious 1 and 0. In addition, Booleans make truth values more explicit. For instance, an infinite loop can now be coded as `while True:` instead of the less intuitive `while 1:`. Similarly,

flags can be initialized more clearly with `flag = False`. We'll discuss these statements further in [Part III](#).

Again, though, for all other practical purposes, you can treat `True` and `False` as though they are predefined variables set to integer 1 and 0. Most programmers used to preassign `True` and `False` to 1 and 0 anyway; the `bool` type simply makes this standard. Its implementation can lead to curious results, though. Because `True` is just the integer 1 with a custom display format, `True + 4` yields 5 in Python:

```
>>> type(True)
<class 'bool'>
>>> isinstance(True, int)
True
>>> True == 1           # Same value
True
>>> True is 1           # But different object: see the next chapter
False
>>> True or False       # Same as: 1 or 0
True
>>> True + 4            # (Hmmm)
5
```

Since you probably won't come across an expression like the last of these in real Python code, you can safely ignore its deeper metaphysical implications....

We'll revisit Booleans in [Chapter 9](#) (to define Python's notion of truth) and again in [Chapter 12](#) (to see how Boolean operators like `and` and `or` work).

Numeric Extensions

Finally, although Python core numeric types offer plenty of power for most applications, there is a large library of third-party open source extensions available to address more focused needs. Because numeric programming is a popular domain for Python, you'll find a wealth of advanced tools.

For example, if you need to do serious number crunching, an optional extension for Python called *NumPy* (Numeric Python) provides advanced numeric programming tools, such as a matrix data type, vector processing, and sophisticated computation libraries. Hardcore scientific programming groups at places like Los Alamos and NASA use Python with NumPy to implement the sorts of tasks they previously coded in C++, FORTRAN, or Matlab. The combination of Python and NumPy is often compared to a free, more flexible version of Matlab—you get NumPy's performance, plus the Python language and its libraries.

Because it's so advanced, we won't talk further about NumPy in this book. You can find additional support for advanced numeric programming in Python, including graphics and plotting tools, statistics libraries, and the popular *SciPy* package at Python's PyPI site, or by searching the Web. Also note that NumPy is currently an optional extension; it doesn't come with Python and must be installed separately.

Chapter Summary

This chapter has taken a tour of Python's numeric object types and the operations we can apply to them. Along the way, we met the standard integer and floating-point types, as well as some more exotic and less commonly used types such as complex numbers, fractions, and sets. We also explored Python's expression syntax, type conversions, bitwise operations, and various literal forms for coding numbers in scripts.

Later in this part of the book, I'll fill in some details about the next object type, the string. In the next chapter, however, we'll take some time to explore the mechanics of variable assignment in more detail than we have here. This turns out to be perhaps the most fundamental idea in Python, so make sure you check out the next chapter before moving on. First, though, it's time to take the usual chapter quiz.

Test Your Knowledge: Quiz

1. What is the value of the expression `2 * (3 + 4)` in Python?
2. What is the value of the expression `2 * 3 + 4` in Python?
3. What is the value of the expression `2 + 3 * 4` in Python?
4. What tools can you use to find a number's square root, as well as its square?
5. What is the type of the result of the expression `1 + 2.0 + 3`?
6. How can you truncate and round a floating-point number?
7. How can you convert an integer to a floating-point number?
8. How would you display an integer in octal, hexadecimal, or binary notation?
9. How might you convert an octal, hexadecimal, or binary string to a plain integer?

Test Your Knowledge: Answers

1. The value will be **14**, the result of `2 * 7`, because the parentheses force the addition to happen before the multiplication.
2. The value will be **10**, the result of `6 + 4`. Python's operator precedence rules are applied in the absence of parentheses, and multiplication has higher precedence than (i.e., happens before) addition, per [Table 5-2](#).
3. This expression yields **14**, the result of `2 + 12`, for the same precedence reasons as in the prior question.
4. Functions for obtaining the square root, as well as *pi*, tangents, and more, are available in the imported `math` module. To find a number's square root, import `math` and call `math.sqrt(N)`. To get a number's square, use either the exponent

expression `X ** 2` or the built-in function `pow(X, 2)`. Either of these last two can also compute the square root when given a power of `0.5` (e.g., `X ** .5`).

5. The result will be a floating-point number: the integers are converted up to floating point, the most complex type in the expression, and floating-point math is used to evaluate it.
6. The `int(N)` and `math.trunc(N)` functions truncate, and the `round(N, digits)` function rounds. We can also compute the floor with `math.floor(N)` and round for display with string formatting operations.
7. The `float(I)` function converts an integer to a floating point; mixing an integer with a floating point within an expression will result in a conversion as well. In some sense, Python 3.0 / division converts too—it always returns a floating-point result that includes the remainder, even if both operands are integers.
8. The `oct(I)` and `hex(I)` built-in functions return the octal and hexadecimal string forms for an integer. The `bin(I)` call also returns a number's binary digits string in Python 2.6 and 3.0. The `%` string formatting expression and `format` string method also provide targets for some such conversions.
9. The `int(S, base)` function can be used to convert from octal and hexadecimal strings to normal integers (pass in `8`, `16`, or `2` for the base). The `eval(S)` function can be used for this purpose too, but it's more expensive to run and can have security issues. Note that integers are always stored in binary in computer memory; these are just display string format conversions.

The Dynamic Typing Interlude

In the prior chapter, we began exploring Python’s core object types in depth with a look at Python numbers. We’ll resume our object type tour in the next chapter, but before we move on, it’s important that you get a handle on what may be the most fundamental idea in Python programming and is certainly the basis of much of both the conciseness and flexibility of the Python language—dynamic typing, and the polymorphism it yields.

As you’ll see here and later in this book, in Python, we do not declare the specific types of the objects our scripts use. In fact, programs should not even care about specific types; in exchange, they are naturally applicable in more contexts than we can sometimes even plan ahead for. Because dynamic typing is the root of this flexibility, let’s take a brief look at the model here.

The Case of the Missing Declaration Statements

If you have a background in compiled or statically typed languages like C, C++, or Java, you might find yourself a bit perplexed at this point in the book. So far, we’ve been using variables without declaring their existence or their types, and it somehow works. When we type `a = 3` in an interactive session or program file, for instance, how does Python know that `a` should stand for an integer? For that matter, how does Python know what `a` is at all?

Once you start asking such questions, you’ve crossed over into the domain of Python’s *dynamic typing* model. In Python, types are determined automatically at runtime, not in response to declarations in your code. This means that you never declare variables ahead of time (a concept that is perhaps simpler to grasp if you keep in mind that it all boils down to variables, objects, and the links between them).

Variables, Objects, and References

As you've seen in many of the examples used so far in this book, when you run an assignment statement such as `a = 3` in Python, it works even if you've never told Python to use the name `a` as a variable, or that `a` should stand for an integer-type object. In the Python language, this all pans out in a very natural way, as follows:

Variable creation

A variable (i.e., name), like `a`, is created when your code first assigns it a value. Future assignments change the value of the already created name. Technically, Python detects some names before your code runs, but you can think of it as though initial assignments make variables.

Variable types

A variable never has any type information or constraints associated with it. The notion of type lives with objects, not names. Variables are generic in nature; they always simply refer to a particular object at a particular point in time.

Variable use

When a variable appears in an expression, it is immediately replaced with the object that it currently refers to, whatever that may be. Further, all variables must be explicitly assigned before they can be used; referencing unassigned variables results in errors.

In sum, variables are created when assigned, can reference any type of object, and must be assigned before they are referenced. This means that you never need to declare names used by your script, but you must initialize names before you can update them; counters, for example, must be initialized to zero before you can add to them.

This dynamic typing model is strikingly different from the typing model of traditional languages. When you are first starting out, the model is usually easier to understand if you keep clear the distinction between names and objects. For example, when we say this:

```
>>> a = 3
```

at least conceptually, Python will perform three distinct steps to carry out the request. These steps reflect the operation of all assignments in the Python language:

1. Create an object to represent the value 3.
2. Create the variable `a`, if it does not yet exist.
3. Link the variable `a` to the new object 3.

The net result will be a structure inside Python that resembles [Figure 6-1](#). As sketched, variables and objects are stored in different parts of memory and are associated by links (the link is shown as a pointer in the figure). Variables always link to objects and never to other variables, but larger objects may link to other objects (for instance, a list object has links to the objects it contains).

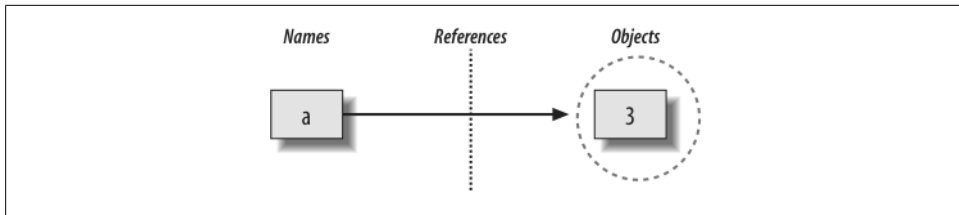


Figure 6-1. Names and objects after running the assignment `a = 3`. Variable `a` becomes a reference to the object 3. Internally, the variable is really a pointer to the object’s memory space created by running the literal expression `3`.

These links from variables to objects are called *references* in Python—that is, a reference is a kind of association, implemented as a pointer in memory.* Whenever the variables are later used (i.e., referenced), Python automatically follows the variable-to-object links. This is all simpler than the terminology may imply. In concrete terms:

- *Variables* are entries in a system table, with spaces for links to objects.
- *Objects* are pieces of allocated memory, with enough space to represent the values for which they stand.
- *References* are automatically followed pointers from variables to objects.

At least conceptually, each time you generate a new value in your script by running an expression, Python creates a new object (i.e., a chunk of memory) to represent that value. Internally, as an optimization, Python caches and reuses certain kinds of unchangeable objects, such as small integers and strings (each `0` is not really a new piece of memory—more on this caching behavior later). But, from a logical perspective, it works as though each expression’s result value is a distinct object and each object is a distinct piece of memory.

Technically speaking, objects have more structure than just enough space to represent their values. Each object also has two standard header fields: a *type designator* used to mark the type of the object, and a *reference counter* used to determine when it’s OK to reclaim the object. To understand how these two header fields factor into the model, we need to move on.

Types Live with Objects, Not Variables

To see how object types come into play, watch what happens if we assign a variable multiple times:

* Readers with a background in C may find Python references similar to C pointers (memory addresses). In fact, references are implemented as pointers, and they often serve the same roles, especially with objects that can be changed in-place (more on this later). However, because references are always automatically dereferenced when used, you can never actually do anything useful with a reference itself; this is a feature that eliminates a vast category of C bugs. You can think of Python references as C “void*” pointers, which are automatically followed whenever used.

```

>>> a = 3           # It's an integer
>>> a = 'spam'      # Now it's a string
>>> a = 1.23        # Now it's a floating point

```

This isn't typical Python code, but it does work—`a` starts out as an integer, then becomes a string, and finally becomes a floating-point number. This example tends to look especially odd to ex-C programmers, as it appears as though the *type* of `a` changes from integer to string when we say `a = 'spam'`.

However, that's not really what's happening. In Python, things work more simply. Names have no types; as stated earlier, types live with objects, not names. In the preceding listing, we've simply changed `a` to reference different objects. Because variables have no type, we haven't actually changed the type of the variable `a`; we've simply made the variable reference a different type of object. In fact, again, all we can ever say about a variable in Python is that it references a particular object at a particular point in time.

Objects, on the other hand, know what type they are—each object contains a header field that tags the object with its type. The integer object `3`, for example, will contain the value `3`, plus a designator that tells Python that the object is an integer (strictly speaking, a pointer to an object called `int`, the name of the integer type). The type designator of the `'spam'` string object points to the string type (called `str`) instead. Because objects know their types, variables don't have to.

To recap, types are associated with objects in Python, not with variables. In typical code, a given variable usually will reference just one kind of object. Because this isn't a requirement, though, you'll find that Python code tends to be much more flexible than you may be accustomed to—if you use Python well, your code might work on many types automatically.

I mentioned that objects have two header fields, a type designator and a reference counter. To understand the latter of these, we need to move on and take a brief look at what happens at the end of an object's life.

Objects Are Garbage-Collected

In the prior section's listings, we assigned the variable `a` to different types of objects in each assignment. But when we reassign a variable, what happens to the value it was previously referencing? For example, after the following statements, what happens to the object `3`?

```

>>> a = 3
>>> a = 'spam'

```

The answer is that in Python, whenever a name is assigned to a new object, the space held by the prior object is reclaimed (if it is not referenced by any other name or object). This automatic reclamation of objects' space is known as *garbage collection*.

To illustrate, consider the following example, which sets the name `x` to a different object on each assignment:


```
>>> x = 42
>>> x = 'shrubbery'           # Reclaim 42 now (unless referenced elsewhere)
>>> x = 3.1415                # Reclaim 'shrubbery' now
>>> x = [1, 2, 3]             # Reclaim 3.1415 now
```

First, notice that `x` is set to a different type of object each time. Again, though this is not really the case, the effect is as though the type of `x` is changing over time. Remember, in Python types live with objects, not names. Because names are just generic references to objects, this sort of code works naturally.

Second, notice that references to objects are discarded along the way. Each time `x` is assigned to a new object, Python reclaims the prior object's space. For instance, when it is assigned the string `'shrubbery'`, the object `42` is immediately reclaimed (assuming it is not referenced anywhere else)—that is, the object's space is automatically thrown back into the free space pool, to be reused for a future object.

Internally, Python accomplishes this feat by keeping a counter in every object that keeps track of the number of references currently pointing to that object. As soon as (and exactly when) this counter drops to zero, the object's memory space is automatically reclaimed. In the preceding listing, we're assuming that each time `x` is assigned to a new object, the prior object's reference counter drops to zero, causing it to be reclaimed.

The most immediately tangible benefit of garbage collection is that it means you can use objects liberally without ever needing to free up space in your script. Python will clean up unused space for you as your program runs. In practice, this eliminates a substantial amount of bookkeeping code required in lower-level languages such as C and C++.



Technically speaking, Python's garbage collection is based mainly upon *reference counters*, as described here; however, it also has a component that detects and reclaims objects with *cyclic references* in time. This component can be disabled if you're sure that your code doesn't create cycles, but it is enabled by default.

Because references are implemented as pointers, it's possible for an object to reference itself, or reference another object that does. For example, exercise 3 at the end of [Part I](#) and its solution in [Appendix B](#) show how to create a cycle by embedding a reference to a list within itself. The same phenomenon can occur for assignments to attributes of objects created from user-defined classes. Though relatively rare, because the reference counts for such objects never drop to zero, they must be treated specially.

For more details on Python's cycle detector, see the documentation for the `gc` module in Python's library manual. Also note that this description of Python's garbage collector applies to the standard CPython only; Jython and IronPython may use different schemes, though the net effect in all is similar—unused space is reclaimed for you automatically.

Shared References

So far, we've seen what happens as a single variable is assigned references to objects. Now let's introduce another variable into our interaction and watch what happens to its names and objects:

```
>>> a = 3
>>> b = a
```

Typing these two statements generates the scene captured in [Figure 6-2](#). The second line causes Python to create the variable `b`; the variable `a` is being used and not assigned here, so it is replaced with the object it references (3), and `b` is made to reference that object. The net effect is that the variables `a` and `b` wind up referencing the same object (that is, pointing to the same chunk of memory). This scenario, with multiple names referencing the same object, is called a *shared reference* in Python.

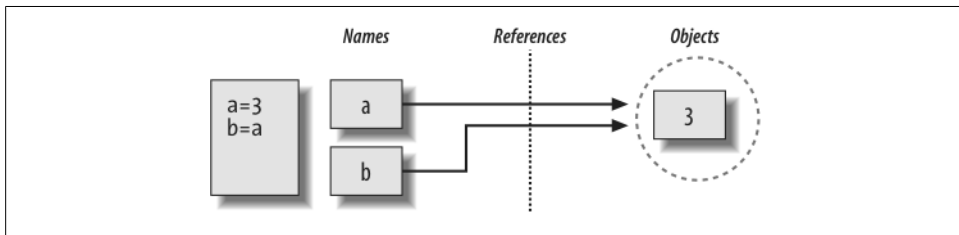


Figure 6-2. Names and objects after next running the assignment `b = a`. Variable `b` becomes a reference to the object 3. Internally, the variable is really a pointer to the object's memory space created by running the literal expression 3.

Next, suppose we extend the session with one more statement:

```
>>> a = 3
>>> b = a
>>> a = 'spam'
```

As with all Python assignments, this statement simply makes a new object to represent the string value 'spam' and sets `a` to reference this new object. It does not, however, change the value of `b`; `b` still references the original object, the integer 3. The resulting reference structure is shown in [Figure 6-3](#).

The same sort of thing would happen if we changed `b` to 'spam' instead—the assignment would change only `b`, not `a`. This behavior also occurs if there are no type differences at all. For example, consider these three statements:

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

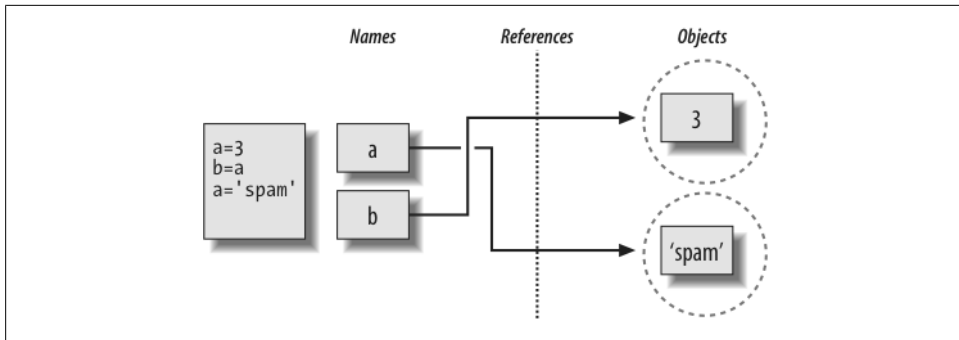


Figure 6-3. Names and objects after finally running the assignment `a = 'spam'`. Variable `a` references the new object (i.e., piece of memory) created by running the literal expression `'spam'`, but variable `b` still refers to the original object 3. Because this assignment is not an in-place change to the object 3, it changes only variable `a`, not `b`.

In this sequence, the same events transpire. Python makes the variable `a` reference the object 3 and makes `b` reference the same object as `a`, as in [Figure 6-2](#); as before, the last assignment then sets `a` to a completely different object (in this case, the integer 5, which is the result of the `+` expression). It does not change `b` as a side effect. In fact, there is no way to ever overwrite the value of the object 3—as introduced in [Chapter 4](#), integers are immutable and thus can never be changed in-place.

One way to think of this is that, unlike in some languages, in Python variables are always pointers to objects, not labels of changeable memory areas: setting a variable to a new value does not alter the original object, but rather causes the variable to reference an entirely different object. The net effect is that assignment to a variable can impact only the single variable being assigned. When mutable objects and in-place changes enter the equation, though, the picture changes somewhat; to see how, let's move on.

Shared References and In-Place Changes

As you'll see later in this part's chapters, there are objects and operations that perform in-place object changes. For instance, an assignment to an offset in a list actually changes the list object itself in-place, rather than generating a brand new list object. For objects that support such in-place changes, you need to be more aware of shared references, since a change from one name may impact others.

To further illustrate, let's take another look at the list objects introduced in [Chapter 4](#). Recall that lists, which do support in-place assignments to positions, are simply collections of other objects, coded in square brackets:

```
>>> L1 = [2, 3, 4]
>>> L2 = L1
```

L1 here is a list containing the objects 2, 3, and 4. Items inside a list are accessed by their positions, so L1[0] refers to object 2, the first item in the list L1. Of course, lists are also objects in their own right, just like integers and strings. After running the two prior assignments, L1 and L2 reference the same object, just like a and b in the prior example (see Figure 6-2). Now say that, as before, we extend this interaction to say the following:

```
>>> L1 = 24
```

This assignment simply sets L1 is to a different object; L2 still references the original list. If we change this statement's syntax slightly, however, it has a radically different effect:

```
>>> L1 = [2, 3, 4]      # A mutable object
>>> L2 = L1            # Make a reference to the same object
>>> L1[0] = 24         # An in-place change

>>> L1                 # L1 is different
[24, 3, 4]
>>> L2                 # But so is L2!
[24, 3, 4]
```

Really, we haven't changed L1 itself here; we've changed a component of the *object* that L1 references. This sort of change overwrites part of the list object in-place. Because the list object is shared by (referenced from) other variables, though, an in-place change like this doesn't only affect L1—that is, you must be aware that when you make such changes, they can impact other parts of your program. In this example, the effect shows up in L2 as well because it references the same object as L1. Again, we haven't actually changed L2, either, but its value will appear different because it has been overwritten.

This behavior is usually what you want, but you should be aware of how it works, so that it's expected. It's also just the default: if you don't want such behavior, you can request that Python *copy* objects instead of making references. There are a variety of ways to copy a list, including using the built-in `list` function and the standard library `copy` module. Perhaps the most common way is to slice from start to finish (see Chapters 4 and 7 for more on slicing):

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]         # Make a copy of L1
>>> L1[0] = 24

>>> L1
[24, 3, 4]
>>> L2                 # L2 is not changed
[2, 3, 4]
```

Here, the change made through L1 is not reflected in L2 because L2 references a copy of the object L1 references; that is, the two variables point to different pieces of memory.

Note that this slicing technique won't work on the other major mutable core types, dictionaries and sets, because they are not sequences—to copy a dictionary or set, instead use their `X.copy()` method call. Also, note that the standard library `copy` module has a call for copying any object type generically, as well as a call for copying nested object structures (a dictionary with nested lists, for example):

```
import copy
X = copy.copy(Y)           # Make top-level "shallow" copy of any object Y
X = copy.deepcopy(Y)       # Make deep copy of any object Y: copy all nested parts
```

We'll explore lists and dictionaries in more depth, and revisit the concept of shared references and copies, in Chapters 8 and 9. For now, keep in mind that objects that can be changed in-place (that is, mutable objects) are always open to these kinds of effects. In Python, this includes lists, dictionaries, and some objects defined with `class` statements. If this is not the desired behavior, you can simply copy your objects as needed.

Shared References and Equality

In the interest of full disclosure, I should point out that the garbage-collection behavior described earlier in this chapter may be more conceptual than literal for certain types. Consider these statements:

```
>>> x = 42
>>> x = 'shrubbery'      # Reclaim 42 now?
```

Because Python caches and reuses small integers and small strings, as mentioned earlier, the object `42` here is probably not literally reclaimed; instead, it will likely remain in a system table to be reused the next time you generate a `42` in your code. Most kinds of objects, though, are reclaimed immediately when they are no longer referenced; for those that are not, the caching mechanism is irrelevant to your code.

For instance, because of Python's reference model, there are two different ways to check for equality in a Python program. Let's create a shared reference to demonstrate:

```
>>> L = [1, 2, 3]
>>> M = L                # M and L reference the same object
>>> L == M                # Same value
True
>>> L is M                # Same object
True
```

The first technique here, the `==` operator, tests whether the two referenced objects have the same values; this is the method almost always used for equality checks in Python. The second method, the `is` operator, instead tests for object identity—it returns `True` only if both names point to the exact same object, so it is a much stronger form of equality testing.

Really, `is` simply compares the pointers that implement references, and it serves as a way to detect shared references in your code if needed. It returns `False` if the names point to equivalent but different objects, as is the case when we run two different literal expressions:

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3]      # M and L reference different objects
>>> L == M             # Same values
True
>>> L is M             # Different objects
False
```

Now, watch what happens when we perform the same operations on small numbers:

```
>>> X = 42
>>> Y = 42             # Should be two different objects
>>> X == Y
True
>>> X is Y             # Same object anyhow: caching at work!
True
```

In this interaction, `X` and `Y` should be `==` (same value), but not `is` (same object) because we ran two different literal expressions. Because small integers and strings are cached and reused, though, `is` tells us they reference the same single object.

In fact, if you really want to look under the hood, you can always ask Python how many references there are to an object: the `getrefcount` function in the standard `sys` module returns the object's reference count. When I ask about the integer object `1` in the IDLE GUI, for instance, it reports 837 reuses of this same object (most of which are in IDLE's system code, not mine):

```
>>> import sys
>>> sys.getrefcount(1)  # 837 pointers to this shared piece of memory
837
```

This object caching and reuse is irrelevant to your code (unless you run the `is` check!). Because you cannot change numbers or strings in-place, it doesn't matter how many references there are to the same object. Still, this behavior reflects one of the many ways Python optimizes its model for execution speed.

Dynamic Typing Is Everywhere

Of course, you don't really need to draw name/object diagrams with circles and arrows to use Python. When you're starting out, though, it sometimes helps you understand unusual cases if you can trace their reference structures. If a mutable object changes out from under you when passed around your program, for example, chances are you are witnessing some of this chapter's subject matter firsthand.

Moreover, even if dynamic typing seems a little abstract at this point, you probably will care about it eventually. Because *everything* seems to work by assignment and references in Python, a basic understanding of this model is useful in many different

contexts. As you'll see, it works the same in assignment statements, function arguments, `for` loop variables, module imports, class attributes, and more. The good news is that there is just one assignment model in Python; once you get a handle on dynamic typing, you'll find that it works the same everywhere in the language.

At the most practical level, dynamic typing means there is less code for you to write. Just as importantly, though, dynamic typing is also the root of Python's *polymorphism*, a concept we introduced in [Chapter 4](#) and will revisit again later in this book. Because we do not constrain types in Python code, it is highly flexible. As you'll see, when used well, dynamic typing and the polymorphism it provides produce code that automatically adapts to new requirements as your systems evolve.

Chapter Summary

This chapter took a deeper look at Python's dynamic typing model—that is, the way that Python keeps track of object types for us automatically, rather than requiring us to code declaration statements in our scripts. Along the way, we learned how variables and objects are associated by references in Python; we also explored the idea of garbage collection, learned how shared references to objects can affect multiple variables, and saw how references impact the notion of equality in Python.

Because there is just one assignment model in Python, and because assignment pops up everywhere in the language, it's important that you have a handle on the model before moving on. The following quiz should help you review some of this chapter's ideas. After that, we'll resume our object tour in the next chapter, with strings.

Test Your Knowledge: Quiz

1. Consider the following three statements. Do they change the value printed for A?

```
A = "spam"
B = A
B = "shrubbery"
```

2. Consider these three statements. Do they change the printed value of A?

```
A = ["spam"]
B = A
B[0] = "shrubbery"
```

3. How about these—is A changed now?

```
A = ["spam"]
B = A[: ]
B[0] = "shrubbery"
```

Test Your Knowledge: Answers

1. No: A still prints as "spam". When B is assigned to the string "shrubbery", all that happens is that the variable B is reset to point to the new string object. A and B initially share (i.e., reference/point to) the same single string object "spam", but two names are never linked together in Python. Thus, setting B to a different object has no effect on A. The same would be true if the last statement here was `B = B + 'shrubbery'`, by the way—the concatenation would make a new object for its result, which would then be assigned to B only. We can never overwrite a string (or number, or tuple) in-place, because strings are immutable.
2. Yes: A now prints as ["shrubbery"]. Technically, we haven't really changed either A or B; instead, we've changed part of the object they both reference (point to) by overwriting that object in-place through the variable B. Because A references the same object as B, the update is reflected in A as well.
3. No: A still prints as ["spam"]. The in-place assignment through B has no effect this time because the slice expression made a copy of the list object before it was assigned to B. After the second assignment statement, there are two different list objects that have the same value (in Python, we say they are `==`, but not `is`). The third statement changes the value of the list object pointed to by B, but not that pointed to by A.

Strings

The next major type on our built-in object tour is the Python *string*—an ordered collection of characters used to store and represent text-based information. We looked briefly at strings in [Chapter 4](#). Here, we will revisit them in more depth, filling in some of the details we skipped then.

From a functional perspective, strings can be used to represent just about anything that can be encoded as text: symbols and words (e.g., your name), contents of text files loaded into memory, Internet addresses, Python programs, and so on. They can also be used to hold the absolute binary values of bytes, and multibyte Unicode text used in internationalized programs.

You may have used strings in other languages, too. Python’s strings serve the same role as character arrays in languages such as C, but they are a somewhat higher-level tool than arrays. Unlike in C, in Python, strings come with a powerful set of processing tools. Also unlike languages such as C, Python has no distinct type for individual characters; instead, you just use one-character strings.

Strictly speaking, Python strings are categorized as immutable sequences, meaning that the characters they contain have a left-to-right positional order and that they cannot be changed in-place. In fact, strings are the first representative of the larger class of objects called *sequences* that we will study here. Pay special attention to the sequence operations introduced in this chapter, because they will work the same on other sequence types we’ll explore later, such as lists and tuples.

[Table 7-1](#) previews common string literals and operations we will discuss in this chapter. Empty strings are written as a pair of quotation marks (single or double) with nothing in between, and there are a variety of ways to code strings. For processing, strings support *expression* operations such as concatenation (combining strings), slicing (extracting sections), indexing (fetching by offset), and so on. Besides expressions, Python also provides a set of string *methods* that implement common string-specific tasks, as well as *modules* for more advanced text-processing tasks such as pattern matching. We’ll explore all of these later in the chapter.

Table 7-1. Common string literals and operations

Operation	Interpretation
<code>S = ''</code>	Empty string
<code>S = "spam's"</code>	Double quotes, same as single
<code>S = 's\np\ta\x00m'</code>	Escape sequences
<code>S = """..."""</code>	Triple-quoted block strings
<code>S = r'\temp\spam'</code>	Raw strings
<code>S = b'spam'</code>	Byte strings in 3.0 (Chapter 36)
<code>S = u'spam'</code>	Unicode strings in 2.6 only (Chapter 36)
<code>S1 + S2</code>	Concatenate, repeat
<code>S * 3</code>	
<code>S[i]</code>	Index, slice, length
<code>S[i:j]</code>	
<code>len(S)</code>	
<code>"a %s parrot" % kind</code>	String formatting expression
<code>"a {0} parrot".format(kind)</code>	String formatting method in 2.6 and 3.0
<code>S.find('pa')</code>	String method calls: search,
<code>S.rstrip()</code>	remove whitespace,
<code>S.replace('pa', 'xx')</code>	replacement,
<code>S.split(',')</code>	split on delimiter,
<code>S.isdigit()</code>	content test,
<code>S.lower()</code>	case conversion,
<code>S.endswith('spam')</code>	end test,
<code>'spam'.join(strlist)</code>	delimiter join,
<code>S.encode('latin-1')</code>	Unicode encoding, etc.
<code>for x in S: print(x)</code>	Iteration, membership
<code>'spam' in S</code>	
<code>[c * 2 for c in S]</code>	
<code>map(ord, S)</code>	

Beyond the core set of string tools in [Table 7-1](#), Python also supports more advanced pattern-based string processing with the standard library’s `re` (regular expression) module, introduced in [Chapter 4](#), and even higher-level text processing tools such as XML parsers, discussed briefly in [Chapter 36](#). This book’s scope, though, is focused on the fundamentals represented by [Table 7-1](#).

To cover the basics, this chapter begins with an overview of string literal forms and string expressions, then moves on to look at more advanced tools such as string methods and formatting. Python comes with many string tools, and we won't look at them all here; the complete story is chronicled in the Python library manual. Our goal here is to explore enough commonly used tools to give you a representative sample; methods we won't see in action here, for example, are largely analogous to those we will.



Content note: Technically speaking, this chapter tells only part of the string story in Python—the part most programmers need to know. It presents the basic `str` string type, which handles ASCII text and works the same regardless of which version of Python you use. That is, this chapter intentionally limits its scope to the string processing essentials that are used in most Python scripts.

From a more formal perspective, ASCII is a simple form of Unicode text. Python addresses the distinction between text and binary data by including distinct object types:

- In Python 3.0 there are three string types: `str` is used for Unicode text (ASCII or otherwise), `bytes` is used for binary data (including encoded text), and `bytearray` is a mutable variant of `bytes`.
- In Python 2.6, `unicode` strings represent wide Unicode text, and `str` strings handle both 8-bit text and binary data.

The `bytearray` type is also available as a back-port in 2.6, but not earlier, and it's not as closely bound to binary data as it is in 3.0. Because most programmers don't need to dig into the details of Unicode encodings or binary data formats, though, I've moved all such details to the Advanced Topics part of this book, in [Chapter 36](#).

If you do need to deal with more advanced string concepts such as alternative character sets or packed binary data and files, see [Chapter 36](#) after reading the material here. For now, we'll focus on the basic string type and its operations. As you'll find, the basics we'll study here also apply directly to the more advanced string types in Python's toolset.

String Literals

By and large, strings are fairly easy to use in Python. Perhaps the most complicated thing about them is that there are so many ways to write them in your code:

- Single quotes: `'spa'm'`
- Double quotes: `"spa'm"`
- Triple quotes: `'''... spam ...''', """... spam ..."""`
- Escape sequences: `"s\tp\na\0m"`
- Raw strings: `r"C:\new\test.spm"`

- Byte strings in 3.0 (see [Chapter 36](#)): `b'sp\x01am'`
- Unicode strings in 2.6 only (see [Chapter 36](#)): `u'eggs\u0020spam'`

The single- and double-quoted forms are by far the most common; the others serve specialized roles, and we're postponing discussion of the last two advanced forms until [Chapter 36](#). Let's take a quick look at all the other options in turn.

Single- and Double-Quoted Strings Are the Same

Around Python strings, single and double quote characters are interchangeable. That is, string literals can be written enclosed in either two single or two double quotes—the two forms work the same and return the same type of object. For example, the following two strings are identical, once coded:

```
>>> 'shrubbery', "shrubbery"
('shrubbery', 'shrubbery')
```

The reason for supporting both is that it allows you to embed a quote character of the other variety inside a string without escaping it with a backslash. You may embed a single quote character in a string enclosed in double quote characters, and vice versa:

```
>>> 'knight"s', "knight's"
('knight"s', "knight's")
```

Incidentally, Python automatically concatenates adjacent string literals in any expression, although it is almost as simple to add a `+` operator between them to invoke concatenation explicitly (as we'll see in [Chapter 12](#), wrapping this form in parentheses also allows it to span multiple lines):

```
>>> title = "Meaning " 'of' " Life"          # Implicit concatenation
>>> title
'Meaning of Life'
```

Notice that adding commas between these strings would result in a tuple, not a string. Also notice in all of these outputs that Python prefers to print strings in single quotes, unless they embed one. You can also embed quotes by escaping them with backslashes:

```
>>> 'knight\'s', "knight\"s"
('knight's', 'knight"s')
```

To understand why, you need to know how escapes work in general.

Escape Sequences Represent Special Bytes

The last example embedded a quote inside a string by preceding it with a backslash. This is representative of a general pattern in strings: backslashes are used to introduce special byte codings known as *escape sequences*.

Escape sequences let us embed byte codes in strings that cannot easily be typed on a keyboard. The character `\`, and one or more characters following it in the string literal, are replaced with a single character in the resulting string object, which has the binary

value specified by the escape sequence. For example, here is a five-character string that embeds a newline and a tab:

```
>>> s = 'a\nb\tc'
```

The two characters `\n` stand for a single character—the byte containing the binary value of the newline character in your character set (usually, ASCII code 10). Similarly, the sequence `\t` is replaced with the tab character. The way this string looks when printed depends on how you print it. The interactive echo shows the special characters as escapes, but `print` interprets them instead:

```
>>> s
'a\nb\tc'
>>> print(s)
a
b      c
```

To be completely sure how many bytes are in this string, use the built-in `len` function—it returns the actual number of bytes in a string, regardless of how it is displayed:

```
>>> len(s)
5
```

This string is five bytes long: it contains an ASCII *a* byte, a newline byte, an ASCII *b* byte, and so on. Note that the original backslash characters are not really stored with the string in memory; they are used to tell Python to store special byte values in the string. For coding such special bytes, Python recognizes a full set of escape code sequences, listed in [Table 7-2](#).

Table 7-2. String backslash characters

Escape	Meaning
<code>\newline</code>	Ignored (continuation line)
<code>\\</code>	Backslash (stores one <code>\</code>)
<code>\'</code>	Single quote (stores <code>'</code>)
<code>\"</code>	Double quote (stores <code>"</code>)
<code>\a</code>	Bell
<code>\b</code>	Backspace
<code>\f</code>	Formfeed
<code>\n</code>	Newline (linefeed)
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\xhh</code>	Character with hex value <i>hh</i> (at most 2 digits)
<code>\ooo</code>	Character with octal value <i>ooo</i> (up to 3 digits)
<code>\0</code>	Null: binary 0 character (doesn't end string)

Escape	Meaning
<code>\N{ id }</code>	Unicode database ID
<code>\uhhhh</code>	Unicode 16-bit hex
<code>\Uhhhhhhhh</code>	Unicode 32-bit hex ^a
<code>\other</code>	Not an escape (keeps both <code>\</code> and <i>other</i>)

^a The `\Uhhhh...` escape sequence takes exactly eight hexadecimal digits (*h*); both `\u` and `\U` can be used only in Unicode string literals.

Some escape sequences allow you to embed absolute binary values into the bytes of a string. For instance, here’s a five-character string that embeds two binary zero bytes (coded as octal escapes of one digit):

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

In Python, the zero (null) byte does not terminate a string the way it typically does in C. Instead, Python keeps both the string’s length and text in memory. In fact, no character terminates a string in Python. Here’s a string that is all absolute binary escape codes—a binary 1 and 2 (coded in octal), followed by a binary 3 (coded in hexadecimal):

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

Notice that Python displays nonprintable characters in hex, regardless of how they were specified. You can freely combine absolute value escapes and the more symbolic escape types in [Table 7-2](#). The following string contains the characters “spam”, a tab and newline, and an absolute zero value byte coded in hex:

```
>>> S = "s\tp\na\x00m"
>>> S
's\tp\na\x00m'
>>> len(S)
7
>>> print(S)
s      p
a m
```

This becomes more important to know when you process binary data files in Python. Because their contents are represented as strings in your scripts, it’s OK to process binary files that contain any sorts of binary byte values (more on files in [Chapter 9](#)).*

* If you need to care about binary data files, the chief distinction is that you open them in binary mode (using open mode flags with a `b`, such as `'rb'`, `'wb'`, and so on). In Python 3.0, binary file content is a `bytes` string, with an interface similar to that of normal strings; in 2.6, such content is a normal `str` string. See also the standard `struct` module introduced in [Chapter 9](#), which can parse binary data loaded from a file, and the extended coverage of binary files and byte strings in [Chapter 36](#).

Finally, as the last entry in [Table 7-2](#) implies, if Python does not recognize the character after a `\` as being a valid escape code, it simply keeps the backslash in the resulting string:

```
>>> x = "C:\py\code"          # Keeps \ literally
>>> x
'C:\\py\\code'
>>> len(x)
10
```

Unless you're able to commit all of [Table 7-2](#) to memory, though, you probably shouldn't rely on this behavior.[†] To code literal backslashes explicitly such that they are retained in your strings, double them up (`\\` is an escape for one `\`) or use raw strings; the next section shows how.

Raw Strings Suppress Escapes

As we've seen, escape sequences are handy for embedding special byte codes within strings. Sometimes, though, the special treatment of backslashes for introducing escapes can lead to trouble. It's surprisingly common, for instance, to see Python newcomers in classes trying to open a file with a filename argument that looks something like this:

```
myfile = open('C:\new\text.dat', 'w')
```

thinking that they will open a file called *text.dat* in the directory *C:\new*. The problem here is that `\n` is taken to stand for a newline character, and `\t` is replaced with a tab. In effect, the call tries to open a file named *C:(newline)ew(tab)ext.dat*, with usually less than stellar results.

This is just the sort of thing that raw strings are useful for. If the letter `r` (uppercase or lowercase) appears just before the opening quote of a string, it turns off the escape mechanism. The result is that Python retains your backslashes literally, exactly as you type them. Therefore, to fix the filename problem, just remember to add the letter `r` on Windows:

```
myfile = open(r'C:\new\text.dat', 'w')
```

Alternatively, because two backslashes are really an escape sequence for one backslash, you can keep your backslashes by simply doubling them up:

```
myfile = open('C:\\new\\text.dat', 'w')
```

In fact, Python itself sometimes uses this doubling scheme when it prints strings with embedded backslashes:

```
>>> path = r'C:\new\text.dat'
>>> path          # Show as Python code
'C:\\new\\text.dat'
>>> print(path)   # User-friendly format
```

[†] In classes, I've met people who have indeed committed most or all of this table to memory; I'd probably think that was really sick, but for the fact that I'm a member of the set, too.

```
C:\new\text.dat
>>> len(path)           # String length
15
```

As with numeric representation, the default format at the interactive prompt prints results as if they were code, and therefore escapes backslashes in the output. The `print` statement provides a more user-friendly format that shows that there is actually only one backslash in each spot. To verify this is the case, you can check the result of the built-in `len` function, which returns the number of bytes in the string, independent of display formats. If you count the characters in the `print(path)` output, you'll see that there really is just 1 character per backslash, for a total of 15.

Besides directory paths on Windows, raw strings are also commonly used for regular expressions (text pattern matching, supported with the `re` module introduced in [Chapter 4](#)). Also note that Python scripts can usually use *forward* slashes in directory paths on Windows and Unix because Python tries to interpret paths portably (i.e., '`C:/new/text.dat`' works when opening files, too). Raw strings are useful if you code paths using native Windows backslashes, though.



Despite its role, even a raw string cannot end in a single backslash, because the backslash escapes the following quote character—you still must escape the surrounding quote character to embed it in the string. That is, `r"...\"` is not a valid string literal—a raw string cannot end in an odd number of backslashes. If you need to end a raw string with a single backslash, you can use two and slice off the second (`r'1\nb\tc\'[:-1]`), tack one on manually (`r'1\nb\tc' + '\\'`), or skip the raw string syntax and just double up the backslashes in a normal string (`'1\\nb\\tc\\'`). All three of these forms create the same eight-character string containing three backslashes.

Triple Quotes Code Multiline Block Strings

So far, you've seen single quotes, double quotes, escapes, and raw strings in action. Python also has a triple-quoted string literal format, sometimes called a *block string*, that is a syntactic convenience for coding multiline text data. This form begins with three quotes (of either the single or double variety), is followed by any number of lines of text, and is closed with the same triple-quote sequence that opened it. Single and double quotes embedded in the string's text may be, but do not have to be, escaped—the string does not end until Python sees three unescaped quotes of the same kind used to start the literal. For example:

```
>>> mantra = """Always look
...   on the bright
...   side of life."""
>>>
>>> mantra
'Always look\n on the bright\nside of life.'
```


This string spans three lines (in some interfaces, the interactive prompt changes to ... on continuation lines; IDLE simply drops down one line). Python collects all the triple-quoted text into a single multiline string, with embedded newline characters (`\n`) at the places where your code has line breaks. Notice that, as in the literal, the second line in the result has a leading space, but the third does not—what you type is truly what you get. To see the string with the newlines interpreted, print it instead of echoing:

```
>>> print(mantra)
Always look
  on the bright
side of life.
```

Triple-quoted strings are useful any time you need multiline text in your program; for example, to embed multiline error messages or HTML or XML code in your source code files. You can embed such blocks directly in your scripts without resorting to external text files or explicit concatenation and newline characters.

Triple-quoted strings are also commonly used for documentation strings, which are string literals that are taken as comments when they appear at specific points in your file (more on these later in the book). These don't have to be triple-quoted blocks, but they usually are to allow for multiline comments.

Finally, triple-quoted strings are also sometimes used as a “horribly hackish” way to temporarily disable lines of code during development (OK, it's not really too horrible, and it's actually a fairly common practice). If you wish to turn off a few lines of code and run your script again, simply put three quotes above and below them, like this:

```
X = 1
"""
import os                                # Disable this code temporarily
print(os.getcwd())
"""
Y = 2
```

I said this was hackish because Python really does make a string out of the lines of code disabled this way, but this is probably not significant in terms of performance. For large sections of code, it's also easier than manually adding hash marks before each line and later removing them. This is especially true if you are using a text editor that does not have support for editing Python code specifically. In Python, practicality often beats aesthetics.

Strings in Action

Once you've created a string with the literal expressions we just met, you will almost certainly want to do things with it. This section and the next two demonstrate string expressions, methods, and formatting—the first line of text-processing tools in the Python language.

Basic Operations

Let's begin by interacting with the Python interpreter to illustrate the basic string operations listed earlier in [Table 7-1](#). Strings can be concatenated using the `+` operator and repeated using the `*` operator:

```
% python
>>> len('abc')           # Length: number of items
3
>>> 'abc' + 'def'        # Concatenation: a new string
'abcdef'
>>> 'Ni!' * 4             # Repetition: like "Ni!" + "Ni!" + ...
'Ni!Ni!Ni!Ni!'
```

Formally, adding two string objects creates a new string object, with the contents of its operands joined. Repetition is like adding a string to itself a number of times. In both cases, Python lets you create arbitrarily sized strings; there's no need to predeclare anything in Python, including the sizes of data structures.[‡] The `len` built-in function returns the length of a string (or any other object with a length).

Repetition may seem a bit obscure at first, but it comes in handy in a surprising number of contexts. For example, to print a line of 80 dashes, you can count up to 80, or let Python count for you:

```
>>> print('----- ...more... ---')    # 80 dashes, the hard way
>>> print('-' * 80)                     # 80 dashes, the easy way
```

Notice that operator overloading is at work here already: we're using the same `+` and `*` operators that perform addition and multiplication when using numbers. Python does the correct operation because it knows the types of the objects being added and multiplied. But be careful: the rules aren't quite as liberal as you might expect. For instance, Python doesn't allow you to mix numbers and strings in `+` expressions: `'abc'+9` raises an error instead of automatically converting `9` to a string.

As shown in the last row in [Table 7-1](#), you can also iterate over strings in loops using `for` statements and test membership for both characters and substrings with the `in` expression operator, which is essentially a search. For substrings, `in` is much like the `str.find()` method covered later in this chapter, but it returns a Boolean result instead of the substring's position:

```
>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')    # Step through items
... 
```

[‡] Unlike with C character arrays, you don't need to allocate or manage storage arrays when using Python strings; you can simply create string objects as needed and let Python manage the underlying memory space. As discussed in [Chapter 6](#), Python reclaims unused objects' memory space automatically, using a reference-count garbage-collection strategy. Each object keeps track of the number of names, data structures, etc., that reference it; when the count reaches zero, Python frees the object's space. This scheme means Python doesn't have to stop and scan all the memory to find unused space to free (an additional garbage component also collects cyclic objects).

```

h a c k e r
>>> "k" in myjob           # Found
True
>>> "z" in myjob           # Not found
False
>>> 'spam' in 'abcspamdef'  # Substring search, no position returned
True

```

The `for` loop assigns a variable to successive items in a sequence (here, a string) and executes one or more statements for each item. In effect, the variable `c` becomes a cursor stepping across the string here. We will discuss iteration tools like these and others listed in [Table 7-1](#) in more detail later in this book (especially in Chapters [14](#) and [20](#)).

Indexing and Slicing

Because strings are defined as ordered collections of characters, we can access their components by position. In Python, characters in a string are fetched by *indexing*—providing the numeric offset of the desired component in square brackets after the string. You get back the one-character string at the specified position.

As in the C language, Python offsets start at 0 and end at one less than the length of the string. Unlike C, however, Python also lets you fetch items from sequences such as strings using *negative* offsets. Technically, a negative offset is added to the length of a string to derive a positive offset. You can also think of negative offsets as counting backward from the end. The following interaction demonstrates:

```

>>> S = 'spam'
>>> S[0], S[-2]           # Indexing from front or end
('s', 'a')
>>> S[1:3], S[1:], S[:-1]  # Slicing: extract a section
('pa', 'pam', 'spa')

```

The first line defines a four-character string and assigns it the name `S`. The next line indexes it in two ways: `S[0]` fetches the item at offset 0 from the left (the one-character string 's'), and `S[-2]` gets the item at offset 2 back from the end (or equivalently, at offset $(4 + (-2))$ from the front). Offsets and slices map to cells as shown in [Figure 7-1](#).[§]

The last line in the preceding example demonstrates *slicing*, a generalized form of indexing that returns an entire *section*, not a single item. Probably the best way to think of slicing is that it is a type of *parsing* (analyzing structure), especially when applied to strings—it allows us to extract an entire section (substring) in a single step. Slices can be used to extract columns of data, chop off leading and trailing text, and more. In fact, we'll explore slicing in the context of text parsing later in this chapter.

The basics of slicing are straightforward. When you index a sequence object such as a string on a pair of offsets separated by a colon, Python returns a new object containing

[§] More mathematically minded readers (and students in my classes) sometimes detect a small asymmetry here: the leftmost item is at offset 0, but the rightmost is at offset -1 . Alas, there is no such thing as a distinct -0 value in Python.

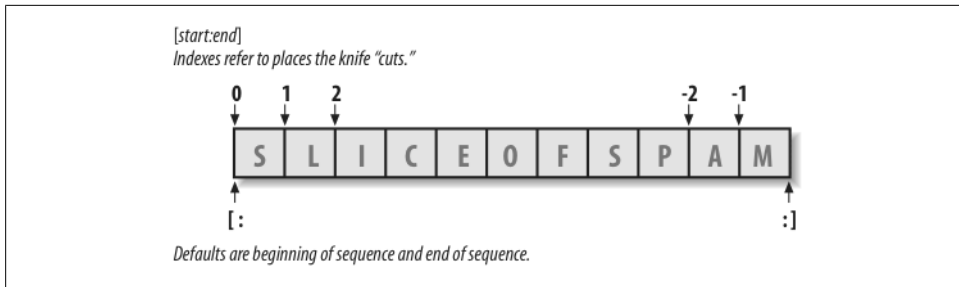


Figure 7-1. Offsets and slices: positive offsets start from the left end (offset 0 is the first item), and negatives count back from the right end (offset -1 is the last item). Either kind of offset can be used to give positions in indexing and slicing operations.

the contiguous section identified by the offset pair. The left offset is taken to be the lower bound (*inclusive*), and the right is the upper bound (*noninclusive*). That is, Python fetches all items from the lower bound up to but not including the upper bound, and returns a new object containing the fetched items. If omitted, the left and right bounds default to 0 and the length of the object you are slicing, respectively.

For instance, in the example we just saw, `S[1:3]` extracts the items at offsets 1 and 2: it grabs the second and third items, and stops before the fourth item at offset 3. Next, `S[1:]` gets *all items beyond the first*—the upper bound, which is not specified, defaults to the length of the string. Finally, `S[:-1]` fetches *all but the last item*—the lower bound defaults to 0, and -1 refers to the last item, noninclusive.

This may seem confusing at first glance, but indexing and slicing are simple and powerful tools to use, once you get the knack. Remember, if you're unsure about the effects of a slice, try it out interactively. In the next chapter, you'll see that it's even possible to change an entire section of another object in one step by assigning to a slice (though not for immutables like strings). Here's a summary of the details for reference:

- *Indexing* (`S[i]`) fetches components at offsets:
 - The first item is at offset 0.
 - Negative indexes mean to count backward from the end or right.
 - `S[0]` fetches the first item.
 - `S[-2]` fetches the second item from the end (like `S[len(S)-2]`).
- *Slicing* (`S[i:j]`) extracts contiguous sections of sequences:
 - The upper bound is noninclusive.
 - Slice boundaries default to 0 and the sequence length, if omitted.
 - `S[1:3]` fetches items at offsets 1 up to but not including 3.
 - `S[1:]` fetches items at offset 1 through the end (the sequence length).

- `S[:3]` fetches items at offset 0 up to but not including 3.
- `S[:-1]` fetches items at offset 0 up to but not including the last item.
- `S[:]` fetches items at offsets 0 through the end—this effectively performs a top-level copy of `S`.

The last item listed here turns out to be a very common trick: it makes a full top-level *copy* of a sequence object—an object with the same value, but a distinct piece of memory (you’ll find more on copies in [Chapter 9](#)). This isn’t very useful for immutable objects like strings, but it comes in handy for objects that may be changed in-place, such as lists.

In the next chapter, you’ll see that the syntax used to index by offset (square brackets) is used to index dictionaries by key as well; the operations look the same but have different interpretations.

Extended slicing: the third limit and slice objects

In Python 2.3 and later, slice expressions have support for an optional third index, used as a *step* (sometimes called a *stride*). The step is added to the index of each item extracted. The full-blown form of a slice is now `X[I:J:K]`, which means “extract all the items in `X`, from offset `I` through `J-1`, by `K`.” The third limit, `K`, defaults to 1, which is why normally all items in a slice are extracted from left to right. If you specify an explicit value, however, you can use the third limit to skip items or to reverse their order.

For instance, `X[1:10:2]` will fetch *every other item* in `X` from offsets 1–9; that is, it will collect the items at offsets 1, 3, 5, 7, and 9. As usual, the first and second limits default to 0 and the length of the sequence, respectively, so `X[::2]` gets every other item from the beginning to the end of the sequence:

```
>>> S = 'abcdefghijklmnop'
>>> S[1:10:2]
'bdfhj'
>>> S[::2]
'acegikmo'
```

You can also use a negative stride. For example, the slicing expression `"hello"[::-1]` returns the new string `"olleh"`—the first two bounds default to 0 and the length of the sequence, as before, and a stride of `-1` indicates that the slice should go from right to left instead of the usual left to right. The effect, therefore, is to *reverse* the sequence:

```
>>> S = 'hello'
>>> S[::-1]
'olleh'
```

With a negative stride, the meanings of the first two bounds are essentially reversed. That is, the slice `S[5:1:-1]` fetches the items from 2 to 5, in reverse order (the result contains items from offsets 5, 4, 3, and 2):

```
>>> S = 'abcdefg'
>>> S[5:1:-1]
'fdec'
```

Skipping and reversing like this are the most common use cases for three-limit slices, but see Python’s standard library manual for more details (or run a few experiments interactively). We’ll revisit three-limit slices again later in this book, in conjunction with the `for` loop statement.

Later in the book, we’ll also learn that slicing is equivalent to indexing with a *slice object*, a finding of importance to class writers seeking to support both operations:

```
>>> 'spam'[1:3]                # Slicing syntax
'pa'
>>> 'spam'[slice(1, 3)]        # Slice objects
'pa'
>>> 'spam'[::-1]
'maps'
>>> 'spam'[slice(None, None, -1)]
'maps'
```

Why You Will Care: Slices

Throughout this book, I will include common use case sidebars (such as this one) to give you a peek at how some of the language features being introduced are typically used in real programs. Because you won’t be able to make much sense of real use cases until you’ve seen more of the Python picture, these sidebars necessarily contain many references to topics not introduced yet; at most, you should consider them previews of ways that you may find these abstract language concepts useful for common programming tasks.

For instance, you’ll see later that the argument words listed on a system command line used to launch a Python program are made available in the `argv` attribute of the built-in `sys` module:

```
# File echo.py
import sys
print(sys.argv)

% python echo.py -a -b -c
['echo.py', '-a', '-b', '-c']
```

Usually, you’re only interested in inspecting the arguments that follow the program name. This leads to a very typical application of slices: a single slice expression can be used to return all but the first item of a list. Here, `sys.argv[1:]` returns the desired list, `['-a', '-b', '-c']`. You can then process this list without having to accommodate the program name at the front.

Slices are also often used to clean up lines read from input files. If you know that a line will have an end-of-line character at the end (a `\n` newline marker), you can get rid of it with a single expression such as `line[:-1]`, which extracts all but the last character in the line (the lower limit defaults to 0). In both cases, slices do the job of logic that must be explicit in a lower-level language.

Note that calling the `line.rstrip` method is often preferred for stripping newline characters because this call leaves the line intact if it has no newline character at the end—a common case for files created with some text-editing tools. Slicing works if you’re sure the line is properly terminated.

String Conversion Tools

One of Python’s design mottos is that it refuses the temptation to guess. As a prime example, you cannot add a number and a string together in Python, even if the string looks like a number (i.e., is all digits):

```
>>> "42" + 1
TypeError: cannot concatenate 'str' and 'int' objects
```

This is by design: because `+` can mean both addition and concatenation, the choice of conversion would be ambiguous. So, Python treats this as an error. In Python, magic is generally omitted if it will make your life more complex.

What to do, then, if your script obtains a number as a text string from a file or user interface? The trick is that you need to employ conversion tools before you can treat a string like a number, or vice versa. For instance:

```
>>> int("42"), str(42)           # Convert from/to string
(42, '42')
>>> repr(42)                   # Convert to as-code string
'42'
```

The `int` function converts a string to a number, and the `str` function converts a number to its string representation (essentially, what it looks like when printed). The `repr` function (and the older backquotes expression, removed in Python 3.0) also converts an object to its string representation, but returns the object as a string of code that can be rerun to recreate the object. For strings, the result has quotes around it if displayed with a `print` statement:

```
>>> print(str('spam'), repr('spam'))
('spam', "'spam'")
```

See the sidebar [“str and repr Display Formats” on page 116](#) for more on this topic. Of these, `int` and `str` are the generally prescribed conversion techniques.

Now, although you can’t mix strings and number types around operators such as `+`, you can manually convert operands before that operation if needed:

```
>>> S = "42"
>>> I = 1
>>> S + I
TypeError: cannot concatenate 'str' and 'int' objects

>>> int(S) + I           # Force addition
43
```

```
>>> S + str(I)           # Force concatenation
'421'
```

Similar built-in functions handle floating-point number conversions to and from strings:

```
>>> str(3.1415), float("1.5")
('3.1415', 1.5)

>>> text = "1.234E-10"
>>> float(text)
1.2340000000000001e-010
```

Later, we'll further study the built-in `eval` function; it runs a string containing Python expression code and so can convert a string to any kind of object. The functions `int` and `float` convert only to numbers, but this restriction means they are usually faster (and more secure, because they do not accept arbitrary expression code). As we saw briefly in [Chapter 5](#), the string formatting expression also provides a way to convert numbers to strings. We'll discuss formatting further later in this chapter.

Character code conversions

On the subject of conversions, it is also possible to convert a single character to its underlying ASCII integer code by passing it to the built-in `ord` function—this returns the actual binary value of the corresponding byte in memory. The `chr` function performs the inverse operation, taking an ASCII integer code and converting it to the corresponding character:

```
>>> ord('s')
115
>>> chr(115)
's'
```

You can use a loop to apply these functions to all characters in a string. These tools can also be used to perform a sort of string-based math. To advance to the next character, for example, convert and do the math in integer:

```
>>> S = '5'
>>> S = chr(ord(S) + 1)
>>> S
'6'
>>> S = chr(ord(S) + 1)
>>> S
'7'
```

At least for single-character strings, this provides an alternative to using the built-in `int` function to convert from string to integer:

```
>>> int('5')
5
>>> ord('5') - ord('0')
5
```


Such conversions can be used in conjunction with looping statements, introduced in [Chapter 4](#) and covered in depth in the next part of this book, to convert a string of binary digits to their corresponding integer values. Each time through the loop, multiply the current value by 2 and add the next digit's integer value:

```
>>> B = '1101'          # Convert binary digits to integer with ord
>>> I = 0
>>> while B != '':
...     I = I * 2 + (ord(B[0]) - ord('0'))
...     B = B[1:]
...
>>> I
13
```

A left-shift operation (`I << 1`) would have the same effect as multiplying by 2 here. We'll leave this change as a suggested exercise, though, both because we haven't studied loops in detail yet and because the `int` and `bin` built-ins we met in [Chapter 5](#) handle binary conversion tasks for us in Python 2.6 and 3.0:

```
>>> int('1101', 2)      # Convert binary to integer: built-in
13
>>> bin(13)              # Convert integer to binary
'0b1101'
```

Given enough time, Python tends to automate most common tasks!

Changing Strings

Remember the term “immutable sequence”? The *immutable* part means that you can't change a string in-place (e.g., by assigning to an index):

```
>>> S = 'spam'
>>> S[0] = "x"
Raises an error!
```

So, how do you modify text information in Python? To change a string, you need to build and assign a new string using tools such as concatenation and slicing, and then, if desired, assign the result back to the string's original name:

```
>>> S = S + 'SPAM!'      # To change a string, make a new one
>>> S
'spamSPAM!'
>>> S = S[:4] + 'Burger' + S[-1]
>>> S
'spamBurger!'
```

The first example adds a substring at the end of `S`, by concatenation (really, it makes a new string and assigns it back to `S`, but you can think of this as “changing” the original string). The second example replaces four characters with six by slicing, indexing, and concatenating. As you'll see in the next section, you can achieve similar effects with string method calls like `replace`:

```
>>> S = 'sploit'
>>> S = S.replace('pl', 'pamal')
>>> S
'spamalot'
```

Like every operation that yields a new string value, string methods generate new string objects. If you want to retain those objects, you can assign them to variable names. Generating a new string object for each string change is not as inefficient as it may sound—remember, as discussed in the preceding chapter, Python automatically garbage collects (reclaims the space of) old unused string objects as you go, so newer objects reuse the space held by prior values. Python is usually more efficient than you might expect.

Finally, it's also possible to build up new text values with string formatting expressions. Both of the following substitute objects into a string, in a sense converting the objects to strings and changing the original string according to a format specification:

```
>>> 'That is %d %s bird!' % (1, 'dead')           # Format expression
That is 1 dead bird!
>>> 'That is {0} {1} bird!'.format(1, 'dead')     # Format method in 2.6 and 3.0
'That is 1 dead bird!'
```

Despite the substitution metaphor, though, the result of formatting is a new string object, not a modified one. We'll study formatting later in this chapter; as we'll find, formatting turns out to be more general and useful than this example implies. Because the second of the preceding calls is provided as a method, though, let's get a handle on string method calls before we explore formatting further.



As we'll see in [Chapter 36](#), Python 3.0 and 2.6 introduce a new string type known as `bytearray`, which is mutable and so may be changed in place. `bytearray` objects aren't really strings; they're sequences of small, 8-bit integers. However, they support most of the same operations as normal strings and print as ASCII characters when displayed. As such, they provide another option for large amounts of text that must be changed frequently. In [Chapter 36](#) we'll also see that `ord` and `chr` handle Unicode characters, too, which might not be stored in single bytes.

String Methods

In addition to expression operators, strings provide a set of *methods* that implement more sophisticated text-processing tasks. Methods are simply functions that are associated with particular objects. Technically, they are attributes attached to objects that happen to reference callable functions. In Python, expressions and built-in functions may work across a range of types, but methods are generally *specific to object types*—string methods, for example, work only on string objects. The method sets of some types intersect in Python 3.0 (e.g., many types have a `count` method), but they are still more type-specific than other tools.

In finer-grained detail, functions are packages of code, and method calls combine two operations at once (an attribute fetch and a call):

Attribute fetches

An expression of the form *object.attribute* means “fetch the value of *attribute* in *object*.”

Call expressions

An expression of the form *function(arguments)* means “invoke the code of *function*, passing zero or more comma-separated *argument* objects to it, and return *function*’s result value.”

Putting these two together allows us to call a method of an object. The method call expression *object.method(arguments)* is evaluated from left to right—Python will first fetch the *method* of the *object* and then call it, passing in the *arguments*. If the method computes a result, it will come back as the result of the entire method-call expression.

As you’ll see throughout this part of the book, most objects have callable methods, and all are accessed using this same method-call syntax. To call an object method, as you’ll see in the following sections, you have to go through an existing object.

[Table 7-3](#) summarizes the methods and call patterns for built-in string objects in Python 3.0; these change frequently, so be sure to check Python’s standard library manual for the most up-to-date list, or run a `help` call on any string interactively. Python 2.6’s string methods vary slightly; it includes a `decode`, for example, because of its different handling of Unicode data (something we’ll discuss in [Chapter 36](#)). In this table, *S* is a string object, and optional arguments are enclosed in square brackets. String methods in this table implement higher-level operations such as splitting and joining, case conversions, content tests, and substring searches and replacements.

Table 7-3. String method calls in Python 3.0

<code>S.capitalize()</code>	<code>S.ljust(width [, fill])</code>
<code>S.center(width [, fill])</code>	<code>S.lower()</code>
<code>S.count(sub [, start [, end]])</code>	<code>S.lstrip([chars])</code>
<code>S.encode([encoding [,errors]])</code>	<code>S.maketrans(x[, y[, z]])</code>
<code>S.endswith(suffix [, start [, end]])</code>	<code>S.partition(sep)</code>
<code>S.expandtabs([tabsize])</code>	<code>S.replace(old, new [, count])</code>
<code>S.find(sub [, start [, end]])</code>	<code>S.rfind(sub [,start [,end]])</code>
<code>S.format(fmtstr, *args, **kwargs)</code>	<code>S.rindex(sub [, start [, end]])</code>
<code>S.index(sub [, start [, end]])</code>	<code>S.rjust(width [, fill])</code>
<code>S.isalnum()</code>	<code>S.rpartition(sep)</code>
<code>S.isalpha()</code>	<code>S.rsplit([sep[, maxsplit]])</code>
<code>S.isdecimal()</code>	<code>S.rstrip([chars])</code>
<code>S.isdigit()</code>	<code>S.split([sep [,maxsplit]])</code>

<code>S.isidentifier()</code>	<code>S.splitlines([keepends])</code>
<code>S.islower()</code>	<code>S.startswith(prefix [, start [, end]])</code>
<code>S.isnumeric()</code>	<code>S.strip([chars])</code>
<code>S.isprintable()</code>	<code>S.swapcase()</code>
<code>S.isspace()</code>	<code>S.title()</code>
<code>S.istitle()</code>	<code>S.translate(map)</code>
<code>S.isupper()</code>	<code>S.upper()</code>
<code>S.join(iterable)</code>	<code>S.zfill(width)</code>

As you can see, there are quite a few string methods, and we don't have space to cover them all; see Python's library manual or reference texts for all the fine points. To help you get started, though, let's work through some code that demonstrates some of the most commonly used methods in action, and illustrates Python text-processing basics along the way.

String Method Examples: Changing Strings

As we've seen, because strings are immutable, they cannot be changed in-place directly. To make a new text value from an existing string, you construct a new string with operations such as slicing and concatenation. For example, to replace two characters in the middle of a string, you can use code like this:

```
>>> S = 'spammy'
>>> S = S[:3] + 'xx' + S[5:]
>>> S
'spaxxy'
```

But, if you're really just out to replace a substring, you can use the string `replace` method instead:

```
>>> S = 'spammy'
>>> S = S.replace('mm', 'xx')
>>> S
'spaxxy'
```

The `replace` method is more general than this code implies. It takes as arguments the original substring (of any length) and the string (of any length) to replace it with, and performs a global search and replace:

```
>>> 'aa$bb$cc$dd'.replace('$', 'SPAM')
'aaSPAMbbSPAMccSPAMdd'
```

In such a role, `replace` can be used as a tool to implement template replacements (e.g., in form letters). Notice that this time we simply printed the result, instead of assigning it to a name—you need to assign results to names only if you want to retain them for later use.

If you need to replace one fixed-size string that can occur at any offset, you can do a replacement again, or search for the substring with the string `find` method and then slice:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> where = S.find('SPAM')           # Search for position
>>> where                                     # Occurs at offset 4
4
>>> S = S[:where] + 'EGGS' + S[(where+4):]
>>> S
'xxxxEGGSxxxxSPAMxxxx'
```

The `find` method returns the offset where the substring appears (by default, searching from the front), or `-1` if it is not found. As we saw earlier, it's a substring search operation just like the `in` expression, but `find` returns the position of a located substring.

Another option is to use `replace` with a third argument to limit it to a single substitution:

```
>>> S = 'xxxxSPAMxxxxSPAMxxxx'
>>> S.replace('SPAM', 'EGGS')       # Replace all
'xxxxEGGSxxxxEGGSxxxx'

>>> S.replace('SPAM', 'EGGS', 1)    # Replace one
'xxxxEGGSxxxxSPAMxxxx'
```

Notice that `replace` returns a new string object each time. Because strings are immutable, methods never really change the subject strings in-place, even if they are called “replace”!

The fact that concatenation operations and the `replace` method generate new string objects each time they are run is actually a potential downside of using them to change strings. If you have to apply many changes to a very large string, you might be able to improve your script's performance by converting the string to an object that does support in-place changes:

```
>>> S = 'spammy'
>>> L = list(S)
>>> L
['s', 'p', 'a', 'm', 'm', 'y']
```

The built-in `list` function (or an object construction call) builds a new list out of the items in any sequence—in this case, “exploding” the characters of a string into a list. Once the string is in this form, you can make multiple changes to it without generating a new copy for each change:

```
>>> L[3] = 'x'                       # Works for lists, not strings
>>> L[4] = 'x'
>>> L
['s', 'p', 'a', 'x', 'x', 'y']
```

If, after your changes, you need to convert back to a string (e.g., to write to a file), use the string `join` method to “implode” the list back into a string:

```
>>> S = ''.join(L)
>>> S
'spaxxy'
```

The `join` method may look a bit backward at first sight. Because it is a method of strings (not of lists), it is called through the desired delimiter. `join` puts the strings in a list (or other iterable) together, with the delimiter between list items; in this case, it uses an empty string delimiter to convert from a list back to a string. More generally, any string delimiter and iterable of strings will do:

```
>>> 'SPAM'.join(['eggs', 'sausage', 'ham', 'toast'])
'eggsSPAMsausageSPAMhamSPAMtoast'
```

In fact, joining substrings all at once this way often runs much faster than concatenating them individually. Be sure to also see the earlier note about the mutable `bytearray` string in Python 3.0 and 2.6, described fully in [Chapter 36](#); because it may be changed in place, it offers an alternative to this `list/join` combination for some kinds of text that must be changed often.

String Method Examples: Parsing Text

Another common role for string methods is as a simple form of text *parsing*—that is, analyzing structure and extracting substrings. To extract substrings at fixed offsets, we can employ slicing techniques:

```
>>> line = 'aaa bbb ccc'
>>> col1 = line[0:3]
>>> col3 = line[8:]
>>> col1
'aaa'
>>> col3
'ccc'
```

Here, the columns of data appear at fixed offsets and so may be sliced out of the original string. This technique passes for parsing, as long as the components of your data have fixed positions. If instead some sort of delimiter separates the data, you can pull out its components by splitting. This will work even if the data may show up at arbitrary positions within the string:

```
>>> line = 'aaa bbb ccc'
>>> cols = line.split()
>>> cols
['aaa', 'bbb', 'ccc']
```

The string `split` method chops up a string into a list of substrings, around a delimiter string. We didn't pass a delimiter in the prior example, so it defaults to whitespace—the string is split at groups of one or more spaces, tabs, and newlines, and we get back a list of the resulting substrings. In other applications, more tangible delimiters may separate the data. This example splits (and hence parses) the string at commas, a separator common in data returned by some database tools:

```
>>> line = 'bob,hacker,40'
>>> line.split(',')
['bob', 'hacker', '40']
```

Delimiters can be longer than a single character, too:

```
>>> line = "i'mSPAMaSPAMlumberjack"
>>> line.split("SPAM")
['i'm', 'a', 'lumberjack']
```

Although there are limits to the parsing potential of slicing and splitting, both run very fast and can handle basic text-extraction chores.

Other Common String Methods in Action

Other string methods have more focused roles—for example, to strip off whitespace at the end of a line of text, perform case conversions, test content, and test for a substring at the end or front:

```
>>> line = "The knights who say Ni!\n"
>>> line.rstrip()
'The knights who say Ni!'
>>> line.upper()
'THE KNIGHTS WHO SAY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
>>> line.startswith('The')
True
```

Alternative techniques can also sometimes be used to achieve the same results as string methods—the `in` membership operator can be used to test for the presence of a substring, for instance, and length and slicing operations can be used to mimic `endswith`:

```
>>> line
'The knights who say Ni!\n'

>>> line.find('Ni') != -1      # Search via method call or expression
True
>>> 'Ni' in line
True

>>> sub = 'Ni!\n'
>>> line.endswith(sub)        # End test via method call or slice
True
>>> line[-len(sub):] == sub
True
```

See also the `format` string formatting method described later in this chapter; it provides more advanced substitution tools that combine many operations in a single step.

Again, because there are so many methods available for strings, we won't look at every one here. You'll see some additional string examples later in this book, but for more

details you can also turn to the Python library manual and other documentation sources, or simply experiment interactively on your own. You can also check the `help(S.method)` results for a *method* of any string object *S* for more hints.

Note that none of the string methods accepts *patterns*—for pattern-based text processing, you must use the Python `re` standard library module, an advanced tool that was introduced in [Chapter 4](#) but is mostly outside the scope of this text (one further example appears at the end of [Chapter 36](#)). Because of this limitation, though, string methods may sometimes run more quickly than the `re` module’s tools.

The Original string Module (Gone in 3.0)

The history of Python’s string methods is somewhat convoluted. For roughly the first decade of its existence, Python provided a standard library module called `string` that contained functions that largely mirrored the current set of string object methods. In response to user requests, in Python 2.0 these functions were made available as methods of string objects. Because so many people had written so much code that relied on the original `string` module, however, it was retained for backward compatibility.

Today, you should use *only string methods*, not the original `string` module. In fact, the original module-call forms of today’s string methods have been removed completely from Python in Release 3.0. However, because you may still see the module in use in older Python code, a brief look is in order here.

The upshot of this legacy is that in Python 2.6, there technically are still two ways to invoke advanced string operations: by calling object methods, or by calling `string` module functions and passing in the objects as arguments. For instance, given a variable *X* assigned to a string object, calling an object method:

```
X.method(arguments)
```

is usually equivalent to calling the same operation through the `string` module (provided that you have already imported the module):

```
string.method(X, arguments)
```

Here’s an example of the method scheme in action:

```
>>> S = 'a+b+c+'
>>> x = S.replace('+', 'spam')
>>> x
'aspambspamcspam'
```

To access the same operation through the `string` module in Python 2.6, you need to import the module (at least once in your process) and pass in the object:

```
>>> import string
>>> y = string.replace(S, '+', 'spam')
>>> y
'aspambspamcspam'
```


Because the module approach was the standard for so long, and because strings are such a central component of most programs, you might see both call patterns in Python 2.X code you come across.

Again, though, today you should always use method calls instead of the older module calls. There are good reasons for this, besides the fact that the module calls have gone away in Release 3.0. For one thing, the module call scheme requires you to import the `string` module (methods do not require imports). For another, the module makes calls a few characters longer to type (when you load the module with `import`, that is, not using `from`). And, finally, the module runs more slowly than methods (the module maps most calls back to the methods and so incurs an extra call along the way).

The original `string` module itself, without its string method equivalents, is retained in Python 3.0 because it contains additional tools, including predefined string constants and a template object system (a relatively obscure tool omitted here—see the Python library manual for details on template objects). Unless you really want to have to change your 2.6 code to use 3.0, though, you should consider the basic string operation calls in it to be just ghosts from the past.

String Formatting Expressions

Although you can get a lot done with the string methods and sequence operations we’ve already met, Python also provides a more advanced way to combine string processing tasks—*string formatting* allows us to perform multiple type-specific substitutions on a string in a single step. It’s never strictly required, but it can be convenient, especially when formatting text to be displayed to a program’s users. Due to the wealth of new ideas in the Python world, string formatting is available in two flavors in Python today:

String formatting expressions

The original technique, available since Python’s inception; this is based upon the C language’s “printf” model and is used in much existing code.

String formatting method calls

A newer technique added in Python 2.6 and 3.0; this is more unique to Python and largely overlaps with string formatting expression functionality.

Since the method call flavor is new, there is some chance that one or the other of these may become deprecated over time. The expressions are more likely to be deprecated in later Python releases, though this should depend on the future practice of real Python programmers. As they are largely just variations on a theme, though, either technique is valid to use today. Since string formatting expressions are the original in this department, let’s start with them.

Python defines the `%` binary operator to work on strings (you may recall that this is also the remainder of division, or modulus, operator for numbers). When applied to strings, the `%` operator provides a simple way to format values as strings according to a format

definition. In short, the % operator provides a compact way to code multiple string substitutions all at once, instead of building and concatenating parts individually.

To format strings:

1. On the left of the % operator, provide a format string containing one or more embedded conversion targets, each of which starts with a % (e.g., %d).
2. On the right of the % operator, provide the object (or objects, embedded in a tuple) that you want Python to insert into the format string on the left in place of the conversion target (or targets).

For instance, in the formatting example we saw earlier in this chapter, the integer 1 replaces the %d in the format string on the left, and the string 'dead' replaces the %s. The result is a new string that reflects these two substitutions:

```
>>> 'That is %d %s bird!' % (1, 'dead')           # Format expression
That is 1 dead bird!
```

Technically speaking, string formatting expressions are usually optional—you can generally do similar work with multiple concatenations and conversions. However, formatting allows us to combine many steps into a single operation. It's powerful enough to warrant a few more examples:

```
>>> exclamation = "Ni"
>>> "The knights who say %s!" % exclamation
'The knights who say Ni!'

>>> "%d %s %d you" % (1, 'spam', 4)
'1 spam 4 you'

>>> "%s -- %s -- %s" % (42, 3.14159, [1, 2, 3])
'42 -- 3.14159 -- [1, 2, 3]'
```

The first example here plugs the string "Ni" into the target on the left, replacing the %s marker. In the second example, three values are inserted into the target string. Note that when you're inserting more than one value, you need to group the values on the right in parentheses (i.e., put them in a tuple). The % formatting expression operator expects either a single item or a tuple of one or more items on its right side.

The third example again inserts three values—an integer, a floating-point object, and a list object—but notice that all of the targets on the left are %s, which stands for conversion to string. As every type of object can be converted to a string (the one used when printing), every object type works with the %s conversion code. Because of this, unless you will be doing some special formatting, %s is often the only code you need to remember for the formatting expression.

Again, keep in mind that formatting always makes a new string, rather than changing the string on the left; because strings are immutable, it must work this way. As before, assign the result to a variable name if you need to retain it.

Advanced String Formatting Expressions

For more advanced type-specific formatting, you can use any of the conversion type codes listed in [Table 7-4](#) in formatting expressions; they appear after the % character in substitution targets. C programmers will recognize most of these because Python string formatting supports all the usual C `printf` format codes (but returns the result, instead of displaying it, like `printf`). Some of the format codes in the table provide alternative ways to format the same type; for instance, `%e`, `%f`, and `%g` provide alternative ways to format floating-point numbers.

Table 7-4. String formatting type codes

Code	Meaning
s	String (or any object's <code>str(X)</code> string)
r	s, but uses <code>repr</code> , not <code>str</code>
c	Character
d	Decimal (integer)
i	Integer
u	Same as d (obsolete: no longer unsigned)
o	Octal integer
x	Hex integer
X	x, but prints uppercase
e	Floating-point exponent, lowercase
E	Same as e, but prints uppercase
f	Floating-point decimal
F	Floating-point decimal
g	Floating-point e or f
G	Floating-point E or F
%	Literal %

In fact, conversion targets in the format string on the expression's left side support a variety of conversion operations with a fairly sophisticated syntax all their own. The general structure of conversion targets looks like this:

```
%(name)[flags][width][.precision]typecode
```

The character type codes in [Table 7-4](#) show up at the end of the target string. Between the % and the character code, you can do any of the following: provide a dictionary key; list flags that specify things like left justification (-), numeric sign (+), and zero fills (0); give a total minimum field width and the number of digits after a decimal point; and more. Both *width* and *precision* can also be coded as a * to specify that they should take their values from the next item in the input values.

Formatting target syntax is documented in full in the Python standard manuals, but to demonstrate common usage, let's look at a few examples. This one formats integers by default, and then in a six-character field with left justification and zero padding:

```
>>> x = 1234
>>> res = "integers: ...%d...%-6d...%06d" % (x, x, x)
>>> res
'integers: ...1234...1234  ...001234'
```

The %e, %f, and %g formats display floating-point numbers in different ways, as the following interaction demonstrates (%E is the same as %e but the exponent is uppercase):

```
>>> x = 1.23456789
>>> x
1.2345678899999999

>>> '%e | %f | %g' % (x, x, x)
'1.234568e+00 | 1.234568 | 1.23457'

>>> '%E' % x
'1.234568E+00'
```

For floating-point numbers, you can achieve a variety of additional formatting effects by specifying left justification, zero padding, numeric signs, field width, and digits after the decimal point. For simpler tasks, you might get by with simply converting to strings with a format expression or the `str` built-in function shown earlier:

```
>>> '%-6.2f | %05.2f | %+06.1f' % (x, x, x)
'1.23   | 01.23 | +001.2'

>>> "%s" % x, str(x)
('1.23456789', '1.23456789')
```

When sizes are not known until runtime, you can have the width and precision computed by specifying them with a * in the format string to force their values to be taken from the next item in the inputs to the right of the % operator—the 4 in the tuple here gives precision:

```
>>> '%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, 0.3333'
```

If you're interested in this feature, experiment with some of these examples and operations on your own for more information.

Dictionary-Based String Formatting Expressions

String formatting also allows conversion targets on the left to refer to the keys in a dictionary on the right and fetch the corresponding values. I haven't told you much about dictionaries yet, so here's an example that demonstrates the basics:

```
>>> "%(n)d %(x)s" % {"n":1, "x":"spam"}
'1 spam'
```

Here, the (n) and (x) in the format string refer to keys in the dictionary literal on the right and fetch their associated values. Programs that generate text such as HTML or XML often use this technique—you can build up a dictionary of values and substitute them all at once with a single formatting expression that uses key-based references:

```
>>> reply = ""                                     # Template with substitution targets
Greetings...
Hello %(name)s!
Your age squared is %(age)s
"""

>>> values = {'name': 'Bob', 'age': 40}             # Build up values to substitute
>>> print(reply % values)                           # Perform substitutions

Greetings...
Hello Bob!
Your age squared is 40
```

This trick is also used in conjunction with the `vars` built-in function, which returns a dictionary containing all the variables that exist in the place it is called:

```
>>> food = 'spam'
>>> age = 40
>>> vars()
{'food': 'spam', 'age': 40, ...many more... }
```

When used on the right of a format operation, this allows the format string to refer to variables by name (i.e., by dictionary key):

```
>>> "%(age)d %(food)s" % vars()
'40 spam'
```

We'll study dictionaries in more depth in [Chapter 8](#). See also [Chapter 5](#) for examples that convert to hexadecimal and octal number strings with the `%x` and `%o` formatting target codes.

String Formatting Method Calls

As mentioned earlier, Python 2.6 and 3.0 introduced a new way to format strings that is seen by some as a bit more Python-specific. Unlike formatting expressions, formatting method calls are not closely based upon the C language's "printf" model, and they are more verbose and explicit in intent. On the other hand, the new technique still relies on some "printf" concepts, such as type codes and formatting specifications. Moreover, it largely overlaps with (and sometimes requires a bit more code than) formatting expressions, and it can be just as complex in advanced roles. Because of this, there is no best-use recommendation between expressions and method calls today, so most programmers would be well served by a cursory understanding of both schemes.

The Basics

In short, the new string object's `format` method in 2.6 and 3.0 (and later) uses the subject string as a template and takes any number of arguments that represent values to be substituted according to the template. Within the subject string, curly braces designate substitution targets and arguments to be inserted either by position (e.g., `{1}`) or keyword (e.g., `{food}`). As we'll learn when we study argument passing in depth in [Chapter 18](#), arguments to functions and methods may be passed by position or keyword name, and Python's ability to collect arbitrarily many positional and keyword arguments allows for such general method call patterns. In Python 2.6 and 3.0, for example:

```
>>> template = '{0}, {1} and {2}'                                     # By position
>>> template.format('spam', 'ham', 'eggs')
'spam, ham and eggs'

>>> template = '{motto}, {pork} and {food}'                           # By keyword
>>> template.format(motto='spam', pork='ham', food='eggs')
'spam, ham and eggs'

>>> template = '{motto}, {0} and {food}'                               # By both
>>> template.format('ham', motto='spam', food='eggs')
'spam, ham and eggs'
```

Naturally, the string can also be a literal that creates a temporary string, and arbitrary object types can be substituted:

```
>>> '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
'3.14, 42 and [1, 2]'
```

Just as with the `%` expression and other string methods, `format` creates and returns a new string object, which can be printed immediately or saved for further work (recall that strings are immutable, so `format` really *must* make a new object). String formatting is not just for display:

```
>>> X = '{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])
>>> X
'3.14, 42 and [1, 2]'
```

```
>>> X.split(' and ')
['3.14, 42', '[1, 2]']
```

```
>>> Y = X.replace('and', 'but under no circumstances')
>>> Y
'3.14, 42 but under no circumstances [1, 2]'
```

Adding Keys, Attributes, and Offsets

Like `%` formatting expressions, `format` calls can become more complex to support more advanced usage. For instance, `format` strings can name object attributes and dictionary keys—as in normal Python syntax, square brackets name dictionary keys and dots denote object attributes of an item referenced by position or keyword. The first of the

following examples indexes a dictionary on the key “spam” and then fetches the attribute “platform” from the already imported `sys` module object. The second does the same, but names the objects by keyword instead of position:

```
>>> import sys

>>> 'My {1[spam]} runs {0.platform}'.format(sys, {'spam': 'laptop'})
'My laptop runs win32'

>>> 'My {config[spam]} runs {sys.platform}'.format(sys=sys,
                                                    config={'spam': 'laptop'})
'My laptop runs win32'
```

Square brackets in format strings can name list (and other sequence) offsets to perform indexing, too, but only single positive offsets work syntactically within format strings, so this feature is not as general as you might think. As with `%` expressions, to name negative offsets or slices, or to use arbitrary expression results in general, you must run expressions outside the format string itself:

```
>>> somelist = list('SPAM')
>>> somelist
['S', 'P', 'A', 'M']

>>> 'first={0[0]}, third={0[2]}'.format(somelist)
'first=S, third=A'

>>> 'first={0}, last={1}'.format(somelist[0], somelist[-1])  # [-1] fails in fmt
'first=S, last=M'

>>> parts = somelist[0], somelist[-1], somelist[1:3]          # [1:3] fails in fmt
>>> 'first={0}, last={1}, middle={2}'.format(*parts)
'first=S, last=M, middle=['P', 'A']"
```

Adding Specific Formatting

Another similarity with `%` expressions is that more specific layouts can be achieved by adding extra syntax in the format string. For the formatting method, we use a colon after the substitution target’s identification, followed by a format specifier that can name the field size, justification, and a specific type code. Here’s the formal structure of what can appear as a substitution target in a format string:

```
{fieldname!conversionflag:formatspec}
```

In this substitution target syntax:

- *fieldname* is a number or keyword naming an argument, followed by optional “name” attribute or “[index]” component references.
- *conversionflag* can be `r`, `s`, or `a` to call `repr`, `str`, or `ascii` built-in functions on the value, respectively.

- *formatspec* specifies how the value should be presented, including details such as field width, alignment, padding, decimal precision, and so on, and ends with an optional data type code.

The *formatspec* component after the colon character is formally described as follows (brackets denote optional components and are not coded literally):

```
[[fill]align][sign][#][0][width][.precision][typecode]
```

align may be <, >, =, or ^, for left alignment, right alignment, padding after a sign character, or centered alignment, respectively. The *formatspec* also contains nested {} format strings with field names only, to take values from the arguments list dynamically (much like the * in formatting expressions).

See Python’s library manual for more on substitution syntax and a list of the available type codes—they almost completely overlap with those used in % expressions and listed previously in Table 7-4, but the format method also allows a “b” type code used to display integers in binary format (it’s equivalent to using the bin built-in call), allows a “%” type code to display percentages, and uses only “d” for base-10 integers (not “i” or “u”).

As an example, in the following {0:10} means the first positional argument in a field 10 characters wide, {1:<10} means the second positional argument left-justified in a 10-character-wide field, and {0.platform:>10} means the platform attribute of the first argument right-justified in a 10-character-wide field:

```
>>> '{0:10} = {1:10}'.format('spam', 123.4567)
'spam      =    123.457'

>>> '{0:>10} = {1:<10}'.format('spam', 123.4567)
'      spam = 123.457 '

>>> '{0.platform:>10} = {1[item]:<10}'.format(sys, dict(item='laptop'))
'      win32 = laptop'
```

Floating-point numbers support the same type codes and formatting specificity in formatting method calls as in % expressions. For instance, in the following {2:g} means the third argument formatted by default according to the “g” floating-point representation, {1:.2f} designates the “f” floating-point format with just 2 decimal digits, and {2:06.2f} adds a field with a width of 6 characters and zero padding on the left:

```
>>> '{0:e}, {1:.3e}, {2:g}'.format(3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '{0:f}, {1:.2f}, {2:06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Hex, octal, and binary formats are supported by the format method as well. In fact, string formatting is an alternative to some of the built-in functions that format integers to a given base:


```

>>> '{0:X}, {1:o}, {2:b}'.format(255, 255, 255)    # Hex, octal, binary
'FF, 377, 11111111'

>>> bin(255), int('11111111', 2), 0b11111111      # Other to/from binary
('0b11111111', 255, 255)

>>> hex(255), int('FF', 16), 0xFF                 # Other to/from hex
('0xff', 255, 255)

>>> oct(255), int('377', 8), 0o377, 0377          # Other to/from octal
('0377', 255, 255, 255)                          # 0377 works in 2.6, not 3.0!

```

Formatting parameters can either be hardcoded in format strings or taken from the arguments list dynamically by nested format syntax, much like the star syntax in formatting expressions:

```

>>> '{0:.2f}'.format(1 / 3.0)                      # Parameters hardcoded
'0.33'
>>> '%.2f' % (1 / 3.0)
'0.33'

>>> '{0:.{1}f}'.format(1 / 3.0, 4)                 # Take value from arguments
'0.3333'
>>> '%.*f' % (4, 1 / 3.0)                          # Ditto for expression
'0.3333'

```

Finally, Python 2.6 and 3.0 also provide a new built-in `format` function, which can be used to format a single item. It's a more concise alternative to the string `format` method, and is roughly similar to formatting a single item with the `%` formatting expression:

```

>>> '{0:.2f}'.format(1.2345)                       # String method
'1.23'
>>> format(1.2345, '.2f')                          # Built-in function
'1.23'
>>> '%.2f' % 1.2345                                # Expression
'1.23'

```

Technically, the `format` built-in runs the subject object's `__format__` method, which the `str.format` method does internally for each formatted item. It's still more verbose than the original `%` expression's equivalent, though—which leads us to the next section.

Comparison to the % Formatting Expression

If you study the prior sections closely, you'll probably notice that at least for positional references and dictionary keys, the string `format` method looks very much like the `%` formatting expression, especially in advanced use with type codes and extra formatting syntax. In fact, in common use cases formatting expressions may be easier to code than formatting method calls, especially when using the generic `%s` print-string substitution target:

```

print('%s=%s' % ('spam', 42))                      # 2.X+ format expression

print('{0}={1}'.format('spam', 42))                 # 3.0 (and 2.6) format method

```

As we'll see in a moment, though, more complex formatting tends to be a draw in terms of complexity (difficult tasks are generally difficult, regardless of approach), and some see the formatting method as largely redundant.

On the other hand, the formatting method also offers a few potential advantages. For example, the original `%` expression can't handle keywords, attribute references, and binary type codes, although dictionary key references in `%` format strings can often achieve similar goals. To see how the two techniques overlap, compare the following `%` expressions to the equivalent `format` method calls shown earlier:

```
# The basics: with % instead of format()
```

```
>>> template = '%s, %s, %s'
>>> template % ('spam', 'ham', 'eggs')           # By position
'spam, ham, eggs'

>>> template = '%(motto)s, %(pork)s and %(food)s'
>>> template % dict(motto='spam', pork='ham', food='eggs') # By key
'spam, ham and eggs'

>>> '%s, %s and %s' % (3.14, 42, [1, 2])         # Arbitrary types
'3.14, 42 and [1, 2]'
```

```
# Adding keys, attributes, and offsets
```

```
>>> 'My %(spam)s runs %(platform)s' % {'spam': 'laptop', 'platform': sys.platform}
'My laptop runs win32'

>>> 'My %(spam)s runs %(platform)s' % dict(spam='laptop', platform=sys.platform)
'My laptop runs win32'

>>> somelist = list('SPAM')
>>> parts = somelist[0], somelist[-1], somelist[1:3]
>>> 'first=%s, last=%s, middle=%s' % parts
'first=S, last=M, middle=[\'P\', \'A\']'
```

When more complex formatting is applied the two techniques approach parity in terms of complexity, although if you compare the following with the `format` method call equivalents listed earlier you'll again find that the `%` expressions tend to be a bit simpler and more concise:

```
# Adding specific formatting
```

```
>>> '%-10s = %10s' % ('spam', 123.4567)
'spam          = 123.4567'

>>> '%10s = %-10s' % ('spam', 123.4567)
'      spam = 123.4567 '

>>> '%(plat)10s = %(item)-10s' % dict(plat=sys.platform, item='laptop')
'      win32 = laptop '
```

Floating-point numbers

```
>>> '%e, %.3e, %g' % (3.14159, 3.14159, 3.14159)
'3.141590e+00, 3.142e+00, 3.14159'

>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Hex and octal, but not binary

```
>>> '%x, %o' % (255, 255)
'ff, 377'
```

The `format` method has a handful of advanced features that the `%` expression does not, but even more involved formatting still seems to be essentially a draw in terms of complexity. For instance, the following shows the same result generated with both techniques, with field sizes and justifications and various argument reference methods:

Hardcoded references in both

```
>>> import sys

>>> 'My {1[spam]:<8} runs {0.platform:>8}'.format(sys, {'spam': 'laptop'})
'My laptop runs win32'

>>> 'My %(spam)-8s runs %(plat)8s' % dict(spam='laptop', plat=sys.platform)
'My laptop runs win32'
```

In practice, programs are less likely to hardcode references like this than to execute code that builds up a set of substitution data ahead of time (to collect data to substitute into an HTML template all at once, for instance). When we account for common practice in examples like this, the comparison between the `format` method and the `%` expression is even more direct (as we'll see in [Chapter 18](#), the `**data` in the method call here is special syntax that unpacks a dictionary of keys and values into individual “name=value” keyword arguments so they can be referenced by name in the format string):

Building data ahead of time in both

```
>>> data = dict(platform=sys.platform, spam='laptop')

>>> 'My {spam:<8} runs {platform:>8}'.format(**data)
'My laptop runs win32'

>>> 'My %(spam)-8s runs %(platform)8s' % data
'My laptop runs win32'
```

As usual, the Python community will have to decide whether `%` expressions, `format` method calls, or a toolset with both techniques proves better over time. Experiment with these techniques on your own to get a feel for what they offer, and be sure to see the Python 2.6 and 3.0 library manuals for more details.



String format method enhancements in Python 3.1: The upcoming 3.1 release (in alpha form as this chapter was being written) will add a thousand-separator syntax for numbers, which inserts commas between three-digit groups. Add a comma before the type code to make this work, as follows:

```
>>> '{0:d}'.format(999999999999)
'999999999999'
```

```
>>> '{0:,d}'.format(999999999999)
'999,999,999,999'
```

Python 3.1 also assigns relative numbers to substitution targets automatically if they are not included explicitly, though using this extension may negate one of the main benefits of the formatting method, as the next section describes:

```
>>> '{:,d}'.format(999999999999)
'999,999,999,999'
```

```
>>> '{:,d} {:,d}'.format(9999999, 8888888)
'9,999,999 8,888,888'
```

```
>>> '{:,.2f}'.format(296999.2567)
'296,999.26'
```

This book doesn't cover 3.1 officially, so you should take this as a preview. Python 3.1 will also address a major performance issue in 3.0 related to the speed of file input/output operations, which made 3.0 impractical for many types of programs. See the 3.1 release notes for more details. See also the *formats.py* comma-insertion and money-formatting function examples in [Chapter 24](#) for a manual solution that can be imported and used prior to Python 3.1.

Why the New Format Method?

Now that I've gone to such lengths to compare and contrast the two formatting techniques, I need to explain why you might want to consider using the `format` method variant at times. In short, although the formatting method can sometimes require more code, it also:

- Has a few extra features not found in the `%` expression
- Can make substitution value references more explicit
- Trades an operator for an arguably more mnemonic method name
- Does not support different syntax for single and multiple substitution value cases

Although both techniques are available today and the formatting expression is still widely used, the `format` method might eventually subsume it. But because the choice is currently still yours to make, let's briefly expand on some of the differences before moving on.

Extra features

The method call supports a few extras that the expression does not, such as binary type codes and (coming in Python 3.1) thousands groupings. In addition, the method call supports key and attribute references directly. As we've seen, though, the formatting expression can usually achieve the same effects in other ways:

```
>>> '{0:b}'.format((2 ** 16) - 1)
'1111111111111111'

>>> '%b' % ((2 ** 16) - 1)
ValueError: unsupported format character 'b' (0x62) at index 1

>>> bin((2 ** 16) - 1)
'0b1111111111111111'

>>> '%s' % bin((2 ** 16) - 1)[2:]
'1111111111111111'
```

See also the prior examples that compare dictionary-based formatting in the % expression to key and attribute references in the `format` method; especially in common practice, the two seem largely variations on a theme.

Explicit value references

One use case where the `format` method is at least debatably clearer is when there are many values to be substituted into the format string. The *lister.py* classes example we'll meet in [Chapter 30](#), for example, substitutes six items into a single string, and in this case the method's `{i}` position labels seem easier to read than the expression's `%s`:

```
'\n%s<Class %s, address %s:\n%s%s>\n' % (...)           # Expression

'\n{0}<Class {1}, address {2}:\n{3}{4}{5}>\n'.format(...)  # Method
```

On the other hand, using dictionary keys in % expressions can mitigate much of this difference. This is also something of a worst-case scenario for formatting complexity, and not very common in practice; more typical use cases seem largely a tossup. Moreover, in Python 3.1 (still in alpha release form as I write these words), numbering substitution values will become optional, thereby subverting this purported benefit altogether:

```
C:\misc> C:\Python31\python
>>> 'The {0} side {1} {2}'.format('bright', 'of', 'life')
'The bright side of life'
>>>
>>> 'The {} side {} {}'.format('bright', 'of', 'life')           # Python 3.1+
'The bright side of life'
>>>
>>> 'The %s side %s %s' % ('bright', 'of', 'life')
'The bright side of life'
```

Using 3.1’s automatic relative numbering like this seems to negate a large part of the method’s advantage. Compare the effect on floating-point formatting, for example—the formatting expression is still more concise, and still seems less cluttered:

```
C:\misc> C:\Python31\python
>>> '{0:f}, {1:.2f}, {2:05.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 03.14'
>>>
>>> '{:f}, {:.2f}, {06.2f}'.format(3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
>>>
>>> '%f, %.2f, %06.2f' % (3.14159, 3.14159, 3.14159)
'3.141590, 3.14, 003.14'
```

Method names and general arguments

Given this 3.1 auto-numbering change, the only clearly remaining potential advantages of the formatting method are that it replaces the % operator with a more mnemonic `format` method name and does not distinguish between single and multiple substitution values. The former may make the method appear simpler to beginners at first glance (“format” may be easier to parse than multiple “%” characters), though this is too subjective to call.

The latter difference might be more significant—with the `format` expression, a single value can be given by itself, but multiple values must be enclosed in a tuple:

```
>>> '%.2f' % 1.2345
'1.23'
>>> '%.2f %s' % (1.2345, 99)
'1.23 99'
```

Technically, the formatting expression accepts either a single substitution value, or a tuple of one or more items. In fact, because a single item can be given *either* by itself or within a tuple, a tuple to be formatted must be provided as nested tuples:

```
>>> '%s' % 1.23
'1.23'
>>> '%s' % (1.23,)
'1.23'
>>> '%s' % ((1.23,))
'(1.23,)'
```

The formatting method, on the other hand, tightens this up by accepting general function arguments in both cases:

```
>>> '{0:.2f}'.format(1.2345)
'1.23'
>>> '{0:.2f} {1}'.format(1.2345, 99)
'1.23 99'

>>> '{0}'.format(1.23)
'1.23'
>>> '{0}'.format((1.23,))
'(1.23,)'
```

Consequently, it might be less confusing to beginners and cause fewer programming mistakes. This is still a fairly minor issue, though—if you always enclose values in a tuple and ignore the nontupled option, the expression is essentially the same as the method call here. Moreover, the method incurs an extra price in inflated code size to achieve its limited flexibility. Given that the expression has been used extensively throughout Python’s history, it’s not clear that this point justifies breaking existing code for a new tool that is so similar, as the next section argues.

Possible future deprecation?

As mentioned earlier, there is some risk that Python developers may deprecate the `%` expression in favor of the `format` method in the future. In fact, there is a note to this effect in Python 3.0’s manuals.

This has not yet occurred, of course, and both formatting techniques are fully available and reasonable to use in Python 2.6 and 3.0 (the versions of Python this book covers). Both techniques are supported in the upcoming Python 3.1 release as well, so deprecation of either seems unlikely for the foreseeable future. Moreover, because formatting expressions are used extensively in almost all existing Python code written to date, most programmers will benefit from being familiar with both techniques for many years to come.

If this deprecation ever does occur, though, you may need to recode all your `%` expressions as `format` methods, and translate those that appear in this book, in order to use a newer Python release. At the risk of editorializing here, I hope that such a change will be based upon the future common practice of actual Python programmers, not the whims of a handful of core developers—particularly given that the window for Python 3.0’s many incompatible changes is now closed. Frankly, this deprecation would seem like trading one complicated thing for another complicated thing—one that is largely equivalent to the tool it would replace! If you care about migrating to future Python releases, though, be sure to watch for developments on this front over time.

General Type Categories

Now that we’ve explored the first of Python’s collection objects, the string, let’s pause to define a few general type concepts that will apply to most of the types we look at from here on. With regard to built-in types, it turns out that operations work the same for all the types in the same category, so we’ll only need to define most of these ideas once. We’ve only examined numbers and strings so far, but because they are representative of two of the three major type categories in Python, you already know more about several other types than you might think.

Types Share Operation Sets by Categories

As you’ve learned, strings are immutable sequences: they cannot be changed in-place (the *immutable* part), and they are positionally ordered collections that are accessed by offset (the *sequence* part). Now, it so happens that all the sequences we’ll study in this part of the book respond to the same sequence operations shown in this chapter at work on strings—concatenation, indexing, iteration, and so on. More formally, there are three major type (and operation) categories in Python:

Numbers (integer, floating-point, decimal, fraction, others)

Support addition, multiplication, etc.

Sequences (strings, lists, tuples)

Support indexing, slicing, concatenation, etc.

Mappings (dictionaries)

Support indexing by key, etc.

Sets are something of a category unto themselves (they don’t map keys to values and are not positionally ordered sequences), and we haven’t yet explored mappings on our in-depth tour (dictionaries are discussed in the next chapter). However, many of the other types we will encounter will be similar to numbers and strings. For example, for any sequence objects *X* and *Y*:

- *X* + *Y* makes a new sequence object with the contents of both operands.
- *X* * *N* makes a new sequence object with *N* copies of the sequence operand *X*.

In other words, these operations work the same way on any kind of sequence, including strings, lists, tuples, and some user-defined object types. The only difference is that the new result object you get back is of the same type as the operands *X* and *Y*—if you concatenate lists, you get back a new list, not a string. Indexing, slicing, and other sequence operations work the same on all sequences, too; the type of the objects being processed tells Python which flavor of the task to perform.

Mutable Types Can Be Changed In-Place

The immutable classification is an important constraint to be aware of, yet it tends to trip up new users. If an object type is immutable, you cannot change its value in-place; Python raises an error if you try. Instead, you must run code to make a new object containing the new value. The major core types in Python break down as follows:

Immutable (numbers, strings, tuples, frozensets)

None of the object types in the immutable category support in-place changes, though we can always run expressions to make new objects and assign their results to variables as needed.

Mutables (lists, dictionaries, sets)

Conversely, the mutable types can always be changed in-place with operations that do not create new objects. Although such objects can be copied, in-place changes support direct modification.

Generally, immutable types give some degree of integrity by guaranteeing that an object won't be changed by another part of a program. For a refresher on why this matters, see the discussion of shared object references in [Chapter 6](#). To see how lists, dictionaries, and tuples participate in type categories, we need to move ahead to the next chapter.

Chapter Summary

In this chapter, we took an in-depth tour of the string object type. We learned about coding string literals, and we explored string operations, including sequence expressions, string method calls, and string formatting with both expressions and method calls. Along the way, we studied a variety of concepts in depth, such as slicing, method call syntax, and triple-quoted block strings. We also defined some core ideas common to a variety of types: sequences, for example, share an entire set of operations.

In the next chapter, we'll continue our types tour with a look at the most general object collections in Python—lists and dictionaries. As you'll find, much of what you've learned here will apply to those types as well. And as mentioned earlier, in the final part of this book we'll return to Python's string model to flesh out the details of Unicode text and binary data, which are of interest to some, but not all, Python programmers. Before moving on, though, here's another chapter quiz to review the material covered here.

Test Your Knowledge: Quiz

1. Can the string `find` method be used to search a list?
2. Can a string slice expression be used on a list?
3. How would you convert a character to its ASCII integer code? How would you convert the other way, from an integer to a character?
4. How might you go about changing a string in Python?
5. Given a string `S` with the value `"s,pa,m"`, name two ways to extract the two characters in the middle.
6. How many characters are there in the string `"a\nb\x1f\000d"`?
7. Why might you use the `string` module instead of string method calls?

Test Your Knowledge: Answers

1. No, because methods are always type-specific; that is, they only work on a single data type. Expressions like `X+Y` and built-in functions like `len(X)` are generic, though, and may work on a variety of types. In this case, for instance, the `in` membership expression has a similar effect as the string `find`, but it can be used to search both strings and lists. In Python 3.0, there is some attempt to group methods by categories (for example, the mutable sequence types `list` and `bytearray` have similar method sets), but methods are still more type-specific than other operation sets.
2. Yes. Unlike methods, expressions are generic and apply to many types. In this case, the slice expression is really a sequence operation—it works on any type of sequence object, including strings, lists, and tuples. The only difference is that when you slice a list, you get back a new list.
3. The built-in `ord(S)` function converts from a one-character string to an integer character code; `chr(I)` converts from the integer code back to a string.
4. Strings cannot be changed; they are immutable. However, you can achieve a similar effect by creating a new string—by concatenating, slicing, running formatting expressions, or using a method call like `replace`—and then assigning the result back to the original variable name.
5. You can slice the string using `S[2:4]`, or split on the comma and index the string using `S.split(',')[1]`. Try these interactively to see for yourself.
6. Six. The string `"a\nb\x1f\000d"` contains the bytes `a`, newline (`\n`), `b`, binary 31 (a hex escape `\x1f`), binary 0 (an octal escape `\000`), and `d`. Pass the string to the built-in `len` function to verify this, and print each of its character's `ord` results to see the actual byte values. See [Table 7-2](#) for more details.
7. You should never use the `string` module instead of string object method calls today—it's deprecated, and its calls are removed completely in Python 3.0. The only reason for using the `string` module at all is for its other tools, such as predefined constants. You might also see it appear in what is now very old and dusty Python code.

Lists and Dictionaries

This chapter presents the list and dictionary object types, both of which are collections of other objects. These two types are the main workhorses in almost all Python scripts. As you'll see, both types are remarkably flexible: they can be changed in-place, can grow and shrink on demand, and may contain and be nested in any other kind of object. By leveraging these types, you can build up and process arbitrarily rich information structures in your scripts.

Lists

The next stop on our built-in object tour is the Python *list*. Lists are Python's most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists. Also, unlike strings, lists may be changed in-place by assignment to offsets and slices, list method calls, deletion statements, and more—they are *mutable* objects.

Python lists do the work of most of the collection data structures you might have to implement manually in lower-level languages such as C. Here is a quick look at their main properties. Python lists are:

Ordered collections of arbitrary objects

From a functional view, lists are just places to collect other objects so you can treat them as groups. Lists also maintain a left-to-right positional ordering among the items they contain (i.e., they are sequences).

Accessed by offset

Just as with strings, you can fetch a component object out of a list by indexing the list on the object's offset. Because items in lists are ordered by their positions, you can also do tasks such as slicing and concatenation.

Variable-length, heterogeneous, and arbitrarily nestable

Unlike strings, lists can grow and shrink in-place (their lengths can vary), and they can contain any sort of object, not just one-character strings (they’re heterogeneous). Because lists can contain other complex objects, they also support arbitrary nesting; you can create lists of lists of lists, and so on.

Of the category “mutable sequence”

In terms of our type category qualifiers, lists are mutable (i.e., can be changed in-place) and can respond to all the sequence operations used with strings, such as indexing, slicing, and concatenation. In fact, sequence operations work the same on lists as they do on strings; the only difference is that sequence operations such as concatenation and slicing return new lists instead of new strings when applied to lists. Because lists are mutable, however, they also support other operations that strings don’t (such as deletion and index assignment operations, which change the lists in-place).

Arrays of object references

Technically, Python lists contain zero or more references to other objects. Lists might remind you of arrays of pointers (addresses) if you have a background in some other languages. Fetching an item from a Python list is about as fast as indexing a C array; in fact, lists really are arrays inside the standard Python interpreter, not linked structures. As we learned in [Chapter 6](#), though, Python always follows a reference to an object whenever the reference is used, so your program deals only with objects. Whenever you assign an object to a data structure component or variable name, Python always stores a reference to that same object, not a copy of it (unless you request a copy explicitly).

[Table 8-1](#) summarizes common and representative list object operations. As usual, for the full story see the Python standard library manual, or run a `help(list)` or `dir(list)` call interactively for a complete list of list methods—you can pass in a real list, or the word `list`, which is the name of the list data type.

Table 8-1. Common list literals and operations

Operation	Interpretation
<code>L = []</code>	An empty list
<code>L = [0, 1, 2, 3]</code>	Four items: indexes 0..3
<code>L = ['abc', ['def', 'ghi']]</code>	Nested sublists
<code>L = list('spam')</code>	Lists of an iterable’s items, list of successive integers
<code>L = list(range(-4, 4))</code>	
<code>L[i]</code>	Index, index of index, slice, length
<code>L[i][j]</code>	
<code>L[i:j]</code>	
<code>len(L)</code>	

Operation	Interpretation
<code>L1 + L2</code>	Concatenate, repeat
<code>L * 3</code>	
<code>for x in L: print(x)</code>	Iteration, membership
<code>3 in L</code>	
<code>L.append(4)</code>	Methods: growing
<code>L.extend([5,6,7])</code>	
<code>L.insert(I, X)</code>	
<code>L.index(1)</code>	Methods: searching
<code>L.count(X)</code>	
<code>L.sort()</code>	Methods: sorting, reversing, etc.
<code>L.reverse()</code>	
<code>del L[k]</code>	Methods, statement: shrinking
<code>del L[i:j]</code>	
<code>L.pop()</code>	
<code>L.remove(2)</code>	
<code>L[i:j] = []</code>	
<code>L[i] = 1</code>	Index assignment, slice assignment
<code>L[i:j] = [4,5,6]</code>	
<code>L = [x**2 for x in range(5)]</code>	List comprehensions and maps (Chapters 14, 20)
<code>list(map(ord, 'spam'))</code>	

When written down as a literal expression, a list is coded as a series of objects (really, expressions that return objects) in square brackets, separated by commas. For instance, the second row in [Table 8-1](#) assigns the variable `L` to a four-item list. A nested list is coded as a nested square-bracketed series (row 3), and the empty list is just a square-bracket pair with nothing inside (row 1).*

Many of the operations in [Table 8-1](#) should look familiar, as they are the same sequence operations we put to work on strings—indexing, concatenation, iteration, and so on. Lists also respond to list-specific method calls (which provide utilities such as sorting, reversing, adding items to the end, etc.), as well as in-place change operations (deleting items, assignment to indexes and slices, and so forth). Lists have these tools for change operations because they are a mutable object type.

* In practice, you won't see many lists written out like this in list-processing programs. It's more common to see code that processes lists constructed dynamically (at runtime). In fact, although it's important to master literal syntax, most data structures in Python are built by running program code at runtime.

Lists in Action

Perhaps the best way to understand lists is to see them at work. Let's once again turn to some simple interpreter interactions to illustrate the operations in [Table 8-1](#).

Basic List Operations

Because they are sequences, lists support many of the same operations as strings. For example, lists respond to the `+` and `*` operators much like strings—they mean concatenation and repetition here too, except that the result is a new list, not a string:

```
% python
>>> len([1, 2, 3])           # Length
3
>>> [1, 2, 3] + [4, 5, 6]    # Concatenation
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4              # Repetition
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

Although the `+` operator works the same for lists and strings, it's important to know that it expects the same sort of sequence on both sides—otherwise, you get a type error when the code runs. For instance, you cannot concatenate a list and a string unless you first convert the list to a string (using tools such as `str` or `%` formatting) or convert the string to a list (the `list` built-in function does the trick):

```
>>> str([1, 2]) + "34"      # Same as "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list("34")     # Same as [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

List Iteration and Comprehensions

More generally, lists respond to all the sequence operations we used on strings in the prior chapter, including iteration tools:

```
>>> 3 in [1, 2, 3]          # Membership
True
>>> for x in [1, 2, 3]:
...     print(x, end=' ')   # Iteration
...
1 2 3
```

We will talk more formally about `for` iteration and the `range` built-ins in [Chapter 13](#), because they are related to statement syntax. In short, `for` loops step through items in any sequence from left to right, executing one or more statements for each item.

The last items in [Table 8-1](#), list comprehensions and `map` calls, are covered in more detail in [Chapter 14](#) and expanded on in [Chapter 20](#). Their basic operation is straightforward, though—as introduced in [Chapter 4](#), list comprehensions are a way to build a new list

by applying an expression to each item in a sequence, and are close relatives to `for` loops:

```
>>> res = [c * 4 for c in 'SPAM']           # List comprehensions
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

This expression is functionally equivalent to a `for` loop that builds up a list of results manually, but as we'll learn in later chapters, list comprehensions are simpler to code and faster to run today:

```
>>> res = []
>>> for c in 'SPAM':                         # List comprehension equivalent
...     res.append(c * 4)
...
>>> res
['SSSS', 'PPPP', 'AAAA', 'MMMM']
```

As also introduced in [Chapter 4](#), the `map` built-in function does similar work, but applies a function to items in a sequence and collects all the results in a new list:

```
>>> list(map(abs, [-1, -2, 0, 1, 2]))        # map function across sequence
[1, 2, 0, 1, 2]
```

Because we're not quite ready for the full iteration story, we'll postpone further details for now, but watch for a similar comprehension expression for dictionaries later in this chapter.

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings. However, the result of indexing a list is whatever type of object lives at the offset you specify, while slicing a list always returns a new list:

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                                     # Offsets start at zero
'SPAM!'
>>> L[-2]                                   # Negative: count from the right
'Spam'
>>> L[1:]                                   # Slicing fetches sections
['Spam', 'SPAM!']
```

One note here: because you can nest lists and other object types within lists, you will sometimes need to string together index operations to go deeper into a data structure. For example, one of the simplest ways to represent matrixes (multidimensional arrays) in Python is as lists with nested sublists. Here's a basic 3×3 two-dimensional list-based array:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

With one index, you get an entire row (really, a nested sublist), and with two, you get an item within the row:

```

>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
>>> matrix = [[1, 2, 3],
...           [4, 5, 6],
...           [7, 8, 9]]
>>> matrix[1][1]
5

```

Notice in the preceding interaction that lists can naturally span multiple lines if you want them to because they are contained by a pair of brackets (more on syntax in the next part of the book). Later in this chapter, you'll also see a dictionary-based matrix representation. For high-powered numeric work, the NumPy extension mentioned in [Chapter 5](#) provides other ways to handle matrixes.

Changing Lists In-Place

Because lists are mutable, they support operations that change a list object *in-place*. That is, the operations in this section all modify the list object directly, without requiring that you make a new copy, as you had to for strings. Because Python deals only in object references, this distinction between changing an object in-place and creating a new object matters—as discussed in [Chapter 6](#), if you change an object in-place, you might impact more than one reference to it at the same time.

Index and slice assignments

When using a list, you can change its contents by assigning to either a particular item (offset) or an entire section (slice):

```

>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'                                     # Index assignment
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']                           # Slice assignment: delete+insert
>>> L                                                  # Replaces items 0,1
['eat', 'more', 'SPAM!']

```

Both index and slice assignments are in-place changes—they modify the subject list directly, rather than generating a new list object for the result. Index assignment in Python works much as it does in C and most other languages: Python replaces the object reference at the designated offset with a new one.

Slice assignment, the last operation in the preceding example, replaces an entire section of a list in a single step. Because it can be a bit complex, it is perhaps best thought of as a combination of two steps:

1. *Deletion.* The slice you specify to the left of the = is deleted.
2. *Insertion.* The new items contained in the object to the right of the = are inserted into the list on the left, at the place where the old slice was deleted.[†]

This isn't what really happens, but it tends to help clarify why the number of items inserted doesn't have to match the number of items deleted. For instance, given a list `L` that has the value `[1,2,3]`, the assignment `L[1:2]=[4,5]` sets `L` to the list `[1,4,5,3]`. Python first deletes the `2` (a one-item slice), then inserts the `4` and `5` where the deleted `2` used to be. This also explains why `L[1:2]=[]` is really a deletion operation—Python deletes the slice (the item at offset 1), and then inserts nothing.

In effect, slice assignment replaces an entire section, or “column,” all at once. Because the length of the sequence being assigned does not have to match the length of the slice being assigned to, slice assignment can be used to replace (by overwriting), expand (by inserting), or shrink (by deleting) the subject list. It's a powerful operation, but frankly, one that you may not see very often in practice. There are usually more straightforward ways to replace, insert, and delete (concatenation and the `insert`, `pop`, and `remove` list methods, for example), which Python programmers tend to prefer in practice.

List method calls

Like strings, Python list objects also support type-specific method calls, many of which change the subject list in-place:

```
>>> L.append('please')           # Append method call: add item at end
>>> L
['eat', 'more', 'SPAM!', 'please']
>>> L.sort()                     # Sort list items ('S' < 'e')
>>> L
['SPAM!', 'eat', 'more', 'please']
```

Methods were introduced in [Chapter 7](#). In brief, they are functions (really, attributes that reference functions) that are associated with particular objects. Methods provide type-specific tools; the list methods presented here, for instance, are generally available only for lists.

Perhaps the most commonly used list method is `append`, which simply tacks a single item (object reference) onto the end of the list. Unlike concatenation, `append` expects you to pass in a single object, not a list. The effect of `L.append(X)` is similar to `L+[X]`, but while the former changes `L` in-place, the latter makes a new list.[‡]

Another commonly seen method, `sort`, orders a list in-place; it uses Python standard comparison tests (here, string comparisons), and by default sorts in ascending order.

[†] This description needs elaboration when the value and the slice being assigned overlap: `L[2:5]=L[3:6]`, for instance, works fine because the value to be inserted is fetched before the deletion happens on the left.

[‡] Unlike + concatenation, `append` doesn't have to generate new objects, so it's usually faster. You can also mimic `append` with clever slice assignments: `L[len(L):]=[X]` is like `L.append(X)`, and `L[:0]=[X]` is like appending at the front of a list. Both delete an empty slice and insert `X`, changing `L` in-place quickly, like `append`.

You can modify sort behavior by passing in *keyword arguments*—a special “name=value” syntax in function calls that specifies passing by name and is often used for giving configuration options. In sorts, the `key` argument gives a one-argument function that returns the value to be used in sorting, and the `reverse` argument allows sorts to be made in descending instead of ascending order:

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                                     # Sort with mixed case
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                       # Normalize to lowercase
>>> L
['abc', 'ABD', 'aBe']
>>>
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)         # Change sort order
>>> L
['aBe', 'ABD', 'abc']
```

The sort `key` argument might also be useful when sorting lists of dictionaries, to pick out a sort key by indexing each dictionary. We’ll study dictionaries later in this chapter, and you’ll learn more about keyword function arguments in [Part IV](#).



Comparison and sorts in 3.0: In Python 2.6 and earlier, comparisons of differently typed objects (e.g., a string and a list) work—the language defines a fixed ordering among different types, which is deterministic, if not aesthetically pleasing. That is, the ordering is based on the names of the types involved: all integers are less than all strings, for example, because “int” is less than “str”. Comparisons never automatically convert types, except when comparing numeric type objects.

In Python 3.0, this has changed: comparison of mixed types raises an exception instead of falling back on the fixed cross-type ordering. Because sorting uses comparisons internally, this means that `[1, 2, 'spam'].sort()` succeeds in Python 2.X but will raise an exception in Python 3.0 and later.

Python 3.0 also no longer supports passing in an arbitrary *comparison function* to sorts, to implement different orderings. The suggested work-around is to use the `key=func` keyword argument to code value transformations during the sort, and use the `reverse=True` keyword argument to change the sort order to descending. These were the typical uses of comparison functions in the past.

One warning here: beware that `append` and `sort` change the associated list object in-place, but don’t return the list as a result (technically, they both return a value called `None`). If you say something like `L=L.append(X)`, you won’t get the modified value of `L` (in fact, you’ll lose the reference to the list altogether!). When you use attributes such as `append` and `sort`, objects are changed as a side effect, so there’s no reason to reassign.

Partly because of such constraints, sorting is also available in recent Pythons as a built-in function, which sorts any collection (not just lists) and returns a new list for the result (instead of in-place changes):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L, key=str.lower, reverse=True)      # Sorting built-in
['aBe', 'ABD', 'abc']

>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True)  # Pretransform items: differs!
['abe', 'abd', 'abc']
```

Notice the last example here—we can convert to lowercase prior to the sort with a list comprehension, but the result does not contain the original list’s values as it does with the `key` argument. The latter is applied temporarily during the sort, instead of changing the values to be sorted. As we move along, we’ll see contexts in which the `sorted` built-in can sometimes be more useful than the `sort` method.

Like strings, lists have other methods that perform other specialized operations. For instance, `reverse` reverses the list in-place, and the `extend` and `pop` methods insert multiple items at the end of and delete an item from the end of the list, respectively. There is also a `reversed` built-in function that works much like `sorted`, but it must be wrapped in a `list` call because it’s an iterator (more on iterators later):

```
>>> L = [1, 2]
>>> L.extend([3,4,5])      # Add many items at end
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                # Delete and return last item
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()            # In-place reversal method
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))      # Reversal built-in with a result
[1, 2, 3, 4]
```

In some types of programs, the list `pop` method used here is often used in conjunction with `append` to implement a quick last-in-first-out (LIFO) *stack* structure. The end of the list serves as the top of the stack:

```
>>> L = []
>>> L.append(1)             # Push onto stack
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()                # Pop off stack
2
>>> L
[1]
```

The `pop` method also accepts an optional offset of the item to be deleted and returned (the default is the last item). Other list methods remove an item by value (`remove`), insert an item at an offset (`insert`), search for an item's offset (`index`), and more:

```
>>> L = ['spam', 'eggs', 'ham']
>>> L.index('eggs')           # Index of an object
1
>>> L.insert(1, 'toast')      # Insert at position
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove('eggs')         # Delete by value
>>> L
['spam', 'toast', 'ham']
>>> L.pop(1)                 # Delete by position
'toast'
>>> L
['spam', 'ham']
```

See other documentation sources or experiment with these calls interactively on your own to learn more about list methods.

Other common list operations

Because lists are mutable, you can use the `del` statement to delete an item or section in-place:

```
>>> L
['SPAM!', 'eat', 'more', 'please']
>>> del L[0]                 # Delete one item
>>> L
['eat', 'more', 'please']
>>> del L[1:]                # Delete an entire section
>>> L                        # Same as L[1:] = []
['eat']
```

Because slice assignment is a deletion plus an insertion, you can also delete a section of a list by assigning an empty list to a slice (`L[i:j]=[]`); Python deletes the slice named on the left, and then inserts nothing. Assigning an empty list to an index, on the other hand, just stores a reference to the empty list in the specified slot, rather than deleting it:

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

Although all the operations just discussed are typical, there are additional list methods and operations not illustrated here (including methods for inserting and searching). For a comprehensive and up-to-date list of type tools, you should always consult

Python’s manuals, Python’s `dir` and `help` functions (which we first met in [Chapter 4](#)), or one of the reference texts mentioned in the Preface.

I’d also like to remind you one more time that all the in-place change operations discussed here work only for mutable objects: they won’t work on strings (or tuples, discussed in [Chapter 9](#)), no matter how hard you try. Mutability is an inherent property of each object type.

Dictionaries

Apart from lists, *dictionaries* are perhaps the most flexible built-in data type in Python. If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by *key*, instead of by positional offset.

Being a built-in type, dictionaries can replace many of the searching algorithms and data structures you might have to implement manually in lower-level languages—indexing a dictionary is a very fast search operation. Dictionaries also sometimes do the work of records and symbol tables used in other languages, can represent sparse (mostly empty) data structures, and much more. Here’s a rundown of their main properties. Python dictionaries are:

Accessed by key, not offset

Dictionaries are sometimes called *associative arrays* or *hashes*. They associate a set of values with keys, so you can fetch an item out of a dictionary using the key under which you originally stored it. You use the same indexing operation to get components in a dictionary as you do in a list, but the index takes the form of a key, not a relative offset.

Unordered collections of arbitrary objects

Unlike in a list, items stored in a dictionary aren’t kept in any particular order; in fact, Python randomizes their left-to-right order to provide quick lookup. Keys provide the symbolic (not physical) locations of items in a dictionary.

Variable-length, heterogeneous, and arbitrarily nestable

Like lists, dictionaries can grow and shrink in-place (without new copies being made), they can contain objects of any type, and they support nesting to any depth (they can contain lists, other dictionaries, and so on).

Of the category “mutable mapping”

Dictionaries can be changed in-place by assigning to indexes (they are mutable), but they don’t support the sequence operations that work on strings and lists. Because dictionaries are unordered collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don’t make sense. Instead, dictionaries are the only built-in representatives of the mapping type category (objects that map keys to values).

Tables of object references (hash tables)

If lists are arrays of object references that support access by position, dictionaries are unordered tables of object references that support access by key. Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, dictionaries store object references (not copies).

Table 8-2 summarizes some of the most common and representative dictionary operations (again, see the library manual or run a `dir(dict)` or `help(dict)` call for a complete list—`dict` is the name of the type). When coded as a literal expression, a dictionary is written as a series of *key:value* pairs, separated by commas, enclosed in curly braces.[§] An empty dictionary is an empty set of braces, and dictionaries can be nested by writing one as a value inside another dictionary, or within a list or tuple.

Table 8-2. Common dictionary literals and operations

Operation	Interpretation
<code>D = {}</code>	Empty dictionary
<code>D = {'spam': 2, 'eggs': 3}</code>	Two-item dictionary
<code>D = {'food': {'ham': 1, 'egg': 2}}</code>	Nesting
<code>D = dict(name='Bob', age=40)</code>	Alternative construction techniques:
<code>D = dict(zip(keylist, valslst))</code>	keywords, zipped pairs, key lists
<code>D = dict.fromkeys(['a', 'b'])</code>	
<code>D['eggs']</code>	Indexing by key
<code>D['food']['ham']</code>	
<code>'eggs' in D</code>	Membership: key present test
<code>D.keys()</code>	Methods: keys,
<code>D.values()</code>	values,
<code>D.items()</code>	keys+values,
<code>D.copy()</code>	copies,
<code>D.get(key, default)</code>	defaults,
<code>D.update(D2)</code>	merge,
<code>D.pop(key)</code>	delete, etc.
<code>len(D)</code>	Length: number of stored entries
<code>D[key] = 42</code>	Adding/changing keys

[§] As with lists, you won't often see dictionaries constructed using literals. Lists and dictionaries are grown in different ways, though. As you'll see in the next section, dictionaries are typically built up by assigning to new keys at runtime; this approach fails for lists (lists are commonly grown with `append` instead).

Operation	Interpretation
<code>del D[key]</code>	Deleting entries by key
<code>list(D.keys())</code>	Dictionary views (Python 3.0)
<code>D1.keys() & D2.keys()</code>	
<code>D = {x: x*2 for x in range(10)}</code>	Dictionary comprehensions (Python 3.0)

Dictionaries in Action

As [Table 8-2](#) suggests, dictionaries are indexed by key, and nested dictionary entries are referenced by a series of indexes (keys in square brackets). When Python creates a dictionary, it stores its items in any left-to-right order it chooses; to fetch a value back, you supply the key with which it is associated, not its relative position. Let's go back to the interpreter to get a feel for some of the dictionary operations in [Table 8-2](#).

Basic Dictionary Operations

In normal operation, you create dictionaries with literals and store and access items by key with indexing:

```
% python
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}      # Make a dictionary
>>> D['spam']                                  # Fetch a value by key
2
>>> D                                          # Order is scrambled
{'eggs': 3, 'ham': 1, 'spam': 2}
```

Here, the dictionary is assigned to the variable `D`; the value of the key `'spam'` is the integer 2, and so on. We use the same square bracket syntax to index dictionaries by key as we did to index lists by offset, but here it means access by key, not by position.

Notice the end of this example: the left-to-right order of keys in a dictionary will almost always be different from what you originally typed. This is on purpose: to implement fast key lookup (a.k.a. hashing), keys need to be reordered in memory. That's why operations that assume a fixed left-to-right order (e.g., slicing, concatenation) do not apply to dictionaries; you can fetch values only by key, not by position.

The built-in `len` function works on dictionaries, too; it returns the number of items stored in the dictionary or, equivalently, the length of its keys list. The dictionary in membership operator allows you to test for key existence, and the `keys` method returns all the keys in the dictionary. The latter of these can be useful for processing dictionaries sequentially, but you shouldn't depend on the order of the keys list. Because the `keys` result can be used as a normal list, however, it can always be sorted if order matters (more on sorting and dictionaries later):

```
>>> len(D)                                    # Number of entries in dictionary
3
>>> 'ham' in D                                # Key membership test alternative
```

```
True
>>> list(D.keys())                                # Create a new list of my keys
['eggs', 'ham', 'spam']
```

Notice the second expression in this listing. As mentioned earlier, the `in` membership test used for strings and lists also works on dictionaries—it checks whether a key is stored in the dictionary. Technically, this works because dictionaries define *iterators* that step through their keys lists. Other types provide iterators that reflect their common uses; files, for example, have iterators that read line by line. We’ll discuss iterators in Chapters 14 and 20.

Also note the syntax of the last example in this listing. We have to enclose it in a `list` call in Python 3.0 for similar reasons—`keys` in 3.0 returns an iterator, instead of a physical list. The `list` call forces it to produce all its values at once so we can print them. In 2.6, `keys` builds and returns an actual list, so the `list` call isn’t needed to display results. More on this later in this chapter.



The order of keys in a dictionary is arbitrary and can change from release to release, so don’t be alarmed if your dictionaries print in a different order than shown here. In fact, the order has changed for me too—I’m running all these examples with Python 3.0, but their keys had a different order in an earlier edition when displayed. You shouldn’t depend on dictionary key ordering, in either programs or books!

Changing Dictionaries In-Place

Let’s continue with our interactive session. Dictionaries, like lists, are mutable, so you can change, expand, and shrink them in-place without making new dictionaries: simply assign a value to a key to change or create an entry. The `del` statement works here, too; it deletes the entry associated with the key specified as an index. Notice also the nesting of a list inside a dictionary in this example (the value of the key ‘ham’). All collection data types in Python can nest inside each other arbitrarily:

```
>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}

>>> D['ham'] = ['grill', 'bake', 'fry']           # Change entry
>>> D
{'eggs': 3, 'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>> del D['eggs']                                   # Delete entry
>>> D
{'ham': ['grill', 'bake', 'fry'], 'spam': 2}

>>> D['brunch'] = 'Bacon'                           # Add new entry
>>> D
{'brunch': 'Bacon', 'ham': ['grill', 'bake', 'fry'], 'spam': 2}
```


As with lists, assigning to an existing index in a dictionary changes its associated value. Unlike with lists, however, whenever you assign a *new* dictionary key (one that hasn't been assigned before) you create a new entry in the dictionary, as was done in the previous example for the key 'brunch'. This doesn't work for lists because Python considers an offset beyond the end of a list out of bounds and throws an error. To expand a list, you need to use tools such as the `append` method or slice assignment instead.

More Dictionary Methods

Dictionary methods provide a variety of tools. For instance, the dictionary `values` and `items` methods return the dictionary's values and (*key,value*) pair tuples, respectively (as with keys, wrap them in a `list` call in Python 3.0 to collect their values for display):

```
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}
>>> list(D.values())
[3, 1, 2]
>>> list(D.items())
[('eggs', 3), ('ham', 1), ('spam', 2)]
```

Such lists are useful in loops that need to step through dictionary entries one by one. Fetching a nonexistent key is normally an error, but the `get` method returns a default value (`None`, or a passed-in default) if the key doesn't exist. It's an easy way to fill in a default for a key that isn't present and avoid a missing-key error:

```
>>> D.get('spam')                # A key that is there
2
>>> print(D.get('toast'))         # A key that is missing
None
>>> D.get('toast', 88)
88
```

The `update` method provides something similar to concatenation for dictionaries, though it has nothing to do with left-to-right ordering (again, there is no such thing in dictionaries). It merges the keys and values of one dictionary into another, blindly overwriting values of the same key:

```
>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}
>>> D2 = {'toast': 4, 'muffin': 5}
>>> D.update(D2)
>>> D
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
```

Finally, the dictionary `pop` method deletes a key from a dictionary and returns the value it had. It's similar to the list `pop` method, but it takes a key instead of an optional position:

```
# pop a dictionary by key
>>> D
{'toast': 4, 'muffin': 5, 'eggs': 3, 'ham': 1, 'spam': 2}
>>> D.pop('muffin')
```

```

5
>>> D.pop('toast')                # Delete and return from a key
4
>>> D
{'eggs': 3, 'ham': 1, 'spam': 2}

# pop a list by position
>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()                        # Delete and return from the end
'dd'
>>> L
['aa', 'bb', 'cc']
>>> L.pop(1)                      # Delete from a specific position
'bb'
>>> L
['aa', 'cc']

```

Dictionaries also provide a `copy` method; we'll discuss this in [Chapter 9](#), as it's a way to avoid the potential side effects of shared references to the same dictionary. In fact, dictionaries come with many more methods than those listed in [Table 8-2](#); see the Python library manual or other documentation sources for a comprehensive list.

A Languages Table

Let's look at a more realistic dictionary example. The following example creates a table that maps programming language names (the keys) to their creators (the values). You fetch creator names by indexing on language names:

```

>>> table = {'Python': 'Guido van Rossum',
...          'Perl':    'Larry Wall',
...          'Tcl':     'John Ousterhout' }
>>>
>>> language = 'Python'
>>> creator  = table[language]
>>> creator
'Guido van Rossum'

>>> for lang in table:                # Same as: for lang in table.keys()
...     print(lang, '\t', table[lang])
...
Tcl      John Ousterhout
Python   Guido van Rossum
Perl     Larry Wall

```

The last command uses a `for` loop, which we haven't covered in detail yet. If you aren't familiar with `for` loops, this command simply iterates through each key in the table and prints a tab-separated list of keys and their values. We'll learn more about `for` loops in [Chapter 13](#).

Dictionaries aren't sequences like lists and strings, but if you need to step through the items in a dictionary, it's easy—calling the dictionary `keys` method returns all stored

keys, which you can iterate through with a `for`. If needed, you can index from key to value inside the `for` loop, as was done in this code.

In fact, Python also lets you step through a dictionary's keys list without actually calling the `keys` method in most `for` loops. For any dictionary `D`, saying `for key in D:` works the same as saying the complete `for key in D.keys():`. This is really just another instance of the iterators mentioned earlier, which allow the `in` membership operator to work on dictionaries as well (more on iterators later in this book).

Dictionary Usage Notes

Dictionaries are fairly straightforward tools once you get the hang of them, but here are a few additional pointers and reminders you should be aware of when using them:

- **Sequence operations don't work.** Dictionaries are mappings, not sequences; because there's no notion of ordering among their items, things like concatenation (an ordered joining) and slicing (extracting a contiguous section) simply don't apply. In fact, Python raises an error when your code runs if you try to do such things.
- **Assigning to new indexes adds entries.** Keys can be created when you write a dictionary literal (in which case they are embedded in the literal itself), or when you assign values to new keys of an existing dictionary object. The end result is the same.
- **Keys need not always be strings.** Our examples so far have used strings as keys, but any other *immutable* objects (i.e., not lists) work just as well. For instance, you can use integers as keys, which makes the dictionary look much like a list (when indexing, at least). Tuples are sometimes used as dictionary keys too, allowing for compound key values. Class instance objects (discussed in [Part VI](#)) can also be used as keys, as long as they have the proper protocol methods; roughly, they need to tell Python that their values are hashable and won't change, as otherwise they would be useless as fixed keys.

Using dictionaries to simulate flexible lists

The last point in the prior list is important enough to demonstrate with a few examples. When you use lists, it is illegal to assign to an offset that is off the end of the list:

```
>>> L = []
>>> L[99] = 'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Although you can use repetition to preallocate as big a list as you'll need (e.g., `[0]*100`), you can also do something that looks similar with dictionaries that does not require such space allocations. By using integer keys, dictionaries can emulate lists that seem to grow on offset assignment:

```
>>> D = {}
>>> D[99] = 'spam'
>>> D[99]
'spam'
>>> D
{99: 'spam'}
```

Here, it looks as if `D` is a 100-item list, but it's really a dictionary with a single entry; the value of the key `99` is the string `'spam'`. You can access this structure with offsets much like a list, but you don't have to allocate space for all the positions you might ever need to assign values to in the future. When used like this, dictionaries are like more flexible equivalents of lists.

Using dictionaries for sparse data structures

In a similar way, dictionary keys are also commonly leveraged to implement *sparse* data structures—for example, multidimensional arrays where only a few positions have values stored in them:

```
>>> Matrix = {}
>>> Matrix[(2, 3, 4)] = 88
>>> Matrix[(7, 8, 9)] = 99
>>>
>>> X = 2; Y = 3; Z = 4           # ; separates statements
>>> Matrix[(X, Y, Z)]
88
>>> Matrix
{(2, 3, 4): 88, (7, 8, 9): 99}
```

Here, we've used a dictionary to represent a three-dimensional array that is empty except for the two positions `(2,3,4)` and `(7,8,9)`. The keys are *tuples* that record the coordinates of nonempty slots. Rather than allocating a large and mostly empty three-dimensional matrix to hold these values, we can use a simple two-item dictionary. In this scheme, accessing an empty slot triggers a nonexistent key exception, as these slots are not physically stored:

```
>>> Matrix[(2,3,6)]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: (2, 3, 6)
```

Avoiding missing-key errors

Errors for nonexistent key fetches are common in sparse matrixes, but you probably won't want them to shut down your program. There are at least three ways to fill in a default value instead of getting such an error message—you can test for keys ahead of time in `if` statements, use a `try` statement to catch and recover from the exception explicitly, or simply use the dictionary `get` method shown earlier to provide a default for keys that do not exist:

```
>>> if (2,3,6) in Matrix:           # Check for key before fetch
...     print(Matrix[(2,3,6)])      # See Chapter 12 for if/else
```

```

... else:
...     print(0)
...
0
>>> try:
...     print(Matrix[(2,3,6)])          # Try to index
... except KeyError:                    # Catch and recover
...     print(0)                        # See Chapter 33 for try/except
...
0
>>> Matrix.get((2,3,4), 0)              # Exists; fetch and return
88
>>> Matrix.get((2,3,6), 0)              # Doesn't exist; use default arg
0

```

Of these, the `get` method is the most concise in terms of coding requirements; we'll study the `if` and `try` statements in more detail later in this book.

Using dictionaries as “records”

As you can see, dictionaries can play many roles in Python. In general, they can replace search data structures (because indexing by key is a search operation) and can represent many types of structured information. For example, dictionaries are one of many ways to describe the properties of an item in your program's domain; that is, they can serve the same role as “records” or “structs” in other languages.

The following, for example, fills out a dictionary by assigning to new keys over time:

```

>>> rec = {}
>>> rec['name'] = 'mel'
>>> rec['age'] = 45
>>> rec['job'] = 'trainer/writer'
>>>
>>> print(rec['name'])
mel

```

Especially when nested, Python's built-in data types allow us to easily represent structured information. This example again uses a dictionary to capture object properties, but it codes it all at once (rather than assigning to each key separately) and nests a list and a dictionary to represent structured property values:

```

>>> mel = {'name': 'Mark',
...        'jobs': ['trainer', 'writer'],
...        'web': 'www.rmi.net/~lutz',
...        'home': {'state': 'CO', 'zip': 80513}}

```

To fetch components of nested objects, simply string together indexing operations:

```

>>> mel['name']
'Mark'
>>> mel['jobs']
['trainer', 'writer']
>>> mel['jobs'][1]
'writer'

```

```
>>> mel['home']['zip']
80513
```

Although we'll learn in [Part VI](#) that classes (which group both data and logic) can be better in this record role, dictionaries are an easy-to-use tool for simpler requirements.

Why You Will Care: Dictionary Interfaces

Dictionaries aren't just a convenient way to store information by key in your programs—some Python extensions also present interfaces that look like and work the same as dictionaries. For instance, Python's interface to DBM access-by-key files looks much like a dictionary that must be opened. Strings are stored and fetched using key indexes:

```
import anydbm
file = anydbm.open("filename") # Link to file
file['key'] = 'data'           # Store data by key
data = file['key']              # Fetch data by key
```

In [Chapter 27](#), you'll see that you can store entire Python objects this way, too, if you replace `anydbm` in the preceding code with `shelve` (shelves are access-by-key databases of persistent Python objects). For Internet work, Python's CGI script support also presents a dictionary-like interface. A call to `cgi.FieldStorage` yields a dictionary-like object with one entry per input field on the client's web page:

```
import cgi
form = cgi.FieldStorage() # Parse form data
if 'name' in form:
    showReply('Hello, ' + form['name'].value)
```

All of these, like dictionaries, are instances of mappings. Once you learn dictionary interfaces, you'll find that they apply to a variety of built-in tools in Python.

Other Ways to Make Dictionaries

Finally, note that because dictionaries are so useful, more ways to build them have emerged over time. In Python 2.3 and later, for example, the last two calls to the `dict` constructor (really, type name) shown here have the same effect as the literal and key-assignment forms above them:

```
{'name': 'mel', 'age': 45} # Traditional literal expression

D = {}                     # Assign by keys dynamically
D['name'] = 'mel'
D['age'] = 45

dict(name='mel', age=45)   # dict keyword argument form

dict([('name', 'mel'), ('age', 45)]) # dict key/value tuples form
```

All four of these forms create the same two-key dictionary, but they are useful in differing circumstances:

- The first is handy if you can spell out the entire dictionary ahead of time.
- The second is of use if you need to create the dictionary one field at a time on the fly.
- The third involves less typing than the first, but it requires all keys to be strings.
- The last is useful if you need to build up keys and values as sequences at runtime.

We met keyword arguments earlier when sorting; the third form illustrated in this code listing has become especially popular in Python code today, since it has less syntax (and hence there is less opportunity for mistakes). As suggested previously in [Table 8-2](#), the last form in the listing is also commonly used in conjunction with the `zip` function, to combine separate lists of keys and values obtained dynamically at runtime (parsed out of a data file’s columns, for instance). More on this option in the next section.

Provided all the key’s values are the same initially, you can also create a dictionary with this special form—simply pass in a list of keys and an initial value for all of the values (the default is `None`):

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

Although you could get by with just literals and key assignments at this point in your Python career, you’ll probably find uses for all of these dictionary-creation forms as you start applying them in realistic, flexible, and dynamic Python programs.

The listings in this section document the various ways to create dictionaries in both Python 2.6 and 3.0. However, there is yet another way to create dictionaries, available only in Python 3.0 (and later): the *dictionary comprehension* expression. To see how this last form looks, we need to move on to the next section.

Dictionary Changes in Python 3.0

This chapter has so far focused on dictionary basics that span releases, but the dictionary’s functionality has mutated in Python 3.0. If you are using Python 2.X code, you may come across some dictionary tools that either behave differently or are missing altogether in 3.0. Moreover, 3.0 coders have access to additional dictionary tools not available in 2.X. Specifically, dictionaries in 3.0:

- Support a new dictionary comprehension expression, a close cousin to list and set comprehensions
- Return iterable views instead of lists for the methods `D.keys`, `D.values`, and `D.items`
- Require new coding styles for scanning by sorted keys, because of the prior point
- No longer support relative magnitude comparisons directly—compare manually instead
- No longer have the `D.has_key` method—the `in` membership test is used instead

Let’s take a look at what’s new in 3.0 dictionaries.

Dictionary comprehensions

As mentioned at the end of the prior section, dictionaries in 3.0 can also be created with dictionary comprehensions. Like the set comprehensions we met in [Chapter 5](#), dictionary comprehensions are available only in 3.0 (not in 2.6). Like the longstanding list comprehensions we met briefly in [Chapter 4](#) and earlier in this chapter, they run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary. A loop variable allows the comprehension to use loop iteration values along the way.

For example, a standard way to initialize a dictionary dynamically in both 2.6 and 3.0 is to zip together its keys and values and pass the result to the `dict` call. As we'll learn in more detail in [Chapter 13](#), the `zip` function is a way to construct a dictionary from key and value lists in a single call. If you cannot predict the set of keys and values in your code, you can always build them up as lists and zip them together:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))      # Zip together keys and values
[('a', 1), ('b', 2), ('c', 3)]

>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))  # Make a dict from zip result
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

In Python 3.0, you can achieve the same effect with a dictionary comprehension expression. The following builds a new dictionary with a key/value pair for every such pair in the `zip` result (it reads almost the same in Python, but with a bit more formality):

```
C:\misc> c:\python30\python                  # Use a dict comprehension

>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

Comprehensions actually require more code in this case, but they are also more general than this example implies—we can use them to map a single stream of values to dictionaries as well, and keys can be computed with expressions just like values:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}      # Or: range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}

>>> D = {c: c * 4 for c in 'SPAM'}            # Loop over any iterable
>>> D
{'A': 'AAAA', 'P': 'PPPP', 'S': 'SSSS', 'M': 'MMMM'}

>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'eggs': 'EGGS!', 'ham': 'HAM!', 'spam': 'SPAM!'}
```

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the `fromkeys` method we met at the end of the preceding section:


```

>>> D = dict.fromkeys(['a', 'b', 'c'], 0)      # Initialize dict from keys
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = {k:0 for k in ['a', 'b', 'c']}         # Same, but with a comprehension
>>> D
{'a': 0, 'c': 0, 'b': 0}

>>> D = dict.fromkeys('spam')                  # Other iterators, default value
>>> D
{'a': None, 'p': None, 's': None, 'm': None}

>>> D = {k: None for k in 'spam'}
>>> D
{'a': None, 'p': None, 's': None, 'm': None}

```

Like related tools, dictionary comprehensions support additional syntax not shown here, including nested loops and `if` clauses. Unfortunately, to truly understand dictionary comprehensions, we need to also know more about iteration statements and concepts in Python, and we don't yet have enough information to address that story well. We'll learn much more about all flavors of comprehensions (list, set, and dictionary) in Chapters 14 and 20, so we'll defer further details until later. We'll also study the `zip` built-in we used in this section in more detail in Chapter 13, when we explore for loops.

Dictionary views

In 3.0 the dictionary `keys`, `values`, and `items` methods all return *view objects*, whereas in 2.6 they return actual result lists. View objects are *iterables*, which simply means objects that generate result items one at a time, instead of producing the result list all at once in memory. Besides being iterable, dictionary views also retain the original order of dictionary components, reflect future changes to the dictionary, and may support set operations. On the other hand, they are not lists, and they do not support operations like indexing or the list `sort` method; nor do they display their items when printed.

We'll discuss the notion of iterables more formally in Chapter 14, but for our purposes here it's enough to know that we have to run the results of these three methods through the `list` built-in if we want to apply list operations or display their values:

```

>>> D = dict(a=1, b=2, c=3)
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> K = D.keys()                               # Makes a view object in 3.0, not a list
>>> K
<dict_keys object at 0x026D83C0>
>>> list(K)                                     # Force a real list in 3.0 if needed
['a', 'c', 'b']

>>> V = D.values()                             # Ditto for values and items views
>>> V
<dict_values object at 0x026D8260>

```

```

>>> list(V)
[1, 3, 2]

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> K[0]                                     # List operations fail unless converted
TypeError: 'dict_keys' object does not support indexing
>>> list(K)[0]
'a'

```

Apart from when displaying results at the interactive prompt, you will probably rarely even notice this change, because looping constructs in Python automatically force iterable objects to produce one result on each iteration:

```

>>> for k in D.keys(): print(k)             # Iterators used automatically in loops
...
a
c
b

```

In addition, 3.0 dictionaries still have iterators themselves, which return successive keys—as in 2.6, it’s still often not necessary to call keys directly:

```

>>> for key in D: print(key)                # Still no need to call keys() to iterate
...
a
c
b

```

Unlike 2.X’s list results, though, dictionary views in 3.0 are not carved in stone when created—they *dynamically reflect future changes* made to the dictionary after the view object has been created:

```

>>> D = {'a':1, 'b':2, 'c':3}
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> K = D.keys()
>>> V = D.values()
>>> list(K)                                     # Views maintain same order as dictionary
['a', 'c', 'b']
>>> list(V)
[1, 3, 2]

>>> del D['b']                                 # Change the dictionary in-place
>>> D
{'a': 1, 'c': 3}

>>> list(K)                                     # Reflected in any current view objects
['a', 'c']
>>> list(V)                                     # Not true in 2.X!
[1, 3]

```

Dictionary views and sets

Also unlike 2.X's list results, 3.0's view objects returned by the `keys` method are *set-like* and support common set operations such as intersection and union; `values` views are not, since they aren't unique, but `items` results are if their (*key*, *value*) pairs are unique and hashable. Given that sets behave much like valueless dictionaries (and are even coded in curly braces like dictionaries in 3.0), this is a logical symmetry. Like dictionary keys, set items are unordered, unique, and immutable.

Here is what keys lists look like when used in set operations. In set operations, views may be mixed with other views, sets, and dictionaries (dictionaries are treated the same as their keys views in this context):

```
>>> K | {'x': 4}                                # Keys (and some items) views are set-like
{'a', 'x', 'c'}

>>> V & {'x': 4}
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'
>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'

>>> D = {'a':1, 'b':2, 'c':3}
>>> D.keys() & D.keys()                          # Intersect keys views
{'a', 'c', 'b'}
>>> D.keys() & {'b'}                             # Intersect keys and set
{'b'}
>>> D.keys() & {'b': 1}                          # Intersect keys and dict
{'b'}
>>> D.keys() | {'b', 'c', 'd'}                  # Union keys and set
{'a', 'c', 'b', 'd'}
```

Dictionary items views are set-like too if they are hashable—that is, if they contain only immutable objects:

```
>>> D = {'a': 1}
>>> list(D.items())                             # Items set-like if hashable
[('a', 1)]
>>> D.items() | D.keys()                       # Union view and view
{('a', 1), 'a'}
>>> D.items() | D                               # dict treated same as its keys
{('a', 1), 'a'}

>>> D.items() | {('c', 3), ('d', 4)}           # Set of key/value pairs
{('a', 1), ('d', 4), ('c', 3)}
>>> dict(D.items()) | {('c', 3), ('d', 4)}      # dict accepts iterable sets too
{'a': 1, 'c': 3, 'd': 4}
```

For more details on set operations in general, see [Chapter 5](#). Now, let's look at three other quick coding notes for 3.0 dictionaries.

Sorting dictionary keys

First of all, because `keys` does not return a list, the traditional coding pattern for scanning a dictionary by sorted keys in 2.X won't work in 3.0. You must either convert to a list manually or use the `sorted` call introduced in [Chapter 4](#) and earlier in this chapter on either a `keys` view or the dictionary itself:

```
>>> D = {'a':1, 'b':2, 'c':3}
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> Ks = D.keys()
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'
# Sorting a view object doesn't work!

>>> Ks = list(Ks)
>>> Ks.sort()
>>> for k in Ks: print(k, D[k])
# Force it to be a list and then sort
...
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> Ks = D.keys()
>>> for k in sorted(Ks): print(k, D[k])
# Or you can use sorted() on the keys
# sorted() accepts any iterable
# sorted() returns its result
...
a 1
b 2
c 3

>>> D
{'a': 1, 'c': 3, 'b': 2}
>>> for k in sorted(D): print(k, D[k])
# Better yet, sort the dict directly
# dict iterators return keys
...
a 1
b 2
c 3
```

Dictionary magnitude comparisons no longer work

Secondly, while in Python 2.6 dictionaries may be compared for relative magnitude directly with `<`, `>`, and so on, in Python 3.0 this no longer works. However, it can be simulated by comparing sorted keys lists manually:

```
sorted(D1.items()) < sorted(D2.items())    # Like 2.6 D1 < D2
```

Dictionary equality tests still work in 3.0, though. Since we'll revisit this in the next chapter in the context of comparisons at large, we'll defer further details here.

The `has_key` method is dead: long live `in`!

Finally, the widely used dictionary `has_key` key presence test method is gone in 3.0. Instead, use the `in` membership expression, or a `get` with a default test (of these, `in` is generally preferred):

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D.has_key('c')
AttributeError: 'dict' object has no attribute 'has_key'           # 2.X only: True/False

>>> 'c' in D
True
>>> 'x' in D
False
>>> if 'c' in D: print('present', D['c'])
...
present 3

>>> print(D.get('c'))
3
>>> print(D.get('x'))
None
>>> if D.get('c') != None: print('present', D['c'])
...
present 3                                     # Preferred in 3.0

...
present 3                                     # Another option
```

If you work in 2.6 and care about 3.0 compatibility, note that the first two changes (comprehensions and views) can only be coded in 3.0, but the last three (`sorted`, manual comparisons, and `in`) can be coded in 2.6 today to ease 3.0 migration in the future.

Chapter Summary

In this chapter, we explored the list and dictionary types—probably the two most common, flexible, and powerful collection types you will see and use in Python code. We learned that the list type supports positionally ordered collections of arbitrary objects, and that it may be freely nested and grown and shrunk on demand. The dictionary type is similar, but it stores items by key instead of by position and does not maintain any reliable left-to-right order among its items. Both lists and dictionaries are mutable, and so support a variety of in-place change operations not available for strings: for example, lists can be grown by `append` calls, and dictionaries by assignment to new keys.

In the next chapter, we will wrap up our in-depth core object type tour by looking at tuples and files. After that, we'll move on to statements that code the logic that processes our objects, taking us another step toward writing complete programs. Before we tackle those topics, though, here are some chapter quiz questions to review.

Test Your Knowledge: Quiz

1. Name two ways to build a list containing five integer zeros.
2. Name two ways to build a dictionary with two keys, 'a' and 'b', each having an associated value of 0.
3. Name four operations that change a list object in-place.
4. Name four operations that change a dictionary object in-place.

Test Your Knowledge: Answers

1. A literal expression like `[0, 0, 0, 0, 0]` and a repetition expression like `[0] * 5` will each create a list of five zeros. In practice, you might also build one up with a loop that starts with an empty list and appends 0 to it in each iteration:
`L.append(0)`. A list comprehension (`[0 for i in range(5)]`) could work here, too, but this is more work than you need to do.
2. A literal expression such as `{ 'a': 0, 'b': 0 }` or a series of assignments like `D = {}`, `D['a'] = 0`, and `D['b'] = 0` would create the desired dictionary. You can also use the newer and simpler-to-code `dict(a=0, b=0)` keyword form, or the more flexible `dict([('a', 0), ('b', 0)])` key/value sequences form. Or, because all the values are the same, you can use the special form `dict.fromkeys('ab', 0)`. In 3.0, you can also use a dictionary comprehension: `{k:0 for k in 'ab'}`.
3. The `append` and `extend` methods grow a list in-place, the `sort` and `reverse` methods order and reverse lists, the `insert` method inserts an item at an offset, the `remove` and `pop` methods delete from a list by value and by position, the `del` statement deletes an item or slice, and index and slice assignment statements replace an item or entire section. Pick any four of these for the quiz.
4. Dictionaries are primarily changed by assignment to a new or existing key, which creates or changes the key's entry in the table. Also, the `del` statement deletes a key's entry, the dictionary `update` method merges one dictionary into another in-place, and `D.pop(key)` removes a key and returns the value it had. Dictionaries also have other, more exotic in-place change methods not listed in this chapter, such as `setdefault`; see reference sources for more details.

Tuples, Files, and Everything Else

This chapter rounds out our in-depth look at the core object types in Python by exploring the *tuple*, a collection of other objects that cannot be changed, and the *file*, an interface to external files on your computer. As you'll see, the tuple is a relatively simple object that largely performs operations you've already learned about for strings and lists. The file object is a commonly used and full-featured tool for processing files; the basic overview of files here is supplemented by larger examples in later chapters.

This chapter also concludes this part of the book by looking at properties common to all the core object types we've met—the notions of equality, comparisons, object copies, and so on. We'll also briefly explore other object types in the Python toolbox; as you'll see, although we've covered all the primary built-in types, the object story in Python is broader than I've implied thus far. Finally, we'll close this part of the book by taking a look at a set of common object type pitfalls and exploring some exercises that will allow you to experiment with the ideas you've learned.

Tuples

The last collection type in our survey is the Python tuple. Tuples construct simple groups of objects. They work exactly like lists, except that tuples can't be changed in-place (they're immutable) and are usually written as a series of items in parentheses, not square brackets. Although they don't support as many methods, tuples share most of their properties with lists. Here's a quick look at the basics. Tuples are:

Ordered collections of arbitrary objects

Like strings and lists, tuples are positionally ordered collections of objects (i.e., they maintain a left-to-right order among their contents); like lists, they can embed any kind of object.

Accessed by offset

Like strings and lists, items in a tuple are accessed by offset (not by key); they support all the offset-based access operations, such as indexing and slicing.

Of the category “immutable sequence”

Like strings and lists, tuples are sequences; they support many of the same operations. However, like strings, tuples are immutable; they don’t support any of the in-place change operations applied to lists.

Fixed-length, heterogeneous, and arbitrarily nestable

Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples can hold any type of object, including other compound objects (e.g., lists, dictionaries, other tuples), and so support arbitrary nesting.

Arrays of object references

Like lists, tuples are best thought of as object reference arrays; tuples store access points to other objects (references), and indexing a tuple is relatively quick.

Table 9-1 highlights common tuple operations. A tuple is written as a series of objects (technically, expressions that generate objects), separated by commas and normally enclosed in parentheses. An empty tuple is just a parentheses pair with nothing inside.

Table 9-1. Common tuple literals and operations

Operation	Interpretation
()	An empty tuple
T = (0,)	A one-item tuple (not an expression)
T = (0, 'Ni', 1.2, 3)	A four-item tuple
T = 0, 'Ni', 1.2, 3	Another four-item tuple (same as prior line)
T = ('abc', ('def', 'ghi'))	Nested tuples
T = tuple('spam')	Tuple of items in an iterable
T[i]	Index, index of index, slice, length
T[i][j]	
T[i:j]	
len(T)	
T1 + T2	Concatenate, repeat
T * 3	
for x in T: print(x)	Iteration, membership
'spam' in T	
[x ** 2 for x in T]	
T.index('Ni')	Methods in 2.6 and 3.0: search, count
T.count('Ni')	

Tuples in Action

As usual, let's start an interactive session to explore tuples at work. Notice in [Table 9-1](#) that tuples do not have all the methods that lists have (e.g., an `append` call won't work here). They do, however, support the usual sequence operations that we saw for both strings and lists:

```
>>> (1, 2) + (3, 4)           # Concatenation
(1, 2, 3, 4)

>>> (1, 2) * 4                # Repetition
(1, 2, 1, 2, 1, 2, 1, 2)

>>> T = (1, 2, 3, 4)          # Indexing, slicing
>>> T[0], T[1:3]
(1, (2, 3))
```

Tuple syntax peculiarities: Commas and parentheses

The second and fourth entries in [Table 9-1](#) merit a bit more explanation. Because parentheses can also enclose expressions (see [Chapter 5](#)), you need to do something special to tell Python when a single object in parentheses is a tuple object and not a simple expression. If you really want a single-item tuple, simply add a trailing comma after the single item, before the closing parenthesis:

```
>>> x = (40)                  # An integer!
>>> x
40
>>> y = (40,)                 # A tuple containing an integer
>>> y
(40,)
```

As a special case, Python also allows you to omit the opening and closing parentheses for a tuple in contexts where it isn't syntactically ambiguous to do so. For instance, the fourth line of [Table 9-1](#) simply lists four items separated by commas. In the context of an assignment statement, Python recognizes this as a tuple, even though it doesn't have parentheses.

Now, some people will tell you to always use parentheses in your tuples, and some will tell you to never use parentheses in tuples (and still others have lives, and won't tell you what to do with your tuples!). The only significant places where the parentheses are *required* are when a tuple is passed as a literal in a function call (where parentheses matter), and when one is listed in a Python 2.X `print` statement (where commas are significant).

For beginners, the best advice is that it's probably easier to use the parentheses than it is to figure out when they are optional. Many programmers (myself included) also find that parentheses tend to aid script readability by making the tuples more explicit, but your mileage may vary.

Conversions, methods, and immutability

Apart from literal syntax differences, tuple operations (the middle rows in [Table 9-1](#)) are identical to string and list operations. The only differences worth noting are that the `+`, `*`, and slicing operations return new *tuples* when applied to tuples, and that tuples don't provide the same methods you saw for strings, lists, and dictionaries. If you want to sort a tuple, for example, you'll usually have to either first convert it to a list to gain access to a sorting method call and make it a mutable object, or use the newer `sorted` built-in that accepts any sequence object (and more):

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)           # Make a list from a tuple's items
>>> tmp.sort()              # Sort the list
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)         # Make a tuple from the list's items
>>> T
('aa', 'bb', 'cc', 'dd')

>>> sorted(T)              # Or use the sorted built-in
['aa', 'bb', 'cc', 'dd']
```

Here, the `list` and `tuple` built-in functions are used to convert the object to a list and then back to a tuple; really, both calls make new objects, but the net effect is like a conversion.

List comprehensions can also be used to convert tuples. The following, for example, makes a list from a tuple, adding 20 to each item along the way:

```
>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]
```

List comprehensions are really sequence operations—they always build new lists, but they may be used to iterate over any sequence objects, including tuples, strings, and other lists. As we'll see later in the book, they even work on some things that are not physically stored sequences—any iterable objects will do, including files, which are automatically read line by line.

Although tuples don't have the same methods as lists and strings, they do have two of their own as of Python 2.6 and 3.0—`index` and `count` works as they do for lists, but they are defined for tuple objects:

```
>>> T = (1, 2, 3, 2, 4, 2)   # Tuple methods in 2.6 and 3.0
>>> T.index(2)              # Offset of first appearance of 2
1
>>> T.index(2, 2)           # Offset of appearance after offset 2
3
>>> T.count(2)              # How many 2s are there?
3
```

Prior to 2.6 and 3.0, tuples have no methods at all—this was an old Python convention for immutable types, which was violated years ago on grounds of practicality with strings, and more recently with both numbers and tuples.

Also, note that the rule about tuple *immutability* applies only to the top level of the tuple itself, not to its contents. A list inside a tuple, for instance, can be changed as usual:

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'spam'                # This fails: can't change tuple itself
TypeError: object doesn't support item assignment

>>> T[1][0] = 'spam'             # This works: can change mutables inside
>>> T
(1, ['spam', 3], 4)
```

For most programs, this one-level-deep immutability is sufficient for common tuple roles. Which, coincidentally, brings us to the next section.

Why Lists and Tuples?

This seems to be the first question that always comes up when teaching beginners about tuples: why do we need tuples if we have lists? Some of the reasoning may be historic; Python’s creator is a mathematician by training, and he has been quoted as seeing a tuple as a simple association of objects and a list as a data structure that changes over time. In fact, this use of the word “tuple” derives from mathematics, as does its frequent use for a row in a relational database table.

The best answer, however, seems to be that the immutability of tuples provides some *integrity*—you can be sure a tuple won’t be changed through another reference elsewhere in a program, but there’s no such guarantee for lists. Tuples, therefore, serve a similar role to “constant” declarations in other languages, though the notion of constantness is associated with objects in Python, not variables.

Tuples can also be used in places that lists cannot—for example, as dictionary keys (see the sparse matrix example in [Chapter 8](#)). Some built-in operations may also require or imply tuples, not lists, though such operations have often been generalized in recent years. As a rule of thumb, lists are the tool of choice for ordered collections that might need to change; tuples can handle the other cases of fixed associations.

Files

You may already be familiar with the notion of files, which are named storage compartments on your computer that are managed by your operating system. The last major built-in object type that we’ll examine on our object types tour provides a way to access those files inside Python programs.

In short, the built-in `open` function creates a Python file object, which serves as a link to a file residing on your machine. After calling `open`, you can transfer strings of data to and from the associated external file by calling the returned file object's methods.

Compared to the types you've seen so far, file objects are somewhat unusual. They're not numbers, sequences, or mappings, and they don't respond to expression operators; they export only methods for common file-processing tasks. Most file methods are concerned with performing input from and output to the external file associated with a file object, but other file methods allow us to seek to a new position in the file, flush output buffers, and so on. [Table 9-2](#) summarizes common file operations.

Table 9-2. Common file operations

Operation	Interpretation
<code>output = open(r'C:\spam', 'w')</code>	Create output file ('w' means write)
<code>input = open('data', 'r')</code>	Create input file ('r' means read)
<code>input = open('data')</code>	Same as prior line ('r' is the default)
<code>aString = input.read()</code>	Read entire file into a single string
<code>aString = input.read(N)</code>	Read up to next N characters (or bytes) into a string
<code>aString = input.readline()</code>	Read next line (including \n newline) into a string
<code>aList = input.readlines()</code>	Read entire file into list of line strings (with \n)
<code>output.write(aString)</code>	Write a string of characters (or bytes) into file
<code>output.writelines(aList)</code>	Write all line strings in a list into file
<code>output.close()</code>	Manual close (done for you when file is collected)
<code>output.flush()</code>	Flush output buffer to disk without closing
<code>anyFile.seek(N)</code>	Change file position to offset N for next operation
<code>for line in open('data'): use line</code>	File iterators read line by line
<code>open('f.txt', encoding='latin-1')</code>	Python 3.0 Unicode text files (str strings)
<code>open('f.bin', 'rb')</code>	Python 3.0 binary bytes files (bytes strings)

Opening Files

To open a file, a program calls the built-in `open` function, with the external filename first, followed by a processing *mode*. The mode is typically the string `'r'` to open for text input (the default), `'w'` to create and open for text output, or `'a'` to open for appending text to the end. The processing mode argument can specify additional options:

- Adding a `b` to the mode string allows for *binary* data (end-of-line translations and 3.0 Unicode encodings are turned off).

- Adding a `+` opens the file for *both* input and output (i.e., you can both read and write to the same file object, often in conjunction with seek operations to reposition in the file).

Both arguments to `open` must be Python strings, and an optional third argument can be used to control output buffering—passing a zero means that output is unbuffered (it is transferred to the external file immediately on a write method call). The external filename argument may include a platform-specific and absolute or relative directory path prefix; without a directory path, the file is assumed to exist in the current working directory (i.e., where the script runs). We’ll cover file fundamentals and explore some basic examples here, but we won’t go into all file-processing mode options; as usual, consult the Python library manual for additional details.

Using Files

Once you make a file object with `open`, you can call its methods to read from or write to the associated external file. In all cases, file text takes the form of strings in Python programs; reading a file returns its text in strings, and text is passed to the write methods as strings. Reading and writing methods come in multiple flavors; [Table 9-2](#) lists the most common. Here are a few fundamental usage notes:

File iterators are best for reading lines

Though the reading and writing methods in the table are common, keep in mind that probably the best way to read lines from a text file today is to not read the file at all—as we’ll see in [Chapter 14](#), files also have an *iterator* that automatically reads one line at a time in a `for` loop, list comprehension, or other iteration context.

Content is strings, not objects

Notice in [Table 9-2](#) that data read from a file always comes back to your script as a string, so you’ll have to convert it to a different type of Python object if a string is not what you need. Similarly, unlike with the `print` operation, Python does not add any formatting and does not convert objects to strings automatically when you write data to a file—you must send an already formatted string. Because of this, the tools we have already met to convert objects to and from strings (e.g., `int`, `float`, `str`, and the string formatting expression and method) come in handy when dealing with files. Python also includes advanced standard library tools for handling generic object storage (such as the `pickle` module) and for dealing with packed binary data in files (such as the `struct` module). We’ll see both of these at work later in this chapter.

close is usually optional

Calling the file `close` method terminates your connection to the external file. As discussed in [Chapter 6](#), in Python an object’s memory space is automatically reclaimed as soon as the object is no longer referenced anywhere in the program. When file objects are reclaimed, Python also automatically closes the files if they are still open (this also happens when a program shuts down). This means you

don't always need to manually close your files, especially in simple scripts that don't run for long. On the other hand, including manual `close` calls can't hurt and is usually a good idea in larger systems. Also, strictly speaking, this auto-close-on-collection feature of files is not part of the language definition, and it may change over time. Consequently, manually issuing file `close` method calls is a good habit to form. (For an alternative way to guarantee automatic file closes, also see this section's later discussion of the file object's *context manager*, used with the new `with/as` statement in Python 2.6 and 3.0.)

Files are buffered and seekable.

The prior paragraph's notes about closing files are important, because closing both frees up operating system resources and flushes output buffers. By default, output files are always buffered, which means that text you write may not be transferred from memory to disk immediately—closing a file, or running its `flush` method, forces the buffered data to disk. You can avoid buffering with extra `open` arguments, but it may impede performance. Python files are also random-access on a byte offset basis—their `seek` method allows your scripts to jump around to read and write at specific locations.

Files in Action

Let's work through a simple example that demonstrates file-processing basics. The following code begins by opening a new text file for output, writing two lines (strings terminated with a newline marker, `\n`), and closing the file. Later, the example opens the same file again in input mode and reads the lines back one at a time with `readline`. Notice that the third `readline` call returns an empty string; this is how Python file methods tell you that you've reached the end of the file (empty lines in the file come back as strings containing just a newline character, not as empty strings). Here's the complete interaction:

```
>>> myfile = open('myfile.txt', 'w')           # Open for text output: create/empty
>>> myfile.write('hello text file\n')          # Write a line of text: string
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close()                             # Flush output buffers to disk

>>> myfile = open('myfile.txt')                 # Open for text input: 'r' is default
>>> myfile.readline()                           # Read the lines back
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()                           # Empty string: end of file
''
```

Notice that file `write` calls return the number of characters written in Python 3.0; in 2.6 they don't, so you won't see these numbers echoed interactively. This example writes each line of text, including its end-of-line terminator, `\n`, as a string; `write`

methods don't add the end-of-line character for us, so we must include it to properly terminate our lines (otherwise the next write will simply extend the current line in the file).

If you want to display the file's content with end-of-line characters interpreted, read the entire file into a string *all at once* with the file object's `read` method and print it:

```
>>> open('myfile.txt').read()           # Read all at once into string
'hello text file\ngoodbye text file\n'

>>> print(open('myfile.txt').read())     # User-friendly display
hello text file
goodbye text file
```

And if you want to scan a text file line by line, *file iterators* are often your best option:

```
>>> for line in open('myfile'):         # Use file iterators, not reads
...     print(line, end='')
...
hello text file
goodbye text file
```

When coded this way, the temporary file object created by `open` will automatically read and return one line on each loop iteration. This form is usually easiest to code, good on memory use, and may be faster than some other options (depending on many variables, of course). Since we haven't reached statements or iterators yet, though, you'll have to wait until [Chapter 14](#) for a more complete explanation of this code.

Text and binary files in Python 3.0

Strictly speaking, the example in the prior section uses text files. In both Python 3.0 and 2.6, file type is determined by the second argument to `open`, the mode string—an included “b” means binary. Python has always supported both text and binary files, but in Python 3.0 there is a sharper distinction between the two:

- *Text files* represent content as normal `str` strings, perform Unicode encoding and decoding automatically, and perform end-of-line translation by default.
- *Binary files* represent content as a special `bytes` string type and allow programs to access file content unaltered.

In contrast, Python 2.6 text files handle both 8-bit text and binary data, and a special string type and file interface (`unicode` strings and `codecs.open`) handles Unicode text. The differences in Python 3.0 stem from the fact that simple and Unicode text have been merged in the normal string type—which makes sense, given that all text is Unicode, including ASCII and other 8-bit encodings.

Because most programmers deal only with ASCII text, they can get by with the basic text file interface used in the prior example, and normal strings. All strings are technically Unicode in 3.0, but ASCII users will not generally notice. In fact, files and strings work the same in 3.0 and 2.6 if your script's scope is limited to such simple forms of text.

If you need to handle internationalized applications or byte-oriented data, though, the distinction in 3.0 impacts your code (usually for the better). In general, you must use `bytes` strings for binary files, and normal `str` strings for text files. Moreover, because text files implement Unicode encodings, you cannot open a binary data file in text mode—decoding its content to Unicode text will likely fail.

Let’s look at an example. When you read a binary data file you get back a `bytes` object—a sequence of small integers that represent absolute byte values (which may or may not correspond to characters), which looks and feels almost exactly like a normal string:

```
>>> data = open('data.bin', 'rb').read()    # Open binary file: rb=read binary
>>> data                                     # bytes string holds binary data
b'\x00\x00\x00\x07spam\x00\x08'
>>> data[4:8]                               # Act like strings
b'spam'
>>> data[0]                                 # But really are small 8-bit integers
115
>>> bin(data[0])                             # Python 3.0 bin() function
'0b1110011'
```

In addition, binary files do not perform any end-of-line translation on data; text files by default map all forms to and from `\n` when written and read and implement Unicode encodings on transfers. Since Unicode and binary data is of marginal interest to many Python programmers, we’ll postpone the full story until [Chapter 36](#). For now, let’s move on to some more substantial file examples.

Storing and parsing Python objects in files

Our next example writes a variety of Python objects into a text file on multiple lines. Notice that it must convert objects to strings using conversion tools. Again, file data is always strings in our scripts, and write methods do not do any automatic to-string formatting for us (for space, I’m omitting byte-count return values from `write` methods from here on):

```
>>> X, Y, Z = 43, 44, 45                    # Native Python objects
>>> S = 'Spam'                              # Must be strings to store in file
>>> D = {'a': 1, 'b': 2}
>>> L = [1, 2, 3]
>>>
>>> F = open('datafile.txt', 'w')           # Create output file
>>> F.write(S + '\n')                       # Terminate lines with \n
>>> F.write('%s,%s,%s\n' % (X, Y, Z))       # Convert numbers to strings
>>> F.write(str(L) + '$' + str(D) + '\n')   # Convert and separate with $
>>> F.close()
```

Once we have created our file, we can inspect its contents by opening it and reading it into a string (a single operation). Notice that the interactive echo gives the exact byte contents, while the `print` operation interprets embedded end-of-line characters to render a more user-friendly display:

```
>>> chars = open('datafile.txt').read()     # Raw string display
>>> chars
```



```
"Spam\n43,44,45\n[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> print(chars)                                # User-friendly display
Spam
43,44,45
[1, 2, 3]${'a': 1, 'b': 2}
```

We now have to use other conversion tools to translate from the strings in the text file to real Python objects. As Python never converts strings to numbers (or other types of objects) automatically, this is required if we need to gain access to normal object tools like indexing, addition, and so on:

```
>>> F = open('datafile.txt')                    # Open again
>>> line = F.readline()                          # Read one line
>>> line
'Spam\n'
>>> line.rstrip()                                # Remove end-of-line
'Spam'
```

For this first line, we used the string `rstrip` method to get rid of the trailing end-of-line character; a `line[:-1]` slice would work, too, but only if we can be sure all lines end in the `\n` character (the last line in a file sometimes does not).

So far, we've read the line containing the string. Now let's grab the next line, which contains numbers, and parse out (that is, extract) the objects on that line:

```
>>> line = F.readline()                          # Next line from file
>>> line                                          # It's a string here
'43,44,45\n'
>>> parts = line.split(',')                      # Split (parse) on commas
>>> parts
['43', '44', '45\n']
```

We used the string `split` method here to chop up the line on its comma delimiters; the result is a list of substrings containing the individual numbers. We still must convert from strings to integers, though, if we wish to perform math on these:

```
>>> int(parts[1])                                # Convert from string to int
44
>>> numbers = [int(P) for P in parts]            # Convert all in list at once
>>> numbers
[43, 44, 45]
```

As we have learned, `int` translates a string of digits into an integer object, and the list comprehension expression introduced in [Chapter 4](#) can apply the call to each item in our list all at once (you'll find more on list comprehensions later in this book). Notice that we didn't have to run `rstrip` to delete the `\n` at the end of the last part; `int` and some other converters quietly ignore whitespace around digits.

Finally, to convert the stored list and dictionary in the third line of the file, we can run them through `eval`, a built-in function that treats a string as a piece of executable program code (technically, a string containing a Python expression):

```
>>> line = F.readline()
>>> line
```

```

"[1, 2, 3]${'a': 1, 'b': 2}\n"
>>> parts = line.split('$')           # Split (parse) on $
>>> parts
['[1, 2, 3]', "${'a': 1, 'b': 2}\n"]
>>> eval(parts[0])                     # Convert to any object type
[1, 2, 3]
>>> objects = [eval(P) for P in parts] # Do same for all in list
>>> objects
[[1, 2, 3], {'a': 1, 'b': 2}]

```

Because the end result of all this parsing and converting is a list of normal Python objects instead of strings, we can now apply list and dictionary operations to them in our script.

Storing native Python objects with pickle

Using `eval` to convert from strings to objects, as demonstrated in the preceding code, is a powerful tool. In fact, sometimes it's *too* powerful. `eval` will happily run any Python expression—even one that might delete all the files on your computer, given the necessary permissions! If you really want to store native Python objects, but you can't trust the source of the data in the file, Python's standard library `pickle` module is ideal.

The `pickle` module is an advanced tool that allows us to store almost any Python object in a file directly, with no to- or from-string conversion requirement on our part. It's like a super-general data formatting and parsing utility. To store a dictionary in a file, for instance, we pickle it directly:

```

>>> D = {'a': 1, 'b': 2}
>>> F = open('datafile.pkl', 'wb')
>>> import pickle
>>> pickle.dump(D, F)                  # Pickle any object to file
>>> F.close()

```

Then, to get the dictionary back later, we simply use `pickle` again to re-create it:

```

>>> F = open('datafile.pkl', 'rb')
>>> E = pickle.load(F)                 # Load any object from file
>>> E
{'a': 1, 'b': 2}

```

We get back an equivalent dictionary object, with no manual splitting or converting required. The `pickle` module performs what is known as *object serialization*—converting objects to and from strings of bytes—but requires very little work on our part. In fact, `pickle` internally translates our dictionary to a string form, though it's not much to look at (and may vary if we pickle in other data protocol modes):

```

>>> open('datafile.pkl', 'rb').read() # Format is prone to change!
b'\x80\x03q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.'

```

Because `pickle` can reconstruct the object from this format, we don't have to deal with that ourselves. For more on the `pickle` module, see the Python standard library manual, or import `pickle` and pass it to `help` interactively. While you're exploring, also take a look at the `shelve` module. `shelve` is a tool that uses `pickle` to store Python objects in an access-by-key filesystem, which is beyond our scope here (though you will get to see

an example of `shelve` in action in [Chapter 27](#), and other `pickle` examples in [Chapters 30](#) and [36](#)).



Note that I opened the file used to store the pickled object in *binary mode*; binary mode is always required in Python 3.0, because the pickler creates and uses a `bytes` string object, and these objects imply binary-mode files (text-mode files imply `str` strings in 3.0). In earlier Pythons it's OK to use text-mode files for protocol 0 (the default, which creates ASCII text), as long as text mode is used consistently; higher protocols require binary-mode files. Python 3.0's default protocol is 3 (binary), but it creates `bytes` even for protocol 0. See [Chapter 36](#), Python's library manual, or reference books for more details on this.

Python 2.6 also has a `cPickle` module, which is an optimized version of `pickle` that can be imported directly for speed. Python 3.0 renames this module `_pickle` and uses it automatically in `pickle`—scripts simply import `pickle` and let Python optimize itself.

Storing and parsing packed binary data in files

One other file-related note before we move on: some advanced applications also need to deal with packed binary data, created perhaps by a C language program. Python's standard library includes a tool to help in this domain—the `struct` module knows how to both compose and parse packed binary data. In a sense, this is another data-conversion tool that interprets strings in files as binary data.

To create a packed binary data file, for example, open it in `'wb'` (write binary) mode, and pass `struct` a format string and some Python objects. The format string used here means pack as a 4-byte integer, a 4-character string, and a 2-byte integer, all in big-endian form (other format codes handle padding bytes, floating-point numbers, and more):

```
>>> F = open('data.bin', 'wb')                # Open binary output file
>>> import struct
>>> data = struct.pack('>i4sh', 7, 'spam', 8)    # Make packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
>>> F.write(data)                              # Write byte string
>>> F.close()
```

Python creates a binary `bytes` data string, which we write out to the file normally—one consists mostly of nonprintable characters printed in hexadecimal escapes, and is the same binary file we met earlier. To parse the values out to normal Python objects, we simply read the string back and unpack it using the same format string. Python extracts the values into normal Python objects—integers and a string:

```
>>> F = open('data.bin', 'rb')
>>> data = F.read()                            # Get packed binary data
>>> data
b'\x00\x00\x00\x07spam\x00\x08'
```

```
>>> values = struct.unpack('>i4sh', data)           # Convert to Python objects
>>> values
(7, 'spam', 8)
```

Binary data files are advanced and somewhat low-level tools that we won't cover in more detail here; for more help, see [Chapter 36](#), consult the Python library manual, or import `struct` and pass it to the `help` function interactively. Also note that the binary file-processing modes `'wb'` and `'rb'` can be used to process a simpler binary file such as an image or audio file as a whole without having to unpack its contents.

File context managers

You'll also want to watch for [Chapter 33](#)'s discussion of the file's *context manager* support, new in Python 3.0 and 2.6. Though more a feature of exception processing than files themselves, it allows us to wrap file-processing code in a logic layer that ensures that the file will be closed automatically on exit, instead of relying on the auto-close on garbage collection:

```
with open(r'C:\misc\data.txt') as myfile:           # See Chapter 33 for details
    for line in myfile:
        ...use line here...
```

The `try/finally` statement we'll look at in [Chapter 33](#) can provide similar functionality, but at some cost in extra code—three extra lines, to be precise (though we can often avoid both options and let Python close files for us automatically):

```
myfile = open(r'C:\misc\data.txt')
try:
    for line in myfile:
        ...use line here...
finally:
    myfile.close()
```

Since both these options require more information than we have yet obtained, we'll postpone details until later in this book.

Other File Tools

There are additional, more advanced file methods shown in [Table 9-2](#), and even more that are not in the table. For instance, as mentioned earlier, `seek` resets your current position in a file (the next read or write happens at that position), `flush` forces buffered output to be written out to disk (by default, files are always buffered), and so on.

The Python standard library manual and the reference books described in the Preface provide complete lists of file methods; for a quick look, run a `dir` or `help` call interactively, passing in an open file object (in Python 2.6 but not 3.0, you can pass in the name `file` instead). For more file-processing examples, watch for the sidebar [“Why You Will Care: File Scanners” on page 340](#). It sketches common file-scanning loop code patterns with statements we have not covered enough yet to use here.

Also, note that although the `open` function and the file objects it returns are your main interface to external files in a Python script, there are additional file-like tools in the Python toolset. Also available, to name a few, are:

Standard streams

Preopened file objects in the `sys` module, such as `sys.stdout` (see “[Print Operations](#)” on page 297)

Descriptor files in the `os` module

Integer file handles that support lower-level tools such as file locking

Sockets, pipes, and FIFOs

File-like objects used to synchronize processes or communicate over networks

Access-by-key files known as “shelves”

Used to store unaltered Python objects directly, by key (used in [Chapter 27](#))

Shell command streams

Tools such as `os.popen` and `subprocess.Popen` that support spawning shell commands and reading and writing to their standard streams

The third-party open source domain offers even more file-like tools, including support for communicating with serial ports in the *PySerial* extension and interactive programs in the *pexpect* system. See more advanced Python texts and the Web at large for additional information on file-like tools.



Version skew note: In Python 2.5 and earlier, the built-in name `open` is essentially a synonym for the name `file`, and files may technically be opened by calling either `open` or `file` (though `open` is generally preferred for opening). In Python 3.0, the name `file` is no longer available, because of its redundancy with `open`.

Python 2.6 users may also use the name `file` as the file object type, in order to customize files with object-oriented programming (described later in this book). In Python 3.0, files have changed radically. The classes used to implement file objects live in the standard library module `io`. See this module’s documentation or code for the classes it makes available for customization, and run a `type(F)` call on open files *F* for hints.

Type Categories Revisited

Now that we’ve seen all of Python’s core built-in types in action, let’s wrap up our object types tour by reviewing some of the properties they share. [Table 9-3](#) classifies all the major types we’ve seen so far according to the type categories introduced earlier. Here are some points to remember:

- Objects share operations according to their category; for instance, strings, lists, and tuples all share sequence operations such as concatenation, length, and indexing.
- Only mutable objects (lists, dictionaries, and sets) may be changed in-place; you cannot change numbers, strings, or tuples in-place.
- Files export only methods, so mutability doesn't really apply to them—their state may be changed when they are processed, but this isn't quite the same as Python core type mutability constraints.
- “Numbers” in [Table 9-3](#) includes all number types: integer (and the distinct long integer in 2.6), floating-point, complex, decimal, and fraction.
- “Strings” in [Table 9-3](#) includes `str`, as well as `bytes` in 3.0 and `unicode` in 2.6; the `bytearray` string type in 3.0 is mutable.
- Sets are something like the keys of a valueless dictionary, but they don't map to values and are not ordered, so sets are neither a mapping nor a sequence type; `frozenset` is an immutable variant of `set`.
- In addition to type category operations, as of Python 2.6 and 3.0 all the types in [Table 9-3](#) have callable methods, which are generally specific to their type.

Table 9-3. Object classifications

Object type	Category	Mutable?
Numbers (all)	Numeric	No
Strings	Sequence	No
Lists	Sequence	Yes
Dictionaries	Mapping	Yes
Tuples	Sequence	No
Files	Extension	N/A
Sets	Set	Yes
<code>frozenset</code>	Set	No
<code>bytearray</code> (3.0)	Sequence	Yes

Why You Will Care: Operator Overloading

In [Part VI](#) of this book, we'll see that objects we implement with classes can pick and choose from these categories arbitrarily. For instance, if we want to provide a new kind of specialized sequence object that is consistent with built-in sequences, we can code a class that overloads things like indexing and concatenation:

```
class MySequence:
    def __getitem__(self, index):
        # Called on self[index], others
    def __add__(self, other):
        # Called on self + other
```

and so on. We can also make the new object mutable or not by selectively implementing methods called for in-place change operations (e.g., `__setitem__` is called on `self[index]=value` assignments). Although it's beyond this book's scope, it's also possible to implement new objects in an external language like C as C extension types. For these, we fill in C function pointer slots to choose between number, sequence, and mapping operation sets.

Object Flexibility

This part of the book introduced a number of compound object types (collections with components). In general:

- Lists, dictionaries, and tuples can hold any kind of object.
- Lists, dictionaries, and tuples can be arbitrarily nested.
- Lists and dictionaries can dynamically grow and shrink.

Because they support arbitrary structures, Python's compound object types are good at representing complex information in programs. For example, values in dictionaries may be lists, which may contain tuples, which may contain dictionaries, and so on. The nesting can be as deep as needed to model the data to be processed.

Let's look at an example of nesting. The following interaction defines a tree of nested compound sequence objects, shown in [Figure 9-1](#). To access its components, you may include as many index operations as required. Python evaluates the indexes from left to right, and fetches a reference to a more deeply nested object at each step. [Figure 9-1](#) may be a pathologically complicated data structure, but it illustrates the syntax used to access nested objects in general:

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

References Versus Copies

[Chapter 6](#) mentioned that assignments always store references to objects, not copies of those objects. In practice, this is usually what you want. Because assignments can generate multiple references to the same object, though, it's important to be aware that changing a mutable object in-place may affect other references to the same object

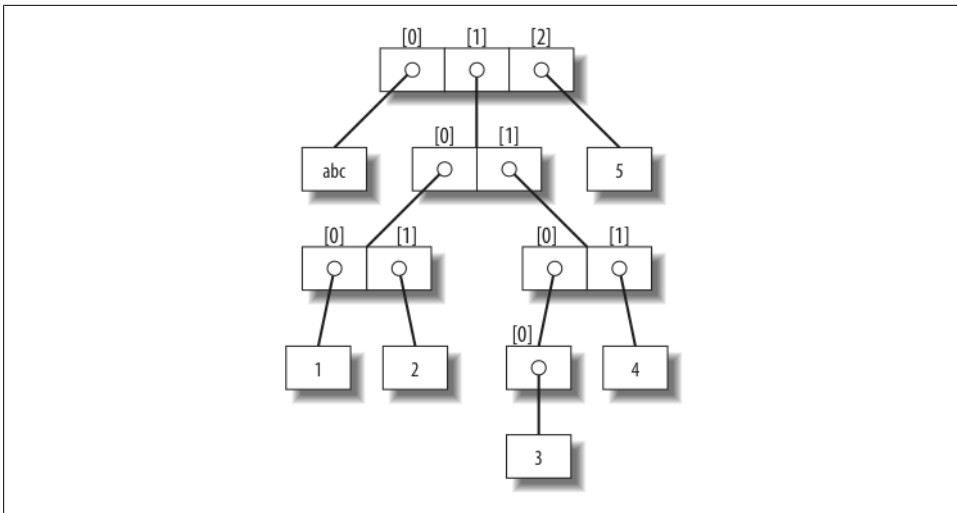


Figure 9-1. A nested object tree with the offsets of its components, created by running the literal expression `['abc', [(1, 2), ([3], 4)], 5]`. Syntactically nested objects are internally represented as references (i.e., pointers) to separate pieces of memory.

elsewhere in your program. If you don't want such behavior, you'll need to tell Python to copy the object explicitly.

We studied this phenomenon in [Chapter 6](#), but it can become more subtle when larger objects come into play. For instance, the following example creates a list assigned to `X`, and another list assigned to `L` that embeds a reference back to list `X`. It also creates a dictionary `D` that contains another reference back to list `X`:

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']           # Embed references to X's object
>>> D = {'x':X, 'y':2}
```

At this point, there are three references to the first list created: from the name `X`, from inside the list assigned to `L`, and from inside the dictionary assigned to `D`. The situation is illustrated in [Figure 9-2](#).

Because lists are mutable, changing the shared list object from any of the three references also changes what the other two reference:

```
>>> X[1] = 'surprise'          # Changes all three references!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```

References are a higher-level analog of pointers in other languages. Although you can't grab hold of the reference itself, it's possible to store the same reference in more than one place (variables, lists, and so on). This is a feature—you can pass a large object

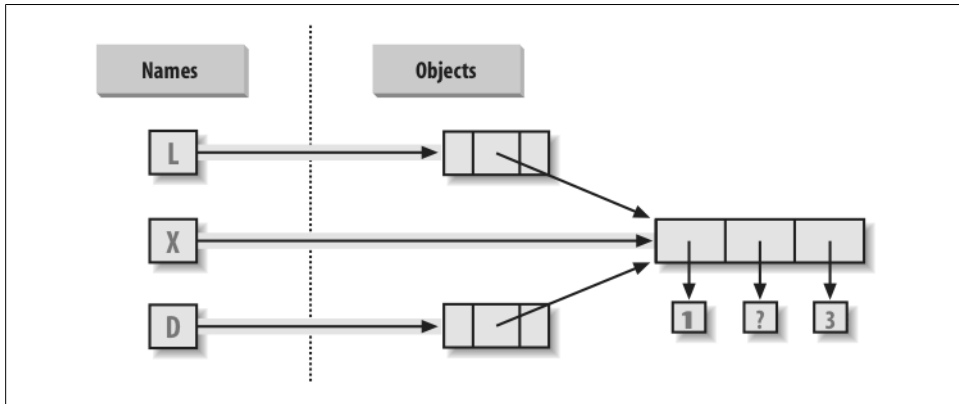


Figure 9-2. Shared object references: because the list referenced by variable X is also referenced from within the objects referenced by L and D, changing the shared list from X makes it look different from L and D, too.

around a program without generating expensive copies of it along the way. If you really do want copies, however, you can request them:

- Slice expressions with empty limits (`L[:]`) copy sequences.
- The dictionary and set `copy` method (`x.copy()`) copies a dictionary or set.
- Some built-in functions, such as `list`, make copies (`list(L)`).
- The `copy` standard library module makes full copies.

For example, say you have a list and a dictionary, and you don't want their values to be changed through other variables:

```
>>> L = [1,2,3]
>>> D = {'a':1, 'b':2}
```

To prevent this, simply assign copies to the other variables, not references to the same objects:

```
>>> A = L[:]                # Instead of A = L (or list(L))
>>> B = D.copy()           # Instead of B = D (ditto for sets)
```

This way, changes made from the other variables will change the copies, not the originals:

```
>>> A[1] = 'Ni'
>>> B['c'] = 'spam'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'spam', 'b': 2})
```

In terms of our original example, you can avoid the reference side effects by slicing the original list instead of simply naming it:

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']           # Embed copies of X's object
>>> D = {'x':X[:], 'y':2}
```

This changes the picture in [Figure 9-2](#)—L and D will now point to different lists than X. The net effect is that changes made through X will impact only X, not L and D; similarly, changes to L or D will not impact X.

One final note on copies: empty-limit slices and the dictionary `copy` method only make *top-level* copies; that is, they do not copy nested data structures, if any are present. If you need a complete, fully independent copy of a deeply nested data structure, use the standard `copy` module: include an `import copy` statement and say `X = copy.deepcopy(Y)` to fully copy an arbitrarily nested object Y. This call recursively traverses objects to copy all their parts. This is a much more rare case, though (which is why you have to say more to make it go). References are usually what you will want; when they are not, slices and copy methods are usually as much copying as you'll need to do.

Comparisons, Equality, and Truth

All Python objects also respond to comparisons: tests for equality, relative magnitude, and so on. Python comparisons always inspect all parts of compound objects until a result can be determined. In fact, when nested objects are present, Python automatically traverses data structures to apply comparisons *recursively* from left to right, and as deeply as needed. The first difference found along the way determines the comparison result.

For instance, a comparison of list objects compares all their components automatically:

```
>>> L1 = [1, ('a', 3)]             # Same value, unique objects
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2             # Equivalent? Same object?
(True, False)
```

Here, L1 and L2 are assigned lists that are equivalent but distinct objects. Because of the nature of Python references (studied in [Chapter 6](#)), there are two ways to test for equality:

- **The `==` operator tests value equivalence.** Python performs an equivalence test, comparing all nested objects recursively.
- **The `is` operator tests object identity.** Python tests whether the two are really the same object (i.e., live at the same address in memory).

In the preceding example, L1 and L2 pass the `==` test (they have equivalent values because all their components are equivalent) but fail the `is` check (they reference two different objects, and hence two different pieces of memory). Notice what happens for short strings, though:

```
>>> S1 = 'spam'
>>> S2 = 'spam'
```

```
>>> S1 == S2, S1 is S2
(True, True)
```

Here, we should again have two distinct objects that happen to have the same value: `==` should be true, and `is` should be false. But because Python internally caches and reuses some strings as an optimization, there really is just a single string `'spam'` in memory, shared by `S1` and `S2`; hence, the `is` identity test reports a true result. To trigger the normal behavior, we need to use longer strings:

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
>>> S1 == S2, S1 is S2
(True, False)
```

Of course, because strings are immutable, the object caching mechanism is irrelevant to your code—strings can't be changed in-place, regardless of how many variables refer to them. If identity tests seem confusing, see [Chapter 6](#) for a refresher on object reference concepts.

As a rule of thumb, the `==` operator is what you will want to use for almost all equality checks; `is` is reserved for highly specialized roles. We'll see cases where these operators are put to use later in the book.

Relative magnitude comparisons are also applied recursively to nested data structures:

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2      # Less, equal, greater: tuple of results
(False, False, True)
```

Here, `L1` is greater than `L2` because the nested `3` is greater than `2`. The result of the last line is really a tuple of three objects—the results of the three expressions typed (an example of a tuple without its enclosing parentheses).

In general, Python compares types as follows:

- Numbers are compared by relative magnitude.
- Strings are compared lexicographically, character by character (`"abc" < "ac"`).
- Lists and tuples are compared by comparing each component from left to right.
- Dictionaries compare as equal if their sorted (*key*, *value*) lists are equal. Relative magnitude comparisons are not supported for dictionaries in Python 3.0, but they work in 2.6 and earlier as though comparing sorted (*key*, *value*) lists.
- Nonnumeric mixed-type comparisons (e.g., `1 < 'spam'`) are errors in Python 3.0. They are allowed in Python 2.6, but use a fixed but arbitrary ordering rule. By proxy, this also applies to sorts, which use comparisons internally: nonnumeric mixed-type collections cannot be sorted in 3.0.

In general, comparisons of structured objects proceed as though you had written the objects as literals and compared all their parts one at a time from left to right. In later chapters, we'll see other object types that can change the way they get compared.

Python 3.0 Dictionary Comparisons

The second to last point in the preceding section merits illustration. In Python 2.6 and earlier, dictionaries support magnitude comparisons, as though you were comparing sorted key/value lists:

```
C:\misc> c:\python26\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
True
```

In Python 3.0, magnitude comparisons for dictionaries are removed because they incur too much overhead when equality is desired (equality uses an optimized scheme in 3.0 that doesn't literally compare sorted key/value lists). The alternative in 3.0 is to either write loops to compare values by key or compare the sorted key/value lists manually—the `items` dictionary methods and `sorted` built-in suffice:

```
C:\misc> c:\python30\python
>>> D1 = {'a':1, 'b':2}
>>> D2 = {'a':1, 'b':3}
>>> D1 == D2
False
>>> D1 < D2
TypeError: unorderable types: dict() < dict()

>>> list(D1.items())
[('a', 1), ('b', 2)]
>>> sorted(D1.items())
[('a', 1), ('b', 2)]

>>> sorted(D1.items()) < sorted(D2.items())
True
>>> sorted(D1.items()) > sorted(D2.items())
False
```

In practice, most programs requiring this behavior will develop more efficient ways to compare data in dictionaries than either this workaround or the original behavior in Python 2.6.

The Meaning of True and False in Python

Notice that the test results returned in the last two examples represent true and false values. They print as the words `True` and `False`, but now that we're using logical tests like these in earnest, I should be a bit more formal about what these names really mean.

In Python, as in most programming languages, an integer `0` represents false, and an integer `1` represents true. In addition, though, Python recognizes any empty data structure as false and any nonempty data structure as true. More generally, the notions of

true and false are intrinsic properties of every object in Python—each object is either true or false, as follows:

- Numbers are true if nonzero.
- Other objects are true if nonempty.

Table 9-4 gives examples of true and false objects in Python.

Table 9-4. Example object truth values

Object	Value
"spam"	True
""	False
[]	False
{}	False
1	True
0.0	False
None	False

As one application, because objects are true or false themselves, it's common to see Python programmers code tests like `if X:`, which, assuming `X` is a string, is the same as `if X != ''`. In other words, you can test the object itself, instead of comparing it to an empty object. (More on `if` statements in [Part III](#).)

The None object

As shown in the last item in [Table 9-4](#), Python also provides a special object called `None`, which is always considered to be false. `None` was introduced in [Chapter 4](#); it is the only value of a special data type in Python and typically serves as an empty placeholder (much like a `NULL` pointer in C).

For example, recall that for lists you cannot assign to an offset unless that offset already exists (the list does not magically grow if you make an out-of-bounds assignment). To preallocate a 100-item list such that you can add to any of the 100 offsets, you can fill it with `None` objects:

```
>>> L = [None] * 100
>>>
>>> L
[None, None, None, None, None, None, None, ... ]
```

This doesn't limit the size of the list (it can still grow and shrink later), but simply presets an initial size to allow for future index assignments. You could initialize a list with zeros the same way, of course, but best practice dictates using `None` if the list's contents are not yet known.

Keep in mind that `None` does not mean “undefined.” That is, `None` is something, not nothing (despite its name!)—it is a real object and piece of memory, given a built-in name by Python. Watch for other uses of this special object later in the book; it is also the default return value of functions, as we’ll see in [Part IV](#).

The bool type

Also keep in mind that the Python Boolean type `bool`, introduced in [Chapter 5](#), simply augments the notions of true and false in Python. As we learned in [Chapter 5](#), the built-in words `True` and `False` are just customized versions of the integers `1` and `0`—it’s as if these two words have been preassigned to `1` and `0` everywhere in Python. Because of the way this new type is implemented, this is really just a minor extension to the notions of true and false already described, designed to make truth values more explicit:

- When used explicitly in truth test code, the words `True` and `False` are equivalent to `1` and `0`, but they make the programmer’s intent clearer.
- Results of Boolean tests run interactively print as the words `True` and `False`, instead of as `1` and `0`, to make the type of result clearer.

You are not required to use only Boolean types in logical statements such as `if`; all objects are still inherently true or false, and all the Boolean concepts mentioned in this chapter still work as described if you use other types. Python also provides a `bool` built-in function that can be used to test the Boolean value of an object (i.e., whether it is `True`—that is, nonzero or nonempty):

```
>>> bool(1)
True
>>> bool('spam')
True
>>> bool({})
False
```

In practice, though, you’ll rarely notice the Boolean type produced by logic tests, because Boolean results are used automatically by `if` statements and other selection tools. We’ll explore Booleans further when we study logical statements in [Chapter 12](#).

Python’s Type Hierarchies

[Figure 9-3](#) summarizes all the built-in object types available in Python and their relationships. We’ve looked at the most prominent of these; most of the other kinds of objects in [Figure 9-3](#) correspond to program units (e.g., functions and modules) or exposed interpreter internals (e.g., stack frames and compiled code).

The main point to notice here is that *everything* in a Python system is an object type and may be processed by your Python programs. For instance, you can pass a class to a function, assign it to a variable, stuff it in a list or dictionary, and so on.

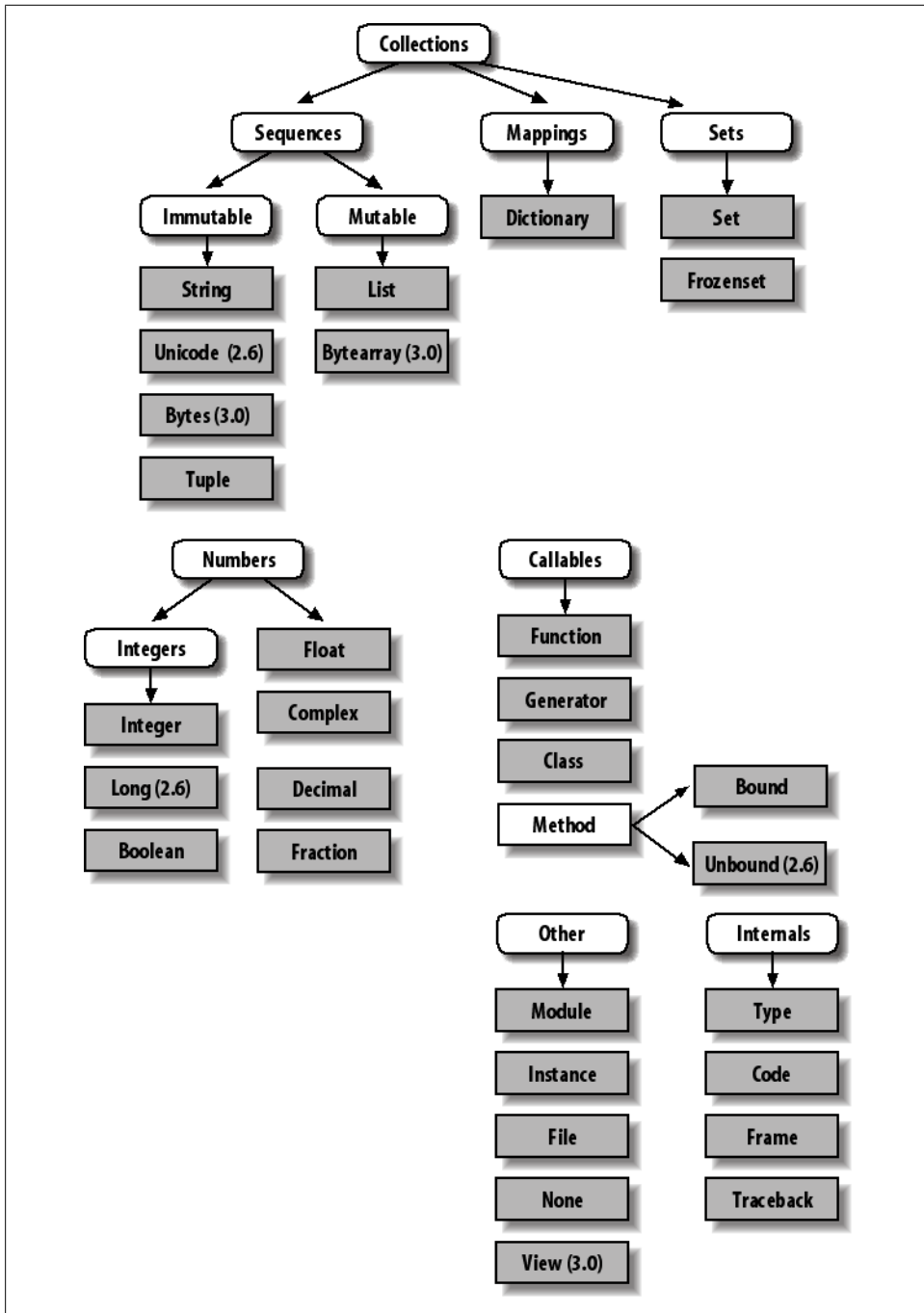


Figure 9-3. Python’s major built-in object types, organized by categories. Everything is a type of object in Python, even the type of an object!

Type Objects

In fact, even types themselves are an object type in Python: the type of an object is an object of type `type` (say that three times fast!). Seriously, a call to the built-in function `type(X)` returns the type object of object `X`. The practical application of this is that type objects can be used for manual type comparisons in Python `if` statements. However, for reasons introduced in [Chapter 4](#), manual type testing is usually not the right thing to do in Python, since it limits your code’s flexibility.

One note on type names: as of Python 2.2, each core type has a new built-in name added to support type customization through object-oriented subclassing: `dict`, `list`, `str`, `tuple`, `int`, `float`, `complex`, `bytes`, `type`, `set`, and more (in Python 2.6 but not 3.0, `file` is also a type name and a synonym for `open`). Calls to these names are really object constructor calls, not simply conversion functions, though you can treat them as simple functions for basic usage.

In addition, the `types` standard library module in Python 3.0 provides additional type names for types that are not available as built-ins (e.g., the type of a function; in Python 2.6 but not 3.0, this module also includes synonyms for built-in type names), and it is possible to do type tests with the `isinstance` function. For example, all of the following type tests are true:

```
type([1]) == type([])           # Type of another list
type([1]) == list               # List type name
instance([1], list)             # List or customization thereof

import types                    # types has names for other types
def f(): pass
type(f) == types.FunctionType
```

Because types can be subclassed in Python today, the `isinstance` technique is generally recommended. See [Chapter 31](#) for more on subclassing built-in types in Python 2.2 and later.

Also in [Chapter 31](#), we will explore how `type(X)` and type-testing in general apply to instances of user-defined *classes*. In short, in Python 3.0 and for new-style classes in Python 2.6, the type of a class instance is the class from which the instance was made. For classic classes in Python 2.6 and earlier, all class instances are of the type “instance,” and we must compare instance `__class__` attributes to compare their types meaningfully. Since we’re not ready for classes yet, we’ll postpone the rest of this story until [Chapter 31](#).

Other Types in Python

Besides the core objects studied in this part of the book, and the program-unit objects such as functions, modules, and classes that we’ll meet later, a typical Python installation has dozens of additional object types available as linked-in C extensions or

Python classes—regular expression objects, DBM files, GUI widgets, network sockets, and so on.

The main difference between these extra tools and the built-in types we’ve seen so far is that the built-ins provide special language creation syntax for their objects (e.g., `4` for an integer, `[1,2]` for a list, the `open` function for files, and `def` and `lambda` for functions). Other tools are generally made available in standard library modules that you must first import to use. For instance, to make a regular expression object, you import `re` and call `re.compile()`. See Python’s library reference for a comprehensive guide to all the tools available to Python programs.

Built-in Type Gotchas

That’s the end of our look at core data types. We’ll wrap up this part of the book with a discussion of common problems that seem to bite new users (and the occasional expert), along with their solutions. Some of this is a review of ideas we’ve already covered, but these issues are important enough to warn about again here.

Assignment Creates References, Not Copies

Because this is such a central concept, I’ll mention it again: you need to understand what’s going on with shared references in your program. For instance, in the following example, the list object assigned to the name `L` is referenced from `L` and from inside the list assigned to the name `M`. Changing `L` in-place changes what `M` references, too:

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']           # Embed a reference to L
>>> M
['X', [1, 2, 3], 'Y']

>>> L[1] = 0                    # Changes M too
>>> M
['X', [1, 0, 3], 'Y']
```

This effect usually becomes important only in larger programs, and shared references are often exactly what you want. If they’re not, you can avoid sharing objects by copying them explicitly. For lists, you can always make a top-level copy by using an empty-limits slice:

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']        # Embed a copy of L
>>> L[1] = 0                    # Changes only L, not M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

Remember, slice limits default to 0 and the length of the sequence being sliced; if both are omitted, the slice extracts every item in the sequence and so makes a top-level copy (a new, unshared object).

Repetition Adds One Level Deep

Repeating a sequence is like adding it to itself a number of times. However, when mutable sequences are nested, the effect might not always be what you expect. For instance, in the following example *X* is assigned to *L* repeated four times, whereas *Y* is assigned to a list *containing* *L* repeated four times:

```
>>> L = [4, 5, 6]
>>> X = L * 4           # Like [4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4         # [L] + [L] + ... = [L, L,...]

>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Because *L* was nested in the second repetition, *Y* winds up embedding references back to the original list assigned to *L*, and so is open to the same sorts of side effects noted in the last section:

```
>>> L[1] = 0           # Impacts Y but not X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

The same solutions to this problem apply here as in the previous section, as this is really just another way to create the shared mutable object reference case. If you remember that repetition, concatenation, and slicing copy only the top level of their operand objects, these sorts of cases make much more sense.

Beware of Cyclic Data Structures

We actually encountered this concept in a prior exercise: if a collection object contains a reference to itself, it's called a *cyclic object*. Python prints a [...] whenever it detects a cycle in the object, rather than getting stuck in an infinite loop:

```
>>> L = ['grail']
>>> L.append(L)         # Append reference to same object
>>> L                  # Generates cycle in object: [...]
['grail', [...]]
```

Besides understanding that the three dots in square brackets represent a cycle in the object, this case is worth knowing about because it can lead to gotchas—cyclic structures may cause code of your own to fall into unexpected loops if you don't anticipate them. For instance, some programs keep a list or dictionary of already visited items and

check it to determine whether they’re in a cycle. See the solutions to the “[Test Your Knowledge: Part I Exercises](#)” in [Appendix B](#) for more on this problem, and check out the *reloadall.py* program in [Chapter 24](#) for a solution.

Don’t use cyclic references unless you really need to. There are good reasons to create cycles, but unless you have code that knows how to handle them, you probably won’t want to make your objects reference themselves very often in practice.

Immutable Types Can’t Be Changed In-Place

You can’t change an immutable object in-place. Instead, you construct a new object with slicing, concatenation, and so on, and assign it back to the original reference, if needed:

```
T = (1, 2, 3)

T[2] = 4           # Error!

T = T[:2] + (4,)   # OK: (1, 2, 4)
```

That might seem like extra coding work, but the upside is that the previous gotchas can’t happen when you’re using immutable objects such as tuples and strings; because they can’t be changed in-place, they are not open to the sorts of side effects that lists are.

Chapter Summary

This chapter explored the last two major core object types—the tuple and the file. We learned that tuples support all the usual sequence operations, have just a few methods, and do not allow any in-place changes because they are immutable. We also learned that files are returned by the built-in `open` function and provide methods for reading and writing data. We explored how to translate Python objects to and from strings for storing in files, and we looked at the `pickle` and `struct` modules for advanced roles (object serialization and binary data). Finally, we wrapped up by reviewing some properties common to all object types (e.g., shared references) and went through a list of common mistakes (“gotchas”) in the object type domain.

In the next part, we’ll shift gears, turning to the topic of statement syntax in Python—we’ll explore all of Python’s basic procedural statements in the chapters that follow. The next chapter kicks off that part of the book with an introduction to Python’s general syntax model, which is applicable to all statement types. Before moving on, though, take the chapter quiz, and then work through the end-of-part lab exercises to review type concepts. Statements largely just create and process objects, so make sure you’ve mastered this domain by working through all the exercises before reading on.

Test Your Knowledge: Quiz

1. How can you determine how large a tuple is? Why is this tool located where it is?
2. Write an expression that changes the first item in a tuple. `(4, 5, 6)` should become `(1, 5, 6)` in the process.
3. What is the default for the processing mode argument in a file `open` call?
4. What module might you use to store Python objects in a file without converting them to strings yourself?
5. How might you go about copying all parts of a nested structure at once?
6. When does Python consider an object true?
7. What is your quest?

Test Your Knowledge: Answers

1. The built-in `len` function returns the length (number of contained items) for any container object in Python, including tuples. It is a built-in function instead of a type method because it applies to many different types of objects. In general, built-in functions and expressions may span many object types; methods are specific to a single object type, though some may be available on more than one type (`index`, for example, works on lists and tuples).
2. Because they are immutable, you can't really change tuples in-place, but you can generate a new tuple with the desired value. Given `T = (4, 5, 6)`, you can change the first item by making a new tuple from its parts by slicing and concatenating: `T = (1,) + T[1:]`. (Recall that single-item tuples require a trailing comma.) You could also convert the tuple to a list, change it in-place, and convert it back to a tuple, but this is more expensive and is rarely required in practice—simply use a list if you know that the object will require in-place changes.
3. The default for the processing mode argument in a file `open` call is `'r'`, for reading text input. For input text files, simply pass in the external file's name.
4. The `pickle` module can be used to store Python objects in a file without explicitly converting them to strings. The `struct` module is related, but it assumes the data is to be in packed binary format in the file.
5. Import the `copy` module, and call `copy.deepcopy(X)` if you need to copy all parts of a nested structure `X`. This is also rarely seen in practice; references are usually the desired behavior, and shallow copies (e.g., `aList[:]`, `aDict.copy()`) usually suffice for most copies.

6. An object is considered true if it is either a nonzero number or a nonempty collection object. The built-in words `True` and `False` are essentially predefined to have the same meanings as integer 1 and 0, respectively.
7. Acceptable answers include “To learn Python,” “To move on to the next part of the book,” or “To seek the Holy Grail.”

Test Your Knowledge: Part II Exercises

This session asks you to get your feet wet with built-in object fundamentals. As before, a few new ideas may pop up along the way, so be sure to flip to the answers in [Appendix B](#) when you’re done (or when you’re not, if necessary). If you have limited time, I suggest starting with exercises 10 and 11 (the most practical of the bunch), and then working from first to last as time allows. This is all fundamental material, though, so try to do as many of these as you can.

1. *The basics.* Experiment interactively with the common type operations found in the various operation tables in this part of the book. To get started, bring up the Python interactive interpreter, type each of the following expressions, and try to explain what’s happening in each case. Note that the semicolon in some of these is being used as a statement separator, to squeeze multiple statements onto a single line: for example, `X=1;X` assigns and then prints a variable (more on statement syntax in the next part of the book). Also remember that a comma between expressions usually builds a tuple, even if there are no enclosing parentheses: `X,Y,Z` is a three-item tuple, which Python prints back to you in parentheses.

```
2 ** 16
2 / 5, 2 / 5.0

"spam" + "eggs"
S = "ham"
"eggs " + S
S * 5
S[:0]
"green %s and %s" % ("eggs", S)
'green {0} and {1}'.format('eggs', S)

('x',)[0]
('x', 'y')[1]

L = [1,2,3] + [4,5,6]
L, L[:], L[:0], L[-2], L[-2:]
([1,2,3] + [4,5,6])[2:4]
[L[2], L[3]]
L.reverse(); L
L.sort(); L
L.index(4)

{'a':1, 'b':2}['b']
D = {'x':1, 'y':2, 'z':3}
```

```
D['w'] = 0
D['x'] + D['w']
D[(1,2,3)] = 4
list(D.keys()), list(D.values()), (1,2,3) in D
```

```
[[[]], [""], [], (), {}, None]
```

2. *Indexing and slicing.* At the interactive prompt, define a list named `L` that contains four strings or numbers (e.g., `L=[0,1,2,3]`). Then, experiment with some boundary cases; you may not ever see these cases in real programs, but they are intended to make you think about the underlying model, and some may be useful in less artificial forms:

- a. What happens when you try to index out of bounds (e.g., `L[4]`)?
- b. What about slicing out of bounds (e.g., `L[-1000:100]`)?
- c. Finally, how does Python handle it if you try to extract a sequence in reverse, with the lower bound greater than the higher bound (e.g., `L[3:1]`)? Hint: try assigning to this slice (`L[3:1]='?'`), and see where the value is put. Do you think this may be the same phenomenon you saw when slicing out of bounds?

3. *Indexing, slicing, and del.* Define another list `L` with four items, and assign an empty list to one of its offsets (e.g., `L[2]=[]`). What happens? Then, assign an empty list to a slice (`L[2:3]=[]`). What happens now? Recall that slice assignment deletes the slice and inserts the new value where it used to be.

The `del` statement deletes offsets, keys, attributes, and names. Use it on your list to delete an item (e.g., `del L[0]`). What happens if you delete an entire slice (`del L[1:]`)? What happens when you assign a nonsequence to a slice (`L[1:2]=1`)?

4. *Tuple assignment.* Type the following lines:

```
>>> X = 'spam'
>>> Y = 'eggs'
>>> X, Y = Y, X
```

What do you think is happening to `X` and `Y` when you type this sequence?

5. *Dictionary keys.* Consider the following code fragments:

```
>>> D = {}
>>> D[1] = 'a'
>>> D[2] = 'b'
```

You've learned that dictionaries aren't accessed by offsets, so what's going on here? Does the following shed any light on the subject? (Hint: strings, integers, and tuples share which type category?)

```
>>> D[(1, 2, 3)] = 'c'
>>> D
{1: 'a', 2: 'b', (1, 2, 3): 'c'}
```

6. *Dictionary indexing.* Create a dictionary named `D` with three entries, for keys `'a'`, `'b'`, and `'c'`. What happens if you try to index a nonexistent key (`D['d']`)? What does Python do if you try to assign to a nonexistent key `'d'` (e.g., `D['d']='spam'`)? How does this compare to out-of-bounds assignments and references for lists? Does this sound like the rule for variable names?
7. *Generic operations.* Run interactive tests to answer the following questions:
 - a. What happens when you try to use the `+` operator on different/mixed types (e.g., `string + list`, `list + tuple`)?
 - b. Does `+` work when one of the operands is a dictionary?
 - c. Does the `append` method work for both lists and strings? How about using the `keys` method on lists? (Hint: what does `append` assume about its subject object?)
 - d. Finally, what type of object do you get back when you slice or concatenate two lists or two strings?
8. *String indexing.* Define a string `S` of four characters: `S = "spam"`. Then type the following expression: `S[0][0][0][0][0]`. Any clue as to what's happening this time? (Hint: recall that a string is a collection of characters, but Python characters are one-character strings.) Does this indexing expression still work if you apply it to a list such as `['s', 'p', 'a', 'm']`? Why?
9. *Immutable types.* Define a string `S` of four characters again: `S = "spam"`. Write an assignment that changes the string to `"slam"`, using only slicing and concatenation. Could you perform the same operation using just indexing and concatenation? How about index assignment?
10. *Nesting.* Write a data structure that represents your personal information: name (first, middle, last), age, job, address, email address, and phone number. You may build the data structure with any combination of built-in object types you like (lists, tuples, dictionaries, strings, numbers). Then, access the individual components of your data structures by indexing. Do some structures make more sense than others for this object?
11. *Files.* Write a script that creates a new output file called *myfile.txt* and writes the string `"Hello file world!"` into it. Then write another script that opens *myfile.txt* and reads and prints its contents. Run your two scripts from the system command line. Does the new file show up in the directory where you ran your scripts? What if you add a different directory path to the filename passed to `open`? Note: file `write` methods do not add newline characters to your strings; add an explicit `\n` at the end of the string if you want to fully terminate the line in the file.

Statements and Syntax

Introducing Python Statements

Now that you're familiar with Python's core built-in object types, this chapter begins our exploration of its fundamental statement forms. As in the previous part, we'll begin here with a general introduction to statement syntax, and we'll follow up with more details about specific statements in the next few chapters.

In simple terms, *statements* are the things you write to tell Python what your programs should do. If programs “do things with stuff,” statements are the way you specify what sort of things a program does. Python is a procedural, statement-based language; by combining statements, you specify a procedure that Python performs to satisfy a program's goals.

Python Program Structure Revisited

Another way to understand the role of statements is to revisit the concept hierarchy introduced in [Chapter 4](#), which talked about built-in objects and the expressions used to manipulate them. This chapter climbs the hierarchy to the next level:

1. Programs are composed of modules.
2. Modules contain statements.
3. *Statements contain expressions.*
4. Expressions create and process objects.

At its core, Python syntax is composed of statements and expressions. Expressions process objects and are embedded in statements. Statements code the larger *logic* of a program's operation—they use and direct expressions to process the objects we studied in the preceding chapters. Moreover, statements are where objects spring into existence (e.g., in expressions within assignment statements), and some statements create entirely new kinds of objects (functions, classes, and so on). Statements always exist in modules, which themselves are managed with statements.

Python's Statements

[Table 10-1](#) summarizes Python's statement set. This part of the book deals with entries in the table from the top through `break` and `continue`. You've informally been introduced to a few of the statements in [Table 10-1](#) already; this part of the book will fill in details that were skipped earlier, introduce the rest of Python's procedural statement set, and cover the overall syntax model. Statements lower in [Table 10-1](#) that have to do with larger program units—functions, classes, modules, and exceptions—lead to larger programming ideas, so they will each have a section of their own. More focused statements (like `del`, which deletes various components) are covered elsewhere in the book, or in Python's standard manuals.

Table 10-1. Python 3.0 statements

Statement	Role	Example
Assignment	Creating references	<code>a, *b = 'good', 'bad', 'ugly'</code>
Calls and other expressions	Running functions	<code>log.write("spam, ham")</code>
<code>print</code> calls	Printing objects	<code>print('The Killer', joke)</code>
<code>if/elif/else</code>	Selecting actions	<code>if "python" in text: print(text)</code>
<code>for/else</code>	Sequence iteration	<code>for x in mylist: print(x)</code>
<code>while/else</code>	General loops	<code>while X > Y: print('hello')</code>
<code>pass</code>	Empty placeholder	<code>while True: pass</code>
<code>break</code>	Loop exit	<code>while True: if exittest(): break</code>
<code>continue</code>	Loop continue	<code>while True: if skiptest(): continue</code>
<code>def</code>	Functions and methods	<code>def f(a, b, c=1, *d): print(a+b+c+d[0])</code>
<code>return</code>	Functions results	<code>def f(a, b, c=1, *d): return a+b+c+d[0]</code>
<code>yield</code>	Generator functions	<code>def gen(n): for i in n: yield i*2</code>
<code>global</code>	Namespaces	<code>x = 'old' def function(): global x, y; x = 'new'</code>
<code>nonlocal</code>	Namespaces (3.0+)	<code>def outer(): x = 'old' def function(): nonlocal x; x = 'new'</code>
<code>import</code>	Module access	<code>import sys</code>
<code>from</code>	Attribute access	<code>from sys import stdin</code>
<code>class</code>	Building objects	<code>class Subclass(Superclass): staticData = [] def method(self): pass</code>

Statement	Role	Example
<code>try/except/finally</code>	Catching exceptions	<pre>try: action() except: print('action error')</pre>
<code>raise</code>	Triggering exceptions	<code>raise EndSearch(location)</code>
<code>assert</code>	Debugging checks	<code>assert X > Y, 'X too small'</code>
<code>with/as</code>	Context managers (2.6+)	<pre>with open('data') as myfile: process(myfile)</pre>
<code>del</code>	Deleting references	<pre>del data[k] del data[i:j] del obj.attr del variable</pre>

Table 10-1 reflects the statement forms in Python 3.0—units of code that each have a specific syntax and purpose. Here are a few fine points about its content:

- Assignment statements come in a variety of syntax flavors, described in [Chapter 11](#): basic, sequence, augmented, and more.
- `print` is technically neither a reserved word nor a statement in 3.0, but a built-in function call; because it will nearly always be run as an expression statement, though (that is, on a line by itself), it’s generally thought of as a statement type. We’ll study print operations in [Chapter 11](#) the next chapter.
- `yield` is actually an expression instead of a statement too, as of 2.5; like `print`, it’s typically used in a line by itself and so is included in this table, but scripts occasionally assign or otherwise use its result, as we’ll see in [Chapter 20](#). As an expression, `yield` is also a reserved word, unlike `print`.

Most of this table applies to Python 2.6, too, except where it doesn’t—if you are using Python 2.6 or older, here are a few notes for your Python, too:

- In 2.6, `nonlocal` is not available; as we’ll see in [Chapter 17](#), there are alternative ways to achieve this statement’s writeable state-retention effect.
- In 2.6, `print` is a statement instead of a built-in function call, with specific syntax covered in [Chapter 11](#).
- In 2.6, the 3.0 `exec` code execution built-in function is a statement, with specific syntax; since it supports enclosing parentheses, though, you can generally use its 3.0 call form in 2.6 code.
- In 2.5, the `try/except` and `try/finally` statements were merged: the two were formerly separate statements, but we can now say both `except` and `finally` in the same `try` statement.
- In 2.5, `with/as` is an optional extension, and it is not available unless you explicitly turn it on by running the statement `from __future__ import with_statement` (see [Chapter 33](#)).

A Tale of Two ifs

Before we delve into the details of any of the concrete statements in [Table 10-1](#), I want to begin our look at Python statement syntax by showing you what you are *not* going to type in Python code so you can compare and contrast it with other syntax models you might have seen in the past.

Consider the following `if` statement, coded in a C-like language:

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

This might be a statement in C, C++, Java, JavaScript, or Perl. Now, look at the equivalent statement in the Python language:

```
if x > y:  
    x = 1  
    y = 2
```

The first thing that may pop out at you is that the equivalent Python statement is less, well, cluttered—that is, there are fewer syntactic components. This is by design; as a scripting language, one of Python’s goals is to make programmers’ lives easier by requiring less typing.

More specifically, when you compare the two syntax models, you’ll notice that Python adds one new thing to the mix, and that three items that are present in the C-like language are not present in Python code.

What Python Adds

The one new syntax component in Python is the colon character (:). All Python *compound statements* (i.e., statements that have statements nested inside them) follow the same general pattern of a header line terminated in a colon, followed by a nested block of code usually indented underneath the header line, like this:

```
Header line:  
    Nested statement block
```

The colon is required, and omitting it is probably the most common coding mistake among new Python programmers—it’s certainly one I’ve witnessed thousands of times in Python training classes. In fact, if you are new to Python, you’ll almost certainly forget the colon character very soon. Most Python-friendly editors make this mistake easy to spot, and including it eventually becomes an unconscious habit (so much so that you may start typing colons in your C++ code, too, generating many entertaining error messages from your C++ compiler!).

What Python Removes

Although Python requires the extra colon character, there are three things programmers in C-like languages must include that you don't generally have to in Python.

Parentheses are optional

The first of these is the set of parentheses around the tests at the top of the statement:

```
if (x < y)
```

The parentheses here are required by the syntax of many C-like languages. In Python, though, they are not—we simply omit the parentheses, and the statement works the same way:

```
if x < y
```

Technically speaking, because every expression can be enclosed in parentheses, including them will not hurt in this Python code, and they are not treated as an error if present. *But don't do that:* you'll be wearing out your keyboard needlessly, and broadcasting to the world that you're an ex-C programmer still learning Python (I was once, too). The Python way is to simply omit the parentheses in these kinds of statements altogether.

End of line is end of statement

The second and more significant syntax component you won't find in Python code is the semicolon. You don't need to terminate statements with semicolons in Python the way you do in C-like languages:

```
x = 1;
```

In Python, the general rule is that the end of a line automatically terminates the statement that appears on that line. In other words, you can leave off the semicolons, and it works the same way:

```
x = 1
```

There are some ways to work around this rule, as you'll see in a moment. But, in general, you write one statement per line for the vast majority of Python code, and no semicolon is required.

Here, too, if you are pining for your C programming days (if such a state is possible...) you can continue to use semicolons at the end of each statement—the language lets you get away with them if they are present. *But don't do that either* (really!); again, doing so tells the world that you're still a C programmer who hasn't quite made the switch to Python coding. The Pythonic style is to leave off the semicolons altogether.

End of indentation is end of block

The third and final syntax component that Python removes, and the one that may seem the most unusual to soon-to-be-ex-C programmers (until they've used it for 10 minutes and realize it's actually a feature), is that you do not type anything explicit in your code to syntactically mark the beginning and end of a nested block of code. You don't need to include `begin/end`, `then/endif`, or braces around the nested block, as you do in C-like languages:

```
if (x > y) {  
    x = 1;  
    y = 2;  
}
```

Instead, in Python, we consistently indent all the statements in a given single nested block the same distance to the right, and Python uses the statements' physical indentation to determine where the block starts and stops:

```
if x > y:  
    x = 1  
    y = 2
```

By *indentation*, I mean the blank whitespace all the way to the left of the two nested statements here. Python doesn't care how you indent (you may use either spaces or tabs), or how much you indent (you may use any number of spaces or tabs). In fact, the indentation of one nested block can be totally different from that of another. The syntax rule is only that for a given single nested block, all of its statements must be indented the same distance to the right. If this is not the case, you will get a syntax error, and your code will not run until you repair its indentation to be consistent.

Why Indentation Syntax?

The indentation rule may seem unusual at first glance to programmers accustomed to C-like languages, but it is a deliberate feature of Python, and it's one of the main ways that Python almost forces programmers to produce uniform, regular, and readable code. It essentially means that you must line up your code vertically, in columns, according to its logical structure. The net effect is to make your code more consistent and readable (unlike much of the code written in C-like languages).

To put that more strongly, aligning your code according to its logical structure is a major part of making it readable, and thus reusable and maintainable, by yourself and others. In fact, even if you never use Python after reading this book, you should get into the habit of aligning your code for readability in any block-structured language. Python forces the issue by making this a part of its syntax, but it's an important thing to do in any programming language, and it has a huge impact on the usefulness of your code.

Your experience may vary, but when I was still doing development on a full-time basis, I was mostly paid to work on large old C++ programs that had been worked on by many programmers over the years. Almost invariably, each programmer had his or her

own style for indenting code. For example, I'd often be asked to change a `while` loop coded in the C++ language that began like this:

```
while (x > 0) {
```

Before we even get into indentation, there are three or four ways that programmers can arrange these braces in a C-like language, and organizations often have political debates and write standards manuals to address the options (which seems more than a little off-topic for the problem to be solved by programming). Ignoring that, here's the scenario I often encountered in C++ code. The first person who worked on the code indented the loop four spaces:

```
while (x > 0) {  
    -----;  
    -----;
```

That person eventually moved on to management, only to be replaced by someone who liked to indent further to the right:

```
while (x > 0) {  
    -----;  
    -----;  
        -----;  
        -----;
```

That person later moved on to other opportunities, and someone else picked up the code who liked to indent less:

```
while (x > 0) {  
    -----;  
    -----;  
        -----;  
        -----;  
-----;  
-----;  
}
```

And so on. Eventually, the block is terminated by a closing brace (`}`), which of course makes this “block-structured code” (he says, sarcastically). In any block-structured language, Python or otherwise, if nested blocks are not indented consistently, they become very difficult for the reader to interpret, change, or reuse, because the code no longer visually reflects its logical meaning. Readability matters, and indentation is a major component of readability.

Here is another example that may have burned you in the past if you've done much programming in a C-like language. Consider the following statement in C:

```
if (x)  
    if (y)  
        statement1;  
else  
    statement2;
```

Which `if` does the `else` here go with? Surprisingly, the `else` is paired with the nested `if` statement (`if (y)`), even though it looks visually as though it is associated with the outer `if (x)`. This is a classic pitfall in the C language, and it can lead to the reader completely misinterpreting the code and changing it incorrectly in ways that might not be uncovered until the Mars rover crashes into a giant rock!

This cannot happen in Python—because indentation is significant, the way the code looks is the way it will work. Consider an equivalent Python statement:

```
if x:
    if y:
        statement1
    else:
        statement2
```

In this example, the `if` that the `else` lines up with vertically is the one it is associated with logically (the outer `if x`). In a sense, Python is a WYSIWYG language—what you see is what you get because the way code looks is the way it runs, regardless of who coded it.

If this still isn't enough to underscore the benefits of Python's syntax, here's another anecdote. Early in my career, I worked at a successful company that developed systems software in the C language, where consistent indentation is not required. Even so, when we checked our code into source control at the end of the day, this company ran an automated script that analyzed the indentation used in the code. If the script noticed that we'd indented our code inconsistently, we received an automated email about it the next morning—and so did our managers!

The point is that even when a language doesn't require it, good programmers know that consistent use of indentation has a huge impact on code readability and quality. The fact that Python promotes this to the level of syntax is seen by most as a feature of the language.

Also keep in mind that nearly every programmer-friendly text editor has built-in support for Python's syntax model. In the IDLE Python GUI, for example, lines of code are automatically indented when you are typing a nested block; pressing the Backspace key backs up one level of indentation, and you can customize how far to the right IDLE indents statements in a nested block. There is no universal standard on this: four spaces or one tab per level is common, but it's up to you to decide how and how much you wish to indent. Indent further to the right for further nested blocks, and less to close the prior block.

As a rule of thumb, you probably shouldn't mix tabs and spaces in the same block in Python, unless you do so consistently; use tabs or spaces in a given block, but not both (in fact, Python 3.0 now issues an error for inconsistent use of tabs and spaces, as we'll see in [Chapter 12](#)). But you probably shouldn't mix tabs or spaces in indentation in *any* structured language—such code can cause major readability issues if the next programmer has his or her editor set to display tabs differently than yours. C-like languages

might let coders get away with this, but they shouldn't: the result can be a mangled mess.

I can't stress enough that regardless of which language you code in, you should be indenting consistently for readability. In fact, if you weren't taught to do this earlier in your career, your teachers did you a disservice. Most programmers—especially those who must read others' code—consider it a major asset that Python elevates this to the level of syntax. Moreover, generating tabs instead of braces is no more difficult in practice for tools that must output Python code. In general, if you do what you should be doing in a C-like language anyhow, but get rid of the braces, your code will satisfy Python's syntax rules.

A Few Special Cases

As mentioned previously, in Python's syntax model:

- The end of a line terminates the statement on that line (without semicolons).
- Nested statements are blocked and associated by their physical indentation (without braces).

Those rules cover almost all Python code you'll write or see in practice. However, Python also provides some special-purpose rules that allow customization of both statements and nested statement blocks.

Statement rule special cases

Although statements normally appear one per line, it is possible to squeeze more than one statement onto a single line in Python by separating them with semicolons:

```
a = 1; b = 2; print(a + b)           # Three statements on one line
```

This is the only place in Python where semicolons are required: as *statement separators*. This only works, though, if the statements thus combined are not themselves compound statements. In other words, you can chain together only simple statements, like assignments, `prints`, and function calls. Compound statements must still appear on lines of their own (otherwise, you could squeeze an entire program onto one line, which probably would not make you very popular among your coworkers!).

The other special rule for statements is essentially the inverse: you can make a single statement span across multiple lines. To make this work, you simply have to enclose part of your statement in a bracketed pair—parentheses `(())`, square brackets `[]`, or curly braces `{ }`. Any code enclosed in these constructs can cross multiple lines: your statement doesn't end until Python reaches the line containing the closing part of the pair. For instance, to continue a list literal:

```
mlist = [111,
          222,
          333]
```

Because the code is enclosed in a square brackets pair, Python simply drops down to the next line until it encounters the closing bracket. The curly braces surrounding dictionaries (as well as set literals and dictionary and set comprehensions in 3.0) allow them to span lines this way too, and parentheses handle tuples, function calls, and expressions. The indentation of the continuation lines does not matter, though common sense dictates that the lines should be aligned somehow for readability.

Parentheses are the catchall device—because any expression can be wrapped up in them, simply inserting a left parenthesis allows you to drop down to the next line and continue your statement:

```
X = (A + B +  
     C + D)
```

This technique works with compound statements, too, by the way. Anywhere you need to code a large expression, simply wrap it in parentheses to continue it on the next line:

```
if (A == 1 and  
    B == 2 and  
    C == 3):  
    print('spam' * 3)
```

An older rule also allows for continuation lines when the prior line ends in a backslash:

```
X = A + B + \  
    C + D          # An error-prone alternative
```

This alternative technique is dated, though, and is frowned on today because it's difficult to notice and maintain the backslashes, and it's fairly brittle—there can be no spaces after the backslash, and omitting it can have unexpected effects if the next line is mistaken to be a new statement. It's also another throwback to the C language, where it is commonly used in “#define” macros; again, when in Pythonland, do as Pythonistas do, not as C programmers do.

Block rule special case

As mentioned previously, statements in a nested block of code are normally associated by being indented the same amount to the right. As one special case here, the body of a compound statement can instead appear on the same line as the header in Python, after the colon:

```
if x > y: print(x)
```

This allows us to code single-line `if` statements, single-line loops, and so on. Here again, though, this will work only if the body of the compound statement itself does not contain any compound statements. That is, only simple statements—assignments, `prints`, function calls, and the like—are allowed after the colon. Larger statements must still appear on lines by themselves. Extra parts of compound statements (such as the `else` part of an `if`, which we'll meet later) must also be on separate lines of their own. The body can consist of multiple simple statements separated by semicolons, but this tends to be frowned upon.

In general, even though it's not always required, if you keep all your statements on individual lines and always indent your nested blocks, your code will be easier to read and change in the future. Moreover, some code profiling and coverage tools may not be able to distinguish between multiple statements squeezed onto a single line or the header and body of a one-line compound statement. It is almost always to your advantage to keep things simple in Python.

To see a prime and common exception to one of these rules in action, however (the use of a single-line `if` statement to break out of a loop), let's move on to the next section and write some real code.

A Quick Example: Interactive Loops

We'll see all these syntax rules in action when we tour Python's specific compound statements in the next few chapters, but they work the same everywhere in the Python language. To get started, let's work through a brief, realistic example that demonstrates the way that statement syntax and statement nesting come together in practice, and introduces a few statements along the way.

A Simple Interactive Loop

Suppose you're asked to write a Python program that interacts with a user in a console window. Maybe you're accepting inputs to send to a database, or reading numbers to be used in a calculation. Regardless of the purpose, you need to code a loop that reads one or more inputs from a user typing on a keyboard, and prints back a result for each. In other words, you need to write a classic read/evaluate/print loop program.

In Python, typical boilerplate code for such an interactive loop might look like this:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(reply.upper())
```

This code makes use of a few new ideas:

- The code leverages the Python `while` loop, Python's most general looping statement. We'll study the `while` statement in more detail later, but in short, it consists of the word `while`, followed by an expression that is interpreted as a true or false result, followed by a nested block of code that is repeated while the test at the top is true (the word `True` here is considered always true).
- The `input` built-in function we met earlier in the book is used here for general console input—it prints its optional argument string as a prompt and returns the user's typed reply as a string.
- A single-line `if` statement that makes use of the special rule for nested blocks also appears here: the body of the `if` appears on the header line after the colon instead

of being indented on a new line underneath it. This would work either way, but as it's coded, we've saved an extra line.

- Finally, the Python `break` statement is used to exit the loop immediately—it simply jumps out of the loop statement altogether, and the program continues after the loop. Without this exit statement, the `while` would loop forever, as its test is always true.

In effect, this combination of statements essentially means “read a line from the user and print it in uppercase until the user enters the word ‘stop.’” There are other ways to code such a loop, but the form used here is very common in Python code.

Notice that all three lines nested under the `while` header line are indented the same amount—because they line up vertically in a column this way, they are the block of code that is associated with the `while` test and repeated. Either the end of the source file or a lesser-indented statement will terminate the loop body block.

When run, here is the sort of interaction we get from this code:

```
Enter text:spam
SPAM
Enter text:42
42
Enter text:stop
```



Version skew note: This example is coded for Python 3.0. If you are working in Python 2.6 or earlier, the code works the same, but you should use `raw_input` instead of `input`, and you can omit the outer parentheses in `print` statements. In 3.0 the former was renamed, and the latter is a built-in function instead of a statement (more on `prints` in the next chapter).

Doing Math on User Inputs

Our script works, but now suppose that instead of converting a text string to uppercase, we want to do some math with numeric input—squaring it, for example, perhaps in some misguided effort to discourage users who happen to be obsessed with youth. We might try statements like these to achieve the desired effect:

```
>>> reply = '20'
>>> reply ** 2
...error text omitted...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

This won't quite work in our script, though, because (as discussed in the prior part of the book) Python won't convert object types in expressions unless they are all numeric, and input from a user is always returned to our script as a string. We cannot raise a string of digits to a power unless we convert it manually to an integer:

```
>>> int(reply) ** 2
400
```

Armed with this information, we can now recode our loop to perform the necessary math. Type the following in a file to test it:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    print(int(reply) ** 2)
print('Bye')
```

This script uses a single-line `if` statement to exit on “stop” as before, but it also converts inputs to perform the required math. This version also adds an exit message at the bottom. Because the `print` statement in the last line is not indented as much as the nested block of code, it is not considered part of the loop body and will run only once, after the loop is exited:

```
Enter text:2
4
Enter text:40
1600
Enter text:stop
Bye
```

One note here: I’m assuming that this code is stored in and run from a script file. If you are entering this code interactively, be sure to include a blank line (i.e., press Enter twice) before the final `print` statement, to terminate the loop. The final `print` doesn’t quite make sense in interactive mode, though (you’ll have to code it after interacting with the loop!).

Handling Errors by Testing Inputs

So far so good, but notice what happens when the input is invalid:

```
Enter text:xxx
...error text omitted...
ValueError: invalid literal for int() with base 10: 'xxx'
```

The built-in `int` function raises an exception here in the face of a mistake. If we want our script to be robust, we can check the string’s content ahead of time with the string object’s `isdigit` method:

```
>>> S = '123'
>>> T = 'xxx'
>>> S.isdigit(), T.isdigit()
(True, False)
```

This also gives us an excuse to further nest the statements in our example. The following new version of our interactive script uses a full-blown `if` statement to work around the exception on errors:

```
while True:
    reply = input('Enter text:')
```

```

if reply == 'stop':
    break
elif not reply.isdigit():
    print('Bad!' * 8)
else:
    print(int(reply) ** 2)
print('Bye')

```

We'll study the `if` statement in more detail in [Chapter 12](#), but it's a fairly lightweight tool for coding logic in scripts. In its full form, it consists of the word `if` followed by a test and an associated block of code, one or more optional `elif` (“else if”) tests and code blocks, and an optional `else` part, with an associated block of code at the bottom to serve as a default. Python runs the block of code associated with the first test that is true, working from top to bottom, or the `else` part if all tests are false.

The `if`, `elif`, and `else` parts in the preceding example are associated as part of the same statement because they all line up vertically (i.e., share the same level of indentation). The `if` statement spans from the word `if` to the start of the `print` statement on the last line of the script. In turn, the entire `if` block is part of the `while` loop because all of it is indented under the loop's header line. Statement nesting is natural once you get the hang of it.

When we run our new script, its code catches errors before they occur and prints an (arguably silly) error message to demonstrate:

```

Enter text:5
25
Enter text:xyz
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:10
100
Enter text:stop

```

Handling Errors with `try` Statements

The preceding solution works, but as you'll see later in the book, the most general way to handle errors in Python is to catch and recover from them completely using the Python `try` statement. We'll explore this statement in depth in [Part VII](#) of this book, but as a preview, using a `try` here can lead to code that some would claim is simpler than the prior version:

```

while True:
    reply = input('Enter text:')
    if reply == 'stop': break
    try:
        num = int(reply)
    except:
        print('Bad!' * 8)
    else:
        print(int(reply) ** 2)
print('Bye')

```


This version works exactly like the previous one, but we've replaced the explicit error check with code that assumes the conversion will work and wraps it up in an exception handler for cases when it doesn't. This `try` statement is composed of the word `try`, followed by the main block of code (the action we are trying to run), followed by an `except` part that gives the exception handler code and an `else` part to be run if no exception is raised in the `try` part. Python first runs the `try` part, then runs either the `except` part (if an exception occurs) or the `else` part (if no exception occurs).

In terms of statement nesting, because the words `try`, `except`, and `else` are all indented to the same level, they are all considered part of the same single `try` statement. Notice that the `else` part is associated with the `try` here, not the `if`. As we've seen, `else` can appear in `if` statements in Python, but it can also appear in `try` statements and loops—its indentation tells you what statement it is a part of. In this case, the `try` statement spans from the word `try` through the code indented under the word `else`, because the `else` is indented to the same level as `try`. The `if` statement in this code is a one-liner and ends after the `break`.

Again, we'll come back to the `try` statement later in this book. For now, be aware that because `try` can be used to intercept any error, it reduces the amount of error-checking code you have to write, and it's a very general approach to dealing with unusual cases. If we wanted to support input of floating-point numbers instead of just integers, for example, using `try` would be much easier than manual error testing—we could simply run a `float` call and catch its exceptions, instead of trying to analyze all possible floating-point syntax.

Nesting Code Three Levels Deep

Let's look at one last mutation of our script. Nesting can take us even further if we need it to—we could, for example, branch to one of a set of alternatives based on the relative magnitude of a valid input:

```
while True:
    reply = input('Enter text:')
    if reply == 'stop':
        break
    elif not reply.isdigit():
        print('Bad!' * 8)
    else:
        num = int(reply)
        if num < 20:
            print('low')
        else:
            print(num ** 2)
print('Bye')
```

This version includes an `if` statement nested in the `else` clause of another `if` statement, which is in turn nested in the `while` loop. When code is conditional, or repeated like this, we simply indent it further to the right. The net effect is like that of the prior versions, but we'll now print "low" for numbers less than 20:

```
Enter text:19
low
Enter text:20
400
Enter text:spam
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
Enter text:stop
Bye
```

Chapter Summary

That concludes our quick look at Python statement syntax. This chapter introduced the general rules for coding statements and blocks of code. As you've learned, in Python we normally code one statement per line and indent all the statements in a nested block the same amount (indentation is part of Python's syntax). However, we also looked at a few exceptions to these rules, including continuation lines and single-line tests and loops. Finally, we put these ideas to work in an interactive script that demonstrated a handful of statements and showed statement syntax in action.

In the next chapter, we'll start to dig deeper by going over each of Python's basic procedural statements in depth. As you'll see, though, all statements follow the same general rules introduced here.

Test Your Knowledge: Quiz

1. What three things are required in a C-like language but omitted in Python?
2. How is a statement normally terminated in Python?
3. How are the statements in a nested block of code normally associated in Python?
4. How can you make a single statement span multiple lines?
5. How can you code a compound statement on a single line?
6. Is there any valid reason to type a semicolon at the end of a statement in Python?
7. What is a `try` statement for?
8. What is the most common coding mistake among Python beginners?

Test Your Knowledge: Answers

1. C-like languages require parentheses around the tests in some statements, semicolons at the end of each statement, and braces around a nested block of code.
2. The end of a line terminates the statement that appears on that line. Alternatively, if more than one statement appears on the same line, they can be terminated with semicolons; similarly, if a statement spans many lines, you must terminate it by closing a bracketed syntactic pair.
3. The statements in a nested block are all indented the same number of tabs or spaces.
4. A statement can be made to span many lines by enclosing part of it in parentheses, square brackets, or curly braces; the statement ends when Python sees a line that contains the closing part of the pair.
5. The body of a compound statement can be moved to the header line after the colon, but only if the body consists of only noncompound statements.
6. Only when you need to squeeze more than one statement onto a single line of code. Even then, this only works if all the statements are noncompound, and it's discouraged because it can lead to code that is difficult to read.
7. The `try` statement is used to catch and recover from exceptions (errors) in a Python script. It's usually an alternative to manually checking for errors in your code.
8. Forgetting to type the colon character at the end of the header line in a compound statement is the most common beginner's mistake. If you haven't made it yet, you probably will soon!

Assignments, Expressions, and Prints

Now that we've had a quick introduction to Python statement syntax, this chapter begins our in-depth tour of specific Python statements. We'll begin with the basics: assignment statements, expression statements, and print operations. We've already seen all of these in action, but here we'll fill in important details we've skipped so far. Although they're fairly simple, as you'll see, there are optional variations for each of these statement types that will come in handy once you begin writing real Python programs.

Assignment Statements

We've been using the Python assignment statement for a while to assign objects to names. In its basic form, you write the *target* of an assignment on the left of an equals sign, and the *object* to be assigned on the right. The target on the left may be a name or object component, and the object on the right can be an arbitrary expression that computes an object. For the most part, assignments are straightforward, but here are a few properties to keep in mind:

- **Assignments create object references.** As discussed in [Chapter 6](#), Python assignments store references to objects in names or data structure components. They always create references to objects instead of copying the objects. Because of that, Python variables are more like pointers than data storage areas.
- **Names are created when first assigned.** Python creates a variable name the first time you assign it a value (i.e., an object reference), so there's no need to predeclare names ahead of time. Some (but not all) data structure slots are created when assigned, too (e.g., dictionary entries, some object attributes). Once assigned, a name is replaced with the value it references whenever it appears in an expression.
- **Names must be assigned before being referenced.** It's an error to use a name to which you haven't yet assigned a value. Python raises an exception if you try, rather than returning some sort of ambiguous default value; if it returned a default instead, it would be more difficult for you to spot typos in your code.

- **Some operations perform assignments implicitly.** In this section we're concerned with the `=` statement, but assignment occurs in many contexts in Python. For instance, we'll see later that module imports, function and class definitions, `for` loop variables, and function arguments are all implicit assignments. Because assignment works the same everywhere it pops up, all these contexts simply bind names to object references at runtime.

Assignment Statement Forms

Although assignment is a general and pervasive concept in Python, we are primarily interested in assignment *statements* in this chapter. [Table 11-1](#) illustrates the different assignment statement forms in Python.

Table 11-1. Assignment statement forms

Operation	Interpretation
<code>spam = 'Spam'</code>	Basic form
<code>spam, ham = 'yum', 'YUM'</code>	Tuple assignment (positional)
<code>[spam, ham] = ['yum', 'YUM']</code>	List assignment (positional)
<code>a, b, c, d = 'spam'</code>	Sequence assignment, generalized
<code>a, *b = 'spam'</code>	Extended sequence unpacking (Python 3.0)
<code>spam = ham = 'lunch'</code>	Multiple-target assignment
<code>spams += 42</code>	Augmented assignment (equivalent to <code>spams = spams + 42</code>)

The first form in [Table 11-1](#) is by far the most common: binding a name (or data structure component) to a single object. In fact, you could get all your work done with this basic form alone. The other table entries represent special forms that are all optional, but that programmers often find convenient in practice:

Tuple- and list-unpacking assignments

The second and third forms in the table are related. When you code a tuple or list on the left side of the `=`, Python pairs objects on the right side with targets on the left by position and assigns them from left to right. For example, in the second line of [Table 11-1](#), the name `spam` is assigned the string `'yum'`, and the name `ham` is bound to the string `'YUM'`. In this case Python internally makes a tuple of the items on the right, which is why this is called tuple-unpacking assignment.

Sequence assignments

In recent versions of Python, tuple and list assignments have been generalized into instances of what we now call *sequence assignment*—any sequence of names can be assigned to any sequence of values, and Python assigns the items one at a time by position. We can even mix and match the types of the sequences involved. The fourth line in [Table 11-1](#), for example, pairs a tuple of names with a string of characters: `a` is assigned `'s'`, `b` is assigned `'p'`, and so on.

Extended sequence unpacking

In Python 3.0, a new form of sequence assignment allows us to be more flexible in how we select portions of a sequence to assign. The fifth line in [Table 11-1](#), for example, matches `a` with the first character in the string on the right and `b` with the rest: `a` is assigned `'s'`, and `b` is assigned `'pam'`. This provides a simpler alternative to assigning the results of manual slicing operations.

Multiple-target assignments

The sixth line in [Table 11-1](#) shows the multiple-target form of assignment. In this form, Python assigns a reference to the same object (the object farthest to the right) to all the targets on the left. In the table, the names `spam` and `ham` are both assigned references to the same string object, `'lunch'`. The effect is the same as if we had coded `ham = 'lunch'` followed by `spam = ham`, as `ham` evaluates to the original string object (i.e., not a separate copy of that object).

Augmented assignments

The last line in [Table 11-1](#) is an example of *augmented assignment*—a shorthand that combines an expression and an assignment in a concise way. Saying `spam += 42`, for example, has the same effect as `spam = spam + 42`, but the augmented form requires less typing and is generally quicker to run. In addition, if the subject is mutable and supports the operation, an augmented assignment may run even quicker by choosing an in-place update operation instead of an object copy. There is one augmented assignment statement for every binary expression operator in Python.

Sequence Assignments

We’ve already used basic assignments in this book. Here are a few simple examples of sequence-unpacking assignments in action:

```
% python
>>> nudge = 1
>>> wink = 2
>>> A, B = nudge, wink           # Tuple assignment
>>> A, B                         # Like A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink]      # List assignment
>>> C, D
(1, 2)
```

Notice that we really are coding two tuples in the third line in this interaction—we’ve just omitted their enclosing parentheses. Python pairs the values in the tuple on the right side of the assignment operator with the variables in the tuple on the left side and assigns the values one at a time.

Tuple assignment leads to a common coding trick in Python that was introduced in a solution to the exercises at the end of [Part II](#). Because Python creates a temporary tuple that saves the original values of the variables on the right while the statement runs,

unpacking assignments are also a way to *swap* two variables' values without creating a temporary variable of your own—the tuple on the right remembers the prior values of the variables automatically:

```
>>> nudge = 1
>>> wink = 2
>>> nudge, wink = wink, nudge      # Tuples: swaps values
>>> nudge, wink                   # Like T = nudge; nudge = wink; wink = T
(2, 1)
```

In fact, the original tuple and list assignment forms in Python have been generalized to accept any type of sequence on the right as long as it is of the same length as the sequence on the left. You can assign a tuple of values to a list of variables, a string of characters to a tuple of variables, and so on. In all cases, Python assigns items in the sequence on the right to variables in the sequence on the left by position, from left to right:

```
>>> [a, b, c] = (1, 2, 3)          # Assign tuple of values to list of names
>>> a, c
(1, 3)
>>> (a, b, c) = "ABC"              # Assign string of characters to tuple
>>> a, c
('A', 'C')
```

Technically speaking, sequence assignment actually supports any *iterable* object on the right, not just any sequence. This is a more general concept that we will explore in Chapters 14 and 20.

Advanced sequence assignment patterns

Although we can mix and match sequence types around the = symbol, we must have the *same number* of items on the right as we have variables on the left, or we'll get an error. Python 3.0 allows us to be more general with extended unpacking syntax, described in the next section. But normally, and always in Python 2.X, the number of items in the assignment target and subject must match:

```
>>> string = 'SPAM'
>>> a, b, c, d = string              # Same number on both sides
>>> a, d
('S', 'M')

>>> a, b, c = string                 # Error if not
...error text omitted...
ValueError: too many values to unpack
```

To be more general, we can slice. There are a variety of ways to employ slicing to make this last case work:

```
>>> a, b, c = string[0], string[1], string[2:]    # Index and slice
>>> a, b, c
('S', 'P', 'AM')

>>> a, b, c = list(string[:2]) + [string[2:]]     # Slice and concatenate
>>> a, b, c
```



```

('S', 'P', 'AM')

>>> a, b = string[:2]                                # Same, but simpler
>>> c = string[2:]
>>> a, b, c
('S', 'P', 'AM')

>>> (a, b), c = string[:2], string[2:]                # Nested sequences
>>> a, b, c
('S', 'P', 'AM')

```

As the last example in this interaction demonstrates, we can even assign *nested* sequences, and Python unpacks their parts according to their shape, as expected. In this case, we are assigning a tuple of two items, where the first item is a nested sequence (a string), exactly as though we had coded it this way:

```

>>> ((a, b), c) = ('SP', 'AM')                       # Paired by shape and position
>>> a, b, c
('S', 'P', 'AM')

```

Python pairs the first string on the right ('SP') with the first tuple on the left ((a, b)) and assigns one character at a time, before assigning the entire second string ('AM') to the variable c all at once. In this event, the sequence-nesting shape of the object on the left must match that of the object on the right. Nested sequence assignment like this is somewhat advanced, and rare to see, but it can be convenient for picking out the parts of data structures with known shapes.

For example, we'll see in [Chapter 13](#) that this technique also works in **for** loops, because loop items are assigned to the target given in the loop header:

```

for (a, b, c) in [(1, 2, 3), (4, 5, 6)]: ...           # Simple tuple assignment

for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: ...    # Nested tuple assignment

```

In a note in [Chapter 18](#), we'll also see that this nested tuple (really, sequence) unpacking assignment form works for function argument lists in Python 2.6 (though not in 3.0), because function arguments are passed by assignment as well:

```

def f(((a, b), c)):                                   # For arguments too in Python 2.6, but not 3.0
    f(((1, 2), 3))

```

Sequence-unpacking assignments also give rise to another common coding idiom in Python—assigning an integer series to a set of variables:

```

>>> red, green, blue = range(3)
>>> red, blue
(0, 2)

```

This initializes the three names to the integer codes 0, 1, and 2, respectively (it's Python's equivalent of the *enumerated* data types you may have seen in other languages). To make sense of this, you need to know that the **range** built-in function generates a list of successive integers:

```
>>> range(3)                                     # Use list(range(3)) in Python 3.0
[0, 1, 2]
```

Because `range` is commonly used in `for` loops, we'll say more about it in [Chapter 13](#).

Another place you may see a tuple assignment at work is for splitting a sequence into its front and the rest in loops like this:

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, L = L[0], L[1:]           # See next section for 3.0 alternative
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

The tuple assignment in the loop here could be coded as the following two lines instead, but it's often more convenient to string them together:

```
...     front = L[0]
...     L = L[1:]
```

Notice that this code is using the list as a sort of stack data structure, which can often also be achieved with the `append` and `pop` methods of list objects; here, `front = L.pop(0)` would have much the same effect as the tuple assignment statement, but it would be an in-place change. We'll learn more about `while` loops, and other (often better) ways to step through a sequence with `for` loops, in [Chapter 13](#).

Extended Sequence Unpacking in Python 3.0

The prior section demonstrated how to use manual slicing to make sequence assignments more general. In Python 3.0 (but not 2.6), sequence assignment has been generalized to make this easier. In short, a single *starred name*, `*X`, can be used in the assignment target in order to specify a more general matching against the sequence—the starred name is assigned a list, which collects all items in the sequence not assigned to other names. This is especially handy for common coding patterns such as splitting a sequence into its “front” and “rest”, as in the preceding section's last example.

Extended unpacking in action

Let's look at an example. As we've seen, sequence assignments normally require exactly as many names in the target on the left as there are items in the subject on the right. We get an error if the lengths disagree (unless we manually sliced on the right, as shown in the prior section):

```
C:\misc> c:\python30\python
>>> seq = [1, 2, 3, 4]
>>> a, b, c, d = seq
>>> print(a, b, c, d)
1 2 3 4
```

```
>>> a, b = seq
ValueError: too many values to unpack
```

In Python 3.0, though, we can use a single starred name in the target to match more generally. In the following continuation of our interactive session, `a` matches the first item in the sequence, and `b` matches the rest:

```
>>> a, *b = seq
>>> a
1
>>> b
[2, 3, 4]
```

When a starred name is used, the number of items in the target on the left need not match the length of the subject sequence. In fact, the starred name can appear anywhere in the target. For instance, in the next interaction `b` matches the last item in the sequence, and `a` matches everything before the last:

```
>>> *a, b = seq
>>> a
[1, 2, 3]
>>> b
4
```

When the starred name appears in the middle, it collects everything between the other names listed. Thus, in the following interaction `a` and `c` are assigned the first and last items, and `b` gets everything in between them:

```
>>> a, *b, c = seq
>>> a
1
>>> b
[2, 3]
>>> c
4
```

More generally, wherever the starred name shows up, it will be assigned a list that collects every unassigned name at that position:

```
>>> a, b, *c = seq
>>> a
1
>>> b
2
>>> c
[3, 4]
```

Naturally, like normal sequence assignment, extended sequence unpacking syntax works for any sequence types, not just lists. Here it is unpacking characters in a string:

```
>>> a, *b = 'spam'
>>> a, b
('s', ['p', 'a', 'm'])
>>> a, *b, c = 'spam'
```

```
>>> a, b, c
('s', ['p', 'a'], 'm')
```

This is similar in spirit to slicing, but not exactly the same—a sequence unpacking assignment always returns a *list* for multiple matched items, whereas slicing returns a sequence of the same type as the object sliced:

```
>>> S = 'spam'

>>> S[0], S[1:]    # Slices are type-specific, * assignment always returns a list
('s', 'pam')

>>> S[0], S[1:3], S[3]
('s', 'pa', 'm')
```

Given this extension in 3.0, as long as we’re processing a list the last example of the prior section becomes even simpler, since we don’t have to manually slice to get the first and rest of the items:

```
>>> L = [1, 2, 3, 4]
>>> while L:
...     front, *L = L                # Get first, rest without slicing
...     print(front, L)
...
1 [2, 3, 4]
2 [3, 4]
3 [4]
4 []
```

Boundary cases

Although extended sequence unpacking is flexible, some boundary cases are worth noting. First, the starred name may match just a single item, but is always assigned a list:

```
>>> seq
[1, 2, 3, 4]

>>> a, b, c, *d = seq
>>> print(a, b, c, d)
1 2 3 [4]
```

Second, if there is nothing left to match the starred name, it is assigned an empty list, regardless of where it appears. In the following, *a*, *b*, *c*, and *d* have matched every item in the sequence, but Python assigns *e* an empty list instead of treating this as an error case:

```
>>> a, b, c, d, *e = seq
>>> print(a, b, c, d, e)
1 2 3 4 []

>>> a, b, *e, c, d = seq
>>> print(a, b, c, d, e)
1 2 3 4 []
```

Finally, errors can still be triggered if there is more than one starred name, if there are too few values and no star (as before), and if the starred name is not itself coded inside a sequence:

```
>>> a, *b, c, *d = seq
SyntaxError: two starred expressions in assignment

>>> a, b = seq
ValueError: too many values to unpack

>>> *a = seq
SyntaxError: starred assignment target must be in a list or tuple

>>> *a, = seq
>>> a
[1, 2, 3, 4]
```

A useful convenience

Keep in mind that extended sequence unpacking assignment is just a convenience. We can usually achieve the same effects with explicit indexing and slicing (and in fact must in Python 2.X), but extended unpacking is simpler to code. The common “first, rest” splitting coding pattern, for example, can be coded either way, but slicing involves extra work:

```
>>> seq
[1, 2, 3, 4]

>>> a, *b = seq                                # First, rest
>>> a, b
(1, [2, 3, 4])

>>> a, b = seq[0], seq[1:]                      # First, rest: traditional
>>> a, b
(1, [2, 3, 4])
```

The also common “rest, last” splitting pattern can similarly be coded either way, but the new extended unpacking syntax requires noticeably fewer keystrokes:

```
>>> *a, b = seq                                # Rest, last
>>> a, b
([1, 2, 3], 4)

>>> a, b = seq[:-1], seq[-1]                  # Rest, last: traditional
>>> a, b
([1, 2, 3], 4)
```

Because it is not only simpler but, arguably, more natural, extended sequence unpacking syntax will likely become widespread in Python code over time.

Application to for loops

Because the loop variable in the `for` loop statement can be any assignment target, extended sequence assignment works here too. We met the `for` loop iteration tool briefly in [Part II](#) and will study it formally in [Chapter 13](#). In Python 3.0, extended assignments may show up after the word `for`, where a simple variable name is more commonly used:

```
for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
    ...
```

When used in this context, on each iteration Python simply assigns the next tuple of values to the tuple of names. On the first loop, for example, it's as if we'd run the following assignment statement:

```
a, *b, c = (1, 2, 3, 4)                # b gets [2, 3]
```

The names `a`, `b`, and `c` can be used within the loop's code to reference the extracted components. In fact, this is really not a special case at all, but just an instance of general assignment at work. As we saw earlier in this chapter, we can do the same thing with simple tuple assignment in both Python 2.X and 3.X:

```
for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:    # a, b, c = (1, 2, 3), ...
```

And we can always emulate 3.0's extended assignment behavior in 2.6 by manually slicing:

```
for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:  
    a, b, c = all[0], all[1:3], all[3]
```

Since we haven't learned enough to get more detailed about the syntax of `for` loops, we'll return to this topic in [Chapter 13](#).

Multiple-Target Assignments

A multiple-target assignment simply assigns all the given names to the object all the way to the right. The following, for example, assigns the three variables `a`, `b`, and `c` to the string `'spam'`:

```
>>> a = b = c = 'spam'  
>>> a, b, c  
( 'spam', 'spam', 'spam' )
```

This form is equivalent to (but easier to code than) these three assignments:

```
>>> c = 'spam'  
>>> b = c  
>>> a = b
```

Multiple-target assignment and shared references

Keep in mind that there is just one object here, shared by all three variables (they all wind up pointing to the same object in memory). This behavior is fine for immutable types—for example, when initializing a set of counters to zero (recall that variables

must be assigned before they can be used in Python, so you must initialize counters to zero before you can start adding to them):

```
>>> a = b = 0
>>> b = b + 1
>>> a, b
(0, 1)
```

Here, changing `b` only changes `b` because numbers do not support in-place changes. As long as the object assigned is immutable, it's irrelevant if more than one name references it.

As usual, though, we have to be more cautious when initializing variables to an empty mutable object such as a list or dictionary:

```
>>> a = b = []
>>> b.append(42)
>>> a, b
([42], [42])
```

This time, because `a` and `b` reference the same object, appending to it in-place through `b` will impact what we see through `a` as well. This is really just another example of the shared reference phenomenon we first met in [Chapter 6](#). To avoid the issue, initialize mutable objects in separate statements instead, so that each creates a distinct empty object by running a distinct literal expression:

```
>>> a = []
>>> b = []
>>> b.append(42)
>>> a, b
([], [42])
```

Augmented Assignments

Beginning with Python 2.0, the set of additional assignment statement formats listed in [Table 11-2](#) became available. Known as *augmented assignments*, and borrowed from the C language, these formats are mostly just shorthand. They imply the combination of a binary expression and an assignment. For instance, the following two formats are now roughly equivalent:

<code>x = x + y</code>	<i># Traditional form</i>
<code>x += y</code>	<i># Newer augmented form</i>

Table 11-2. Augmented assignment statements

<code>x += y</code>	<code>x &= y</code>	<code>x -= y</code>	<code>x = y</code>
<code>x *= y</code>	<code>x ^= y</code>	<code>x /= y</code>	<code>x >>= y</code>
<code>x %= y</code>	<code>x <=<= y</code>	<code>x **= y</code>	<code>x //>= y</code>

Augmented assignment works on any type that supports the implied binary expression. For example, here are two ways to add 1 to a name:

```

>>> x = 1
>>> x = x + 1           # Traditional
>>> x
2
>>> x += 1             # Augmented
>>> x
3

```

When applied to a string, the augmented form performs concatenation instead. Thus, the second line here is equivalent to typing the longer `S = S + "SPAM"`:

```

>>> S = "spam"
>>> S += "SPAM"         # Implied concatenation
>>> S
'spamSPAM'

```

As shown in [Table 11-2](#), there are analogous augmented assignment forms for every Python binary expression operator (i.e., each operator with values on the left and right side). For instance, `X *= Y` multiplies and assigns, `X >>= Y` shifts right and assigns, and so on. `X //= Y` (for floor division) was added in version 2.2.

Augmented assignments have three advantages:*

- There's less for you to type. Need I say more?
- The left side only has to be evaluated once. In `X += Y`, `X` may be a complicated object expression. In the augmented form, it only has to be evaluated once. However, in the long form, `X = X + Y`, `X` appears twice and must be run twice. Because of this, augmented assignments usually run faster.
- The optimal technique is automatically chosen. That is, for objects that support in-place changes, the augmented forms automatically perform in-place change operations instead of slower copies.

The last point here requires a bit more explanation. For augmented assignments, in-place operations may be applied for mutable objects as an optimization. Recall that lists can be extended in a variety of ways. To add a single item to the end of a list, we can concatenate or call `append`:

```

>>> L = [1, 2]
>>> L = L + [3]         # Concatenate: slower
>>> L
[1, 2, 3]
>>> L.append(4)         # Faster, but in-place
>>> L
[1, 2, 3, 4]

```

* C/C++ programmers take note: although Python now supports statements like `X += Y`, it still does not have C's auto-increment/decrement operators (e.g., `X++`, `--X`). These don't quite map to the Python object model because Python has no notion of in-place changes to immutable objects like numbers.

And to add a set of items to the end, we can either concatenate again or call the list `extend` method:[†]

```
>>> L = L + [5, 6]           # Concatenate: slower
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])        # Faster, but in-place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

In both cases, concatenation is less prone to the side effects of shared object references but will generally run slower than the in-place equivalent. Concatenation operations must create a new object, copy in the list on the left, and then copy in the list on the right. By contrast, in-place method calls simply add items at the end of a memory block.

When we use augmented assignment to extend a list, we can forget these details—for example, Python automatically calls the quicker `extend` method instead of using the slower concatenation operation implied by `+`:

```
>>> L += [9, 10]            # Mapped to L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Augmented assignment and shared references

This behavior is usually what we want, but notice that it implies that the `+=` is an in-place change for lists; thus, it is not exactly like `+` concatenation, which always makes a new object. As for all shared reference cases, this difference might matter if other names reference the object being changed:

```
>>> L = [1, 2]
>>> M = L                   # L and M reference the same object
>>> L = L + [3, 4]          # Concatenation makes a new object
>>> L, M                    # Changes L but not M
([1, 2, 3, 4], [1, 2])

>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]             # But += really means extend
>>> L, M                    # M sees the in-place change too!
([1, 2, 3, 4], [1, 2, 3, 4])
```

This only matters for mutables like lists and dictionaries, and it is a fairly obscure case (at least, until it impacts your code!). As always, make copies of your mutable objects if you need to break the shared reference structure.

[†] As suggested in [Chapter 6](#), we can also use slice assignment (e.g., `L[len(L):] = [11,12,13]`), but this works roughly the same as the simpler list `extend` method.

Variable Name Rules

Now that we’ve explored assignment statements, it’s time to get more formal about the use of variable names. In Python, names come into existence when you assign values to them, but there are a few rules to follow when picking names for things in your programs:

Syntax: (underscore or letter) + (any number of letters, digits, or underscores)

Variable names must start with an underscore or letter, which can be followed by any number of letters, digits, or underscores. `_spam`, `spam`, and `Spam_1` are legal names, but `1_Spam`, `spam$`, and `@#!` are not.

Case matters: SPAM is not the same as spam

Python always pays attention to case in programs, both in names you create and in reserved words. For instance, the names `X` and `x` refer to two different variables. For portability, case also matters in the names of imported module files, even on platforms where the filesystems are case-insensitive.

Reserved words are off-limits

Names you define cannot be the same as words that mean special things in the Python language. For instance, if you try to use a variable name like `class`, Python will raise a syntax error, but `klass` and `Class` work fine. [Table 11-3](#) lists the words that are currently reserved (and hence off-limits for names of your own) in Python.

Table 11-3. Python 3.0 reserved words

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

[Table 11-3](#) is specific to Python 3.0. In Python 2.6, the set of reserved words differs slightly:

- `print` is a reserved word, because printing is a statement, not a built-in (more on this later in this chapter).
- `exec` is a reserved word, because it is a statement, not a built-in function.
- `nonlocal` is not a reserved word because this statement is not available.

In older Pythons the story is also more or less the same, with a few variations:

- `with` and `as` were not reserved until 2.6, when context managers were officially enabled.
- `yield` was not reserved until Python 2.3, when generator functions were enabled.
- `yield` morphed from statement to expression in 2.5, but it’s still a reserved word, not a built-in function.

As you can see, most of Python’s reserved words are all lowercase. They are also all truly reserved—unlike names in the built-in scope that you will meet in the next part of this book, you cannot redefine reserved words by assignment (e.g., `and = 1` results in a syntax error).‡

Besides being of mixed case, the first three entries in [Table 11-3](#), `True`, `False`, and `None`, are somewhat unusual in meaning—they also appear in the built-in scope of Python described in [Chapter 17](#), and they are technically names assigned to objects. They are truly reserved in all other senses, though, and cannot be used for any other purpose in your script other than that of the objects they represent. All the other reserved words are hardwired into Python’s syntax and can appear only in the specific contexts for which they are intended.

Furthermore, because module names in `import` statements become variables in your scripts, variable name constraints extend to your *module filenames* too. For instance, you can code files called *and.py* and *my-code.py* and run them as top-level scripts, but you cannot import them: their names without the “.py” extension become *variables* in your code and so must follow all the variable rules just outlined. Reserved words are off-limits, and dashes won’t work, though underscores will. We’ll revisit this idea in [Part V](#) of this book.

Python’s Deprecation Protocol

It is interesting to note how reserved word changes are gradually phased into the language. When a new feature might break existing code, Python normally makes it an option and begins issuing “deprecation” warnings one or more releases before the feature is officially enabled. The idea is that you should have ample time to notice the warnings and update your code before migrating to the new release. This is not true for major new releases like 3.0 (which breaks existing code freely), but it is generally true in other cases.

For example, `yield` was an optional extension in Python 2.2, but is a standard keyword as of 2.3. It is used in conjunction with generator functions. This was one of a small handful of instances where Python broke with backward compatibility. Still, `yield` was phased in over time: it began generating deprecation warnings in 2.2 and was not enabled until 2.3.

‡ In the Jython Java-based implementation of Python, though, user-defined variable names can sometimes be the same as Python reserved words. See [Chapter 2](#) for an overview of the Jython system.

Similarly, in Python 2.6, the words `with` and `as` become new reserved words for use in context managers (a newer form of exception handling). These two words are not reserved in 2.5, unless the context manager feature is turned on manually with a `from __future__ import` (discussed later in this book). When used in 2.5, `with` and `as` generate warnings about the upcoming change—except in the version of IDLE in Python 2.5, which appears to have enabled this feature for you (that is, using these words as variable names does generate errors in 2.5, but only in its version of the IDLE GUI).

Naming conventions

Besides these rules, there is also a set of naming *conventions*—rules that are not required but are followed in normal practice. For instance, because names with two leading and trailing underscores (e.g., `__name__`) generally have special meaning to the Python interpreter, you should avoid this pattern for your own names. Here is a list of the conventions Python follows:

- Names that begin with a single underscore (`_X`) are not imported by a `from module import *` statement (described in [Chapter 22](#)).
- Names that have two leading and trailing underscores (`__X__`) are system-defined names that have special meaning to the interpreter.
- Names that begin with two underscores and do not end with two more (`__X`) are localized (“mangled”) to enclosing classes (see the discussion of pseudoprivate attributes in [Chapter 30](#)).
- The name that is just a single underscore (`_`) retains the result of the last expression when working interactively.

In addition to these Python interpreter conventions, there are various other conventions that Python programmers usually follow. For instance, later in the book we’ll see that class names commonly start with an uppercase letter and module names with a lowercase letter, and that the name `self`, though not reserved, usually has a special role in classes. In [Chapter 17](#) we’ll also study another, larger category of names known as the *built-ins*, which are predefined but not reserved (and so can be reassigned: `open = 42` works, though sometimes you might wish it didn’t!).

Names have no type, but objects do

This is mostly review, but remember that it’s crucial to keep Python’s distinction between names and objects clear. As described in [Chapter 6](#), objects have a type (e.g., integer, list) and may be mutable or not. Names (a.k.a. variables), on the other hand, are always just references to objects; they have no notion of mutability and have no associated type information, apart from the type of the object they happen to reference at a given point in time.

Thus, it's OK to assign the same name to different kinds of objects at different times:

```
>>> x = 0           # x bound to an integer object
>>> x = "Hello"     # Now it's a string
>>> x = [1, 2, 3]    # And now it's a list
```

In later examples, you'll see that this generic nature of names can be a decided advantage in Python programming. In [Chapter 17](#), you'll also learn that names also live in something called a *scope*, which defines where they can be used; the place where you assign a name determines where it is visible.[§]



For additional naming suggestions, see the previous section “[Naming conventions](#)” of Python’s semi-official style guide, known as *PEP 8*. This guide is available at <http://www.python.org/dev/peps/pep-0008>, or via a web search for “Python PEP 8.” Technically, this document formalizes coding standards for Python library code.

Though useful, the usual caveats about coding standards apply here. For one thing, PEP 8 comes with more detail than you are probably ready for at this point in the book. And frankly, it has become more complex, rigid, and subjective than it needs to be—some of its suggestions are not at all universally accepted or followed by Python programmers doing real work. Moreover, some of the most prominent companies using Python today have adopted coding standards of their own that differ.

PEP 8 does codify useful rule-of-thumb Python knowledge, though, and it’s a great read for Python beginners, as long as you take its recommendations as guidelines, not gospel.

Expression Statements

In Python, you can use an expression as a statement, too—that is, on a line by itself. But because the result of the expression won’t be saved, it usually makes sense to do so only if the expression does something useful as a side effect. Expressions are commonly used as statements in two situations:

For calls to functions and methods

Some functions and methods do lots of work without returning a value. Such functions are sometimes called *procedures* in other languages. Because they don’t return values that you might be interested in retaining, you can call these functions with expression statements.

[§] If you’ve used a more restrictive language like C++, you may be interested to know that there is no notion of C++’s `const` declaration in Python; certain objects may be *immutable*, but names can always be assigned. Python also has ways to hide names in classes and modules, but they’re not the same as C++’s declarations (if hiding attributes matters to you, see the coverage of `_X` module names in [Chapter 24](#), `__X` class names in [Chapter 30](#), and the `Private` and `Public` class decorators example in [Chapter 38](#)).

For printing values at the interactive prompt

Python echoes back the results of expressions typed at the interactive command line. Technically, these are expression statements, too; they serve as a shorthand for typing `print` statements.

Table 11-4 lists some common expression statement forms in Python. Calls to functions and methods are coded with zero or more argument objects (really, expressions that evaluate to objects) in parentheses, after the function/method name.

Table 11-4. Common Python expression statements

Operation	Interpretation
<code>spam(eggs, ham)</code>	Function calls
<code>spam.ham(eggs)</code>	Method calls
<code>spam</code>	Printing variables in the interactive interpreter
<code>print(a, b, c, sep='')</code>	Printing operations in Python 3.0
<code>yield x ** 2</code>	Yielding expression statements

The last two entries in Table 11-4 are somewhat special cases—as we’ll see later in this chapter, printing in Python 3.0 is a function call usually coded on a line by itself, and the `yield` operation in generator functions (discussed in Chapter 20) is often coded as a statement as well. Both are really just instances of expression statements.

For instance, though you normally run a `print` call on a line by itself as an expression statement, it returns a value like any other function call (its return value is `None`, the default return value for functions that don’t return anything meaningful):

```
>>> x = print('spam')      # print is a function call expression in 3.0
spam
>>> print(x)               # But it is coded as an expression statement
None
```

Also keep in mind that although expressions can appear as statements in Python, statements cannot be used as expressions. For example, Python doesn’t allow you to embed assignment statements (`=`) in other expressions. The rationale for this is that it avoids common coding mistakes; you can’t accidentally change a variable by typing `=` when you really mean to use the `==` equality test. You’ll see how to code around this when you meet the Python `while` loop in Chapter 13.

Expression Statements and In-Place Changes

This brings up a mistake that is common in Python work. Expression statements are often used to run list methods that change a list in-place:

```
>>> L = [1, 2]
>>> L.append(3)           # Append is an in-place change
>>> L
[1, 2, 3]
```

However, it's not unusual for Python newcomers to code such an operation as an assignment statement instead, intending to assign `L` to the larger list:

```
>>> L = L.append(4)          # But append returns None, not L
>>> print(L)                # So we lose our list!
None
```

This doesn't quite work, though. Calling an in-place change operation such as `append`, `sort`, or `reverse` on a list always changes the list in-place, but these methods do not return the list they have changed; instead, they return the `None` object. Thus, if you assign such an operation's result back to the variable name, you effectively lose the list (and it is probably garbage collected in the process!).

The moral of the story is, don't do this. We'll revisit this phenomenon in the section [“Common Coding Gotchas” on page 387](#) at the end of this part of the book because it can also appear in the context of some looping statements we'll meet in later chapters.

Print Operations

In Python, `print` prints things—it's simply a programmer-friendly interface to the standard output stream.

Technically, printing converts one or more objects to their textual representations, adds some minor formatting, and sends the resulting text to either standard output or another file-like stream. In a bit more detail, `print` is strongly bound up with the notions of files and streams in Python:

File object methods

In [Chapter 9](#), we learned about file object methods that write text (e.g., `file.write(str)`). Printing operations are similar, but more focused—whereas file write methods write strings to arbitrary files, `print` writes objects to the `stdout` stream by default, with some automatic formatting added. Unlike with file methods, there is no need to convert objects to strings when using print operations.

Standard output stream

The standard output stream (often known as `stdout`) is simply a default place to send a program's text output. Along with the standard input and error streams, it's one of three data connections created when your script starts. The standard output stream is usually mapped to the window where you started your Python program, unless it's been redirected to a file or pipe in your operating system's shell. Because the standard output stream is available in Python as the `stdout` file object in the built-in `sys` module (i.e., `sys.stdout`), it's possible to emulate `print` with file write method calls. However, `print` is noticeably easier to use and makes it easy to print text to other files and streams.

Printing is also one of the most visible places where Python 3.0 and 2.6 have diverged. In fact, this divergence is usually the first reason that most 2.X code won't run unchanged under 3.X. Specifically, the way you code print operations depends on which version of Python you use:

- In Python 3.X, printing is a *built-in function*, with keyword arguments for special modes.
- In Python 2.X, printing is a *statement* with specific syntax all its own.

Because this book covers both 3.0 and 2.6, we will look at each form in turn here. If you are fortunate enough to be able to work with code written for just one version of Python, feel free to pick the section that is relevant to you; however, as your circumstances may change, it probably won't hurt to be familiar with both cases.

The Python 3.0 print Function

Strictly speaking, printing is not a separate statement form in 3.0. Instead, it is simply an instance of the *expression statement* we studied in the preceding section.

The `print` built-in function is normally called on a line of its own, because it doesn't return any value we care about (technically, it returns `None`). Because it is a normal function, though, printing in 3.0 uses *standard function-call syntax*, rather than a special statement form. Because it provides special operation modes with keyword arguments, this form is both more general and supports future enhancements better.

By comparison, Python 2.6 `print` statements have somewhat ad-hoc syntax to support extensions such as end-of-line suppression and target files. Further, the 2.6 statement does not support separator specification at all; in 2.6, you wind up building strings ahead of time more often than you do in 3.0.

Call format

Syntactically, calls to the 3.0 `print` function have the following form:

```
print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout])
```

In this formal notation, items in square brackets are optional and may be omitted in a given call, and values after `=` give argument defaults. In English, this built-in function prints the textual representation of one or more **objects** separated by the string `sep` and followed by the string `end` to the stream `file`.

The `sep`, `end`, and `file` parts, if present, must be given as *keyword arguments*—that is, you must use a special “name=value” syntax to pass the arguments by name instead of position. Keyword arguments are covered in depth in [Chapter 18](#), but they're straightforward to use. The keyword arguments sent to this call may appear in any left-to-right order following the objects to be printed, and they control the `print` operation:

- **sep** is a string inserted between each object’s text, which defaults to a single space if not passed; passing an empty string suppresses separators altogether.
- **end** is a string added at the end of the printed text, which defaults to a `\n` newline character if not passed. Passing an empty string avoids dropping down to the next output line at the end of the printed text—the next `print` will keep adding to the end of the current output line.
- **file** specifies the file, standard stream, or other file-like object to which the text will be sent; it defaults to the `sys.stdout` standard output stream if not passed. Any object with a file-like `write(string)` method may be passed, but real files should be already opened for output.

The textual representation of each object to be printed is obtained by passing the object to the `str` built-in call; as we’ve seen, this built-in returns a “user friendly” display string for any object.^{||} With no arguments at all, the `print` function simply prints a newline character to the standard output stream, which usually displays a blank line.

The 3.0 `print` function in action

Printing in 3.0 is probably simpler than some of its details may imply. To illustrate, let’s run some quick examples. The following prints a variety of object types to the default standard output stream, with the default separator and end-of-line formatting added (these are the defaults because they are the most common use case):

```
C:\misc> c:\python30\python
>>>
>>> print()                                # Display a blank line

>>> x = 'spam'
>>> y = 99
>>> z = ['eggs']
>>>
>>> print(x, y, z)                          # Print 3 objects per defaults
spam 99 ['eggs']
```

There’s no need to convert objects to strings here, as would be required for file write methods. By default, `print` calls add a space between the objects printed. To suppress this, send an empty string to the `sep` keyword argument, or send an alternative separator of your choosing:

```
>>> print(x, y, z, sep='')                  # Suppress separator
spam99['eggs']
>>>
>>> print(x, y, z, sep=', ')                # Custom separator
spam, 99, ['eggs']
```

^{||} Technically, printing uses the equivalent of `str` in the internal implementation of Python, but the effect is the same. Besides this to-string conversion role, `str` is also the name of the string data type and can be used to decode Unicode strings from raw bytes with an extra encoding argument, as we’ll learn in [Chapter 36](#); this latter role is an advanced usage that we can safely ignore here.

Also by default, `print` adds an end-of-line character to terminate the output line. You can suppress this and avoid the line break altogether by passing an empty string to the `end` keyword argument, or you can pass a different terminator of your own (include a `\n` character to break the line manually):

```
>>> print(x, y, z, end='')                                # Suppress line break
spam 99 ['eggs']>>>
>>>
>>> print(x, y, z, end=''); print(x, y, z)                # Two prints, same output line
spam 99 ['eggs']spam 99 ['eggs']
>>> print(x, y, z, end='...\n')                           # Custom line end
spam 99 ['eggs']...
>>>
```

You can also combine keyword arguments to specify both separators and end-of-line strings—they may appear in any order but must appear after all the objects being printed:

```
>>> print(x, y, z, sep='...', end='!\n')                  # Multiple keywords
spam...99...['eggs']!
>>> print(x, y, z, end='!\n', sep='...')                  # Order doesn't matter
spam...99...['eggs']!
```

Here is how the `file` keyword argument is used—it directs the printed text to an open output file or other compatible object for the duration of the single `print` (this is really a form of stream redirection, a topic we will revisit later in this section):

```
>>> print(x, y, z, sep='...', file=open('data.txt', 'w')) # Print to a file
>>> print(x, y, z)                                         # Back to stdout
spam 99 ['eggs']
>>> print(open('data.txt').read())                         # Display file text
spam...99...['eggs']
```

Finally, keep in mind that the separator and end-of-line options provided by `print` operations are just conveniences. If you need to display more specific formatting, don't `print` this way. Instead, build up a more complex string ahead of time or within the `print` itself using the string tools we met in [Chapter 7](#), and `print` the string all at once:

```
>>> text = '%s: %-.4f, %05d' % ('Result', 3.14159, 42)
>>> print(text)
Result: 3.1416, 00042
>>> print('%s: %-.4f, %05d' % ('Result', 3.14159, 42))
Result: 3.1416, 00042
```

As we'll see in the next section, almost everything we've just seen about the 3.0 `print` function also applies directly to 2.6 `print` statements—which makes sense, given that the function was intended to both emulate and improve upon 2.6 printing support.

The Python 2.6 `print` Statement

As mentioned earlier, printing in Python 2.6 uses a statement with unique and specific syntax, rather than a built-in function. In practice, though, 2.6 printing is mostly a variation on a theme; with the exception of separator strings (which are supported in

3.0 but not 2.6), everything we can do with the 3.0 `print` function has a direct translation to the 2.6 `print` statement.

Statement forms

Table 11-5 lists the `print` statement’s forms in Python 2.6 and gives their Python 3.0 `print` function equivalents for reference. Notice that the *comma* is significant in `print` statements—it separates objects to be printed, and a trailing comma suppresses the end-of-line character normally added at the end of the printed text (not to be confused with tuple syntax!). The `>>` syntax, normally used as a bitwise right-shift operation, is used here as well, to specify a target output stream other than the `sys.stdout` default.

Table 11-5. Python 2.6 `print` statement forms

Python 2.6 statement	Python 3.0 equivalent	Interpretation
<code>print x, y</code>	<code>print(x, y)</code>	Print objects’ textual forms to <code>sys.stdout</code> ; add a space between the items and an end-of-line at the end
<code>print x, y,</code>	<code>print(x, y, end='')</code>	Same, but don’t add end-of-line at end of text
<code>print >> afile, x, y</code>	<code>print(x, y, file=afile)</code>	Send text to <code>myfile.write</code> , not to <code>sys.stdout.write</code>

The 2.6 `print` statement in action

Although the 2.6 `print` statement has more unique syntax than the 3.0 function, it’s similarly easy to use. Let’s turn to some basic examples again. By default, the 2.6 `print` statement adds a space between the items separated by commas and adds a line break at the end of the current output line:

```
C:\misc> c:\python26\python
>>>
>>> x = 'a'
>>> y = 'b'
>>> print x, y
a b
```

This formatting is just a default; you can choose to use it or not. To suppress the line break so you can add more text to the current line later, end your `print` statement with a comma, as shown in the second line of Table 11-5 (the following is two statements on one line, separated by a semicolon):

```
>>> print x, y,; print x, y
a b a b
```

To suppress the space between items, again, don't print this way. Instead, build up an output string using the string concatenation and formatting tools covered in [Chapter 7](#), and print the string all at once:

```
>>> print x + y
ab
>>> print '%s...%s' % (x, y)
a...b
```

As you can see, apart from their special syntax for usage modes, 2.6 `print` statements are roughly as simple to use as 3.0's function. The next section uncovers the way that files are specified in 2.6 `prints`.

Print Stream Redirection

In both Python 3.0 and 2.6, printing sends text to the standard output stream by default. However, it's often useful to send it elsewhere—to a text file, for example, to save results for later use or testing purposes. Although such redirection can be accomplished in system shells outside Python itself, it turns out to be just as easy to redirect a script's streams from within the script.

The Python “hello world” program

Let's start off with the usual (and largely pointless) language benchmark—the “hello world” program. To print a “hello world” message in Python, simply print the string per your version's print operation:

```
>>> print('hello world')           # Print a string object in 3.0
hello world

>>> print 'hello world'           # Print a string object in 2.6
hello world
```

Because expression results are echoed on the interactive command line, you often don't even need to use a `print` statement there—simply type the expressions you'd like to have printed, and their results are echoed back:

```
>>> 'hello world'                 # Interactive echoes
'hello world'
```

This code isn't exactly an earth-shattering piece of software mastery, but it serves to illustrate printing behavior. Really, the `print` operation is just an ergonomic feature of Python—it provides a simple interface to the `sys.stdout` object, with a bit of default formatting. In fact, if you enjoy working harder than you must, you can also code print operations this way:

```
>>> import sys                   # Printing the hard way
>>> sys.stdout.write('hello world\n')
hello world
```

This code explicitly calls the `write` method of `sys.stdout`—an attribute preset when Python starts up to an open file object connected to the output stream. The `print` operation hides most of those details, providing a simple tool for simple printing tasks.

Manual stream redirection

So, why did I just show you the hard way to print? The `sys.stdout` print equivalent turns out to be the basis of a common technique in Python. In general, `print` and `sys.stdout` are directly related as follows. This statement:

```
print(X, Y)                                # Or, in 2.6: print X, Y
```

is equivalent to the longer:

```
import sys
sys.stdout.write(str(X) + ' ' + str(Y) + '\n')
```

which manually performs a string conversion with `str`, adds a separator and newline with `+`, and calls the output stream's `write` method. Which would you rather code? (He says, hoping to underscore the programmer-friendly nature of prints....)

Obviously, the long form isn't all that useful for printing by itself. However, it is useful to know that this is exactly what `print` operations do because it is possible to *reassign* `sys.stdout` to something different from the standard output stream. In other words, this equivalence provides a way of making your `print` operations send their text to other places. For example:

```
import sys
sys.stdout = open('log.txt', 'a')           # Redirects prints to a file
...
print(x, y, x)                             # Shows up in log.txt
```

Here, we reset `sys.stdout` to a manually opened file named `log.txt`, located in the script's working directory and opened in append mode (so we add to its current content). After the reset, every `print` operation anywhere in the program will write its text to the end of the file `log.txt` instead of to the original output stream. The `print` operations are happy to keep calling `sys.stdout`'s `write` method, no matter what `sys.stdout` happens to refer to. Because there is just one `sys` module in your process, assigning `sys.stdout` this way will redirect every `print` anywhere in your program.

In fact, as this chapter's upcoming sidebar about `print` and `stdout` will explain, you can even reset `sys.stdout` to an object that isn't a file at all, as long as it has the expected interface: a method named `write` to receive the printed text string argument. When that object is a *class*, printed text can be routed and processed arbitrarily per a `write` method you code yourself.

This trick of resetting the output stream is primarily useful for programs originally coded with `print` statements. If you know that output should go to a file to begin with, you can always call file write methods instead. To redirect the output of a `print`-based

program, though, resetting `sys.stdout` provides a convenient alternative to changing every `print` statement or using system shell-based redirection syntax.

Automatic stream redirection

This technique of redirecting printed text by assigning `sys.stdout` is commonly used in practice. One potential problem with the last section's code, though, is that there is no direct way to restore the original output stream should you need to switch back after printing to a file. Because `sys.stdout` is just a normal file object, you can always save it and restore it if needed: #

```
C:\misc> c:\python30\python
>>> import sys
>>> temp = sys.stdout                # Save for restoring later
>>> sys.stdout = open('log.txt', 'a') # Redirect prints to a file
>>> print('spam')                   # Prints go to file, not here
>>> print(1, 2, 3)
>>> sys.stdout.close()              # Flush output to disk
>>> sys.stdout = temp               # Restore original stream

>>> print('back here')              # Prints show up here again
back here
>>> print(open('log.txt').read())    # Result of earlier prints
spam
1 2 3
```

As you can see, though, manual saving and restoring of the original output stream like this involves quite a bit of extra work. Because this crops up fairly often, a `print` extension is available to make it unnecessary.

In 3.0, the `file` keyword allows a single `print` call to send its text to a file's `write` method, without actually resetting `sys.stdout`. Because the redirection is temporary, normal `print` calls keep printing to the original output stream. In 2.6, a `print` statement that begins with a `>>` followed by an output file object (or other compatible object) has the same effect. For example, the following again sends printed text to a file named `log.txt`:

```
log = open('log.txt', 'a')           # 3.0
print(x, y, z, file=log)             # Print to a file-like object
print(a, b, c)                       # Print to original stdout

log = open('log.txt', 'a')           # 2.6
print >> log, x, y, z                 # Print to a file-like object
print a, b, c                        # Print to original stdout
```

These redirected forms of `print` are handy if you need to print to *both* files and the standard output stream in the same program. If you use these forms, however, be sure

#In both 2.6 and 3.0 you may also be able to use the `__stdout__` attribute in the `sys` module, which refers to the original value `sys.stdout` had at program startup time. You still need to restore `sys.stdout` to `sys.__stdout__` to go back to this original stream value, though. See the `sys` module documentation for more details.

to give them a file object (or an object that has the same `write` method as a file object), not a file's name string. Here is the technique in action:

```
C:\misc> c:\python30\python
>>> log = open('log.txt', 'w')
>>> print(1, 2, 3, file=log)           # 2.6: print >> log, 1, 2, 3
>>> print(4, 5, 6, file=log)
>>> log.close()
>>> print(7, 8, 9)                     # 2.6: print 7, 8, 9
7 8 9
>>> print(open('log.txt').read())
1 2 3
4 5 6
```

These extended forms of `print` are also commonly used to print error messages to the standard error stream, available to your script as the preopened file object `sys.stderr`. You can either use its file `write` methods and format the output manually, or print with redirection syntax:

```
>>> import sys
>>> sys.stderr.write(('Bad!' * 8) + '\n')
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!

>>> print('Bad!' * 8, file=sys.stderr)   # 2.6: print >> sys.stderr, 'Bad' * 8
Bad!Bad!Bad!Bad!Bad!Bad!Bad!Bad!
```

Now that you know all about print redirections, the equivalence between printing and file `write` methods should be fairly obvious. The following interaction prints both ways in 3.0, then redirects the output to an external file to verify that the same text is printed:

```
>>> X = 1; Y = 2
>>> print(X, Y)                        # Print: the easy way
1 2
>>> import sys
>>> sys.stdout.write(str(X) + ' ' + str(Y) + '\n')   # Print: the hard way
1 2
4
>>> print(X, Y, file=open('temp1', 'w'))           # Redirect text to file

>>> open('temp2', 'w').write(str(X) + ' ' + str(Y) + '\n') # Send to file manually
4
>>> print(open('temp1', 'rb').read())               # Binary mode for bytes
b'1 2\r\n'
>>> print(open('temp2', 'rb').read())
b'1 2\r\n'
```

As you can see, unless you happen to enjoy typing, print operations are usually the best option for displaying text. For another example of the equivalence between prints and file writes, watch for a 3.0 `print` function emulation example in [Chapter 18](#); it uses this code pattern to provide a general 3.0 `print` function equivalent for use in Python 2.6.

Version-Neutral Printing

Finally, if you cannot restrict your work to Python 3.0 but still want your prints to be compatible with 3.0, you have some options. For one, you can code 2.6 `print` statements and let 3.0's `2to3` conversion script translate them to 3.0 function calls automatically. See the Python 3.0 documentation for more details about this script; it attempts to translate 2.X code to run under 3.0.

Alternatively, you can code 3.0 `print` function calls in your 2.6 code, by enabling the function call variant with a statement like the following:

```
from __future__ import print_function
```

This statement changes 2.6 to support 3.0's `print` functions exactly. This way, you can use 3.0 print features and won't have to change your prints if you later migrate to 3.0.

Also keep in mind that simple prints, like those in the first row of [Table 11-5](#), work in *either* version of Python—because any expression may be enclosed in parentheses, we can always pretend to be calling a 3.0 `print` function in 2.6 by adding outer parentheses. The only downside to this is that it makes a tuple out of your printed objects if there are more than one—they will print with extra enclosing parentheses. In 3.0, for example, any number of objects may be listed in the call's parentheses:

```
C:\misc> c:\python30\python
>>> print('spam')                # 3.0 print function call syntax
spam
>>> print('spam', 'ham', 'eggs') # These are mutiple arguments
spam ham eggs
```

The first of these works the same in 2.6, but the second generates a tuple in the output:

```
C:\misc> c:\python26\python
>>> print('spam')                # 2.6 print statement, enclosing parens
spam
>>> print('spam', 'ham', 'eggs') # This is really a tuple object!
('spam', 'ham', 'eggs')
```

To be truly portable, you can format the print string as a single object, using the string formatting expression or method call, or other string tools that we studied in [Chapter 7](#):

```
>>> print('%s %s %s' % ('spam', 'ham', 'eggs'))
spam ham eggs
>>> print('{0} {1} {2}'.format('spam', 'ham', 'eggs'))
spam ham eggs
```

Of course, if you can use 3.0 exclusively you can forget such mappings entirely, but many Python programmers will at least encounter, if not write, 2.X code and systems for some time to come.



I use Python 3.0 `print` function calls throughout this book. I'll usually warn you that the results may have extra enclosing parentheses in 2.6 because multiple items are a tuple, but I sometimes don't, so please consider this note a blanket warning—if you see extra parentheses in your printed text in 2.6, either drop the parentheses in your `print` statements, recode your prints using the version-neutral scheme outlined here, or learn to love superfluous text.

Why You Will Care: `print` and `stdout`

The equivalence between the `print` operation and writing to `sys.stdout` is important. It makes it possible to reassign `sys.stdout` to any user-defined object that provides the same `write` method as files. Because the `print` statement just sends text to the `sys.stdout.write` method, you can capture printed text in your programs by assigning `sys.stdout` to an object whose `write` method processes the text in arbitrary ways.

For instance, you can send printed text to a GUI window, or tee it off to multiple destinations, by defining an object with a `write` method that does the required routing. You'll see an example of this trick when we study classes in [Part VI](#) of this book, but abstractly, it looks like this:

```
class FileFaker:
    def write(self, string):
        # Do something with printed text in string

import sys
sys.stdout = FileFaker()
print(someObjects)                # Sends to class write method
```

This works because `print` is what we will call in the next part of this book a *polymorphic* operation—it doesn't care what `sys.stdout` is, only that it has a method (i.e., interface) called `write`. This redirection to objects is made even simpler with the `file` keyword argument in 3.0 and the `>>` extended form of `print` in 2.6, because we don't need to reset `sys.stdout` explicitly—normal prints will still be routed to the `stdout` stream:

```
myobj = FileFaker()                # 3.0: Redirect to object for one print
print(someObjects, file=myobj)     # Does not reset sys.stdout

myobj = FileFaker()                # 2.6: same effect
print >> myobj, someObjects        # Does not reset sys.stdout
```

Python's built-in `input` function reads from the `sys.stdin` file, so you can intercept read requests in a similar way, using classes that implement file-like `read` methods instead. See the `input` and `while` loop example in [Chapter 10](#) for more background on this.

Notice that because printed text goes to the `stdout` stream, it's the way to print HTML in CGI scripts used on the Web. It also enables you to redirect Python script input and output at the operating system's shell command line, as usual:

```
python script.py < inputfile > outputfile  
python script.py | filterProgram
```

Python's print operation redirection tools are essentially pure-Python alternatives to these shell syntax forms.

Chapter Summary

In this chapter, we began our in-depth look at Python statements by exploring assignments, expressions, and print operations. Although these are generally simple to use, they have some alternative forms that, while optional, are often convenient in practice: augmented assignment statements and the redirection form of `print` operations, for example, allow us to avoid some manual coding work. Along the way, we also studied the syntax of variable names, stream redirection techniques, and a variety of common mistakes to avoid, such as assigning the result of an `append` method call back to a variable.

In the next chapter, we'll continue our statement tour by filling in details about the `if` statement, Python's main selection tool; there, we'll also revisit Python's syntax model in more depth and look at the behavior of Boolean expressions. Before we move on, though, the end-of-chapter quiz will test your knowledge of what you've learned here.

Test Your Knowledge: Quiz

1. Name three ways that you can assign three variables to the same value.
2. Why might you need to care when assigning three variables to a mutable object?
3. What's wrong with saying `L = L.sort()`?
4. How might you use the `print` operation to send text to an external file?

Test Your Knowledge: Answers

1. You can use multiple-target assignments (`A = B = C = 0`), sequence assignment (`A, B, C = 0, 0, 0`), or multiple assignment statements on three separate lines (`A = 0`, `B = 0`, and `C = 0`). With the latter technique, as introduced in [Chapter 10](#), you can also string the three separate statements together on the same line by separating them with semicolons (`A = 0; B = 0; C = 0`).

2. If you assign them this way:

```
A = B = C = []
```

all three names reference the same object, so changing it in-place from one (e.g., `A.append(99)`) will affect the others. This is true only for in-place changes to mutable objects like lists and dictionaries; for immutable objects such as numbers and strings, this issue is irrelevant.

3. The list `sort` method is like `append` in that it makes an in-place change to the subject list—it returns `None`, not the list it changes. The assignment back to `L` sets `L` to `None`, not to the sorted list. As we'll see later in this part of the book, a newer built-in function, `sorted`, sorts any sequence and returns a new list with the sorting result; because this is not an in-place change, its result can be meaningfully assigned to a name.
4. To print to a file for a single `print` operation, you can use 3.0's `print(X, file=F)` call form, use 2.6's extended `print >> file, X` statement form, or assign `sys.stdout` to a manually opened file before the `print` and restore the original after. You can also redirect all of a program's printed text to a file with special syntax in the system shell, but this is outside Python's scope.

if Tests and Syntax Rules

This chapter presents the Python `if` statement, which is the main statement used for selecting from alternative actions based on test results. Because this is our first in-depth look at *compound statements*—statements that embed other statements—we will also explore the general concepts behind the Python statement syntax model here in more detail than we did in the introduction in [Chapter 10](#). Because the `if` statement introduces the notion of tests, this chapter will also deal with Boolean expressions and fill in some details on truth tests in general.

if Statements

In simple terms, the Python `if` statement selects actions to perform. It's the primary selection tool in Python and represents much of the *logic* a Python program possesses. It's also our first compound statement. Like all compound Python statements, the `if` statement may contain other statements, including other `ifs`. In fact, Python lets you combine statements in a program sequentially (so that they execute one after another), and in an arbitrarily nested fashion (so that they execute only under certain conditions).

General Format

The Python `if` statement is typical of `if` statements in most procedural languages. It takes the form of an `if` test, followed by one or more optional `elif` (“else if”) tests and a final optional `else` block. The tests and the `else` part each have an associated block of nested statements, indented under a header line. When the `if` statement runs, Python executes the block of code associated with the first test that evaluates to true, or the `else` block if all tests prove false. The general form of an `if` statement looks like this:

```
if <test1>:                # if test
    <statements1>          # Associated block
elif <test2>:              # Optional elifs
    <statements2>
else:                      # Optional else
    <statements3>
```

Basic Examples

To demonstrate, let's look at a few simple examples of the `if` statement at work. All parts are optional, except the initial `if` test and its associated statements. Thus, in the simplest case, the other parts are omitted:

```
>>> if 1:
...     print('true')
...
true
```

Notice how the prompt changes to `...` for continuation lines when typing interactively in the basic interface used here; in IDLE, you'll simply drop down to an indented line instead (hit Backspace to back up). A blank line (which you can get by pressing Enter twice) terminates and runs the entire statement. Remember that `1` is Boolean `true`, so this statement's test always succeeds. To handle a false result, code the `else`:

```
>>> if not 1:
...     print('true')
... else:
...     print('false')
...
false
```

Multiway Branching

Now here's an example of a more complex `if` statement, with all its optional parts present:

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print("how's jessica?")
... elif x == 'bugs':
...     print("what's up doc?")
... else:
...     print('Run away! Run away!')
...
Run away! Run away!
```

This multiline statement extends from the `if` line through the `else` block. When it's run, Python executes the statements nested under the first test that is true, or the `else` part if all tests are false (in this example, they are). In practice, both the `elif` and `else` parts may be omitted, and there may be more than one statement nested in each section. Note that the words `if`, `elif`, and `else` are associated by the fact that they line up vertically, with the same indentation.

If you've used languages like C or Pascal, you might be interested to know that there is no `switch` or `case` statement in Python that selects an action based on a variable's value. Instead, *multiway branching* is coded either as a series of `if/elif` tests, as in the prior example, or by indexing dictionaries or searching lists. Because dictionaries and lists can be built at runtime, they're sometimes more flexible than hardcoded `if` logic:

```
>>> choice = 'ham'
>>> print({'spam': 1.25,          # A dictionary-based 'switch'
...       'ham': 1.99,          # Use has_key or get for default
...       'eggs': 0.99,
...       'bacon': 1.10}[choice])
1.99
```

Although it may take a few moments for this to sink in the first time you see it, this dictionary is a multiway branch—indexing on the key `choice` branches to one of a set of values, much like a `switch` in C. An almost equivalent but more verbose Python `if` statement might look like this:

```
>>> if choice == 'spam':
...     print(1.25)
... elif choice == 'ham':
...     print(1.99)
... elif choice == 'eggs':
...     print(0.99)
... elif choice == 'bacon':
...     print(1.10)
... else:
...     print('Bad choice')
...
1.99
```

Notice the `else` clause on the `if` here to handle the default case when no key matches. As we saw in [Chapter 8](#), dictionary defaults can be coded with `in` expressions, `get` method calls, or exception catching. All of the same techniques can be used here to code a default action in a dictionary-based multiway branch. Here's the `get` scheme at work with defaults:

```
>>> branch = {'spam': 1.25,
...           'ham': 1.99,
...           'eggs': 0.99}

>>> print(branch.get('spam', 'Bad choice'))
1.25
>>> print(branch.get('bacon', 'Bad choice'))
Bad choice
```

An `in` membership test in an `if` statement can have the same default effect:

```
>>> choice = 'bacon'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Bad choice')
...
Bad choice
```

Dictionaries are good for associating values with keys, but what about the more complicated actions you can code in the statement blocks associated with `if` statements? In [Part IV](#), you'll learn that dictionaries can also contain *functions* to represent more complex branch actions and implement general jump tables. Such functions appear as

dictionary values, may be coded as function names or `lambdas`, and are called by adding parentheses to trigger their actions; stay tuned for more on this topic in [Chapter 19](#).

Although dictionary-based multiway branching is useful in programs that deal with more dynamic data, most programmers will probably find that coding an `if` statement is the most straightforward way to perform multiway branching. As a rule of thumb in coding, when in doubt, err on the side of simplicity and readability; it's the "Pythonic" way.

Python Syntax Rules

I introduced Python's syntax model in [Chapter 10](#). Now that we're stepping up to larger statements like the `if`, this section reviews and expands on the syntax ideas introduced earlier. In general, Python has a simple, statement-based syntax. However, there are a few properties you need to know about:

- **Statements execute one after another, until you say otherwise.** Python normally runs statements in a file or nested block in order from first to last, but statements like `if` (and, as you'll see, loops) cause the interpreter to jump around in your code. Because Python's path through a program is called the *control flow*, statements such as `if` that affect it are often called *control-flow statements*.
- **Block and statement boundaries are detected automatically.** As we've seen, there are no braces or "begin/end" delimiters around blocks of code in Python; instead, Python uses the indentation of statements under a header to group the statements in a nested block. Similarly, Python statements are not normally terminated with semicolons; rather, the end of a line usually marks the end of the statement coded on that line.
- **Compound statements = header + ":" + indented statements.** All compound statements in Python follow the same pattern: a header line terminated with a colon, followed by one or more nested statements, usually indented under the header. The indented statements are called a *block* (or sometimes, a *suite*). In the `if` statement, the `elif` and `else` clauses are part of the `if`, but they are also header lines with nested blocks of their own.
- **Blank lines, spaces, and comments are usually ignored.** Blank lines are ignored in files (but not at the interactive prompt, when they terminate compound statements). Spaces inside statements and expressions are almost always ignored (except in string literals, and when used for indentation). Comments are always ignored: they start with a `#` character (not inside a string literal) and extend to the end of the current line.
- **Docstrings are ignored but are saved and displayed by tools.** Python supports an additional comment form called documentation strings (*docstrings* for short), which, unlike `#` comments, are retained at runtime for inspection. Docstrings are simply strings that show up at the top of program files and some statements. Python

ignores their contents, but they are automatically attached to objects at runtime and may be displayed with documentation tools. Docstrings are part of Python's larger documentation strategy and are covered in the last chapter in this part of the book.

As you've seen, there are no variable type declarations in Python; this fact alone makes for a much simpler language syntax than what you may be used to. However, for most new users the lack of the braces and semicolons used to mark blocks and statements in many other languages seems to be the most novel syntactic feature of Python, so let's explore what this means in more detail.

Block Delimiters: Indentation Rules

Python detects block boundaries automatically, by line *indentation*—that is, the empty space to the left of your code. All statements indented the same distance to the right belong to the same block of code. In other words, the statements within a block line up vertically, as in a column. The block ends when the end of the file or a lesser-indented line is encountered, and more deeply nested blocks are simply indented further to the right than the statements in the enclosing block.

For instance, [Figure 12-1](#) demonstrates the block structure of the following code:

```
x = 1
if x:
    y = 2
    if y:
        print('block2')
    print('block1')
print('block0')
```

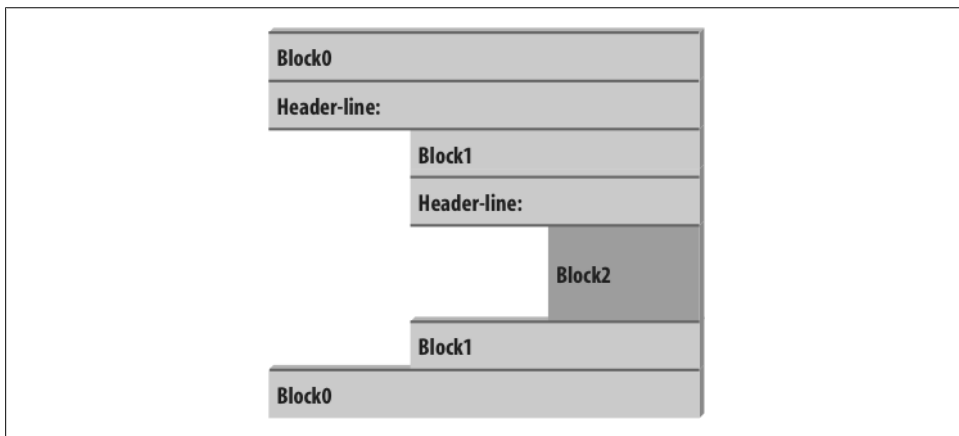


Figure 12-1. Nested blocks of code: a nested block starts with a statement indented further to the right and ends with either a statement that is indented less, or the end of the file.

This code contains three blocks: the first (the top-level code of the file) is not indented at all, the second (within the outer `if` statement) is indented four spaces, and the third (the `print` statement under the nested `if`) is indented eight spaces.

In general, top-level (unnested) code must start in column 1. Nested blocks can start in any column; indentation may consist of any number of spaces and tabs, as long as it's the same for all the statements in a given single block. That is, Python doesn't care *how* you indent your code; it only cares that it's done consistently. Four spaces or one tab per indentation level are common conventions, but there is no absolute standard in the Python world.

Indenting code is quite natural in practice. For example, the following (arguably silly) code snippet demonstrates common indentation errors in Python code:

```
x = 'SPAM'                                # Error: first line indented
if 'rubbery' in 'shrubbery':
    print(x * 8)
    x += 'NI'                             # Error: unexpected indentation
    if x.endswith('NI'):
        x *= 2
        print(x)                         # Error: inconsistent indentation
```

The properly indented version of this code looks like the following—even for an artificial example like this, proper indentation makes the code's intent much more apparent:

```
x = 'SPAM'
if 'rubbery' in 'shrubbery':
    print(x * 8)
    x += 'NI'
    if x.endswith('NI'):
        x *= 2
        print(x)                        # Prints "SPAMNISPAMNI"
```

It's important to know that the only major place in Python where whitespace matters is where it's used to the left of your code, for indentation; in most other contexts, space can be coded or not. However, indentation is really part of Python syntax, not just a stylistic suggestion: all the statements within any given single block must be indented to the same level, or Python reports a syntax error. This is intentional—because you don't need to explicitly mark the start and end of a nested block of code, some of the syntactic clutter found in other languages is unnecessary in Python.

As described in [Chapter 10](#), making indentation part of the syntax model also enforces consistency, a crucial component of readability in structured programming languages like Python. Python's syntax is sometimes described as “what you see is what you get”—the indentation of each line of code unambiguously tells readers what it is associated with. This uniform and consistent appearance makes Python code easier to maintain and reuse.

Indentation is more natural than the details might imply, and it makes your code reflect its logical structure. Consistently indented code always satisfies Python’s rules. Moreover, most text editors (including IDLE) make it easy to follow Python’s indentation model by automatically indenting code as you type it.

Avoid mixing tabs and spaces: New error checking in 3.0

One rule of thumb: although you can use spaces or tabs to indent, it’s usually not a good idea to mix the two within a block—use one or the other. Technically, tabs count for enough spaces to move the current column number up to a multiple of 8, and your code will work if you mix tabs and spaces consistently. However, such code can be difficult to change. Worse, mixing tabs and spaces makes your code difficult to read—tabs may look very different in the next programmer’s editor than they do in yours.

In fact, Python 3.0 now issues an error, for these very reasons, when a script mixes tabs and spaces for indentation inconsistently within a block (that is, in a way that makes it dependent on a tab’s equivalent in spaces). Python 2.6 allows such scripts to run, but it has a `-t` command-line flag that will warn you about inconsistent tab usage and a `-tt` flag that will issue errors for such code (you can use these switches in a command line like `python -t main.py` in a system shell window). Python 3.0’s error case is equivalent to 2.6’s `-tt` switch.

Statement Delimiters: Lines and Continuations

A statement in Python normally ends at the end of the line on which it appears. When a statement is too long to fit on a single line, though, a few special rules may be used to make it span multiple lines:

- **Statements may span multiple lines if you’re continuing an open syntactic pair.** Python lets you continue typing a statement on the next line if you’re coding something enclosed in a `()`, `{}`, or `[]` pair. For instance, expressions in parentheses and dictionary and list literals can span any number of lines; your statement doesn’t end until the Python interpreter reaches the line on which you type the closing part of the pair (a `)`, `}`, or `]`). Continuation lines (lines 2 and beyond of the statement) can start at any indentation level you like, but you should try to make them align vertically for readability if possible. This open pairs rule also covers set and dictionary comprehensions in Python 3.0.
- **Statements may span multiple lines if they end in a backslash.** This is a somewhat outdated feature, but if a statement needs to span multiple lines, you can also add a backslash (a `\` not embedded in a string literal or comment) at the end of the prior line to indicate you’re continuing on the next line. Because you can also continue by adding parentheses around most constructs, backslashes are almost never used. This approach is error-prone: accidentally forgetting a `\` usually generates a syntax error and might even cause the next line to be silently mistaken to be a new statement, with unexpected results.

- **Special rules for string literals.** As we learned in [Chapter 7](#), triple-quoted string blocks are designed to span multiple lines normally. We also learned in [Chapter 7](#) that adjacent string literals are implicitly concatenated; when used in conjunction with the open pairs rule mentioned earlier, wrapping this construct in parentheses allows it to span multiple lines.
- **Other rules.** There are a few other points to mention with regard to statement delimiters. Although uncommon, you can terminate a statement with a semicolon—this convention is sometimes used to squeeze more than one simple (noncompound) statement onto a single line. Also, comments and blank lines can appear anywhere in a file; comments (which begin with a # character) terminate at the end of the line on which they appear.

A Few Special Cases

Here's what a continuation line looks like using the open syntactic pairs rule. Delimited constructs, such as lists in square brackets, can span across any number of lines:

```
L = ["Good",
     "Bad",
     "Ugly"]                # Open pairs may span lines
```

This also works for anything in parentheses (expressions, function arguments, function headers, tuples, and generator expressions), as well as anything in curly braces (dictionaries and, in 3.0, set literals and set and dictionary comprehensions). Some of these are tools we'll study in later chapters, but this rule naturally covers most constructs that span lines in practice.

If you like using backslashes to continue lines, you can, but it's not common practice in Python:

```
if a == b and c == d and \
    d == e and f == g:
    print('olde')           # Backslashes allow continuations...
```

Because any expression can be enclosed in parentheses, you can usually use the open pairs technique instead if you need your code to span multiple lines—simply wrap a part of your statement in parentheses:

```
if (a == b and c == d and
    d == e and e == f):
    print('new')            # But parentheses usually do too
```

In fact, backslashes are frowned on, because they're too easy to not notice and too easy to omit altogether. In the following, `x` is assigned `10` with the backslash, as intended; if the backslash is accidentally omitted, though, `x` is assigned `6` instead, and no error is reported (the `+4` is a valid expression statement by itself).

In a real program with a more complex assignment, this could be the source of a very nasty bug:

```
x = 1 + 2 + 3 \           # Omitting the \ makes this very different
+4
```

As another special case, Python allows you to write more than one noncompound statement (i.e., statements without nested statements) on the same line, separated by semicolons. Some coders use this form to save program file real estate, but it usually makes for more readable code if you stick to one statement per line for most of your work:

```
x = 1; y = 2; print(x)    # More than one simple statement
```

As we learned in [Chapter 7](#), triple-quoted string literals span lines too. In addition, if two string literals appear next to each other, they are concatenated as if a + had been added between them—when used in conjunction with the open pairs rule, wrapping in parentheses allows this form to span multiple lines. For example, the first of the following inserts newline characters at line breaks and assigns `S` to `'\naaaa\nbbbb\ncccc'`, and the second implicitly concatenates and assigns `S` to `'aaaabbbbcccc'`; comments are ignored in the second form, but included in the string in the first:

```
S = """
aaaa
bbbb
cccc"""

S = ('aaaa'
     'bbbb'
     'cccc')           # Comments here are ignored
```

Finally, Python lets you move a compound statement’s body up to the header line, provided the body is just a simple (noncompound) statement. You’ll most often see this used for simple `if` statements with a single test and action:

```
if 1: print('hello')    # Simple statement on header line
```

You can combine some of these special cases to write code that is difficult to read, but I don’t recommend it; as a rule of thumb, try to keep each statement on a line of its own, and indent all but the simplest of blocks. Six months down the road, you’ll be happy you did.

* Frankly, it’s surprising that this wasn’t removed in Python 3.0, given some of its other changes! (See [Table P-2](#) of the Preface for a list of 3.0 removals; some seem fairly innocuous in comparison with the dangers inherent in backslash continuations.) Then again, this book’s goal is Python instruction, not populist outrage, so the best advice I can give is simply: don’t do this.

Truth Tests

The notions of comparison, equality, and truth values were introduced in [Chapter 9](#). Because the `if` statement is the first statement we’ve looked at that actually uses test results, we’ll expand on some of these ideas here. In particular, Python’s Boolean operators are a bit different from their counterparts in languages like C. In Python:

- Any nonzero number or nonempty object is true.
- Zero numbers, empty objects, and the special object `None` are considered false.
- Comparisons and equality tests are applied recursively to data structures.
- Comparisons and equality tests return `True` or `False` (custom versions of `1` and `0`).
- Boolean `and` and `or` operators return a true or false operand object.

In short, Boolean operators are used to combine the results of other tests. There are three Boolean expression operators in Python:

`X and Y`

Is true if both `X` and `Y` are true

`X or Y`

Is true if either `X` or `Y` is true

`not X`

Is true if `X` is false (the expression returns `True` or `False`)

Here, `X` and `Y` may be any truth value, or any expression that returns a truth value (e.g., an equality test, range comparison, and so on). Boolean operators are typed out as words in Python (instead of C’s `&&`, `||`, and `!`). Also, Boolean `and` and `or` operators return a true or false *object* in Python, not the values `True` or `False`. Let’s look at a few examples to see how this works:

```
>>> 2 < 3, 3 < 2          # Less-than: return True or False (1 or 0)
(True, False)
```

Magnitude comparisons such as these return `True` or `False` as their truth results, which, as we learned in [Chapters 5](#) and [9](#), are really just custom versions of the integers `1` and `0` (they print themselves differently but are otherwise the same).

On the other hand, the `and` and `or` operators always return an object—either the object on the left side of the operator or the object on the right. If we test their results in `if` or other statements, they will be as expected (remember, every object is inherently true or false), but we won’t get back a simple `True` or `False`.

For `or` tests, Python evaluates the operand objects from left to right and returns the first one that is true. Moreover, Python stops at the first true operand it finds. This is usually called *short-circuit evaluation*, as determining a result short-circuits (terminates) the rest of the expression:

```
>>> 2 or 3, 3 or 2      # Return left operand if true
(2, 3)                 # Else, return right operand (true or false)
>>> [] or 3
3
>>> [] or {}
{}

```

In the first line of the preceding example, both operands (2 and 3) are true (i.e., are nonzero), so Python always stops and returns the one on the left. In the other two tests, the left operand is false (an empty object), so Python simply evaluates and returns the object on the right (which may happen to have either a true or a false value when tested).

`and` operations also stop as soon as the result is known; however, in this case Python evaluates the operands from left to right and stops at the first *false* object:

```
>>> 2 and 3, 3 and 2    # Return left operand if false
(3, 2)                 # Else, return right operand (true or false)
>>> [] and {}
[]
>>> 3 and []
[]

```

Here, both operands are true in the first line, so Python evaluates both sides and returns the object on the right. In the second test, the left operand is false (`[]`), so Python stops and returns it as the test result. In the last test, the left side is true (3), so Python evaluates and returns the object on the right (which happens to be a false `[]`).

The end result of all this is the same as in C and most other languages—you get a value that is logically true or false if tested in an `if` or `while`. However, in Python Booleans return either the left or the right object, not a simple integer flag.

This behavior of `and` and `or` may seem esoteric at first glance, but see this chapter’s sidebar [“Why You Will Care: Booleans” on page 323](#) for examples of how it is sometimes used to advantage in coding by Python programmers. The next section also shows a common way to leverage this behavior, and its replacement in more recent versions of Python.

The if/else Ternary Expression

One common role for the prior section’s Boolean operators is to code an expression that runs the same as an `if` statement. Consider the following statement, which sets A to either Y or Z, based on the truth value of X:

```

if X:
    A = Y
else:
    A = Z

```

Sometimes, though, the items involved in such a statement are so simple that it seems like overkill to spread them across four lines. At other times, we may want to nest such a construct in a larger statement instead of assigning its result to a variable. For these reasons (and, frankly, because the C language has a similar tool[†]), Python 2.5 introduced a new expression format that allows us to say the same thing in one expression:

```
A = Y if X else Z
```

This expression has the exact same effect as the preceding four-line `if` statement, but it's simpler to code. As in the statement equivalent, Python runs expression `Y` only if `X` turns out to be true, and runs expression `Z` only if `X` turns out to be false. That is, it *short-circuits*, just like the Boolean operators described in the prior section. Here are some examples of it in action:

```

>>> A = 't' if 'spam' else 'f'      # Nonempty is true
>>> A
't'
>>> A = 't' if '' else 'f'
>>> A
'f'

```

Prior to Python 2.5 (and after 2.5, if you insist), the same effect can often be achieved by a careful combination of the `and` and `or` operators, because they return either the object on the left side or the object on the right:

```
A = ((X and Y) or Z)
```

This works, but there is a catch—you have to be able to assume that `Y` will be Boolean true. If that is the case, the effect is the same: the `and` runs first and returns `Y` if `X` is true; if it's not, the `or` simply returns `Z`. In other words, we get “if `X` then `Y` else `Z`.”

This `and/or` combination also seems to require a “moment of great clarity” to understand the first time you see it, and it's no longer required as of 2.5—use the equivalent and more robust and mnemonic `Y if X else Z` instead if you need this as an expression, or use a full `if` statement if the parts are nontrivial.

As a side note, using the following expression in Python is similar because the `bool` function will translate `X` into the equivalent of integer `1` or `0`, which can then be used to pick true and false values from a list:

```
A = [Z, Y][bool(X)]
```

[†] In fact, Python's `X if Y else Z` has a slightly different order than C's `Y ? X : Z`. This was reportedly done in response to analysis of common use patterns in Python code. According to rumor, this order was also chosen in part to discourage ex-C programmers from overusing it! Remember, simple is better than complex, in Python and elsewhere.

For example:

```
>>> ['f', 't'][bool('')]
'f'
>>> ['f', 't'][bool('spam')]
't'
```

However, this isn't exactly the same, because Python will not short-circuit—it will always run both `Z` and `Y`, regardless of the value of `X`. Because of such complexities, you're better off using the simpler and more easily understood `if/else` expression as of Python 2.5 and later. Again, though, you should use even that sparingly, and only if its parts are all fairly simple; otherwise, you're better off coding the full `if` statement form to make changes easier in the future. Your coworkers will be happy you did.

Still, you may see the `and/or` version in code written prior to 2.5 (and in code written by C programmers who haven't quite let go of their dark coding pasts...).

Why You Will Care: Booleans

One common way to use the somewhat unusual behavior of Python Boolean operators is to select from a set of objects with an `or`. A statement such as this:

```
X = A or B or C or None
```

sets `X` to the first nonempty (that is, true) object among `A`, `B`, and `C`, or to `None` if all of them are empty. This works because the `or` operator returns one of its two objects, and it turns out to be a fairly common coding paradigm in Python: to select a nonempty object from among a fixed-size set, simply string them together in an `or` expression. In simpler form, this is also commonly used to designate a default—the following sets `X` to `A` if `A` is true (or nonempty), and to `default` otherwise:

```
X = A or default
```

It's also important to understand short-circuit evaluation because expressions on the right of a Boolean operator might call functions that perform substantial or important work, or have side effects that won't happen if the short-circuit rule takes effect:

```
if f1() or f2(): ...
```

Here, if `f1` returns a true (or nonempty) value, Python will never run `f2`. To guarantee that both functions will be run, call them before the `or`:

```
tmp1, tmp2 = f1(), f2()
if tmp1 or tmp2: ...
```

You've already seen another application of this behavior in this chapter: because of the way Booleans work, the expression `((A and B) or C)` can be used to emulate an `if/else` statement—almost (see this chapter's discussion of this form for details).

We met additional Boolean use cases in prior chapters. As we saw in [Chapter 9](#), because all objects are inherently true or false, it's common and easier in Python to test an object directly (`if X:`) than to compare it to an empty value (`if X != ''`). For a string, the two tests are equivalent. As we also saw in [Chapter 5](#), the preset Boolean values `True` and `False` are the same as the integers `1` and `0` and are useful for initializing variables

(`X = False`), for loop tests (`while True:`), and for displaying results at the interactive prompt.

Also watch for the discussion of operator overloading in [Part VI](#): when we define new object types with classes, we can specify their Boolean nature with either the `__bool__` or `__len__` methods (`__bool__` is named `__nonzero__` in 2.6). The latter of these is tried if the former is absent and designates false by returning a length of zero—an empty object is considered false.

Chapter Summary

In this chapter, we studied the Python `if` statement. Additionally, because this was our first compound and logical statement, we reviewed Python’s general syntax rules and explored the operation of truth tests in more depth than we were able to previously. Along the way, we also looked at how to code multiway branching in Python and learned about the `if/else` expression introduced in Python 2.5.

The next chapter continues our look at procedural statements by expanding on the `while` and `for` loops. There, we’ll learn about alternative ways to code loops in Python, some of which may be better than others. Before that, though, here is the usual chapter quiz.

Test Your Knowledge: Quiz

1. How might you code a multiway branch in Python?
2. How can you code an `if/else` statement as an expression in Python?
3. How can you make a single statement span many lines?
4. What do the words `True` and `False` mean?

Test Your Knowledge: Answers

1. An `if` statement with multiple `elif` clauses is often the most straightforward way to code a multiway branch, though not necessarily the most concise. Dictionary indexing can often achieve the same result, especially if the dictionary contains callable functions coded with `def` statements or `lambda` expressions.
2. In Python 2.5 and later, the expression form `Y if X else Z` returns `Y` if `X` is true, or `Z` otherwise; it’s the same as a four-line `if` statement. The `and/or` combination `((X and Y) or Z)` can work the same way, but it’s more obscure and requires that the `Y` part be true.

3. Wrap up the statement in an open syntactic pair (`()`, `[]`, or `{}`), and it can span as many lines as you like; the statement ends when Python sees the closing (right) half of the pair, and lines 2 and beyond of the statement can begin at any indentation level.
4. `True` and `False` are just custom versions of the integers `1` and `0`, respectively: they always stand for Boolean `true` and `false` values in Python. They're available for use in truth tests and variable initialization and are printed for expression results at the interactive prompt.

while and for Loops

This chapter concludes our tour of Python procedural statements by presenting the language’s two main *looping* constructs—statements that repeat an action over and over. The first of these, the `while` statement, provides a way to code general loops. The second, the `for` statement, is designed for stepping through the items in a sequence object and running a block of code for each.

We’ve seen both of these informally already, but we’ll fill in additional usage details here. While we’re at it, we’ll also study a few less prominent statements used within loops, such as `break` and `continue`, and cover some built-ins commonly used with loops, such as `range`, `zip`, and `map`.

Although the `while` and `for` statements covered here are the primary syntax provided for coding repeated actions, there are additional looping operations and concepts in Python. Because of that, the iteration story is continued in the next chapter, where we’ll explore the related ideas of Python’s *iteration protocol* (used by the `for` loop) and *list comprehensions* (a close cousin to the `for` loop). Later chapters explore even more exotic iteration tools such as *generators*, `filter`, and `reduce`. For now, though, let’s keep things simple.

while Loops

Python’s `while` statement is the most general iteration construct in the language. In simple terms, it repeatedly executes a block of (normally indented) statements as long as a test at the top keeps evaluating to a true value. It is called a “loop” because control keeps looping back to the start of the statement until the test becomes false. When the test becomes false, control passes to the statement that follows the `while` block. The net effect is that the loop’s body is executed repeatedly while the test at the top is true; if the test is false to begin with, the body never runs.

General Format

In its most complex form, the `while` statement consists of a header line with a test expression, a body of one or more indented statements, and an optional `else` part that is executed if control exits the loop without a `break` statement being encountered. Python keeps evaluating the test at the top and executing the statements nested in the loop body until the test returns a false value:

```
while <test>:           # Loop test
    <statements1>       # Loop body
else:                   # Optional else
    <statements2>       # Run if didn't exit loop with break
```

Examples

To illustrate, let's look at a few simple `while` loops in action. The first, which consists of a `print` statement nested in a `while` loop, just prints a message forever. Recall that `True` is just a custom version of the integer 1 and always stands for a Boolean true value; because the test is always true, Python keeps executing the body forever, or until you stop its execution. This sort of behavior is usually called an *infinite loop*:

```
>>> while True:
...     print('Type Ctrl-C to stop me!')
```

The next example keeps slicing off the first character of a string until the string is empty and hence false. It's typical to test an object directly like this instead of using the more verbose equivalent (`while x != ''`). Later in this chapter, we'll see other ways to step more directly through the items in a string with a `for` loop.

```
>>> x = 'spam'
>>> while x:           # While x is not empty
...     print(x, end=' ')
...     x = x[1:]      # Strip first character off x
...
spam pam am m
```

Note the `end=' '` keyword argument used here to place all outputs on the same line separated by a space; see [Chapter 11](#) if you've forgotten why this works as it does. The following code counts from the value of `a` up to, but not including, `b`. We'll see an easier way to do this with a Python `for` loop and the built-in `range` function later:

```
>>> a=0; b=10
>>> while a < b:       # One way to code counter loops
...     print(a, end=' ')
...     a += 1        # Or, a = a + 1
...
0 1 2 3 4 5 6 7 8 9
```

Finally, notice that Python doesn't have what some languages call a "do until" loop statement. However, we can simulate one with a test and `break` at the bottom of the loop body:

```
while True:
    ...loop body...
    if exitTest(): break
```

To fully understand how this structure works, we need to move on to the next section and learn more about the **break** statement.

break, continue, pass, and the Loop else

Now that we've seen a few Python loops in action, it's time to take a look at two simple statements that have a purpose only when nested inside loops—the **break** and **continue** statements. While we're looking at oddballs, we will also study the loop **else** clause here, because it is intertwined with **break**, and Python's empty placeholder statement, the **pass** (which is not tied to loops per se, but falls into the general category of simple one-word statements). In Python:

break

Jumps out of the closest enclosing loop (past the entire loop statement)

continue

Jumps to the top of the closest enclosing loop (to the loop's header line)

pass

Does nothing at all: it's an empty statement placeholder

Loop else block

Runs if and only if the loop is exited normally (i.e., without hitting a **break**)

General Loop Format

Factoring in **break** and **continue** statements, the general format of the **while** loop looks like this:

```
while <test1>:
    <statements1>
    if <test2>: break           # Exit loop now, skip else
    if <test3>: continue       # Go to top of loop now, to test1
else:
    <statements2>             # Run if we didn't hit a 'break'
```

break and **continue** statements can appear anywhere inside the **while** (or **for**) loop's body, but they are usually coded further nested in an **if** test to take action in response to some condition.

Let's turn to a few simple examples to see how these statements come together in practice.

pass

Simple things first: the `pass` statement is a no-operation placeholder that is used when the syntax requires a statement, but you have nothing useful to say. It is often used to code an empty body for a compound statement. For instance, if you want to code an infinite loop that does nothing each time through, do it with a `pass`:

```
while True: pass                                # Type Ctrl-C to stop me!
```

Because the body is just an empty statement, Python gets stuck in this loop. `pass` is roughly to statements as `None` is to objects—an explicit nothing. Notice that here the `while` loop’s body is on the same line as the header, after the colon; as with `if` statements, this only works if the body isn’t a compound statement.

This example does nothing forever. It probably isn’t the most useful Python program ever written (unless you want to warm up your laptop computer on a cold winter’s day!); frankly, though, I couldn’t think of a better `pass` example at this point in the book.

We’ll see other places where `pass` makes more sense later—for instance, to ignore exceptions caught by `try` statements, and to define empty `class` objects with attributes that behave like “structs” and “records” in other languages. A `pass` is also sometime coded to mean “to be filled in later,” to stub out the bodies of functions temporarily:

```
def func1():
    pass                                # Add real code here later

def func2():
    pass
```

We can’t leave the body empty without getting a syntax error, so we say `pass` instead.



Version skew note: Python 3.0 (but not 2.6) allows *ellipses* coded as `...` (literally, three consecutive dots) to appear any place an expression can. Because ellipses do nothing by themselves, this can serve as an alternative to the `pass` statement, especially for code to be filled in later—a sort of Python “TBD”:

```
def func1():
    ...                                # Alternative to pass

def func2():
    ...

func1()                               # Does nothing if called
```

Ellipses can also appear on the same line as a statement header and may be used to initialize variable names if no specific type is required:

```
def func1(): ...                      # Works on same line too
def func2(): ...

>>> X = ...                          # Alternative to None
```



```
>>> X
Ellipsis
```

This notation is new in Python 3.0 (and goes well beyond the original intent of `...` in slicing extensions), so time will tell if it becomes widespread enough to challenge `pass` and `None` in these roles.

continue

The `continue` statement causes an immediate jump to the top of a loop. It also sometimes lets you avoid statement nesting. The next example uses `continue` to skip odd numbers. This code prints all even numbers less than 10 and greater than or equal to 0. Remember, 0 means false and `%` is the remainder of division operator, so this loop counts down to 0, skipping numbers that aren't multiples of 2 (it prints 8 6 4 2 0):

```
x = 10
while x:
    x = x-1                # Or, x -= 1
    if x % 2 != 0: continue # Odd? -- skip print
    print(x, end=' ')
```

Because `continue` jumps to the top of the loop, you don't need to nest the `print` statement inside an `if` test; the `print` is only reached if the `continue` is not run. If this sounds similar to a “goto” in other languages, it should. Python has no “goto” statement, but because `continue` lets you jump about in a program, many of the warnings about readability and maintainability you may have heard about goto apply. `continue` should probably be used sparingly, especially when you're first getting started with Python. For instance, the last example might be clearer if the `print` were nested under the `if`:

```
x = 10
while x:
    x = x-1
    if x % 2 == 0:          # Even? -- print
        print(x, end=' ')
```

break

The `break` statement causes an immediate exit from a loop. Because the code that follows it in the loop is not executed if the `break` is reached, you can also sometimes avoid nesting by including a `break`. For example, here is a simple interactive loop (a variant of a larger example we studied in [Chapter 10](#)) that inputs data with `input` (known as `raw_input` in Python 2.6) and exits when the user enters “stop” for the name request:

```
>>> while True:
...     name = input('Enter name:')
...     if name == 'stop': break
...     age = input('Enter age: ')
...     print('Hello', name, '=>', int(age) ** 2)
...
Enter name:mel
Enter age: 40
```

```
Hello mel => 1600
Enter name: bob
Enter age: 30
Hello bob => 900
Enter name: stop
```

Notice how this code converts the `age` input to an integer with `int` before raising it to the second power; as you'll recall, this is necessary because `input` returns user input as a string. In [Chapter 35](#), you'll see that `input` also raises an exception at end-of-file (e.g., if the user types Ctrl-Z or Ctrl-D); if this matters, wrap `input` in `try` statements.

Loop else

When combined with the loop `else` clause, the `break` statement can often eliminate the need for the search status flags used in other languages. For instance, the following piece of code determines whether a positive integer `y` is prime by searching for factors greater than 1:

```
x = y // 2                                # For some y > 1
while x > 1:
    if y % x == 0:                        # Remainder
        print(y, 'has factor', x)
        break                             # Skip else
    x -= 1
else:                                     # Normal exit
    print(y, 'is prime')
```

Rather than setting a flag to be tested when the loop is exited, it inserts a `break` where a factor is found. This way, the loop `else` clause can assume that it will be executed only if no factor is found; if you don't hit the `break`, the number is prime.

The loop `else` clause is also run if the body of the loop is never executed, as you don't run a `break` in that event either; in a `while` loop, this happens if the test in the header is false to begin with. Thus, in the preceding example you still get the “is prime” message if `x` is initially less than or equal to 1 (for instance, if `y` is 2).



This example determines primes, but only informally so. Numbers less than 2 are not considered prime by the strict mathematical definition. To be really picky, this code also fails for negative numbers and succeeds for floating-point numbers with no decimal digits. Also note that its code must use `//` instead of `/` in Python 3.0 because of the migration of `/` to “true division,” as described in [Chapter 5](#) (we need the initial division to truncate remainders, not retain them!). If you want to experiment with this code, be sure to see the exercise at the end of [Part IV](#), which wraps it in a function for reuse.

More on the loop else

Because the loop `else` clause is unique to Python, it tends to perplex some newcomers. In general terms, the loop `else` provides explicit syntax for a common coding scenario—it is a coding structure that lets us catch the “other” way out of a loop, without setting and checking flags or conditions.

Suppose, for instance, that we are writing a loop to search a list for a value, and we need to know whether the value was found after we exit the loop. We might code such a task this way:

```
found = False
while x and not found:
    if match(x[0]):
        print('Ni')
        found = True
    else:
        x = x[1:]
if not found:
    print('not found')
```

Value at front?

Slice off front and repeat

Here, we initialize, set, and later test a flag to determine whether the search succeeded or not. This is valid Python code, and it does work; however, this is exactly the sort of structure that the loop `else` clause is there to handle. Here’s an `else` equivalent:

```
while x:
    if match(x[0]):
        print('Ni')
        break
    x = x[1:]
else:
    print('Not found')
```

Exit when x empty

Exit, go around else

Only here if exhausted x

This version is more concise. The flag is gone, and we’ve replaced the `if` test at the loop end with an `else` (lined up vertically with the word `while`). Because the `break` inside the main part of the `while` exits the loop and goes around the `else`, this serves as a more structured way to catch the search-failure case.

Some readers might have noticed that the prior example’s `else` clause could be replaced with a test for an empty `x` after the loop (e.g., `if not x:`). Although that’s true in this example, the `else` provides explicit syntax for this coding pattern (it’s more obviously a search-failure clause here), and such an explicit empty test may not apply in some cases. The loop `else` becomes even more useful when used in conjunction with the `for` loop—the topic of the next section—because sequence iteration is not under your control.

Why You Will Care: Emulating C while Loops

The section on expression statements in [Chapter 11](#) stated that Python doesn't allow statements such as assignments to appear in places where it expects an expression. That means this common C language coding pattern won't work in Python:

```
while ((x = next()) != NULL) {...process x...}
```

C assignments return the value assigned, but Python assignments are just statements, not expressions. This eliminates a notorious class of C errors (you can't accidentally type `=` in Python when you mean `==`). If you need similar behavior, though, there are at least three ways to get the same effect in Python `while` loops without embedding assignments in loop tests. You can move the assignment into the loop body with a `break`:

```
while True:
    x = next()
    if not x: break
    ...process x...
```

or move the assignment into the loop with tests:

```
x = True
while x:
    x = next()
    if x:
        ...process x...
```

or move the first assignment outside the loop:

```
x = next()
while x:
    ...process x...
    x = next()
```

Of these three coding patterns, the first may be considered by some to be the least structured, but it also seems to be the simplest and is the most commonly used. A simple Python `for` loop may replace some C loops as well.

for Loops

The `for` loop is a generic sequence iterator in Python: it can step through the items in any ordered sequence object. The `for` statement works on strings, lists, tuples, other built-in iterables, and new objects that we'll see how to create later with classes. We met it in brief when studying sequence object types; let's expand on its usage more formally here.

General Format

The Python `for` loop begins with a header line that specifies an assignment target (or targets), along with the object you want to step through. The header is followed by a block of (normally indented) statements that you want to repeat:

```

for <target> in <object>:           # Assign object items to target
    <statements>                   # Repeated loop body: use target
else:
    <statements>                   # If we didn't hit a 'break'

```

When Python runs a **for** loop, it assigns the items in the sequence object to the target one by one and executes the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as though it were a cursor stepping through the sequence.

The name used as the assignment target in a **for** header line is usually a (possibly new) variable in the scope where the **for** statement is coded. There's not much special about it; it can even be changed inside the loop's body, but it will automatically be set to the next item in the sequence when control returns to the top of the loop again. After the loop this variable normally still refers to the last item visited, which is the last item in the sequence unless the loop exits with a **break** statement.

The **for** statement also supports an optional **else** block, which works exactly as it does in a **while** loop—it's executed if the loop exits without running into a **break** statement (i.e., if all items in the sequence have been visited). The **break** and **continue** statements introduced earlier also work the same in a **for** loop as they do in a **while**. The **for** loop's complete format can be described this way:

```

for <target> in <object>:           # Assign object items to target
    <statements>
    if <test>: break                # Exit loop now, skip else
    if <test>: continue            # Go to top of loop now
else:
    <statements>                   # If we didn't hit a 'break'

```

Examples

Let's type a few **for** loops interactively now, so you can see how they are used in practice.

Basic usage

As mentioned earlier, a **for** loop can step across any kind of sequence object. In our first example, for instance, we'll assign the name **x** to each of the three items in a list in turn, from left to right, and the **print** statement will be executed for each. Inside the **print** statement (the loop body), the name **x** refers to the current item in the list:

```

>>> for x in ["spam", "eggs", "ham"]:
...     print(x, end=' ')
...
spam eggs ham

```

The next two examples compute the sum and product of all the items in a list. Later in this chapter and later in the book we'll meet tools that apply operations such as **+** and ***** to items in a list automatically, but it's usually just as easy to use a **for**:

```

>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24

```

Other data types

Any sequence works in a `for`, as it's a generic tool. For example, `for` loops work on strings and tuples:

```

>>> S = "lumberjack"
>>> T = ("and", "I'm", "okay")

>>> for x in S: print(x, end=' ')    # Iterate over a string
...
l u m b e r j a c k

>>> for x in T: print(x, end=' ')    # Iterate over a tuple
...
and I'm okay

```

In fact, as we'll in the next chapter when we explore the notion of “iterables,” `for` loops can even work on some objects that are not sequences—files and dictionaries work, too!

Tuple assignment in for loops

If you're iterating through a sequence of tuples, the loop target itself can actually be a *tuple* of targets. This is just another case of the tuple-unpacking assignment we studied in [Chapter 11](#) at work. Remember, the `for` loop assigns items in the sequence object to the target, and assignment works the same everywhere:

```

>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:                # Tuple assignment at work
...     print(a, b)
...
1 2
3 4
5 6

```

Here, the first time through the loop is like writing `(a,b) = (1,2)`, the second time is like writing `(a,b) = (3,4)`, and so on. The net effect is to automatically unpack the current tuple on each iteration.

This form is commonly used in conjunction with the `zip` call we'll meet later in this chapter to implement parallel traversals. It also makes regular appearances in conjunction with SQL databases in Python, where query result tables are returned as sequences

of sequences like the list used here—the outer list is the database table, the nested tuples are the rows within the table, and tuple assignment extracts columns.

Tuples in `for` loops also come in handy to iterate through *both* keys and values in dictionaries using the `items` method, rather than looping through the keys and indexing to fetch the values manually:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
...     print(key, '=>', D[key])           # Use dict keys iterator and index
...
a => 1
c => 3
b => 2

>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]

>>> for (key, value) in D.items():
...     print(key, '=>', value)           # Iterate over both keys and values
...
a => 1
c => 3
b => 2
```

It's important to note that tuple assignment in `for` loops isn't a special case; any assignment target works syntactically after the word `for`. Although we can always assign manually within the loop to unpack:

```
>>> T
[(1, 2), (3, 4), (5, 6)]

>>> for both in T:
...     a, b = both
...     print(a, b)                       # Manual assignment equivalent
...
1 2
3 4
5 6
```

Tuples in the loop header save us an extra step when iterating through sequences of sequences. As suggested in [Chapter 11](#), even *nested* structures may be automatically unpacked this way in a `for`:

```
>>> ((a, b), c) = ((1, 2), 3)             # Nested sequences work too
>>> a, b, c
(1, 2, 3)

>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)
...
1 2 3
4 5 6
```

But this is no special case—the `for` loop simply runs the sort of assignment we ran just before it, on each iteration. Any nested sequence structure may be unpacked this way, just because *sequence assignment* is so generic:

```
>>> for ((a, b), c) in [[(1, 2), 3], ['XY', 6]]: print(a, b, c)
...
1 2 3
X Y 6
```

Python 3.0 extended sequence assignment in `for` loops

In fact, because the loop variable in a `for` loop can really be any assignment target, we can also use Python 3.0’s extended sequence-unpacking assignment syntax here to extract items and sections of sequences within sequences. Really, this isn’t a special case either, but simply a new assignment form in 3.0 (as discussed in [Chapter 11](#)); because it works in assignment statements, it automatically works in `for` loops.

Consider the tuple assignment form introduced in the prior section. A tuple of values is assigned to a tuple of names on each iteration, exactly like a simple assignment statement:

```
>>> a, b, c = (1, 2, 3)                                     # Tuple assignment
>>> a, b, c
(1, 2, 3)

>>> for (a, b, c) in [(1, 2, 3), (4, 5, 6)]:                 # Used in for loop
...     print(a, b, c)
...
1 2 3
4 5 6
```

In Python 3.0, because a sequence can be assigned to a more general set of names with a starred name to collect multiple items, we can use the same syntax to extract parts of nested sequences in the `for` loop:

```
>>> a, *b, c = (1, 2, 3, 4)                                 # Extended seq assignment
>>> a, b, c
(1, [2, 3], 4)

>>> for (a, *b, c) in [(1, 2, 3, 4), (5, 6, 7, 8)]:
...     print(a, b, c)
...
1 [2, 3] 4
5 [6, 7] 8
```

In practice, this approach might be used to pick out multiple columns from rows of data represented as nested sequences. In Python 2.X starred names aren’t allowed, but you can achieve similar effects by slicing. The only difference is that slicing returns a type-specific result, whereas starred names always are assigned lists:

```
>>> for all in [(1, 2, 3, 4), (5, 6, 7, 8)]:                 # Manual slicing in 2.6
...     a, b, c = all[0], all[1:3], all[3]
...     print(a, b, c)
```



```
...
1 (2, 3) 4
5 (6, 7) 8
```

See [Chapter 11](#) for more on this assignment form.

Nested for loops

Now let's look at a `for` loop that's a bit more sophisticated than those we've seen so far. The next example illustrates statement nesting and the loop `else` clause in a `for`. Given a list of objects (`items`) and a list of keys (`tests`), this code searches for each key in the objects list and reports on the search's outcome:

```
>>> items = ["aaa", 111, (4, 5), 2.01]  # A set of objects
>>> tests = [(4, 5), 3.14]              # Keys to search for
>>>
>>> for key in tests:                    # For all keys
...     for item in items:              # For all items
...         if item == key:            # Check for match
...             print(key, "was found")
...             break
...     else:
...         print(key, "not found!")
>>>
(4, 5) was found
3.14 not found!
```

Because the nested `if` runs a `break` when a match is found, the loop `else` clause can assume that if it is reached, the search has failed. Notice the nesting here. When this code runs, there are two loops going at the same time: the outer loop scans the keys list, and the inner loop scans the items list for each key. The nesting of the loop `else` clause is critical; it's indented to the same level as the header line of the inner `for` loop, so it's associated with the inner loop, not the `if` or the outer `for`.

Note that this example is easier to code if we employ the `in` operator to test membership. Because `in` implicitly scans an object looking for a match (at least logically), it replaces the inner loop:

```
>>> for key in tests:                  # For all keys
...     if key in items:               # Let Python check for a match
...         print(key, "was found")
...     else:
...         print(key, "not found!")
>>>
(4, 5) was found
3.14 not found!
```

In general, it's a good idea to let Python do as much of the work as possible (as in this solution) for the sake of brevity and performance.

The next example performs a typical data-structure task with a `for`—collecting common items in two sequences (strings). It's roughly a simple set intersection routine; after the loop runs, `res` refers to a list that contains all the items found in `seq1` and `seq2`:

```

>>> seq1 = "spam"
>>> seq2 = "scam"
>>>
>>> res = []                                # Start empty
>>> for x in seq1:                          # Scan first sequence
...     if x in seq2:                      # Common item?
...         res.append(x)                 # Add to result end
...
>>> res
['s', 'a', 'm']

```

Unfortunately, this code is equipped to work only on two specific variables: `seq1` and `seq2`. It would be nice if this loop could somehow be generalized into a tool you could use more than once. As you'll see, that simple idea leads us to *functions*, the topic of the next part of the book.

Why You Will Care: File Scanners

In general, loops come in handy anywhere you need to repeat an operation or process something more than once. Because files contain multiple characters and lines, they are one of the more typical use cases for loops. To load a file's contents into a string all at once, you simply call the file object's `read` method:

```

file = open('test.txt', 'r')  # Read contents into a string
print(file.read())

```

But to load a file in smaller pieces, it's common to code either a `while` loop with breaks on end-of-file, or a `for` loop. To read by characters, either of the following codings will suffice:

```

file = open('test.txt')
while True:
    char = file.read(1)        # Read by character
    if not char: break
    print(char)

for char in open('test.txt').read():
    print(char)

```

The `for` loop here also processes each character, but it loads the file into memory all at once (and assumes it fits!). To read by lines or blocks instead, you can use `while` loop code like this:

```

file = open('test.txt')
while True:
    line = file.readline()    # Read line by line
    if not line: break
    print(line, end='')       # Line already has a \n

file = open('test.txt', 'rb')
while True:
    chunk = file.read(10)     # Read byte chunks: up to 10 bytes
    if not chunk: break
    print(chunk)

```

You typically read binary data in blocks. To read text files line by line, though, the `for` loop tends to be easiest to code and the quickest to run:

```
for line in open('test.txt').readlines():
    print(line, end='')

for line in open('test.txt'):    # Use iterators: best text input mode
    print(line, end='')

```

The file `readlines` method loads a file all at once into a line-string list, and the last example here relies on file *iterators* to automatically read one line on each loop iteration (iterators are covered in detail in [Chapter 14](#)). See the library manual for more on the calls used here. The last example here is generally the best option for text files—besides its simplicity, it works for arbitrarily large files and doesn't load the entire file into memory all at once. The iterator version may be the quickest, but I/O performance is less clear-cut in Python 3.0.

In some 2.X Python code, you may also see the name `open` replaced with `file` and the file object's older `xreadlines` method used to achieve the same effect as the file's automatic line iterator (it's like `readlines` but doesn't load the file into memory all at once). Both `file` and `xreadlines` are removed in Python 3.0, because they are redundant; you shouldn't use them in 2.6 either, but they may pop up in older code and resources. Watch for more on reading files in [Chapter 36](#); as we'll see there, text and binary files have slightly different semantics in 3.0.

Loop Coding Techniques

The `for` loop subsumes most counter-style loops. It's generally simpler to code and quicker to run than a `while`, so it's the first tool you should reach for whenever you need to step through a sequence. But there are also situations where you will need to iterate in more specialized ways. For example, what if you need to visit every second or third item in a list, or change the list along the way? How about traversing more than one sequence in parallel, in the same `for` loop?

You can always code such unique iterations with a `while` loop and manual indexing, but Python provides two built-ins that allow you to specialize the iteration in a `for`:

- The built-in `range` function produces a series of successively higher integers, which can be used as indexes in a `for`.
- The built-in `zip` function returns a series of parallel-item tuples, which can be used to traverse multiple sequences in a `for`.

Because `for` loops typically run quicker than `while`-based counter loops, it's to your advantage to use tools like these that allow you to use `for` when possible. Let's look at each of these built-ins in turn.

Counter Loops: while and range

The `range` function is really a general tool that can be used in a variety of contexts. Although it's used most often to generate indexes in a `for`, you can use it anywhere you need a list of integers. In Python 3.0, `range` is an *iterator* that generates items on demand, so we need to wrap it in a `list` call to display its results all at once (more on iterators in [Chapter 14](#)):

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

With one argument, `range` generates a list of integers from zero up to but not including the argument's value. If you pass in two arguments, the first is taken as the lower bound. An optional third argument can give a *step*; if it is used, Python adds the step to each successive integer in the result (the step defaults to 1). Ranges can also be nonpositive and nonascending, if you want them to be:

```
>>> list(range(-5, 5))
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(range(5, -5, -1))
[5, 4, 3, 2, 1, 0, -1, -2, -3, -4]
```

Although such `range` results may be useful all by themselves, they tend to come in most handy within `for` loops. For one thing, they provide a simple way to repeat an action a specific number of times. To print three lines, for example, use a `range` to generate the appropriate number of integers; `for` loops force results from `range` automatically in 3.0, so we don't need `list` here:

```
>>> for i in range(3):
...     print(i, 'Pythons')
...
0 Pythons
1 Pythons
2 Pythons
```

`range` is also commonly used to iterate over a sequence indirectly. The easiest and fastest way to step through a sequence exhaustively is always with a simple `for`, as Python handles most of the details for you:

```
>>> X = 'spam'
>>> for item in X: print(item, end=' ')           # Simple iteration
...
s p a m
```

Internally, the `for` loop handles the details of the iteration automatically when used this way. If you really need to take over the indexing logic explicitly, you can do it with a `while` loop:

```
>>> i = 0
>>> while i < len(X):
...     print(X[i], end=' ')
...     i += 1                                     # while loop iteration
```

```
...
s p a m
```

You can also do manual indexing with a `for`, though, if you use `range` to generate a list of indexes to iterate through. It's a multistep process, but it's sufficient to generate offsets, rather than the items at those offsets:

```
>>> X
'spam'
>>> len(X)                                # Length of string
4
>>> list(range(len(X)))                    # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print(X[i], end=' ') # Manual for indexing
...
s p a m
```

Note that because this example is stepping over a list of *offsets* into `X`, not the actual *items* of `X`, we need to index back into `X` within the loop to fetch each item.

Nonexhaustive Traversals: range and Slices

The last example in the prior section works, but it's not the fastest option. It's also more work than we need to do. Unless you have a special indexing requirement, you're always better off using the simple `for` loop form in Python—as a general rule, use `for` instead of `while` whenever possible, and don't use `range` calls in `for` loops except as a last resort. This simpler solution is better:

```
>>> for item in X: print(item)              # Simple iteration
...
```

However, the coding pattern used in the prior example does allow us to do more specialized sorts of traversals. For instance, we can skip items as we go:

```
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]

>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k
```

Here, we visit every *second* item in the string `S` by stepping over the generated `range` list. To visit every third item, change the third `range` argument to be 3, and so on. In effect, using `range` this way lets you skip items in loops while still retaining the simplicity of the `for` loop construct.

Still, this is probably not the ideal best-practice technique in Python today. If you really want to skip items in a sequence, the extended three-limit form of the *slice expression*, presented in [Chapter 7](#), provides a simpler route to the same goal. To visit every second character in `S`, for example, slice with a stride of 2:

```
>>> S = 'abcdefghijk'
>>> for c in S[::2]: print(c, end=' ')
...
a c e g i k
```

The result is the same, but substantially easier for you to write and for others to read. The only real advantage to using `range` here instead is that it does not copy the string and does not create a list in 3.0; for very large strings, it may save memory.

Changing Lists: range

Another common place where you may use the `range` and `for` combination is in loops that change a list as it is being traversed. Suppose, for example, that you need to add 1 to every item in a list. You can try this with a simple `for` loop, but the result probably won't be exactly what you want:

```
>>> L = [1, 2, 3, 4, 5]

>>> for x in L:
...     x += 1
...
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

This doesn't quite work—it changes the loop variable `x`, not the list `L`. The reason is somewhat subtle. Each time through the loop, `x` refers to the next integer already pulled out of the list. In the first iteration, for example, `x` is integer 1. In the next iteration, the loop body sets `x` to a different object, integer 2, but it does not update the list where 1 originally came from.

To really change the list as we march across it, we need to use indexes so we can assign an updated value to each position as we go. The `range/len` combination can produce the required indexes for us:

```
>>> L = [1, 2, 3, 4, 5]

>>> for i in range(len(L)):          # Add one to each item in L
...     L[i] += 1                    # Or L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

When coded this way, the list is changed as we proceed through the loop. There is no way to do the same with a simple `for x in L:`-style loop, because such a loop iterates through actual items, not list positions. But what about the equivalent `while` loop? Such a loop requires a bit more work on our part, and likely runs more slowly:

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
```

```

...     i += 1
...
>>> L
[3, 4, 5, 6, 7]

```

Here again, though, the `range` solution may not be ideal either. A list comprehension expression of the form:

```
[x+1 for x in L]
```

would do similar work, albeit without changing the original list in-place (we could assign the expression's new list object result back to `L`, but this would not update any other references to the original list). Because this is such a central looping concept, we'll save a complete exploration of list comprehensions for the next chapter.

Parallel Traversals: `zip` and `map`

As we've seen, the `range` built-in allows us to traverse sequences with `for` in a nonexhaustive fashion. In the same spirit, the built-in `zip` function allows us to use `for` loops to visit multiple sequences *in parallel*. In basic operation, `zip` takes one or more sequences as arguments and returns a series of tuples that pair up parallel items taken from those sequences. For example, suppose we're working with two lists:

```

>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]

```

To combine the items in these lists, we can use `zip` to create a list of tuple pairs (like `range`, `zip` is an iterable object in 3.0, so we must wrap it in a `list` call to display all its results at once—more on iterators in the next chapter):

```

>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2))
[(1, 5), (2, 6), (3, 7), (4, 8)]

```

list() required in 3.0, not 2.6

Such a result may be useful in other contexts as well, but when wedded with the `for` loop, it supports parallel iterations:

```

>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12

```

Here, we step over the result of the `zip` call—that is, the pairs of items pulled from the two lists. Notice that this `for` loop again uses the tuple assignment form we met earlier to unpack each tuple in the `zip` result. The first time through, it's as though we ran the assignment statement `(x, y) = (1, 5)`.

The net effect is that we scan both `L1` *and* `L2` in our loop. We could achieve a similar effect with a `while` loop that handles indexing manually, but it would require more typing and would likely run more slowly than the `for/zip` approach.

Strictly speaking, the `zip` function is more general than this example suggests. For instance, it accepts any type of sequence (really, any iterable object, including files), and it accepts more than two arguments. With three arguments, as in the following example, it builds a list of three-item tuples with items from each sequence, essentially projecting by columns (technically, we get an N-ary tuple for N arguments):

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3))
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Moreover, `zip` truncates result tuples at the length of the shortest sequence when the argument lengths differ. In the following, we `zip` together two strings to pick out characters in parallel, but the result has only as many tuples as the length of the shortest sequence:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2))
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

map equivalence in Python 2.6

In Python 2.X, the related built-in `map` function pairs items from sequences in a similar fashion, but it pads shorter sequences with `None` if the argument lengths differ instead of truncating to the shortest length:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'

>>> map(None, S1, S2)
[('a', 'x'), ('b', 'y'), ('c', 'z'), (None, '1'), (None, '2'), (None, '3')]
# 2.X only
```

This example is using a degenerate form of the `map` built-in, which is no longer supported in 3.0. Normally, `map` takes a function and one or more sequence arguments and collects the results of calling the function with parallel items taken from the sequence(s). We'll study `map` in detail in Chapters 19 and 20, but as a brief example, the following maps the built-in `ord` function across each item in a string and collects the results (like `zip`, `map` is a value generator in 3.0 and so must be passed to `list` to collect all its results at once):

```
>>> list(map(ord, 'spam'))
[115, 112, 97, 109]
```


This works the same as the following loop statement, but is often quicker:

```
>>> res = []
>>> for c in 'spam': res.append(ord(c))
>>> res
[115, 112, 97, 109]
```



Version skew note: The degenerate form of `map` using a function argument of `None` is no longer supported in Python 3.0, because it largely overlaps with `zip` (and was, frankly, a bit at odds with `map`'s function-application purpose). In 3.0, either use `zip` or write loop code to pad results yourself. We'll see how to do this in [Chapter 20](#), after we've had a chance to study some additional iteration concepts.

Dictionary construction with `zip`

In [Chapter 8](#), I suggested that the `zip` call used here can also be handy for generating dictionaries when the sets of keys and values must be computed at runtime. Now that we're becoming proficient with `zip`, I'll explain how it relates to dictionary construction. As you've learned, you can always create a dictionary by coding a dictionary literal, or by assigning to keys over time:

```
>>> D1 = {'spam':1, 'eggs':3, 'toast':5}
>>> D1
{'toast': 5, 'eggs': 3, 'spam': 1}

>>> D1 = {}
>>> D1['spam'] = 1
>>> D1['eggs'] = 3
>>> D1['toast'] = 5
```

What to do, though, if your program obtains dictionary keys and values in *lists* at runtime, after you've coded your script? For example, say you had the following keys and values lists:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]
```

One solution for turning those lists into a dictionary would be to `zip` the lists and step through them in parallel with a `for` loop:

```
>>> list(zip(keys, vals))
[('spam', 1), ('eggs', 3), ('toast', 5)]

>>> D2 = {}
>>> for (k, v) in zip(keys, vals): D2[k] = v
...
>>> D2
{'toast': 5, 'eggs': 3, 'spam': 1}
```

It turns out, though, that in Python 2.2 and later you can skip the `for` loop altogether and simply pass the zipped keys/values lists to the built-in `dict` constructor call:

```
>>> keys = ['spam', 'eggs', 'toast']
>>> vals = [1, 3, 5]

>>> D3 = dict(zip(keys, vals))
>>> D3
{'toast': 5, 'eggs': 3, 'spam': 1}
```

The built-in name `dict` is really a type name in Python (you'll learn more about type names, and subclassing them, in [Chapter 31](#)). Calling it achieves something like a list-to-dictionary conversion, but it's really an object construction request. In the next chapter we'll explore a related but richer concept, the *list comprehension*, which builds lists in a single expression; we'll also revisit 3.0 *dictionary comprehensions* an alternative to the `dict` call for zipped key/value pairs.

Generating Both Offsets and Items: `enumerate`

Earlier, we discussed using `range` to generate the offsets of items in a string, rather than the items at those offsets. In some programs, though, we need both: the item to use, plus an offset as we go. Traditionally, this was coded with a simple `for` loop that also kept a counter of the current offset:

```
>>> S = 'spam'
>>> offset = 0
>>> for item in S:
...     print(item, 'appears at offset', offset)
...     offset += 1
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

This works, but in recent Python releases a new built-in named `enumerate` does the job for us:

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
...     print(item, 'appears at offset', offset)
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

The `enumerate` function returns a *generator object*—a kind of object that supports the iteration protocol that we will study in the next chapter and will discuss in more detail in the next part of the book. In short, it has a `__next__` method called by the `next` built-in function, which returns an *(index, value)* tuple each time through the loop. We can unpack these tuples with tuple assignment in the `for` loop (much like using `zip`):

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x02765AA8>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

As usual, we don't normally see this machinery because iteration contexts—including list comprehensions, the subject of [Chapter 14](#)—run the iteration protocol automatically:

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']
```

To fully understand iteration concepts like `enumerate`, `zip`, and list comprehensions, we need to move on to the next chapter for a more formal dissection.

Chapter Summary

In this chapter, we explored Python's looping statements as well as some concepts related to looping in Python. We looked at the `while` and `for` loop statements in depth, and we learned about their associated `else` clauses. We also studied the `break` and `continue` statements, which have meaning only inside loops, and met several built-in tools commonly used in `for` loops, including `range`, `zip`, `map`, and `enumerate` (although their roles as iterators in Python 3.0 won't be fully uncovered until the next chapter).

In the next chapter, we continue the iteration story by discussing list comprehensions and the iteration protocol in Python—concepts strongly related to `for` loops. There, we'll also explain some of the subtleties of iterable tools we met here, such as `range` and `zip`. As always, though, before moving on let's exercise what you've picked up here with a quiz.

Test Your Knowledge: Quiz

1. What are the main functional differences between a `while` and a `for`?
2. What's the difference between `break` and `continue`?
3. When is a loop's `else` clause executed?
4. How can you code a counter-based loop in Python?
5. What can a `range` be used for in a `for` loop?