

About Python

This chapter covers

- Why use Python?
- What Python does well
- What Python doesn't do as well
- Why learn Python 3?

Read this chapter if you want to know how Python compares to other languages and its place in the grand scheme of things. Skip this chapter if you want to start learning Python right away. The information in this chapter is a valid part of this book—but it's certainly not necessary for programming with Python.

1.1 Why should I use Python?

Hundreds of programming languages are available today, from mature languages like C and C++, to newer entries like Ruby, C#, and Lua, to enterprise juggernauts like Java. Choosing a language to learn is difficult. Although no one language is the right choice for every possible situation, I think that Python is a good choice for a large number of programming problems, and it's also a good choice if you're learning to program. Hundreds of thousands of programmers around the world use Python, and the number grows every year.

Python continues to attract new users for a variety of reasons. It's a true cross-platform language, running equally well on Windows, Linux/UNIX, and Macintosh platforms, as well as others, ranging from supercomputers to cell phones. It can be used to develop small applications and rapid prototypes, but it scales well to permit development of large programs. It comes with a powerful and easy-to-use graphical user interface (GUI) toolkit, web programming libraries, and more. *And it's free.*

1.2 **What Python does well**

Python is a modern programming language developed by Guido van Rossum in the 1990s (and named after a famous comedic troupe). Although Python isn't perfect for every application, its strengths make it a good choice for many situations.

1.2.1 **Python is easy to use**

Programmers familiar with traditional languages will find it easy to learn Python. All of the familiar constructs such as loops, conditional statements, arrays, and so forth are included, but many are easier to use in Python. Here are a few of the reasons why:

- *Types are associated with objects, not variables.* A variable can be assigned a value of any type, and a list can contain objects of many different types. This also means that type casting usually isn't necessary, and your code isn't locked into the straitjacket of predeclared types.
- *Python typically operates at a much higher level of abstraction.* This is partly the result of the way the language is built and partly the result of an extensive standard code library that comes with the Python distribution. A program to download a web page can be written in two or three lines!
- *Syntax rules are very simple.* Although becoming an expert Pythonista takes time and effort, even beginners can absorb enough Python syntax to write useful code quickly.

Python is well suited for rapid application development. It isn't unusual for coding an application in Python to take one-fifth the time it would if coded in C or Java and to take as little as one-fifth the number of lines of the equivalent C program. This depends on the particular application, of course; for a numerical algorithm performing mostly integer arithmetic in `for` loops, there would be much less of a productivity gain. For the average application, the productivity gain can be significant.

1.2.2 **Python is expressive**

Python is a very expressive language. *Expressive* in this context means that a single line of Python code can do more than a single line of code in most other languages. The advantages of a more expressive language are obvious: the fewer lines of code you have to write, the faster you can complete the project. Not only that, but the fewer lines of code there are, the easier the program will be to maintain and debug.

To get an idea of how Python’s expressiveness can simplify code, let’s consider swapping the values of two variables, `var1` and `var2`. In a language like Java, this requires three lines of code and an extra variable:

```
int temp = var1;
var1 = var2;
var2 = temp;
```

The variable `temp` is needed to save the value of `var1` when `var2` is put into it, and then that saved value is put into `var2`. The process isn’t terribly complex, but reading those three lines and understanding that a swap has taken place takes a certain amount of overhead, even for experienced coders.

In contrast, Python lets you make the same swap in one line and in a way that makes it obvious that a swap of values has occurred:

```
var2, var1 = var1, var2
```

Of course, this is a very simple example, but you find the same advantages throughout the language.

1.2.3 Python is readable

Another advantage of Python is that it’s easy to read. You might think that a programming language needs to be read only by a computer, but humans have to read your code as well—whoever debugs your code (quite possibly you), whoever maintains your code (could be you again), and whoever might want to modify your code in the future. In all of those situations, the easier the code is to read and understand, the better it is.

The easier code is to understand, the easier it is to debug, maintain, and modify. Python’s main advantage in this department is its use of indentation. Unlike most languages, Python *insists* that blocks of code be indented. Although this strikes some as odd, it has the benefit that your code is always formatted in a very easy-to-read style.

Following are two short programs, one written in Perl and one in Python. Both take two equal-sized lists of numbers and return the pairwise sum of those lists. I think the Python code is more readable than the Perl code; it’s visually cleaner and contains fewer inscrutable symbols:

```
# Perl version.
sub pairwise_sum {
    my($arg1, $arg2) = @_;
    my(@result) = ();
    @list1 = @$arg1;
    @list2 = @$arg2;
    for($i=0; $i < length(@list1); $i++) {
        push(@result, $list1[$i] + $list2[$i]);
    }
    return(\@result);
}
```

```
# Python version.
def pairwise_sum(list1, list2):
    result = []
    for i in range(len(list1)):
        result.append(list1[i] + list2[i])
    return result
```

Both pieces of code do the same thing, but the Python code wins in terms of readability.

1.2.4 Python is complete—“batteries included”

Another advantage of Python is its “batteries included” philosophy when it comes to libraries. The idea is that when you install Python, you should have everything you need to do real work, without the need to install additional libraries. This is why the Python standard library comes with modules for handling email, web pages, databases, operating system calls, GUI development, and more.

For example, with Python, you can write a web server to share the files in a directory with just two lines of code:

```
import http.server
http.server.test(HandlerClass=http.server.SimpleHTTPRequestHandler)
```

There’s no need to install libraries to handle network connections and HTTP—it’s already in Python, right out of the box.

1.2.5 Python is cross-platform

Python is also an excellent cross-platform language. Python runs on many different platforms: Windows, Mac, Linux, UNIX, and so on. Because it’s interpreted, the same code can run on any platform that has a Python interpreter, and almost all current platforms have one. There are even versions of Python that run on Java (Jython) and .NET (IronPython), giving you even more possible platforms that run Python.

1.2.6 Python is free

Python is also free. Python was originally, and continues to be, developed under the open source model, and it’s freely available. You can download and install practically any version of Python and use it to develop software for commercial or personal applications, and you don’t need to pay a dime.

Although attitudes are changing, some people are still leery of free software because of concerns about a lack of support, fearing they lack the clout of a paying customer. But Python is used by many established companies as a key part of their business; Google, Rackspace, Industrial Light & Magic, and Honeywell are just a few examples. These companies and many others know Python for what it is—a very stable, reliable, and well-supported product with an active and knowledgeable user community. You’ll get an answer to even the most difficult Python question more quickly on the Python internet newsgroup than you will on most tech-support phone lines, and the Python answer will be free and correct.

Python and open source software

Not only is Python free of cost, but its source code is also freely available, and you're free to modify, improve, and extend it if you want. Because the source code is freely available, you have the ability to go in yourself and change it (or to hire someone to go in and do so for you). You rarely have this option at any reasonable cost with proprietary software.

If this is your first foray into the world of open source software, you should understand that not only are you free to use and modify Python, but you're also able (and encouraged) to contribute to it and improve it. Depending on your circumstances, interests, and skills, those contributions might be financial, as in a donation to the Python Software Foundation (PSF), or they may involve participating in one of the special interest groups (SIGs), testing and giving feedback on releases of the Python core or one of the auxiliary modules, or contributing some of what you or your company develops back to the community. The level of contribution (if any) is, of course, up to you; but if you're able to give back, definitely consider doing so. Something of significant value is being created here, and you have an opportunity to add to it.

Python has a lot going for it: expressiveness, readability, rich included libraries, and cross-platform capabilities, plus it's open source. What's the catch?

1.3 What Python doesn't do as well

Although Python has many advantages, no language can do everything, so Python isn't the perfect solution for all your needs. To decide whether Python is the right language for your situation, you also need to consider the areas where Python doesn't do as well.

1.3.1 Python is not the fastest language

A possible drawback with Python is its speed of execution. It isn't a fully compiled language. Instead, it's first semicompiled to an internal byte-code form, which is then executed by a Python interpreter. There are some tasks, such as string parsing using regular expressions, for which Python has efficient implementations and is as fast as, or faster than, any C program you're likely to write. Nevertheless, most of the time, using Python results in slower programs than a language like C. But you should keep this in perspective. Modern computers have so much computing power that for the vast majority of applications, the speed of the program isn't as important as the speed of development, and Python programs can typically be written much more quickly. In addition, it's easy to extend Python with modules written in C or C++, which can be used to run the CPU-intensive portions of a program.

1.3.2 *Python doesn't have the most libraries*

Although Python comes with an excellent collection of libraries, and many more are available, Python doesn't hold the lead in this department. Languages like C, Java, and Perl have even larger collections of libraries available, in some cases offering a solution where Python has none or a choice of several options where Python might have only one. These situations tend to be fairly specialized, however, and Python is easy to extend, either in Python itself or by using existing libraries in C and other languages. For almost all common computing problems, Python's library support is excellent.

1.3.3 *Python doesn't check variable types at compile time*

Unlike some languages, Python's variables are more like labels that reference various objects: integers, strings, class instances, whatever. That means that although those objects themselves have types, the variables referring to them aren't bound to that particular type. It's possible (if not necessarily desirable) to use the variable `x` to refer to a string in one line and an integer in another:

```
>>> x = "2"
>>> print(x)
'2'
```

← **x is string "2"**

```
>>> x = int(x)
>>> print(x)
2
```

← **x is now integer 2**

The fact that Python associates types with objects and not with variables means that the interpreter doesn't help you catch variable type mismatches. If you intend a variable `count` to hold an integer, Python won't complain if you assign the string "two" to it. Traditional coders count this as a disadvantage, because you lose an additional free check on your code. But errors like this usually aren't hard to find and fix, and Python's testing features makes avoiding type errors manageable. Most Python programmers feel that the flexibility of dynamic typing more than outweighs the cost.

1.4 *Why learn Python 3?*

Python has been around for a number of years and has evolved over that time. The first edition of this book was based on Python 1.5.2, and Python 2.x has been the dominant version for several years. This book is based on Python 3.1.

Python 3, originally whimsically dubbed Python 3000, is notable because it's the first version of Python in the history of the language to break backward compatibility. What this means is that code written for earlier versions of Python probably won't run on Python 3 without some changes. In earlier versions of Python, for example, the `print` statement didn't require parentheses around its arguments:

```
print "hello"
```

In Python 3, `print` is a function and needs the parentheses:

```
print("hello")
```

You may be thinking, “Why change details like this, if it’s going to break old code?” Because this kind of change is a big step for any language, the core developers of Python thought about this issue carefully. Although the changes in Python 3 break compatibility with older code, those changes are fairly small and for the better—they make the language more consistent, more readable, and less ambiguous. Python 3 isn’t a dramatic rewrite of the language; it’s a well-thought-out evolution. The core developers also took care to provide a strategy and tools to safely and efficiently migrate old code to Python 3, which will be discussed in a later chapter.

Why learn Python 3? Because it’s the best Python so far; and as projects switch to take advantage of its improvements, it will be the dominant Python version for years to come. If you need a library that hasn’t been converted yet, by all means stick with Python 2.x; but if you’re starting to learn Python or starting a project, then go with Python 3—not only is it better, but it’s the future.

1.5 Summary

Python is a modern, high-level language, with many features:

- Dynamic typing
- Simple, consistent syntax and semantics
- Multiplatform
- Well-planned design and evolution of features
- Highly modular
- Suited for both rapid development and large-scale programming
- Reasonably fast and easily extended with C or C++ modules for higher speeds
- Easy access to various GUI toolkits
- Built-in advanced features such as persistent object storage, advanced hash tables, expandable class syntax, universal comparison functions, and so forth
- Powerful included libraries such as numeric processing, image manipulation, user interfaces, web scripting, and others
- Supported by a dynamic Python community
- Can be integrated with a number of other languages to let you take advantage of the strengths of both while obviating their weaknesses

Let’s get going. The first step is to make sure you have Python 3 installed on your machine. In the next chapter, we’ll look at how to get Python up and running on Windows, Mac, and Linux platforms.

Getting started



This chapter covers

- Installing Python
- Using IDLE and the basic interactive mode
- Writing a simple program
- Using IDLE's Python shell window

This chapter guides you through downloading, installing, and starting up Python and IDLE, an integrated development environment for Python. At the time of this writing, the Python language is fairly mature, and version 3.1 has just been released. After going through years of refinement, Python 3 is the first version of the language that isn't fully backward compatible with earlier versions. It should be several years before another such dramatic change occurs, and any future enhancements will be developed with concern to avoid impacting an already significant existing code base. Therefore, the material presented after this chapter isn't likely to become dated anytime soon.

2.1 Installing Python

Installing Python is a simple matter, regardless of which platform you're using. The first step is to obtain a recent distribution for your machine; the most recent one can always be found at www.python.org. This book is based on Python 3.1.

Having more than one version of Python

You may already have an earlier version of Python installed on your machine. Many Linux distributions and Mac OS X come with Python 2.x as part of the operating system. Because Python 3 isn't completely compatible with Python 2, it's reasonable to wonder if installing both versions on the same computer will cause a conflict.

There's no need to worry; you can have multiple versions of Python on the same computer. In the case of UNIX-based systems like OS X and Linux, Python 3 installs alongside the older version and doesn't replace it. When your system looks for "python," it still finds the one it expects; and when you want to access Python 3, you can run `python3.1` or `idle3.1`. In Windows, the different versions are installed in separate locations and have separate menu entries.

Some basic platform-specific descriptions for the Python installation are given next. The specifics can vary quite a bit depending on your platform, so be sure to read the instructions on the download pages and with the various versions. You're probably familiar with installing software on your particular machine, so I'll keep these descriptions short:

- *Microsoft Windows*—Python can be installed in most versions of Windows by using the Python installer program, currently called `python-3.1.msi`. Just download it, execute it, and follow the installer's prompts. You may need to be logged in as administrator to run the install. If you're on a network and don't have the administrator password, ask your system administrator to do the installation for you.
- *Macintosh*—You need to get a version of Python 3 that matches your OS X version and your processor. After you determine the correct version, download the disk image file, double-click to mount it, and run the installer inside. The OS X installer sets up everything automatically, and Python 3 will be in a subfolder inside the Applications folder, labeled with the version number. Mac OS X ships with various versions of Python as part of the system, but you don't need to worry about that—Python 3 will be installed *in addition to* the system version. You can find more information about using Python on OS X by following the links on the Python home page.
- *Linux/UNIX*—Most Linux distributions come with Python installed. But the versions of Python vary, and the version of Python installed may not be version 3; for this book, you need to be sure you have the Python 3 packages installed. It's also possible that IDLE isn't installed by default, and you'll need to install that package separately. Although it's also possible to build Python 3 from the source code available on the www.python.org website, a number of additional libraries are needed, and the process isn't for novices. If a precompiled version of Python exists for your distribution of Linux, I recommend using that. Use

the software management system for your distribution to locate and install the correct packages for Python 3 and IDLE. Versions are also available for running Python under many other operating systems. See www.python.org for a current list of supported platforms and specifics on installation.

2.2 *IDLE and the basic interactive mode*

You have two built-in options for obtaining interactive access to the Python interpreter: the original basic (command-line) mode and IDLE. IDLE is available on many platforms, including Windows, Mac, and Linux, but it may not be available on others. You may need to do more work and install additional software packages to get IDLE running, but it will be worth it because it's a large step up from the basic interactive mode. On the other hand, even if you normally use IDLE, at times you'll likely want to fire up the basic mode. You should be familiar enough to start and use either one.

2.2.1 *The basic interactive mode*

The basic interactive mode is a rather primitive environment. But the interactive examples in this book are generally small; and later in this book, you'll learn how to easily bring code you've placed in a file into your session (using the module mechanism). Let's look at how to start a basic session on Windows, Mac OS X, and UNIX:

- *Starting a basic session on Windows*—For version 3.x of Python, you navigate to the Python (command-line) entry on the Python 3.x submenu of the Programs folder on the Start menu, and click it. Alternatively, you can directly find the Python.exe executable (for example, in C:\Python31) and double-click it. Doing so brings up the window shown in figure 2.1.
- *Starting a basic session on Mac OS X*—Open a terminal window, and type `python3`. If you get a “Command not found” error, run the `Update Shell Profile.command` script found in the Python3 subfolder in the Applications folder.
- *Starting a basic session on UNIX*—Type `python3.1` at a command prompt. A version message similar to the one shown in figure 2.1 followed by the Python prompt `>>>` appears in the current window.

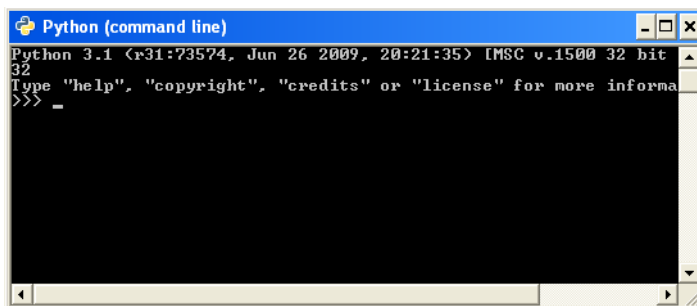


Figure 2.1 Basic interactive mode on Windows XP

Exiting the interactive shell

To exit from a basic session, press Ctrl-Z (if you're on Windows) or Ctrl-D (if you're on Linux or UNIX) or type `exit()` at a command prompt.

Most platforms have a command-line-editing and command-history mechanism. You can use the up and down arrows, as well as the Home, End, Page Up, and Page Down keys, to scroll through past entries and repeat them by pressing the Enter key. (See “Basic Python interactive mode summary” at the end of the appendix.) This is all you need to work your way through this book as you're learning Python. Another option is to use the excellent Python mode available for Emacs, which, among other things, provides access to the interactive mode of Python through an integrated shell buffer.

2.2.2 The IDLE integrated development environment

IDLE is the built-in development environment for Python. Its name is based on the acronym for *integrated development environment* (of course, it may have been influenced by the last name of a certain cast member of a particular British television show). IDLE combines an interactive interpreter with code editing and debugging tools to give you one-stop shopping as far as creating Python code is concerned. IDLE's various tools make it an attractive place to start as you learn Python. Let's look at how you run IDLE on Windows, Mac OS X, and Linux:

- *Starting IDLE on Windows*—For version 3.1 of Python, you navigate to the IDLE (Python GUI) entry of the Python 3.1 submenu of the Programs folder of your Start menu, and click it. Doing so brings up the window shown in figure 2.2.
- *Starting IDLE on Mac OS X*—Navigate to the Python 3.x subfolder in the Applications folder, and run IDLE from there.
- *Starting IDLE on Linux or UNIX*—Type `idle3.1` at a command prompt. This brings up a window similar to the one shown in figure 2.2. If you installed IDLE through your distribution's package manager, there should also be a menu entry for IDLE under the Programming submenu or something similar.

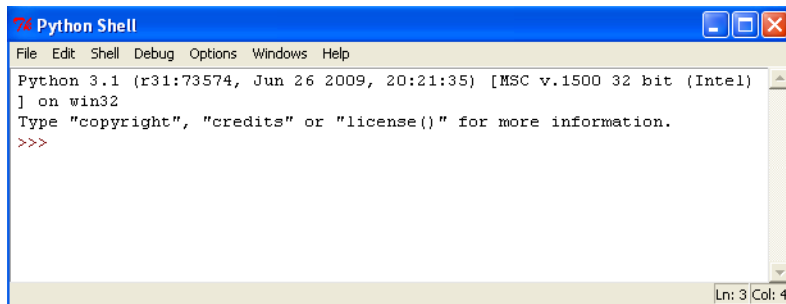


Figure 2.2 IDLE on Windows

2.2.3 Choosing between basic interactive mode and IDLE

Which should you use, IDLE or the basic shell window? To begin, use either IDLE or the Python Shell window. Both have all you need to work through the code examples in this book until you reach chapter 10. From there, we'll cover writing your own modules, and IDLE will be a convenient way to create and edit files. But if you have a strong preference for another editor, you may find that a basic shell window and your favorite editor serve you just as well. If you don't have any strong editor preferences, I suggest using IDLE from the beginning.

2.3 Using IDLE's Python Shell window

The Python Shell window (figure 2.3) opens when you fire up IDLE. It provides automatic indentation and colors your code as you type it in, based on Python syntax types.

You can move around the buffer using the mouse, the arrow keys, the Page Up and Page Down keys, and/or a number of the standard Emacs key bindings. Check the Help menu for the details.

Everything in your session is buffered. You can scroll or search up, place the cursor on any line, and press Enter (creating a hard return), and that line will be copied to the bottom of the screen, where you can edit it and then send it to the interpreter by pressing the Enter key again. Or, leaving the cursor at the bottom, you can toggle up and down through the previously entered commands using Alt-P and Alt-N. This will successively bring copies of the lines to the bottom. When you have the one you want, you can again edit it and then send it to the interpreter by pressing the Enter key. You can complete Python keywords or user-defined values by pressing Alt-/.

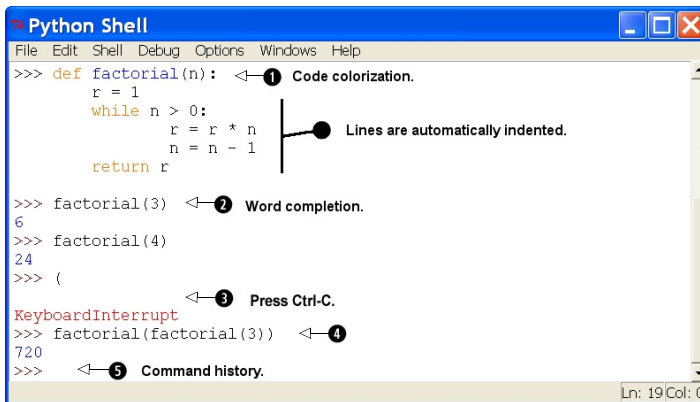


Figure 2.3 Using the Python shell in IDLE. ❶ Code is automatically colored (based on Python syntax) as it's typed in. ❷ Here I typed `factorial` and then pressed Alt-/, and automatic completion finished the word `factorial`. ❸ I lost the prompt, so I pressed Ctrl-C to interrupt the interpreter and get the prompt back (a closed bracket would have worked here as well). ❹ Placing the cursor on any previous command and pressing the Enter key moves the command and the cursor to the bottom, where you can edit the command and then press Enter to send it to the interpreter. ❺ Placing the cursor at the bottom, you can toggle up and down through the history of previous commands using Alt-P and Alt-N. When you have the command you want, edit it as desired and press Enter, and it will be sent to the interpreter.

If you ever find yourself in a situation where you seem to be hung and can't get a new prompt, the interpreter is likely in a state where it's waiting for you to enter something specific. Pressing Ctrl-C sends an interrupt and should get you back to a prompt. It can also be used to interrupt any running command. To exit IDLE, choose Exit from the File menu.

The Edit menu is the one you'll likely be using the most to begin with. Like any of the other menus, you can tear it off by double-clicking the dotted line at its top and leaving it up beside your window.

2.4 Hello, world

Regardless of how you're accessing Python's interactive mode, you should see a prompt consisting of three angle braces: `>>>`. This is the Python command prompt, and it indicates that you can type in a command to be executed or an expression to be evaluated. Start with the obligatory "Hello, World" program, which is a one-liner in Python. (End each line you type with a hard return.)

```
>>> print("Hello, World")
Hello, World
```

Here I entered the `print` command at the command prompt, and the result appeared on the screen.

Executing the `print` function causes its argument to be printed to the standard output, usually the screen. If the command had been executed while Python was running a Python program from a file, exactly the same thing would have happened: "Hello, World" would have been printed to the screen.

Congratulations! You've just written your first Python program, and we haven't even started talking about the language.

2.5 Using the interactive prompt to explore Python

Whether you're in IDLE or at a standard interactive prompt, there are a couple of handy tools to help you explore Python. The first is the `help()` function, which has two modes. You can just enter `help()` at the prompt to enter the help system, where you can get help on modules, keywords, or topics. When you're in the help system, you see a `help>` prompt, and you can enter a module name, such as `math` or some other topic, to browse Python's documentation on that topic.

Usually it's more convenient to use `help()` in a more targeted way. Entering a type or variable name as a parameter for `help()` gives you an immediate display of that type's documentation:

```
>>> x = 2
>>> help(x)
Help on int object:

class int(object)
|   int(x[, base]) -> integer
|
```

```
| Convert a string or number to an integer, if possible. A floating
| point argument will be truncated towards zero (this does not include a
| string representation of a floating point number!) When converting a
| string, use the optional base. It is an error to supply a base when
| converting a non-string.
|
| Methods defined here:
... (continues with a list of methods for an int)
```

Using `help()` in this way is handy for checking the exact syntax of a method or the behavior of an object.

The `help()` function is part of the `pydoc` library, which has several options for accessing the documentation built into Python libraries. Because every Python installation comes with complete documentation, you can have all of the official documentation at your fingertips, even if you aren't online. See the appendix for more information on accessing Python's documentation.

The other useful function is `dir()`, which lists the objects in a particular namespace. Used with no parameters, it lists the current globals, but it can also list objects for a module or even a type:

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'x']
>>> dir(int)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
 '__delattr__', '__divmod__', '__doc__', '__eq__', '__float__',
 '__floor__', '__floordiv__', '__format__', '__ge__', '__getattr__',
 '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
 '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__',
 '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
 '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
 '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
 '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
 '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
 '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
 '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator',
 'imag', 'numerator', 'real']
>>>
```

`dir()` is useful for finding out what methods and data are defined, for reminding yourself at a glance of all the members that belong an object or module, and for debugging, because you can see what is defined where.

You can use two other functions to see local and global variables, respectively. They are `globals` and `locals`:

```
>>> globals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__',
 '__doc__': None, '__package__': None}
```

Unlike `dir`, both `globals` and `locals` show the values associated with the objects. You'll find out more about both of these functions in chapter 10; for now, it's enough

that you're aware that you have several options for examining what's going on within a Python session.

2.6 **Summary**

Installing Python 3 is the first step. On Windows systems it's as simple as downloading the latest installer from python.org and running it. On Linux, UNIX, and Mac systems the steps will vary. Check for instructions on the www.python.org website, and use your system's software package installer where possible.

After you've installed Python, you can use either the basic interactive shell (and later, your favorite editor) or the IDLE integrated development environment. Whichever you decide to use, it's time to move to chapter 3, where we'll make a quick survey of Python the language.

The Quick Python overview

This chapter covers

- Surveying Python
- Using built-in data types
- Controlling program flow
- Creating modules
- Using object-oriented programming

The purpose of this chapter is to give you a basic feeling for the syntax, semantics, capabilities, and philosophy of the Python language. It has been designed to provide you with an initial perspective or conceptual framework on which you'll be able to add details as you encounter them in the rest of the book.

On an initial read, you needn't be concerned about working through and understanding the details of the code segments. You'll be doing fine if you pick up a bit of an idea about what is being done. The subsequent chapters of this book will walk you through the specifics of these features and won't assume prior knowledge. You can always return to this chapter and work through the examples in the appropriate sections as a review after you've read the later chapters.

3.1 Python synopsis

Python has a number of built-in data types such as integers, floats, complex numbers, strings, lists, tuples, dictionaries, and file objects. These can be manipulated using language operators, built-in functions, library functions, or a data type's own methods.

Programmers can also define their own classes and instantiate their own class instances.¹ These can be manipulated by programmer-defined methods as well as the language operators and built-in functions for which the programmer has defined the appropriate special method attributes.

Python provides conditional and iterative control flow through an `if-elif-else` construct along with `while` and `for` loops. It allows function definition with flexible argument-passing options. Exceptions (errors) can be raised using the `raise` statement and caught and handled using the `try-except-else` construct.

Variables don't have to be declared and can have any built-in data type, user-defined object, function, or module assigned to them.

3.2 Built-in data types

Python has several built-in data types, from scalars like numbers and Booleans, to more complex structures like lists, dictionaries, and files.

3.2.1 Numbers

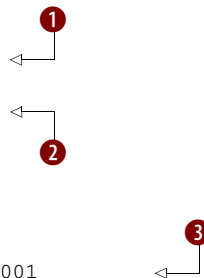
Python's four number types are integers, floats, complex numbers, and Booleans:

- *Integers*—1, -3, 42, 355, 8888888888888888, -7777777777
- *Floats*—3.0, 31e12, -6e-4
- *Complex numbers*—3 + 2j, -4 - 2j, 4.2 + 6.3j
- *Booleans*—True, False

You can manipulate them using the arithmetic operators: + (addition), - (subtraction), * (multiplication), / (division), ** (exponentiation), and % (modulus).

The following examples use integers:

```
>>> x = 5 + 2 - 3 * 2
>>> x
1
>>> 5 / 2
2.5
>>> 5 // 2
2
>>> 5 % 2
1
>>> 2 ** 8
256
>>> 1000000001 ** 3
1000000003000000003000000001
```



¹ The Python documentation and this book use the term *object* to refer to instances of any Python data type, not just what many other languages would call *class instances*. This is because all Python objects are instances of one class or another.

Division of integers with `/` ❶ results in a float (new in Python 3.x), and division of integers with `//` ❷ results in truncation. Note that integers are of unlimited size ❸; they will grow as large as you need them to.

These examples work with floats, which are based on the doubles in C:

```
>>> x = 4.3 ** 2.4
>>> x
33.137847377716483
>>> 3.5e30 * 2.77e45
9.6950000000000002e+75
>>> 1000000001.0 ** 3
1.000000003e+27
```

Next, the following examples use complex numbers:

```
>>> (3+2j) ** (2+3j)
(0.68176651908903363-2.1207457766159625j)
>>> x = (3+2j) * (4+9j)
>>> x
(-6+35j)
>>> x.real
-6.0
>>> x.imag
35.0
```

❶

Complex numbers consist of both a real element and an imaginary element, suffixed with a `j`. In the preceding code, variable `x` is assigned to a complex number ❶. You can obtain its “real” part using the attribute notation `x.real`.

Several built-in functions can operate on numbers. There are also the library module `cmath` (which contains functions for complex numbers) and the library module `math` (which contains functions for the other three types):

```
>>> round(3.49)
3
>>> import math
>>> math.ceil(3.49)
4
```

❶ Built-in function

❷ Library module function

Built-in functions are always available and are called using a standard function calling syntax. In the preceding code, `round` is called with a float as its input argument ❶.

The functions in library modules are made available using the `import` statement. At ❷, the `math` library module is imported, and its `ceil` function is called using attribute notation: `module.function(arguments)`.

The following examples use Booleans:

```
>>> x = False
>>> x
False
>>> not x
True
>>> y = True * 2
>>> y
2
```


❶

Other than their representation as `True` and `False`, Booleans behave like the numbers 1 (True) and 0 (False) ❶.

3.2.2 Lists

Python has a powerful built-in list type:

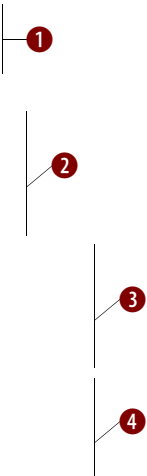
```
[ ]
[1]
[1, 2, 3, 4, 5, 6, 7, 8, 12]
[1, "two", 3L, 4.0, ["a", "b"], (5,6)]
```



A list can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number ❶.

A list can be indexed from its front or back. You can also refer to a subsegment, or *slice*, of a list using slice notation:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'
>>> x[2]
'third'
>>> x[-1]
'fourth'
>>> x[-2]
'third'
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']
>>> x[-2:-1]
['third']
>>> x[:3]
['first', 'second', 'third']
>>> x[-2:]
['third', 'fourth']
```



Index from the front ❶ using positive indices (starting with 0 as the first element). Index from the back ❷ using negative indices (starting with -1 as the last element). Obtain a slice using `[m:n]` ❸, where `m` is the inclusive starting point and `n` is the exclusive ending point (see table 3.1). An `[:n]` slice ❹ starts at its beginning, and an `[m:]` slice goes to a list's end.

Table 3.1 List indices

x=	["first" ,	"second" ,	"third" ,	"fourth"]
Positive indices		0	1	2	3	
Negative indices		-4	-3	-2	-1	

You can use this notation to add, remove, and replace elements in a list or to obtain an element or a new list that is a slice from it:

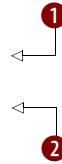
```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[1] = "two"
>>> x[8:9] = []
>>> x
[1, 'two', 3, 4, 5, 6, 7, 8]
>>> x[5:7] = [6.0, 6.5, 7.0]
>>> x
[1, 'two', 3, 4, 5, 6.0, 6.5, 7.0, 8]
>>> x[5:]
[6.0, 6.5, 7.0, 8]
```



The size of the list increases or decreases if the new slice is bigger or smaller than the slice it's replacing ❶.

Some built-in functions (`len`, `max`, and `min`), some operators (`in`, `+`, and `*`), the `del` statement, and the list methods (`append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, and `sort`) will operate on lists:

```
>>> x = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(x)
9
>>> [-1, 0] + x
[-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.reverse()
>>> x
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```



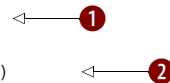
The operators `+` and `*` each create a new list, leaving the original unchanged ❶. A list's methods are called using attribute notation on the list itself: `x.method(arguments)` ❷.

A number of these operations repeat functionality that can be performed with slice notation, but they improve code readability.

3.2.3 Tuples

Tuples are similar to lists but are *immutable*—that is, they can't be modified after they have been created. The operators (`in`, `+`, and `*`) and built-in functions (`len`, `max`, and `min`), operate on them the same way as they do on lists, because none of them modify the original. Index and slice notation work the same way for obtaining elements or slices but can't be used to add, remove, or replace elements. There are also only two tuple methods: `count` and `index`. A major purpose of tuples is for use as keys for dictionaries. They're also more efficient to use when you don't need modifiability.

```
()
(1,)
(1, 2, 3, 4, 5, 6, 7, 8, 12)
(1, "two", 3L, 4.0, ["a", "b"], (5, 6))
```



A one-element tuple ❶ needs a comma. A tuple, like a list, can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number ❷.

A list can be converted to a tuple using the built-in function `tuple`:

```
>>> x = [1, 2, 3, 4]
>>> tuple(x)
(1, 2, 3, 4)
```

Conversely, a tuple can be converted to a list using the built-in function `list`:

```
>>> x = (1, 2, 3, 4)
>>> list(x)
[1, 2, 3, 4]
```

3.2.4 Strings

String processing is one of Python's strengths. There are many options for delimiting strings:

```
"A string in double quotes can contain 'single quote' characters."
'A string in single quotes can contain "double quote" characters.'
''\tThis string starts with a tab and ends with a newline character.\n''
"""This is a triple double quoted string, the only kind that can
    contain real newlines."""
```

Strings can be delimited by single (`' '`), double (`" "`), triple single (`''' '''`), or triple double (`""" """`) quotations and can contain tab (`\t`) and newline (`\n`) characters.

Strings are also immutable. The operators and functions that work with them return new strings derived from the original. The operators (`in`, `+`, and `*`) and built-in functions (`len`, `max`, and `min`) operate on strings as they do on lists and tuples. Index and slice notation works the same for obtaining elements or slices but can't be used to add, remove, or replace elements.

Strings have several methods to work with their contents, and the `re` library module also contains functions for working with strings:

```
>>> x = "live and      let \t    \tlive"
>>> x.split()
['live', 'and', 'let', 'live']
>>> x.replace("      let \t    \tlive", "enjoy life")
'live and enjoy life'
>>> import re
>>> regexpr = re.compile(r"[\t ]+")
>>> regexpr.sub(" ", x)
'live and let live'
```



The `re` module ❶ provides regular expression functionality. It provides more sophisticated pattern extraction and replacement capability than the `string` module.

The `print` function outputs strings. Other Python data types can be easily converted to strings and formatted:

```
>>> e = 2.718
>>> x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
>>> print("The constant e is:", e, "and the list x is:", x)
```



```

The constant e is: 2.718 and the list x is: [1, 'two', 3, 4.0,
['a', 'b'], (5, 6)]
>>> print("the value of %s is: %.2f" % ("e", e))
the value of e is: 2.72

```



Objects are automatically converted to string representations for printing ❶. The % operator ❷ provides a formatting capability similar to that of C's `sprintf`.

3.2.5 Dictionaries

Python's built-in dictionary data type provides associative array functionality implemented using hash tables. The built-in `len` function returns the number of key-value pairs in a dictionary. The `del` statement can be used to delete a key-value pair. As is the case for lists, a number of dictionary methods (`clear`, `copy`, `get`, `has_key`, `items`, `keys`, `update`, and `values`) are available.

```

>>> x = {1: "one", 2: "two"}
>>> x["first"] = "one"
>>> x[("Delorme", "Ryan", 1995)] = (1, 2, 3)
>>> list(x.keys())
['first', 2, 1, ('Delorme', 'Ryan', 1995)]
>>> x[1]
'one'
>>> x.get(1, "not available")
'one'
>>> x.get(4, "not available")
'not available'

```



Keys must be of an immutable type ❶. This includes numbers, strings, and tuples. Values can be any kind of object, including mutable types such as lists and dictionaries. The dictionary method `get` ❷ optionally returns a user-definable value when a key isn't in a dictionary.

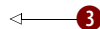
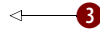
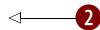
3.2.6 Sets

A set in Python is an unordered collection of objects, used in situations where membership and uniqueness in the set are the main things you need to know about that object. You can think of sets as a collection of dictionary keys without any associated values:

```

>>> x = set([1, 2, 3, 1, 3, 5])
>>> x
{1, 2, 3, 5}
>>> 1 in x
True
>>> 4 in x
False
>>>

```



You can create a set by using `set` on a sequence, like a list ❶. When a sequence is made into a set, duplicates are removed ❷. The `in` keyword ❸ is used to check for membership of an object in a set.

3.2.7 File objects

A file is accessed through a Python file object:

```
>>> f = open("myfile", "w")
>>> f.write("First line with necessary newline character\n")
44
>>> f.write("Second line to write to the file\n")
33
>>> f.close()
>>> f = open("myfile", "r")
>>> line1 = f.readline()
>>> line2 = f.readline()
>>> f.close()
>>> print(line1, line2)
First line with necessary newline character
Second line to write to the file
>>> import os
>>> print(os.getcwd())
c:\My Documents\test
>>> os.chdir(os.path.join("c:\\", "My Documents", "images"))
>>> filename = os.path.join("c:\\", "My Documents", "test", "myfile")
>>> print(filename)
c:\My Documents\test\myfile
>>> f = open(filename, "r")
>>> print(f.readline())
First line with necessary newline character
>>> f.close()
```

The `open` statement ❶ creates a file object. Here the file `myfile` in the current working directory is being opened in write ("`w`") mode. After writing two lines to it and closing it ❷, we open the same file again, this time in the read ("`r`") mode. The `os` module ❸ provides a number of functions for moving around the file system and working with the pathnames of files and directories. Here, we move to another directory ❹. But by referring to the file by an absolute pathname ❺, we are still able to access it.

A number of other input/output capabilities are available. You can use the built-in `input` function to prompt and obtain a string from the user. The `sys` library module allows access to `stdin`, `stdout`, and `stderr`. The `struct` library module provides support for reading and writing files that were generated by or are to be used by C programs. The `Pickle` library module delivers data persistence through the ability to easily read and write the Python data types to and from files.

3.3 Control flow structures

Python has a full range of structures to control code execution and program flow, including common branching and looping structures.

3.3.1 Boolean values and expressions

Python has several ways of expressing Boolean values; the Boolean constant `False`, `0`, the Python nil value `None`, and empty values (for example, the empty list `[]` or empty

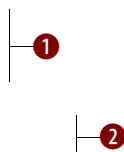
string `"")` are all taken as `False`. The Boolean constant `True` and everything else are considered `True`.

You can create comparison expressions using the comparison operators (`<`, `<=`, `==`, `>`, `>=`, `!=`, `is`, `is not`, `in`, `not in`) and the logical operators (`and`, `not`, `or`), which all return `True` or `False`.

3.3.2 The if-elif-else statement

The block of code after the first `true` condition (of an `if` or an `elif`) is executed. If none of the conditions is `true`, the block of code after the `else` is executed:

```
x = 5
if x < 5:
    y = -1
    z = 5
elif x > 5:
    y = 1
    z = 11
else:
    y = 0
    z = 10
print(x, y, z)
```



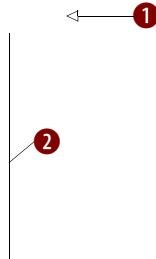
The `elif` and `else` clauses are optional ❶, and there can be any number of `elif` clauses. Python uses indentation to delimit blocks ❷. No explicit delimiters such as brackets or braces are necessary. Each block consists of one or more statements separated by newlines. These statements must all be at the same level of indentation.

The output here would be `5 0 10`.

3.3.3 The while loop

The `while` loop is executed as long as the condition (which here is `x > y`) is `true`:

```
u, v, x, y = 0, 0, 100, 30
while x > y:
    u = u + y
    x = x - y
    if x < y + 2:
        v = v + x
        x = 0
    else:
        v = v + y + 2
        x = x - y - 2
print(u, v)
```



This is a shorthand notation. Here, `u` and `v` are assigned a value of 0, `x` is set to 100, and `y` obtains a value of 30 ❶. This is the loop block ❷. It's possible for it to contain `break` (which ends the loop) and `continue` statements (which abort the current iteration of the loop).

The output here would be `60 40`.

3.3.4 The for loop

The `for` loop is simple but powerful because it's possible to iterate over any iterable type, such as a list or tuple. Unlike in many languages, Python's `for` loop iterates over each of the items in a sequence, making it more of a `foreach` loop. The following loop finds the first occurrence of an integer that is divisible by 7:

```
item_list = [3, "string1", 23, 14.0, "string2", 49, 64, 70]
for x in item_list:
    if not isinstance(x, int):
        continue
    if not x % 7:
        print("found an integer divisible by seven: %d" % x)
        break
```

`x` is sequentially assigned each value in the list ❶. If `x` isn't an integer, then the rest of this iteration is aborted by the `continue` statement ❷. Flow control continues with `x` set to the next item from the list. After the first appropriate integer is found, the loop is ended by the `break` statement ❸.

The output here would be

```
found an integer divisible by seven: 49
```

3.3.5 Function definition

Python provides flexible mechanisms for passing arguments to functions:

```
>>> def funct1(x, y, z):
...     value = x + 2*y + z**2
...     if value > 0:
...         return x + 2*y + z**2
...     else:
...         return 0
...
>>> u, v = 3, 4
>>> funct1(u, v, 2)
15
>>> funct1(u, z=v, y=2)
23
>>> def funct2(x, y=1, z=1):
...     return x + 2 * y + z ** 2
...
>>> funct2(3, z=4)
21
>>> def funct3(x, y=1, z=1, *tup):
...     print((x, y, z) + tup)
...
>>> funct3(2)
(2, 1, 1)
>>> funct3(1, 2, 3, 4, 5, 6, 7, 8, 9)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> def funct4(x, y=1, z=1, **dictionary):
...     print(x, y, z, dict)
>>> funct4(1, 2, m=5, n=9, z=3)
1 2 3 {'n': 9, 'm': 5}
```

Functions are defined using the `def` statement ❶. The `return` statement ❷ is what a function uses to return a value. This value can be of any type. If no `return` statement is encountered, Python's `None` value is returned. Function arguments can be entered either by position or by name (keyword). Here `z` and `y` are entered by name ❸. Function parameters can be defined with defaults that are used if a function call leaves them out ❹. A special parameter can be defined that will collect all extra positional arguments in a function call into a tuple ❺. Likewise, a special parameter can be defined that will collect all extra keyword arguments in a function call into a dictionary ❻.

3.3.6 Exceptions

Exceptions (errors) can be caught and handled using the `try-except-finally-else` compound statement. This statement can also catch and handle exceptions you define and raise yourself. Any exception that isn't caught will cause the program to exit. Listing 3.1 shows basic exception handling.

Listing 3.1 File exception.py

```
class EmptyFileError(Exception):
    pass
filenames = ["myfile1", "nonExistent", "emptyFile", "myfile2"]
for file in filenames:
    try:
        f = open(file, 'r')
        line = f.readline()
        if line == "":
            f.close()
            raise EmptyFileError("%s: is empty" % file)
    except IOError as error:
        print("%s: could not be opened: %s" % (file, error.strerror))
    except EmptyFileError as error:
        print(error)
    else:
        print("%s: %s" % (file, f.readline()))
    finally:
        print("Done processing", file)
```

Here we define our own exception type inheriting from the base `Exception` type ❶. If an `IOError` or `EmptyFileError` occurs during the execution of the statements in the `try` block, the associated `except` block is executed ❷. This is where an `IOError` might be raised ❸. Here we raise the `EmptyFileError` ❹. The `else` clause is optional ❺. It's executed if no exception occurs in the `try` block (note that in this example, `continue` statements in the `except` blocks could have been used instead). The `finally` clause is optional ❻. It's executed at the end of the block whether an exception was raised or not.

3.4 Module creation

It's easy to create your own modules, which can be imported and used in the same way as Python's built-in library modules. The example in listing 3.2 is a simple module with one function that prompts the user to enter a filename and determines the number of times words occur in this file.

Listing 3.2 File `wo.py`

```

"""wo module. Contains function: words_occur()"""
# interface functions
def words_occur():
    """words_occur() - count the occurrences of words in a file."""
    # Prompt user for the name of the file to use.
    file_name = input("Enter the name of the file: ")
    # Open the file, read it and store its words in a list.
    f = open(file_name, 'r')
    word_list = f.read().split()
    f.close()
    # Count the number of occurrences of each word in the file.
    occurs_dict = {}
    for word in word_list:
        # increment the occurrences count for this word
        occurs_dict[word] = occurs_dict.get(word, 0) + 1
    # Print out the results.
    print("File %s has %d words (%d are unique)" \
          % (file_name, len(word_list), len(occurs_dict)))
    print(occurs_dict)
if __name__ == '__main__':
    words_occur()

```

Documentation strings are a standard way of documenting modules, functions, methods, and classes ❶. Comments are anything beginning with a `#` character ❷. `read` returns a string containing all the characters in a file ❸, and `split` returns a list of the words of a string “split out” based on whitespace. You can use a `\` to break a long statement across multiple lines ❹. This allows the program to also be run as a script by typing `python wo.py` at a command line ❺.

If you place a file in one of the directories on the module search path, which can be found in `sys.path`, then it can be imported like any of the built-in library modules using the `import` statement:

```

>>> import wo
>>> wo.words_occur()

```

This function is called ❶ using the same attribute syntax as used for library module functions.

Note that if you change the file `wo.py` on disk, `import` won't bring your changes in to the same interactive session. You use the `reload` function from the `imp` library in this situation:

```
>>> import imp
>>> imp.reload(wo)
<module 'wo'>
```

For larger projects, there is a generalization of the module concept called *packages*. This allows you to easily group a number of modules together in a directory or directory subtree and import and hierarchically refer to them using a `package.subpackage.module` syntax. This entails little more than the creation of a possibly empty initialization file for each package or subpackage.

3.5 Object-oriented programming

Python provides full support for OOP. Listing 3.3 is an example that might be the start of a simple shapes module for a drawing program. It's intended mainly to serve as reference if you're already familiar with object-oriented programming. The callout notes relate Python's syntax and semantics to the standard features found in other languages.

Listing 3.3 File `sh.py`

```
"""sh module. Contains classes Shape, Square and Circle"""
class Shape:
    """Shape class: has method move"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, deltaX, deltaY):
        self.x = self.x + deltaX
        self.y = self.y + deltaY
class Square(Shape):
    """Square Class: inherits from Shape"""
    def __init__(self, side=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.side = side
class Circle(Shape):
    """Circle Class: inherits from Shape and has method area"""
    pi = 3.14159
    def __init__(self, r=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.radius = r
    def area(self):
        """Circle area method: returns the area of the circle."""
        return self.radius * self.radius * self.pi
    def __str__(self):
        return "Circle of radius %s at coordinates (%d, %d)" \
            % (self.radius, self.x, self.y)
```

Classes are defined using the `class` keyword ❶. The instance initializer method (constructor) for a class is always called `__init__` ❷. Instance variables `x` and `y` are created and initialized here ❸. Methods, like functions, are defined using the `def` keyword ❹. The first argument of any method is by convention called `self`. When the method is invoked, `self` is set to the instance that invoked the method. Class `Circle` inherits from class `Shape` ❺. This is similar to but not exactly like a standard class variable ❻. A class must, in its initializer, explicitly call the initializer of its base class ❼. The `__str__` method is used by the `print` function ❽. Other special attribute methods permit operator overloading or are employed by built-in methods such as the length (`len`) function.

Importing this file makes these classes available:

```
>>> import sh
>>> c1 = sh.Circle()
>>> c2 = sh.Circle(5, 15, 20)
>>> print(c1)
Circle of radius 1 at coordinates (0, 0)
>>> print(c2)
Circle of radius 5 at coordinates (15, 20)
>>> c2.area()
78.539749999999998
>>> c2.move(5,6)
>>> print(c2)
Circle of radius 5 at coordinates (20, 26)
```

The initializer is implicitly called, and a circle instance is created ❶. The `print` function implicitly uses the special `__str__` method ❷. Here we see that the `move` method of `Circle`'s parent class `Shape` is available ❸. A method is called using attribute syntax on the object instance: `object.method()`. The first (`self`) parameter is set implicitly.

3.6 Summary

This ends our overview of Python. Don't worry if some parts were confusing. You need an understanding of only the broad strokes at this point. The chapters in part 2 and part 3 won't assume prior knowledge of their concepts and will walk you through these features in detail. You can also think of this as an early preview of what your level of knowledge will be when you're ready to move on to the chapters in part 4. You may find it valuable to return here and work through the appropriate examples as a review after we cover the features in subsequent chapters.

If this chapter was mostly a review for you, or there were only a few features you would like to learn more about, feel free to jump ahead, using the index, the table of contents, or the appendix. You can always slow down if anything catches your eye. You probably should have an understanding of Python to the level that you have no trouble understanding most of this chapter before you move on to the chapters in part 4.

Part 2

The essentials

In the chapters that follow, we'll show you the essentials of Python. We'll start from the absolute basics of what makes a Python program and move through Python's built-in data types and control structures, as well as defining functions and using modules.

The last section of this part moves on to show you how to write standalone Python programs, manipulate files, handle errors, and use classes. The section ends with chapter 16, which is a brief introduction to GUI programming using Python's `tkinter` module.

The absolute basics



This chapter covers

- Indenting and block structuring
- Differentiating comments
- Assigning variables
- Evaluating expressions
- Using common data types
- Getting user input
- Using correct Pythonic style

This chapter describes the absolute basics in Python: assignments and expressions, how to type a number or a string, how to indicate comments in code, and so forth. It starts out with a discussion of how Python block structures its code, which is different from any other major language.

4.1 Indentation and block structuring

Python differs from most other programming languages because it uses whitespace and indentation to determine block structure (that is, to determine what constitutes the body of a loop, the `else` clause of a conditional, and so on). Most languages use

braces of some sort to do this. Here is C code that calculates the factorial of 9, leaving the result in the variable `r`:

```
/* This is C code */
int n, r;
n = 9;
r = 1;
while (n > 0) {
    r *= n;
    n--;
}
```


The `{` and `}` delimit the body of the `while` loop, the code that is executed with each repetition of the loop. The code is usually indented more or less as shown, to make clear what's going on, but it could also be written like this:

```
/* And this is C code with arbitrary indentation */
    int n, r;
        n = 9;
        r = 1;
    while (n > 0) {
r *= n;
n--;
    }
```


It still would execute correctly, even though it's rather difficult to read.

Here's the Python equivalent:

```
# This is Python code. (Yea!)
n = 9
r = 1
while n > 0:
    r = r * n
    n = n - 1
```



**Python also supports
C-style `r *= n`**



**Python also supports
`n -= 1`**

Python doesn't use braces to indicate code structure; instead, the indentation itself is used. The last two lines of the previous code are the body of the `while` loop because they come immediately after the `while` statement and are indented one level further than the `while` statement. If they weren't indented, they wouldn't be part of the body of the `while`.

Using indentation to structure code rather than braces may take some getting used to, but there are significant benefits:

- It's impossible to have missing or extra braces. You'll never need to hunt through your code for the brace near the bottom that matches the one a few lines from the top.
- The visual structure of the code reflects its real structure. This makes it easy to grasp the skeleton of code just by looking at it.
- Python coding styles are mostly uniform. In other words, you're unlikely to go crazy from dealing with someone's idea of aesthetically pleasing code. Their code will look pretty much like yours.

You probably use consistent indentation in your code already, so this won't be a big step for you. If you're using IDLE, it automatically indents lines. You just need to back-space out of levels of indentation when desired. Most programming editors and IDEs, including Emacs, VIM, and Eclipse, to name a few, provide this functionality as well. One thing that may trip you up once or twice until you get used to it is that the Python interpreter returns an error message if you have a space (or spaces) preceding the commands you enter at a prompt.

4.2 Differentiating comments

For the most part, anything following a `#` symbol in a Python file is a comment and is disregarded by the language. The obvious exception is a `#` in a string, which is just a character of that string:

```
# Assign 5 to x
x = 5
x = 3                # Now x is 3
x = "# This is not a comment"
```

We'll put comments into Python code frequently.

4.3 Variables and assignments

The most commonly used command in Python is assignment, which looks pretty close to what you might've used in other languages. Python code to create a variable called `x` and assign the value 5 to that variable is

```
x = 5
```

In Python, neither a variable type declaration nor an end-of-line delimiter is necessary, unlike in many other computer languages. The line is ended by the end of the line. Variables are created automatically when they're first assigned.

Python variables can be set to any object, unlike C or many other languages' variables, which can store only the type of value they're declared as. The following is perfectly legal Python code:

```
>>> x = "Hello"
>>> print(x)
Hello
>>> x = 5
>>> print(x)
5
```

`x` starts out referring to the string object `"Hello"` and then refers to the integer object `5`. Of course, this feature can be abused because arbitrarily assigning the same variable name to refer successively to different data types can make code confusing to understand.

A new assignment overrides any previous assignments. The `del` statement deletes the variable. Trying to print the variable's contents after deleting it gives an error the same as if the variable had never been created in the first place.

```
>>> x = 5
>>> print(x)
5
>>> del x
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

Here we have our first look at a *traceback*, which is printed when an error, called an *exception*, has been detected. The last line tells us what exception was detected, which in this case is a `NameError` exception on `x`. After its deletion, `x` is no longer a valid variable name. In this example, the trace returns only `line 1, in <module>` because only the single line has been sent in the interactive mode. In general, the full dynamic call structure of the existing function calls at the time of the occurrence of the error is returned. If you're using IDLE, you obtain the same information with some small differences; it may look something like this:

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

Chapter 14 will describe this mechanism in more detail. A full list of the possible exceptions and what causes them is in the appendix of this book. Use the index to find any specific exception (such as `NameError`) you receive.

Variable names are case sensitive and can include any alphanumeric character as well as underscores but must start with a letter or underscore. See section 4.10 for more guidance on the Pythonic style for creating variable names.

4.4 Expressions

Python supports arithmetic and similar expressions; these will be familiar to most readers. The following code calculates the average of 3 and 5, leaving the result in the variable `z`:

```
x = 3
y = 5
z = (x + y) / 2
```

Note that unlike the arithmetic rules of C in terms of type coercions, arithmetic operators involving only integers do *not* always return an integer. Even though all the values are integers, division (starting with Python 3) returns a floating-point number, so the fractional part isn't truncated. If you want traditional integer division returning a truncated integer, you can use the `//` instead.

Standard rules of arithmetic precedence apply; if we'd left out the parentheses in the last line, it would've been calculated as `x + (y / 2)`.

Expressions don't have to involve just numerical values; strings, Boolean values, and many other types of objects can be used in expressions in various ways. We'll discuss these in more detail as they're used.

4.5 Strings

You've already seen that Python, like most other programming languages, indicates strings through the use of double quotes. This line leaves the string `"Hello, World"` in the variable `x`:

```
x = "Hello, World"
```

Backslashes can be used to escape characters, to give them special meanings. `\n` means the newline character, `\t` means the tab character, `\\` means a single normal backslash character, and `\"` is a plain double-quote character. It doesn't end the string:

```
x = "\tThis string starts with a \"tab\"."
x = "This string contains a single backslash(\\)."
```

You can use single quotes instead of double quotes. The following two lines do the same thing:

```
x = "Hello, World"
x = 'Hello, World'
```

The only difference is that you don't need to backslash `"` characters in single-quoted strings or `'` characters in double-quoted strings:

```
x = "Don't need a backslash"
x = 'Can\'t get by without a backslash'
x = "Backslash your \" character!"
x = 'You can leave the " alone'
```

You can't split a normal string across lines; this code won't work:

```
# This Python code will cause an ERROR -- you can't split the string
across two lines.
x = "This is a misguided attempt to
put a newline into a string without using backslash-n"
```

But Python offers triple-quoted strings, which let you do this and permit single and double quotes to be included without backslashes:

```
x = """Starting and ending a string with triple " characters
permits embedded newlines, and the use of " and ' without
backslashes"""
```

Now `x` is the entire sentence between the `"""` delimiters. (You can also use triple single quotes—`' '`—instead of triple double quotes to do the same thing.)

Python offers enough string-related functionality that chapter 6 is devoted to the topic.

4.6 Numbers

Because you're probably familiar with standard numeric operations from other languages, this book doesn't contain a separate chapter describing Python's numeric abilities. This section describes the unique features of Python numbers, and the Python documentation lists the available functions.

Python offers four kinds of numbers: *integers*, *floats*, *complex numbers*, and *Booleans*. An integer constant is written as an integer—0, -11, +33, 123456—and has unlimited range, restricted only by the resources of your machine. A float can be written with a decimal point or using scientific notation: 3.14, -2E-8, 2.718281828. The precision of these values is governed by the underlying machine but is typically equal to double (64-bit) types in C. Complex numbers are probably of limited interest and are discussed separately later in the section. Booleans are either `True` or `False` and behave identically to 1 and 0 except for their string representations.

Arithmetic is much like it is in C. Operations involving two integers produce an integer, except for division (`/`), where a float results. If the `//` division symbol is used, the result is an integer, with truncation. Operations involving a float always produce a float. Here are a few examples:

```
>>> 5 + 2 - 3 * 2
1
>>> 5 / 2          # floating point result with normal division
2.5
>>> 5 / 2.0        # also a floating point result
2.5
>>> 5 // 2         # integer result with truncation when divided using '//'
2
>>> 30000000000    # This would be too large to be an int in many languages
30000000000
>>> 30000000000 * 3
90000000000
>>> 30000000000 * 3.0
90000000000.0
>>> 2.0e-8         # Scientific notation gives back a float
2e-08
>>> 3000000 * 3000000
9000000000000
>>> int(200.2)
200
>>> int(2e2)
200
>>> float(200)
200.0
```

These are explicit conversions between types ❶. `int` will truncate float values.

Numbers in Python have two advantages over C or Java. First, integers can be arbitrarily large; and second, the division of two integers results in a float.

4.6.1 Built-in numeric functions

Python provides the following number-related functions as part of its core:

`abs`, `divmod`, `float`, `hex`, `long`, `max`, `min`, `oct`, `pow`, `round`

See the documentation for details.

4.6.2 Advanced numeric functions

More advanced numeric functions such as the trig and hyperbolic trig functions, as well as a few useful constants, aren't built-ins in Python but are provided in a standard module called `math`. Modules will be explained in detail later; for now, it's sufficient to know that the math functions in this section must be made available by starting your Python program or interactive session with the statement

```
from math import *
```

The `math` module provides the following functions and constants:

`acos`, `acosh`, `asin`, `asinh`, `atan`, `atan2`, `atanh`, `ceil`, `copysign`,
`cos`, `cosh`, `degrees`, `e`, `exp`, `fabs`, `factorial`, `floor`, `fmod`,
`frexp`, `fsum`, `hypot`, `isinf`, `isnan`, `ldexp`, `log`, `log10`, `log1p`,
`modf`, `pi`, `pow`, `radians`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `trunc`

See the documentation for details.

4.6.3 Numeric computation

The core Python installation isn't well suited to intensive numeric computation because of speed constraints. But the powerful Python extension `NumPy` provides highly efficient implementations of many advanced numeric operations. The emphasis is on array operations, including multidimensional matrices and more advanced functions such as the Fast Fourier Transform. You should be able to find `NumPy` (or links to it) at www.scipy.org.

4.6.4 Complex numbers

Complex numbers are created automatically whenever an expression of the form `nj` is encountered, with `n` having the same form as a Python integer or float. `j` is, of course, standard engineering notation for the imaginary number equal to the square root of -1 , for example:

```
>>> (3+2j)
(3+2j)
```

Note that Python expresses the resulting complex number in parentheses, as a way of indicating that what is printed to the screen represents the value of a single object:

```
>>> 3 + 2j - (4+4j)
(-1-2j)
```

```
>>> (1+2j) * (3+4j)
(-5+10j)
>>> 1j * 1j
(-1+0j)
```

Calculating `j * j` gives the expected answer of `-1`, but the result remains a Python complex number object. Complex numbers are never converted automatically to equivalent real or integer objects. But you can easily access their real and imaginary parts with `real` and `imag`:

```
>>> z = (3+5j)
>>> z.real
3.0
>>> z.imag
5.0
```

Note that real and imaginary parts of a complex number are always returned as floating-point numbers.

4.6.5 *Advanced complex-number functions*

The functions in the `math` module don't apply to complex numbers; the rationale is that most users want the square root of `-1` to generate an error, not an answer! Instead, similar functions, which can operate on complex numbers, are provided in the `cmath` module:

```
acos, acosh, asin, asinh, atan, atanh, cos, cosh, e, exp,
isinf, isnan, log, log10, phase, pi, polar, rect, sin, sinh,
sqrt, tan, tanh
```

In order to make clear in the code that these are special-purpose complex-number functions and to avoid name conflicts with the more normal equivalents, it's best to import the `cmath` module by saying

```
import cmath
```

and then to explicitly refer to the `cmath` package when using the function:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
```

Minimizing from <module> import *

This is a good example of why it's best to minimize the use of the `from <module> import *` form of the `import` statement. If you imported first the `math` and then the `cmath` modules using it, the commonly named functions in `cmath` would override those of `math`. It's also more work for someone reading your code to figure out the source of the specific functions you use. Some modules are explicitly designed to use this form of import.

See chapter 10 for more details on how to use modules and module names.

The important thing to keep in mind is that by importing the `cmath` module, you can do almost anything you can do with other numbers.

4.7 The None value


In addition to standard types such as strings and numbers, Python has a special basic data type that defines a single special data object called `None`. As the name suggests, `None` is used to represent an empty value. It appears in various guises throughout Python. For example, a procedure in Python is just a function that doesn't explicitly return a value, which means that, by default, it returns `None`.

`None` is often useful in day-to-day Python programming as a placeholder, to indicate a point in a data structure where meaningful data will eventually be found, even though that data hasn't yet been calculated. You can easily test for the presence of `None`, because there is only one instance of `None` in the entire Python system (all references to `None` point to the same object), and `None` is equivalent only to itself.

4.8 Getting input from the user

You can also use the `input()` function to get input from the user. Use the prompt string you want displayed to the user as `input`'s parameter:

```
>>> name = input("Name? ")
Name? Vern
>>> print(name)
Vern
>>> age = int(input("Age? "))
Age? 28
>>> print(age)
28
>>>
```



Converts input
from string to int

This is a fairly simple way to get user input. The one catch is that the input comes in as a string, so if you want to use it as a number, you have to use the `int()` or `float()` function to convert it.

4.9 Built-in operators

Python provides various built-in operators, from the standard (such as `+`, `*`, and so on) to the more esoteric, such as operators for performing bit shifting, bitwise logical functions, and so forth. Most of these operators are no more unique to Python than to any other language, and hence I won't explain them in the main text. You can find a complete list of the Python built-in operators in the documentation.

4.10 Basic Python style

Python has relatively few limitations on coding style with the obvious exception of the requirement to use indentation to organize code into blocks. Even in that case, the amount of indentation and type of indentation (tabs versus spaces) isn't mandated. However, there are preferred stylistic conventions for Python, which are contained in

Python Enhancement Proposal (PEP) 8, which is summarized in the appendix and can be found online at www.python.org/dev/peps/pep-0008/. A selection of Pythonic conventions is provided in table 4.1, but to fully absorb Pythonic style you'll need to periodically reread PEP 8.

Table 4.1 Pythonic coding conventions

Situation	Suggestion	Example
Module/package names	short, all lowercase, underscores only if needed	<code>imp, sys</code>
Function names	all lowercase, underscores_for_readability	<code>foo(), my_func()</code>
Variable names	all lowercase, underscores_for_readability	<code>my_var</code>
Class names	CapitalizeEachWord	<code>MyClass</code>
Constant names	ALL_CAPS_WITH_UNDERSCORES	<code>PI, TAX_RATE</code>
Indentation	4 spaces per level, don't use tabs	
Comparisons	Don't compare explicitly to True or False	<code>if my_var:</code> <code>if not my_var:</code>

I strongly urge you to follow the conventions of PEP 8. They're wisely chosen and time tested and will make your code easier for you and other Python programmers to understand.

4.11 Summary

That's the view of Python from 30,000 feet. If you're an experienced programmer, you're probably already seeing how you can write your code in Python. If that's the case, you should feel free to start experimenting with your own code. Many programmers find it surprisingly easy to pick up Python syntax, because there are relatively few surprises. Once you pick up the basics of the language, it's very predictable and consistent.

In any case, we have just covered the broadest outlines of the language, and there are lots of details that we still need to cover, beginning in the next chapter with one of the workhorses of Python, lists.

5

Lists, tuples, and sets

This chapter covers

- Manipulating lists and list indices
- Modifying lists
- Sorting
- Using common list operations
- Handling nested lists and deep copies
- Using tuples
- Creating and using sets

In this chapter, we'll discuss the two major Python sequence types: lists and tuples. At first, lists may remind you of arrays in many other languages, but don't be fooled—lists are a good deal more flexible and powerful than plain arrays. This chapter also discusses a newer Python collection type: sets. Sets are useful when an object's membership in the collection, as opposed to its position, is important.

Tuples are like lists that can't be modified—you can think of them as a restricted type of list or as a basic record type. We'll discuss why we need such a restricted data type later in the chapter.

Most of the chapter is devoted to lists, because if you understand lists, you pretty much understand tuples. The last part of the chapter discusses the differences between lists and tuples, in both functional and design terms.

5.1 *Lists are like arrays*

A list in Python is much the same thing as an array in Java or C or any other language. It's an ordered collection of objects. You create a list by enclosing a comma-separated list of elements in square brackets, like so:

```
# This assigns a three-element list to x
x = [1, 2, 3]
```

Note that you don't have to worry about declaring the list or fixing its size ahead of time. This line creates the list as well as assigns it, and a list automatically grows or shrinks in size as needed.

Arrays in Python

A typed `array` module is available in Python that provides arrays based on C data types. Information on its use can be found in the *Python Library Reference*. I suggest you look into it only if you run into a situation where you really need the performance improvement. If a situation calls for numerical computations, you should consider using `NumPy`, mentioned in section 4.6.3, available at www.scipy.org.

Unlike lists in many other languages, Python lists can contain different types of elements; a list element can be any Python object. Here's a list that contains a variety of elements:

```
# First element is a number, second is a string, third is another list.
x = [2, "two", [1, 2, 3]]
```

Probably the most basic built-in list function is the `len` function, which returns the number of elements in a list:

```
>>> x = [2, "two", [1, 2, 3]]
>>> len(x)
3
```

Note that the `len` function doesn't count the items in the inner, nested list.

5.2 *List indices*

Understanding how list indices work will make Python much more useful to you. Please read the whole section!

Elements can be extracted from a Python list using a notation like C's array indexing. Like C and many other languages, Python starts counting from 0; asking for element 0

returns the first element of the list, asking for element 1 returns the second element, and so forth. Here are a few examples:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[0]
'first'
>>> x[2]
'third'
```

But Python indexing is more flexible than C indexing; if indices are negative numbers, they indicate positions counting from the end of the list, with -1 being the last position in the list, -2 being the second-to-last position, and so forth. Continuing with the same list `x`, we can do the following:

```
>>> a = x[-1]
>>> a
'fourth'
>>> x[-2]
'third'
```

For operations involving a single list index, it's generally satisfactory to think of the index as pointing at a particular element in the list. For more advanced operations, it's more correct to think of list indices as indicating positions *between* elements. In the list `["first", "second", "third", "fourth"]`, you can think of the indices as pointing like this:

<code>x = [</code>		<code>"first",</code>		<code>"second",</code>		<code>"third",</code>		<code>"fourth"</code>		<code>]</code>
Positive indices	0		1		2		3			
Negative indices	-4		-3		-2		-1			

This is irrelevant when you're extracting a single element, but Python can extract or assign to an entire sublist at once, an operation known as *slicing*. Instead of entering `list[index]` to extract the item just after `index`, enter `list[index1:index2]` to extract all items including `index1` and up to (but not including) `index2` into a new list. Here are some examples:

```
>>> x = ["first", "second", "third", "fourth"]
>>> x[1:-1]
['second', 'third']
>>> x[0:3]
['first', 'second', 'third']
>>> x[-2:-1]
['third']
```

It may seem reasonable that if the second index indicates a position in the list *before* the first index, this would return the elements between those indices in reverse order, but this isn't what happens. Instead, this returns an empty list:

```
>>> x[-1:2]
[]
```

When slicing a list, it's also possible to leave out `index1` or `index2`. Leaving out `index1` means “go from the beginning of the list,” and leaving out `index2` means “go to the end of the list”:

```
>>> x[:3]
['first', 'second', 'third']
>>> x[2:]
['third', 'fourth']
```

Omitting both indices makes a new list that goes from the beginning to the end of the original list; that is, it copies the list. This is useful when you wish to make a copy that you can modify, without affecting the original list:

```
>>> y = x[:]
>>> y[0] = '1 st'
>>> y
['1 st', 'second', 'third', 'fourth']
>>> x
['first', 'second', 'third', 'fourth']
```

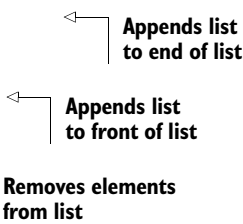
5.3 *Modifying lists*

You can use list index notation to modify a list as well as to extract an element from it. Put the index on the left side of the assignment operator:

```
>>> x = [1, 2, 3, 4]
>>> x[1] = "two"
>>> x
[1, 'two', 3, 4]
```

Slice notation can be used here too. Saying something like `lista[index1:index2] = listb` causes all elements of `lista` between `index1` and `index2` to be replaced with the elements in `listb`. `listb` can have more or fewer elements than are removed from `lista`, in which case the length of `lista` will be altered. You can use slice assignment to do a number of different things, as shown here:

```
>>> x = [1, 2, 3, 4]
>>> x[len(x):] = [5, 6, 7]
>>> x
[1, 2, 3, 4, 5, 6, 7]
>>> x[:0] = [-1, 0]
>>> x
[-1, 0, 1, 2, 3, 4, 5, 6, 7]
>>> x[1:-1] = []
>>> x
[-1, 7]
```



Appending a single element to a list is such a common operation that there's a special `append` method to do it:

```
>>> x = [1, 2, 3]
>>> x.append("four")
>>> x
[1, 2, 3, 'four']
```

One problem can occur if you try to append one list to another. The list gets appended as a single element of the main list:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.append(y)
>>> x
[1, 2, 3, 4, [5, 6, 7]]
```

The `extend` method is like the `append` method, except that it allows you to add one list to another:

```
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 5, 6, 7]
```

There is also a special `insert` method to insert new list elements between two existing elements or at the front of the list. `insert` is used as a method of lists and takes two additional arguments; the first is the index position in the list where the new element should be inserted, and the second is the new element itself:

```
>>> x = [1, 2, 3]
>>> x.insert(2, "hello")
>>> print(x)
[1, 2, 'hello', 3]
>>> x.insert(0, "start")
>>> print(x)
['start', 1, 2, 'hello', 3]
```

`insert` understands list indices as discussed in the section on slice notation, but for most uses it's easiest to think of `list.insert(n, elem)` as meaning `insert elem` just before the n th element of list. `insert` is just a convenience method. Anything that can be done with `insert` can also be done using slice assignment; that is, `list.insert(n, elem)` is the same thing as `list[n:n] = [elem]` when n is nonnegative. Using `insert` makes for somewhat more readable code, and `insert` even handles negative indices:

```
>>> x = [1, 2, 3]
>>> x.insert(-1, "hello")
>>> print(x)
[1, 2, 'hello', 3]
```

The `del` statement is the preferred method of deleting list items or slices. It doesn't do anything that can't be done with slice assignment, but it's usually easier to remember and easier to read:

```
>>> x = ['a', 2, 'c', 7, 9, 11]
>>> del x[1]
>>> x
['a', 'c', 7, 9, 11]
>>> del x[:2]
>>> x
[7, 9, 11]
```

In general, `del list[n]` does the same thing as `list[n:n+1] = []`, whereas `del list[m:n]` does the same thing as `list[m:n] = []`.

The `remove` method isn't the converse of `insert`. Whereas `insert` inserts an element at a specified location, `remove` looks for the first instance of a given value in a list and removes that value from the list:

```
>>> x = [1, 2, 3, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 3, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 5]
>>> x.remove(3)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

If `remove` can't find anything to remove, it raises an error. You can catch this error using the exception-handling abilities of Python, or you can avoid the problem by using `in` to check for the presence of something in a list before attempting to remove it.

The `reverse` method is a more specialized list modification method. It efficiently reverses a list in place:

```
>>> x = [1, 3, 5, 6, 7]
>>> x.reverse()
>>> x
[7, 6, 5, 3, 1]
```

5.4 **Sorting lists**

Lists can be sorted using the built-in Python `sort` method:

```
>>> x = [3, 8, 4, 0, 2, 1]
>>> x.sort()
>>> x
[0, 1, 2, 3, 4, 8]
```

This does an in-place sort—that is, it changes the list being sorted. To sort a list without changing the original list, make a copy of it first.

```
>>> x = [2, 4, 1, 3]
>>> y = x[:]
>>> y.sort()
>>> y
[1, 2, 3, 4]
>>> x
[2, 4, 1, 3]
```

Sorting works with strings, too:

```
>>> x = ["Life", "Is", "Enchanting"]
>>> x.sort()
>>> x
['Enchanting', 'Is', 'Life']
```


The `sort` method can sort just about anything, because Python can compare just about anything. But there is one caveat in sorting. The default key method used by `sort` requires that all items in the list be of comparable types. That means that using the `sort` method on a list containing both numbers and strings will raise an exception:

```
>>> x = [1, 2, 'hello', 3]
>>> x.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

On the other hand, we can sort a list of lists:

```
>>> x = [[3, 5], [2, 9], [2, 3], [4, 1], [3, 2]]
>>> x.sort()
>>> x
[[2, 3], [2, 9], [3, 2], [3, 5], [4, 1]]
```

According to the built-in Python rules for comparing complex objects, the sublists are sorted first by ascending first element and then by ascending second element.

`sort` is even more flexible than this—it's possible to use your own key function to determine how elements of a list are sorted.

5.4.1 Custom sorting

To use custom sorting, you need to be able to define functions, something we haven't talked about. In this section we'll also use the fact that `len(string)` returns the number of characters in a string. String operations are discussed more fully in chapter 6.

By default, `sort` uses built-in Python comparison functions to determine ordering, which is satisfactory for most purposes. There will be times, though, when you want to sort a list in a way that doesn't correspond to this default ordering. For example, let's say we wish to sort a list of words by the number of characters in each word, in contrast to the lexicographic sort that would normally be carried out by Python.

To do this, write a function that will return the value, or key, that we want to sort on, and use it with the `sort` method. That function in the context of `sort` is a function that takes one argument and returns the key or value the `sort` function is to use.

For our number-of-characters ordering, a suitable key function could be

```
def compare_num_of_chars(string1):
    return len(string1)
```

This key function is trivial. It passes the length of each string back to the `sort` method, rather than the strings themselves.

After you define the key function, using it is a matter of passing it to the `sort` method using the `key` keyword. Because functions are Python objects, they can be passed around like any other Python object. Here's a small program that illustrates the difference between a default sort and our custom sort:

```
>>> def compare_num_of_chars(string1):
...     return len(string1)
```

```
>>> word_list = ['Python', 'is', 'better', 'than', 'C']
>>> word_list.sort()
>>> print(word_list)
['C', 'Python', 'better', 'is', 'than']
>>> word_list = ['Python', 'is', 'better', 'than', 'C']
>>> word_list.sort(key=compare_num_of_chars)
>>> print(word_list)
['C', 'is', 'than', 'Python', 'better']
```

The first list is in lexicographic order (with uppercase coming before lowercase), and the second list is ordered by ascending number of characters.

Custom sorting is very useful, but if performance is critical, it may be slower than the default. Usually this impact is minimal, but if the key function is particularly complex, the effect may be more than desired, especially for sorts involving hundreds of thousands or millions of elements.

One particular place to avoid custom sorts is where you want to sort a list in descending, rather than ascending, order. In this case, use the `sort` method's `reverse` parameter set to `True`. If for some reason you don't want to do that, it's still better to sort the list normally and then use the `reverse` method to invert the order of the resulting list. These two operations together—the standard sort and the reverse—will still be much faster than a custom sort.

5.4.2 *The sorted() function*

Lists have a built-in method to sort themselves, but other iterables in Python, like the keys of a dictionary, for example, don't have a `sort` method. Python also has the built-in function `sorted()`, which returns a sorted list from any iterable. `sorted()` uses the same `key` and `reverse` parameters as the `sort` method:

```
>>> x = (4, 3, 1, 2)
>>> y = sorted(x)
>>> y
[1, 2, 3, 4]
```

5.5 *Other common list operations*

A number of other list methods are frequently useful, but they don't fall into any specific category.

5.5.1 *List membership with the in operator*

It's easy to test if a value is in a list using the `in` operator, which returns a Boolean value. You can also use the converse, the `not in` operator:

```
>>> 3 in [1, 3, 4, 5]
True
>>> 3 not in [1, 3, 4, 5]
False
>>> 3 in ["one", "two", "three"]
False
>>> 3 not in ["one", "two", "three"]
True
```

5.5.2 List concatenation with the + operator

To create a list by concatenating two existing lists, use the `+` (list concatenation) operator. This will leave the argument lists unchanged.

```
>>> z = [1, 2, 3] + [4, 5]
>>> z
[1, 2, 3, 4, 5]
```

5.5.3 List initialization with the * operator

Use the `*` operator to produce a list of a given size, which is initialized to a given value. This is a common operation for working with large lists whose size is known ahead of time. Although you can use `append` to add elements and automatically expand the list as needed, you obtain greater efficiency by using `*` to correctly size the list at the start of the program. A list that doesn't change in size doesn't incur any memory reallocation overhead:

```
>>> z = [None] * 4
>>> z
[None, None, None, None]
```

When used with lists in this manner, `*` (which in this context is called the *list multiplication operator*) replicates the given list the indicated number of times and joins all the copies to form a new list. This is the standard Python method for defining a list of a given size ahead of time. A list containing a single instance of `None` is commonly used in list multiplication, but the list can be anything:

```
>>> z = [3, 1] * 2
>>> z
[3, 1, 3, 1]
```

5.5.4 List minimum or maximum with min and max

You can use `min` and `max` to find the smallest and largest elements in a list. You'll probably use these mostly with numerical lists, but they can be used with lists containing any type of element. Trying to find the maximum or minimum object in a set of objects of different types causes an error if it doesn't make sense to compare those types:

```
>>> min([3, 7, 0, -2, 11])
-2
>>> max([4, "Hello", [1, 2]])
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    max([4, "Hello", [1, 2]])
TypeError: unorderable types: str() > int()
```

5.5.5 List search with index

If you wish to find where in a list a value can be found (rather than wanting to know only if the value is in the list), use the `index` method. It searches through a

list looking for a list element equivalent to a given value and returns the position of that list element:

```
>>> x = [1, 3, "five", 7, -2]
>>> x.index(7)
3
>>> x.index(5)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
```

Attempting to find the position of an element that doesn't exist in the list at all raises an error, as shown here. This can be handled in the same manner as the analogous error that can occur with the `remove` method (that is, by testing the list with `in` before using `index`).

5.5.6 List matches with count

`count` also searches through a list, looking for a given value, but it returns the number of times that value is found in the list rather than positional information:

```
>>> x = [1, 2, 2, 3, 5, 2, 5]
>>> x.count(2)
3
>>> x.count(5)
2
>>> x.count(4)
0
```

5.5.7 Summary of list operations

You can see that lists are very powerful data structures, with possibilities that go far beyond plain old arrays. List operations are so important in Python programming that it's worth laying them out for easy reference, as shown in table 5.1.

Table 5.1 List operations

List operation	Explanation	Example
<code>[]</code>	Creates an empty list	<code>x = []</code>
<code>len</code>	Returns the length of a list	<code>len(x)</code>
<code>append</code>	Adds a single element to the end of a list	<code>x.append('y')</code>
<code>insert</code>	Inserts a new element at a given position in the list	<code>x.insert(0, 'y')</code>
<code>del</code>	Removes a list element or slice	<code>del(x[0])</code>
<code>remove</code>	Searches for and removes a given value from a list	<code>x.remove('y')</code>
<code>reverse</code>	Reverses a list in place	<code>x.reverse()</code>

Table 5.1 List operations (*continued*)

List operation	Explanation	Example
<code>sort</code>	Sorts a list in place	<code>x.sort()</code>
<code>+</code>	Adds two lists together	<code>x1 + x2</code>
<code>*</code>	Replicates a list	<code>x = ['y'] * 3</code>
<code>min</code>	Returns the smallest element in a list	<code>min(x)</code>
<code>max</code>	Returns the largest element in a list	<code>max(x)</code>
<code>index</code>	Returns the position of a value in a list	<code>x.index('y')</code>
<code>count</code>	Counts the number of times a value occurs in a list	<code>x.count('y')</code>
<code>in</code>	Returns whether an item is in a list	<code>'y' in x</code>

Being familiar with these list operations will make your life as a Python coder much easier.

5.6 Nested lists and deep copies

This is another advanced topic that you may want to skip if you're just learning the language.

Lists can be nested. One application of this is to represent two-dimensional matrices. The members of these can be referred to using two-dimensional indices. Indices for these work as follows:

```
>>> m = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
>>> m[0]
[0, 1, 2]
>>> m[0][1]
1
>>> m[2]
[20, 21, 22]
>>> m[2][2]
22
```

This mechanism scales to higher dimensions in the manner you would expect.

Most of the time, this is all you need to concern yourself with. But there is an issue with nested lists that you may run into. This is the result of the combination of the way variables refer to objects and the fact that some objects (such as lists) can be modified (they're mutable). An example is the best way to illustrate:

```
>>> nested = [0]
>>> original = [nested, 1]
>>> original
[[0], 1]
```

Figure 5.1 shows what this looks like.

The value in the nested list can now be changed using either the nested or the original variables:

```
>>> nested[0] = 'zero'
>>> original
[['zero'], 1]
>>> original[0][0] = 0
>>> nested
[0]
>>> original
[[0], 1]
```

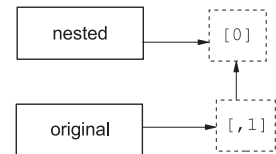


Figure 5.1 A list with its first item referring to a nested list

But if `nested` is set to another list, the connection between them is broken:

```
>>> nested = [2]
>>> original
[[0], 1]
```

Figure 5.2 illustrates this.

You've seen that you can obtain a copy of a list by taking a full slice (that is, `x[:]`). You can also obtain a copy of a list using the `+` or `*` operator (for example, `x + []` or `x * 1`). These are slightly less efficient than the slice method. All three create what is called a *shallow* copy of the list. This is probably what you want most of the time. But if your list has other lists nested in it, you may want to make a *deep* copy. You can do this with the `deepcopy` function of the `copy` module:

```
>>> original = [[0], 1]
>>> shallow = original[:]
>>> import copy
>>> deep = copy.deepcopy(original)
```

See figure 5.3 for an illustration.

The lists pointed at by the original or shallow variables are connected. Changing the value in the nested list through either one of them affects the other:

```
>>> shallow[1] = 2
>>> shallow
[[0], 2]
>>> original
[[0], 1]
>>> shallow[0][0] = 'zero'
>>> original
[['zero'], 1]
```

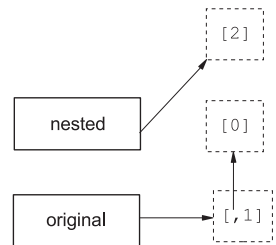


Figure 5.2 The first item of the original list is still a nested list, but the nested variable refers to a different list.

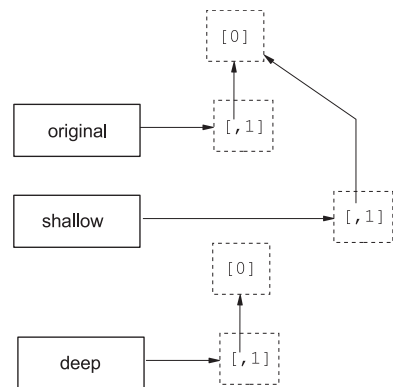


Figure 5.3 A shallow copy doesn't copy nested lists.

The deep copy is independent of the original, and no change to it has any effect on the original list:

```
>>> deep[0][0] = 5
>>> deep
```

```
[[5], 1]
>>> original
[['zero'], 1]
```

This behavior is the same for any other nested objects in a list that are modifiable (such as dictionaries).

Now that you've seen what lists can do, it's time to look at tuples.

5.7 Tuples

Tuples are data structures that are very similar to lists, but they can't be modified. They can only be created. Tuples are so much like lists that you may wonder why Python bothers to include them. The reason is that tuples have important roles that can't be efficiently filled by lists, as keys for dictionaries.

5.7.1 Tuple basics

Creating a tuple is similar to creating a list: assign a sequence of values to a variable. A list is a sequence that is enclosed by `[` and `]`; a tuple is a sequence that is enclosed by `(` and `)`:

```
>>> x = ('a', 'b', 'c')
```

This line creates a three-element tuple.

After a tuple is created, using it is so much like using a list that it's easy to forget they're different data types:

```
>>> x[2]
'c'
>>> x[1:]
('b', 'c')
>>> len(x)
3
>>> max(x)
'c'
>>> min(x)
'a'
>>> 5 in x
False
>>> 5 not in x
True
```

The main difference between tuples and lists is that tuples are immutable. An attempt to modify a tuple results in a confusing error message, which is Python's way of saying it doesn't know how to set an item in a tuple.

```
>>> x[2] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

You can create tuples from existing ones by using the `+` and `*` operators.

```
>>> x + x
('a', 'b', 'c', 'a', 'b', 'c')
```

```
>>> 2 * x
('a', 'b', 'c', 'a', 'b', 'c')
```

A copy of a tuple can be made in any of the same ways as for lists:

```
>>> x[:]
('a', 'b', 'c')
>>> x * 1
('a', 'b', 'c')
>>> x + ()
('a', 'b', 'c')
```

If you didn't read section 5.6, "Nested lists and deep copies," you can skip the rest of this paragraph. Tuples themselves can't be modified. But if they contain any mutable objects (for example, lists or dictionaries), these may be changed if they're still assigned to their own variables. Tuples that contain mutable objects aren't allowed as keys for dictionaries.

5.7.2 *One-element tuples need a comma*

A small syntactical point is associated with using tuples. Because the square brackets used to enclose a list aren't used elsewhere in Python, it's clear that `[]` means an empty list and `[1]` means a list with one element. The same thing isn't true with the parentheses used to enclose tuples. Parentheses can also be used to group items in expressions in order to force a certain evaluation order. If we say `(x + y)` in a Python program, do we mean that `x` and `y` should be added and then put into a one-element tuple, or do we mean that the parentheses should be used to force `x` and `y` to be added, before any expressions to either side come into play?

This is only a problem for tuples with one element, because tuples with more than one element always include commas to separate the elements, and the commas tell Python the parentheses indicate a tuple, not a grouping. In the case of one-element tuples, Python requires that the element in the tuple be followed by a comma, to disambiguate the situation. In the case of zero-element (empty) tuples, there's no problem. An empty set of parentheses must be a tuple, because it's meaningless otherwise.

```
>>> x = 3
>>> y = 4
>>> (x + y)    # This adds x and y.
7
>>> (x + y,)   # Including a comma indicates the parentheses denote a tuple.
(7,)
>>> ()         # To create an empty tuple, use an empty pair of parentheses.
()
```

5.7.3 *Packing and unpacking tuples*

As a convenience, Python permits tuples to appear on the left-hand side of an assignment operator, in which case variables in the tuple receive the corresponding values

from the tuple on the right-hand side of the assignment operator. Here's a simple example:

```
>>> (one, two, three, four) = (1, 2, 3, 4)
>>> one
1
>>> two
2
```

This can be written even more simply, because Python recognizes tuples in an assignment context even without the enclosing parentheses. The values on the right-hand side are packed into a tuple and then unpacked into the variables on the left-hand side:

```
one, two, three, four = 1, 2, 3, 4
```

One line of code has replaced the following four lines of code:

```
one = 1
two = 2
three = 3
four = 4
```

This is a convenient way to swap values between variables. Instead of saying

```
temp = var1
var1 = var2
var2 = temp
```

just say

```
var1, var2 = var2, var1
```

To make things even more convenient, Python 3 has an extended unpacking feature, allowing an element marked with a `*` to absorb any number elements not matching the other elements. Again, some examples will make this clearer:

```
>>> x = (1, 2, 3, 4)
>>> a, b, *c = x
>>> a, b, c
(1, 2, [3, 4])
>>> a, *b, c = x
>>> a, b, c
(1, [2, 3], 4)
>>> *a, b, c = x
>>> a, b, c
([1, 2], 3, 4)
>>> a, b, c, d, *e = x
>>> a, b, c, d, e
(1, 2, 3, 4, [])
```

Note that the starred element receives all the surplus items as a list, and that if there are no surplus elements, it receives an empty list.

Packing and unpacking can be performed using list delimiters as well:

```
>>> [a, b] = [1, 2]
>>> [c, d] = 3, 4
>>> [e, f] = (5, 6)
>>> (g, h) = 7, 8
>>> i, j = [9, 10]
>>> k, l = (11, 12)
>>> a
1
>>> [b, c, d]
[2, 3, 4]
>>> (e, f, g)
(5, 6, 7)
>>> h, i, j, k, l
(8, 9, 10, 11, 12)
```

5.7.4 Converting between lists and tuples

Tuples can be easily converted to lists with the `list` function (which takes any sequence as an argument and produces a new list with the same elements as the original sequence). Similarly, lists can be converted to tuples with the `tuple` function (which does the same thing but produces a new tuple instead of a new list):

```
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]
>>> tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

As an interesting side note, `list` is a convenient way to break a string into characters:

```
>>> list("Hello")
['H', 'e', 'l', 'l', 'o']
```

This works because `list` (and `tuple`) apply to any Python sequence, and a string is just a sequence of characters. (Strings are discussed fully in the next chapter.)

5.8 Sets

A *set* in Python is an unordered collection of objects used in situations where membership and uniqueness in the set are main things you need to know about that object. Just as with dictionary keys (as you'll see in chapter 7), the items in a set must be immutable and hashable. This means that ints, floats, strings, and tuples can be members of a set, but lists, dictionaries, and sets themselves can't.

5.8.1 Set operations

In addition to the operations that apply to collections in general, like `in`, `len`, and being able to use a `for` loop to iterate over all of their elements, sets also have several set-specific operations:

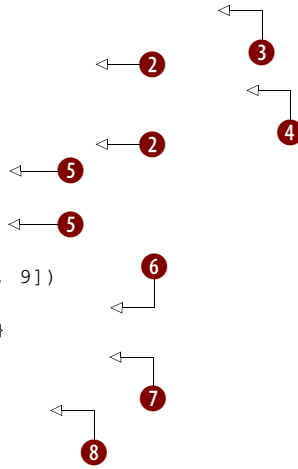
```
>>> x = set([1, 2, 3, 1, 3, 5])
>>> x
{1, 2, 3, 5}
```



```

>>> x.add(6)
>>> x
{1, 2, 3, 5, 6}
>>> x.remove(5)
>>> x
{1, 2, 3, 6}
>>> 1 in x
True
>>> 4 in x
False
>>> y = set([1, 7, 8, 9])
>>> x | y
{1, 2, 3, 6, 7, 8, 9}
>>> x & y
{1}
>>> x ^ y
{2, 3, 6, 7, 8, 9}
>>>

```



You can create a set by using `set` on a sequence, like a list **1**. When a sequence is made into a set, duplicates are removed **2**. After creating a set using the `set` function, you can use `add` **3** and `remove` **4** to change the elements in the set. The `in` keyword is used to check for membership of an object in a set **5**. You can also use `|` **6** to get the union, or combination, of two sets, `&` to get their intersection **7**, and `^` **8** to find their symmetric difference—that is, elements that are in one set or the other but not both.

These examples aren't a complete listing of set operations but are enough to give you a good idea of how sets work. For more information, refer to the official Python documentation.

5.8.2 Frozensets

Because sets aren't immutable and hashable, they can't belong to other sets. To remedy that situation there is another set type, `frozenset`, which is just like a set but can't be changed after creation. Because frozensets are immutable and hashable, they can be members of other sets:

```

>>> x = set([1, 2, 3, 1, 3, 5])
>>> z = frozenset(x)
>>> z
frozenset({1, 2, 3, 5})
>>> z.add(6)
Traceback (most recent call last):
  File "<pyshell#79>", line 1, in <module>
    z.add(6)
AttributeError: 'frozenset' object has no attribute 'add'
>>> x.add(z)
>>> x
{1, 2, 3, 5, frozenset({1, 2, 3, 5})}

```

5.9 **Summary**

Lists are a basic and highly useful data structure built into the Python language. In addition to demonstrating fairly standard array-like behavior, lists possess additional functionality, such as automatic resizing, the ability to use slice notation, and a good set of convenience functions, methods, and operators. Note that there are a few more list methods than were covered in this chapter. You'll find details on these in the Python documentation.

Tuples are similar to lists but can't be modified. They take up slightly less memory and are faster to access. They aren't as flexible but are more efficient than lists. Their normal use (from the point of view of a Python coder) is to serve as dictionary keys, which is discussed in chapter 7.

Sets are also sequence structures but with two differences from lists and tuples. Although sets do have an order, that order is arbitrary and not under the programmer's control. The second difference is that the elements in a set must be unique.

Lists, tuples, and sets are structures that embody the idea of a sequence of elements. As you'll see in the next chapter, strings are also sequences with some additional methods.