Module 1 - Why Python?

Why do people use Python?
Python is a popular open source programming language used for both standalone programs and scripting applications in a wide variety of domains. It is free, portable, powerful, and is both relatively easy and remarkably fun to use. Programmers of different software industry found Python's focus on developer productivity and software quality to be a strategic advantage in projects both large and small. It is multi-paradigm computer programming language.

Most Python programs run unchanged on all major computer platforms. Porting Python code between Linux and Windows, for example, s usually just a matter of copy a script's code between machines. Moreover, Python offers multi options for coding portability graphical use interfaces, database access programs, web-based systems, and more.

Python comes with a large collection of prebuilt and portable functionality, known as the standard library. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both homegrown libraries and a vast collection of third-party application support software. The NumPy extension, for instance, has been described as a free and more powerful equivalent to the MatLab numerical programming system.

Python scripts can easily communicate with other parts of an application, using a variety of integrations mechanisms. Such integrations allow Python to be used as a product customization and extension tool. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks with interface like SOAP, XML-RPC, and CORBA.

Is Python a scripting language?
Python is a general-purpose programming language that is often applied in scripting roles. It is commonly defined as an object-oriented language – a definition that blends support for OOP with an overall orientation toward scripting roles. If pressed for a one-liner, Python is probably better known as a general-purpose programming language that blends procedural, functional, and object-oriented paradigms.

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

What are Python's technical strengths?
Listed below is a quick introduction to some of the Python's top technical features:

- **It is Object-Oriented and Functional** – Python is an object-oriented language, from ground up. Its class model supports advanced notions such as Polymorphism, Operator Overloading, and Multiple Inheritance. In recent years, Python has acquired built-in support for functional programming – a set that by most measures include generators, comprehensions, closures, maps, decorators, anonymous function lambdas, and first-class function objects. These can serve as both complement and alternative to its OOP tools.
- **It is Free** – Python is completely free to use and distribute. As with open source software, such as Tcl, Perl, Linux, and Apache. You can fetch the entire Python system's source code for free on the Internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products. In fact, you can also sell Python's source code, if you are so inclined.
- **It is Portable** – The standard implementation of Python is written in portable ANSI C, and it compiles and runs on virtually every major platform currently in use. For example, Python programs run today on everything from PDA to supercomputers.
- **It is Powerful** – From a feature perspective, Python is something of a hybrid. Its toolset places it between traditional scripting languages (such as Tcl, Scheme, and Perl) and systems development languages (Sch as C, C++, and Java). It provides all the simplicity and ease of use of a scripting language, along with more

advanced software-engineering tools typically found in compiled languages. Unlike some scripting languages, this combination makes Python useful for large-scale development projects. In Python's toolbox, you will find Dynamic Typing, Automatic Memory Management, Programming-in-the-large support, Built-in object types, Built-in tools, and Library utilities.

- **It is Mixable** – Python programs can easily be glued to components written in other languages in a variety ways.
- **It is Relatively Easy to Use** – Compared to alternatives like C++, Java, and C#, Python programming seems astonishing simple to most observers. To run a Python program, you simply type it and run it. There are no intermediate compile and link steps. Python executes programs immediately, which makes or an interactive programming experience and rapid turnaround after program changes – in many cases, you can witness the effect of program change nearly as fast as you type it. It also provides deliberately simple and powerful built-in tools. In fact, some have gone so far to call Python executable pseudo-code. Because it eliminates much of the complexity in other tools, Python programs are simpler, smaller, and more flexible than equivalent programs in other popular languages.
- **It is Relatively Easy to Learn –** Especially when compared to other widely used programming languages, the core Python language is remarkably easy to learn.

Module 2 - How Python Runs Programs

This module and the next take a quick look at program execution – How you launch code, and how Python runs it. In this module we'll study how the Python Interpreter executes programs in general. Module 3 then show you how to get your own programs up and running.

Startup details are inherently platform-specific, and some of the material in these two modules may not apply to the platform you work on, so more advanced developers should feel free to skip parts not relevant to their intended use. Likewise, developers who have used similar tools in the past and prefer to get to the meat of the language quickly may want to file some of the modules away as "for future reference". For the rest, let's take a brief look at the way that Python will run our code, before we learn how to write it.

Introducing the Python interpreter
Python language is considered as 4GL. Therefore, it required a special translator called **Interpreter** for it execution. An interpreter is a kind of program that executes other programs. When you write a Python program, the Python interpreter reads your program and carried out the instructions it contains. In effect, the interpreter is a layer of software logic between your code and the computer hardware on your machine.

When the Python package is installed on your machine, it generates a number of components – Minimally, an interpreter and supporting library. Depending how you use it, the Python interpreter may take the form of an executable program, or a set of libraries linked into another program. Depending on which flavor of Python you run, the interpreter itself may be implemented in C program, a set of Java classes, or something else. Whatever form it takes, the Python code you write must always be run by this interpreter. And to enable that, you must install a Python interpreter on your computer.

Python itself may be fetched from the download page on its main website, http://www.python.org. It may also be found through various other distribution channels. Keep in mind that you should always check to see whether Python is already present before installing it. If you're on Windows 7 and earlier, you'll usually find Python in the Start menu. On Unix and Linux, Python probably lives in your /usr directory tree.

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

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

Programmer's View
In its simplest form, a Python program is just a text file containing Python statements. Fro example, the following file, named script0.py, contains two statements below:

```
print('hello world')
print(2 ** 100)
```

At the moment, don't worry about the syntax of this code yet, we are interested only in getting it to run.

You can create such a file of statements with any text editor. By convention, Python program files are given names that end in extension .py; technically, this naming scheme required only for files that are "imported" – a term clarified in the next module – but most Python files have .py names for consistency.

After you have typed these statements into a text file, you must tell Python to execute the file – which simply means to run all the statements in the file from top to bottom, one after another. As you'll see in the next module, you can launch Python program files by shell command lines, by clicking their icons, from within IDEs, and with other

standard techniques. If the statements in the file well declared without compilation error, you will see the results of the two print statements show up somewhere on your computer – by default, usually in the same window you were in when ran the program:

```
hello world
1267650600228229401496703205376
```

See Module 3 for the full story on this process, especially if you are new to programming; we will get into all the gory details of writing and launching programs there. For our purposes here, we have just run a Python script that prints a string and a number, just enough to capture the basics of program execution.

Python's View

Then brief description in the prior section is fairly standard for scripting languages, and it is usually all that most Python programmers need to know. You type code into text files, and run those files through the interpreter. Under the hood, though, a bit more happens when you tell Python to "go". Although knowledge of Python internals is not strictly required for Python programming, a basic understanding of the runtime structure of Python can help you grasp the bigger picture of program execution.

When you instruct Python to run your script, there are a few steps that Python carries before your code actually starts crunching away. Specially, it is first compiled to something call **byte code** and then routed to something call a **virtual machine**.

Byte code is a lower-level platform independent representation of your source code. Roughly, Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This byte code translation is performed to speed execution.

Prior paragraph said that this is almost completely hidden from you. If the Python process has write access on your machine, it will store the byte code of your programs in files that end with a .pyc extension. Prior to Python 3.2, you will see these files show up on your computer after you have run a few programs alongside the corresponding source code files – that is, in the same directories. For instance, you will notice a script.pyc after importing a script.py. In 3.2 and later, Python instead saves its .pyc byte code files in a subdirectory named __pycache__ located in the directory where your source files reside, and in files whose names identify the Python version that created them. The name __pycache__ subdirectory helps to avoid clutter, and the new naming convention for byte code files prevents different Python versions installed on the same computer from overwriting each other's saved byte code. We will study these byte code file models in more details in Module 22, though there are automatic and irrelevant to most Python programs, and are free to vary among the alternatives Python implementations described ahead.

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

- **Source Changes:** Python automatically checks the last-modified timestamps of source and byte code files to know when it must recompile.
- **Python Versions:** Imports also checks to see if the file must be recompiled because it was created by different Python version, using either a version number in the bye code file itself in 3.2 and earlier, or the information present in byte code filenames in 3.2 and later.

The result is that both source code changes and differing Python version numbers will trigger a new byte code file. If Python cannot write the bye code files to your machine, your program still works – the byte code is generated in memory and simply discarded on program exit. However, because .pyc files speed startup time, you will want to make sure they are written for large programs. Byte code files are also one way to ship Python programs – Python is happy to run a program if all it can find are .pyc files, even if the

original .py source file are absent. With this arrangement, the developer can protect their coding logic from the program users.

Finally, keep in mind that byte code is saved in files only for files that are imported, not for the top-level files of a program that are only run as scripts (strictly speaking, it is an import optimization). We will explore import basics in Module 3, and take a deeper look at imports later. Moreover, a given file is only imported (and possibly compiled) once per program run, and byte code is also never saved for code typed at the interactive prompt.

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

The PVM sound more impressive than it is; really, it is not a separate program, and it need not be installed by itself. In fact, the PVM is just a big code loop that iterates through your byte code instructions, on by one, to carry out their operations. The PVM is the runtime engine of Python; It is always present as part of the Python system, and it is the component that truly runs your scripts. Technically, it is just the last step of what is called the **Python Interpreter**.

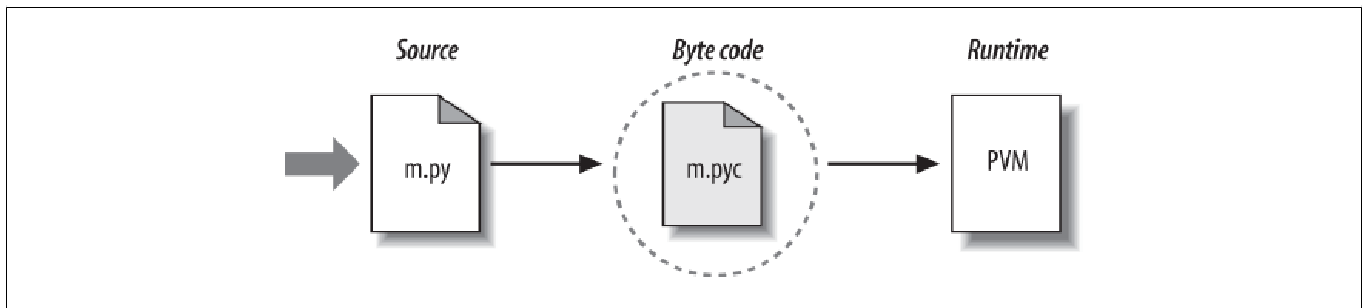


Figure above illustrates the runtime structure described here. Keep in mind that all of this complexity is deliberately hidden from Python programmers. Byte code compilation is automatic, and the PVM is just part of the Python system that you have installed on your machine. Again, programmers simply code and run files of statements, and Python handles the logistic of running them.

Developers with a background in fully compiled language such as C and C++ might notice a few differences in Python model. For one thing, there is usually no build or make step in Python work: code runs immediately after it is written. For another, Python byte code is not machine native code (e.g., instructions for an Intel or ARM chip). Bye code is a Python-specific representation.

Technically, Python is considered as 4 GL, this is why some Python code may not run as fast as native codes compiled by C or C++ programs. On the other hand, unlike classic interpreter, there is still an internal compile step – Python does not need to reanalyze and reparse each source statement's text repeatedly. The net effect is that pure Python code runs at speed somewhere between those of traditional compile language and a traditional interpreted language.

Another ramification of Python's execution model is that there is really no distinction between the development and execution environment. That is, the systems that compile and execute your source code are really one and the same. This similarity may have a bit more significance to developers with background in traditional compiled languages, but in Python, the compiler is always present at runtime and is part of the system that runs programs.

This makes for a much more rapid development cycle. There is no need to precompile and link before execution may begin; simply type and run the code. This also adds a much more dynamic flavor to the language – it is possible, and often very convenient, for Python programs to construct and execute other Python programs at runtime. The **eval** and **exec** built-ins, for instance, accept and run strings containing Python program code. This structure is also why Python lends itself to product customization – because Python code can be changed on the fly, user can modify the Python parts of the system onsite without needing to have or compile the entire system's code.

At a more fundamental level, keep in mind that all we really have in Python in runtime – there is no initial compile-time phase at all, and everything happens as the program is running. This even includes operations such as the creation of functions and classes and the linkage of modules. Such events occur before execution in more static languages, but happen as programs execute in Python. As we will see, this makes for a much more dynamic programming experience than that to which developers may be accustomed.

Execution model variations

Now that we have studied the internal execution flow described in the prior section, take note that it reflects the standard implementation of Python today but is not really a requirement of the Python language itself. Because of that, the execution model is prone to changing with time. Before moving on, let's briefly explore the most prominent of these variations.

Python Implementation Alternatives

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

In brief, CPython is the standard implementation, and the system that most developers will wish to use (if you are not sure, this probably includes you). This is also the version used in this training, though the core Python language presented here is almost entirely the same in the alternatives. All the other Python implementations have specific purposes and roles, though they can often serve in most of CPython's capacities too. All implement the same Python language but execute programs in different ways.

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

Module 3 –How You Run Programs

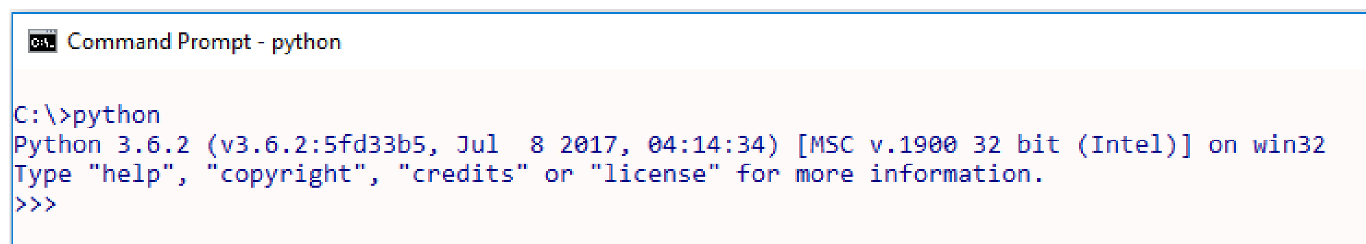
Now you have a handle on the program execution model, you are finally ready to start some real Python programming. At this point, you are assumed to have Python installed on your computer; if you don't, see the start of the prior module and Appendix A for installation and configurations hints on various platforms. Our goal here is to learn how to run Python program code.

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

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

The Interactive Prompts
This section gets us starts with interactive coding basic. Because it is our first look at running code, we also cover some preliminaries here, such as setting up a working directory and the system path, so be sure to read this section first if you are relatively new to programming. This section also explains some conventions used throughout the training, so most developers should probably take at least a quick look here.

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



To gets you out of this session, type Ctrl-Z on Windows, and Ctrl-D on Unix.

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

On most platforms, you can start the interactive prompt in additional ways that don't require typing a command. For instance, on Windows 7 and earlier, you can also begin similar interactive session by starting the IDLE GUI, or by selecting the "Python (Command line)" menu option from the Start button menu for Python. Both spawn a Python interactive prompt with the same functionality obtained with a python prompt.

Anytime you see the >>> prompt, you are in an interactive Python interpreter session – you can type any Python statement or expression here and run it immediately.

When we typed ***python*** in the last section to start an interactive session, we relied on the fact that the system located the Python program for us on its program search path. Depending on your Python version and platform, if you not set your system's PATH environment variable to include Python's install directory, you may need to replace the word ***python*** with the full path to the Python executable on your machine. Or you first switch to the executable directory before running the prompt. But you will probably want to set your PATH eventually, so a simple ***python*** can turn the session on.

Running Code Interactively

Under Python interactive session, you will see the >>> prompt. Now the interpreter is waiting for your statements or expression.

When working interactively, the result of your code are displayed below the >>> input line after you press the Enter key. For instance here are the results of two Python print statements:

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

There it is – we have just run some Python cod3. Don't worry about the details of the print statements shown here yet; we will start digging into syntax in other modules. In short, they print a Python string and an integer.

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

```
>>> greeting = 'Hello World'
>>> greeting
'Hello World'
>>> 2**8
256
>>>
```

There it is – we have just run some Python cod3. Don't worry about the details of the print statements shown here yet; we will start digging into syntax in other modules. In short, they print a Python string and an integer.

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

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

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

More generally, the interactive prompt is a place to test program components, regardless of their source – you can import and test functions and classes in your Python files, type calls and linked-in C functions, exercise Java classes under Jython, and more. Partly because of its interactive nature, Python supports an experimental and exploratory programming style you will find convenient when getting started. As a new developer, this interactive prompt will be very handy for you to learn the basic of the language such as operators without involve preparing codes in files.

Although the interactive prompt is simple to use, there are a few tips that beginners should keep in mind. Following are lists of common mistakes and also might spare you from a few headaches if you read them up front:

- **Type Python command only** – First of all, remember that you can only type Python code at Python's >>> prompt, not system commands.
- **Print statements are required only in files**. Because the interactive interpreter automatically prints the results of executions, you do not need to type complete print statements interactively. This is a nice feature, but it tends to confuse users when they on to writing code in files: within a code file, you must use print statements to see your output because expression results are not automatically echoed.
- **Don't indent at the interactive prompt (yet).** When typing Python programs, either interactively or into a text file, be sure to start all your un-nested statements in column 1 (that is, all the way to the left). If you don't, Python may print a ***SyntaxError*** message, because blank space to the left of your code is taken to be indentation that groups nested statements.
- **Watch out for prompt changes for compound (multi-lines) statements.** You should know that when typing lines 2 and beyond of a compound statement interactively, the prompt may change. In the simple shell window interface, the interactive prompt changes to … instead of >>> for lines 2 and beyond; in the IDLE GUI interface, lines after the firt are instead automatically indented.

  If you happened to come across a … prompt or a blank line when entering your code, it probably means that you have somehow confused interactive Python into thinking you are typing a multiline statement. Try hitting the Enter key or a Ctrl-C combination to get back to the main prompt.
- **Terminate compound statements at the interactive prompt with a blank line.** At the interactive prompt, inserting a blank line is necessary to tell interactive Python that you are done typing the multiline statement. That is, you must press Enter twice to make a compound statement run. By contrast, blank lines are not required in files and are simply ignored if present. If you don't press Enter twice at the end of a compound statement when working interactively, you will appear to be stack in a limbo state, because the interactive interpreter will do nothing at all – It is waiting for you to press Enter again!
- **The interactive prompt runs one statement at a time.** At the interactive prompt, you must run one statement to completion before typing another. This is natural for simple statements, because pressing the Enter key runs the statement entered. For compound statements, though, remember that you must submit a blank line to terminate the statement and make it run before you can type the next statement.

System Command Lines and Files

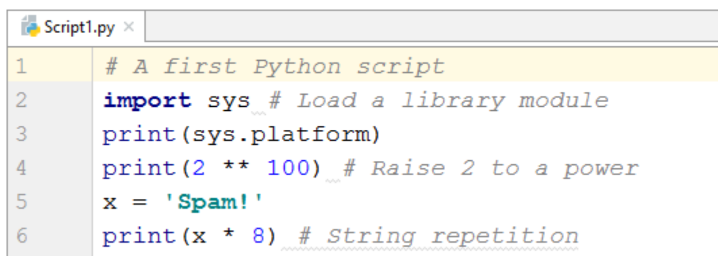
Although the interactive prompt is great for experimenting and testing, it has one big disadvantage: programs you type there go away as soon as the Python interpreter executes them. Because the code you type interactively is never stored in a file, you can't run it again without retyping if from scratch.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

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

Terminology in this domain can vary somewhat. For instance, module files are often referred to as programs in Python—that is, a program is considered to be a series of pre-coded statements stored in a file for repeated execution. Module files that are run directly are also sometimes called scripts—an informal term usually meaning a top-level program file. Some reserve the term “module” for a file imported from another file, and “script” for the main file of a program; we generally will here, too (though you’ll have to stay tuned for more on the meaning of “top-level,” imports, and main files later in this module).
Whatever you call them, the next few sections explore ways to run code typed into module files. In this section, you’ll learn how to run files in the most basic way: by listing their names in a python command line entered at your computer’s system prompt. Though it might seem primitive to some—and can often be avoided altogether by using a GUI like IDLE, discussed later—for many programmers a system shell command-line window, together with a text editor window, constitutes as much of an integrated development environment as they will ever need, and provides more direct control over programs.

A First Script

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



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

- Imports a Python module (libraries of additional tools), to fetch the name of the platform
- Runs three print function calls, to display the script’s results
- Uses a variable named x, created when it’s assigned, to hold onto a string object
- Applies various object operations

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

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

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

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

Running Files with Command Lines

Once you've saved this text file, you can ask Python to run it by listing its full filename as the first argument to a python command like the following typed at the system shell prompt (don't type this at Python's interactive prompt, and read on to the next paragraph if this doesn't work right away for you):

```
Administrator: Command Prompt

C:\Users\Administrator\PycharmProjects\untitled>python script1.py
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!

C:\Users\Administrator\PycharmProjects\untitled>
```

Again, you can type such a system shell command in whatever your system provides for command-line entry—a Windows Command Prompt window, an xterm window, or similar. But be sure to run this in the same working directory where you've saved your script file ("cd" there first if needed), and be sure to run this at the system prompt, not Python's ">>>" prompt. Also remember to replace the command's word "python" with a full directory path as we did before if your PATH setting is not configured, though this isn't required for the "py" Windows launcher program, and may not be required in 3.3 and later.

Another note to beginners: do not type any of the preceding text in the script1.py source file you created in the prior section. This text is a system command and program output, not program code. The first line here is the shell command used to run the source file, and the lines following it are the results produced by the source file's print statements. And again, remember that the system prompt for UNIX is %—don't type it yourself (not to nag, but it's a remarkably common early mistake).

If all works as planned, this shell command makes Python run the code in this file line by line, and you will see the output of the script's three print statements—the name of the underlying platform as known Python, 2 raised to the power 100, and the result of the same string repetition expression we saw earlier

If all didn't work as planned, you'll get an error message—make sure you've entered the code in your file exactly as shown, and try again. The next section has additional options and pointers on this process. And if all else fails, you might also try running under the IDLE GUI discussed ahead—a tool that sugarcoats some launching details, though sometimes at the expense of the more explicit control you have when using command lines.

Command-Line Usage Variations

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

```
C:\Users\Administrator\PycharmProjects\untitled>python script1.py > saveit.txt
```

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

Alternatively, if you're using the Windows launcher new in Python 3.3 (described earlier), a py command will have the same effect, but does not require a directory path or PATH settings, and allows you to specify Python version numbers on the command line too:

```
Select Administrator: Command Prompt

C:\Users\Administrator\PycharmProjects\untitled>py -3 script1.py
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

On all recent versions of Windows, you can also type just the name of your script, and omit the name of Python itself. Because newer Windows systems use the Windows Registry (a.k.a. filename associations) to find a program with which to run a file, you don't need to name "python" or "py" on the command line explicitly to run a .py file. The prior command, for example, could be simplified to the following on most Windows machines, and will automatically be run by python prior to 3.3, and by py in 3.3 and later—just as though you had clicked on the file's icon in Explorer (more on this option ahead):

```
Administrator: Command Prompt

C:\Users\Administrator\PycharmProjects\untitled>script1.py
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

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

```
C:\code> cd D:\other
D:\other> python c:\code\script1.py
```

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

```
D:\other> C:\Python33\python c:\code\script1.py
```

Unix-Style Executable Scripts: #!

Our next launching technique is really a specialized form of the prior, which, despite this section's title, can apply to program files run on both Unix and Windows today. Since it has its roots on Unix, let's begin this story there.

Unix Script Basics

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

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

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

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

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

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

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

The Unix env Lookup Trick

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

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

When coded this way, the env program locates the Python interpreter according to your system search path settings (in most Unix shells, by looking in all the directories listed in your PATH environment variable). This scheme can be more portable, as you don't need to hardcode a Python install path in the first line of all your scripts. That way, if your scripts ever move to a new machine, or your Python ever moves to a new location, you must update just PATH, not all your scripts.

Provided you have access to env everywhere, your scripts will run no matter where Python lives on your system. In fact, this env form is generally recommended today over even something as generic as /usr/bin/python, because some platforms may install Python elsewhere. Of course, this assumes that env lives in the same place everywhere (on some machines, it may be in /sbin, /bin, or elsewhere); if not, all portability bets are off!

The Python 3.3 Windows Launcher: #! Comes to Windows

A note for Windows users running Python 3.2 and earlier: the method described here is a Unix trick, and it may not work on your platform. Not to worry; just use the basic command-line technique explored earlier. List the file's name on an explicit python command line:

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

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

If you're using Python 3.3 or later, though, or have its Windows launcher installed separately, it turns out that Unix-style #! lines do mean something on Windows too. Besides offering the py executable described earlier, the new Windows launcher mentioned earlier attempts to parse #! lines to determine which Python version to launch to run your script's code. Moreover, it allows you to give the version number in full or partial forms, and recognizes most common Unix patterns for this line, including the /usr/bin/env form.

The launcher's #! parsing mechanism is applied when you run scripts from command lines with the py program, and when you click Python file icons (in which case py is run implicitly by filename associations). Unlike Unix, you do not need to mark files with executable privileges for this to work on Windows, because filename associations achieve similar results.

For example, the first of the following is run by Python 3.X and the second by 2.X (without an explicit number, the launcher defaults to 2.X unless you set a PY_PYTHON environment variable):

```
c:\code> type robin3.py
#!/usr/bin/python3
print('Run', 'away!...') # 3.X function
c:\code> py robin3.py # Run file per #! line version
Run away!...
c:\code> type robin2.py
#!python2
print 'Run', 'away more!...' # 2.X statement
c:\code> py robin2.py # Run file per #! line version
Run away more!...
```

This works in addition to passing versions on command lines—we saw this briefly earlier for starting the interactive prompt, but it works the same when launching a script file:

```
c:\code> py −3.1 robin3.py # Run per command-line argument
Run away!...
```

The net effect is that the launcher allows Python versions to be specified on both a per-file and per-command basis, by using #! lines and command-line arguments, respectively. At least that's the very short version of the launcher's story.

Clicking File Icons

If you're not a fan of command lines, you can generally avoid them by launching Python scripts with file icon clicks, development GUIs, and other schemes that vary per platform. Let's take a quick look at the first of these alternatives here.

Icon-Click Basics

Icon clicks are supported on most platforms in one form or another. Here's a rundown of how these might be structured on your computer:

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

  Specifically, a clicked file will be run by one of two Python programs, depending on its extension and the Python you're running. In Pythons 3.2 and earlier, .py files are run by python.exe with a console (Command Prompt) window, and .pyw files are run by pythonw.exe files without a console. Byte code files are also run by these programs if clicked. In Python 3.3 and later (and where it's installed separately), the new Window's launchers' py.exe and pyw.exe programs serve the same roles, opening .py and .pyw files, respectively.
- **Non-Windows icon clicks -** On non-Windows systems, you will probably be able to perform a similar feat, but the icons, file explorer navigation schemes, and more may differ slightly. On Mac OS X, for instance, you might use Python Launcher in the MacPython (or Python N.M) folder of your Applications folder to run by clicking in Finder. On some Linux and other Unix systems, you may need to register the .py extension with your file explorer GUI, make your script executable using the #! line scheme of the preceding section, or associate the file MIME type with an application or command by editing files, installing programs, or using other tools. See your file explorer's documentation for more details.

In other words, icon clicks generally work as you'd expect for your platform, but be sure to see the platform usage documentation "Python Setup and Usage" in Python's standard manual set for more details as needed.

Module Imports and Reloads

In simple terms, every file of Python source code whose name ends in a .py extension is a module. No special code or syntax is required to make a file a module: any such file will do. Other files can access the items a module defines by importing that module—import operations essentially load another file and grant access to that file's contents. The contents of a module are made available to the outside world through its attributes.

This module-based services model turns out to be the core idea behind program architecture in Python. Larger programs usually take the form of multiple module files, which import tools from other module files. One of the modules is designated as the main or top-level file, or "script"—the file launched to start the entire program, which runs line by line as usual. Below this level, it's all modules importing modules.

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

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

```
C:\code> C:\python33\python
>>> import script1
win32
1267650600228229401496703205376
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

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

```
...Change script1.py in a text edit window to print 2 ** 16...
>>> import script1
>>> import script1
```

This is by design; imports are too expensive an operation to repeat more than once per file, per program run. Imports must find files, compile them to byte code, and run the code.

If you really want to force Python to run the file again in the same session without stopping and restarting the session, you need to instead call the reload function available in the imp standard library module (this function is also a simple built-in in Python 2.X, but not in 3.X):

```
>>> from imp import reload # Must load from module in 3.X (only)
>>> reload(script1)
win32
65536
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
<module 'script1' from '.\\script1.py'>
>>>
```

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

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

The reload function expects the name of an already loaded module object, so you have to have successfully imported a module once before you reload it (if the import reported an error, you can't yet reload and must import again). Notice that reload also expects parentheses around the module object name, whereas import does not. reload is a function that is called, and import is a statement.

That's why you must pass the module name to reload as an argument in parentheses, and that's why you get back an extra output line when reloading—the last output line is just the display representation of the reload call's return value, a Python module object. For now, when you hear "function," remember that parentheses are required to run a call.

The Grander Module Story: Attributes

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

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

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

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

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

You can access this module's title attribute in other components in two different ways. First, you can load the module as a whole with an import statement, and then qualify the module name with the attribute name to fetch it (note that we're letting the interpreter print automatically here):

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

In general, the dot expression syntax object.attribute lets you fetch any attribute attached to any object, and is one of the most common operations in Python code.

Here, we've used it to access the string variable title inside the module myfile—in other words, myfile.title. Alternatively, you can fetch (really, copy) names out of a module with from statements:

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

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

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

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

```
a = 'dead' # Define three attributes
b = 'parrot' # Exported to other files
c = 'sketch'
print(a, b, c) # Also used in this file (in 2.X: print a, b, c)
```

This file, threenames.py, assigns three variables, and so generates three attributes for the outside world. It also uses its own three variables in a 3.X print statement, as we see when we run this as a top-level file:

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

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

```
% python
>>> import threenames # Grab the whole module: it runs here
dead parrot sketch
>>>
>>> threenames.b, threenames.c # Access its attributes
('parrot', 'sketch')
>>>
>>> from threenames import a, b, c # Copy multiple names out
>>> b, c
('parrot', 'sketch')
```

The results here are printed in parentheses because they are really tuples—a kind of object created by the comma in the inputs (and covered in the next part of this course) —that you can safely ignore for now.

Once you start coding modules with multiple names like this, the built-in ***dir*** function starts to come in handy—you can use it to fetch a list of all the names available inside a module. The following returns a Python list of strings in square brackets :

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

The contents of this list have been edited here because they vary per Python version.

The point to notice here is that when the ***dir*** function is called with the name of an imported module in parentheses like this, it returns all the attributes inside that module.

Some of the names it returns are names you get "for free": names with leading and trailing double underscores (__X__) are built-in names that are always predefined by Python and have special meaning to the interpreter, but they aren't important at this point in this course. The variables our code defined by assignment—a, b, and c—show up last in the ***dir*** result.

Modules and namespaces

Module imports are a way to run files of code, but, as we'll expand on later in the course, modules are also the largest program structure in Python programs, and one of the first key concepts in the language.

As we've seen, Python programs are composed of multiple module files linked together by import statements, and each module file is a package of variables—that is, a namespace. Just as importantly, each module is a self-contained namespace: one module file cannot see the names defined in another file unless it explicitly imports that other file. Because of this, modules serve to minimize name collisions in your code—because each file is a self-contained namespace, the names in one file cannot clash with those in another, even if they are spelled the same way.

In fact, as you'll see, modules are one of a handful of ways that Python goes to great lengths to package your variables into compartments to avoid name clashes. We'll discuss modules and other namespace constructs—including local scopes defined by classes and functions—further later in the course. For now, modules will come in handy as a way to run your code many times without having to retype it, and will prevent your file's names from accidentally replacing each other.

Usage Notes: import and reload

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

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

In addition, you may run into trouble if you use modules in unusual ways at this point in the course. For instance, if you want to import a module file that is stored in a directory other than the one you're working in, you'll have learn about the module search path. For now, if you must import, try to keep all your files in the directory you are working in to avoid complications.

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

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

```
% python
>>> exec(open('script1.py').read())
win32
65536
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
...Change script1.py in a text edit window to print 2 ** 32...
>>> exec(open('script1.py').read())
win32
4294967296
Spam!Spam!Spam!Spam!Spam!Spam!Spam!Spam!
```

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

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

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

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

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

In short, IDLE is a desktop GUI that lets you edit, run, browse, and debug Python programs, all from a single interface. It runs portably on most Python platforms, including Microsoft Windows, X Windows (for Linux, Unix, and Unix-like platforms), and the Mac OS (both Classic and OS X). For many, IDLE represents an easy-to-use alternative to typing command lines, a less problem-prone alternative to clicking on icons, and a great way for newcomers to get started editing and running code. You'll sacrifice some control in the bargain, but this typically becomes important later in your Python career.

Because IDLE is just a Python script on the module search path in the standard library, you can also generally run it on any platform and from any directory by typing the following in a system command shell window (e.g., in a Command Prompt on Windows:

```
c:\code> python -m idlelib.idle # Run idle.py in a package on module path
```

Other IDEs

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

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

- **Komodo** - A full-featured development environment GUI for Python (and other languages), Komodo includes standard syntax coloring, text editing, debugging, and other features. In addition, Komodo offers many advanced features that IDLE don't, including project files, source-control integration, and regular-expression debugging. At this writing, Komodo is not free, but see the Web for its current status— it is available at http://www.activestate.com from ActiveState.
- **NetBeans IDE for Python** - NetBeans is a powerful open source development environment GUI with support for many advanced features for Python developers: code completion, automatic indentation and code colorization, editor hints, code folding, refactoring, debugging, code coverage and testing, projects, and more. It may be used to develop both CPython and Jython code. Like Eclipse, NetBeans requires installation steps beyond those of the included IDLE GUI, but it is seen by many as more than worth the effort. Search the Web for the latest information and links.
- **PythonWin** - PythonWin is a free Windows-only IDE for Python that ships as part of Active- State's ActivePython distribution (and may also be fetched separately from http://www.python.org resources). It is roughly like IDLE, with a handful of useful Windows-specific extensions added; for example, PythonWin has support for COM objects. Today, IDLE is probably more advanced than PythonWin (for instance, IDLE's dual-process architecture often prevents it from hanging). However, PythonWin still offers tools for Windows developers that IDLE does not. See http://www.activestate.com for more information.
- **Wing, Visual Studio, and others** - Other IDEs are popular among Python developers too, including the mostly commercial Wing IDE, Microsoft Visual Studio via a plug-in, and PyCharm, PyScripter, Pyshield, and Spyder—but I do not have space to do justice to them here, and more will undoubtedly appear over time. In fact, almost every programmer-friendly text editor has some sort of support for Python development these days, whether it be preinstalled or fetched separately. Emacs and Vim, for instance, have substantial Python support.

IDE choices are often subjective, so I encourage you to browse to find tools that fit your development style and goals. For more information, see the resources available at http://www.python.org or search the Web for "Python IDE" or similar. A search for "Python editors" today leads you to a wiki page that maintains information about dozens of IDE and text-editor options for Python programming.

Other Launch Option
At this point, we've seen how to run code typed interactively, and how to launch code saved in files in a variety of ways—system command lines, icon clicks, imports and execs, GUIs like IDLE, and more. That covers most of the techniques in common use, and enough to run the code you'll see in this course. There are additional ways to run Python code, though, most of which have special or narrow roles. For completeness and reference, the next few sections take a quick look at some of these.

Embedding Calls
In some specialized domains, Python code may be run automatically by an enclosing system. In such cases, we say that the Python programs are embedded in (i.e., run by) another program. The Python code itself may be entered into a text file, stored in a database, fetched from an HTML page, parsed from an XML document, and so on.

But from an operational perspective, another system—not you—may tell Python to run the code you've created. Such an embedded execution mode is commonly used to support end-user customization—a game program, for instance, might allow for play modifications by running user-accessible embedded Python code at strategic points in time. Users can modify this type of system by providing or changing Python code. Because Python code is interpreted, there is no need to recompile the entire system to incorporate the change
.
In this mode, the enclosing system that runs your code might be written in C, C++, or even Java when the Jython system is used. As an example, it's possible to create and run strings of Python code from a C program by calling functions in the Python runtime API (a set of services exported by the libraries created when Python is compiled on your machine):

```
#include <Python.h>
...
Py_Initialize(); // This is C, not Python
PyRun_SimpleString("x = 'brave ' + 'sir robin'"); // But it runs Python code
```

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

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

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

See the prior module for more on this option.

Text Editor Launch Options
Text Editor Launch Options as mentioned previously, although they're not full-blown IDE GUIs, most programmer-friendly text editors have support for editing, and possibly running, Python programs.

Such support may be built in or fetchable on the Web. For instance, if you are familiar with the Emacs text editor, you can do all your Python editing and launching from inside that text editor. See the text editor resources page at http://www.python.org/editors for more details, or search the Web for the phrase "Python editors."

Still Other Launch Options

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

Future Possibilities?

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

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

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

Which Option Should I Use?

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

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

Module 4 - Introducing to Python Object Types

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

Somewhat more formally, in Python, dat takes the form of objects – either buit-in objects that Python provides, or objects we crerate using Python classes or external language tools such as C extension library. Although we will firm up this definition later, obejcts are essentially just a pieces of memory, with values and sets of associated operations. As we will see, everything is an object in Python script. Even simple numbers qualify, with value (e.g., 99), and supported operations (addition, subtraction, and so on).

Because objects are also the most fundamental notion in Pyhon programming, we will start this module with a survey of Python’s built-in object types. Other modules provide a second pass that fills in details we will gloss over in this survey. Here, our goal is a brief tour to introduce the basics.

The Python Conceptual Hierarchy

Before we get to the code, let’s first extablish a clear picure of how this module fits into the overall Python picture. From a more concrete perspective, Python programs can be decomposed into modules, statements, expressions, and objects, as follows:

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

The previous module introduced the highest level of the this hierarchy. This and following few modules begin at the bottom – exploring both built-in objects and the expressions you can code to use them.

We will move on to study statements in the next part of the course, though we will find that they largely exist to manage the objects we will meet here. Moreover, by the time we reach classes in the OOP part of this course, we will discover that they allow us to define new object types of our own, by both using and emulating the object types we will explore here. Because of all this, built-in objects are mandatory point of embarkation for all Python journeys.

Why use built-in types?

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

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

- Built-in objects make programs easy to write. For simple tasks, built-in types are often all you need to represent the structure of problem domains. Because you get powerful tools such as collections (lists) and search tables (dictionaries) for free, you can use them immediately. You can get a lot of work done with Python’s builtin object types alone.
- Built-in objects are components of extensions. For more complex tasks, you may need to provide your own objects using Python classes or C language interfaces. But as you’ll see in later parts of this course, objects implemented manually are often built on top of built-in types such as lists and dictionaries. For instance, a stack data structure may be implemented as a class that manages or customizes a built-in list.
- Built-in objects are often more efficient than custom data structures. Python’s built-in types employ already optimized data structure algorithms that are implemented in C for speed. Although you can write similar

object types on your own, you'll usually be hard-pressed to get the level of performance built-in object types provide.

- Built-in objects are a standard part of the language. In some ways, Python borrows both from languages that rely on built-in tools (e.g., LISP) and languages that rely on the programmer to provide tool implementations or frameworks of their own (e.g., C++). Although you can implement unique object types in Python, you don't need to do so just to get started. Moreover, because Python's built-ins are standard, they're always the same; proprietary frameworks, on the other hand, tend to differ from site to site.

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

Python's Core Data Types

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

Table: *Built-in objects preview*

Object type	Example literals/creation
Numbers	1234, 3.1415, 3+4j, 0b111, Decimal(), Fraction()
Strings	'spam', "Bob's", b'a\x01c', u'sp\xc4m'
Lists	[1, [2, 'three'], 4.5], list(range(10))
Dictionaries	{'food': 'spam', 'taste': 'yum'}, dict(hours=10)
Tuples	(1, 'spam', 4, 'U'), tuple('spam'), namedtuple
Files	open('eggs.txt'), open(r'C:\ham.bin', 'wb')
Sets	set('abc'), {'a', 'b', 'c'}
Other core types	Booleans, types, None
Program unit types	Functions, modules, classes
Implementation-related types	Compiled code, stack tracebacks

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

Also shown in Table above, program units such as functions, modules, and classes are objects in Python too; they are created with statements and expressions such as def, class, import, and lambda and may be passed around scripts freely, stored within other objects, and so on. Python also provides a set of implementation-related types such as compiled code objects, which are generally of interest to tool builders more than application developers.

Despite its title, Table above isn't really complete, because everything we process in Python programs is a kind of object. For instance, when we perform text pattern matching in Python, we create pattern objects, and when we perform network scripting, we use socket objects. These other kinds of objects are generally created by importing and using functions in library modules—for example, in the re and socket modules for patterns and sockets—and have behavior all their own.

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

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

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

Just as importantly, once you create an object, you bind its operation set for all time—you can perform only string operations on a string and list operations on a list. In formal terms, this means that Python is dynamically typed, a model that keeps track of types for you automatically instead of requiring declaration code, but it is also strongly typed, a constraint that means you can perform on an object only operations that are valid for its type.

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

Numbers

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

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

A complete inventory of Python's numeric toolbox includes:

- Integer and floating-point objects
- Complex number objects
- Decimal: fixed-precision objects
- Fraction: rational number objects
- Sets: collections with numeric operations
- Booleans: true and false
- Built-in functions and modules: round, math, random, etc
- Expressions; unlimited integer precision; bitwise operations; hex, octal, and binary formats
- Third-party extensions: vectors, libraries, visualization, plotting, etc.

Because the types in this list's first bullet item tend to see the most action in Python code, this module starts with basic numbers and fundamentals, then moves on to explore the other types on this list, which serve specialized roles. We'll also study sets here, which have both numeric and collection qualities, but are generally considered more the former than the latter. Before we jump into code, though, the next few sections get us started with a brief overview of how we write and process numbers in our scripts.

Among its basic types, Python provides integers, which are positive and negative whole numbers, and floating-point numbers, which are numbers with a fractional part (sometimes called "floats" for verbal economy). Python also allows us to write integers using hexadecimal, octal, and binary literals; offers a complex number type; and allows integers to have unlimited precision—they can grow to have as many digits as your memory space allows. Table below shows what Python's numeric types look like when written out in a program as literals or constructor function calls.

Table: *Numeric literals and constructors*

Literal	Interpretation

1234, −24, 0, 99999999999999	Integers (unlimited size)
1.23, 1., 3.14e-10, 4E210, 4.0e+210	Floating-point numbers
0o177, 0x9ff, 0b101010	Octal, hex, and binary literals in 3.X
0177, 0o177, 0x9ff, 0b101010	Octal, octal, hex, and binary literals in 2.X
3+4j, 3.0+4.0j, 3J	Complex number literals
set('spam'), {1, 2, 3, 4}	Sets: 2.X and 3.X construction forms
Decimal('1.0'), Fraction(1, 3)	Decimal and fraction extension types
bool(X), True, False	Boolean type and constants

Built-in Numeric Tools
Python provides a set of tools for processing number objects:

Expression operators
+, -, *, /, >>, **, &, etc.
Built-in mathematical functions
pow, abs, round, int, hex, bin, etc.
Utility modules
random, math, etc.

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

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

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

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

Many operator expressions available in Python are self-explanatory; for instance, the usual mathematical operators (+, −, *, /, and so on) are supported.

A few will be familiar if you've used other languages in the past: % computes a division remainder, << performs a bitwise left-shift, & computes a bitwise AND result, and so on. Others are more Python-specific, and not all are numeric in nature: for example, the is operator tests object identity (i.e., address in memory, a strict form of equality), and lambda creates unnamed functions.

Table: *Python expression operators and precedence*

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

Numbers in action

On to the code! Probably the best way to understand numeric objects and expressions is to see them in action, so with those basics in hand let's start up the interactive command line and try some simple but illustrative operations.

Variables and Basic Expressions
First of all, let's exercise some basic math. In the following interaction, we first assign two variables (a and b) to integers so we can use them later in a larger expression.

Variables are simply names—created by you or Python—that are used to keep track of information in your program. We'll say more about this in the next module, but in
Python:

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

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

```
% python
>>> a = 3        # Name created: not declared ahead of time
>>> b = 4
```

We also used a comment here. Recall that in Python code, text after a # mark and continuing to the end of the line is considered to be a comment and is ignored b Python. Comments are a way to write human-readable documentation for your code, and an important part of programming. I've added them to most of this course's examples to help explain the code. In the next part of the course, we'll meet a related but more functional feature—documentation strings—that attaches the text of your comments to objects so it's available after your code is loaded.

Because code you type interactively is temporary, though, you won't normally write comments in this context. If you're working along, this means you don't need to type any of the comment text from the # through to the end of the line; it's not a required part of the statements we're running this way.

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

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

Technically, the results being echoed back here are tuples of two values because the lines typed at the prompt contain two expressions separated by commas; that's why the results are displayed in parentheses (more on tuples later). Note that the expressions work because the variables a and b within them have been assigned values. If you use a different variable that has not yet been assigned, Python reports an error rather than filling in some default value:

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

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

Here are two slightly larger expressions to illustrate operator grouping and more about conversions, and preview a difference in the division operator in Python 3.X and 2.X:

```
>>> b / 2 + a # Same as ((4 / 2) + 3) [use 2.0 in 2.X]
5.0
>>> b / (2.0 + a) # Same as (4 / (2.0 + 3)) [use print before 2.7]
0.8
```

In the first expression, there are no parentheses, so Python automatically groups the components according to its precedence rules—because / is higher precedence than +, it binds more tightly and so is evaluated first. The result is as if the expression had been organized with parentheses as shown in the comment to the right of the code.

Also, notice that all the numbers are integers in the first expression. Because of that, Python 2.X's / performs integer division and addition and will give a result of 5, whereas Python 3.X's / performs true division, which always retains fractional remainders and gives the result 5.0 shown. If you want 2.X's integer division in 3.X, code this as b //2 + a; if you want 3.X's true division in 2.X, code this as b / 2.0 + a (more on division in a moment).

In the second expression, parentheses are added around the + part to force Python to evaluate it first (i.e., before the /). We also made one of the operands floating point by adding a decimal point: 2.0. Because of the mixed types, Python converts the integer referenced by a to a floating-point value (3.0) before performing the +. If instead all the numbers in this expression were integers, integer division (4 / 5) would yield the truncated integer 0 in Python 2.X but the floating point 0.8 shown in Python 3.X. Again, stay tuned for formal division details.

Numeric Display Formats

If you're using Python 2.6, Python 3.0, or earlier, the result of the last of the preceding examples may look a bit odd the first time you see it:

```
>>> b / (2.0 + a)           # Pythons <= 2.6: echoes give more (or fewer) digits
0.80000000000000004
>>> print(b / (2.0 + a)) # But print rounds off digits
0.8
```

We met this phenomenon briefly in the prior module, and it's not present in Pythons 2.7, 3.1, and later. The full story behind this odd result has to do with the limitations of floating-point hardware and its inability to exactly represent some values in a limited number of bits. Because computer architecture is well beyond this course's scope, though, we'll finesse this by saying that your computer's floating-point hardware is doing the best it can, and neither it nor Python is in error here.

In fact, this is really just a display issue—the interactive prompt's automatic result echo shows more digits than the print statement here only because it uses a different algorithm.

It's the same number in memory. If you don't want to see all the digits, use print. As of 2.7 and 3.1, Python's floating-point display logic tries to be more intelligent, usually showing fewer decimal digits, but occasionally more.

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

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

and that there are more ways to display the bits of a number inside your computer than using print and automatic echoes (the following are all run in Python 3.3, and mayvary slightly in older versions):

```
>>> num = 1 / 3.0
>>> num                    # Auto-echoes
0.3333333333333333
>>> print(num)             # Print explicitly
0.3333333333333333
>>> '%e' % num             # String formatting expression
'3.333333e-01'
>>> '%4.2f' % num          # Alternative floating-point format
'0.33'
>>> '{0:4.2f}'.format(num) # String formatting method: Python 2.6, 3.0, and later
'0.33'
```

The last three of these expressions employ string formatting, a tool that allows for format flexibility, which we will explore in the later on strings. Its results are strings that are typically printed to displays or reports.

Comparisons: Normal and Chained

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

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

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

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

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

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

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

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

```
>>> X < Y > Z
False
>>> X < Y and Y > Z
False
>>> 1 < 2 < 3.0 < 4
True
>>> 1 > 2 > 3.0 > 4
False
```

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

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

Python does not compare the 1 == 2 expression's False result to 3—this would technically mean the same as 0 < 3, which would be True (True and False are just customized 1 and 0).

One last note here before we move on: chaining aside, numeric comparisons are based on magnitudes, which are generally simple—though floating-point numbers may not always work as you'd expect, and may require conversions or other massaging to be compared meaningfully:

```
>>> 1.1 + 2.2 == 3.3              # Shouldn't this be True?...
False
>>> 1.1 + 2.2                     # Close to 3.3, but not exactly: limited precision
3.3000000000000003
>>> int(1.1 + 2.2) == int(3.3)    # OK if convert: see also round, floor, trunc ahead
True                              # Decimals and fractions (ahead) may help here too
```

This stems from the fact that floating-point numbers cannot represent some values exactly due to their limited number of bits—a fundamental issue in numeric programming not unique to Python, which we'll learn more about later when we meet decimals and fractions, tools that can address such limitations. First, though, let's continue our tour of Python's core numeric operations, with a deeper look at division.

Division: Classic, Floor, and True

You've seen how division works in the previous sections, so you should know that it behaves slightly differently in Python 3.X and 2.X. In fact, there are actually three flavors of division, and two different division operators, one of which changes in 3.X. This story gets a bit detailed, but it's another major change in 3.X and can break 2.X code, so let's get the division operator facts straight:

- **X / Y** - Classic and true division. In Python 2.X, this operator performs classic division, truncating results for integers, and keeping remainders (i.e., fractional parts) for floating-point numbers. In Python 3.X, it performs true division, always keepingremainders in floating-point results, regardless of types.
- **X // Y -** Floor division. Added in Python 2.2 and available in both Python 2.X and 3.X, this operator always truncates fractional remainders down to their floor, regardless of types. Its result type depends on the types of its operands.

True division was added to address the fact that the results of the original classic division model are dependent on operand types, and so can be difficult to anticipate in a dynamically typed language like Python. Classic division was removed in 3.X because of this constraint—the / and // operators implement true and floor division in 3.X. Python2.X defaults to classic and floor division, but you can enable true division as an option.

In sum:

- In 3.X, the / now always performs true division, returning a float result that includes any remainder, regardless of operand types. The // performs floor division, which truncates the remainder and returns an integer for integer operands or a float if anyoperand is a float.
- In 2.X, the / does classic division, performing truncating integer division if both operands are integers and float division (keeping remainders) otherwise. The // does floor division and works as it does in 3.X, performing truncating division for integers and floor division for floats.

Here are the two operators at work in 3.X and 2.X—the first operation in each set is the crucial difference between the lines that may impact code:

```
% python
>>>
>>> 10 / 4         # Differs in 3.X: keeps remainder
2.5
>>> 10 / 4.0       # Same in 3.X: keeps remainder
2.5
>>> 10 // 4        # Same in 3.X: truncates remainder
2
>>> 10 // 4.0       # Same in 3.X: truncates to floor
2.0

C:\code> C:\Python27\python
>>>
>>> 10 / 4         # This might break on porting to 3.X!
2
>>> 10 / 4.0
2.5
>>> 10 // 4        # Use this in 2.X if truncation needed
2
>>> 10 // 4.0
2.0
```

Notice that the data type of the result for // is still dependent on the operand types in 3.X: if either is a float, the result is a float; otherwise, it is an integer. Although this may seem similar to the type-dependent behavior of / in 2.X that motivated its change in 3.X, the type of the return value is much less critical than differences in the return value itself.

Moreover, because // was provided in part as a compatibility tool for programs that rely on truncating integer division (and this is more common than you might expect), it must return integers for integers. Using // instead of / in 2.X when integer truncation is required helps make code 3.X-compatible.

Supporting either Python

Although / behavior differs in 2.X and 3.X, you can still support both versions in your code. If your programs depend on truncating integer division, use // in both 2.X and 3.X as just mentioned. If your programs require floating-point results with remainders for integers, use float to guarantee that one operand is a float around a / when run in 2.X:

```
X = Y // Z              # Always truncates, always an int result for ints in 2.X and 3.X
X = Y / float(Z)        # Guarantees float division with remainder in either 2.X or 3.X
```

Alternatively, you can enable 3.X / division in 2.X with a __future__ import, rather than forcing it with float conversions:

```
C:\code> C:\Python27\python
>>> from __future__ import division      # Enable 3.X "/" behavior
```

```
>>> 10 / 4
2.5
>>> 10 // 4                                    # Integer // is the same in both
2
```

This special from statement applies to the rest of your session when typed interactively like this, and must appear as the first executable line when used in a script file (and alas, we can import from the future in Python, but not the past; insert something about talking to “the Doc” here...).

Floor versus truncation

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

```
>>> import math
>>> math.floor(2.5)      # Closest number below value
2
>>> math.floor(-2.5)
-3
>>> math.trunc(2.5)      # Truncate fractional part (toward zero)
2
>>> math.trunc(-2.5)
-2
```

When running division operators, you only really truncate for positive results, since truncation is the same as floor; for negatives, it’s a floor result (really, they are both floor, but floor is the same as truncation for positives). Here’s the case for 3.X:

```
C:\code> c:\python33\python
>>> 5 / 2, 5 / −2
(2.5, −2.5)
>>> 5 // 2, 5 // −2                # Truncates to floor: rounds to first lower integer
(2, −3)                            # 2.5 becomes 2, −2.5 becomes −3
>>> 5 / 2.0, 5 / −2.0
(2.5, −2.5)
>>> 5 // 2.0, 5 // −2.0            # Ditto for floats, though result is float too
(2.0, −3.0)
```

The 2.X case is similar, but / results differ again:

```
C:code> c:\python27\python
>>> 5 / 2, 5 / −2          # Differs in 3.X
(2, −3)
>>> 5 // 2, 5 // −2        # This and the rest are the same in 2.X and 3.X
(2, −3)
>>> 5 / 2.0, 5 / −2.0
(2.5, −2.5)
>>> 5 // 2.0, 5 // −2.0
(2.0, −3.0)
```

If you really want truncation toward zero regardless of sign, you can always run a float division result through math.trunc, regardless of Python version (also see the round built-in for related functionality, and the int built-in, which has the same effect here but requires no import):

```
C:\code> c:\python33\python
>>> import math
>>> 5 / −2                  # Keep remainder
−2.5
>>> 5 // −2                 # Floor below result
-3
>>> math.trunc(5 / −2)  # Truncate instead of floor (same as int())
−2
C:\code> c:\python27\python
>>> import math
>>> 5 / float(−2)                # Remainder in 2.X
−2.5
>>> 5 / −2, 5 // −2              # Floor in 2.X
(−3, −3)
>>> math.trunc(5 / float(−2))      # Truncate in 2.X
−2
```

Why does truncation matter?
As a wrap-up, if you are using 3.X, here is the short story on division operators for reference:

```
>>> (5 / 2), (5 / 2.0), (5 / −2.0), (5 / −2)            # 3.X true division
(2.5, 2.5, −2.5, −2.5)
>>> (5 // 2), (5 // 2.0), (5 // −2.0), (5 // −2)        # 3.X floor division
(2, 2.0, −3.0, −3)
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)            # Both
(3.0, 3.0, 3, 3.0)
```

For 2.X developers, division works as follows (the three bold outputs of integer division differ from 3.X):

```
>>> (5 / 2), (5 / 2.0), (5 / −2.0), (5 / −2)            # 2.X classic division (differs)
(2, 2.5, −2.5, −3)
>>> (5 // 2), (5 // 2.0), (5 // −2.0), (5 // −2)        # 2.X floor division (same)
(2, 2.0, −3.0, −3)
>>> (9 / 3), (9.0 / 3), (9 // 3), (9 // 3.0)            # Both
(3, 3.0, 3, 3.0)
```

It's possible that the nontruncating behavior of / in 3.X may break a significant number of 2.X programs. Perhaps because of a C language legacy, many programmers rely on division truncation for integers and will have to learn to use // in such contexts instead.

You should do so in all new 2.X and 3.X code you write today—in the former for 3.X compatibility, and in the latter because / does not truncate in 3.X.

Integer Precision
Division may differ slightly across Python releases, but it's still fairly standard. Here's something a bit more exotic. As mentioned earlier, Python 3.X integers support unlimited size:

```
>>> 999999999999999999999999999999 + 1 # 3.X
1000000000000000000000000000000
```

Python 2.X has a separate type for long integers, but it automatically converts any number too large to store in a normal integer to this type. Hence, you don't need to code any special syntax to use longs, and the only way you can tell that you're using 2.X longs is that they print with a trailing "L":

```
>>> 999999999999999999999999999999 + 1 # 2.X
1000000000000000000000000000000L
```

Unlimited-precision integers are a convenient built-in tool. For instance, you can use them to count the U.S. national debt in pennies in Python directly (if you are so inclined, and have enough memory on your computer for this year's budget). Here are the 3.X and 2.X cases:

```
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376
>>> 2 ** 200
1606938044258990275541962092341162602522202993782792835301376L
```

Because Python must do extra work to support their extended precision, integer math is usually substantially slower than normal when numbers grow large. However, if you need the precision, the fact that it's built in for you to use will likely outweigh its performance penalty.

Complex Numbers

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

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

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

Complex numbers also allow us to extract their parts as attributes, support all the usual mathematical expressions, and may be processed with tools in the standard cmath module (the complex version of the standard math module). Because complex numbers are rare in most programming domains, though, we'll skip the rest of this story here. Check Python's language reference manual for additional details.

Hex, Octal, Binary: Literals and Conversions

Python integers can be coded in hexadecimal, octal, and binary notation, in addition to the normal base-10 decimal coding we've been using so far. The first three of these may at first seem foreign to 10-fingered beings, but some programmers find them convenient alternatives for specifying values, especially when their mapping to bytes and bits is important. The coding rules were introduced briefly at the start of this module; let's look at some live examples here.

Keep in mind that these literals are simply an alternative syntax for specifying the value of an integer object. For example, the following literals coded in Python 3.X or 2.X produce normal integers with the specified values in all three bases. In memory, an integer's value is the same, regardless of the base we use to specify it:

```
>>> 0o1, 0o20, 0o377            # Octal literals: base 8, digits 0-7 (3.X, 2.6+)
(1, 16, 255)
>>> 0x01, 0x10, 0xFF            # Hex literals: base 16, digits 0-9/A-F (3.X, 2.X)
```

```
(1, 16, 255)
>>> 0b1, 0b10000, 0b11111111 # Binary literals: base 2, digits 0-1 (3.X, 2.6+)
(1, 16, 255)
```

Here, the octal value 0o377, the hex value 0xFF, and the binary value 0b11111111 are all decimal 255. The F digits in the hex value, for example, each mean 15 in decimal and a 4-bit 1111 in binary, and reflect powers of 16. Thus, the hex value 0xFF and others convert to decimal values as follows:

```
>>> 0xFF, (15 * (16 ** 1)) + (15 * (16 ** 0)) # How hex/binary map to decimal
(255, 255)
>>> 0x2F, (2 * (16 ** 1)) + (15 * (16 ** 0))
(47, 47)
>>> 0xF, 0b1111, (1*(2**3) + 1*(2**2) + 1*(2**1) + 1*(2**0))
(15, 15, 15)
```

Python prints integer values in decimal (base 10) by default but provides built-in functions that allow you to convert integers to other bases' digit strings, in Python-literal form—useful when programs or users expect to see values in a given base:

```
>>> oct(64), hex(64), bin(64)                # Numbers=>digit strings
('0o100', '0x40', '0b1000000')
```

The oct function converts decimal to octal, hex to hexadecimal, and bin to binary. To go the other way, the built-in int function converts a string of digits to an integer, and an optional second argument lets you specify the numeric base—useful for numbers read from files as strings instead of coded in scripts:

```
>>> 64, 0o100, 0x40, 0b1000000                    # Digits=>numbers in scripts and strings
(64, 64, 64, 64)
>>> int('64'), int('100', 8), int('40', 16), int('1000000', 2)
(64, 64, 64, 64)
>>> int('0x40', 16), int('0b1000000', 2)          # Literal forms supported too
(64, 64)
```

The eval function, which you'll meet later in this course, treats strings as though they were Python code. Therefore, it has a similar effect, but usually runs more slowly—it actually compiles and runs the string as a piece of a program, and it assumes the string being run comes from a trusted source—a clever user might be able to submit a string that deletes files on your machine, so be careful with this call:

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

Finally, you can also convert integers to base-specific strings with string formatting method calls and expressions, which return just digits, not Python literal strings:

```
>>> '{0:o}, {1:x}, {2:b}'.format(64, 64, 64)       # Numbers=>digits, 2.6+
'100, 40, 1000000'
>>> '%o, %x, %x, %X' % (64, 64, 255, 255)          # Similar, in all Pythons
'100, 40, ff, FF'
```

Two notes before moving on. First, per the start of this module, Python 2.X users should remember that you can code octals with simply a leading zero, the original octal format in Python:

```
>>> 0o1, 0o20, 0o377              # New octal format in 2.6+ (same as 3.X)
(1, 16, 255)
>>> 01, 020, 0377                 # Old octal literals in all 2.X (error in 3.X)
```

```
(1, 16, 255)
```

In 3.X, the syntax in the second of these examples generates an error. Even though it's not an error in 2.X, be careful not to begin a string of digits with a leading zero unless you really mean to code an octal value. Python 2.X will treat it as base 8, which may not work as you'd expect—010 is always decimal 8 in 2.X, not decimal 10 (despite what you may or may not think!). This, along with symmetry with the hex and binary forms, is why the octal format was changed in 3.X—you must use 0o010 in 3.X, and probably should in 2.6 and 2.7 both for clarity and forward-compatibility with 3.X.

Secondly, note that these literals can produce arbitrarily long integers. The following, for instance, creates an integer with hex notation and then displays it first in decimal and then in octal and binary with converters (run in 3.X here: in 2.X the decimal and octal displays have a trailing L to denote its separate long type, and octals display without the letter o):

```
>>> X = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFF
>>> X
5192296858534827628530496329220095
>>> oct(X)
'0o17777777777777777777777777777777777777'
>>> bin(X)
'0b1111111111111111111111111111111111111111111111111111 ...and so on... 11111'
```

Speaking of binary digits, the next section shows tools for processing individual bits.

<u>Bitwise Operations</u>

Besides the normal numeric operations (addition, subtraction, and so on), Python supports most of the numeric expressions available in the C language. This includes operators that treat integers as strings of binary bits, and can come in handy if your Python code must deal with things like network packets, serial ports, or packed binary data produced by a C program.

We can't dwell on the fundamentals of Boolean math here—again, those who must use it probably already know how it works, and others can often postpone the topic altogether—but the basics are straightforward. For instance, here are some of Python's bitwise expression operators at work performing bitwise shift and Boolean operations on integers:

```
>>> x = 1        # 1 decimal is 0001 in bits
>>> x << 2       # Shift left 2 bits: 0100
4
>>> x | 2        # Bitwise OR (either bit=1): 0011
3
>>> x & 1        # Bitwise AND (both bits=1): 0001
1
```

In the first expression, a binary 1 (in base 2, 0001) is shifted left two slots to create a binary 4 (0100). The last two operations perform a binary OR to combine bits (0001|0010 = 0011) and a binary AND to select common bits (0001&0001 = 0001). Such bitmasking operations allow us to encode and extract multiple flags and other values within a single integer.

This is one area where the binary and hexadecimal number support in Python as of 3.0 and 2.6 become especially useful—they allow us to code and inspect numbers by bitstrings:

```
>>> X = 0b0001               # Binary literals
>>> X << 2                   # Shift left
4
```

```
>>> bin(X << 2)                # Binary digits string
'0b100'
>>> bin(X | 0b010)             # Bitwise OR: either
'0b11'
>>> bin(X & 0b1)               # Bitwise AND: both
'0b1'
```

This is also true for values that begin life as hex literals, or undergo base conversions:

```
>>> X = 0xFF                   # Hex literals
>>> bin(X)
'0b11111111'
>>> X ^ 0b10101010             # Bitwise XOR: either but not both
85
>>> bin(X ^ 0b10101010)
'0b1010101'
>>> int('01010101', 2)         # Digits=>number: string to int per base
85
>>> hex(85)                    # Number=>digits: Hex digit string
'0x55'
```

Also in this department, Python 3.1 and 2.7 introduced a new integer bit_length method, which allows you to query the number of bits required to represent a number's value in binary. You can often achieve the same effect by subtracting 2 from the length of the bin string using the len built-in, though it may be less efficient:

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

We won't go into much more detail on such "bit twiddling" here. It's supported if you need it, but bitwise operations are often not as important in a high-level language such as Python as they are in a low-level language such as C. As a rule of thumb, if you find yourself wanting to flip bits in Python, you should think about which language you're really coding. Python's lists, dictionaries, and the like provide richer—and usually better—ways to encode information than bit strings, especially when your data's audience includes readers of the human variety

Other Numeric Types

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

Decimal Type

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

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

The last point merits elaboration. As previewed briefly when we explored comparisons, floating-point math is less than exact because of the limited space used to store values.

For example, the following should yield zero, but it does not. The result is close to zero, but there are not enough bits to be precise here:

```
>>> 0.1 + 0.1 + 0.1 - 0.3          # Python 3.3
5.551115123125783e-17
```

On Pythons prior to 3.1 and 2.7, printing the result to produce the user-friendly display format doesn't completely help either, because the hardware related to floating-point math is inherently limited in terms of accuracy (a.k.a. precision). The following in 3.3 gives the same result as the previous output:

```
>>> print(0.1 + 0.1 + 0.1 - 0.3)  # Pythons < 2.7, 3.1
5.55111512313e-17
```

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

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

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

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

In Pythons 2.7, 3.1, and later, it's also possible to create a decimal object from a floatingpoint object, with a call of the form decimal.Decimal.from_float(1.25), and recent Pythons allow floating-point numbers to be used directly. The conversion is exact but can sometimes yield a large default number of digits, unless they are fixed per the next section:

```
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)
Decimal('2.775557561565156540423631668E-17')
```

In Python 3.3 and later, the decimal module was also optimized to improve its performance radically: the reported speedup for the new version is 10X to 100X, depending on the type of program benchmarked.

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

```
>>> import decimal
>>> decimal.Decimal(1) / decimal.Decimal(7)                # Default: 28 digits
Decimal('0.1428571428571428571428571429')
>>> decimal.getcontext().prec = 4                          # Fixed precision
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal('0.1429')
>>> Decimal(0.1) + Decimal(0.1) + Decimal(0.1) - Decimal(0.3)          # Closer to 0
Decimal('1.110E-17')
```

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

```
>>> 1999 + 1.33           # This has more digits in memory than displayed in 3.3
2000.33
>>>
>>> decimal.getcontext().prec = 2
>>> pay = decimal.Decimal(str(1999 + 1.33))
>>> pay
Decimal('2000.33')
```

In Python 2.6 and 3.0 and later, it's also possible to reset precision temporarily by using the with context manager statement. The precision is reset to its original value on statement exit; in a new Python 3.3 session (The "..." here is Python's interactive prompt for continuation lines in some interfaces and requires manual indentation; IDLE omits this prompt and indents for you):

```
C:\code> C:\Python33\python
>>> import decimal
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.3333333333333333333333333333')
>>>
>>> with decimal.localcontext() as ctx:
... ctx.prec = 2
... decimal.Decimal('1.00') / decimal.Decimal('3.00')
...
Decimal('0.33')
>>>
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.3333333333333333333333333333')
```

Though useful, this statement requires much more background knowledge than you've obtained at this point.

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

Fraction Type

Python 2.6 and 3.0 debuted a new numeric type, Fraction, which implements a rational number object. It essentially keeps both a numerator and a denominator explicitly, so as to avoid some of the inaccuracies and limitations of floating-point math. Like decimals, fractions do not map as closely to computer hardware as floating-point numbers. This means their performance may not be as good, but it also allows them to provide extra utility in a standard tool where required or useful.

Fraction is a functional cousin to the Decimal fixed-precision type described in the prior section, as both can be used to address the floating-point type's numerical inaccuracies. It's also used in similar ways—like Decimal, Fraction resides in a module; import its constructor and pass in a numerator and a denominator to make one (among other schemes). The following interaction shows how:

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

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

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

```
>>> x + y
Fraction(1, 1)
>>> x − y                    # Results are exact: numerator, denominator
Fraction(−1, 3)
>>> x * y
Fraction(2, 9)
```

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

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

Notice that this is different from floating-point-type math, which is constrained by the underlying limitations of floating-point hardware. To compare, here are the same operations run with floating-point objects, and notes on their limited accuracy—they may display fewer digits in recent Pythons than they used to, but they still aren't exact values in memory:

```
>>> a = 1 / 3.0              # Only as accurate as floating-point hardware
>>> b = 4 / 6.0              # Can lose precision over many calculations
>>> a
0.3333333333333333
>>> b
0.6666666666666666
>>> a + b
1.0
>>> a - b
-0.3333333333333333
>>> a * b
0.2222222222222222
```

This floating-point limitation is especially apparent for values that cannot be represented accurately given their limited number of bits in memory. Both Fraction and Decimal provide ways to get exact results, albeit at the cost of some speed and code verbosity. For instance, in the following example (repeated from the prior section), floating-point numbers do not accurately give the zero answer expected, but both of the other types do:

```
>>> 0.1 + 0.1 + 0.1 - 0.3                  # This should be zero (close, but not exact)
5.551115123125783e-17
>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10) - Fraction(3, 10)
Fraction(0, 1)
```

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

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

```
>>> 1 / 3                          # Use a ".0" in Python 2.X for true "/"
0.3333333333333333
>>> Fraction(1, 3)                 # Numeric accuracy, two ways
Fraction(1, 3)
>>> import decimal
>>> decimal.getcontext().prec = 2
>>> Decimal(1) / Decimal(3)
Decimal('0.33')
```

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

```
>>> (1 / 3) + (6 / 12)                   # Use a ".0" in Python 2.X for true "/"
0.8333333333333333
>>> Fraction(6, 12)                      # Automatically simplified
Fraction(1, 2)
>>> Fraction(1, 3) + Fraction(6, 12)
Fraction(5, 6)
>>> decimal.Decimal(str(1/3)) + decimal.Decimal(str(6/12))
Decimal('0.83')
>>> 1000.0 / 1234567890
8.100000073710001e-07
>>> Fraction(1000, 1234567890)           # Substantially simpler!
Fraction(100, 123456789)
```

To support fraction conversions, floating-point objects now have a method that yields their numerator and denominator ratio, fractions have a from_float method, and float accepts a Fraction as an argument. Trace through the following interaction to see how this pans out (the * in the second test is special syntax that expands a tuple into individual arguments

```
>>> (2.5).as_integer_ratio()             # float object method
(5, 2)
>>> f = 2.5
>>> z = Fraction(*f.as_integer_ratio())  # Convert float -> fraction: two args
>>> z                                    # Same as Fraction(5, 2)
Fraction(5, 2)
>>> x                                    # x from prior interaction
Fraction(1, 3)
>>> x + z
Fraction(17, 6)                          # 5/2 + 1/3 = 15/6 + 2/6
>>> float(x)                             # Convert fraction -> float
0.3333333333333333
>>> float(z)
2.5
>>> float(x + z)
2.8333333333333335
>>> 17 / 6
2.8333333333333335
```

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

>>> Fraction(*(1.75).as_integer_ratio())
Fraction(7, 4)
```

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

```
>>> x
Fraction(1, 3)
>>> x + 2                         # Fraction + int -> Fraction
Fraction(7, 3)
>>> x + 2.0                       # Fraction + float -> float
2.3333333333333335
>>> x + (1./3)                    # Fraction + float -> float
0.6666666666666666
>>> x + (4./3)
1.6666666666666665
>>> x + Fraction(4, 3)            # Fraction + Fraction -> Fraction
Fraction(5, 3)
```

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

```
>>> 4.0 / 3
1.3333333333333333
>>> (4.0 / 3).as_integer_ratio()                      # Precision loss from float
(6004799503160661, 4503599627370496)
>>> x
Fraction(1, 3)
>>> a = x + Fraction(*(4.0 / 3).as_integer_ratio())
>>> a
Fraction(22517998136852479, 13510798882111488)
>>> 22517998136852479 / 13510798882111488.           # 5 / 3 (or close to it!)
1.6666666666666667
>>> a.limit_denominator(10)                           # Simplify to closest fraction
Fraction(5, 3)
```

For more details on the Fraction type, experiment further on your own and consult the Python 2.6, 2.7, and 3.X library manuals and other documentation.

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

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

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

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

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

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

Again, though, for most practical purposes, you can treat True and False as though they are predefined variables set to integers 1 and 0. Most programmers had been preassigning True and False to 1 and 0 anyway; the bool type simply makes this standard.

Its implementation can lead to curious results, though. Because True is just the integer 1 with a custom display format, True + 4 yields integer 5 in Python!

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

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

Numeric Extensions

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

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

Because it's so advanced, we won't talk further about NumPy in this course. You can find additional support for advanced numeric programming in Python, including graphics and plotting tools, extended precision floats, statistics libraries, and the popular SciPy package by searching the Web. Also note that NumPy is currently an optional extension; it doesn't come with Python and must be installed separately, though you'll probably want to do so if you care enough about this domain to look it up on the Web.

Module 5 - The dynamic typing interlude

In the prior module, we began exploring Python's core object types in depth by studying Python numeric types and operations. We'll resume our object type tour in the next module, but before we move on, it's important that you get a handle on what may be the most fundamental idea in Python programming and is certainly the basis of much of both the conciseness and flexibility of the Python language—dynamic typing, and the polymorphism it implies.

Throughout this course, in Python, we do not declare the specific types of the objects our scripts use. In fact, most programs should not even care about specific types; in exchange, they are naturally applicable in more contexts than we can sometimes even plan ahead for. Because dynamic typing is the root of this flexibility, and is also a potential stumbling block for newcomers, let's take a brief side trip to explore the model here.

The Case of the Missing Declaration Statements

If you have a background in compiled or statically typed languages like C, C++, or Java, you might find yourself a bit perplexed at this point in the course. So far, we've been using variables without declaring their existence or their types, and it somehow works.

When we type a = 3 in an interactive session or program file, for instance, how does Python know that a should stand for an integer? For that matter, how does Python know what a is at all?

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

Variables, Objects, and References

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

- **Variable creation -** A variable (i.e., name), like a, is created when your code first assigns it a value. Future assignments change the value of the already created name. Technically, Python detects some names before your code runs, but you can think of it as thoughinitial assignments make variables
- **Variable types -** A variable never has any type information or constraints associated with it. The notion of type lives with objects, not names. Variables are generic in nature; they always simply refer to a particular object at a particular point in time.
- **Variable use –** When a variable appears in an expression, it is immediately replaced with the object that it currently refers to, whatever that may be. Further, all variables must be explicitly assigned before they can be used; referencing unassigned variables results in errors.

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

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

```
>>> a = 3          # Assign a name to an object
```

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

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

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

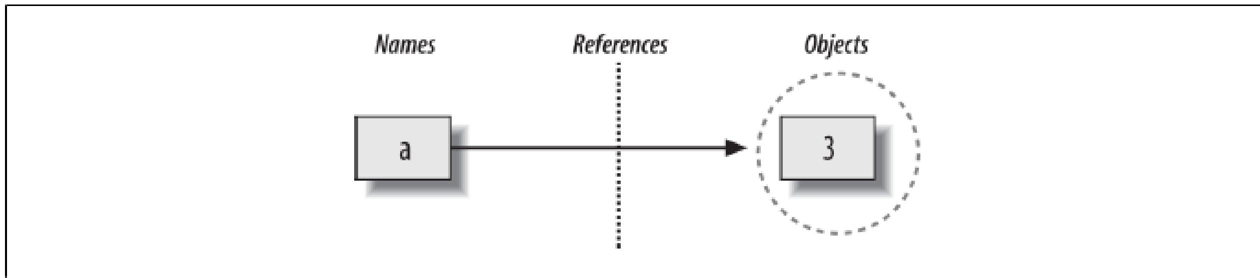


Figure: *Names and objects after running the assignment a = 3. Variable a becomes a reference to the object 3. Internally, the variable is really a pointer to the object's memory space created by running the literal expression 3.*

These links from variables to objects are called references in Python—that is, a reference is a kind of association, implemented as a pointer in memory.1 Whenever the variables are later used (i.e., referenced), Python automatically follows the variable-to-object links. This is all simpler than the terminology may imply. In concrete terms:

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

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

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

Types Live with Objects, Not Variables

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

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

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

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

Objects, on the other hand, know what type they are—each object contains a header field that tags the object with its type. The integer object 3, for example, will contain the value 3, plus a designator that tells Python that the object is

an integer (strictly speaking, a pointer to an object called int, the name of the integer type). The type designator of the 'spam' string object points to the string type (called str) instead.

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

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

Objects Are Garbage-Collected

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

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

The answer is that in Python, whenever a name is assigned to a new object, the space held by the prior object is reclaimed if it is not referenced by any other name or object. This automatic reclamation of objects' space is known as garbage collection, and makes life much simpler for programmers of languages like Python that support it.

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

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

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

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

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

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

Shared References

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

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

Typing these two statements generates the scene captured in Figure below. The second command causes Python to create the variable b; the variable a is being used and not assigned here, so it is replaced with the object it references (3), and b is made to reference that object. The net effect is that the variables a and b wind up referencing the same object (that is, pointing to the same chunk of memory).

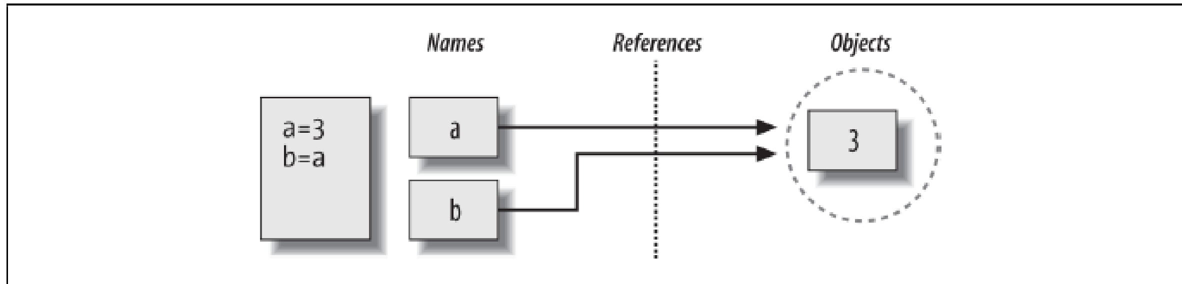


Figure: *Names and objects after next running the assignment b = a. Variable b becomes a reference to the object 3. Internally, the variable is really a pointer to the object's memory space created by running the literal expression 3.*

This scenario in Python—with multiple names referencing the same object—is usually called a shared reference (and sometimes just a shared object). Note that the names a and b are not linked to each other directly when this happens; in fact, there is no way to ever link a variable to another variable in Python. Rather, both variables point to the same object via their references.

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

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

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

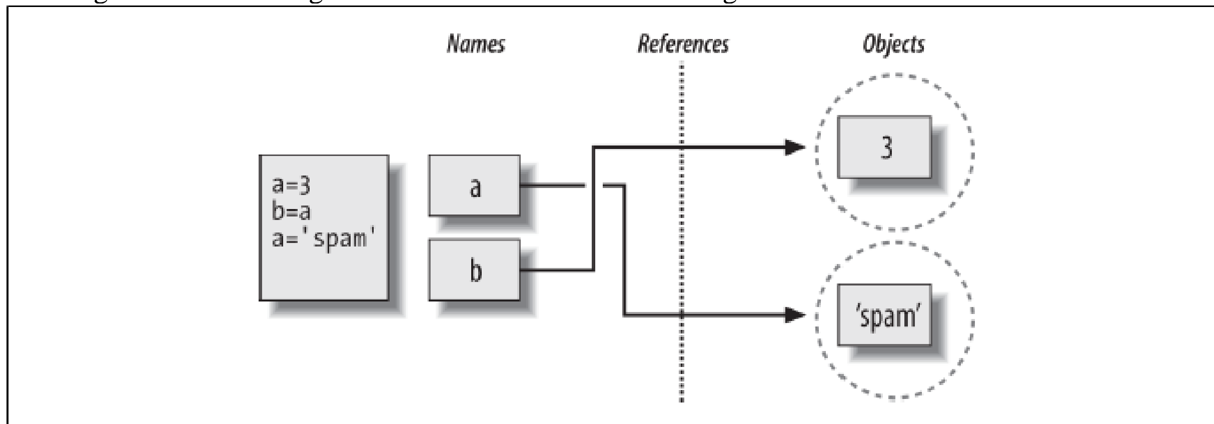


Figure: *Names and objects after finally running the assignment a = 'spam'. Variable a references the new object (i.e., piece of memory) created by running the literal expression 'spam', but variable b still refers to the original object 3. Because this assignment is not an in-place change to the object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

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

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

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

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

Shared References and In-Place Changes

As you'll see later, there are objects and operations that perform in-place object changes—Python's mutable types, including lists, dictionaries, and sets. For instance, an assignment to an offset in a list actually changes the list object itself in place, rather than generating a brand-new list object.

Though you must take it somewhat on faith at this point in the course, this distinction can matter much in your programs. For objects that support such in-place changes, you need to be more aware of shared references, since a change from one name may impact others. Otherwise, your objects may seem to change for no apparent reason.

Given that all assignments are based on references (including function argument passing), it's a pervasive potential. To illustrate, let's take another look at the list objects introduced. Recall that lists, which do support in-place assignments to positions, are simply collections of other objects, coded in square brackets:

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

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

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

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

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

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

This behavior only occurs for mutable objects that support in-place changes, and is usually what you want, but you should be aware of how it works, so that it's expected. It's also just the default: if you don't want such behavior, you can request that Python copy objects instead of making references. There are a variety of ways to copy a list, including using the built-in list function and the standard library copy module. Perhaps the most common way is to slice from start to finish

```
>>> L1 = [2, 3, 4]
>>> L2 = L1[:]            # Make a copy of L1 (or list(L1), copy.copy(L1), etc.)
>>> L1[0] = 24
>>> L1
[24, 3, 4]
>>> L2                    # L2 is not changed
[2, 3, 4]
```

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

Note that this slicing technique won't work on the other major mutable core types, dictionaries and sets, because they are not sequences—to copy a dictionary or set, instead use their X.copy() method call (lists have one as of Python 3.3 as well), or pass the original object to their type names, dict and set. Also, note that the standard library copy module has a call for copying any object type generically, as well as a call for copying nested object structures—a dictionary with nested lists, for example:

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

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

Shared References and Equality

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

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

Because Python caches and reuses small integers and small strings, as mentioned earlier, the object 42 here is probably not literally reclaimed; instead, it will likely remain in a system table to be reused the next time you generate a 42 in your code. Most kinds of objects, though, are reclaimed immediately when they are no longer referenced; for those that are not, the caching mechanism is irrelevant to your code. For instance, because of Python's reference model, there are two different ways to check for equality in a Python program. Let's create a shared reference to demonstrate:

```
>>> L = [1, 2, 3]
```

```
>>> M = L                    # M and L reference the same object
>>> L == M                   # Same values
True
>>> L is M                   # Same objects
True
```

The first technique here, the == operator, tests whether the two referenced objects have the same values; this is the method almost always used for equality checks in Python.

The second method, the is operator, instead tests for object identity—it returns True only if both names point to the exact same object, so it is a much stronger form of equality testing and is rarely applied in most programs.

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

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

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

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

In this interaction, X and Y should be == (same value), but not is (same object) because we ran two different literal expressions (42). Because small integers and strings are cached and reused, though, is tells us they reference the same single object.

In fact, if you really want to look under the hood, you can always ask Python how many references there are to an object: the getrefcount function in the standard sys module returns the object's reference count. When I ask about the integer object 1 in the IDLE GUI, for instance, it reports 647 reuses of this same object (most of which are in IDLE's system code, not mine, though this returns 173 outside IDLE so Python must be hoarding 1s as well):

```
>>> import sys
>>> sys.getrefcount(1)            # 647 pointers to this shared piece of memory
647
```

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

Dynamic Typing Is Everywhere

Of course, you don't really need to draw name/object diagrams with circles and arrows to use Python. When you're starting out, though, it sometimes helps you understand unusual cases if you can trace their reference structures as we've done here. If a mutable object changes out from under you when passed around your program, for example, chances are you are witnessing some of this module's subject matter firsthand. Moreover, even if dynamic typing seems a little abstract at this point, you probably will care about it eventually. Because everything seems to work by assignment and references in Python, a basic understanding of this model is useful in many different contexts.

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

At the most practical level, dynamic typing means there is less code for you to write. Just as importantly, though, dynamic typing is also the root of Python's polymorphism, and will revisit again later in this course.

Because we do not constrain types in Python code, it is both concise and highly flexible. As you'll see, when used well, dynamic typing—and the polymorphism it implies— produces code that automatically adapts to new requirements as your systems evolve.

Module 6 - Strings

So far, we've studied numbers and explored Python's dynamic typing model. The next major type on our in-depth core object tour is the Python string—an ordered collection of characters used to store and represent text- and bytes-based information. We looked briefly at strings in prior modules. Here, we will revisit them in more depth, filling in some of the details we skipped earlier.

Before we get started, I also want to clarify what we won't be covering here. Prior modules briefly previewed Unicode strings and files—tools for dealing with non-ASCII text. Unicode is a key tool for some programmers, especially those who work in the Internet domain. It can pop up, for example, in web pages, email content and headers, FTP transfers, GUI APIs, directory tools, and HTML, XML and JSON text.

At the same time, Unicode can be a heavy topic for programmers just starting out, and many (or most) of the Python programmers I meet today still do their jobs in blissful ignorance of the entire topic. In light of that, this course will not cover the Unicode in detail.

That is, this module tells only part of the string story in Python—the part that most scripts use and most programmers need to know. It explores the fundamental str string type, which handles ASCII text, and works the same regardless of which version of Python you use. Despite this intentionally limited scope, because str also handles Unicode in Python 3.X, and the separate unicode type works almost identically to str in 2.X, everything we learn here will apply directly to Unicode processing too.

From a functional perspective, strings can be used to represent just about anything that can be encoded as text or bytes. In the text department, this includes symbols and words (e.g., your name), contents of text files loaded into memory, Internet addresses, Python source code, and so on. Strings can also be used to hold the raw bytes used for media files and network transfers, and both the encoded and decoded forms of non- ASCII Unicode text used in internationalized programs. You may have used strings in other languages, too. Python's strings serve the same role as character arrays in languages such as C, but they are a somewhat higher-level tool than arrays. Unlike in C, in Python, strings come with a powerful set of processing tools. Also unlike languages such as C, Python has no distinct type for individual characters; instead, you just use one-character strings.

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

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

Table: Common string literals and operations	
Operation	Interpretation
S = ''	Empty string
S = "spam's"	Double quotes, same as single
S = 's\np\ta\x00m	Escape sequences
S = """...multiline..."""	Triple-quoted block strings
S = r'\temp\spam'	Raw strings (no escapes)
B = b'sp\xc4m'	Byte strings in 2.6, 2.7, and 3.X
U = u'sp\u00c4m'	Unicode strings in 2.X and 3.3
S1 + S2	Concatenate
S * 3	Repeat
S[i]	Index
S[i:j]	Slice
len(S)	Length
"a %s parrot" % kind	String formatting expression
"a {0} parrot".format(kind)	String formatting method in 2.6, 2.7, and 3.
S.find('pa')	String methods (see ahead for all 43): search
S.rstrip()	remove whitespace
S.replace('pa', 'xx')	Replacement
S.split(',')	Split on delimiter
S.isdigit()	Content test
S.lower()	Lower case conversion
S.endswith('spam')	End test
'spam'.join(strlist)	Delimiter join
S.encode('latin-1')	Unicode encoding
B.decode('utf8')	Unicode decoding, etc.
for x in S: print(x)	Iteration, membership
'spam' in S	Pattern matching: library module
[c * 2 for c in S]	
map(ord, S)	
re.match('sp(.*)am', line)	

Beyond the core set of string tools in Table above, Python also supports more advanced pattern-based string processing with the standard library's re (for "regular expression") module, and even higher-level text processing tools such as XML parsers. This course's scope, though, is focused on the fundamentals represented by Table above.

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

String literals

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

- Single quotes: 'spa"m'
- Double quotes: "spa'm"
- Triple quotes: '''... spam ...''', """... spam ..."""
- Escape sequences: "s\tp\na\0m"
- Raw strings: r"C:\new\test.spm"
- Bytes literals in 3.X and 2.6+ : b'sp\x01am'
- Unicode literals in 2.X and 3.3+ : u'eggs\u0020spam'

The single- and double-quoted forms are by far the most common; the others serve specialized roles. Let's take a quick look at all the other options in turn.

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

Basic Operations
You can concatenate strings using the + operator and repeat them using the * operator:

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

The len built-in function here returns the length of a string (or any other object with a length). Formally, adding two string objects with + creates a new string object, with the contents of its operands joined, and repetition with * is like adding a string to itself a number of times. In both cases, Python lets you create arbitrarily sized strings; there's no need to predeclare anything in Python, including the sizes of data structures—you simply create string objects as needed and let Python manage the underlying memory space automatically.

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

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

Notice that operator overloading is at work here already: we're using the same + and * operators that perform addition and multiplication when using numbers. Python does the correct operation because it knows the types of the objects being added and multiplied.

But be careful: the rules aren't quite as liberal as you might expect. For instance, Python doesn't allow you to mix numbers and strings in + expressions: 'abc'+9 raises an error instead of automatically converting 9 to a string.

You can also iterate over strings in loops using for statements, which repeat actions, and test membership for both characters and substrings with the in expression operator, which is essentially a search. For substrings, in is much like the str.find() method covered later in this Module, but it returns a Boolean result instead of the substring's position (the following uses a 3.X print call and may leave your cursor a bit indented; in 2.X say print c, instead):

```
>>> myjob = "hacker"
>>> for c in myjob: print(c, end=' ')      # Step through items, print each (3.X form)
h a c k e r
>>> "k" in myjob                           # Found
True
>>> "z" in myjob                           # Not found
False
>>> 'spam' in 'abcspamdef'                 # Substring search, no position returned
True
```

The for loop assigns a variable to successive items in a sequence (here, a string) and executes one or more statements for each item. In effect, the variable c becomes a cursor stepping across the string's characters here.

Indexing and Slicing

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

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

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

The first line defines a four-character string and assigns it the name S. The next line indexes it in two ways: S[0] fetches the item at offset 0 from the left—the one-character string 's'; S[−2] gets the item at offset 2 back from the end—or equivalently, at offset (4 + (−2)) from the front. In more graphic terms, offsets and slices map to cells as shown in Figure below:

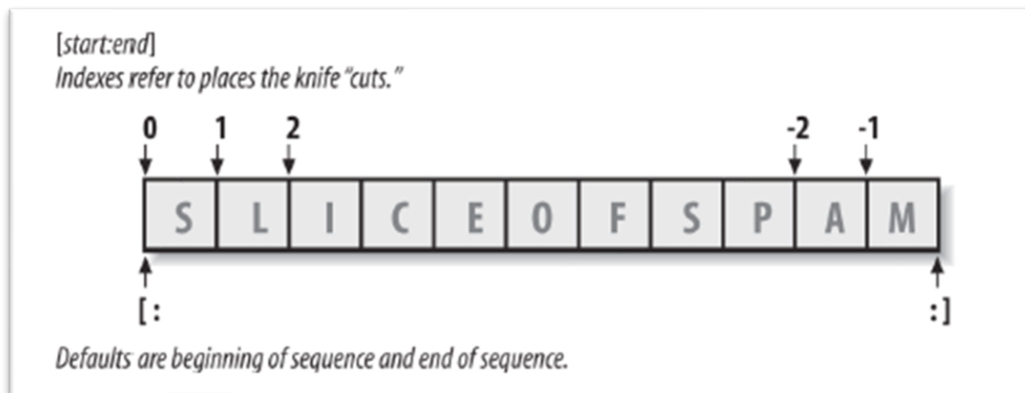


Figure: *Offsets and slices: positive offsets start from the left end (offset 0 is the first item), and negatives count back from the right end (offset −1 is the last item). Either kind of offset can be used to give positions in indexing and slicing operations.*

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

The basics of slicing are straightforward. When you index a sequence object such as a string on a pair of offsets separated by a colon, Python returns a new object containing the contiguous section identified by the offset pair. The left offset is taken to be the lower bound (inclusive), and the right is the upper bound (noninclusive). That is, Python fetches all items from the lower bound up to but not including the upper bound, and returns a new object containing the fetched items. If omitted, the left and right bounds default to 0 and the length of the object you are slicing, respectively.

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

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

Indexing (S[i]) fetches components at offsets:

- The first item is at offset 0.
- Negative indexes mean to count backward from the end or right.
- S[0] fetches the first item.
- S[−2] fetches the second item from the end (like S[len(S)−2]).

Slicing (S[i:j]) extracts contiguous sections of sequences:

- The upper bound is noninclusive.
- Slice boundaries default to 0 and the sequence length, if omitted.
- S[1:3] fetches items at offsets 1 up to but not including 3.
- S[1:] fetches items at offset 1 through the end (the sequence length).
- S[:3] fetches items at offset 0 up to but not including 3.
- S[:−1] fetches items at offset 0 up to but not including the last item.
- S[:] fetches items at offsets 0 through the end—making a top-level copy of S.

Extended slicing (S[i:j:k]) accepts a step (or stride) k, which defaults to +1:

- Allows for skipping items and reversing order—see the next section.

The second-to-last bullet item listed here turns out to be a very common technique: it makes a full top-level copy of a sequence object—an object with the same value, but a distinct piece of memory. This isn't very useful for immutable objects like strings, but it comes in handy for objects that may be changed in place, such as lists.

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

<u>Extended slicing: The third limit and slice objects</u>

In Python 2.3 and later, slice expressions have support for an optional third index, used as a step (sometimes called a stride). The step is added to the index of each item extracted.

The full-blown form of a slice is now X[I:J:K], which means "extract all the items in X, from offset I through J−1, by K." The third limit, K, defaults to +1, which is why normally all items in a slice are extracted from left to right. If you specify an explicit value, however, you can use the third limit to skip items or to reverse their order.

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

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

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

```
>>> S = 'hello'
>>> S[::−1]                    # Reversing items
'olleh'
```

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

```
>>> S = 'abcedfg'
>>> S[5:1:−1]                  # Bounds roles differ
'fdec'
```

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

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

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

String formatting
Although you can get a lot done with the string methods and sequence operations we've already met, Python also provides a more advanced way to combine string processing tasks—string formatting allows us to perform multiple type-specific substitutions on a string in a single step. It's never strictly required, but it can be convenient, especially when formatting text to be displayed to a program's users. Due to the wealth of new ideas in the Python world, string formatting is available in two flavors in Python today (not counting the less-used string module Template system mentioned in the prior section):

String formatting expressions: '...%s...' % (values)

The original technique available since Python's inception, this form is based upon the C language's "printf" model, and sees widespread use in much existing code.

String formatting method calls: '...{}...'.format(values)

A newer technique added in Python 2.6 and 3.0, this form is derived in part from a same-named tool in C#/.NET, and overlaps with string formatting expression functionality.

Since the method call flavor is newer, there is some chance that one or the other of these may become deprecated and removed over time. When 3.0 was released in 2008, the expression seemed more likely to be deprecated in later Python releases. Indeed, 3.0's documentation threatened deprecation in 3.1 and removal thereafter. This hasn't

happened as of 2013 and 3.3, and now looks unlikely given the expression's wide use—in fact, it still appears even in Python's own standard library thousands of times today!

Naturally, this story's development depends on the future practice of Python's users. On the other hand, because both the expression and method are valid to use today and either may appear in code you'll come across, this course covers both techniques in full here. As you'll see, the two are largely variations on a theme, though the method has some extra features (such as thousands separators), and the expression is often more concise and seems second nature to most Python programmers.

This course itself uses both techniques in later examples for illustrative purposes. If its author has a preference, he will keep it largely classified, except to quote from Python's import this motto:

There should be one—and preferably only one—obvious way to do it. Unless the newer string formatting method is compellingly better than the original and widely used expression, its doubling of Python programmers' knowledge base requirements in this domain seems unwarranted—and even un-Pythonic, per the original and longstanding meaning of that term. Programmers should not have to learn two complicated tools if those tools largely overlap. You'll have to judge for yourself whether formatting merits the added language heft, of course, so let's give both a fair hearing.

Since string formatting expressions are the original in this department, we'll start with them. Python defines the % binary operator to work on strings (you may recall that this is also the remainder of division, or modulus, operator for numbers). When applied to strings, the % operator provides a simple way to format values as strings according to format definition. In short, the % operator provides a compact way to code multiple string substitutions all at once, instead of building and concatenating parts individually.

To format strings:

1) On the left of the % operator, provide a format string containing one or more embedded conversion targets, each of which starts with a % (e.g., %d).
2) On the right of the % operator, provide the object (or objects, embedded in a tuple) that you want Python to insert into the format string on the left in place of the conversion target (or targets).

For instance, in the formatting example we saw earlier in this Module, the integer 1 replaces the %d in the format string on the left, and the string 'dead' replaces the %s.

The result is a new string that reflects these two substitutions, which may be printed or saved for use in other roles:

```
>>> 'That is %d %s bird!' % (1, 'dead')             # Format expression
That is 1 dead bird!
```

Technically speaking, string formatting expressions are usually optional—you can generally do similar work with multiple concatenations and conversions. However, formatting allows us to combine many steps into a single operation. It's powerful enough to warrant a few more examples:

```
>>> exclamation = 'Ni'
>>> 'The knights who say %s!' % exclamation    # String substitution
'The knights who say Ni!'
>>> '%d %s %g you' % (1, 'spam', 4.0)          # Type-specific substitutions
'1 spam 4 you'
>>> '%s -- %s -- %s' % (42, 3.14159, [1, 2, 3])  # All types match a %s target
'42 -- 3.14159 -- [1, 2, 3]'
```

The first example here plugs the string 'Ni' into the target on the left, replacing the %s marker. In the second example, three values are inserted into the target string. Note that when you're inserting more than one value, you need to group the values on the right in parentheses (i.e., put them in a tuple). The % formatting expression operator

expects either a single item or a tuple of one or more items on its right side.

The third example again inserts three values—an integer, a floating-point object, and a list object—but notice that all of the targets on the left are %s, which stands for conversion to string. As every type of object can be converted to a string (the one used when printing), every object type works with the %s conversion code. Because of this, unless you will be doing some special formatting, %s is often the only code you need to remember for the formatting expression.

Again, keep in mind that formatting always makes a new string, rather than changing the string on the left; because strings are immutable, it must work this way. As before, assign the result to a variable name if you need to retain it.

Advanced Formatting Expression Syntax

For more advanced type-specific formatting, you can use any of the conversion type codes listed in Table below in formatting expressions; they appear after the % character in substitution targets. C programmers will recognize most of these because Python string formatting supports all the usual C printf format codes (but returns the result, instead of displaying it, like printf). Some of the format codes in the table provide alternative ways to format the same type; for instance, %e, %f, and %g provide alternative ways to format floating-point numbers.

Table: *String formatting type codes*

Code	Meaning
s	String (or any object's str(X) string)
r	Same as s, but uses repr, not str
c	Character (int or str)
d	Decimal (base-10 integer)
i	Integer
u	Same as d (obsolete: no longer unsigned)
o	Octal integer (base 8)
x	Hex integer (base 16)
X	Same as x, but with uppercase letters
e	Floating point with exponent, lowercase
E	Same as e, but uses uppercase letters
f	Floating-point decimal
F	Same as f, but uses uppercase letters
g	Floating-point e or f
G	Floating-point E or F
%	Literal % (coded as %%)

In fact, conversion targets in the format string on the expression's left side support a variety of conversion operations with a fairly sophisticated syntax all their own. The general structure of conversion targets looks like this:

```
%[(keyname)][flags][width][.precision]typecode
```

The type code characters in the first column of Table above show up at the end of this target string's format. Between the % and the type code character, you can do any of the following:

- Provide a key name for indexing the dictionary used on the right side of the expression
- List flags that specify things like left justification (−), numeric sign (+), a blank before positive numbers and a – for negatives (a space), and zero fills (0)
- Give a total minimum field width for the substituted text
- Set the number of digits (precision) to display after a decimal point for floating point numbers

Both the width and precision parts can also be coded as a * to specify that they should take their values from the next item in the input values on the expression's right side (useful when this isn't known until runtime). And if you don't need any of these extra tools, a simple %s in the format string will be replaced by the corresponding value's default print string, regardless of its type.

Dictionary-Based Formatting Expressions

As a more advanced extension, string formatting also allows conversion targets on the left to refer to the keys in a dictionary coded on the right and fetch the corresponding values. This opens the door to using formatting as a sort of template tool. We've only met dictionaries briefly, but here's an example that demonstrates the basics:

```
>>> '%(qty)d more %(food)s' % {'qty': 1, 'food': 'spam'}
'1 more spam'
```

Here, the (qty) and (food) in the format string on the left refer to keys in the dictionary literal on the right and fetch their associated values. Programs that generate text such as HTML or XML often use this technique—you can build up a dictionary of values and substitute them all at once with a single formatting expression that uses key-based references (notice the first comment is above the triple quote so it's not added to the string, and I'm typing this in IDLE without a "..." prompt for continuation lines):

```
>>> # Template with substitution targets
>>> reply = """
Greetings...
Hello %(name)s!
Your age is %(age)s
"""
>>> values = {'name': 'Bob', 'age': 40} # Build up values to substitute
>>> print(reply % values) # Perform substitutions
Greetings...
Hello Bob!
Your age is 40
```

This trick is also used in conjunction with the vars built-in function, which returns a dictionary containing all the variables that exist in the place it is called:

```
>>> food = 'spam'
>>> qty = 10
>>> vars()
{'food': 'spam', 'qty': 10, ...plus built-in names set by Python... }
```

When used on the right side of a format operation, this allows the format string to refer to variables by name—as dictionary keys:

```
>>> '%(qty)d more %(food)s' % vars() # Variables are keys in vars()
'10 more spam'
```

We'll study dictionaries in more depth later. Additional formatting expression examples also appear ahead as comparisons to the formatting method—this module's next and final string topic.

String methods

In addition to expression operators, strings provide a set of methods that implement more sophisticated text-processing tasks. In Python, expressions and built-in functions may work across a range of types, but methods are generally specific to object types—string methods, for example, work only on string objects. The method sets of some types intersect in Python 3.X (e.g., many types have count and copy methods), but they are still more type-specific than other tools.

Method Call Syntax

Methods are simply functions that are associated with and act upon particular objects. Technically, they are attributes attached to objects that happen to reference callable functions which always have an implied subject. In finergrained detail, functions are packages of code, and method calls combine two operations at once—an attribute fetch and a call:

- **Attribute fetches -** An expression of the form object.attribute means "fetch the value of attribute in object."
- **Call expressions** - An expression of the form function(arguments) means "invoke the code of function, passing zero or more comma-separated argument objects to it, and return function's result value."

Putting these two together allows us to call a method of an object. The method call expression:

```
object.method(arguments)
```

is evaluated from left to right—Python will first fetch the method of the object and then call it, passing in both object and the arguments. Or, in plain words, the method call expression means this:

```
Call method to process object with arguments.
```

If the method computes a result, it will also come back as the result of the entire methodcall expression. As a more tangible example:

```
>>> S = 'spam'
>>> result = S.find('pa')            # Call the find method to look for 'pa' in string S
```

This mapping holds true for methods of both built-in types, as well as user-defined classes we'll study later. As you'll see throughout this part of the course, most objects have callable methods, and all are accessed using this same method-call syntax. To call an object method, as you'll see in the following sections, you have to go through an existing object; methods cannot be run (and make little sense) without a subject.

Methods of Strings

Table below summarizes the methods and call patterns for built-in string objects in Python 3.3; these change frequently, so be sure to check Python's standard library manual for the most up-to-date list, or run a dir or help call on any string (or the str type name) interactively. Python 2.X's string methods vary slightly; it includes a decode, for example, because of its different handling of Unicode. In this table, S is a string object, and optional arguments are enclosed in square brackets. String methods in this table implement higher-level operations such as splitting and joining, case conversions, content tests, and substring searches and replacements.

Table: *String method calls in Python 3.3*	
S.capitalize()	S.ljust(width [, fill])
S.casefold()	S.lower()
S.center(width [, fill])	S.lstrip([chars])
S.count(sub [, start [, end]])	S.maketrans(x[, y[, z]])
S.encode([encoding [,errors]])	S.partition(sep)
S.endswith(suffix [, start [, end]])	S.replace(old, new [, count])
S.expandtabs([tabsize])	S.rfind(sub [,start [,end]])
S.find(sub [, start [, end]])	S.rindex(sub [, start [, end]])
S.format(fmtstr, *args, **kwargs)	S.rjust(width [, fill])
S.index(sub [, start [, end]])	S.rpartition(sep)
S.isalnum()	S.rsplit([sep[, maxsplit]])
S.isalpha()	S.rstrip([chars])
S.isdecimal()	S.split([sep [,maxsplit]])
S.isdigit()	S.splitlines([keepends])
S.isidentifier()	S.startswith(prefix [, start [, end]])
S.islower()	S.strip([chars])

S.isnumeric()	S.swapcase()
S.isprintable()	S.title()
S.isspace()	S.translate(map)
S.istitle()	S.upper()
S.isupper()	S.zfill(width)
S.join(iterable)	

As you can see, there are quite a few string methods, and we don't have space to cover them all; see Python's library manual or reference texts for all the fine points. To help you get started, though, let's work through some code that demonstrates some of the most commonly used methods in action, and illustrates Python text-processing basics

Other Common String Methods in Action

Other string methods have more focused roles—for example, to strip off whitespace at the end of a line of text, perform case conversions, test content, and test for a substring at the end or front:

```
>>> line = "The knights who say Ni!\n"
>>> line.rstrip()
'The knights who say Ni!'
>>> line.upper()
'THE KNIGHTS WHO SAY NI!\n'
>>> line.isalpha()
False
>>> line.endswith('Ni!\n')
True
>>> line.startswith('The')
True
```

Alternative techniques can also sometimes be used to achieve the same results as string methods—the in membership operator can be used to test for the presence of a substring, for instance, and length and slicing operations can be used to mimic endswith:

```
>>> line
'The knights who say Ni!\n'
>>> line.find('Ni') != −1          # Search via method call or expression
True
>>> 'Ni' in line
True
>>> sub = 'Ni!\n'
>>> line.endswith(sub)             # End test via method call or slice
True
>>> line[-len(sub):] == sub
True
```

See also the format string formatting method described later in this module; it provides more advanced substitution tools that combine many operations in a single step.

Again, because there are so many methods available for strings, we won't look at every one here. You'll see some additional string examples later in this course, but for more details you can also turn to the Python library manual and other documentation sources, or simply experiment interactively on your own. You can also check the help(S.method) results for a method of any string object S for more hints; running help on str.method likely gives the same details.
Note that none of the string methods accepts patterns—for pattern-based text processing, you must use the Python re standard library module. Because of this limitation, though, string methods may sometimes run more quickly than the re module's tools.

General type categories
Now that we've explored the first of Python's collection objects, the string, let's close this module by defining a few general type concepts that will apply to most of the types we look at from here on. With regard to built-in types, it turns out that operations work the same for all the types in the same category, so we'll only need to define most of these ideas once. We've only examined numbers and strings so far, but because they are representative of two of the three major type categories in Python, you already know more about several other types than you might think.

Types Share Operation Sets by Categories
As you've learned, strings are immutable sequences: they cannot be changed in place (the immutable part), and they are positionally ordered collections that are accessed by offset (the sequence part). It so happens that all the sequences we'll study in this part of the course respond to the same sequence operations shown in this module at work on strings—concatenation, indexing, iteration, and so on. More formally, there are three major type (and operation) categories in Python that have this generic nature:

Numbers (integer, floating-point, decimal, fraction, others) - Support addition, multiplication, etc.

Sequences (strings, lists, tuples) - Support indexing, slicing, concatenation, etc.

Mappings (dictionaries) - Support indexing by key, etc.

I'm including the Python 3.X byte strings and 2.X Unicode strings I mentioned at the start of this module under the general "strings" label here. Sets are something of a category unto themselves (they don't map keys to values and are not positionally ordered sequences), and we haven't yet explored mappings on our in-depth
tour. However, many of the other types we will encounter will be similar to numbers and strings. For example, for any sequence objects X and Y:

- X + Y makes a new sequence object with the contents of both operands.
- X * N makes a new sequence object with N copies of the sequence operand X.

In other words, these operations work the same way on any kind of sequence, including strings, lists, tuples, and some user-defined object types. The only difference is that the new result object you get back is of the same type as the operands X and Y—if you concatenate lists, you get back a new list, not a string. Indexing, slicing, and other sequence operations work the same on all sequences, too; the type of the objects being processed tells Python which flavor of the task to perform.

Mutable Types Can Be Changed in Place
The immutable classification is an important constraint to be aware of, yet it tends to trip up new users. If an object type is immutable, you cannot change its value in place; Python raises an error if you try. Instead, you must run code to make a new object containing the new value. The major core types in Python break down as follows:

Immutables (numbers, strings, tuples, frozensets)

> None of the object types in the immutable category support in-place changes, though we can always run expressions to make new objects and assign their results to variables as needed.

Mutables (lists, dictionaries, sets, bytearray)

> Conversely, the mutable types can always be changed in place with operations that do not create new objects. Although such objects can be copied, in-place changes support direct modification.

Generally, immutable types give some degree of integrity by guaranteeing that an object won't be changed by another part of a program.

Module 7 - Lists & Dictionaries

Now that we've learned about numbers and strings, this module moves on to give the full story on Python's list and dictionary object types—collections of other objects, and the main workhorses in almost all Python scripts. As you'll

see, both types are remarkably flexible: they can be changed in place, can grow and shrink on demand, and may contain and be nested in any other kind of object. By leveraging these types, you can build up and process arbitrarily rich information structures in your scripts.

Lists
The next stop on our built-in object tour is the Python list. Lists are Python's most flexible ordered collection object type. Unlike strings, lists can contain any sort of object: numbers, strings, and even other lists. Also, unlike strings, lists may be changed in place by assignment to offsets and slices, list method calls, deletion statements, and more—they are ***mutable*** objects.

Python lists do the work of many of the collection data structures you might have to implement manually in lower-level languages such as C. Here is a quick look at their main properties. Python lists are:

Ordered collections of arbitrary objects- From a functional view, lists are just places to collect other objects so you can treat them as groups. Lists also maintain a left-to-right positional ordering among the items they contain (i.e., they are sequences).

Accessed by offset-Just as with strings, you can fetch a component object out of a list by indexing the list on the object's offset. Because items in lists are ordered by their positions, you can also do tasks such as slicing and concatenation.

Variable-length, heterogeneous, and arbitrarily nestable-Unlike strings, lists can grow and shrink in place (their lengths can vary), and they can contain any sort of object, not just one-character strings (they're heterogeneous). Because lists can contain other complex objects, they also support arbitrary nesting; you can create lists of lists of lists, and so on.

Of the category "mutable sequence"-In terms of our type category qualifiers, lists are mutable (i.e., can be changed in place) and can respond to all the sequence operations used with strings, such as indexing, slicing, and concatenation. In fact, sequence operations work the same on lists as they do on strings; the only difference is that sequence operations such as concatenation and slicing return new lists instead of new strings when applied to lists. Because lists are mutable, however, they also support other operations that strings don't, such as deletion and index assignment operations, which change the lists in place.

Arrays of object references-Technically, Python lists contain zero or more references to other objects. Lists might remind you of arrays of pointers (addresses) if you have a background in some other languages. Fetching an item from a Python list is about as fast as indexing a C array; in fact, lists really are arrays inside the standard Python interpreter, not linked structures. As we learned, though, Python always follows a reference to an object whenever the reference is used, so your program deals only with objects. Whenever you assign an object to a data structure component or variable name, Python always stores a reference to that same object, not a copy of it (unless you request a copy explicitly).

As a preview and reference, Table below summarizes common and representative list object operations. It is fairly complete for Python 3.3, but for the full story, consult the Python standard library manual, or run a help(list) or dir(list) call interactively for a complete list of list methods—you can pass in a real list, or the word list, which is the name of the list data type. The set of methods here is especially prone to change—in fact, two are new as of Python 3.3.

Table: *Common list literals and operations*

Operation	Interpretation
L = []	An empty list

L = [123, 'abc', 1.23, { }]	Four items: indexes 0..3
L = ['Bob', 40.0, ['dev', 'mgr']]	Nested sublists
L = list('spam')	List of an iterable’s items, list of successive integers
L = list(range(-4, 4))	
L[i]	Index, index of index, slice, length
L[i][j]	
L[i:j]	
len(L)	
L1 + L2	Concatenate, repeat
L * 3	Iteration, membership
for x in L: print(x)	
3 in L	
L.append(4)	Methods: growing
L.extend([5,6,7])	
L.insert(i, X)	
L.index(X)	Methods: searching
L.count(X)	
L.sort()	Methods: sorting, reversing, copying (3.3+), clearing (3.3+)
L.reverse()	
L.copy()	
L.clear()	
L.pop(i)	Methods, statements: shrinking
L.remove(X)	
del L[i]	
del L[i:j]	
L[i:j] = []	
L[i] = 3	Index assignment, slice assignment
L[i:j] = [4,5,6]	
L = [x**2 for x in range(5)]	List comprehensions and maps
list(map(ord, 'spam'))	

When written down as a literal expression, a list is coded as a series of objects (really, expressions that return objects) in square brackets, separated by commas. For instance, the second row in Table 8-1 assigns the variable L to a four-item list. A nested list is coded as a nested square-bracketed series (row 3), and the empty list is just a squarebracket pair with nothing inside (row 1).

Many of the operations in Table 8-1 should look familiar, as they are the same sequence operations we put to work on strings earlier—indexing, concatenation, iteration, and so on. Lists also respond to list-specific method calls (which provide utilities such as sorting, reversing, adding items to the end, etc.), as well as in-place change operations (deleting items, assignment to indexes and slices, and so forth). Lists have these tools for change operations because they are a mutable object type.

Lists in action
Perhaps the best way to understand lists is to see them at work. Let's once again turn to some simple interpreter interactions to illustrate the operations in Table above.

Basic List Operations
Because they are sequences, lists support many of the same operations as strings. For example, lists respond to the + and * operators much like strings—they mean concatenation and repetition here too, except that the result is a new list, not a string:

```
% python
>>> len([1, 2, 3])                # Length
3
>>> [1, 2, 3] + [4, 5, 6]         # Concatenation
[1, 2, 3, 4, 5, 6]
>>> ['Ni!'] * 4                   # Repetition
['Ni!', 'Ni!', 'Ni!', 'Ni!']
```

Although the + operator works the same for lists and strings, it's important to know that it expects the same sort of sequence on both sides—otherwise, you get a type error when the code runs. For instance, you cannot concatenate a list and a string unless you first convert the list to a string (using tools such as str or % formatting) or convert the string to a list (the list built-in function does the trick):

```
>>> str([1, 2]) + "34"            # Same as "[1, 2]" + "34"
'[1, 2]34'
>>> [1, 2] + list("34")           # Same as [1, 2] + ["3", "4"]
[1, 2, '3', '4']
```

List Iteration and Comprehensions
More generally, lists respond to all the sequence operations we used on strings in the prior module, including iteration tools:

```
>>> 3 in [1, 2, 3]                # Membership
True
>>> for x in [1, 2, 3]:
... print(x, end=' ')             # Iteration (2.X uses: print x,)
...
1 2 3
```

As also introduced briefly, the map built-in function does similar work, but applies a function to items in a sequence and collects all the results in a new list:

```
>>> list(map(abs, [−1, −2, 0, 1, 2]))       # Map a function across a sequence
[1, 2, 0, 1, 2]
```

Because we're not quite ready for the full iteration story, we'll postpone further details for now, but watch for a similar comprehension expression for dictionaries later in this module.

Indexing, Slicing, and Matrixes
Because lists are sequences, indexing and slicing work the same way for lists as they do for strings. However, the result of indexing a list is whatever type of object lives at the offset you specify, while slicing a list always returns a new list:

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[2]                    # Offsets start at zero
```

```
'SPAM!'
>>> L[−2]                # Negative: count from the right
'Spam'
>>> L[1:]                # Slicing fetches sections
['Spam', 'SPAM!']
```

One note here: because you can nest lists and other object types within lists, you will sometimes need to string together index operations to go deeper into a data structure. For example, one of the simplest ways to represent matrixes (multidimensional arrays) in Python is as lists with nested sublists. Here's a basic 3 × 3 two-dimensional list-based array:

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

With one index, you get an entire row (really, a nested sublist), and with two, you get an item within the row:

```
>>> matrix[1]
[4, 5, 6]
>>> matrix[1][1]
5
>>> matrix[2][0]
7
>>> matrix = [[1, 2, 3],
... [4, 5, 6],
... [7, 8, 9]]
>>> matrix[1][1]
5
```

Notice in the preceding interaction that lists can naturally span multiple lines if you want them to because they are contained by a pair of brackets; the "..."s here are Python's continuation line prompt .

<u>Changing Lists in Place</u>

Because lists are mutable, they support operations that change a list object in place. That is, the operations in this section all modify the list object directly—overwriting its former value—without requiring that you make a new copy, as you had to for strings.

Because Python deals only in object references, this distinction between changing an object in place and creating a new object matters; as discussed, if you change an object in place, you might impact more than one reference to it at the same time.

<u>Index and slice assignments</u>

When using a list, you can change its contents by assigning to either a particular item (offset) or an entire section (slice):

```
>>> L = ['spam', 'Spam', 'SPAM!']
>>> L[1] = 'eggs'                # Index assignment
>>> L
['spam', 'eggs', 'SPAM!']
>>> L[0:2] = ['eat', 'more']     # Slice assignment: delete+insert
>>> L # Replaces items 0,1
['eat', 'more', 'SPAM!']
```

Both index and slice assignments are in-place changes—they modify the subject list directly, rather than generating a new list object for the result. Index assignment in Python works much as it does in C and most other languages: Python replaces the single object reference at the designated offset with a new one.

Slice assignment, the last operation in the preceding example, replaces an entire section of a list in a single step. Because it can be a bit complex, it is perhaps best thought of as a combination of two steps:

1. **Deletion**. The slice you specify to the left of the = is deleted.
2. **Insertion**. The new items contained in the iterable object to the right of the = are inserted into the list on the left, at the place where the old slice was deleted.

This isn't what really happens, but it can help clarify why the number of items inserted doesn't have to match the number of items deleted. For instance, given a list L of two or more items, an assignment L[1:2]=[4,5] replaces one item with two—Python first deletes the one-item slice at [1:2] (from offset 1, up to but not including offset 2), then inserts both 4 and 5 where the deleted slice used to be.

This also explains why the second slice assignment in the following is really an insert—Python replaces an empty slice at [1:1] with two items; and why the third is really a deletion—Python deletes the slice (the item at offset 1), and then inserts nothing:

```
>>> L = [1, 2, 3]
>>> L[1:2] = [4, 5]        # Replacement/insertion
>>> L
[1, 4, 5, 3]
>>> L[1:1] = [6, 7]        # Insertion (replace nothing)
>>> L
[1, 6, 7, 4, 5, 3]
>>> L[1:2] = []            # Deletion (insert nothing)
>>> L
[1, 7, 4, 5, 3]
```

In effect, slice assignment replaces an entire section, or "column," all at once—even if the column or its replacement is empty. Because the length of the sequence being assigned does not have to match the length of the slice being assigned to, slice assignment can be used to replace (by overwriting), expand (by inserting), or shrink (by deleting) the subject list. It's a powerful operation, but frankly, one that you may not see very often in practice. There are often more straightforward and mnemonic ways to replace, insert, and delete (concatenation, and the insert, pop, and remove list methods, for example), which Python programmers tend to prefer in practice.

On the other hand, this operation can be used as a sort of in-place concatenation at the front of the list—per the next section's method coverage, something the list's extend does more mnemonically at list end:

```
>>> L = [1]
>>> L[:0] = [2, 3, 4]              # Insert all at :0, an empty slice at front
>>> L
[2, 3, 4, 1]
>>> L[len(L):] = [5, 6, 7]         # Insert all at len(L):, an empty slice at end
>>> L
[2, 3, 4, 1, 5, 6, 7]
>>> L.extend([8, 9, 10])           # Insert all at end, named method
>>> L
[2, 3, 4, 1, 5, 6, 7, 8, 9, 10]
```

List method calls

Python list objects like string also support type-specific method calls, many of which change the subject list in place:

```
>>> L = ['eat', 'more', 'SPAM!']
>>> L.append('please')             # Append method call: add item at end
>>> L
```

```
['eat', 'more', 'SPAM!', 'please']
>>> L.sort()                          # Sort list items ('S' < 'e')
>>> L
['SPAM!', 'eat', 'more', 'please']
```

Methods were introduced in brief, they are functions (really, object attributes that reference functions) that are associated with and act upon particular objects.

Methods provide type-specific tools; the list methods presented here, for instance, are generally available only for lists.

Perhaps the most commonly used list method is append, which simply tacks a single item (object reference) onto the end of the list. Unlike concatenation, append expects you to pass in a single object, not a list. The effect of L.append(X) is similar to L+[X], but while the former changes L in place, the latter makes a new list.3 The sort method orders the list's items here, but merits a section of its own.

More on sorting lists

Another commonly seen method, sort, orders a list in place; it uses Python standard comparison tests (here, string comparisons, but applicable to every object type), and by default sorts in ascending order. You can modify sort behavior by passing in keyword arguments—a special "name=value" syntax in function calls that specifies passing by name and is often used for giving configuration options.

In sorts, the reverse argument allows sorts to be made in descending instead of ascending order, and the key argument gives a one-argument function that returns the value to be used in sorting—the string object's standard lower case converter in the following (though its newer casefold may handle some types of Unicode text better):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort()                                    # Sort with mixed case
>>> L
['ABD', 'aBe', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower)                       # Normalize to lowercase
>>> L
['abc', 'ABD', 'aBe']
>>>
>>> L = ['abc', 'ABD', 'aBe']
>>> L.sort(key=str.lower, reverse=True)         # Change sort order
>>> L
['aBe', 'ABD', 'abc']
```

The sort key argument might also be useful when sorting lists of dictionaries, to pick out a sort key by indexing each dictionary.

One warning here: beware that append and sort change the associated list object in place, but don't return the list as a result (technically, they both return a value called None). If you say something like L=L.append(X), you won't get the modified value of L (in fact, you'll lose the reference to the list altogether!). When you use attributes such as append and sort, objects are changed as a side effect, so there's no reason to reassign.

Partly because of such constraints, sorting is also available in recent Pythons as a builtin function, which sorts any collection (not just lists) and returns a new list for the result (instead of in-place changes):

```
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted(L, key=str.lower, reverse=True)      # Sorting built-in
```

```
['aBe', 'ABD', 'abc']
>>> L = ['abc', 'ABD', 'aBe']
>>> sorted([x.lower() for x in L], reverse=True) # Pretransform items: differs!
['abe', 'abd', 'abc']
```

Notice the last example here—we can convert to lowercase prior to the sort with a list comprehension, but the result does not contain the original list's values as it does with the key argument. The latter is applied temporarily during the sort, instead of changing the values to be sorted altogether. As we move along, we'll see contexts in which the sorted built-in can sometimes be more useful than the sort method.

Other common list methods

Like strings, lists have other methods that perform other specialized operations. For instance, reverse reverses the list in-place, and the extend and pop methods insert multiple items at and delete an item from the end of the list, respectively. There is also a reversed built-in function that works much like sorted and returns a new result object, but it must be wrapped in a list call in both 2.X and 3.X here because its result is an iterator that produces results on demand:

```
>>> L = [1, 2]
>>> L.extend([3, 4, 5])          # Add many items at end (like in-place +)
>>> L
[1, 2, 3, 4, 5]
>>> L.pop()                      # Delete and return last item (by default: −1)
5
>>> L
[1, 2, 3, 4]
>>> L.reverse()                  # In-place reversal method
>>> L
[4, 3, 2, 1]
>>> list(reversed(L))            # Reversal built-in with a result (iterator)
[1, 2, 3, 4]
```

Technically, the extend method always iterates through and adds each item in an iterable object, whereas append simply adds a single item as is without iterating through it. For now, it's enough to know that extend adds many items, and append adds one. In some types of programs, the list pop method is often used in conjunction with append to implement a quick lastin-first-out (LIFO) stack structure. The end of the list serves as the top of the stack:

```
>>> L = []
>>> L.append(1)          # Push onto stack
>>> L.append(2)
>>> L
[1, 2]
>>> L.pop()              # Pop off stack
2
>>> L
[1]
```

The pop method also accepts an optional offset of the item to be deleted and returned (the default is the last item at offset −1). Other list methods remove an item by value (remove), insert an item at an offset (insert), count the number of occurrences (count), and search for an item's offset (index—a search for the index of an item, not
to be confused with indexing!):

```
>>> L = ['spam', 'eggs', 'ham']
>>> L.index('eggs')              # Index of an object (search/find)
1
```

```
>>> L.insert(1, 'toast')           # Insert at position
>>> L
['spam', 'toast', 'eggs', 'ham']
>>> L.remove('eggs')               # Delete by value
>>> L
['spam', 'toast', 'ham']
>>> L.pop(1)                       # Delete by position
'toast'
>>> L
['spam', 'ham']
>>> L.count('spam')                # Number of occurrences
1
```

Note that unlike other list methods, count and index do not change the list itself, but return information about its content. See other documentation sources or experiment with these calls interactively on your own to learn more about list methods.

<u>Other common list operations</u>
Because lists are mutable, you can use the del statement to delete an item or section in place:

```
>>> L = ['spam', 'eggs', 'ham', 'toast']
>>> del L[0]              # Delete one item
>>> L
['eggs', 'ham', 'toast']
>>> del L[1:]             # Delete an entire section
>>> L # Same as L[1:] = []
['eggs']
```

As we saw earlier, because slice assignment is a deletion plus an insertion, you can also delete a section of a list by assigning an empty list to a slice (L[i:j]=[]); Python deletes

the slice named on the left, and then inserts nothing. Assigning an empty list to an index, on the other hand, just stores a reference to the empty list object in the specified slot, rather than deleting an item:

```
>>> L = ['Already', 'got', 'one']
>>> L[1:] = []
>>> L
['Already']
>>> L[0] = []
>>> L
[[]]
```

Although all the operations just discussed are typical, there may be additional list methods and operations not illustrated here. The method set, for example, may change over time, and in fact has in Python 3.3—its new L.copy() method makes a top-level copy of the list, much like L[:] and list(L), but is symmetric with copy in sets and dictionaries. For a comprehensive and up-to-date list of type tools, you should always consult Python's manuals, Python's dir and help functions, or one of the reference texts mentioned in the preface. And because it's such a common hurdle, I'd also like to remind you again that all the in-place change operations discussed here work only for mutable objects: they won't work on strings, no matter how hard you try. Mutability is an inherent property of each object type.

Dictionaries

Along with lists, dictionaries are one of the most flexible built-in data types in Python. If you think of lists as ordered collections of objects, you can think of dictionaries as unordered collections; the chief distinction is that in dictionaries, items are stored and fetched by key, instead of by positional offset. While lists can serve roles similar to arrays in other languages, dictionaries take the place of records, search tables, and any ther sort of aggregation where item names are more meaningful than item positions.

For example, dictionaries can replace many of the searching algorithms and data structures you might have to implement manually in lower-level languages—as a highly optimized built-in type, indexing a dictionary is a very fast search operation. Dictionaries also sometimes do the work of records, structs, and symbol tables used in other languages; can be used to represent sparse (mostly empty) data structures; and much more. Here's a rundown of their main properties. Python dictionaries are:

Accessed by key, not offset position

Dictionaries are sometimes called associative arrays or hashes (especially by users of other scripting languages). They associate a set of values with keys, so you can fetch an item out of a dictionary using the key under which you originally stored it. You use the same indexing operation to get components in a dictionary as you do in a list, but the index takes the form of a key, not a relative offset.

Unordered collections of arbitrary objects

Unlike in a list, items stored in a dictionary aren't kept in any particular order; in fact, Python pseudo-randomizes their left-to-right order to provide quick lookup. Keys provide the symbolic (not physical) locations of items in a dictionary.

Variable-length, heterogeneous, and arbitrarily nestable

Like lists, dictionaries can grow and shrink in place (without new copies being made), they can contain objects of any type, and they support nesting to any depth (they can contain lists, other dictionaries, and so on). Each key can have just one associated value, but that value can be a collection of multiple objects if needed, and a given value can be stored under any number of keys.

Of the category "mutable mapping"

You can change dictionaries in place by assigning to indexes (they are mutable), but they don't support the sequence operations that work on strings and lists. Because dictionaries are unordered collections, operations that depend on a fixed positional order (e.g., concatenation, slicing) don't make sense. Instead, dictionaries are the only built-in, core type representatives of the mapping category—objects that map keys to values. Other mappings in Python are created by imported modules.

Tables of object references (hash tables)

If lists are arrays of object references that support access by position, dictionaries are unordered tables of object references that support access by key. Internally, dictionaries are implemented as hash tables (data structures that support very fast retrieval), which start small and grow on demand. Moreover, Python employs optimized hashing algorithms to find keys, so retrieval is quick. Like lists, dictionaries store object references (not copies, unless you ask for them explicitly).

For reference and preview again, Table below summarizes some of the most common and representative dictionary operations, and is relatively complete as of Python 3.3. As usual, though, see the library manual or run a dir(dict) or help(dict) call for a complete list—dict is the name of the type. When coded as a literal expression, a dictionary is written as a series of key:value pairs, separated by commas, enclosed in curly braces.4 An empty dictionary is an empty set of braces, and you can nest dictionaries by simply coding one as a value inside another dictionary, or within a list or tuple.

Table: *Common dictionary literal and operations*	
Operation	**Interpretation**
D={}	Empty dictionary
D={name': 'Bob', 'age': 40}	Two-item dictionary
E={'cto': {name:'Bob', 'age': 40}}	Nesting
D = dict(name='Bob', age=40)	Alternative construction techniques
D = dict([('name', 'Bob'), ('age', 40)])	keywords, key/value pairs, zipped key/value pairs, key lists
D = dict(zip(keyslist, valueslist))	
D = dict.fromkeys(['name', 'age'])	
D['name']	Indexing by key
E['cto']['age']	
'age' in D	Membership: key present test
D.keys()	Methods: all keys,
D.values()	all values,
D.items()	all key+value tuples,
D.copy()	copy (top-level),
D.clear()	clear (remove all items),
D.update(D2)	merge by keys,
D.get(key, default?)	fetch by key, if absent default (or None),
D.pop(key, default?)	remove by key, if absent default (or error)
D.setdefault(key, default?)	fetch by key, if absent set default (or None),
D.popitem()	remove/return any (key, value) pair; etc.
len(D)	Length: number of stored entries
D[key] = 42	Adding/changing keys
del D[key]	Deleting entries by key
list(D.keys())	Dictionary views (Python 3.X)
D1.keys() & D2.keys()	
D.viewkeys(), D.viewvalues()	Dictionary views (Python 2.7)
D = {x: x*2 for x in range(10)}	Dictionary comprehensions (Python 3.X, 2.7)

Dictionaries in action

As Table above suggests, dictionaries are indexed by key, and nested dictionary entries are referenced by a series of indexes (keys in square brackets). When Python creates a dictionary, it stores its items in any left-to-right order it chooses; to fetch a value back, you supply the key with which it is associated, not its relative position. Let’s go back to the interpreter to get a feel for some of the dictionary operations in Table above.

Basic Dictionary Operations

In normal operation, you create dictionaries with literals and store and access items by key with indexing:

```
% python
>>> D = {'spam': 2, 'ham': 1, 'eggs': 3}   # Make a dictionary
>>> D['spam']                              # Fetch a value by key
2
>>> D                                      # Order is "scrambled"
{'eggs': 3, 'spam': 2, 'ham': 1}
```

Here, the dictionary is assigned to the variable D; the value of the key 'spam' is the integer 2, and so on. We use the same square bracket syntax to index dictionaries by key as we did to index lists by offset, but here it means access by key, not by position.

Notice the end of this example—much like sets, the left-to-right order of keys in a dictionary will almost always be different from what you originally typed. This is on purpose: to implement fast key lookup (a.k.a. hashing), keys need

to be reordered in memory. That's why operations that assume a fixed left-to-right order (e.g., slicing, concatenation) do not apply to dictionaries; you can fetch values only by key, not by position. Technically, the ordering is pseudo-random—it's not truly random (you might be able to decipher it given Python's source code and a lot of time to kill), but it's arbitrary, and might vary per release and platform, and even per interactive session in Python 3.3.

The built-in len function works on dictionaries, too; it returns the number of items stored in the dictionary or, equivalently, the length of its keys list. The dictionary in membership operator allows you to test for key existence, and the keys method returns all the keys in the dictionary. The latter of these can be useful for processing dictionaries sequentially, but you shouldn't depend on the order of the keys list. Because the keys result can be used as a normal list, however, it can always be sorted if order matters (more on sorting and dictionaries later):

```
>>> len(D)              # Number of entries in dictionary
3
>>> 'ham' in D          # Key membership test alternative
True
>>> list(D.keys())      # Create a new list of D's keys
['eggs', 'spam', 'ham']
```

Observe the second expression in this listing. As mentioned earlier, the in membership test used for strings and lists also works on dictionaries—it checks whether a key is stored in the dictionary. Technically, this works because dictionaries define iterators that step through their keys lists automatically. Other types provide iterators that reflect their common uses; files, for example, have iterators that read line by line.

Also note the syntax of the last example in this listing. We have to enclose it in a list call in Python 3.X for similar reasons—keys in 3.X returns an iterable object, instead of a physical list. The list call forces it to produce all its values at once so we can print them interactively, though this call isn't required some other contexts. In 2.X, keys builds and returns an actual list, so the list call isn't even needed to display a result.

Changing Dictionaries in Place

Let's continue with our interactive session. Dictionaries, like lists, are mutable, so you can change, expand, and shrink them in place without making new dictionaries: simply assign a value to a key to change or create an entry. The del statement works here, too; it deletes the entry associated with the key specified as an index. Notice also the nesting of a list inside a dictionary in this example (the value of the key 'ham'). All collection data types in Python can nest inside each other arbitrarily:

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
>>> D['ham'] = ['grill', 'bake', 'fry']      # Change entry (value=list)
>>> D
{'eggs': 3, 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> del D['eggs']                            # Delete entry
>>> D
{'spam': 2, 'ham': ['grill', 'bake', 'fry']}
>>> D['brunch'] = 'Bacon'                    # Add new entry
>>> D
{'brunch': 'Bacon', 'spam': 2, 'ham': ['grill', 'bake', 'fry']}
```

Like lists, assigning to an existing index in a dictionary changes its associated value. Unlike lists, however, whenever you assign a new dictionary key (one that hasn't been assigned before) you create a new entry in the dictionary, as was done in the previous example for the key 'brunch'. This doesn't work for lists because you can only assign to existing list offsets—Python considers an offset beyond the end of a list out of bounds and raises an error. To expand a list, you need to use tools such as the append method or slice assignment instead.

More Dictionary Methods

Dictionary methods provide a variety of type-specific tools. For instance, the dictionary values and items methods return all of the dictionary's values and (key,value) pair tuples, respectively; along with keys, these are useful in loops that need to step through dictionary entries one by one (we'll start coding examples of such loops in the next section). As for keys, these two methods also iterable objects in 3.X, so wrap them in a list call there to collect their values all at once for display:

```
>>> D = {'spam':2, 'ham':1, 'egg':3}
>>> list(D.values())
[3,2,1]
>>> list(D.items())
[('eggs', 3), ('spam', 2), ('ham', 1)]
```

In realistic programs that gather data as they run, you often won't be able to predict what will be in a dictionary before the program is launched, much less when it's coded. Fetching a nonexistent key is normally an error, but the get method returns a default value—None, or a passed-in default—if the key doesn't exist. It's an easy way to fill in a default for a key that isn't present, and avoid a missing-key error when your program can't anticipate contents ahead of time:

```
>>> D.get('spam')                 # A key that is there
2
>>> print(D.get('toast'))         # A key that is missing
None
>>> D.get('toast', 88)
88
```

The update method provides something similar to concatenation for dictionaries, though it has nothing to do with left-to-right ordering (again, there is no such thing in dictionaries). It merges the keys and values of one dictionary into another, blindly overwriting values of the same key if there's a clash:

```
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
>>> D2 = {'toast':4, 'muffin':5}  # Lots of delicious scrambled order here
>>> D.update(D2)
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
```

Notice how mixed up the key order is in the last result; again, that's just how dictionaries work. Finally, the dictionary pop method deletes a key from a dictionary and returns the value it had. It's similar to the list pop method, but it takes a key instead of n optional position:

```
# pop a dictionary by key
>>> D
{'eggs': 3, 'muffin': 5, 'toast': 4, 'spam': 2, 'ham': 1}
>>> D.pop('muffin')
5
>>> D.pop('toast')          # Delete and return from a key
4
>>> D
{'eggs': 3, 'spam': 2, 'ham': 1}
# pop a list by position
>>> L = ['aa', 'bb', 'cc', 'dd']
>>> L.pop()                 # Delete and return from the end
'dd'
>>> L
```

```
['aa', 'bb', 'cc']
>>> L.pop(1)                # Delete from a specific position
'bb'
>>> L
['aa', 'cc']
```

Dictionaries also provide a copy method, as it's a way to avoid the potential side effects of shared references to the same dictionary. In fact, dictionaries come with more methods than those listed in the previous table; see the Python library manual, dir and help, or other reference sources for a comprehensive list.

Dictionary Usage Notes

Dictionaries are fairly straightforward tools once you get the hang of them, but here are a few additional pointers and reminders you should be aware of when using them:

- **Sequence operations don't work.** Dictionaries are mappings, not sequences; because there's no notion of ordering among their items, things like concatenation (an ordered joining) and slicing (extracting a contiguous section) simply don't apply. In fact, Python raises an error when your code runs if you try to do such things.
- **Assigning to new indexes adds entries.** Keys can be created when you write a dictionary literal (embedded in the code of the literal itself), or when you assign values to new keys of an existing dictionary object individually. The end result is the same.
- **Keys need not always be strings.** Our examples so far have used strings as keys, but any other immutable objects work just as well. For instance, you can use integers as keys, which makes the dictionary look much like a list (when indexing, at least). Tuples may be used as dictionary keys too, allowing compound key values—such as dates and IP addresses—to have associated values. User-defined class instance objects (discussed in Part VI) can also be used as keys, as long as they have the proper protocol methods; roughly, they need to tell Python that their values are "hashable" and thus won't change, as otherwise they would be useless as fixed keys. Mutable objects such as lists, sets, and other dictionaries don't work as keys, but are allowed as values.

Other Ways to Make Dictionaries

Other Ways to Make Dictionaries Finally, note that because dictionaries are so useful, more ways to build them have emerged over time. In Python 2.3 and later, for example, the last two calls to the dict constructor (really, type name) shown here have the same effect as the literal and keyassignment forms above them:

```
{'name': 'Bob', 'age': 40}                # Traditional literal expression
D = {}                                    # Assign by keys dynamically
D['name'] = 'Bob'
D['age'] = 40
dict(name='Bob', age=40)                  # dict keyword argument form
dict([('name', 'Bob'), ('age', 40)])      # dict key/value tuples form
```

All four of these forms create the same two-key dictionary, but they are useful in differing circumstances:

- The first is handy if you can spell out the entire dictionary ahead of time.
- The second is of use if you need to create the dictionary one field at a time on the fly.
- The third involves less typing than the first, but it requires all keys to be strings.
- The last is useful if you need to build up keys and values as sequences at runtime.

We met keyword arguments earlier when sorting; the third form illustrated in this code listing has become especially popular in Python code today, since it has less syntax (and hence there is less opportunity for mistakes). As suggested in previous table, the last form in the listing is also commonly used in conjunction with the zip function, to combine separate lists of keys and values obtained dynamically at runtime (parsed out of a data file's columns, for instance):

```
dict(zip(keyslist, valueslist))       # Zipped key/value tuples form (ahead)
```

More on zipping dictionary keys in the next section. Provided all the key's values are the same initially, you can also create a dictionary with this special form—simply pass in a list of keys and an initial value for all of the values (the default is None):

```
>>> dict.fromkeys(['a', 'b'], 0)
{'a': 0, 'b': 0}
```

Although you could get by with just literals and key assignments at this point in your Python career, you'll probably find uses for all of these dictionary-creation forms as you start applying them in realistic, flexible, and dynamic Python programs.

The listings in this section document the various ways to create dictionaries in both Python 2.X and 3.X. However, there is yet another way to create dictionaries, available only in Python 3.X and 2.7: the dictionary comprehension expression. To see how this last form looks, we need to move on to the next and final section of this module.

Dictionary Changes in Python 3.X and 2.7

This module has so far focused on dictionary basics that span releases, but the dictionary's functionality has mutated in Python 3.X. If you are using Python 2.X code, you may come across some dictionary tools that either behave differently or are missing altogether in 3.X. Moreover, 3.X coders have access to additional dictionary tools not available in 2.X, apart from two back-ports to 2.7.

Specifically, dictionaries in Python 3.X:

- Support a new dictionary comprehension expression, a close cousin to list and set comprehensions
- Return set-like iterable views instead of lists for the methods D.keys, D.values, and D.items
- Require new coding styles for scanning by sorted keys, because of the prior point
- No longer support relative magnitude comparisons directly—compare manually instead
- No longer have the D.has_key method—the in membership test is used instead

As later back-ports from 3.X, dictionaries in Python 2.7 (but not earlier in 2.X):

- Support item 1 in the prior list—dictionary comprehensions—as a direct back-port from 3.X
- Support item 2 in the prior list—set-like iterable views—but do so with special method names D.viewkeys, D.viewvalues, D.viewitems); their nonview methods return lists as before Because of this overlap, some of the material in this section pertains both to 3.X and 2.7, but is presented here in the context of 3.X extensions because of its origin. With that in mind, let's take a look at what's new in dictionaries in 3.X and 2.7.

Dictionary comprehensions in 3.X and 2.7

As mentioned at the end of the prior section, dictionaries in 3.X and 2.7 can also be created with dictionary comprehensions. Like the set comprehensions we met , dictionary comprehensions are available only in 3.X and 2.7 (not in 2.6 and earlier). Like the longstanding list comprehensions we met briefly and earlier in this module, they run an implied loop, collecting the key/value results of expressions on each iteration and using them to fill out a new dictionary. A loop variable allows the comprehension to use loop iteration values along the way.

To illustrate, a standard way to initialize a dictionary dynamically in both 2.X and 3.X is to combine its keys and values with zip, and pass the result to the dict call. The zip built-in function is the hook that allows us to construct a dictionary from key and value lists this way—if you cannot predict the set of keys and values in your code, you can always build them up as lists and zip them together. We'll study zip in detail after exploring statements; it's an iterable in 3.X, so we must wrap it in a list call to show its results there, but its basic usage is otherwise straightforward:

```
>>> list(zip(['a', 'b', 'c'], [1, 2, 3]))          # Zip together keys and values
[('a', 1), ('b', 2), ('c', 3)]
>>> D = dict(zip(['a', 'b', 'c'], [1, 2, 3]))  # Make a dict from zip result
>>> D
```

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

In Python 3.X and 2.7, though, you can achieve the same effect with a dictionary comprehension expression. The following builds a new dictionary with a key/value pair for every such pair in the zip result (it reads almost the same in Python, but with a bit more formality):

```
>>> D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
>>> D
{'b': 2, 'c': 3, 'a': 1}
```

Comprehensions actually require more code in this case, but they are also more general than this example implies—we can use them to map a single stream of values to dictionariesas well, and keys can be computed with expressions just like values:

```
>>> D = {x: x ** 2 for x in [1, 2, 3, 4]}  # Or: range(1, 5)
>>> D
{1: 1, 2: 4, 3: 9, 4: 16}
>>> D = {c: c * 4 for c in 'SPAM'}         # Loop over any iterable
>>> D
{'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}
>>> D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}
>>> D
{'eggs': 'EGGS!', 'spam': 'SPAM!', 'ham': 'HAM!'}
```

Dictionary comprehensions are also useful for initializing dictionaries from keys lists, in much the same way as the fromkeys method we met at the end of the preceding section:

```
>>> D = dict.fromkeys(['a', 'b', 'c'], 0)     # Initialize dict from keys
>>> D
{'b': 0, 'c': 0, 'a': 0}
>>> D = {k:0 for k in ['a', 'b', 'c']}          # Same, but with a comprehension
>>> D
{'b': 0, 'c': 0, 'a': 0}
>>> D = dict.fromkeys('spam')   # Other iterables, default value
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}
>>> D = {k: None for k in 'spam'}
>>> D
{'s': None, 'p': None, 'a': None, 'm': None}
```

Like related tools, dictionary comprehensions support additional syntax not shown here, including nested loops and if clauses. Unfortunately, to truly understand dictionary comprehensions, we need to also know more about iteration statements and concepts in Python, and we don’t yet have enough information to address that story well.

Dictionary views in 3.X (and 2.7 via new methods)
In 3.X the dictionary keys, values, and items methods all return view objects, whereas in 2.X they return actual result lists. This functionality is also available in Python 2.7, but in the guise of the special, distinct method names listed at the start of this section (2.7's normal methods still return simple lists, so as to avoid breaking existing 2.X code); because of this, I'll refer to this as a 3.X feature in this section.

View objects are iterables, which simply means objects that generate result items one at a time, instead of producing the result list all at once in memory. Besides being iterable, dictionary views also retain the original order of dictionary components, reflect future changes to the dictionary, and may support set operations. On the other hand, because they are not lists, they do not directly support operations like indexing or the list sort method, and do not display their items as a normal list when printed (they do show their components as of Python 3.1 but not as a list, and are still a divergence from 2.X).

We'll discuss the notion of iterables more formally later, but for our purposes here it's enough to know that we have to run the results of these three methods through the list built-in if we want to apply list operations or display their values. For example, in Python 3.3 (other version's outputs may differ slightly):

```
>>> D = dict(a=1, b=2, c=3)
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> K = D.keys()                    # Makes a view object in 3.X, not a list
>>> K
dict_keys(['b', 'c', 'a'])
>>> list(K)                         # Force a real list in 3.X if needed
['b', 'c', 'a']
>>> V = D.values()                  # Ditto for values and items views
>>> V
dict_values([2, 3, 1])
>>> list(V)
[2, 3, 1]
>>> D.items()
dict_items([('b', 2), ('c', 3), ('a', 1)])
>>> list(D.items())
[('b', 2), ('c', 3), ('a', 1)]
>>> K[0]                            # List operations fail unless converted
TypeError: 'dict_keys' object does not support indexing
>>> list(K)[0]
'b'
```

Apart from result displays at the interactive prompt, you will probably rarely even notice this change, because looping constructs in Python automatically force iterable objects to produce one result on each iteration:

```
>>> for k in D.keys(): print(k)     # Iterators used automatically in loops
...
b
c
a
```

In addition, 3.X dictionaries still have iterators themselves, which return successive keys—as in 2.X, it's still often not necessary to call keys directly:

```
>>> for key in D: print(key)        # Still no need to call keys() to iterate
...
b
```

```
c
a
```

Unlike 2.X’s list results, though, dictionary views in 3.X are not carved in stone when created—they dynamically reflect future changes made to the dictionary after the view object has been created:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> K = D.keys()
>>> V = D.values()
>>> list(K)                # Views maintain same order as dictionary
['b', 'c', 'a']
>>> list(V)
[2, 3, 1]
>>> del D['b']             # Change the dictionary in place
>>> D
{'c': 3, 'a': 1}
>>> list(K)                # Reflected in any current view objects
['c', 'a']
>>> list
```

Dictionary views and sets

Also unlike 2.X’s list results, 3.X’s view objects returned by the keys method are set-like and support common set operations such as intersection and union; values views are not set-like, but items results are if their (key, value) pairs are unique and hashable(immutable). Given that sets behave much like valueless dictionaries (and may even be coded in curly braces like dictionaries in 3.X and 2.7), this is a logical symmetry. As mentioned, set items are unordered, unique, and immutable, just like dictionary keys.

Here is what keys views look like when used in set operations (continuing the prior section’s session); dictionary value views are never set-like, since their items are not necessarily unique or immutable:

```
>>> K, V
(dict_keys(['c', 'a']), dict_values([3, 1]))
>>> K | {'x': 4}           # Keys (and some items) views are set-like
{'c', 'x', 'a'}
>>> V & {'x': 4}
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict'
>>> V & {'x': 4}.values()
TypeError: unsupported operand type(s) for &: 'dict_values' and 'dict_values'
```

In set operations, views may be mixed with other views, sets, and dictionaries; dictionaries are treated the same as their keys views in this context:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D.keys() & D.keys()          # Intersect keys views
{'b', 'c', 'a'}
>>> D.keys() & {'b'}             # Intersect keys and set
{'b'}
>>> D.keys() & {'b': 1}          # Intersect keys and dict
{'b'}
>>> D.keys() | {'b', 'c', 'd'}   # Union keys and set
{'b', 'c', 'a', 'd'}
```

Items views are set-like too if they are hashable—that is, if they contain only immutable objects:

```
>>> D = {'a': 1}
>>> list(D.items())                         # Items set-like if hashable
[('a', 1)]
>>> D.items() | D.keys()                    # Union view and view
{('a', 1), 'a'}
>>> D.items() | D                           # dict treated same as its keys
{('a', 1), 'a'}
>>> D.items() | {('c', 3), ('d', 4)}        # Set of key/value pairs
{('d', 4), ('a', 1), ('c', 3)}
>>> dict(D.items() | {('c', 3), ('d', 4)})  # dict accepts iterable sets too
{'c': 3, 'a': 1, 'd': 4}
```

Here, let's wrap up with three other quick coding notes for 3.X dictionaries.

Sorting dictionary keys in 3.X

First of all, because keys does not return a list in 3.X, the traditional coding pattern for scanning a dictionary by sorted keys in 2.X won't work in 3.X:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> Ks = D.keys()        # Sorting a view object doesn't work!
>>> Ks.sort()
AttributeError: 'dict_keys' object has no attribute 'sort'
```

To work around this, in 3.X you must either convert to a list manually or use the sorted call on either a keys view or the dictionary itself:

```
>>> Ks = list(Ks)                          # Force it to be a list and then sort
>>> Ks.sort()
>>> for k in Ks: print(k, D[k])            # 2.X: omit outer parens in prints
...
a 1
b 2
c 3
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> Ks = D.keys()                          # Or you can use sorted() on the keys
>>> for k in sorted(Ks): print(k, D[k])    # sorted() accepts any iterable
... # sorted() returns its result
a 1
b 2
c 3
```

Of these, using the dictionary's keys iterator is probably preferable in 3.X, and works in 2.X as well:

```
>>> D
{'b': 2, 'c': 3, 'a': 1}    # Better yet, sort the dict directly
>>> for k in sorted(D): print(k, D[k])   # dict iterators return keys
...
a 1
b 2
c 3
```

Dictionary magnitude comparisons no longer work in 3.X

Secondly, while in Python 2.X dictionaries may be compared for relative magnitude directly with <, >, and so on, in Python 3.X this no longer works. However, you can simulate it by comparing sorted keys lists manually:

```
sorted(D1.items()) < sorted(D2.items())       # Like 2.X D1 < D2
```

Dictionary equality tests (e.g., D1 == D2) still work in 3.X, though. Since we'll revisit this near the end of the next module in the context of comparisons at large, we'll postpone further details here.

The has_key method is dead in 3.X: Long live in!

Finally, the widely used dictionary has_key key presence test method is gone in 3.X. Instead, use the in membership expression, or a get with a default test (of these, in is generally preferred):

```
>>> D
{'b': 2, 'c': 3, 'a': 1}
>>> D.has_key('c')        # 2.X only: True/False
AttributeError: 'dict' object has no attribute 'has_key'
>>> 'c' in D              # Required in 3.X
True
>>> 'x' in D              # Preferred in 2.X today
False
>>> if 'c' in D: print('present', D['c'])       # Branch on result
...
present 3
>>> print(D.get('c'))      # Fetch with default
3
>>> print(D.get('x'))
None
>>> if D.get('c') != None: print('present', D['c'])  # Another option
...
present 3
```

To summarize, the dictionary story changes substantially in 3.X. If you work in 2.X and care about 3.X compatibility (or suspect that you might someday), here are some pointers. Of the 3.X changes we've met in this section:

- The first (dictionary comprehensions) can be coded only in 3.X and 2.7.
- The second (dictionary views) can be coded only in 3.X, and with special method names in 2.7.

However, the last three techniques—sorted, manual comparisons, and in—can be coded in 2.X today to ease 3.X migration in the future.

Module 8 - Tuples, Files & Everything Else

This module rounds out our in-depth tour of the core object types in Python by exploring the tuple, a collection of other objects that cannot be changed, and the file, an interface to external files on your computer. As you'll see, the tuple is a relatively simple object that largely performs operations you've already learned about for strings and lists. The file object is a commonly used and full-featured tool for processing files on your computer.

This module also concludes this part of the course by looking at properties common to all the core object types we've met—the notions of equality, comparisons, object copies, and so on. We'll also briefly explore other object types in Python's toolbox, including the None placeholder and the namedtuple hybrid; as you'll see, although we've covered all the primary built-in types, the object story in Python is broader than I've implied thus far. Finally, we'll close this part of the course by taking a look at a set of common object type pitfalls and exploring some exercises that will allow you to experiment with the ideas you've learned.

Tuples

The last collection type in our survey is the Python tuple. Tuples construct simple groups of objects. They work exactly like lists, except that tuples can't be changed in place (they're immutable) and are usually written as a series of items in parentheses, not square brackets. Although they don't support as many methods, tuples share most of their properties with lists. Here's a quick look at the basics. Tuples are:

Ordered collections of arbitrary objects

Like strings and lists, tuples are positionally ordered collections of objects (i.e., they maintain a left-to-right order among their contents); like lists, they can embed any kind of object.

Accessed by offset

Like strings and lists, items in a tuple are accessed by offset (not by key); they support all the offset-based access operations, such as indexing and slicing.

Of the category "immutable sequence"

Like strings and lists, tuples are sequences; they support many of the same operations. However, like strings, tuples are immutable; they don't support any of the in-place change operations applied to lists.

Fixed-length, heterogeneous, and arbitrarily nestable

Because tuples are immutable, you cannot change the size of a tuple without making a copy. On the other hand, tuples can hold any type of object, including other compound objects (e.g., lists, dictionaries, other tuples), and so support arbitrary nesting.

Arrays of object references

Like lists, tuples are best thought of as object reference arrays; tuples store access points to other objects (references), and indexing a tuple is relatively quick.

Table below highlights common tuple operations. A tuple is written as a series of objects (technically, expressions that generate objects), separated by commas and normally enclosed in parentheses. An empty tuple is just a parentheses pair with nothing inside.

Table: *Common tuple literals and operations*

Operation	Interpretation
()	An empty tuple
T = (0,)	A one-item tuple (not an expression)
T = (0, 'Ni', 1.2, 3)	A four-item tuple
T = 0, 'Ni', 1.2, 3	Another four-item tuple (same as prior line)
T = ('Bob', ('dev', 'mgr'))	Nested tuples
T = tuple('spam')	Tuple of items in an iterable
T[i]	Index, index of index, slice, length
T[i][j]	
T[i:j]	
len(T)	
T1 + T2	Concatenate, repeat
T * 3	

for x in T: print(x)	Iteration, membership
'spam' in T	
[x ** 2 for x in T]	
T.index('Ni')	Methods in 2.6, 2.7, and 3.X: search, count
T.count('Ni')	
namedtuple('Emp', ['name', 'jobs'])	Named tuple extension type

Tuples in Action

As usual, let's start an interactive session to explore tuples at work. Notice in Table above that tuples do not have all the methods that lists have (e.g., an append call won't work here). They do, however, support the usual sequence operations that we saw for both strings and lists:

```
>>> (1, 2) + (3, 4)         # Concatenation
(1, 2, 3, 4)
>>> (1, 2) * 4              # Repetition
(1, 2, 1, 2, 1, 2, 1, 2)
>>> T = (1, 2, 3, 4)        # Indexing, slicing
>>> T[0], T[1:3]
(1, (2, 3))
```

Tuple syntax peculiarities: Commas and parentheses

The second and fourth entries in Table above merit a bit more explanation. Because parentheses can also enclose expressions, you need to do something special to tell Python when a single object in parentheses is a tuple object and not a simple expression. If you really want a single-item tuple, simply add a trailing comma after the single item, before the closing parenthesis:

```
>>> x = (40)     # An integer!
>>> x
>>> y = (40,)    # A tuple containing an integer
>>> y
(40,)
```

As a special case, Python also allows you to omit the opening and closing parentheses for a tuple in contexts where it isn't syntactically ambiguous to do so. For instance, the fourth line of pevious Table simply lists four items separated by commas. In the context of an assignment statement, Python recognizes this as a tuple, even though it doesn't have parentheses.

Now, some people will tell you to always use parentheses in your tuples, and some will tell you to never use parentheses in tuples (and still others have lives, and won't tell you what to do with your tuples!). The most common places where the parentheses are required for tuple literals are those where:

- Parentheses matter—within a function call, or nested in a larger expression.
- Commas matter—embedded in the literal of a larger data structure like a list or dictionary, or listed in a Python 2.X print statement.

In most other contexts, the enclosing parentheses are optional. For beginners, the best advice is that it's probably easier to use the parentheses than it is to remember when they are optional or required. Many programmers (myself included) also find that parentheses tend to aid script readability by making the tuples more explicit and obvious, but your mileage may vary.

Conversions, methods, and immutability

Apart from literal syntax differences, tuple operations (the middle rows in previous Table) are identical to string and list operations. The only differences worth noting are that the +, *, and slicing operations return new tuples when applied to tuples, and that tuples don't provide the same methods you saw for strings, lists, and dictionaries. If you want to sort a tuple, for example, you'll usually have to either first convert it to a list to gain access to a sorting method call and make it a mutable object, or use the newer sorted built-in that accepts any sequence object:

```
>>> T = ('cc', 'aa', 'dd', 'bb')
>>> tmp = list(T)          # Make a list from a tuple's items
>>> tmp.sort()             # Sort the list
>>> tmp
['aa', 'bb', 'cc', 'dd']
>>> T = tuple(tmp)         # Make a tuple from the list's items
>>> T
('aa', 'bb', 'cc', 'dd')
>>> sorted(T)              # Or use the sorted built-in, and save two steps
['aa', 'bb', 'cc', 'dd']
```

Here, the list and tuple built-in functions are used to convert the object to a list and then back to a tuple; really, both calls make new objects, but the net effect is like a conversion. List comprehensions can also be used to convert tuples. The following, for example, makes a list from a tuple, adding 20 to each item along the way:

```
>>> T = (1, 2, 3, 4, 5)
>>> L = [x + 20 for x in T]
>>> L
[21, 22, 23, 24, 25]
```

List comprehensions are really sequence operations—they always build new lists, but they may be used to iterate over any sequence objects, including tuples, strings, and other lists. As we'll see later in the course, they even work on some things that are not physically stored sequences—any iterable objects will do, including files, which are automatically read line by line. Given this, they may be better called iteration tools. Although tuples don't have the same methods as lists and strings, they do have two of their own as of Python 2.6 and 3.0—index and count work as they do for lists, but they are defined for tuple objects:

```
>>> T = (1, 2, 3, 2, 4, 2)        # Tuple methods in 2.6, 3.0, and later
>>> T.index(2)                    # Offset of first appearance of 2
1
>>> T.index(2, 2)                 # Offset of appearance after offset 2
3
>>> T.count(2)                    # How many 2s are there?
3
```

Prior to 2.6 and 3.0, tuples have no methods at all—this was an old Python convention for immutable types, which was violated years ago on grounds of practicality with strings, and more recently with both numbers and tuples. Also, note that the rule about tuple immutability applies only to the top level of the tuple itself, not to its contents. A list inside a tuple, for instance, can be changed as usual:

```
>>> T = (1, [2, 3], 4)
>>> T[1] = 'spam'          # This fails: can't change tuple itself
TypeError: object doesn't support item assignment
>>> T[1][0] = 'spam'       # This works: can change mutables inside
>>> T
(1, ['spam', 3], 4)
```

For most programs, this one-level-deep immutability is sufficient for common tuple roles. Which, coincidentally, brings us to the next section.

Files
You may already be familiar with the notion of files, which are named storage compartments on your computer that are managed by your operating system. The last major built-in object type that we'll examine on our object types tour provides a way to access those files inside Python programs.

In short, the built-in open function creates a Python file object, which serves as a link to a file residing on your machine. After calling open, you can transfer strings of data to and from the associated external file by calling the returned file object's methods.

Compared to the types you've seen so far, file objects are somewhat unusual. They are considered a core type because they are created by a built-in function, but they're not numbers, sequences, or mappings, and they don't respond to expression operators; they export only methods for common file-processing tasks. Most file methods are concerned with performing input from and output to the external file associated with a file object, but other file methods allow us to seek to a new position in the file, flush output buffers, and so on. Table below summarizes common file operations.

Table: *Common file operations*

Operation	Interpretation
output = open(r'C:\spam', 'w')	Create output file ('w' means write)
input = open('data', 'r')	Create input file ('r' means read)
input = open('data')	Same as prior line ('r' is the default)
aString = input.read()	Read entire file into a single string
aString = input.read(N)	Read up to next N characters (or bytes) into a string
aString = input.readline()	Read next line (including \n newline) into a string
aList = input.readlines()	Read entire file into list of line strings (with \n)
output.write(aString)	Write a string of characters (or bytes) into file
output.writelines(aList)	Write all line strings in a list into file
output.close()	Manual close (done for you when file is collected)
output.flush()	Flush output buffer to disk without closing
anyFile.seek(N)	Change file position to offset N for next operation
for line in open('data'): use line	File iterators read line by line
open('f.txt', encoding='latin-1')	Python 3.X Unicode text files (str strings)
open('f.bin', 'rb')	Python 3.X bytes files (bytes strings)
codecs.open('f.txt', encoding='utf8')	Python 2.X Unicode text files (unicode strings)
open('f.bin', 'rb')	Python 2.X bytes files (str strings)

Opening Files
To open a file, a program calls the built-in open function, with the external filename first, followed by a processing mode. The call returns a file object, which in turn has methods for data transfer:

```
afile = open(filename, mode)
afile.method()
```

The first argument to open, the external filename, may include a platform-specific and absolute or relative directory path prefix. Without a directory path, the file is assumed to exist in the current working directory (i.e., where the script runs).

The filename may also contain non-ASCII Unicode characters that Python automatically translates to and from the underlying platform's encoding, or be provided as a pre-encoded byte string.

The second argument to open, processing mode, is typically the string 'r' to open for text input (the default), 'w' to create and open for text output, or 'a' to open for appending text to the end (e.g., for adding to logfiles). The processing mode argument can specify additional options:

- Adding a b to the mode string allows for binary data (end-of-line translations and 3.X Unicode encodings are turned off).
- Adding a + opens the file for both input and output (i.e., you can both read and write to the same file object, often in conjunction with seek operations to reposition in the file).

Both of the first two arguments to open must be Python strings. An optional third argument can be used to control output buffering—passing a zero means that output is unbuffered (it is transferred to the external file immediately on a write method call), and additional arguments may be provided for special types of files (e.g., an encoding for Unicode text files in Python 3.X).

We'll cover file fundamentals and explore some basic examples here, but we won't go into all file-processing mode options; as usual, consult the Python library manual for additional details.

Using Files

Once you make a file object with open, you can call its methods to read from or write to the associated external file. In all cases, file text takes the form of strings in Python programs; reading a file returns its content in strings, and content is passed to the write methods as strings. Reading and writing methods come in multiple flavors; Table 9-2 lists the most common. Here are a few fundamental usage notes:

File iterators are best for reading lines

Though the reading and writing methods in the table are common, keep in mind that probably the best way to read lines from a text file today is to not read the file at all, files also have an iterator that automatically reads one line at a time in a for loop, list comprehension, or other iteration context.

Content is strings, not objects

Notice in Table 9-2 that data read from a file always comes back to your script as a string, so you'll have to convert it to a different type of Python object if a string is not what you need. Similarly, unlike with the print operation, Python does not add any formatting and does not convert objects to strings automatically when you write data to a file—you must send an already formatted string. Because of this, the tools we have already met to convert objects to and from strings (e.g., int, float, str, and the string formatting expression and method) come in handy when dealing with files.

Python also includes advanced standard library tools for handling generic object storage (the pickle module), for dealing with packed binary data in files (the struct module), and for processing special types of content such as JSON, XML, and CSV text. We'll see these at work later in this module and course, but Python's manuals document them in full.

Files are buffered and seekable

By default, output files are always buffered, which means that text you write may not be transferred from memory to disk immediately—closing a file, or running its flush method, forces the buffered data to disk. You can avoid buffering with extra open arguments, but it may impede performance. Python files are also randomaccess on a byte offset basis—their seek method allows your scripts to jump around to read and write at specific locations.

close is often optional: auto-close on collection

Calling the file close method terminates your connection to the external file, releases its system resources, and flushes its buffered output to disk if any is still in memory. As discussed, in Python an object's memory space is automatically reclaimed as soon as the object is no longer referenced anywhere in the program. When file objects are reclaimed, Python also automatically closes the files if they are still open (this also happens when a program shuts down). This means you don't always need to manually close your files in standard Python, especially those in simple scripts with short runtimes, and temporary files used by a single line or expression.

On the other hand, including manual close calls doesn't hurt, and may be a good habit to form, especially in long-running systems. Strictly speaking, this auto-closeon-collection feature of files is not part of the language definition—it may change over time, may not happen when you expect it to in interactive shells, and may not work the same in other Python implementations whose garbage collectors may not reclaim and close files at the same points as standard CPython. In fact, when many files are opened within loops, Pythons other than CPython may require close calls to free up system resources immediately, before garbage collection can get around to freeing objects. Moreover, close calls may sometimes be required to flush buffered output of file objects not

yet reclaimed. For an alternative way to guarantee automatic file closes, also see this section's later discussion of the file object's context manager, used with the with/as statement in Python 2.6, 2.7, and 3.X.

Files in Action

Let's work through a simple example that demonstrates file-processing basics. The following code begins by opening a new text file for output, writing two lines (strings terminated with a newline marker, \n), and closing the file. Later, the example opens the same file again in input mode and reads the lines back one at a time with read line. Notice that the third readline call returns an empty string; this is how Python file methods tell you that you've reached the end of the file (empty lines in the file come back as strings containing just a newline character, not as empty strings). Here's the complete interaction:

```
>>> myfile = open('myfile.txt', 'w')       # Open for text output: create/empty
>>> myfile.write('hello text file\n')      # Write a line of text: string
16
>>> myfile.write('goodbye text file\n')
18
>>> myfile.close()                         # Flush output buffers to disk
>>> myfile = open('myfile.txt')            # Open for text input: 'r' is default
>>> myfile.readline()                      # Read the lines back
'hello text file\n'
>>> myfile.readline()
'goodbye text file\n'
>>> myfile.readline()                      # Empty string: end-of-file
''
```

Notice that file write calls return the number of characters written in Python 3.X; in 2.X they don't, so you won't see these numbers echoed interactively. This example writes each line of text, including its end-of-line terminator, \n, as a string; write methods don't add the end-of-line character for us, so we must include it to properly terminate our lines (otherwise the next write will simply extend the current line in the file).

If you want to display the file's content with end-of-line characters interpreted, read the entire file into a string all at once with the file object's read method and print it:

```
>>> open('myfile.txt').read()              # Read all at once into string
'hello text file\ngoodbye text file\n'
>>> print(open('myfile.txt').read())       # User-friendly display
hello text file
goodbye text file
```

And if you want to scan a text file line by line, file iterators are often your best option:

```
>>> for line in open('myfile.txt'):        # Use file iterators, not reads
... print(line, end='')
...
hello text file
goodbye text file
```

When coded this way, the temporary file object created by open will automatically read and return one line on each loop iteration. This form is usually easiest to code, good on memory use, and may be faster than some other options (depending on many variables, of course).

Storing Python Objects in JSON Format

The prior section's pickle module translates nearly arbitrary Python objects to a proprietary format developed specifically for Python, and honed for performance over many years. JSON is a newer and emerging data interchange format, which is both programming-language-neutral and supported by a variety of systems. MongoDB, for instance, stores data in a JSON document database (using a binary JSON format).

JSON does not support as broad a range of Python object types as pickle, but its portability is an advantage in some contexts, and it represents another way to serialize a specific category of Python objects for storage and transmission. Moreover, because JSON is so close to Python dictionaries and lists in syntax, the translation to and from Python objects is trivial, and is automated by the json standard library module.

For example, a Python dictionary with nested structures is very similar to JSON data, though Python's variables and expressions support richer structuring options (any part of the following can be an arbitrary expression in Python code):

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

The final dictionary format displayed here is a valid literal in Python code, and almost passes for JSON when printed as is, but the json module makes the translation official —here translating Python objects to and from a JSON serialized string representation in memory:

```
>>> import json
>>> json.dumps(rec)
'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'
>>> S = json.dumps(rec)
>>> S

'{"job": ["dev", "mgr"], "name": {"last": "Smith", "first": "Bob"}, "age": 40.5}'
>>> O = json.loads(S)
>>> O
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
>>> O == rec
True
```

It's similarly straightforward to translate Python objects to and from JSON data strings in files. Prior to being stored in a file, your data is simply Python objects; the JSON module recreates them from the JSON textual representation when it loads it from the file:

```
>>> json.dump(rec, fp=open('testjson.txt', 'w'), indent=4)
>>> print(open('testjson.txt').read())
{
"job": [
"dev",
"mgr"
],
"name": {
"last": "Smith",
"first": "Bob"
},
"age": 40.5
}
>>> P = json.load(open('testjson.txt'))
>>> P
{'job': ['dev', 'mgr'], 'name': {'last': 'Smith', 'first': 'Bob'}, 'age': 40.5}
```

Once you've translated from JSON text, you process the data using normal Python object operations in your script. For more details on JSON-related topics, see Python's library manuals and search the Web.

Note that strings are all Unicode in JSON to support text drawn from international character sets, so you'll see a leading u on strings after translating from JSON data in Python 2.X (but not in 3.X. Because Unicode text strings support all the usual string operations, the difference is negligible to your code while text resides in memory; the distinction matters most when transferring text to and from files, and then usually only for non-ASCII types of text where encodings come into play.

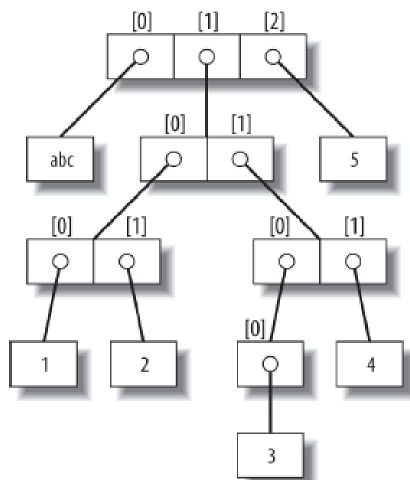
Object Flexibility

This part of the course introduced a number of compound object types—collections with components. In general:

- Lists, dictionaries, and tuples can hold any kind of object.
- Sets can contain any type of immutable object.
- Lists, dictionaries, and tuples can be arbitrarily nested.
- Lists, dictionaries, and sets can dynamically grow and shrink.

Because they support arbitrary structures, Python's compound object types are good at representing complex information in programs. For example, values in dictionaries may be lists, which may contain tuples, which may contain dictionaries, and so on. The nesting can be as deep as needed to model the data to be processed.

Let's look at an example of nesting. The following interaction defines a tree of nested compound sequence objects, shown in Figure below.



To access its components, you may include as many index operations as required. Python evaluates the indexes from left to right, and fetches a reference to a more deeply nested object at each step. Figure above may be a pathologically complicated data structure, but it illustrates the syntax used to access nested objects in general:

```
>>> L = ['abc', [(1, 2), ([3], 4)], 5]
>>> L[1]
[(1, 2), ([3], 4)]
>>> L[1][1]
([3], 4)
>>> L[1][1][0]
[3]
>>> L[1][1][0][0]
3
```

References versus copies

As mentioned that assignments always store references to objects, not copies of those objects. In practice, this is usually what you want. Because assignments can generate multiple references to the same object, though, it's important to be aware that previuos Figure. A nested object tree with the offsets of its components, created by running the literal expression [‘abc’, [(1, 2), ([3], 4)], 5]. Syntactically nested objects are internally represented as references (i.e., pointers) to separate pieces of memory.

changing a mutable object in place may affect other references to the same object elsewhere in your program. If you don't want such behavior, you'll need to tell Python to copy the object explicitly.

We studied this phenomenon before, but it can become more subtle when larger objects of the sort we've explored since then come into play. For instance, the following example creates a list assigned to X, and another list assigned to L that embeds a reference back to list X. It also creates a dictionary D that contains another reference back to list X:

```
>>> X = [1, 2, 3]
>>> L = ['a', X, 'b']        # Embed references to X's object
>>> D = {'x':X, 'y':2}
```

At this point, there are three references to the first list created: from the name X, from inside the list assigned to L, and from inside the dictionary assigned to D. The situation is illustrated in Figure below.

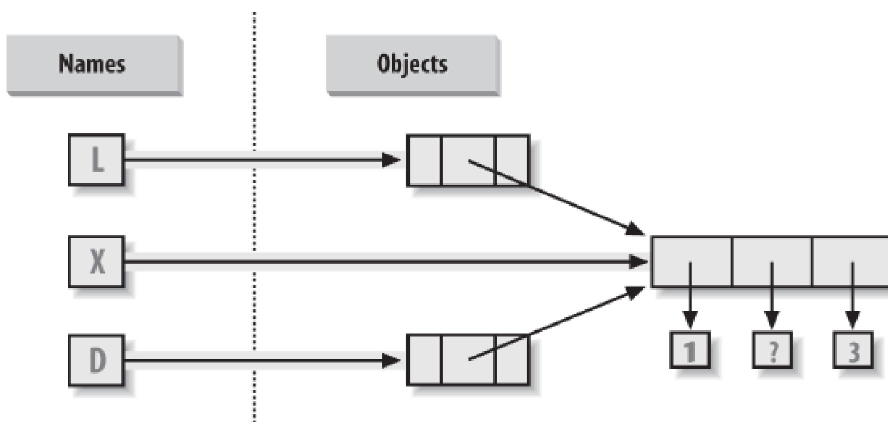


Figure: *Shared object references: because the list referenced by variable X is also referenced from within the objects referenced by L and D, changing the shared list from X makes it look different from L and D, too.*

Because lists are mutable, changing the shared list object from any of the three references also changes what the other two reference:

```
>>> X[1] = 'surprise'      # Changes all three references!
>>> L
['a', [1, 'surprise', 3], 'b']
>>> D
{'x': [1, 'surprise', 3], 'y': 2}
```

References are a higher-level analog of pointers in other languages that are always followed when used. Although you can't grab hold of the reference itself, it's possible to store the same reference in more than one place (variables, lists, and so on). This is a feature—you can pass a large object around a program without generating expensive copies of it along the way. If you really do want copies, however, you can request them:

- Slice expressions with empty limits (L[:]) copy sequences.

- The dictionary, set, and list copy method (X.copy()) copies a dictionary, set, or list (the list's copy is new as of 3.3).
- Some built-in functions, such as list and dict make copies (list(L), dict(D), set(S)).
- The copy standard library module makes full copies when needed.

For example, say you have a list and a dictionary, and you don't want their values to be changed through other variables:

```
>>> L = [1,2,3]
>>> D = {'a':1, 'b':2}
```

To prevent this, simply assign copies to the other variables, not references to the same objects:

```
>>> A = L[:]            # Instead of A = L (or list(L))
>>> B = D.copy()        # Instead of B = D (ditto for sets)
```

This way, changes made from the other variables will change the copies, not the originals:

```
>>> A[1] = 'Ni'
>>> B['c'] = 'spam'
>>>
>>> L, D
([1, 2, 3], {'a': 1, 'b': 2})
>>> A, B
([1, 'Ni', 3], {'a': 1, 'c': 'spam', 'b': 2})
```

In terms of our original example, you can avoid the reference side effects by slicing the original list instead of simply naming it:

```
>>> X = [1, 2, 3]
>>> L = ['a', X[:], 'b']    # Embed copies of X's object
>>> D = {'x':X[:], 'y':2}
```

This changes the picture in previous Figure—L and D will now point to different lists than X. The net effect is that changes made through X will impact only X, not L and D; similarly, changes to L or D will not impact X.

One final note on copies: empty-limit slices and the dictionary copy method only make top-level copies; that is, they do not copy nested data structures, if any are present. If you need a complete, fully independent copy of a deeply nested data structure , use the standard copy module:

```
import copy
X = copy.deepcopy(Y)  # Fully copy an arbitrarily nested object Y
```

This call recursively traverses objects to copy all their parts. This is a much more rare case, though, which is why you have to say more to use this scheme. References are usually what you will want; when they are not, slices and copy methods are usually as much copying as you'll need to do.

Comparisons, equality, and truth

All Python objects also respond to comparisons: tests for equality, relative magnitude, and so on. Python comparisons always inspect all parts of compound objects until a result can be determined. In fact, when nested objects are present, Python automatically traverses data structures to apply comparisons from left to right, and as deeply as needed. The first difference found along the way determines the comparison result.

This is sometimes called a recursive comparison—the same comparison requested on the top-level objects is applied to each of the nested objects, and to each of their nested objects, and so on, until a result is found. We'll see later how to write recursive functions of our own that work similarly on nested structures.

For now, think about comparing all the linked pages at two websites if you want a metaphor for such structures, and a reason for writing recursive functions to process them.

In terms of core types, the recursion is automatic. For instance, a comparison of list objects compares all their components automatically until a mismatch is found or the end is reached:

```
>>> L1 = [1, ('a', 3)]            # Same value, unique objects
>>> L2 = [1, ('a', 3)]
>>> L1 == L2, L1 is L2            # Equivalent? Same object?
(True, False)
```

Here, L1 and L2 are assigned lists that are equivalent but distinct objects. As a review of what we saw, because of the nature of Python references, there are two ways to test for equality:

- The == operator tests value equivalence. Python performs an equivalence test, comparing all nested objects recursively.
- The is operator tests object identity. Python tests whether the two are really the same object (i.e., live at the same address in memory).

In the preceding example, L1 and L2 pass the == test (they have equivalent values because all their components are equivalent) but fail the is check (they reference two different objects, and hence two different pieces of memory). Notice what happens for short strings, though:

```
>>> S1 = 'spam'
>>> S2 = 'spam'
>>> S1 == S2, S1 is S2
(True, True)
```

Here, we should again have two distinct objects that happen to have the same value: == should be true, and is should be false. But because Python internally caches and reuses some strings as an optimization, there really is just a single string 'spam' in memory, shared by S1 and S2; hence, the is identity test reports a true result. To trigger the normal behavior, we need to use longer strings:

```
>>> S1 = 'a longer string'
>>> S2 = 'a longer string'
>>> S1 == S2, S1 is S2
(True, False)
```

Of course, because strings are immutable, the object caching mechanism is irrelevant to your code—strings can't be changed in place, regardless of how many variables refer to them. If identity tests seem confusing.

As a rule of thumb, the == operator is what you will want to use for almost all equality checks; is is reserved for highly specialized roles. We'll see cases later in the course where both operators are put to use.

Relative magnitude comparisons are also applied recursively to nested data structures:

```
>>> L1 = [1, ('a', 3)]
>>> L2 = [1, ('a', 2)]
>>> L1 < L2, L1 == L2, L1 > L2          # Less, equal, greater: tuple of results
(False, False, True)
```

Here, L1 is greater than L2 because the nested 3 is greater than 2. By now you should know that the result of the last line is really a tuple of three objects—the results of the three expressions typed (an example of a tuple without its enclosing parentheses).

More specifically, Python compares types as follows:

- Numbers are compared by relative magnitude, after conversion to the common highest type if needed.
- Strings are compared lexicographically (by the character set code point values returned by ord), and character by character until the end or first mismatch ("abc" < "ac").
- Lists and tuples are compared by comparing each component from left to right, and recursively for nested structures, until the end or first mismatch ([2] > [1, 2]).
- Sets are equal if both contain the same items (formally, if each is a subset of the other), and set relative magnitude comparisons apply subset and superset tests.
- Dictionaries compare as equal if their sorted (key, value) lists are equal. Relative magnitude comparisons are not supported for dictionaries in Python 3.X, but they work in 2.X as though comparing sorted (key, value) lists.
- Nonnumeric mixed-type magnitude comparisons (e.g., 1 < 'spam') are errors in Python 3.X. They are allowed in Python 2.X, but use a fixed but arbitrary ordering rule based on type name string. By proxy, this also applies to sorts, which use comparisons internally: nonnumeric mixed-type collections cannot be sorted in 3.X.

In general, comparisons of structured objects proceed as though you had written the objects as literals and compared all their parts one at a time from left to right..

Python's type hierarchies

As a summary and reference, Figure 9-3 sketches all the built-in object types available in Python and their relationships. We've looked at the most prominent of these; most of the other kinds of objects in Figure 9-3 correspond to program units (e.g., functions and modules) or exposed interpreter internals (e.g., stack frames and compiled code).

The largest point to notice here is that everything in a Python system is an object type and may be processed by your Python programs. For instance, you can pass a class to a function, assign it to a variable, stuff it in a list or dictionary, and so on.

In fact, even types themselves are an object type in Python: the type of an object is an object of type type (say that three times fast!). Seriously, a call to the built-in function type(X) returns the type object of object X. The practical application of this is that type objects can be used for manual type comparisons in Python if statements. However, manual type testing is usually not the right thing to do in Python, since it limits your code's flexibility.

One note on type names: as of Python 2.2, each core type has a new built-in name added to support type customization through object-oriented subclassing: dict, list, str, tuple, int, float, complex, bytes, type, set, and more. In Python 3.X names all references classes, and in Python 2.X but not 3.X, file is also a type name and a synonym for open. Calls to these names are really object constructor calls, not simply conversion functions, though you can treat them as simple functions for basic usage.

In addition, the types standard library module in Python 3.X provides additional type names for types that are not available as built-ins (e.g., the type of a function; in Python 2.X but not 3.X, this module also includes synonyms for built-in type names), and it is possible to do type tests with the isinstance function. For example, all of the following type tests are true:

```
type([1]) == type([])       # Compare to type of another list
type([1]) == list           # Compare to list type name
isinstance([1], list)       # Test if list or customization thereof
import types                # types has names for other types
def f(): pass
type(f) == types.FunctionType
```

Because types can be subclassed in Python today, the isinstance technique is generally recommended.

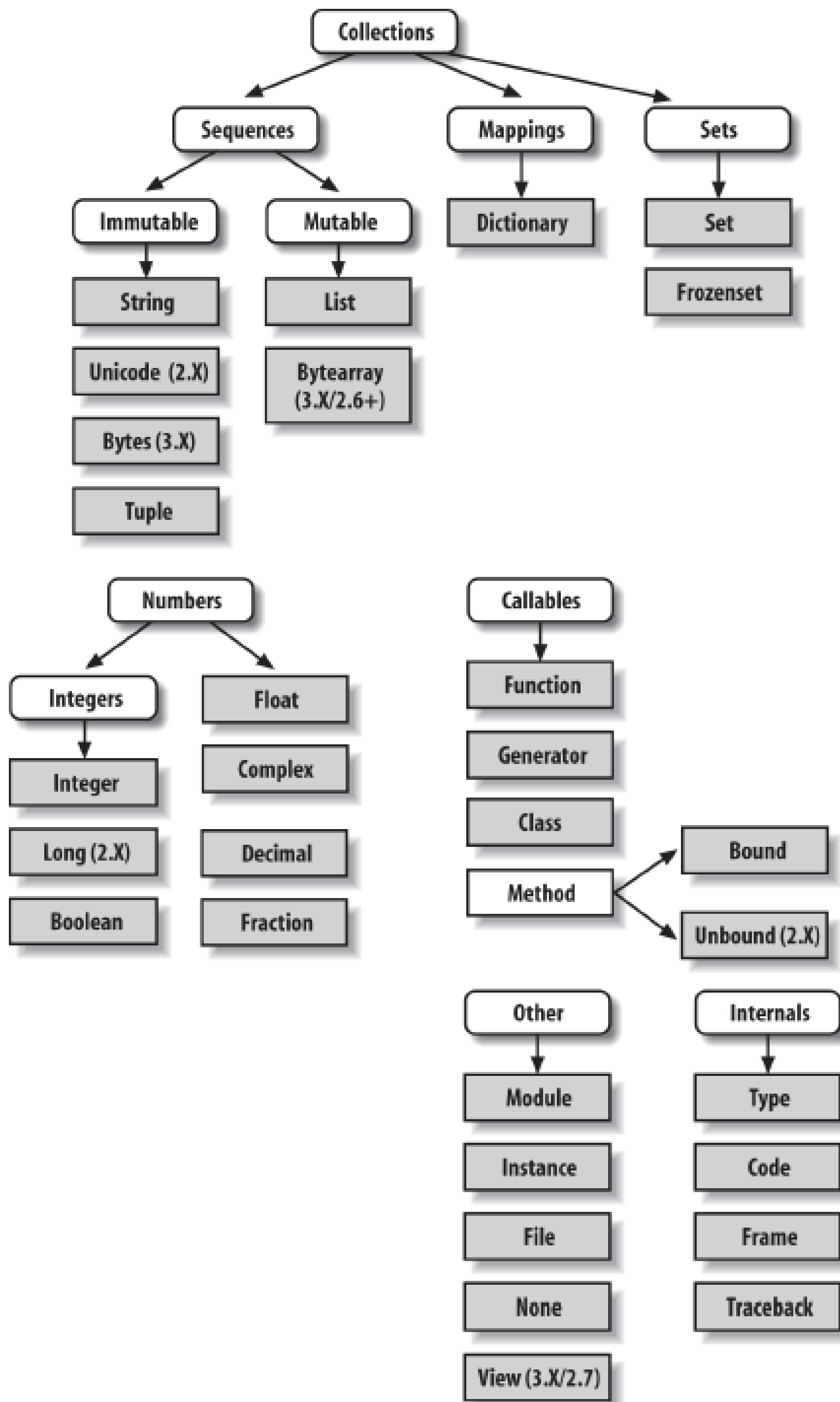


Figure: *Python's major built-in object types, organized by categories. Everything is a type of object in Python, even the type of an object! Some extension types, such as named tuples, might belong in this figure too, but the criteria for inclusion in the core types set are not formal.*

Other types in Python
Besides the core objects studied in this part of the course, and the program-unit objects such as functions, modules, and classes that we'll meet later, a typical Python installation has dozens of additional object types available as linked-in C extensions or Python classes—regular expression objects, DBM files, GUI widgets, network sockets, and so on. Depending on whom you ask, the named tuple we met earlier in this module may fall in this category too.

The main difference between these extra tools and the built-in types we've seen so far is that the built-ins provide special language creation syntax for their objects (e.g., 4 for an integer, [1,2] for a list, the open function for files, and def and lambda for functions).

Other tools are generally made available in standard library modules that you must first import to use, and aren't usually considered core types. For instance, to make a regular expression object, you import re and call re.compile(). See Python's library reference for a comprehensive guide to all the tools available to Python programs.

Built-in type gotchas
That's the end of our look at core data types. We'll wrap up this part of the course with a discussion of common problems that seem to trap new users (and the occasional expert), along with their solutions. Some of this is a review of ideas we've already covered, but these issues are important enough to warn about again here.

Assignment Creates References, Not Copies
Because this is such a central concept, I'll mention it again: shared references to mutable objects in your program can matter. For instance, in the following example, the list object assigned to the name L is referenced both from L and from inside the list assigned to the name M. Changing L in place changes what M references, too:

```
>>> L = [1, 2, 3]
>>> M = ['X', L, 'Y']      # Embed a reference to L
>>> M
['X', [1, 2, 3], 'Y']
>>> L[1] = 0               # Changes M too
>>> M
['X', [1, 0, 3], 'Y']
```

This effect usually becomes important only in larger programs, and shared references are often exactly what you want. If objects change out from under you in unwanted ways, you can avoid sharing objects by copying them explicitly. For lists, you can always make a top-level copy by using an empty-limits slice, among other techniques described earlier:

```
>>> L = [1, 2, 3]
>>> M = ['X', L[:], 'Y']          # Embed a copy of L (or list(L), or L.copy())
>>> L[1] = 0                      # Changes only L, not M
>>> L
[1, 0, 3]
>>> M
['X', [1, 2, 3], 'Y']
```

Remember, slice limits default to 0 and the length of the sequence being sliced; if both are omitted, the slice extracts every item in the sequence and so makes a top-level copy (a new, unshared object).

Repetition Adds One Level Deep

Repeating a sequence is like adding it to itself a number of times. However, when mutable sequences are nested, the effect might not always be what you expect. For instance, in the following example X is assigned to L repeated four times, whereas Y is assigned to a list containing L repeated four times:

```
>>> L = [4, 5, 6]
>>> X = L * 4 # Like [4, 5, 6] + [4, 5, 6] + ...
>>> Y = [L] * 4 # [L] + [L] + ... = [L, L,...]
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Because L was nested in the second repetition, Y winds up embedding references back to the original list assigned to L, and so is open to the same sorts of side effects noted in the preceding section:

```
>>> L[1] = 0     # Impacts Y but not X
>>> X
[4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 5, 6]
>>> Y
[[4, 0, 6], [4, 0, 6], [4, 0, 6], [4, 0, 6]]
```

This may seem artificial and academic—until it happens unexpectedly in your code!

The same solutions to this problem apply here as in the previous section, as this is really just another way to create the shared mutable object reference case—make copies when you don't want shared references:

```
>>> L = [4, 5, 6]
>>> Y = [list(L)] * 4      # Embed a (shared) copy of L
>>> L[1] = 0
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

Even more subtly, although Y doesn't share an object with L anymore, it still embeds four references to the same copy of it. If you must avoid that sharing too, you'll want to make sure each embedded copy is unique:

```
>>> Y[0][1] = 99           # All four copies are still the same
>>> Y
[[4, 99, 6], [4, 99, 6], [4, 99, 6], [4, 99, 6]]
>>> L = [4, 5, 6]
>>> Y = [list(L) for i in range(4)]
>>> Y
[[4, 5, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
>>> Y[0][1] = 99
>>> Y
[[4, 99, 6], [4, 5, 6], [4, 5, 6], [4, 5, 6]]
```

If you remember that repetition, concatenation, and slicing copy only the top level of their operand objects, these sorts of cases make much more sense.

Beware of Cyclic Data Structures

We actually encountered this concept in a prior exercise: if a collection object contains a reference to itself, it's called a cyclic object. Python prints a [...] whenever it detects a cycle in the object, rather than getting stuck in an infinite loop (as it once did long ago):

```
>>> L = ['grail']           # Append reference to same object
>>> L.append(L)             # Generates cycle in object: [...]
>>> L
['grail', [...]]
```

Besides understanding that the three dots in square brackets represent a cycle in the object, this case is worth knowing about because it can lead to gotchas—cyclic structures may cause code of your own to fall into unexpected loops if you don't anticipate them.

For instance, some programs that walk through structured data must keep a list, dictionary, or set of already visited items, and check it when they're about to step into a cycle that could cause an unwanted loop.

The solution is knowledge: don't use cyclic references unless you really need to, and make sure you anticipate them in programs that must care. There are good reasons to create cycles, but unless you have code that knows how to handle them, objects that reference themselves may be more surprise than asset.

Immutable Types Can't Be Changed in Place

Immutable Types Can't Be Changed in Place And once more for completeness: you can't change an immutable object in place. Instead, you construct a new object with slicing, concatenation, and so on, and assign it back to the original reference, if needed:

```
T = (1, 2, 3)
T[2] = 4                    # Error!
T = T[:2] + (4,)            # OK: (1, 2, 4)
```

That might seem like extra coding work, but the upside is that the previous gotchas in this section can't happen when you're using immutable objects such as tuples and strings; because they can't be changed in place, they are not open to the sorts of side effects that lists are.

Module 9 - Assignment, Expressions, & Print

Now that we've had a quick introduction to Python statement syntax, this module begins our in-depth tour of specific Python statements. We'll begin with the basics: assignment statements, expression statements, and print operations. We've already seen all of these in action, but here we'll fill in important details we've skipped so far.

Although they're relatively simple, as you'll see, there are optional variations for each of these statement types that will come in handy once you begin writing realistic Python programs.

Assignment statements

We've been using the Python assignment statement for a while to assign objects to names. In its basic form, you write the target of an assignment on the left of an equals sign, and the object to be assigned on the right. The target on the left may be a name or object component, and the object on the right can be an arbitrary expression that computes an object. For the most part, assignments are straightforward, but here are a few properties to keep in mind:

- **Assignments create object references.** As discussed, Python assignments store references to objects in names or data structure components. They always create references to objects instead of copying the objects. Because of that,
  Python variables are more like pointers than data storage areas.
- **Names are created when first assigned.** Python creates a variable name the first time you assign it a value (i.e., an object reference), so there's no need to predeclare names ahead of time. Some (but not all) data structure slots are created when assigned, too (e.g., dictionary entries, some object attributes). Once assigned, a name is replaced with the value it references whenever it appears in an expression.
- **Names must be assigned before being referenced.** It's an error to use a name to which you haven't yet assigned a value. Python raises an exception if you try, rather than returning some sort of ambiguous default value. This turns out to be crucial in Python because names are not predeclared—if Python provided default values for unassigned names used in your program instead of treating them as errors, it would be much more difficult for you to spot name typos in your code.
- **Some operations perform assignments implicitly.** In this section we're concerned with the = statement, but assignment occurs in many contexts in Python.
  For instance, we'll see later that module imports, function and class definitions, for loop variables, and function arguments are all implicit assignments. Because assignment works the same everywhere it pops up, all these contexts simply bind names to object references at runtime.

Assignment Statement Forms

Although assignment is a general and pervasive concept in Python, we are primarily interested in assignment statements in this module. Table below illustrates the different assignment statement forms in Python, and their syntax patterns.

Table: *Assignment statement forms*

Operation	Interpretation
spam = 'Spam'	Basic form
spam, ham = 'yum', 'YUM'	Tuple assignment (positional)
[spam, ham] = ['yum', 'YUM']	List assignment (positional)
a, b, c, d = 'spam'	Sequence assignment, generalized
a, *b = 'spam'	Extended sequence unpacking (Python 3.X)
spam = ham = 'lunch'	Multiple-target assignment
spams += 42	Augmented assignment (equivalent to spams = spams + 42)

The first form in Table above is by far the most common: binding a name (or data structure component) to a single object. In fact, you could get all your work done with this basic form alone. The other table entries represent special forms that are all optional, but that programmers often find convenient in practice:

Tuple- and list-unpacking assignments

The second and third forms in the table are related. When you code a tuple or list on the left side of the =, Python pairs objects on the right side with targets on the left by position and assigns them from left to right. For example, in the second line of previuos Table, the name spam is assigned the string 'yum', and the name ham is bound o the string 'YUM'. In this case Python internally may make a tuple of the items on the right, which is why this is called tuple-unpacking assignment.

Sequence assignments

In later versions of Python, tuple and list assignments were generalized into instances of what we now call sequence assignment—any sequence of names can be assigned to any sequence of values, and Python assigns the items one at a time by position. We can even mix and match the types of the sequences involved. The fourth line in Table 11-1, for example, pairs a tuple of names with a string of characters: a is assigned 's', b is assigned 'p', and so on.

Extended sequence unpacking

In Python 3.X (only), a new form of sequence assignment allows us to be more flexible in how we select portions of a sequence to assign. The fifth line in previous Table, for example, matches a with the first character in the string on the right and b with the rest: a is assigned 's', and b is assigned 'pam'. This provides a simpler alternative to assigning the results of manual slicing operations.

Multiple-target assignments

The sixth line in previous Table shows the multiple-target form of assignment. In this form, Python assigns a reference to the same object (the object farthest to the right) to all the targets on the left. In the table, the names spam and ham are both assigned references to the same string object, 'lunch'. The effect is the same as if we had coded ham = 'lunch' followed by spam = ham, as ham evaluates to the original string object (i.e., not a separate copy of that object).

Augmented assignments

The last line in previous Table is an example of augmented assignment—a shorthand that combines an expression and an assignment in a concise way. Saying spam += 42, for example, has the same effect as spam = spam + 42, but the augmented form requires less typing and is generally quicker to run. In addition, if the subject is mutable and supports the operation, an augmented assignment may run even quicker by choosing an in-place update operation instead of an object copy. There is one augmented assignment statement for every binary expression operator in
Python.

Sequence Assignments

We've already used and explored basic assignments in this course, so we'll take them as a given. Here are a few simple examples of sequence-unpacking assignments in action:

```
% python
>>> nudge = 1                     # Basic assignment
>>> wink = 2
>>> A, B = nudge, wink            # Tuple assignment
>>> A, B # Like A = nudge; B = wink
(1, 2)
>>> [C, D] = [nudge, wink]        # List assignment
>>> C, D
(1, 2)
```

Notice that we really are coding two tuples in the third line in this interaction—we've just omitted their enclosing parentheses. Python pairs the values in the tuple on the right side of the assignment operator with the variables in the tuple on the left side and assigns the values one at a time.

Tuple assignment leads to a common coding trick in Python that was introduced in a solution to the exercises at the end of Part II. Because Python creates a temporary tuple that saves the original values of the variables on the right while the statement runs, unpacking assignments are also a way to swap two variables' values without creating a temporary variable of your own—the tuple on the right remembers the prior values of the variables automatically:

```
>>> nudge = 1
```

```
>>> wink = 2
>>> nudge, wink = wink, nudge            # Tuples: swaps values
>>> nudge, wink                          # Like T = nudge; nudge = wink; wink = T
(2, 1)
```

In fact, the original tuple and list assignment forms in Python have been generalized to accept any type of sequence (really, iterable) on the right as long as it is of the same length as the sequence on the left. You can assign a tuple of values to a list of variables, a string of characters to a tuple of variables, and so on. In all cases, Python assigns items in the sequence on the right to variables in the sequence on the left by position, from left to right:

```
>>> [a, b, c] = (1, 2, 3)         # Assign tuple of values to list of names
>>> a, c
(1, 3)
>>> (a, b, c) = "ABC"             # Assign string of characters to tuple
>>> a, c
('A', 'C')
```

Technically speaking, sequence assignment actually supports any iterable object on the right, not just any sequence. This is a more general category that includes collections both physical (e.g., lists) and virtual (e.g., a file's lines).

Multiple-Target Assignments

A multiple-target assignment simply assigns all the given names to the object all the way to the right. The following, for example, assigns the three variables a, b, and c to the string 'spam':

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

This form is equivalent to (but easier to code than) these three assignments:

```
>>> c = 'spam'
>>> b = c
>>> a = b
```

Augmented Assignments

Beginning with Python 2.0, the set of additional assignment statement formats listed in Table below became available. Known as augmented assignments, and borrowed from the C language, these formats are mostly just shorthand. They imply the combination of a binary expression and an assignment. For instance, the following two formats are roughly equivalent:

```
X = X + Y        # Traditional form
X += Y           # Newer augmented form
```

Table: *Augmented assignment statements*

X += Y	X &= Y	X −= Y	X |= Y
X *= Y	X ^= Y	X /= Y	X >>= Y
X %= Y	X <<= Y	X **= Y	X //= Y

Augmented assignment works on any type that supports the implied binary expression. For example, here are two ways to add 1 to a name:

```
>>> x = 1
>>> x = x + 1    # Traditional
>>> x
2
>>> x += 1       # Augmented
>>> x
3
```

When applied to a sequence such as a string, the augmented form performs concatenation instead. Thus, the second line here is equivalent to typing the longer S = S + "SPAM":

```
>>> S = "spam"
>>> S += "SPAM" # Implied concatenation
>>> S
'spamSPAM'
```

As shown in previous Table, there are analogous augmented assignment forms for every Python binary expression operator (i.e., each operator with values on the left and right side). For instance, X *= Y multiplies and assigns, X >>= Y shifts right and assigns, and so on. X //= Y (for floor division) was added in version 2.2.

Augmented assignments have three advantages:

- There's less for you to type. Need I say more?
- The left side has to be evaluated only once. In X += Y, X may be a complicated object expression. In the augmented form, its code must be run only once. However, in the long form, X = X + Y, X appears twice and must be run twice. Because of this, augmented assignments usually run faster.
- The optimal technique is automatically chosen. That is, for objects that support in-place changes, the augmented forms automatically perform in-place change operations instead of slower copies.

The last point here requires a bit more explanation. For augmented assignments, inplace operations may be applied for mutable objects as an optimization. Recall that lists can be extended in a variety of ways. To add a single item to the end of a list, we can concatenate or call append:

```
>>> L = [1, 2]
>>> L = L + [3]         # Concatenate: slower
>>> L
[1, 2, 3]
>>> L.append(4)         # Faster, but in place
>>> L
[1, 2, 3, 4]
```

And to add a set of items to the end, we can either concatenate again or call the list extend method:2

```
>>> L = L + [5, 6]      # Concatenate: slower
>>> L
[1, 2, 3, 4, 5, 6]
>>> L.extend([7, 8])    # Faster, but in place
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

In both cases, concatenation is less prone to the side effects of shared object references but will generally run slower than the in-place equivalent. Concatenation operations must create a new object, copy in the list on the left, and then copy in the list on the right. By contrast, in-place method calls simply add items at the end of a memory block (it can be a bit more complicated than that internally, but this description suffices).

When we use augmented assignment to extend a list, we can largely forget these details —Python automatically calls the quicker extend method instead of using the slower concatenation operation implied by +:

```
>>> L += [9, 10] # Mapped to L.extend([9, 10])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Note however, that because of this equivalence += for a list is not exactly the same as a + and = in all cases—for lists += allows arbitrary sequences (just like extend), but concatenation normally does not:

```
>>> L = []
>>> L += 'spam'         # += and extend allow any sequence, but + does not!
>>> L
```

```
['s', 'p', 'a', 'm']
>>> L = L + 'spam'
TypeError: can only concatenate list (not "str") to list
```

Augmented assignment and shared references

This behavior is usually what we want, but notice that it implies that the += is an inplace hange for lists; thus, it is not exactly like + concatenation, which always makes a new object. As for all shared reference cases, this difference might matter if other names reference the object being changed:

```
>>> L = [1, 2]
>>> M = L                  # L and M reference the same object
>>> L = L + [3, 4]         # Concatenation makes a new object
>>> L, M # Changes L but not M
([1, 2, 3, 4], [1, 2])
>>> L = [1, 2]
>>> M = L
>>> L += [3, 4]            # But += really means extend
>>> L, M                   # M sees the in-place change too!
([1, 2, 3, 4], [1, 2, 3, 4])
```

This only matters for mutables like lists and dictionaries, and it is a fairly obscure case (at least, until it impacts your code!). As always, make copies of your mutable objects if you need to break the shared reference structure.

Variable Name Rules

Now that we've explored assignment statements, it's time to get more formal about the use of variable names. In Python, names come into existence when you assign values to them, but there are a few rules to follow when choosing names for the subjects of your programs:

Syntax: (underscore or letter) + (any number of letters, digits, or underscores)

Variable names must start with an underscore or letter, which can be followed by any number of letters, digits, or underscores. _spam, spam, and Spam_1 are legal names, but 1_Spam, spam$, and @#! are not.

Case matters: SPAM is not the same as spam

Python always pays attention to case in programs, both in names you create and in reserved words. For instance, the names X and x refer to two different variables.

For portability, case also matters in the names of imported module files, even on platforms where the filesystems are case-insensitive. That way, your imports still work after programs are copied to differing platforms.

Reserved words are off-limits

Names you define cannot be the same as words that mean special things in the Python language. For instance, if you try to use a variable name like class, Python will raise a syntax error, but klass and Class work fine. Table below lists the words that are currently reserved (and hence off-limits for names of your own) in Python.

Table: *Python 3.X reserved words*

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	While
and	del	global	not	with
as	elif	if	or	Yield
assert	else	import	pass	
break	except	in	raise	

Table above is specific to Python 3.X. In Python 2.X, the set of reserved words differs slightly:

- print is a reserved word, because printing is a statement, not a built-in function (more on this later in this module).
- exec is a reserved word, because it is a statement, not a built-in function.
- nonlocal is not a reserved word because this statement is not available.

In older Pythons the story is also more or less the same, with a few variations:

- with and as were not reserved until 2.6, when context managers were officially enabled.
- yield was not reserved until Python 2.3, when generator functions came online.
- yield morphed from statement to expression in 2.5, but it's still a reserved word, not a built-in function.

As you can see, most of Python's reserved words are all lowercase. They are also all truly reserved—unlike names in the built-in scope that you will meet in the next part of this course, you cannot redefine reserved words by assignment (e.g., and = 1 results in a syntax error).

Besides being of mixed case, the first three entries in previous Table, True, False, and None, are somewhat unusual in meaning—they also appear in the built-in scope of Python, and they are technically names assigned to objects. In 3.X they are truly reserved in all other senses, though, and cannot be used for any other purpose in your script other than that of the objects they represent. All the other reserved words are hardwired into Python's syntax and can appear only in the specific contexts for which they are intended.

Furthermore, because module names in import statements become variables in your scripts, variable name constraints extend to your module filenames too. For instance, you can code files called and.py and my-code.py and run them as top-level scripts, but you cannot import them: their names without the ".py" extension become variables in your code and so must follow all the variable rules just outlined. Reserved words are off-limits, and dashes won't work, though underscores will.

Naming conventions

Besides these rules, there is also a set of naming conventions—rules that are not required but are followed in normal practice. For instance, because names with two leading and trailing underscores (e.g., __name__) generally have special meaning to the Python interpreter, you should avoid this pattern for your own names. Here is a list of the conventions Python follows:

- Names that begin with a single underscore (_X) are not imported by a from module import * statement.
- Names that have two leading and trailing underscores (__X__) are system-defined names that have special meaning to the interpreter.
- Names that begin with two underscores and do not end with two more (__X) are localized ("mangled") to enclosing classes.
- The name that is just a single underscore (_) retains the result of the last expression when you are working interactively.

In addition to these Python interpreter conventions, there are various other conventions that Python programmers usually follow. For instance, later in the course we'll see that class names commonly start with an uppercase letter and module names with a lowercase letter, and that the name self, though not reserved, usually has a special role in classes. We'll also study another, larger category of names known as the built-ins, which are predefined but not reserved (and so can be reassigned: open = 42 works, though sometimes you might wish it didn't!).

Names have no type, but objects do

This is mostly review, but remember that it's crucial to keep Python's distinction between names and objects clear. As describe, objects have a type (e.g., integer, list) and may be mutable or not. Names (a.k.a. variables), on the other hand, are always just references to objects; they have no notion of mutability and have no associated type information, apart from the type of the object they happen to reference at a given point in time.

Thus, it's OK to assign the same name to different kinds of objects at different times:

```
>>> x = 0              # x bound to an integer object
>>> x = "Hello"        # Now it's a string
>>> x = [1, 2, 3]      # And now it's a list
```

In later examples, you'll see that this generic nature of names can be a decided advantage in Python programming. You'll also learn that names also live in something called a scope, which defines where they can be used; the place where you assign a name determines where it is visible.

Expression statements

In Python, you can use an expression as a statement, too—that is, on a line by itself. But because the result of the expression won't be saved, it usually makes sense to do so only if the expression does something useful as a side effect. Expressions are commonly used as statements in two situations:

For calls to functions and methods

Some functions and methods do their work without returning a value. Such functions are sometimes called procedures in other languages. Because they don't return values that you might be interested in retaining, you can call these functions with expression statements.

For printing values at the interactive prompt

Python echoes back the results of expressions typed at the interactive command line. Technically, these are expression statements, too; they serve as a shorthand for typing print statements.

Table below lists some common expression statement forms in Python. Calls to functions and methods are coded with zero or more argument objects (really, expressions that evaluate to objects) in parentheses, after the function/method name.

Table: *Common Python expression statements*

Operation	Interpretation
spam(eggs, ham)	Function calls
spam.ham(eggs)	Method calls
spam	Printing variables in the interactive interpreter
print(a, b, c, sep='')	Printing operations in Python 3.X
yield x ** 2	Yielding expression statements

The last two entries in Table above are somewhat special cases—as we'll see later in this module, printing in Python 3.X is a function call usually coded on a line by itself, and the yield operation in generator functions is often coded as a statement as well. Both are really just instances of expression statements.

For instance, though you normally run a 3.X print call on a line by itself as an expression statement, it returns a value like any other function call (its return value is None, the default return value for functions that don't return anything meaningful):

```
>>> x = print('spam')    # print is a function call expression in 3.X
spam
>>> print(x)             # But it is coded as an expression statement
None
```

Also keep in mind that although expressions can appear as statements in Python, statements cannot be used as expressions. A statement that is not an expression must generally appear on a line all by itself, not nested in a larger syntactic structure. For example, Python doesn't allow you to embed assignment statements (=) in other expressions. The rationale for this is that it avoids common coding mistakes; you can't accidentally change a variable by typing = when you really mean to use the == equality test.

Expression Statements and In-Place Changes

This brings up another mistake that is common in Python work. Expression statements are often used to run list methods that change a list in place:

```
>>> L = [1, 2]
>>> L.append(3)          # Append is an in-place change
>>> L
[1, 2, 3]
```

However, it's not unusual for Python newcomers to code such an operation as an assignment statement instead, intending to assign L to the larger list:

```
>>> L = L.append(4)     # But append returns None, not L
>>> print(L)             # So we lose our list!
None
```

This doesn't quite work, though. Calling an in-place change operation such as append, sort, or reverse on a list always changes the list in place, but these methods do not return the list they have changed; instead, they return the None object. Thus, if you assign such an operation's result back to the variable name, you effectively lose the list
(and it is probably garbage-collected in the process!).

The moral of the story is, don't do this—call in-place change operations without assigning their results. We'll revisit this phenomenon in the section "Common Coding Gotchas" because it can also appear in the context of some looping statements.

Print statements

In Python, print prints things—it's simply a programmer-friendly interface to the standard output stream.
Technically, printing converts one or more objects to their textual representations, adds some minor formatting, and sends the resulting text to either standard output or another file-like stream. In a bit more detail, print is strongly bound up with the notions of files and streams in Python:

File object methods

We learned about file object methods that write text (e.g.,file.write(str)). Printing operations are similar, but more focused—whereas file write methods write strings to arbitrary files, print writes objects to the stdout stream by default, with some automatic formatting added. Unlike with file methods, there is no need to convert objects to strings when using print operations.

Standard output stream

The standard output stream (often known as stdout) is simply a default place to send a program's text output. Along with the standard input and error streams, it's one of three data connections created when your script starts. The standard output stream is usually mapped to the window where you started your Python program, unless it's been redirected to a file or pipe in your operating system's shell. Because the standard output stream is available in Python as the stdout file object in the built-in sys module (i.e., sys.stdout), it's possible to emulate print with file
write method calls. However, print is noticeably easier to use and makes it easy to print text to other files and streams.

Printing is also one of the most visible places where Python 3.X and 2.X have diverged. In fact, this divergence is usually the first reason that most 2.X code won't run unchanged under 3.X. Specifically, the way you code print operations depends on which version of Python you use:

- In Python 3.X, printing is a built-in function, with keyword arguments for special modes.
- In Python 2.X, printing is a statement with specific syntax all its own.

Because this course covers both 3.X and 2.X, we will look at each form in turn here. If you are fortunate enough to be able to work with code written for just one version of Python, feel free to pick the section that is relevant to you. Because your needs may change, however, it probably won't hurt to be familiar with both cases. Moreover, users of recent Python 2.X releases can also import and use 3.X's flavor of printing in their Pythons if desired—both for its extra functionality and to ease future migration to 3.X.

Module 10 - If Tests

This module presents the Python if statement, which is the main statement used for selecting from alternative actions based on test results. Because this is our first in-depth look at compound statements—statements that embed other statements—we will also explore the general concepts behind the Python statement syntax model here in more detail than we did in the introduction before. Because the if statement introduces the notion of tests, this module will also deal with Boolean expressions, cover the "ternary" if expression, and fill in some details on truth tests in general.

If statements

In simple terms, the Python if statement selects actions to perform. Along with its expression counterpart, it's the primary selection tool in Python and represents much of the logic a Python program possesses. It's also our first compound statement. Like all compound Python statements, the if statement may contain other statements, including other ifs. In fact, Python lets you combine statements in a program sequentially (so that they execute one after another), and in an arbitrarily nested fashion (so that they execute only under certain conditions such as selections and loops).

General Format

The Python if statement is typical of if statements in most procedural languages. It takes the form of an if test, followed by one or more optional elif ("else if") tests and a final optional else block. The tests and the else part each have an associated block of nested statements, indented under a header line. When the if statement runs, Python executes the block of code associated with the first test that evaluates to true, or the else block if all tests prove false. The general form of an if statement looks like this:

```
if test1: # if test
statements1 # Associated block
elif test2: # Optional elifs
statements2
else: # Optional else
statements3
```

Basic Examples

To demonstrate, let's look at a few simple examples of the if statement at work. All parts are optional, except the initial if test and its associated statements. Thus, in the simplest case, the other parts are omitted:

```
>>> if 1:
...     print('true')
...
true
```

Notice how the prompt changes to ... for continuation lines when you're typing interactively in the basic interface used here; in IDLE, you'll simply drop down to an indented line instead (hit Backspace to back up). A blank line (which you can get by pressing Enter twice) terminates and runs the entire statement. Remember that 1 is Boolean true (as we'll see later, the word True is its equivalent), so this statement's test always succeeds. To handle a false result, code the else:

```
>>> if not 1:
...     print('true')
... else:
...     print('false')
...
false
```

Multiway Branching

Now here's an example of a more complex if statement, with all its optional parts present:

```
>>> x = 'killer rabbit'
>>> if x == 'roger':
...     print("shave and a haircut")
... elif x == 'bugs':
...     print("what's up doc?")
... else:
...     print('Run away! Run away!')
...
Run away! Run away!
```

This multiline statement extends from the if line through the block nested under the else. When it's run, Python executes the statements nested under the first test that is true, or the else part if all tests are false (in this example, they are). In practice, both the elif and else parts may be omitted, and there may be more than one statement nested in each section. Note that the words if, elif, and else are associated by the fact that they line up vertically, with the same indentation.

If you've used languages like C or Pascal, you might be interested to know that there is no switch or case statement in Python that selects an action based on a variable's value. Instead, you usually code multiway branching as a series of if/elif tests, as in the prior example, and occasionally by indexing dictionaries or searching lists. Because dictionaries and lists can be built at runtime dynamically, they are sometimes more flexible than hardcoded if logic in your script:

```
>>> choice = 'ham'
>>> print({'spam': 1.25,          # A dictionary-based 'switch'
...        'ham': 1.99,           # Use has_key or get for default
...        'eggs': 0.99,
...        'bacon': 1.10}[choice])
1.99
```

Although it may take a few moments for this to sink in the first time you see it, this dictionary is a multiway branch—indexing on the key choice branches to one of a set of values, much like a switch in C. An almost equivalent but more verbose Python if statement might look like the following:

```
>>> if choice == 'spam': # The equivalent if statement
...     print(1.25)
... elif choice == 'ham':
...     print(1.99)
... elif choice == 'eggs':
...     print(0.99)
... elif choice == 'bacon':
...     print(1.10)
... else:
...     print('Bad choice')
...
1.99
```

Though it's perhaps more readable, the potential downside of an if like this is that, short of constructing it as a string and running it with tools like the prior module's eval or exec, you cannot construct it at runtime as easily as a dictionary. In more dynamic programs, data structures offer added flexibility.

Handling switch defaults
Notice the else clause on the if here to handle the default case when no key matches.
Dictionary defaults can be coded with in expressions, get method calls, or exception catching with the try statement. All of the same techniques can be used here to code a default action in a dictionary-based multiway branch. As a review in the context of this use case, here's the get scheme at work with defaults:

```
>>> branch = {'spam': 1.25,
...           'ham': 1.99,
...           'eggs': 0.99}
>>> print(branch.get('spam', 'Bad choice'))
1.25
>>> print(branch.get('bacon', 'Bad choice'))
Bad choice
```

An in membership test in an if statement can have the same default effect:

```
>>> choice = 'bacon'
>>> if choice in branch:
...     print(branch[choice])
... else:
...     print('Bad choice')
...
Bad choice
```

And the try statement is a general way to handle defaults by catching and handling the exceptions they'd otherwise trigger:

```
>>> try:
...     print(branch[choice])
... except KeyError:
...     print('Bad choice')
...
Bad choice
```

Handling larger actions
Dictionaries are good for associating values with keys, but what about the more complicated actions you can code in the statement blocks associated with if statements?

LaterV, you'll learn that dictionaries can also contain functions to represent more complex branch actions and implement general jump tables. Such functions appear as dictionary values, they may be coded as function names or inline lambdas, and they are called by adding parentheses to trigger their actions. Here's an abstract sampler:

```
def function(): ...
def default(): ...
branch = {'spam': lambda: ..., # A table of callable function objects
          'ham': function,
          'eggs': lambda: ...}
branch.get(choice, default)()
```

Although dictionary-based multiway branching is useful in programs that deal with more dynamic data, most programmers will probably find that coding an if statement is the most straightforward way to perform multiway branching. As a rule of thumb in coding, when in doubt, err on the side of simplicity and readability; it's the "Pythonic" way.

Python syntax rules

Now that we're stepping up to larger statements like if, this section reviews and expands on the syntax ideas introduced earlier. In general, Python has a simple, statement-based syntax. However, there are a few properties you need to know about:

- **Statements execute one after another, until you say otherwise.** Python normally runs statements in a file or nested block in order from first to last as a sequence, but statements like if (as well as loops and exceptions) cause the interpreter to jump around in your code. Because Python's path through a program is called the control flow, statements such as if that affect it are often called controlflow statements.
- **Block and statement boundaries are detected automatically.** As we've seen, there are no braces or "begin/end" delimiters around blocks of code in Python; instead, Python uses the indentation of statements under a header to group the statements in a nested block. Similarly, Python statements are not normally terminated with semicolons; rather, the end of a line usually marks the end of the statement coded on that line. As a special case, statements can span lines and be combined on a line with special syntax.
- **Compound statements = header + ":" + indented statements.** All Python compound statements—those with nested statements—follow the same pattern: a header line terminated with a colon, followed by one or more nested statements, usually indented under the header. The indented statements are called a block (or sometimes, a suite). In the if statement, the elif and else clauses are part of the if, but they are also header lines with nested blocks of their own. As a special case, blocks can show up on the same line as the header if they are simple noncompound code.
- **Blank lines, spaces, and comments are usually ignored.** Blank lines are both optional and ignored in files (but not at the interactive prompt, when they terminate compound statements). Spaces inside statements and expressions are almost always ignored (except in string literals, and when used for indentation). Comments are always ignored: they start with a # character (not inside a string literal) and extend to the end of the current line.
- **Docstrings are ignored but are saved and displayed by tools.** Python supports an additional comment form called documentation strings (docstrings for short), which, unlike # comments, are retained at runtime for inspection. Docstrings are simply strings that show up at the top of program files and some statements. Python ignores their contents, but they are automatically attached to objects at runtime and may be displayed with documentation tools like PyDoc.

As you've seen, there are no variable type declarations in Python; this fact alone makes for a much simpler language syntax than what you may be used to. However, for most new users the lack of the braces and semicolons used to mark blocks and statements in many other languages seems to be the most novel syntactic feature of Python, so let's explore what this means in more detail.

<u>Block Delimiters: Indentation Rules</u>

As introduced, Python detects block boundaries automatically, by line indentation—that is, the empty space to the left of your code. All statements indented the same distance to the right belong to the same block of code. In other words, the statements within a block line up vertically, as in a column. The block ends when the end of the file or a lesser-indented line is encountered, and more deeply nested blocks are simply indented further to the right than the statements in the enclosing block.

Compound statement bodies can appear on the header's line in some cases we'll explore later, but most are indented under it.

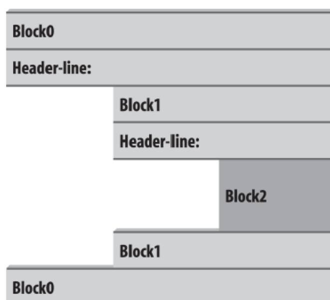


Figure: *Nested blocks of code: a nested block starts with a statement indented further to the right and ends with either a statement that is indented less, or the end of the file.*

For instance, previous Figure demonstrates the block structure of the following code:

```
x = 1
if x:
    y = 2
    if y:
        print('block2')
  print('block1')
print('block0')
```

This code contains three blocks: the first (the top-level code of the file) is not indented at all, the second (within the outer if statement) is indented four spaces, and the third (the print statement under the nested if) is indented eight spaces.

In general, top-level (unnested) code must start in column 1. Nested blocks can start in any column; indentation may consist of any number of spaces and tabs, as long as it's the same for all the statements in a given single block. That is, Python doesn't care how you indent your code; it only cares that it's done consistently. Four spaces or one tab per indentation level are common conventions, but there is no absolute standard in the Python world.

Indenting code is quite natural in practice. For example, the following (arguably silly) code snippet demonstrates common indentation errors in Python code:

```
    x = 'SPAM' # Error: first line indented
if 'rubbery' in 'shrubbery':
        print(x * 8)
            x += 'NI' # Error: unexpected indentation
            if x.endswith('NI'):
                   x *= 2
                print(x) # Error: inconsistent indentation
```

The properly indented version of this code looks like the following—even for an artificial example like this, proper indentation makes the code's intent much more apparent:

```
x = 'SPAM'
if 'rubbery' in 'shrubbery':
    print(x * 8) # Prints 8 "SPAM"
    x += 'NI'
    if x.endswith('NI'):
        x *= 2
        print(x) # Prints "SPAMNISPAMNI"
```

It's important to know that the only major place in Python where whitespace matters is where it's used to the left of your code, for indentation; in most other contexts, space can be coded or not. However, indentation is really part of Python syntax, not just a stylistic suggestion: all the statements within any given single block must be indented to the same level, or Python reports a syntax error. This is intentional—because you don't need to explicitly mark the start and end of a nested block of code, some of the syntactic clutter found in other languages is unnecessary in Python.

Making indentation part of the syntax model also enforces consistency, a crucial component of readability in structured programming languages like Python. Python's syntax is sometimes described as "what you see is what you get" —the indentation of each line of code unambiguously tells readers what it is associated with. This uniform and consistent appearance makes Python code easier to maintain and reuse.

Indentation is simpler in practice than its details might initially imply, and it makes your code reflect its logical structure. Consistently indented code always satisfies Python's rules. Moreover, most text editors (including IDLE) make it easy to follow Python's indentation model by automatically indenting code as you type it.

Statement Delimiters: Lines and Continuations

A statement in Python normally ends at the end of the line on which it appears. When a statement is too long to fit on a single line, though, a few special rules may be used to make it span multiple lines:

- **Statements may span multiple lines if you're continuing an open syntactic**
- pair. Python lets you continue typing a statement on the next line if you're coding something enclosed in a (), {}, or [] pair. For instance, expressions in parentheses and dictionary and list literals can span any number of lines; your statement doesn't end until the Python interpreter reaches the line on which you type the closing part of the pair (a), }, or]). Continuation lines—lines 2 and beyond of the statement —can start at any indentation level you like, but you should try to make them align vertically for readability if possible. This open pairs rule also covers set and dictionary comprehensions in Python 3.X and 2.7.
- **Statements may span multiple lines if they end in a backslash.** This is a somewhat outdated feature that's not generally recommended, but if a statement needs to span multiple lines, you can also add a backslash (a \ not embedded in a string literal or comment) at the end of the prior line to indicate you're continuing on the next line. Because you can also continue by adding parentheses around most constructs, backslashes are rarely used today. This approach is also error-prone: accidentally forgetting a \ usually generates a syntax error and might even cause the next line to be silently mistaken (i.e., without warning) for a new statement, with unexpected results.
- **Special rules for string literals**. Triple-quoted string blocks are designed to span multiple lines normally. Adjacent string literals are implicitly concatenated; when it's used in conjunction with the open pairs rule mentioned earlier, wrapping this construct in parentheses allows it to span multiple lines.
- **Other rules.** There are a few other points to mention with regard to statement delimiters. Although it is uncommon, you can terminate a statement with a semicolon—this convention is sometimes used to squeeze more than one simple (noncompound) statement onto a single line. Also, comments and blank lines can appear anywhere in a file; comments (which begin with a # character) terminate at the end of the line on which they appear.

Truth tests

The notions of comparison, equality, and truth values were introduced before.

Because the if statement is the first statement we've looked at that actually uses test results, we'll expand on some of these ideas here. In particular, Python's Boolean operators are a bit different from their counterparts in languages like C. In Python:

- All objects have an inherent Boolean true or false value.
- Any nonzero number or nonempty object is true.
- Zero numbers, empty objects, and the special object None are considered false.
- Comparisons and equality tests are applied recursively to data structures.
- Comparisons and equality tests return True or False (custom versions of 1 and 0).
- Boolean and and or operators return a true or false operand object.
- Boolean operators stop evaluating ("short circuit") as soon as a result is known.

The if statement takes action on truth values, but Boolean operators are used to combine the results of other tests in richer ways to produce new truth values. More formally, there are three Boolean expression operators in Python:

```
X and Y
Is true if both X and Y are true
X or Y
Is true if either X or Y is true
not X
Is true if X is false (the expression returns True or False)
```

Here, X and Y may be any truth value, or any expression that returns a truth value (e.g., an equality test, range comparison, and so on). Boolean operators are typed out as words in Python (instead of C's &&, ||, and !). Also, Boolean and and or operators return a true or false object in Python, not the values True or False. Let's look at a few examples to see how this works:

```
>>> 2 < 3, 3 < 2 # Less than: return True or False (1 or 0)
(True, False)
```

Magnitude comparisons such as these return True or False as their truth results, which, as we learned before, are really just custom versions of the integers 1 and 0 (they print themselves differently but are otherwise the same).

On the other hand, the and and or operators always return an object—either the object on the left side of the operator or the object on the right. If we test their results in if or other statements, they will be as expected (remember, every object is inherently true or false), but we won't get back a simple True or False.

For or tests, Python evaluates the operand objects from left to right and returns the first one that is true. Moreover, Python stops at the first true operand it finds. This is usually called short-circuit evaluation, as determining a result short-circuits (terminates) the rest of the expression as soon as the result is known:

```
>>> 2 or 3, 3 or 2 # Return left operand if true
(2, 3) # Else, return right operand (true or false)
>>> [] or 3
3
>>> [] or {}
{}
```

In the first line of the preceding example, both operands (2 and 3) are true (i.e., are nonzero), so Python always stops and returns the one on the left—it determines the result because true or anything is always true. In the other two tests, the left operand is false (an empty object), so Python simply evaluates and returns the object on the right—which may happen to have either a true or a false value when tested.

Python and operations also stop as soon as the result is known; however, in this case Python evaluates the operands from left to right and stops if the left operand is a false object because it determines the result—false and anything is always false:

```
>>> 2 and 3, 3 and 2 # Return left operand if false
(3, 2) # Else, return right operand (true or false)
>>> [] and {}
[]
>>> 3 and []
[]
```

Here, both operands are true in the first line, so Python evaluates both sides and returns the object on the right. In the second test, the left operand is false ([]), so Python stops and returns it as the test result. In the last test, the left side is true (3), so Python evaluates and returns the object on the right—which happens to be a false [].

The end result of all this is the same as in C and most other languages—you get a value that is logically true or false if tested in an if or while according to the normal definitions of or and and. However, in Python Booleans return either the left or the right object, not a simple integer flag.

This behavior of and and or may seem esoteric at first glance, sometimes used to advantage in coding by Python programmers.

The if/else Ternary Expression

One common role for the prior section's Boolean operators is to code an expression that runs the same as an if statement. Consider the following statement, which sets A to either Y or Z, based on the truth value of X:

```
if X:
   A = Y
else:
   A = Z
```

Sometimes, though, the items involved in such a statement are so simple that it seems like overkill to spread them across four lines. At other times, we may want to nest such a construct in a larger statement instead of assigning its

result to a variable. For these reasons (and, frankly, because the C language has a similar tool), Python 2.5 introduced a new expression format that allows us to say the same thing in one expression:

```
A = Y if X else Z
```

This expression has the exact same effect as the preceding four-line if statement, but it's simpler to code. As in the statement equivalent, Python runs expression Y only if X turns out to be true, and runs expression Z only if X turns out to be false. That is, it short-circuits, just like the Boolean operators described in the prior section, running just Y or Z but not both. Here are some examples of it in action:

```
>>> A = 't' if 'spam' else 'f'        # For strings, nonempty means true
>>> A
't'
>>> A = 't' if '' else 'f'
>>> A
'f'
```

Prior to Python 2.5 (and after 2.5, if you insist), the same effect can often be achieved by a careful combination of the and and or operators, because they return either the object on the left side or the object on the right as the preceding section described:

```
A = ((X and Y) or Z)
```

This works, but there is a catch—you have to be able to assume that Y will be Boolean true. If that is the case, the effect is the same: the and runs first and returns Y if X is true; if X if false the and skips Y, and the or simply returns Z. In other words, we get "if X then Y else Z." This is equivalent to the ternary form:

```
A = Y if X else Z
```

The and/or combination form also seems to require a "moment of great clarity" to understand the first time you see it, and it's no longer required as of 2.5—use the equivalent and more robust and mnemonic if/else expression when you need this structure, or use a full if statement if the parts are nontrivial.

As a side note, using the following expression in Python is similar because the bool function will translate X into the equivalent of integer 1 or 0, which can then be used as offsets to pick true and false values from a list:

```
A = [Z, Y][bool(X)]
```

For example:

```
>>> ['f', 't'][bool('')]
'f'
>>> ['f', 't'][bool('spam')]
't'
```

However, this isn't exactly the same, because Python will not short-circuit—it will always run both Z and Y, regardless of the value of X. Because of such complexities, you're better off using the simpler and more easily understood if/else expression as of Python 2.5 and later. Again, though, you should use even that sparingly, and only if its parts are all fairly simple; otherwise, you're better off coding the full if statement form to make changes easier in the future. Your coworkers will be happy you did.

Still, you may see the and/or version in code written prior to 2.5 (and in Python code written by ex–C programmers who haven't quite let go of their dark coding pasts).

Module 11 - While And For Loops

This module concludes our tour of Python procedural statements by presenting the language's two main looping constructs—statements that repeat an action over and over. The first of these, the while statement, provides a way to code general loops. The second, the for statement, is designed for stepping through the items in a sequence or other iterable object and running a block of code for each.

We've seen both of these informally already, but we'll fill in additional usage details here. While we're at it, we'll also study a few less prominent statements used within loops, such as break and continue, and cover some built-ins commonly used with loops, such as range, zip, and map.

Although the while and for statements covered here are the primary syntax provided for coding repeated actions, there are additional looping operations and concepts in Python. Because of that, the iteration story is continued later, where we'll explore the related ideas of Python's iteration protocol (used by the for loop) and list comprehensions (a close cousin to the for loop). Later we will explore even more exotic iteration tools such as generators, filter, and reduce. For now, though, let's keep things simple.

While loops

Python's while statement is the most general iteration construct in the language. In simple terms, it repeatedly executes a block of (normally indented) statements as long as a test at the top keeps evaluating to a true value. It is called a "loop" because control keeps looping back to the start of the statement until the test becomes false. When the test becomes false, control passes to the statement that follows the while block. The net effect is that the loop's body is executed repeatedly while the test at the top is true.

If the test is false to begin with, the body never runs and the while statement is skipped.

General Format

In its most complex form, the while statement consists of a header line with a test expression, a body of one or more normally indented statements, and an optional else part that is executed if control exits the loop without a break statement being encountered. Python keeps evaluating the test at the top and executing the statements nested in the loop body until the test returns a false value:

```
while test:                 # Loop test
    statements              # Loop body
else:                       # Optional else
    statements              # Run if didn't exit loop with break
```

Examples

To illustrate, let's look at a few simple while loops in action. The first, which consists of a print statement nested in a while loop, just prints a message forever. Recall that True is just a custom version of the integer 1 and always stands for a Boolean true value; because the test is always true, Python keeps executing the body forever, or until you stop its execution. This sort of behavior is usually called an infinite loop—it's not really immortal, but you may need a Ctrl-C key combination to forcibly terminate one:

```
>>> while True:
...     print('Type Ctrl-C to stop me!')
```

The next example keeps slicing off the first character of a string until the string is empty and hence false. It's typical to test an object directly like this instead of using the more verbose equivalent (while x != '':). Later in this module, we'll see other ways to step through the items in a string more easily with a for loop.

```
>>> x = 'spam'
>>> while x:               # While x is not empty
...     print(x, end=' ')  # In 2.X use print x,
...     x = x[1:]          # Strip first character off x
...
spam pam am m
```

Note the end=' ' keyword argument used here to place all outputs on the same line separated by a space. This may leave your input prompt in an odd state at the end of your output; type Enter to reset. Python 2.X developers: also remember to use a trailing comma instead of end in the prints like this.

The following code counts from the value of a up to, but not including, b. We'll also see an easier way to do this with a Python for loop and the built-in range function later:

```
>>> a=0; b=10
>>> while a < b:          # One way to code counter loops
...     print(a, end=' ')
...     a += 1              # Or, a = a + 1
...
0 1 2 3 4 5 6 7 8 9
```

Finally, notice that Python doesn't have what some languages call a "do until" loop statement. However, we can simulate one with a test and break at the bottom of the loop body, so that the loop's body is always run at least once:

```
while True:
   ...loop body...
   if exitTest(): break
```

To fully understand how this structure works, we need to move on to the next section and learn more about the break statement.

Break, continue, pass, and the loop else

Now that we've seen a few Python loops in action, it's time to take a look at two simple statements that have a purpose only when nested inside loops—the break and continue statements. While we're looking at oddballs, we will also study the loop else clause here because it is intertwined with break, and Python's empty placeholder statement, pass (which is not tied to loops per se, but falls into the general category of simple one-word statements). In Python:

break
    Jumps out of the closest enclosing loop (past the entire loop statement)

continue
    Jumps to the top of the closest enclosing loop (to the loop's header line)

pass
    Does nothing at all: it's an empty statement placeholder

Loop else block
    Runs if and only if the loop is exited normally (i.e., without hitting a break)

General Loop Format

Factoring in break and continue statements, the general format of the while loop looks like this:

```
while test:
   statements
   if test: break          # Exit loop now, skip else if present
   if test: continue       # Go to top of loop now, to test1
else:
   statements              # Run if we didn't hit a 'break'
```

break and continue statements can appear anywhere inside the while (or for) loop's body, but they are usually coded further nested in an if test to take action in response to some condition.

Let's turn to a few simple examples to see how these statements come together in practice.

Pass

Simple things first: the pass statement is a no-operation placeholder that is used when the syntax requires a statement, but you have nothing useful to say. It is often used to code an empty body for a compound statement. For instance, if you want to code an infinite loop that does nothing each time through, do it with a pass:

```
while True: pass            # Type Ctrl-C to stop me!
```

Because the body is just an empty statement, Python gets stuck in this loop. pass is roughly to statements as None is to objects—an explicit nothing. Notice that here the while loop's body is on the same line as the header, after the colon; as with if statements, this only works if the body isn't a compound statement.

This example does nothing forever. It probably isn't the most useful Python program ever written (unless you want to warm up your laptop computer on a cold winter's day!); frankly, though, I couldn't think of a better pass example at this point in the course.

We'll see other places where pass makes more sense later—for instance, to ignore exceptions caught by try statements, and to define empty class objects with attributes that behave like "structs" and "records" in other languages. A pass is also sometime coded to mean "to be filled in later," to stub out the bodies of functions temporarily:

```
def func1():
    pass            # Add real code here later
def func2():
    pass
```

We can't leave the body empty without getting a syntax error, so we say pass instead.

continue

The continue statement causes an immediate jump to the top of a loop. It also sometimes lets you avoid statement nesting. The next example uses continue to skip odd numbers. This code prints all even numbers less than 10 and greater than or equal to 0. Remember, 0 means false and % is the remainder of division (modulus) operator, so this loop counts down to 0, skipping numbers that aren't multiples of 2—it prints 8 6 4 2 0:

```
x = 10
while x:
    x = x−1                         # Or, x -= 1
    if x % 2 != 0: continue         # Odd? -- skip print
    print(x, end=' ')
```

Because continue jumps to the top of the loop, you don't need to nest the print statement here inside an if test; the print is only reached if the continue is not run. If this sounds similar to a "go to" in other languages, it should. Python has no "go to" statement, but because continue lets you jump about in a program, many of the warnings about readability and maintainability you may have heard about "go to" apply. continue should probably be used sparingly, especially when you're first getting started with Python. For instance, the last example might be clearer if the print were nested under the if:

```
x = 10
while x:
    x = x−1
    if x % 2 == 0:          # Even? -- print
        print(x, end=' ')
```

Later in this course, we'll also learn that raised and caught exceptions can also emulate "go to" statements in limited and structured ways; stay tuned for more on this technique.

break

The break statement causes an immediate exit from a loop. Because the code that follows it in the loop is not executed if the break is reached, you can also sometimes avoid nesting by including a break. For example, here is a simple interactive loop that inputs data with input (known as raw_input in Python 2.X) and exits when the user enters “stop” for the name request:

```
>>> while True:
...     name = input('Enter name:')            # Use raw_input() in 2.X
...     if name == 'stop': break
...     age = input('Enter age: ')
...     print('Hello', name, '=>', int(age) ** 2)
...
Enter name:bob
Enter age: 40
Hello bob => 1600
Enter name:sue
Enter age: 30
Hello sue => 900
Enter name:stop
```

Notice how this code converts the age input to an integer with int before raising it to the second power; as you’ll recall, this is necessary because input returns user input as a string.

Loop else

When combined with the loop else clause, the break statement can often eliminate the need for the search status flags used in other languages. For instance, the following piece of code determines whether a positive integer y is prime by searching for factors greater than 1:

```
x = y // 2                 # For some y > 1
while x > 1:
    if y % x == 0:         # Remainder
        print(y, 'has factor', x)
        break              # Skip else
      x -= 1
    else: # Normal exit
        print(y, 'is prime')
```

Rather than setting a flag to be tested when the loop is exited, it inserts a break where a factor is found. This way, the loop else clause can assume that it will be executed only if no factor is found; if you don’t hit the break, the number is prime. Trace through this code to see how this works.

The loop else clause is also run if the body of the loop is never executed, as you don’t run a break in that event either; in a while loop, this happens if the test in the header is false to begin with. Thus, in the preceding example you still get the “is prime” message if x is initially less than or equal to 1 (for instance, if y is 2).

For loops

The for loop is a generic iterator in Python: it can step through the items in any ordered sequence or other iterable object. The for statement works on strings, lists, tuples, and other built-in iterables, as well as new user-defined objects that we’ll learn how to create later with classes. We met for briefly and in conjunction with sequence object types; let’s expand on its usage more formally here.

General Format

The Python for loop begins with a header line that specifies an assignment target (or targets), along with the object you want to step through. The header is followed by a block of (normally indented) statements that you want to repeat:

```
for target in object:               # Assign object items to target
```

```
    statements                          # Repeated loop body: use target
else:                                   # Optional else part
    statements                          # If we didn't hit a 'break'
```

When Python runs a for loop, it assigns the items in the iterable object to the target one by one and executes the loop body for each. The loop body typically uses the assignment target to refer to the current item in the sequence as though it were a cursor stepping through the sequence.

The name used as the assignment target in a for header line is usually a (possibly new) variable in the scope where the for statement is coded. There's not much unique about this name; it can even be changed inside the loop's body, but it will automatically be set to the next item in the sequence when control returns to the top of the loop again.

After the loop this variable normally still refers to the last item visited, which is the last item in the sequence unless the loop exits with a break statement.

The for statement also supports an optional else block, which works exactly as it does in a while loop—it's executed if the loop exits without running into a break statement (i.e., if all items in the sequence have been visited). The break and continue statements introduced earlier also work the same in a for loop as they do in a while. The for loop's complete format can be described this way:

```
for target in object:                   # Assign object items to target
    statements
    if test: break                      # Exit loop now, skip else
        if test: continue                       # Go to top of loop now
else:
    statements                          # If we didn't hit a 'break'
```

Examples

Let's type a few for loops interactively now, so you can see how they are used in practice.

Basic usage

As mentioned earlier, a for loop can step across any kind of sequence object. In our first example, for instance, we'll assign the name x to each of the three items in a list in turn, from left to right, and the print statement will be executed for each. Inside the print statement (the loop body), the name x refers to the current item in the list:

```
>>> for x in ["spam", "eggs", "ham"]:
...     print(x, end=' ')
...
spam eggs ham
```

The next two examples compute the sum and product of all the items in a list. Later in this module and later in the course we'll meet tools that apply operations such as + and * to items in a list automatically, but it's often just as easy to use a for:

```
>>> sum = 0
>>> for x in [1, 2, 3, 4]:
...     sum = sum + x
...
>>> sum
10
>>> prod = 1
>>> for item in [1, 2, 3, 4]: prod *= item
...
>>> prod
24
```

Other data types

Any sequence works in a for, as it's a generic tool. For example, for loops work on strings and tuples:

```
>>> S = "lumberjack"
>>> T = ("and", "I'm", "okay")
>>> for x in S: print(x, end=' ') # Iterate over a string
...
l u m b e r j a c k
>>> for x in T: print(x, end=' ') # Iterate over a tuple
...
and I'm okay
```

In fact, as we'll learn in the later when we explore the notion of "iterables," for loops can even work on some objects that are not sequences—files and dictionaries work, too.

Tuple assignment in for loops

If you're iterating through a sequence of tuples, the loop target itself can actually be a tuple of targets. This is just another case of the tuple-unpacking assignment we studied. Remember, the for loop assigns items in the sequence object to the target, and assignment works the same everywhere:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T: # Tuple assignment at work
...     print(a, b)
...
1 2
3 4
5 6
```

Here, the first time through the loop is like writing (a,b) = (1,2), the second time is like writing (a,b) = (3,4), and so on. The net effect is to automatically unpack the current tuple on each iteration.

This form is commonly used in conjunction with the zip call we'll meet later in this module to implement parallel traversals. It also makes regular appearances in conjunction with SQL databases in Python, where query result tables are returned as sequences of sequences like the list used here—the outer list is the database table, the nested tuples are the rows within the table, and tuple assignment extracts columns.

Tuples in for loops also come in handy to iterate through both keys and values in dictionaries using the items method, rather than looping through the keys and indexing to fetch the values manually:

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> for key in D:
... print(key, '=>', D[key])        # Use dict keys iterator and index
...
a => 1
c => 3
b => 2
>>> list(D.items())
[('a', 1), ('c', 3), ('b', 2)]
>>> for (key, value) in D.items():
...     print(key, '=>', value)     # Iterate over both keys and values
...
a => 1
c => 3
b => 2
```

It's important to note that tuple assignment in for loops isn't a special case; any assignment target works syntactically after the word for. We can always assign manually within the loop to unpack:

```
>>> T
```

```
[(1, 2), (3, 4), (5, 6)]
>>> for both in T:
...     a, b = both          # Manual assignment equivalent
...     print(a, b)          # 2.X: prints with enclosing tuple "()"
...
1 2
3 4
5 6
```

But tuples in the loop header save us an extra step when iterating through sequences of sequences. As suggested before, even nested structures may be automatically unpacked this way in a for:

```
>>> ((a, b), c) = ((1, 2), 3)          # Nested sequences work too
>>> a, b, c
(1, 2, 3)
>>> for ((a, b), c) in [((1, 2), 3), ((4, 5), 6)]: print(a, b, c)
...
1 2 3
4 5 6
```

Even this is not a special case, though—the for loop simply runs the sort of assignment we ran just before it, on each iteration. Any nested sequence structure may be unpacked this way, simply because sequence assignment is so generic:

```
>>> for ((a, b), c) in [([1, 2], 3), ['XY', 6]]: print(a, b, c)
...
1 2 3
X Y 6
```

Nested for loops

Now let's look at a for loop that's a bit more sophisticated than those we've seen so far. The next example illustrates statement nesting and the loop else clause in a for.

Given a list of objects (items) and a list of keys (tests), this code searches for each key in the objects list and reports on the search's outcome:

```
>>> items = ["aaa", 111, (4, 5), 2.01] # A set of objects
>>> tests = [(4, 5), 3.14] # Keys to search for
>>>
>>> for key in tests: # For all keys
...     for item in items: # For all items
...         if item == key: # Check for match
...         print(key, "was found")
...         break
...   else:
...  print(key, "not found!")
...
(4, 5) was found
3.14 not found!
```

Because the nested if runs a break when a match is found, the loop else clause can assume that if it is reached, the search has failed. Notice the nesting here. When this code runs, there are two loops going at the same time: the outer loop scans the keys list, and the inner loop scans the items list for each key. The nesting of the loop else clause is critical; it's indented to the same level as the header line of the inner for loop, so it's associated with the inner loop, not the if or the outer for.

This example is illustrative, but it may be easier to code if we employ the in operator to test membership. Because in implicitly scans an object looking for a match (at least logically), it replaces the inner loop:

```
>>> for key in tests:               # For all keys
...     if key in items:            # Let Python check for a match
...         print(key, "was found")
...     else:
...         print(key, "not found!")
...
(4, 5) was found
not found!
```

In general, it's a good idea to let Python do as much of the work as possible (as in this solution) for the sake of brevity and performance.

The next example is similar, but builds a list as it goes for later use instead of printing. It performs a typical data-structure task with a for—collecting common items in two sequences (strings)—and serves as a rough set intersection routine. After the loop runs, res refers to a list that contains all the items found in seq1 and seq2:

```
>>> seq1 = "spam"
>>> seq2 = "scam"
>>>
>>> res = []              # Start empty
>>> for x in seq1:        # Scan first sequence
...     if x in seq2:     # Common item?
...         res.append(x) # Add to result end
...
>>> res
['s', 'a', 'm']
```

Unfortunately, this code is equipped to work only on two specific variables: seq1 and seq2. It would be nice if this loop could somehow be generalized into a tool you could use more than once. As you'll see, that simple idea leads us to functions, the topic of the next part of the course.

This code also exhibits the classic list comprehension pattern—collecting a results list with an iteration and optional filter test—and could be coded more concisely too:

```
>>> [x for x in seq1 if x in seq2]          # Let Python collect results
['s', 'a', 'm']
```

Loop variations

The for loop we just studied subsumes most counter-style loops. It's generally simpler to code and often quicker to run than a while, so it's the first tool you should reach for whenever you need to step through a sequence or other iterable. In fact, as a general rule, you should resist the temptation to count things in Python—its iteration tools automate much of the work you do to loop over collections in lower-level languages like C.

Still, there are situations where you will need to iterate in more specialized ways. For example, what if you need to visit every second or third item in a list, or change the list along the way? How about traversing more than one sequence in parallel, in the same for loop? What if you need indexes too?

You can always code such unique iterations with a while loop and manual indexing, but Python provides a set of built-ins that allow you to specialize the iteration in a for:

- The built-in range function (available since Python 0.X) produces a series of successively higher integers, which can be used as indexes in a for.
- The built-in zip function (available since Python 2.0) returns a series of parallelitem tuples, which can be used to traverse multiple sequences in a for.
- The built-in enumerate function (available since Python 2.3) generates both the values and indexes of items in an iterable, so we don't need to count manually.
- The built-in map function (available since Python 1.0) can have a similar effect to zip in Python 2.X, though this role is removed in 3.X.

Because for loops may run quicker than while-based counter loops, though, it's to your advantage to use tools like these that allow you to use for whenever possible. Let's look at each of these built-ins in turn, in the context of common use cases. As we'll see, their usage may differ slightly between 2.X and 3.X, and some of their applications are more valid than others.

Counter Loops: range

Our first loop-related function, range, is really a general tool that can be used in a variety of contexts. Although it's used most often to generate indexes in a for, you can use it anywhere you need a series of integers. In Python 2.X range creates a physical list; in 3.X, range is an iterable that generates items on demand, so we need to wrap it in a list call to display its results all at once in 3.X only:

```
>>> list(range(5)), list(range(2, 5)), list(range(0, 10, 2))
([0, 1, 2, 3, 4], [2, 3, 4], [0, 2, 4, 6, 8])
```

With one argument, range generates a list of integers from zero up to but not including the argument's value. If you pass in two arguments, the first is taken as the lower bound. An optional third argument can give a step; if it is used, Python adds the step to each successive integer in the result (the step defaults to +1). Ranges can also be nonpositive and nonascending, if you want them to be:

```
>>> list(range(−5, 5))
[−5, −4, −3, −2, −1, 0, 1, 2, 3, 4]
>>> list(range(5, −5, −1))
[5, 4, 3, 2, 1, 0, −1, −2, −3, −4]
```

We'll get more formal about iterables like this one later. There, we'll also see that Python 2.X has a cousin named xrange, which is like its range but doesn't build the result list in memory all at once. This is a space optimization, which is subsumed in 3.X by the generator behavior of its range.

Although such range results may be useful all by themselves, they tend to come in most handy within for loops. For one thing, they provide a simple way to repeat an action a specific number of times. To print three lines, for example, use a range to generate the appropriate number of integers:

```
>>> for i in range(3):
...     print(i, 'Pythons')
...
0 Pythons
```

```
1 Pythons
2 Pythons
```

Note that for loops force results from range automatically in 3.X, so we don't need to use a list wrapper here in 3.X (in 2.X we get a temporary list unless we call xrange instead).

Sequence Scans: while and range Versus for

The range call is also sometimes used to iterate over a sequence indirectly, though it's often not the best approach in this role. The easiest and generally fastest way to step through a sequence exhaustively is always with a simple for, as Python handles most of the details for you:

```
>>> X = 'spam'
>>> for item in X: print(item, end=' ') # Simple iteration
...
s p a m
```

Internally, the for loop handles the details of the iteration automatically when used this way. If you really need to take over the indexing logic explicitly, you can do it with a while loop:

```
>>> i = 0
>>> while i < len(X): # while loop iteration
... print(X[i], end=' ')
... i += 1
...
s p a m
```

You can also do manual indexing with a for, though, if you use range to generate a list of indexes to iterate through. It's a multistep process, but it's sufficient to generate offsets, rather than the items at those offsets:

```
>>> X
'spam'
>>> len(X) # Length of string
4
>>> list(range(len(X))) # All legal offsets into X
[0, 1, 2, 3]
>>>
>>> for i in range(len(X)): print(X[i], end=' ') # Manual range/len iteration
...
s p a m
```

Note that because this example is stepping over a list of offsets into X, not the actual items of X, we need to index back into X within the loop to fetch each item. If this seems like overkill, though, it's because it is: there's really no reason to work this hard in this example.

Although the range/len combination suffices in this role, it's probably not the best option. It may run slower, and it's also more work than we need to do. Unless you have a special indexing requirement, you're better off using the simple for loop form in Python:

```
>>> for item in X: print(item, end=' ') # Use simple iteration if you can
```

As a general rule, use for instead of while whenever possible, and don't use range calls in for loops except as a last resort. This simpler solution is almost always better. Like every good rule, though, there are plenty of exceptions—as the next section demonstrates.

Sequence Shufflers: range and len

Though not ideal for simple sequence scans, the coding pattern used in the prior example does allow us to do more specialized sorts of traversals when required. For example, some algorithms can make use of sequence reordering—to generate alternatives in searches, to test the effect of different value orderings, and so on. Such cases may require offsets in order to pull sequences apart and put them back together, as in the following; the range's integers provide a repeat count in the first, and a position for slicing in the second:

```
>>> S = 'spam'
>>> for i in range(len(S)): # For repeat counts 0..3
...     S = S[1:] + S[:1] # Move front item to end
...     print(S, end=' ')
...
pams amsp mspa spam
>>> S
'spam'
>>> for i in range(len(S)): # For positions 0..3
...     X = S[i:] + S[:i] # Rear part + front part
...     print(X, end=' ')
...
spam pams amsp mspa
```

Trace through these one iteration at a time if they seem confusing. The second creates the same results as the first, though in a different order, and doesn't change the original variable as it goes. Because both slice to obtain parts to concatenate, they also work on any type of sequence, and return sequences of the same type as that being shuffled—if you shuffle a list, you create reordered lists:

```
>>> L = [1, 2, 3]
>>> for i in range(len(L)):
...     X = L[i:] + L[:i] # Works on any sequence type
...     print(X, end=' ')
...
[1, 2, 3] [2, 3, 1] [3, 1, 2]
```

We'll make use of code like this to test functions with different argument orderings in later, and will extend it to functions, generators, and more complete permutations—it's a widely useful tool.

Nonexhaustive Traversals: range Versus Slices

Cases like that of the prior section are valid applications for the range/len combination. We might also use this technique to skip items as we go:

```
>>> S = 'abcdefghijk'
>>> list(range(0, len(S), 2))
[0, 2, 4, 6, 8, 10]
>>> for i in range(0, len(S), 2): print(S[i], end=' ')
...
a c e g i k
```

Here, we visit every second item in the string S by stepping over the generated range list. To visit every third item, change the third range argument to be 3, and so on. In effect, using range this way lets you skip items in loops while still retaining the simplicity of the for loop construct.

In most cases, though, this is also probably not the "best practice" technique in Python today. If you really mean to skip items in a sequence, the extended three-limit form of the slice expression, provides a simpler route to the same goal.

To visit every second character in S, for example, slice with a stride of 2:

```
>>> S = 'abcdefghijk'
```

```
>>> for c in S[::2]: print(c, end=' ')
...
a c e g i k
```

The result is the same, but substantially easier for you to write and for others to read.
The potential advantage to using range here instead is space: slicing makes a copy of the string in both 2.X and 3.X, while range in 3.X and xrange in 2.X do not create a list; for very large strings, they may save memory.

Changing Lists: range Versus Comprehensions

Another common place where you may use the range/len combination with for is in loops that change a list as it is being traversed. Suppose, for example, that you need to add 1 to every item in a list (maybe you're giving everyone a raise in an employee database list). You can try this with a simple for loop, but the result probably won't be exactly what you want:

```
>>> L = [1, 2, 3, 4, 5]
>>> for x in L:
...     x += 1 # Changes x, not L
...
>>> L
[1, 2, 3, 4, 5]
>>> x
6
```

This doesn't quite work—it changes the loop variable x, not the list L. The reason is somewhat subtle. Each time through the loop, x refers to the next integer already pulled out of the list. In the first iteration, for example, x is integer 1. In the next iteration, the loop body sets x to a different object, integer 2, but it does not update the list where 1 originally came from; it's a piece of memory separate from the list.

To really change the list as we march across it, we need to use indexes so we can assign an updated value to each position as we go. The range/len combination can produce the required indexes for us:

```
>>> L = [1, 2, 3, 4, 5]
>>> for i in range(len(L)): # Add one to each item in L
...     L[i] += 1 # Or L[i] = L[i] + 1
...
>>> L
[2, 3, 4, 5, 6]
```

When coded this way, the list is changed as we proceed through the loop. There is no way to do the same with a simple for x in L:–style loop, because such a loop iterates through actual items, not list positions. But what about the equivalent while loop? Such a loop requires a bit more work on our part, and might run more slowly depending on your Python:

```
>>> i = 0
>>> while i < len(L):
...     L[i] += 1
...     i += 1
...
>>> L
[3, 4, 5, 6, 7]
```

Here again, though, the range solution may not be ideal either. A list comprehension expression of the form:
[x + 1 for x in L] likely runs faster today and would do similar work, albeit without changing the original list in place (we could assign the expression's new list object result back to L, but this would not update any other references to the original list).

Parallel Traversals: zip and map

Our next loop coding technique extends a loop's scope. As we've seen, the range builtin allows us to traverse sequences with for in a nonexhaustive fashion. In the same spirit, the built-in zip function allows us to use for loops to visit multiple sequences in parallel—not overlapping in time, but during the same loop. In basic operation, zip takes one or more sequences as arguments and returns a series of tuples that pair up parallel items taken from those sequences. For example, suppose we're working with two lists (a list of names and addresses paired by position, perhaps):

```
>>> L1 = [1,2,3,4]
>>> L2 = [5,6,7,8]
```

To combine the items in these lists, we can use zip to create a list of tuple pairs. Like range, zip is a list in Python 2.X, but an iterable object in 3.X where we must wrap it in a list call to display all its results at once:

```
>>> zip(L1, L2)
<zip object at 0x026523C8>
>>> list(zip(L1, L2)) # list() required in 3.X, not 2.X
[(1, 5), (2, 6), (3, 7), (4, 8)]
```

Such a result may be useful in other contexts as well, but when wedded with the for loop, it supports parallel iterations:

```
>>> for (x, y) in zip(L1, L2):
...     print(x, y, '--', x+y)
...
1 5 -- 6
2 6 -- 8
3 7 -- 10
4 8 -- 12
```

Here, we step over the result of the zip call—that is, the pairs of items pulled from the two lists. Notice that this for loop again uses the tuple assignment form we met earlier to unpack each tuple in the zip result. The first time through, it's as though we ran the assignment statement (x, y) = (1, 5).

The net effect is that we scan both L1 and L2 in our loop. We could achieve a similar effect with a while loop that handles indexing manually, but it would require more typing and would likely run more slowly than the for/zip approach.

Strictly speaking, the zip function is more general than this example suggests. For instance, it accepts any type of sequence (really, any iterable object, including files), and it accepts more than two arguments. With three arguments, as in the following example, it builds a list of three-item tuples with items from each sequence, essentially projecting by columns (technically, we get an N-ary tuple for N arguments):

```
>>> T1, T2, T3 = (1,2,3), (4,5,6), (7,8,9)
>>> T3
(7, 8, 9)
>>> list(zip(T1, T2, T3)) # Three tuples for three arguments
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Moreover, zip truncates result tuples at the length of the shortest sequence when the argument lengths differ. In the following, we zip together two strings to pick out characters in parallel, but the result has only as many tuples as the length of the shortest sequence:

```
>>> S1 = 'abc'
>>> S2 = 'xyz123'
>>>
>>> list(zip(S1, S2)) # Truncates at len(shortest)
[('a', 'x'), ('b', 'y'), ('c', 'z')]
```

Generating Both Offsets and Items: enumerate

Our final loop helper function is designed to support dual usage modes. Earlier, we discussed using range to generate the offsets of items in a string, rather than the items at those offsets. In some programs, though, we need both: the item to use, plus an offset as we go. Traditionally, this was coded with a simple for loop that also kept a counter of the current offset:

```
>>> S = 'spam'
>>> offset = 0
>>> for item in S:
...     print(item, 'appears at offset', offset)
...     offset += 1
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

This works, but in all recent Python 2.X and 3.X releases (since 2.3) a new built-in named enumerate does the job for us—its net effect is to give loops a counter "for free," without sacrificing the simplicity of automatic iteration:

```
>>> S = 'spam'
>>> for (offset, item) in enumerate(S):
...     print(item, 'appears at offset', offset)
...
s appears at offset 0
p appears at offset 1
a appears at offset 2
m appears at offset 3
```

The enumerate function returns a generator object—a kind of object that supports the iteration protocol. In short, it has a method called by the next built-in function, which returns an (index, value) tuple each time through the loop. The for steps through these tuples automatically, which allows us to unpack their values with tuple assignment, much as we did for zip:

```
>>> E = enumerate(S)
>>> E
<enumerate object at 0x0000000002A8B900>
>>> next(E)
(0, 's')
>>> next(E)
(1, 'p')
>>> next(E)
(2, 'a')
```

We don't normally see this machinery because all iteration contexts—including list comprehensions—run the iteration protocol automatically:

```
>>> [c * i for (i, c) in enumerate(S)]
['', 'p', 'aa', 'mmm']
>>> for (i, l) in enumerate(open('test.txt')):
...     print('%s) %s' % (i, l.rstrip()))
...
0) aaaaaa
1) bbbbbb
2) cccccc
```

To fully understand iteration concepts like enumerate, zip, and list comprehensions, though, we need to move on for a more formal dissection later.

Module 12 - Function Basics

In Part III, we studied basic procedural statements in Python. Here, we'll move on to explore a set of additional statements and expressions that we can use to create functions of our own.

In simple terms, a function is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name. Functions also can compute a result value and let us specify parameters that serve as function inputs and may differ each time the code is run. Coding an operation as a function makes it a generally useful tool, which we can use in a variety of contexts.

More fundamentally, functions are the alternative to programming by cutting and pasting—rather than having multiple redundant copies of an operation's code, we can factor it into a single function. In so doing, we reduce our future work radically: if the operation must be changed later, we have only one copy to update in the function, not many scattered throughout the program.

Functions are also the most basic program structure Python provides for maximizing code reuse, and lead us to the larger notions of program design. As we'll see, functions let us split complex systems into manageable parts. By implementing each part as a function, we make it both reusable and easier to code.

Table below previews the primary function-related tools we'll study in this part of the course—a set that includes call expressions, two ways to make functions (def and lambda), two ways to manage scope visibility (global and nonlocal), and two ways to send results back to callers (return and yield).

Table: *Function-related statements and expressions*

Statement or expression	Examples
Call expressions	myfunc('spam', 'eggs', meat=ham, *rest)
def	def printer(messge): print('Hello ' + message)
return	def adder(a, b=1, *c): return a + b + c[0]
global	x = 'old' def changer(): global x; x = 'new'

nonlocal (3.X)	def outer():
	    x = 'old'
	    def changer():
	        nonlocal x; x = 'new'

yield	def squares(x): for i in range(x): yield i ** 2
lambda	funcs = [lambda x: x**2, lambda x: x**3]

Why use functions?

Before we get into the details, let's establish a clear picture of what functions are all about. Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called subroutines or procedures. As a brief introduction, functions serve two primary development roles:

- **Maximizing code reuse and minimizing redundancy**
  As in most programming languages, Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Up until now, all the code we've been writing has run immediately. Functions allow us to group and generalize code to be used arbitrarily many times later. Because they allow us to code an operation in a single place and use it in many places, Python functions are the most basic factoring tool in the language: they allow us to reduce code redundancy in our programs, and thereby reduce maintenance effort.
- **Procedural decomposition**
  Functions also provide a tool for splitting systems into pieces that have well-defined roles. For instance, to make a pizza from scratch, you would start by mixing the dough, rolling it out, adding toppings, baking it, and so on. If

you were programming a pizza-making robot, functions would help you divide the overall "make pizza" task into chunks—one function for each subtask in the process. It's easier to implement the smaller tasks in isolation than it is to implement the entire process at once. In general, functions are about procedure—how to do something, rather than what you're doing it to.

In this part of the course, we'll explore the tools used to code functions in Python: function basics, scope rules, and argument passing, along with a few related concepts such as generators and functional tools. Because its importance begins to become more apparent at this level of coding, we'll also revisit the notion of polymorphism, which was introduced earlier in the course. As you'll see, functions don't imply much new syntax, but they do lead us to some bigger programming ideas.

Coding functions

Although it wasn't made very formal, we've already used some functions befores. For instance, to make a file object, we called the built-in open function; similarly, we used the len built-in function to ask for the number of items in a collection object.

In this module, we will explore how to write new functions in Python. Functions we write behave the same way as the built-ins we've already seen: they are called in expressions, are passed values, and return results. But writing new functions requires the application of a few additional ideas that haven't yet been introduced. Moreover, functions behave very differently in Python than they do in compiled languages like C. Here is a brief introduction to the main concepts behind Python functions, all of which we will study in this part of the course:

- **def is executable code.** Python functions are written with a new statement, the def. Unlike functions in compiled languages such as C, def is an executable statement—your function does not exist until Python reaches and runs the def. In fact, it's legal (and even occasionally useful) to nest def statements inside if statements, while loops, and even other defs. In typical operation, def statements are coded in module files and are naturally run to generate functions when the module file they reside in is first imported.
- **def creates an object and assigns it to a name.** When Python reaches and runs a def statement, it generates a new function object and assigns it to the function's name. As with all assignments, the function name becomes a reference to the function object. There's nothing magic about the name of a function—as you'll see, the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined attributes attached to them to record data.
- **lambda creates an object but returns it as a result.** Functions may also be created with the lambda expression, a feature that allows us to in-line function definitions in places where a def statement won't work syntactically.
- **return sends a result object back to the caller.** When a function is called, the caller stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a return statement; the returned value becomes the result of the function call. A return without a value simply returns to the caller (and sends back None, the default result).
- **yield sends a result object back to the caller, but remembers where it left off.** Functions known as generators may also use the yield statement to send back a value and suspend their state such that they may be resumed later, to produce a series of results over time. This is another advanced topic covered later in this part of the course.
- **global declares module-level variables that are to be assigned.** By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a global statement. More generally, names are always looked up in scopes—places where variables are stored—and assignments bind names to scopes.
- **nonlocal declares enclosing function variables that are to be assigned.** Similarly, the nonlocal statement added in Python 3.X allows a function to assign a name that exists in the scope of a syntactically enclosing def statement. This allows enclosing functions to serve as a place to retain state—information remembered between function calls—without using shared global names.
- **Arguments are passed by assignment (object reference).** In Python, arguments are passed to functions by assignment (which, as we've learned, means by object reference). As you'll see, in Python's model the caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.

- **Arguments are passed by position, unless you say otherwise.** Values you pass in a function call match argument names in a function's definition from left to right by default. For flexibility, function calls can also pass arguments by name with name=value keyword syntax, and unpack arbitrarily many arguments to send with *pargs and **kargs starred-argument notation. Function definitions use the same two forms to specify argument defaults, and collect arbitrarily many arguments
- received.
- **Arguments, return values, and variables are not declared.** As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on. As one consequence, a single function can often be applied to a variety of object types—any objects that sport a compatible interface (methods and expressions) will do, regardless of their specific types.

If some of the preceding words didn't sink in, don't worry—we'll explore all of these concepts with real code in this part of the course. Let's get started by expanding on some of these ideas and looking at a few examples.

def Statements

The def statement creates a function object and assigns it to a name. Its general format is as follows:

```
def name(arg1, arg2,... argN):
    statements
```

As with all compound Python statements, def consists of a header line followed by a block of statements, usually indented (or a simple statement after the colon). The statement block becomes the function's body—that is, the code Python executes each time the function is later called.

The def header line specifies a function name that is assigned the function object, along with a list of zero or more arguments (sometimes called parameters) in parentheses.

The argument names in the header are assigned to the objects passed in parentheses at the point of call. Function bodies often contain a return statement:

```
def name(arg1, arg2,... argN):
   ...
   return value
```

The Python return statement can show up anywhere in a function body; when reached, it ends the function call and sends a result back to the caller. The return statement consists of an optional object value expression that gives the function's result. If the value is omitted, return sends back a None.

The return statement itself is optional too; if it's not present, the function exits when the control flow falls off the end of the function body. Technically, a function without a return statement also returns the None object automatically, but this return value is sually ignored at the call.

Functions may also contain yield statements, which are designed to produce a series of values over time, but we'll defer discussion of these until we survey generator topics.

<u>def Executes at Runtime</u>

The Python def is a true executable statement: when it runs, it creates a new function object and assigns it to a name. (Remember, all we have in Python is runtime; there is no such thing as a separate compile time.) Because it's a statement, a def can appear anywhere a statement can—even nested in other statements. For instance, although defs normally are run when the module enclosing them is imported, it's also completely legal to nest a function def inside an if statement to select between alternative definitions:

```
if test:
    def func(): # Define func this way
        ...
```

```
else:
    def func(): # Or else this way
        ...
...
func() # Call the version selected and built
```

One way to understand this code is to realize that the def is much like an = statement: it simply assigns a name at runtime. Unlike in compiled languages such as C, Python functions do not need to be fully defined before the program runs. More generally, defs are not evaluated until they are reached and run, and the code inside defs is not evaluated until the functions are later called.

Because function definition happens at runtime, there's nothing special about the function name. What's important is the object to which it refers:

```
othername = func # Assign function object
othername() # Call func again
```

Here, the function was assigned to a different name and called through the new name.

Like everything else in Python, functions are just objects; they are recorded explicitly in memory at program execution time. In fact, besides calls, functions allow arbitrary attributes to be attached to record information for later use:

```
def func(): ... # Create function object
func() # Call object
func.attr = value # Attach attributes
```

Polymorphism in Python

As we just saw, the very meaning of the expression x * y in our simple times function depends completely upon the kinds of objects that x and y are—thus, the same function can perform multiplication in one instance and repetition in another. Python leaves it up to the objects to do something reasonable for the syntax. Really, * is just a dispatch mechanism that routes control to the objects being processed.

This sort of type-dependent behavior is known as polymorphism, a term essentially means that the meaning of an operation depends on the objects being operated upon. Because it's a dynamically typed language, polymorphism runs rampant in Python. In fact, every operation is a polymorphic operation in Python: printing, indexing, the * operator, and much more.

This is deliberate, and it accounts for much of the language's conciseness and flexibility.

A single function, for instance, can generally be applied to a whole category of object types automatically. As long as those objects support the expected interface (a.k.a. protocol), the function can process them. That is, if the objects passed into a function have the expected methods and expression operators, they are plug-and-play compatible with the function's logic.

Even in our simple times function, this means that any two objects that support a * will work, no matter what they may be, and no matter when they are coded. This function will work on two numbers (performing multiplication), or a string and a number (performing repetition), or any other combination of objects supporting the expected interface—even class-based objects we have not even imagined yet.

Moreover, if the objects passed in do not support this expected interface, Python will detect the error when the * expression is run and raise an exception automatically. It's therefore usually pointless to code error checking ourselves. In fact, doing so would limit our function's utility, as it would be restricted to work only on objects whose types we test for.

This turns out to be a crucial philosophical difference between Python and statically typed languages like C++ and Java: in Python, your code is not supposed to care about specific data types. If it does, it will be limited to working on just the types you anticipated when you wrote it, and it will not support other compatible object types that may be

coded in the future. Although it is possible to test for types with tools like the type built-in function, doing so breaks your code's flexibility. By and large, we code to object interfaces in Python, not data types.

Of course, some programs have unique requirements, and this polymorphic model of programming means we have to test our code to detect errors, rather than providing type declarations a compiler can use to detect some types of errors for us ahead of time.

In exchange for an initial bit of testing, though, we radically reduce the amount of code we have to write and radically increase our code's flexibility. As you'll learn, it's a net win in practice.

Local Variables

Probably the most interesting part of this example, though, is its names. It turns out that the variable res inside intersect is what in Python is called a local variable—a name that is visible only to code inside the function def and that exists only while the function runs. In fact, because all names assigned in any way inside a function are classified as local variables by default, nearly all the names in intersect are local variables:

- res is obviously assigned, so it is a local variable.
- Arguments are passed by assignment, so seq1 and seq2 are, too.
- The for loop assigns items to a variable, so the name x is also local.

All these local variables appear when the function is called and disappear when the function exits—the return statement at the end of intersect sends back the result object, but the name res goes away. Because of this, a function's variables won't remember values between calls; although the object returned by a function lives on, retaining other sorts of state information requires other sorts of techniques.

Module 13 - Scopes & Arguments

As we saw, Python's core function model is simple to use, but even simple function examples quickly led us to questions about the meaning of variables in our code. This module moves on to present the details behind Python's scopes—the places where variables are defined and looked up. Like module files, scopes help prevent name clashes across your program's code: names defined in one program unit don't interfere with names in another.

As we'll see, the place where a name is assigned in our code is crucial to determining what the name means. We'll also find that scope usage can have a major impact on program maintenance effort; overuse of globals, for example, is a generally bad thing.

On the plus side, we'll learn that scopes can provide a way to retain state information between function calls, and offer an alternative to classes in some roles.

Scope rules

Now that you're ready to start writing your own functions, we need to get more formal about what names mean in Python. When you use a name in a program, Python creates, changes, or looks up the name in what is known as a namespace—a place where names live. When we talk about the search for a name's value in relation to code, the term scope refers to a namespace: that is, the location of a name's assignment in your source code determines the scope of the name's visibility to your code.

Just about everything related to names, including scope classification, happens at assignment time in Python. As we've seen, names in Python spring into existence when they are first assigned values, and they must be assigned before they are used. Because names are not declared ahead of time, Python uses the location of the assignment of a name to associate it with (i.e., bind it to) a particular namespace. In other words, the place where you assign a name in your source code determines the namespace it will live in, and hence its scope of visibility.

Besides packaging code for reuse, functions add an extra namespace layer to your programs to minimize the potential for collisions among variables of the same name—by default, all names assigned inside a function are associated with that function's namespace, and no other. This rule means that:

- Names assigned inside a def can only be seen by the code within that def. You cannot even refer to such names from outside the function.
- Names assigned inside a def do not clash with variables outside the def, even if the same names are used elsewhere. A name X assigned outside a given def (i.e., in a different def or at the top level of a module file) is a completely different variable from a name X assigned inside that def.

In all cases, the scope of a variable (where it can be used) is always determined by where it is assigned in your source code and has nothing to do with which functions call which. In fact, as we'll learn in this module, variables may be assigned in three different places, corresponding to three different scopes:

- If a variable is assigned inside a def, it is local to that function.
- If a variable is assigned in an enclosing def, it is nonlocal to nested functions.
- If a variable is assigned outside all defs, it is global to the entire file.

We call this lexical scoping because variable scopes are determined entirely by the locations of the variables in the source code of your program files, not by function calls.

For example, in the following module file, the X = 99 assignment creates a global variable named X (visible everywhere in this file), but the X = 88 assignment creates a local variable X (visible only within the def statement):

```
X = 99                    # Global (module) scope X
def func():
    X = 88                # Local (function) scope X: a different variable
```

Even though both variables are named X, their scopes make them different. The net effect is that function scopes help to avoid name clashes in your programs and help to make functions more self-contained program units—their code need not be concerned with names used elsewhere.

Scope Details

Before we started writing functions, all the code we wrote was at the top level of a module (i.e., not nested in a def), so the names we used either lived in the module itself or were built-ins predefined by Python (e.g., open). Technically, the interactive prompt is a module named __main__ that prints results and doesn't save its code; in all other ways, though, it's like the top level of a module file.

Functions, though, provide nested namespaces (scopes) that localize the names they use, such that names inside a function won't clash with those outside it (in a module or another function). Functions define a local scope and modules define a global scope with the following properties:

- **The enclosing module is a global scope.** Each module is a global scope—that is, a namespace in which variables created (assigned) at the top level of the module file live. Global variables become attributes of a module object to the outside world after imports but can also be used as simple variables within the module file itself.
- **The global scope spans a single file only.** Don't be fooled by the word "global" here—names at the top level of a file are global to code within that single file only.
  There is really no notion of a single, all-encompassing global file-based scope in Python. Instead, names are partitioned into modules, and you must always import a module explicitly if you want to be able to use the names its file defines. When you hear "global" in Python, think "module."
- **Assigned names are local unless declared global or nonlocal.** By default, all the names assigned inside a function definition are put in the local scope (the namespace associated with the function call). If you need to assign a name that lives at the top level of the module enclosing the function, you can do so by declaring it in a global statement inside the function. If you need to assign a name that lives in an enclosing def, as of Python 3.X you can do so by declaring it in a nonlocal statement.
- **All other names are enclosing function locals, globals, or built-ins.** Names not assigned a value in the function definition are assumed to be enclosing scope locals, defined in a physically surrounding def statement; globals that live in the enclosing module's namespace; or built-ins in the predefined built-ins module Python provides.
- **Each call to a function creates a new local scope.** Every time you call a function, you create a new local scope—that is, a namespace in which the names created inside that function will usually live. You can think of each def statement (and lambda expression) as defining a new local scope, but the local scope actually corresponds to a function call. Because Python allows functions to call themselves to loop—an advanced technique known as recursion and noted briefly when we explored comparisons—each active call receives its own copy of the function's local variables. Recursion is useful in functions we write as well, to process structures whose shapes can't be predicted ahead of time.

There are a few subtleties worth underscoring here. First, keep in mind that code typed at the interactive command prompt lives in a module, too, and follows the normal scope rules: they are global variables, accessible to the entire interactive session. You'll learn more about modules in the next part of this course.

Also note that any type of assignment within a function classifies a name as local. This includes = statements, module names in import, function names in def, function argument names, and so on. If you assign a name in any way within a def, it will become a local to that function by default.

Conversely, in-place changes to objects do not classify names as locals; only actual name assignments do. For instance, if the name L is assigned to a list at the top level of a module, a statement L = X within a function will classify L as a local, but L.append(X) will not. In the latter case, we are changing the list object that L references, not L itself —L is found in the global scope as usual, and Python happily modifies it without requiring a global (or nonlocal) declaration. As usual, it helps to keep the distinction between names and objects clear: changing an object is not an assignment to a name.

The Built-in Scope

We’ve been talking about the built-in scope in the abstract, but it’s a bit simpler than you may think. Really, the built-in scope is just a built-in module called builtins, but you have to import builtins to query built-ins because the name builtins is not itself built in...

No, I’m serious! The built-in scope is implemented as a standard library module named builtins in 3.X, but that name itself is not placed in the built-in scope, so you have to import it in order to inspect it. Once you do, you can run a dir call to see which names are predefined. In Python 3.3 (see ahead for 2.X usage):

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
...many more names omitted...
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed',
'round', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum',
'super', 'tuple', 'type', 'vars', 'zip']
```

The names in this list constitute the built-in scope in Python; roughly the first half are built-in exceptions, and the second half are built-in functions. Also in this list are the special names None, True, and False, though they are treated as reserved words in 3.X.

Because Python automatically searches this module last in its LEGB lookup, you get all the names in this list “for free”—that is, you can use them without importing any modules. Thus, there are really two ways to refer to a built-in function—by taking advantage of the LEGB rule, or by manually importing the builtins module:

```
>>> zip # The normal way
<class 'zip'>
>>> import builtins # The hard way: for customizations
>>> builtins.zip
<class 'zip'>
>>> zip is builtins.zip # Same object, different lookups
True
```

The global statement

The global statement and its nonlocal 3.X cousin are the only things that are remotely like declaration statements in Python. They are not type or size declarations, though; they are namespace declarations. The global statement tells Python that a function plans to change one or more global names—that is, names that live in the enclosing module’s scope (namespace).

We’ve talked about global in passing already. Here’s a summary:

- Global names are variables assigned at the top level of the enclosing module file.
- Global names must be declared only if they are assigned within a function.
- Global names may be referenced within a function without being declared.

In other words, global allows us to change names that live outside a def at the top level of a module file. As we’ll see later, the nonlocal statement is almost identical but applies to names in the enclosing def’s local scope, rather than names in the enclosing module.

The global statement consists of the keyword global, followed by one or more names separated by commas. All the listed names will be mapped to the enclosing module’s scope when assigned or referenced within the function body. For instance:

```
X = 88          # Global X
def func():
    global X
    X = 99      # Global X: outside def
func()
```

```
print(X)            # Prints 99
```

We've added a global declaration to the example here, such that the X inside the def now refers to the X outside the def; they are the same variable this time, so changing X inside the function changes the X outside it. Here is a slightly more involved example of global at work:

```
y, z = 1, 2         # Global variables in module
def all_global():
    global x        # Declare globals assigned
    x = y + z       # No need to declare y, z: LEGB rule
```

Here, x, y, and z are all globals inside the function all_global. y and z are global because they aren't assigned in the function; x is global because it was listed in a global statement to map it to the module's scope explicitly. Without the global here, x would be considered local by virtue of the assignment.

Notice that y and z are not declared global; Python's LEGB lookup rule finds them in the module automatically. Also, notice that x does not even exist in the enclosing module before the function runs; in this case, the first assignment in the function creates x in the module.

Other Ways to Access Globals

Interestingly, because global-scope variables morph into the attributes of a loaded module object, we can emulate the global statement by importing the enclosing module and assigning to its attributes, as in the following example module file. Code in this file imports the enclosing module, first by name, and then by indexing the sys.modules loaded modules table:

```
# thismod.py
var = 99                              # Global variable == module attribute
def local():
    var = 0                           # Change local var
def glob1():
    global var                        # Declare global (normal)
    var += 1                          # Change global var
def glob2():
    var = 0                           # Change local var
    import thismod                    # Import myself
    thismod.var += 1                  # Change global var
def glob3():
    var = 0                           # Change local var
    import sys                        # Import system table
    glob = sys.modules['thismod']     # Get module object (or use __name__)
    glob.var += 1                     # Change global var
def test():
    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)
```

When run, this adds 3 to the global variable (only the first function does not impact it):

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

This works, and it illustrates the equivalence of globals to module attributes, but it's much more work than using the global statement to make your intentions explicit.

As we've seen, global allows us to change names in a module outside a function. It has a close relative named nonlocal that can be used to change names in enclosing functions, too—but to understand how that can be useful, we first need to explore enclosing functions in general.

Scopes and nested functions

So far, I've omitted one part of Python's scope rules on purpose, because it's relatively uncommon to encounter it in practice. However, it's time to take a deeper look at the letter E in the LEGB lookup rule. The E layer was added in Python 2.2; it takes the form of the local scopes of any and all enclosing function's local scopes. Enclosing scopes are sometimes also called statically nested scopes. Really, the nesting is a lexical one—nested scopes correspond to physically and syntactically nested code structures in your program's source code text.

Nested Scope Details

With the addition of nested function scopes, variable lookup rules become slightly more complex. Within a function:

- **A reference** (X) looks for the name X first in the current local scope (function); then in the local scopes of any lexically enclosing functions in your source code, from inner to outer; then in the current global scope (the module file); and finally in the built-in scope (the module builtins). global declarations make the search begin in the global (module file) scope instead.
- **An assignment** (X = value) creates or changes the name X in the current local scope, by default. If X is declared global within the function, the assignment creates or changes the name X in the enclosing module's scope instead. If, on the other hand, X is declared nonlocal within the function in 3.X (only), the assignment changes the name X in the closest enclosing function's local scope.

Notice that the global declaration still maps variables to the enclosing module. When nested functions are present, variables in enclosing functions may be referenced, but they require 3.X nonlocal declarations to be changed.

Retaining Enclosing Scope State with Defaults

In early versions of Python (prior to 2.2), the sort of code in the prior section failed because nested defs did not do anything about scopes—a reference to a variable within f2 in the following would search only the local (f2), then global (the code outside f1), and then built-in scopes. Because it skipped the scopes of enclosing functions, an error would result. To work around this, programmers typically used default argument values to pass in and remember the objects in an enclosing scope:

```
def f1():
   x = 88
   def f2(x=x):   # Remember enclosing scope X with defaults
      print(x)
   f2()
f1()              # Prints 88
```

This coding style works in all Python releases, and you'll still see this pattern in some existing Python code. In fact, it's still required for loop variables, as we'll see in a moment, which is why it remains worth studying today. In short, the syntax arg=val in a def header means that the argument arg will default to the value val if no real value is passed to arg in a call. This syntax is used here to explicitly assign enclosing scope state to be retained.

Specifically, in the modified f2 here, the x=x means that the argument x will default to the value of x in the enclosing scope—because the second x is evaluated before Python steps into the nested def, it still refers to the x in f1. In effect, the default argument remembers what x was in f1: the object 88.

That's fairly complex, and it depends entirely on the timing of default value evaluations. In fact, the nested scope lookup rule was added to Python to make defaults unnecessary for this role—today, Python automatically remembers any values required in the enclosing scope for use in nested defs.

Of course, the best prescription for much code is simply to avoid nesting defs within defs, as it will make your programs much simpler—in the Pythonic view, flat is generally better than nested. The following is an equivalent of the prior example that avoids nesting altogether. Notice the forward reference in this code—it's OK to call a function

defined after the function that calls it, as long as the second def runs before the first function is actually called. Code inside a def is never evaluated until the function is actually called:

```
>>> def f1():
        x = 88    # Pass x along instead of nesting
        f2(x)     # Forward reference OK
>>> def f2(x):
        print(x)  # Flat is still often better than nested!
>>> f1()
88
```

If you avoid nesting this way, you can almost forget about the nested scopes concept in Python. On the other hand, the nested functions of closure (factory) functions are fairly common in modern Python code, as are lambda functions—which almost naturally appear nested in defs and often rely on the nested scopes layer, as the next section explains.

Nested scopes, defaults, and lambdas

Although they see increasing use in defs these days, you may be more likely to care about nested function scopes when you start coding or reading lambda expressions.

We've met lambda briefly, but in short, it's an expression that generates a new function to be called later, much like a def statement.

Because it's an expression, though, it can be used in places that def cannot, such as within list and dictionary literals. Like a def, a lambda expression also introduces a new local scope for the function it creates. Thanks to the enclosing scopes lookup layer, lambdas can see all the variables that live in the functions in which they are coded. Thus, the following code—a variation on the factory we saw earlier—works, but only because the nested scope rules are applied:

```
def func():
    x = 4
    action = (lambda n: x ** n)    # x remembered from enclosing def
    return action
x = func()
print(x(2))                        # Prints 16, 4 ** 2
```

Prior to the introduction of nested function scopes, programmers used defaults to pass values from an enclosing scope into lambdas, just as for defs. For instance, the following works on all Pythons:

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n) # Pass x in manually
    return action
```

Because lambdas are expressions, they naturally (and even normally) nest inside enclosing defs. Hence, they were perhaps the biggest initial beneficiaries of the addition of enclosing function scopes in the lookup rules; in most cases, it is no longer necessary to pass values into lambdas with defaults.

Loop variables may require defaults, not scopes

There is one notable exception to the rule I just gave (and a reason why I've shown you the otherwise dated default argument technique we just saw): if a lambda or def defined within a function is nested inside a loop, and the nested function references an enclosing scope variable that is changed by that loop, all functions generated within the loop will have the same value—the value the referenced variable had in the last loop iteration.

In such cases, you must still use defaults to save the variable's current value instead. This may seem a fairly obscure case, but it can come up in practice more often than you may think, especially in code that generates callback handler functions for a number of widgets in a GUI—for instance, handlers for button-clicks for all the buttons in a row. If these are created in a loop, you may need to be careful to save state with defaults, or all your buttons' callbacks may wind up doing the same thing.

Here's an illustration of this phenomenon reduced to simple code: the following attempts to build up a list of functions that each remember the current variable i from the enclosing scope:

```
>>> def makeActions():
        acts = []
        for i in range(5): # Tries to remember each i
            acts.append(lambda x: i ** x) # But all remember same last i!
        return acts
>>> acts = makeActions()
>>> acts[0]
<function makeActions.<locals>.<lambda> at 0x0000000002A4A400>
```

This doesn't quite work, though—because the enclosing scope variable is looked up when the nested functions are later called, they all effectively remember the same value: the value the loop variable had on the last loop iteration. That is, when we pass a power argument of 2 in each of the following calls, we get back 4 to the power of 2 for each function in the list, because i is the same in all of them—4:

```
>>> acts[0](2) # All are 4 ** 2, 4=value of last i
16
>>> acts[1](2) # This should be 1 ** 2 (1)
16
>>> acts[2](2) # This should be 2 ** 2 (4)
16
>>> acts[4](2) # Only this should be 4 ** 2 (16)
16
```

This is the one case where we still have to explicitly retain enclosing scope values with default arguments, rather than enclosing scope references. That is, to make this sort of code work, we must pass in the current value of the enclosing scope's variable with a default. Because defaults are evaluated when the nested function is created (not when it's later called), each remembers its own value for i:

```
>>> def makeActions():
        acts = []
        for i in range(5): # Use defaults instead
            acts.append(lambda x, i=i: i ** x) # Remember current i
        return acts
>>> acts = makeActions()
>>> acts[0](2) # 0 ** 2
0
>>> acts[1](2) # 1 ** 2
1
>>> acts[2](2) # 2 ** 2
4
>>> acts[4](2) # 4 ** 2
16
```

This seems an implementation artifact that is prone to change, and may become more important as you start writing larger programs.

Passing arguments

As we learned, the place where a name is defined in our code determines much of its meaning. This module continues the function story by studying the concepts in Python argument passing—the way that objects are sent to functions as inputs. As we'll see, arguments (a.k.a. parameters) are assigned to names in a function, but they have more to do with object references than with variable scopes.

We'll also find that Python provides extra tools, such as keywords, defaults, and arbitrary argument collectors and extractors that allow for wide flexibility in the way arguments are sent to a function, and we'll put them to work in examples.

Argument-Passing Basics

Earlier in this part of the course, I noted that arguments are passed by assignment. This has a few ramifications that aren't always obvious to newcomers, which I'll expand on in this section. Here is a rundown of the key points in passing arguments to functions:

- **Arguments are passed by automatically assigning objects to local variable names.** Function arguments—references to (possibly) shared objects sent by the caller—are just another instance of Python assignment at work. Because references are implemented as pointers, all arguments are, in effect, passed by pointer. Objects passed as arguments are never automatically copied.
- **Assigning to argument names inside a function does not affect the caller.** Argument names in the function header become new, local names when the function runs, in the scope of the function. There is no aliasing between function argument names and variable names in the scope of the caller.
- **Changing a mutable object argument in a function may impact the caller.** On the other hand, as arguments are simply assigned to passed-in objects, functions can change passed-in mutable objects in place, and the results may affect the caller. Mutable arguments can be input and output for functions.

Python's pass-by-assignment scheme isn't quite the same as C++'s reference parameters option, but it turns out to be very similar to the argument-passing model of the C language (and others) in practice:

- **Immutable arguments are effectively passed "by value."** Objects such as integersand strings are passed by object reference instead of by copying, but because you can't change immutable objects in place anyhow, the effect is much like making a copy.
- **Mutable arguments are effectively passed "by pointer."** Objects such as lists and dictionaries are also passed by object reference, which is similar to the way C passes arrays as pointers—mutable objects can be changed in place in the function, much like C arrays.

Of course, if you've never used C, Python's argument-passing mode will seem simpler still—it involves just the assignment of objects to names, and it works the same whether the objects are mutable or not.

Arguments and Shared References

To illustrate argument-passing properties at work, consider the following code:

```
>>> def f(a):    # a is assigned to (references) the passed object
        a = 99   # Changes local variable a only
>>> b = 88
>>> f(b)         # a and b both reference same 88 initially
>>> print(b)     # b is not changed
88
```

In this example the variable a is assigned the object 88 at the moment the function is called with f(b), but a lives only within the called function. Changing a inside the function has no effect on the place where the function is called; it simply resets the local variable a to a completely different object.

That's what is meant by a lack of name aliasing—assignment to an argument name inside a function (e.g., a=99) does not magically change a variable like b in the scope of the function call. Argument names may share passed objects initially (they are essentially pointers to those objects), but only temporarily, when the function is first called. As soon as an argument name is reassigned, this relationship ends.

At least, that's the case for assignment to argument names themselves. When arguments are passed mutable objects like lists and dictionaries, we also need to be aware that inplace changes to such objects may live on after a function exits, and hence impact callers.

Here's an example that demonstrates this behavior:

```
>>> def changer(a, b):   # Arguments assigned references to objects
        a = 2            # Changes local name's value only
        b[0] = 'spam'    # Changes shared object in place
>>> X = 1
>>> L = [1, 2]           # Caller:
>>> changer(X, L)        # Pass immutable and mutable objects
>>> X, L                 # X is unchanged, L is different!
(1, ['spam', 2])
```

In this code, the changer function assigns values to argument a itself, and to a component of the object referenced by argument b. These two assignments within the function are only slightly different in syntax but have radically different results:

- Because a is a local variable name in the function's scope, the first assignment has no effect on the caller—it simply changes the local variable a to reference a completely different object, and does not change the binding of the name X in the caller's scope. This is the same as in the prior example.
- Argument b is a local variable name, too, but it is passed a mutable object (the list that L references in the caller's scope). As the second assignment is an in-place object change, the result of the assignment to b[0] in the function impacts the value of L after the function returns.

Really, the second assignment statement in changer doesn't change b—it changes part of the object that b currently references. This in-place change impacts the caller only because the changed object outlives the function call. The name L hasn't changed either —it still references the same, changed object—but it seems as though L differs after the call because the value it references has been modified within the function. In effect, the list name L serves as both input to and output from the function.

Figure below illustrates the name/object bindings that exist immediately after the function has been called, and before its code has run.

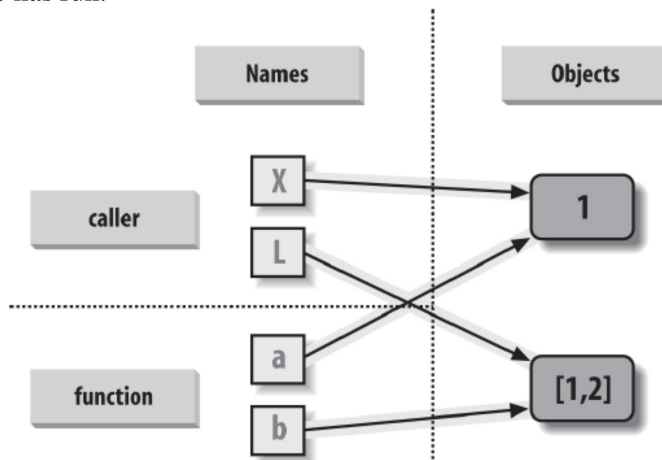


Figure: *References: arguments. Because arguments are passed by assignment, argument names in the function may share objects with variables in the scope of the call. Hence, in-place changes to mutable arguments in a function can impact the caller. Here, a and b in the function initially reference the objects referenced by variables X and L when the function is first called. Changing the list throughvariable b makes L appear different after the call returns.*

If this example is still confusing, it may help to notice that the effect of the automatic assignments of the passed-in arguments is the same as running a series of simple assignment statements. In terms of the first argument, the assignment has no effect on the caller:

```
>>> X = 1
>>> a = X # They share the same object
>>> a = 2 # Resets 'a' only, 'X' is still 1
>>> print(X)
1
```

The assignment through the second argument does affect a variable at the call, though, because it is an in-place object change:

```
>>> L = [1, 2]
>>> b = L # They share the same object
>>> b[0] = 'spam' # In-place change: 'L' sees the change too
>>> print(L)
['spam', 2]
```

If you recall our discussions about shared mutable objects, you'll recognize the phenomenon at work: changing a mutable object in place can impact other references to that object. Here, the effect is to make one of the arguments work like both an input and an output of the function.

Avoiding Mutable Argument Changes

This behavior of in-place changes to mutable arguments isn't a bug—it's simply the way argument passing works in Python, and turns out to be widely useful in practice.

Arguments are normally passed to functions by reference because that is what we normally want. It means we can pass large objects around our programs without making multiple copies along the way, and we can easily update these objects as we go. In fact, Python's class model depends upon changing a passed-in "self" argument in place, to update object state.

If we don't want in-place changes within functions to impact objects we pass to them, though, we can simply make explicit copies of mutable objects, as we learned. For function arguments, we can always copy the list at the point of call, with tools like list, list.copy as of 3.3, or an empty slice:

```
L = [1, 2]
changer(X, L[:]) # Pass a copy, so our 'L' does not change
```

We can also copy within the function itself, if we never want to change passed-in objects, regardless of how the function is called:

```
def changer(a, b):
    b = b[:] # Copy input list so we don't impact caller
    a = 2
    b[0] = 'spam' # Changes our list copy only
```

Both of these copying schemes don't stop the function from changing the object—they just prevent those changes from impacting the caller. To really prevent changes, we can always convert to immutable objects to force the issue. Tuples, for example, raise an exception when changes are attempted:

```
L = [1, 2]
changer(X, tuple(L)) # Pass a tuple, so changes are errors
```

This scheme uses the built-in tuple function, which builds a new tuple out of all the items in a sequence (really, any iterable). It's also something of an extreme—because it forces the function to be written to never change passed-in arguments, this solution might impose more limitations on the function than it should, and so should generally be avoided (you never know when changing arguments might come in handy for other calls in the future). Using this technique will also make the function lose the ability to call any list-specific methods on the argument, including methods that do not change the object in place.

The main point to remember here is that functions might update mutable objects like lists and dictionaries passed into them. This isn't necessarily a problem if it's expected, and often serves useful purposes. Moreover, functions that change passed-in mutable objects in place are probably designed and intended to do so—the change is likely part of a well-defined API that you shouldn't violate by making copies.

However, you do have to be aware of this property—if objects change out from under you unexpectedly, check whether a called function might be responsible, and make copies when objects are passed if needed.

Simulating Output Parameters and Multiple Results

We've already discussed the return statement and used it in a few examples. Here's another way to use this statement: because return can send back any sort of object, it can return multiple values by packaging them in a tuple or other collection type. In fact, although Python doesn't support what some languages label "call by reference" argument passing, we can usually simulate it by returning tuples and assigning the results back to the original argument names in the caller:

```
>>> def multiple(x, y):
        x = 2 # Changes local names only
        y = [3, 4]
        return x, y # Return multiple new values in a tuple
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L) # Assign results to caller's names
>>> X, L
(2, [3, 4])
```

It looks like the code is returning two values here, but it's really just one—a two-item tuple with the optional surrounding parentheses omitted. After the call returns, we can use tuple assignment to unpack the parts of the returned tuple. (If you've forgotten why this works, flip back to "Tuples" and "Assignment Statements". The net effect of this coding pattern is to both send back multiple results and simulate the output parameters of other languages by explicit assignments.

Here, X and L change after the call, but only because the code said so.

Special argument matching modes

As we've just seen, arguments are always passed by assignment in Python; names in the def header are assigned to passed-in objects. On top of this model, though, Python provides additional tools that alter the way the argument objects in a call are matched with argument names in the header prior to assignment. These tools are all optional, but they allow us to write functions that support more flexible calling patterns, and you may encounter some libraries that require them. By default, arguments are matched by position, from left to right, and you must pass exactly as many arguments as there are argument names in the function header. However, you can also specify matching by name, provide default values, and use collectors for extra arguments.

Module 14 - The Big Picture

This module begins our in-depth look at the Python module—the highest-level program organization unit, which packages program code and data for reuse, and provides selfcontained namespaces that minimize variable name clashes across your programs. In concrete terms, modules typically correspond to Python program files. Each file is a module, and modules import other modules to use the names they define. Modules might also correspond to extensions coded in external languages such as C, Java, or C#, and even to directories in package imports. Modules are processed with two statements and one important function:

import
Lets a client (importer) fetch a module as a whole
from
Allows clients to fetch particular names from a module
imp.reload (reload in 2.X)
Provides a way to reload a module's code without stopping Python

The goal here is to expand on the core module concepts you're already familiar with, and move on to explore more advanced module usage. First we reviews module basics, and offers a general look at the role of modules in overall program structure.

later, we'll dig into the coding details behind the theory. Along the way, we'll flesh out module details omitted so far—you'll learn about reloads, the __name__ and __all__ attributes, package imports, relative import syntax, 3.3 namespace packages, and so on. Because modules and classes are really just glorified namespaces,
we'll formalize namespace concepts here as well.

In short, modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as namespaces.

Why use modules?
the top level of a module file become attributes of the imported module object. As we saw in the last part of this course, imports give access to names in a module's global scope. That is, the module file's global scope morphs into the module object's attribute namespace when it is imported. Ultimately, Python's modules allow us to link individual files into a larger program system.

More specifically, modules have at least three roles:

- **Code reuse -** Modules let you save code in files permanently. Unlike code you type at the Python interactive prompt, which goes away when you exit Python, code in module files is persistent—it can be reloaded and rerun as many times as needed. Just as importantly, modules are a place to define names, known as attributes, which may be referenced by multiple external clients. When used well, this supports a modular program design that groups functionality into reusable units.
- **System namespace partitioning -** Modules are also the highest-level program organization unit in Python. Although they are fundamentally just packages of names, these packages are also self-contained—you can never see a name in another file, unless you explicitly import that file. Much like the local scopes of functions, this helps avoid name clashes across your programs. In fact, you can't avoid this feature—everything "lives" in a module, both the code you run and the objects you create are always implicitly enclosed in modules. Because of that, modules are natural tools for grouping system components.
- **Implementing shared services or data -** From an operational perspective, modules are also useful for implementing components that are shared across a system and hence require only a single copy. For instance, if you need to provide a global object that's used by more than one function or file, you can code it in a module that can then be imported by many clients.

At least that's the abstract story—for you to truly understand the role of modules in a Python system, we need to digress for a moment and explore the general structure of a Python program.

Python program architecture

So far in this course, I've sugarcoated some of the complexity in my descriptions of Python programs. In practice, programs usually involve more than just one file. For all but the simplest scripts, your programs will take the form of multifile systems. Even if you can get by with coding a single file yourself, you will almost certainly wind up using external files that someone else has already written.

This section introduces the general architecture of Python programs—the way you divide a program into a collection of source files (a.k.a. modules) and link the parts into a whole. As we'll see, Python fosters a modular program structure that groups functionality into coherent and reusable units, in ways that are natural, and almost automatic.

Along the way, we'll also explore the central concepts of Python modules, imports, and object attributes.

How to Structure a Program

At a base level, a Python program consists of text files containing Python statements, with one main top-level file, and zero or more supplemental files known as modules.

Here's how this works. The top-level (a.k.a. script) file contains the main flow of control of your program—this is the file you run to launch your application. The module files are libraries of tools used to collect components used by the top-level file, and possibly elsewhere. Top-level files use tools defined in module files, and modules use tools defined in other modules.

Although they are files of code too, module files generally don't do anything when run directly; rather, they define tools intended for use in other files. A file imports a module to gain access to the tools it defines, which are known as its attributes—variable names attached to objects such as functions. Ultimately, we import modules and access their attributes to use their tools.

Imports and Attributes

Let's make this a bit more concrete. Figure 22-1 sketches the structure of a Python program composed of three files: a.py, b.py, and c.py. The file a.py is chosen to be the top-level file; it will be a simple text file of statements, which is executed from top to bottom when launched. The files b.py and c.py are modules; they are simple text files of statements as well, but they are not usually launched directly. Instead, as explained previously, modules are normally imported by other files that wish to use the tools the modules define.

For instance, suppose the file b.py in Figure before this defines a function called spam, for external use. As we learned when studying functions in Part IV, b.py will contain a Python def statement to generate the function, which you can later run by passing zero or more values in parentheses after the function's name:

```
def spam(text): # File b.py print(text, 'spam')
```

Now, suppose a.py wants to use spam. To this end, it might contain Python statements such as the following:

```
import b # File a.py
b.spam('gumby') # Prints "gumby spam"
```

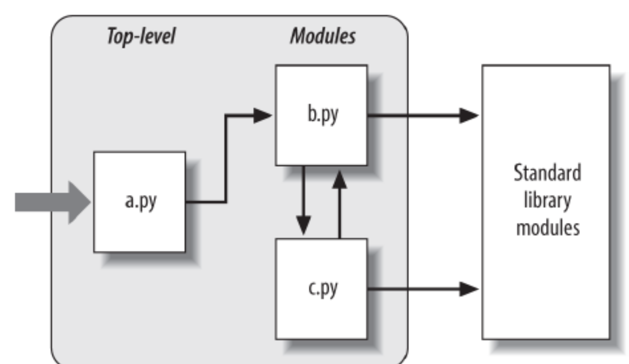


Figure: *Program architecture in Python. A program is a system of modules. It has one top-levelscript file (launched to run the program), and multiple module files (imported libraries of tools).*

Scripts and modules are both text files containing Python statements, though the statements in modulesusually just create objects to be used later. Python's standard library provides a collection of precoded modules.

The first of these, a Python import statement, gives the file a.py access to everything defined by top-level code in the file b.py. The code import b roughly means:

Load the file b.py (unless it's already loaded), and give me access to all its attributes through the name b.

To satisfy such goals, import (and, as you'll see later, from) statements execute and load other files on request. More formally, in Python, cross-file module linking is not resolved until such import statements are executed at runtime; their net effect is to assign module names—simple variables like b—to loaded module objects. In fact, the module name used in an import statement serves two purposes: it identifies the external file to be loaded, but it also becomes a variable assigned to the loaded module.

Similarly, objects defined by a module are also created at runtime, as the import is executing: import literally runs statements in the target file one at a time to create its contents. Along the way, every name assigned at the top-level of the file becomes an attribute of the module, accessible to importers. For example, the second of the statements in a.py calls the function spam defined in the module b—created by running its def statement during the import—using object attribute notation. The code b.spam means:

Fetch the value of the name spam that lives within the object b.

This happens to be a callable function in our example, so we pass a string in parentheses ('gumby'). If you actually type these files, save them, and run a.py, the words "gumby spam" will be printed.

As we've seen, the object.attribute notation appears throughout Python code—most objects have useful attributes that are fetched with the "." operator. Some reference callable objects like functions that take action (e.g., a salary computer), and others are simple data values that denote more static objects and properties (e.g., a person's name).

The notion of importing is also completely general throughout Python. Any file can import tools from any other file. For instance, the file a.py may import b.py to call its function, but b.py might also import c.py to leverage different tools defined there. Import chains can go as deep as you like: in this example, the module a can import b, which can import c, which can import b again, and so on.

Besides serving as the highest organizational structure, modules are also the highest level of code reuse in Python. Coding components in module files makes them useful in your original program, and in any other programs you may write later. For instance, if after coding the program in Figure before this, we discover that the function b.spam is a general-purpose tool, we can reuse it in a completely different program; all we have to do is import the file b.py again from the other program's files.

Standard Library Modules

Notice the rightmost portion of Figure before this. Some of the modules that your programs will import are provided by Python itself and are not files you will code.

Python automatically comes with a large collection of utility modules known as the standard library. This collection, over 200 modules large at last count, contains platform-independent support for common programming tasks: operating system interfaces, object persistence, text pattern matching, network and Internet scripting, GUI construction, and much more. None of these tools are part of the Python language itself, but you can use them by importing the appropriate modules on any standard Python installation. Because they are standard library modules, you can also be reasonably sure that they will be available and will work portably on most platforms on
which you will run Python.

This course's examples employ a few of the standard library's modules—timeit, sys, and os, for instance—but we'll really only scratch the surface of the libraries story here. For a complete look, you should browse the standard Python

library reference manual, available either online at http://www.python.org, or with your Python installation (via IDLE or Python's Start button menu on some Windows). The PyDoc tool is another way to explore standard library modules.

Because there are so many modules, this is really the only way to get a feel for what tools are available. You can also find tutorials on Python library tools in commercial books that cover application-level programming.

How imports work

The prior section talked about importing modules without really explaining what happens when you do so. Because imports are at the heart of program structure in Python, this section goes into more formal detail on the import operation to make this process less abstract.

Some C programmers like to compare the Python module import operation to a C #include, but they really shouldn't—in Python, imports are not just textual insertions of one file into another. They are really runtime operations that perform three distinct steps the first time a program imports a given file:

1. Find the module's file.
2. Compile it to byte code (if needed).
3. Run the module's code to build the objects it defines.

To better understand module imports, we'll explore these steps in turn. Bear in mind that all three of these steps are carried out only the first time a module is imported during a program's execution; later imports of the same module in a program run bypass all of these steps and simply fetch the already loaded module object in memory.

Technically, Python does this by storing loaded modules in a table named sys.modules and checking there at the start of an import operation. If the module is not present, a three-step process begins.

1. Find It

First, Python must locate the module file referenced by an import statement. Notice that the import statement in the prior section's example names the file without a .py extension and without its directory path: it just says import b, instead of something like import c:\dir1\b.py. Path and extension details are omitted on purpose; instead, Python uses a standard module search path and known file types to locate the module file corresponding to an import statement.1 Because this is the main part of the import operation that programmers must know about, we'll return to this topic in a moment.

2. Compile It (Maybe)

After finding a source code file that matches an import statement by traversing the module search path, Python next compiles it to byte code, if necessary. We discussed byte code briefly, but it's a bit richer than explained there. During an import operation Python checks both file modification times and the byte code's Python version number to decide how to proceed. The former uses file "timestamps," and the latter uses either a "magic" number embedded in the byte code or a filename, depending on the Python release being used. This step chooses an action as follows:

- **Compile** If the byte code file is older than the source file (i.e., if you've changed the source) or was created by a different Python version, Python automatically regenerates the byte code when the program is run. As discussed ahead, this model is modified somewhat in Python 3.2 and later— byte code files are segregated in a __pycache__ subdirectory and named with their Python version to avoid contention and recompiles when multiple Pythons are installed. This obviates the need to check version numbers in the byte code, but the timestamp check is still used to detect changes in the source.
- **Don't compile** If, on the other hand, Python finds a .pyc byte code file that is not older than the corresponding .py source file and was created by the same Python version, it skips the source-to-byte-code compile step.

In addition, if Python finds only a byte code file on the search path and no source, it simply loads the byte code directly; this means you can ship a program as just byte code files and avoid sending source. In other words, the compile step is bypassed if possible to speed program startup.

Notice that compilation happens when a file is being imported. Because of this, you will not usually see a .pyc byte code file for the top-level file of your program, unless it is also imported elsewhere—only imported files leave behind .pyc files on your machine.

The byte code of top-level files is used internally and discarded; byte code of imported files is saved in files to speed future imports.

Top-level files are often designed to be executed directly and not imported at all. Later, we'll see that it is possible to design a file that serves both as the top-level code of a program and as a module of tools to be imported. Such a file may be both executed and imported, and thus does generate a .pyc.

3. Run It

The final step of an import operation executes the byte code of the module. All statements in the file are run in turn, from top to bottom, and any assignments made to names during this step generate attributes of the resulting module object. This is how the tools defined by the module's code are created. For instance, def statements in a file are run at import time to create functions and assign attributes within the module to those functions. The functions can then be called later in the program by the file's importers.

Because this last import step actually runs the file's code, if any top-level code in a module file does real work, you'll see its results at import time. For example, top-level print statements in a module show output when the file is imported. Function def statements simply define objects for later use.

As you can see, import operations involve quite a bit of work—they search for files, possibly run a compiler, and run Python code. Because of this, any given module is imported only once per process by default. Future imports skip all three import steps and reuse the already loaded module in memory. If you need to import a file again after it has already been loaded (for example, to support dynamic end-user customizations), you have to force the issue with an imp.reload call.

The Module Search Path

As mentioned earlier, the part of the import procedure that most programmers will need to care about is usually the first—locating the file to be imported (the "find it" part). Because you may need to tell Python where to look to find files to import, you need to know how to tap into its search path in order to extend it.

In many cases, you can rely on the automatic nature of the module import search path and won't need to configure this path at all. If you want to be able to import userdefined files across directory boundaries, though, you will need to know how the search path works in order to customize it. Roughly, Python's module search path is composed of the concatenation of these major components, some of which are preset for you and some of which you can tailor to tell Python where to look:

1. The home directory of the program
2. PYTHONPATH directories (if set)
3. Standard library directories
4. The contents of any .pth files (if present)
5. The site-packages home of third-party extensions

Ultimately, the concatenation of these four components becomes sys.path, a mutable list of directory name strings that I'll expand upon later in this section. The first and third elements of the search path are defined automatically. Because Python searches the concatenation of these components from first to last, though, the second and fourth elements can be used to extend the path to include your own source code directories.

Here is how Python uses each of these path components:

- **Home directory (automatic) -** Python first looks for the imported file in the home directory. The meaning of this entry depends on how you are running the code. When you're running a program, this entry is the directory containing your program's top-level script file. When you're working interactively, this entry is the directory in which you are working (i.e., the current working directory). Because this directory is always searched first, if a program is located entirely in a single directory, all of its imports will work automatically with no path configuration required. On the other hand, because this directory is searched first, its files will also override modules of the same name in directories elsewhere on the path; be careful not to accidentally hide library modules this way if you need them in your program, or use package tools we'll meet later that can partially sidestep this issue.
- **PYTHONPATH directories (configurable) -** Next, Python searches all directories listed in your PYTHONPATH environment variable setting, from left to right (assuming you have set this at all: it's not preset for you). In brief, PYTHONPATH is simply a list of user-defined and platform-specific names of directories that contain Python code files. You can add all the directories from which you wish to be able to import, and Python will extend the module search path to include all the directories your PYTHONPATH lists. Because Python searches the home directory first, this setting is only important when importing files across directory boundaries—that is, if you need to import a file that is stored in a different directory from the file that imports it. You'll probably want to set your PYTHONPATH variable once you start writing substantial programs, but when you're first starting out, as long as you save all your module files in the directory in which you're working (i.e., the home directory, like the C:\code used in this course) your imports will work without you needing to worry about this setting at all.
- **Standard library directories (automatic) -** Next, Python automatically searches the directories where the standard library modules are installed on your machine. Because these are always searched, they normally do not need to be added to your PYTHONPATH or included in path files (discussed next). pth path file directories (configurable)

Next, a lesser-used feature of Python allows users to add directories to the module search path by simply listing them, one per line, in a text file whose name ends with a .pth suffix (for "path"). These path configuration files are a somewhat advanced installation-related feature; we won't cover them fully here, but they provide an alternative to PYTHONPATH settings.

In short, text files of directory names dropped in an appropriate directory can serve roughly the same role as the PYTHONPATH environment variable setting. For instance, if you're running Windows and Python 3.3, a file named myconfig.pth may be placed at the top level of the Python install directory (C:\Python33) or in the sitepackages subdirectory of the standard library there (C:\Python33\Lib\site-packages) to extend the module search path. On Unix-like systems, this file might be located in usr/local/lib/python3.3/site-packages or /usr/local/lib/site-python instead.

When such a file is present, Python will add the directories listed on each line of the file, from first to last, near the end of the module search path list—currently, after PYTHONPATH and standard libraries, but before the site-packages directory where third-party extensions are often installed. In fact, Python will collect the directory names in all the .pth path files it finds and will filter out any duplicates and nonexistent directories. Because they are files rather than shell settings, path files can apply to all users of an installation, instead of just one user or shell.

Moreover, for some users and applications, text files may be simpler to code than environment settings.
This feature is more sophisticated than I've described here. For more details, consult the Python library manual, and especially its documentation for the standard library module site—this module allows the locations of Python libraries and path files to be configured, and its documentation describes the expected locations of path files in general. I recommend that beginners use PYTHONPATH or perhaps a single .pth file, and then only if you must import across directories. Path files are used more often by third-party libraries, which commonly install a path file in Python's site-packages, described next.

The Lib\site-packages directory of third-party extensions (automatic). Finally, Python automatically adds the site-packages subdirectory of its standard library to the module search path. By convention, this is the place that most third party extensions are installed, often automatically by the distutils utility described in an upcoming sidebar.

Because their install directory is always part of the module search path, clients can import the modules of such extensions without any path settings.

Configuring the Search Path

The net effect of all of this is that both the PYTHONPATH and path file components of the search path allow you to tailor the places where imports look for files. The way you set environment variables and where you store path files varies per platform. For instance, on Windows, you might use your Control Panel's System icon to set PYTHONPATH to a list of directories separated by semicolons, like this:

```
c:\pycode\utilities;d:\pycode\package1
```

Or you might instead create a text file called C:\Python33\pydirs.pth, which looks like this:

```
c:\pycode\utilities
d:\pycode\package1
```

These settings are analogous on other platforms, but the details can vary too widely for us to cover in this module. See Appendix A for pointers on extending your module search path with PYTHONPATH or .pth files on various platforms.

Search Path Variations

This description of the module search path is accurate, but generic; the exact configuration of the search path is prone to changing across platforms, Python releases, and even Python implementations. Depending on your platform, additional directories may automatically be added to the module search path as well.

For instance, some Pythons may add an entry for the current working directory—the directory from which you launched your program—in the search path before the PYTHONPATH directories. When you're launching from a command line, the current working directory may not be the same as the home directory of your top-level file (i.e., the directory where your program file resides), which is always added. Because the current working directory can vary each time your program runs, you normally shouldn't depend on its value for import purposes.

To see how your Python configures the module search path on your platform, you can always inspect sys.path—the topic of the next section.

The sys.path List

If you want to see how the module search path is truly configured on your machine, you can always inspect the path as Python knows it by printing the built-in sys.path list (that is, the path attribute of the standard library module sys). This list of directory name strings is the actual search path within Python; on imports, Python searches each directory in this list from left to right, and uses the first file match it finds.

Really, sys.path is the module search path. Python configures it at program startup, automatically merging the home directory of the top-level file (or an empty string to designate the current working directory), any PYTHONPATH directories, the contents of any .pth file paths you've created, and all the standard library directories. The result is a list of directory name strings that Python searches on each import of a new file.

Python exposes this list for two good reasons. First, it provides a way to verify the search path settings you've made—if you don't see your settings somewhere in this list, you need to recheck your work. For example, here is what my module search path looks like on Windows under Python 3.3, with my PYTHONPATH set to C:\code and a C:\Python33\mypath.pth path file that lists C:\Users\mark. The empty string at the front means current directory, and my two settings are merged in; the rest are standard library directories and files and the site-packages home for third-party extensions:

```
>>> import sys
```

```
>>> sys.path
['', 'C:\\code', 'C:\\Windows\\system32\\python33.zip', 'C:\\Python33\\DLLs',
'C:\\Python33\\lib', 'C:\\Python33', 'C:\\Users\\mark',
'C:\\Python33\\lib\\site-packages']
```

Second, if you know what you're doing, this list provides a way for scripts to tailor their search paths manually. As you'll see by example later in this part of the course, by modifying the sys.path list, you can modify the search path for all future imports made in a program's run. Such changes last only for the duration of the script, however;

PYTHONPATH and .pth files offer more permanent ways to modify the path—the first per user, and the second per installation.

On the other hand, some programs really do need to change sys.path. Scripts that run on web servers, for example, often run as the user "nobody" to limit machine access.

Because such scripts cannot usually depend on "nobody" to have set PYTHONPATH in any particular way, they often set sys.path manually to include required source directories, prior to running any import statements. A sys.path.append or sys.path.insert will often suffice, though will endure for a single program run only.

Module File Selection

Keep in mind that filename extensions (e.g., .py) are omitted from import statements intentionally. Python chooses the first file it can find on the search path that matches the imported name. In fact, imports are the point of interface to a host of external components—source code, multiple flavors of byte code, compiled extensions, and more. Python automatically selects any type that matches a module's name.

Module sources

For example, an import statement of the form import b might today load or resolve to:

- A source code file named b.py
- A byte code file named b.pyc
- An optimized byte code file named b.pyo (a less common format)
- A directory named b, for package imports
- A compiled extension module, coded in C, C++, or another language, and dynamically

linked when imported (e.g., b.so on Linux, or b.dll or b.pyd on Cygwin and Windows)

- A compiled built-in module coded in C and statically linked into Python
- A ZIP file component that is automatically extracted when imported
- An in-memory image, for frozen executables
- A Java class, in the Jython version of Python
- A .NET component, in the IronPython version of Python

C extensions, Jython, and package imports all extend imports beyond simple files. To importers, though, differences in the loaded file type are completely irrelevant, both when importing and when fetching module attributes. Saying import b gets whatever module b is, according to your module search path, and b.attr fetches an item in the module, be it a Python variable or a linked-in C function. Some standard modules we will use in this course are actually coded in C, not Python; because they look just like Python-coded module files, their clients don't have to care.

Selection priorities

If you have both a b.py and a b.so in different directories, Python will always load the one found in the first (leftmost) directory of your module search path during the leftto-right search of sys.path. But what happens if it finds both a b.py and a b.so in the same directory? In this case, Python follows a standard picking order, though this order is not guaranteed to stay the same over time or across implementations. In general, you should not depend on which type of file Python will choose within a given directory—make your module names distinct, or configure your module search path to make your module selection preferences explicit.

Import hooks and ZIP files

Normally, imports work as described in this section—they find and load files on your machine. However, it is possible to redefine much of what an import operation does in Python, using what are known as import hooks. These hooks can be used to make imports do various useful things, such as loading files from archives, performing decryption, and so on.

In fact, Python itself makes use of these hooks to enable files to be directly imported from ZIP archives: archived files are automatically extracted at import time when a .zip file is selected from the module import search path. One of the standard library directories in the earlier sys.path display, for example, is a .zip file today. For more details, see the Python standard library manual's description of the built-in __import__ function, the customizable tool that import statements actually run.

Optimized byte code files

Finally, Python also supports the notion of .pyo optimized byte code files, created and run with the -O Python command-line flag, and automatically generated by some install tools. Because these run only slightly faster than normal .pyc files (typically 5 percent faster), however, they are infrequently used. The PyPy system, for example, provides more substantial speedups

Module 15 - Coding Basics

Now that we've looked at the larger ideas behind modules, let's turn to some examples of modules in action. Although some of the early topics in this module will be review for linear readers who have already applied them in previous examples, we'll find that they quickly lead us to further details surrounding Python's modules that we haven't yet met, such as nesting, reloads, scopes, and more.

Python modules are easy to create; they're just files of Python program code created with a text editor. You don't need to write special syntax to tell Python you're making a module; almost any text file will do. Because Python handles all the details of finding and loading modules, modules are also easy to use; clients simply import a module, or specific names a module defines, and use the objects they reference.

Module creation

To define a module, simply use your text editor to type some Python code into a text file, and save it with a ".py" extension; any such file is automatically considered a Python module. All the names assigned at the top level of the module become its attributes (names associated with the module object) and are exported for clients to use —they morph from variable to module object attribute automatically.

For instance, if you type the following def into a file called module1.py and import it, you create a module object with one attribute—the name printer, which happens to be a reference to a function object:

```
def printer(x): # Module attribute
print(x)
```

Module Filenames

Before we go on, I should say a few more words about module filenames. You can call modules just about anything you like, but module filenames should end in a .py suffix if you plan to import them. The .py is technically optional for top-level files that will be run but not imported, but adding it in all cases makes your files' types more obvious and allows you to import any of your files in the future.

Because module names become variable names inside a Python program (without the .py), they should also follow the normal variable name rules. For instance, you can create a module file named if.py, but you cannot import it because if is a reserved word—when you try to run import if, you'll get a syntax error. In fact, both the names of module files and the names of directories used in package imports must conform to the rules for variable names; they may, for instance, contain only letters, digits, and underscores. Package directories also cannot contain platform-specific syntax such as spaces in their names.

When a module is imported, Python maps the internal module name to an external filename by adding a directory path from the module search path to the front, and a .py or other extension at the end. For instance, a module named M ultimately maps to some external file <directory>\M.<extension> that contains the module's code.

Other Kinds of Modules

As mentioned, it is also possible to create a Python module by writing code in an external language such as C, C++, and others (e.g., Java, in the Jython implementation of the language). Such modules are called extension modules, and they are generally used to wrap up external libraries for use in Python scripts. When imported by Python code, extension modules look and feel the same as modules coded as Python source code files—they are accessed with import statements, and they provide functions and objects as module attributes. Extension modules are beyond the scope of this course; see Python's standard manuals or advanced texts such as Programming Python for more details.

Module usage

Clients can use the simple module file we just wrote by running an import or from statement. Both statements find, compile, and run a module file's code, if it hasn't yet been loaded. The chief difference is that import fetches the module as a whole, so you must qualify to fetch its names; in contrast, from fetches (or copies) specific names out of the module.

Let's see what this means in terms of code. All of the following examples wind up calling the printer function defined in the prior section's module1.py module file, but in different ways.

The import Statement

In the first example, the name module1 serves two different purposes—it identifies an external file to be loaded, and it becomes a variable in the script, which references the module object after the file is loaded:

```
>>> import module1 # Get module as a whole (one or more)
>>> module1.printer('Hello world!') # Qualify to get names
Hello world!
```

The import statement simply lists one or more names of modules to load, separated by commas. Because it gives a name that refers to the whole module object, we must go through the module name to fetch its attributes (e.g., module1.printer).

The from Statement

By contrast, because from copies specific names from one file over to another scope, it allows us to use the copied names directly in the script without going through the module (e.g., printer):

```
>>> from module1 import printer # Copy out a variable (one or more)
>>> printer('Hello world!') # No need to qualify name
Hello world!
```

This form of from allows us to list one or more names to be copied out, separated by commas. Here, it has the same effect as the prior example, but because the imported name is copied into the scope where the from statement appears, using that name in the script requires less typing—we can use it directly instead of naming the enclosing module. In fact, we must; from doesn't assign the name of the module itself.

As you'll see in more detail later, the from statement is really just a minor extension to the import statement—it imports the module file as usual , but adds an extra step that copies one or more names (not objects) out of the file. The entire file is loaded, but you're given names for more direct access to its parts.

The from * Statement

Finally, the next example uses a special form of from: when we use a * instead of specific names, we get copies of all names assigned at the top level of the referenced module.

Here again, we can then use the copied name printer in our script without going through the module name:

```
>>> from module1 import * # Copy out _all_ variables
>>> printer('Hello world!')
Hello world!
```

Technically, both import and from statements invoke the same import operation; the from * form simply adds an extra step that copies all the names in the module into the importing scope. It essentially collapses one module's namespace into another; again, the net effect is less typing for us. Note that only * works in this context; you can't use pattern matching to select a subset of names (though you could with more work and a loop through a module's __dict__, discussed ahead).

And that's it—modules really are simple to use. To give you a better understanding of what really happens when you define and use modules, though, let's move on to look at some of their properties in more detail.

Imports Happen Only Once

One of the most common questions people seem to ask when they start using modules is, "Why won't my imports keep working?" They often report that the first import works fine, but later imports during an interactive session (or program run) seem to have no effect. In fact, they're not supposed to. This section explains why.

Modules are loaded and run on the first import or from, and only the first. This is on purpose—because importing is an expensive operation, by default Python does it just once per file, per process. Later import operations simply fetch the already loaded module object.

Initialization code

As one consequence, because top-level code in a module file is usually executed only once, you can use it to initialize variables. Consider the file simple.py, for example:

```
print('hello')
spam = 1 # Initialize variable
In this example, the print and = statements run the first time the module is imported,
and the variable spam is initialized at import time:
% python
>>> import simple # First import: loads and runs file's code
hello
>>> simple.spam # Assignment makes an attribute
1
```

Second and later imports don't rerun the module's code; they just fetch the already created module object from Python's internal modules table. Thus, the variable spam is not reinitialized:

```
>>> simple.spam = 2 # Change attribute in module
>>> import simple # Just fetches already loaded module
>>> simple.spam # Code wasn't rerun: attribute unchanged
2
```

Of course, sometimes you really want a module's code to be rerun on a subsequent import. We'll see how to do this with Python's reload function later in this module.

import and from Are Assignments

Just like def, import and from are executable statements, not compile-time declarations.

They may be nested in if tests, to select among options; appear in function defs, to be loaded only on calls (subject to the preceding note); be used in try statements, to provide defaults; and so on. They are not resolved or run until Python reaches them while executing your program. In other words, imported modules and names are not available until their associated import or from statements run.

Changing mutables in modules

Also, like def, the import and from are implicit assignments:

- import assigns an entire module object to a single name.
- from assigns one or more names to objects of the same names in another module.

All the things we've already discussed about assignment apply to module access, too.

For instance, names copied with a from become references to shared objects; as with function arguments, reassigning a copied name has no effect on the module from which it was copied, but changing a shared mutable object through a copied name can also change it in the module from which it was imported. To illustrate, consider the following file, small.py:

```
x = 1
y = [1, 2]
```

When importing with from, we copy names to the importer's scope that initially share objects referenced by the module's names:

```
% python
>>> from small import x, y # Copy two names out
>>> x = 42 # Changes local x only
```

```
>>> y[0] = 42 # Changes shared mutable in place
```

Here, x is not a shared mutable object, but y is. The names y in the importer and the importee both reference the same list object, so changing it from one place changes it in the other:

```
>>> import small # Get module name (from doesn't)
>>> small.x # Small's x is not my x
1
>>> small.y # But we share a changed mutable
[42, 2]
```

The effect is the same, except that here we're dealing with names in modules, not functions. Assignment works the same everywhere in Python.

Cross-file name changes

Recall from the preceding example that the assignment to x in the interactive session changed the name x in that scope only, not the x in the file—there is no link from a name copied with from back to the file it came from. To really change a global name in another file, you must use import:

```
% python
>>> from small import x, y # Copy two names out
>>> x = 42 # Changes my x only
>>> import small # Get module name
>>> small.x = 42 # Changes x in other module
```

This phenomenon was introduced. Because changing variables in other modules like this is a common source of confusion (and often a bad design choice), we'll revisit this technique again later in this part of the course. Note that the change to y[0] in the prior session is different; it changes an object, not a name, and the name in both modules references the same, changed object.

import and from Equivalence

Notice in the prior example that we have to execute an import statement after the from to access the small module name at all. from only copies names from one module to another; it does not assign the module name itself. At least conceptually, a from statement like this one:

```
from module import name1, name2        # Copy these two names out (only)
```

is equivalent to this statement sequence:

```
import module # Fetch the module object
name1 = module.name1          # Copy names out by assignment
name2 = module.name2
del module                    # Get rid of the module name
```

Like all assignments, the from statement creates new variables in the importer, which initially refer to objects of the same names in the imported file. Only the names are copied out, though, not the objects they reference, and not the name of the module itself. When we use the from * form of this statement (from module import *), the equivalence is the same, but all the top-level names in the module are copied over to the importing scope this way.

Notice that the first step of the from runs a normal import operation, with all the semantics outlined. Because of this, the from always imports the entire module into memory if it has not yet been imported, regardless of how many names it copies out of the file. There is no way to load just part of a module file (e.g., just one function), but because modules are byte code in Python instead of machine code, the performance implications are generally negligible.

Potential Pitfalls of the from Statement

Because the from statement makes the location of a variable more implicit and obscure (name is less meaningful to the reader than module.name), some Python users recommend using import instead of from most of the time. I'm not sure this advice is warranted, though; from is commonly and widely used, without too many dire consequences.

In practice, in realistic programs, it's often convenient not to have to type a module's name every time you wish to use one of its tools. This is especially true for large modules that provide many attributes—the standard library's tkinter GUI module, for example.

It is true that the from statement has the potential to corrupt namespaces, at least in principle—if you use it to import variables that happen to have the same names as existing variables in your scope, your variables will be silently overwritten. This problem doesn't occur with the simple import statement because you must always go through a module's name to get to its contents (module.attr will not clash with a variable named attr in your scope).

As long as you understand and expect that this can happen when using from, though, this isn't a major concern in practice, especially if you list the imported names explicitly (e.g., from module import x, y, z).

On the other hand, the from statement has more serious issues when used in conjunction with the reload call, as imported names might reference prior versions of objects.

Moreover, the from module import * form really can corrupt namespaces and make names difficult to understand, especially when applied to more than one file—in this case, there is no way to tell which module a name came from, short of searching the external source files. In effect, the from * form collapses one namespace into another, and so defeats the namespace partitioning feature of modules. We will explore these issues in more detail in the section "Module Gotchas".

Probably the best real-world advice here is to generally prefer import to from for simple modules, to explicitly list the variables you want in most from statements, and to limit the from * form to just one import per file. That way, any undefined names can be assumed to live in the module referenced with the from *. Some care is required when using the from statement, but armed with a little knowledge, most programmers find it to be a convenient way to access modules.

<u>When import is required</u>

The only time you really must use import instead of from is when you must use the same name defined in two different modules. For example, if two files define the same name differently:

```
# M.py
def func():
...do something...
# N.py
def func():
...do something else...
```

and you must use both versions of the name in your program, the from statement will fail—you can have only one assignment to the name in your scope:

```
# O.py
from M import func
from N import func # This overwrites the one we fetched from M
func() # Calls N.func only!
```

An import will work here, though, because including the name of the enclosing module makes the two names unique:

```
# O.py
import M, N # Get the whole modules, not their names
M.func() # We can call both names now
N.func() # The module names make them unique
```

This case is unusual enough that you're unlikely to encounter it very often in practice.

If you do, though, import allows you to avoid the name collision. Another way out of this dilemma is using the as extension:

```
# O.py
from M import func as mfunc # Rename uniquely with "as"
from N import func as nfunc
mfunc(); nfunc() # Calls one or the other
```

The as extension works in both import and from as a simple renaming tool (it can also be used to give a shorter synonym for a long module name in import).

Module namespaces

Modules are probably best understood as simply packages of names—i.e., places to define names you want to make visible to the rest of a system. Technically, modules usually correspond to files, and Python creates a module object to contain all the names assigned in a module file. But in simple terms, modules are just namespaces (places where names are created), and the names that live in a module are called its attributes.

This section expands on the details behind this model.

<u>Files Generate Namespaces</u>

I've mentioned that files morph into namespaces, but how does this actually happen?

The short answer is that every name that is assigned a value at the top level of a module file (i.e., not nested in a function or class body) becomes an attribute of that module.

For instance, given an assignment statement such as X = 1 at the top level of a module file M.py, the name X becomes an attribute of M, which we can refer to from outside the module as M.X. The name X also becomes a global variable to other code inside M.py, but we need to consider the notion of module loading and scopes a bit more formally to understand why:

- Module statements run on the first import. The first time a module is imported anywhere in a system, Python creates an empty module object and executes the statements in the module file one after another, from the top of the file to the bottom.
- Top-level assignments create module attributes. During an import, statements at the top level of the file not nested in a def or class that assign names (e.g., =, def) create attributes of the module object; assigned names are stored in the module's namespace.
- Module namespaces can be accessed via the attribute__dict__ or dir(M).
- Module namespaces created by imports are dictionaries; they may be accessed through the built-in __dict__ attribute associated with module objects and may be inspected with the dir function. The dir function is roughly equivalent to the sorted keys list of an object's __dict__ attribute, but it includes inherited names for classes, may not be complete, and is prone to changing from release to release.
- Modules are a single scope (local is global). As we saw, names at the top level of a module follow the same reference/assignment rules as names in a function, but the local and global scopes are the same—or, more formally, they follow the LEGB scope rule we met, but without the L and E lookup layers.

Crucially, though, the module's global scope becomes an attribute dictionary of a module object after the module has been loaded. Unlike function scopes, where the local namespace exists only while the function runs, a module file's scope becomes a module object's attribute namespace and lives on after the import, providing a source of tools to importers.

Here's a demonstration of these ideas. Suppose we create the following module file in a text editor and call it module2.py:

```
print('starting to load...')
import sys
name = 42
def func(): pass
class klass: pass
print('done loading.')
```

The first time this module is imported (or run as a program), Python executes its statements from top to bottom. Some statements create names in the module's namespace as a side effect, but others do actual work while the import is going on. For instance, the two print statements in this file execute at import time:

```
>>> import module2
starting to load...
done loading.
```

Once the module is loaded, its scope becomes an attribute namespace in the module object we get back from import. We can then access attributes in this namespace by qualifying them with the name of the enclosing module:

```
>>> module2.sys
<module 'sys' (built-in)>
>>> module2.name
42
>>> module2.func
<function func at 0x000000000222E7B8>
>>> module2.klass
<class 'module2.klass'>
```

Here, sys, name, func, and klass were all assigned while the module's statements were being run, so they are attributes after the import. We'll talk about classes in Part VI, but notice the sys attribute—import statements really assign module objects to names, and any type of assignment to a name at the top level of a file generates a module attribute.

Namespace Dictionaries: __dict__

In fact, internally, module namespaces are stored as dictionary objects. These are just normal dictionaries with all the usual methods. When needed we can access a module's namespace dictionary through the module's __dict__ attribute. Continuing the prior section's example (remember to wrap this in a list call in Python 3.X—it's a view object there, and contents may vary outside 3.3 used here):

```
>>> list(module2.__dict__.keys())
['__loader__', 'func', 'klass', '__builtins__', '__doc__', '__file__', '__name__',
'name', '__package__', 'sys', '__initializing__', '__cached__']
```

The names we assigned in the module file become dictionary keys internally, so some of the names here reflect top-level assignments in our file. However, Python also adds some names in the module's namespace for us; for instance, __file__ gives the name of the file the module was loaded from, and __name__ gives its name as known to importers (without the .py extension and directory path). To see just the names your code assigns, filter out the double-underscore names as we've done before, in built-in scope coverage:

```
>>> list(name for name in module2.__dict__.keys() if not name.startswith('__'))
['func', 'klass', 'name', 'sys']
>>> list(name for name in module2.__dict__ if not name.startswith('__'))
['func', 'sys', 'name', 'klass']
```

This time we're filtering with a generator instead of a list comprehension, and can omit the .keys() because dictionaries generate their keys automatically though implicitly; the effect is the same. We'll see similar __dict__ dictionaries on class-related objects in Part VI too. In both cases, attribute fetch is similar to dictionary indexing, though only the former kicks off inheritance in classes:

```
>>> module2.name, module2.__dict__['name']
(42, 42)
```

Attribute Name Qualification
Speaking of attribute fetch, now that you're becoming more familiar with modules, we should firm up the notion of name qualification more formally too. In Python, you can access the attributes of any object that has attributes using the qualification (a.k.a. attribute fetch) syntax object.attribute.

Qualification is really an expression that returns the value assigned to an attribute name associated with an object. For example, the expression module2.sys in the previous example fetches the value assigned to sys in module2. Similarly, if we have a built-in list object L, L.append returns the append method object associated with that list.

It's important to keep in mind that attribute qualification has nothing to do with the scope rules; it's an independent concept. When you use qualification to access names, you give Python an explicit object from which to fetch the specified names. The LEGB scope rule applies only to bare, unqualified names—it may be used for the leftmost name in a name path, but later names after dots search specific objects instead. Here are the rules:

Simple variables
X means search for the name X in the current scopes'

Qualification
X.Y means find X in the current scopes, then search for the attribute Y in the object
X (not in scopes).

Qualification paths
X.Y.Z means look up the name Y in the object X, then look up Z in the object X.Y.

Generality
Qualification works on all objects with attributes: modules, classes, C extension types, etc.
In Part VI, we'll see that attribute qualification means a bit more for classes—it's also the place where something called inheritance happens—but in general, the rules outlined here apply to all names in Python.

Imports Versus Scopes
As we've learned, it is never possible to access names defined in another module file without first importing that file. That is, you never automatically get to see names in another file, regardless of the structure of imports or function calls in your program. A variable's meaning is always determined by the locations of assignments in your source code, and attributes are always requested of an object explicitly.

For example, consider the following two simple modules. The first, moda.py, defines a variable X global to code in its file only, along with a function that changes the global X in this file:

```
X = 88 # My X: global to this file only
def f():
global X # Change this file's X
X = 99 # Cannot see names in other modules
```

The second module, modb.py, defines its own global variable X and imports and calls the function in the first module:

```
X = 11 # My X: global to this file only
import moda # Gain access to names in moda
moda.f() # Sets moda.X, not this file's X
print(X, moda.X)
```

When run, moda.f changes the X in moda, not the X in modb. The global scope for moda.f is always the file enclosing it, regardless of which module it is ultimately called from:

```
% python modb.py
11 99
```

In other words, import operations never give upward visibility to code in imported files —an imported file cannot see names in the importing file. More formally:

- Functions can never see names in other functions, unless they are physically enclosing.
- Module code can never see names in other modules, unless they are explicitly imported.

Such behavior is part of the lexical scoping notion—in Python, the scopes surrounding a piece of code are completely determined by the code's physical position in your file.

Scopes are never influenced by function calls or module imports.

Namespace Nesting

In some sense, although imports do not nest namespaces upward, they do nest downward.

That is, although an imported module never has direct access to names in a file that imports it, using attribute qualification paths it is possible to descend into arbitrarily nested modules and access their attributes. For example, consider the next three files. mod3.py defines a single global name and attribute by assignment:

```
X = 3
```

mod2.py in turn defines its own X, then imports mod3 and uses qualification to access the imported module's attribute:

```
X = 2
import mod3
print(X, end=' ') # My global X
print(mod3.X) # mod3's X
```

mod1.py also defines its own X, then imports mod2, and fetches attributes in both the first and second files:

```
X = 1
import mod2
print(X, end=' ') # My global X
print(mod2.X, end=' ') # mod2's X
print(mod2.mod3.X) # Nested mod3's X
```

Really, when mod1 imports mod2 here, it sets up a two-level namespace nesting. By using the path of names mod2.mod3.X, it can descend into mod3, which is nested in the imported mod2. The net effect is that mod1 can see the Xs in all three files, and hence has access to all three global scopes:

```
% python mod1.py
2 3
1 2 3
```

The reverse, however, is not true: mod3 cannot see names in mod2, and mod2 cannot see names in mod1. This example may be easier to grasp if you don't think in terms of namespaces and scopes, but instead focus on the objects involved. Within mod1, mod2 is just a name that refers to an object with attributes, some of which may refer to other objects with attributes (import is an assignment). For paths like mod2.mod3.X, Python simply evaluates from left to right, fetching attributes from objects along the way.

Note that mod1 can say import mod2, and then mod2.mod3.X, but it cannot say import mod2.mod3—this syntax invokes something called package (directory) imports, described later. Package imports also create module namespace nesting, but their import statements are taken to reflect directory trees, not simple file import chains.

Reloading modules

As we've seen, a module's code is run only once per process by default. To force a module's code to be reloaded and rerun, you need to ask Python to do so explicitly by calling the reload built-in function. In this section, we'll explore how to use reloads to make your systems more dynamic. In a nutshell:

- Imports (via both import and from statements) load and run a module's code only the first time the module is imported in a process.
- Later imports use the already loaded module object without reloading or rerunning the file's code.

- The reload function forces an already loaded module's code to be reloaded and rerun. Assignments in the file's new code change the existing module object in place.

Why care about reloading modules? In short, dynamic customization: the reload function allows parts of a program to be changed without stopping the whole program.

With reload, the effects of changes in components can be observed immediately. Reloading doesn't help in every situation, but where it does, it makes for a much shorter development cycle. For instance, imagine a database program that must connect to a server on startup; because program changes or customizations can be tested immediately after reloads, you need to connect only once while debugging. Long-running servers can update themselves this way, too.

Because Python is interpreted (more or less), it already gets rid of the compile/link steps you need to go through to get a C program to run: modules are loaded dynamically when imported by a running program. Reloading offers a further performance advantage by allowing you to also change parts of running programs without stopping.

Though beyond this course's scope, note that reload currently only works on modules written in Python; compiled extension modules coded in a language such as C can be dynamically loaded at runtime, too, but they can't be reloaded (though most users probably prefer to code customizations in Python anyhow!).

reload Basics

Unlike import and from:

- reload is a function in Python, not a statement.
- reload is passed an existing module object, not a new name.
- reload lives in a module in Python 3.X and must be imported itself.

Because reload expects an object, a module must have been previously imported successfully before you can reload it (if the import was unsuccessful due to a syntax or other error, you may need to repeat it before you can reload the module). Furthermore, the syntax of import statements and reload calls differs: as a function reloads require parentheses, but import statements do not. Abstractly, reloading looks like this:

```
import module # Initial import
...use module.attributes...
... # Now, go change the module file
...
from imp import reload # Get reload itself (in 3.X)
reload(module) # Get updated exports
...use module.attributes...
```

The typical usage pattern is that you import a module, then change its source code in a text editor, and then reload it. This can occur when working interactively, but also in larger programs that reload periodically.

When you call reload, Python rereads the module file's source code and reruns its toplevel statements. Perhaps the most important thing to know about reload is that it changes a module object in place; it does not delete and re-create the module object.

Because of that, every reference to an entire module object anywhere in your program is automatically affected by a reload. Here are the details:

- Reload runs a module file's new code in the module's current namespace. Rerunning a module file's code overwrites its existing namespace, rather than deleting and re-creating it.
- Top-level assignments in the file replace names with new values. For instance, rerunning a def statement replaces the prior version of the function in the module's namespace by reassigning the function name.
- Reloads impact all clients that use import to fetch modules. Because clients that use import qualify to fetch attributes, they'll find new values in the module object after a reload.
- Reloads impact future from clients only. Clients that used from to fetch attributes in the past won't be affected by a reload; they'll still have references to the old objects fetched before the reload.

- Reloads apply to a single module only. You must run them on each module you wish to update, unless you use code or tools that apply reloads transitively.

reload Example

To demonstrate, here's a more concrete example of reload in action. In the following, we'll change and reload a module file without stopping the interactive Python session.

Reloads are used in many other scenarios, too, but we'll keep things simple for illustration here.

First, in the text editor of your choice, write a module file named changer.py with the following contents:

```
message = "First version"
def printer():
print(message)
```

This module creates and exports two names—one bound to a string, and another to a function. Now, start the Python interpreter, import the module, and call the function it exports. The function will print the value of the global message variable:

```
% python
>>> import changer
>>> changer.printer()
First version
```

Keeping the interpreter active, now edit the module file in another window:

```
...modify changer.py without stopping Python...
% notepad changer.py
```

Change the global message variable, as well as the printer function body:

```
message = "After editing"
def printer():
print('reloaded:', message)
```

Then, return to the Python window and reload the module to fetch the new code. Notice in the following interaction that importing the module again has no effect; we get the original message, even though the file's been changed. We have to call reload in order to get the new version:

```
...back to the Python interpreter...
>>> import changer
>>> changer.printer() # No effect: uses loaded module
First version
>>> from imp import reload
>>> reload(changer) # Forces new code to load/run
<module 'changer' from '.\\changer.py'>
>>> changer.printer() # Runs the new version now
reloaded: After editing
```

Notice that reload actually returns the module object for us—its result is usually ignored, but because expression results are printed at the interactive prompt, Python shows a default <module 'name'...> representation.

Two final notes here: first, if you use reload, you'll probably want to pair it with import instead of from, as the latter isn't updated by reload operations—leaving your names in a state that's strange enough to warrant postponing further elaboration until this part's "gotchas". Second, reload by itself updates only a single module, but it's straightforward to code a function that applies it transitively to related modules.

Module 16 - Packages

So far, when we've imported modules, we've been loading files. This represents typical module usage, and it's probably the technique you'll use for most imports you'll code early on in your Python career. However, the module import story is a bit richer than I have thus far implied.

In addition to a module name, an import can name a directory path. A directory of Python code is said to be a package, so such imports are known as package imports. In effect, a package import turns a directory on your computer into another Python namespace, with attributes corresponding to the subdirectories and module files that the directory contains.

This is a somewhat advanced feature, but the hierarchy it provides turns out to be handy for organizing the files in a large system and tends to simplify module search path settings. As we'll see, package imports are also sometimes required to resolve import ambiguities when multiple program files of the same name are installed on a single machine.

Because it is relevant to code in packages only, we'll also introduce Python's recent relative imports model and syntax here. As we'll see, this model modifies search paths in 3.X, and extends the from statement for imports within packages in both 2.X and 3.X. This model can make such intrapackage imports more explicit and succinct, but comes with some tradeoffs that can impact your programs.

Finally, for developers using Python 3.3 and later, its new namespace package model—which allows packages to span multiple directories and requires no initialization file—is also introduced here. This new-style package model is optional and can be used in concert with the original (now known as "regular") package model, but it upends some of the original model's basic ideas and rules. Because of that, we'll explore regular packages here first for all developers, and present namespace packages last as an optional topic.

Package import basics

At a base level, package imports are straightforward—in the place where you have been naming a simple file in your import statements, you can instead list a path of names separated by periods:

```
import dir1.dir2.mod
```

The same goes for from statements:

```
from dir1.dir2.mod import x
```

The "dotted" path in these statements is assumed to correspond to a path through the directory hierarchy on your computer, leading to the file mod.py (or similar; the extension may vary). That is, the preceding statements indicate that on your machine there is a directory dir1, which has a subdirectory dir2, which contains a module file mod.py (or similar).

Furthermore, these imports imply that dir1 resides within some container directory dir0, which is a component of the normal Python module search path. In other words, these two import statements imply a directory structure that looks something like this (shown with Windows backslash separators):

```
dir0\dir1\dir2\mod.py # Or mod.pyc, mod.so, etc.
```

The container directory dir0 needs to be added to your module search path unless it's the home directory of the top-level file, exactly as if dir1 were a simple module file.

More formally, the leftmost component in a package import path is still relative to a directory included in the sys.path module search path list we explored. From there down, though, the import statements in your script explicitly give the directory paths leading to modules in packages.

Packages and Search Path Settings

If you use this feature, keep in mind that the directory paths in your import statements can be only variables separated by periods. You cannot use any platform-specific path syntax in your import statements, such as C:\dir1, My Documents.dir2, or ../dir1—these do not work syntactically. Instead, use any such platform-specific syntax in your module search path settings to name the container directories.

For instance, in the prior example, dir0—the directory name you add to your module search path—can be an arbitrarily long and platform-specific directory path leading up to dir1. You cannot use an invalid statement like this:

```
import C:\mycode\dir1\dir2\mod # Error: illegal syntax
```

But you can add C:\mycode to your PYTHONPATH variable or a .pth file, and say this in your script:

```
import dir1.dir2.mod
```

In effect, entries on the module search path provide platform-specific directory path prefixes, which lead to the leftmost names in import and from statements. These import statements themselves provide the remainder of the directory path in a platform-neutral fashion.1

As for simple file imports, you don't need to add the container directory dir0 to your module search path if it's already there, it will be if it's the home directory of the top-level file, the directory you're working in interactively, a standard library directory, or the site-packages third-party install root. One way or another, though, your module search path must include all the directories containing leftmost components in your code's package import statements.

Package __init__.py Files

If you choose to use package imports, there is one more constraint you must follow: at least until Python 3.3, each directory named within the path of a package import statement must contain a file named __init__.py, or your package imports will fail. That is, in the example we've been using, both dir1 and dir2 must contain a file called __init__.py; the container directory dir0 does not require such a file because it's not listed in the import statement itself.

More formally, for a directory structure such as this:

```
dir0\dir1\dir2\mod.py
```

and an import statement of the form:

```
import dir1.dir2.mod
```

the following rules apply:

- dir1 and dir2 both must contain an __init__.py file.
- dir0, the container, does not require an __init__.py file; this file will simply be ignored if present.
- dir0, not dir0\dir1, must be listed on the module search path sys.path.

To satisfy the first two of these rules, package creators must create files of the sort we'll explore here. To satisfy the latter of these, dir0 must be an automatic path component (the home, libraries, or site-packages directories), or be given in PYTHONPATH or .pth file settings or manual sys.path changes.

The net effect is that this example's directory structure should be as follows, with indentation designating directory nesting:

```
dir0\ # Container on module search path
dir1\
__init__.py
```

```
dir2\
__init__.py
mod.py
```

The __init__.py files can contain Python code, just like normal module files. Their names are special because their code is run automatically the first time a Python program imports a directory, and thus serves primarily as a hook for performing initialization steps required by the package. These files can also be completely empty, though, and sometimes have additional roles—as the next section explains.

Package initialization file roles

In more detail, the __init__.py file serves as a hook for package initialization-time actions, declares a directory as a Python package, generates a module namespace for a directory, and implements the behavior of from * (i.e., from .. import *) statements when used with directory imports:

Package initialization

The first time a Python program imports through a directory, it automatically runs all the code in the directory's __init__.py file. Because of that, these files are a natural place to put code to initialize the state required by files in a package. For instance, a package might use its initialization file to create required data files, open connections to databases, and so on. Typically, __init__.py files are not meant to be useful if executed directly; they are run automatically when a package is first accessed.

Module usability declarations

Package __init__.py files are also partly present to declare that a directory is a Python package. In this role, these files serve to prevent directories with common names from unintentionally hiding true modules that appear later on the module search path. Without this safeguard, Python might pick a directory that has nothing to do with your code, just because it appears nested in an earlier directory on the search path. As we'll see later, Python 3.3's namespace packages obviate much of this role, but achieve a similar effect algorithmically by scanning ahead on the path to find later files.

Module namespace initialization

In the package import model, the directory paths in your script become real nested object paths after an import. For instance, in the preceding example, after the import the expression dir1.dir2 works and returns a module object whose namespace contains all the names assigned by dir2's __init__.py initialization file.

Such files provide a namespace for module objects created for directories, which would otherwise have no real associated module file.

from * statement behavior

As an advanced feature, you can use __all__ lists in __init__.py files to define what is exported when a directory is imported with the from * statement form. In an __init__.py file, the __all__ list is taken to be the list of submodule names that should be automatically imported when from * is used on the package (directory) name. If __all__ is not set, the from * statement does not automatically load submodules nested in the directory; instead, it loads just names defined by assignments in the directory's _init__.py file, including any submodules explicitly imported by code in this file. For instance, the statement from submodule import X in a directory's __init__.py makes the name X available in that directory's namespace. (We'll see additional roles for __all__: it serves to declare from * exports of simple files as well.)

You can also simply leave these files empty, if their roles are beyond your needs (and frankly, they are often empty in practice). They must exist, though, for your directory imports to work at all.

Package import example
Let's actually code the example we've been talking about to show how initialization files and paths come into play. The following three files are coded in a directory dir1 and its subdirectory dir2—comments give the pathnames of these files:

```
# dir1\__init__.py
print('dir1 init')
x = 1
# dir1\dir2\__init__.py
print('dir2 init')
y = 2
# dir1\dir2\mod.py
print('in mod.py')
z = 3
```

Here, dir1 will be either an immediate subdirectory of the one we're working in (i.e., the home directory), or an immediate subdirectory of a directory that is listed on the module search path (technically, on sys.path). Either way, dir1's container does not need an __init__.py file.

import statements run each directory's initialization file the first time that directory is traversed, as Python descends the path; print statements are included here to trace their execution:

```
C:\code> python # Run in dir1's container directory
>>> import dir1.dir2.mod # First imports run init files
dir1 init
dir2 init
in mod.py
>>>
>>> import dir1.dir2.mod # Later imports do not
```

Just like module files, an already imported directory may be passed to reload to force reexecution of that single item. As shown here, reload accepts a dotted pathname to reload nested directories and files:

```
>>> from imp import reload # from needed in 3.X only
>>> reload(dir1)
dir1 init
<module 'dir1' from '.\\dir1\\__init__.py'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from '.\\dir1\\dir2\\__init__.py'>
```

Once imported, the path in your import statement becomes a nested object path in your script. Here, mod is an object nested in the object dir2, which in turn is nested in the object dir1:

```
>>> dir1
<module 'dir1' from '.\\dir1\\__init__.py'>
>>> dir1.dir2
<module 'dir1.dir2' from '.\\dir1\\dir2\\__init__.py'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from '.\\dir1\\dir2\\mod.py'>
```

In fact, each directory name in the path becomes a variable assigned to a module object whose namespace is initialized by all the assignments in that directory's __init__.py file. dir1.x refers to the variable x assigned in dir1__init__.py, much as mod.z refers to the variable z assigned in mod.py:

```
>>> dir1.x
1
>>> dir1.dir2.y
```

```
2
>>> dir1.dir2.mod.z
3
```

from Versus import with Packages

import statements can be somewhat inconvenient to use with packages, because you may have to retype the paths frequently in your program. In the prior section's example, for instance, you must retype and rerun the full path from dir1 each time you want to reach z. If you try to access dir2 or mod directly, you'll get an error:

```
>>> dir2.mod
NameError: name 'dir2' is not defined
>>> mod.z
NameError: name 'mod' is not defined
```

It's often more convenient, therefore, to use the from statement with packages to avoid retyping the paths at each access. Perhaps more importantly, if you ever restructure your directory tree, the from statement requires just one path update in your code, whereas imports may require many. The import as extension, discussed formally in the next module, can also help here by providing a shorter synonym for the full path, and a renaming tool when the same name appears in multiple modules:

```
C:\code> python
>>> from dir1.dir2 import mod # Code path here only
dir1 init
dir2 init
in mod.py
>>> mod.z # Don't repeat path
3
>>> from dir1.dir2.mod import z
>>> z
3
>>> import dir1.dir2.mod as mod # Use shorter name
>>> mod.z
3
>>> from dir1.dir2.mod import z as modz # Ditto if names clash
>>> modz
3
```

Why use package imports?

If you're new to Python, make sure that you've mastered simple modules before stepping up to packages, as they are a somewhat more advanced feature. They do serve useful roles, though, especially in larger programs: they make imports more informative, serve as an organizational tool, simplify your module search path, and can resolve ambiguities.

First of all, because package imports give some directory information in program files, they both make it easier to locate your files and serve as an organizational tool. Without package paths, you must often resort to consulting the module search path to find files Moreover, if you organize your files into subdirectories for functional areas, package imports make it more obvious what role a module plays, and so make your code more readable. For example, a normal import of a file in a directory somewhere on the module search path, like this:

```
import utilities
```

offers much less information than an import that includes the path:

```
import database.client.utilities
```

Package imports can also greatly simplify your PYTHONPATH and .pth file search path settings. In fact, if you use explicit package imports for all your cross-directory imports, and you make those package imports relative to a common root directory where all your Python code is stored, you really only need a single entry on your search path:

the common root. Finally, package imports serve to resolve ambiguities by making explicit exactly which files you want to import—and resolve conflicts when the same module name appears in more than one place. The next section explores this role in more detail.

Module 17 - OOP: The Big Picture

So far in this course, we've been using the term "object" generically. Really, the code written up to this point has been object-based—we've passed objects around our scripts, used them in expressions, called their methods, and so on. For our code to qualify as being truly object-oriented (OO), though, our objects will generally need to also participate in something called an inheritance hierarchy.

This module begins our exploration of the Python class—a coding structure and device used to implement new kinds of objects in Python that support inheritance. Classes are Python's main object-oriented programming (OOP) tool, so we'll also look at OOP basics along the way in this part of the course. OOP offers a different and often more effective way of programming, in which we factor code to minimize redundancy, and write new programs by customizing existing code instead of changing it in place.

In Python, classes are created with a new statement: the class. As you'll see, the objects defined with classes can look a lot like the built-in types we studied earlier in the course.

In fact, classes really just apply and extend the ideas we've already covered; roughly, they are packages of functions that use and process built-in object types. Classes, though, are designed to create and manage new objects, and support inheritance—a mechanism of code customization and reuse above and beyond anything we've seen so far.

One note up front: in Python, OOP is entirely optional, and you don't need to use classes just to get started. You can get plenty of work done with simpler constructs such as functions, or even simple top-level script code. Because using classes well requires some up-front planning, they tend to be of more interest to people who work in strategic mode (doing long-term product development) than to people who work in tactical mode (where time is in very short supply).

Still, as you'll see in this part of the course, classes turn out to be one of the most useful tools Python provides. When used well, classes can actually cut development time radically. They're also employed in popular Python tools like the tkinter GUI API, so most Python programmers will usually find at least a working knowledge of class basics helpful.

Why use classes?
In simple terms, classes are just a way to define new sorts of stuff, reflecting real objects in a program's domain. If we implement it using classes, we can model more of its real-world structure and relationships. Two aspects of OOP prove useful here:

Inheritance
Pizza-making robots are kinds of robots, so they possess the usual robot-y properties. In OOP terms, we say they "inherit" properties from the general category of all robots. These common properties need to be implemented only once for the general case and can be reused in part or in full by all types of robots we may build in the future.

Composition
Pizza-making robots are really collections of components that work together as a team. For instance, for our robot to be successful, it might need arms to roll dough, motors to maneuver to the oven, and so on. In OOP parlance, our robot is an example of composition; it contains other objects that it activates to do its bidding. Each component might be coded as a class, which defines its own behavior and relationships.

General OOP ideas like inheritance and composition apply to any application that can be decomposed into a set of objects. For example, in typical GUI systems, interfaces are written as collections of widgets—buttons, labels, and so on—which are all drawn when their container is drawn (composition). Moreover, we may be able to write our own custom widgets—buttons with unique fonts, labels with new color schemes, and the like—which are specialized versions of more general interface devices (inheritance).

From a more concrete programming perspective, classes are Python program units, just like functions and modules: they are another compartment for packaging logic and data. In fact, classes also define new namespaces, much like

modules. But, compared to other program units we've already seen, classes have three critical distinctions that make them more useful when it comes to building new objects:

Multiple instances
Classes are essentially factories for generating one or more objects. Every time we call a class, we generate a new object with a distinct namespace. Each object generated from a class has access to the class's attributes and gets a namespace of its own for data that varies per object. Classes offer a complete programming solution.

Customization via inheritance
Classes also support the OOP notion of inheritance; we can extend a class by redefining its attributes outside the class itself in new software components coded as subclasses. More generally, classes can build up namespace hierarchies, which define names to be used by objects created from classes in the hierarchy. This supports multiple customizable behaviors more directly than other tools.

Operator overloading
By providing special protocol methods, classes can define objects that respond to the sorts of operations we saw at work on built-in types. For instance, objects made with classes can be sliced, concatenated, indexed, and so on.

Python provides hooks that classes can use to intercept and implement any built-in type operation.

At its base, the mechanism of OOP in Python is largely just two bits of magic: a special first argument in functions (to receive the subject of a call) and inheritance attribute search (to support programming by customization). Other than this, the model is largely just functions that ultimately process built-in types. While not radically new, though, OOP adds an extra layer of structure that supports better programming than flat procedural models. Along with the functional tools we met earlier, it represents a major abstraction step above computer hardware that helps us build more sophisticated programs.

OOP from 30,000 feet
Before we see what this all means in terms of code, I'd like to say a few words about the general ideas behind OOP. If you've never done anything object-oriented in your life before now, some of the terminology in this module may seem a bit perplexing on the first pass. Moreover, the motivation for these terms may be elusive until you've had a chance to study the ways that programmers apply them in larger systems. OOP is as much an experience as a technology.

Attribute Inheritance Search
The good news is that OOP is much simpler to understand and use in Python than in other languages, such as C++ or Java. As a dynamically typed scripting language, Python removes much of the syntactic clutter and complexity that clouds OOP in other tools. In fact, much of the OOP story in Python boils down to this expression:

```
object.attribute
```

We've been using this expression throughout the course to access module attributes, call methods of objects, and so on. When we say this to an object that is derived from a class statement, however, the expression kicks off a search in Python—it searches a tree of linked objects, looking for the first appearance of attribute that it can find.

When classes are involved, the preceding Python expression effectively translates to the following in natural language: Find the first occurrence of attribute by looking in object, then in all classes above it, from bottom to top and left to right.

In other words, attribute fetches are simply tree searches. The term inheritance is applied because objects lower in a tree inherit attributes attached to objects higher in that tree. As the search proceeds from the bottom up, in a sense, the objects linked into a tree are the union of all the attributes defined in all their tree parents, all the way up the tree.

In Python, this is all very literal: we really do build up trees of linked objects with code, and Python really does climb this tree at runtime searching for attributes every time we use the object.attribute expression. To make this more concrete, Figure 26-1 sketches an example of one of these trees.

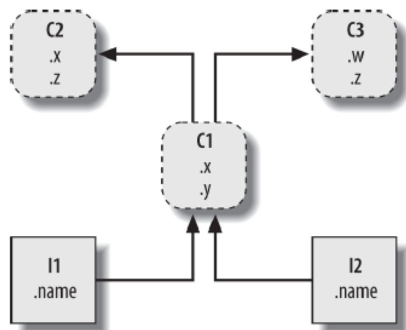


Figure 26-1. A class tree, with two instances at the bottom (I1 and I2), a class above them (C1), and two superclasses at the top (C2 and C3). All of these objects are namespaces (packages of variables), and the inheritance search is simply a search of the tree from bottom to top looking for the lowest occurrence of an attribute name. Code implies the shape of such trees.

In this figure, there is a tree of five objects labeled with variables, all of which have attached attributes, ready to be searched. More specifically, this tree links together three class objects (the ovals C1, C2, and C3) and two instance objects (the rectangles I1 and I2) into an inheritance search tree. Notice that in the Python object model, classes and the instances you generate from them are two distinct object types:

Classes
Serve as instance factories. Their attributes provide behavior—data and functions —that is inherited by all the instances generated from them (e.g., a function to compute an employee's salary from pay and hours).

Instances
Represent the concrete items in a program's domain. Their attributes record data that varies per specific object (e.g., an employee's Social Security number).

In terms of search trees, an instance inherits attributes from its class, and a class inherits attributes from all classes above it in the tree.

In Figure 26-1, we can further categorize the ovals by their relative positions in the tree.

We usually call classes higher in the tree (like C2 and C3) superclasses; classes lower in the tree (like C1) are known as subclasses. These terms refer to both relative tree positions and roles. Superclasses provide behavior shared by all their subclasses, but because the search proceeds from the bottom up, subclasses may override behavior defined in their superclasses by redefining superclass names lower in the tree.1

As these last few words are really the crux of the matter of software customization in OOP, let's expand on this concept. Suppose we build up the tree in Figure 26-1, and then say this:

```
I2.w
```

Right away, this code invokes inheritance. Because this is an object.attribute expression, it triggers a search of the tree in Figure 26-1—Python will search for the attribute w by looking in I2 and above. Specifically, it will search the linked objects in this order:

```
I2, C1, C2, C3
```

and stop at the first attached w it finds (or raise an error if w isn't found at all). In this case, w won't be found until C3 is searched because it appears only in that object. In other words, I2.w resolves to C3.w by virtue of the automatic search. In OOP terminology, I2 "inherits" the attribute w from C3.

Ultimately, the two instances inherit four attributes from their classes: w, x, y, and z.

Other attribute references will wind up following different paths in the tree. For example:
- I1.x and I2.x both find x in C1 and stop because C1 is lower than C2.
- I1.y and I2.y both find y in C1 because that's the only place y appears.
- I1.z and I2.z both find z in C2 because C2 is further to the left than C3.
- I2.name finds name in I2 without climbing the tree at all.

Trace these searches through the tree in Figure 26-1 to get a feel for how inheritance searches work in Python.

The first item in the preceding list is perhaps the most important to notice—because C1 redefines the attribute x lower in the tree, it effectively replaces the version above it in C2. As you'll see in a moment, such redefinitions are at the heart of software customization in OOP—by redefining and replacing the attribute, C1 effectively customizes what it inherits from its superclasses.

Classes and Instances

Although they are technically two separate object types in the Python model, the classes and instances we put in these trees are almost identical—each type's main purpose is to serve as another kind of namespace—a package of variables, and a place where we can attach attributes. If classes and instances therefore sound like modules, they should; however, the objects in class trees also have automatically searched links to other namespace objects, and classes correspond to statements, not entire files.

The primary difference between classes and instances is that classes are a kind of factory for generating instances. For example, in a realistic application, we might have an Employee class that defines what it means to be an employee; from that class, we generate actual Employee instances. This is another difference between classes and modules—we only ever have one instance of a given module in memory (that's why we have to reload a module to get its new code), but with classes, we can make as many instances as we need.

Operationally, classes will usually have functions attached to them (e.g., computeSalary), and the instances will have more basic data items used by the class's functions (e.g., hoursWorked). In fact, the object-oriented model is not that different from the classic data-processing model of programs plus records—in OOP, instances are like records with "data," and classes are the "programs" for processing those records. In OOP, though, we also have the notion of an inheritance hierarchy, which supports software customization better than earlier models.

Method Calls

In the prior section, we saw how the attribute reference I2.w in our example class tree was translated to C3.w by the inheritance search procedure in Python. Perhaps just as important to understand as the inheritance of attributes, though, is what happens when we try to call methods—functions attached to classes as attributes.

If this I2.w reference is a function call, what it really means is "call the C3.w function to process I2." That is, Python will automatically map the call I2.w() into the call C3.w(I2), passing in the instance as the first argument to the inherited function.

In fact, whenever we call a function attached to a class in this fashion, an instance of the class is always implied. This implied subject or context is part of the reason we refer to this as an object-oriented model—there is always a subject object when an operation is run. In a more realistic example, we might invoke a method called giveRaise attached as an attribute to an Employee class; such a call has no meaning unless qualified with the employee to whom the raise should be given.

As we'll see later, Python passes in the implied instance to a special first argument in the method, called self by convention. Methods go through this argument to process the subject of the call. As we'll also learn, methods can be called through either an instance—bob.giveRaise()—or a class—Employee.giveRaise(bob)—and both forms serve purposes in our scripts. These calls also illustrate both of the key ideas in OOP: to run a bob.giveRaise() method call, Python:

1. Looks up giveRaise from bob, by inheritance search
2. Passes bob to the located giveRaise function, in the special self argument

When you call Employee.giveRaise(bob), you're just performing both steps yourself.

This description is technically the default case (Python has additional method types we'll meet later), but it applies to the vast majority of the OOP code written in the language. To see how methods receive their subjects, though, we need to move on to some code.

Coding Class Trees

Although we are speaking in the abstract here, there is tangible code behind all these ideas, of course. We construct trees and their objects with class statements and class calls, which we'll meet in more detail later. In short:

- Each class statement generates a new class object.
- Each time a class is called, it generates a new instance object.
- Instances are automatically linked to the classes from which they are created.
- Classes are automatically linked to their superclasses according to the way we list them in parentheses in a class header line; the left-to-right order there gives the order in the tree.

To build the tree in Figure 26-1, for example, we would run Python code of the following form. Like function definition, classes are normally coded in module files and are run during an import (I've omitted the guts of the class statements here for brevity):

```
class C2: ... # Make class objects (ovals)
class C3: ...
class C1(C2, C3): ... # Linked to superclasses (in this order)
I1 = C1() # Make instance objects (rectangles)
I2 = C1() # Linked to their classes
```

Here, we build the three class objects by running three class statements, and make the two instance objects by calling the class C1 twice, as though it were a function. The instances remember the class they were made from, and the class C1 remembers its listed superclasses.

Technically, this example is using something called multiple inheritance, which simply means that a class has more than one superclass above it in the class tree—a useful technique when you wish to combine multiple tools. In Python, if there is more than one superclass listed in parentheses in a class statement (like C1's here), their left-toright order gives the order in which those superclasses will be searched for attributes by inheritance. The leftmost version of a name is used by default, though you can always choose a name by asking for it from the class it lives in (e.g., C3.z).

Because of the way inheritance searches proceed, the object to which you attach an attribute turns out to be crucial—it determines the name's scope. Attributes attached to instances pertain only to those single instances, but attributes attached to classes are shared by all their subclasses and instances. Later, we'll study the code that hangs attributes on these objects in depth. As we'll find:

- Attributes are usually attached to classes by assignments made at the top level in class statement blocks, and not nested inside function def statements there.
- Attributes are usually attached to instances by assignments to the special argument passed to functions coded inside classes, called self.

For example, classes provide behavior for their instances with method functions we create by coding def statements inside class statements. Because such nested defs assign names within the class, they wind up attaching attributes to the class object that will be inherited by all instances and subclasses:

```
class C2: ... # Make superclass objects
class C3: ...
class C1(C2, C3): # Make and link class C1
def setname(self, who): # Assign name: C1.setname
self.name = who # Self is either I1 or I2
```

```
I1 = C1() # Make two instances
I2 = C1()
I1.setname('bob') # Sets I1.name to 'bob'
I2.setname('sue') # Sets I2.name to 'sue'
print(I1.name) # Prints 'bob'
```

There's nothing syntactically unique about def in this context. Operationally, though, when a def appears inside a class like this, it is usually known as a method, and it automatically receives a special first argument—called self by convention—that provides a handle back to the instance to be processed. Any values you pass to the method yourself go to arguments after self (here, to who).

Because classes are factories for multiple instances, their methods usually go through this automatically passed-in self argument whenever they need to fetch or set attributes of the particular instance being processed by a method call. In the preceding code, self is used to store a name in one of two instances.

Like simple variables, attributes of classes and instances are not declared ahead of time, but spring into existence the first time they are assigned values. When a method assigns to a self attribute, it creates or changes an attribute in an instance at the bottom of the class tree (i.e., one of the rectangles in Figure 26-1) because self automatically refers to the instance being processed—the subject of the call.

In fact, because all the objects in class trees are just namespace objects, we can fetch or set any of their attributes by going through the appropriate names. Saying C1.setname is as valid as saying I1.setname, as long as the names C1 and I1 are in your code's scopes.

Operator Overloading

As currently coded, our C1 class doesn't attach a name attribute to an instance until the setname method is called. Indeed, referencing I1.name before calling I1.setname would produce an undefined name error. If a class wants to guarantee that an attribute like name is always set in its instances, it more typically will fill out the attribute at construction time, like this:

```
class C2: ... # Make superclass objects
class C3: ...
class C1(C2, C3):
def __init__(self, who): # Set name when constructed
self.name = who # Self is either I1 or I2
I1 = C1('bob') # Sets I1.name to 'bob'
I2 = C1('sue') # Sets I2.name to 'sue'
print(I1.name) # Prints 'bob'
```

If it's coded or inherited, Python automatically calls a method named __init__ each time an instance is generated from a class. The new instance is passed in to the self argument of __init__ as usual, and any values listed in parentheses in the class call go to arguments two and beyond. The effect here is to initialize instances when they are
made, without requiring extra method calls.

The __init__ method is known as the constructor because of when it is run. It's the most commonly used representative of a larger class of methods called operator overloading methods, which we'll discuss in more detail later. Such methods are inherited in class trees as usual and have double underscores at the start and end of their names to make them distinct. Python runs them automatically when instances that support them appear in the corresponding operations, and they are mostly an alternative to using simple method calls. They're also optional: if omitted, the operations are not supported. If no __init__ is present, class calls return an empty instance, without initializing it.

For example, to implement set intersection, a class might either provide a method named intersect, or overload the & expression operator to dispatch to the required logic by coding a method named __and__. Because the operator scheme makes instances look and feel more like built-in types, it allows some classes to provide a consistent and natural interface, and be compatible with code that expects a built-in type. Still, apart from the __init__ constructor—

which appears in most realistic classes—many programs may be better off with simpler named methods unless their objects are similar to built-ins. A giveRaise may make sense for an Employee, but a & might not.

OOP Is About Code Reuse

And that, along with a few syntax details, is most of the OOP story in Python. Of course, there's a bit more to it than just inheritance. For example, operator overloading is much more general than I've described so far—classes may also provide their own implementations of operations such as indexing, fetching attributes, printing, and more. By and large, though, OOP is about looking up attributes in trees with a special first argument in functions.

So why would we be interested in building and searching trees of objects? Although it takes some experience to see how, when used well, classes support code reuse in ways that other Python program components cannot. In fact, this is their highest purpose.

With classes, we code by customizing existing software, instead of either changing existing code in place or starting from scratch for each new project. This turns out to be a powerful paradigm in realistic programming.

At a fundamental level, classes are really just packages of functions and other names, much like modules. However, the automatic attribute inheritance search that we get with classes supports customization of software above and beyond what we can do with modules and functions. Moreover, classes provide a natural structure for code that packages and localizes logic and names, and so aids in debugging.

For instance, because methods are simply functions with a special first argument, we can mimic some of their behavior by manually passing objects to be processed to simple functions. The participation of methods in class inheritance, though, allows us to naturally customize existing software by coding subclasses with new method definitions, rather than changing existing code in place. There is really no such concept with modules and functions.

Polymorphism and classes

As an example, suppose you're assigned the task of implementing an employee database application. As a Python OOP programmer, you might begin by coding a general superclass that defines default behaviors common to all the kinds of employees in your organization:

```
class Employee: # General superclass
def computeSalary(self): ... # Common or default behaviors
def giveRaise(self): ...
def promote(self): ...
def retire(self): ...
```

Once you've coded this general behavior, you can specialize it for each specific kind of employee to reflect how the various types differ from the norm. That is, you can code subclasses that customize just the bits of behavior that differ per employee type; the rest of the employee types' behavior will be inherited from the more general class. For example, if engineers have a unique salary computation rule (perhaps it's not hours times rate), you can replace just that one method in a subclass:

```
class Engineer(Employee): # Specialized subclass
def computeSalary(self): ... # Something custom here
```

Because the computeSalary version here appears lower in the class tree, it will replace (override) the general version in Employee. You then create instances of the kinds of employee classes that the real employees belong to, to get the correct behavior:

```
bob = Employee() # Default behavior
sue = Employee() # Default behavior
tom = Engineer() # Custom salary calculator
```

Notice that you can make instances of any class in a tree, not just the ones at the bottom —the class you make an instance from determines the level at which the attribute search will begin, and thus which versions of the methods it will employ.

Ultimately, these three instance objects might wind up embedded in a larger container object—for instance, a list, or an instance of another class—that represents a department or company using the composition idea mentioned at the start of this module.

When you later ask for these employees' salaries, they will be computed according to the classes from which the objects were made, due to the principles of the inheritance search:

```
company = [bob, sue, tom] # A composite object
for emp in company:
print(emp.computeSalary()) # Run this object's version: default or custom
```

This is yet another instance of the idea of polymorphism. Recall that polymorphism means that the meaning of an operation depends on the object being operated on. That is, code shouldn't care about what an object is, only about what it does. Here, the method computeSalary is located by inheritance search in each object before it is called. The net effect is that we automatically run the correct version for the object being processed. Trace the code to see why.

In other applications, polymorphism might also be used to hide (i.e., encapsulate) interface differences. For example, a program that processes data streams might be coded to expect objects with input and output methods, without caring what those methods actually do:

```
def processor(reader, converter, writer):
  while True:
    data = reader.read()
    if not data: break
    data = converter(data)
    writer.write(data)
```

By passing in instances of subclasses that specialize the required read and write method interfaces for various data sources, we can reuse the processor function for any data source we need to use, both now and in the future:

```
class Reader:
def read(self): ... # Default behavior and tools
def other(self): ...
class FileReader(Reader):
def read(self): ... # Read from a local file
class SocketReader(Reader):
def read(self): ... # Read from a network socket
...
processor(FileReader(...), Converter, FileWriter(...))
processor(SocketReader(...), Converter, TapeWriter(...))
processor(FtpReader(...), Converter, XmlWriter(...))
```

Moreover, because the internal implementations of those read and write methods have been factored into single locations, they can be changed without impacting code such as this that uses them. The processor function might even be a class itself to allow the conversion logic of converter to be filled in by inheritance, and to allow readers and writers to be embedded by composition.

Module 18 - Class Coding Basics

Now that we've talked about OOP in the abstract, it's time to see how this translates to actual code. This module begins to fill in the syntax details behind the class model in Python.

If you've never been exposed to OOP in the past, classes can seem somewhat complicated if taken in a single dose. To make class coding easier to absorb, we'll begin our detailed exploration of OOP by taking a first look at some basic classes in action in this module. In their basic form, Python classes are easy to understand.

In fact, classes have just three primary distinctions. At a base level, they are mostly just namespaces, much like the modules we studied in Part V. Unlike modules, though, classes also have support for generating multiple objects, for namespace inheritance, and for operator overloading. Let's begin our class statement tour by exploring each of these three distinctions in turn.

Classes generate multiple instance objects

To understand how the multiple objects idea works, you have to first understand that there are two kinds of objects in Python's OOP model: class objects and instance objects.

Class objects provide default behavior and serve as factories for instance objects.

Instance objects are the real objects your programs process—each is a namespace in its own right, but inherits (i.e., has automatic access to) names in the class from which it was created. Class objects come from statements, and instances come from calls; each time you call a class, you get a new instance of that class.

This object-generation concept is very different from most of the other program constructs we've seen so far in this course. In effect, classes are essentially factories for generating multiple instances. By contrast, only one copy of each module is ever imported into a single program. In fact, this is why reload works as it does, updating a singleinstance shared object in place. With classes, each instance can have its own, independent data, supporting multiple versions of the object that the class models.

In this role, class instances are similar to the per-call state of the closure (a.k.a. factory) functions, but this is a natural part of the class model, and state in classes is explicit attributes instead of implicit scope references. Moreover, this is just part of what classes do—they also support customization by inheritance, operator overloading,
and multiple behaviors via methods. Generally speaking, classes are a more complete programming tool, though OOP and function programming are not mutually exclusive paradigms. We may combine them by using functional tools in methods, by coding methods that are themselves generators, by writing user-defined iterators, and so on.

The following is a quick summary of the bare essentials of Python OOP in terms of its two object types. As you'll see, Python classes are in some ways similar to both defs and modules, but they may be quite different from what you're used to in other languages.

Class Objects Provide Default Behavior

When we run a class statement, we get a class object. Here's a rundown of the main properties of Python classes:

- The class statement creates a class object and assigns it a name. Just like the function def statement, the Python class statement is an executable statement.
- When reached and run, it generates a new class object and assigns it to the name in the class header. Also, like defs, class statements typically run when the files they are coded in are first imported.
- Assignments inside class statements make class attributes. Just like in module files, top-level assignments within a class statement (not nested in a def) generate attributes in a class object. Technically, the class statement defines a local scope that morphs into the attribute namespace of the class object, just like a module's global scope. After running a class statement, class attributes are accessed by name qualification: object.name.
- Class attributes provide object state and behavior. Attributes of a class object record state information and behavior to be shared by all instances created from the class; function def statements nested inside a class generate methods, which process instances.

Instance Objects Are Concrete Items

When we call a class object, we get an instance object. Here's an overview of the key points behind class instances:

- Calling a class object like a function makes a new instance object. Each time a class is called, it creates and returns a new instance object. Instances represent concrete items in your program's domain.
- Each instance object inherits class attributes and gets its own namespace. Instance objects created from classes are new namespaces; they start out empty but inherit attributes that live in the class objects from which they were generated.
- Assignments to attributes of self in methods make per-instance attributes. Inside a class's method functions, the first argument (called self by convention) references the instance object being processed; assignments to attributes of self create or change data in the instance, not the class.

The end result is that classes define common, shared data and behavior, and generate instances. Instances reflect concrete application entities, and record per-instance data that may vary per object.

Classes are customized by inheritance

Let's move on to the second major distinction of classes. Besides serving as factories for generating multiple instance objects, classes also allow us to make changes by introducing new components (called subclasses), instead of changing existing components in place.

As we've seen, instance objects generated from a class inherit the class's attributes.

Python also allows classes to inherit from other classes, opening the door to coding hierarchies of classes that specialize behavior—by redefining attributes in subclasses that appear lower in the hierarchy, we override the more general definitions of those attributes higher in the tree. In effect, the further down the hierarchy we go, the more specific the software becomes. Here, too, there is no parallel with modules, whose attributes live in a single, flat namespace that is not as amenable to customization.

In Python, instances inherit from classes, and classes inherit from superclasses. Here are the key ideas behind the machinery of attribute inheritance:

- Superclasses are listed in parentheses in a class header. To make a class inherit attributes from another class, just list the other class in parentheses in the new class statement's header line. The class that inherits is usually called a subclass, and the class that is inherited from is its superclass.
- Classes inherit attributes from their superclasses. Just as instances inherit the attribute names defined in their classes, classes inherit all of the attribute names defined in their superclasses; Python finds them automatically when they're accessed, if they don't exist in the subclasses.
- Instances inherit attributes from all accessible classes. Each instance gets names from the class it's generated from, as well as all of that class's superclasses. When looking for a name, Python checks the instance, then its class, then all superclasses.
- Each object.attribute reference invokes a new, independent search. Python performs an independent search of the class tree for each attribute fetch expression. This includes references to instances and classes made outside class statements (e.g., X.attr), as well as references to attributes of the self instance argument in a class's method functions. Each self.attr expression in a method invokes a new search for attr in self and above.
- Logic changes are made by subclassing, not by changing superclasses. By redefining superclass names in subclasses lower in the hierarchy (class tree), subclasses replace and thus customize inherited behavior.

The net effect—and the main purpose of all this searching—is that classes support factoring and customization of code better than any other language tool we've seen so far. On the one hand, they allow us to minimize code redundancy (and so reduce maintenance costs) by factoring operations into a single, shared implementation; on the other, they allow us to program by customizing what already exists, rather than changing it in place or starting from scratch.

Classes Are Attributes in Modules

Before we move on, remember that there's nothing magic about a class name. It's just a variable assigned to an object when the class statement runs, and the object can be referenced with any normal expression. For instance, if our FirstClass were coded in a module file instead of being typed interactively, we could import it and use its name normally in a class header line:

```
from modulename import FirstClass # Copy name into my scope
class SecondClass(FirstClass): # Use class name directly
def display(self): ...
Or, equivalently:
import modulename # Access the whole module
class SecondClass(modulename.FirstClass): # Qualify to reference
def display(self): ...
```

Like everything else, class names always live within a module, so they must follow all the rules we studied in Part V. For example, more than one class can be coded in a single module file—like other statements in a module, class statements are run during imports to define names, and these names become distinct module attributes. More generally, each module may arbitrarily mix any number of variables, functions, and classes, and all names in a module behave the same way. The file food.py demonstrates:

```
# food.py
var = 1 # food.var
def func(): ... # food.func
class spam: ... # food.spam
class ham: ... # food.ham
class eggs: ... # food.eggs
```

This holds true even if the module and class happen to have the same name. For example, given the following file, person.py:

```
class person: ...
```

we need to go through the module to fetch the class as usual:

```
import person # Import module
x = person.person() # Class within module
```

Although this path may look redundant, it's required: person.person refers to the per son class inside the person module. Saying just person gets the module, not the class, unless the from statement is used:

```
from person import person # Get class from module
x = person() # Use class name
```

As with any other variable, we can never see a class in a file without first importing and somehow fetching it from its enclosing file. If this seems confusing, don't use the same name for a module and a class within it. In fact, common convention in Python dictates that class names should begin with an uppercase letter, to help make them more distinct:

```
import person # Lowercase for modules
x = person.Person() # Uppercase for classes
```

Also, keep in mind that although classes and modules are both namespaces for attaching attributes, they correspond to very different source code structures: a module reflects an entire file, but a class is a statement within a file. We'll say more about such distinctions later in this part of the course.

Classes can intercept python operators

Let's move on to the third and final major difference between classes and modules:
operator overloading. In simple terms, operator overloading lets objects coded with classes intercept and respond to operations that work on built-in types: addition, slicing, printing, qualification, and so on. It's mostly just an automatic dispatch mechanism—expressions and other built-in operations route control to implementations in classes. Here, too, there is nothing similar in modules: modules can implement function calls, but not the behavior of expressions.

Although we could implement all class behavior as method functions, operator overloading lets objects be more tightly integrated with Python's object model. Moreover, because operator overloading makes our own objects act like built-ins, it tends to foster object interfaces that are more consistent and easier to learn, and it allows class-based objects to be processed by code written to expect a built-in type's interface. Here is a quick rundown of the main ideas behind overloading operators:

- Methods named with double underscores (__X__) are special hooks. In Python classes we implement operator overloading by providing specially named methods to intercept operations. The Python language defines a fixed and unchangeable mapping from each of these operations to a specially named method.
- Such methods are called automatically when instances appear in built-in operations. For instance, if an instance object inherits an __add__ method, that method is called whenever the object appears in a + expression. The method's return value becomes the result of the corresponding expression.
- Classes may override most built-in type operations. There are dozens of special operator overloading method names for intercepting and implementing nearly every operation available for built-in types. This includes expressions, but also basic operations like printing and object creation.
- There are no defaults for operator overloading methods, and none are required. If a class does not define or inherit an operator overloading method, it just means that the corresponding operation is not supported for the class's instances. If there is no __add__, for example, + expressions raise exceptions.
- New-style classes have some defaults, but not for common operations. In Python 3.X, and so-called "new style" classes in 2.X that we'll define later, a root class named object does provide defaults for some __X__ methods, but not for many, and not for most commonly used operations.
- Operators allow classes to integrate with Python's object model. By overloading type operations, the user-defined objects we implement with classes can act just like built-ins, and so provide consistency as well as compatibility with expected interfaces.

Operator overloading is an optional feature; it's used primarily by people developing tools for other Python programmers, not by application developers. And, candidly, you probably shouldn't use it just because it seems clever or "cool." Unless a class needs to mimic built-in type interfaces, it should usually stick to simpler named methods. Why would an employee database application support expressions like * and +, for example? Named methods like giveRaise and promote would usually make more sense.

Because of this, we won't go into details on every operator overloading method available in Python in this course. Still, there is one operator overloading method you are likely to see in almost every realistic Python class: the __init__ method, which is known as the constructor method and is used to initialize objects' state. You should pay special attention to this method, because __init__, along with the self argument, turns out to be a key requirement to reading and understanding most OOP code in Python.

Module 19 - Class Coding Details

If you haven't quite gotten all of Python OOP yet, don't worry; now that we've had a first tour, we're going to dig a bit deeper and study the concepts introduced earlier in further detail. We will take another look at class mechanics.

Here, we're going to study classes, methods, and inheritance, formalizing and expanding on some of the coding ideas introduced. Because the class is our last namespace tool, we'll summarize Python's namespace and scope concepts as well.

Following module continues this in-depth second pass over class mechanics by covering one specific aspect: operator overloading. Besides presenting additional details, this module and the next also give us an opportunity to explore some larger classes than those we have studied so far.

The class statement

Although the Python class statement may seem similar to tools in other OOP languages on the surface, on closer inspection, it is quite different from what some programmers are used to. For example, as in C++, the class statement is Python's main OOP tool, but unlike in C++, Python's class is not a declaration. Like a def, a class statement is an object builder, and an implicit assignment—when run, it generates a class object and stores a reference to it in the name used in the header. Also like a def, a class statement is true executable code—your class doesn't exist until Python reaches and runs the class statement that defines it. This typically occurs while importing the module it is coded in, but not before.

General Form

class is a compound statement, with a body of statements typically indented appearing under the header. In the header, superclasses are listed in parentheses after the class name, separated by commas. Listing more than one superclass leads to multiple inheritance.

Here is the statement's general form:

```
class name(superclass,...): # Assign to name
attr = value # Shared class data
def method(self,...): # Methods
self.attr = value # Per-instance data
```

Within the class statement, any assignments generate class attributes, and specially named methods overload operators; for instance, a function called __init__ is called at instance object construction time, if defined.

Methods

Because you already know about functions, you also know about methods in classes.

Methods are just function objects created by def statements nested in a class statement's body. From an abstract perspective, methods provide behavior for instance objects to inherit. From a programming perspective, methods work in exactly the same way as simple functions, with one crucial exception: a method's first argument always receives the instance object that is the implied subject of the method call.

In other words, Python automatically maps instance method calls to a class's method functions as follows. Method calls made through an instance, like this:

```
instance.method(args...)
```

are automatically translated to class method function calls of this form:

```
class.method(instance, args...)
```

where Python determines the class by locating the method name using the inheritance search procedure. In fact, both call forms are valid in Python.

Besides the normal inheritance of method attribute names, the special first argument is the only real magic behind method calls. In a class's method, the first argument is usually called self by convention (technically, only its position is significant, not its name). This argument provides methods with a hook back to the instance that is the subject of the call—because classes generate many instance objects, they need to use this argument to manage data that varies per instance.

C++ programmers may recognize Python's self argument as being similar to C++'s this pointer. In Python, though, self is always explicit in your code: methods must always go through self to fetch or change attributes of the instance being processed by the current method call. This explicit nature of self is by design—the presence of this name makes it obvious that you are using instance attribute names in your script, not names in the local or global scope.

Calling Superclass Constructors

Methods are normally called through instances. Calls to methods through a class, though, do show up in a variety of special roles. One common scenario involves the constructor method. The __init__ method, like all attributes, is looked up by inheritance.

This means that at construction time, Python locates and calls just one __init__. If subclass constructors need to guarantee that superclass construction-time logic runs, too, they generally must call the superclass's __init__ method explicitly through the class:

```
class Super:
  def __init__(self, x):
    ...default code...

class Sub(Super):
  def __init__(self, x, y):
    Super.__init__(self, x) # Run superclass __init__
    ...custom code... # Do my init actions

I = Sub(1, 2)
```

This is one of the few contexts in which your code is likely to call an operator overloading method directly. Naturally, you should call the superclass constructor this way only if you really want it to run—without the call, the subclass replaces it completely.

Other Method Call Possibilities

This pattern of calling methods through a class is the general basis of extending—instead of completely replacing—inherited method behavior. It requires an explicit instance to be passed because all methods do by default. Technically, this is because methods are instance methods in the absence of any special code.

There is newer option added in Python 2.2, static methods, that allow you to code methods that do not expect instance objects in their first arguments.

Such methods can act like simple instanceless functions, with names that are local to the classes in which they are coded, and may be used to manage class data. The class method, receives a class when called instead of an instance and can be used to manage per-class data, and is implied in metaclasses.

These are both advanced and usually optional extensions, though. Normally, an instance must always be passed to a method—whether automatically when it is called through an instance, or manually when you call through a class.

Inheritance
Of course, the whole point of the namespace created by the class statement is to support name inheritance. This section expands on some of the mechanisms and roles of attribute inheritance in Python.

As we've seen, in Python, inheritance happens when an object is qualified, and it involves searching an attribute definition tree—one or more namespaces. Every time you use an expression of the form object.attr where object is an instance or class object, Python searches the namespace tree from bottom to top, beginning with object, looking for the first attr it can find. This includes references to self attributes in your methods.

Because lower definitions in the tree override higher ones, inheritance forms the basis of specialization.

Attribute Tree Construction
Figure 29-1 summarizes the way namespace trees are constructed and populated with names. Generally:
- Instance attributes are generated by assignments to self attributes in methods.
- Class attributes are created by statements (assignments) in class statements.
- Superclass links are made by listing classes in parentheses in a class statement header.

The net result is a tree of attribute namespaces that leads from an instance, to the class it was generated from, to all the superclasses listed in the class header. Python searches upward in this tree, from instances to superclasses, each time you use qualification to fetch an attribute name from an instance object.

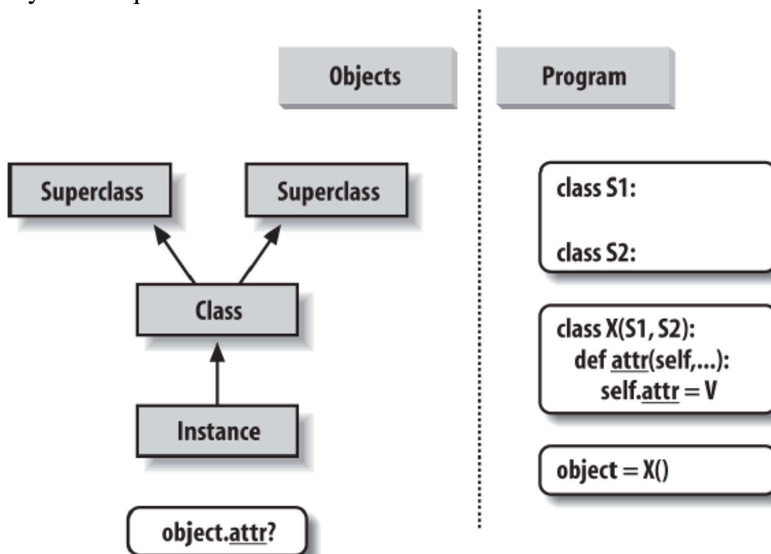


Figure 29-1. Program code creates a tree of objects in memory to be searched by attribute inheritance.

Calling a class creates a new instance that remembers its class, running a class statement creates a new class, and superclasses are listed in parentheses in the class statement header. Each attribute reference triggers a new bottom-up tree search—even references to self attributes within a class's methods.

Specializing Inherited Methods
The tree-searching model of inheritance just described turns out to be a great way to specialize systems. Because inheritance finds names in subclasses before it checks superclasses, subclasses can replace default behavior by redefining their superclasses'. Two fine points here: first, this description isn't 100% complete, because we can also create instance and class attributes by assigning them to objects outside class statements—but that's a much less common and sometimes more error-prone approach (changes aren't isolated to class statements). In Python, all attributes are always accessible by default.

In common usage, though, it's simply a way to redefine, and hence customize, behavior coded in classes.

In fact, you can build entire systems as hierarchies of classes, which you extend by adding new external subclasses rather than changing existing logic in place.

The idea of redefining inherited names leads to a variety of specialization techniques.

For instance, subclasses may replace inherited attributes completely, provide attributes that a superclass expects to find, and extend superclass methods by calling back to the superclass from an overridden method. We've already seen some of these patterns in action; here's a self-contained example of extension at work:

```
>>> class Super:
def method(self):
print('in Super.method')
>>> class Sub(Super):
def method(self): # Override method
print('starting Sub.method') # Add actions here
Super.method(self) # Run default action
print('ending Sub.method')
```

Direct superclass method calls are the crux of the matter here. The Sub class replaces Super's method function with its own specialized version, but within the replacement, Sub calls back to the version exported by Super to carry out the default behavior. In other words, Sub.method just extends Super.method's behavior, rather than replacing it completely:

```
>>> x = Super() # Make a Super instance
>>> x.method() # Runs Super.method
in Super.method
>>> x = Sub() # Make a Sub instance
>>> x.method() # Runs Sub.method, calls Super.method
starting Sub.method
in Super.method
ending Sub.method.
```

Abstract Superclasses

Of the prior example's classes, Provider may be the most crucial to understand. When we call the delegate method through a Provider instance, two independent inheritance searches occur:

1. On the initial x.delegate call, Python finds the delegate method in Super by searching the Provider instance and above. The instance x is passed into the method's self argument as usual.
2. Inside the Super.delegate method, self.action invokes a new, independent inheritance search of self and above. Because self references a Provider instance, the action method is located in the Provider subclass.

This "filling in the blanks" sort of coding structure is typical of OOP frameworks. In a more realistic context, the method filled in this way might handle an event in a GUI, provide data to be rendered as part of a web page, process a tag's text in an XML file, and so on—your subclass provides specific actions, but the framework handles the rest of the overall job.

At least in terms of the delegate method, the superclass in this example is what is sometimes called an abstract superclass—a class that expects parts of its behavior to be provided by its subclasses. If an expected method is not defined in a subclass, Python raises an undefined name exception when the inheritance search fails.

Class coders sometimes make such subclass requirements more obvious with assert statements, or by raising the built-in NotImplementedError exception with raise statements.

We'll study statements that may trigger exceptions in depth in the next part of this course; as a quick preview, here's the assert scheme in action:

```
class Super:
  def delegate(self):
```

```
    self.action()
  def action(self):
    assert False, 'action must be defined!' # If this version is called
>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
```

assert in short, if its first expression evaluates to false, it raises an exception with the provided error message. Here, the expression is always false so as to trigger an error message if a method is not redefined, and inheritance locates the version here. Alternatively, some classes simply raise a NotImplementedError exception directly in such method stubs to signal the mistake:

```
class Super:
  def delegate(self):
    self.action()
  def action(self):
    raise NotImplementedError('action must be defined!')
>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
```

For instances of subclasses, we still get the exception unless the subclass provides the expected method to replace the default in the superclass:

```
>>> class Sub(Super): pass
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!
>>> class Sub(Super):
def action(self): print('spam')
>>> X = Sub()
>>> X.delegate()
spam
```

Classes Versus Modules

Finally, let's wrap up this module by briefly comparing the topics of this course's last two parts: modules and classes. Because they're both about namespaces, the distinction can be confusing. In short:

- Modules
    - — Implement data/logic packages
    - — Are created with Python files or other-language extensions
    - — Are used by being imported
    - — Form the top-level in Python program structure
- Classes
    - — Implement new full-featured objects
    - — Are created with class statements
    - — Are used by being called
    - — Always live within a module

Classes also support extra features that modules don't, such as operator overloading, multiple instance generation, and inheritance. Although both classes and modules are namespaces, you should be able to tell by now that they are very different things. We need to move ahead to see just how different classes can be.

Operator overloading

Really "operator overloading" simply means intercepting built-in operations in a class's methods—Python automatically invokes your methods when instances of the class appear in built-in operations, and your method's return value becomes the result of the corresponding operation. Here's a review of the key ideas behind overloading:

- Operator overloading lets classes intercept normal Python operations.
- Classes can overload all Python expression operators.
- Classes can also overload built-in operations such as printing, function calls, attribute access, etc.
- Overloading makes class instances act more like built-in types.
- Overloading is implemented by providing specially named methods in a class.

In other words, when certain specially named methods are provided in a class, Python automatically calls them when instances of the class appear in their associated expressions.

Your class provides the behavior of the corresponding operation for instance objects created from it.

As we've learned, operator overloading methods are never required and generally don't have defaults (apart from a handful that some classes get from object); if you don't code or inherit one, it just means that your class does not support the corresponding operation. When used, though, these methods allow classes to emulate the interfaces of built-in objects, and so appear more consistent.

Constructors and Expressions: __init__ and __sub__

As a review, consider the following simple example: its Number class, coded in the file number.py, provides a method to intercept instance construction (__init__), as well as one for catching subtraction expressions (__sub__). Special methods such as these are the hooks that let you tie into built-in operations:

```
# File number.py
class Number:
  def __init__(self, start): # On Number(start)
    self.data = start
  def __sub__(self, other): # On instance - other
    return Number(self.data - other) # Result is a new instance
>>> from number import Number # Fetch class from module
>>> X = Number(5) # Number.__init__(X, 5)
>>> Y = X - 2 # Number.__sub__(X, 2)
>>> Y.data # Y is new Number instance
3
```

As we've already learned, the __init__ constructor method seen in this code is the most commonly used operator overloading method in Python; it's present in most classes, and used to initialize the newly created instance object using any arguments passed to the class name. The __sub__ method plays the binary operator role that __add__ did , intercepting subtraction expressions and returning a new instance of the class as its result (and running __init__ along the way).

We've already studied __init__ and basic binary operators like __sub__ in some depth, so we won't rehash their usage further here. In this module, we will tour some of the other tools available in this domain and look at example code that applies them in common use cases.

Common Operator Overloading Methods

Just about everything you can do to built-in objects such as integers and lists has a corresponding specially named method for overloading in classes. Table 30-1 lists a few of the most common; there are many more. In fact, many overloading methods come in multiple versions (e.g., __add__, __radd__, and __iadd__ for addition), which is one reason there are so many. See other Python books, or the Python language reference manual, for an exhaustive list of the special method names available.

Table 30-1: *Common operator overloading methods*

Method	Implements Called for
__init__	Constructor Object creation: X = Class(args)
__del__	Destructor Object reclamation of X
__add__	Operator + X + Y, X += Y if no __iadd__
__or__	Operator | (bitwise OR) X | Y, X |= Y if no __ior__
__repr__, __str__	Printing, conversions print(X), repr(X), str(X)
__call__	Function calls X(*args, **kargs)
__getattr__	Attribute fetch X.undefined
__setattr__	Attribute assignment X.any = value
__delattr__	Attribute deletion del X.any
__getattribute__	Attribute fetch X.any
__getitem__ __iter__	Indexing, slicing, iteration X[key], X[i:j], for loops and other iterations if no
__setitem__	Index and slice assignment X[key] = value, X[i:j] = iterable
__delitem__	Index and slice deletion del X[key], del X[i:j]
__len__	Length len(X), truth tests if no __bool__
__bool__	Boolean tests bool(X), truth tests (named __nonzero__ in 2.X)
__lt__, __gt__, __le__, __ge__, __eq__, __ne__	Comparisons X < Y, X > Y, X <= Y, X >= Y, X == Y, X != Y (or else __cmp__ in 2.X only)
__radd__	Right-side operators Other + X
__iadd__	In-place augmented operators X += Y (or else __add__)
__iter__, __next__	Iteration contexts I=iter(X), next(I); for loops, in if no __con tains__, all comprehensions, map(F,X), others (__next__ is named next in 2.X)
__contains__	Membership test item in X (any iterable)
__index__	Integer value hex(X), bin(X), oct(X), O[X], O[X:] (replaces 2.X __oct__, __hex__)
__enter__, __exit__	Context manager with obj as var
__get__, __set__, __delete__	Descriptor attributes X.attr, X.attr = value, del X.attr
__new__	Creation Object creation, before __init__

All overloading methods have names that start and end with two underscores to keep them distinct from other names you define in your classes. The mappings from special method names to expressions or operations are predefined by the Python language, and documented in full in the standard language manual and other reference resources.

For example, the name __add__ always maps to + expressions by Python language definition, regardless of what an __add__ method's code actually does.

Operator overloading methods may be inherited from superclasses if not defined, just like any other methods. Operator overloading methods are also all optional—if you don't code or inherit one, that operation is simply unsupported by your class, and attempting it will raise an exception. Some built-in operations, like printing, have defaults (inherited from the implied object class in Python 3.X), but most built-ins fail for class instances if no corresponding operator overloading method is present.

Most overloading methods are used only in advanced programs that require objects to behave like built-ins, though the __init__ constructor we've already met tends to appear in most classes. Let's explore some of the additional methods in Table 30-1 by example.

Module 20 - Designing with Classes

So far in this part of the course, we've concentrated on using Python's OOP tool, the class. But OOP is also about design issues—that is, how to use classes to model useful objects. This module will touch on a few core OOP ideas and present some additional examples that are more realistic than many shown so far.

Along the way, we'll code some common OOP design patterns in Python, such as inheritance, composition, delegation, and factories. We'll also investigate some designfocused class concepts, such as pseudoprivate attributes, multiple inheritance, and bound methods.

One note up front: some of the design terms mentioned here require more explanation than I can provide in this course. If this material sparks your curiosity, I suggest exploring a text on OOP design or design patterns as a next step. As we'll see, the good news is that Python makes many traditional design patterns trivial.

Python and OOP

Let's begin with a review—Python's implementation of OOP can be summarized by three ideas:

- **Inheritance -** Inheritance is based on attribute lookup in Python (in X.name expressions).
- **Polymorphism -** In X.method, the meaning of method depends on the type (class) of subject object X.
- **Encapsulation -** Methods and operators implement behavior, though data hiding is a convention by default.

By now, you should have a good feel for what inheritance is all about in Python. We've also talked about Python's polymorphism a few times already; it flows from Python's lack of type declarations. Because attributes are always resolved at runtime, objects that implement the same interfaces are automatically interchangeable; clients don't need to know what sorts of objects are implementing the methods they call.

Encapsulation means packaging in Python—that is, hiding implementation details behind an object's interface. It does not mean enforced privacy, though that can be implemented with code. Encapsulation is available and useful in Python nonetheless: it allows the implementation of an object's interface to be changed without impacting the users of that object.

Polymorphism Means Interfaces, Not Call Signatures

Some OOP languages also define polymorphism to mean overloading functions based on the type signatures of their arguments—the number passed and/or their types. Because there are no type declarations in Python, this concept doesn't really apply; as we've seen, polymorphism in Python is based on object interfaces, not types.

If you're pining for your C++ days, you can try to overload methods by their argument lists, like this:

```
class C:
  def meth(self, x):
    ...
  def meth(self, x, y, z):
    ...
```

This code will run, but because the def simply assigns an object to a name in the class's scope, the last definition of the method function is the only one that will be retained.

Put another way, it's just as if you say X = 1 and then X = 2; X will be 2. Hence, there can be only one definition of a method name.

If they are truly required, you can always code type-based selections using the typetesting, or the argument list tools:

```
class C:
def meth(self, *args):
if len(args) == 1: # Branch on number arguments
...
elif type(arg[0]) == int: # Branch on argument types (or isinstance())
...
```

You normally shouldn't do this, though—it's not the Python way. You should write your code to expect only an object interface, not a specific data type. That way, it will be useful for a broader category of types and applications, both now and in the future:

```
class C:
def meth(self, x):
x.operation() # Assume x does the right thing
```

It's also generally considered better to use distinct method names for distinct operations, rather than relying on call signatures (no matter what language you code in).

Although Python's object model is straightforward, much of the art in OOP is in the way we combine classes to achieve a program's goals. The next section begins a tour of some of the ways larger programs use classes to their advantage.

OOP and inheritance: "is-a" relationships

We've explored the mechanics of inheritance in depth already, but I'd now like to show you an example of how it can be used to model real-world relationships. From a programmer's point of view, inheritance is kicked off by attribute qualifications, which trigger searches for names in instances, their classes, and then any superclasses. From a designer's point of view, inheritance is a way to specify set membership: a class defines a set of properties that may be inherited and customized by more specific sets (i.e., subclasses).

To illustrate, let's put that pizza-making robot we talked about at the start of this part of the course to work. Suppose we've decided to explore alternative career paths and open a pizza restaurant (not bad, as career paths go). One of the first things we'll need to do is hire employees to serve customers, prepare the food, and so on. Being engineers at heart, we've decided to build a robot to make the pizzas; but being politically and cybernetically correct, we've also decided to make our robot a full-fledged employee with a salary.

Our pizza shop team can be defined by the four classes in the following Python 3.X and 2.X example file, employees.py. The most general class, Employee, provides common behavior such as bumping up salaries (giveRaise) and printing (__repr__). There are two kinds of employees, and so two subclasses of Employee—Chef and Server. Both override the inherited work method to print more specific messages. Finally, our pizza robot is modeled by an even more specific class—PizzaRobot is a kind of Chef, which is a kind of Employee. In OOP terms, we call these relationships "is-a" links: a robot is a chef, which is an employee. Here's the employees.py file:

```
# File employees.py (2.X + 3.X)
from __future__ import print_function
class Employee:
  def __init__(self, name, salary=0):
    self.name = name
    self.salary = salary
  def giveRaise(self, percent):
    self.salary = self.salary + (self.salary * percent)
  def work(self):
    print(self.name, "does stuff")
  def __repr__(self):
    return "<Employee: name=%s, salary=%s>" % (self.name, self.salary)

class Chef(Employee):
  def __init__(self, name):
    Employee.__init__(self, name, 50000)
  def work(self):
    print(self.name, "makes food")

class Server(Employee):
  def __init__(self, name):
```

```
    Employee.__init__(self, name, 40000)
  def work(self):
    print(self.name, "interfaces with customer")

class PizzaRobot(Chef):
  def __init__(self, name):
    Chef.__init__(self, name)
  def work(self):
    print(self.name, "makes pizza")
    if __name__ == "__main__":
      bob = PizzaRobot('bob') # Make a robot named bob
      print(bob) # Run inherited __repr__
      bob.work() # Run type-specific action
      bob.giveRaise(0.20) # Give bob a 20% raise
      print(bob); print()
      for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__)
        obj.work()
```

When we run the self-test code included in this module, we create a pizza-making robot named bob, which inherits names from three classes: PizzaRobot, Chef, and Employee.

For instance, printing bob runs the Employee.__repr__ method, and giving bob a raise invokes Employee.giveRaise because that’s where the inheritance search finds that method:

```
c:\code> python employees.py
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000.0>
Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

In a class hierarchy like this, you can usually make instances of any of the classes, not just the ones at the bottom.

For instance, the for loop in this module’s self-test code creates instances of all four classes; each responds differently when asked to work because the work method is different in each. bob the robot, for example, gets work from the most specific (i.e., lowest) PizzaRobot class.

Of course, these classes just simulate real-world objects; work prints a message for the time being, but it could be expanded to do real work later (see Python’s interfaces to devices such as serial ports, Arduino boards, and the Raspberry Pi if you’re taking this section much too literally!).

OOP and composition: "has-a" relationships

The notion of composition from a programmer’s point of view, composition involves embedding other objects in a container object, and activating them to implement container methods. To a designer, composition is another way to represent relationships in a problem domain. But, rather than set membership, composition has to do with components—parts of a whole.

Composition also reflects the relationships between parts, called “has-a” relationships.

Some OOP design texts refer to composition as aggregation, or distinguish between the two terms by using aggregation to describe a weaker dependency between container and contained. In this text, a “composition” simply

refers to a collection of embedded objects. The composite class generally provides an interface all its own and implements it by directing the embedded objects.

Now that we've implemented our employees, let's put them in the pizza shop and let them get busy. Our pizza shop is a composite object: it has an oven, and it has employees like servers and chefs. When a customer enters and places an order, the components of the shop spring into action—the server takes the order, the chef makes the pizza, and so on. The following example—file pizzashop.py—runs the same on Python 3.X and 2.X and simulates all the objects and relationships in this scenario:

```
# File pizzashop.py (2.X + 3.X)
from __future__ import print_function
from employees import PizzaRobot, Server
class Customer:
  def __init__(self, name):
    self.name = name
  def order(self, server):
    print(self.name, "orders from", server)
  def pay(self, server):
    print(self.name, "pays for item to", server)

class Oven:
  def bake(self):
    print("oven bakes")

class PizzaShop:
  def __init__(self):
    self.server = Server('Pat') # Embed other objects
    self.chef = PizzaRobot('Bob') # A robot named bob
    self.oven = Oven()
  def order(self, name):
    customer = Customer(name) # Activate other objects
    customer.order(self.server) # Customer orders from server
    self.chef.work()
    self.oven.bake()
    customer.pay(self.server)
    if __name__ == "__main__":
      scene = PizzaShop() # Make the composite
      scene.order('Homer') # Simulate Homer's order
      print('...')
      scene.order('Shaggy') # Simulate Shaggy's order
```

The PizzaShop class is a container and controller; its constructor makes and embeds instances of the employee classes we wrote in the prior section, as well as an Oven class defined here. When this module's self-test code calls the PizzaShop order method, the embedded objects are asked to carry out their actions in turn. Notice that we make a new Customer object for each order, and we pass on the embedded Server object to Customer methods; customers come and go, but the server is part of the pizza shop composite. Also notice that employees are still involved in an inheritance relationship; composition and inheritance are complementary tools.

When we run this module, our pizza shop handles two orders—one from Homer, and then one from Shaggy:

```
c:\code> python pizzashop.py
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
```

```
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

Again, this is mostly just a toy simulation, but the objects and interactions are representative of composites at work. As a rule of thumb, classes can represent just about any objects and relationships you can express in a sentence; just replace nouns with classes (e.g., Oven), and verbs with methods (e.g., bake), and you'll have a first cut at a design.

OOP and delegation

Beside inheritance and composition, object-oriented programmers often speak of delegation, which usually implies controller objects that embed other objects to which they pass off operation requests. The controllers can take care of administrative activities, such as logging or validating accesses, adding extra steps to interface components, or monitoring active instances.

In a sense, delegation is a special form of composition, with a single embedded object managed by a wrapper (sometimes called a proxy) class that retains most or all of the embedded object's interface. The notion of proxies sometimes applies to other mechanisms too, such as function calls; in delegation, we're concerned with proxies for all of an object's behavior, including method calls and other operations.

This concept in Python is often implemented with the __getattr__ method hook. Because this operator overloading method intercepts accesses to nonexistent attributes, a wrapper class can use __getattr__ to route arbitrary accesses to a wrapped object. Because this method allows attribute requests to be routed generically, the wrapper class retains the interface of the wrapped object and may add additional operations of its own.

By way of review, consider the file trace.py (which runs the same in 2.X and 3.X):

```
class Wrapper:
def __init__(self, object):
self.wrapped = object # Save object
def __getattr__(self, attrname):
print('Trace: ' + attrname) # Trace fetch
return getattr(self.wrapped, attrname) # Delegate fetch
```

__getattr__ gets the attribute name as a string. This code makes use of the getattr built-in function to fetch an attribute from the wrapped object by name string—getattr(X,N) is like X.N, except that N is an expression that evaluates to a string at runtime, not a variable. In fact, getattr(X,N) is similar to X.__dict__[N], but the former also performs an inheritance search, like X.N, while the latter does not.

You can use the approach of this module's wrapper class to manage access to any object with attributes—lists, dictionaries, and even classes and instances. Here, the Wrapper class simply prints a trace message on each attribute access and delegates the attribute request to the embedded wrapped object:

```
>>> from trace import Wrapper
>>> x = Wrapper([1, 2, 3]) # Wrap a list
>>> x.append(4) # Delegate to list method
Trace: append
>>> x.wrapped # Print my member
[1, 2, 3, 4]
>>> x = Wrapper({'a': 1, 'b': 2}) # Wrap a dictionary
>>> list(x.keys()) # Delegate to dictionary method
Trace: keys
['a', 'b']
```

The net effect is to augment the entire interface of the wrapped object, with additional code in the Wrapper class. We can use this to log our method calls, route method calls to extra or custom logic, adapt a class to a new interface, and so on.

We'll revive the notions of wrapped objects and delegated operations as one way to extend built-in types. If you are interested in the delegation design pattern, also watch for function decorators, a strongly related concept designed to augment a specific function or method call rather than the entire interface of an object, and class decorators, which serve as a way to automatically add such delegation-based wrappers to all instances of a class.

Multiple inheritance

Our last design pattern is one of the most useful, and will serve as a subject for a more realistic example to wrap up this module and point toward the next. As a bonus, the code we'll write here may be a useful tool.

Many class-based designs call for combining disparate sets of methods. As we've seen, in a class statement, more than one superclass can be listed in parentheses in the header line. When you do this, you leverage multiple inheritance—the class and its instances inherit names from all the listed superclasses.

When searching for an attribute, Python's inheritance search traverses all superclasses in the class header from left to right until a match is found. Technically, because any of the superclasses may have superclasses of its own, this search can be a bit more complex for larger class trees:

- In classic classes (the default until Python 3.0), the attribute search in all cases proceeds depth-first all the way to the top of the inheritance tree, and then from left to right. This order is usually called DFLR, for its depth-first, left-to-right path.
- In new-style classes (optional in 2.X and standard in 3.X), the attribute search is usually as before, but in diamond patterns proceeds across by tree levels before moving up, in a more breadth-first fashion. This order is usually called the new-style MRO, for method resolution order, though it's used for all attributes, not just methods.

In general, multiple inheritance is good for modeling objects that belong to more than one set. For instance, a person may be an engineer, a writer, a musician, and so on, and inherit properties from all such sets. With multiple inheritance, objects obtain the union of the behavior in all their superclasses. As we'll see ahead, multiple inheritance also allows classes to function as general packages of mixable attributes.

Though a useful pattern, multiple inheritance's chief downside is that it can pose a conflict when the same method (or other attribute) name is defined in more than one superclass. When this occurs, the conflict is resolved either automatically by the inheritance search order, or manually in your code:

- Default: By default, inheritance chooses the first occurrence of an attribute it finds when an attribute is referenced normally—by self.method(), for example. In this mode, Python chooses the lowest and leftmost in classic classes, and in nondiamond patterns in all classes; new-style classes may choose an option to the right before one above in diamonds.
- Explicit: In some class models, you may sometimes need to select an attribute explicitly by referencing it through its class name—with superclass.method(self), for instance. Your code breaks the conflict and overrides the search's default—to select an option to the right of or above the inheritance search's default.

This is an issue only when the same name appears in multiple superclasses, and you do not wish to use the first one inherited. Because this isn't as common an issue in typical Python code as it may sound, we'll defer details on this topic until we study new-style classes and their MRO and super tools, and revisit this as a "gotcha". First, though, the next section demonstrates a practical use case for multiple inheritance-based tools.

Classes are objects: Generic object factories

Sometimes, class-based designs require objects to be created in response to conditions that can't be predicted when a program is written. The factory design pattern allows such a deferred approach. Due in large part to Python's flexibility, factories can take multiple forms, some of which don't seem special at all.

Because classes are also “first class” objects, it’s easy to pass them around a program, store them in data structures, and so on. You can also pass classes to functions that generate arbitrary kinds of objects; such functions are sometimes called factories in OOP design circles. Factories can be a major undertaking in a strongly typed language
such as C++ but are almost trivial to implement in Python.

For example, the call syntax we met can call any class with any number of positional or keyword constructor arguments in one step to generate any sort of instance:

```
def factory(aClass, *pargs, **kargs): # Varargs tuple, dict
  return aClass(*pargs, **kargs) # Call aClass (or apply in 2.X only)

class Spam:
  def doit(self, message):
    print(message)

class Person:
  def __init__(self, name, job=None):
    self.name = name
    self.job = job
    object1 = factory(Spam) # Make a Spam object
    object2 = factory(Person, "Arthur", "King") # Make a Person object
    object3 = factory(Person, name='Brian') # Ditto, with keywords and default
```

In this code, we define an object generator function called factory. It expects to be passed a class object (any class will do) along with one or more arguments for the class’s constructor. The function uses special “varargs” call syntax to call the function and return an instance.

The rest of the example simply defines two classes and generates instances of both by passing them to the factory function. And that’s the only factory function you’ll ever need to write in Python; it works for any class and any constructor arguments. If you run this live (factory.py), your objects will look like this:

```
>>> object1.doit(99)
99
>>> object2.name, object2.job
('Arthur', 'King')
>>> object3.name, object3.job
('Brian', None)
```

By now, you should know that everything is a “first class” object in Python—including classes, which are usually just compiler input in languages like C++. It’s natural to pass them around this way. As mentioned at the start of this part of the course, though, only objects derived from classes do full OOP in Python.

Methods are objects: Bound or unbound

Methods in general, and bound methods in particular, simplify the implementation of many design goals in Python. We met bound methods briefly while studying __call__. The full story, which we’ll flesh out here, turns out to be more general and flexible than you might expect.

We learned how functions can be processed as normal objects. Methods are a kind of object too, and can be used generically in much the same way as other objects—they can be assigned to names, passed to functions, stored in data structures, and so on—and like simple functions, qualify as “first class” objects. Because a class’s methods can be accessed from an instance or a class, though, they actually come in two flavors in Python:

- Unbound (class) method
- objects: no self

Accessing a function attribute of a class by qualifying the class returns an unbound method object. To call the method, you must provide an instance object explicitly as the first argument. In Python 3.X, an unbound method is the same

as a simple function and can be called through the class's name; in 2.X it's a distinct type and cannot be called without providing an instance.

Bound (instance) method objects: self + function pairs Accessing a function attribute of a class by qualifying an instance returns a bound method object. Python automatically packages the instance with the function in the bound method object, so you don't need to pass an instance to call the method.

Both kinds of methods are full-fledged objects; they can be transferred around a program at will, just like strings and numbers. Both also require an instance in their first argument when run (i.e., a value for self). This is why we've had to pass in an instance explicitly when calling superclass methods from subclass methods in previous examples (including this module's employees.py); technically, such calls produce unbound method objects along the way.

When calling a bound method object, Python provides an instance for you automatically—the instance used to create the bound method object. This means that bound method objects are usually interchangeable with simple function objects, and makes them especially useful for interfaces originally written for functions.

To illustrate in simple terms, suppose we define the following class:

```
class Spam:
  def doit(self, message):
    print(message)
```

Now, in normal operation, we make an instance and call its method in a single step to print the passed-in argument:

```
object1 = Spam()
object1.doit('hello world')
```

Really, though, a bound method object is generated along the way, just before the method call's parentheses. In fact, we can fetch a bound method without actually calling it. An object.name expression evaluates to an object as all expressions do. In the following, it returns a bound method object that packages the instance (object1) with the method function (Spam.doit). We can assign this bound method pair to another name and then call it as though it were a simple function:

```
object1 = Spam()
x = object1.doit # Bound method object: instance+function
x('hello world') # Same effect as object1.doit('...')
```

On the other hand, if we qualify the class to get to doit, we get back an unbound method object, which is simply a reference to the function object. To call this type of method, we must pass in an instance as the leftmost argument—there isn't one in the expression otherwise, and the method expects it:

```
object1 = Spam()
t = Spam.doit # Unbound method object (a function in 3.X: see ahead)
t(object1, 'howdy') # Pass in instance (if the method expects one in 3.X)
```

By extension, the same rules apply within a class's method if we reference self attributes that refer to functions in the class. A self.method expression is a bound method object because self is an instance object:

```
class Eggs:
  def m1(self, n):
    print(n)
  def m2(self):
    x = self.m1 # Another bound method object
    x(42) # Looks like a simple function

Eggs().m2() # Prints 42
```

Most of the time, you call methods immediately after fetching them with attribute qualification, so you don't always notice the method objects generated along the way.

But if you start writing code that calls objects generically, you need to be careful to treat unbound methods specially—they normally require an explicit instance object to be passed in.

Classes versus modules

Finally, let's wrap up this module by briefly comparing the topics of this course's last two parts: modules and classes. Because they're both about namespaces, the distinction can be confusing. In short:

- Modules
    - Implement data/logic packages
    - Are created with Python files or other-language extensions
    - Are used by being imported
    - Form the top-level in Python program structure
- Classes
    - Implement new full-featured objects
    - Are created with class statements
    - Are used by being called
    - Always live within a module

Classes also support extra features that modules don't, such as operator overloading, multiple instance generation, and inheritance. Although both classes and modules are namespaces, you should be able to tell by now that they are very different things. We need to move ahead to see just how different classes can be.

Module 21 - Exception Basics

This part of the course deals with exceptions, which are events that can modify the flow of control through a program. In Python, exceptions are triggered automatically on errors, and they can be triggered and intercepted by your code. They are processed by four statements we'll study in this part, the first of which has two variations (listed separately here) and the last of which was an optional extension until Python 2.6 and 3.0:

try/except	Catch and recover from exceptions raised by Python, or by you.
try/finally	Perform cleanup actions, whether exceptions occur or not.
raise	Trigger an exception manually in your code.
assert	Conditionally trigger an exception in your code.
with/as	Implement context managers in Python 2.6, 3.0, and later (optional in 2.5).

This topic was saved until nearly the end of the course because you need to know about classes to code exceptions of your own. With a few exceptions (pun intended), though, you'll find that exception handling is simple in Python because it's integrated into the language itself as another high-level tool.

Why use exceptions?

In a nutshell, exceptions let us jump out of arbitrarily large chunks of a program. Consider the hypothetical pizza-making robot we discussed earlier in the course. Suppose we took the idea seriously and actually built such a machine. To make a pizza, our culinary automaton would need to execute a plan, which we would implement as a Python program: it would take an order, prepare the dough, add toppings, bake the pie, and so on.

Now, suppose that something goes very wrong during the "bake the pie" step. Perhaps the oven is broken, or perhaps our robot miscalculates its reach and spontaneously combusts. Clearly, we want to be able to jump to code that handles such states quickly.

As we have no hope of finishing the pizza task in such unusual cases, we might as well abandon the entire plan.

That's exactly what exceptions let you do: you can jump to an exception handler in a single step, abandoning all function calls begun since the exception handler was entered.

Code in the exception handler can then respond to the raised exception as appropriate (by calling the fire department, for instance!).

One way to think of an exception is as a sort of structured "super go to." An exception handler (try statement) leaves a marker and executes some code. Somewhere further ahead in the program, an exception is raised that makes Python jump back to that marker, abandoning any active functions that were called after the marker was left.

This protocol provides a coherent way to respond to unusual events. Moreover, because Python jumps to the handler statement immediately, your code is simpler—there is usually no need to check status codes after every call to a function that could possibly fail.

Exception handling: The short story

Compared to some other core language topics we've met in this course, exceptions are a fairly lightweight tool in Python. Because they are so simple, let's jump right into some code.

Default Exception Handler

Suppose we write the following function:

```
>>> def fetcher(obj, index):
        return obj[index]
```

There's not much to this function—it simply indexes an object on a passed-in index. In normal operation, it returns the result of a legal index:

```
>>> x = 'spam'
```

```
>>> fetcher(x, 3) # Like x[3]
'm'
```

However, if we ask this function to index off the end of the string, an exception will be triggered when the function tries to run obj[index]. Python detects out-of-bounds indexing for sequences and reports it by raising (triggering) the built-in IndexError exception:

```
>>> fetcher(x, 4) # Default handler - shell interface
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in fetcher
IndexError: string index out of range
```

Because our code does not explicitly catch this exception, it filters back up to the top level of the program and invokes the default exception handler, which simply prints the standard error message. By this point in the course, you've probably seen your share of standard error messages. They include the exception that was raised, along with a stack trace—a list of all the lines and functions that were active when the exception occurred.

The error message text here was printed by Python 3.3; it can vary slightly per release, and even per interactive shell, so you shouldn't rely upon its exact form—in either this course or your code. When you're coding interactively in the basic shell interface, the filename is just "<stdin>," meaning the standard input stream.

When working in the IDLE GUI's interactive shell, the filename is "<pyshell>," and source lines are displayed, too. Either way, file line numbers are not very meaningful when there is no file:

```
>>> fetcher(x, 4) # Default handler - IDLE GUI interface
Traceback (most recent call last):
File "<pyshell#6>", line 1, in <module>
fetcher(x, 4)
File "<pyshell#3>", line 2, in fetcher
return obj[index]
IndexError: string index out of range
```

In a more realistic program launched outside the interactive prompt, after printing an error message the default handler at the top also terminates the program immediately.

That course of action makes sense for simple scripts; errors often should be fatal, and the best you can do when they occur is inspect the standard error message.

Catching Exceptions

Sometimes, this isn't what you want, though. Server programs, for instance, typically need to remain active even after internal errors. If you don't want the default exception behavior, wrap the call in a try statement to catch exceptions yourself:

```
>>> try:
... fetcher(x, 4)
... except IndexError: # Catch and recover
... print('got exception')
...
got exception
>>>
```

Now, Python jumps to your handler—the block under the except clause that names the exception raised—automatically when an exception is triggered while the try block is running. The net effect is to wrap a nested block of code in an error handler that intercepts the block's exceptions.

When working interactively like this, after the except clause runs, we wind up back at the Python prompt. In a more realistic program, try statements not only catch exceptions, but also recover from them:

```
>>> def catcher():
try:
fetcher(x, 4)
except IndexError:
print('got exception')
print('continuing')
>>> catcher()
got exception
continuing
>>>
```

This time, after the exception is caught and handled, the program resumes execution after the entire try statement that caught it—which is why we get the "continuing" message here. We don't see the standard error message, and the program continues on its way normally.

Notice that there's no way in Python to go back to the code that triggered the exception (short of rerunning the code that reached that point all over again, of course). Once you've caught the exception, control continues after the entire try that caught the exception, not after the statement that kicked it off. In fact, Python clears the memory of any functions that were exited as a result of the exception, like fetcher in our example; they're not resumable. The try both catches exceptions, and is where the program resumes.

Raising Exceptions

So far, we've been letting Python raise exceptions for us by making mistakes (on purpose this time!), but our scripts can raise exceptions too—that is, exceptions can be raised by Python or by your program, and can be caught or not. To trigger an exception manually, simply run a raise statement. User-triggered exceptions are caught the same way as those Python raises. The following may not be the most useful Python code ever penned, but it makes the point—raising the built-in IndexError exception:

```
>>> try:
... raise IndexError # Trigger exception manually
... except IndexError:
... print('got exception')
...
got exception
```

As usual, if they're not caught, user-triggered exceptions are propagated up to the toplevel default exception handler and terminate the program with a standard error message:

```
>>> raise IndexError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError
```

As we'll see in the later, the assert statement can be used to trigger exceptions, too—it's a conditional raise, used mostly for debugging purposes during development:

```
>>> assert False, 'Nobody expects the Spanish Inquisition!'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AssertionError: Nobody expects the Spanish Inquisition!
```

User-Defined Exceptions

The raise statement introduced in the prior section raises a built-in exception defined in Python's built-in scope. As you'll learn later in this part of the course, you can also define new exceptions of your own that are specific to your programs. User-defined exceptions are coded with classes, which inherit from a built-in exception class: usually the class named Exception:

```
>>> class AlreadyGotOne(Exception): pass # User-defined exception
>>> def grail():
raise AlreadyGotOne() # Raise an instance
>>> try:
... grail()
... except AlreadyGotOne: # Catch class name
... print('got exception')
...
got exception
>>>
```

As we'll see later, an as clause on an except can gain access to the exception object itself. Class-based exceptions allow scripts to build exception categories, which can inherit behavior, and have attached state information and methods. They can also customize their error message text displayed if they're not caught:

```
>>> class Career(Exception):
def __str__(self): return 'So I became a waiter...'
>>> raise Career()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.Career: So I became a waiter...
>>>
```

Termination Actions

Finally, try statements can say "finally"—that is, they may include finally blocks.

These look like except handlers for exceptions, but the try/finally combination specifies termination actions that always execute "on the way out," regardless of whether an exception occurs in the try block or not;

```
>>> try:
... fetcher(x, 3)
... finally: # Termination actions
... print('after fetch')
...
'm'
after fetch
>>>
```

Here, if the try block finishes without an exception, the finally block will run, and the program will resume after the entire try. In this case, this statement seems a bit silly —we might as well have simply typed the print right after a call to the function, and skipped the try altogether:

```
fetcher(x, 3)
print('after fetch')
```

There is a problem with coding this way, though: if the function call raises an exception, the print will never be reached. The try/finally combination avoids this pitfall—when an exception does occur in a try block, finally blocks are executed while the program is being unwound:

```
>>> def after():
try:
fetcher(x, 4)
finally:
print('after fetch')
print('after try?')
>>> after()
after fetch
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in after
File "<stdin>", line 2, in fetcher
IndexError: string out of range
>>>
```

Here, we don't get the "after try?" message because control does not resume after the try/finally block when an exception occurs. Instead, Python jumps back to run the finally action, and then propagates the exception up to a prior handler (in this case, to the default handler at the top). If we change the call inside this function so as not to trigger an exception, the finally code still runs, but the program continues after the try:

```
>>> def after():
try:
fetcher(x, 3)
finally:
print('after fetch')
print('after try?')
>>> after()
after fetch
after try?
>>>
```

In practice, try/except combinations are useful for catching and recovering from exceptions, and try/finally combinations come in handy to guarantee that termination actions will fire regardless of any exceptions that may occur in the try block's code.

For instance, you might use try/except to catch errors raised by code that you import from a third-party library, and try/finally to ensure that calls to close files or terminate server connections are always run. We'll see some such practical examples later in this part of the course.

Although they serve conceptually distinct purposes, as of Python 2.5, we can mix except and finally clauses in the same try statement—the finally is run on the way out regardless of whether an exception was raised, and regardless of whether the exception was caught by an except clause.

Python 2.X and 3.X both provide an alternative to try/finally when using some types of objects. The with/as statement runs an object's context management logic to guarantee that termination actions occur, irrespective of
any exceptions in its nested block:

```
>>> with open('lumberjack.txt', 'w') as file: # Always close file on exit
file.write('The larch!\n')
```

Although this option requires fewer lines of code, it's applicable only when processing certain object types, so try/finally is a more general termination structure, and is often simpler than coding a class in cases where with is not already supported. On the other hand, with/as may also run startup actions too, and supports user-defined context management code with access to Python's full OOP toolset.

The try/except/else statement

Now that we've seen the basics, it's time for the details. In the following discussion, I'll first present try/except/else and try/finally as separate statements, because in versions of Python prior to 2.5 they serve distinct roles and cannot be combined, and still are at least logically distinct today. Per the preceding note, in Python 2.5 and later except and finally can be mixed in a single try statement; we'll see the implications of that merging after we've explored the two original forms in isolation.

Syntactically, the try is a compound, multipart statement. It starts with a try header line, followed by a block of (usually) indented statements; then one or more except clauses that identify exceptions to be caught and blocks to process them; and an optional else clause and block at the end. You associate the words try, except, and else by

indenting them to the same level (i.e., lining them up vertically). For reference, here's the general and most complete format in Python 3.X:

```
try:
statements # Run this main action first
except name1:
statements # Run if name1 is raised during try block
except (name2, name3):
statements # Run if any of these exceptions occur
except name4 as var:
statements # Run if name4 is raised, assign instance raised to var
except:
statements # Run for all other exceptions raised
else:
statements # Run if no exception was raised during try block
```

Semantically, the block under the try header in this statement represents the main action of the statement—the code you're trying to run and wrap in error processing logic. The except clauses define handlers for exceptions raised during the try block, and the else clause (if coded) provides a handler to be run if no exceptions occur. The var entry here has to do with a feature of raise statements and exception classes, which we will discuss in full later in this module.

How try Statements Work

Operationally, here's how try statements are run. When a try statement is entered, Python marks the current program context so it can return to it if an exception occurs.

The statements nested under the try header are run first. What happens next depends on whether exceptions are raised while the try block's statements are running, and whether they match those that the try is watching for:

- If an exception occurs while the try block's statements are running, and the exception matches one that the statement names, Python jumps back to the try and runs the statements under the first except clause that matches the raised exception, after assigning the raised exception object to the variable named after the as keyword in the clause (if present). After the except block runs, control then resumes below the entire try statement (unless the except block itself raises another exception, in which case the process is started anew from this point in the code).
- If an exception occurs while the try block's statements are running, but the exception does not match one that the statement names, the exception is propagated up to the next most recently entered try statement that matches the exception; if no such matching try statement can be found and the search reaches the top level of the process, Python kills the program and prints a default error message.
- If an exception does not occur while the try block's statements are running, Python runs the statements under the else line (if present), and control then resumes below the entire try statement.

In other words, except clauses catch any matching exceptions that happen while the try block is running, and the else clause runs only if no exceptions happen while the try block runs. Exceptions raised are matched to exceptions named in except clauses by superclass relationships, and the empty except clause (with no exception name) matches all (or all other) exceptions.

The except clauses are focused exception handlers—they catch exceptions that occur only within the statements in the associated try block. However, as the try block's statements can call functions coded elsewhere in a program, the source of an exception may be outside the try statement itself.

In fact, a try block might invoke arbitrarily large amounts of program code—including code that may have try statements of its own, which will be searched first when exceptions occur. That is, try statements can nest at runtime.
.

try Statement Clauses
When you write a try statement, a variety of clauses can appear after the try header.

Table 34-1 summarizes all the possible forms—you must use at least one. We've already met some of these: as you know, except clauses catch exceptions, finally clauses run on the way out, and else clauses run if no exceptions are encountered.

Formally, there may be any number of except clauses, but you can code else only if there is at least one except, and there can be only one else and one finally. Through Python 2.4, the finally clause must appear alone (without else or except); the try/finally is really a different statement. As of Python 2.5, however, a finally can appear in the same statement as except and else (more on the ordering rules later in this module when we meet the unified try statement).

Table 34-1. try statement clause forms

Clause form	**Interpretation**
except:	Catch all (or all other) exception types
except name:	Catch a specific exception only
except name as value:	Catch the listed exception and assign its instance
except (name1, name2):	Catch any of the listed exceptions
except (name1, name2) as value:	Catch any listed exception and assign its instance
else:	Run if no exceptions are raised in the try block
finally:	Always perform this block on exit

We'll explore the entries with the extra as value part in more detail when we meet the raise statement later in this module. They provide access to the objects that are raised as exceptions.

Catching any and all exceptions
The first and fourth entries in Table 34-1 are new here:

- except clauses that list no exception name (except:) catch all exceptions not previously listed in the try statement.
- except clauses that list a set of exceptions in parentheses (except (e1, e2, e3):) catch any of the listed exceptions.

Because Python looks for a match within a given try by inspecting the except clauses from top to bottom, the parenthesized version has the same effect as listing each exception in its own except clause, but you have to code the statement body associated with each only once. Here's an example of multiple except clauses at work, which demonstrates just how specific your handlers can be:

```
try:
action()
except NameError:
...
except IndexError:
...
except KeyError:
...
except (AttributeError, TypeError, SyntaxError):
...
else:
...
```

In this example, if an exception is raised while the call to the action function is running, Python returns to the try and searches for the first except that names the exception raised. It inspects the except clauses from top to bottom and left to right, and runs the statements under the first one that matches. If none match, the exception is propagated
past this try. Note that the else runs only when no exception occurs in action—it does not run when an exception without a matching except is raised.

Catching all: The empty except and Exception

If you really want a general "catchall" clause, an empty except does the trick:

```
try:
action()
except NameError:
... # Handle NameError
except IndexError:
... # Handle IndexError
except:
... # Handle all other exceptions
else:
... # Handle the no-exception case
```

The empty except clause is a sort of wildcard feature—because it catches everything, it allows your handlers to be as general or specific as you like. In some scenarios, this form may be more convenient than listing all possible exceptions in a try. For example, the following catches everything without listing anything:

```
try:
action()
except:
... # Catch all possible exceptions
```

Empty excepts also raise some design issues, though. Although convenient, they may catch unexpected system exceptions unrelated to your code, and they may inadvertently intercept exceptions meant for another handler. For example, even system exit calls and Ctrl-C key combinations in Python trigger exceptions, and you usually want these to pass. Even worse, the empty except may also catch genuine programming mistakes for which you probably want to see an error message. We'll revisit this as a gotcha at the end of this part of the course. For now, I'll just say, "use with care." Python 3.X more strongly supports an alternative that solves one of these problems—catching an exception named Exception has almost the same effect as an empty except, but ignores exceptions related to system exits:

```
try:
action()
except Exception:
... # Catch all possible exceptions, except exits
```

In short, it works because exceptions match if they are a subclass of one named in an except clause, and Exception is a superclass of all the exceptions you should generally catch this way. This form has most of the same convenience of the empty except, without the risk of catching exit events. Though better, it also has some of the same dangers—especially with regard to masking programming errors.

The try else Clause

The purpose of the else clause is not always immediately obvious to Python newcomers.

Without it, though, there is no direct way to tell (without setting and checking Boolean flags) whether the flow of control has proceeded past a try statement because no exception was raised, or because an exception occurred and was handled. Either way, we wind up after the try:

```
try:
...run code...
except IndexError:
...handle exception...
# Did we get here because the try failed or not?
```

Much like the way else clauses in loops make the exit cause more apparent, the else clause provides syntax in a try that makes what has happened obvious and unambiguous:

```
try:
...run code...
```

```
except IndexError:
...handle exception...
else:
...no exception occurred...
You can almost emulate an else clause by moving its code into the try block:
try:
...run code...
...no exception occurred...
except IndexError:
...handle exception...
```

This can lead to incorrect exception classifications, though. If the "no exception occurred" action triggers an IndexError, it will register as a failure of the try block and erroneously trigger the exception handler below the try (subtle, but true!). By using an explicit else clause instead, you make the logic more obvious and guarantee that except handlers will run only for real failures in the code you're wrapping in a try, not for failures in the else no-exception case's action.

The try/finally statement

The other flavor of the try statement is a specialization that has to do with finalization (a.k.a. termination) actions. If a finally clause is included in a try, Python will always run its block of statements "on the way out" of the try statement, whether an exception occurred while the try block was running or not. Its general form is:

```
try:
statements # Run this action first
finally:
statements # Always run this code on the way out
```

With this variant, Python begins by running the statement block associated with the try header line as usual. What happens next depends on whether an exception occurs during the try block:

- If an exception does not occur while the try block is running, Python continues on to run the finally block, and then continues execution past the try statement.
- If an exception does occur during the try block's run, Python still comes back and runs the finally block, but it then propagates the exception up to a previously entered try or the top-level default handler; the program does not resume execution below the finally clause's try statement. That is, the finally block is run even if an exception is raised, but unlike an except, the finally does not terminate the exception—it continues being raised after the finally block runs.

The try/finally form is useful when you want to be completely sure that an action will happen after some code runs, regardless of the exception behavior of the program. In practice, it allows you to specify cleanup actions that always must occur, such as file closes and server disconnects where required.

Note that the finally clause cannot be used in the same try statement as except and else in Python 2.4 and earlier, so the try/finally is best thought of as a distinct statement form if you are using an older release. In Python 2.5, and later, however, finally can appear in the same statement as except and else, so today there is really a single try statement with many optional clauses (more about this shortly). Whichever version you use, though, the finally clause still serves the same purpose—to specify "cleanup" actions that must always be run, regardless of any exceptions.

Unified try/except/finally

In all versions of Python prior to release 2.5 (for its first 15 years of life, more or less),
the try statement came in two flavors and was really two separate statements—we could either use a finally to ensure that cleanup code was always run, or write except blocks to catch and recover from specific exceptions and optionally specify an else clause to be run if no exceptions occurred.

That is, the finally clause could not be mixed with except and else. This was partly because of implementation issues, and partly because the meaning of mixing the two seemed obscure—catching and recovering from exceptions seemed a disjoint concept from performing cleanup actions.

In Python 2.5 and later, though, the two statements have merged. Today, we can mix finally, except, and else clauses in the same statement—in part because of similar utility in the Java language. That is, we can now write a statement of this form:

```
try: # Merged form
main-action
except Exception1:
handler1
except Exception2: # Catch exceptions
handler2
...
else: # No-exception handler
else-block
finally: # The finally encloses all else
finally-block
```

The code in this statement's main-action block is executed first, as usual. If that code raises an exception, all the except blocks are tested, one after another, looking for a match to the exception raised. If the exception raised is Exception1, the handler1 block is executed; if it's Exception2, handler2 is run, and so on. If no exception is raised, the else-block is executed.

No matter what's happened previously, the finally-block is executed once the main action block is complete and any raised exceptions have been handled. In fact, the code in the finally-block will be run even if there is an error in an exception handler or the else-block and a new exception is raised.

As always, the finally clause does not end the exception—if an exception is active when the finally-block is executed, it continues to be propagated after the finallyblock runs, and control jumps somewhere else in the program (to another try, or to the default top-level handler). If no exception is active when the finally is run, control
resumes after the entire try statement.

The net effect is that the finally is always run, regardless of whether:

- An exception occurred in the main action and was handled.
- An exception occurred in the main action and was not handled.
- No exceptions occurred in the main action.
- A new exception was triggered in one of the handlers.

Again, the finally serves to specify cleanup actions that must always occur on the way out of the try, regardless of what exceptions have been raised or handled.

Unified try Statement Syntax

When combined like this, the try statement must have either an except or a finally, and the order of its parts must be like this:

```
try -> except -> else -> finally
```

where the else and finally are optional, and there may be zero or more excepts, but there must be at least one except if an else appears. Really, the try statement consists of two parts: excepts with an optional else, and/or the finally.
In fact, it's more accurate to describe the merged statement's syntactic form this way (square brackets mean optional and star means zero-or-more here):

```
try: # Format 1
statements
except [type [as value]]: # [type [, value]] in Python 2.X
```

```
statements
[except [type [as value]]:
statements]*
[else:
statements]
[finally:
statements]
try: # Format 2
statements
finally:
statements
```

Because of these rules, the else can appear only if there is at least one except, and it's always possible to mix except and finally, regardless of whether an else appears or not. It's also possible to mix finally and else, but only if an except appears too (though the except can omit an exception name to catch everything and run a raise statement, described later, to reraise the current exception). If you violate any of these ordering rules, Python will raise a syntax error exception before your code runs.

Combining finally and except by Nesting

Prior to Python 2.5, it is actually possible to combine finally and except clauses in a try by syntactically nesting a try/except in the try block of a try/finally statement.

The following has the same effect as the new merged form shown at the start of this section:

```
try: # Nested equivalent to merged form
try:
  main-action
except Exception1:
  handler1
except Exception2:
  handler2
...
else:
  no-error
finally:
  cleanup
```

Again, the finally block is always run on the way out, regardless of what happened in the main action and regardless of any exception handlers run in the nested try (trace through the four cases listed previously to see how this works the same). Since an else always requires an except, this nested form even sports the same mixing constraints of the unified statement form outlined in the preceding section.

However, this nested equivalent seems more obscure to some, and requires more code than the new merged form—though just one four-character line plus extra indentation.

Mixing finally into the same statement makes your code arguably easier to write and read, and is a generally preferred technique today.

The raise statement

To trigger exceptions explicitly, you can code raise statements. Their general form is simple—a raise statement consists of the word raise, optionally followed by the class to be raised or an instance of it:

raise instance # Raise instance of class

```
raise class # Make and raise instance of class: makes an instance
```

```
raise # Reraise the most recent exception
```

As mentioned earlier, exceptions are always instances of classes in Python 2.6, 3.0, and later. Hence, the first raise form here is the most common—we provide an instance directly, either created before the raise or within the raise statement itself. If we pass a class instead, Python calls the class with no constructor arguments, to create an instance to be raised; this form is equivalent to adding parentheses after the class reference.

The last form reraises the most recently raised exception; it's commonly used in exception handlers to propagate exceptions that have been caught.

Raising Exceptions

To make this clearer, let's look at some examples. With built-in exceptions, the following two forms are equivalent—both raise an instance of the exception class named, but the first creates the instance implicitly:

```
raise IndexError # Class (instance created)
raise IndexError() # Instance (created in statement)
```

We can also create the instance ahead of time—because the raise statement accepts any kind of object reference, the following two examples raise IndexError just like the prior two:

```
exc = IndexError() # Create instance ahead of time
raise exc
excs = [IndexError, TypeError]
raise excs[0]
```

When an exception is raised, Python sends the raised instance along with the exception.

If a try includes an except name as X: clause, the variable X will be assigned the instance provided in the raise:

```
try:
...
except IndexError as X: # X assigned the raised instance object
...
```

The as is optional in a try handler (if it's omitted, the instance is simply not assigned to a name), but including it allows the handler to access both data in the instance and methods in the exception class.

This model works the same for user-defined exceptions we code with classes—the following, for example, passes to the exception class constructor arguments that become available in the handler through the assigned instance:

```
class MyExc(Exception): pass
...
raise MyExc('spam') # Exception class with constructor args
...
try:
...
except MyExc as X: # Instance attributes available in handler
print(X.args)
```

Regardless of how you name them, exceptions are always identified by class instance objects, and at most one is active at any given time. Once caught by an except clause anywhere in the program, an exception dies (i.e., won't propagate to another try), unless it's reraised by another raise statement or error.

Scopes and try except Variables

Now that we've seen the as variable in action, though, we can finally clarify the related version-specific scope issue. In Python 2.X, the exception reference variable name in an except clause is not localized to the clause itself, and is available after the associated block runs:

```
c:\code> py −2
```

```
>>> try:
... 1 / 0
... except Exception as X: # 2.X does not localize X either way
... print X
...
integer division or modulo by zero
>>> X
ZeroDivisionError('integer division or modulo by zero',)
```

This is true in 2.X whether we use the 3.X-style as or the earlier comma syntax:

```
>>> try:
... 1 / 0
... except Exception, X:
... print X
...
integer division or modulo by zero
>>> X
ZeroDivisionError('integer division or modulo by zero',)
```

By contrast, Python 3.X localizes the exception reference name to the except block—the variable is not available after the block exits, much like a temporary loop variable in 3.X comprehension expressions (3.X also doesn't accept 2.X's except comma syntax, as noted earlier):

```
c:\code> py −3
>>> try:
... 1 / 0
... except Exception, X:
SyntaxError: invalid syntax
>>> try:
... 1 / 0
... except Exception as X: # 3.X localizes 'as' names to except block
... print(X)
...
division by zero
>>> X
NameError: name 'X' is not defined
```

Unlike compression loop variables, though, this variable is removed after the except block exits in 3.X. It does so because it would otherwise retain a reference to the runtime call stack, which would defer garbage collection and thus retain excess memory space.

This removal occurs, though, even if you're using the name elsewhere, and is more extreme policy than that used for comprehensions:

```
>>> X = 99
>>> try:
... 1 / 0
... except Exception as X: # 3.X localizes _and_ removes on exit!
... print(X)
...
division by zero
>>> X
NameError: name 'X' is not defined
>>> X = 99
>>> {X for X in 'spam'} # 2.X/3.X localizes only: not removed
{'s', 'a', 'p', 'm'}
```

```
>>> X
99
```

Because of this, you should generally use unique variable names in your try statement's except clauses, even if they are localized by scope. If you do need to reference the exception instance after the try statement, simply assign it to another name that won't be automatically removed:

```
>>> try:
... 1 / 0
... except Exception as X: # Python removes this reference
... print(X)
... Saveit = X # Assign exc to retain exc if needed
...
division by zero
>>> X
NameError: name 'X' is not defined
>>> Saveit
ZeroDivisionError('division by zero',)
```

Propagating Exceptions with raise

The raise statement is a bit more feature-rich than we've seen thus far. For example, a raise that does not include an exception name or extra data value simply reraises the current exception. This form is typically used if you need to catch and handle an exception but don't want the exception to die in your code:

```
>>> try:
... raise IndexError('spam') # Exceptions remember arguments
... except IndexError:
... print('propagating')
... raise # Reraise most recent exception
...
propagating
Traceback (most recent call last):
File "<stdin>", line 2, in <module>
IndexError: spam
```

Running a raise this way reraises the exception and propagates it to a higher handler (or the default handler at the top, which stops the program with a standard error message).

Notice how the argument we passed to the exception class shows up in the error messages.

The assert statement

As a somewhat special case for debugging purposes, Python includes the assert statement.

It is mostly just syntactic shorthand for a common raise usage pattern, and an assert can be thought of as a conditional raise statement. A statement of the form:

```
assert test, data # The data part is optional
works like the following code:
if __debug__:
if not test:
raise AssertionError(data)
```

In other words, if the test evaluates to false, Python raises an exception: the data item (if it's provided) is used as the exception's constructor argument. Like all exceptions, the AssertionError exception will kill your program if it's not caught with a try, in which case the data item shows up as part of the standard error message.

As an added feature, assert statements may be removed from a compiled program's byte code if the -O Python command-line flag is used, thereby optimizing the program.

AssertionError is a built-in exception, and the __debug__ flag is a built-in name that is automatically set to True unless the -O flag is used. Use a command line like python –O main.py to run in optimized mode and disable (and hence skip) asserts.

Module 22 - Exception Objects

So far, I've been deliberately vague about what an exception actually is. As suggested earlier, as of Python 2.6 and 3.0 both built-in and user-defined exceptions are identified by class instance objects. This is what is raised and propagated along by exception processing, and the source of the class matched against exceptions named in try statements.

Although this means you must use object-oriented programming to define new exceptions in your programs—and introduces a knowledge dependency that deferred full exception coverage to this part of the course—basing exceptions on classes and OOP offers a number of benefits. Among them, class-based exceptions:

- Can be organized into categories. Exceptions coded as classes support future changes by providing categories—adding new exceptions in the future won't generally require changes in try statements.
- Have state information and behavior. Exception classes provide a natural place for us to store context information and tools for use in the try handler—instances have access to both attached state information and callable methods.
- Support inheritance. Class-based exceptions can participate in inheritance hierarchies to obtain and customize common behavior—inherited display methods, for example, can provide a common look and feel for error messages.

Because of these advantages, class-based exceptions support program evolution and larger systems well. As we'll find, all built-in exceptions are identified by classes and are organized into an inheritance tree, for the reasons just listed. You can do the same with user-defined exceptions of your own.

In fact, in Python 3.X the built-in exceptions we'll study here turn out to be integral to new exceptions you define. Because 3.X requires user-defined exceptions to inherit from built-in exception superclasses that provide useful defaults for printing and state retention, the task of coding user-defined exceptions also involves understanding the roles of these built-ins.

String-based exceptions

Prior to Python 2.6 and 3.0, it was possible to define exceptions with both class instances and string objects. String-based exceptions began issuing deprecation warnings in 2.5 and were removed in 2.6 and 3.0, so today you should use class-based exceptions, as shown in this course. If you work with legacy code, though, you might still come across string exceptions. They might also appear in books, tutorials, and web resources written a few years ago (which qualifies as an eternity in Python years!).

String exceptions were straightforward to use—any string would do, and they matched by object identity, not value (that is, using is, not ==):

```
C:\code> C:\Python25\python
>>> myexc = "My exception string" # Were we ever this young?...
>>> try:
... raise myexc
... except myexc:
... print('caught')
...
caught
```

This form of exception was removed because it was not as good as classes for larger programs and code maintenance. In modern Pythons, string exceptions trigger exceptions instead:

```
C:\code> py −3
>>> raise 'spam'
TypeError: exceptions must derive from BaseException
C:\code> py −2
>>> raise 'spam'
TypeError: exceptions must be old-style classes or derived from BaseException, ...etc
```

Although you can't use string exceptions today, they actually provide a natural vehicle for introducing the class-based exceptions model.

Class-based exceptions

Strings were a simple way to define exceptions. As described earlier, however, classes have some added advantages that merit a quick look. Most prominently, they allow us to identify exception categories that are more flexible to use and maintain than simple strings. Moreover, classes naturally allow for attached exception details and support inheritance. Because they are seen by many as the better approach, they are now required.

Coding details aside, the chief difference between string and class exceptions has to do with the way that exceptions raised are matched against except clauses in try statements:

- String exceptions were matched by simple object identity: the raised exception was matched to except clauses by Python's is test.
- Class exceptions are matched by superclass relationships: the raised exception matches an except clause if that except clause names the exception instance's class or any superclass of it.

That is, when a try statement's except clause lists a superclass, it catches instances of that superclass, as well as instances of all its subclasses lower in the class tree. The net effect is that class exceptions naturally support the construction of exception hierarchies:

- Superclasses become category names, and subclasses become specific kinds of exceptions within a category. By naming a general exception superclass, an except clause can catch an entire category of exceptions—any more specific subclass will match.
- String exceptions had no such concept: because they were matched by simple object identity, there was no direct way to organize exceptions into more flexible categories or groups. The net result was that exception handlers were coupled with exception sets in a way that made changes difficult.

In addition to this category idea, class-based exceptions better support exception state information (attached to instances) and allow exceptions to participate in inheritance hierarchies (to obtain common behaviors). Because they offer all the benefits of classes and OOP in general, they provide a more powerful alternative to the now-defunct string-based exceptions model in exchange for a small amount of additional code.

Coding Exceptions Classes

Let's look at an example to see how class exceptions translate to code. In the following file, classexc.py, we define a superclass called General and two subclasses called Specific1 and Specific2. This example illustrates the notion of exception categories—General is a category name, and its two subclasses are specific types of exceptions within the category. Handlers that catch General will also catch any subclasses of it, including Specific1 and Specific2:

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass
 def raiser0():
   X = General() # Raise superclass instance
   raise X
 def raiser1():
   X = Specific1() # Raise subclass instance
   raise X
 def raiser2():
   X = Specific2() # Raise different subclass instance
   raise X
   for func in (raiser0, raiser1, raiser2):
     try:
       func()
     except General: # Match General or any subclass of it

import sys
```

```
print('caught: %s' % sys.exc_info()[0])
C:\code> python classexc.py
caught: <class '__main__.General'>
caught: <class '__main__.Specific1'>
caught: <class '__main__.Specific2'>
```

This code is mostly straightforward, but here are a few points to notice:

Exception superclass

Classes used to build exception category trees have very few requirements—in fact, in this example they are mostly empty, with bodies that do nothing but pass. Notice, though, how the top-level class here inherits from the built-in Exception class.

This is required in Python 3.X; Python 2.X allows standalone classic classes to serve as exceptions too, but it requires new-style classes to be derived from built-in exception classes just as in 3.X. Although we don't employ it here, because Exception provides some useful behavior we'll meet later, it's a good idea to inherit from it in either Python.

Raising instances

In this code, we call classes to make instances for the raise statements. In the class exception model, we always raise and catch a class instance object. If we list a class name without parentheses in a raise, Python calls the class with no constructor argument to make an instance for us. Exception instances can be created before the raise, as done here, or within the raise statement itself.

Catching categories

This code includes functions that raise instances of all three of our classes as exceptions, as well as a top-level try that calls the functions and catches General exceptions. The same try also catches the two specific exceptions, because they are subclasses of General—members of its category.

Exception details

The exception handler here uses the sys.exc_info call, it's how we can grab hold of the most recently raised exception in a generic fashion. Briefly, the first item in its result is the class of the exception raised, and the second is the actual instance raised. In a general except clause like the one here that catches all classes in a category, sys.exc_info is one way to determine exactly what's occurred. In this particular case, it's equivalent to fetching the instance's __class__ attribute. The sys.exc_info scheme is also commonly used with empty except clauses that catch everything.

The last point merits further explanation. When an exception is caught, we can be sure that the instance raised is an instance of the class listed in the except, or one of its more specific subclasses. Because of this, the __class__ attribute of the instance also gives the exception type. The following variant in classexc2.py, for example, works the same as the prior example—it uses the as extension in its except clause to assign a variable to the instance actually raised:

```
class General(Exception): pass
class Specific1(General): pass
class Specific2(General): pass
 def raiser0(): raise General()
 def raiser1(): raise Specific1()
 def raiser2(): raise Specific2()
 for func in (raiser0, raiser1, raiser2):
   try:
     func()
   except General as X: # X is the raised instance
     print('caught: %s' % X.__class__) # Same as sys.exc_info()[0]
```

Because __class__ can be used like this to determine the specific type of exception raised, sys.exc_info is more useful for empty except clauses that do not otherwise have a way to access the instance or its class. Furthermore, more realistic programs usually should not have to care about which specific exception was raised at all—by calling methods of the exception class instance generically, we automatically dispatch to behavior tailored for the exception raised.

Why Exception Hierarchies?

Because there are only three possible exceptions in the prior section's example, it doesn't really do justice to the utility of class exceptions. In fact, we could achieve the same effects by coding a list of exception names in parentheses within the except clause:

```
try:
func()
except (General, Specific1, Specific2): # Catch any of these
...
```

This approach worked for the defunct string exception model too. For large or high exception hierarchies, however, it may be easier to catch categories using class-based categories than to list every member of a category in a single except clause. Perhaps more importantly, you can extend exception hierarchies as software needs evolve by adding new subclasses without breaking existing code.

Suppose, for example, you code a numeric programming library in Python, to be used by a large number of people. While you are writing your library, you identify two things that can go wrong with numbers in your code—division by zero, and numeric overflow.

You document these as the two standalone exceptions that your library may raise:

```
# mathlib.py
class Divzero(Exception): pass
class Oflow(Exception): pass
def func():
...
raise Divzero()
...and so on...
```

Now, when people use your library, they typically wrap calls to your functions or classes in try statements that catch your two exceptions; after all, if they do not catch your exceptions, exceptions from your library will kill their code:

```
# client.py
import mathlib
try:
  mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow):
  ...handle and recover...
```

This works fine, and lots of people start using your library. Six months down the road, though, you revise it (as programmers are prone to do!). Along the way, you identify a new thing that can go wrong—underflow, perhaps—and add that as a new exception:

```
# mathlib.py
class Divzero(Exception): pass
class Oflow(Exception): pass
class Uflow(Exception): pass
```

Unfortunately, when you re-release your code, you create a maintenance problem for your users. If they've listed your exceptions explicitly, they now have to go back and change every place they call your library to include the newly added exception name:

```
# client.py
```

```
try:
  mathlib.func(...)
except (mathlib.Divzero, mathlib.Oflow, mathlib.Uflow):
  ...handle and recover...
```

This may not be the end of the world. If your library is used only in-house, you can make the changes yourself. You might also ship a Python script that tries to fix such code automatically (it would probably be only a few dozen lines, and it would guess right at least some of the time). If many people have to change all their try statements each time you alter your exception set, though, this is not exactly the most polite of upgrade policies.

Your users might try to avoid this pitfall by coding empty except clauses to catch all possible exceptions:

```
# client.py
try:
  mathlib.func(...)
except: # Catch everything here (or catch Exception super)
  ...handle and recover...
```

But this workaround might catch more than they bargained for—things like running out of memory, keyboard interrupts (Ctrl-C), system exits, and even typos in their own try block's code will all trigger exceptions, and such things should pass, not be caught and erroneously classified as library errors. Catching the Exception super class improves on this, but still intercepts—and thus may mask—program errors.

And really, in this scenario users want to catch and recover from only the specific exceptions the library is defined and documented to raise. If any other exception occurs during a library call, it's likely a genuine bug in the library (and probably time to contact the vendor!). As a rule of thumb, it's usually better to be specific than general in exception handlers.

So what to do, then? Class exception hierarchies fix this dilemma completely. Rather than defining your library's exceptions as a set of autonomous classes, arrange them into a class tree with a common superclass to encompass the entire category:

```
# mathlib.py
class NumErr(Exception): pass
class Divzero(NumErr): pass
class Oflow(NumErr): pass
def func():
  ...
raise DivZero()
  ...and so on...
```

This way, users of your library simply need to list the common superclass (i.e., category) to catch all of your library's exceptions, both now and in the future:

```
# client.py
import mathlib
try:
  mathlib.func(...)
except mathlib.NumErr:
  ...report and recover...
```

When you go back and hack (update) your code again, you can add new exceptions as new subclasses of the common superclass:

```
# mathlib.py
...
class Uflow(NumErr): pass
```

The end result is that user code that catches your library's exceptions will keep working, unchanged. In fact, you are free to add, delete, and change exceptions arbitrarily in the future—as long as clients name the superclass, and that superclass remains intact, they are insulated from changes in your exceptions set. In other words, class exceptions provide a better answer to maintenance issues than strings could.

Class-based exception hierarchies also support state retention and inheritance in ways that make them ideal in larger programs. To understand these roles, though, we first need to see how user-defined exception classes relate to the built-in exceptions from which they inherit.

Built-in Exception Classes

I didn't really pull the prior section's examples out of thin air. All built-in exceptions that Python itself may raise are predefined class objects. Moreover, they are organized into a shallow hierarchy with general superclass categories and specific subclass types, much like the prior section's exceptions class tree.

In Python 3.X, all the familiar exceptions you've seen (e.g., SyntaxError) are really just predefined classes, available as built-in names in the module named builtins; in Python 2.X, they instead live in __builtin__ and are also attributes of the standard library module exceptions. In addition, Python organizes the built-in exceptions into a hierarchy, to support a variety of catching modes. For example:

```
BaseException: topmost root, printing and constructor defaults
```

The top-level root superclass of exceptions. This class is not supposed to be directly inherited by user-defined classes (use Exception instead). It provides default printing and state retention behavior inherited by subclasses. If the str built-in is called on an instance of this class (e.g., by print), the class returns the display strings of the constructor arguments passed when the instance was created (or an empty string if there were no arguments). In addition, unless subclasses replace this class's constructor, all of the arguments passed to this class at instance construction time are stored in its args attribute as a tuple.

Exception: root of user-defined exceptions

The top-level root superclass of application-related exceptions. This is an immediate subclass of BaseException and is a superclass to every other built-in exception, except the system exit event classes (SystemExit, KeyboardInterrupt, and Genera torExit). Nearly all user-defined classes should inherit from this class, not BaseEx ception. When this convention is followed, naming Exception in a try statement's handler ensures that your program will catch everything but system exit events, which should normally be allowed to pass. In effect, Exception becomes a catchall in try statements and is more accurate than an empty except.

ArithmeticError: root of numeric errors

A subclass of Exception, and the superclass of all numeric errors. Its subclasses identify specific numeric errors:

- OverflowError, ZeroDivisionError, and Floating
- PointError.
- LookupError: root of indexing errors

A subclass of Exception, and the superclass category for indexing errors for both sequences and mappings—IndexError and KeyError—as well as some Unicode lookup errors.

And so on—because the built-in exception set is prone to frequent changes, this course doesn't document it exhaustively. You can read further about this structure in reference texts such as Python Pocket Reference or the Python library manual. In fact, the exceptions class tree differs slightly between Python 3.X and 2.X in ways we'll omit here, because they are not relevant to examples.

You can also see the built-in exceptions class tree in the help text of the exceptions module in Python 2.X only:

```
>>> import exceptions
>>> help(exceptions)
...lots of text omitted...
```

This module is removed in 3.X, where you'll find up-to-date help in the other resources mentioned.

Built-in Exception Categories

The built-in class tree allows you to choose how specific or general your handlers will be. For example, because the built-in exception ArithmeticError is a superclass for more specific exceptions such as OverflowError and ZeroDivisionError:

- By listing ArithmeticError in a try, you will catch any kind of numeric error raised.
- By listing ZeroDivisionError, you will intercept just that specific type of error, and no others.

Similarly, because Exception is the superclass of all application-level exceptions in Python 3.X, you can generally use it as a catchall—the effect is much like an empty except, but it allows system exit exceptions to pass and propagate as they usually should:

```
try:
  action()
except Exception: # Exits not caught here
  ...handle all application exceptions...
else:
  ...handle no-exception case...
```

This doesn't quite work universally in Python 2.X, however, because standalone userdefined exceptions coded as classic classes are not required to be subclasses of the Exception root class. This technique is more reliable in Python 3.X, since it requires all classes to derive from built-in exceptions. Even in Python 3.X, though, this scheme suffers most of the same potential pitfalls as the empty except—it might intercept exceptions intended for elsewhere, and it might mask genuine programming errors. Since this is such a common issue.

Whether or not you will leverage the categories in the built-in class tree, it serves as a good example; by using similar techniques for class exceptions in your own code, you can provide exception sets that are flexible and easily modified.

Default Printing and State

Built-in exceptions also provide default print displays and state retention, which is often as much logic as user-defined classes require. Unless you redefine the constructors your classes inherit from them, any constructor arguments you pass to these classes are automatically saved in the instance's args tuple attribute, and are automatically displayed when the instance is printed. An empty tuple and display string are used if no constructor arguments are passed, and a single argument displays as itself (not as a tuple).

This explains why arguments passed to built-in exception classes show up in error messages—any constructor arguments are attached to the instance and displayed when the instance is printed:

```
>>> raise IndexError # Same as IndexError(): no arguments
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError
>>> raise IndexError('spam') # Constructor argument attached, printed
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: spam
>>> I = IndexError('spam') # Available in object attribute
>>> I.args
('spam',)
>>> print(I) # Displays args when printed manually
spam
```

The same holds true for user-defined exceptions in Python 3.X (and for new-style classes in 2.X), because they inherit the constructor and display methods present in their builtin superclasses:

```
>>> class E(Exception): pass
...
```

```
>>> raise E
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.E
>>> raise E('spam')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
__main__.E: spam
>>> I = E('spam')
>>> I.args
('spam',)
>>> print(I)
spam
```

When intercepted in a try statement, the exception instance object gives access to both the original constructor arguments and the display method:

```
>>> try:
... raise E('spam')
... except E as X:
... print(X) # Displays and saves constructor arguments
... print(X.args)
... print(repr(X))
...
spam
('spam',)
E('spam',)

>>> try: # Multiple arguments save/display a tuple
... raise E('spam', 'eggs', 'ham')
... except E as X:
... print('%s %s' % (X, X.args))
...
('spam', 'eggs', 'ham') ('spam', 'eggs', 'ham')
```

Note that exception instance objects are not strings themselves, but use the __str__ operator overloading protocol we studied to provide display strings when printed; to concatenate with real strings, perform manual conversions:

```
str(X) + 'as
tr', '%s' % X, and the like.
```

Although this automatic state and display support is useful by itself, for more specific display and state retention needs you can always redefine inherited methods such as __str__ and __init__ in Exception subclasses—as the next section shows.

Module 23 - Designing with Exceptions

This module rounds out this part of the course with a collection of exception design topics and common use case examples, followed by this part's gotchas and exercises.

Because this module also closes out the fundamentals portion of the course at large, it includes a brief overview of development tools as well to help you as you make the migration from Python beginner to Python application developer

Nesting exception handlers

Most of our examples so far have used only a single try to catch exceptions, but what happens if one try is physically nested inside another? For that matter, what does it mean if a try calls a function that runs another try? Technically, try statements can nest, in terms of both syntax and the runtime control flow through your code. I've mentioned this briefly, but let's clarify the idea here.

Both of these cases can be understood if you realize that Python stacks try statements at runtime. When an exception is raised, Python returns to the most recently entered try statement with a matching except clause.

Because each try statement leaves a marker, Python can jump back to earlier trys by inspecting the stacked markers.

This nesting of active handlers is what we mean when we talk about propagating exceptions up to "higher" handlers—such handlers are simply try statements entered earlier in the program's execution flow.

Figure 36-1 illustrates what occurs when try statements with except clauses nest at runtime. The amount of code that goes into a try block can be substantial, and it may contain function calls that invoke other code watching for the same exceptions. When an exception is eventually raised, Python jumps back to the most recently entered try statement that names that exception, runs that statement's except clause, and then resumes execution after that try.

Once the exception is caught, its life is over—control does not jump back to all matching trys that name the exception; only the first (i.e., most recent) one is given the opportunity to handle it. In Figure 36-1, for instance, the raise statement in the function func2 sends control back to the handler in func1, and then the program continues within func1

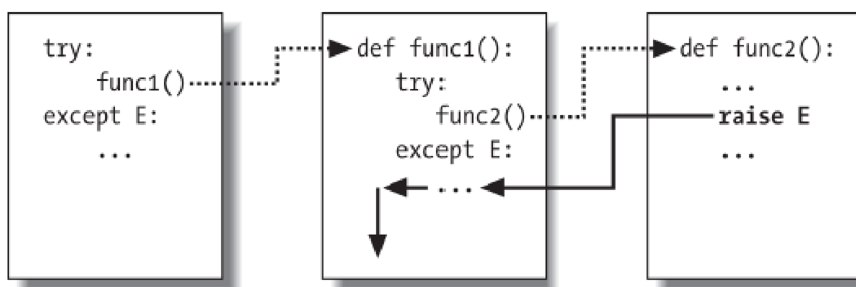


Figure: *Nested try/except statements: when an exception is raised (by you or by Python), control jumps back to the most recently entered try statement with a matching except clause, and the program resumes after that try statement. except clauses intercept and stop the exception—they are where you process and recover from exceptions.*

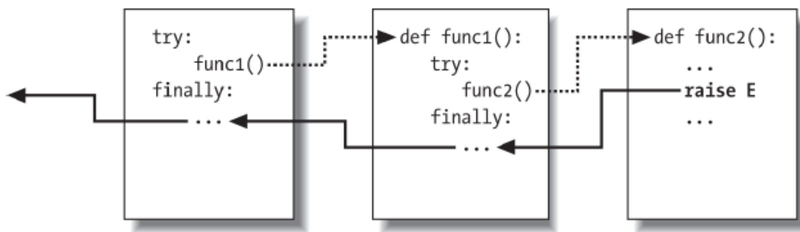


Figure: *Nested try/finally statements: when an exception is raised here, control returns to the most recently entered try to run its finally statement, but then the exception keeps propagating to all finallys in all active try statements and eventually reaches the default top-level handler, where an error message is printed. finally clauses intercept (but do not stop) an exception—they are for actions to be performed "on the way out."*

By contrast, when try statements that contain only finally clauses are nested, each finally block is run in turn when an exception occurs—Python continues propagating the exception up to other trys, and eventually perhaps to the top-level default handler (the standard error message printer). As Figure 36-2 illustrates, the finally clauses do not kill the exception—they just specify code to be run on the way out of each try during the exception propagation process. If there are many try/finally clauses active when an exception occurs, they will all be run, unless a try/except catches the exception somewhere along the way.

In other words, where the program goes when an exception is raised depends entirely upon where it has been—it's a function of the runtime flow of control through the script, not just its syntax. The propagation of an exception essentially proceeds backward through time to try statements that have been entered but not yet exited. This propagation stops as soon as control is unwound to a matching except clause, but not as it passes through finally clauses on the way.

Exception idioms

We've seen the mechanics behind exceptions. Now let's take a look at some of the other ways they are typically used.

Breaking Out of Multiple Nested Loops: "go to"

As mentioned at the start of this part of the course, exceptions can often be used to serve the same roles as other languages' "go to" statements to implement more arbitrary control transfers. Exceptions, however, provide a more structured option that localizes the jump to a specific block of nested code.

In this role, raise is like "go to," and except clauses and exception names take the place of program labels. You can jump only out of code wrapped in a try this way, but that's a crucial feature—truly arbitrary "go to" statements can make code extraordinarily difficult to understand and maintain.

For example, Python's break statement exits just the single closest enclosing loop, but we can always use exceptions to break out of more than one loop level if needed:

```
>>> class Exitloop(Exception): pass
...
>>> try:
... while True:
... while True:
... for i in range(10):
... if i > 3: raise Exitloop # break exits just one level
... print('loop3: %s' % i)
... print('loop2')
... print('loop1')
... except Exitloop:
... print('continuing') # Or just pass, to move on
...
```

```
loop3: 0
loop3: 1
loop3: 2
loop3: 3
continuing
>>> i
4
```

If you change the raise in this to break, you'll get an infinite loop, because you'll break only out of the most deeply nested for loop, and wind up in the second-level loop nesting. The code would then print "loop2" and start the for again.

Also notice that variable i is still what it was after the try statement exits. Variable assignments made in a try are not undone in general, though as we've seen, exception instance variables listed in except clause headers are localized to that clause, and the local variables of any functions that are exited as a result of a raise are discarded.

Technically, active functions' local variables are popped off the call stack and the objects they reference may be garbage-collected as a result, but this is an automatic step.

Exceptions Aren't Always Errors

In Python, all errors are exceptions, but not all exceptions are errors. For instance, we saw that file object read methods return an empty string at the end of a file. In contrast, the built-in input function—which we first met, deployed in an interactive loop, and learned is named raw_input in 2.X—reads a line of text from the standard input stream, sys.stdin, at each call and raises the builtin EOFError at end-of-file.

Unlike file methods, this function does not return an empty string—an empty string from input means an empty line. Despite its name, though, the EOFError exception is just a signal in this context, not an error. Because of this behavior, unless the end-offile should terminate a script, input often appears wrapped in a try handler and nested in a loop, as in the following code:

```
while True:
  try:
    line = input() # Read line from stdin (raw_input in 2.X)
  except EOFError:
    break # Exit loop at end-of-file
  else:
    ...process next line here...
```

Several other built-in exceptions are similarly signals, not errors—for example, calling sys.exit() and pressing Ctrl-C on your keyboard raise SystemExit and KeyboardInter rupt, respectively.

Python also has a set of built-in exceptions that represent warnings rather than errors; some of these are used to signal use of deprecated (phased out) language features. See the standard library manual's description of built-in exceptions for more information, and consult the warnings module's documentation for more on exceptions raised as warnings.

Exception design tips

I'm lumping design tips and gotchas together in this module, because it turns out that the most common gotchas largely stem from design issues. By and large, exceptions are easy to use in Python. The real art behind them is in deciding how specific or general your except clauses should be and how much code to wrap up in try statements.

Let's address the second of these concerns first.

What Should Be Wrapped

In principle, you could wrap every statement in your script in its own try, but that would just be silly (the try statements would then need to be wrapped in try statements!).

What to wrap is really a design issue that goes beyond the language itself, and it will become more apparent with use. But for now, here are a few rules of thumb:

- Operations that commonly fail should generally be wrapped in try statements. For example, operations that interface with system state (file opens, socket calls, and the like) are prime candidates for try.
- However, there are exceptions to the prior rule—in a simple script, you may want failures of such operations to kill your program instead of being caught and ignored. This is especially true if the failure is a showstopper. Failures in Python typically result in useful error messages (not hard crashes), and this is the best outcome some programs could hope for.
- You should implement termination actions in try/finally statements to guarantee their execution, unless a context manager is available as a with/as option. The try/finally statement form allows you to run code whether exceptions occur or not in arbitrary scenarios.
- It is sometimes more convenient to wrap the call to a large function in a single try statement, rather than littering the function itself with many try statements.

That way, all exceptions in the function percolate up to the try around the call, and you reduce the amount of code within the function.

The types of programs you write will probably influence the amount of exception handling you code as well. Servers, for instance, must generally keep running persistently and so will likely require try statements to catch and recover from exceptions. Inprocess testing programs of the kind we saw in this module will probably handle exceptions as well. Simpler one-shot scripts, though, will often ignore exception handling completely because failure at any step requires script shutdown.

Catching Too Much: Avoid Empty except and Exception

As mentioned, exception handler generality is a key design choice. Python lets you pick and choose which exceptions to catch, but you sometimes have to be careful to not be too inclusive. For example, you've seen that an empty except clause catches every exception that might be raised while the code in the try block runs.

That's easy to code, and sometimes desirable, but you may also wind up intercepting an error that's expected by a try handler higher up in the exception nesting structure.

For example, an exception handler such as the following catches and stops every exception that reaches it, regardless of whether another handler is waiting for it:

```
def func():
  try:
    ... # IndexError is raised in here
  except:
    ... # But everything comes here and dies!
  try:
    func()
  except IndexError: # Exception should be processed here
    ...
```

Perhaps worse, such code might also catch unrelated system exceptions. Even things like memory errors, genuine programming mistakes, iteration stops, keyboard interrupts, and system exits raise exceptions in Python. Unless you're writing a debugger or similar tool, such exceptions should not usually be intercepted in your code.

For example, scripts normally exit when control falls off the end of the top-level file.

However, Python also provides a built-in sys.exit(statuscode) call to allow early terminations.

This actually works by raising a built-in SystemExit exception to end the program, so that try/finally handlers run on the way out and special types of programs can intercept the event.1 Because of this, a try with an empty except might unknowingly prevent a crucial exit, as in the following file (exiter.py):

```
import sys
def bye():
  sys.exit(40) # Crucial error: abort now!
  try:
    bye()
  except:
    print('got it') # Oops--we ignored the exit
    print('continuing...')
% python exiter.py
got it
continuing...
```

You simply might not expect all the kinds of exceptions that could occur during an operation. Using the built-in exception classes can help in this particular case, because the Exception superclass is not a superclass of SystemExit:

```
try:
  bye()
except Exception: # Won't catch exits, but _will_ catch many others
  ...
```

In other cases, though, this scheme is no better than an empty except clause—because Exception is a superclass above all built-in exceptions except system-exit events, it still has the potential to catch exceptions meant for elsewhere in the program.

Probably worst of all, both using an empty except and catching the Exception superclass will also catch genuine programming errors, which should be allowed to pass most of the time. In fact, these two techniques can effectively turn off Python's error-reporting machinery, making it difficult to notice mistakes in your code. Consider this code, for example:

```
mydictionary = {...}
...
try:
x = myditctionary['spam'] # Oops: misspelled
except:
x = None # Assume we got KeyError
...continue here with x...
```

The coder here assumes that the only sort of error that can happen when indexing a dictionary is a missing key error. But because the name myditctionary is misspelled (it should say mydictionary), Python raises a NameError instead for the undefined name reference, which the handler will silently catch and ignore. The event handler will incorrectly fill in a None default for the dictionary access, masking the program error.

Moreover, catching Exception here will not help—it would have the exact same effect as an empty except, happily and silently filling in a default and masking a genuine program error you will probably want to know about. If this happens in code that is far removed from the place where the fetched values are used, it might make for a very interesting debugging task!

As a rule of thumb, be as specific in your handlers as you can be—empty except clauses and Exception catchers are handy, but potentially error-prone. In the last example, for instance, you would be better off saying except KeyError: to make your intentions explicit and avoid intercepting unrelated events. In simpler scripts, the potential for problems might not be significant enough to outweigh the convenience of a catchall, but in general, general handlers are generally trouble.

Catching Too Little: Use Class-Based Categories

On the other hand, neither should handlers be too specific. When you list specific exceptions in a try, you catch only what you actually list. This isn't necessarily a bad thing, but if a system evolves to raise other exceptions in the future, you may need to go back and add them to exception lists elsewhere in your code.

We saw this phenomenon at work. For instance, the following handler is written to treat MyExcept1 and MyExcept2 as normal cases and everything else as an error. If you add a MyExcept3 in the future, though, it will be processed as an error unless you update the exception list:

```
try:
...
except (MyExcept1, MyExcept2): # Breaks if you add a MyExcept3 later
... # Nonerrors
else:
... # Assumed to be an error
```

Luckily, careful use of the class-based exceptions can make this code maintenance trap go away completely. As we saw, if you catch a general superclass, you can add and raise more specific subclasses in the future without having to extend except clause lists manually—the superclass becomes an extendible exceptions category:

```
try:
...
except SuccessCategoryName: # OK if you add a MyExcept3 subclass later
... # Nonerrors
else:
... # Assumed to be an error
```

In other words, a little design goes a long way. The moral of the story is to be careful to be neither too general nor too specific in exception handlers, and to pick the granularity of your try statement wrappings wisely. Especially in larger systems, exception policies should be a part of the overall design.

Core language summary

Congratulations! This concludes your look at the fundamentals of the Python programming language. If you've gotten this far, you've become a fully operational Python programmer. In terms of the essentials, though, the Python story—and this course's main journey—is now complete.

Along the way, you've seen just about everything there is to see in the language itself, and in enough depth to apply to most of the code you are likely to encounter in the open source "wild." You've studied built-in types, statements, and exceptions, as well as tools used to build up the larger program units of functions, modules, and classes.

You've also explored important software design issues, the complete OOP paradigm, functional programing tools, program architecture concepts, alternative tool tradeoffs, and more—compiling a skill set now qualified to be turned loose on the task of developing real applications.