PART V

Modules

CHAPTER 21

Modules: The Big Picture

This chapter begins our in-depth look at the Python *module*, the highest-level program organization unit, which packages program code and data for reuse. In concrete terms, modules usually correspond to Python program files (or extensions coded in external languages such as C, Java, or C#). Each file is a module, and modules import other modules to use the names they define. 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`
: Provides a way to reload a module's code without stopping Python

Chapter 3 introduced module fundamentals, and we've been using them ever since. This part of the book begins by expanding on core module concepts, then moves on to explore more advanced module usage. This first chapter offers a general look at the role of modules in overall program structure. In the following chapters, 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, and so on. Because modules and classes are really just glorified namespaces, we'll formalize namespace concepts here as well.

Why Use Modules?

In short, modules provide an easy way to organize components into a system by serving as self-contained packages of variables known as *namespaces*. All the names defined at the top level of a module file become attributes of the imported module object. As we saw in the last part of this book, 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, from an abstract perspective, modules have at least three roles:

Code reuse
: As discussed in Chapter 3, 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. More to the point, modules are a place to define names, known as *attributes*, which may be referenced by multiple external clients.

System namespace partitioning
: Modules are also the highest-level program organization unit in Python. Fundamentally, they are just packages of names. Modules seal up names into self-contained packages, which helps avoid name clashes—you can never see a name in another file, unless you explicitly import that file. In fact, everything "lives" in a module—code you execute and 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 also come in handy 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.

For you to truly understand the role of modules in a Python system, though, we need to digress for a moment and explore the general structure of a Python program.

Python Program Architecture

So far in this book, 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. And 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. Along the way, we'll also explore the central concepts of Python modules, imports, and object attributes.

How to Structure a Program

Generally, a Python program consists of multiple text files containing Python *statements*. The program is structured as one main, *top-level* file, along with zero or more supplemental files known as *modules* in Python.

In Python, 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.

Module files generally don't do anything when run directly; rather, they define tools intended for use in other files. In Python, a file *imports* a module to gain access to the tools it defines, which are known as its *attributes* (i.e., 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 21-1 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 they define.

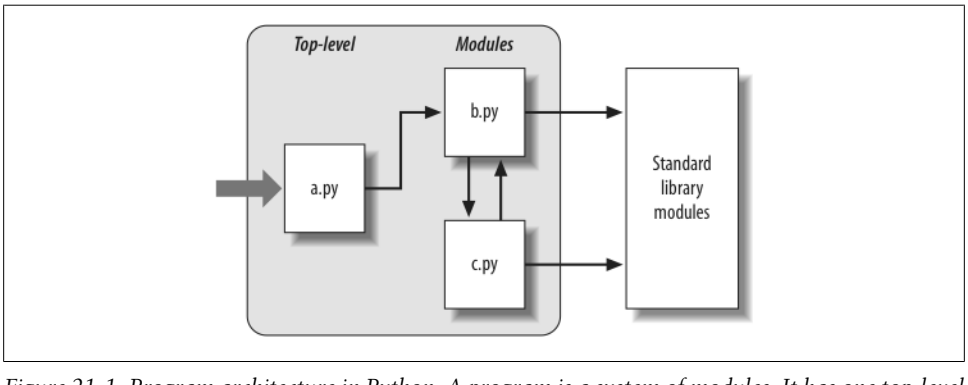

Figure 21-1. Program architecture in Python. A program is a system of modules. It has one top-level script file (launched to run the program), and multiple module files (imported libraries of tools). Scripts and modules are both text files containing Python statements, though the statements in modules usually just create objects to be used later. Python's standard library provides a collection of precoded modules.

For instance, suppose the file *b.py* in Figure 21-1 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 can later be run by passing zero or more values in parentheses after the function's name:

```
def spam(text):
    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
b.spam('gumby')
```

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*. It roughly means "load the file *b.py* (unless it's already loaded), and give me access to all its attributes through the name `b`." `import` (and, as you'll see later, `from`) statements execute and load other files at runtime.

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—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. 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.

The second of the statements in *a.py* calls the function `spam` defined in the module `b`, 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.

You'll see the `object.attribute` notation used throughout Python scripts—most objects have useful attributes that are fetched with the "." operator. Some are callable things like functions, and others are simple data values that give object 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 (and module packages, described in Chapter 23) 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. For instance, if after coding the program in Figure 21-1 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 21-1. 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, roughly 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.

You will see a few of the standard library modules in action in this book's examples, but for a complete look you should browse the standard Python library reference manual, available either with your Python installation (via IDLE or the Python Start button menu on Windows) or online at *http://www.python.org*.

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, such as O'Reilly's *Programming Python*, but the manuals are free, viewable in any web browser (they ship in HTML format), and updated each time Python is rereleased.

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 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 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* suffix and without its directory path: it just says `import b`, instead of something like `import c:\dir1\b.py`. In fact, you can only list a simple name; path and suffix details are omitted on purpose and Python uses a standard *module search path* to locate the module file corresponding to an `import` statement.* 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 in Chapter 2.)

Python checks the file timestamps and, if the byte code file is older than the source file (i.e., if you've changed the source), automatically regenerates the byte code when the program is run. If, on the other hand, it finds a *.pyc* byte code file that is not older than the corresponding *.py* source file, 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

* It's actually syntactically illegal to include path and suffix details in a standard `import`. *Package imports*, which we'll discuss in Chapter 23, allow `import` statements to include part of the directory path leading to a file as a set of period-separated names; however, package imports still rely on the normal module search path to locate the leftmost directory in a package path (i.e., they are relative to a directory in the search path). They also cannot make use of any platform-specific directory syntax in the `import` statements; such syntax only works on the search path. Also, note that module file search path issues are not as relevant when you run *frozen executables* (discussed in Chapter 2); they typically embed byte code in the binary image.

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*. To learn how this works, watch for the discussion of the special `__name__` attribute and `__main__` in Chapter 24.

3. Run It

The final step of an import operation executes the byte code of the module. All statements in the file are executed in turn, from top to bottom, and any assignments made to names during this step generate attributes of the resulting module object. This execution step therefore generates all the tools that the module's code defines. 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 end-user customization), you have to force the issue with an `imp.reload` call—a tool we'll meet in the next chapter.†

The Module Search Path

As mentioned earlier, the part of the import procedure that is most important to programmers 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.

† As described earlier, Python keeps already imported modules in the built-in `sys.modules` dictionary so it can keep track of what's been loaded. In fact, if you want to see which modules are loaded, you can import `sys` and print `list(sys.modules.keys())`. More on other uses for this internal table in Chapter 24.

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 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)

Ultimately, the concatenation of these four components becomes `sys.path`, a 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
: 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.

`PYTHONPATH` *directories*
: Next, Python searches all directories listed in your `PYTHONPATH` environment variable setting, from left to right (assuming you have set this at all). In brief, `PYTHONPATH` is simply set to 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, described earlier) your imports will work without you needing to worry about this setting at all.

Standard library directories
: 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
: Finally, 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 them cover 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 3.0, a file named *myconfig.pth* may be placed at the top level of the Python install directory (*C:\Python30*) or in the *site-packages* subdirectory of the standard library there (*C:\Python30\Lib\site-packages*) to extend the module search path. On Unix-like systems, this file might be located in *usr/local/lib/python3.0/site-packages* or */usr/local/lib/site-python* instead.

  When 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. In fact, Python will collect the directory names in all the 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 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* directory so that user settings are not required (Python's `distutils` install system, described in an upcoming sidebar, automates many install steps).

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:\Python30\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 chapter. 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 and Python releases. Depending on your platform, additional directories may automatically be added to the module search path as well.

For instance, Python may add an entry for the *current working directory*—the directory from which you launched your program—in the search path after the PYTHONPATH directories, and before the standard library entries. 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). Because the current working directory can vary each time your program runs, you normally shouldn't depend on its value for import purposes. See Chapter 3 for more on launching programs from command lines.‡

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.

‡ See also Chapter 23's discussion of the new *relative import syntax* in Python 3.0; this modifies the search path for from statements in files inside packages when "." characters are used (e.g., from . import string). By default, a package's own directory is not automatically searched by imports in Python 3.0, unless relative imports are used by files in the package itself.

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 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 3.0, with my `PYTHONPATH` set to `C:\users` and a *C:\Python30\mypath.py* 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):

```
>>> import sys
>>> sys.path
['', 'C:\\users', 'C:\\Windows\\system32\\python30.zip', 'c:\\Python30\\DLLs',
'c:\\Python30\\lib', 'c:\\Python30\\lib\\plat-win', 'c:\\Python30',
'C:\\Users\\Mark', 'c:\\Python30\\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 later in this part of the book, by modifying the `sys.path` list, you can modify the search path for all future imports. Such changes only last for the duration of the script, however; `PYTHONPATH` and *.pth* files offer more permanent ways to modify the path.§

Module File Selection

Keep in mind that filename suffixes (e.g., *.py*) are intentionally omitted from `import` statements. Python chooses the first file it can find on the search path that matches the imported name. For example, an `import` statement of the form `import b` might load:

- A source code file named *b.py*
- A byte code file named *b.pyc*
- A directory named *b*, for package imports (described in Chapter 23)
- A compiled extension module, usually coded in C or C++ 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

§ Some programs really need to change `sys.path`, though. 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(dirname)` will often suffice.

- 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 transparent, 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 book are actually coded in C, not Python; because of this transparency, their clients don't have to care.

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 left-to-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. 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 more obvious.

Advanced Module Selection Concepts

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.

Python also supports the notion of *.pyo* optimized byte code files, created and run with the `-O` Python command-line flag; because these run only slightly faster than normal *.pyc* files (typically 5 percent faster), however, they are infrequently used. The Psyco system (see Chapter 2) provides more substantial speedups.

Third-Party Software: distutils

This chapter's description of module search path settings is targeted mainly at user-defined source code that you write on your own. Third-party extensions for Python typically use the `distutils` tools in the standard library to automatically install themselves, so no path configuration is required to use their code.

Systems that use `distutils` generally come with a *setup.py* script, which is run to install them; this script imports and uses `distutils` modules to place such systems in a directory that is automatically part of the module search path (usually in the *Lib\site-packages* subdirectory of the Python install tree, wherever that resides on the target machine).

For more details on distributing and installing with `distutils`, see the Python standard manual set; its use is beyond the scope of this book (for instance, it also provides ways to automatically compile C-coded extensions on the target machine). Also check out the emerging third-party open source *eggs* system, which adds dependency checking for installed Python software.

Chapter Summary

In this chapter, we covered the basics of modules, attributes, and imports and explored the operation of `import` statements. We learned that imports find the designated file on the module search path, compile it to byte code, and execute all of its statements to generate its contents. We also learned how to configure the search path to be able to import from directories other than the home directory and the standard library directories, primarily with `PYTHONPATH` settings.

As this chapter demonstrated, the import operation and modules are at the heart of program architecture in Python. Larger programs are divided into multiple files, which are linked together at runtime by imports. Imports in turn use the module search path to locate files, and modules define attributes for external use.

Of course, the whole point of imports and modules is to provide a structure to your program, which divides its logic into self-contained software components. Code in one module is isolated from code in another; in fact, no file can ever see the names defined in another, unless explicit `import` statements are run. Because of this, modules minimize name collisions between different parts of your program.

You'll see what this all means in terms of actual statements and code in the next chapter. Before we move on, though, let's run through the chapter quiz.

Test Your Knowledge: Quiz

1. How does a module source code file become a module object?
2. Why might you have to set your `PYTHONPATH` environment variable?
3. Name the four major components of the module import search path.
4. Name four file types that Python might load in response to an import operation.
5. What is a namespace, and what does a module's namespace contain?

Test Your Knowledge: Answers

1. A module's source code file automatically becomes a module object when that module is imported. Technically, the module's source code is run during the import, one statement at a time, and all the names assigned in the process become attributes of the module object.
2. You only need to set `PYTHONPATH` to import from directories other than the one in which you are working (i.e., the current directory when working interactively, or the directory containing your top-level file).
3. The four major components of the module import search path are the top-level script's home directory (the directory containing it), all directories listed in the `PYTHONPATH` environment variable, the standard library directories, and all directories listed in *.pth* path files located in standard places. Of these, programmers can customize `PYTHONPATH` and *.pth* files.
4. Python might load a source code (*.py*) file, a byte code (*.pyc*) file, a C extension module (e.g., a *.so* file on Linux or a *.dll* or *.pyd* file on Windows), or a directory of the same name for package imports. Imports may also load more exotic things such as ZIP file components, Java classes under the Jython version of Python, .NET components under IronPython, and statically linked C extensions that have no files present at all. With import hooks, imports can load anything.
5. A namespace is a self-contained package of variables, which are known as the *attributes* of the namespace object. A module's namespace contains all the names assigned by code at the top level of the module file (i.e., not nested in `def` or `class` statements). Technically, a module's global scope morphs into the module object's attributes namespace. A module's namespace may also be altered by assignments from other files that import it, though this is frowned upon (see Chapter 17 for more on this issue).

CHAPTER 22
Module Coding Basics

Now that we've looked at the larger ideas behind modules, let's turn to a simple example of modules in action. 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.

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)
```

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 outlined in Chapter 11. 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 (discussed in the next chapter) must conform to the rules for variable names presented in Chapter 11; 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.

As mentioned in the preceding chapter, it is also possible to create a Python module by writing code in an external language such as C or C++ (or 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 book; 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
>>> module1.printer('Hello world!')     # Qualify to get names
Hello world!
```

Because `import` 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` also copies 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 one variable
>>> printer('Hello world!')              # No need to qualify name
Hello world!
```

This 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.

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 out of the file.

The from * Statement

Finally, the next example uses a special form of `from`: when we use a `*`, we get copies of *all* the 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.

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.

In Python 3.0, the `from ...*` statement form described here can be used *only* at the top level of a module file, not within a function. Python 2.6 allows it to be used within a function, but issues a warning. It's extremely rare to see this statement used inside a function in practice; when present, it makes it impossible for Python to detect variables statically, before the function runs.

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.

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 chapter.

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, appear in function `def`s, and so on, and 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. Also, like `def`, `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 fetched name has no effect on the module from which it was copied, but changing a fetched *mutable object* can change it in the module from which it was imported. To illustrate, consider the following file, *small.py*:

```
x = 1
y = [1, 2]
```

```
% 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 name `y` in the importer and the importee 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]
```

For a graphical picture of what `from` assignments do with references, flip back to Figure 18-1 (function argument passing), and mentally replace "caller" and "function" with "imported" and "importer." 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 in Chapter 17. 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 book. Note that the change to `y[0]` in the prior session is different; it changes an object, not a name.

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 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. 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" on page 599 at the end of this part of the book (see Chapter 24).

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.

When import is required

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 only have one assignment to the name in your scope:

```
# O.py

from M import func
from N import func          # This overwites the one we got 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.

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*. We'll explore how all this works in this section.

Files Generate Namespaces

So, how do files morph into namespaces? The short story 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 explain 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 in Chapter 17, 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 (more formally, they follow the LEGB scope rule we met in Chapter 17, but without the L and E lookup layers). But, in modules, the module *scope* becomes an attribute dictionary of a module *object* after the module has been loaded. Unlike with functions (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.

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 0xO26D3BB8>

>>> 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.

Internally, module namespaces are stored as dictionary objects. These are just normal dictionary objects with the usual methods. We can access a module's namespace dictionary through the module's `__dict__` attribute (remember to wrap this in a `list` call in Python 3.0—it's a view object):

```
>>> list(module2.__dict__.keys())
['name', '__builtins__', '__file__', '__package__', 'sys', 'klass', 'func',
'__name__', '__doc__']
```

The names we assigned in the module file become dictionary keys internally, so most 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).

Attribute Name Qualification

Now that you're becoming more familiar with modules, we should look at the notion of name *qualification* (fetching attributes) in more depth. In Python, you can access the attributes of any object that has attributes using the qualification 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.

So, what does attribute qualification do to the scope rules we studied in Chapter 17? Nothing, really: 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 rule applies only to bare, unqualified names. Here are the rules:

Simple variables
: `X` means search for the name `X` in the current scopes (following the LEGB rule).

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 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. Using attribute qualification paths, it's 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:

* Some languages act differently and provide for *dynamic scoping*, where scopes really may depend on runtime calls. This tends to make code trickier, though, because the meaning of a variable can differ over time.

```
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 `X`s 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 in the next chapter. Package imports also create module namespace nesting, but their `import` statements are taken to reflect directory trees, not simple 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 all the fuss about reloading modules? The `reload` function allows parts of a program to be changed without stopping the whole program. With `reload`, therefore, 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. 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.

Version skew note: In Python 2.6, `reload` is available as a built-in function. In Python 3.0, it has been moved to the `imp` standard library module—it's known as `imp.reload` in 3.0. This simply means that an extra `import` or `from` statement is required to load this tool (in 3.0 only). Readers using 2.6 can ignore these imports in this book's examples, or use them anyhow—2.6 also has a `reload` in its `imp` module to ease migration to 3.0. Reloading works the same regardless of its packaging.

reload Basics

Unlike `import` and `from`:

- `reload` is a function in Python, not a statement.
- `reload` is passed an existing module object, not a name.
- `reload` lives in a module in Python 3.0 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: reloads require parentheses, but imports do not. 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 3.0)
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. When you call `reload`, Python rereads the module file's source code and reruns its top-level 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 a 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.

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 (see the sidebar "Why You Will Care: Module Reloads" on page 557), 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...
% vi 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/program...

>>> 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.

Why You Will Care: Module Reloads

Besides allowing you to reload (and hence rerun) modules at the interactive prompt, module reloads are also useful in larger systems, especially when the cost of restarting the entire application is prohibitive. For instance, systems that must connect to servers over a network on startup are prime candidates for dynamic reloads.

They're also useful in GUI work (a widget's callback action can be changed while the GUI remains active), and when Python is used as an embedded language in a C or C++ program (the enclosing program can request a reload of the Python code it runs, without having to stop). See *Programming Python* for more on reloading GUI callbacks and embedded Python code.

More generally, reloads allow programs to provide highly dynamic interfaces. For instance, Python is often used as a *customization* language for larger systems—users can customize products by coding bits of Python code onsite, without having to recompile the entire product (or even having its source code at all). In such worlds, the Python code already adds a dynamic flavor by itself.

To be even more dynamic, though, such systems can automatically reload the Python customization code periodically at runtime. That way, users' changes are picked up while the system is running; there is no need to stop and restart each time the Python code is modified. Not all systems require such a dynamic approach, but for those that do, module reloads provide an easy-to-use dynamic customization tool.

Chapter Summary

This chapter delved into the basics of module coding tools—the `import` and `from` statements, and the `reload` call. We learned how the `from` statement simply adds an extra step that copies names out of a file after it has been imported, and how `reload` forces a file to be imported again without stopping and restarting Python. We also surveyed namespace concepts, saw what happens when imports are nested, explored the way files become module namespaces, and learned about some potential pitfalls of the `from` statement.

Although we've already seen enough to handle module files in our programs, the next chapter extends our coverage of the import model by presenting *package imports*—a way for our `import` statements to specify part of the directory path leading to the desired module. As we'll see, package imports give us a hierarchy that is useful in larger systems and allow us to break conflicts between same-named modules. Before we move on, though, here's a quick quiz on the concepts presented here.

Test Your Knowledge: Quiz

1. How do you make a module?
2. How is the `from` statement related to the `import` statement?
3. How is the `reload` function related to imports?
4. When must you use `import` instead of `from`?
5. Name three potential pitfalls of the `from` statement.
6. What...is the airspeed velocity of an unladen swallow?

Test Your Knowledge: Answers

1. To create a module, you just write a text file containing Python statements; every source code file is automatically a module, and there is no syntax for declaring one. Import operations load module files into module objects in memory. You can also make a module by writing code in an external language like C or Java, but such extension modules are beyond the scope of this book.

2. The `from` statement imports an entire module, like the `import` statement, but as an extra step it also copies one or more variables from the imported module into the scope where the `from` appears. This enables you to use the imported names directly (`name`) instead of having to go through the module (`module.name`).
3. By default, a module is imported only once per process. The `reload` function forces a module to be imported again. It is mostly used to pick up new versions of a module's source code during development, and in dynamic customization scenarios.
4. You must use `import` instead of `from` only when you need to access the same name in two different modules; because you'll have to specify the names of the enclosing modules, the two names will be unique.
5. The `from` statement can obscure the meaning of a variable (which module it is defined in), can have problems with the `reload` call (names may reference prior versions of objects), and can corrupt namespaces (it might silently overwrite names you are using in your scope). The `from *` form is worse in most regards—it can seriously corrupt namespaces and obscure the meaning of variables, so it is probably best used sparingly.
6. What do you mean? An African or European swallow?

CHAPTER 23

Module 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 and extends the `from` statement for imports within packages.

Package Import Basics

So, how do package imports work? 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 machine, 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 Python module search path. In other words, the two `import` statements imply a directory structure that looks something like this (shown with DOS 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 generally, the leftmost component in a package import path is still relative to a directory included in the `sys.path` module search path list we met in Chapter 21. From there down, though, the `import` statements in your script give the directory paths leading to the modules explicitly.

Packages and Search Path Settings

If you use this feature, keep in mind that the directory paths in your `import` statements can only be 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 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*. Instead of using an invalid statement like this:

```
import C:\mycode\dir1\dir2\mod       # Error: illegal syntax
```

add *C:\mycode* to your `PYTHONPATH` variable or a *.pth* file (assuming it is not the program's home directory, in which case this step is not necessary), 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` statements. `import` statements provide directory path tails in a platform-neutral fashion.*

* The dot path syntax was chosen partly for platform neutrality, but also because paths in `import` statements become real nested object paths. This syntax also means that you get odd error messages if you forget to omit the *.py* in your `import` statements. For example, `import mod.py` is assumed to be a directory path import—it loads *mod.py*, then tries to load a *mod\py.py*, and ultimately issues a potentially confusing "No module named py" error message.

Package __init__.py Files

If you choose to use package imports, there is one more constraint you must follow: 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 (i.e., it must be the home directory, or be listed in your `PYTHONPATH`, etc.).

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. They are partly present as a declaration to Python, however, and can be completely empty. As declarations, 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 in an earlier directory on the search path.

More generally, the *__init__.py* file serves as a hook for package-initialization-time actions, 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 Python 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 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* file. Such files provide a namespace for module objects created for directories, which have no real associated module files.

`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 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__` in Chapter 24.)

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.

Don't confuse package *__init__.py* files with the class `__init__` constructor methods we'll meet in the next part of the book. The former are files of code run when `import`s first step through a package directory, while the latter are called when an instance is created. Both have initialization roles, but they are otherwise very different.

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 path names 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 a subdirectory of the one we're working in (i.e., the home directory), or a 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. As with 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:

```
% python
>>> import dir1.dir2.mod       # First imports run init files
dir1 init
dir2 init
in mod.py
>>>
>>> import dir1.dir2.mod       # Later imports do not
>>>
>>> from imp import reload     # Needed in 3.0
>>> reload(dir1)
dir1 init
<module 'dir1' from 'dir1\__init__.pyc'>
>>>
>>> reload(dir1.dir2)
dir2 init
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
```

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__.pyc'>
>>> dir1.dir2
<module 'dir1.dir2' from 'dir1\dir2\__init__.pyc'>
>>> dir1.dir2.mod
<module 'dir1.dir2.mod' from 'dir1\dir2\mod.pyc'>
```

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 `import`s may require many. The `import as` extension, discussed formally in the next chapter, can also help here by providing a shorter synonym for the full path:

```
% 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 (see Chapter 24)
>>> mod.z
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 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. The next section explores this role in more detail.

A Tale of Three Systems

The only time package imports are actually required is to resolve ambiguities that may arise when multiple programs with same-named files are installed on a single machine. This is something of an install issue, but it can also become a concern in general practice. Let's turn to a hypothetical scenario to illustrate.

Suppose that a programmer develops a Python program that contains a file called *utilities.py* for common utility code and a top-level file named *main.py* that users launch to start the program. All over this program, its files say `import utilities` to load and use the common code. When the program is shipped, it arrives as a single *.tar* or *.zip* file containing all the program's files, and when it is installed, it unpacks all its files into a single directory named *system1* on the target machine:

```
system1\
    utilities.py        # Common utility functions, classes
    main.py             # Launch this to start the program
    other.py            # Import utilities to load my tools
```

Now, suppose that a second programmer develops a different program with files also called *utilities.py* and *main.py*, and again uses `import utilities` throughout the program to load the common code file. When this second system is fetched and installed on the same computer as the first system, its files will unpack into a new directory called *system2* somewhere on the receiving machine (ensuring that they do not overwrite same-named files from the first system):

```
system2\
    utilities.py        # Common utilities
    main.py             # Launch this to run
    other.py            # Imports utilities
```

So far, there's no problem: both systems can coexist and run on the same machine. In fact, you won't even need to configure the module search path to use these programs on your computer—because Python always searches the home directory first (that is, the directory containing the top-level file), imports in either system's files will automatically see all the files in that system's directory. For instance, if you click on *system1\main.py*, all imports will search *system1* first. Similarly, if you launch

system2\main.py, *system2* will be searched first instead. Remember, module search path settings are only needed to import across directory boundaries.

However, suppose that after you've installed these two programs on your machine, you decide that you'd like to use some of the code in each of the *utilities.py* files in a system of your own. It's common utility code, after all, and Python code by nature wants to be reused. In this case, you want to be able to say the following from code that you're writing in a third directory to load one of the two files:

```
import utilities
utilities.func('spam')
```

Now the problem starts to materialize. To make this work at all, you'll have to set the module search path to include the directories containing the *utilities.py* files. But which directory do you put first in the path—*system1* or *system2*?

The problem is the *linear* nature of the search path. It is always scanned from left to right, so no matter how long you ponder this dilemma, you will always get *utilities.py* from the directory listed first (leftmost) on the search path. As is, you'll never be able to import it from the other directory at all. You could try changing `sys.path` within your script before each import operation, but that's both extra work and highly error prone. By default, you're stuck.

This is the issue that packages actually fix. Rather than installing programs as flat lists of files in standalone directories, you can package and install them as *subdirectories* under a common root. For instance, you might organize all the code in this example as an install hierarchy that looks like this:

```
root\
    system1\
        __init__.py
        utilities.py
        main.py
        other.py
    system2\
        __init__.py
        utilities.py
        main.py
        other.py
    system3\                    # Here or elsewhere
        __init__.py             # Your new code here
        myfile.py
```

Now, add just the common root directory to your search path. If your code's imports are all relative to this common root, you can import *either* system's utility file with a package import—the enclosing directory name makes the path (and hence, the module reference) unique. In fact, you can import *both* utility files in the same module, as long as you use an `import` statement and repeat the full path each time you reference the utility modules:

```
import system1.utilities
import system2.utilities
system1.utilities.function('spam')
system2.utilities.function('eggs')
```

The names of the enclosing directories here make the module references unique.

Note that you have to use `import` instead of `from` with packages only if you need to access the same attribute in two or more paths. If the name of the called function here was different in each path, `from` statements could be used to avoid repeating the full package path whenever you call one of the functions, as described earlier.

Also, notice in the install hierarchy shown earlier that *__init__.py* files were added to the *system1* and *system2* directories to make this work, but not to the *root* directory. Only directories listed within `import` statements in your code require these files; as you'll recall, they are run automatically the first time the Python process imports through a package directory.

Technically, in this case the *system3* directory doesn't have to be under *root*—just the packages of code from which you will import. However, because you never know when your own modules might be useful in other programs, you might as well place them under the common *root* directory as well to avoid similar name-collision problems in the future.

Finally, notice that both of the two original systems' imports will keep working unchanged. Because their *home* directories are searched first, the addition of the common root on the search path is irrelevant to code in *system1* and *system2*; they can keep saying just `import utilities` and expect to find their own files. Moreover, if you're careful to unpack all your Python systems under a common root like this, path configuration becomes simple: you'll only need to add the common root directory, once.

Package Relative Imports

The coverage of package imports so far has focused mostly on importing package files from *outside* the package. Within the package itself, imports of package files can use the same path syntax as outside imports, but they can also make use of special intra-package search rules to simplify `import` statements. That is, rather than listing package import paths, imports within the package can be relative to the package.

The way this works is version-dependent today: Python 2.6 implicitly searches package directories first on imports, while 3.0 requires explicit relative import syntax. This 3.0 change can enhance code readability, by making same-package imports more obvious. If you're starting out in Python with version 3.0, your focus in this section will likely be on its new import syntax. If you've used other Python packages in the past, though, you'll probably also be interested in how the 3.0 model differs.

Changes in Python 3.0

The way import operations in packages work has changed slightly in Python 3.0. This change applies only to imports within files located in the package directories we've been studying in this chapter; imports in other files work as before. For imports in packages, though, Python 3.0 introduces two changes:

- It modifies the module import search path semantics to skip the package's own directory by default. Imports check only other components of the search path. These are known as "absolute" imports.
- It extends the syntax of `from` statements to allow them to explicitly request that imports search the package's directory only. This is known as "relative" import syntax.

These changes are fully present in Python 3.0. The new `from` statement relative syntax is also available in Python 2.6, but the default search path change must be enabled as an option. It's currently scheduled to be added in the 2.7 release†—this change is being phased in this way because the search path portion is not backward compatible with earlier Pythons.

The impact of this change is that in 3.0 (and optionally in 2.6), you must generally use special `from` syntax to import modules located in the same package as the importer, unless you spell out a complete path from a package root. Without this syntax, your package is not automatically searched.

Relative Import Basics

In Python 3.0 and 2.6, `from` statements can now use leading dots (".") to specify that they require modules located within the same package (known as *package relative imports*), instead of modules located elsewhere on the module import search path (called *absolute imports*). That is:

- In both Python 3.0 and 2.6, you can use leading dots in `from` statements to indicate that imports should be *relative* to the containing package—such imports will search for modules inside the package only and will not look for same-named modules located elsewhere on the import search path (`sys.path`). The net effect is that package modules override outside modules.
- In Python 2.6, normal imports in a package's code (without leading dots) currently default to a relative-then-absolute search path order—that is, they search the package's own directory first. However, in Python 3.0, imports within a package are absolute by default—in the absence of any special dot syntax, imports skip the containing package itself and look elsewhere on the `sys.path` search path.

† Yes, there will be a 2.7 release, and possibly 2.8 and later releases, in parallel with new releases in the 3.X line. As described in the Preface, both the Python 2 and Python 3 lines are expected to be fully supported for years to come, to accommodate the large existing Python 2 user and code bases.

For example, in both Python 3.0 and 2.6, a statement of the form:

```
from . import spam                      # Relative to this package
```

instructs Python to import a module named `spam` located in the same package directory as the file in which this statement appears. Similarly, this statement:

```
from .spam import name
```

means "from a module named `spam` located in the same package as the file that contains this statement, import the variable `name`."

The behavior of a statement *without* the leading dot depends on which version of Python you use. In 2.6, such an import will still default to the current relative-then-absolute search path order (i.e., searching the package's directory first), unless a statement of the following form is included in the importing file:

```
from __future__ import  absolute_import    # Required until 2.7?
```

If present, this statement enables the Python 3.0 absolute-by-default default search path change, described in the next paragraph.

In 3.0, an import without a leading dot always causes Python to skip the relative components of the module import search path and look instead in the absolute directories that `sys.path` contains. For instance, in 3.0's model, a statement of the following form will always find a `string` module somewhere on `sys.path`, instead of a module of the same name in the package:

```
import string                           # Skip this package's version
```

Without the `from __future__` statement in 2.6, if there's a `string` module in the package, it will be imported instead. To get the same behavior in 3.0 and in 2.6 when the absolute import change is enabled, run a statement of the following form to force a relative import:

```
from . import string                    # Searches this package only
```

This works in both Python 2.6 and 3.0 today. The only difference in the 3.0 model is that it is *required* in order to load a module that is located in the same package directory as the file in which this appears, when the module is given with a simple name.

Note that leading dots can be used to force relative imports only with the `from` statement, not with the `import` statement. In Python 3.0, the `import modname` statement is always absolute, skipping the containing package's directory. In 2.6, this statement form still performs relative imports today (i.e., the package's directory is searched first), but these will become absolute in Python 2.7, too. `from` statements without leading dots behave the same as `import` statements—absolute in 3.0 (skipping the package directory), and relative-then-absolute in 2.6 (searching the package directory first).

Other dot-based relative reference patterns are possible, too. Within a module file located in a package directory named *mypkg*, the following alternative import forms work as described:

```
from .string import name1, name2          # Imports names from mypkg.string
from . import string                      # Imports mypkg.string
from .. import string                     # Imports string sibling of mypkg
```

To understand these latter forms better, we need to understand the rationale behind this change.

Why Relative Imports?

This feature is designed to allow scripts to resolve ambiguities that can arise when a same-named file appears in multiple places on the module search path. Consider the following package directory:

```
mypkg\
    __init__.py
    main.py
    string.py
```

This defines a package named `mypkg` containing modules named `mypkg.main` and `mypkg.string`. Now, suppose that the `main` module tries to import a module named `string`. In Python 2.6 and earlier, Python will first look in the *mypkg* directory to perform a *relative* import. It will find and import the *string.py* file located there, assigning it to the name `string` in the `mypkg.main` module's namespace.

It could be, though, that the intent of this import was to load the Python standard library's `string` module instead. Unfortunately, in these versions of Python, there's no straightforward way to ignore `mypkg.string` and look for the standard library's `string` module located on the module search path. Moreover, we cannot resolve this with package import paths, because we cannot depend on any extra package directory structure above the standard library being present on every machine.

In other words, imports in packages can be ambiguous—within a package, it's not clear whether an `import spam` statement refers to a module within or outside the package. More accurately, a local module or package can hide another hanging directly off of `sys.path`, whether intentionally or not.

In practice, Python users can avoid reusing the names of standard library modules they need for modules of their own (if you need the standard `string`, don't name a new module `string`!). But this doesn't help if a package accidentally hides a standard module; moreover, Python might add a new standard library module in the future that has the same name as a module of your own. Code that relies on relative imports is also less easy to understand, because the reader may be confused about which module is intended to be used. It's better if the resolution can be made explicit in code.

The relative imports solution in 3.0

To address this dilemma, imports run within packages have changed in Python 3.0 (and as an option in 2.6) to be absolute. Under this model, an `import` statement of the

following form in our example file *mypkg/main.py* will always find a `string` outside the package, via an absolute import search of `sys.path`:

```
import string                    # Imports string outside package
```

A `from` import without leading-dot syntax is considered absolute as well:

```
from string import name          # Imports name from string outside package
```

If you really want to import a module from your package without giving its full path from the package root, though, relative imports are still possible by using the dot syntax in the `from` statement:

```
from . import string             # Imports mypkg.string (relative)
```

This form imports the `string` module relative to the current package only and is the relative equivalent to the prior `import` example's absolute form; when this special relative syntax is used, the package's directory is the only directory searched.

We can also copy specific names from a module with relative syntax:

```
from .string import name1, name2     # Imports names from mypkg.string
```

This statement again refers to the `string` module relative to the current package. If this code appears in our `mypkg.main` module, for example, it will import `name1` and `name2` from `mypkg.string`.

In effect, the "." in a relative import is taken to stand for the package directory *containing* the file in which the import appears. An additional leading dot performs the relative import starting from the *parent* of the current package. For example, this statement:

```
from .. import spam              # Imports a sibling of mypkg
```

will load a sibling of `mypkg`—i.e., the `spam` module located in the package's own container directory, next to `mypkg`. More generally, code located in some module `A.B.C` can do any of these:

```
from . import D                  # Imports A.B.D    (. means A.B)
from .. import E                 # Imports A.E      (.. means A)

from .D import X                 # Imports A.B.D.X  (. means A.B)
from ..E import X                # Imports A.E.X    (.. means A)
```

Relative imports versus absolute package paths

Alternatively, a file can sometimes name its own package explicitly in an absolute import statement. For example, in the following, `mypkg` will be found in an absolute directory on `sys.path`:

```
from mypkg import string             # Imports mypkg.string (absolute)
```

However, this relies on both the configuration and the order of the module search path settings, while relative import dot syntax does not. In fact, this form requires that the directory immediately containing `mypkg` be included in the module search path. In

general, absolute import statements must list all the directories below the package's root entry in `sys.path` when naming packages explicitly like this:

```
from system.section.mypkg import string      # system container on sys.path only
```

In large or deep packages, that could be much more work than a dot:

```
from . import string                         # Relative import syntax
```

With this latter form, the containing package is searched automatically, regardless of the search path settings.

The Scope of Relative Imports

Relative imports can seem a bit perplexing on first encounter, but it helps if you remember a few key points about them:

- **Relative imports apply to imports within packages only**. Keep in mind that this feature's module search path change applies only to `import` statements within module files located in a package. Normal imports coded outside package files still work exactly as described earlier, automatically searching the directory containing the top-level script first.
- **Relative imports apply to the from statement only**. Also remember that this feature's new syntax applies only to `from` statements, not `import` statements. It's detected by the fact that the module name in a `from` begins with one or more dots (periods). Module names that contain dots but don't have a leading dot are package imports, not relative imports.
- **The terminology is ambiguous**. Frankly, the terminology used to describe this feature is probably more confusing than it needs to be. Really, all imports are relative to something. Outside a package, imports are still relative to directories listed on the `sys.path` module search path. As we learned in Chapter 21, this path includes the program's container directory, `PYTHONPATH` settings, path file settings, and standard libraries. When working interactively, the program container directory is simply the current working directory.

  For imports made inside packages, 2.6 augments this behavior by searching the package itself first. In the 3.0 model, all that really changes is that normal "absolute" import syntax skips the package directory, but special "relative" import syntax causes it to be searched first and only. When we talk about 3.0 imports as being "absolute," what we really mean is that they are relative to the directories on `sys.path`, but not the package itself. Conversely, when we speak of "relative" imports, we mean they are relative to the package directory only. Some `sys.path` entries could, of course, be absolute or relative paths too. (And I could probably make up something more confusing, but it would be a stretch!)

In other words, "package relative imports" in 3.0 really just boil down to a removal of 2.6's special search path behavior for packages, along with the addition of special `from` syntax to explicitly request relative behavior. If you wrote your package imports in the past to not depend on 2.6's special implicit relative lookup (e.g., by always spelling out full paths from a package root), this change is largely a moot point. If you didn't, you'll need to update your package files to use the new `from` syntax for local package files.

Module Lookup Rules Summary

With packages and relative imports, the module search story in Python 3.0 in its entirety can be summarized as follows:

- Simple module names (e.g., `A`) are looked up by searching each directory on the `sys.path` list, from left to right. This list is constructed from both system defaults and user-configurable settings.
- Packages are simply directories of Python modules with a special *__init__.py* file, which enables `A.B.C` directory path syntax in imports. In an import of `A.B.C`, for example, the directory named `A` is located relative to the normal module import search of `sys.path`, `B` is another package subdirectory within `A`, and `C` is a module or other importable item within `B`.
- Within a package's files, normal `import` statements use the same `sys.path` search rule as imports elsewhere. Imports in packages using `from` statements and leading dots, however, are relative to the package; that is, only the package directory is checked, and the normal `sys.path` lookup is not used. In `from . import A`, for example, the module search is restricted to the directory containing the file in which this statement appears.

Relative Imports in Action

But enough theory: let's run some quick tests to demonstrate the concepts behind relative imports.

Imports outside packages

First of all, as mentioned previously, this feature does not impact imports outside a package. Thus, the following finds the standard library `string` module as expected:

```
C:\test> c:\Python30\python
>>> import string
>>> string
<module 'string' from 'c:\Python30\lib\string.py'>
```

But if we add a module of the same name in the directory we're working in, it is selected instead, because the first entry on the module search path is the current working directory (CWD):

```
# test\string.py
print('string' * 8)

C:\test> c:\Python30\python
>>> import string
stringstringstringstringstringstringstringstring
>>> string
<module 'string' from 'string.py'>
```

In other words, normal imports are still relative to the "home" directory (the top-level script's container, or the directory you're working in). In fact, relative import syntax is not even allowed in code that is not in a file being used as part of a package:

```
>>> from . import string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Attempted relative import in non-package
```

In this and all examples in this section, code entered at the interactive prompt behaves the same as it would if run in a top-level script, because the first entry on `sys.path` is either the interactive working directory or the directory containing the top-level file. The only difference is that the start of `sys.path` is an absolute directory, not an empty string:

```
# test\main.py
import string
print(string)

C:\test> C:\python30\python main.py                    # Same results in 2.6
stringstringstringstringstringstringstringstring
<module 'string' from 'C:\test\string.py'>
```

Imports within packages

Now, let's get rid of the local `string` module we coded in the CWD and build a package directory there with two modules, including the required but empty *test\pkg__init__.py* file (which I'll omit here):

```
C:\test> del string*
C:\test> mkdir pkg

# test\pkg\spam.py
import eggs                    # <== Works in 2.6 but not 3.0!
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)
```

The first file in this package tries to import the second with a normal `import` statement. Because this is taken to be relative in 2.6 but absolute in 3.0, it fails in the latter. That is, 2.6 searches the containing package first, but 3.0 does not. This is the noncompatible behavior you have to be aware of in 3.0:

```
C:\test> c:\Python26\python
>>> import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test> c:\Python30\python
>>> import pkg.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pkg\spam.py", line 1, in <module>
    import eggs
ImportError: No module named eggs
```

To make this work in both 2.6 and 3.0, change the first file to use the special relative import syntax, so that its import searches the package directory in 3.0, too:

```
# test\pkg\spam.py
from . import eggs              # <== Use package relative import in 2.6 or 3.0
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string
print(string)

C:\test> c:\Python26\python
>>> import pkg.spam
<module 'string' from 'c:\Python26\lib\string.pyc'>
99999

C:\test> c:\Python30\python
>>> import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>
99999
```

Imports are still relative to the CWD

Notice in the preceding example that the package modules still have access to standard library modules like `string`. Really, their imports are still relative to the entries on the module search path, even if those entries are relative themselves. If you add a `string` module to the CWD again, imports in a package will find it there instead of in the standard library. Although you can skip the package directory with an absolute import in 3.0, you still can't skip the home directory of the program that imports the package:

```
# test\string.py
print('string' * 8)

# test\pkg\spam.py
from . import eggs
```

```
print(eggs.X)

# test\pkg\eggs.py
X = 99999
import string                   # <== Gets string in CWD, not Python lib!
print(string)

C:\test> c:\Python30\python     # Same result in 2.6
>>> import pkg.spam
stringstringstringstringstringstringstringstringstring
<module 'string' from 'string.py'>
99999
```

Selecting modules with relative and absolute imports

To show how this applies to imports of standard library modules, reset the package one more time. Get rid of the local `string` module, and define a new one inside the package itself:

```
C:\test> del string*

# test\pkg\spam.py
import string                   # <== Relative in 2.6, absolute in 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

Now, which version of the `string` module you get depends on which Python you use. As before, 3.0 interprets the import in the first file as absolute and skips the package, but 2.6 does not:

```
C:\test> c:\Python30\python
>>> import pkg.spam
<module 'string' from 'c:\Python30\lib\string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

Using relative import syntax in 3.0 forces the package to be searched again, as it is in 2.6—by using absolute or relative import syntax in 3.0, you can either skip or select the package directory explicitly. In fact, this is the use case that the 3.0 model addresses:

```
# test\pkg\spam.py
from . import string            # <== Relative in both 2.6 and 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)

C:\test> c:\Python30\python
>>> import pkg.spam
NiNiNiNiNiNiNiNi
```

```
<module 'pkg.string' from 'pkg\string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

It's important to note that relative import syntax is really a binding declaration, not just a preference. If we delete the *string.py* file in this example, the relative import in *spam.py fails* in both 3.0 and 2.6, instead of falling back on the standard library's version of this module (or any other):

```
# test\pkg\spam.py
from . import string            # <== Fails if no string.py here!

C:\test> C:\python30\python
>>> import pkg.spam
...text omitted...
ImportError: cannot import name string
```

Modules referenced by relative imports must exist in the package directory.

Imports are still relative to the CWD (again)

Although absolute imports let you skip package modules, they still rely on other components of `sys.path`. For one last test, let's define two `string` modules of our own. In the following, there is one module by that name in the CWD, one in the package, and another in the standard library:

```
# test\string.py
print('string' * 8)

# test\pkg\spam.py
from . import string            # <== Relative in both 2.6 and 3.0
print(string)

# test\pkg\string.py
print('Ni' * 8)
```

When we import the `string` module with relative import syntax, we get the version in the package, as desired:

```
C:\test> c:\Python30\python     # Same result in 2.6
>>> import pkg.spam
NiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.py'>
```

When absolute syntax is used, though, the module we get varies per version again. 2.6 interprets this as relative to the package, but 3.0 makes it "absolute," which in this case really just means it skips the package and loads the version relative to the CWD (*not* the version the standard library):

```
# test\string.py
print('string' * 8)
```

```
# test\pkg\spam.py
import string                   # <== Relative in 2.6, "absolute" in 3.0: CWD!
print(string)

# test\pkg\string.py
print('Ni' * 8)

C:\test> c:\Python30\python
>>> import pkg.spam
stringstringstringstringstringstringstringstring
<module 'string' from 'string.py'>

C:\test> c:\Python26\python
>>> import pkg.spam
NiNiNiNiNiNiNiNi
<module 'pkg.string' from 'pkg\string.pyc'>
```

As you can see, although packages can explicitly request modules within their own directories, their imports are otherwise still relative to the rest of the normal module search path. In this case, a file in the program using the package hides the standard library module the package may want. All that the change in 3.0 really accomplishes is allowing package code to select files either inside or outside the package (i.e., relatively or absolutely). Because import resolution can depend on an enclosing context that may not be foreseen, absolute imports in 3.0 are not a guarantee of finding a module in the standard library.

Experiment with these examples on your own for more insight. In practice, this is not usually as ad-hoc as it might seem: you can generally structure your imports, search paths, and module names to work the way you wish during development. You should keep in mind, though, that imports in larger systems may depend upon context of use, and the module import protocol is part of a successful library's design.

Now that you've learned about package-relative imports, also keep in mind that they may not always be your best option. Absolute package imports, relative to a directory on `sys.path`, are still sometimes preferred over both implicit package-relative imports in Python 2, and explicit package-relative import syntax in both Python 2 and 3.

Package-relative import syntax and Python 3.0's new absolute import search rules at least require relative imports from a package to be made explicit, and thus easier to understand and maintain. Files that use imports with dots, though, are implicitly bound to a package directory and cannot be used elsewhere without code changes.

Naturally, the extent to which this may impact your modules can vary per package; absolute imports may also require changes when directories are reorganized.

Why You Will Care: Module Packages

Now that packages are a standard part of Python, it's common to see larger third-party extensions shipped as sets of package directories, rather than flat lists of modules. The *win32all* Windows extensions package for Python, for instance, was one of the first to jump on the package bandwagon. Many of its utility modules reside in packages imported with paths. For instance, to load client-side COM tools, you use a statement like this:

```
from win32com.client import constants, Dispatch
```

This line fetches names from the `client` module of the `win32com` package (an install subdirectory).

Package imports are also pervasive in code run under the Jython Java-based implementation of Python, because Java libraries are organized into hierarchies as well. In recent Python releases, the email and XML tools are likewise organized into package subdirectories in the standard library, and Python 3.0 groups even more related modules into packages (including tkinter GUI tools, HTTP networking tools, and more). The following imports access various standard library tools in 3.0:

```
from email.message import Message
from tkinter.filedialog import askopenfilename
from http.server import CGIHTTPRequestHandler
```

Whether you create package directories or not, you will probably import from them eventually.

Chapter Summary

This chapter introduced Python's package import model—an optional but useful way to explicitly list part of the directory path leading up to your modules. Package imports are still relative to a directory on your module import search path, but rather than relying on Python to traverse the search path manually, your script gives the rest of the path to the module explicitly.

As we've seen, packages not only make imports more meaningful in larger systems, but also simplify import search path settings (if all cross-directory imports are relative to a common root directory) and resolve ambiguities when there is more than one module of the same name (including the name of the enclosing directory in a package import helps distinguish between them).

Because it's relevant only to code in packages, we also explored the newer relative import model here—a way for imports in package files to select modules in the same package using leading dots in a `from`, instead of relying on an older implicit package search rule.

In the next chapter, we will survey a handful of more advanced module-related topics, such as relative import syntax and the `__name__` usage mode variable. As usual, though, we'll close out this chapter with a short quiz to test what you've learned here.

Test Your Knowledge: Quiz

1. What is the purpose of an *__init__.py* file in a module package directory?
2. How can you avoid repeating the full package path every time you reference a package's content?
3. Which directories require *__init__.py* files?
4. When must you use `import` instead of `from` with packages?
5. What is the difference between `from mypkg import spam` and `from . import spam`?

Test Your Knowledge: Answers

1. The *__init__.py* file serves to declare and initialize a module package; Python automatically runs its code the first time you import through a directory in a process. Its assigned variables become the attributes of the module object created in memory to correspond to that directory. It is also not optional—you can't import through a directory with package syntax unless it contains this file.
2. Use the `from` statement with a package to copy names out of the package directly, or use the `as` extension with the `import` statement to rename the path to a shorter synonym. In both cases, the path is listed in only one place, in the `from` or `import` statement.
3. Each directory listed in an `import` or `from` statement must contain an *__init__.py* file. Other directories, including the directory containing the leftmost component of a package path, do not need to include this file.
4. You must use `import` instead of `from` with packages only if you need to access the same name defined in more than one path. With `import`, the path makes the references unique, but `from` allows only one version of any given name.
5. `from mypkg import spam` is an *absolute* import—the search for `mypkg` skips the package directory and the module is located in an absolute directory in `sys.path`. A statement `from . import spam`, on the other hand, is a *relative* import—`spam` is looked up relative to the package in which this statement is contained before `sys.path` is searched.