CHAPTER 16
Function Basics

In Part III, we studied basic procedural statements in Python. Here, we'll move on to explore a set of additional statements and expressions that we can use to create functions of our own.

In simple terms, a *function* is a device that groups a set of statements so they can be run more than once in a program—a packaged procedure invoked by name. Functions also can compute a result value and let us specify parameters that serve as function inputs and may differ each time the code is run. Coding an operation as a function makes it a generally useful tool, which we can use in a variety of contexts.

More fundamentally, functions are the alternative to programming by *cutting and pasting*—rather than having multiple redundant copies of an operation's code, we can factor it into a single function. In so doing, we reduce our future work radically: if the operation must be changed later, we have only one copy to update in the function, not many scattered throughout the program.

Functions are also the most basic program structure Python provides for maximizing *code reuse*, and lead us to the larger notions of program *design*. As we'll see, functions let us split complex systems into manageable parts. By implementing each part as a function, we make it both reusable and easier to code.

Table 16-1 previews the primary function-related tools we'll study in this part of the book—a set that includes call expressions, two ways to make functions (`def` and `lambda`), two ways to manage scope visibility (`global` and `nonlocal`), and two ways to send results back to callers (`return` and `yield`).

Table 16-1. Function-related statements and expressions

Statement or expression	Examples
Call expressions	`myfunc('spam', 'eggs', meat=ham, *rest)`
`def`	`def printer(messge):` `  print('Hello ' + message)`
`return`	`def adder(a, b=1, *c):` `  return a + b + c[0]`

Statement or expression	Examples
global	x = 'old' def changer(): global x; x = 'new'

nonlocal (3.X)	def outer():
	  x = 'old'
	  def changer():
	  nonlocal x; x = 'new'

yield	def squares(x): for i in range(x): yield i ** 2
lambda	funcs = [lambda x: x**2, lambda x: x**3]

Why Use Functions?

Before we get into the details, let's establish a clear picture of what functions are all about. Functions are a nearly universal program-structuring device. You may have come across them before in other languages, where they may have been called *subroutines* or *procedures*. As a brief introduction, functions serve two primary development roles:

Maximizing code reuse and minimizing redundancy

As in most programming languages, Python functions are the simplest way to package logic you may wish to use in more than one place and more than one time. Up until now, all the code we've been writing has run immediately. Functions allow us to group and generalize code to be used arbitrarily many times later. Because they allow us to code an operation in a single place and use it in many places, Python functions are the most basic *factoring* tool in the language: they allow us to reduce code redundancy in our programs, and thereby reduce maintenance effort.

Procedural decomposition

Functions also provide a tool for splitting systems into pieces that have well-defined roles. For instance, to make a pizza from scratch, you would start by mixing the dough, rolling it out, adding toppings, baking it, and so on. If you were programming a pizza-making robot, functions would help you divide the overall "make pizza" task into chunks—one function for each subtask in the process. It's easier to implement the smaller tasks in isolation than it is to implement the entire process at once. In general, functions are about *procedure*—how to do something, rather than what you're doing it to. We'll see why this distinction matters in Part VI, when we start making new objects with classes.

In this part of the book, we'll explore the tools used to code functions in Python: function basics, scope rules, and argument passing, along with a few related concepts such as generators and functional tools. Because its importance begins to become more apparent at this level of coding, we'll also revisit the notion of polymorphism, which was

introduced earlier in the book. As you'll see, functions don't imply much new syntax, but they do lead us to some bigger programming ideas.

Coding Functions

Although it wasn't made very formal, we've already used some functions in earlier chapters. For instance, to make a file object, we called the built-in `open` function; similarly, we used the `len` built-in function to ask for the number of items in a collection object.

In this chapter, we will explore how to write *new* functions in Python. Functions we write behave the same way as the built-ins we've already seen: they are called in expressions, are passed values, and return results. But writing new functions requires the application of a few additional ideas that haven't yet been introduced. Moreover, functions behave very differently in Python than they do in compiled languages like C. Here is a brief introduction to the main concepts behind Python functions, all of which we will study in this part of the book:

- **`def` is executable code**. Python functions are written with a new statement, the `def`. Unlike functions in compiled languages such as C, `def` is an executable statement—your function does not exist until Python reaches and runs the `def`. In fact, it's legal (and even occasionally useful) to nest `def` statements inside `if` statements, `while` loops, and even other `def`s. In typical operation, `def` statements are coded in module files and are naturally run to generate functions when the module file they reside in is first imported.
- **`def` creates an object and assigns it to a name**. When Python reaches and runs a `def` statement, it generates a new function object and assigns it to the function's name. As with all assignments, the function name becomes a reference to the function object. There's nothing magic about the name of a function—as you'll see, the function object can be assigned to other names, stored in a list, and so on. Function objects may also have arbitrary user-defined *attributes* attached to them to record data.
- **`lambda` creates an object but returns it as a result**. Functions may also be created with the `lambda` expression, a feature that allows us to *in-line* function definitions in places where a `def` statement won't work syntactically. This is a more advanced concept that we'll defer until Chapter 19.
- **`return` sends a result object back to the caller**. When a function is called, the caller stops until the function finishes its work and returns control to the caller. Functions that compute a value send it back to the caller with a `return` statement; the returned value becomes the result of the function call. A `return` without a value simply returns to the caller (and sends back `None`, the default result).
- **`yield` sends a result object back to the caller, but remembers where it left off**. Functions known as *generators* may also use the `yield` statement to send back

a value and suspend their state such that they may be resumed later, to produce a series of results over time. This is another advanced topic covered later in this part of the book.

- **`global` declares module-level variables that are to be assigned**. By default, all names assigned in a function are local to that function and exist only while the function runs. To assign a name in the enclosing module, functions need to list it in a `global` statement. More generally, names are always looked up in *scopes*—places where variables are stored—and assignments bind names to scopes.
- **`nonlocal` declares enclosing function variables that are to be assigned**. Similarly, the `nonlocal` statement added in Python 3.X allows a function to assign a name that exists in the scope of a syntactically enclosing `def` statement. This allows enclosing functions to serve as a place to retain *state*—information remembered between function calls—without using shared global names.
- **Arguments are passed by assignment (object reference)**. In Python, arguments are passed to functions by assignment (which, as we've learned, means by object reference). As you'll see, in Python's model the caller and function share objects by references, but there is no name aliasing. Changing an argument name within a function does not also change the corresponding name in the caller, but changing passed-in mutable objects in place can change objects shared by the caller, and serve as a function result.
- **Arguments are passed by position, unless you say otherwise**. Values you pass in a function call match argument names in a function's definition from left to right by default. For flexibility, function *calls* can also pass arguments by name with `name=value` keyword syntax, and unpack arbitrarily many arguments to send with `*pargs` and `**kargs` starred-argument notation. Function *definitions* use the same two forms to specify argument defaults, and collect arbitrarily many arguments received.
- **Arguments, return values, and variables are not declared**. As with everything in Python, there are no type constraints on functions. In fact, nothing about a function needs to be declared ahead of time: you can pass in arguments of any type, return any kind of object, and so on. As one consequence, a single function can often be applied to a variety of object types—any objects that sport a compatible *interface* (methods and expressions) will do, regardless of their specific types.

If some of the preceding words didn't sink in, don't worry—we'll explore all of these concepts with real code in this part of the book. Let's get started by expanding on some of these ideas and looking at a few examples.

def Statements

The `def` statement creates a function object and assigns it to a name. Its general format is as follows:

```
def name(arg1, arg2,... argN):
    statements
```

As with all compound Python statements, `def` consists of a header line followed by a block of statements, usually indented (or a simple statement after the colon). The statement block becomes the function's *body*—that is, the code Python executes each time the function is later called.

The `def` header line specifies a function *name* that is assigned the function object, along with a list of zero or more *arguments* (sometimes called *parameters*) in parentheses. The argument names in the header are assigned to the objects passed in parentheses at the point of call.

Function bodies often contain a `return` statement:

```
def name(arg1, arg2,... argN):
    ...
    return value
```

The Python `return` statement can show up anywhere in a function body; when reached, it ends the function call and sends a result back to the caller. The `return` statement consists of an optional object value expression that gives the function's result. If the value is omitted, `return` sends back a `None`.

The `return` statement itself is optional too; if it's not present, the function exits when the control flow falls off the end of the function body. Technically, a function without a `return` statement also returns the `None` object automatically, but this return value is usually ignored at the call.

Functions may also contain `yield` statements, which are designed to produce a series of values over time, but we'll defer discussion of these until we survey generator topics in Chapter 20.

def Executes at Runtime

The Python `def` is a true executable statement: when it runs, it creates a new function object and assigns it to a name. (Remember, all we have in Python is *runtime*; there is no such thing as a separate compile time.) Because it's a statement, a `def` can appear anywhere a statement can—even nested in other statements. For instance, although `def`s normally are run when the module enclosing them is imported, it's also completely legal to nest a function `def` inside an `if` statement to select between alternative definitions:

```
if test:
    def func():             # Define func this way
        ...
else:
    def func():             # Or else this way
        ...
...
func()                      # Call the version selected and built
```

One way to understand this code is to realize that the `def` is much like an `=` statement: it simply assigns a name at runtime. Unlike in compiled languages such as C, Python functions do not need to be fully defined before the program runs. More generally, `def`s are not evaluated until they are reached and run, and the code *inside* `def`s is not evaluated until the functions are later called.

Because function definition happens at runtime, there's nothing special about the function name. What's important is the object to which it refers:

```
othername = func          # Assign function object
othername()               # Call func again
```

Here, the function was assigned to a different name and called through the new name. Like everything else in Python, functions are just *objects*; they are recorded explicitly in memory at program execution time. In fact, besides calls, functions allow arbitrary *attributes* to be attached to record information for later use:

```
def func(): ...           # Create function object
func()                    # Call object
func.attr = value         # Attach attributes
```

A First Example: Definitions and Calls

Apart from such runtime concepts (which tend to seem most unique to programmers with backgrounds in traditional compiled languages), Python functions are straightforward to use. Let's code a first real example to demonstrate the basics. As you'll see, there are two sides to the function picture: a *definition* (the `def` that creates a function) and a *call* (an expression that tells Python to run the function's body).

Definition

Here's a definition typed interactively that defines a function called `times`, which returns the product of its two arguments:

```
>>> def times(x, y):      # Create and assign function
...     return x * y       # Body executed when called
...
```

When Python reaches and runs this `def`, it creates a new function object that packages the function's code and assigns the object to the name `times`. Typically, such a statement is coded in a module file and runs when the enclosing file is imported; for something this small, though, the interactive prompt suffices.

Calls

The `def` statement makes a function but does not call it. After the `def` has run, you can call (run) the function in your program by adding parentheses after the function's name.

The parentheses may optionally contain one or more object arguments, to be passed (assigned) to the names in the function's header:

```
>>> times(2, 4)               # Arguments in parentheses
8
```

This expression passes two arguments to `times`. As mentioned previously, arguments are passed by assignment, so in this case the name `x` in the function header is assigned the value `2`, `y` is assigned the value `4`, and the function's body is run. For this function, the body is just a `return` statement that sends back the result as the value of the call expression. The returned object was printed here interactively (as in most languages, `2 * 4` is `8` in Python), but if we needed to use it later we could instead assign it to a variable. For example:

```
>>> x = times(3.14, 4)        # Save the result object
>>> x
12.56
```

Now, watch what happens when the function is called a third time, with very different kinds of objects passed in:

```
>>> times('Ni', 4)            # Functions are "typeless"
'NiNiNiNi'
```

This time, our function means something completely different (Monty Python reference again intended). In this third call, a string and an integer are passed to `x` and `y`, instead of two numbers. Recall that * works on both numbers and sequences; because we never declare the types of variables, arguments, or return values in Python, we can use `times` to either *multiply* numbers or *repeat* sequences.

In other words, what our `times` function means and does depends on what we pass into it. This is a core idea in Python (and perhaps the key to using the language well), which merits a bit of expansion here.

Polymorphism in Python

As we just saw, the very meaning of the expression `x * y` in our simple `times` function depends completely upon the kinds of objects that `x` and `y` are—thus, the same function can perform multiplication in one instance and repetition in another. Python leaves it up to the *objects* to do something reasonable for the syntax. Really, * is just a dispatch mechanism that routes control to the objects being processed.

This sort of type-dependent behavior is known as *polymorphism*, a term we first met in Chapter 4 that essentially means that the meaning of an operation depends on the objects being operated upon. Because it's a dynamically typed language, polymorphism runs rampant in Python. In fact, *every* operation is a polymorphic operation in Python: printing, indexing, the * operator, and much more.

This is deliberate, and it accounts for much of the language's conciseness and flexibility. A single function, for instance, can generally be applied to a whole category of object

types automatically. As long as those objects support the expected *interface* (a.k.a. protocol), the function can process them. That is, if the objects passed into a function have the expected methods and expression operators, they are plug-and-play compatible with the function's logic.

Even in our simple `times` function, this means that *any* two objects that support a * will work, no matter what they may be, and no matter when they are coded. This function will work on two numbers (performing multiplication), or a string and a number (performing repetition), or any other combination of objects supporting the expected interface—even class-based objects we have not even imagined yet.

Moreover, if the objects passed in do *not* support this expected interface, Python will detect the error when the * expression is run and raise an exception automatically. It's therefore usually pointless to code error checking ourselves. In fact, doing so would limit our function's utility, as it would be restricted to work only on objects whose types we test for.

This turns out to be a crucial philosophical difference between Python and statically typed languages like C++ and Java: in Python, your code is *not supposed to care* about specific data types. If it does, it will be limited to working on just the types you anticipated when you wrote it, and it will not support other compatible object types that may be coded in the future. Although it is possible to test for types with tools like the `type` built-in function, doing so breaks your code's flexibility. By and large, we code to object *interfaces* in Python, not data types.[1]

Of course, some programs have unique requirements, and this polymorphic model of programming means we have to test our code to detect errors, rather than providing type declarations a compiler can use to detect some types of errors for us ahead of time. In exchange for an initial bit of testing, though, we radically reduce the amount of code we have to write and radically increase our code's flexibility. As you'll learn, it's a net win in practice.

A Second Example: Intersecting Sequences

Let's look at a second function example that does something a bit more useful than multiplying arguments and further illustrates function basics.

In Chapter 13, we coded a `for` loop that collected items held in common in two strings. We noted there that the code wasn't as useful as it could be because it was set up to work only on specific variables and could not be rerun later. Of course, we could copy

1. This polymorphic behavior has in recent years come to also be known as *duck typing*—the essential idea being that your code is not supposed to care if an object is a *duck*, only that it *quacks*. Anything that quacks will do, duck or not, and the implementation of quacks is up to the object, a principle which will become even more apparent when we study classes in Part VI. Graphic metaphor to be sure, though this is really just a new label for an older idea, and use cases for quacking software would seem limited in the tangible world (he says, bracing for emails from militant ornithologists...).

the code and paste it into each place where it needs to be run, but this solution is neither good nor general—we'd still have to edit each copy to support different sequence names, and changing the algorithm would then require changing multiple copies.

Definition

By now, you can probably guess that the solution to this dilemma is to package the `for` loop inside a function. Doing so offers a number of advantages:

- Putting the code in a function makes it a tool that you can run as many times as you like.
- Because callers can pass in arbitrary arguments, functions are general enough to work on any two sequences (or other iterables) you wish to intersect.
- When the logic is packaged in a function, you have to change code in only one place if you ever need to change the way the intersection works.
- Coding the function in a module file means it can be imported and reused by any program run on your machine.

In effect, wrapping the code in a function makes it a general intersection utility:

```
def intersect(seq1, seq2):
    res = []                    # Start empty
    for x in seq1:              # Scan seq1
        if x in seq2:           # Common item?
            res.append(x)       # Add to end
    return res
```

The transformation from the simple code of Chapter 13 to this function is straightforward; we've just nested the original logic under a `def` header and made the objects on which it operates passed-in parameter names. Because this function computes a result, we've also added a `return` statement to send a result object back to the caller.

Calls

Before you can call a function, you have to make it. To do this, run its `def` statement, either by typing it interactively or by coding it in a module file and importing the file. Once you've run the `def`, you can call the function by passing any two sequence objects in parentheses:

```
>>> s1 = "SPAM"
>>> s2 = "SCAM"
>>> intersect(s1, s2)           # Strings
['S', 'A', 'M']
```

Here, we've passed in two strings, and we get back a list containing the characters in common. The algorithm the function uses is simple: "for every item in the first argument, if that item is also in the second argument, append the item to the result." It's a little shorter to say that in Python than in English, but it works out the same.

To be fair, our intersect function is fairly slow (it executes nested loops), isn't really mathematical intersection (there may be duplicates in the result), and isn't required at all (as we've seen, Python's set data type provides a built-in intersection operation). Indeed, the function could be replaced with a single list comprehension expression, as it exhibits the classic loop collector code pattern:

```
>>> [x for x in s1 if x in s2]
['S', 'A', 'M']
```

As a function basics example, though, it does the job—this single piece of code can apply to an entire range of object types, as the next section explains. In fact, we'll improve and extend this to support arbitrarily many operands in Chapter 18, after we learn more about argument passing modes.

Polymorphism Revisited

Like all good functions in Python, `intersect` is polymorphic. That is, it works on arbitrary types, as long as they support the expected object interface:

```
>>> x = intersect([1, 2, 3], (1, 4))      # Mixed types
>>> x                                     # Saved result object
[1]
```

This time, we passed in different types of objects to our function—a list and a tuple (mixed types)—and it still picked out the common items. Because you don't have to specify the types of arguments ahead of time, the `intersect` function happily iterates through any kind of sequence objects you send it, as long as they support the expected interfaces.

For `intersect`, this means that the first argument has to support the `for` loop, and the second has to support the `in` membership test. Any two such objects will work, regardless of their specific types—that includes physically stored sequences like strings and lists; all the iterable objects we met in Chapter 14, including files and dictionaries; and even any class-based objects we code that apply operator overloading techniques we'll discuss later in the book.[2]

Here again, if we pass in objects that do not support these interfaces (e.g., numbers), Python will automatically detect the mismatch and raise an exception for us—which is exactly what we want, and the best we could do on our own if we coded explicit type

2. This code will always work if we intersect files' contents obtained with `file.readlines()`. It may not work to intersect lines in open input files directly, though, depending on the file object's implementation of the `in` operator or general iteration. Files must generally be rewound (e.g., with a `file.seek(0)` or another `open`) after they have been read to end-of-file once, and so are single-pass iterators. As we'll see in Chapter 30 when we study operator overloading, objects implement the `in` operator either by providing the specific `__contains__` method or by supporting the general iteration protocol with the `__iter__` or older `__getitem__` methods; classes can code these methods arbitrarily to define what iteration means for their data.

tests. By not coding type tests and allowing Python to detect the mismatches for us, we both reduce the amount of code we need to write and increase our code's flexibility.

Local Variables

Probably the most interesting part of this example, though, is its names. It turns out that the variable `res` inside `intersect` is what in Python is called a *local variable*—a name that is visible only to code inside the function `def` and that exists only while the function runs. In fact, because all names *assigned* in any way inside a function are classified as local variables by default, nearly all the names in `intersect` are local variables:

- `res` is obviously assigned, so it is a local variable.
- Arguments are passed by assignment, so `seq1` and `seq2` are, too.
- The `for` loop assigns items to a variable, so the name `x` is also local.

All these local variables appear when the function is called and disappear when the function exits—the `return` statement at the end of `intersect` sends back the result *object*, but the *name* `res` goes away. Because of this, a function's variables won't remember values between calls; although the object returned by a function lives on, retaining other sorts of state information requires other sorts of techniques. To fully explore the notion of locals and state, though, we need to move on to the scopes coverage of Chapter 17.

Chapter Summary

This chapter introduced the core ideas behind function definition—the syntax and operation of the `def` and `return` statements, the behavior of function call expressions, and the notion and benefits of polymorphism in Python functions. As we saw, a `def` statement is executable code that creates a function object at runtime; when the function is later called, objects are passed into it by assignment (recall that assignment means object reference in Python, which, as we learned in Chapter 6, really means pointer internally), and computed values are sent back by `return`. We also began exploring the concepts of local variables and scopes in this chapter, but we'll save all the details on those topics for Chapter 17. First, though, a quick quiz.

Test Your Knowledge: Quiz

1. What is the point of coding functions?
2. At what time does Python create a function?
3. What does a function return if it has no `return` statement in it?
4. When does the code nested inside the function definition statement run?

5. What's wrong with checking the types of objects passed into a function?

Test Your Knowledge: Answers

1. Functions are the most basic way of avoiding code *redundancy* in Python—factoring code into functions means that we have only one copy of an operation's code to update in the future. Functions are also the basic unit of code *reuse* in Python—wrapping code in functions makes it a reusable tool, callable in a variety of programs. Finally, functions allow us to divide a complex system into manageable parts, each of which may be developed individually.
2. A function is created when Python reaches and runs the `def` statement; this statement creates a function object and assigns it the function's name. This normally happens when the enclosing module file is imported by another module (recall that imports run the code in a file from top to bottom, including any `def`s), but it can also occur when a `def` is typed interactively or nested in other statements, such as `if`s.
3. A function returns the `None` object by default if the control flow falls off the end of the function body without running into a `return` statement. Such functions are usually called with expression statements, as assigning their `None` results to variables is generally pointless. A `return` statement with no expression in it also returns `None`.
4. The function body (the code nested inside the function definition statement) is run when the function is later called with a call expression. The body runs anew each time the function is called.
5. Checking the types of objects passed into a function effectively breaks the function's flexibility, constraining the function to work on specific types only. Without such checks, the function would likely be able to process an entire range of object types—any objects that support the interface expected by the function will work. (The term *interface* means the set of methods and expression operators the function's code runs.)

CHAPTER 22

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, and provides self-contained namespaces that minimize variable name clashes across your programs. In concrete terms, modules typically correspond to Python program files. Each file is a module, and modules import other modules to use the names they define. Modules might also correspond to extensions coded in external languages such as C, Java, or C#, and even to directories in package imports. Modules are processed with two statements and one important function:

`import`
: Lets a client (importer) fetch a module as a whole

`from`
: Allows clients to fetch particular names from a module

`imp.reload` *(*`reload` *in 2.X)*
: Provides a way to reload a module's code without stopping Python

Chapter 3 introduced module fundamentals, and we've been using them ever since. The goal here is to expand on the core module concepts you're already familiar with, and move on to explore more advanced module usage. This first chapter reviews module basics, and offers a general look at the role of modules in overall program structure. In the chapters that follow, we'll dig into the coding details behind the theory.

Along the way, we'll flesh out module details omitted so far—you'll learn about reloads, the `__name__` and `__all__` attributes, package imports, relative import syntax, 3.3 namespace packages, and so on. Because modules and classes are really just glorified *namespaces*, we'll formalize namespace concepts here as well.

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, 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. Just as importantly, modules are a place to define names, known as *attributes*, which may be referenced by multiple external clients. When used well, this supports a *modular* program design that groups functionality into reusable units.

System namespace partitioning
: Modules are also the highest-level program organization unit in Python. Although they are fundamentally just packages of names, these packages are also *self-contained*—you can never see a name in another file, unless you explicitly import that file. Much like the local scopes of functions, this helps avoid name clashes across your programs. In fact, you can't avoid this feature—everything "lives" in a module, both the code you run and the objects you create are always implicitly enclosed in modules. Because of that, modules are natural tools for grouping system components.

Implementing shared services or data
: From an operational perspective, modules are also useful for implementing components that are shared across a system and hence require only a *single copy*. For instance, if you need to provide a global object that's used by more than one function or file, you can code it in a module that can then be imported by many clients.

At least that's the abstract story—for you to truly understand the role of modules in a Python system, we need to digress for a moment and explore the general structure of a Python program.

Python Program Architecture

So far in this 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—as the code timing programs of the preceding chapter illustrate. Even if you can get by with coding a single file yourself, you will almost certainly wind up using external files that someone else has already written.

This section introduces the general *architecture* of Python programs—the way you divide a program into a collection of source files (a.k.a. modules) and link the parts into a whole. As we'll see, Python fosters a modular program structure that groups functionality into coherent and reusable units, in ways that are natural, and almost automatic. Along the way, we'll also explore the central concepts of Python modules, imports, and object attributes.

How to Structure a Program

At a base level, a Python program consists of text files containing Python *statements*, with one main *top-level* file, and zero or more supplemental files known as *modules*.

Here's how this works. The top-level (a.k.a. script) file contains the main flow of control of your program—this is the file you run to launch your application. The module files are libraries of tools used to collect components used by the top-level file, and possibly elsewhere. Top-level files use tools defined in module files, and modules use tools defined in other modules.

Although they are files of code too, module files generally don't do anything when run directly; rather, they define tools intended for use in other files. A file *imports* a module to gain access to the tools it defines, which are known as its *attributes*—variable names attached to objects such as functions. Ultimately, we import modules and access their attributes to use their tools.

Imports and Attributes

Let's make this a bit more concrete. Figure 22-1 sketches the structure of a Python program composed of three files: *a.py*, *b.py*, and *c.py*. The file *a.py* is chosen to be the top-level file; it will be a simple text file of statements, which is executed from top to bottom when launched. The files *b.py* and *c.py* are modules; they are simple text files of statements as well, but they are not usually launched directly. Instead, as explained previously, modules are normally imported by other files that wish to use the tools the modules define.

For instance, suppose the file *b.py* in Figure 22-1 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 you can later run by passing zero or more values in parentheses after the function's name:

```
def spam(text):                 # File b.py
    print(text, 'spam')
```

Now, suppose *a.py* wants to use `spam`. To this end, it might contain Python statements such as the following:

```
import b                        # File a.py
b.spam('gumby')                 # Prints "gumby spam"
```

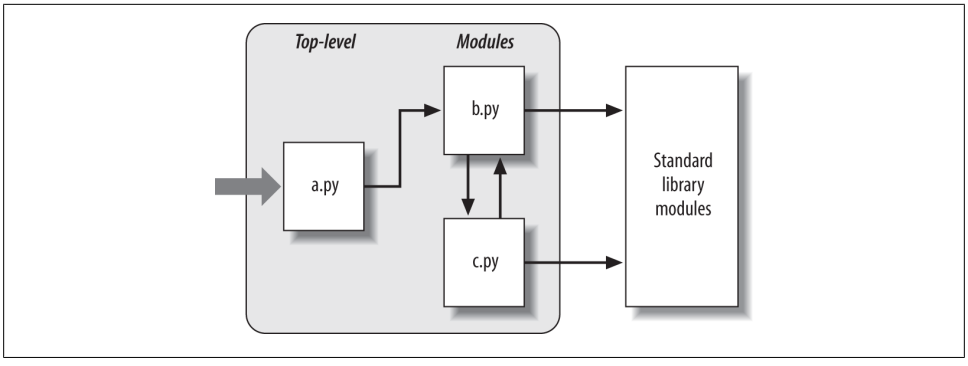

Figure 22-1. Program architecture in Python. A program is a system of modules. It has one top-level script file (launched to run the program), and multiple module files (imported libraries of tools). Scripts and modules are both text files containing Python statements, though the statements in modules usually just create objects to be used later. Python's standard library provides a collection of precoded modules.

The first of these, a Python `import` statement, gives the file *a.py* access to everything defined by top-level code in the file *b.py*. The code `import b` roughly means:

> Load the file *b.py* (unless it's already loaded), and give me access to all its attributes through the name `b`.

To satisfy such goals, `import` (and, as you'll see later, `from`) statements execute and load other files on request. More formally, in Python, cross-file module linking is not resolved until such `import` statements are executed at *runtime*; their net effect is to assign module names—simple variables like `b`—to loaded module objects. In fact, the module name used in an `import` statement serves two purposes: it identifies the external *file* to be loaded, but it also becomes a *variable* assigned to the loaded module.

Similarly, objects *defined* by a module are also created at runtime, as the `import` is executing: `import` literally runs statements in the target file one at a time to create its contents. Along the way, every name assigned at the top-level of the file becomes an attribute of the module, accessible to importers. For example, the second of the statements in *a.py* calls the function `spam` defined in the module `b`—created by running its `def` statement during the import—using object attribute notation. The code `b.spam` means:

> Fetch the value of the name `spam` that lives within the object `b`.

This happens to be a callable function in our example, so we pass a string in parentheses (`'gumby'`). If you actually type these files, save them, and run *a.py*, the words "gumby spam" will be printed.

As we've seen, the `object.attribute` notation appears throughout Python code—most objects have useful attributes that are fetched with the "." operator. Some reference callable objects like functions that take action (e.g., a salary computer), and others are

simple data values that denote more static objects and properties (e.g., a person's name).

The notion of importing is also completely general throughout Python. Any file can import tools from any other file. For instance, the file *a.py* may import *b.py* to call its function, but *b.py* might also import *c.py* to leverage different tools defined there. Import chains can go as deep as you like: in this example, the module `a` can import `b`, which can import `c`, which can import `b` again, and so on.

Besides serving as the highest organizational structure, modules (and module packages, described in Chapter 24) are also the highest level of *code reuse* in Python. Coding components in module files makes them useful in your original program, and in any other programs you may write later. For instance, if after coding the program in Figure 22-1 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 22-1. 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, over 200 modules large at last count, contains platform-independent support for common programming tasks: operating system interfaces, object persistence, text pattern matching, network and Internet scripting, GUI construction, and much more. None of these tools are part of the Python language itself, but you can use them by importing the appropriate modules on any standard Python installation. Because they are standard library modules, you can also be reasonably sure that they will be available and will work portably on most platforms on which you will run Python.

This book's examples employ a few of the standard library's modules—`timeit`, `sys`, and `os` in last chapter's code, for instance—but we'll really only scratch the surface of the libraries story here. For a complete look, you should browse the standard Python library reference manual, available either online at *http://www.python.org*, or with your Python installation (via IDLE or Python's Start button menu on some Windows). The *PyDoc* tool discussed in Chapter 15 is another way to explore standard library modules.

Because there are so many modules, this is really the only way to get a feel for what tools are available. You can also find tutorials on Python library tools in commercial books that cover application-level programming, such as O'Reilly's *Programming Python*, but the manuals are free, viewable in any web browser (in HTML format), viewable in other formats (e.g., Windows help), and updated each time Python is rereleased. See Chapter 15 for more pointers.

How Imports Work

The prior section talked about importing modules without really explaining what happens when you do so. Because imports are at the heart of program structure in Python, this section goes into more formal detail on the import operation to make this process less abstract.

Some C programmers like to compare the Python module import operation to a `C #include`, but they really shouldn't—in Python, imports are not just textual insertions of one file into another. They are really runtime operations that perform three distinct steps the first time a program imports a given file:

1. *Find* the module's file.
2. *Compile* it to byte code (if needed).
3. *Run* the module's code to build the objects it defines.

To better understand module imports, we'll explore these steps in turn. Bear in mind that all three of these steps are carried out only the *first time* a module is imported during a program's execution; later imports of the same module in a program run bypass all of these steps and simply fetch the already loaded module object in memory. Technically, Python does this by storing loaded modules in a table named `sys.modules` and checking there at the start of an import operation. If the module is not present, a three-step process begins.

1. Find It

First, Python must locate the module file referenced by an `import` statement. Notice that the `import` statement in the prior section's example names the file without a *.py* extension and without its directory path: it just says `import b`, instead of something like `import c:\dir1\b.py`. Path and extension details are omitted on purpose; instead, Python uses a standard *module search path* and known file types to locate the module file corresponding to an `import` statement.[1] Because this is the main part of the import operation that programmers must know about, we'll return to this topic in a moment.

1. It's syntactically illegal to include path and extension details in a standard `import`. However, *package imports*, which we'll discuss in Chapter 24, allow `import` statements to include part of the directory path leading to a file as a set of period-separated names. Package imports, though, 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), which typically embed byte code in the binary image.

2. Compile It (Maybe)

After finding a source code file that matches an `import` statement by traversing the module search path, Python next compiles it to byte code, if necessary. We discussed byte code briefly in Chapter 2, but it's a bit richer than explained there. During an import operation Python checks both file modification times and the byte code's Python version number to decide how to proceed. The former uses file "timestamps," and the latter uses either a "magic" number embedded in the byte code or a filename, depending on the Python release being used. This step chooses an action as follows:

Compile
: If the byte code file is *older* than the source file (i.e., if you've changed the source) or was created by a different Python *version*, Python automatically regenerates the byte code when the program is run.

  As discussed ahead, this model is modified somewhat in Python 3.2 and later—byte code files are segregated in a *__pycache__* subdirectory and named with their Python version to avoid contention and recompiles when multiple Pythons are installed. This obviates the need to check version numbers in the byte code, but the timestamp check is still used to detect changes in the source.

Don't compile
: If, on the other hand, Python finds a *.pyc* byte code file that is *not older* than the corresponding *.py* source file and was created by the same Python version, it skips the source-to-byte-code compile step.

  In addition, if Python finds only a byte code file on the search path and no source, it simply loads the byte code directly; this means you can ship a program as just byte code files and avoid sending source. In other words, the compile step is *bypassed* if possible to speed program startup.

Notice that compilation happens when a file is being imported. Because of this, you will not usually see a *.pyc* byte code file for the *top-level* file of your program, unless it is also imported elsewhere—only imported files leave behind *.pyc* files on your machine. The byte code of top-level files is used internally and discarded; byte code of imported files is saved in files to speed future imports.

Top-level files are often designed to be executed directly and not imported at all. Later, we'll see that it is possible to design a file that serves both as the top-level code of a program and as a module of tools to be imported. Such a file may be both executed and imported, and thus does generate a *.pyc*. To learn how this works, watch for the discussion of the special `__name__` attribute and `__main__` in Chapter 25.

3. Run It

The final step of an import operation executes the byte code of the module. All statements in the file are run in turn, from top to bottom, and any assignments made to names during this step generate attributes of the resulting module object. This is how

the tools defined by the module's code are created. For instance, `def` statements in a file are run at import time to create functions and assign attributes within the module to those functions. The functions can then be called later in the program by the file's importers.

Because this last import step actually runs the file's code, if any top-level code in a module file does real work, you'll see its results at import time. For example, top-level `print` statements in a module show output when the file is imported. Function `def` statements simply define objects for later use.

As you can see, import operations involve quite a bit of work—they search for files, possibly run a compiler, and run Python code. Because of this, any given module is imported only *once* per process by default. Future imports skip all three import steps and reuse the already loaded module in memory. If you need to import a file again after it has already been loaded (for example, to support dynamic end-user customizations), you have to force the issue with an `imp.reload` call—a tool we'll meet in the next chapter.[2]

Byte Code Files: __pycache__ in Python 3.2+

As mentioned briefly, the way that Python stores files to retain the byte code that results from compiling your source has changed in Python 3.2 and later. First of all, if Python cannot write a file to save this on your computer for any reason, your program still runs fine—Python simply creates and uses the byte code in memory and discards it on exit. To speed startups, though, it will try to save byte code in a file in order to skip the compile step next time around. The way it does this varies per Python version:

In Python 3.1 and earlier (including all of Python 2.X)
: Byte code is stored in files in the *same* directory as the corresponding source files, normally with the filename extension *.pyc* (e.g., *module.pyc*). Byte code files are also stamped internally with the version of Python that created them (known as a "magic" field to developers) so Python knows to *recompile* when this differs in the version of Python running your program. For instance, if you upgrade to a new Python whose byte code differs, all your byte code files will be recompiled automatically due to a version number mismatch, even if you haven't changed your source code.

In Python 3.2 and later
: Byte code is instead stored in files in a subdirectory named *__pycache__*, which Python creates if needed, and which is located in the directory containing the corresponding source files. This helps avoid *clutter* in your source directories by segregating the byte code files in their own directory. In addition, although byte code

2. 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())`. There's more on other uses for this internal table in Chapter 25.

files still get the *.pyc* extension as before, they are given more descriptive names that include text identifying the *version* of Python that created them (e.g., *module.cpython-32.pyc*). This avoids contention and *recompiles*: because each version of Python installed can have its own uniquely named version of byte code files in the *__pycache__* subdirectory, running under a given version doesn't overwrite the byte code of another, and doesn't require recompiles. Technically, byte code filenames also include the *name* of the Python that created them, so CPython, Jython, and other implementations mentioned in the preface and Chapter 2 can coexist on the same machine without stepping on each other's work (once they support this model).

In *both* models, Python always recreates the byte code file if you've changed the source code file since the last compile, but version differences are handled differently—by magic numbers and replacement prior to 3.2, and by filenames that allow for multiple copies in 3.2 and later.

Byte Code File Models in Action

The following is a quick example of these two models in action under 2.X and 3.3. I've omitted much of the text displayed by the `dir` directory listing on Windows here to save space, and the script used here isn't listed because it is not relevant to this discussion (it's from Chapter 2, and simply prints two values). *Prior to 3.2*, byte code files show up alongside their source files after being created by import operations:

```
c:\code\py2x> dir
10/31/2012  10:58 AM                39 script0.py

c:\code\py2x> C:\python27\python
>>> import script0
hello world
126765060022822940149670320537 6
>>> ^Z

c:\code\py2x> dir
10/31/2012  10:58 AM                39 script0.py
10/31/2012  11:00 AM               154 script0.pyc
```

However, in *3.2 and later* byte code files are saved in the *__pycache__* subdirectory and include versions and Python implementation details in their names to avoid clutter and contention among the Pythons on your computer:

```
c:\code\py2x> cd ..\py3x
c:\code\py3x> dir
10/31/2012  10:58 AM                39 script0.py

c:\code\py3x> C:\python33\python
>>> import script0
hello world
126765060022822940149670320537 6
>>> ^Z
```

```
c:\code\py3x> dir
10/31/2012  10:58 AM                39 script0.py
10/31/2012  11:00 AM    <DIR>          __pycache__

c:\code\py3x> dir __pycache__
10/31/2012  11:00 AM               184 script0.cpython-33.pyc
```

Crucially, under the model used in 3.2 and later, importing the same file with a different Python creates a *different* byte code file, instead of overwriting the *single* file as done by the pre-3.2 model—in the newer model, each Python version and implementation has its own byte code files, ready to be loaded on the next program run (earlier Pythons will happily continue using their scheme on the same machine):

```
c:\code\py3x> C:\python32\python
>>> import script0
hello world
126765060022822940149670320537 6
>>> ^Z

c:\code\py3x> dir __pycache__
10/31/2012  12:28 PM               178 script0.cpython-32.pyc
10/31/2012  11:00 AM               184 script0.cpython-33.pyc
```

Python 3.2's newer byte code file model is probably superior, as it avoids recompiles when there is more than one Python on your machine—a common case in today's mixed 2.X/3.X world. On the other hand, it is not without potential incompatibilities in programs that rely on the prior file and directory structure. This may be a compatibility issue in some tools programs, for instance, though most well-behaved tools should work as before. See Python 3.2's "What's New?" document for details on potential impacts.

Also keep in mind that this process is completely *automatic*—it's a side effect of running programs—and most programmers probably won't care about or even notice the difference, apart from faster startups due to fewer recompiles.

The Module Search Path

As mentioned earlier, the part of the import procedure that most programmers *will* need to care about is usually the first—locating the file to be imported (the "find it" part). Because you may need to tell Python where to look to find files to import, you need to know how to tap into its search path in order to extend it.

In many cases, you can rely on the automatic nature of the module import search path and won't need to configure this path at all. If you want to be able to import user-defined files across directory boundaries, though, you will need to know how the search path works in order to customize it. Roughly, Python's module search path is composed of the concatenation of these major components, some of which are preset for you and some of which you can tailor to tell Python where to look:

1. The home directory of the program
2. PYTHONPATH directories (if set)
3. Standard library directories
4. The contents of any *.pth* files (if present)
5. The *site-packages* home of third-party extensions

Ultimately, the concatenation of these four components becomes `sys.path`, a mutable list of directory name strings that I'll expand upon later in this section. The first and third elements of the search path are defined automatically. Because Python searches the concatenation of these components from first to last, though, the *second* and *fourth* elements can be used to extend the path to include your own source code directories. Here is how Python uses each of these path components:

Home directory (automatic)
: Python first looks for the imported file in the home directory. The meaning of this entry depends on how you are running the code. When you're running a *program*, this entry is the directory containing your program's top-level script file. When you're working *interactively*, this entry is the directory in which you are working (i.e., the current working directory).

  Because this directory is always searched first, if a program is located entirely in a single directory, all of its imports will work automatically with no path configuration required. On the other hand, because this directory is searched first, its files will also override modules of the same name in directories elsewhere on the path; be careful not to accidentally hide library modules this way if you need them in your program, or use package tools we'll meet later that can partially sidestep this issue.

PYTHONPATH *directories (configurable)*
: Next, Python searches all directories listed in your PYTHONPATH environment variable setting, from left to right (assuming you have set this at all: it's not preset for you). In brief, PYTHONPATH is simply a list of user-defined and platform-specific names of directories that contain Python code files. You can add all the directories from which you wish to be able to import, and Python will extend the module search path to include all the directories your PYTHONPATH lists.

  Because Python searches the home directory first, this setting is only important when importing files across directory boundaries—that is, if you need to import a file that is stored in a *different* directory from the file that imports it. You'll probably want to set your PYTHONPATH variable once you start writing substantial programs, but when you're first starting out, as long as you save all your module files in the directory in which you're working (i.e., the home directory, like the *C:\code* used in this book) your imports will work without you needing to worry about this setting at all.

Standard library directories (automatic)
: Next, Python automatically searches the directories where the standard library modules are installed on your machine. Because these are always searched, they normally do not need to be added to your `PYTHONPATH` or included in path files (discussed next).

.pth path file directories (configurable)
: Next, a lesser-used feature of Python allows users to add directories to the module search path by simply listing them, one per line, in a text file whose name ends with a *.pth* suffix (for "path"). These path configuration files are a somewhat advanced installation-related feature; we won't cover them fully here, but they provide an alternative to `PYTHONPATH` settings.

  In short, text files of directory names dropped in an appropriate directory can serve roughly the same role as the `PYTHONPATH` environment variable setting. For instance, if you're running Windows and Python 3.3, a file named *myconfig.pth* may be placed at the top level of the Python install directory (*C:\Python33*) or in the *site-packages* subdirectory of the standard library there (*C:\Python33\Lib\site-packages*) to extend the module search path. On Unix-like systems, this file might be located in *usr/local/lib/python3.3/site-packages* or */usr/local/lib/site-python* instead.

  When such a file is present, Python will add the directories listed on each line of the file, from first to last, near the end of the module search path list—currently, after `PYTHONPATH` and standard libraries, but before the *site-packages* directory where third-party extensions are often installed. In fact, Python will collect the directory names in all the *.pth* path files it finds and will filter out any duplicates and nonexistent directories. Because they are files rather than shell settings, path files can apply to all users of an installation, instead of just one user or shell. Moreover, for some users and applications, text files may be simpler to code than environment settings.

  This feature is more sophisticated than I've described here. For more details, consult the Python library manual, and especially its documentation for the standard library module `site`—this module allows the locations of Python libraries and path files to be configured, and its documentation describes the expected locations of path files in general. I recommend that beginners use `PYTHONPATH` or perhaps a single *.pth* file, and then only if you must import across directories. Path files are used more often by third-party libraries, which commonly install a path file in Python's *site-packages*, described next.

The Lib\site-packages directory of third-party extensions (automatic)
: Finally, Python automatically adds the *site-packages* subdirectory of its standard library to the module search path. By convention, this is the place that most third-party extensions are installed, often automatically by the `distutils` utility described in an upcoming sidebar. Because their install directory is always part of the module search path, clients can import the modules of such extensions without any path settings.

Configuring the Search Path

The net effect of all of this is that both the PYTHONPATH and path file components of the search path allow you to tailor the places where imports look for files. The way you set environment variables and where you store path files varies per platform. For instance, on Windows, you might use your Control Panel's System icon to set PYTHONPATH to a list of directories separated by semicolons, like this:

```
c:\pycode\utilities;d:\pycode\package1
```

Or you might instead create a text file called *C:\Python33\pydirs.pth*, which looks like this:

```
c:\pycode\utilities
d:\pycode\package1
```

These settings are analogous on other platforms, but the details can vary too widely for us to cover in this 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, Python releases, and even Python implementations. Depending on your platform, additional directories may automatically be added to the module search path as well.

For instance, some Pythons may add an entry for the *current working directory*—the directory from which you launched your program—in the search path before the PYTHONPATH directories. When you're launching from a command line, the current working directory may not be the same as the home directory of your top-level file (i.e., the directory where your program file resides), which is always added. Because the current working directory can vary each time your program runs, you normally shouldn't depend on its value for import purposes. See Chapter 3 for more on launching programs from command lines.[3]

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`

3. Also watch for Chapter 24's discussion of the new *relative import syntax* and search rules in Python 3.X; they modify 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.X, unless such relative imports are used by files in the package itself.

list (that is, the `path` attribute of the standard library module `sys`). This list of directory name strings is the actual search path within Python; on imports, Python searches each directory in this list from left to right, and uses the first file match it finds.

Really, `sys.path` *is* the module search path. Python configures it at program startup, automatically merging the home directory of the top-level file (or an empty string to designate the current working directory), any `PYTHONPATH` directories, the contents of any *.pth* file paths you've created, and all the standard library directories. The result is a list of directory name strings that Python searches on each import of a new file.

Python exposes this list for two good reasons. First, it provides a way to verify the search path settings you've made—if you don't see your settings somewhere in this list, you need to recheck your work. For example, here is what my module search path looks like on Windows under Python 3.3, with my `PYTHONPATH` set to `C:\code` and a *C:\Python33\mypath.pth* path file that lists `C:\Users\mark`. The empty string at the front means current directory, and my two settings are merged in; the rest are standard library directories and files and the *site-packages* home for third-party extensions:

```
>>> import sys
>>> sys.path
['', 'C:\\code', 'C:\\Windows\\system32\\python33.zip', 'C:\\Python33\\DLLs',
'C:\\Python33\\lib', 'C:\\Python33', 'C:\\Users\\mark',
'C:\\Python33\\lib\\site-packages']
```

Second, if you know what you're doing, this list provides a way for scripts to tailor their search paths manually. As you'll see by example later in this part of the book, by *modifying* the `sys.path` list, you can modify the search path for all future imports made in a program's run. Such changes last only for the duration of the script, however; `PYTHONPATH` and *.pth* files offer more permanent ways to modify the path—the first per user, and the second per installation.

On the other hand, some programs really *do* need to change `sys.path`. Scripts that run on web servers, for example, often run as the user "nobody" to limit machine access. Because such scripts cannot usually depend on "nobody" to have set `PYTHONPATH` in any particular way, they often set `sys.path` manually to include required source directories, prior to running any import statements. A `sys.path.append` or `sys.path.insert` will often suffice, though will endure for a single program run only.

Module File Selection

Keep in mind that filename extensions (e.g., *.py*) are omitted from `import` statements intentionally. Python chooses the first file it can find on the search path that matches the imported name. In fact, imports are the point of interface to a host of external components—source code, multiple flavors of byte code, compiled extensions, and more. Python automatically selects any type that matches a module's name.

Module sources

For example, an `import` statement of the form `import b` might today load or resolve to:

- A source code file named *b.py*
- A byte code file named *b.pyc*
- An optimized byte code file named *b.pyo* (a less common format)
- A directory named *b*, for package imports (described in Chapter 24)
- A compiled extension module, coded in C, C++, or another language, and dynamically linked when imported (e.g., *b.so* on Linux, or *b.dll* or *b.pyd* on Cygwin and Windows)
- A compiled built-in module coded in C and statically linked into Python
- A ZIP file component that is automatically extracted when imported
- An in-memory image, for frozen executables
- A Java class, in the Jython version of Python
- A .NET component, in the IronPython version of Python

C extensions, Jython, and package imports all extend imports beyond simple files. To importers, though, differences in the loaded file type are completely irrelevant, both when importing and when fetching module attributes. Saying `import b` gets whatever module `b` is, according to your module search path, and `b.attr` fetches an item in the module, be it a Python variable or a linked-in C function. Some standard modules we will use in this book are actually coded in C, not Python; because they look just like Python-coded module files, their clients don't have to care.

Selection priorities

If you have both a *b.py* and a *b.so* in different directories, Python will always load the one found in the first (leftmost) directory of your module search path during the 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 or across implementations. In general, you should not depend on which type of file Python will choose within a given directory—make your module names distinct, or configure your module search path to make your module selection preferences explicit.

Import hooks and ZIP files

Normally, imports work as described in this section—they find and load files on your machine. However, it is possible to redefine much of what an import operation does in Python, using what are known as *import hooks*. These hooks can be used to make imports do various useful things, such as loading files from archives, performing decryption, and so on.

In fact, Python itself makes use of these hooks to enable files to be directly imported from ZIP archives: archived files are automatically extracted at import time when a *.zip* file is selected from the module import search path. One of the standard library directories in the earlier `sys.path` display, for example, is a *.zip* file today. For more details, see the Python standard library manual's description of the built-in `__import__` function, the customizable tool that `import` statements actually run.

Also see Python 3.3's "What's New?" document for updates on this front that we'll mostly omit here for space. In short, in this version and later, the `__import__` function is now implemented by `imporlib.__import__`, in part to unify and more clearly expose its implementation.

The latter of these calls is also wrapped by `importlib.import_module`—a tool that, per Python's current manuals, is generally preferred over `__import__` for direct calls to import by name string, a technique discussed in Chapter 25. Both calls still work today, though the `__import__` function supports customizing imports by replacement in the built-in scope (see Chapter 17), and other techniques support similar roles. See the Python library manuals for more details.

Optimized byte code files

Finally, Python also supports the notion of *.pyo* optimized byte code files, created and run with the `-O` Python command-line flag, and automatically generated by some install tools. Because these run only slightly faster than normal *.pyc* files (typically 5 percent faster), however, they are infrequently used. The PyPy system (see Chapter 2 and Chapter 21), for example, provides more substantial speedups. See Appendix A and Chapter 36 for more on *.pyo* files.

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 third-party open source *eggs* system, which adds dependency checking for installed Python software.

Note: as this fifth edition is being written, there is some talk of deprecating `distutils` and replacing it with a newer `distutils2` package in the Python standard library. The status of this is unclear—it was anticipated in 3.3 but did not appear—so be sure to see Python's "What's New" documents for updates on this front that may emerge after this book is released.

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 five 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 im-

port, 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). In practice, this will be a common case for nontrivial programs.
3. The five 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, all directories listed in *.pth* path files located in standard places, and the *site-packages* root directory for third-party extension installs. Of these, programmers can customize `PYTHONPATH` and *.pth* files.
4. Python might load a source code (*.py*) file, a byte code (*.pyc* or *.pyo*) 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. In fact, with import hooks, imports can load arbitrary items.
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 generally frowned upon (see Chapter 17 for more on the downsides of cross-file changes).

CHAPTER 23

Module Coding Basics

Now that we've looked at the larger ideas behind modules, let's turn to some examples of modules in action. Although some of the early topics in this chapter will be review for linear readers who have already applied them in previous chapters' examples, we'll find that they quickly lead us to further details surrounding Python's modules that we haven't yet met, such as nesting, reloads, scopes, and more.

Python modules are easy to *create*; they're just files of Python program code created with a text editor. You don't need to write special syntax to tell Python you're making a module; almost any text file will do. Because Python handles all the details of finding and loading modules, modules are also easy to *use*; clients simply import a module, or specific names a module defines, and use the objects they reference.

Module Creation

To define a module, simply use your text editor to type some Python code into a text file, and save it with a ".py" extension; any such file is automatically considered a Python module. All the names assigned at the top level of the module become its *attributes* (names associated with the module object) and are exported for clients to use —they morph from variable to module object attribute automatically.

For instance, if you type the following `def` into a file called *module1.py* and import it, you create a module object with one attribute—the name `printer`, which happens to be a reference to a function object:

```
def printer(x):                    # Module attribute
    print(x)
```

Module Filenames

Before we go on, I should say a few more words about module filenames. You can call modules just about anything you like, but module filenames should end in a *.py* suffix if you plan to import them. The *.py* is technically optional for top-level files that will

be run but not imported, but adding it in all cases makes your files' types more obvious and allows you to import any of your files in the future.

Because module names become variable names inside a Python program (without the *.py*), they should also follow the normal variable name rules 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.

Other Kinds of Modules

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, C++, and others (e.g., Java, in the Jython implementation of the language). Such modules are called *extension modules*, and they are generally used to wrap up external libraries for use in Python scripts. When imported by Python code, extension modules look and feel the same as modules coded as Python source code files—they are accessed with `import` statements, and they provide functions and objects as module attributes. Extension modules are beyond the scope of this 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 (one or more)
>>> module1.printer('Hello world!')     # Qualify to get names
Hello world!
```

The `import` statement simply lists one or more names of modules to load, separated by commas. Because it gives a name that refers to the *whole module* object, we must go through the module name to fetch its attributes (e.g., `module1.printer`).

The from Statement

By contrast, because `from` copies *specific names* from one file over to another scope, it allows us to use the copied names directly in the script without going through the module (e.g., `printer`):

```
>>> from module1 import printer         # Copy out a variable (one or more)
>>> printer('Hello world!')             # No need to qualify name
Hello world!
```

This form of `from` allows us to list one or more names to be copied out, separated by commas. Here, it has the same effect as the prior example, but because the imported name is copied into the scope where the `from` statement appears, using that name in the script requires less typing—we can use it directly instead of naming the enclosing module. In fact, we must; `from` doesn't assign the name of the module itself.

As you'll see in more detail later, the `from` statement is really just a minor extension to the `import` statement—it imports the module file as usual (running the full three-step procedure of the preceding chapter), but adds an extra step that copies one or more names (not objects) out of the file. The entire file is loaded, but you're given names for more direct access to its parts.

The from * Statement

Finally, the next example uses a special form of `from`: when we use a * instead of specific names, we get copies of *all names* assigned at the top level of the referenced module. Here again, we can then use the copied name `printer` in our script without going through the module name:

```
>>> from module1 import *               # Copy out _all_ variables
>>> printer('Hello world!')
Hello world!
```

Technically, both `import` and `from` statements invoke the same import operation; the `from *` form simply adds an extra step that copies all the names in the module into the importing scope. It essentially collapses one module's namespace into another; again,

the net effect is less typing for us. Note that only * works in this context; you can't use pattern matching to select a subset of names (though you could with more work and a loop through a module's `__dict__`, discussed ahead).

And that's it—modules really are simple to use. To give you a better understanding of what really happens when you define and use modules, though, let's move on to look at some of their properties in more detail.

In Python 3.X, the `from ...*` statement form described here can be used *only* at the top level of a module file, not within a function. Python 2.X allows it to be used within a function, but issues a warning anyhow. It's 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. Best practice in all Pythons recommends listing *all* your imports at the top of a module file; it's not required, but makes them easier to spot.

Imports Happen Only Once

One of the most common questions people seem to ask when they start using modules is, "Why won't my imports keep working?" They often report that the first import works fine, but later imports during an interactive session (or program run) seem to have no effect. In fact, they're not supposed to. This section explains why.

Modules are loaded and run on the first `import` or `from`, and only the first. This is on purpose—because importing is an expensive operation, by default Python does it just once per file, per process. Later import operations simply fetch the already loaded module object.

Initialization code

As one consequence, because top-level code in a module file is usually executed only once, you can use it to initialize variables. Consider the file *simple.py*, for example:

```
print('hello')
spam = 1                    # Initialize variable
```

In this example, the `print` and `=` statements run the first time the module is imported, and the variable `spam` is initialized at import time:

```
% python
>>> import simple           # First import: loads and runs file's code
hello
>>> simple.spam             # Assignment makes an attribute
1
```

Second and later imports don't rerun the module's code; they just fetch the already created module object from Python's internal modules table. Thus, the variable `spam` is not reinitialized:

```
>>> simple.spam = 2          # Change attribute in module
>>> import simple            # Just fetches already loaded module
>>> simple.spam              # Code wasn't rerun: attribute unchanged
2
```

Of course, sometimes you really *want* a module's code to be rerun on a subsequent import. We'll see how to do this with Python's `reload` function later in this 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, to select among options; appear in function `def`s, to be loaded only on calls (subject to the preceding note); be used in `try` statements, to provide defaults; and so on. They are not resolved or run until Python reaches them while executing your program. In other words, imported modules and names are not available until their associated `import` or `from` statements run.

Changing mutables in modules

Also, like `def`, the `import` and `from` are *implicit assignments*:

- `import` assigns an entire module object to a single name.
- `from` assigns one or more names to objects of the same names in another module.

All the things we've already discussed about assignment apply to module access, too. For instance, names copied with a `from` become references to shared objects; as with function arguments, reassigning a copied name has no effect on the module from which it was copied, but changing a shared *mutable object* through a copied name can also change it in the module from which it was imported. To illustrate, consider the following file, *small.py*:

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

When importing with `from`, we copy names to the importer's scope that initially share objects referenced by the module's names:

```
% python
>>> from small import x, y       # Copy two names out
>>> x = 42                       # Changes local x only
>>> y[0] = 42                    # Changes shared mutable in place
```

Here, `x` is not a shared mutable object, but `y` is. The names `y` in the importer and the importee both reference the same list object, so changing it from one place changes it in the other:

```
>>> import small                 # Get module name (from doesn't)
>>> small.x                      # Small's x is not my x
1
>>> small.y                      # But we share a changed mutable
[42, 2]
```

For more background on this, see Chapter 6. And 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, and the name in both modules references the same, changed object.

import and from Equivalence

Notice in the prior example that we have to execute an `import` statement after the `from` to access the `small` module name at all. `from` only copies names from one module to another; it does not assign the module name itself. At least conceptually, a `from` statement like this one:

```
from module import name1, name2      # Copy these two names out (only)
```

is equivalent to this statement sequence:

```
import module                        # Fetch the module object
name1 = module.name1                 # Copy names out by assignment
name2 = module.name2
del module                           # Get rid of the module name
```

Like all assignments, the `from` statement creates new variables in the importer, which initially refer to objects of the same names in the imported file. Only the *names* are copied out, though, not the objects they reference, and not the name of the module itself. When we use the `from *` form of this statement (`from module import *`), the equivalence is the same, but all the top-level names in the module are copied over to the importing scope this way.

Notice that the first step of the `from` runs a normal `import` operation, with all the semantics outlined in the preceding chapter. 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 770 (see Chapter 25).

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 have only one assignment to the name in your scope:

```
# O.py
from M import func
from N import func              # This overwrites the one we fetched from M
func()                          # Calls N.func only!
```

An `import` will work here, though, because including the name of the enclosing module makes the two names unique:

```
# O.py
import M, N                     # Get the whole modules, not their names
M.func()                        # We can call both names now
N.func()                        # The module names make them unique
```

This case is unusual enough that you're unlikely to encounter it very often in practice. If you do, though, `import` allows you to avoid the name collision. Another way out of this dilemma is using the `as` extension, which we'll cover in Chapter 25 but is simple enough to introduce here:

```
# O.py
from M import func as mfunc     # Rename uniquely with "as"
from N import func as nfunc
mfunc(); nfunc()                # Calls one or the other
```

The `as` extension works in both `import` and `from` as a simple renaming tool (it can also be used to give a shorter synonym for a long module name in `import`); more on this form in Chapter 25.

Module Namespaces

Modules are probably best understood as simply packages of names—i.e., places to define names you want to make visible to the rest of a system. Technically, modules usually correspond to files, and Python creates a module object to contain all the names assigned in a module file. But in simple terms, modules are just namespaces (places where names are created), and the names that live in a module are called its *attributes*. This section expands on the details behind this model.

Files Generate Namespaces

I've mentioned that files *morph* into namespaces, but how does this actually happen? The short answer is that every name that is assigned a value at the top level of a module file (i.e., not nested in a function or class body) becomes an attribute of that module.

For instance, given an assignment statement such as `X = 1` at the top level of a module file *M.py*, the name `X` becomes an attribute of `M`, which we can refer to from outside the module as `M.X`. The name `X` also becomes a global variable to other code inside *M.py*, but we need to consider the notion of module loading and scopes a bit more formally to understand why:

- **Module statements run on the first import**. The first time a module is imported anywhere in a system, Python creates an empty module object and executes the statements in the module file one after another, from the top of the file to the bottom.
- **Top-level assignments create module attributes**. During an import, statements at the top level of the file not nested in a `def` or `class` that assign names (e.g., `=`, `def`) create attributes of the module object; assigned names are stored in the module's namespace.
- **Module namespaces can be accessed via the attribute `__dict__` or `dir(M)`**. Module namespaces created by imports are dictionaries; they may be accessed through the built-in `__dict__` attribute associated with module objects and may be inspected with the `dir` function. The `dir` function is roughly equivalent to the sorted keys list of an object's `__dict__` attribute, but it includes inherited names for classes, may not be complete, and is prone to changing from release to release.
- **Modules are a single scope (local is global)**. As we saw 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—or, more formally, they follow the LEGB scope rule we met in Chapter 17, but without the *L* and *E* lookup layers.

  Crucially, though, the module's global *scope* becomes an attribute dictionary of a module *object* after the module has been loaded. Unlike function scopes, where the local namespace exists only while the function runs, a module file's scope becomes a module object's attribute namespace and *lives on* after the import, providing a source of tools to importers.

Here's a demonstration of these ideas. Suppose we create the following module file in a text editor and call it *module2.py*:

```
print('starting to load...')
import sys
name = 42

def func(): pass
```

```
class klass: pass

print('done loading.')
```

The first time this module is imported (or run as a program), Python executes its statements from top to bottom. Some statements create names in the module's namespace as a side effect, but others do actual work while the import is going on. For instance, the two `print` statements in this file execute at import time:

```
>>> import module2
starting to load...
done loading.
```

Once the module is loaded, its scope becomes an attribute namespace in the module object we get back from `import`. We can then access attributes in this namespace by qualifying them with the name of the enclosing module:

```
>>> module2.sys
<module 'sys' (built-in)>

>>> module2.name
42

>>> module2.func
<function func at 0x000000000222E7B8>

>>> module2.klass
<class 'module2.klass'>
```

Here, `sys`, `name`, `func`, and `klass` were all assigned while the module's statements were being run, so they are attributes after the import. We'll talk about classes in Part VI, but notice the `sys` attribute—`import` statements really *assign* module objects to names, and any type of assignment to a name at the top level of a file generates a module attribute.

Namespace Dictionaries: __dict__

In fact, internally, module namespaces are stored as *dictionary* objects. These are just normal dictionaries with all the usual methods. When needed—for instance, to write tools that list module content generically as we will in Chapter 25—we can access a module's namespace dictionary through the module's `__dict__` attribute. Continuing the prior section's example (remember to wrap this in a `list` call in Python 3.X—it's a view object there, and contents may vary outside 3.3 used here):

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

The names we assigned in the module file become dictionary keys internally, so some of the names here reflect top-level assignments in our file. However, Python also adds some names in the module's namespace for us; for instance, `__file__` gives the name

of the file the module was loaded from, and `__name__` gives its name as known to importers (without the *.py* extension and directory path). To see just the names your code assigns, filter out the double-underscore names as we've done before, in Chapter 15's `dir` coverage and Chapter 17's built-in scope coverage:

```
>>> list(name for name in module2.__dict__.keys() if not name.startswith('__'))
['func', 'klass', 'name', 'sys']
>>> list(name for name in module2.__dict__ if not name.startswith('__'))
['func', 'sys', 'name', 'klass']
```

This time we're filtering with a *generator* instead of a list comprehension, and can omit the `.keys()` because dictionaries generate their keys automatically though implicitly; the effect is the same. We'll see similar `__dict__` dictionaries on *class*-related objects in Part VI too. In both cases, attribute fetch is similar to dictionary indexing, though only the former kicks off inheritance in classes:

```
>>> module2.name, module2.__dict__['name']
(42, 42)
```

Attribute Name Qualification

Speaking of attribute fetch, now that you're becoming more familiar with modules, we should firm up the notion of name qualification more formally too. In Python, you can access the attributes of any object that has attributes using the *qualification* (a.k.a. attribute fetch) syntax *`object.attribute`*.

Qualification is really an expression that returns the value assigned to an attribute name associated with an object. For example, the expression `module2.sys` in the previous example fetches the value assigned to `sys` in `module2`. Similarly, if we have a built-in list object `L`, `L.append` returns the `append` method object associated with that list.

It's important to keep in mind that attribute qualification has nothing to do with the scope rules we studied in Chapter 17; it's an independent concept. When you use qualification to access names, you give Python an explicit object from which to fetch the specified names. The LEGB scope rule applies only to bare, unqualified names—it may be used for the leftmost name in a name path, but later names after dots search specific objects instead. Here are the rules:

Simple variables
: `X` means search for the name `X` in the current scopes (following the LEGB rule of Chapter 17).

Qualification
: `X.Y` means find `X` in the current scopes, then search for the attribute `Y` in the object `X` (not in scopes).

Qualification paths
: `X.Y.Z` means look up the name `Y` in the object `X`, then look up `Z` in the object `X.Y`.

Generality
: Qualification works on all objects with attributes: modules, classes, C extension types, etc.

In Part VI, we'll see that attribute qualification means a bit more for classes—it's also the place where something called *inheritance* happens—but in general, the rules outlined here apply to all names in Python.

Imports Versus Scopes

As we've learned, it is never possible to access names defined in another module file without first importing that file. That is, you never automatically get to see names in another file, regardless of the structure of imports or function calls in your program. A variable's meaning is always determined by the locations of assignments in your source code, and attributes are always requested of an object explicitly.

For example, consider the following two simple modules. The first, *moda.py*, defines a variable `X` global to code in its file only, along with a function that changes the global `X` in this file:

```
X = 88                          # My X: global to this file only
def f():
    global X                    # Change this file's X
    X = 99                      # Cannot see names in other modules
```

The second module, *modb.py*, defines its own global variable `X` and imports and calls the function in the first module:

```
X = 11                          # My X: global to this file only

import moda                     # Gain access to names in moda
moda.f()                        # Sets moda.X, not this file's X
print(X, moda.X)
```

When run, `moda.f` changes the `X` in `moda`, not the `X` in `modb`. The global scope for `moda.f` is always the file enclosing it, regardless of which module it is ultimately called from:

```
% python modb.py
11 99
```

In other words, import operations never give upward visibility to code in imported files—an imported file cannot see names in the importing file. More formally:

- Functions can never see names in other functions, unless they are physically enclosing.
- Module code can never see names in other modules, unless they are explicitly imported.

Such behavior is part of the *lexical scoping* notion—in Python, the scopes surrounding a piece of code are completely determined by the code's physical position in your file. Scopes are never influenced by function calls or module imports.[1]

Namespace Nesting

In some sense, although imports do not nest namespaces upward, they do nest downward. That is, although an imported module never has direct access to names in a file that imports it, using attribute qualification paths it is possible to descend into arbitrarily nested modules and access their attributes. For example, consider the next three files. *mod3.py* defines a single global name and attribute by assignment:

```
X = 3
```

mod2.py in turn defines its own `X`, then imports `mod3` and uses qualification to access the imported module's attribute:

```
X = 2
import mod3

print(X, end=' ')              # My global X
print(mod3.X)                 # mod3's X
```

mod1.py also defines its own `X`, then imports `mod2`, and fetches attributes in both the first and second files:

```
X = 1
import mod2

print(X, end=' ')              # My global X
print(mod2.X, end=' ')         # mod2's X
print(mod2.mod3.X)             # Nested mod3's X
```

Really, when `mod1` imports `mod2` here, it sets up a two-level namespace nesting. By using the path of names `mod2.mod3.X`, it can descend into `mod3`, which is nested in the imported `mod2`. The net effect is that `mod1` can see the `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

1. 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. In Python, scopes more simply correspond to the text of your program.

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 file import chains.

Reloading Modules

As we've seen, a module's code is run only once per process by default. To force a module's code to be reloaded and rerun, you need to ask Python to do so explicitly by calling the `reload` built-in function. In this section, we'll explore how to use reloads to make your systems more dynamic. In a nutshell:

- Imports (via both `import` and `from` statements) load and run a module's code only the first time the module is imported in a process.
- Later imports use the already loaded module object without reloading or rerunning the file's code.
- The `reload` function forces an already loaded module's code to be reloaded and rerun. Assignments in the file's new code change the existing module object in place.

Why care about reloading modules? In short, *dynamic customization*: the `reload` function allows parts of a program to be changed without stopping the whole program. With `reload`, the effects of changes in components can be observed immediately. Reloading doesn't help in every situation, but where it does, it makes for a much shorter development cycle. For instance, imagine a database program that must connect to a server on startup; because program changes or customizations can be tested immediately after reloads, you need to connect only once while debugging. Long-running servers can update themselves this way, too.

Because Python is interpreted (more or less), it already gets rid of the compile/link steps you need to go through to get a C program to run: modules are loaded dynamically when imported by a running program. Reloading offers a further performance advantage by allowing you to also change parts of running programs without stopping.

Though beyond this book's scope, note that `reload` currently only works on modules written in Python; compiled extension modules coded in a language such as C can be dynamically loaded at runtime, too, but they can't be reloaded (though most users probably prefer to code customizations in Python anyhow!).

Version skew note: In Python 2.X, `reload` is available as a built-in function. In Python 3.X, it has been moved to the `imp` standard library module—it's known as `imp.reload` in 3.X. This simply means that an extra `import` or `from` statement is required to load this tool in 3.X only. Readers using 2.X can ignore these imports in this book's examples, or use them anyhow—2.X also has a `reload` in its `imp` module to ease migration to 3.X. 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 new name.
- `reload` lives in a module in Python 3.X and must be imported itself.

Because `reload` expects an object, a module must have been previously imported successfully before you can reload it (if the import was unsuccessful due to a syntax or other error, you may need to repeat it before you can reload the module). Furthermore, the syntax of `import` statements and `reload` calls differs: as a function reloads require parentheses, but import statements do not. Abstractly, reloading looks like this:

```
import module                   # Initial import
...use module.attributes...
...                             # Now, go change the module file
...
from imp import reload          # Get reload itself (in 3.X)
reload(module)                  # Get updated exports
...use module.attributes...
```

The typical usage pattern is that you import a module, then change its source code in a text editor, and then reload it. This can occur when working interactively, but also in larger programs that reload periodically.

When you call `reload`, Python rereads the module file's source code and reruns its 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 an entire module *object* anywhere in your program is automatically affected by a reload. Here are the details:

- **`reload` runs a module file's new code in the module's current namespace**. Rerunning a module file's code overwrites its existing namespace, rather than deleting and re-creating it.
- **Top-level assignments in the file replace names with new values**. For instance, rerunning a `def` statement replaces the prior version of the function in the module's namespace by reassigning the function name.

- **Reloads impact all clients that use `import` to fetch modules**. Because clients that use `import` qualify to fetch attributes, they'll find new values in the module object after a reload.
- **Reloads impact future `from` clients only**. Clients that used `from` to fetch attributes in the past won't be affected by a reload; they'll still have references to the old objects fetched before the reload.
- **Reloads apply to a single module only**. You must run them on each module you wish to update, unless you use code or tools that apply reloads transitively.

reload Example

To demonstrate, here's a more concrete example of `reload` in action. In the following, we'll change and reload a module file without stopping the interactive Python session. Reloads are used in many other scenarios, too (see the sidebar "Why You Will Care: Module Reloads" on page 703), but we'll keep things simple for illustration here. First, in the text editor of your choice, write a module file named *changer.py* with the following contents:

```
message = "First version"
def printer():
    print(message)
```

This module creates and exports two names—one bound to a string, and another to a function. Now, start the Python interpreter, import the module, and call the function it exports. The function will print the value of the global `message` variable:

```
% python
>>> import changer
>>> changer.printer()
First version
```

Keeping the interpreter active, now edit the module file in another window:

```
...modify changer.py without stopping Python...
% notepad changer.py
```

Change the global `message` variable, as well as the `printer` function body:

```
message = "After editing"
def printer():
    print('reloaded:', message)
```

Then, return to the Python window and reload the module to fetch the new code. Notice in the following interaction that importing the module again has no effect; we get the original message, even though the file's been changed. We have to call `reload` in order to get the new version:

```
...back to the Python interpreter...
>>> import changer
>>> changer.printer()                  # No effect: uses loaded module
First version
>>> from imp import reload
```

```
>>> reload(changer)                 # Forces new code to load/run
<module 'changer' from '.\\changer.py'>
>>> changer.printer()               # Runs the new version now
reloaded: After editing
```

Notice that `reload` actually *returns* the module object for us—its result is usually ignored, but because expression results are printed at the interactive prompt, Python shows a default `<module 'name'...>` representation.

Two final notes here: first, if you use `reload`, you'll probably want to pair it with `import` instead of `from`, as the latter isn't updated by reload operations—leaving your names in a state that's strange enough to warrant postponing further elaboration until this part's "gotchas" at the end of Chapter 25. Second, `reload` by itself updates only a *single* module, but it's straightforward to code a function that applies it transitively to related modules—an extension we'll save for a case study near the end of Chapter 25.

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, game servers and 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 essentials 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 simply 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. The `as` extension can render `from` usable in this context as well.
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?