PART IV

Functions

CHAPTER 16
Function Basics

In Part III, we looked at basic procedural statements in Python. Here, we'll move on to explore a set of additional statements 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. Functions also can compute a result value and let us specify parameters that serve as function inputs, which 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 only have one copy to update, not many.

Functions are the most basic program structure Python provides for maximizing *code reuse* and minimizing *code redundancy*. As we'll see, functions are also a design tool that lets us split complex systems into manageable parts. Table 16-1 summarizes the primary function-related tools we'll study in this part of the book.

Table 16-1. Function-related statements and expressions

Statement	Examples
Calls	`myfunc('spam', 'eggs', meat=ham)`
def, return	`def adder(a, b=1, *c):` `  return a + b + c[0]`
global	`def changer():` `  global x; x = 'new'`
nonlocal	`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 object 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 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 a module file 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.
- **`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 3.0 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 when a function is called—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 can change objects shared by the caller.
- **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 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; it ends the function call and sends a result back to the caller. The `return` statement consists of an object expression that gives the function's result. The `return` statement is optional; 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 returns the `None` object automatically, but this return value is usually ignored.

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

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 we'll explore in the next section.

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 coded 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 pointless to code error checking ourselves. In fact, doing so would limit our function's utility, as it would be restricted to work only on objects whose types we test for.

This turns out to be a crucial philosophical difference between Python and statically typed languages like C++ and Java: in Python, your code is *not supposed to care* about specific data types. If it does, it will be limited to working on just the types you anticipated when you wrote it, and it will not support other compatible object types that

may be coded in the future. Although it is possible to test for types with tools like the `type` built-in function, doing so breaks your code's flexibility. By and large, we code to object *interfaces* in Python, not data types.

Of course, 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 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 only have to change code in 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.

Polymorphism Revisited

Like all 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 these later in the book).*

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 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 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. To fully explore the notion of locals, though, we need to move on to 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

* 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. As we'll see in Chapter 29 when we study operator overloading, classes 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; if coded, classes can define what iteration means for their data.

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.
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 17
Scopes

Chapter 16 introduced basic function definitions and calls. As we saw, Python's basic function model is simple to use, but even simple function examples quickly led us to questions about the meaning of variables in our code. This chapter moves on to present the details behind Python's *scopes*—the places where variables are defined and looked up. As we'll see, the place where a name is assigned in our code is crucial to determining what the name means. We'll also find that scope usage can have a major impact on program maintenance effort; overuse of globals, for example, is a generally bad thing.

Python Scope Basics

Now that you're ready to start writing your own functions, we need to get more formal about what names mean in Python. When you use a name in a program, Python creates, changes, or looks up the name in what is known as a *namespace*—a place where names live. When we talk about the search for a name's value in relation to code, the term *scope* refers to a namespace: that is, the location of a name's assignment in your code determines the scope of the name's visibility to your code.

Just about everything related to names, including scope classification, happens at assignment time in Python. As we've seen, names in Python spring into existence when they are first assigned values, and they must be assigned before they are used. Because names are not declared ahead of time, Python uses the location of the assignment of a name to associate it with (i.e., *bind* it to) a particular namespace. In other words, the place where you assign a name in your source code determines the namespace it will live in, and hence its scope of visibility.

Besides packaging code, functions add an extra namespace layer to your programs—by default, all names assigned inside a function are associated with that function's namespace, and no other. This means that:

- Names defined inside a `def` can only be seen by the code within that `def`. You cannot even refer to such names from outside the function.

- Names defined inside a `def` do not clash with variables outside the `def`, even if the same names are used elsewhere. A name `X` assigned outside a given `def` (i.e., in a different `def` or at the top level of a module file) is a completely different variable from a name `X` assigned inside that `def`.

In all cases, the scope of a variable (where it can be used) is always determined by where it is assigned in your source code and has nothing to do with which functions call which. In fact, as we'll learn in this chapter, variables may be assigned in three different places, corresponding to three different scopes:

- If a variable is assigned inside a `def`, it is *local* to that function.
- If a variable is assigned in an enclosing `def`, it is *nonlocal* to nested functions.
- If a variable is assigned outside all `def`s, it is *global* to the entire file.

We call this *lexical scoping* because variable scopes are determined entirely by the locations of the variables in the source code of your program files, not by function calls.

For example, in the following module file, the `X = 99` assignment creates a *global* variable named `X` (visible everywhere in this file), but the `X = 88` assignment creates a *local* variable `X` (visible only within the `def` statement):

```
X = 99

def func():
    X = 88
```

Even though both variables are named `X`, their scopes make them different. The net effect is that function scopes help to avoid name clashes in your programs and help to make functions more self-contained program units.

Scope Rules

Before we started writing functions, all the code we wrote was at the top level of a module (i.e., not nested in a `def`), so the names we used either lived in the module itself or were built-ins predefined by Python (e.g., `open`). Functions provide nested namespaces (scopes) that localize the names they use, such that names inside a function won't clash with those outside it (in a module or another function). Again, functions define a *local scope*, and modules define a *global scope*. The two scopes are related as follows:

- **The enclosing module is a global scope**. Each module is a global scope—that is, a namespace in which variables created (assigned) at the top level of the module file live. Global variables become attributes of a module object to the outside world but can be used as simple variables within a module file.
- **The global scope spans a single file only**. Don't be fooled by the word "global" here—names at the top level of a file are only global to code within that single file. There is really no notion of a single, all-encompassing global file-based scope in

Python. Instead, names are partitioned into modules, and you must always import a module explicitly if you want to be able to use the names its file defines. When you hear "global" in Python, think "module."

- **Each call to a function creates a new local scope**. Every time you call a function, you create a new local scope—that is, a namespace in which the names created inside that function will usually live. You can think of each `def` statement (and `lambda` expression) as defining a new local scope, but because Python allows functions to call themselves to loop (an advanced technique known as *recursion*), the local scope in fact technically corresponds to a function call—in other words, each call creates a new local namespace. Recursion is useful when processing structures whose shapes can't be predicted ahead of time.
- **Assigned names are local unless declared global or nonlocal**. By default, all the names assigned inside a function definition are put in the local scope (the namespace associated with the function call). If you need to assign a name that lives at the top level of the module enclosing the function, you can do so by declaring it in a `global` statement inside the function. If you need to assign a name that lives in an enclosing `def`, as of Python 3.0 you can do so by declaring it in a `nonlocal` statement.
- **All other names are enclosing function locals, globals, or built-ins**. Names not assigned a value in the function definition are assumed to be enclosing scope locals (in an enclosing `def`), globals (in the enclosing module's namespace), or built-ins (in the predefined `__builtin__` module Python provides).

There are a few subtleties to note here. First, keep in mind that code typed at the *interactive command prompt* follows these same rules. You may not know it yet, but code run interactively is really entered into a built-in module called `__main__`; this module works just like a module file, but results are echoed as you go. Because of this, interactively created names live in a module, too, and thus follow the normal scope rules: they are global to the interactive session. You'll learn more about modules in the next part of this book.

Also note that *any type of assignment* within a function classifies a name as local. This includes `=` statements, module names in `import`, function names in `def`, function argument names, and so on. If you assign a name in any way within a `def`, it will become a local to that function.

Conversely, *in-place changes* to objects do not classify names as locals; only actual name assignments do. For instance, if the name `L` is assigned to a list at the top level of a module, a statement `L = X` within a function will classify `L` as a local, but `L.append(X)` will not. In the latter case, we are changing the list object that `L` references, not `L` itself—`L` is found in the global scope as usual, and Python happily modifies it without requiring a `global` (or `nonlocal`) declaration. As usual, it helps to keep the distinction between names and objects clear: changing an object is not an assignment to a name.

Name Resolution: The LEGB Rule

If the prior section sounds confusing, it really boils down to three simple rules. With a `def` statement:

- Name references search at most four scopes: local, then enclosing functions (if any), then global, then built-in.
- Name assignments create or change local names by default.
- `global` and `nonlocal` declarations map assigned names to enclosing module and function scopes.

In other words, all names assigned inside a function `def` statement (or a `lambda`, an expression we'll meet later) are locals by default. Functions can freely use names assigned in syntactically enclosing functions and the global scope, but they must declare such nonlocals and globals in order to change them.

Python's name-resolution scheme is sometimes called the *LEGB rule*, after the scope names:

- When you use an unqualified name inside a function, Python searches up to four scopes—the local (*L*) scope, then the local scopes of any enclosing (*E*) `def`s and `lambda`s, then the global (*G*) scope, and then the built-in (*B*) scope—and stops at the first place the name is found. If the name is not found during this search, Python reports an error. As we learned in Chapter 6, names must be assigned before they can be used.
- When you assign a name in a function (instead of just referring to it in an expression), Python always creates or changes the name in the local scope, unless it's declared to be global or nonlocal in that function.
- When you assign a name outside any function (i.e., at the top level of a module file, or at the interactive prompt), the local scope is the same as the global scope—the module's namespace.

Figure 17-1 illustrates Python's four scopes. Note that the second scope lookup layer, *E*—the scopes of enclosing `def`s or `lambda`s—can technically correspond to more than one lookup layer. This case only comes into play when you nest functions within functions, and it is addressed by the `nonlocal` statement.*

Also keep in mind that these rules apply only to simple *variable* names (e.g., `spam`). In Parts V and VI, we'll see that qualified *attribute* names (e.g., `object.spam`) live in particular objects and follow a completely different set of lookup rules than those

* The scope lookup rule was called the "LGB rule" in the first edition of this book. The enclosing `def` "E" layer was added later in Python to obviate the task of passing in enclosing scope names explicitly with default arguments—a topic usually of marginal interest to Python beginners that we'll defer until later in this chapter. Since this scope is addressed by the `nonlocal` statement in Python 3.0, I suppose the lookup rule might now be better named "LNGB," but backward compatibility matters in books, too!

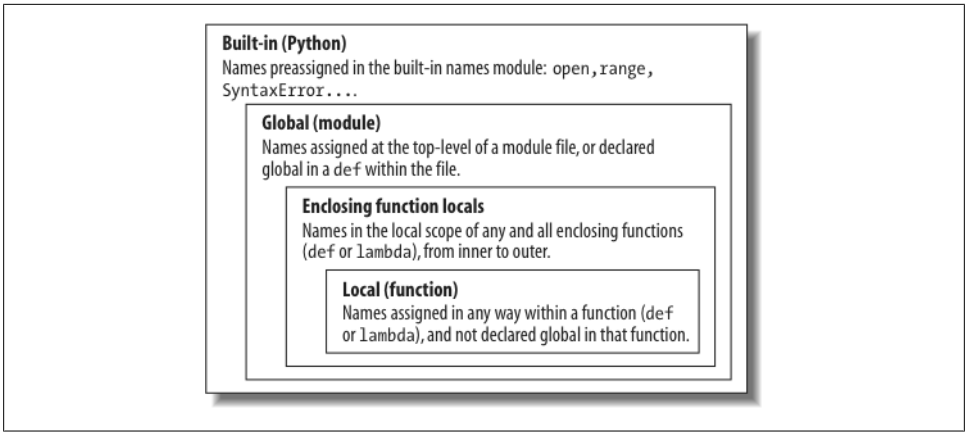

Figure 17-1. The LEGB scope lookup rule. When a variable is referenced, Python searches for it in this order: in the local scope, in any enclosing functions' local scopes, in the global scope, and finally in the built-in scope. The first occurrence wins. The place in your code where a variable is assigned usually determines its scope. In Python 3, nonlocal declarations can also force names to be mapped to enclosing function scopes, whether assigned or not.

covered here. References to attribute names following periods (`.`) search one or more *objects*, not scopes, and may invoke something called "inheritance"; more on this in Part VI of this book.

Scope Example

Let's look at a larger example that demonstrates scope ideas. Suppose we wrote the following code in a module file:

```
# Global scope
X = 99                  # X and func assigned in module: global

def func(Y):            # Y and Z assigned in function: locals
    # Local scope
    Z = X + Y           # X is a global
    return Z

func(1)                 # func in module: result=100
```

This module and the function it contains use a number of names to do their business. Using Python's scope rules, we can classify the names as follows:

Global names: `X`, `func`
: `X` is global because it's assigned at the top level of the module file; it can be referenced inside the function without being declared global. `func` is global for the same reason; the `def` statement assigns a function object to the name `func` at the top level of the module.

Local names: `Y`, `Z`

`Y` and `Z` are local to the function (and exist only while the function runs) because they are both assigned values in the function definition: `Z` by virtue of the `=` statement, and `Y` because arguments are always passed by assignment.

The whole point behind this name-segregation scheme is that local variables serve as temporary names that you need only while a function is running. For instance, in the preceding example, the argument `Y` and the addition result `Z` exist only inside the function; these names don't interfere with the enclosing module's namespace (or any other function, for that matter).

The local/global distinction also makes functions easier to understand, as most of the names a function uses appear in the function itself, not at some arbitrary place in a module. Also, because you can be sure that local names will not be changed by some remote function in your program, they tend to make programs easier to debug and modify.

The Built-in Scope

We've been talking about the built-in scope in the abstract, but it's a bit simpler than you may think. Really, the built-in scope is just a built-in module called `builtins`, but you have to import `builtins` to query built-ins because the name `builtins` is not itself built-in....

No, I'm serious! The built-in scope is implemented as a standard library module named `builtins`, but that name itself is not placed in the built-in scope, so you have to import it in order to inspect it. Once you do, you can run a `dir` call to see which names are predefined. In Python 3.0:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis',
    ...many more names omitted...
'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set',
'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

The names in this list constitute the built-in scope in Python; roughly the first half are built-in exceptions, and the second half are built-in functions. Also in this list are the special names `None`, `True`, and `False`, though they are treated as reserved words. Because Python automatically searches this module last in its LEGB lookup, you get all the names in this list "for free;" that is, you can use them without importing any modules. Thus, there are really two ways to refer to a built-in function—by taking advantage of the LEGB rule, or by manually importing the `builtins` module:

```
>>> zip                          # The normal way
<class 'zip'>
```

```
>>> import builtins              # The hard way
>>> builtins.zip
<class 'zip'>
```

The second of these approaches is sometimes useful in advanced work. The careful reader might also notice that because the LEGB lookup procedure takes the first occurrence of a name that it finds, names in the local scope may override variables of the same name in both the global and built-in scopes, and global names may override built-ins. A function can, for instance, create a local variable called `open` by assigning to it:

```
def hider():
    open = 'spam'               # Local variable, hides built-in
    ...
    open('data.txt')            # This won't open a file now in this scope!
```

However, this will hide the built-in function called `open` that lives in the built-in (outer) scope. It's also usually a bug, and a nasty one at that, because Python will not issue a warning message about it (there are times in advanced programming where you may really want to replace a built-in name by redefining it in your code).

Functions can similarly hide global variables of the same name with locals:

```
X = 88                          # Global X

def func():
    X = 99                      # Local X: hides global

func()
print(X)                        # Prints 88: unchanged
```

Here, the assignment within the function creates a local `X` that is a completely different variable from the global `X` in the module outside the function. Because of this, there is no way to change a name outside a function without adding a `global` (or `nonlocal`) declaration to the `def`, as described in the next section.

Version skew note: Actually, the tongue twisting gets a bit worse. The Python 3.0 `builtins` module used here is named `__builtin__` in Python 2.6. And just for fun, the name `__builtins__` (with the "s") is preset in most global scopes, including the interactive session, to reference the module known as `builtins` (a.k.a. `__builtin__` in 2.6).

That is, after importing `builtins`, `__builtins__ is builtins` is `True` in 3.0, and `__builtins__ is __builtin__` is `True` in 2.6. The net effect is that we can inspect the built-in scope by simply running `dir(__builtins__)` with no import in both 3.0 and 2.6, but we are advised to use `builtins` for real work in 3.0. Who said documenting this stuff was easy?

Breaking the Universe in Python 2.6

Here's another thing you can do in Python that you probably shouldn't—because the names `True` and `False` in 2.6 are just variables in the built-in scope and are not reserved, it's possible to reassign them with a statement like `True = False`. Don't worry, you won't actually break the logical consistency of the universe in so doing! This statement merely redefines the word `True` for the single scope in which it appears. All other scopes still find the originals in the built-in scope.

For more fun, though, in Python 2.6 you could say `__builtin__.True = False`, to reset `True` to `False` for the entire Python process. Alas, this type of assignment has been disallowed in Python 3.0, because `True` and `False` are treated as actual reserved words, just like `None`. In 2.6, though, it sends IDLE into a strange panic state that resets the user code process.

This technique can be useful, however, both to illustrate the underlying namespace model and for tool writers who must change built-ins such as `open` to customized functions. Also, note that third-party tools such as PyChecker will warn about common programming mistakes, including accidental assignment to built-in names (this is known as "shadowing" a built-in in PyChecker).

The global Statement

The `global` statement and its `nonlocal` cousin are the only things that are remotely like declaration statements in Python. They are not type or size declarations, though; they are *namespace declarations*. The `global` statement tells Python that a function plans to change one or more global names—i.e., names that live in the enclosing module's scope (namespace).

We've talked about `global` in passing already. Here's a summary:

- Global names are variables assigned at the top level of the enclosing module file.
- Global names must be declared only if they are assigned within a function.
- Global names may be referenced within a function without being declared.

In other words, `global` allows us to change names that live outside a `def` at the top level of a module file. As we'll see later, the `nonlocal` statement is almost identical but applies to names in the enclosing `def`'s local scope, rather than names in the enclosing module.

The `global` statement consists of the keyword `global`, followed by one or more names separated by commas. All the listed names will be mapped to the enclosing module's scope when assigned or referenced within the function body. For instance:

```
X = 88                          # Global X

def func():
    global X
    X = 99                      # Global X: outside def
```

```
func()
print(X)                    # Prints 99
```

We've added a `global` declaration to the example here, such that the `X` inside the `def` now refers to the `X` outside the `def`; they are the same variable this time. Here is a slightly more involved example of `global` at work:

```
y, z = 1, 2                 # Global variables in module
def all_global():
    global x                # Declare globals assigned
    x = y + z               # No need to declare y, z: LEGB rule
```

Here, `x`, `y`, and `z` are all globals inside the function `all_global`. `y` and `z` are global because they aren't assigned in the function; `x` is global because it was listed in a `global` statement to map it to the module's scope explicitly. Without the `global` here, `x` would be considered local by virtue of the assignment.

Notice that `y` and `z` are not declared global; Python's LEGB lookup rule finds them in the module automatically. Also, notice that `x` might not exist in the enclosing module before the function runs; in this case, the assignment in the function creates `x` in the module.

Minimize Global Variables

By default, names assigned in functions are locals, so if you want to change names outside functions you have to write extra code (e.g., `global` statements). This is by design—as is common in Python, you have to say more to do the potentially "wrong" thing. Although there are times when globals are useful, variables assigned in a `def` are local by default because that is normally the best policy. Changing globals can lead to well-known software engineering problems: because the variables' values are dependent on the order of calls to arbitrarily distant functions, programs can become difficult to debug.

Consider this module file, for example:

```
X = 99
def func1():
    global X
    X = 88

def func2():
    global X
    X = 77
```

Now, imagine that it is your job to modify or reuse this module file. What will the value of `X` be here? Really, that question has no meaning unless it's qualified with a point of reference in time—the value of `X` is timing-dependent, as it depends on which function was called last (something we can't tell from this file alone).

The net effect is that to understand this code, you have to trace the flow of control through the *entire program*. And, if you need to reuse or modify the code, you have to keep the entire program in your head all at once. In this case, you can't really use one of these functions without bringing along the other. They are dependent on (that is, *coupled* with) the global variable. This is the problem with globals—they generally make code more difficult to understand and use than code consisting of self-contained functions that rely on locals.

On the other hand, short of using object-oriented programming and classes, global variables are probably the most straightforward way to retain shared state information (information that a function needs to remember for use the next time it is called) in Python—local variables disappear when the function returns, but globals do not. Other techniques, such as default mutable arguments and enclosing function scopes, can achieve this, too, but they are more complex than pushing values out to the global scope for retention.

Some programs designate a single module to collect globals; as long as this is expected, it is not as harmful. In addition, programs that use multithreading to do parallel processing in Python commonly depend on global variables—they become shared memory between functions running in parallel threads, and so act as a communication device.†

For now, though, especially if you are relatively new to programming, avoid the temptation to use globals whenever you can—try to communicate with passed-in arguments and return values instead. Six months from now, both you and your coworkers will be happy you did.

Minimize Cross-File Changes

Here's another scope-related issue: although we *can* change variables in another file directly, we usually shouldn't. Module files were introduced in Chapter 3 and are covered in more depth in the next part of this book. To illustrate their relationship to scopes, consider these two module files:

```
# first.py
X = 99                  # This code doesn't know about second.py

# second.py
import first
print(first.X)          # Okay: references a name in another file
first.X = 88            # But changing it can be too subtle and implicit
```

† *Multithreading* runs function calls in parallel with the rest of the program and is supported by Python's standard library modules `_thread`, `threading`, and `queue` (`thread`, `threading`, and `Queue` in Python 2.6). Because all threaded functions run in the same process, global scopes often serve as shared memory between them. Threading is commonly used for long-running tasks in GUIs, to implement nonblocking operations in general and to leverage CPU capacity. It is also beyond this book's scope; see the Python library manual, as well as the follow-up texts listed in the Preface (such as O'Reilly's *Programming Python*), for more details.

The first defines a variable X, which the second prints and then changes by assignment. Notice that we must import the first module into the second file to get to its variable at all—as we've learned, each module is a self-contained namespace (package of variables), and we must import one module to see inside it from another. That's the main point about modules: by segregating variables on a per-file basis, they avoid name collisions across files.

Really, though, in terms of this chapter's topic, the global scope of a module file *becomes* the attribute namespace of the module object once it is imported—importers automatically have access to all of the file's global variables, because a file's global scope morphs into an object's attribute namespace when it is imported.

After importing the first module, the second module prints its variable and then assigns it a new value. Referencing the module's variable to print it is fine—this is how modules are linked together into a larger system normally. The problem with the assignment, however, is that it is far too implicit: whoever's charged with maintaining or reusing the first module probably has no clue that some arbitrarily far-removed module on the import chain can change X out from under him at runtime. In fact, the second module may be in a completely different directory, and so difficult to notice at all.

Although such cross-file variable changes are always possible in Python, they are usually much more subtle than you will want. Again, this sets up too strong a *coupling* between the two files—because they are both dependent on the value of the variable X, it's difficult to understand or reuse one file without the other. Such implicit cross-file dependencies can lead to inflexible code at best, and outright bugs at worst.

Here again, the best prescription is generally to not do this—the best way to communicate across file boundaries is to call functions, passing in arguments and getting back return values. In this specific case, we would probably be better off coding an *accessor function* to manage the change:

```
# first.py
X = 99

def setX(new):
    global X
    X = new

# second.py
import first
first.setX(88)
```

This requires more code and may seem like a trivial change, but it makes a huge difference in terms of readability and maintainability—when a person reading the first module by itself sees a function, that person will know that it is a point of *interface* and will expect the change to the X. In other words, it removes the element of surprise that is rarely a good thing in software projects. Although we cannot prevent cross-file changes from happening, common sense dictates that they should be minimized unless widely accepted across the program.

Other Ways to Access Globals

Interestingly, because global-scope variables morph into the attributes of a loaded module object, we can emulate the `global` statement by importing the enclosing module and assigning to its attributes, as in the following example module file. Code in this file imports the enclosing module, first by name, and then by indexing the `sys.modules` loaded modules table (more on this table in Chapter 21):

```
# thismod.py

var = 99                            # Global variable == module attribute

def local():
    var = 0                         # Change local var

def glob1():
    global var                      # Declare global (normal)
    var += 1                        # Change global var

def glob2():
    var = 0                         # Change local var
    import thismod                  # Import myself
    thismod.var += 1                # Change global var

def glob3():
    var = 0                         # Change local var
    import sys                      # Import system table
    glob = sys.modules['thismod']   # Get module object (or use __name__)
    glob.var += 1                   # Change global var

def test():
    print(var)
    local(); glob1(); glob2(); glob3()
    print(var)
```

When run, this adds 3 to the global variable (only the first function does not impact it):

```
>>> import thismod
>>> thismod.test()
99
102
>>> thismod.var
102
```

This works, and it illustrates the equivalence of globals to module attributes, but it's much more work than using the `global` statement to make your intentions explicit.

As we've seen, `global` allows us to change names in a module outside a function. It has a cousin named `nonlocal` that can be used to change names in enclosing functions, too, but to understand how that can be useful, we first need to explore enclosing functions in general.

Scopes and Nested Functions

So far, I've omitted one part of Python's scope rules on purpose, because it's relatively rare to encounter it in practice. However, it's time to take a deeper look at the letter *E* in the LEGB lookup rule. The *E* layer is fairly new (it was added in Python 2.2); it takes the form of the local scopes of any and all enclosing function `def`s. Enclosing scopes are sometimes also called *statically nested scopes*. Really, the nesting is a lexical one—nested scopes correspond to physically and syntactically nested code structures in your program's source code.

Nested Scope Details

With the addition of nested function scopes, variable lookup rules become slightly more complex. Within a function:

- A reference (`X`) looks for the name `X` first in the current local scope (function); then in the local scopes of any lexically enclosing functions in your source code, from inner to outer; then in the current global scope (the module file); and finally in the built-in scope (the module `builtins`). `global` declarations make the search begin in the global (module file) scope instead.
- An assignment (`X = value`) creates or changes the name `X` in the current local scope, by default. If `X` is declared *global* within the function, the assignment creates or changes the name `X` in the enclosing module's scope instead. If, on the other hand, `X` is declared *nonlocal* within the function, the assignment changes the name `X` in the closest enclosing function's local scope.

Notice that the `global` declaration still maps variables to the enclosing module. When nested functions are present, variables in enclosing functions may be referenced, but they require `nonlocal` declarations to be changed.

Nested Scope Examples

To clarify the prior section's points, let's illustrate with some real code. Here is what an enclosing function scope looks like:

```
X = 99                      # Global scope name: not used

def f1():
    X = 88                  # Enclosing def local
    def f2():
        print(X)            # Reference made in nested def
    f2()

f1()                        # Prints 88: enclosing def local
```

First off, this is legal Python code: the `def` is simply an executable statement, which can appear anywhere any other statement can—including nested in another `def`. Here, the

nested `def` runs while a call to the function `f1` is running; it generates a function and assigns it to the name `f2`, a local variable within `f1`'s local scope. In a sense, `f2` is a temporary function that lives only during the execution of (and is visible only to code in) the enclosing `f1`.

But notice what happens inside `f2`: when it prints the variable `X`, it refers to the `X` that lives in the enclosing `f1` function's local scope. Because functions can access names in all physically enclosing `def` statements, the `X` in `f2` is automatically mapped to the `X` in `f1`, by the LEGB lookup rule.

This enclosing scope lookup works even if the enclosing function has already returned. For example, the following code defines a function that makes and returns another function:

```
def f1():
    X = 88
    def f2():
        print(X)          # Remembers X in enclosing def scope
    return f2             # Return f2 but don't call it

action = f1()             # Make, return function
action()                  # Call it now: prints 88
```

In this code, the call to `action` is really running the function we named `f2` when `f1` ran. `f2` remembers the enclosing scope's `X` in `f1`, even though `f1` is no longer active.

Factory functions

Depending on whom you ask, this sort of behavior is also sometimes called a *closure* or *factory* function. These terms refer to a function object that remembers values in enclosing scopes regardless of whether those scopes are still present in memory. Although classes (described in Part VI of this book) are usually best at remembering state because they make it explicit with attribute assignments, such functions provide an alternative.

For instance, factory functions are sometimes used by programs that need to generate event handlers on the fly in response to conditions at runtime (e.g., user inputs that cannot be anticipated). Look at the following function, for example:

```
>>> def maker(N):
...     def action(X):                  # Make and return action
...         return X ** N               # action retains N from enclosing scope
...     return action
...
```

This defines an outer function that simply generates and returns a nested function, without calling it. If we call the outer function:

```
>>> f = maker(2)                        # Pass 2 to N
>>> f
<function action at 0x014720B0>
```

what we get back is a reference to the generated nested function—the one created by running the nested `def`. If we now call what we got back from the outer function:

```
>>> f(3)                          # Pass 3 to X, N remembers 2: 3 ** 2
9
>>> f(4)                          # 4 ** 2
16
```

it invokes the nested function—the one called `action` within `maker`. The most unusual part of this is that the nested function remembers integer `2`, the value of the variable `N` in `maker`, even though `maker` has returned and exited by the time we call `action`. In effect, `N` from the enclosing local scope is retained as state information attached to `action`, and we get back its argument squared.

If we now call the outer function again, we get back a new nested function with different state information attached. That is, we get the argument cubed instead of squared, but the original still squares as before:

```
>>> g = maker(3)                  # g remembers 3, f remembers 2
>>> g(3)                          # 3 ** 3
27
>>> f(3)                          # 3 ** 2
9
```

This works because each call to a factory function like this gets its own set of state information. In our case, the function we assign to name `g` remembers `3`, and `f` remembers `2`, because each has its own state information retained by the variable `N` in `maker`.

This is an advanced technique that you're unlikely to see very often in most code, except among programmers with backgrounds in functional programming languages. On the other hand, enclosing scopes are often employed by `lambda` function-creation expressions (discussed later in this chapter)—because they are expressions, they are almost always nested within a `def`. Moreover, function nesting is commonly used for *decorators* (explored in Chapter 38)—in some cases, it's the most reasonable coding pattern.

As a general rule, *classes* are better at "memory" like this because they make the state retention explicit in attributes. Short of using classes, though, globals, enclosing scope references like these, and default arguments are the main ways that Python functions can retain state information. To see how they compete, Chapter 18 provides complete coverage of defaults, but the next section gives enough of an introduction to get us started.

Retaining enclosing scopes' state with defaults

In earlier versions of Python, the sort of code in the prior section failed because nested `def`s did not do anything about scopes—a reference to a variable within `f2` would search only the local (`f2`), then global (the code outside `f1`), and then built-in scopes. Because it skipped the scopes of enclosing functions, an error would result. To work around this, programmers typically used *default argument values* to pass in and remember the objects in an enclosing scope:

```
def f1():
    x = 88
    def f2(x=x):                # Remember enclosing scope X with defaults
        print(x)
    f2()

f1()                            # Prints 88
```

This code works in all Python releases, and you'll still see this pattern in some existing Python code. In short, the syntax `arg = val` in a `def` header means that the argument `arg` will default to the value `val` if no real value is passed to `arg` in a call.

In the modified `f2` here, the `x=x` means that the argument `x` will default to the value of `x` in the enclosing scope—because the second `x` is evaluated before Python steps into the nested `def`, it still refers to the `x` in `f1`. In effect, the default remembers what `x` was in `f1` (i.e., the object `88`).

That's fairly complex, and it depends entirely on the timing of default value evaluations. In fact, the nested scope lookup rule was added to Python to make defaults unnecessary for this role—today, Python automatically remembers any values required in the enclosing scope for use in nested `def`s.

Of course, the best prescription for most code is simply to avoid nesting `def`s within `def`s, as it will make your programs much simpler. The following is an equivalent of the prior example that banishes the notion of nesting. Notice the forward reference in this code—it's OK to call a function defined after the function that calls it, as long as the second `def` runs before the first function is actually called. Code inside a `def` is never evaluated until the function is actually called:

```
>>> def f1():
...     x = 88                 # Pass x along instead of nesting
...     f2(x)                  # Forward reference okay
...
>>> def f2(x):
...     print(x)
...
>>> f1()
88
```

If you avoid nesting this way, you can almost forget about the nested scopes concept in Python, unless you need to code in the factory function style discussed earlier—at least, for `def` statements. `lambda`s, which almost naturally appear nested in `def`s, often rely on nested scopes, as the next section explains.

Nested scopes and lambdas

While they're rarely used in practice for `def`s themselves, you are more likely to care about nested function scopes when you start coding `lambda` expressions. We won't cover `lambda` in depth until Chapter 19, but in short, it's an expression that generates a new function to be called later, much like a `def` statement. Because it's an expression,

though, it can be used in places that `def` cannot, such as within list and dictionary literals.

Like a `def`, a `lambda` expression introduces a new local scope for the function it creates. Thanks to the enclosing scopes lookup layer, `lambda`s can see all the variables that live in the functions in which they are coded. Thus, the following code works, but only because the nested scope rules are applied:

```
def func():
    x = 4
    action = (lambda n: x ** n)          # x remembered from enclosing def
    return action

x = func()
print(x(2))                              # Prints 16, 4 ** 2
```

Prior to the introduction of nested function scopes, programmers used defaults to pass values from an enclosing scope into `lambda`s, just as for `def`s. For instance, the following works on all Python releases:

```
def func():
    x = 4
    action = (lambda n, x=x: x ** n)     # Pass x in manually
    return action
```

Because `lambda`s are expressions, they naturally (and even normally) nest inside enclosing `def`s. Hence, they are perhaps the biggest beneficiaries of the addition of enclosing function scopes in the lookup rules; in most cases, it is no longer necessary to pass values into `lambda`s with defaults.

Scopes versus defaults with loop variables

There is one notable exception to the rule I just gave: if a `lambda` or `def` defined within a function is nested inside a loop, and the nested function references an enclosing scope variable that is changed by that loop, all functions generated within the loop will have the same value—the value the referenced variable had in the last loop iteration.

For instance, the following attempts to build up a list of functions that each remember the current variable `i` from the enclosing scope:

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                      # Tries to remember each i
...         acts.append(lambda x: i ** x)       # All remember same last i!
...     return acts
...
>>> acts = makeActions()
>>> acts[0]
<function <lambda> at 0x012B16B0>
```

This doesn't quite work, though—because the enclosing scope variable is looked up when the nested functions are later *called*, they all effectively remember the same value

(the value the loop variable had on the *last* loop iteration). That is, we get back 4 to the power of 2 for each function in the list, because `i` is the same in all of them:

```
>>> acts[0](2)                          # All are 4 ** 2, value of last i
16
>>> acts[2](2)                          # This should be 2 ** 2
16
>>> acts[4](2)                          # This should be 4 ** 2
16
```

This is the one case where we still have to explicitly retain enclosing scope values with default arguments, rather than enclosing scope references. That is, to make this sort of code work, we must pass in the *current* value of the enclosing scope's variable with a default. Because defaults are evaluated when the nested function is *created* (not when it's later *called*), each remembers its own value for `i`:

```
>>> def makeActions():
...     acts = []
...     for i in range(5):                     # Use defaults instead
...         acts.append(lambda x, i=i: i ** x) # Remember current i
...     return acts
...
>>> acts = makeActions()
>>> acts[0](2)                          # 0 ** 2
0
>>> acts[2](2)                          # 2 ** 2
4
>>> acts[4](2)                          # 4 ** 2
16
```

This is a fairly obscure case, but it can come up in practice, especially in code that generates callback handler functions for a number of widgets in a GUI (e.g., button-press handlers). We'll talk more about defaults in Chapter 18 and `lambda`s in Chapter 19, so you may want to return and review this section later.‡

Arbitrary scope nesting

Before ending this discussion, I should note that scopes may nest arbitrarily, but only enclosing function `def` statements (not classes, described in Part VI) are searched:

```
>>> def f1():
...     x = 99
...     def f2():
...         def f3():
...             print(x)          # Found in f1's local scope!
...         f3()
```

‡ In the section "Function Gotchas" on page 518 at the end of this part of the book, we'll also see that there is an issue with using mutable objects like lists and dictionaries for default arguments (e.g., `def f(a=[])`)—because defaults are implemented as single objects attached to functions, mutable defaults retain state from call to call, rather then being initialized anew on each call. Depending on whom you ask, this is either considered a feature that supports state retention, or a strange wart on the language. More on this at the end of Chapter 20.

```
...     f2()
...
>>> f1()
99
```

Python will search the local scopes of *all* enclosing `def`s, from inner to outer, after the referencing function's local scope and before the module's global scope or built-ins. However, this sort of code is even less likely to pop up in practice. In Python, we say *flat is better than nested*—except in very limited contexts, your life (and the lives of your coworkers) will generally be better if you minimize nested function definitions.

The nonlocal Statement

In the prior section we explored the way that nested functions can *reference* variables in an enclosing function's scope, even if that function has already returned. It turns out that, as of Python 3.0, we can also *change* such enclosing scope variables, as long as we declare them in `nonlocal` statements. With this statement, nested `def`s can have both read and write access to names in enclosing functions.

The `nonlocal` statement is a close cousin to `global`, covered earlier. Like `global`, `nonlocal` declares that a name will be changed in an enclosing scope. Unlike `global`, though, `nonlocal` applies to a name in an enclosing function's scope, not the global module scope outside all `def`s. Also unlike `global`, `nonlocal` names must already exist in the enclosing function's scope when declared—they can exist only in enclosing functions and cannot be created by a first assignment in a nested `def`.

In other words, `nonlocal` both allows assignment to names in enclosing function scopes and limits scope lookups for such names to enclosing `def`s. The net effect is a more direct and reliable implementation of changeable scope information, for programs that do not desire or need classes with attributes.

nonlocal Basics

Python 3.0 introduces a new `nonlocal` statement, which has meaning only inside a function:

```
def func():
    nonlocal name1, name2, ...
```

This statement allows a nested function to change one or more names defined in a syntactically enclosing function's scope. In Python 2.X (including 2.6), when one function `def` is nested in another, the nested function can reference any of the names defined by assignment in the enclosing `def`'s scope, but it cannot change them. In 3.0, declaring the enclosing scopes' names in a `nonlocal` statement enables nested functions to assign and thus change such names as well.

This provides a way for enclosing functions to provide *writeable* state information, remembered when the nested function is later called. Allowing the state to change

makes it more useful to the nested function (imagine a counter in the enclosing scope, for instance). In 2.X, programmers usually achieve similar goals by using classes or other schemes. Because nested functions have become a more common coding pattern for state retention, though, `nonlocal` makes it more generally applicable.

Besides allowing names in enclosing `def`s to be changed, the `nonlocal` statement also forces the issue for references—just like the `global` statement, `nonlocal` causes searches for the names listed in the statement to begin in the enclosing `def`s' scopes, not in the local scope of the declaring function. That is, `nonlocal` also means "skip my local scope entirely."

In fact, the names listed in a `nonlocal` *must* have been previously defined in an enclosing `def` when the `nonlocal` is reached, or an error is raised. The net effect is much like `global`: `global` means the names reside in the enclosing module, and `nonlocal` means they reside in an enclosing `def`. `nonlocal` is even more strict, though—scope search is restricted to *only* enclosing `def`s. That is, nonlocal names can appear only in enclosing `def`s, not in the module's global scope or built-in scopes outside the `def`s.

The addition of `nonlocal` does not alter name reference scope rules in general; they still work as before, per the "LEGB" rule described earlier. The `nonlocal` statement mostly serves to allow names in enclosing scopes to be changed rather than just referenced. However, `global` and `nonlocal` statements do both restrict the lookup rules somewhat, when coded in a function:

- `global` makes scope lookup begin in the enclosing module's scope and allows names there to be assigned. Scope lookup continues on to the built-in scope if the name does not exist in the module, but assignments to global names always create or change them in the module's scope.
- `nonlocal` restricts scope lookup to just enclosing `def`s, requires that the names already exist there, and allows them to be assigned. Scope lookup does not continue on to the global or built-in scopes.

In Python 2.6, references to enclosing `def` scope names are allowed, but not assignment. However, you can still use classes with explicit attributes to achieve the same changeable state information effect as nonlocals (and you may be better off doing so in some contexts); globals and function attributes can sometimes accomplish similar goals as well. More on this in a moment; first, let's turn to some working code to make this more concrete.

nonlocal in Action

On to some examples, all run in 3.0. References to enclosing `def` scopes work as they do in 2.6. In the following, `tester` builds and returns the function `nested`, to be called later, and the `state` reference in `nested` maps the local scope of `tester` using the normal scope lookup rules:

```
C:\\misc>c:\python30\python

>>> def tester(start):
...     state = start             # Referencing nonlocals works normally
...     def nested(label):
...         print(label, state)   # Remembers state in enclosing scope
...     return nested
...
>>> F = tester(0)
>>> F('spam')
spam 0
>>> F('ham')
ham 0
```

Changing a name in an enclosing `def`'s scope is not allowed by default, though; this is the normal case in 2.6 as well:

```
>>> def tester(start):
...     state = start
...     def nested(label):
...         print(label, state)
...         state += 1            # Cannot change by default (or in 2.6)
...     return nested
...
>>> F = tester(0)
>>> F('spam')
UnboundLocalError: local variable 'state' referenced before assignment
```

Using nonlocal for changes

Now, under 3.0, if we declare `state` in the `tester` scope as `nonlocal` within `nested`, we get to change it inside the nested function, too. This works even though `tester` has returned and exited by the time we call the returned `nested` function through the name `F`:

```
>>> def tester(start):
...     state = start             # Each call gets its own state
...     def nested(label):
...         nonlocal state        # Remembers state in enclosing scope
...         print(label, state)
...         state += 1            # Allowed to change it if nonlocal
...     return nested
...
>>> F = tester(0)
>>> F('spam')                     # Increments state on each call
spam 0
>>> F('ham')
ham 1
>>> F('eggs')
eggs 2
```

As usual with enclosing scope references, we can call the `tester` factory function multiple times to get multiple copies of its state in memory. The `state` object in the enclosing scope is essentially attached to the `nested` function object returned; each call makes a

new, distinct `state` object, such that updating one function's state won't impact the other. The following continues the prior listing's interaction:

```
>>> G = tester(42)              # Make a new tester that starts at 42
>>> G('spam')
spam 42

>>> G('eggs')                   # My state information updated to 43
eggs 43

>>> F('bacon')                  # But F's is where it left off: at 3
bacon 3                         # Each call has different state information
```

Boundary cases

There are a few things to watch out for. First, unlike the `global` statement, `nonlocal` names really *must* have previously been assigned in an enclosing `def`'s scope when a `nonlocal` is evaluated, or else you'll get an error—you cannot create them dynamically by assigning them anew in the enclosing scope:

```
>>> def tester(start):
...     def nested(label):
...         nonlocal state       # Nonlocals must already exist in enclosing def!
...         state = 0
...         print(label, state)
...     return nested
...
SyntaxError: no binding for nonlocal 'state' found

>>> def tester(start):
...     def nested(label):
...         global state         # Globals don't have to exist yet when declared
...         state = 0            # This creates the name in the module now
...         print(label, state)
...     return nested
...
>>> F = tester(0)
>>> F('abc')
abc 0
>>> state
0
```

Second, `nonlocal` restricts the scope lookup to just enclosing `def`s; nonlocals are not looked up in the enclosing module's global scope or the built-in scope outside all `def`s, even if they are already there:

```
>>> spam = 99
>>> def tester():
...     def nested():
...         nonlocal spam        # Must be in a def, not the module!
...         print('Current=', spam)
...         spam += 1
...     return nested
...
SyntaxError: no binding for nonlocal 'spam' found
```

These restrictions make sense once you realize that Python would not otherwise generally know which enclosing scope to create a brand new name in. In the prior listing, should `spam` be assigned in `tester`, or the module outside? Because this is ambiguous, Python must resolve nonlocals at function *creation* time, not function *call* time.

Why nonlocal?

Given the extra complexity of nested functions, you might wonder what the fuss is about. Although it's difficult to see in our small examples, state information becomes crucial in many programs. There are a variety of ways to "remember" information across function and method calls in Python. While there are tradeoffs for all, `nonlocal` does improve this story for enclosing scope references—the `nonlocal` statement allows multiple copies of changeable state to be retained in memory and addresses simple state-retention needs where classes may not be warranted.

As we saw in the prior section, the following code allows state to be retained and modified in an enclosing scope. Each call to `tester` creates a little self-contained *package of changeable information*, whose names do not clash with any other part of the program:

```
def tester(start):
    state = start                    # Each call gets its own state
    def nested(label):
        nonlocal state               # Remembers state in enclosing scope
        print(label, state)
        state += 1                   # Allowed to change it if nonlocal
    return nested

F = tester(0)
F('spam')
```

Unfortunately, this code only works in Python 3.0. If you are using Python 2.6, other options are available, depending on your goals. The next two sections present some alternatives.

Shared state with globals

One usual prescription for achieving the `nonlocal` effect in 2.6 and earlier is to simply move the state out to the *global scope* (the enclosing module):

```
>>> def tester(start):
...     global state                # Move it out to the module to change it
...     state = start               # global allows changes in module scope
...     def nested(label):
...         global state
...         print(label, state)
...         state += 1
...     return nested
...
>>> F = tester(0)
>>> F('spam')                       # Each call increments shared global state
```

```
spam 0
>>> F('eggs')
eggs 1
```

This works in this case, but it requires `global` declarations in both functions and is prone to name collisions in the global scope (what if "state" is already being used?). A worse, and more subtle, problem is that it only allows for a *single shared copy* of the state information in the module scope—if we call `tester` again, we'll wind up resetting the module's `state` variable, such that prior calls will see their `state` overwritten:

```
>>> G = tester(42)                  # Resets state's single copy in global scope
>>> G('toast')
toast 42

>>> G('bacon')
bacon 43

>>> F('ham')                        # Oops -- my counter has been overwritten!
ham 44
```

As shown earlier, when using `nonlocal` instead of `global`, each call to `tester` remembers its own unique copy of the `state` object.

State with classes (preview)

The other prescription for changeable state information in 2.6 and earlier is to use *classes with attributes* to make state information access more explicit than the implicit magic of scope lookup rules. As an added benefit, each instance of a class gets a fresh copy of the state information, as a natural byproduct of Python's object model.

We haven't explored classes in detail yet, but as a brief preview, here is a reformulation of the `tester`/`nested` functions used earlier as a class—state is recorded in objects explicitly as they are created. To make sense of this code, you need to know that a `def` within a `class` like this works exactly like a `def` outside of a `class`, except that the function's `self` argument automatically receives the implied subject of the call (an instance object created by calling the class itself):

```
>>> class tester:                   # Class-based alternative (see Part VI)
...     def __init__(self, start):  # On object construction,
...         self.state = start      # save state explicitly in new object
...     def nested(self, label):
...         print(label, self.state)    # Reference state explicitly
...         self.state += 1             # Changes are always allowed
...
>>> F = tester(0)                   # Create instance, invoke __init__
>>> F.nested('spam')                # F is passed to self
spam 0
>>> F.nested('ham')
ham 1

>>> G = tester(42)                  # Each instance gets new copy of state
>>> G.nested('toast')               # Changing one does not impact others
toast 42
```

```
>>> G.nested('bacon')
bacon 43

>>> F.nested('eggs')                    # F's state is where it left off
eggs 2
>>> F.state                             # State may be accessed outside class
3
```

With just slightly more magic, which we'll delve into later in this book, we could also make our class look like a callable function using operator overloading. `__call__` intercepts direct calls on an instance, so we don't need to call a named method:

```
>>> class tester:
...     def __init__(self, start):
...         self.state = start
...     def __call__(self, label):          # Intercept direct instance calls
...         print(label, self.state)        # So .nested() not required
...         self.state += 1
...
>>> H = tester(99)
>>> H('juice')                          # Invokes __call__
juice 99
>>> H('pancakes')
pancakes 100
```

Don't sweat the details in this code too much at this point in the book; we'll explore classes in depth in Part VI and will look at specific operator overloading tools like `__call__` in Chapter 29, so you may wish to file this code away for future reference. The point here is that classes can make state information more obvious, by leveraging explicit attribute assignment instead of scope lookups.

While using classes for state information is generally a good rule of thumb to follow, they might be overkill in cases like this, where state is a single counter. Such trivial state cases are more common than you might think; in such contexts, nested `def`s are sometimes more lightweight than coding classes, especially if you're not familiar with OOP yet. Moreover, there are some scenarios in which nested `def`s may actually work better than classes (see the description of *method decorators* in Chapter 38 for an example that is far beyond this chapter's scope).

State with function attributes

As a final state-retention option, we can also sometimes achieve the same effect as nonlocals with *function attributes*—user-defined names attached to functions directly. Here's a final version of our example based on this technique—it replaces a nonlocal with an attribute attached to the nested function. Although this scheme may not be as intuitive to some, it also allows the state variable to be accessed *outside* the nested function (with nonlocals, we can only see state variables within the nested `def`):

```
>>> def tester(start):
...     def nested(label):
...         print(label, nested.state)      # nested is in enclosing scope
...         nested.state += 1               # Change attr, not nested itself
```

```
...     nested.state = start              # Initial state after func defined
...     return nested
...
>>> F = tester(0)
>>> F('spam')                            # F is a 'nested' with state attached
spam 0
>>> F('ham')
ham 1
>>> F.state                              # Can access state outside functions too
2
>>>
>>> G = tester(42)                       # G has own state, doesn't overwrite F's
>>> G('eggs')
eggs 42
>>> F('ham')
ham 2
```

This code relies on the fact that the function name `nested` is a local variable in the `tester` scope enclosing `nested`; as such, it can be referenced freely inside `nested`. This code also relies on the fact that changing an object in-place is not an assignment to a name; when it increments `nested.state`, it is changing part of the object `nested` references, not the name `nested` itself. Because we're not really assigning a name in the enclosing scope, no `nonlocal` is needed.

As you can see, globals, nonlocals, classes, and function attributes all offer state-retention options. Globals only support shared data, classes require a basic knowledge of OOP, and both classes and function attributes allow state to be accessed outside the nested function itself. As usual, the best tool for your program depends upon your program's goals.

Chapter Summary

In this chapter, we studied one of two key concepts related to functions: *scopes* (how variables are looked up when they are used). As we learned, variables are considered local to the function definitions in which they are assigned, unless they are specifically declared to be global or nonlocal. We also studied some more advanced scope concepts here, including nested function scopes and function attributes. Finally, we looked at some general design ideas, such as the need to avoid globals and cross-file changes.

In the next chapter, we're going to continue our function tour with the second key function-related concept: argument passing. As we'll find, arguments are passed into a function by assignment, but Python also provides tools that allow functions to be flexible in how items are passed. Before we move on, let's take this chapter's quiz to review the scope concepts we've covered here.

Test Your Knowledge: Quiz

1. What is the output of the following code, and why?

```
>>> X = 'Spam'
>>> def func():
...     print(X)
...
>>> func()
```

2. What is the output of this code, and why?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI!'
...
>>> func()
>>> print(X)
```

3. What does this code print, and why?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     print(X)
...
>>> func()
>>> print(X)
```

4. What output does this code produce? Why?

```
>>> X = 'Spam'
>>> def func():
...     global X
...     X = 'NI'
...
>>> func()
>>> print(X)
```

5. What about this code—what's the output, and why?

```
>>> X = 'Spam'
>>> def func():
...     X = 'NI'
...     def nested():
...         print(X)
...     nested()
...
>>> func()
>>> X
```

6. How about this example: what is its output in Python 3.0, and why?

```
>>> def func():
...     X = 'NI'
...     def nested():
...         nonlocal X
...         X = 'Spam'
...     nested()
...     print(X)
...
>>> func()
```

7. Name three or more ways to retain state information in a Python function.

Test Your Knowledge: Answers

1. The output here is `'Spam'`, because the function references a global variable in the enclosing module (because it is not assigned in the function, it is considered global).
2. The output here is `'Spam'` again because assigning the variable inside the function makes it a local and effectively hides the global of the same name. The `print` statement finds the variable unchanged in the global (module) scope.
3. It prints `'NI'` on one line and `'Spam'` on another, because the reference to the variable within the function finds the assigned local and the reference in the `print` statement finds the global.
4. This time it just prints `'NI'` because the global declaration forces the variable assigned inside the function to refer to the variable in the enclosing global scope.
5. The output in this case is again `'NI'` on one line and `'Spam'` on another, because the `print` statement in the nested function finds the name in the enclosing function's local scope, and the `print` at the end finds the variable in the global scope.
6. This example prints `'Spam'`, because the `nonlocal` statement (available in Python 3.0 but not 2.6) means that the assignment to `X` inside the nested function changes `X` in the enclosing function's local scope. Without this statement, this assignment would classify `X` as local to the nested function, making it a different variable; the code would then print `'NI'` instead.
7. Although the values of local variables go away when a function returns, you can make a Python function retain state information by using shared global variables, enclosing function scope references within nested functions, or using default argument values. Function attributes can sometimes allow state to be attached to the function itself, instead of looked up in scopes. Another alternative, using OOP with classes, sometimes supports state retention better than any of the scope-based techniques because it makes it explicit with attribute assignments; we'll explore this option in Part VI.

CHAPTER 18
Arguments

Chapter 17 explored the details behind Python's *scopes*—the places where variables are defined and looked up. As we learned, the place where a name is defined in our code determines much of its meaning. This chapter continues the function story by studying the concepts in Python *argument passing*—the way that objects are sent to functions as inputs. As we'll see, arguments (a.k.a. parameters) are assigned to names in a function, but they have more to do with object references than with variable scopes. We'll also find that Python provides extra tools, such as keywords, defaults, and arbitrary argument collectors, that allow for wide flexibility in the way arguments are sent to a function.

Argument-Passing Basics

Earlier in this part of the book, I noted that arguments are passed by *assignment*. This has a few ramifications that aren't always obvious to beginners, which I'll expand on in this section. Here is a rundown of the key points in passing arguments to functions:

- **Arguments are passed by automatically assigning objects to local variable names**. Function arguments—references to (possibly) shared objects sent by the caller—are just another instance of Python assignment at work. Because references are implemented as pointers, all arguments are, in effect, passed by pointer. Objects passed as arguments are never automatically copied.
- **Assigning to argument names inside a function does not affect the caller**. Argument names in the function header become new, local names when the function runs, in the scope of the function. There is no aliasing between function argument names and variable names in the scope of the caller.
- **Changing a mutable object argument in a function may impact the caller**. On the other hand, as arguments are simply assigned to passed-in objects, functions can change passed-in mutable objects in place, and the results may affect the caller. Mutable arguments can be input and output for functions.

For more details on *references*, see Chapter 6; everything we learned there also applies to function arguments, though the assignment to argument names is automatic and implicit.

Python's pass-by-assignment scheme isn't quite the same as C++'s reference parameters option, but it turns out to be very similar to the C language's argument-passing model in practice:

- **Immutable arguments are effectively passed "by value."** Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in-place anyhow, the effect is much like making a copy.
- **Mutable arguments are effectively passed "by pointer."** Objects such as lists and dictionaries are also passed by object reference, which is similar to the way C passes arrays as pointers—mutable objects can be changed in-place in the function, much like C arrays.

Of course, if you've never used C, Python's argument-passing mode will seem simpler still—it involves just the assignment of objects to names, and it works the same whether the objects are mutable or not.

Arguments and Shared References

To illustrate argument-passing properties at work, consider the following code:

```
>>> def f(a):                  # a is assigned to (references) passed object
...     a = 99                 # Changes local variable a only
...
>>> b = 88
>>> f(b)                       # a and b both reference same 88 initially
>>> print(b)                   # b is not changed
88
```

In this example the variable `a` is assigned the object `88` at the moment the function is called with `f(b)`, but `a` lives only within the called function. Changing `a` inside the function has no effect on the place where the function is called; it simply resets the local variable `a` to a completely different object.

That's what is meant by a lack of name *aliasing*—assignment to an argument name inside a function (e.g., `a=99`) does not magically change a variable like `b` in the scope of the function call. Argument names may share passed objects initially (they are essentially pointers to those objects), but only temporarily, when the function is first called. As soon as an argument name is reassigned, this relationship ends.

At least, that's the case for assignment to argument *names* themselves. When arguments are passed *mutable* objects like lists and dictionaries, we also need to be aware that in-place changes to such *objects* may live on after a function exits, and hence impact callers. Here's an example that demonstrates this behavior:

```
>>> def changer(a, b):          # Arguments assigned references to objects
...     a = 2                   # Changes local name's value only
...     b[0] = 'spam'           # Changes shared object in-place
...
>>> X = 1
>>> L = [1, 2]                  # Caller
>>> changer(X, L)               # Pass immutable and mutable objects
>>> X, L                        # X is unchanged, L is different!
(1, ['spam', 2])
```

In this code, the `changer` function assigns values to argument `a` itself, and to a component of the object referenced by argument `b`. These two assignments within the function are only slightly different in syntax but have radically different results:

- Because `a` is a local variable name in the function's scope, the first assignment has no effect on the caller—it simply changes the local variable `a` to reference a completely different object, and does not change the binding of the name `X` in the caller's scope. This is the same as in the prior example.
- Argument `b` is a local variable name, too, but it is passed a mutable object (the list that `L` references in the caller's scope). As the second assignment is an in-place object change, the result of the assignment to `b[0]` in the function impacts the value of `L` after the function returns.

Really, the second assignment statement in `changer` doesn't change `b`—it changes part of the object that `b` currently references. This in-place change impacts the caller only because the changed object outlives the function call. The name `L` hasn't changed either—it still references the same, changed object—but it seems as though `L` differs after the call because the value it references has been modified within the function.

Figure 18-1 illustrates the name/object bindings that exist immediately after the function has been called, and before its code has run.

If this example is still confusing, it may help to notice that the effect of the automatic assignments of the passed-in arguments is the same as running a series of simple assignment statements. In terms of the first argument, the assignment has no effect on the caller:

```
>>> X = 1
>>> a = X               # They share the same object
>>> a = 2               # Resets 'a' only, 'X' is still 1
>>> print(X)
1
```

The assignment through the second argument does affect a variable at the call, though, because it is an in-place object change:

```
>>> L = [1, 2]
>>> b = L               # They share the same object
>>> b[0] = 'spam'       # In-place change: 'L' sees the change too
>>> print(L)
['spam', 2]
```

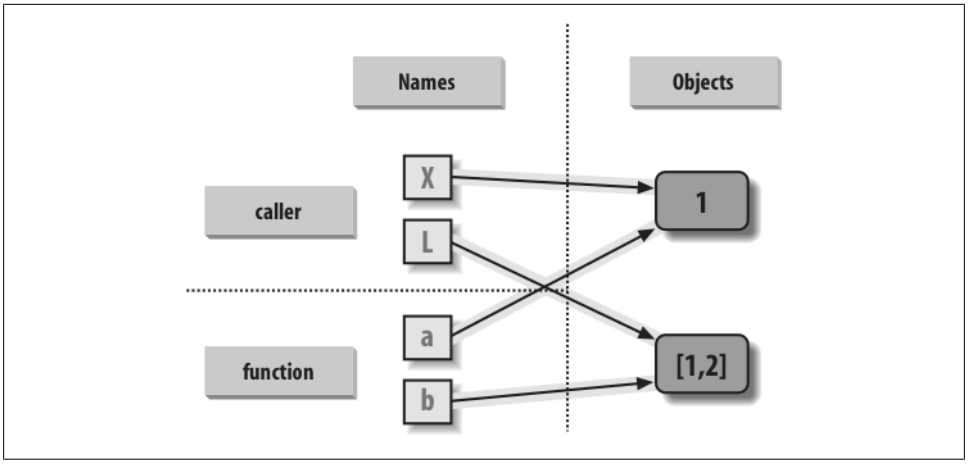

Figure 18-1. References: arguments. Because arguments are passed by assignment, argument names in the function may share objects with variables in the scope of the call. Hence, in-place changes to mutable arguments in a function can impact the caller. Here, a and b in the function initially reference the objects referenced by variables X and L when the function is first called. Changing the list through variable b makes L appear different after the call returns.

If you recall our discussions about shared mutable objects in Chapters 6 and 9, you'll recognize the phenomenon at work: changing a mutable object in-place can impact other references to that object. Here, the effect is to make one of the arguments work like both an input and an *output* of the function.

Avoiding Mutable Argument Changes

This behavior of in-place changes to mutable arguments isn't a bug—it's simply the way argument passing works in Python. Arguments are passed to functions by reference (a.k.a. pointer) by default because that is what we normally want. It means we can pass large objects around our programs without making multiple copies along the way, and we can easily update these objects as we go. In fact, as we'll see in Part VI, Python's class model *depends* upon changing a passed-in "self" argument in-place, to update object state.

If we don't want in-place changes within functions to impact objects we pass to them, though, we can simply make explicit copies of mutable objects, as we learned in Chapter 6. For function arguments, we can always copy the list at the point of call:

```
L = [1, 2]
changer(X, L[:])        # Pass a copy, so our 'L' does not change
```

We can also copy within the function itself, if we never want to change passed-in objects, regardless of how the function is called:

```
def changer(a, b):
    b = b[:]            # Copy input list so we don't impact caller
```

```
a = 2
b[0] = 'spam'          # Changes our list copy only
```

Both of these copying schemes don't stop the function from changing the object—they just prevent those changes from impacting the caller. To really prevent changes, we can always convert to immutable objects to force the issue. Tuples, for example, throw an exception when changes are attempted:

```
L = [1, 2]
changer(X, tuple(L))    # Pass a tuple, so changes are errors
```

This scheme uses the built-in `tuple` function, which builds a new tuple out of all the items in a sequence (really, any iterable). It's also something of an extreme—because it forces the function to be written to never change passed-in arguments, this solution might impose more limitations on the function than it should, and so should generally be avoided (you never know when changing arguments might come in handy for other calls in the future). Using this technique will also make the function lose the ability to call any list-specific methods on the argument, including methods that do not change the object in-place.

The main point to remember here is that functions might update mutable objects like lists and dictionaries passed into them. This isn't necessarily a problem if it's expected, and often serves useful purposes. Moreover, functions that change passed-in mutable objects in place are probably designed and intended to do so—the change is likely part of a well-defined API that you shouldn't violate by making copies.

However, you do have to be aware of this property—if objects change out from under you unexpectedly, check whether a called function might be responsible, and make copies when objects are passed if needed.

Simulating Output Parameters

We've already discussed the `return` statement and used it in a few examples. Here's another way to use this statement: because `return` can send back any sort of object, it can return *multiple values* by packaging them in a tuple or other collection type. In fact, although Python doesn't support what some languages label "call-by-reference" argument passing, we can usually simulate it by returning tuples and assigning the results back to the original argument names in the caller:

```
>>> def multiple(x, y):
...     x = 2                  # Changes local names only
...     y = [3, 4]
...     return x, y            # Return new values in a tuple
...
>>> X = 1
>>> L = [1, 2]
>>> X, L = multiple(X, L)     # Assign results to caller's names
>>> X, L
(2, [3, 4])
```

It looks like the code is returning two values here, but it's really just one—a two-item tuple with the optional surrounding parentheses omitted. After the call returns, we can use tuple assignment to unpack the parts of the returned tuple. (If you've forgotten why this works, flip back to "Tuples" on page 225 in Chapter 4, Chapter 9, and "Assignment Statements" on page 279 in Chapter 11.) The net effect of this coding pattern is to simulate the output parameters of other languages by explicit assignments. `X` and `L` change after the call, but only because the code said so.

Unpacking arguments in Python 2.X: The preceding example unpacks a tuple returned by the function with tuple assignment. In Python 2.6, it's also possible to automatically unpack tuples in arguments passed to a function. In 2.6, a function defined by this header:

```
def f((a, (b, c))):
```

can be called with tuples that match the expected structure: `f((1, (2, 3)))` assigns `a`, `b`, and `c` to `1`, `2`, and `3`, respectively. Naturally, the passed tuple can also be an object created before the call (`f(T)`). This `def` syntax is no longer supported in Python 3.0. Instead, code this function as:

```
def f(T): (a, (b, c)) = T
```

to unpack in an explicit assignment statement. This explicit form works in both 3.0 and 2.6. Argument unpacking is an obscure and rarely used feature in Python 2.X. Moreover, a function header in 2.6 supports only the tuple form of sequence assignment; more general sequence assignments (e.g., `def f((a, [b, c])):`) fail on syntax errors in 2.6 as well and require the explicit assignment form.

Tuple unpacking argument syntax is also disallowed by 3.0 in `lambda` function argument lists: see the sidebar "Why You Will Care: List Comprehensions and map" on page 491 for an example. Somewhat asymmetrically, tuple unpacking assignment is still automatic in 3.0 `for` loops targets, though; see Chapter 13 for examples.

Special Argument-Matching Modes

As we've just seen, arguments are always passed by *assignment* in Python; names in the `def` header are assigned to passed-in objects. On top of this model, though, Python provides additional tools that alter the way the argument objects in a call are *matched* with argument names in the header prior to assignment. These tools are all optional, but they allow us to write functions that support more flexible calling patterns, and you may encounter some libraries that require them.

By default, arguments are matched by position, from left to right, and you must pass exactly as many arguments as there are argument names in the function header. However, you can also specify matching by name, default values, and collectors for extra arguments.

The Basics

Before we go into the syntactic details, I want to stress that these special modes are optional and only have to do with matching objects to names; the underlying passing mechanism after the matching takes place is still assignment. In fact, some of these tools are intended more for people writing libraries than for application developers. But because you may stumble across these modes even if you don't code them yourself, here's a synopsis of the available tools:

Positionals: matched from left to right
: The normal case, which we've mostly been using so far, is to match passed argument values to argument names in a function header by position, from left to right.

Keywords: matched by argument name
: Alternatively, callers can specify which argument in the function is to receive a value by using the argument's name in the call, with the `name=value` syntax.

Defaults: specify values for arguments that aren't passed
: Functions themselves can specify default values for arguments to receive if the call passes too few values, again using the `name=value` syntax.

Varargs collecting: collect arbitrarily many positional or keyword arguments
: Functions can use special arguments preceded with one or two * characters to collect an arbitrary number of extra arguments (this feature is often referred to as *varargs*, after the varargs feature in the C language, which also supports variable-length argument lists).

Varargs unpacking: pass arbitrarily many positional or keyword arguments
: Callers can also use the * syntax to unpack argument collections into discrete, separate arguments. This is the inverse of a * in a function header—in the header it means collect arbitrarily many arguments, while in the call it means pass arbitrarily many arguments.

Keyword-only arguments: arguments that must be passed by name
: In Python 3.0 (but not 2.6), functions can also specify arguments that must be passed by name with keyword arguments, not by position. Such arguments are typically used to define configuration options in addition to actual arguments.

Matching Syntax

Table 18-1 summarizes the syntax that invokes the special argument-matching modes.

Table 18-1. Function argument-matching forms

Syntax	Location	Interpretation
`func(value)`	Caller	Normal argument: matched by position
`func(name=value)`	Caller	Keyword argument: matched by name
`func(*sequence)`	Caller	Pass all objects in sequence as individual positional arguments
`func(**dict)`	Caller	Pass all key/value pairs in `dict` as individual keyword arguments
`def func(name)`	Function	Normal argument: matches any passed value by position or name
`def func(name=value)`	Function	Default argument value, if not passed in the call
`def func(*name)`	Function	Matches and collects remaining positional arguments in a tuple
`def func(**name)`	Function	Matches and collects remaining keyword arguments in a dictionary
`def func(*args, name)`	Function	Arguments that must be passed by keyword only in calls (3.0)
`def func(*, name=value)`		

These special matching modes break down into function calls and definitions as follows:

- In a *function call* (the first four rows of the table), simple values are matched by position, but using the `name=value` form tells Python to match by name to arguments instead; these are called *keyword arguments*. Using a `*sequence` or `**dict` in a call allows us to package up arbitrarily many positional or keyword objects in sequences and dictionaries, respectively, and unpack them as separate, individual arguments when they are passed to the function.
- In a *function header* (the rest of the table), a simple `name` is matched by position or name depending on how the caller passes it, but the `name=value` form specifies a *default value*. The `*name` form collects any extra unmatched positional arguments in a tuple, and the `**name` form collects extra keyword arguments in a dictionary. In Python 3.0 and later, any normal or defaulted argument names following a `*name` or a bare * are *keyword-only* arguments and must be passed by keyword in calls.

Of these, keyword arguments and defaults are probably the most commonly used in Python code. We've informally used both of these earlier in this book:

- We've already used *keywords* to specify options to the 3.0 `print` function, but they are more general—keywords allow us to label any argument with its name, to make calls more informational.

- We met *defaults* earlier, too, as a way to pass in values from the enclosing function's scope, but they are also more general—they allow us to make any argument optional, providing its default value in a function definition.

As we'll see, the combination of defaults in a function header and keywords in a call further allows us to pick and choose which defaults to override.

In short, special argument-matching modes let you be fairly liberal about how many arguments must be passed to a function. If a function specifies defaults, they are used if you pass *too few* arguments. If a function uses the * variable argument list forms, you can pass *too many* arguments; the * names collect the extra arguments in data structures for processing in the function.

The Gritty Details

If you choose to use and combine the special argument-matching modes, Python will ask you to follow these ordering rules:

- In a function *call*, arguments must appear in this order: any positional arguments (`value`), followed by a combination of any keyword arguments (`name=value`) and the `*sequence` form, followed by the `**dict` form.
- In a function *header*, arguments must appear in this order: any normal arguments (`name`), followed by any default arguments (`name=value`), followed by the `*name` (or * in 3.0) form if present, followed by any `name` or `name=value` keyword-only arguments (in 3.0), followed by the `**name` form.

In both the call and header, the `**arg` form must appear last if present. If you mix arguments in any other order, you will get a syntax error because the combinations can be ambiguous. The steps that Python internally carries out to match arguments before assignment can roughly be described as follows:

1. Assign nonkeyword arguments by position.
2. Assign keyword arguments by matching names.
3. Assign extra nonkeyword arguments to `*name` tuple.
4. Assign extra keyword arguments to `**name` dictionary.
5. Assign default values to unassigned arguments in header.

After this, Python checks to make sure each argument is passed just one value; if not, an error is raised. When all matching is complete, Python assigns argument names to the objects passed to them.

The actual matching algorithm Python uses is a bit more complex (it must also account for keyword-only arguments in 3.0, for instance), so we'll defer to Python's standard language manual for a more exact description. It's not required reading, but tracing Python's matching algorithm may help you to understand some convoluted cases, especially when modes are mixed.

In Python 3.0, argument names in a function header can also have *annotation* values, specified as `name:value` (or `name:value=default` when defaults are present). This is simply additional syntax for arguments and does not augment or change the argument-ordering rules described here. The function itself can also have an annotation value, given as `def f()->value`. See the discussion of function annotation in Chapter 19 for more details.

Keyword and Default Examples

This is all simpler in code than the preceding descriptions may imply. If you don't use any special matching syntax, Python matches names by position from left to right, like most other languages. For instance, if you define a function that requires three arguments, you must call it with three arguments:

```
>>> def f(a, b, c): print(a, b, c)
...
```

Here, we pass them by position—`a` is matched to `1`, `b` is matched to `2`, and so on (this works the same in Python 3.0 and 2.6, but extra tuple parentheses are displayed in 2.6 because we're using 3.0 `print` calls):

```
>>> f(1, 2, 3)
1 2 3
```

Keywords

In Python, though, you can be more specific about what goes where when you call a function. Keyword arguments allow us to match by *name*, instead of by position:

```
>>> f(c=3, b=2, a=1)
1 2 3
```

The `c=3` in this call, for example, means send `3` to the argument named `c`. More formally, Python matches the name `c` in the call to the argument named `c` in the function definition's header, and then passes the value `3` to that argument. The net effect of this call is the same as that of the prior call, but notice that the left-to-right order of the arguments no longer matters when keywords are used because arguments are matched by name, not by position. It's even possible to combine positional and keyword arguments in a single call. In this case, all positionals are matched first from left to right in the header, before keywords are matched by name:

```
>>> f(1, c=3, b=2)
1 2 3
```

When most people see this the first time, they wonder why one would use such a tool. Keywords typically have two roles in Python. First, they make your calls a bit more self-documenting (assuming that you use better argument names than `a`, `b`, and `c`). For example, a call of this form:

```
func(name='Bob', age=40, job='dev')
```

is much more meaningful than a call with three naked values separated by commas—the keywords serve as labels for the data in the call. The second major use of keywords occurs in conjunction with defaults, which we turn to next.

Defaults

We talked about defaults in brief earlier, when discussing nested function scopes. In short, defaults allow us to make selected function arguments optional; if not passed a value, the argument is assigned its default before the function runs. For example, here is a function that requires one argument and defaults two:

```
>>> def f(a, b=2, c=3): print(a, b, c)
...
```

When we call this function, we must provide a value for `a`, either by position or by keyword; however, providing values for `b` and `c` is optional. If we don't pass values to `b` and `c`, they default to `2` and `3`, respectively:

```
>>> f(1)
1 2 3
>>> f(a=1)
1 2 3
```

If we pass two values, only `c` gets its default, and with three values, no defaults are used:

```
>>> f(1, 4)
1 4 3
>>> f(1, 4, 5)
1 4 5
```

Finally, here is how the keyword and default features interact. Because they subvert the normal left-to-right positional mapping, keywords allow us to essentially skip over arguments with defaults:

```
>>> f(1, c=6)
1 2 6
```

Here, `a` gets `1` by position, `c` gets `6` by keyword, and `b`, in between, defaults to `2`.

Be careful not to confuse the special `name=value` syntax in a function header and a function call; in the call it means a match-by-name keyword argument, while in the header it specifies a default for an optional argument. In both cases, this is not an assignment statement (despite its appearance); it is special syntax for these two contexts, which modifies the default argument-matching mechanics.

Combining keywords and defaults

Here is a slightly larger example that demonstrates keywords and defaults in action. In the following, the caller must always pass at least two arguments (to match `spam` and `eggs`), but the other two are optional. If they are omitted, Python assigns `toast` and `ham` to the defaults specified in the header:

```
def func(spam, eggs, toast=0, ham=0):    # First 2 required
    print((spam, eggs, toast, ham))

func(1, 2)                               # Output: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                   # Output: (1, 0, 0, 1)
func(spam=1, eggs=0)                     # Output: (1, 0, 0, 0)
func(toast=1, eggs=2, spam=3)            # Output: (3, 2, 1, 0)
func(1, 2, 3, 4)                         # Output: (1, 2, 3, 4)
```

Notice again that when keyword arguments are used in the call, the order in which the arguments are listed doesn't matter; Python matches by name, not by position. The caller must supply values for `spam` and `eggs`, but they can be matched by position or by name. Again, keep in mind that the form `name=value` means different things in the call and the `def`: a keyword in the call and a default in the header.

Arbitrary Arguments Examples

The last two matching extensions, * and **, are designed to support functions that take any number of arguments. Both can appear in either the function definition or a function call, and they have related purposes in the two locations.

Collecting arguments

The first use, in the function definition, collects unmatched *positional* arguments into a tuple:

```
>>> def f(*args): print(args)
...
```

When this function is called, Python collects all the positional arguments into a new tuple and assigns the variable `args` to that tuple. Because it is a normal tuple object, it can be indexed, stepped through with a `for` loop, and so on:

```
>>> f()
()
>>> f(1)
(1,)
>>> f(1, 2, 3, 4)
(1, 2, 3, 4)
```

The ** feature is similar, but it only works for *keyword* arguments—it collects them into a new dictionary, which can then be processed with normal dictionary tools. In a sense, the ** form allows you to convert from keywords to dictionaries, which you can then step through with `keys` calls, dictionary iterators, and the like:

```
>>> def f(**args): print(args)
...
>>> f()
{}
>>> f(a=1, b=2)
{'a': 1, 'b': 2}
```

Finally, function headers can combine normal arguments, the *, and the ** to implement wildly flexible call signatures. For instance, in the following, `1` is passed to `a` by position, `2` and `3` are collected into the `pargs` positional tuple, and `x` and `y` wind up in the `kargs` keyword dictionary:

```
>>> def f(a, *pargs, **kargs): print(a, pargs, kargs)
...
>>> f(1, 2, 3, x=1, y=2)
1 (2, 3) {'y': 2, 'x': 1}
```

In fact, these features can be combined in even more complex ways that may seem ambiguous at first glance—an idea we will revisit later in this chapter. First, though, let's see what happens when * and ** are coded in function calls instead of definitions.

Unpacking arguments

In recent Python releases, we can use the * syntax when we call a function, too. In this context, its meaning is the inverse of its meaning in the function definition—it unpacks a collection of arguments, rather than building a collection of arguments. For example, we can pass four arguments to a function in a tuple and let Python unpack them into individual arguments:

```
>>> def func(a, b, c, d): print(a, b, c, d)
...
>>> args = (1, 2)
>>> args += (3, 4)
>>> func(*args)
1 2 3 4
```

Similarly, the ** syntax in a function call unpacks a dictionary of key/value pairs into separate keyword arguments:

```
>>> args = {'a': 1, 'b': 2, 'c': 3}
>>> args['d'] = 4
>>> func(**args)
1 2 3 4
```

Again, we can combine normal, positional, and keyword arguments in the call in very flexible ways:

```
>>> func(*(1, 2), **{'d': 4, 'c': 4})
1 2 4 4

>>> func(1, *(2, 3), **{'d': 4})
1 2 3 4

>>> func(1, c=3, *(2,), **{'d': 4})
```

```
1 2 3 4

>>> func(1, *(2, 3), d=4)
1 2 3 4

>>> f(1, *(2,), c=3, **{'d':4})
1 2 3 4
```

This sort of code is convenient when you cannot predict the number of arguments that will be passed to a function when you write your script; you can build up a collection of arguments at runtime instead and call the function generically this way. Again, don't confuse the */** syntax in the function header and the function call—in the header it collects any number of arguments, while in the call it unpacks any number of arguments.

As we saw in Chapter 14, the `*pargs` form in a call is an *iteration context*, so technically it accepts any iterable object, not just tuples or other sequences as shown in the examples here. For instance, a file object works after the `*`, and unpacks its lines into individual arguments (e.g., `func(*open('fname'))`).

This generality is supported in both Python 3.0 and 2.6, but it holds true only for *calls*—a `*pargs` in a call allows any iterable, but the same form in a `def` header always bundles extra arguments into a *tuple*. This header behavior is similar in spirit and syntax to the `*` in Python 3.0 extended sequence unpacking assignment forms we met in Chapter 11 (e.g., `x, *y = z`), though that feature always creates lists, not tuples.

Applying functions generically

The prior section's examples may seem obtuse, but they are used more often than you might expect. Some programs need to call arbitrary functions in a generic fashion, without knowing their names or arguments ahead of time. In fact, the real power of the special "varargs" call syntax is that you don't need to know how many arguments a function call requires before you write a script. For example, you can use `if` logic to select from a set of functions and argument lists, and call any of them generically:

```
if <test>:
    action, args = func1, (1,)          # Call func1 with 1 arg in this case
else:
    action, args = func2, (1, 2, 3)     # Call func2 with 3 args here
...
action(*args)                           # Dispatch generically
```

More generally, this varargs call syntax is useful any time you cannot predict the arguments list. If your user selects an arbitrary function via a user interface, for instance, you may be unable to hardcode a function call when writing your script. To work around this, simply build up the arguments list with sequence operations, and call it with starred names to unpack the arguments:

```
>>> args = (2,3)
>>> args += (4,)
>>> args
(2, 3, 4)
>>> func(*args)
```

Because the arguments list is passed in as a tuple here, the program can build it at runtime. This technique also comes in handy for functions that test or time other functions. For instance, in the following code we support any function with any arguments by passing along whatever arguments were sent in:

```
def tracer(func, *pargs, **kargs):          # Accept arbitrary arguments
    print('calling:', func.__name__)
    return func(*pargs, **kargs)            # Pass along arbitrary arguments

def func(a, b, c, d):
    return a + b + c + d

print(tracer(func, 1, 2, c=3, d=4))
```

When this code is run, arguments are collected by the tracer and then *propagated* with varargs call syntax:

```
calling: func
10
```

We'll see larger examples of such roles later in this book; see especially the sequence timing example in Chapter 20 and the various decorator tools we will code in Chapter 38.

The defunct apply built-in (Python 2.6)

Prior to Python 3.0, the effect of the `*args` and `**args` varargs call syntax could be achieved with a built-in function named `apply`. This original technique has been removed in 3.0 because it is now redundant (3.0 cleans up many such dusty tools that have been subsumed over the years). It's still available in Python 2.6, though, and you may come across it in older 2.X code.

In short, the following are equivalent prior to Python 3.0:

```
func(*pargs, **kargs)           # Newer call syntax: func(*sequence, **dict)

apply(func, pargs, kargs)       # Defunct built-in:  apply(func, sequence, dict)
```

For example, consider the following function, which accepts any number of positional or keyword arguments:

```
>>> def echo(*args, **kwargs): print(args, kwargs)
...
>>> echo(1, 2, a=3, b=4)
(1, 2) {'a': 3, 'b': 4}
```

In Python 2.6, we can call it generically with `apply`, or with the call syntax that is now required in 3.0:

```
>>> pargs = (1, 2)
>>> kargs = {'a':3, 'b':4}

>>> apply(echo, pargs, kargs)
(1, 2) {'a': 3, 'b': 4}

>>> echo(*pargs, **kargs)
(1, 2) {'a': 3, 'b': 4}
```

The unpacking call syntax form is newer than the `apply` function, is preferred in general, and is required in 3.0. Apart from its symmetry with the `*pargs` and `**kargs` collector forms in `def` headers, and the fact that it requires fewer keystrokes overall, the newer call syntax also allows us to pass along additional arguments without having to manually extend argument sequences or dictionaries:

```
>>> echo(0, c=5, *pargs, **kargs)      # Normal, keyword, *sequence, **dictionary
(0, 1, 2) {'a': 3, 'c': 5, 'b': 4}
```

That is, the call syntax form is *more general*. Since it's required in 3.0, you should now disavow all knowledge of `apply` (unless, of course, it appears in 2.X code you must use or maintain...).

Python 3.0 Keyword-Only Arguments

Python 3.0 generalizes the ordering rules in function headers to allow us to specify *keyword-only arguments*—arguments that must be passed by keyword only and will never be filled in by a positional argument. This is useful if we want a function to both process any number of arguments and accept possibly optional configuration options.

Syntactically, keyword-only arguments are coded as named arguments that appear after `*args` in the arguments list. All such arguments must be passed using keyword syntax in the call. For example, in the following, `a` may be passed by name or position, `b` collects any extra positional arguments, and `c` must be passed by keyword only:

```
>>> def kwonly(a, *b, c):
...     print(a, b, c)
...
>>> kwonly(1, 2, c=3)
1 (2,) 3
>>> kwonly(a=1, c=3)
1 () 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() needs keyword-only argument c
```

We can also use a * character by itself in the arguments list to indicate that a function does not accept a variable-length argument list but still expects all arguments following the * to be passed as keywords. In the next function, `a` may be passed by position or name again, but `b` and `c` must be keywords, and no extra positionals are allowed:

```
>>> def kwonly(a, *, b, c):
...     print(a, b, c)
...
>>> kwonly(1, c=3, b=2)
1 2 3
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes exactly 1 positional argument (3 given)
>>> kwonly(1)
TypeError: kwonly() needs keyword-only argument b
```

You can still use defaults for keyword-only arguments, even though they appear after the * in the function header. In the following code, `a` may be passed by name or position, and `b` and `c` are optional but must be passed by keyword if used:

```
>>> def kwonly(a, *, b='spam', c='ham'):
...     print(a, b, c)
...
>>> kwonly(1)
1 spam ham
>>> kwonly(1, c=3)
1 spam 3
>>> kwonly(a=1)
1 spam ham
>>> kwonly(c=3, b=2, a=1)
1 2 3
>>> kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)
```

In fact, keyword-only arguments with defaults are optional, but those without defaults effectively become required keywords for the function:

```
>>> def kwonly(a, *, b, c='spam'):
...     print(a, b, c)
...
>>> kwonly(1, b='eggs')
1 eggs spam
>>> kwonly(1, c='eggs')
TypeError: kwonly() needs keyword-only argument b
>>> kwonly(1, 2)
TypeError: kwonly() takes exactly 1 positional argument (2 given)

>>> def kwonly(a, *, b=1, c, d=2):
...     print(a, b, c, d)
...
>>> kwonly(3, c=4)
3 1 4 2
>>> kwonly(3, c=4, b=5)
3 5 4 2
>>> kwonly(3)
TypeError: kwonly() needs keyword-only argument c
>>> kwonly(1, 2, 3)
TypeError: kwonly() takes exactly 1 positional argument (3 given)
```

Ordering rules

Finally, note that keyword-only arguments must be specified after a single star, not two—named arguments cannot appear after the **`**args`** arbitrary keywords form, and a ** can't appear by itself in the arguments list. Both attempts generate a syntax error:

```
>>> def kwonly(a, **pargs, b, c):
SyntaxError: invalid syntax
>>> def kwonly(a, **, b, c):
SyntaxError: invalid syntax
```

This means that in a function *header*, keyword-only arguments must be coded before the **`**args`** arbitrary keywords form and after the **`*args`** arbitrary positional form, when both are present. Whenever an argument name appears before **`*args`**, it is a possibly default positional argument, not keyword-only:

```
>>> def f(a, *b, **d, c=6): print(a, b, c, d)          # Keyword-only before **!
SyntaxError: invalid syntax

>>> def f(a, *b, c=6, **d): print(a, b, c, d)          # Collect args in header
...
>>> f(1, 2, 3, x=4, y=5)                               # Default used
1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, 2, 3, x=4, y=5, c=7)                          # Override default
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, 2, 3, c=7, x=4, y=5)                          # Anywhere in keywords
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> def f(a, c=6, *b, **d): print(a, b, c, d)          # c is not keyword-only!
...
>>> f(1, 2, 3, x=4)
1 (3,) 2 {'x': 4}
```

In fact, similar ordering rules hold true in function *calls*: when keyword-only arguments are passed, they must appear before a **`**args`** form. The keyword-only argument can be coded either before or after the **`*args`**, though, and may be included in **`**args`**:

```
>>> def f(a, *b, c=6, **d): print(a, b, c, d)          # KW-only between * and **
...
>>> f(1, *(2, 3), **dict(x=4, y=5))                    # Unpack args at call
1 (2, 3) 6 {'y': 5, 'x': 4}

>>> f(1, *(2, 3), **dict(x=4, y=5), c=7)               # Keywords before **args!
SyntaxError: invalid syntax

>>> f(1, *(2, 3), c=7, **dict(x=4, y=5))               # Override default
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, c=7, *(2, 3), **dict(x=4, y=5))               # After or before *
1 (2, 3) 7 {'y': 5, 'x': 4}

>>> f(1, *(2, 3), **dict(x=4, y=5, c=7))               # Keyword-only in **
1 (2, 3) 7 {'y': 5, 'x': 4}
```

Trace through these cases on your own, in conjunction with the general argument-ordering rules described formally earlier. They may appear to be worst cases in the artificial examples here, but they can come up in real practice, especially for people who write libraries and tools for other Python programmers to use.

Why keyword-only arguments?

So why care about keyword-only arguments? In short, they make it easier to allow a function to accept both any number of positional arguments to be processed, and configuration options passed as keywords. While their use is optional, without keyword-only arguments extra work may be required to provide defaults for such options and to verify that no superfluous keywords were passed.

Imagine a function that processes a set of passed-in objects and allows a tracing flag to be passed:

```
process(X, Y, Z)                  # use flag's default
process(X, Y, notify=True)        # override flag default
```

Without keyword-only arguments we have to use both `*args` and `**args` and manually inspect the keywords, but with keyword-only arguments less code is required. The following guarantees that no positional argument will be incorrectly matched against `notify` and requires that it be a keyword if passed:

```
def process(*args, notify=False): ...
```

Since we're going to see a more realistic example of this later in this chapter, in "Emulating the Python 3.0 print Function" on page 457, I'll postpone the rest of this story until then. For an additional example of keyword-only arguments in action, see the iteration options timing case study in Chapter 20. And for additional function definition enhancements in Python 3.0, stay tuned for the discussion of function annotation syntax in Chapter 19.

The min Wakeup Call!

Time for something more realistic. To make this chapter's concepts more concrete, let's work through an exercise that demonstrates a practical application of argument-matching tools.

Suppose you want to code a function that is able to compute the minimum value from an arbitrary set of arguments and an arbitrary set of object data types. That is, the function should accept zero or more arguments, as many as you wish to pass. Moreover, the function should work for all kinds of Python object types: numbers, strings, lists, lists of dictionaries, files, and even `None`.

The first requirement provides a natural example of how the * feature can be put to good use—we can collect arguments into a tuple and step over each of them in turn with a simple `for` loop. The second part of the problem definition is easy: because every

object type supports comparisons, we don't have to specialize the function per type (an application of polymorphism); we can simply compare objects blindly and let Python worry about what sort of comparison to perform.

Full Credit

The following file shows three ways to code this operation, at least one of which was suggested by a student in one of my courses:

- The first function fetches the first argument (`args` is a tuple) and traverses the rest by slicing off the first (there's no point in comparing an object to itself, especially if it might be a large structure).
- The second version lets Python pick off the first and rest of the arguments automatically, and so avoids an index and slice.
- The third converts from a tuple to a list with the built-in `list` call and employs the list `sort` method.

The `sort` method is coded in C, so it can be quicker than the other approaches at times, but the linear scans of the first two techniques will make them faster most of the time.* The file *mins.py* contains the code for all three solutions:

```
def min1(*args):
    res = args[0]
    for arg in args[1:]:
        if arg < res:
            res = arg
    return res

def min2(first, *rest):
    for arg in rest:
        if arg < first:
            first = arg
    return first

def min3(*args):
    tmp = list(args)              # Or, in Python 2.4+: return sorted(args)[0]
    tmp.sort()
    return tmp[0]

print(min1(3,4,1,2))
```

* Actually, this is fairly complicated. The Python `sort` routine is coded in C and uses a highly optimized algorithm that attempts to take advantage of partial ordering in the items to be sorted. It's named "timsort" after Tim Peters, its creator, and in its documentation it claims to have "supernatural performance" at times (pretty good, for a sort!). Still, sorting is an inherently exponential operation (it must chop up the sequence and put it back together many times), and the other versions simply perform one linear left-to-right scan. The net effect is that sorting is quicker if the arguments are partially ordered, but is likely to be slower otherwise. Even so, Python performance can change over time, and the fact that sorting is implemented in the C language can help greatly; for an exact analysis, you should time the alternatives with the `time` or `timeit` modules we'll meet in Chapter 20.

```
print(min2("bb", "aa"))
print(min3([2,2], [1,1], [3,3]))
```

All three solutions produce the same result when the file is run. Try typing a few calls interactively to experiment with these on your own:

```
% python mins.py
1
aa
[1, 1]
```

Notice that none of these three variants tests for the case where no arguments are passed in. They could, but there's no point in doing so here—in all three solutions, Python will automatically raise an exception if no arguments are passed in. The first variant raises an exception when we try to fetch item 0, the second when Python detects an argument list mismatch, and the third when we try to return item 0 at the end.

This is exactly what we want to happen—because these functions support any data type, there is no valid sentinel value that we could pass back to designate an error. There are exceptions to this rule (e.g., if you have to run expensive actions before you reach the error), but in general it's better to assume that arguments will work in your functions' code and let Python raise errors for you when they do not.

Bonus Points

You can get can get bonus points here for changing these functions to compute the *maximum*, rather than minimum, values. This one's easy: the first two versions only require changing `<` to `>`, and the third simply requires that we return `tmp[-1]` instead of `tmp[0]`. For an extra point, be sure to set the function name to "max" as well (though this part is strictly optional).

It's also possible to generalize a single function to compute either a minimum *or* a maximum value, by evaluating comparison expression strings with a tool like the `eval` built-in function (see the library manual) or passing in an arbitrary comparison function. The file *minmax.py* shows how to implement the latter scheme:

```
def minmax(test, *args):
    res = args[0]
    for arg in args[1:]:
        if test(arg, res):
            res = arg
    return res

def lessthan(x, y): return x < y                # See also: lambda
def grtrthan(x, y): return x > y

print(minmax(lessthan, 4, 2, 1, 5, 6, 3))       # Self-test code
print(minmax(grtrthan, 4, 2, 1, 5, 6, 3))

% python minmax.py
```

```
1
6
```

Functions are another kind of object that can be passed into a function like this one. To make this a `max` (or other) function, for example, we could simply pass in the right sort of `test` function. This may seem like extra work, but the main point of generalizing functions this way (instead of cutting and pasting to change just a single character) is that we'll only have one version to change in the future, not two.

The Punch Line...

Of course, all this was just a coding exercise. There's really no reason to code `min` or `max` functions, because both are built-ins in Python! We met them briefly in Chapter 5 in conjunction with numeric tools, and again in Chapter 14 when exploring iteration contexts. The built-in versions work almost exactly like ours, but they're coded in C for optimal speed and accept either a single iterable or multiple arguments. Still, though it's superfluous in this context, the general coding pattern we used here might be useful in other scenarios.

Generalized Set Functions

Let's look at a more useful example of special argument-matching modes at work. At the end of Chapter 16, we wrote a function that returned the intersection of two sequences (it picked out items that appeared in both). Here is a version that intersects an arbitrary number of sequences (one or more) by using the varargs matching form `*args` to collect all the passed-in arguments. Because the arguments come in as a tuple, we can process them in a simple `for` loop. Just for fun, we'll code a `union` function that also accepts an arbitrary number of arguments to collect items that appear in any of the operands:

```
def intersect(*args):
    res = []
    for x in args[0]:                   # Scan first sequence
        for other in args[1:]:          # For all other args
            if x not in other: break    # Item in each one?
        else:                           # No: break out of loop
            res.append(x)               # Yes: add items to end
    return res

def union(*args):
    res = []
    for seq in args:                    # For all args
        for x in seq:                   # For all nodes
            if not x in res:
                res.append(x)           # Add new items to result
    return res
```

Because these are tools worth reusing (and they're too big to retype interactively), we'll store the functions in a module file called *inter2.py* (if you've forgotten how modules and imports work, see the introduction in Chapter 3, or stay tuned for in-depth coverage in Part V). In both functions, the arguments passed in at the call come in as the `args` tuple. As in the original `intersect`, both work on any kind of sequence. Here, they are processing strings, mixed types, and more than two sequences:

```
% python
>>> from inter2 import intersect, union
>>> s1, s2, s3 = "SPAM", "SCAM", "SLAM"

>>> intersect(s1, s2), union(s1, s2)          # Two operands
(['S', 'A', 'M'], ['S', 'P', 'A', 'M', 'C'])

>>> intersect([1,2,3], (1,4))                 # Mixed types
[1]

>>> intersect(s1, s2, s3)                     # Three operands
['S', 'A', 'M']

>>> union(s1, s2, s3)
['S', 'P', 'A', 'M', 'C', 'L']
```

I should note that because Python now has a *set object type* (described in Chapter 5), none of the set-processing examples in this book are strictly required anymore; they are included only as demonstrations of coding techniques. Because it's constantly improving, Python has an uncanny way of conspiring to make my book examples obsolete over time!

Emulating the Python 3.0 print Function

To round out the chapter, let's look at one last example of argument matching at work. The code you'll see here is intended for use in Python 2.6 or earlier (it works in 3.0, too, but is pointless there): it uses both the `*args` arbitrary positional tuple and the `**args` arbitrary keyword-arguments dictionary to simulate most of what the Python 3.0 `print` function does.

As we learned in Chapter 11, this isn't actually required, because 2.6 programmers can always enable the 3.0 `print` function with an import of this form:

```
from __future__ import print_function
```

To demonstrate argument matching in general, though, the following file, *print30.py*, does the same job in a small amount of reusable code:

```
"""
Emulate most of the 3.0 print function for use in 2.X
call signature: print30(*args, sep=' ', end='\n', file=None)
"""
import sys

def print30(*args, **kargs):
    sep  = kargs.get('sep', ' ')            # Keyword arg defaults
    end  = kargs.get('end', '\n')
    file = kargs.get('file', sys.stdout)
    output = ''
    first  = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

To test it, import this into another file or the interactive prompt, and use it like the 3.0 `print` function. Here is a test script, *testprint30.py* (notice that the function must be called "print30", because "print" is a reserved word in 2.6):

```
from print30 import print30
print30(1, 2, 3)
print30(1, 2, 3, sep='')                    # Suppress separator
print30(1, 2, 3, sep='...')
print30(1, [2], (3,), sep='...')            # Various object types

print30(4, 5, 6, sep='', end='')            # Suppress newline
print30(7, 8, 9)
print30()                                   # Add newline (or blank line)

import sys
print30(1, 2, 3, sep='??', end='.\n', file=sys.stderr)    # Redirect to file
```

When run under 2.6, we get the same results as 3.0's `print` function:

```
C:\misc> c:\python26\python testprint30.py
1 2 3
123
1...2...3
1...[2]...(3,)
4567 8 9

1??2??3.
```

Although pointless in 3.0, the results are the same when run there. As usual, the generality of Python's design allows us to prototype or develop concepts in the Python language itself. In this case, argument-matching tools are as flexible in Python code as they are in Python's internal implementation.

Using Keyword-Only Arguments

It's interesting to notice that this example could be coded with Python 3.0 keyword-only arguments, described earlier in this chapter, to automatically validate configuration arguments:

```
# Use keyword-only args

def print30(*args, sep=' ', end='\n', file=sys.stdout):
    output = ''
    first  = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

This version works the same as the original, and it's a prime example of how keyword-only arguments come in handy. The original version assumes that all positional arguments are to be printed, and all keywords are for options only. That's almost sufficient, but any extra keyword arguments are silently ignored. A call like the following, for instance, will generate an exception with the keyword-only form:

```
>>> print30(99, name='bob')
TypeError: print30() got an unexpected keyword argument 'name'
```

but will silently ignore the `name` argument in the original version. To detect superfluous keywords manually, we could use `dict.pop()` to delete fetched entries, and check if the dictionary is not empty. Here is an equivalent to the keyword-only version:

```
# Use keyword args deletion with defaults

def print30(*args, **kargs):
    sep  = kargs.pop('sep', ' ')
    end  = kargs.pop('end', '\n')
    file = kargs.pop('file', sys.stdout)
    if kargs: raise TypeError('extra keywords: %s' % kargs)
    output = ''
    first  = True
    for arg in args:
        output += ('' if first else sep) + str(arg)
        first = False
    file.write(output + end)
```

This works as before, but it now catches extraneous keyword arguments, too:

```
>>> print30(99, name='bob')
TypeError: extra keywords: {'name': 'bob'}
```

This version of the function runs under Python 2.6, but it requires four more lines of code than the keyword-only version. Unfortunately, the extra code is required in this case—the keyword-only version only works on 3.0, which negates most of the reason that I wrote this example in the first place (a 3.0 emulator that only works on 3.0 isn't incredibly useful!). In programs written to run on 3.0, though, keyword-only arguments can simplify a specific category of functions that accept both arguments and options. For another example of 3.0 keyword-only arguments, be sure to see the upcoming iteration timing case study in Chapter 20.

Why You Will Care: Keyword Arguments

As you can probably tell, advanced argument-matching modes can be complex. They are also entirely optional; you can get by with just simple positional matching, and it's probably a good idea to do so when you're starting out. However, because some Python tools make use of them, some general knowledge of these modes is important.

For example, keyword arguments play an important role in `tkinter`, the de facto standard GUI API for Python (this module's name is `Tkinter` in Python 2.6). We touch on `tkinter` only briefly at various points in this book, but in terms of its call patterns, keyword arguments set configuration options when GUI components are built. For instance, a call of the form:

```
from tkinter import *
widget = Button(text="Press me", command=someFunction)
```

creates a new button and specifies its text and callback function, using the `text` and `command` keyword arguments. Since the number of configuration options for a widget can be large, keyword arguments let you pick and choose which to apply. Without them, you might have to either list all the possible options by position or hope for a judicious positional argument defaults protocol that would handle every possible option arrangement.

Many built-in functions in Python expect us to use keywords for usage-mode options as well, which may or may not have defaults. As we learned in Chapter 8, for instance, the `sorted` built-in:

```
sorted(iterable, key=None, reverse=False)
```

expects us to pass an iterable object to be sorted, but also allows us to pass in optional keyword arguments to specify a dictionary sort key and a reversal flag, which default to `None` and `False`, respectively. Since we normally don't use these options, they may be omitted to use defaults.

Chapter Summary

In this chapter, we studied the second of two key concepts related to functions: *arguments* (how objects are passed into a function). As we learned, arguments are passed into a function by assignment, which means by object reference, which really means

by pointer. We also studied some more advanced extensions, including default and keyword arguments, tools for using arbitrarily many arguments, and keyword-only arguments in 3.0. Finally, we saw how mutable arguments can exhibit the same behavior as other shared references to objects—unless the object is explicitly copied when it's sent in, changing a passed-in mutable in a function can impact the caller.

The next chapter continues our look at functions by exploring some more advanced function-related ideas: function annotations, `lambda`s, and functional tools such as `map` and `filter`. Many of these concepts stem from the fact that functions are normal objects in Python, and so support some advanced and very flexible processing modes. Before diving into those topics, however, take this chapter's quiz to review the argument ideas we've studied here.

Test Your Knowledge: Quiz

1. What is the output of the following code, and why?

```
>>> def func(a, b=4, c=5):
...     print(a, b, c)
...
>>> func(1, 2)
```

2. What is the output of this code, and why?

```
>>> def func(a, b, c=5):
...     print(a, b, c)
...
>>> func(1, c=3, b=2)
```

3. How about this code: what is its output, and why?

```
>>> def func(a, *pargs):
...     print(a, pargs)
...
>>> func(1, 2, 3)
```

4. What does this code print, and why?

```
>>> def func(a, **kargs):
...     print(a, kargs)
...
>>> func(a=1, c=3, b=2)
```

5. One last time: what is the output of this code, and why?

```
>>> def func(a, b, c=3, d=4): print(a, b, c, d)
...
>>> func(1, *(5,6))
```

6. Name three or more ways that functions can communicate results to a caller.

Test Your Knowledge: Answers

1. The output here is `'1 2 5'`, because `1` and `2` are passed to `a` and `b` by position, and `c` is omitted in the call and defaults to `5`.
2. The output this time is `'1 2 3'`: `1` is passed to `a` by position, and `b` and `c` are passed `2` and `3` by name (the left-to-right order doesn't matter when keyword arguments are used like this).
3. This code prints `'1 (2, 3)'`, because `1` is passed to `a` and the `*pargs` collects the remaining positional arguments into a new tuple object. We can step through the extra positional arguments tuple with any iteration tool (e.g., `for arg in pargs: ...`).
4. This time the code prints `'1, {'c': 3, 'b': 2}'`, because `1` is passed to `a` by name and the `**kargs` collects the remaining keyword arguments into a dictionary. We could step through the extra keyword arguments dictionary by key with any iteration tool (e.g., `for key in kargs: ...`).
5. The output here is `'1 5 6 4'`: `1` matches `a` by position, `5` and `6` match `b` and `c` by `*name` positionals (`6` overrides `c`'s default), and `d` defaults to `4` because it was not passed a value.
6. Functions can send back results with `return` statements, by changing passed-in mutable arguments, and by setting global variables. Globals are generally frowned upon (except for very special cases, like multithreaded programs) because they can make code more difficult to understand and use. `return` statements are usually best, but changing mutables is fine, if expected. Functions may also communicate with system devices such as files and sockets, but these are beyond our scope here.

CHAPTER 19

Advanced Function Topics

This chapter introduces a collection of more advanced function-related topics: recursive functions, function attributes and annotations, the `lambda` expression, and functional programming tools such as `map` and `filter`. These are all somewhat advanced tools that, depending on your job description, you may not encounter on a regular basis. Because of their roles in some domains, though, a basic understanding can be useful; `lambda`s, for instance, are regular customers in GUIs.

Part of the art of using functions lies in the interfaces between them, so we will also explore some general function design principles here. The next chapter continues this advanced theme with an exploration of generator functions and expressions and a revival of list comprehensions in the context of the functional tools we will study here.

Function Design Concepts

Now that we've had a chance to study function basics in Python, let's begin this chapter with a few words of context. When you start using functions in earnest, you're faced with choices about how to glue components together—for instance, how to decompose a task into purposeful functions (known as *cohesion*), how your functions should communicate (called *coupling*), and so on. You also need to take into account concepts such as the size of your functions, because they directly impact code usability. Some of this falls into the category of structured analysis and design, but it applies to Python code as to any other.

We introduced some ideas related to function and module coupling in the Chapter 17 when studying scopes, but here is a review of a few general guidelines for function beginners:

- **Coupling: use arguments for inputs and `return` for outputs**. Generally, you should strive to make a function independent of things outside of it. Arguments and `return` statements are often the best ways to isolate external dependencies to a small number of well-known places in your code.

- **Coupling: use global variables only when truly necessary**. Global variables (i.e., names in the enclosing module) are usually a poor way for functions to communicate. They can create dependencies and timing issues that make programs difficult to debug and change.
- **Coupling: don't change mutable arguments unless the caller expects it**. Functions can change parts of passed-in mutable objects, but (as with global variables) this creates lots of coupling between the caller and callee, which can make a function too specific and brittle.
- **Cohesion: each function should have a single, unified purpose**. When designed well, each of your functions should do one thing—something you can summarize in a simple declarative sentence. If that sentence is very broad (e.g., "this function implements my whole program"), or contains lots of conjunctions (e.g., "this function gives employee raises *and* submits a pizza order"), you might want to think about splitting it into separate and simpler functions. Otherwise, there is no way to reuse the code behind the steps mixed together in the function.
- **Size: each function should be relatively small**. This naturally follows from the preceding goal, but if your functions start spanning multiple pages on your display, it's probably time to split them. Especially given that Python code is so concise to begin with, a long or deeply nested function is often a symptom of design problems. Keep it simple, and keep it short.
- **Coupling: avoid changing variables in another module file directly**. We introduced this concept in Chapter 17, and we'll revisit it in the next part of the book when we focus on modules. For reference, though, remember that changing variables across file boundaries sets up a coupling between modules similar to how global variables couple functions—the modules become difficult to understand and reuse. Use accessor functions whenever possible, instead of direct assignment statements.

Figure 19-1 summarizes the ways functions can talk to the outside world; inputs may come from items on the left side, and results may be sent out in any of the forms on the right. Good function designers prefer to use only arguments for inputs and `return` statements for outputs, whenever possible.

Of course, there are plenty of exceptions to the preceding design rules, including some related to Python's OOP support. As you'll see in Part VI, Python classes *depend* on changing a passed-in mutable object—class functions set attributes of an automatically passed-in argument called `self` to change per-object state information (e.g., `self.name='bob'`). Moreover, if classes are not used, global variables are often the most straightforward way for functions in modules to retain state between calls. Side effects are dangerous only if they're unexpected.

In general though, you should strive to minimize external dependencies in functions and other program components. The more *self-contained* a function is, the easier it will be to understand, reuse, and modify.

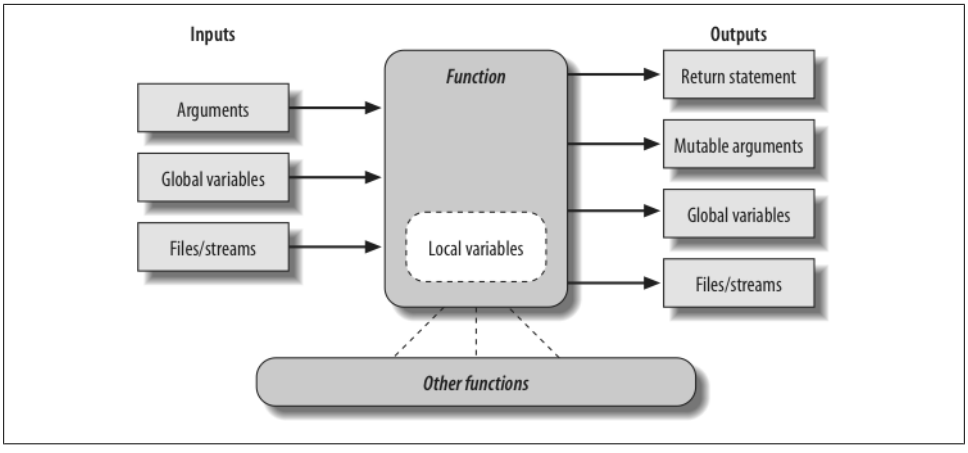

Figure 19-1. Function execution environment. Functions may obtain input and produce output in a variety of ways, though functions are usually easier to understand and maintain if you use arguments for input and return statements and anticipated mutable argument changes for output. In Python 3, outputs may also take the form of declared nonlocal names that exist in an enclosing function scope.

Recursive Functions

While discussing scope rules near the start of Chapter 17, we briefly noted that Python supports *recursive functions*—functions that call themselves either directly or indirectly in order to loop. Recursion is a somewhat advanced topic, and it's relatively rare to see in Python. Still, it's a useful technique to know about, as it allows programs to traverse structures that have arbitrary and unpredictable shapes. Recursion is even an alternative for simple loops and iterations, though not necessarily the simplest or most efficient one.

Summation with Recursion

Let's look at some examples. To sum a list (or other sequence) of numbers, we can either use the built-in `sum` function or write a more custom version of our own. Here's what a custom summing function might look like when coded with recursion:

```
>>> def mysum(L):
...     if not L:
...         return 0
...     else:
...         return L[0] + mysum(L[1:])          # Call myself

>>> mysum([1, 2, 3, 4, 5])
15
```

At each level, this function calls itself recursively to compute the sum of the rest of the list, which is later added to the item at the front. The recursive loop ends and zero is returned when the list becomes empty. When using recursion like this, each open level

of call to the function has its own copy of the function's local scope on the runtime call stack—here, that means `L` is different in each level.

If this is difficult to understand (and it often is for new programmers), try adding a `print` of `L` to the function and run it again, to trace the current list at each call level:

```
>>> def mysum(L):
...     print(L)                                # Trace recursive levels
...     if not L:                               # L shorter at each level
...         return 0
...     else:
...         return L[0] + mysum(L[1:])
...
>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

As you can see, the list to be summed grows smaller at each recursive level, until it becomes empty—the termination of the recursive loop. The sum is computed as the recursive calls unwind.

Coding Alternatives

Interestingly, we can also use Python's `if`/`else` ternary expression (described in Chapter 12) to save some code real-estate here. We can also generalize for any summable type (which is easier if we assume at least one item in the input, as we did in Chapter 18's minimum value example) and use Python 3.0's extended sequence assignment to make the first/rest unpacking simpler (as covered in Chapter 11):

```
def mysum(L):
    return 0 if not L else L[0] + mysum(L[1:])           # Use ternary expression

def mysum(L):
    return L[0] if len(L) == 1 else L[0] + mysum(L[1:])  # Any type, assume one

def mysum(L):
    first, *rest = L
    return first if not rest else first + mysum(rest)    # Use 3.0 ext seq assign
```

The latter two of these fail for empty lists but allow for sequences of any object type that supports `+`, not just numbers:

```
>>> mysum([1])                          # mysum([]) fails in last 2
1
>>> mysum([1, 2, 3, 4, 5])
15
>>> mysum(('s', 'p', 'a', 'm'))         # But various types now work
'spam'
```

```
>>> mysum(['spam', 'ham', 'eggs'])
'spamhameggs'
```

If you study these three variants, you'll find that the latter two also work on a single string argument (e.g., `mysum ('spam')`), because strings are sequences of one-character strings; the third variant works on arbitary iterables, including open input files, but the others do not because they index; and the function header `def mysum(first, * rest)`, although similar to the third variant, wouldn't work at all, because it expects individual arguments, not a single iterable.

Keep in mind that recursion can be direct, as in the examples so far, or *indirect*, as in the following (a function that calls another function, which calls back to its caller). The net effect is the same, though there are two function calls at each level instead of one:

```
>>> def mysum(L):
...     if not L: return 0
...     return nonempty(L)                  # Call a function that calls me
...
>>> def nonempty(L):
...     return L[0] + mysum(L[1:])          # Indirectly recursive
...
>>> mysum([1.1, 2.2, 3.3, 4.4])
11.0
```

Loop Statements Versus Recursion

Though recursion works for summing in the prior sections' examples, it's probably overkill in this context. In fact, recursion is not used nearly as often in Python as in more esoteric languages like Prolog or Lisp, because Python emphasizes simpler procedural statements like loops, which are usually more natural. The `while`, for example, often makes things a bit more concrete, and it doesn't require that a function be defined to allow recursive calls:

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> while L:
...     sum += L[0]
...     L = L[1:]
...
>>> sum
15
```

Better yet, `for` loops iterate for us automatically, making recursion largely extraneous in most cases (and, in all likelihood, less efficient in terms of memory space and execution time):

```
>>> L = [1, 2, 3, 4, 5]
>>> sum = 0
>>> for x in L: sum += x
...
>>> sum
15
```

With looping statements, we don't require a fresh copy of a local scope on the call stack for each iteration, and we avoid the speed costs associated with function calls in general. (Stay tuned for Chapter 20's timer case study for ways to compare the execution times of alternatives like these.)

Handling Arbitrary Structures

On the other hand, recursion (or equivalent explicit stack-based algorithms, which we'll finesse here) can be required to traverse arbitrarily shaped structures. As a simple example of recursion's role in this context, consider the task of computing the sum of all the numbers in a nested sublists structure like this:

```
[1, [2, [3, 4], 5], 6, [7, 8]]          # Arbitrarily nested sublists
```

Simple looping statements won't work here because this not a linear iteration. Nested looping statements do not suffice either, because the sublists may be nested to arbitrary depth and in an arbitrary shape. Instead, the following code accommodates such general nesting by using recursion to visit sublists along the way:

```
def sumtree(L):
    tot = 0
    for x in L:                          # For each item at this level
        if not isinstance(x, list):
            tot += x                     # Add numbers directly
        else:
            tot += sumtree(x)            # Recur for sublists
    return tot

L = [1, [2, [3, 4], 5], 6, [7, 8]]       # Arbitrary nesting
print(sumtree(L))                        # Prints 36


# Pathological cases

print(sumtree([1, [2, [3, [4, [5]]]]]))  # Prints 15 (right-heavy)

print(sumtree([[[[[1], 2], 3], 4], 5]))  # Prints 15 (left-heavy)
```

Trace through the test cases at the bottom of this script to see how recursion traverses their nested lists. Although this example is artificial, it is representative of a larger class of programs; inheritance trees and module import chains, for example, can exhibit similarly general structures. In fact, we will use recursion again in such roles in more realistic examples later in this book:

- In Chapter 24's *reloadall.py*, to traverse import chains
- In Chapter 28's *classtree.py*, to traverse class inheritance trees
- In Chapter 30's *lister.py*, to traverse class inheritance trees again

Although you should generally prefer looping statements to recursion for linear iterations on the grounds of simplicity and efficiency, we'll find that recursion is essential in scenarios like those in these later examples.

Moreover, you sometimes need to be aware of the potential of *unintended* recursion in your programs. As you'll also see later in the book, some operator overloading methods in classes such as `__setattr__` and `__getattribute__` have the potential to recursively loop if used incorrectly. Recursion is a powerful tool, but it tends to be best when expected!

Function Objects: Attributes and Annotations

Python functions are more flexible than you might think. As we've seen in this part of the book, functions in Python are much more than code-generation specifications for a compiler—Python functions are full-blown *objects*, stored in pieces of memory all their own. As such, they can be freely passed around a program and called indirectly. They also support operations that have little to do with calls at all—attribute storage and annotation.

Indirect Function Calls

Because Python functions are objects, you can write programs that process them generically. Function objects may be assigned to other names, passed to other functions, embedded in data structures, returned from one function to another, and more, as if they were simple numbers or strings. Function objects also happen to support a special operation: they can be called by listing arguments in parentheses after a function expression. Still, functions belong to the same general category as other objects.

We've seen some of these generic use cases for functions in earlier examples, but a quick review helps to underscore the object model. For example, there's really nothing special about the name used in a `def` statement: it's just a variable assigned in the current scope, as if it had appeared on the left of an `=` sign. After a `def` runs, the function name is simply a reference to an object—you can *reassign* that object to other names freely and call it through any reference:

```
>>> def echo(message):                  # Name echo assigned to function object
...     print(message)
...
>>> echo('Direct call')                 # Call object through original name
Direct call

>>> x = echo                            # Now x references the function too
>>> x('Indirect call!')                 # Call object through name by adding ()
Indirect call!
```

Because arguments are passed by assigning objects, it's just as easy to *pass* functions to other functions as arguments. The callee may then call the passed-in function just by adding arguments in parentheses:

```
>>> def indirect(func, arg):
...     func(arg)                          # Call the passed-in object by adding ()
...
>>> indirect(echo, 'Argument call!')       # Pass the function to another function
Argument call!
```

You can even stuff function objects into data structures, as though they were integers or strings. The following, for example, *embeds* the function twice in a list of tuples, as a sort of actions table. Because Python compound types like these can contain any sort of object, there's no special case here, either:

```
>>> schedule = [ (echo, 'Spam!'), (echo, 'Ham!') ]
>>> for (func, arg) in schedule:
...     func(arg)                          # Call functions embedded in containers
...
Spam!
Ham!
```

This code simply steps through the `schedule` list, calling the `echo` function with one argument each time through (notice the tuple-unpacking assignment in the `for` loop header, introduced in Chapter 13). As we saw in Chapter 17's examples, functions can also be created and *returned* for use elsewhere:

```
>>> def make(label):                      # Make a function but don't call it
...     def echo(message):
...         print(label + ':' + message)
...     return echo
...
>>> F = make('Spam')                      # Label in enclosing scope is retained
>>> F('Ham!')                             # Call the function that make returned
Spam:Ham!
>>> F('Eggs!')
Spam:Eggs!
```

Python's universal object model and lack of type declarations make for an incredibly flexible programming language.

Function Introspection

Because they are objects, we can also process functions with normal object tools. In fact, functions are more flexible than you might expect. For instance, once we make a function, we can call it as usual:

```
>>> def func(a):
...     b = 'spam'
...     return b * a
...
>>> func(8)
'spamspamspamspamspamspamspamspam'
```

But the call expression is just one operation defined to work on function objects. We can also inspect their attributes generically (the following is run in Python 3.0, but 2.6 results are similar):

```
>>> func.__name__
'func'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...more omitted...
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

Introspection tools allow us to explore implementation details too—functions have attached *code objects*, for example, which provide details on aspects such as the functions' local variables and arguments:

```
>>> func.__code__
<code object func at 0x0257C9B0, file "<stdin>", line 1>

>>> dir(func.__code__)
['__class__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
...more omitted...
'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount', 'co_lnotab',
'co_name', 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']

>>> func.__code__.co_varnames
('a', 'b')
>>> func.__code__.co_argcount
1
```

Tool writers can make use of such information to manage functions (in fact, we will too in Chapter 38, to implement validation of function arguments in decorators).

Function Attributes

Function objects are not limited to the system-defined attributes listed in the prior section, though. As we learned in Chapter 17, it's possible to attach arbitrary user-defined attributes to them as well:

```
>>> func
<function func at 0x0257C738>
>>> func.count = 0
>>> func.count += 1
>>> func.count
1
>>> func.handles = 'Button-Press'
>>> func.handles
'Button-Press'
>>> dir(func)
['__annotations__', '__call__', '__class__', '__closure__', '__code__',
...more omitted...
__str__', '__subclasshook__', 'count', 'handles']
```

As we saw in that chapter, such attributes can be used to attach *state information* to function objects directly, instead of using other techniques such as globals, nonlocals, and classes. Unlike nonlocals, such attributes are accessible anywhere the function itself is. In a sense, this is also a way to emulate "static locals" in other languages—variables whose names are local to a function, but whose values are retained after a function exits. Attributes are related to objects instead of scopes, but the net effect is similar.

Function Annotations in 3.0

In Python 3.0 (but not 2.6), it's also possible to attach *annotation information*—arbitrary user-defined data about a function's arguments and result—to a function object. Python provides special syntax for specifying annotations, but it doesn't do anything with them itself; annotations are completely optional, and when present are simply attached to the function object's `__annotations__` attribute for use by other tools.

We met Python 3.0's keyword-only arguments in the prior chapter; annotations generalize function header syntax further. Consider the following nonannotated function, which is coded with three arguments and returns a result:

```
>>> def func(a, b, c):
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Syntactically, function annotations are coded in `def` header lines, as arbitrary expressions associated with arguments and return values. For arguments, they appear after a colon immediately following the argument's name; for return values, they are written after a `->` following the arguments list. This code, for example, annotates all three of the prior function's arguments, as well as its return value:

```
>>> def func(a: 'spam', b: (1, 10), c: float) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
```

Calls to an annotated function work as usual, but when annotations are present Python collects them in a *dictionary* and attaches it to the function object itself. Argument names become keys, the return value annotation is stored under key "return" if coded, and the values of annotation keys are assigned to the results of the annotation expressions:

```
>>> func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Because they are just Python objects attached to a Python object, annotations are straightforward to process. The following annotates just two of three arguments and steps through the attached annotations generically:

```
>>> def func(a: 'spam', b, c: 99):
...     return a + b + c
...
>>> func(1, 2, 3)
6
>>> func.__annotations__
{'a': 'spam', 'c': 99}

>>> for arg in func.__annotations__:
...    print(arg, '=>', func.__annotations__[arg])
...
a => spam
c => 99
```

There are two fine points to note here. First, you can still use *defaults* for arguments if you code annotations—the annotation (and its `:` character) appear before the default (and its `=` character). In the following, for example, `a: 'spam' = 4` means that argument `a` defaults to `4` and is annotated with the string `'spam'`:

```
>>> def func(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6) -> int:
...     return a + b + c
...
>>> func(1, 2, 3)
6
>>> func()                          # 4 + 5 + 6   (all defaults)
15
>>> func(1, c=10)                   # 1 + 5 + 10  (keywords work normally)
16
>>> func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Second, note that the *blank spaces* in the prior example are all optional—you can use spaces between components in function headers or not, but omitting them might degrade your code's readability to some observers:

```
>>> def func(a:'spam'=4, b:(1,10)=5, c:float=6)->int:
...     return a + b + c
...
>>> func(1, 2)                      # 1 + 2 + 6
9
>>> func.__annotations__
{'a': 'spam', 'c': <class 'float'>, 'b': (1, 10), 'return': <class 'int'>}
```

Annotations are a new feature in 3.0, and some of their potential uses remain to be uncovered. It's easy to imagine annotations being used to specify constraints for argument types or values, though, and larger APIs might use this feature as a way to register function interface information. In fact, we'll see a potential application in Chapter 38, where we'll look at annotations as an alternative to *function decorator arguments* (a more general concept in which information is coded outside the function header and so is not limited to a single role). Like Python itself, annotation is a tool whose roles are shaped by your imagination.

Finally, note that annotations work only in `def` statements, not `lambda` expressions, because `lambda`'s syntax already limits the utility of the functions it defines. Coincidentally, this brings us to our next topic.

Anonymous Functions: lambda

Besides the `def` statement, Python also provides an expression form that generates function objects. Because of its similarity to a tool in the Lisp language, it's called `lambda`.* Like `def`, this expression creates a function to be called later, but it returns the function instead of assigning it to a name. This is why `lambda`s are sometimes known as *anonymous* (i.e., unnamed) functions. In practice, they are often used as a way to inline a function definition, or to defer execution of a piece of code.

lambda Basics

The `lambda`'s general form is the keyword `lambda`, followed by one or more arguments (exactly like the arguments list you enclose in parentheses in a `def` header), followed by an expression after a colon:

```
lambda argument1, argument2,... argumentN :expression using arguments
```

Function objects returned by running `lambda` expressions work exactly the same as those created and assigned by `def`s, but there are a few differences that make `lambda`s useful in specialized roles:

- **`lambda` is an expression, not a statement**. Because of this, a `lambda` can appear in places a `def` is not allowed by Python's syntax—inside a list literal or a function call's arguments, for example. As an expression, `lambda` returns a value (a new function) that can optionally be assigned a name. In contrast, the `def` statement always assigns the new function to the name in the header, instead of returning it as a result.
- **`lambda`'s body is a single expression, not a block of statements**. The `lambda`'s body is similar to what you'd put in a `def` body's `return` statement; you simply type the result as a naked expression, instead of explicitly returning it. Because it is limited to an expression, a `lambda` is less general than a `def`—you can only squeeze so much logic into a `lambda` body without using statements such as `if`. This is by design, to limit program nesting: `lambda` is designed for coding simple functions, and `def` handles larger tasks.

* The `lambda` tends to intimidate people more than it should. This reaction seems to stem from the name "lambda" itself—a name that comes from the Lisp language, which got it from lambda calculus, which is a form of symbolic logic. In Python, though, it's really just a keyword that introduces the expression syntactically. Obscure mathematical heritage aside, `lambda` is simpler to use than you may think.

Apart from those distinctions, defs and lambdas do the same sort of work. For instance, we've seen how to make a function with a def statement:

```
>>> def func(x, y, z): return x + y + z
...
>>> func(2, 3, 4)
9
```

But you can achieve the same effect with a lambda expression by explicitly assigning its result to a name through which you can later call the function:

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Here, f is assigned the function object the lambda expression creates; this is how def works, too, but its assignment is automatic.

Defaults work on lambda arguments, just like in a def:

```
>>> x = (lambda a="fee", b="fie", c="foe": a + b + c)
>>> x("wee")
'weefiefoe'
```

The code in a lambda body also follows the same scope lookup rules as code inside a def. lambda expressions introduce a local scope much like a nested def, which automatically sees names in enclosing functions, the module, and the built-in scope (via the LEGB rule):

```
>>> def knights():
...     title = 'Sir'
...     action = (lambda x: title + ' ' + x)        # Title in enclosing def
...     return action                               # Return a function
...
>>> act = knights()
>>> act('robin')
'Sir robin'
```

In this example, prior to Release 2.2, the value for the name title would typically have been passed in as a default argument value instead; flip back to the scopes coverage in Chapter 17 if you've forgotten why.

Why Use lambda?

Generally speaking, lambdas come in handy as a sort of function shorthand that allows you to embed a function's definition within the code that uses it. They are entirely optional (you can always use defs instead), but they tend to be simpler coding constructs in scenarios where you just need to embed small bits of executable code.

For instance, we'll see later that callback handlers are frequently coded as inline lambda expressions embedded directly in a registration call's arguments list, instead of being defined with a def elsewhere in a file and referenced by name (see the sidebar "Why You Will Care: Callbacks" on page 479 for an example).

lambdas are also commonly used to code *jump tables*, which are lists or dictionaries of actions to be performed on demand. For example:

```
L = [lambda x: x ** 2,          # Inline function definition
     lambda x: x ** 3,
     lambda x: x ** 4]          # A list of 3 callable functions

for f in L:
    print(f(2))                 # Prints 4, 8, 16

print(L[0](3))                  # Prints 9
```

The lambda expression is most useful as a shorthand for def, when you need to stuff small pieces of executable code into places where statements are illegal syntactically. This code snippet, for example, builds up a list of three functions by embedding lambda expressions inside a list literal; a def won't work inside a list literal like this because it is a statement, not an expression. The equivalent def coding would require temporary function names and function definitions outside the context of intended use:

```
def f1(x): return x ** 2
def f2(x): return x ** 3        # Define named functions
def f3(x): return x ** 4

L = [f1, f2, f3]                # Reference by name

for f in L:
    print(f(2))                 # Prints 4, 8, 16

print(L[0](3))                  # Prints 9
```

In fact, you can do the same sort of thing with dictionaries and other data structures in Python to build up more general sorts of action tables. Here's another example to illustrate, at the interactive prompt:

```
>>> key = 'got'
>>> {'already': (lambda: 2 + 2),
...  'got':     (lambda: 2 * 4),
...  'one':     (lambda: 2 ** 6)}[key]()
8
```

Here, when Python makes the temporary dictionary, each of the nested lambdas generates and leaves behind a function to be called later. Indexing by key fetches one of those functions, and parentheses force the fetched function to be called. When coded this way, a dictionary becomes a more general multiway branching tool than what I could show you in Chapter 12's coverage of if statements.

To make this work without lambda, you'd need to instead code three def statements somewhere else in your file, outside the dictionary in which the functions are to be used, and reference the functions by name:

```
>>> def f1(): return 2 + 2
...
>>> def f2(): return 2 * 4
...
```

```
>>> def f3(): return 2 ** 6
...
>>> key = 'one'
>>> {'already': f1, 'got': f2, 'one': f3}[key]()
64
```

This works, too, but your `def`s may be arbitrarily far away in your file, even if they are just little bits of code. The *code proximity* that `lambda`s provide is especially useful for functions that will only be used in a single context—if the three functions here are not useful anywhere else, it makes sense to embed their definitions within the dictionary as `lambda`s. Moreover, the `def` form requires you to make up names for these little functions that may clash with other names in this file (perhaps unlikely, but always possible).

`lambda`s also come in handy in function-call argument lists as a way to inline temporary function definitions not used anywhere else in your program; we'll see some examples of such other uses later in this chapter, when we study `map`.

How (Not) to Obfuscate Your Python Code

The fact that the body of a `lambda` has to be a single expression (not a series of statements) would seem to place severe limits on how much logic you can pack into a `lambda`. If you know what you're doing, though, you can code most statements in Python as expression-based equivalents.

For example, if you want to print from the body of a `lambda` function, simply say `sys.stdout.write(str(x)+'\n')`, instead of `print(x)` (recall from Chapter 11 that this is what `print` really does). Similarly, to nest logic in a `lambda`, you can use the `if`/`else` ternary expression introduced in Chapter 12, or the equivalent but trickier `and`/`or` combination also described there. As you learned earlier, the following statement:

```
if a:
    b
else:
    c
```

can be emulated by either of these roughly equivalent expressions:

```
b if a else c
((a and b) or c)
```

Because expressions like these can be placed inside a `lambda`, they may be used to implement selection logic within a `lambda` function:

```
>>> lower = (lambda x, y: x if x < y else y)
>>> lower('bb', 'aa')
'aa'
>>> lower('aa', 'bb')
'aa'
```

Furthermore, if you need to perform loops within a `lambda`, you can also embed things like `map` calls and list comprehension expressions (tools we met in earlier chapters and will revisit in this and the next chapter):

```
>>> import sys
>>> showall = lambda x: list(map(sys.stdout.write, x))        # Use list in 3.0

>>> t = showall(['spam\n', 'toast\n', 'eggs\n'])
spam
toast
eggs

>>> showall = lambda x: [sys.stdout.write(line) for line in x]

>>> t = showall(('bright\n', 'side\n', 'of\n', 'life\n'))
bright
side
of
life
```

Now that I've shown you these tricks, I am required by law to ask you to please only use them as a last resort. Without due care, they can lead to unreadable (a.k.a. *obfuscated*) Python code. In general, simple is better than complex, explicit is better than implicit, and full statements are better than arcane expressions. That's why `lambda` is limited to expressions. If you have larger logic to code, use `def`; `lambda` is for small pieces of inline code. On the other hand, you may find these techniques useful in moderation.

Nested lambdas and Scopes

`lambda`s are the main beneficiaries of nested function scope lookup (the E in the LEGB scope rule we studied in Chapter 17). In the following, for example, the `lambda` appears inside a `def`—the typical case—and so can access the value that the name `x` had in the enclosing function's scope at the time that the enclosing function was called:

```
>>> def action(x):
...     return (lambda y: x + y)          # Make and return function, remember x
...
>>> act = action(99)
>>> act
<function <lambda> at 0x00A16A88>
>>> act(2)                               # Call what action returned
101
```

What wasn't illustrated in the prior discussion of nested function scopes is that a `lambda` also has access to the names in any enclosing `lambda`. This case is somewhat obscure, but imagine if we recoded the prior `def` with a `lambda`:

```
>>> action = (lambda x: (lambda y: x + y))
>>> act = action(99)
>>> act(3)
102
```

```
>>> ((lambda x: (lambda y: x + y))(99))(4)
103
```

Here, the nested `lambda` structure makes a function that makes a function when called. In both cases, the nested `lambda`'s code has access to the variable `x` in the enclosing `lambda`. This works, but it's fairly convoluted code; in the interest of readability, nested `lambda`s are generally best avoided.

Why You Will Care: Callbacks

Another very common application of `lambda` is to define inline callback functions for Python's `tkinter` GUI API (this module is named `Tkinter` in Python 2.6). For example, the following creates a button that prints a message on the console when pressed, assuming `tkinter` is available on your computer (it is by default on Windows and other OSs):

```
import sys
from tkinter import Button, mainloop     # Tkinter in 2.6
x = Button(
        text ='Press me',
        command=(lambda:sys.stdout.write('Spam\n')))
x.pack()
mainloop()
```

Here, the callback handler is registered by passing a function generated with a `lambda` to the `command` keyword argument. The advantage of `lambda` over `def` here is that the code that handles a button press is right here, embedded in the button-creation call.

In effect, the `lambda` *defers* execution of the handler until the event occurs: the `write` call happens on button presses, not when the button is created.

Because the nested function scope rules apply to `lambda`s as well, they are also easier to use as callback handlers, as of Python 2.2—they automatically see names in the functions in which they are coded and no longer require passed-in defaults in most cases. This is especially handy for accessing the special `self` instance argument that is a local variable in enclosing class method functions (more on classes in Part VI):

```
class MyGui:
    def makewidgets(self):
        Button(command=(lambda: self.onPress("spam")))
    def onPress(self, message):
        ...use message...
```

In prior releases, even `self` had to be passed in to a `lambda` with defaults.

Mapping Functions over Sequences: map

One of the more common things programs do with lists and other sequences is apply an operation to each item and collect the results. For instance, updating all the counters in a list can be done easily with a `for` loop:

```
>>> counters = [1, 2, 3, 4]
>>>
>>> updated = []
>>> for x in counters:
...     updated.append(x + 10)                # Add 10 to each item
...
>>> updated
[11, 12, 13, 14]
```

But because this is such a common operation, Python actually provides a built-in that does most of the work for you. The `map` function applies a passed-in function to each item in an iterable object and returns a list containing all the function call results. For example:

```
>>> def inc(x): return x + 10                 # Function to be run
...
>>> list(map(inc, counters))                  # Collect results
[11, 12, 13, 14]
```

We met `map` briefly in Chapters 13 and 14, as a way to apply a built-in function to items in an iterable. Here, we make better use of it by passing in a user-defined function to be applied to each item in the list—`map` calls `inc` on each list item and collects all the return values into a new list. Remember that `map` is an iterable in Python 3.0, so a `list` call is used to force it to produce all its results for display here; this isn't necessary in 2.6.

Because `map` expects a function to be passed in, it also happens to be one of the places where `lambda` commonly appears:

```
>>> list(map((lambda x: x + 3), counters))    # Function expression
[4, 5, 6, 7]
```

Here, the function adds 3 to each item in the `counters` list; as this little function isn't needed elsewhere, it was written inline as a `lambda`. Because such uses of `map` are equivalent to `for` loops, with a little extra code you can always code a general mapping utility yourself:

```
>>> def mymap(func, seq):
...     res = []
...     for x in seq: res.append(func(x))
...     return res
```

Assuming the function `inc` is still as it was when it was shown previously, we can map it across a sequence with the built-in or our equivalent:

```
>>> list(map(inc, [1, 2, 3]))                 # Built-in is an iterator
[11, 12, 13]
>>> mymap(inc, [1, 2, 3])                     # Ours builds a list (see generators)
[11, 12, 13]
```

However, as `map` is a built-in, it's always available, always works the same way, and has some performance benefits (as we'll prove in the next chapter, it's usually faster than a manually coded `for` loop). Moreover, `map` can be used in more advanced ways than

shown here. For instance, given multiple sequence arguments, it sends items taken from sequences in parallel as distinct arguments to the function:

```
>>> pow(3, 4)                           # 3**4
81
>>> list(map(pow, [1, 2, 3], [2, 3, 4]))  # 1**2, 2**3, 3**4
[1, 8, 81]
```

With multiple sequences, `map` expects an N-argument function for N sequences. Here, the `pow` function takes two arguments on each call—one from each sequence passed to `map`. It's not much extra work to simulate this multiple-sequence generality in code, too, but we'll postpone doing so until later in the next chapter, after we've met some additional iteration tools.

The `map` call is similar to the list comprehension expressions we studied in Chapter 14 and will meet again in the next chapter, but `map` applies a *function* call to each item instead of an arbitrary *expression*. Because of this limitation, it is a somewhat less general tool. However, in some cases `map` may be faster to run than a list comprehension (e.g., when mapping a built-in function), and it may also require less coding.

Functional Programming Tools: filter and reduce

The `map` function is the simplest representative of a class of Python built-ins used for *functional programming*—tools that apply functions to sequences and other iterables. Its relatives filter out items based on a test function (`filter`) and apply functions to pairs of items and running results (`reduce`). Because they return iterables, `range` and `filter` both require `list` calls to display all their results in 3.0. For example, the following `filter` call picks out items in a sequence that are greater than zero:

```
>>> list(range(-5, 5))                          # An iterator in 3.0
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]

>>> list(filter((lambda x: x > 0), range(-5, 5)))   # An iterator in 3.0
[1, 2, 3, 4]
```

Items in the sequence or iterable for which the function returns a true result are added to the result list. Like `map`, this function is roughly equivalent to a `for` loop, but it is built-in and fast:

```
>>> res = []
>>> for x in range(-5, 5):
...     if x > 0:
...         res.append(x)
...
>>> res
[1, 2, 3, 4]
```

`reduce`, which is a simple built-in function in 2.6 but lives in the `functools` module in 3.0, is more complex. It accepts an iterator to process, but it's not an iterator itself—it

returns a single result. Here are two `reduce` calls that compute the sum and product of the items in a list:

```
>>> from functools import reduce       # Import in 3.0, not in 2.6

>>> reduce((lambda x, y: x + y), [1, 2, 3, 4])
10
>>> reduce((lambda x, y: x * y), [1, 2, 3, 4])
24
```

At each step, `reduce` passes the current sum or product, along with the next item from the list, to the passed-in `lambda` function. By default, the first item in the sequence initializes the starting value. To illustrate, here's the `for` loop equivalent to the first of these calls, with the addition hardcoded inside the loop:

```
>>> L = [1,2,3,4]
>>> res = L[0]
>>> for x in L[1:]:
...     res = res + x
...
>>> res
10
```

Coding your own version of `reduce` is actually fairly straightforward. The following function emulates most of the built-in's behavior and helps demystify its operation in general:

```
>>> def myreduce(function, sequence):
...     tally = sequence[0]
...     for next in sequence[1:]:
...         tally = function(tally, next)
...     return tally
...
>>> myreduce((lambda x, y: x + y), [1, 2, 3, 4, 5])
15
>>> myreduce((lambda x, y: x * y), [1, 2, 3, 4, 5])
120
```

The built-in `reduce` also allows an optional third argument placed before the items in the sequence to serve as a default result when the sequence is empty, but we'll leave this extension as a suggested exercise.

If this coding technique has sparked your interest, you might also be interested in the standard library `operator` module, which provides functions that correspond to built-in expressions and so comes in handy for some uses of functional tools (see Python's library manual for more details on this module):

```
>>> import operator, functools
>>> functools.reduce(operator.add, [2, 4, 6])          # Function-based +
12
>>> functools.reduce((lambda x, y: x + y), [2, 4, 6])
12
```

Together with `map`, `filter` and `reduce` support powerful functional programming techniques. Some observers might also extend the functional programming toolset in Python to include `lambda`, discussed earlier, as well as list comprehensions—a topic we will return to in the next chapter.

Chapter Summary

This chapter took us on a tour of advanced function-related concepts: recursive functions; function annotations; `lambda` expression functions; functional tools such as `map`, `filter`, and `reduce`; and general function design ideas. The next chapter continues the advanced topics motif with a look at generators and a reprisal of iterators and list comprehensions—tools that are just as related to functional programming as to looping statements. Before you move on, though, make sure you've mastered the concepts covered here by working through this chapter's quiz.

Test Your Knowledge: Quiz

1. How are `lambda` expressions and `def` statements related?
2. What's the point of using `lamba`?
3. Compare and contrast `map`, `filter`, and `reduce`.
4. What are function annotations, and how are they used?
5. What are recursive functions, and how are they used?
6. What are some general design guidelines for coding functions?

Test Your Knowledge: Answers

1. Both `lambda` and `def` create function objects to be called later. Because `lambda` is an expression, though, it returns a function object instead of assigning it to a name, and it can be used to nest a function definition in places where a `def` will not work syntactically. A `lambda` only allows for a single implicit return value expression, though; because it does not support a block of statements, it is not ideal for larger functions.
2. `lambda`s allow us to "inline" small units of executable code, defer its execution, and provide it with state in the form of default arguments and enclosing scope variables. Using a `lambda` is never required; you can always code a `def` instead and reference the function by name. `lambda`s come in handy, though, to embed small pieces of deferred code that are unlikely to be used elsewhere in a program. They commonly appear in callback-based program such as GUIs, and they have a natural affinity with function tools like `map` and `filter` that expect a processing function.

3. These three built-in functions all apply another function to items in a sequence (iterable) object and collect results. `map` passes each item to the function and collects all results, `filter` collects items for which the function returns a `True` value, and `reduce` computes a single value by applying the function to an accumulator and successive items. Unlike the other two, `reduce` is available in the `functools` module in 3.0, not the built-in scope.
4. Function annotations, available in 3.0 and later, are syntactic embellishments of a function's arguments and result, which are collected into a dictionary assigned to the function's `__annotations__` attribute. Python places no semantic meaning on these annotations, but simply packages them for potential use by other tools.
5. Recursive functions call themselves either directly or indirectly in order to loop. They may be used to traverse arbitrarily shaped structures, but they can also be used for iteration in general (though the latter role is often more simply and efficiently coded with looping statements).
6. Functions should generally be small, as self-contained as possible, have a single unified purpose, and communicate with other components through input arguments and return values. They may use mutable arguments to communicate results too if changes are expected, and some types of programs imply other communication mechanisms.