CHAPTER 11

Fun with Functions

In This Chapter

- ▶ Saving time by writing functions
- ▶ Passing parameters to functions
- ▶ Global and local namespaces

A *function* is a block of code that performs a specific action and returns a result.

In this chapter, you find out about calling functions, writing functions and their docstrings, and passing arguments to functions. You're also inducted into the arcane mysteries of the namespace.

I Love Chunky Code

Here's how functions make your life easier as a programmer and make things easier on people who read and use your code, too:

- **Functions let you write code only once.**

  When you create code that you want to reuse, be sure to define it as a function. This saves you from typing the same code over again when you want to use it in another program.

- **Functions hide unnecessary complexity from the user.**

  No one ever has to care what this function does unless it's included in your program. And even if someone does use the function, they don't have to care *how* it works, as long as it performs the way its documentation says it will.

- **Functions make your code easier to understand.**

  If you name a function according to what its code does, then someone reading the program just has to look at the name to get an understanding of what the program is doing at that point. (Of course, you should also include comments in your program and comments and docstrings in your function to further explain what's happening.)

- **Functions help you organize your program logically.**

  Writing most of your code chunks in functions helps you organize your program's structure logically, and that can make your programs easier to understand.

CALLING A FUNCTION

Using a function is known as *calling* the function. When you call a function, the function's code runs and returns a result.

To call a function, follow these steps:

1. **Type a name to hold the result of the function followed by an equals sign (=).**

   You don't have to do this, but usually you want to do something with the function result, and it's easier if a name has been given to the result.

2. **Type the name of the function.**

3. **In parentheses, give the function information about what you want it to act on.**

   Giving a function information is called *passing arguments*. Some functions don't take any arguments, but most do.

   You can pass the function a value or a name.

In the following example, the sum() function is given one argument: the name of a list (purchases). The name total is given to the result:

Parameters versus arguments

In this chapter, we use the terms *parameter* and *argument* when discussing the data that a function needs to do its work. Here's a way to understand the difference:

A *parameter* is a placeholder for the data. When you are writing a function, you use parameters to tell the function to expect data to be passed. Then you use the parameter name inside the function to stand for that data.

An *argument* is the actual data that's passed when you call the function. Here's an example:

```
>>> def a_function(a_param):
... print "my parameter is a_param and my argument is",
  repr(a_param)
...
>>> a_function("my arg")
my parameter is a_param and my argument is 'my arg'
```

```
>>> purchases = [13.02, 2.99, 4.45]
>>> total = sum(purchases)
>>> print total
20.46
```

Functions versus methods

What's the difference between a function and a method? Not much. A method is a kind of function — both perform a specific action on some information you give them. However, a method is associated with a particular object and performs its action on that object. Often, that object is a data type. For example, a list comes with the append() method, which lets you add items to the end of the list. Whenever you create a new list object by using a statement such as my_list = [1,2,3], the object my_list has access to all the methods for list objects.

Functions, on the other hand, stand on their own. You don't have to create an object in order to use one.

You call functions and methods using different syntax:

- To use a function, type the function name:

my_function(x)

- To use a method, append it to the name of the object, separating the name and the method with a dot, like this:

my_list.append(x)

DEFINING A FUNCTION

Unless you're using a built-in function, you need to define the function before you can call it. These are the basic steps for writing a function:

1. On the first line, write the def **statement for the function.**

Follow these steps:

a. Write def and the function name.

b. In parentheses, add the names of parameters that the function will act on. Separate parameters with commas.

If your function doesn't have parameters, it still needs the parentheses.

c. Type a colon at the end of the statement.

Your def statement might look something like this:

```
def my_function(arg1, arg2):
```

2. (Optional, but very important) On the next lines, write the docstring for the function.

Follow these steps:

a. On a new line, indent four spaces and type three quotation marks (a triple quote).

b. Type an explanation of what the function does and how to use it.

See the sidebar “What’s up, Doc?” to find out what to include.

c. On the next line, indent four spaces and type a triple quote.

Your function and its docstring might look something like this:

```
def my_function(arg1, arg2):
“””
Perform my function on arg1 using arg2. Return a list.
“””
```

3. Write the function’s code.

TIP

Indent each line of code four spaces. (If your function contains another subordinate or nested code block, such as an if statement, that block’s code must be indented an additional four spaces, and so on.)

When you’re finished, the function might look like this (but containing Python code, of course!):

```
def my_function(arg1, arg2):
        """
        my function does this
        """
        do something
        if this:
            do that
        return something else
```

GIVING ANOTHER NAME TO A FUNCTION

After you import a function, you can give it a new name. The additional name refers to the same object at the same location as the old name, as this example shows:

```
>>> tinyfunction
<function tinyfunction at 0x61430>
>>> myfunc = tinyfunction
>>> myfunc
<function tinyfunction at 0x61430>
```

It's important to remember the difference between *giving a new name to a function* and *calling a function while giving a name to the result.* When you call a function, you add parentheses at the end of the function name. These examples show the difference:

```
>>> myfunction = tinymodule.tinyfunction # giving a new name
```

```
    to a function
  >>> myresult = tinymodule.tinyfunction(2) # calling a function
```

What's up, Doc?

If the first code in a function is a string, it becomes the docstring for the function. The docstring defines what the function does and works with Python's help() utility. We recommend writing docstrings for all your functions.

A docstring usually spans multiple lines, so it begins and ends with triple quotes.

By convention among Python programmers, a basic docstring for a function looks like this:

1. The first line (the only line for a simple function) is a description of what the function does, or a summary if the function does several things or has several options.

For example, take this function that prints something:

```
def printme(me):
    print me
```

This function does only one thing, so the docstring might look like this:

```
"""Print the argument"""
```

2. If the function needs more than one line of documentation, the second line should be blank.

3. The third and subsequent lines explain options, defaults

for optional arguments, and other usage notes.

Say we change the printme() function so it takes an optional argument — a list of words not to print:

```
def printme(me, bad_word_list=None):
    if bad_word_list and me in bad_word_list:
        print "How dare you!"
    else:
        print me
```

We might give it the following docstring:

```
"""
Print argument. Optionally check against list of words
not to print.
Keyword argument:
bad_word_list -- list of words not to print (default
None)
"""
```

You can find out more about Python's docstring conventions by reading Python Enhancement Proposal (PEP) 257 at:

www.python.org/dev/peps/pep-0257

RETURNING VALUES FROM A FUNCTION

Names you define inside a function disappear after the function exits. To get values out of a function, you must do two things:

- **Include a** return **statement in the function with a name or**

value, like this:

```
def my_function(b):
    a = b + 42
return a
```

✔ **Use the function in an expression or in an assignment statement:**

```
my_value = my_function(3)
```

The name on the left side of the equals sign in the assignment statement now stores the value that the function returned, as shown:

```
>>> my_value = my_function(3)
>>> print my_value
45
```

If a function has no return statement, or if the return statement doesn't specify what to return, the function returns None (a built-in name). Here's a function without a return statement.

```
>>> def no_return_function(b):
... print b
...
>>> my_value = no_return_function(3)
3
>>> print my_value
None
```

So why would you ever want a function without a return statement? Some functions manipulate an item in place instead of returning a new item. Some functions perform an action that doesn't need to return a result (for example, saving a message to a log). Such a function might not need a return statement.

The return statement not only passes a value out of the function, it also stops Python from running the rest of the code in the function. So return either should be the last statement in the function or should be used in conditional code, such as an if block, to jump out of the function when a certain condition is met.

Argument Clinic: Passing Data

Parameters are placeholders for information that you give to a function so it can carry out an action.

There are many ways to specify a function's parameters. This section introduces specifying positional parameters, default-value parameters, and arbitrary numbers of parameters. It also describes how to avoid the quirks of default-value parameters and mutable objects.

INTRODUCING PARAMETERS AND ARGUMENTS

Here are a few things to know about function parameters:

- **A function can have any number of parameters (or no parameters).**
- **When you define a function, you specify how many parameters it has.**

  It can have a specific number of parameters or an indefinite number of parameters.
- **You can define default values for any or all of the parameters.**

When you call a function, you need to give it a value corresponding to each parameter. (This actual value is called an *argument.*) Here are a few things to know about arguments:

- **Arguments are passed when you call the function.**
- **If the parameter has a default value, you don't have to pass an argument.**
- **You can pass an argument as a literal or as a name.**

  If you pass an argument as a name, the function acts on the object the name refers to.

SPECIFYING ARGUMENTS WHEN YOU CALL A FUNCTION

The basic way of specifying arguments is to *pass* them when you call the function.

To pass an argument, type it between the parentheses of a function when you call the function, like this:

```
>>> a_function(myarg)
```

The following example creates a function that prints the value of its argument, gives a name to a tuple, and passes the name to the function:

```
>>> def a_function(x):
... print "you passed me the argument", repr(x)
...
>>> myarg = (1,2,3)
>>> a_function(myarg)
you passed me the argument (1, 2, 3)
```

The following example passes the same function a string literal without defining a name:

```
>>> a_function("hello")
you passed me the argument 'hello'
```

SPECIFYING ARGUMENTS WITH KEYWORDS AND DEFAULT VALUES

A function can have default values for some of the parameters. (Such default-value parameters are sometimes called *keyword arguments* .) Here are some circumstances in which default-value parameters are useful:

- **A particular value will be used more often than others.**

  For example, Python's round() function takes a required number argument and an optional precision argument. The precision parameter defaults to 0 because, most of the time, when you round a number, you want the integer closest to the number rather than a decimal.

- **The function is designed to work in more than one way, depending on which arguments are passed.**

  For example, the open() function opens a file for reading by default. If you want to open a file for writing, you need to specify a mode argument, 'w'.

When adding default-value parameters to function definitions, follow these rules:

1. **Type any parameters that won't have keywords first.**

   Parameters without keywords are called *positional*.

2. **Type keyword arguments in the form** keyword=value .

The following function definition has one required (positional) parameter (ingredient) and two default-value parameters:

```
def recipes(ingredient, servings=4, mode="Vegetarian"):
```

When calling a function that includes default-value parameters, follow these rules:

1. Pass values for all positional parameters first.

Use the same order in which the function's parameters are defined.

2. Pass values or names for any default-value parameters whose default values you want to override.

Here are some additional guidelines for default-value parameters:

- Don't pass information for default-value parameters whose default values you want to use. For example, this function call uses both defaults:

```
>>> recipes('shiitake mushrooms')
```

- If you include default-value parameters in the order in which they were defined, you can just pass their values, like this:

```
>>> recipes('arugula', 8, "Carnivore")
```

- If you are including default-value parameters in a different order or skipping some parameters, use the keyword=value format, as shown here:

```
>>> recipes('arugula', mode="No peanuts")
```

AVOIDING THE QUIRKS OF DEFAULT VALUES

Default values are useful as function parameters, but you need to be aware of their quirks.

Immutable default values stay the same

Python evaluates default values in function definitions only once — when the function is defined, that is, when the def statement first runs. (Another way to say this is that Python *binds the default to the parameter*

.) def is an executable statement; this is why you have to define a function before you can call it. After that, if you change the value of a name you used as a default in the function definition, the function will continue to use the original value when the value is an immutable data type. In the example below, the def is passed the value of the integer mynum . When mynum is changed later, the function still uses the old value:

```
>>> mynum = 8 # Integers are immutable.
>>> def f(arg=mynum):
... return arg
...
>>> f()
8
>>> mynum = 10 # We're changing mynum here...
>>> f() # ...but the function still uses 8.
8
```

Mutable default values can change

When you use a mutable object (such as a list, dictionary, or class instance) as a default and change the object inside the function, then the next time you call the function, the function uses the changed object. For example, the following function creates a list that accumulates the arguments passed to it.

```
>>> def f(q, mylist=[]):
... mylist.append(q)
... return mylist
...
>>> print f('a')
['a']
>>> print f('b')
['a', 'b']
>>> print f('c')
['a', 'b', 'c']
```

Here's what's going on:

1. When the function is defined, it creates an empty list object named mylist.
2. Inside the function, the append() method adds an item to the list. The append() method changes the contents of the list *in place* — that is, mylist still refers to the same object after being changed.
3. When the function is called again, it grabs the same list from the same place and uses the new contents.

Assignment (=) always changes

Assignment always causes a name to refer to a different object (that is, it *rebinds*), even if the object is mutable. So if you use assignment (=) inside the function rather than the append() method, mylist doesn't accumulate arguments.

```
    >>> def f(q, mylist=[]):
```

```
... mylist = mylist + q
... return mylist
...
>>> a = [1]
>>> f(a)
[1]
>>> a = [2]
>>> f(a)
[2]
```

Augmented assignment (+=) doesn't rebind mutable data types

Augmented assignment operators (+= and *=) rebind when used with an immutable data type, but don't rebind when used with a mutable data type, such as a list. When used with a list, += is the equivalent of the extend() method; it adds items to the end of a list. So if we rewrite the function to use augmented assignment, the function extends mylist (which is mutable) by adding the argument to the end of the list:

```
>>> def f(q, mylist=[]):
... mylist += q
... return mylist
...
>>> a = [1]
>>> f(a)
[1]
```

```
>>> a = [2]
>>> f(a)
[1, 2]
```

Working on a copy of a mutable object

Because using mutable objects as arguments can have unexpected results, you're better off operating on a copy of the object. This function creates a new copy of its list and then adds a value to each list element:

```
def f(mylist, data):
    newlist = mylist[:] # make a copy of mylist
    for i in range(len(mylist)):
        newlist[i] = mylist[i] + data
    return newlist
```

Calling the above function doesn't change the original list:

```
>>> alist = [1, 2, 3]
>>> x = f(alist, 5)
>>> print x
[6, 7, 8]
>>> print alist
[1, 2, 3]
```

Using a default value of None to redefine a mutable object

If you want a function to redefine a list as empty each time you call it, use a default parameter of mylist=None . None is immutable, so the function won't be referring to a list object that might change.

The following code gives the parameter mylist a default value of None in its def statement and then redefines mylist as an empty list inside the function:

```
def f(a, mylist=None):
    if mylist is None:
        mylist = []
    mylist.append(a)
    return mylist
```

When you call the above function several times, it doesn't accumulate arguments:

```
>>> a = 1
>>> f(a)
[1]
>>> a = 2
>>> f(a)
[2]
```

SPECIFYING A FUNCTION WITH AN ARBITRARY NUMBER OF ARGUMENTS

To pass an arbitrary number of arguments to a function, use the parameter *args in the def statement for the function. Here are tips and rules for using *args :

- The parameter *args stands for a set of positional arguments that aren't explicitly named in the function definition.

- When you call the function and pass it arguments, *args automatically creates a tuple out of the arguments.
- The parameter *args must come last in the function definition (or next-to-last if there is also a **kwargs parameter).

To pass an arbitrary number of keyword=value pairs to a function, use the parameter **kwargs in the function's def statement. Here's how it works:

- When you call the function and pass the keyword=value pairs, **kwargs automatically creates a dictionary from them.
- The parameter **kwargs must come last in the function definition.

The names don't have to be *args or **kwargs ; Python cares only that the * or ** operator comes first in the name. But Pythonistas always use *args and **kwargs . This consistency makes programs easier to read by humans.

This example program shows how *args and **kwargs work in a function that also has two positional parameters:

```
def a_function(a, b, *args, **kwargs):
    print "a is", a
    print "b is", b
```

```
        print "*args is this tuple:", args
        print "**kwargs is this dictionary:", kwargs
```

It produces the following result:

```
>>> a_function(1, '2', 'three', 'blind', 'mice', see="how",
  they="run")
a is 1
b is 2
*args is this tuple: ('three', 'blind', 'mice')
**kwargs is this dictionary: {'see': 'how', 'they': 'run'}
```

UNPACKING ARGUMENTS

To pass the elements of a list, tuple, or dictionary as arguments in a function, use the * operator (for lists and tuples) or the ** operator (for dictionaries) when calling the function to *unpack* the elements.

To unpack, you need these things:

- **A function**

  The function can have any kind of parameters — positional or default, *args, or **kwargs . . . or even no parameters at all.

- **A list, tuple, or dictionary with the *same* number of elements as the number of parameters in the function**

Unpacking lists or tuples

To pass the elements of a list or tuple as arguments in a function, use this line of code (substituting your function name and list or tuple name):

```
    myfunction(*my_list_or_tuple)
```

The following code defines and calls a function, passing a list as its argument, using the * operator:

```
    >>> def func_with_three_args(a, b, c):
    ... print “the arguments are:”, a, b, c
    ...
    >>> weapons = [‘fear’, ‘surprise’, ‘ruthless efficiency’]
    >>> func_with_three_args(*weapons)
    the arguments are: fear surprise ruthless efficiency
```

Unpacking dictionaries

To pass the key:value pairs of a dictionary to a function, use this line of code (substituting your function name and dict name):

```
    myfunction(**mydict)
```

The dictionary keys must have the same names as the function’s parameters. The following code passes a dictionary to func_with-_three_args() using the ** operator. The dictionary’s keys match the argument names:

```
    >>> not_in_stock = {“a”:”Red Leicester”, “b”:”Jarlsberg”, “c”:”-
       Camembert”}
    >>> func_with_three_args(**not_in_stock)
    the arguments are: Red Leicester Jarlsberg Camembert
```

What's in a Namespace

A *namespace,* also called a *symbol table* or *scope,* is storage for the names of objects Python knows about.

The most important thing to remember about *function namespaces* is this: When you give a name to a value, you are always assigning within the function's local namespace unless

- You explicitly say that the name is global
- You specify that the name is an attribute of a particular object

DISCOVERING WHERE PYTHON LOOKS FOR NAMES

Python has three basic layers of namespaces, listed here in order of most specific to most general. Python looks for names in this order and stops looking as soon as it finds the name:

1. Local (names defined inside a class, function, or method)
2. Global (names defined inside a module — often function names and class names, but can be other names, too)
3. Built-in (names that are always available)

If Python can't find a name in any of those places, it raises a NameError exception. If you get an AttributeError , that means that Python found the leftmost (first) name but didn't find the name after the dot. Each object has its own namespace that Python searches. For more information about object namespaces, see Chapters 12 and 13.

UNDERSTANDING FUNCTION NAMESPACES

When Python encounters a function definition in a chunk of code, it *executes* the definition. That prepares the function for being called later. When Python executes the definition, the following things happen:

- The function's name is stored in the current namespace.

  If the function is part of a module, this is the module's namespace. If the function was imported directly, it might be the main namespace in interactive mode.
- Python creates a new namespace for storing any local names defined within the function.

REMEMBER

Python searches from local to global for names, but it doesn't search from global to local. That means global namespaces don't know about the names inside local namespaces.

This behavior of Python's is useful for hiding complexity. It also prevents bugs that might occur if all names were automatically global — if that were true, it would be easy to use a name twice without realizing it, and unexpected things might happen. When names are automatically local, they are less likely to conflict.

How local and global names work with assignment

Names are automatically local (unless you tell Python to treat them as global). This means that if you give a name to a value (using an equals sign) inside a function, when you exit the function, Python will forget about the name. Here's a program that demonstrates this feature of namespaces using Groucho's famous saying. You could type it into a text file and save it with the name groucho.py :

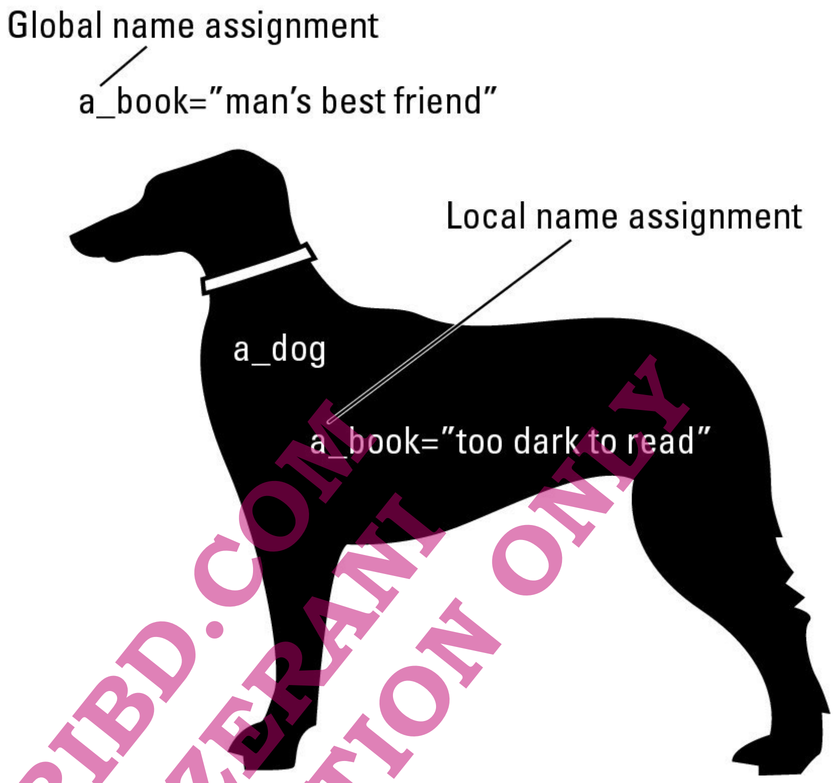
```
# groucho.py
a_book = “man’s best friend”
print “outside of a dog, a book is”, a_book
def a_dog():
    a_book = “too dark to read”
    print “inside of a dog, it’s”, a_book
a_dog()
print “we’re back outside of the dog again”
print “and a book is again”, a_book
```

Figure 11-1 illustrates global and local names.

If we run the program from the command line, it has this output:

```
% python groucho.py
outside of a dog, a book is man’s best friend
inside of a dog, it’s too dark to read
we’re back outside of the dog again
and a book is again man’s best friend
```

Figure 11-1: A global and a local name assignment.



How local and global names work with references

Because Python searches for names first locally and then globally, you can *reference* a global name inside the function — assuming the function doesn't contain a local name that's the same as the global name. This example program (we call it brightdog.py) and Figure 11-2 show that a function can access the name a_book defined outside the function:

```
# brightdog.py
a_book = "man's best friend"
print "outside of a dog, a book is", a_book
def a_bright_dog():
    print "inside of THIS dog, a book is still", a_book
a_bright_dog()
```

This code, run on the command line, produces the following result:

```
% python brightdog.py
outside of a dog, a book is man’s best friend
inside of THIS dog, a book is still man’s best friend
```

You can also define a function inside another function. This creates a new local namespace for the nested function. This namespace is called a *nested scope*. Nested scopes are mostly useful for functional programming. (Chapter 16 has some information on functional programming.) Nested scopes work differently in Python 2.1 and later than they work in earlier versions. We don’t cover them in this book.

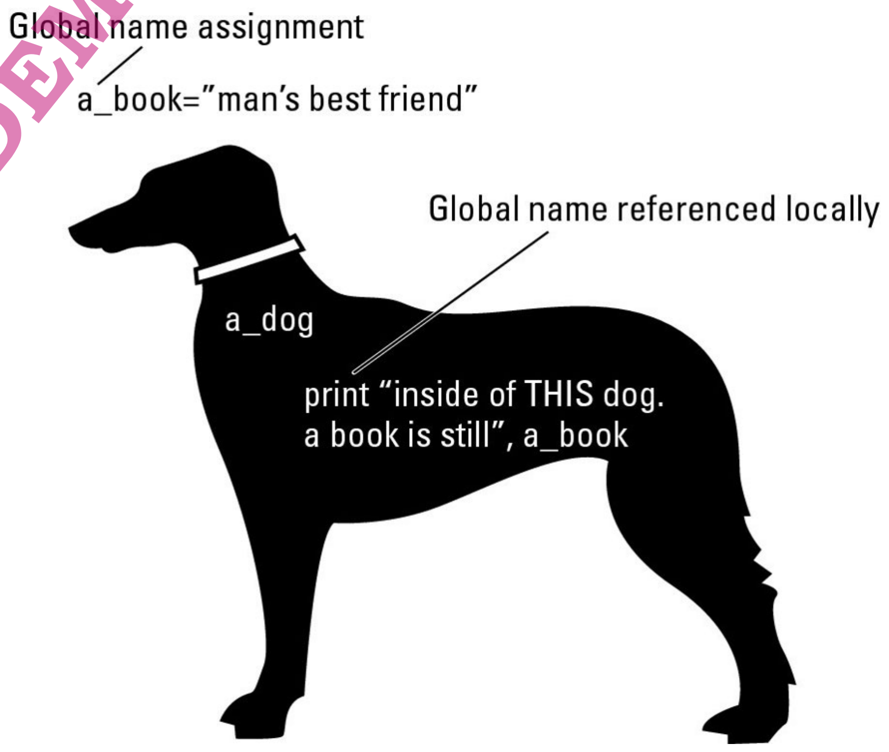

Global name assignment
a_book="man’s best friend"
Global name referenced locally
a_dog
print “inside of THIS dog.
a book is still”, a_book

Figure 11-2: A global name referenced locally.

THINK GLOBALLY, ACT LOCALLY

It's easy to explicitly tell a function that a name is global: You just use the keyword global with the name. In the example program below, which we call colors.py , defining the name eggcolor as global allows the function to change the value of that name, and the changed value remains in effect after Python exits the function. In contrast, the name meat , which isn't declared global, is changed only inside the function.

```
# colors.py
eggcolor = "green"
meat = "ham"
print eggcolor, "eggs and", meat
def breakfast():
    global eggcolor
    eggcolor = "red"
    meat = "bacon"
    print eggcolor, "eggs and", meat
breakfast()
print eggcolor, "eggs and", meat
```

If you run this code on the command line, you get the following result:

```
% python colors.py
green eggs and ham
red eggs and bacon
red eggs and ham
```

REMEMBER

It’s usually a Bad Idea to use global names because they are more likely to cause name conflicts in complex programs.

SORTING OUT MODULE NAMESPACES

Each module (a file with Python code, ending in .py) has its own namespace. All the functions defined in the module use the module’s namespace as their global namespace. The functions won’t look elsewhere (except for the built-in namespace) for names that are referenced within them. This means that the author of a module can use global names in a module without worrying about their conflicting with global names used elsewhere. In other words, whenever you see “global” used in Python, think “module global.”

TIP

For more about modules, see Chapter 12.

When a module imports another module, the imported module’s name becomes part of the importing module’s namespace. You can then access its functions and other information by using the module name, a dot, and the function name, the same way you can when you import a module into interactive mode.

Imagine you have a module called mymod.py that does nothing but import another module:

```
# module 'mymod'
import math
```

If you import this module into interactive mode, you can print the math.pi constant by typing this code:

```
>>> import mymod
>>> mymod.math.pi
3.14159265359
```

But the name math and its constant pi are known only inside the module. The surrounding namespace of interactive mode still doesn't know about the math module:

```
>>> print math.pi
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

It's possible to manipulate a module's global names by using the same notation used to refer to its functions, modulename.itemname. You should do this *only* if you know what you're doing. (It's an advanced feature we don't cover in this book.)