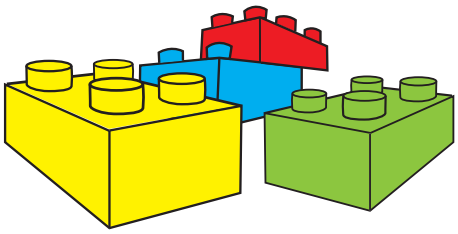
CHAPTER 13

Functions

Pretty soon, our programs are going to start getting bigger and more complicated. We need some ways to organize them in smaller pieces so they're easier to write and keep track of.

There are three main ways to break programs into smaller parts. *Functions* are like building blocks of code that you can use over and over again. *Objects* are a way of describing pieces of your program as self-contained units. *Modules* are just separate files that contain parts of your program. In this chapter, we'll learn about functions, and in the next two chapters, we'll learn about objects and modules. Then we'll have all the basic tools we need to start using graphics and sounds, and to create games.

Functions—the building blocks



In the simplest of terms, a *function* is a chunk of code that does something. It's a small piece that you can use to build a bigger program. You can put the piece together with other pieces, just like building something with toy blocks.

You create or *define* a function with Python's `def` keyword. You then use or *call* the function by using its name. Let's start with a simple example.

Creating a function

The code in the following listing defines a function and then uses it. This function prints a mailing address to the screen.

Listing 13.1 Creating and using a function

```
def printMyAddress():
    print "Warren Sande"
    print "123 Main Street"
    print "Ottawa, Ontario, Canada"
    print "K2M 2E9"
    print
printMyAddress()
```

Defines (creates) the function

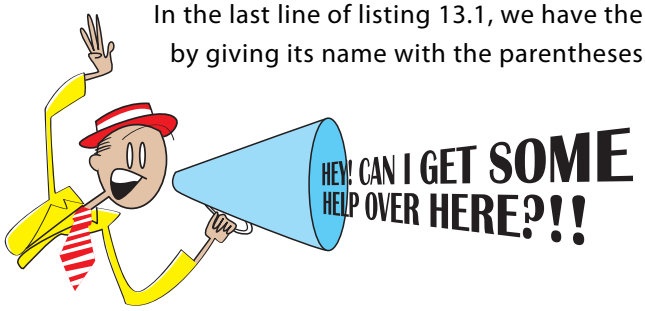
Calls (uses) the function

In line 1, we define a function, using the `def` keyword. We give the name of the function followed by parentheses `()` and then a colon:

```
def printMyAddress():
```

I will explain what the parentheses are for soon. The colon tells Python that a block of code is coming next (just like `for` loops, `while` loops, and `if` statements).

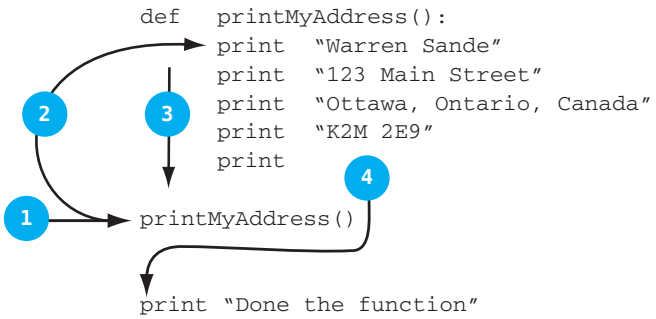
Then, we have the code that makes up the function.



In the last line of listing 13.1, we have the main program: we *call* the function by giving its name with the parentheses. This is where the program starts running. This one line makes the program run the code in the function we just defined.

When the main program calls a function, it's like the function is helping the main program get its job done.

The code inside the `def` block isn't part of the main program, so when the program runs, it skips over that part and starts with the first line that isn't inside a `def` block. The next figure shows what happens when you call a function. I added one extra line at the end of the program that prints a message after the function is done.



These are the steps in the previous figure:

1. Start here. This is the beginning of the main program.
2. When we call the function, we jump to the first line of code in the function.
3. Execute each line of the function.
4. When the function is finished, we continue where we left off in the main program.

Calling a function

Calling a function means running the code that is inside the function. If you define a function but never call it, that code will never run.

You call a function by using its name and a set of parentheses. Sometimes there's something in the parentheses and sometimes not.

Try running the program in listing 13.1 and see what happens. You should see something like this:

```
>>> ================== RESTART ==================
>>>
Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

>>>
```

Now, that's exactly the same output we'd have gotten from a simpler program that looks like this:

```
print "Warren Sande"
print "123 Main Street"
print "Ottawa, Ontario, Canada"
print "K2M 2E9"
print
```

So why did we go to the trouble of making things more complex and using a function in listing 13.1?

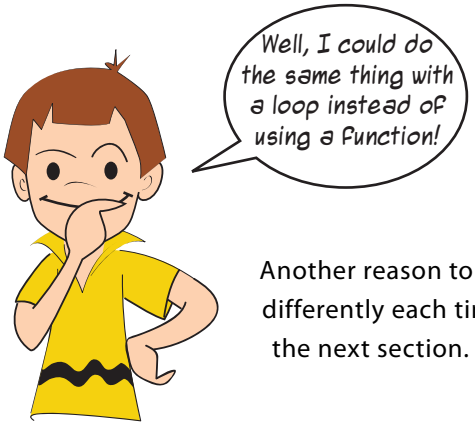
The main reason to use functions is that, once you have defined them, you can use them over and over again just by *calling* them. So if we wanted to print the address five times, we could do this:

```
printMyAddress()
printMyAddress()
printMyAddress()
printMyAddress()
printMyAddress()
```

And the output would be

```
Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
```

You might say that you could do the same thing with a loop instead of a function.

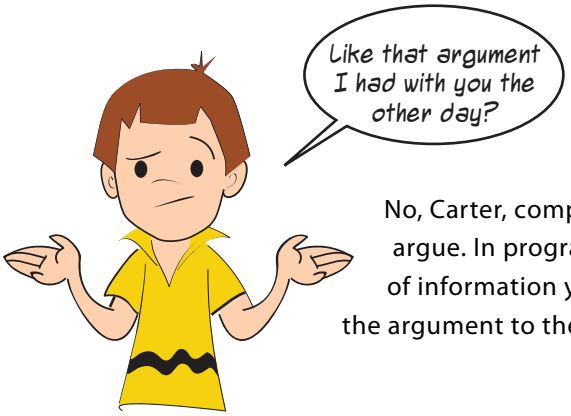


I knew that was coming…. In this case, you *could* do the same thing with a loop. But if you wanted to print the address at different places in a program instead of all at once, a loop wouldn't work.

Another reason to use a function is that you can make it behave differently each time it runs. You're going to see how to do that in the next section.

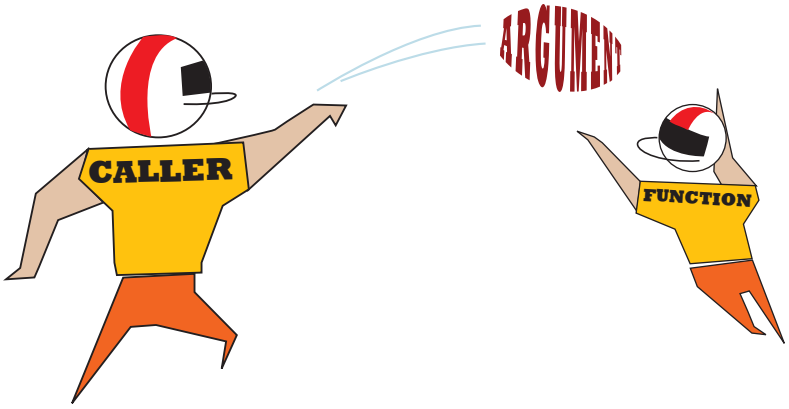
Passing arguments to a function

Now it's time to see what the parentheses are for: *arguments*!



No, Carter, computers are very agreeable—they never argue. In programming, the term *argument* means a piece of information you give to a function. We say that you *pass* the argument to the function.

Imagine that you wanted to be able to use the address-printing function for any member of your family. The address would be the same for everybody, but the name would be different each time. Instead of having the name hard-coded as "Warren Sande" in the function, you can make it a variable. The variable is *passed* to the function when you call it.



An example is the easiest way to see how this works. In listing 13.2, I modified the address-printing function to use one argument for the name. Arguments are named, just like other variables. I called this variable `myName`.

When the function runs, the variable `myName` gets filled in with whatever argument we pass to the function when we call it. We pass the argument to the function by putting it inside the parentheses when we call the function.

So, in listing 13.2, the argument **myName** is assigned the value "Carter Sande".

Listing 13.2 Passing an argument to a function

```
def printMyAddress(myName):
    print myName
    print "123 Main Street"
    print "Ottawa, Ontario, Canada"
    print "K2M 2E9"
    print

printMyAddress("Carter Sande")
```

Passes myName argument to the function

Prints the name

Passes "Carter Sande" as the argument to the function; the variable myName inside the function will have the value "Carter Sande"

If we run the code in listing 13.2, we get exactly what you'd expect:

```
>>> ==================== RESTART ====================
>>>
Carter Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
```

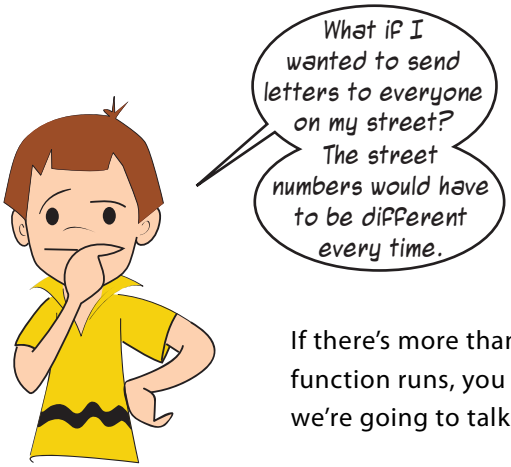
This looks the same as the output we got from the first program, when we didn't use arguments. But now we can make the address print differently every time, like this:

```
printMyAddress("Carter Sande")
printMyAddress("Warren Sande")
printMyAddress("Kyra Sande")
printMyAddress("Patricia Sande")
```

And now, the output is different each time the function is called. The name changes, because we *pass* the function a different name each time:

```
>>> ========================== RESTART ==========================
>>>
Carter Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Warren Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Kyra Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9

Patricia Sande
123 Main Street
Ottawa, Ontario, Canada
K2M 2E9
```

Notice that whatever value we passed to the function was used inside the function and was printed as the name part of the address.



If there's more than one thing that is different every time the function runs, you need more than one argument. That's what we're going to talk about next.

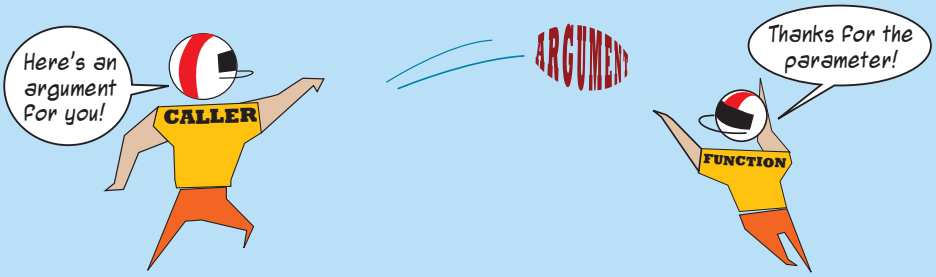
Functions with more than one argument

In listing 13.2, our function had a single argument. But functions can have more than one argument. In fact, they can have as many as you need. Let's try an example with two arguments, and I think you'll get the idea. Then you can keep adding as many arguments as you need for the functions in your programs.

WORD BOX

There's another term you'll hear when talking about passing things to a function: *parameters*. Some people say that the terms *argument* and *parameter* are interchangeable. So you could say, "I passed two parameters to that function," or "I passed two arguments to that function."

Some people say that you should use *argument* when talking about the passing part (when you call the function), and *parameter* when talking about the receiving part (what is inside the function).



As long as you use *argument* or *parameter* to talk about passing values to functions, programmers will know what you mean.

To send Carter's letters to everyone on the street, our address-printing function will need two arguments: one for the name, and one for the house number. The next listing shows what this would look like.

Listing 13.3 Function with two arguments

```
def printMyAddress(someName, houseNum):
    print someName
    print houseNum,
    print "Main Street"
    print "Ottawa, Ontario, Canada"
    print "K2M 2E9"
    print

printMyAddress("Carter Sande", "45")
printMyAddress("Jack Black", "64")
printMyAddress("Tom Green", "22")
printMyAddress("Todd White", "36")
```

Uses two variables, for two arguments

Both variables get printed

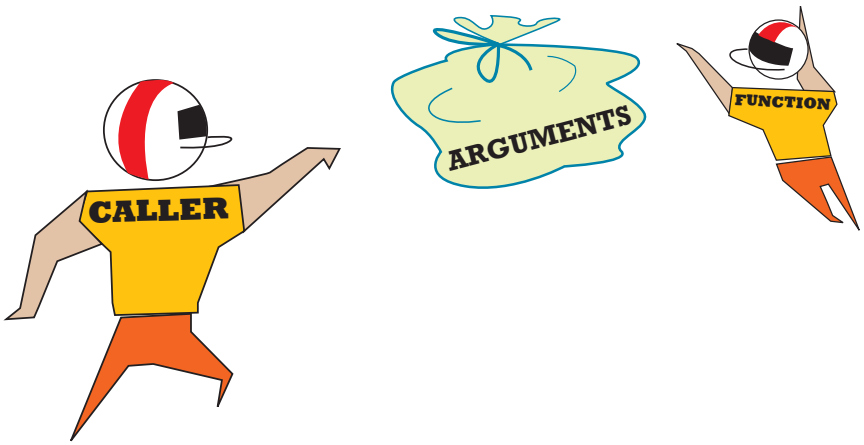
Comma makes house number and street print on the same line

Calls the function, passing it two parameters

When you use multiple arguments (or parameters), you separate them with a comma, just like items in a list, which brings us to our next topic….

How many is too many?

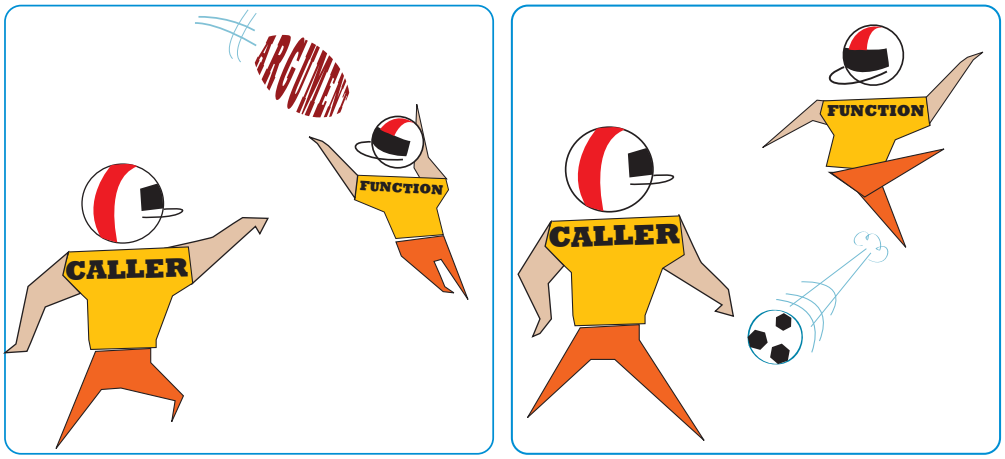
I said before that you can pass as many arguments as you want to a function. That is true, but if your function has more than five or six arguments, it might be time to think of doing things another way. One thing you can do is collect all the arguments in a *list* and then pass the list to the function. That way, you're passing a single variable (the list variable), which just happens to contain a bunch of values. It might make your code easier to read.



Functions that return a value

So far, our functions have just been doing stuff for us. But a very useful thing about functions is that they can also send you something back.

You have seen that you can send information (arguments) to functions, but functions can also send information back to the caller. The value that comes back from a function is called the *result* or *return value*.



Returning a value

The way you make a function return a value is to use the Python **`return`** keyword inside the function. Here's an example:

```
def calculateTax(price, tax_rate):
    taxTotal = price + (price * tax_rate)
    return taxTotal
```

This will send the value **`taxTotal`** back out to the part of the program that called the function.

But when it's sent back, where does it go? Returned values go back to whatever code called the function. Here's an example:

```
totalPrice = calculateTax(7.99, 0.06)
```

The **`calculateTax`** function will return the value 8.4694, and that value will be assigned to **`totalPrice`**.

You can use a function to return values anywhere you'd use an expression. You can assign the return value to a variable (as we just did), use it in another expression, or print it, like this:

```
>>> print calculateTax(7.99, 0.06)
8.4694
>>> total = calculateTax(7.99, 0.06) + calculateTax(6.59,
  0.08)
```

You can also do nothing with the returned value, like this:

```
>>> calculateTax(7.49, 0.07)
```

In the last example, the function ran and calculated the total with tax, but we didn't use the result.

Let's make a program with a function that returns a value. In listing 13.4, the **`calculateTax()`** function returns a value. We give it the price before tax and the tax rate, and it returns the price after tax. We'll assign this value to a variable. So instead of just using the function's name like we did before, we need a variable, an equal sign (**`=`**), and then the function's name. The variable will be assigned the result that the **`calculateTax()`** function gives back.

Listing 13.4 Creating and using a function that returns a value

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    return total

my_price = float(raw_input ("Enter a price: "))

totalPrice = calculateTax(my_price, 0.06)
print "price = ", my_price, " Total price = ", totalPrice
```

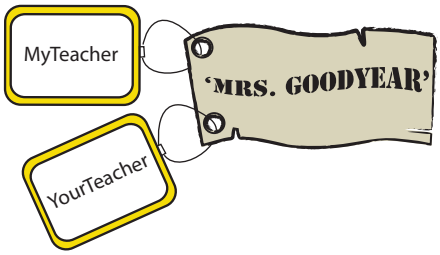
Sends result back to the main program

Function calculates tax and returns total

Calls function and stores the result in totalPrice

Try typing in, saving, and running the program in listing 13.4. Notice that the tax rate is fixed as 0.06 (which equals 6 percent tax) in the code. If the program had to handle different tax rates, you could have the user enter the tax rate as well as the price.

Variable scope



You might have noticed that we have variables outside the function, like `totalPrice`, as well as variables inside the function, like `total`. These are just two names for the same thing. It's like back in chapter 2, when we had `YourTeacher = MyTeacher`.

In our `calculateTax` example, `totalPrice` and `total` are two tags attached to the same thing. With functions, the names inside the function are only created when the function runs. They don't even exist before the function runs or after it has finished running. Python has something called *memory management* that does this automatically. Python creates new names to use inside the function when it runs, *and then deletes them when the function is finished*. That last part is important: when the function is done running, any names inside it cease to exist.

While the function is running, the names *outside* the function are sort of on hold—they're not being used. Only the names inside the function are being used. The part of a program where a variable is used (or available to be used) is called its *scope*.

Local variables

In listing 13.4, the variables `price` and `total` were only used within the function. We say that `price`, `total`, and `tax_rate` are *in the scope* of the `calculateTax()` function. Another term that is used is *local*. The `price`, `total`, and `tax_rate` variables are *local variables* in the `calculateTax()` function.

One way to see what this means is to add a line to the program in listing 13.4 that tries to print the value of `price` somewhere outside the function. The following listing does this.

Listing 13.5 Trying to print a local variable

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    return total

my_price = float(raw_input ("Enter a price: "))

totalPrice = calculateTax(my_price, 0.06)
print "price = ", my_price, " Total price = ", totalPrice
print price
```

Defines a function to calculate tax and return the total

Calls the function and stores and prints the result

Tries to print price

If you run this, you'll get an error that looks like this:

```
Traceback (most recent call last):
  File "C:/.../Listing_13-5.py", line 9, in <module>
    print price
NameError: name 'price' is not defined
```

This line explains the error

The last line of the error message tells the story: when we're not inside the `calculateTax()` function, the variable `price` is *not defined*. It only exists while the function is running. When we tried to print the value of `price` from outside the function (when the function was not running), we got an error.

Global variables

In contrast to the *local* variable `price`, the variables `my_price` and `totalPrice` in listing 13.5 are defined *outside* the function, in the main part of the program. We use the term *global* for a variable that has a wider scope. In this case, *wider* means the main part of the program, not what's inside the function. If we expanded the program in listing 13.5, we could use the variables `my_price` and `totalPrice` in another place in the program, and they would still have the values we gave them earlier. They would still be *in scope*. Because we can use them anywhere in the program, we say they're *global variables*.

In listing 13.5, when we were outside the function and tried to print a variable that was inside the function, we got an error. The variable didn't exist; it was *out of scope*. What do you think will happen if we do the opposite: try to print a global variable from inside the function?

The next listing tries to print the variable **my_price** from inside the **calculateTax()** function. Try it and see what happens.

Listing 13.6 Using a global variable inside a function

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)
    print my_price                          <-- Tries to print my_price
    return total

my_price = float(raw_input ("Enter a price: "))

totalPrice = calculateTax(my_price, 0.06)
print "price = ", my_price, " Total price = ", totalPrice
```

Did it work? Yes! But why?

When we started talking about variable scope, I told you that Python uses memory management to automatically create local variables when a function runs. The memory manager does some other things, too. In a function, if you use a variable name that has been defined in the main program, Python will let you use the global variable as long as you don't try to change it.

So you can do this

```
print my_price
```

or this

```
your_price = my_price
```

because neither of these changes **my_price**.

If any part of the function tries to change the variable, Python creates a new local variable instead. So if you do this

```
my_price = my_price + 10
```

then **my_price** is a new local variable that Python creates when the function runs.

In the example in listing 13.6, the value that was printed was the *global* variable **my_price**, because the function didn't change it. The program in listing 13.7 shows you that, if you do

try to change the global variable inside the function, you get a new, local variable instead. Try running it and see.

Listing 13.7 Trying to modify a global variable inside a function

```
def calculateTax(price, tax_rate):
    total = price + (price * tax_rate)

    my_price = 10000
    print "my_price (inside function) = ", my_price
    return total

my_price = float(raw_input ("Enter a price: "))

totalPrice = calculateTax(my_price, 0.06)
print "price = ", my_price, " Total price = ", totalPrice
print "my_price (outside function) = ", my_price
```

Modifies my_price inside the function

Prints the local version of my_price

The variable my_price here is a different chunk of memory than the my_price here

Prints the global version of my_price

If you run the code in listing 13.7, the output will look like this:

```
>>> ========================== RESTART ==========================
>>>
Enter a price: 7.99
my_price (inside function) =  10000
price =  7.99  Total price =  8.4694
my_price (outside function) =  7.99
```

Prints my_price from inside the function

Prints my_price from outside the function

As you can see, there are now two different variables called `my_price`, with different values. One is the *local variable* inside the `calculateTax()` function that we set to 10,000. The other is the *global variable* we defined in the main program to capture the user's input, which was 7.99.

Forcing a global

In the last section, you saw that, if you try to change the value of a *global variable* from inside a function, Python creates a new *local variable* instead. This is meant to prevent functions from accidentally changing global variables.

However, there are times when you *want* to change a global variable from inside a function. So how do you do it?

Python has a keyword, `global`, that lets you do that. You use it like this:

```
def calculateTax(price, tax_rate):
    global my_price
```

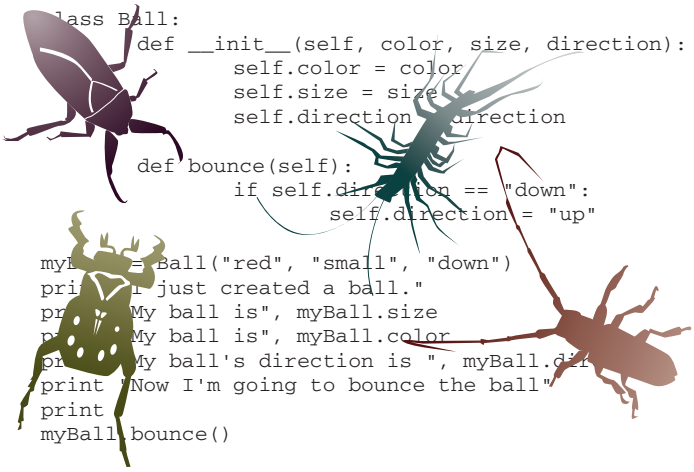
Tells Python you want to use the global version of my_price

If you use the `global` keyword, Python *won't* make a new local variable called `my_price`. It will use the global variable `my_price`. If there's no global variable called `my_price`, it will create one.

A bit of advice on naming variables

You saw in the previous sections that you can use the same names for global variables and local variables. Python will automatically create new local variables when it needs to, or you can prevent that with the `global` keyword. However, I strongly recommend that you don't reuse names.

As you might have noticed from some of the examples, it can be difficult to know whether the variable is the local version or the global version. It makes the code more confusing, because you have different variables with the same name. And wherever there's confusion, bugs love to creep in.



So for now, I recommend you use different names for local variables and global variables. That way, there's no confusion, and you'll keep the bugs at bay.

What did you learn?

In this chapter, you learned

- What a function is
- What arguments (or parameters) are
- How to pass an argument to a function
- How to pass multiple arguments to a function
- How to make a function return a value to the caller

- What variable scope is, and what local and global variables are
- How to use global variables in a function

Test your knowledge

1. What keyword do you use to create a function?
2. How do you call a function?
3. How do you pass information (arguments) to a function?
4. What's the maximum number of arguments a function can have?
5. How do you get information back from a function?
6. What happens to local variables in a function after the function is finished running?

Try it out

1. Write a function to print your name in big letters, like this:

```
   CCCC        A        RRRRR   TTTTTTT   EEEEEE   RRRRR
  C    C      A A       R    R     T      E        R    R
 C           A   A      R    R     T      EEEE     R    R
 C          AAAAAAA     RRRRR      T      E        RRRRR
  C    C   A       A    R    R     T      E        R    R
   CCCC   A         A   R    R     T      EEEEEE   R    R
```

   Write a program that calls the function a number of times.
2. Make a function that will allow you to print any name, address, street, city, state or province, zip or postal code, and country in the world. (Hint: It needs seven arguments. You can pass them as individual arguments or as a list.)
3. Try using the example from listing 13.7, but making `my_price` global so you can see the difference in the resulting output.
4. Write a function to calculate the total value of some change—quarters, dimes, nickels, and pennies (just like in the last "Try it out" question from chapter 5). The function should return the total value of the coins. Then write a program that calls the function. The output should look like this when it runs:

```
quarters: 3
dimes: 6
nickels: 7
pennies: 2
total is $1.72
```