CHAPTER 6

Abstraction

In this chapter, you learn how to group statements into functions, which enables you to tell the computer how to do something, and to tell it only once. You won't need to give it the same detailed instructions over and over. The chapter provides a thorough introduction to parameters and scoping, and you learn what recursion is and what it can do for your programs.

Laziness Is a Virtue

The programs we've written so far have been pretty small, but if you want to make something bigger, you'll soon run into trouble. Consider what happens if you have written some code in one place and need to use it in another place as well. For example, let's say you wrote a snippet of code that computed some *Fibonacci numbers* (a series of numbers in which each number is the sum of the two previous ones):

```
fibs = [0, 1]
for i in range(8):
    fibs.append(fibs[-2] + fibs[-1])
```

After running this, `fibs` contains the first ten Fibonacci numbers:

```
>>> fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

This is all right if what you want is to calculate the first ten Fibonacci numbers once. You could even change the `for` loop to work with a dynamic range, with the length of the resulting sequence supplied by the user:

```
fibs = [0, 1]
num = input('How many Fibonacci numbers do you want? ')
for i in range(num-2):
    fibs.append(fibs[-2] + fibs[-1])
print fibs
```

Note Remember that you can use `raw_input` if you want to read in a plain string. In this case, you would then need to convert it to an integer by using the `int` function.

But what if you also want to use the numbers for something else? You could certainly just write the same loop again when needed, but what if you had written a more complicated piece of code, such as one that downloaded a set of web pages and computed the frequencies of all the words used? Would you still want to write all the code several times, once for each time you needed it? No, real programmers don't do that. Real programmers are lazy. Not lazy in a bad way, but in the sense that they don't do unnecessary work.

So what do real programmers do? They make their programs more *abstract*. You could make the previous program more abstract as follows:

```
num = input('How many numbers do you want? ')
print fibs(num)
```

Here, only what is specific to *this program* is written concretely (reading in the number and printing out the result). Actually, computing the Fibonacci numbers is done in an abstract manner: you simply tell the computer to do it. You don't say specifically how it should be done. You create a function called `fibs`, and use it when you need the functionality of the little Fibonacci program. It saves you a lot of effort if you need it in several places.

Abstraction and Structure

Abstraction can be useful as a labor saver, but it is actually more important than that. It is the key to making computer programs understandable to humans (which is essential, whether you're writing them or reading them). The computers themselves are perfectly happy with very concrete and specific instructions, but humans generally aren't. If you ask me for directions to the cinema, for example, you wouldn't want me to answer, "Walk 10 steps forward, turn 90 degrees to your left, walk another 5 steps, turn 45 degrees to your right, walk 123 steps." You would soon lose track, wouldn't you?

Now, if I instead told you to "Walk down this street until you get to a bridge, cross the bridge, and the cinema is to your left," you would certainly understand me. The point is that you already know how to walk down the street and how to cross a bridge. You don't need explicit instructions on how to do either.

You structure computer programs in a similar fashion. Your programs should be quite abstract, as in "Download page, compute frequencies, and print the frequency of each word." This is easily understandable. In fact, let's translate this high-level description to a Python program right now:

```
page = download_page()
freqs = compute_frequencies(page)
for word, freq in freqs:
    print word, freq
```

From reading this, you can understand what the program does. However, you haven't explicitly said anything about *how* it should do it. You just tell the computer to download the page and compute the frequencies. The specifics of these operations will need to be written somewhere else—in separate *function definitions*.

Creating Your Own Functions

A function is something you can call (possibly with some parameters—the things you put in the parentheses), which performs an action and returns a value.[1] In general, you can tell whether something is callable or not with the built-in function `callable`:

```
 >>> import math
>>> x = 1
>>> y = math.sqrt
>>> callable(x)
False
>>> callable(y)
True
```

■ **Note** The function `callable` no longer exists in Python 3.0. With that version, you will need to use the expression `hasattr(func, __call__)`. For more information about `hasattr`, see Chapter 7.

As you know from the previous section, creating functions is central to structured programming. So how do you define a function? You do this with the `def` (or "function definition") statement:

```
def hello(name):
    return 'Hello, ' + name + '!'
```

After running this, you have a new function available, called `hello`, which returns a string with a greeting for the name given as the only parameter. You can use this function just like you use the built-in ones:

```
>>> print hello('world')
Hello, world!
>>> print hello('Gumby')
Hello, Gumby!
```

Pretty neat, huh? Consider how you would write a function that returned a list of Fibonacci numbers. Easy! You just use the code from before, and instead of reading in a number from the user, you receive it as a parameter:

```
def fibs(num):
    result = [0, 1]
    for i in range(num-2):
        result.append(result[-2] + result[-1])
    return result
```

1. Actually, functions in Python don't always return values. More on this later in the chapter.

After running this statement, you've basically told the interpreter how to calculate Fibonacci numbers. Now you don't have to worry about the details anymore. You simply use the function `fibs`:

```
>>> fibs(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> fibs(15)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

The names `num` and `result` are quite arbitrary in this example, but `return` is important. The `return` statement is used to return something from the function (which is also how we used it in the preceding `hello` function).

Documenting Functions

If you want to document your functions so that you're certain that others will understand them later on, you can add comments (beginning with the hash sign, #). Another way of writing comments is simply to write strings by themselves. Such strings can be particularly useful in some places, such as immediately after a `def` statement (and at the beginning of a module or a class—you learn more about classes in Chapter 7 and modules in Chapter 10). If you put a string at the beginning of a function, it is stored as part of the function and is called a *docstring*. The following code demonstrates how to add a docstring to a function:

```
def square(x):
    'Calculates the square of the number x.'
    return x*x
```

The docstring may be accessed like this:

```
>>> square.__doc__
'Calculates the square of the number x.'
```

■ **Note** `__doc__` is a function attribute. You'll learn a lot more about attributes in Chapter 7. The double underscores in the attribute name mean that this is a special attribute. Special or "magic" attributes like this are discussed in Chapter 9.

A special built-in function called `help` can be quite useful. If you use it in the interactive interpreter, you can get information about a function, including its docstring:

```
>>> help(square)
Help on function square in module __main__:

square(x)
    Calculates the square of the number x.
```

You meet the `help` function again in Chapter 10.

Functions That Aren't Really Functions

Functions, in the mathematical sense, always return something that is calculated from their parameters. In Python, some functions don't return anything. In other languages (such as Pascal), such functions may be called other things (such as *procedures*), but in Python, a function is a function, even if it technically isn't. Functions that don't return anything simply don't have a `return` statement. Or, if they *do* have `return` statements, there is no value after the word `return`:

```
def test():
    print 'This is printed'
    return
    print 'This is not'
```

Here, the `return` statement is used simply to end the function:

```
>>> x = test()
This is printed
```

As you can see, the second `print` statement is skipped. (This is a bit like using `break` in loops, except that you break out of the function.) But if `test` doesn't return anything, what does `x` refer to? Let's see:

```
>>> x
>>>
```

Nothing there. Let's look a bit closer:

```
>>> print x
None
```

That's a familiar value: `None`. So all functions *do* return something; it's just that they return `None` when you don't tell them what to return. I guess I was a bit unfair when I said that some functions aren't really functions.

■ **Caution** Don't let this default behavior trip you up. If you return values from inside `if` statements and the like, be sure you've covered every case, so you don't accidentally return `None` when the caller is expecting a sequence, for example.

The Magic of Parameters

Using functions is pretty straightforward, and creating them isn't all that complicated either. The way parameters work may, however, seem a bit like magic at times. First, let's do the basics.

Where Do the Values Come From?

Sometimes, when defining a function, you may wonder where parameters get their values. In general, you shouldn't worry about that. Writing a function is a matter of providing a service to whatever part of your program (and possibly even other programs) might need it. Your task is to make sure the function does its job if it is supplied with acceptable parameters, and preferably fails in an obvious manner if the parameters are wrong. (You do this with `assert` or exceptions in general. More about exceptions in Chapter 8.)

■ **Note** The variables you write after your function name in `def` statements are often called the **formal** parameters of the function. The values you supply when you **call** the function are called the **actual** parameters, or **arguments**. In general, I won't be too picky about the distinction. If it is important, I will call the actual parameters *values* to distinguish them from the formal parameters.

Can I Change a Parameter?

So, your function gets a set of values through its parameters. Can you change them? And what happens if you do? Well, the parameters are just variables like all others, so this works as you would expect. Assigning a new value to a parameter inside a function won't change the outside world at all:

```
>>> def try_to_change(n):
        n = 'Mr. Gumby'

>>> name = 'Mrs. Entity'
>>> try_to_change(name)
>>> name
'Mrs. Entity'
```

Inside `try_to_change`, the parameter `n` gets a new value, but as you can see, that doesn't affect the variable `name`. After all, it's a completely different variable. It's just as if you did something like this:

```
>>> name = 'Mrs. Entity'
>>> n = name # This is almost what happens when passing a parameter
>>> n = 'Mr. Gumby' # This is done inside the function
>>> name
'Mrs. Entity'
```

Here, the result is obvious. While the variable `n` is changed, the variable `name` is not. Similarly, when you rebind (assign to) a parameter inside a function, variables outside the function will not be affected.

■ **Note** Parameters are kept in what is called a **local scope**. Scoping is discussed later in this chapter.

Strings (and numbers and tuples) are *immutable*, which means that you can't modify them (that is, you can only replace them with new values). Therefore, there isn't much to say about them as parameters. But consider what happens if you use a mutable data structure such as a list:

```
>>> def change(n):
        n[0] = 'Mr. Gumby'

>>> names = ['Mrs. Entity', 'Mrs. Thing']
>>> change(names)
>>> names
['Mr. Gumby', 'Mrs. Thing']
```

In this example, the parameter is changed. There is one crucial difference between this example and the previous one. In the previous one, we simply gave the local variable a new value, but in this one, we actually *modify* the list to which the variable `names` is bound. Does that sound strange? It's not really that strange. Let's do it again without the function call:

```
>>> names = ['Mrs. Entity', 'Mrs. Thing']
>>> n = names # Again pretending to pass names as a parameter
>>> n[0] = 'Mr. Gumby' # Change the list
>>> names
['Mr. Gumby', 'Mrs. Thing']
```

You've seen this sort of thing before. When two variables refer to the same list, they . . . refer to the same list. It's really as simple as that. If you want to avoid this, you must make a *copy* of the list. When you do slicing on a sequence, the returned slice is always a copy. Thus, if you make a slice of the *entire list*, you get a copy:

```
>>> names = ['Mrs. Entity', 'Mrs. Thing']
>>> n = names[:]
```

Now `n` and `names` contain two *separate* (nonidentical) lists that are *equal*:

```
>>> n is names
False
>>> n == names
True
```

If you change `n` now (as you did inside the function `change`), it won't affect `names`:

```
>>> n[0] = 'Mr. Gumby'
>>> n
['Mr. Gumby', 'Mrs. Thing']
>>> names
['Mrs. Entity', 'Mrs. Thing']
```

Let's try this trick with `change`:

```
>>> change(names[:])
>>> names
['Mrs. Entity', 'Mrs. Thing']
```

Now the parameter n contains a copy, and your original list is safe.

■ **Note** In case you're wondering, names that are local to a function, including parameters, do not clash with names outside the function (that is, global ones). For more information about this, see the discussion of scoping, later in this chapter.

Why Would I Want to Modify My Parameters?

Using a function to change a data structure (such as a list or a dictionary) can be a good way of introducing abstraction into your program. Let's say you want to write a program that stores names and that allows you to look up people by their first, middle, or last names. You might use a data structure like this:

```
storage = {}
storage['first'] = {}
storage['middle'] = {}
storage['last'] = {}
```

The data structure `storage` is a dictionary with three keys: `'first'`, `'middle'`, and `'last'`. Under each of these keys, you store another dictionary. In these subdictionaries, you'll use names (first, middle, or last) as keys, and insert lists of people as values. For example, to add me to this structure, you could do the following:

```
>>> me = 'Magnus Lie Hetland'
>>> storage['first']['Magnus'] = [me]
>>> storage['middle']['Lie'] = [me]
>>> storage['last']['Hetland'] = [me]
```

Under each key, you store a list of people. In this case, the lists contain only me.

Now, if you want a list of all the people registered who have the middle name Lie, you could do the following:

```
>>> storage['middle']['Lie']
['Magnus Lie Hetland']
```

As you can see, adding people to this structure is a bit tedious, especially when you get more people with the same first, middle, or last names, because then you need to extend the list that is already stored under that name. Let's add my sister, and let's assume you don't know what is already stored in the database:

```
>>> my_sister = 'Anne Lie Hetland'
>>> storage['first'].setdefault('Anne', []).append(my_sister)
>>> storage['middle'].setdefault('Lie', []).append(my_sister)
>>> storage['last'].setdefault('Hetland', []).append(my_sister)
>>> storage['first']['Anne']
['Anne Lie Hetland']
```

```
>>> storage['middle']['Lie']
['Magnus Lie Hetland', 'Anne Lie Hetland']
```

Imagine writing a large program filled with updates like this. It would quickly become quite unwieldy.

The point of abstraction is to hide all the gory details of the updates, and you can do that with functions. Let's first make a function to initialize a data structure:

```
def init(data):
    data['first'] = {}
    data['middle'] = {}
    data['last'] = {}
```

In the preceding code, I've simply moved the initialization statements inside a function. You can use it like this:

```
>>> storage = {}
>>> init(storage)
>>> storage
{'middle': {}, 'last': {}, 'first': {}}
```

As you can see, the function has taken care of the initialization, making the code much more readable.

■ **Note** The keys of a dictionary don't have a specific order, so when a dictionary is printed out, the order may vary. If the order is different in your interpreter, don't worry about it.

Before writing a function for storing names, let's write one for getting them:

```
def lookup(data, label, name):
    return data[label].get(name)
```

With `lookup`, you can take a label (such as `'middle'`) and a name (such as `'Lie'`) and get a list of full names returned. In other words, assuming my name was stored, you could do this:

```
>>> lookup(storage, 'middle', 'Lie')
['Magnus Lie Hetland']
```

It's important to notice that the list that is returned is the same list that is stored in the data structure. So if you change the list, the change also affects the data structure. (This is not the case if no people are found; then you simply return `None`.)

Now it's time to write the function that stores a name in your structure (don't worry if it doesn't make sense to you immediately):

```
def store(data, full_name):
    names = full_name.split()
    if len(names) == 2: names.insert(1, '')
    labels = 'first', 'middle', 'last'
```

```
    for label, name in zip(labels, names):
        people = lookup(data, label, name)
        if people:
            people.append(full_name)
        else:
            data[label][name] = [full_name]
```

The `store` function performs the following steps:

1. You enter the function with the parameters `data` and `full_name` set to some values that you receive from the outside world.
2. You make yourself a list called `names` by splitting `full_name`.
3. If the length of `names` is 2 (you have only a first and a last name), you insert an empty string as a middle name.
4. You store the strings `'first'`, `'middle'`, and `'last'` as a tuple in `labels`. (You could certainly use a list here; it's just convenient to drop the brackets.)
5. You use the `zip` function to combine the labels and names so they line up properly, and for each pair (`label, name`), you do the following:
   - Fetch the list belonging to the given label and name.
   - Append `full_name` to that list, or insert a new list if needed.

Let's try it out:

```
>>> MyNames = {}
>>> init(MyNames)
>>> store(MyNames, 'Magnus Lie Hetland')
>>> lookup(MyNames, 'middle', 'Lie')
['Magnus Lie Hetland']
```

It seems to work. Let's try some more:

```
>>> store(MyNames, 'Robin Hood')
>>> store(MyNames, 'Robin Locksley')
>>> lookup(MyNames, 'first', 'Robin')
['Robin Hood', 'Robin Locksley']
>>> store(MyNames, 'Mr. Gumby')
>>> lookup(MyNames, 'middle', '')
['Robin Hood', 'Robin Locksley', 'Mr. Gumby']
```

As you can see, if more people share the same first, middle, or last name, you can retrieve them all together.

■ **Note** This sort of application is well suited to object-oriented programming, which is explained in the next chapter.

What If My Parameter Is Immutable?

In some languages (such as C++, Pascal, and Ada), rebinding parameters and having these changes affect variables outside the function is an everyday thing. In Python, it's not directly possible; you can modify only the parameter objects themselves. But what if you have an immutable parameter, such as a number?

Sorry, but it can't be done. What you should do is return all the values you need from your function (as a tuple, if there is more than one). For example, a function that increments the numeric value of a variable by one could be written like this:

```
>>> def inc(x): return x + 1
...

>>> foo = 10
>>> foo = inc(foo)
>>> foo
11
```

If you *really* wanted to modify your parameter, you could use a trick such as wrapping your value in a list, like this:

```
>>> def inc(x): x[0] = x[0] + 1
...
>>> foo = [10]
>>> inc(foo)
>>> foo
[11]
```

Simply returning the new value would be a cleaner solution, though.

Keyword Parameters and Defaults

The parameters we've been using until now are called *positional parameters* because their positions are important—more important than their names, in fact. The techniques introduced in this section let you sidestep the positions altogether, and while they may take some getting used to, you will quickly see how useful they are as your programs grow in size.

Consider the following two functions:

```
def hello_1(greeting, name):
    print '%s, %s!' % (greeting, name)

def hello_2(name, greeting):
    print '%s, %s!' % (name, greeting)
```

They both do *exactly* the same thing, only with their parameter names reversed:

```
>>> hello_1('Hello', 'world')
Hello, world!
>>> hello_2('Hello', 'world')
Hello, world!
```

Sometimes (especially if you have many parameters) the order may be hard to remember. To make things easier, you can supply the *name* of your parameter:

```
>>> hello_1(greeting='Hello', name='world')
Hello, world!
```

The order here doesn't matter at all:

```
>>> hello_1(name='world', greeting='Hello')
Hello, world!
```

The names *do*, however (as you may have gathered):

```
>>> hello_2(greeting='Hello', name='world')
world, Hello!
```

The parameters that are supplied with a name like this are called *keyword parameters*. On their own, the key strength of keyword parameters is that they can help clarify the role of each parameter. Instead of needing to use some odd and mysterious call like this:

```
>>> store('Mr. Brainsample', 10, 20, 13, 5)
```

you could use this:

```
>>> store(patient='Mr. Brainsample', hour=10, minute=20, day=13, month=5)
```

Even though it takes a bit more typing, it is absolutely clear what each parameter does. Also, if you get the order mixed up, it doesn't matter.

What really makes keyword arguments rock, however, is that you can give the parameters in the function default values:

```
def hello_3(greeting='Hello', name='world'):
    print '%s, %s!' % (greeting, name)
```

When a parameter has a default value like this, you don't need to supply it when you call the function! You can supply none, some, or all, as the situation might dictate:

```
>>> hello_3()
Hello, world!
>>> hello_3('Greetings')
Greetings, world!
>>> hello_3('Greetings', 'universe')
Greetings, universe!
```

As you can see, this works well with positional parameters, except that you must supply the greeting if you want to supply the name. What if you want to supply *only* the name, leaving the default value for the greeting? I'm sure you've guessed it by now:

```
>>> hello_3(name='Gumby')
Hello, Gumby!
```

Pretty nifty, huh? And that's not all. You can combine positional and keyword parameters. The only requirement is that all the positional parameters come first. If they don't, the interpreter won't know which ones they are (that is, which position they are supposed to have).

■ **Note** Unless you know what you're doing, you might want to avoid mixing positional and keyword parameters. That approach is generally used when you have a small number of mandatory parameters and many modifying parameters with default values.

For example, our `hello` function might require a name, but allow us to (optionally) specify the greeting and the punctuation:

```
def hello_4(name, greeting='Hello', punctuation='!'):
    print '%s, %s%s' % (greeting, name, punctuation)
```

This function can be called in many ways. Here are some of them:

```
>>> hello_4('Mars')
Hello, Mars!
>>> hello_4('Mars', 'Howdy')
Howdy, Mars!
>>> hello_4('Mars', 'Howdy', '...')
Howdy, Mars...
>>> hello_4('Mars', punctuation='.')
Hello, Mars.
>>> hello_4('Mars', greeting='Top of the morning to ya')
Top of the morning to ya, Mars!
>>> hello_4()
Traceback (most recent call last):
  File "<pyshell#64>", line 1, in ?
    hello_4()
TypeError: hello_4() takes at least 1 argument (0 given)
```

■ **Note** If I had given `name` a default value as well, the last example wouldn't have raised an exception.

That's pretty flexible, isn't it? And we didn't really need to do much to achieve it either. In the next section we get even *more* flexible.

Collecting Parameters

Sometimes it can be useful to allow the user to supply any number of parameters. For example, in the name-storing program (described in the section "Why Would I Want to Modify My Parameters?" earlier in this chapter), you can store only one name at a time. It would be nice to be able to store more names, like this:

```
>>> store(data, name1, name2, name3)
```

For this to be useful, you should be allowed to supply as many names as you want. Actually, that's quite possible.

Try the following function definition:

```
def print_params(*params):
    print params
```

Here, I seemingly specify only one parameter, but it has an odd little star (or asterisk) in front of it. What does that mean? Let's call the function with a single parameter and see what happens:

```
>>> print_params('Testing')
('Testing',)
```

You can see that what is printed out is a tuple because it has a comma in it. (Those tuples of length one are a bit odd, aren't they?) So using a star in front of a parameter puts it in a tuple? The plural in `params` ought to give a clue about what's going on:

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

The star in front of the parameter puts all the values into the same tuple. It gathers them up, so to speak. You may wonder if we can combine this with ordinary parameters. Let's write another function:

```
def print_params_2(title, *params):
    print title
    print params
```

and try it:

```
>>> print_params_2('Params:', 1, 2, 3)
Params:
(1, 2, 3)
```

It works! So the star means "Gather up the rest of the positional parameters." I bet if I don't give any parameters to gather, `params` will be an empty tuple:

```
>>> print_params_2('Nothing:')
Nothing:
()
```

Indeed. How useful! Does it handle keyword arguments (the same as parameters), too?

```
>>> print_params_2('Hmm...', something=42)
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in ?
    print_params_2('Hmm...', something=42)
TypeError: print_params_2() got an unexpected keyword argument 'something'
```

Doesn't look like it. So we probably need another "gathering" operator for keyword arguments. What do you think that might be? Perhaps **?

```
def print_params_3(**params):
    print params
```

At least the interpreter doesn't complain about the function. Let's try to call it:

```
>>> print_params_3(x=1, y=2, z=3)
{'z': 3, 'x': 1, 'y': 2}
```

Yep, we get a dictionary rather than a tuple. Let's put them all together:

```
def print_params_4(x, y, z=3, *pospar, **keypar):
    print x, y, z
    print pospar
    print keypar
```

This works just as expected:

```
>>> print_params_4(1, 2, 3, 5, 6, 7, foo=1, bar=2)
1 2 3
(5, 6, 7)
{'foo': 1, 'bar': 2}
>>> print_params_4(1, 2)
1 2 3
()
{}
```

By combining all these techniques, you can do quite a lot. If you wonder how some combination might work (or whether it's allowed), just try it! (In the next section, you see how * and ** can be used when a function is called as well, regardless of whether they were used in the function definition.)

Now, back to the original problem: how you can use this in the name-storing example. The solution is shown here:

```
def store(data, *full_names):
    for full_name in full_names:
        names = full_name.split()
        if len(names) == 2: names.insert(1, '')
        labels = 'first', 'middle', 'last'
        for label, name in zip(labels, names):
            people = lookup(data, label, name)
            if people:
                people.append(full_name)
            else:
                data[label][name] = [full_name]
```

Using this function is just as easy as using the previous version, which accepted only one name:

```
>>> d = {}
>>> init(d)
>>> store(d, 'Han Solo')
```

But now you can also do this:

```
>>> store(d, 'Luke Skywalker', 'Anakin Skywalker')
>>> lookup(d, 'last', 'Skywalker')
['Luke Skywalker', 'Anakin Skywalker']
```

Reversing the Process

Now you've learned about gathering up parameters in tuples and dictionaries, but it is in fact possible to do the "opposite" as well, with the same two operators, * and **. What might the opposite of parameter gathering be? Let's say we have the following function available:

```
def add(x, y): return x + y
```

■ **Note** You can find a more efficient version of this function in the `operator` module.

Also, let's say you have a tuple with two numbers that you want to add:

```
params = (1, 2)
```

This is more or less the opposite of what we did previously. Instead of gathering the parameters, we want to *distribute* them. This is simply done by using the * operator at the "other end"—that is, when calling the function rather than when defining it:

```
>>> add(*params)
3
```

This works with parts of a parameter list, too, as long as the expanded part is last. You can use the same technique with dictionaries, using the ** operator. Assuming that you have defined `hello_3` as before, you can do the following:

```
>>> params = {'name': 'Sir Robin', 'greeting': 'Well met'}
>>> hello_3(**params)
Well met, Sir Robin!
```

Using * (or **) both when you define and call the function will simply pass the tuple or dictionary right through, so you might as well not have bothered:

```
>>> def with_stars(**kwds):
        print kwds['name'], 'is', kwds['age'], 'years old'

>>> def without_stars(kwds):
        print kwds['name'], 'is', kwds['age'], 'years old'
```

```
>>> args = {'name': 'Mr. Gumby', 'age': 42}
>>> with_stars(**args)
Mr. Gumby is 42 years old
>>> without_stars(args)
Mr. Gumby is 42 years old
```

As you can see, in `with_stars`, I use stars both when defining and calling the function. In `without_stars`, I don't use the stars in either place but achieve exactly the same effect. So the stars are really useful only if you use them *either* when defining a function (to allow a varying number of arguments) *or* when calling a function (to "splice in" a dictionary or a sequence).

■ **Tip** It may be useful to use these splicing operators to "pass through" parameters, without worrying too much about how many there are, and so forth. Here is an example:

```
def foo(x, y, z, m=0, n=0):
    print x, y, z, m, n
def call_foo(*args,**kwds):
    print "Calling foo!"
    foo(*args,**kwds)
```

This can be particularly useful when calling the constructor of a superclass (see Chapter 9 for more on that).

Parameter Practice

With so many ways of supplying and receiving parameters, it's easy to get confused. So let me tie it all together with an example. First, let's define some functions:

```
def story(**kwds):
    return 'Once upon a time, there was a ' \
           '%(job)s called %(name)s.' % kwds

def power(x, y, *others):
    if others:
        print 'Received redundant parameters:', others
    return pow(x, y)

 def interval(start, stop=None, step=1):
    'Imitates range() for step > 0'
    if stop is None:             # If the stop is not supplied...
        start, stop = 0, start   # shuffle the parameters
    result = []
```

```
    i = start                   # We start counting at the start index
    while i < stop:             # Until the index reaches the stop index...
        result.append(i)        # ...append the index to the result...
        i += step               # ...increment the index with the step (> 0)
    return result
```

Now let's try them out:

```
>>> print story(job='king', name='Gumby')
Once upon a time, there was a king called Gumby.
>>> print story(name='Sir Robin', job='brave knight')
Once upon a time, there was a brave knight called Sir Robin.
>>> params = {'job': 'language', 'name': 'Python'}
>>> print story(**params)
Once upon a time, there was a language called Python.
>>> del params['job']
>>> print story(job='stroke of genius', **params)
Once upon a time, there was a stroke of genius called Python.
>>> power(2,3)
8
>>> power(3,2)
9
>>> power(y=3,x=2)
8
>>> params = (5,) * 2
>>> power(*params)
3125
>>> power(3, 3, 'Hello, world')
Received redundant parameters: ('Hello, world',)
27
>>> interval(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> interval(1,5)
[1, 2, 3, 4]
>>> interval(3,12,4)
[3, 7, 11]
>>> power(*interval(3,7))
Received redundant parameters: (5, 6)
81
```

Feel free to experiment with these functions and functions of your own until you are confident that you understand how this stuff works.

Scoping

What *are* variables, really? You can think of them as names referring to values. So, after the assignment `x = 1`, the name `x` refers to the value 1. It's almost like using dictionaries, where keys refer to values, except that you're using an "invisible" dictionary. Actually, this isn't far from the truth. There is a built-in function called `vars`, which returns this dictionary:

```
>>> x = 1
>>> scope = vars()
>>> scope['x']
1
>>> scope['x'] += 1
>>> x
2
```

■ **Caution** In general, you should not modify the dictionary returned by `vars` because, according to the official Python documentation, the result is undefined. In other words, you might not get the result you're after.

This sort of "invisible dictionary" is called a *namespace* or *scope*. So, how many namespaces are there? In addition to the global scope, each function call creates a new one:

```
>>> def foo(): x = 42
...
>>> x = 1
>>> foo()
>>> x
1
```

Here `foo` changes (rebinds) the variable `x`, but when you look at it in the end, it hasn't changed after all. That's because when you call `foo`, a *new* namespace is created, which is used for the block *inside* `foo`. The assignment `x = 42` is performed in this inner scope (the *local* namespace), and therefore it doesn't affect the `x` in the outer (*global*) scope. Variables that are used inside functions like this are called *local variables* (as opposed to global variables). The parameters work just like local variables, so there is no problem in having a parameter with the same name as a global variable:

```
>>> def output(x): print x
...
>>> x = 1
>>> y = 2
>>> output(y)
2
```

So far, so good. But what if you want to access the global variables inside a function? As long as you only want to *read* the value of the variable (that is, you don't want to rebind it), there is generally no problem:

```
>>> def combine(parameter): print parameter + external
...
>>> external = 'berry'
>>> combine('Shrub')
Shrubberry
```

■ **Caution** Referencing global variables like this is a source of many bugs. Use global variables with care.

THE PROBLEM OF SHADOWING

Reading the value of global variables is not a problem in general, but one thing may make it problematic. If a local variable or parameter exists with the same name as the global variable you want to access, you can't do it directly. The global variable is *shadowed* by the local one.

If needed, you can still gain access to the global variable by using the function `globals`, a close relative of `vars`, which returns a dictionary with the global variables. (`locals` returns a dictionary with the local variables.)

For example, if you had a global variable called `parameter` in the previous example, you couldn't access it from within `combine` because you have a parameter with the same name. In a pinch, however, you could have referred to it as `globals()['parameter']`:

```
>>> def combine(parameter):
        print parameter + globals()['parameter']
...
>>> parameter = 'berry'
>>> combine('Shrub')
Shrubberry
```

Rebinding global variables (making them refer to some new value) is another matter. If you assign a value to a variable inside a function, it automatically becomes local unless you tell Python otherwise. And how do you think you can tell it to make a variable global?

```
>>> x = 1
>>> def change_global():
        global x
        x = x + 1

>>> change_global()
>>> x
2
```

Piece of cake!

NESTED SCOPES

Python functions may be nested—you can put one inside another.[2] Here is an example:

```
def foo():
    def bar():
        print "Hello, world!"
    bar()
```

Nesting is normally not all that useful, but there is one particular application that stands out: using one function to "create" another. This means that you can (among other things) write functions like the following:

```
def multiplier(factor):
    def multiplyByFactor(number):
        return number*factor
    return multiplyByFactor
```

One function is inside another, and the outer function *returns the inner one*; that is, the function itself is returned—it is not called. What's important is that the returned function still has access to the scope where it was defined; in other words, it carries its environment (and the associated local variables) with it!

Each time the outer function is called, the inner one gets redefined, and each time, the variable `factor` may have a new value. Because of Python's nested scopes, this variable from the outer local scope (of `multiplier`) is accessible in the inner function later on, as follows:

```
>>> double = multiplier(2)
>>> double(5)
10
>>> triple = multiplier(3)
>>> triple(3)
9
>>> multiplier(5)(4)
20
```

A function such as `multiplyByFactor` that stores its enclosing scopes is called a **closure**.

Normally, you cannot rebind variables in outer scopes. In Python 3.0, however, the keyword `nonlocal` is introduced. It is used in much the same way as `global`, and lets you assign to variables in outer (but non-global) scopes.

Recursion

You've learned a lot about making functions and calling them. You also know that functions can call other functions. What *might* come as a surprise is that functions can call *themselves*.

2. This topic is a bit advanced; if you're new to functions and scopes, you may want to skip it for now.

If you haven't encountered this sort of thing before, you may wonder what this word *recursion* is. It simply means referring to (or, in our case, "calling") yourself. A humorous definition goes like this:

recursion \ri-'k&r-zh&n\ n: see recursion.

Recursive definitions (including recursive function definitions) include references to the term they are defining. Depending on the amount of experience you have with it, recursion can be either mind-boggling or quite straightforward. For a deeper understanding of it, you should probably buy yourself a good textbook on computer science, but playing around with the Python interpreter can certainly help.

In general, you don't want recursive definitions like the humorous one of the word *recursion*, because you won't get anywhere. You look up recursion, which again tells you to look up recursion, and so on. A similar function definition would be

```
def recursion():
    return recursion()
```

It is obvious that this doesn't *do* anything—it's just as silly as the mock dictionary definition. But what happens if you run it? You're welcome to try. You'll find that the program simply crashes (raises an exception) after a while. Theoretically, it should simply run forever. However, each time a function is called, it uses up a little memory, and after enough function calls have been made (before the previous calls have returned), there is no more room, and the program ends with the error message `maximum recursion depth exceeded`.

The sort of recursion you have in this function is called *infinite recursion* (just as a loop beginning with `while True` and containing no `break` or `return` statements is an *infinite loop*) because it never ends (in theory). What you want is a recursive function that does something useful. A useful recursive function usually consists of the following parts:

- A base case (for the smallest possible problem) when the function returns a value directly
- A *recursive case*, which contains one or more recursive calls on *smaller parts of the problem*

The point here is that by breaking the problem up into smaller pieces, the recursion can't go on forever because you always end up with the smallest possible problem, which is covered by the base case.

So you have a function calling itself. But how is that even possible? It's really not as strange as it might seem. As I said before, each time a function is called, a new namespace is created for that specific call. That means that when a function calls "itself," you are actually talking about two different functions (or, rather, the same function with two different namespaces). You might think of it as one creature of a certain species talking to another one of the same species.

Two Classics: Factorial and Power

In this section, we examine two classic recursive functions. First, let's say you want to compute the *factorial* of a number n. The factorial of n is defined as $n \times (n–1) \times (n–2) \times \ldots \times 1$. It's used in

many mathematical applications (for example, in calculating how many different ways there are of putting *n* people in a line). How do you calculate it? You could always use a loop:

```
def factorial(n):
    result = n
    for i in range(1,n):
        result *= i
    return result
```

This works and is a straightforward implementation. Basically, what it does is this: first, it sets the result to *n*; then, the result is multiplied by each number from 1 to *n*–1 in turn; finally, it returns the result. But you can do this differently if you like. The key is the mathematical definition of the factorial, which can be stated as follows:

- The factorial of 1 is 1.
- The factorial of a number *n* greater than 1 is the product of *n* and the factorial of *n*–1.

As you can see, this definition is exactly equivalent to the one given at the beginning of this section.

Now consider how you implement this definition as a function. It is actually pretty straightforward, once you understand the definition itself:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

This is a direct implementation of the definition. Just remember that the function call `factorial(n)` is a different entity from the call `factorial(n-1)`.

Let's consider another example. Assume you want to calculate powers, just like the built-in function `pow`, or the operator **. You can define the (integer) power of a number in several different ways, but let's start with a simple one: `power(x,n)` (`x` to the power of `n`) is the number `x` multiplied by itself `n-1` times (so that `x` is used as a factor `n` times). So `power(2,3)` is 2 multiplied with itself twice, or $2 \times 2 \times 2 = 8$.

This is easy to implement:

```
def power(x, n):
    result = 1
    for i in range(n):
        result *= x
    return result
```

This is a sweet and simple little function, but again you can change the definition to a recursive one:

- `power(x, 0)` is `1` for all numbers `x`.
- `power(x, n)` for `n > 0` is the product of `x` and `power(x, n-1)`.

Again, as you can see, this gives exactly the same result as in the simpler, iterative definition.

Understanding the definition is the hardest part—implementing it is easy:

```
def power(x, n):
    if n == 0:
        return 1
    else:
        return x * power(x, n-1)
```

Again, I have simply translated my definition from a slightly formal textual description into a programming language (Python).

■ **Tip** If a function or an algorithm is complex and difficult to understand, clearly defining it in your own words before actually implementing it can be very helpful. Programs in this sort of "almost-programming-language" are often referred to as **pseudocode**.

So what is the point of recursion? Can't you just use loops instead? The truth is yes, you can, and in most cases, it will probably be (at least slightly) more efficient. But in many cases, recursion can be more readable—sometimes *much* more readable—especially if one understands the recursive definition of a function. And even though you could conceivably avoid ever writing a recursive function, as a programmer you will most likely have to understand recursive algorithms and functions created by others, at the very least.

Another Classic: Binary Search

As a final example of recursion in practice, let's have a look at the algorithm called *binary search*.

You probably know of the game where you are supposed to guess what someone is thinking about by asking 20 yes-or-no questions. To make the most of your questions, you try to cut the number of possibilities in (more or less) half. For example, if you know the subject is a person, you might ask, "Are you thinking of a woman?" You don't start by asking, "Are you thinking of John Cleese?" unless you have a very strong hunch. A version of this game for those more numerically inclined is to guess a number. For example, your partner is thinking of a number between 1 and 100, and you have to guess which one it is. Of course, you could do it in a hundred guesses, but how many do you really need?

As it turns out, you need only seven questions. The first one is something like "Is the number greater than 50?" If it is, then you ask, "Is it greater than 75?" You keep halving the interval (splitting the difference) until you find the number. You can do this without much thought.

The same tactic can be used in many different contexts. One common problem is to find out whether a number is to be found in a (sorted) sequence, and even to find out where it is. Again, you follow the same procedure: "Is the number to the right of the middle of the sequence?" If it isn't, "Is it in the second quarter (to the right of the middle of the left half)?" and so on. You keep an upper and a lower limit to where the number *may* be, and keep splitting that interval in two with every question.

The point is that this algorithm lends itself naturally to a recursive definition and implementation. Let's review the definition first, to make sure we know what we're doing:

- If the upper and lower limits are the same, they both refer to the correct position of the number, so return it.
- Otherwise, find the middle of the interval (the average of the upper and lower bound), and find out if the number is in the right or left half. Keep searching in the proper half.

The key to the recursive case is that the numbers are sorted, so when you have found the middle element, you can just compare it to the number you're looking for. If your number is larger, then it must be to the right, and if it is smaller, it must be to the left. The recursive part is "Keep searching in the proper half," because the search will be performed in exactly the manner described in the definition. (Note that the search algorithm returns the position where the number *should* be—if it's not present in the sequence, this position will, naturally, be occupied by another number.)

You're now ready to implement a binary search:

```
def search(sequence, number, lower, upper):
    if lower == upper:
        assert number == sequence[upper]
        return upper
    else:
        middle = (lower + upper) // 2
        if number > sequence[middle]:
            return search(sequence, number, middle+1, upper)
        else:
            return search(sequence, number, lower, middle)
```

This does exactly what the definition said it should: if `lower == upper`, then return `upper`, which is the upper limit. Note that you assume (`assert`) that the number you are looking for (`number`) has actually been found (`number == sequence[upper]`). If you haven't reached your base case yet, you find the `middle`, check whether your number is to the left or right, and call `search` recursively with new limits. You could even make this easier to use by making the limit specifications optional. You simply add the following conditional to the beginning of the function definition:

```
def search(sequence, number, lower=0, upper=None):
    if upper is None: upper = len(sequence)-1
    ...
```

Now, if you don't supply the limits, they are set to the first and last positions of the sequence. Let's see if this works:

```
>>> seq = [34, 67, 8, 123, 4, 100, 95]
>>> seq.sort()
>>> seq
[4, 8, 34, 67, 95, 100, 123]
>>> search(seq, 34)
2
```

```
>>> search(seq, 100)
5
```

But why go to all this trouble? For one thing, you could simply use the list method `index`, and if you wanted to implement this yourself, you could just make a loop starting at the beginning and iterating along until you found the number.

Sure, using `index` is just fine. But using a simple loop may be a bit inefficient. Remember I said you needed seven questions to find one number (or position) among 100? And the loop obviously needs 100 questions in the worst-case scenario. "Big deal," you say. But if the list has 100,000,000,000,000,000,000,000,000,000,000,000 elements, and the same number of questions with a loop (perhaps a somewhat unrealistic size for a Python list), this sort of thing starts to matter. Binary search would then need only 117 questions. Pretty efficient, huh? [3]

■ **Tip** You can actually find a standard implementation of binary search in the `bisect` module.

THROWING FUNCTIONS AROUND

By now, you are probably used to using functions just like other objects (strings, number, sequences, and so on) by assigning them to variables, passing them as parameters, and returning them from other functions. Some programming languages (such as Scheme or Lisp) use functions in this way to accomplish almost everything. Even though you usually don't rely that heavily on functions in Python (you usually make your own kinds of objects—more about that in the next chapter), you *can*.

Python has a few functions that are useful for this sort of "functional programming": `map`, `filter`, and `reduce`.[4] (In Python 3.0, these are moved to the `functools` module.) The `map` and `filter` functions are not really all that useful in current versions of Python, and you should probably use list comprehensions instead. You can use `map` to pass all the elements of a sequence through a given function:

```
>>> map(str, range(10)) # Equivalent to [str(i) for i in range(10)]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

You use `filter` to filter out items based on a Boolean function:

```
>>> def func(x):
...     return x.isalnum()
...
>>> seq = ["foo", "x41", "?!", "***"]
>>> filter(func, seq)
['foo', 'x41']
```

3. In fact, with the estimated number of particles in the observable universe at 10^{87}, you would need only about 290 questions to discern between them!
4. There is also `apply`, but that was really only needed before we had the splicing discussed previously.

For this example, using a list comprehension would mean you didn't need to define the custom function:

```
>>> [x for x in seq if x.isalnum()]
['foo', 'x41']
```

Actually, there is a feature called *lambda expressions,*[5] which lets you define simple functions in-line (primarily used with `map`, `filter`, and `reduce`):

```
>>> filter(lambda x: x.isalnum(), seq)
['foo', 'x41']
```

Isn't the list comprehension more readable, though?

The `reduce` function cannot easily be replaced by list comprehensions, but you probably won't need its functionality all that often (if ever). It combines the first two elements of a sequence with a given function, combines the result with the third element, and so on until the entire sequence has been processed and a single result remains. For example, if you wanted to sum all the numbers of a sequence, you could use `reduce` with `lambda x, y: x+y` (still using the same numbers):[6]

```
>>> numbers = [72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100, 33]
>>> reduce(lambda x, y: x+y, numbers)
1161
```

Of course, here you could just as well have used the built-in function `sum`.

A Quick Summary

In this chapter, you've learned several things about abstraction in general, and functions in particular:

Abstraction: Abstraction is the art of hiding unnecessary details. You can make your program more abstract by defining functions that handle the details.

Function definition: Functions are defined with the `def` statement. They are blocks of statements that receive values (parameters) from the "outside world" and may return one or more values as the result of their computation.

Parameters: Functions receive what they need to know in the form of parameters—variables that are set when the function is called. There are two types of parameters in Python: positional parameters and keyword parameters. Parameters can be made optional by giving them default values.

5. The name "lambda" comes from the Greek letter, which is used in mathematics to indicate an anonymous function.
6. Actually, instead of this lambda function, you could import the function `add` from the `operator` module, which has a function for each of the built-in operators. Using functions from the `operator` module is always more efficient than using your own functions.

Scopes: Variables are stored in scopes (also called *namespaces*). There are two main scopes in Python: the global scope and the local scope. Scopes may be nested.

Recursion: A function can call itself—and if it does, it's called *recursion*. Everything you can do with recursion can also be done by loops, but sometimes a recursive function is more readable.

Functional programming: Python has some facilities for programming in a functional style. Among these are lambda expressions and the `map`, `filter`, and `reduce` functions.

New Functions in This Chapter

Function	Description
`map(func, seq [, seq, ...])`	Applies the function to all the elements in the sequences
`filter(func, seq)`	Returns a list of those elements for which the function is true
`reduce(func, seq [, initial])`	Equivalent to `func(func(func(seq[0], seq[1]), seq[2]), ...)`
`sum(seq)`	Returns the sum of all the elements of `seq`
`apply(func[, args[, kwargs]])`	Calls the function, optionally supplying argument

What Now?

The next chapter takes abstractions to another level, through *object-oriented programming*. You learn how to make your own types (or *classes*) of objects to use alongside those provided by Python (such as strings, lists, and dictionaries), and you learn how this enables you to write better programs. Once you've worked your way through the next chapter, you'll be able to write some really *big* programs without getting lost in the source code.