Chapter 14

Functions and Variables

“What happens in functions should stay in functions.”

Perhaps the biggest obstacle for computing is complexity. Not everything that one can imagine is computable can actually be done. Unlike science fiction movies where the human protagonist commands the computer to calculate the odds, crack a security code, or find the identify of a suspect from slivers of evidence, the reality is that computing has its limits. The main theoretical limit is complexity: some things are inherently difficult to compute.

On a practical scale, complexity is a major impediment to the construction of software. A typical commercial aircraft has dozens if not hundreds of embedded computing devices; a telephone can have over a million statements from several computing languages. To help manage the complexity, there are several general themes. One theme mentioned in Chapter 8 is the DRY principle. Another is the division of large software projects into chunks of manageable size, with the hope that the complexity of each chunk is relatively low.

Functions help manage complexity in several ways. First, the job of each function can be limited so that it does not need complicated logic. Second, even if what happens inside a function is based on some complex idea (possibly even some mathematical theory), the caller of a function can be unaware of how the function does its job, so interior complexity is hidden and users of the function are shielded from the complexity. And third, the variables used by the caller of a function can be protected from any of the function’s statements, which sets up a boundary between variables of one software component (the calling program) and the variables used inside the function.

Scope

The notion of scope was introduced, informally, in Chapter 13. Here we first revisit the idea of scope for a script, rather than a function, to see the simple
cases. The first case for scope is trivial. To show it, there is a script below with some line numbers on the left (they are not part of the script, but just for reference in the discussion that follows).

```
1  X = 20
2  Y = int(X*1.5)
3  Z = (X+Y)/2
4  X += Y
5  print("value =",Z)
6  del Y, Z
7  print(X)
```

When the script runs, initially no variables (other than Python’s built-in names) are defined. By the same token, when the script finishes, no variable is defined, because the memory of a Python program is transient, unlike the memory of data in a disk file, on CD, or on a flash drive. (In later chapters we shall see how to write to files, so that results can be permanent rather than transient.) Variable X becomes defined in line 1 of the script, and remains defined up to the end of the script. At line 4 we see that X is re-defined, or assigned a new value. Variable Y is defined in line 2, and remains defined up to line 6, where it is deleted. The example illustrates that it matters where, within a script, that we look to decide whether a variable is defined or not. And, because the order of evaluation of lines in a script flows from first line to the last line, it also matters when we consider the question of variable being defined or not.

The scope of a variable refers to the places and times that it is defined. When we say that a variable is “defined” we mean that it has a specific, unambiguous value. The value is fixed at the moment of assignment. In a script, the value of a variable is always its most recent assignment. Thus above, the value of X at lines 5, 6, and 7 is 50 — because the assignment on line 4 re-defines X to be the evaluation of X+Y at the moment that line 4 runs. To emphasize this behavior, consider this little example:

```
1  A = "one"
2  B = "two " + A
3  A = "three"
4  print(B)
```

When it runs, the output is two one. The assignment to A in line 3 does not change the value of B, it only re-defines A. The value of B was fixed in its most recent assignment, at line 2.

Scope is not always trivial in a script. What happens when the following runs?
The scope of \( R \) is the entire script, from line 1 to line 5. The first line defines \( R \) as what some function \texttt{mystery()} returns (suppose we do not know exactly what this returns). Now, can you say what is the scope of variable \( C \)? Possibly, it is lines 3-5; but it may happen that \texttt{mystery()} does not return a string, in which case \( C \) will never be defined. Thus, it can be that the scope of a variable in a Python script can be unpredictable; it can be \textit{dynamic} and depend on the values used and the conditional logic in the script.

Variables in Functions

Variables can freely be used inside functions. The scope of variables assigned within functions is limited to the body of the function and for the duration of a function application.

```python
def myfunction(S):
    x = S.split()
    y = ''.join(x)
    return len(x) - len(y)
def test():
    x = "one more time"
    y = "larger than a breadbox"
    print(myfunction(x))
    print(myfunction(y))
    print(x)
    print(y)
test()
```

The output of this script is

```
2
3
one more time
larger than a breadbox
```

Seeing this output demonstrates that applying \texttt{myfunction} did not redefine variable \texttt{x} assigned in the body of \texttt{test}. Somehow, when \texttt{x} is assigned within \texttt{myfunction}, the value of variables in \texttt{test} is unaffected. The technical reason is
that the scope of a variable is limited to the function body where it is assigned. The way Python achieves this is roughly described by the following rules (later we talk about some exceptions to the rules).

1. To aid the explanation, consider a function application `gfun(5,"it")` with the header of `gfun` being

   ```python
   def gfun(a,b):
   ```

   When `gfun(5,"it")` is evaluated, Python first creates a new variable name dictionary, solely for this function application. At this point, there are thus two variable dictionaries, the one that already existed for the function’s caller (which could be another function or perhaps a script), and the new one, the `gfun`-dictionary. The new `gfun`-dictionary is empty.

2. Python next puts two items into the new dictionary, for parameter `a` and for parameter `b`. The values associated with these two items are obtained by evaluating the arguments of the function application, in this case, `5` and "it". This step is called binding arguments to parameter names.

3. As Python evaluates statements of `gfun`, any assignment or expression evaluation uses the `gfun`-dictionary to find variables, create new variables, change variable values, or delete them.

4. When `gfun` returns, by a return statement, or if Python finishes evaluation all statements in `gfun` (perhaps because it has no return statement), then Python takes the value to be returned (or `None`), sets that aside, and destroys the `gfun`-dictionary.

5. Finally, Python substitutes whatever return value was obtained from the previous step into the expression where the `gfun(5,"it")` appears.

Steps 1–5 reveal Python’s mechanism for avoiding confusion over variable names during function evaluation. During the evaluation of `gfun(5,"it")`, there can safely be two different variables with the same name; but they have different scope, and therefore they are in different dictionaries. Python uses only one variable dictionary at a time, so there is no confusion. The only time the two dictionaries connect is when parameters bind to arguments, which may require getting values from the caller’s dictionary. Each function application operates in a “private world” of variables, oblivious to the outside. A consequence of step 4, which destroys the `gfun`-directory, is that a function cannot “remember” how many times it has been called, nor can it save some information between function applications.

Few programmers think about the work Python does to evaluate function application. That’s probably a good thing. The example at the beginning of the chapter and the description of steps 1–5 mention the simple case of a single function application. In fact, Python can evaluate expressions such as `g(f(h(k(True))))`
which involves four functions, \( f, g, h, k \). Technically, this means Python will need four dictionaries, one for the scope of each function. Each time Python evaluates function application, it *pushes down* the current set of dictionaries. The term “push down” is standard computing jargon, but a better way to describe this might be “put in background”, referring to how applications launched on an a software desktop open windows that cover up existing windows. The older windows move to the background; they will reappear once the current application finishes. To evaluate \( g(f(h(k(True)))) \), Python will push down \( g \), then \( f \), and then \( h \) before it finishes evaluating \( k(True) \). In case you’re curious, the opposite of “push down” is *pop up*: when function evaluation finishes, the previous dictionary of variable names and values that existed before evaluating the function “pops up” and becomes active.

An important consequence of pushing down and managing multiple dictionaries is that there can be *more than one variable with the same name*. However, at any moment during the run of a script of evaluation of a function, the question of scope is determined by the currently active dictionary of variable names and values.

### Global Scope

There are two exceptions to steps ①–⑤ above, which a “loopholes” in the scope restrictions on variables. One of these is an object reference mechanism, which is common to many programming languages, and is a topic of Chapter 15. The other is a way to bypass scoping rules for certain variables. The *global* statement allows functions to say that one or more variables created outside of the function can be used in expressions and assigned. When used in expressions, such *global variables* have their previous values, before the function was called. If a global variable is assigned within the function, the result of the assignment persists even after the function returns.

```python
def Mfun(val):
    global acount
    r = val/2
    acount += 1
    return r*r
def top():
    global acount
    v = max( [ Mfun(i) for i in range(10,18) ] )
    print "max is", v
    print "there were", acount, "function calls"
    top()
```

Copyright © 2010 Ted R. Herman, all rights reserved.
Running this script produces the output

```
max is 64
there were 8 function calls
```

In the example, variable `acount` belongs to two dictionaries, one for `top` and one for `Mfun`. The first time that `Mfun` is called, namely `Mfun(10)`, the value of `acount` is 0; the second time is `Mfun(11)`, and `acount` equals 1.

Superficially it may seem that the `global` statement simplifies writing functions. It is a way to have some memory for how many times a function is called. Using `global` is a way that many functions could share information through variables they have in common to their name dictionaries. However, by the same token, there is some danger to using `global`. For one thing, the caller of a function that has a `global` statement may need to understand more about that function. Suppose `Mfun` is defined as above, and the following interactive Python session tries to call `Mfun(0)`

```
>>> Mfun(0)
line 4, in Mfun
    acount += 1
NameError: global name 'acount' is not defined
```

The error occurs when `Mfun` tries to evaluate `acount += 1`, which is logically the same as `acount = acount + 1`. In order to evaluate this, Python first needs to get the current value of `acount`. But for the interactive session, variable `acount` was never assigned a value, so Python is unable to find a pre-existing `acount` variable with a value, and prints the error message. Because the use of `global` means that the callers of functions need to understand more about what happens inside the function, i.e., that it depends on having the values of the global variables, this programming technique is generally considered to be poor practice. While there may be some circumstances that motivate `global`, nine times out of ten there is another, better way to achieve the same ends. The user of `global` turns out to complicate software rather than simplify it.

Default Global

Even if you never use the `global` statement, it’s worth knowing about. Here’s why: Python automatically uses a “semi-global” way of evaluating variables inside of functions. To explain Python’s behavior, step ③ needs to be revised a bit:

(Step ③’) When Python starts function application, the statements of the function are examined to get the names of all variables that
might be assigned. If some variable might be assigned (whether it is assigned or not may depend on conditional logic), but is not in a global statement, Python considers that variable name to be local. If a variable is not local by this criterion, then by default it is global.

Some program language experts are of the opinion that Python’s “default global” behavior is a bug in the language design. Others like the way that functions refer to variables of the caller even though they are not parameters. In some ways, global variables, be they declared by global or be they default globals, are in effect secret parameters to a function — it isn’t enough to look at the function header, you have to read through the function body to figure out dependency on global variables. Secrecy may introduce more complexity, which is a danger to good software construction. To see why the default global behavior can be a danger, it’s good to see a few examples.

```python
def Rfun(b):
    r = 2
    return b*r
...
>>> r = 7
>>> print Rfun(0), Rfun(8)
0 16
```

This example uses no default global variables and has no surprises. The first statement establishes an initial value for r, which creates an entry in the Rfun’s local variable dictionary. One common mistake programmers make is forgetting to put some statement in a function; this can also happen because of a keystroke error in some editors, where a delete line might happen by pressing some key.

```python
def Rfun(b):
    return b*r
...
>>> print Rfun(0), Rfun(8)
NameError: global name 'r' is not defined
```

This is good, Python actually detected that something is wrong in the definition of Rfun. The error message indicates that the local variable dictionary does not have the name r in it when it evaluated b*r, and Python could not find the name in any pushed down dictionary either.

```python
def Rfun(b):
    return b*r
...
>>> r = 7
>>> print Rfun(0), Rfun(8)
0 56
```
This example confirms that Python uses the default global strategy of finding variable names and their values. The local dictionary takes priority, but will use a global approach when a variable is not assigned in the function body. But is this what the programmer intended? Just because Python did not complain does not mean the result is correct.

```python
def Tfun(b):
    if not b:
        r = 5
        return r
...
>>> r = 7
>>> print Tfun(True)
UnboundLocalError: local variable 'r' referenced before assignment
```

In the last example, we observe that Python classified `r` as a local variable, not a global one, and a bug was detected.

**Terminology Review**

Jargon used in this chapter includes: scope, global variable, push-down, pop-up.

**Exercise**

Use Python’s built-in `dir` function to see what is defined when a script runs. For instance, try to run something like this script:

```python
def foo(M,g):
    t = g in M
    print("foo name dictionary is:",dir())
    print("foo values (M,g,t) are:",M,g,t)
    return t
def moo(t,M):
    print("moo name dictionary is:",dir())
    print("moo values (t,M) are:",t,M)
    return foo(t,M) or foo(t,M.upper())
print("Test")
A,b = "if you find an error", "fi"
print("script name dictionary is:",dir())
print(moo(A,b))
```

Can you reason about the output of the program and determine what’s happening?