F: Functions

Gareth McCaughan

Credits

For the LaTeX source of this sheet, and for more information on LiveWires and on this course, see the LiveWires web site at
http://www.livewires.org.uk/python/

Introduction

A *function* is just a piece of Python program with a name. Functions are very important; this sheet tells you all about functions in Python.

What's a function?

A function is like a recipe. It's a set of instructions that you can refer to by name. Using functions saves thinking, in the same sort of way as using recipes does; if you're planning a meal it's much easier if you can just say "We'll have spaghetti bolognaise and apple crumble" rather than having to plan every detail of the meal "from first principles".

Functions are useful when you want to do the same thing several times: you just define a function once (like writing down a recipe) and then use it several times (like making several cakes with the same recipe).

They're even more useful when you want to do *almost* the same thing several times. You can write a function so that some bits of what it does aren't completely specified, and when the function is used the details get filled in. This is like having a recipe that will make several different kinds of pie, I suppose.

Functions are also useful even if you're only going to use each function once. They don't make your program any shorter then, but they do make it easier to understand. (You'll probably just have to take my word for this, until you've written lots of programs and discovered it for yourself!)

Defining and using functions

To define a simple function (that does the same thing every time you use it), do something like this:

```
def some_name_or_other():
  do something
  do something else
  blah blah blah
```

And to use it, you just say

```
some_name_or_other()
```

That will do basically the same as copying everything inside the function definition would have done.

Local variables

Suppose you say

```
def say_boo_twice():
  boo = 'Boo!!!'
  print boo, boo

boo = 'boo hoo'
say_boo_twice()
```

Then after doing that, boo will still be a name for the string 'boo hoo' – *not* for 'Boo!!!'. (So this is something that's different from what would happen if you just copied the definition of the function instead of "calling" the function by its name.)

There's a good reason for this. It means that functions can't have terribly unexpected effects. Someone reading the second half of that little program wouldn't have any reason to suspect that say_boo_twice() would change the value of boo. So, making changes to variables "local" to the function makes programs easier to understand: you don't need to know all about what's inside say_boo_twice() in order to understand a piece of program that uses it.

The variable boo inside say_boo_twice is called a "local variable".

Global variables

Actually, I lied to you when I said that calling a function can't have unexpected effects on variables' values. It can, but you have to ask for this to happen. If we change the program above by adding one extra line

```
def say_boo_twice():
  global boo                          This is the extra line
  boo = 'Boo!!!'
  print boo, boo

boo = 'boo hoo'
say_boo_twice()
```

then the value of boo *will* change. Saying global boo means: "The variable boo is not a local variable." ("Global" is the opposite of "local".)

You shouldn't need to do this very often, especially because of a useful rule: If a function uses a variable *but never changes its value*, then the variable is assumed to be global. So your functions can *use* variables from the rest of your program without the trouble of writing global; it's only if you want to change those variables that you need to say they're global. And, usually, you shouldn't need to do that.

Functions with arguments

The most interesting functions have "arguments". An argument is a piece of information you give when using a function that makes a difference to what the function does. If you have a recipe that tells you how to make any number of pancakes ("allow a pint of milk for every 12 pancakes", etc) then the number of pancakes is an *argument* to the recipe (though cookery books don't usually put it that way!).

Here's how you define a function with arguments.

```
def print_two_things(x,y):
  print 'x is', x
  print 'y is', y
```

You can probably guess that if you say `print_two_things(1,'boing!')` then the computer will say

```
x is 1
y is boing!
```

(The arguments to a function are really just a special kind of local variable: inside `print_two_things`, x and y are local variables whose values are given when you say `print_two_things(1,'boing!')` or whatever.)

Returning values

Some functions don't just *do* things; they also produce results. For instance, you can write a function that takes two arguments and has, as its result, the sum of those two arguments. (There's no particular reason why you'd want to do that; it just makes a simple example.)

Here's how you do that.

```
def add(a,b):
  return a+b
```

(The magic word `return` means "the following is the result of the function".) So, you could use this function like so:

```
print '3 plus 4 is', add(3,4)
```

which would, unsurprisingly, print 3 plus 4 is 7.

Optional arguments

Sometimes the arguments to a function are *usually* the same, but occasionally you want to do something different. You can save some typing by using "optional arguments". Here's how to do that.

```
def do_something(x, y=999):
  print x,y
```

You can now use this in three ways.

1. As an ordinary function with two arguments. `do_something(5,6)` does just the same as if the definition had just begun with `def do_something(x,y):`.

2. As a function with *one* argument. If you don't say what value y should get, it will be set to 999 for the duration of the function. So, `do_something(1)` is the same as `do_something(1,999)`.

3. As a function with one argument, and a "keyword argument" which you can call by name. In other words, you can say `do_something(8, y=123)`. Obviously this isn't very useful, but if `do_something` had had *several* optional arguments this lets you give special values to some of them but not others.

Weird stuff

There are some slightly strange things you can do with functions. For instance, as far as Python is concerned, a function is just another object, no more unusual than a number or a string. So you can use one function as an argument to another. For

instance, the built-in function (i.e., a function Python provides for you without you having to define it) called `map` works like this:

```
>>> def thrice(x):
...     return 3*x
...                          Just hit Enter here.
>>> map(thrice, [1,2,3,100])
[3, 6, 9, 300]
```

You might find a use for this some day.

Why bother?

It's possible to write quite complicated programs without bothering with functions at all. What does using functions gain you?

- *Your programs will be clearer.* It's easier to understand a program when it's divided up into manageable portions, and functions are a useful way of doing that.

- *You won't have to think so hard.* If your program is divided up sensibly into functions, you can understand what each bit of it does without having to remember the details of every other bit. When you see `update_high_scores()` you'll know what that does, without the effort of reading through the code that actually updates the high scores, so it will be quicker to understand the bit of the program that uses `update_high_scores()`.

- *You'll save typing.* If you have to do the same thing (or even roughly the same thing) several times, put it in a function. Then you only have to explain how to do it once.

- *Your programs will be more adaptable.* If you find that you need to change something about your program, it's much easier to do that if it's neatly packaged up into functions. Otherwise you'll have nightmares wondering whether you really caught *every* place in the program that deals with the player's score and adjusted it for the new scoring rules.

- *You'll be able to re-use more of what you write.* With a bit of luck, you'll find that some functions you write are useful for several things, so that you can use the same function (or a slight variant) in several programs. That saves you the effort of having to write it several times!

You probably won't see the point of lots of the things on this list until you've actually written (or read!) some complicated programs. But once you've done that, you'll discover that functions are essential for keeping large programs under control.