Lesson 6: Functions and Input/Output

In this lesson we will learn to write a function both inside a Python file and as an external module. We will also learn to write and read data to and from a separate file.

1 Functions

Sometimes we need to repeat a section of code from one part of a program to another, or we might need to repeat a section of code in another program. One way to handle this is to simply copy and paste the section of code. Often, a better solution is to write a Python *function* that can be called in multiple places in a program, or called from another program.

A Python function is a small bit of code, with a name, that we can use repeatedly. A function can be a simple mathematical calculation like `sin()` or `sqrt()`, or something more complicated. The syntax for defining a function is

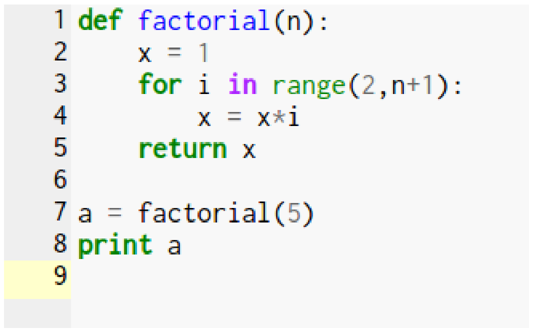
```
def functionname():
    code to execute
    return values
```

The first line contains the function name, *functionname*, which must follow the same naming rules as variables. The parentheses after *functionname* contains a list of variables that the function will use. We say that these variables are passed from the main code to the function. The first line ends with a colon. The remaining lines must be indented. They include the code that the function executes, followed by a `return` statement. Typically, a mathematical function will return some numerical result of the calculation. In some cases it might be useful to return `True` or `False`. The return statement can be omitted if it is not needed.

Once the function is defined, it can be used throughout the rest of the program. The code in Figure 1 defines and uses a function `factorial` to compute 5!. Figure 2 shows a code that computes 5! without using the `return` statement. Most often, your code will be more clear and logical if you use `return` statements with your functions.

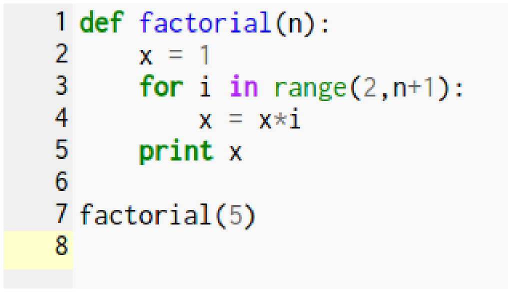
Functions are useful for breaking up code into more understandable pieces. You can define a function at any point in the code, as long as it's before the function is called. However, it is usually best to define all functions at the top of your code, after any import statements.

The variables that are passed to a function (like the variable n in Figures 1 and 2) exist only within the function code. We don't need to worry about using the same variable name within the function as we do within the main code. Thus, in Figure 1, the main code passes



```
1 def factorial(n):
2     x = 1
3     for i in range(2,n+1):
4         x = x*i
5     return x
6
7 a = factorial(5)
8 print a
9
```

Figure 1: Calculation of 5! using a function `factorial`. The function is defined in lines 1–5. The main code is lines 7–8.



```
def factorial(n):
    x = 1
    for i in range(2,n+1):
        x = x*i
    print x

factorial(5)
```

Figure 2: Calculation of 5! using a function `factorial`, without a `return` statement.

the number 5 to the function, which makes the assignment $n = 5$. Even if the main code defines a variable $n = 17$, the command `a = factorial(5)` still assigns 5 to the variable n within the function.

Exercise 1: Type in the code from Figure 1. Add the line $n = 17$ to the main code, before the line `a = factorial(5)`, and modify the print statement to say `print a, n`. Do you understand the output?

We can define functions with default values included for one or more of its arguments. Default values are defined like this:

$$\mathtt{def}\ \mathit{functionname}(\mathtt{a}, \mathtt{b}, \mathtt{c} = \mathtt{3}, \ldots) :$$

Any variables that have a default value must be put at the end of the list, after those variables (like `a` and `b`) that do not have default values.

Exercise 2: Write a function that will raise a number to a power. If only one number is given to the function, it should raise that number to the power 2.

Functions that we want to use in several different programs can be written into a Python program by themselves. In this way, we can build up our own library of useful functions. To use this library, we need to import it, just like we import `numpy` or `pylab`.

Homework: Create a new file in Canopy. In this new file define two functions, the first being the factorial function from Figure 1 and the second being your "raise to a power" function. Save this file as `lastname_functions.py`. Next, write a program that imports your file and uses both functions to compute $(N!)^r$, given integers N and r.

2 More About Functions

Consider the following code:

```
def myfun(aye,kay):
        out = aye + kay + sea
        return out

aye = 2.0
bee = 3.0
sea = 4.0
print myfun(aye,bee)
print myfun(7.0,bee)
print aye
```

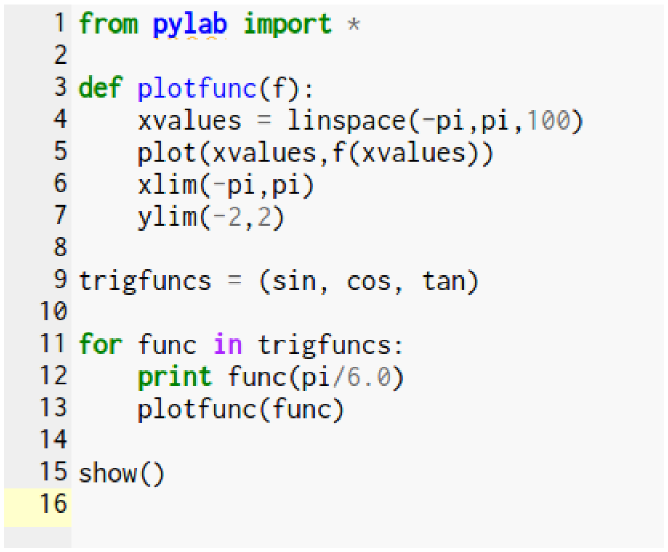
The outputs from the three print statements are `9.0`, `14.0` and `2.0`. This example illustrates several features of functions.

1. The value of the variable `sea` is defined in the main code. It can be used in the function `myfun`, even though it is not passed to `myfun` as an argument.
2. In the first print statement, the main code passes the values `aye = 2.0` and `bee = 3.0` to `myfun`, where these values are called `aye` and `kay`. The names used in the function definition can be the same (like `aye`) or different (like `bee` → `kay`) from the names used in the main code.
3. In the second print statement, `myfun` uses the value 7.0 for its first argument, which is called `aye` in the function definition. This does not change the value of `aye` in the main code.

Keep in mind that when a function is called, it is passed the *values* of the variables in the argument list from the main code. Those values are passed to the function definition, which can assign it's own variable names to these values. Whatever changes might happen to the variables within the function routine will not affect the values of any variables in the main code.

Exercise 3: Write the Python code above and verify the output. Insert the line `aye = 27.0` just after the command `out = aye + kay + sea` in the function definition. Does this change the output? What happens if you place `aye = 27.0` before `out = aye + kay + sea` command?

Functions are treated just like variables by Python. You can store functions inside variables, place them in lists, *etc.* You can use functions inside of other functions. An example is shown in Figure 3.



```python
from pylab import *

def plotfunc(f):
    xvalues = linspace(-pi,pi,100)
    plot(xvalues,f(xvalues))
    xlim(-pi,pi)
    ylim(-2,2)

trigfuncs = (sin, cos, tan)

for func in trigfuncs:
    print func(pi/6.0)
    plotfunc(func)

show()

```

Figure 3: Example code showing a function that plots functions.

Exercise 4: Write the program in Figure 3 and run it. Do you understand how it works?

3 Input and Output

Data (or text) can be inserted into a Python program using the `raw_input` command. This can be cumbersome if the program requires a large amount of data. It is often preferable to have the program read the data from a file. It is also common for a code to produce a large amount of data that we need to write to a file for later analysis.

Reading data from a file and writing data to a file can be a bit tricky, because Python interprets each character, including spaces and carriage returns, quite literally. For scientific purposes, we usually want to read and write arrays of real numbers. Fortunately, `numpy` includes easy–to–use functions for reading and writing such data arrays. Two of these functions are called `loadtxt()` and `savetxt()`.

To load the contents of the file *filename* into the array variable `a`, simply use:

$$\texttt{a} = \texttt{loadtxt}(filename)$$

To save the array `a` to a file, just write:

$$\texttt{savetxt}(filename, \texttt{a})$$

There are optional arguments to `loadtxt()` and `savetxt()` that you should explore on your own. By default, `loadtxt()` and `savetxt()` assume all numbers are real (type *float*).

Exercise 5: Create two files. One should contain a single column of numbers. The other should contain a single row of numbers separated by one or more spaces. Read the files into arrays `a` and `b`, respectively, using the `loadtxt` command. Display the contents of these arrays to the screen using the `print` command. Are the results as you expect?

Create another file consisting of several rows, each row containing several numbers. Read the file into an array `c`. What does `print c` produce? This is an example of a two–dimensional array. You can reference individual elements using the syntax `c[i,j]`. Which element is `c[0,1]`? Which element is `c[1,0]`?

The `savetxt` command will place the elements of a one–dimensional array into a file with a single column. We can save multiple arrays into multiple columns of a single file. For example, let's say our code computes the height of a projectile as a function of time. Let `t` denote the array of times and `y` denote the array of heights. Then the command

$$\texttt{savetxt}(\mathit{filename}, \texttt{zip}(\texttt{t}, \texttt{y}))$$

Will create the file *filename* with `t` in the first column and `y` in the second column.

To read a data file that consists of multiple columns, and load each column into it's own array, you can use

$$\texttt{A}, \texttt{B} = \texttt{loadtxt}(\mathit{filename}, \texttt{usecols} = (0, 1), \texttt{unpack} = \texttt{True})$$

This command extracts the first column of *filename* (which has index `0`) and places it into the array `A`. The second column (which has index `1`) is placed into the array `B`.

Exercise 6: Consider again the multi–column file that you created in the previous exercise. Read the columns of this file into one–dimensional arrays and verify the results using `print`.

Homework: Write a program to create an $n \times n$ multiplication table and write it to a file. The program should tell the user that a square multiplication table will be created, and ask the user for the size n.

4 More About Input/Output

We sometimes need to read or write more general information to a file, information other than real number arrays. Python includes "primative" commands that allow the user detailed control over input and output.

To work with any file we first need to open it:

$$\mathit{filehandle} = \texttt{open}(\mathit{filename}, \mathit{mode})$$

The `open()` function requires two arguments, the name of the file (*filename*) and the *mode*. The *mode* can be one of three options:

1. 'r' opens the file for reading only. The file cannot be changed, only read.
2. 'w' opens the file for writing. If the file does not already exist, it will be created. If the file does already exist, it will be over–written, destroying the original contents.
3. 'a' opens the file for appending. The output will be added to the end of the existing file without destroying what was originally there.

Once the file is opened, we can read the contents with various methods:

- $string = filehandle.$`read`$()$ This method will read the entire file into a single string. The string could end up being incredibly huge and cumbersome.
- $line = filehandle.$`readline`$()$ This method will read a single line of the file and return it as a single string. To read multiple lines in the file, this method needs to be invoked for each line.
- $lines = filehandle.$`readlines`$()$ This method will read the entire file into a list of strings. Each element of the list is a single string containing a single line from the file. Thus $lines[0]$ is a string containing the first line in the file.

All of these methods read the information in the file as strings. Python does not care what the contents of the file mean, just what the characters in the file are. So, if the data are numbers, you will need to convert them from strings into the appropriate numeric types. Once you have finished with the file it's a good idea to close the file:

$$filehandle.\texttt{close}()$$

Python will automatically close the file when the program ends, However, closing it ourselves is good practice and is preferred. It ensures that the write buffer is flushed and everything is actually written to the hard drive.

Exercise 7: Write a program to read in the data file `Lesson6_Data.txt`. Experiment with all three methods.

Writing data is similar to reading data. First we open the file:

$$filehandle = \texttt{open}(filename, '\mathrm{w}')$$

Once the file is opened, we can write strings to it:

$$filehandle.\texttt{write}(string)$$

Writing data to a file is a very literal operation. The `write()` command will send the exact contents of the string to the file. So, it will become important to include a \`newline` character at the end of each line so that the `write()` command will not simply write everything to a single line. We can use the same string formatting commands that we learned earlier to control the size and number of decimal points of the numbers written.

Homework: Create a file with a single column that contains a list of five fruits. Write a program that reads the file and writes to the screen: `My favorite fruits are ...` (and lists the fruits). Your program should then modify the fruit file by adding a sixth fruit to the list.