



Introduction to programming

Reusing code with functions and modules

Defining and calling functions

In week 3 we looked at looping concepts and how they can be used to repeat a block of code. In programming terms a function is similar. A block of code is contained within a function and the function can be repeated a number of times. A function is intended to perform a specific action and they should generally be kept as brief as possible, depending on the task.

Functions help to make your code more manageable by reducing redundant code and making your programs more readable. As functions can be used all through your program there is no need to repeat the same blocks of code, which makes it easier when changes are needed.

Defining a function in Python is fairly simple; use the **def** keyword followed by the name of function and finally parentheses **()** like this:

```
def say_hello():  
    print "Hello!"
```

In the example above a function called **say_hello** is created which contains a single statement which prints **Hello!** As is the same with if statements and loops, the block of code that is contained in the function needs to be indented.

If you were to save the above function in a file and run it in Python you would get no output. Try saving the function or creating your own and running it to see the result. In order to use the function it needs to be **called**.

To call a function, in order to run the block of code it contains, simply type the name you gave the function followed by parentheses.

```
say_hello()
```



The parentheses are important as Python will just return a reference to the function as apposed to actually running it if you do this. For example, creating and calling the function in IDLE:

```
>>> def say_hello():
        print "Hello!"

>>> say_hello
<function say_hello at 0x02B0D7B0>
>>> say_hello()
Hello!
>>> |
```

You can see that without the parentheses IDLE just returns the reference of the function and where it is stored in memory. Running it with the parentheses results in the code being run.

Passing arguments to functions

The **say_hello()** function doesn't provide much extra value to our programs but it is a good example of making your code more maintainable. Imagine if you used the **say_hello()** function in multiple places in your program and then wanted to change the message that is displayed. All you would need to do is update the print statement from within the **say_hello()** function.

However, functions can provide much more use when they're provided with **arguments** to produce a unique result each time it is run. An argument is a variable that is passed to a function, the value of which can then be used within the indented block of code inside the function.

```
def say_hello(name):
    print "Hello", name, "!"

say_hello("James"):
```

The **say_hello()** function now takes an argument called **name** which creates a variable inside the function which can then be used. The print statement has been updated to print out a message to including the **name** variable to create a personalised message.

Now when the function is called you can provide a string which will be added to the print statement. Try creating the above function and running it with your name to see the result.



Once a function has arguments specified in its definition these become mandatory. For example, if you were to run the **say_hello()** function as before without providing a name:

```
>>> say_hello("James")
Hello James !
>>> say_hello()

Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    say_hello()
TypeError: say_hello() takes exactly 1 argument (0 given)
>>>
```

You can see that an error is generated when the function is not given a name argument. As the error explains, the **say_hello()** function has been defined to take 1 argument but none were provided. The simplest solution to this problem is to provide a default value to the argument when the function is defined. This is done inside the parentheses where the arguments are specified.

```
def say_hello(name="Stranger"):
    print "Hello", name, "!"
```

Now, with the **say_hello()** function if the name argument is not provided when it is called the function will take the default value **Stranger** and assign it to name.

```
>>> say_hello("James")
Hello James !
>>> say_hello()
Hello Stranger !
>>> |
```

Returning values

The **say_hello()** function is a good way to demonstrate how a function can be created and then used. However most functions will perform some sort of calculation or operation and provide an answer or result back to the program at the point where the function was called. This mechanism is known as **returning a value**.

A simple although unnecessary example would be a function that adds two numbers together and returns the result.

```
def add(n1, n2):
    result = n1 + n2
    return result

print add(3,6)
```



This could be achieved in a far more elegant way in the code by simply adding the two numbers together on the print statement line but the function demonstrates the idea of returning values. In fact, creating the temporary **result** variable is not needed and the sum can just be put on the same line as the return statement:

```
def add(n1, n2):  
    return n1 + n2  
  
print add(3,6)
```

Try creating your own functions that either multiply, divide or perform other calculations and return the results.

Taking lists of arguments

Sometimes functions need to be a bit more flexible in terms of the amount of arguments they accept. Our **add()** function worked fine but it will only accept two numbers and it would be nice if it could add up a list of numbers. Most programming languages will allow you to provide a list of arguments and store them in one variable, which you can then loop over to perform the required action. In order to do this in Python, you will need to put an asterisk * before a variable name in the function definition.

```
def add(*arguments):  
    total = 0  
    for argument in arguments:  
        total += argument  
    return total  
  
print add(3,6,9,12)
```

The word **arguments** can be changed for any other word that you choose e.g. the function would work exactly the same if we changed **arguments** to **numbers**.

The **add()** function now will accept any amount of numbers, add them together and then return that result.

Recursive functions

Recursion occurs in a program when a certain function calls itself from within its own body. For example with the simple **say_hello()** function above recursion might look like this:

```
def say_hello(name="Stranger"):  
    print "Hello", name, "!"  
    say_hello(name)
```

Do not try to run the above code as if you are following what is happening. You will notice that there is an infinite loop in the above code. When **say_hello()** is called the print statement is executed and then **say_hello()** is called again which starts the whole process again.



Recursive functions might seem like a problem for creating infinite loops but they can be really useful when the same purpose a function provides is needed within that same function. Take for example, working out the [Fibonacci sequence](#) for a set of numbers:

```
def fib(n):
    if n==0 or n==1:
        return n
    return fib(n-1) + fib(n-2)

for i in range(10):
    print fib(i)
```

The Fibonacci numbers are calculated by adding the last two numbers in the sequence together which ends up with: **0, 1, 1, 2, 3, 5, 8, 13, 21, 34**

Here the **fib()** function calls itself from inside it's own function body. This make it easy to determine what the Fibonacci number is by recursively calculating the last two numbers in the sequence and then returning the result. Again, there is an issue where there could be an infinite loop within the code however this is prevented by providing an exit or end condition. The function first checks whether the number passed to the function **n** is equal to **0** or **1** and if it is, the function returns that number rather than calling the function again. The function at some point will have **n** set to **0** or **1** so it will always be able to exit and not get stuck in a loop.

What are modules

A module is a collection of code that is stored in a file. The code can include variables, functions and objects. In the same way that our functions define a block of code that can be easily repeated, by calling the function name we can use the available functions in a module. For example, from the functions that were created in the last section we could have a module file which looks like this:

```
def say_hello(name="Stranger"):
    print "Hello", name, "!"

def add(*arguments):
    total = 0
    for argument in arguments:
        total += argument
    return total

def fib(n):
    if n==0 or n==1:
        return n
    return fib(n-1) + fib(n-2)
```

This could be a file called **useful_module.py** and this would be referred to as the **module file**. Notice how there isn't any calls to the functions in the module, just the definitions of the functions.



Because there are only definitions of functions in the above file it won't actually do anything on its own. You can try running the file and see – there won't be any output. In order to use the functions that have been defined in the module file, they need to be **imported**.

```
import useful_module

useful_module.say_hello("James")
print "Did you know that 2 + 2 = ", useful_module.add(2,2)
print "And here are the first ten fibonacci numbers:"
for i in range(10):
    print useful_module.fib(i)
```

You can import a module file in Python by using the **import** function. You will see this a lot in other programming languages too with various different keywords such as **include** and **use**. You will see from the above code that every time you want to use a function that is defined, you need to prefix the function name with the module name e.g. **useful_module.say_hello()**. This is to tell Python what module the **say_hello()** function has been defined in – there may be functions with the same name so using the module name avoids confusion.

Try creating a module with a few functions in it, and then import it into a separate file to use those functions.

In order to tidy up your code a little, you can give the imported module file a name so you don't need to type out the full name each time you want to use a function.

```
import useful_module as tools
```







Now whenever you want to use one of the functions from the **useful_module** you can just use the **tools** prefix:

```
tools.say_hello("James")
print "Did you know that 2 + 2 = ", tools.add(2,2)
print "And here are the first ten fibonacci numbers:"
for i in range(10):
    print tools.fib(i)
```



Where to put modules

If you create a module file and store it in a particular directory then it will be automatically available for importing into any programs that you create in that same directory.

Name	Date modified	Type	Size
 fib.py	02/02/2016 11:14	Python File	1 KB
 return.py	09/02/2016 09:42	Python File	1 KB
 sums.py	09/02/2016 11:04	Python File	1 KB
 test_module.py	09/02/2016 14:22	Python File	1 KB
 useful_module.py	09/02/2016 14:18	Python File	1 KB
 useful_module.pyc	09/02/2016 14:19	Compiled Python ...	1 KB

You will also notice that when you import a module in Python, it creates a compiled version which is used in the importing process. In the above screenshot you can see the **useful_module.pyc** file has been created.

However if you move the module file or try to create a program in another directory that uses the module then you will get an error when trying to import it.

```
Traceback (most recent call last):
  File "C:\Users\james.bubb\Google Drive\learndirect\SD Lvl 4\python_course\Week
  6\test_module.py", line 1, in <module>
    import useful_module as tools
ImportError: No module named useful_module
```

One solution in this example is to just copy the **useful_module.py** file to the directory you're creating the program in. However that presents a problem that when you want to make changes to the module file, you need to replicate those changes across all of the copies you have made, which can become a big headache.

If you want to make a module available to all Python programs you create, at least on the same computer, then there is a special folder to store them in. You can find out the correct folder to store your modules in by running a command at the command prompt:

```
python -m site --user-site
```

On my computer the response I got was:

C:\Users\james.bubb\AppData\Roaming\Python\Python27\site-packages

So this is the folder I should store my module files in if I want them to be available to other programs.