Functions – Part I

Functions are blocks of code that can be used again and again, can accept arguments and return values after their execution is complete. Unlike languages like C and C++, functions in Python do not follow a *call by value* or a *call by reference* mechanism; instead they employ what is commonly called *call by sharing.* This essentially means that argument objects are shared between the caller and the called function, and mutations made to mutable objects by the called function are visible to the caller, while changes made to immutable objects are not reflected by the caller objects.

A very loose explanation of the above is that mutable objects are passed by reference, and immutable objects are passed by value.

A key difference between functions and methods is that methods are functions that belong to a class. For example, the functions used while manipulating strings are methods, as they belong to the `str` class.

We have already come across several inbuilt functions and methods, so in this tutorial, we will mainly deal with user-defined functions.

Defining a function

In Python, functions are defined using the `def` keyword, as –

```
def foo():

    #do something
```

Functions can receive arguments as well, and return values –

```
def square(x):

    return x*x
```

Notice that unlike C or other languages, the data type of the parameters doesn't have to be specified.

Functions can return multiple values as a tuple, which can then be unpacked –

```
def foo(a, b):

    return a*b, a/b

c, d = foo(2, 3)
```

`c` and `d` get the values 6 and 0.66 respectively.

In Python, there are no function prototypes. The only rule is that a function should be defined before its use.

```
def bar():

    foo()

def foo():

    print('Hello there')

bar()
```

This is allowed, as by the time `bar()` is called, `foo()` has already been defined. However, the following code will not work –

```
def bar():

    foo()

bar()

def foo():

    print('Hello there')
```

Recursion

The process by which a function calls itself directly or indirectly is called recursion, and a function that does so is called a recursive function. They are used to solve problems that can be expressed as smaller sub-units of itself. The simplest example is a function to compute the factorial of a number.

```
def factorial(n):

    if n == 1 or n == 0:

        return 1

    else:

        return n * factorial(n-1)
```

Remember that every recursive problem can be solved by an iterative method, and vice-versa. Recursive functions will have greater space requirements, and the time requirements are also higher due to the number of function calls and return overheads.

Lambda Functions

Lambda functions are anonymous functions; they are defined without a name and are generally throwaway functions – they are just needed where they have been created. The syntax for a lambda function is quite simple -

```
lambda argument_list: expression
```

The argument list consists of a list of comma separated operators, and the expression is generally a mathematical expression involving these arguments. The lambda expression can be stored in another variable to give it a name. For example –

```
sum = lambda x, y: x+y

sum(3, 4)

# Output : 7
```

This may seem redundant; we might as well define a proper function named sum to compute the same operation. However, lambda functions are generally helpful when making calls to higher-order functions – functions that take in other functions as arguments. They are extensively used along with the functions `map()`, `reduce()` and `filter()`.

The `map()` function

`map()` is a function that takes two arguments – a function name, and a list sequence. The syntax for it is-

```
result = map(func, seq)
```

`map()` applies the function to each element in the list, and return an iterator containing the result of each computation (iterators are any Python data type that can be used in a `for…in` loop).

Here's an example –

```
def cubed(x):

    return x**3

a = [1, 2, 3, 4]

res = map(cubed, a)

print(list(res))

# Output : [1, 8, 27, 64]
```

`map()` can be used with multiple lists, and these lists do not have to be of the same length. The function will simply map values until the shortest list has been consumed. The mapping starts with the 0th index of all three lists, then the 1st index, and so on.

```
>>> a = [1, 2, 3]

>>> b = [17, 12, 11, 10]

>>> c = [-1, -4, 5, 9]

>>>

>>> list(map(lambda x, y, z : 2.5*x + 2*y - z, a, b, c))
```

```
[37.5, 33.0, 24.5]
```

In the above example, the parameters x, y and z get their values from a, b, and c respectively.

This function can also be used in conjunction with a lambda function –

```
>>> C = [39.2, 36.5, 37.3, 38, 37.8]

>>> F = list(map(lambda x: (float(9)/5)*x + 32, C))

>>> print(F)

[102.56, 97.7, 99.14, 100.4, 100.03999999999999]

>>> C = list(map(lambda x: (float(5)/9)*(x-32), F))

>>> print(C)

[39.2, 36.5, 37.300000000000004, 38.00000000000001, 37.8]
```

The `filter()` function

The function `filter()` is used to filter results – it filters out all the elements of a sequence for which a function returns `True`. The syntax for this function is –

```
filter(func, seq)
```

An item will be produced by the iterator result of `filter(function, sequence)` if item is included in the sequence "sequence" and if `function(item)` returns `True`.
In other words, the function `filter(f,l)` needs a function `f` as its first argument. `f` must return a Boolean value, i.e. either `True` or `False`. This function will be applied to every element of the list `l`. Only if `f` returns `True` will the element be produced by the iterator, which is the return value of `filter(function, sequence)`.

```
>>> fibonacci = [0,1,1,2,3,5,8,13,21,34,55]

>>> odd_numbers = list(filter(lambda x: x % 2, fibonacci))

>>> print(odd_numbers)

[1, 1, 3, 5, 13, 21, 55]

>>> even_numbers = list(filter(lambda x: x % 2 == 0, fibonacci))

>>> print(even_numbers)

[0, 2, 8, 34]

>>>
```

```
>>>

>>> # or alternatively:

...

>>> even_numbers = list(filter(lambda x: x % 2 -1, fibonacci))

>>> print(even_numbers)

[0, 2, 8, 34]
```

The `reduce()` function

The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.

If `seq = [ s1, s2, s3, ... , sn ]`, calling `reduce(func, seq)` works like this:

1. At first the first two elements of seq will be applied to `func`, i.e. `func(s1, s2)` The list on which reduce() works looks now like this: `[ func(s1, s2), s3, ... , sn ]`.
2. In the next step `func` will be applied on the previous result and the third element of the list, i.e. `func(func(s1, s2), s3)`.
3. The list looks like this now: `[ func(func(s1, s2),s3), ... , sn ]`.
4. Continue like this until just one element is left and return this element as the result of `reduce()`.

Here’s an example –

```
>>> from functools import reduce

>>> reduce(lambda x, y: x+y, range(1,101))

5050
```

The above example computes the sum of all numbers from 1 to 100.

Remember – the module `functools` must be imported to use this function.

As an exercise, try rewriting the function to compute the factorial of a number using `reduce()`.