Nested Functions

A function which is defined inside another function is known as inner function or nested function. Nested functions are able to access variables of the enclosing scope. Inner functions are used so that they can be protected from everything happening outside the function. This process is also known as Encapsulation.

Example

In [1]:

```
name = 'This is a global name'

def greet():
    # Enclosing function
    name = 'Sammy'

    def hello():
        #inner function
        print('Hello '+name)

    hello()

greet()
```

```
Hello Sammy
```

Note how Sammy was used, because the hello() function was enclosed inside of the greet function!

Scope of variable in nested function

The location where we can find a variable and also access it , if is required is called the scope of a variable. It is known how to access a global variable inside a function.

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are local to the function. This is called the scope of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Example:

In [2]:

```
x = 50

def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)

func(x)
print('x is still', x)
```

```
x is 50
Changed local x to 2
x is still 50
```

The first time that we print the value of the name x with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value 2 to x. The name x is local to our function. So, when we change the value of x in the function, the x defined in the main block remains unaffected.

With the last print statement, we display the value of x as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

The `global` statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is global. We do this using the `global` statement. It is impossible to assign a value to a variable defined outside a function without the global statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outermost block. Example:

In [3]:

```
x = 50

def func():
    global x
    print('This function is now using the global x!')
    print('Because of global x is: ', x)
    x = 2
    print('Run func(), changed global x to', x)

print('Before calling func(), x is: ', x)
func()
print('Value of x (outside of func()) is: ', x)
```

```
Before calling func(), x is:  50
This function is now using the global x!
Because of global x is:  50
Run func(), changed global x to 2
Value of x (outside of func()) is:  2
```

The `global` statement is used to declare that x is a global variable - hence, when we assign a value to x inside the function, that change is reflected when we use the value of x in the main block.

You can specify more than one global variable using the same global statement e.g. `global x, y, z`.