Lecture 10 – Functions

A function is a block of instructions that performs an action and, once defined, can be reused.

Functions make code more modular, allowing you to use the same code over and over again.

Python has several built-in functions that you may be familiar with, including:

- **print()** which will print an object to the terminal
- **int()** which will convert a string or number data type to an integer data type
- **len()** which returns the length of an object

Function names include parentheses and may include parameters.

Defining Functions without parameters

Let’s start with turning the classic “Hello, World!” program into a function.

A function is **defined** by using the **def keyword, followed by a name of your choosing, followed by a set of parentheses which hold any parameters the function will take (they can be empty), and ending with a colon.**

For Example

```
def hello() :      #   initial statement for creating a function
```

Next , **setup instructions** for what the function does. In this case, printing Hello, World! to the console.

For Example

```
def hello() :      #   initial statement for creating a function
       print("Hello, World!")
```

The function is now fully defined.

Finally: Outside of the defined function block, **call the function** with hello():

For Example

```
#   initial statement for creating a function

def hello() :
       print("Hello, World!")   #  Body of the function


hello()     # A statement that is calling the function
```

When running this program, the following will be produced.

Hello, World!
>>>

Functions can be more complicated than the hello() function that was defined above.

For example, within the function block one can use :

- **for / while loops**
- **conditional statements**
- **input / print statements**

Example : Write a function that iterate over the letters in the name string and displays every character.

```
# Define function names()
def names():    # Set up name variable with input
    name = str(input('Enter University Name:  '))

# Iterate over name
    for letter in name :
        print(letter)

# Calling  the function
names()
```

Sample Run

```
Enter University Name:  Texas State
T
e
x
a
s

S
t
a
t
e
>>>
```

Example : Write a function that displays a letter grade for a score that is entered from the keyboard

```
def letterGrade():
    score = int(input("Enter Score   "))
    if score >= 90:
        letter = 'A'
    elif score >= 80:
        letter = 'B'
    elif score >= 70:
        letter = 'C'
    elif score >= 60:
        letter = 'D'
    else:
        letter = 'F'
    print("\nYour Letter Grade is    " , letter)

letterGrade()
```

Sample Run

```
Enter Score   50

Your Letter Grade is     F
>>>
```

Poem Function Exercise

Write a program, **poem.py**, that defines a function that prints a short poem or song verse. Give a meaningful name to the function. Using a for loop or a while loop have the program end by calling the function five times, so the poem or verse is repeated five times.

```
def friend():
  print("Friend in need is a friend indeed ")


print("Using For loop\n")
for i in range( 0 , 5 ):
  friend()

number = 5
i = 0

print("\n\nUsing While Loop\n")

while ( i < number ):
  friend()
  i=i+1
```

Defining Functions with Parameters

A parameter is a named entity in a function definition, specifying an argument that the function can accept.

Writ a small program that takes in parameters x, y, and z. Create a function that adds the parameters together in different configurations. The sums of these will be printed by the function. Call the function and pass numbers into the function.

Example

```
def add_numbers(x, y, z):
  a = x + y
  b = x + z
  c = y + z
  print(a, b, c)


add_numbers(1, 2, 3)
```

Sample Run

```
3 4 5
>>>
```

In the previous example :

1 in for the x parameter,
2 in for the y parameter,
and 3 in for the z parameter.

These values correspond with each parameter in the order they are given.

The program is doing the following math based on the values that was passed to the parameters:

a = 1 + 2
b = 1 + 3
c = 2 + 3

The function also prints a, b, and c, and based on the math above

a is equal to 3,
b is equal to 4,
and c is equal to 5.

Keyword Arguments

In addition to calling parameters in order, a keyword argument in a function call can be used, in which the caller **identifies the arguments by the parameter name**.

For example: create a function that will display profile information for a user.

Parameters will be passed to the function in the form of username (intended as a string), and followers (intended as an integer).

```
def profile_info(username, followers):
   print("Username: " + username)
   print("Followers: " + str(followers))
```

Within the function definition statement, username and followers are contained in the parentheses of the profile_info() function.

The block of the function prints out information about the user as strings, making use of the two parameters.

The function is called, and parameters are assign to it.

```
profile_info("sammyshark", 945)
```

Sample Run

Username: sammyshark
Followers: 945
>>>

Call function with keyword arguments

profile_info(username="AlexAnglerfish", followers=342)

Sample Run

Username: AlexAnglerfish
Followers: 342
>>>

The Complete Program

```
def profile_info(username, followers):
  print("Username: " + username)
  print("Followers: " + str(followers))

profile_info("sammyshark", 945)

profile_info(username="AlexAnglerfish", followers=342)
```

Sample Run

Username: sammyshark
Followers: 945
Username: AlexAnglerfish
Followers: 342
>>>

Modifying the order of the parameters

Using the same program with a different call:

Change order of parameters

profile_info(followers=820, username="cameroncatfish")

Example

```
def profile_info(username, followers):
  print("Username: " + username)
  print("Followers: " + str(followers))

profile_info(followers=820, username="cameroncatfish")
```

Sample Run

Username: cameroncatfish
Followers: 820
>>>

By using keyword arguments, it does not matter which order the parameters are passed into the function call.

Default Argument Values

default values for one or both of the parameters can also be provided.

For example: Create a default value for the followers parameter with a value of 1:

 # Use followers to be 1 if followers parameter was not passed

```
def profile_info(username, followers = 1):
    print("Username: " + username)
    print("Followers: " + str(followers))

profile_info(username="JOctopus")
profile_info(username="sammyshark", followers=945)
```

Sample Run

Username: JOctopus
Followers: 1
Username: sammyshark
Followers: 945
>>>

What happens if you enter the following :

```
profile_info(followers=945)
```

Functions That Return Values to Main Program

You can pass a parameter value into a function, and a function can also produce a value.

A function can produce a value with the **return statement**, which will exit a function and **optionally** pass an expression back to the caller. If you use a return statement with no arguments, the function will return None.

For example , create a program with a function that squares the parameter x and returns the variable y. A call to print the **result variable will be issued**, which is formed by running the square() function with 3 passed into it.

```
def square(x):
   y = x ** 2
   return y    #  use return instead of print.

result = square(3)     # the result of power will be stored in the variable
                       # result

print(result)

print("\n",square(4))     # the result of power will be returned
```

Sample run

9
16
>>>

The integer 9 is returned as output, which is the square of 3.

The integer 16 is returned as output, which is the square of 3.

Comment out the return statement in the program and observe what happens

```
def square(x):
    y = x ** 2
    # return y

result = square(3)
print(result)
```

Sample run

```
None
>>>
```

Without using the **return** statement here, the program cannot return a value (which is 9) so the **value defaults to None**

Another way of using return

```
def square(x):

   return   x ** 2


result = square(3)    # the result of power will be stored in the variable
                      # result

print(result)
```

Sample run

9
>>>

The integer 9 is returned as output, which is the square of 3.

As another example, modify the previous program that adds 3 integer numbers so that it return the values back to the main program

```
def add_numbers(x, y, z):
    a = x + y
    b = x + z
    c = y + z
    return a, b, c        # using return instead of print


sums = add_numbers(1, 2, 3)
print(sums)
```

Sample run

```
(3, 4, 5)
>>>
```

What is the output of the following program :

```
def loop_five():
    for x in range(0, 25):
        print(x)
        if x == 5:          # Stop function at x == 5
            return
    print("This line will not execute.")


loop_five()
```

What is the output of the above program if you replace return with Continue ?

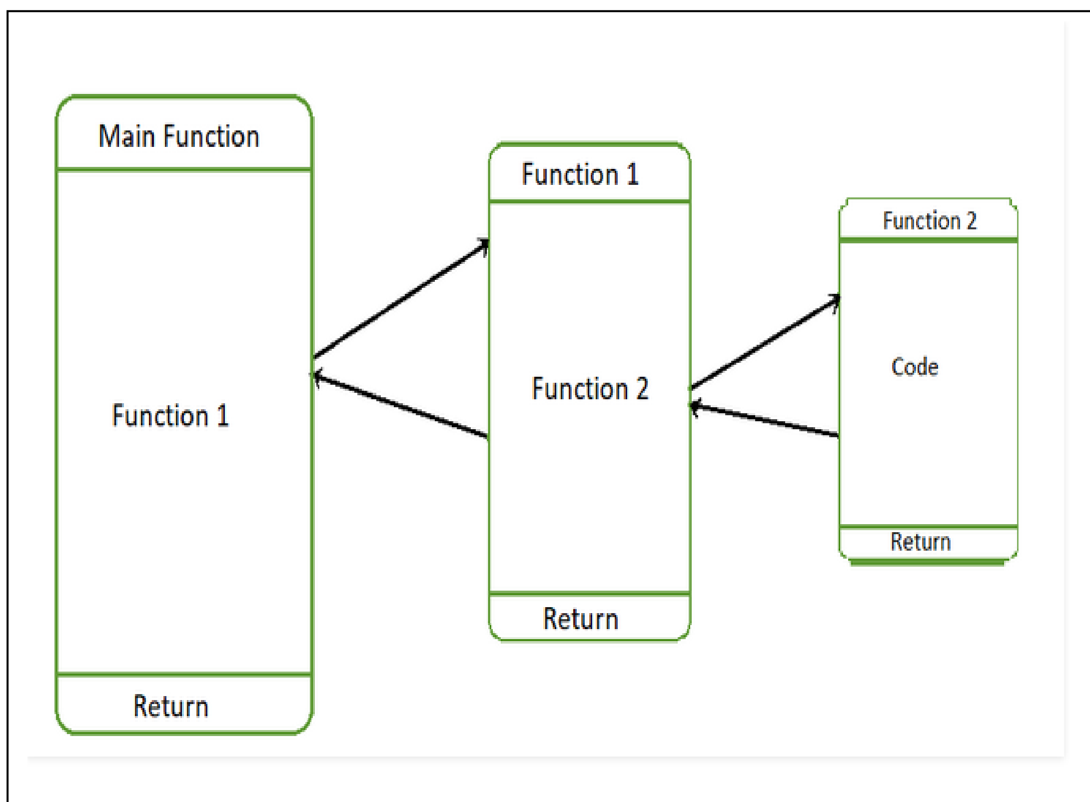
What is the output of the above program if you replace return with break ?

What is the output of the following program:

```
def function_1():
    print("Hello From My Function!")

def function_2(username, greeting):
  print("Hello" , ",", username , ", From My Function!, I wish you " , greeting)

def function_3(username, greeting):
    print("Hello, %s , From My Function!, I wish you %s"%(username,
greeting))

def function_4(a, b):
    return a ** b



function_1()

function_2("Lexi  Rob", "a great year!")

function_3("John Doe", "a great year!")

x = function_4(3,3)

x = function_4(2,3)


print(x)
```

Functions calling other functions

It is important to understand that each of the functions can be used and called from other functions. This is one of the most important ways of take a large problem and break it down into a group of smaller problems. This process of breaking a problem into smaller subproblems is called **functional decomposition**.



Here's a simple example of functional decomposition using two functions. The first function called **square** simply computes the square of a given number. The second function called **sum_of_squares** makes use of square to compute the sum of three numbers that have been squared.

```
def square(x):
    y = x * x
    return y

def sum_of_squares(x, y, z):
    a = square(x)
    b = square(y)
    c = square(z)

    return a + b + c

a = -5
b = 2
c = 10
result = sum_of_squares(a, b, c)
print(result)
```

Sample Run

```
129
>>>
```

What is the output of this program

```
def a():
	print('a() starts')
	b()
	d()
	print('a() returns')

def b():
	print('b() starts')
	c()
	print('b() returns')

def c():
	print('c() starts')
	print('c() returns')

def d():
	print('d() starts')
	print('d() returns')


a()
```

Local and Global Variables

In Python , If you define **a variable within a function block**, you'll only be able to use that variable within that function. It is called **local** variable.

If you would like to **use variables across functions** it may be better to declare the variable as a **global** variable.

Example of using Local Variable in Python :

```
def sum(x,y):
    sum = x + y
    return sum

print(sum(5, 10))
```

Sample run

```
15
>>>
```

The variables **x and y** will only work and used **inside the function sum()** and they don't exist outside of the function.

So trying to use local variable outside their scope, might through NameError. For example, If you place print(x) in the main program your will get the following error and will not work.

```
def sum(x,y):
    sum = x + y
    return sum

print(sum(5, 10))
```

Sample Run

15

```
def sum(x,y):
    sum = x + y
    return sum

print(x)   #  will not work and the program terminates
```

```
Traceback (most recent call last):
  File "C:\Users\HP\AppData\Local\Programs\Python\Python38-32\Program1.py", line 6, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

Example of using Global Variable in Python :

```
x = 99       # x , y are global
y = 17
def  fun(x):   #  x is local to function  fun
   y = 100
   #  y is local to function  fun
   #  which is diff from the Global  y

   print ( x, y )


fun(77)
print ( x, y )
```

Sample Run

```
77 100
99 17
>>>
```

Example of using Global Variable with the keyword global :

```
Z = 25
def func():
    global Z        #  indicates that using the global   Z
                    #  By using the keyword global
    print(Z)
    Z=20            #  value of Z is changed

func()
print(Z)
```

A calling func(), the global variable value Z is changed for **the entire program.**

Sample run

25

20

>>>

Combination of local and global variables and function parameters

```
def func(x, y):
    global a
    a = 45
    x,y = y,x     #   x , y , b , c  are local to func
    b = 33
    b = 17
    c = 100
    print(a,b,x,y)


# values are visible only in the main program
# x , y are different from the x , y that appears in func

a,b,x,y = 3,15,3,4
func(9,81)
print (a,b,x,y)
```

Sample Run

```
45  17  81  9
45  15  3  4
>>>
```

What is the output of the following program :

```
variable_1 = 20
variable_2 = 30

def local():

   global variable_1
   variable_2 = 70
   variable_1 = 80

local()
print("the value of global_variable_1 has become",  variable_1)
print("the value of global_variable_2 did not change" ,  variable_2)
```

What is the output of the following program :

```
scale = 10
def doscale(list):
   newlist = []
   for i in list:
      newlist.append(i / scale)
   return newlist

mylist = [1,2,3,4,5]
otherlist = doscale(mylist)
print (otherlist)
```

What is the output of this program

```
total = 100

def test():
    marks = 19
    print('Marks = ', marks)
    print('Total in test before func1  = ', total)
    func1()
    print('Total in test  after func1  = ', total)
    func2()
    print('Total in test  after func2  = ', total)

def func():
     global total
     if total > 10:
         total = 25
     print('Total in  func  = ', total)

def func1():
    global total
    total = 15
    func()

def func2():
     global total
     if total <= 30:
         total = 15

def main():
    print('Total in main  = ', total)
    test()

main()
```

*args : Functions with unknown number of arguments

In Python, the single-asterisk form of ***args** can be used as a parameter to send a non-keyworded variable-length argument list to functions.

For example , look at a typical function that uses two arguments:

```
def multiply(x, y):
        print (x * y)
```

In the code above :

- function with x and y as arguments is defined,
- and then when the function is called , numbers that correspond to x and y must be used.

In this case, we will pass the integer 5 in for x and the integer 4 in for y:

```
def multiply(x, y):
        print (x * y)

multiply(5, 4)
```

Sample run

```
20
>>>
```

What happens if you call the function as follows :

multiply(5, 4, 3)
or
multiply(5, 4, 3, 2)

The statements multiply(5, 4, 3) or multiply(5, 4, 3, 2) will produce the following errors :

TypeError: multiply() takes 2 positional arguments but 3 were given

TypeError: multiply() takes 2 positional arguments but 4 were given

Thus , if more arguments might be used later on , make the use of *args as the function parameter instead.

Example

```
def multiply(*args):
    z = 1
    for num in args:
        z *= num
    print(z)

multiply(4, 5)
multiply(10, 9)
multiply(2, 3, 4)
multiply(3, 5, 10, 6)
```

Sample Run

```
20
90
24
900
>>>
```

With *args you can create more flexible code that accepts a varied amount of non-keyworded arguments within the function.

Another example

Function which returns the length of the longest of a variable number of arguments. The function will accept a collection of strings as arguments, it will check them all, and return the maximum length of any of them.

```
def longlen(*strings):
    max = 0
    for s in strings:
        if len(s) > max:
            max = len(s)
    return max

print(longlen('apple','banana','cantaloupe','cherry'))
print(longlen('seven','six','five','four','three','two'))
```

Sample Run

```
10
5
>>>
```

In more complex situations, two kinds of special arguments (variable and single asterisks) can be used in the same function.

For example

```
def allargs(one,*args):
   print ('one = ' , one)
   print ('Unnamed arguments:')
   for a in args:
      print ('%s' % str(a))


allargs(12,'Dog','Cat','Horse','Tiger')
```

Sample run

```
one =  12
Unnamed arguments:
Dog
Cat
Horse
Tiger
>>>
```

Practice

1. Write a Python function to multiply all the numbers in a list.

2. Write a Python function to check whether a number is in a given range

3. Write a Python function that accepts a string and calculate the number of upper-case letters, lower case letters and digits. The function will return upper, lower, and digits to the caller. Values are displayed in the main program