06_Functions

April 13, 2020

1 Functions

Most of the times, In a program, the statements keep repeating and it will be a tedious job to execute the same statements again and again and will consume a lot of memory and is not efficient. Enter Functions.

This is the basic syntax of a function

def funcname(arg1, arg2,... argN):

```
''' Document String'''

statements

return <value>
```

Read the above syntax as, A function by name "funcname" is defined, which accepts arguements "arg1,arg2,....argN". The function is documented and it is "'Document String"'. The function after executing the statements returns a "value".

[1]:
```
print ("Hey Kumar!")
print ("Kumar, How do you do?")
```

```
Hey Kumar!
Kumar, How do you do?
```

Instead of writing the above two statements, every single time it can be replaced by defining a function which would do the job in just one line.

Defining a function firstfunc().

[2]:
```
# Function definition
def firstfunc():
    print ("Hey Kumar!")
    print ("Kumar, How do you do?")
```

[3]:
```
# Function calling
firstfunc()
```

```
Hey Kumar!
Kumar, How do you do?
```

firstfunc() every time just prints the message to a single person. We can make our function **firstfunc()** to accept arguements which will store the name and then prints respective to that accepted name. To do so, add a argument within the function as shown.

[4]:
```
def firstfunc(username):
    print ("Hey", username + '!')
    print (username + ',' ,"How do you do?")
```

[5]:
```
name1 = input('Please enter your name:')
```

```
Please enter your name:Kumar
```

The name "Kumar" is actually stored in name1. So we pass this variable to the function **firstfunc()** as the variable username because that is the variable that is defined for this function. i.e name1 is passed as username.

[6]:
```
firstfunc(name1) #function calling
```

```
Hey Kumar!
Kumar, How do you do?
```

Let us simplify this even further by defining another function **secondfunc()** which accepts the name and stores it inside a variable and then calls the **firstfunc()** from inside the function itself.

[7]:
```
def firstfunc(username):
    print ("Hey", username + '!')
    print (username + ',' ,"How do you do?")
```

[8]:
```
def secondfunc():
    name = input("Please enter your name : ")
    firstfunc(name) # calling fristfunc()
```

[9]:
```
secondfunc() #calling
```

```
Please enter your name : Kumar
Hey Kumar!
Kumar, How do you do?
```

1.0.1 Example 2

[10]:
```
def addition(a,b):
    print("sum=",a+b)
```

[11]:
```
x=5
y=6
```

[12]:
```
addition(x,y)
```

```
sum= 11
```

[13]:
```
addition(2,3)
```

```
sum= 5
```

[14]:
```
addition(2) # signature must be matched
```

```
␣
↪---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call␣
↪last)

    <ipython-input-14-931aa23213e7> in <module>
----> 1 addition(2)


    TypeError: addition() missing 1 required positional argument: 'b'
```

1.1 Return Statement

When the function results in some value and that value has to be stored in a variable or needs to be sent back or returned for further operation to the main algorithm, return statement is used.

[ ]:
```
def times(x,y):
    z = x*y
    return z
```

The above defined **times( )** function accepts two arguements and return the variable z which contains the result of the product of the two arguements

[ ]:
```
c = times(4,5)
print (c)
```

[ ]:
```
print (times(4,5))
```

The z value is stored in variable c and can be used for further operations.

Instead of declaring another variable, the entire statement itself can be used in the return statement as shown.

[ ]:
```
def times(x,y):
    '''This multiplies the two input arguments'''
    return x*y
```

[ ]:
```
c = times(4,5)
print (c)
```

[ ]:
```
times(4,5)
```

Since the **times( )** is now defined, we can document it as shown above. This document is returned whenever **times( )** function is called under **help( )** function.

[ ]:
```
help(times)
```

[ ]:
```
times?
```

Multiple variable can also be returned, But keep in mind the order.

[ ]:
```
eglist = [10,50,30,12,6,8,100]
```

[ ]:
```
def egfunc(eglist):
    highest = max(eglist)
    lowest = min(eglist)
    first = eglist[0]
    last = eglist[-1]
    return highest,lowest,first,last
```

If the function is just called without any variable for it to be assigned to, the result is returned inside a tuple. But if the variables are mentioned then the result is assigned to the variable in a particular order which is declared in the return statement.

[ ]:
```
egfunc(eglist) # calling
```

[ ]:
```
a,b,c,d = egfunc(eglist)
print (' a =',a,'\n b =',b,'\n c =',c,'\n d =',d)
```

1.2 Implicit arguments

When an argument of a function is common in majority of the cases or it is "implicit" this concept is used.

[ ]:
```
def implicitadd(x,y=3):
    return x+y
```

implicitadd( ) is a function accepts two arguments but most of the times the first argument needs to be added just by 3. Hence the second argument is assigned the value 3. Here the second argument is implicit.

Now if the second argument is not defined when calling the **implicitadd( )** function then it considered as 3.

[ ]:
```
implicitadd(4) # calling with 1 arg
```

But if the 2^{nd} argument is specified then this value overrides the implicit value assigned to the argument

[ ]:
```
implicitadd(4,4)
```

1.3 Any number of arguments

If the number of arguments that is to be accepted by a function is not known then a asterisk symbol is used before the argument.

[ ]:
```
def add_n(*args):
    reslist = [] # list is empty
    for i in args:
        reslist.append(i) # appending element i
    print (reslist)
    return sum(reslist)
```

The above function accepts any number of arguments, defines a list and appends all the arguments into that list and return the sum of all the arguments.

[ ]:
```
add_n(1,2,3,4,5)
```

[ ]:
```
add_n(1,2,3)
```

1.4 Lambda Functions

These are small functions which are not defined with any name and carry a single expression whose result is returned. Lambda functions comes very handy when operating with lists. These function are defined by the keyword **lambda** followed by the variables, a colon and the respective expression.

[ ]:
```
z = lambda x: x * x
```

[ ]:
```
print(z(8))
print(z(3))
```

[ ]:
```
# addition of two given no.s uding lambda func.
addition = lambda x,y:x+y
addition(2,4)
```

1.5 More Examples

1.5.1 Finding fibonacci series

[ ]:
```
def fib1(n):
    a,b = 0,1
    print(a)
    while(b<n):
        print(b)
        a,b=b,a+b
fib1(10) # fn call
```

[ ]:
```
def fib1(n):
    a,b = 0,1
    print(a)
    print(b)
    while(b<n):
        c=a+b
        print(c)
        a=b
        b=c
fib1(10) # fn call
```

[ ]:
```
# recursive function - which calls itself
def fibonacci(n):
    if(n <= 1):
        return n
    else:
        return(fibonacci(n-1) + fibonacci(n-2))

num = int(input("Enter number of terms:"))
print("Fibonacci sequence:")
for i in range(num):
    print (fibonacci(i))
```

[ ]:
```
# non recursive func.
def Factorial(num):
    fact = 1
    for i in range(1,num+1):
        fact = fact*i
    return fact
# calling func
Factorial(5)

# 1*2*3*4*5
```

[ ]:
```
# recursive func.
def Factorial(n):
    if n==1:
        return n
    else:
        return n*Factorial(n-1)

# calling func
Factorial(5)
```

[ ]:
```

```