3.4. Fruitful functions: return values, parameters, local and global scope, function composition, recursion

Fruitful Functions

Function that returns value are called as fruitful functions.The return statement is followed by an expression which is evaluated, its result is returned to the caller as the "fruit" of calling this function.

Input the value→fruitful function→return the result

len(variable) – which takes input as a string or a list and produce the length of string or a list as an output.

In a fruitful function the return statement includes a return value. This statement means: Return immediately from this function and use the following expression as a return value. The expression provided can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):

return 3.14159 * radius**2
```

On the other hand, temporary variables like temp often make debugging easier. Sometimes it is useful to have multiple return statements, one in each branch of a conditional.

We have already seen the built-in abs, now we see how to write our own:

```
def absolute_value(x):
if x < 0:
        return -x
else:
        return x
```

Since these return statements are in an alternative conditional, only one will be executed. As soon as one is executed, the function terminates without executing any subsequent statements. Another way to write the above function is to leave out the else and just follow the if condition by the second return statement.

```
def absolute_value(x):
if x < 0:
        return -x
return x
```

Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. The following version of absolute_value fails to do this:

```
def absolute_value(x):
 if x < 0:
        return -x
elif x > 0:
        return x
```

This version is not correct because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. In this case, the return value is a special value called **None**:

```
>>>print absolute value(0)
None
```

None is the unique value of a type called the NoneType:

```
>>>type(None)
```

All Python functions return None whenever they do not return another value.

Example:

Write a python program to find distance between two points:

```
import math
def distance(x1,y1,x2,y2):                    # Defining the Function Distance
    dx=x2-x1
    dy=y2-y1
    print("The value of dx is", dx)
    print("The value of dy is", dy)
    d= (dx**2 + dy**2)
    dist=math.sqrt(d)
    return dist

x1 = float(input("Enter the first Number: "))      #Getting inputs from  user
x2 = float(input("Enter the Second Number: "))
y1 = float(input("Enter the third number: "))
y2 = float(input("Enter the forth number: "))
print("The distance between two points are",distance(x1,x2,y1,y2))
                                                   #Calling the function distance
```

Output:

```
>>> Enter the first Number: 2
    Enter the Second Number: 4
    Enter the third number: 6
    Enter the forth number: 12
    The value of dx is 4.0
    The value of dy is 8.0
   The distance between two points are 8.94427190999916
>>>
```

Explanation for Example 2:

Function Name – 'distance()'

Function Definition – def distance(x1,y1,x2,y2)

Formal Parameters - x1, y1, x2, y2

Actual Parameter – dx, dy

Return Keyword – return the output value 'dist'

Function Calling – distance(x1,y1,x2,y2)

Parameter in fruitful function

A function in python

- Take input data,called parameter
- Perform computation
- Return result

```
def funct(param1,param2):
        statements
        return value
```

Once the function is defined,it can be called from main program or from another function.

Functioncall statement syntax

```
Result=function_name(param1,param2
```

Parameter is the input data that is sent from one function to another.The parameters are of two types

1.Formal parameter

- The parameter defined as part of the function definition.
- The actual parameter is received by the formal parameter.

2.Actual parameter

- The parameter is defined in the function call

Example:

```
def cube(x):
        return x*x*x                    #x is the formal parameter
a=input("Enter the number=")
```

```
b=cube(a)                                   #a is the actual parameter
print”cube of given number=”,b
```

Result:

Enter the number=2

Cube of given number=8

Scope and Lifetime of variables

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exits in the memory.

The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

Eg:

```
def my_func():
    x = 10
    print("Value inside function:",x)
x = 20
my_func()
print("Value outside function:",x)
```

Output:

```
Value inside function: 10
Value outside function: 20
```

Local Scope and Local Variables

A **local variable** is a variable that is only accessible from within a given function. Such variables are said to have **local scope** .

```
def func1():
    n = 10
    print('n in func1 = ', n)

def func2():
    n = 20
    print('n in func2 before call to func1 = ', n)
    func1()
    print('n in func2 after call to func1 = ', n)
```

```
>>> func2()
n in func2 before call to func1 =  20
n in func1 =  10
n in func2 after call to func1 =  20
```

Global Variables and Global Scope

A **global variable** is a variable that is defined outside of any function definition. Such variables are said to have **global scope** .

```
max = 100  # global variable

def func1(count):
  if count < max:
  .

def func2(count):
  for i in range(1, max):
  .
```

global variable max accessed

global variable max accessed

Variable max is defined outside func1 and func2 and therefore “global” to each.

Function Composition

We can call one function from within another. This ability is called composition.

As an example, we’ll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the **center point** is stored in the variables **xc** and **yc**, and the **perimeter** point is in **xp** and **yp**. The first step is to find the radius of the circle, which is the distance between the two points.

radius = distance(xc, yc, xp, yp)

The second step is to find the area of a circle with that radius and return it. Again we will use one of our earlier functions:

```
result = area(radius)
return result
```

Wrapping that up in a function, we get:

```
def area2(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius) return result
```

We called this function area2 to distinguish it from the area function defined earlier. There can only be one function with a given name within a given module. The temporary variables radius and result are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def area2(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

Example:

Write a python program to add three numbers by using function:

```
def addition(x,y,z):                                    #function 1
    add=x+y+z
    return add
def get():                                              #function 2
    a=int(input("Enter first number:"))
    b=int(input("Enter second number:"))
    c=int(input("Enter third number:"))
    print("The addition is:",addition(a,b,c))       #Composition function calling
get()                                                   #function calling
```

Output:

Enter first number:5

Enter second number:10

Enter third number:15

The addition is: 30

Recursion:

A Recursive function is the one which calls itself again and again to repeat the code. The recursive function does not check any condition. It executes like normal function definition and the particular function is called again and again

Syntax:

```
def function(parameter):
                #Body of function
```

Example-1:

Write a python program to find factorial of a number using Recursion:

(*Positive value of n ,then n! can be calculated as n!=(n-1)….2.1 it ncan be written as (n-1)!*

Hence n! is the product of n and (n-1)! n!=n.(n-1)!)

```
def fact(n):
        if(n<=1):
                return n
        else:
                return n*fact(n-1)
n=int(input("Enter a number:"))
print("The Factorial is", fact(n))
```

Output:

>>> Enter a number:5

The Factorial is 120

>>>

Explanation:

First Iteration - 5*fact(4)

Second Iteration - 5*4* fact(3)

Third Iteration - 5*4*3*fact(2)

Fourth Iteration - 5*4*3*2* fact(1)

Fifth Iteration - 5*4*3*2*1

Example-2:

Write a python program to find the sum of a 'n' natural number using Recursion:

```
def nat(n):
    if(n<=1):
        return n
    else:
        return n+nat(n-1)
n=int(input("Enter a number:"))
print("The Sum is", nat(n))
```

Output:

```
>>> Enter a number: 5
    The Sum is 15
```

Explanation:

First Iteration – 5+nat(4)

Second Iteration – 5+4+nat(3)

Third Iteration – 5+4+3+nat(2)

Fourth Iteration – 5+4+3+2+nat(1)

Fifth Iteration – 5+4+3+2+1

The Advantages of recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

The Disadvantages of recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.