Chapter 6

Functions

A function is a group of statements that are intended to perform a specific task and, once defined, can be reused. Thus when there are several tasks to be performed, the programmer will write (define or create) several functions to make code more modular, allowing you to use the same code over and over again. There are several built-in functions in Python to perform various tasks. For example:

- 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

Similar to these functions, a programmer can also create his own functions which are called 'user-defined' functions. The following are the advantage of functions:

- ✓ Functions are important in programming because they are used to process data, make calculations or perform any task which is required in the program many times.
- ✓ Once a function is written, it can be reused as and when required.
- ✓ It reduces code redundancy.
- ✓ Make program modular.
- ✓ Code maintenance will become easy because of functions.
- ✓ When there is an error in the program, the corresponding function can be modified without disturbing the other functions in the program. Thus the code debugging will become easy.
- ✓ The use of function in a program will reduce the length of the program.

Difference between a Function and a Method

A function can be written individually in a Python program. A function is called using its name. When a function is written inside a class, it became a 'method'. A method is called using one of the following ways:

```
objectname.methodname()
```

```
str.strip()
```

str.replace('is', 'was')

So, please remember that a function and method are same except their placement and the way they are called.

Defining 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.

Consider the syntax of function

```
def functionname(para1,para2...):
      """ function docstring"""
      statements
```

Example:

```
def hello():                #function without parameter
   """ display Hello, World! String """
    print("Hello, World!")


def sum(a, b):     # function with parameter
     """
      This function find the sum of two numbers
     """
     c = a+ b
     print(c )
```

Our function is now fully defined, but if we run the program at this point, nothing will happen since we didn't call the function.
So, outside of our defined function block, let's call the function with ***hello()***

```
def hello():
    print("Hello, World!")

hello()
```

While calling a function, we should pass the necessary value to the function in the parenthesis as:

```
def sum(a, b):     # function with parameter
```

```
    c = a+ b
    print(c )
sum(10,20)
sum(5.5, 10.5)   # called second time
```

Program 1 To repeat the character

```
def repeat():
   for i in range(10):
       print('* ' , end ='')
   print()

repeat()
repeat()
repeat()
```

Output:

```
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

Every time when we call repeat() function it will display the same output.

So far we have looked at functions with empty parentheses that do not take arguments, but we can define parameters in function definitions within their parentheses.

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

Program 2: A Python function to check whether a number is prime or not

```
def prime(n):
   """
   To check if the n is prime or not
   """
   x =1 # this will be 0 if not prime
   for i in range(2,n):
       if n% i==0:
           x =0
           break
       else:
            x=1
```

```
    return x
num = int(input('Enter a number: '))
result = prime(num)
if result ==1:
    print(num, 'is prime ')
else:
    print(num, 'is not prime')
```

Program 3: A Python function to calculate the factorial values of numbers.

```
# a function to calculate the factorial value
def fact(n):
    prod =1
    while n>1:
        prod *=n
        n-=1
    return prod

for i in range(1,7):
    print('Factorial of {} is {}'.format(i, fact(i)))
```

Output:

```
Factorial of 1 is 1
Factorial of 2 is 2
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
Factorial of 6 is 720
```

Returning Multiple Value from a Function

A function returns a single value in the programming language like C or Java. But in Python, a function can return multiple values. When a function calculates multiple results and wants to return the results. We can use return statement as:

```
return x, y, z
```

Here, three value which are in x,y,z are returned. These values are returned by the function as a tuple. Please remember a tuple is like a list that contains a group of elements. To receive these values, we can use three variable at the time of calling the function as:

x,y,z = function()

Here, the variable x,y,z are receiving the three values returned by the function. For example:

```
def sum_sub_mul(a,b):
    c = a+b
    d = a-b
    e = a*b
    return c,d,e
```

Since this function has three parameters, at the time of calling this function, we should pass three values, as:

x,y,z = sum_sub_mul(10,5)

Pass by Object Reference

In the language like C and Java, when we pass values to a function, we think about two ways:

- Pass by value or call by value
- Pass by reference or call by reference

Pass by value represents that a copy of the variable value is passed to the function and any modifications to that value will not reflect outside the function.

Pass by reference represents sending the reference or memory address of the variable to the function. The variable value is modified by the function through memory address and hence the modified value will reflect outside the function also.

Neither of these two concepts is applicable in Python. The values are sent to functions by means of object references. We know that everything is considered as an object in Python. All numbers are objects, string are objects, and data types like tuples, list and dictionaries are also object.

```
#passing an integer to a function
def modify(x)
   x =15
   print(x, id(x)

x=10
modify(x)
print(x, id(x)
```

Output:

15 1617805901
10 1617805016

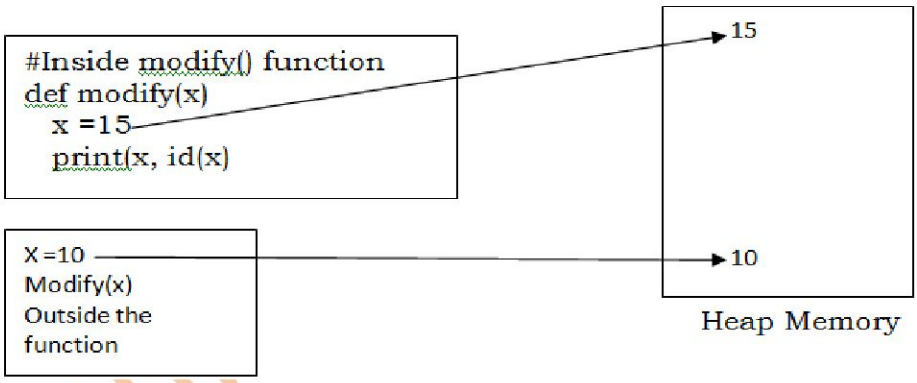
From the output, we can understand that the value of 'x' in the function is 15 and that is not available outside the function. When we call the modify() function and pass 'x' as:

```
modify(x)
```

we should remember that we are passing the object reference to the modify() function. The object is 10 and its reference name is 'x'. This is being passed to the modify() function. Inside the function, we are using:

x = 15

This means another object 15 is created in memory and that object is referenced by the name 'x'. The reason why another object is create in the memory is that the integer object are immutable (not modifiable). Consider the diagram in figure 6.1.



In Python integers, floats strings and tuples are immutable. That means their value cannot be modified. When we try to change their value, a new object is created with modified value. On the other hand, lists and dictionaries are mutable. That means we can change their value, the object gets modified and new object is not created. When we append a new element to the list, the same list is modified and hence the modified list is available outside the function also.

```
#passing a list to a function
def modify(lst)
   lst.append(9,id(lst))

lst = [1,2,3,4]
```

```
modify(lst)
print(lst, id(lst))
```

Output:

[1,2,3,4,9] 37762552
[1,2,3,4,9] 37762552

In the above program, the list 'lst' is the name or tag that represent the list object. Before calling the modify() function, the list contains 4 elements as:

```
lst = [1,2,3,4]
```

Please understand that the object here is [1,2,3,4] and its reference name is 'lst'. When we call the function as modify(lst), we are passing the object reference 'lst' to the function. Inside the function, we are appending a new element '9' to the list. Since lists are mutable, adding a new element to the same object is possible. Hence, append() function modified. So, when we display 'lst' inside the function, we can see:

```
[1,2,3,4,9]
```

After coming out of the function, we can see the modified list as the same object got modified.

```
[1,2,3,4,9]
```

This is shown in the diagram in figure 6.2

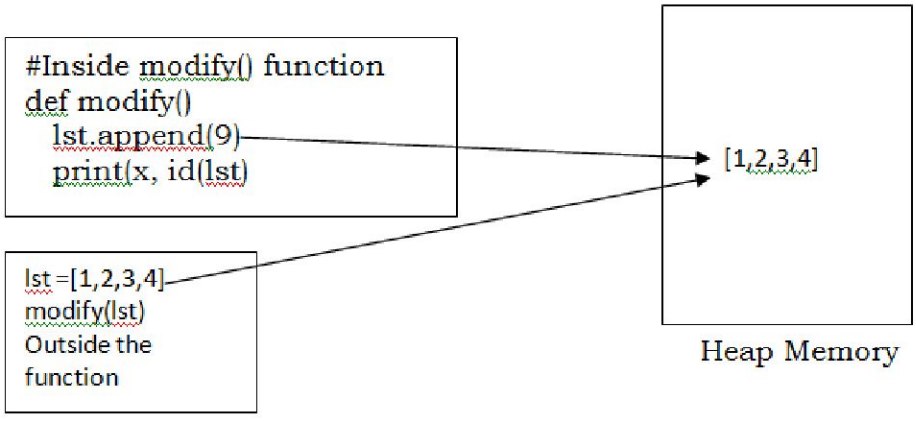
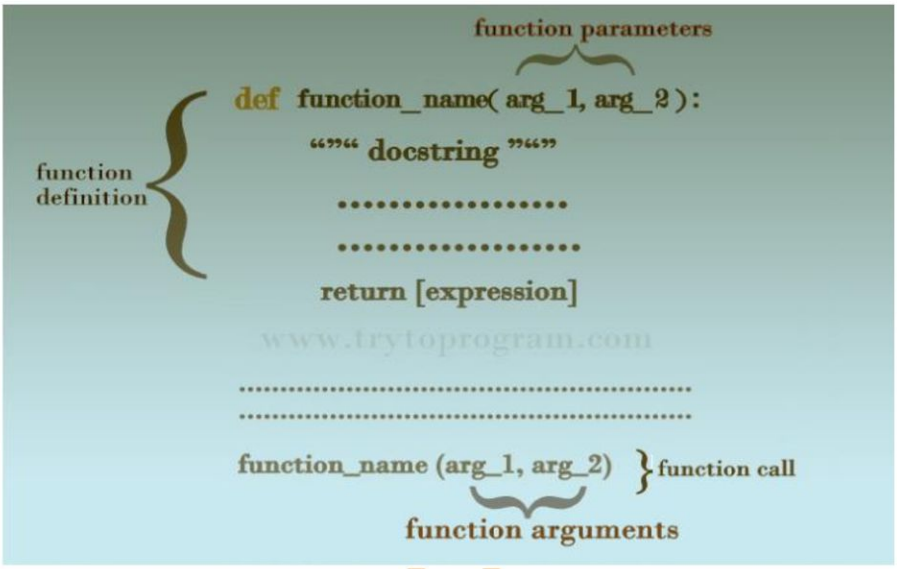


Figure 6.2

Formal and Actual Arguments

The first thing a programmer must be aware of is that parameters and arguments are clearly two different things although people use them synonymously. Parameters are the variables that are defined or used inside parentheses while defining a function, whereas arguments are the value passed for these parameters while calling a function. Arguments are the values that are passed to the function at run-time so that the function can do the designated task using these values.



When a function is defined, it may have some parameters. These parameters are useful to receive values from outside of the function. They are called 'formal arguments'. When we call the function, we should pass data or values to the function. These values are called 'actual arguments'. In the following code, 'a' and 'b' are formal arguments and 'x' and 'y' are actual arguments.

```
def  mul(a,b):   # a,b are formal arguments
    c = a * b
    print(c)

#call the function
x = 10
y = 15
mul(x,y) # x, y are actual arguments
```

The actual arguments used in a function call are of 4 types:

- Positional arguments
- Keyword arguments
- Default arguments

- Variable length arguments

Positional Arguments

These are the arguments passed to a function in correct positional order. Here, the number of arguments and their positions in the function definition should match exactly with the number and position of the argument in the function call. For example, take a function definition with two arguments as:

```
def replicate(x,y):
     for I in range(1,y):
         print(x, end='')

#calling function
replicate('*',10) # It will print '*' 10 times
replicate(10,'*') # generate error
replicate('*')    # generate error
```

Output

```
replicate(10,'*') # generate error
  File "D:/Batch 2018-2022/Python Programs/Demofunction.py", line 2, in replicate
    for I in range(1,y):
TypeError: 'str' object cannot be interpreted as an integer
replicate('*')    # generate error
TypeError: replicate() missing 1 required positional argument: 'y'
```

Keyword Arguments

In function, the values passed through arguments are assigned to parameters in order, by their position. With Keyword arguments, we can use the name of the parameter irrespective of its position while calling the function to supply the values. All the keyword arguments must match one of the arguments accepted by the function.

For example, the definition of a function that display symbol (you want to replicate) and its number(no. of times repeat a symbol) can be written as:

```
replicate(symbol ='*',number=10)
```

Here, we are mentioning a keyword 'symbol' and its value and then another keyword 'number' (repetition number) and its value. Please observe these

keywords are nothing but the parameter names which receive these value. We can change the order of the arguments as:

```
replicate(number =10,symbol='*')
```

In this way, even though we change the order of the arguments, there will not be any problem as the parameter names will guide where to store that value.

Program 4 A Python program to understand the keyword arguments of a function

```
def replicate(symbol,number):
    for I in range(1,number):
        print(symbol, end='')
    print()

#calling function
replicate(symbol ='*',number=10) #'*' assign to x and 10 to y
replicate(number =10,symbol='*') #'*' assign to x and 10 to y
```

Output

```
*********
*********
```

Here is another example to understand the keywords arguments

```
def print_name(name1, name2):
  """ This function prints the name """
  print (name1 + " and " + name2 + " are friends")

#calling the function
print_name(name2 = 'Ajay',name1 = 'Aman')
```

Output:
Ajay and Aman are friends

Notice in above example, if we had supplied arguments as print_name('Ajay','Aman'), the output would have been Ajay and Aman are friends as the values would have been assigned by arguments position.

But using keyword arguments by specifying the name of the argument itself, we don't have to worry about their position. This makes using function easier as we don't need to worry about the order of arguments.

Default Arguments

Sometimes we may want to use parameters in a function that takes default values in case the user doesn't want to provide a value for them.

For this, we can use default arguments which assumes a default value if a value is not supplied as an argument while calling the function. In parameters list, we can give default values to one or more parameters.

We can mention some default value for the function parameters in the definition. Let's take the definition of replicate() function as:

```
def replicate(symbol,number=10):
```

Here, the first arguments is 'symbol' whose default value is not mentioned. But the second argument is 'number' and its default value is mentioned to be 10. At the time of calling this function, if do not pass 'number' value, then the default value of 10 is taken. If we mention the 'number' value, then that mentioned value is utilized. So, a default arguments is an arguments that assumes a default value if a value is not provided in the function call for that argument. Program 5 will clarify this:

Program 5 A Python program to understand the default arguments of a function

```
def replicate(symbol,number=10):
     for I in range(1,number):
         print(symbol, end='')
     print()

#calling function
replicate(symbol ='*',number=15)
replicate(symbol ='*') #default value for number is used.
```

Output

****

****

Here is one more example to understand default arguments

```
def sum(a=4, b=2): #2 is supplied as default argument
  """ This function will print sum of two numbers
      if the arguments are not supplied
      it will add the default value """
  print (a+b)
```

```
sum(1,2)   #calling with arguments
sum( )     #calling without arguments
```

Output:

3

6

In the program above, default arguments 2 and 4 are supplied to the function. First, the user has provided the arguments 1 and 2, hence the function prints their sum which is 3. In the second call, the user has not provided the arguments. Thus the function takes the default arguments and prints their sum.

Note: Common Programming Error

Using a non-default argument after default arguments raise a SyntaxError. In Python functions, a default argument can only be followed by a default argument. Non-default arguments must be placed before default arguments.

Why does Python throw error while using default argument before non-default argument?

In function, the values are assigned to parameters in order, by their position. The value of default argument may be optional as there is the default value if not provided, but the value for a non-default argument is mandatory as there exists no default value for this.

For example, def func(a=1, b) is not allowed because when we will call the function using func(5), the default argument a will be replaced by 5, leaving no value for parameter b. That is why a non-default argument is always placed before default argument.

Variable length Arguments

Sometimes you may need more arguments to process function then you mentioned in the definition. If we don't know in advance about the arguments needed in function, we can use variable-length arguments also called arbitrary arguments.

For this an asterisk (*) is placed before a parameter in function definition which can hold non-keyworded variable-length arguments and a double asterisk (**) is placed before a parameter in function which can hold keyword variable-length arguments.

If we use one asterisk (*) like *var, then all the positional arguments from that point till the end are collected as a tuple called 'var' and if we use two asterisks (**) before a variable like **var, then all the positional arguments from that point till the end are collected as a dictionary called 'var'.

Here is an example.

```
def display(*name, **address):
  for items in name:
    print (items)

  for items in address.items():
    print (items)

#Calling the function
display('john','Mary','Nina',John='LA',Mary='NY',Nina='DC')
Output
John
Mary
Nina
('John', 'LA')
('Mary', 'NY')
('Nina', 'DC')
```

As you can see in the example above, *name takes all the non-keyworded arguments John, Mary, and Nina wrapped into a tuple, whereas **address takes all the keyworded arguments John='LA', Mary ='NY', and Nina='DC' wrapped into a dictionary.

Program6: A Python program to show variable length arguments and its use.

```
# Variable length argument demo
def add(fargs, *args):  # *args can take 0 or more values
    """
        To add given numbers
    """
    print('Formal argument = ', fargs)
    sum =0
    for i in args:
        sum+=i
    print('Sum of all numbers=', (fargs + sum))
# call add() and pass arguments
```

```
add(8,10)
add(9,18,27,36)
add(9)
```

Output:

```
===== RESTART: D:/Batch 2018-2022/Python Programs/VariableLeng
thArgs.py =====
Formal argument =  8
Sum of all numbers= 18
Formal argument =  9
Sum of all numbers= 90
Formal argument =  9
Sum of all numbers= 9
```

As discussed above, a keyword variable length argument is an argument that can accept any number of values provided in the format of keys and values, declare with "**' before the argument.

This argument internally represents a dictionary object. A dictionary store data in the form of key and value pairs. It means, when we provide values for "**kwargs' , we can pass pairs of values using keywords as:

display(5, rno=10)

Here, 5 is stored into 'farg' which is formal argument. 'rno' is stored as key and its value '10' is stored as value in the dictionary that us referenced by kwargs'. Program7 provide better understanding of this concept.

Program7: A Python program to understand keyword variable argument.

```
#keyword variable argument demo
def display(farg, **kwargs):
    print('Formal argument = ', farg)

    for x,y in kwargs.items(): # items() will give pairs of items
        print('Key = {}, Value = {}'.format(x,y))
    print()

display(5, rno = 10)
display(5, rno =10, name = "Rishab")
```

Output:

```
======== RESTART: D:/Batch 2018-2022/Python Programs/KeywordVa
rArg.py ========
Formal argument =  5
Key = rno, Value = 10

Formal argument =  5
Key = rno, Value = 10
Key = name, Value = Rishab
```

Local and Global Variables

Sometimes, the global variable and the local variable may have the same name. In that case, the function, by default, refers to the local variable and ignores the global variables. So, the global variable is not accessible inside the function but outside of it, it is accessible. Consider the following example

#same name for global and local variables

```
a = 1
def myfunction():
   a = 2    # this is local variable
   print('a = ' , a)  #display local variable
myfunction()
print('a =', a)  # display global var
```

Output:

```
a = 2
a = 1
```

When the programmer wants to use the global variable inside a function, he can use the keyword 'global' before the variable in the beginning of the function body as:

```
global a
```

In this way, the global variable is made available to the function and the programmer can work with it as he wishes. In the following program, we are showing how to work with a global variable inside a function.

```
a = 1
def myfunction():
   global a
   print('global a= ', a)    #display global var
```

```
    a = 2    # modify global varibale
    print('modified a = ' , a)  #display new value
myfunction()
print('global a =', a)  # display global var
```

Output:

```
global a = 1
modified a = 2
global a = 2
```

When the global variable name and local variable names are same, the programmer will face difficulty to differentiate between them inside a function. For example there is a global variable 'a' with same value declared above the function. The programmer is writing a local variable with the same name 'a' with some other value inside the function. Consider the following code:

```
a=1  # this is a global var
def myfunction( ):
     a= 2  # this is a local var
```

Now, if the programmer wants to work with global variable, how it is possible? If he uses 'global' keyword, then he can access only global variable and the local variable is no more available. The **global()** function will solve this problem. This is a built in function which returns a table of current global variable in the form of dictionary. Hence, using this function, we can refer to the global variable 'a' as: global() ['a']. Now, this value cane be assigned to another variable, say 'x' and the programmer can work with that value.

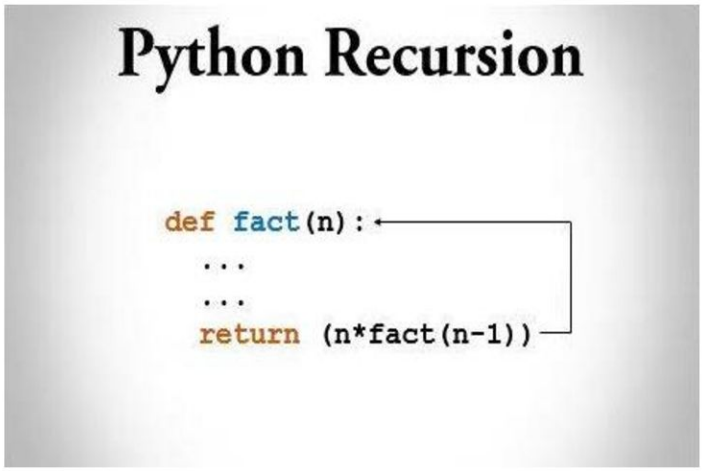
```
a = 1
def myfunction():
    a=2   # a is local var
    x = global()['a']  # get global variable into x
    print('global var a= ', x)    #display global var
    print('local var a = '  a)
    print('modified a = ' , a)  #display new value
myfunction()
print('global a =', a)  # display global var
```

Output:

```
global var a = 1
local var a = 2
global var a = 1
```

Recursive Functions

Recursion means iteration. A function is called recursive, if the body of function calls the function itself until the condition for recursion is true. Thus, a Python recursive function has a termination condition.



Why does a recursive function in Python has termination condition?

Well, the simple answer is to prevent the function from infinite recursion.

When a function body calls itself with any condition, this can go on forever resulting an infinite loop or recursion. This termination condition is also called base condition.

Advantages of Python Recursion

i. Reduces unnecessary calling of function, thus reduces length of program.
ii. Very flexible in data structure like stacks, queues, linked list and quick sort.
iii. Big and complex iterative solutions are easy and simple with Python recursion.
iv. Algorithms can be defined recursively making it much easier to visualize and prove.

Disadvantages of Python Recursion

i. Slow.
ii. Logical but difficult to trace and debug.
iii. Requires extra storage space. For every recursive calls separate memory is allocated for the variables.
iv. Recursive functions often throw a Stack Overflow Exception when processing or operations are too large.

Python Recursion: Example

Let's get an insight of Python recursion with an example to find the factorial of 3.

```
3! = 3 * 2! = 3 * (2 * 1!) = 3 * 2 * 1
```

This is how a factorial is calculated. Let's implement this same logic into a program.

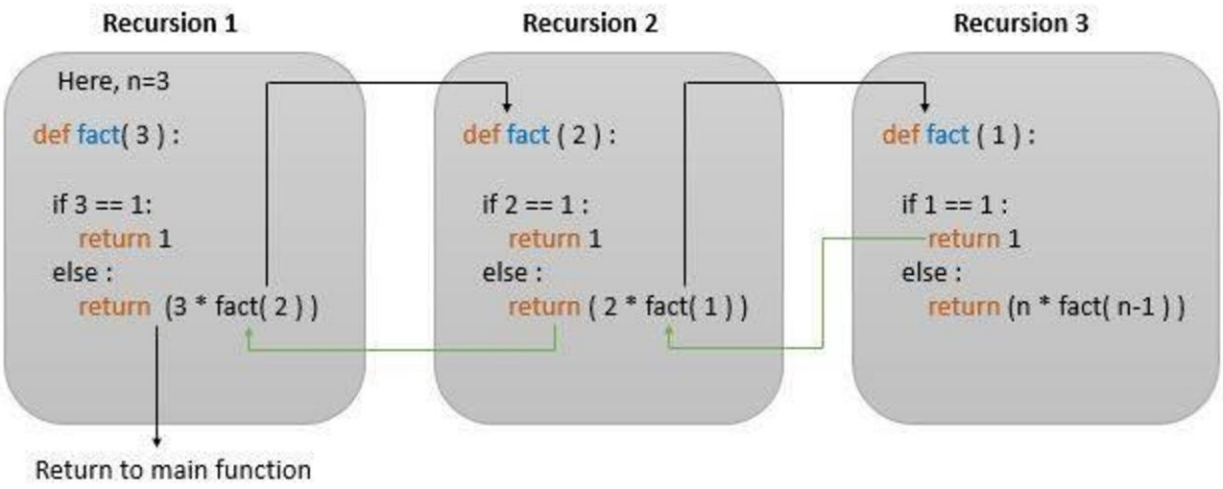
```
#recursive function to calculate factorial
def fact(n):
  """ Function to find factorial """
  if n == 1:
    return 1
  else:
    return (n * fact(n-1))

print ("3! = ",fact(3))
```

Output

```
3! = 6
```

Explanation of the program

First is a base condition in any recursive function. If the value of n is equal to 1, then the function will return 1 and exit. Till this base condition is met, the function will be iterated. In the program above, the value of the argument supplied is 3. Hence, the program operates in following way.



When the function is called with the value of n

In recursion 1: Function returns 3 * fact(2). This invokes the function again with the value of n = 2.

In recursion 2: Function checks if n = 1. Since it's False function return 3 * 2 * fact(1). This again invokes the function with the value of n = 1.

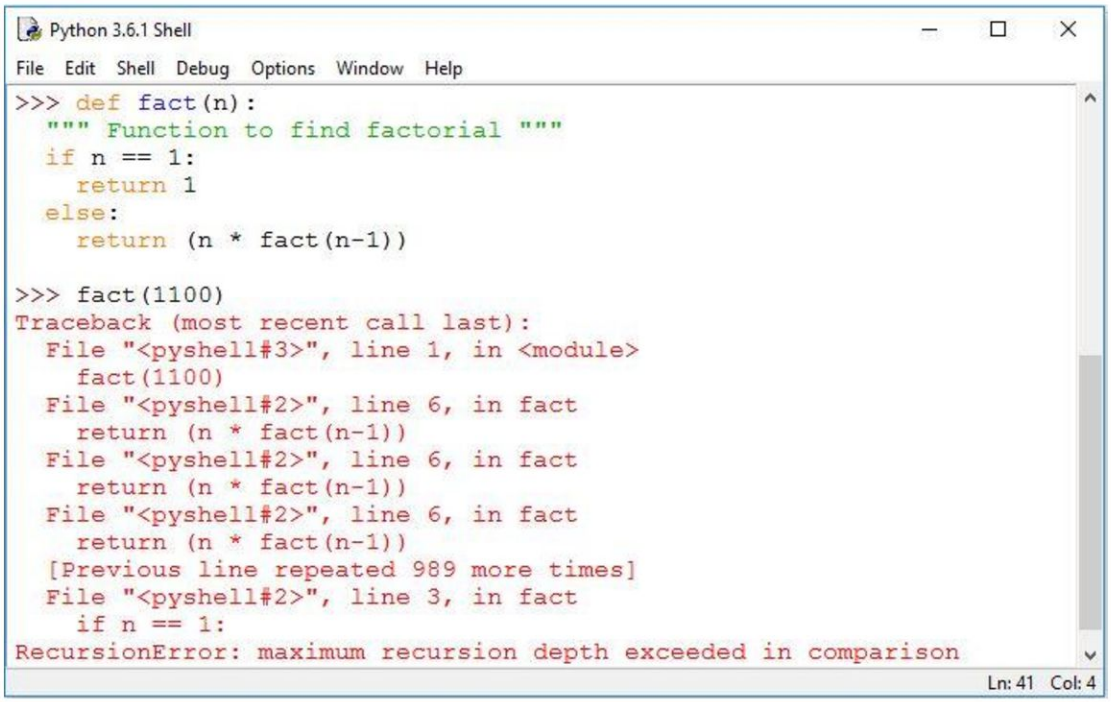
In recursion 3: Function checks if n = 1. This returns True making the function to exit and return 1.

Hence after 3 recursions, the final value returned is 3 * 2 * 1 = 6.

Is there any limit on the number of recursions for a Python recursive function?

The answer is YES.

Unless we explicitly set the maximum limit of recursions, the program by default will throw a Recursion error after 1000 recursions. Here is what happens when we supply 1100 as an argument in Python recursive function and the program have to call itself recursively over 1000 times.



To avoid such errors, we can explicitly set recursion limit using sys.setrecursionlimit( ).

```
import sys
sys.setrecursionlimit(1500)
```

Anonymous Function or Lambdas

A function without a name is called 'anonymous function'. So far, the function we wrote were defined using the keyword *def*. But anonymous functions are not defined using 'def'. They are defined using the keyword ***lambda*** and hence they are called 'Lambda function'. Let's take a normal function that returns square of a given value.

```
def square(x)
    return x*x
```

The same function can be written as anonymous function as:

```
lambda x : x * x
```

Observe the keyword 'lambda'. This represents that an anonymous function is being created. After that, we have written an argument of the function, i.e. 'x'. Then colon (:) represents the beginning of the function that contains an expression x * x. Please observer that we did not use any name for the function here.

Normally, if a function returns some value, we assign that value to a variable as:

```
y = square(3)
```

But, lambda functions returns some value, we assign that value to a variable as:

```
f = lambda x : x * x
```

Here, 'f' is the function name to which the lambda expression is assigned. Now, if we call the function f() as:

```
value = f(3)
```

Now, 'value' contains the square value of 3 i.e. 9.

```
# Program to create lambda function that returns a square value of given number
f = lambda x : x * x    #write lambda function
value = f(5)  call lambda function
print('Square of 5 = ' , value)
```

Python Modules

Python modules are nothing but files that consist of different statements and functions defined inside. A module can define functions, classes, and variables. Modules help in organizing the code making it easier to use and understand. Modules provide reusability of the code. Any file with extension .py can be referred as a module and the functions defined inside the module can be used in another program by simply using the import statement.

Suppose we need a function to find factorial in many programs. So, instead of defining a function to find the factorial in each program, what we can do is create a module with a function to find factorial and use that function in every program by simply importing the module.

How to create a Python Module?

Creating a Python module is as simple as defining a function and saving it as a .py file so that we can use this function later by just importing this module.

For example, let's create a module findfact.py which contains a function to find the factorial of any number and a function to check positive or negative number. We will use recursion to find factorial.

```
#findfact.py
#function to find factorial
def fact(n):
  """ Function to find factorial """
  if n == 1:
    return 1
  else:
    return (n * fact(n-1))

#function to check positive/Negative
def check_num(a):
  """ Function to check positive/Negative number """
  if a > 0:
    print (a ," is a positive number.")
  elif a == 0:
    print ("Number is zero.")
  else:
    print (a ," is negative number.")
```

Save this file as findfact.py and there you have created your first ever Python module. Now let's see how to import this module in other programs and use the function defined in it.

How to import Python modules?

There are tons of standard modules that are included in our local machine inside the folder Lib in the directory we installed Python.

To import a Python module be it standard or user-defined, the keyword import is used followed by the module name. For example, let's import the module findfact.py and use the function to find factorial.

```
>>> import findfact
```

Importing a module is as simple as mentioned above.

Now to use the functions defined inside this module we use (.) operator in following way.

```
>>> import findfact
>>> findfact.fact(5)  #calling factorial function inside module
120
```

Note: In Python, a module name is stored within a module available as the global variable __name__ (note the double underscores).

```
>>> import findfact
>>> findfact.__name__
'findfact'
```

This was a simple demonstration to import modules in Python using the import statement. There are a couple of other ways to import Python modules using different forms of import statements.

Python from .. import statement

Imagine you have multiple functions defined inside a module like we have two functions defined inside findfact.py.

Python from .. import statement allows us to import particular function from the module. Here is the example.

```
>>> #importing only check_num function from findfact.py
>>> from findfact import check_num
>>> findfact.check_num(2)
2 is a positive number.
```

Import module as object

Python modules can be imported as objects. In such case, instead of module_name.function_name( ) we use object.function_name( ).

Here is the example.

```
>>> #importing findfact.py as f
>>> import findfact as f
>>> f.fact(5)
120
>>> f.check_num(0)
Number is zero.
```

Import everything from module

Besides importing certain functions, we can import everything defined inside the module and use the functions directly in the program.

Here is the example.

```
>>> #importing every functions from findfact.py
>>> from findfact import *
>>> check_num(2)
2 is a positive number.
>>> fact(5)
120
```

This imports all names except those beginning with an underscore (_).

Importing everything from a module using (*) might seem easy to use, but this can lead to duplicate methods and definitions. The methods name in the module and the main program may have same names. So it's better to import certain functions or import module as an object.

Reloading a Python Module

Sometimes we may feel need to change the functions or statements in the module that we have already imported in our Python Shell.

Python Packages

A Python package in simple words is a directory that contains Python files. Just like a directory has sub-directories and those sub-directories also have files inside, a Python package also has sub-packages and those sub-packages again have different modules defined inside.

A directory with Python files can only be considered as Python package if the directory has a file with name __init__.py.

So every directory having a file __init__.py is considered Python package and the sub-directories having the file __init__.py are considered sub-packages.

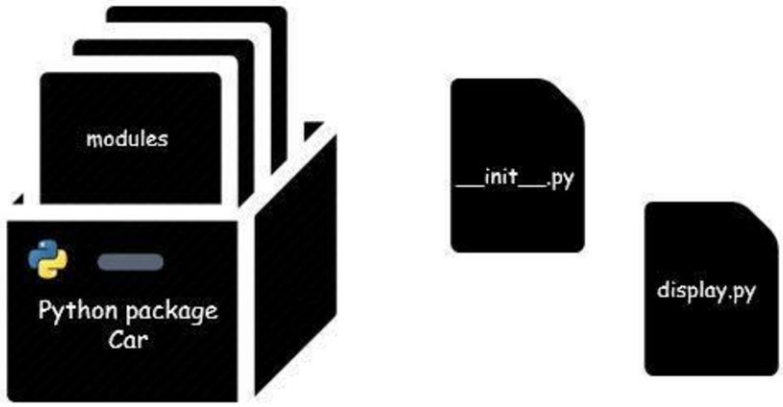
The file __init__.py can be empty or contain code to initialize the package as it's the first file that is executed while importing a package.

How to create a Python Package?

Creating file is like creating directories and sub-directories with a file __init__.py placed inside.

Example 1: let's create a simple package car containing a simple module **display.py**.

Here is the structure for creating this package.



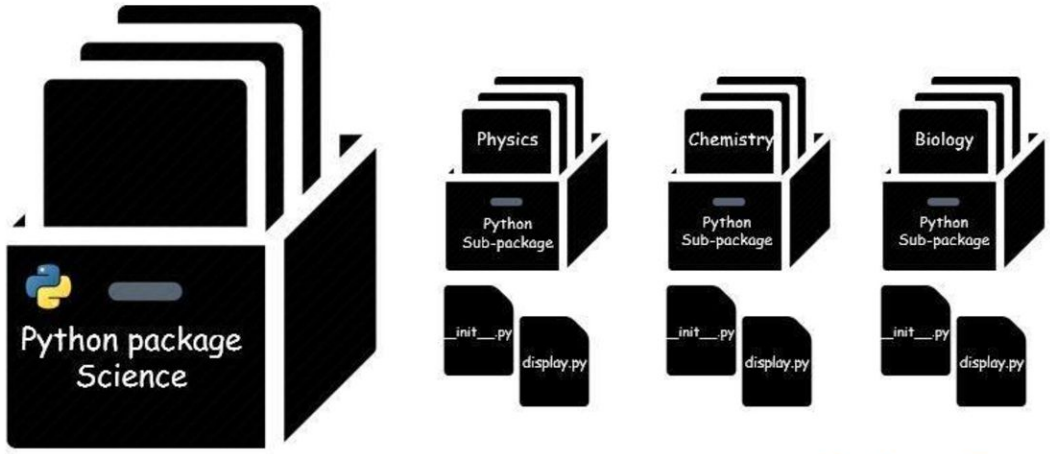
1. Create a directory car to create a package of the same name.
2. Create a file __init__.py and place it inside directory car so that it can be considered a Python package.
3. Create a module inside directory car.The code for module **display.py**.

```
#display.py
def display():
print ('This is a car.')
```

Before discussing how to import Python packages, let's take another example and create a Python package with sub-packages.

Example 2: Create a package named science containing physics, chemistry, and biology as its sub-packages.

Here is the structure.



1. Create a directory science to create a package of the same name.
2. Create a file __init__.py and place it inside directory science so that it can be considered a Python package.
3. Create sub-directories physics, chemistry, and biology and place __init__.py inside each sub-directories so that they can be considered Python sub-packages.
4. Now finally create the corresponding modules inside each sub-packages.**physics/display.py**

```
#display.py
def display():
print ('This is physics.')
```

chemistry/display.py

```
#display.py
def display():
print ('This is chemistry.')
```

biology/display.py

```
#display.py
def display():
print ('This is biology.')
```

Now that we have created Python packages, let's learn about importing them.

How to import Python packages?

We can import Python packages using import statement and dot (.) operator.

If we were to import the package car that we created in example 1, then we have to import it like:

```
>>> import car
```

Now to import a function defined inside a module of this package car, here is the way.

```
>>> from car import display
>>> display.display()  #calling function defined in the module
This is a car.
```

Another way of importing the module in a package is

```
>>> import car.display
>>> display.display()
This is a car.
```

To import Python packages with sub-packages

In example 2, we created a package science with sub-packages. Here is how we import modules and packages in such cases.

```
>>> import science.physics.display
>>> display.display()
This is physics.
```

This is the way to import modules from sub-packages.

What if we want to import functions inside the modules defined inside sub-packages?

Here is how it is done.

```
>>> #importing display function from display module inside sub-package chemistry
>>> from science.chemistry.display import display
```

```
>>> display()
This is chemistry.
```