# FUNCTIONS:

A function is a block of organized, reusable code that is used to perform a single, related action.

➢ Once a function is written, it can be reused as and when required. So, functions are also called reusable code.

➢ Functions provide modularity for programming. A module represents a part of the program. Usually, a programmer divides the main task into smaller sub tasks called modules.

➢ Code maintenance will become easy because of functions. When a new feature has to be added to the existing software, a new function can be written and integrated into the software.

➢ When there is an error in the software, the corresponding function can be modified without disturbing the other functions in the software.

➢ The use of functions in a program will reduce the length of the program.

As you already know, Python gives you many built-in functions like sqrt( ), etc. but you can also create your own functions. These functions are called *user-defined functions.*

## 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 becomes a „method‟. A method is called using object name or class name. A method is called using one of the following ways:

**Objectname.methodname()**
**Classname.methodname()**

## Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword **def** followed by the function name and parentheses ( ).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return none.

**Syntax:**

```
def functionname (parameters):
        """function_docstring"""
        function_suite
        return [expression]
```

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

**Example:**

```
def add(a,b):
        """This function sum the numbers"""
        c=a+b
        print c
        return
```

Here, „*def'* represents starting of function. „*add'* is function name. After this name, parentheses ( ) are compulsory as they denote that it is a function and not a variable or something else. In the parentheses we wrote two variables „*a*" and „*b*" these variables are called „parameters". A parameter is a variable that receives data from outside a function. So, this function receives two values from outside and those are stored in the variables „a" and „b". After parentheses, we put colon (:) that represents the beginning of the function body. The function body contains a group of statements called „suite".

## Calling Function:

A function cannot run by its own. It runs only when we call it. So, the next step is to call function using its name. While calling the function, we should pass the necessary values to the function in the parentheses as:

```
add(5,12)
```

Here, we are calling „add" function and passing two values 5 and 12 to that function. When this statement is executed, the python interpreter jumps to the function definition and copies the values 5 and 12 into the parameters „a" and „b" respectively.

**Example:**

```
def add(a,b):
        """This function sum the numbers"""
        c=a+b
        print c
add(5,12) # 17
```

## Returning Results from a function:

We can return the result or output from the function using a „return" statement in the function body. When a function does not return any result, we need not write the return statement in the body of the function.

Q) Write a program to find the sum of two numbers and return the result from the function.

```
def add(a,b):
        """This function sum the numbers"""
        c=a+b
        return c
print add(5,12) # 17
print add(1.5,6) #6.5
```

## Returning multiple values from a function:

A function can returns a single value in the programming languages like C, C++ and 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 a, b, c**

Here, three values which are in „a‟, „b‟ and „c‟ are returned. These values are returned by the function as a tuple. To grab these values, we can three variables at the time of calling the function as:

x, y, z = functionName( )

Here, „x‟, „y‟ and „z‟ are receiving the three values returned by the function.

**Example:**

```
def
    calc(a,b):
    c=a+b
    d=a-b
    e=a*b
    return c,d,e
x,y,z=calc(5,8)
print "Addition=",x
print "Subtraction=",y
```

**Functions are First Class Objects:**

In Python, functions are considered as first class objects. It means we can use functions as perfect objects. In fact when we create a function, the Python interpreter internally creates an object. Since functions are objects, we can pass a function to another function just like we pass an object (or value) to a function. The following possibilities are:

➢ It is possible to assign a function to a variable.
➢ It is possible to define one function inside another function.
➢ It is possible to pass a function as parameter to another function.
➢ It is possible that a function can return another function.

To understand these points, we will take a few simple programs.

Q) A python program to see how to assign a function to a variable.

```
def display(st):
    return "hai"+st
x=display("cse")
print x
```
                       **Output:** haicse

Q) A python program to know how to define a function inside another function.

```
def display(st):
    def message():
        return "how r u?"
    res=message()+st
    return res
x=display("cse")
print x
```
                       **Output:** how r u?cse

Q) A python program to know how to pass a function as parameter to another function.

```
def display(f):
    return "hai"+f
def message():
    return "how r u?"
fun=display(message())
print fun
```
**Output:** haihow r u?

Q) A python program to know how a function can return another function.

```
def display():
    def message():
        return "how r u?"
    return message
fun=display()
print fun()
```
**Output:** how r u?

**Pass by Value:**

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. In python, the values are sent to functions by means of object references. We know everything is considered as an object in python. All numbers, strings, tuples, lists and dictionaries are objects.

If we store a value into a variable as:

**x=10**

In python, everything is an object. An object can be imagined as a memory block where we can store some value. In this case, an object with the value „10‟ is created in memory for which a name „x‟ is attached. So, 10 is the object and „x‟ is the name or tag given to that object. Also, objects are created on heap memory which is a very huge memory that depends on the RAM of our computer system.

**Example:** A Python program to pass an integer to a function and modify it.

```
def modify(x):
    x=15
    print "inside",x,id(x)
x=10
modify(x)
print "outside",x,id(x)
```

**Output:**

```
inside 15 6356456
outside 10 6356516
```

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 references to the modify( ) function. The object is 10 and its references 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 created in the memory is that the integer objects are immutable (not modifiable). So in the function, when we display „x" value, it will display 15. Once we come outside the function and display „x" value, it will display numbers of „x" inside and outside the function, and we see different numbers since they are different objects.

In python, integers, floats, strings and tuples are immutable. That means their data cannot be modified. When we try to change their value, a new object is created with the modified value.
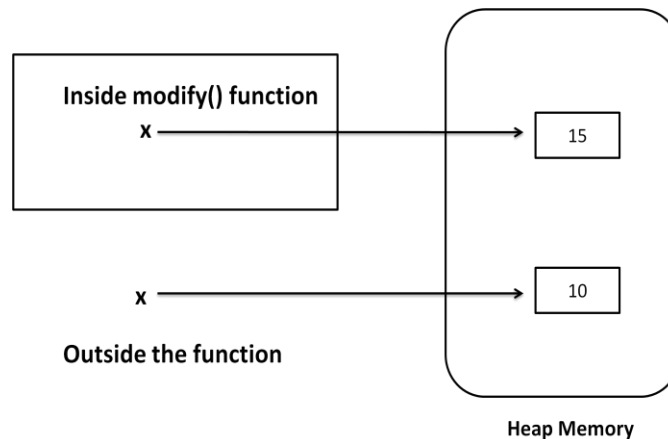


**Fig.** Passing Integer to a Function

**Pass by Reference:**

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.

In python, lists and dictionaries are mutable. That means, when we change their data, the same object gets modified and new object is not created. In the below program, we are passing a list of numbers to modify ( ) function. 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.

**Example:** A Python program to pass a list to a function and modify it.

```
def modify(a):
    a.append(5)
    print "inside",a,id(a)
a=[1,2,3,4]
modify(a)
print "outside",a,id(a)
```

**Output:**

```
inside [1, 2, 3, 4, 5] 45355616
outside [1, 2, 3, 4, 5] 45355616
```

In the above program the list „a" is the name or tag that represents the list object. Before calling the modify( ) function, the list contains 4 elements as: **a=[1,2,3,4]**

Inside the function, we are appending a new element „5" to the list. Since, lists are mutable, adding a new element to the same object is possible. Hence, append( ) method modifies the same object.
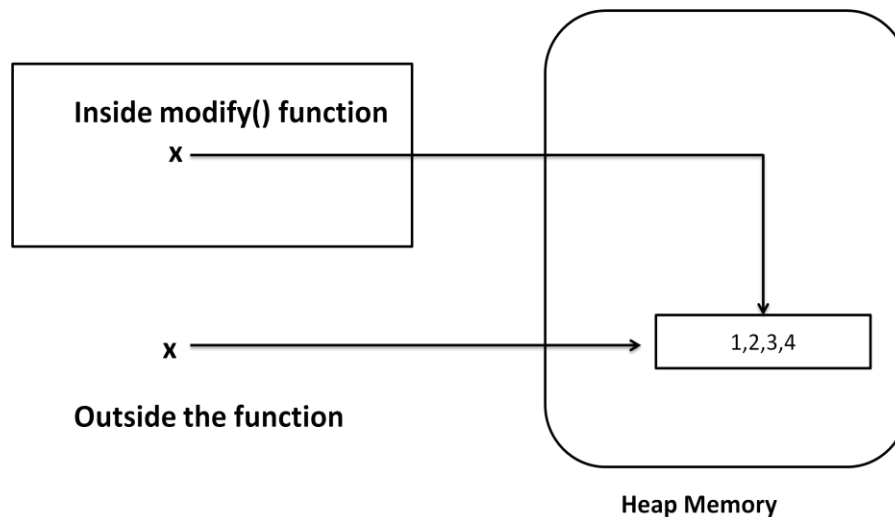
**Fig.** Passing a list to the function

**Formal and Actual Arguments:**

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.

**Example:**

```
def add(a,b): # a, b are formal arguments
    c=a+b
    print c
x,y=10,15
add(x,y)    # x, y are actual arguments
```

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

    **a)** Positional arguments
    **b)** Keyword arguments
    **c)** Default arguments
    **d)** Variable length arguments

**a) Positional Arguments:**

These are the arguments passed to a function in correct positional order. Here, the number of arguments and their position in the function definition should match exactly with the number and position of argument in the function call.

```
def attach(s1,s2):
    s3=s1+s2
    print s3
attach("New","Delhi") #Positional arguments
```

This function expects two strings that too in that order only. Let‟s assume that this function attaches the two strings as s1+s2. So, while calling this function, we are supposed to pass only two strings as: **attach("New","Delhi")**

The preceding statements displays the following output NewDelhi

Suppose, we passed "Delhi" first and then "New", then the result will be: "DelhiNew". Also, if we try to pass more than or less than 2 strings, there will be an error.

**b) Keyword Arguments:**

Keyword arguments are arguments that identify the parameters by their names. For example, the definition of a function that displays grocery item and its price can be written as:

**def grocery(item, price):**

At the time of calling this function, we have to pass two values and we can mention which value is for what. For example,

**grocery(item='sugar', price=50.75)**

Here, we are mentioning a keyword „item" and its value and then another keyword „price" and its value. Please observe these keywords are nothing but the parameter names which receive these values. We can change the order of the arguments as:

**grocery(price=88.00, item='oil')**

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.

```
def grocery(item,price):
    print "item=",item
    print "price=",price
grocery(item="sugar",price=50.75) # keyword arguments
grocery(price=88.00,item="oil") # keyword arguments
```

**Output:**

item= sugar

price= 50.75

item= oil

price= 88.0

**c) Default Arguments:**

We can mention some default value for the function parameters in the definition.

Let"s take the definition of grocery( ) function as:

**def grocery(item, price=40.00)**

Here, the first argument is „item" whose default value is not mentioned. But the second argument is „price" and its default value is mentioned to be 40.00. at the time of calling this function, if we do not pass „price" value, then the default value of 40.00 is taken. If we mention the „price" value, then that mentioned value is utilized. So, a default argument is an argument that assumes a default value if a value is not provided in the function call for that argument.

**Example:**
```
def grocery(item,price=40.00):
    print "item=",item
    print "price=",price
grocery(item="sugar",price=50.75)
grocery(item="oil")
```

**Output:**

> item= sugar
> price= 50.75
> item= oil
> price= 40.0

**d) Variable Length Arguments:**

Sometimes, the programmer does not know how many values a function may receive. In that case, the programmer cannot decide how many arguments to be given in the function definition. for example, if the programmer is writing a function to add two numbers, he/she can write:

> **add(a,b)**

But, the user who is using this function may want to use this function to find sum of three numbers. In that case, there is a chance that the user may provide 3 arguments to this function as:

> **add(10,15,20)**

Then the add( ) function will fail and error will be displayed. If the programmer want to develop a function that can accept „n‟ arguments, that is also possible in python. For this purpose, a variable length argument is used in the function definition. a variable length argument is an argument that can accept any number of values. The variable length argument is written with a „*‟ symbol before it in the function definition as:

> **def add(farg, *args):**

here, „farg‟ is the formal; argument and „*args‟ represents variable length argument. We can pass 1 or more values to this „*args‟ and it will store them all in a tuple.

**Example:**

```
def add(farg,*args):
    sum=0
    for i in args:
        sum=sum+i
    print "sum is",sum+farg
add(5,10)
add(5,10,20)
add(5,10,20,30)
```

**Output:**

> sum is 15
> sum is 35
> sum is 65

**Local and Global Variables:**

When we declare a variable inside a function, it becomes a local variable. A local variable is a variable whose scope is limited only to that function where it is created. That means the local variable value is available only in that function and not outside of that function.

When the variable „a‟ is declared inside myfunction() and hence it is available inside that function. Once we come out of the function, the variable „a‟ is removed from memory and it is not available.

**Example-1:**

```
def myfunction():
    a=10
    print "Inside function",a #display 10
myfunction()
print "outside function",a # Error, not available
```

**Output:**

```
Inside function 10
outside function
```

**NameError: name 'a' is not defined**

When a variable is declared above a function, it becomes global variable. Such variables are available to all the functions which are written after it.

**Example-2:**

```
a=11
def myfunction():
    b=10
    print "Inside function",a #display global var
    print "Inside function",b #display local var
myfunction()
print "outside function",a # available
print "outside function",b # error
```

**Output:**

```
Inside function 11
Inside function 10
outside function 11
outside function
```

**NameError: name 'b' is not defined**

**The Global Keyword:**

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 variable. So, the global variable is not accessible inside the function but outside of it, it is accessible.

**Example-1:**

```
a=11
def myfunction():
    a=10
    print "Inside function",a # display local variable
myfunction()
print "outside function",a # display global variable
```

**Output:**

```
Inside function 10
outside function 11
```

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**

**Example-2:**

```
a=11
def myfunction():
    global a
    a=10
    print "Inside function",a # display global variable
myfunction()
print "outside function",a # display global variable
```

**Output:**

     Inside function 10

     outside function 10

## Recursive Functions:

A function that calls itself is known as „recursive function‟. For example, we can write the factorial of 3 as:

     factorial(3) = 3 * factorial(2)

     Here, factorial(2) = 2 * factorial(1)

     And, factorial(1) = 1 * factorial(0)

Now, if we know that the factorial(0) value is 1, all the preceding statements will evaluate and give the result as:

     factorial(3) = 3 * factorial(2)

               = 3 * 2 * factorial(1)

               = 3 * 2 * 1 * factorial(0)

               = 3 * 2 * 1 * 1

               = 6

From the above statements, we can write the formula to calculate factorial of any number „n‟ as:     factorial(n) = n * factorial(n-1)

**Example-1:**

```
def factorial(n):
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    return result
for i in range(1,5):
    print "factorial of ",i,"is",factorial(i)
```

**Output:**

     factorial of  1 is 1

     factorial of  2 is 2

     factorial of  3 is 6

     factorial of 4 is 24

## Anonymous Function or Lambdas:

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

➢ Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

➢ An anonymous function cannot be a direct call to print because lambda requires an expression.

➢ Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

➢ Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Let"s take a normal function that returns square of given value:

**def square(x):**
**return x*x**

the same function can be written as anonymous function as:

**lambda x: x*x**

The colon (:) represents the beginning of the function that contains an expression x*x. The syntax is:

**lambda argument_list: expression**

**Example:**

f=lambda x:x*x
value = f(5)
print value

**The map() Function**

The advantage of the lambda operator can be seen when it is used in combination with the map() function. map() is a function with two arguments:

**r = map(func, seq)**

The first argument *func* is the name of a function and the second a sequence (e.g. a list) *seq*. *map()* applies the function *func* to all the elements of the sequence *seq*. It returns a new list with the elements changed by *func*

```
def fahrenheit(T):
    return ((float(9)/5)*T + 32)
def celsius(T):
    return (float(5)/9)*(T-32)
temp = (36.5, 37, 37.5,39)
F = map(fahrenheit, temp)
C = map(celsius, F)
```

In the example above we haven't used lambda. By using lambda, we wouldn't have had to define and name the functions fahrenheit() and celsius(). You can see this in the following interactive session:

>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
>>> print Fahrenheit
[102.56, 97.70000000000003, 99.14000000000001, 100.03999999999999]
>>> C = map(lambda x: (float(5)/9)*(x-32), Fahrenheit)

>>> print C

[39.200000000000003, 36.5, 37.300000000000004, 37.799999999999997]

map() can be applied to more than one list. The lists have to have the same length. map() will apply its lambda function to the elements of the argument lists, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

>>> a = [1,2,3,4]

>>> b = [17,12,11,10]

>>> c = [-1,-4,5,9]

>>> map(lambda x,y:x+y, a,b)

[18, 14, 14, 14]

>>> map(lambda x,y,z:x+y+z, a,b,c)

[17, 10, 19, 23]

>>> map(lambda x,y,z:x+y-z, a,b,c)

[19, 18, 9, 5]

We can see in the example above that the parameter x gets its values from the list a, while y gets its values from b and z from list c.

## Filtering

The function filter(function, list) offers an elegant way to filter out all the elements of a list, for which the function *function* returns True. The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False. This function will be applied to every element of the list *l*. Only if f returns True will the element of the list be included in the result list.

>>> fib = [0,1,1,2,3,5,8,13,21,34,55]

>>> result = filter(lambda x: x % 2, fib)

>>> print result

[1, 1, 3, 5, 13, 21, 55]

>>> result = filter(lambda x: x % 2 == 0, fib)

>>> print result

[0, 2, 8, 34]

## Reducing a List

The function reduce(func, seq) continually applies the function func() to the sequence seq. It returns a single value.

If seq = [ $s_1$, $s_2$, $s_3$, ... , $s_n$ ], calling reduce(func, seq) works like this:

- At first the first two elements of seq will be applied to func, i.e. func($s_1$,$s_2$) The list on which reduce() works looks now like this: [ func($s_1$, $s_2$), $s_3$, ... , $s_n$ ]

- In the next step func will be applied on the previous result and the third element of the list, i.e. func(func($s_1$, $s_2$),$s_3$). The list looks like this now: [ func(func($s_1$, $s_2$),$s_3$), ... , $s_n$ ]

- Continue like this until just one element is left and return this element as the result of reduce()

We illustrate this process in the following example:

>>> reduce(lambda x,y: x+y, [47,11,42,13])

113

The following diagram shows the intermediate steps of the calculation:
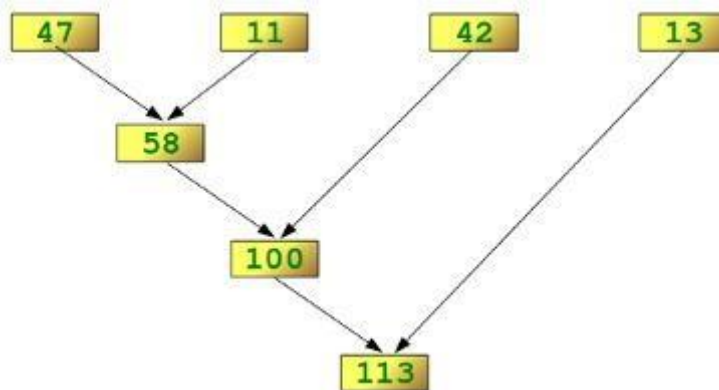


**Examples of reduce ( )**

Determining the maximum of a list of numerical values by using reduce:

>>> f = lambda a,b: a if (a > b) else b

>>> reduce(f, [47,11,42,102,13])

102

>>>

Calculating the sum of the numbers from 1 to 100:

>>> reduce(lambda x, y: x+y, range(1,101))

5050

## Function Decorators:

A decorator is a function that accepts a function as parameter and returns a function. A decorator takes the result of a function, modifies the result and returns it. Thus decorators are useful to perform some additional processing required by a function.

The following steps are generally involved in creation of decorators:

➢ We should define a decorator function with another function name as parameter.

➢ We should define a function inside the decorator function. This function actually modifies or decorates the value of the function passed to the decorator function.

➢ Return the inner function that has processed or decorated the value.

**Example-1:**

```
def decor(fun):
    def inner():
        value=fun()
        return value+2
    return inner
def num():
    return 10
result=decor(num)
print result()
```

**Output:**

12

To apply the decorator to any function, we can use **'@'** symbol and decorator name just above the function definition.

**Example-2:** A python program to create two decorators.

```
def decor1(fun):
   def inner():
      value=fun()
      return value+2
   return inner
def decor2(fun):
   def inner():
      value=fun()
      return value*2
   return inner
def num():
   return 10


result=decor1(decor2(num))
print result()
```

**Output:**

22

**Example-3:** A python program to create two decorators to the same function using „@"
symbol.

```
def decor1(fun):
   def inner():
      value=fun()
      return value+2
   return inner
def decor2(fun):
   def inner():
      value=fun()
      return value*2
   return inner
@decor1
@decor2
def num():
   return 10

print num()
```

**Output:**

22

## Function Generators:

A generator is a function that produces a sequence of results instead of a single value. „yield" statement is used to return the value.

```
def mygen(n):
    i = 0
    while i < n:
        yield i
        i += 1
g=mygen(6)
for i in g:
    print i,
```

**Output:**

0 1 2 3 4 5

**Note:** „yield" statement can be used to hold the sequence of results and return it.


## Modules:

A module is a file containing Python definitions and statements. The file name is the module name with the suffix.py appended. Within a module, the module"s name (as a string) is available as the value of the global variable __name__. For instance, use your favourite text editor to create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module
def fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Now enter the Python interpreter and import this module with the following command:

>>> **import fibo**

This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there. Using the module name you can access the functions:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55,  89]
>>> fibo._name___
'fibo'
```

## from statement:

➢ A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement. (They are also run if the file is executed as a script.)

➢ Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user"s global variables. On the other hand, if you know what you are doing you can touch a module"s global variables with the same notation used to refer to its functions, modname.itemname.

➢ Modules can import other modules. It is customary but not required to place all import statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module"s global symbol table.

➢ There is a variant of the import statement that imports names from a module directly into the importing module"s symbol table. For example:

>>> **from fibo import** fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, fibo is not defined).

There is even a variant to import all names that a module defines:

>>> **from fibo import** *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377

**Namespaces and Scoping**

➢ Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).

➢ A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable shadows the global variable.

➢ Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.

➢ Python makes educated guesses on whether variables are local or global. It assumes that any variable assigned a value in a function is local.

➢ Therefore, in order to assign a value to a global variable within a function, you must first use the global statement.

➢ The statement *global VarName* tells Python that *VarName* is a global variable. Python stops searching the local namespace for the variable.

➢ For example, we define a variable *Money* in the global namespace. Within the function*Money*, we assign *Money* a value, therefore Python assumes *Money* as a local variable. However, we accessed the value of the local variable *Money* before setting it, so an UnboundLocalError is the result. Uncommenting the global statement fixes the problem.

**Packages in Python**

A package is a hierarchical file directory structure that defines a single Python application environment that consists of modules and subpackages and subsubpackages, and so on.

**Third Party Packages:**

The Python has got the greatest community for creating great python packages. There are more tha 1,00,000 Packages available at https://pypi.python.org/pypi .

Python Package is a collection of all modules connected properly into one form and distributed PyPI, the Python Package Index maintains the list of Python packages available. Now when you are done with pip setup Go to command prompt / terminal and say

         pip install <package_name>

**Note:** In windows, pip file is in "Python27\Scripts" folder. To install package you have goto the path C:\Python27\Scripts in command prompt and install.
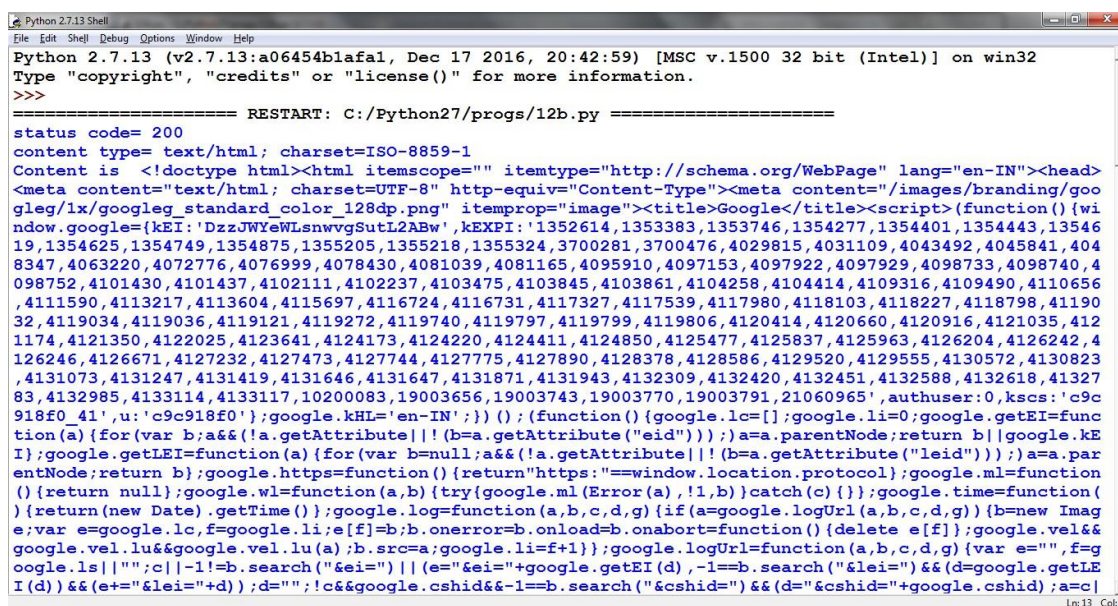
The requests and flask Packages are downloaded from internet. To download install the packages follow the commands

  ➢ **Installation of requests Package:**
       ∝ **Command:** cd C:\Python27\Scripts
       ∝ **Command:** pip install requests

  ➢ **Installation of flask Package:**
       ∝ **Command:** cd C:\Python27\Scripts
       ∝ **Command:** pip install flask

**Example:** Write a script that imports requests and fetch content from the page.

```
import requests
r = requests.get('https://www.google.com/')
print r.status_code
print r.headers['content-type']
print r.text
```

There are some libraries in python:

➢ **Requests**: The most famous HTTP Library. It is a must and an essential criterion for every Python Developer.

➢ **Scrapy**: If you are involved in webscripting then this is a must have library for you. After using this library you won"t use any other.

➢ **Pillow**: A friendly fork of PIL (Python Imaging Library). It is more user-friendly than PIL and is a must have for anyone who works with images.

➢ **SQLAchemy**: It is a database library.

➢ **BeautifulSoup**: This xml and html parsing library.

➢ **Twisted**: The most important tool for any network application developer.

➢ **NumPy**: It provides some advanced math functionalities to python.

➢ **SciPy**: It is a library of algorithms and mathematical tools for python and has caused many scientists to switch from ruby to python.

➢ **Matplotlib**: It is a numerical plotting library. It is very useful for any data scientist or any data analyzer.