Functions in python(Part-2)

Passing a Group of Elements to a Function

To pass a group of elements like numbers or strings, we can accept them into a list and then pass the list to the function where the required processing can be done. In Program, we are accepting a group of integers from the keyboard in a single line using the following statement:

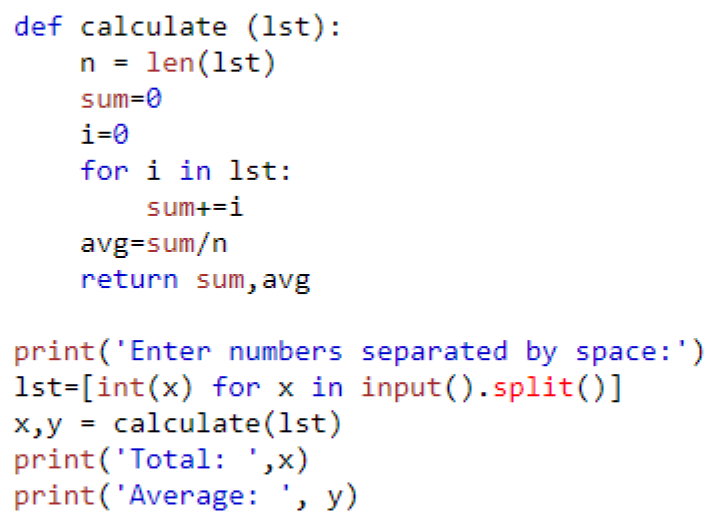
lst = [int() for x in input().split()]

The input() function takes a group of integers as a string. This string is split into pieces where a space is found as a space is the default for split() method. These pieces are then converted into integers by the int() function and returned into the list lst'. We pass this list of integers to the calculate() function as:

x, y = calculate(lst)

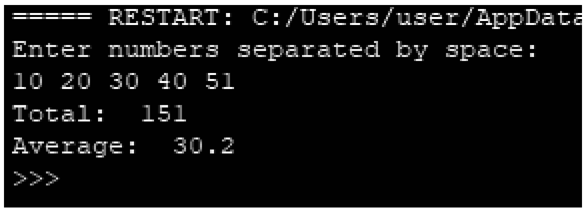
The calculate() function returns sum and average which are stored into x and y respectively.

A function to accept a group of numbers and find their total average. a function to find total and average



```
def calculate (lst):
    n = len(lst)
    sum=0
    i=0
    for i in lst:
        sum+=i
    avg=sum/n
    return sum,avg

print('Enter numbers separated by space:')
lst=[int(x) for x in input().split()]
x,y = calculate(lst)
print('Total: ',x)
print('Average: ', y)
```

Output:



```
===== RESTART: C:/Users/user/AppData
Enter numbers separated by space:
10 20 30 40 51
Total:  151
Average:  30.2
>>>
```

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)

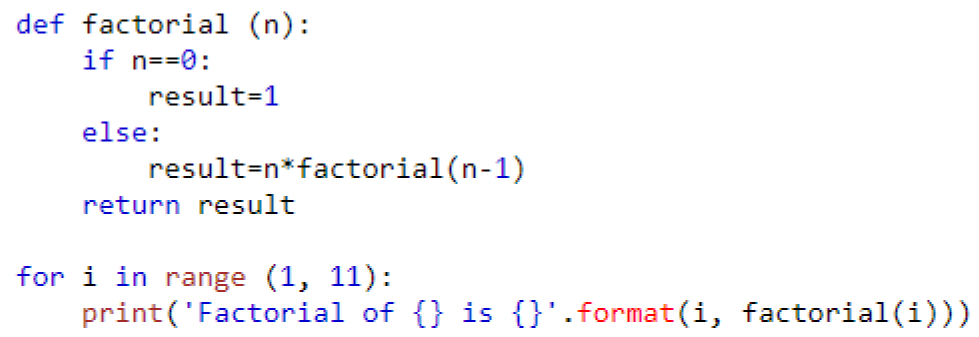
Now, if we know that the factorial(0) value is 1, all the preceding statements and give the result as:

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 ‘n' as:

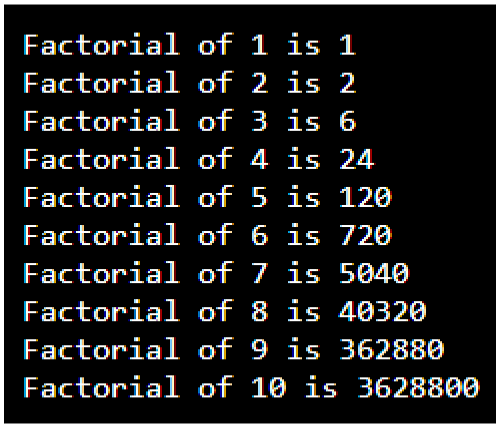
factorial(n) = n* factorial(n-1).

When we observe this formula, we can understand that the factorial() function that calculates factorial of 'n' value will call itself to calculate factorial of 'n-1' value. So, we implement the factorial() function using recursion, as shown in the program below.



```
def factorial (n):
    if n==0:
        result=1
    else:
        result=n*factorial(n-1)
    return result

for i in range (1, 11):
    print('Factorial of {} is {}'.format(i, factorial(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
Factorial of 7 is 5040
Factorial of 8 is 40320
Factorial of 9 is 362880
Factorial of 10 is 3628800
```

Anonymous Functions or Lambdas

A function without a name is called an anonymous function. So far, the functions 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 also called Lambda functions. Let's take a normal function that returns a 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 observe that we did not use any name for the function here. So, the format of lambda functions

lambda argument_list: expression

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

y = square (5)

But, lambda functions return a function and hence they should be assigned to a function as:

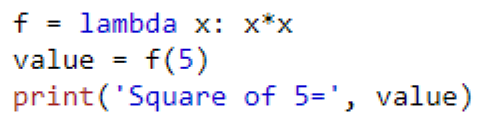
f= lambda x: x*x

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

value = f(5)

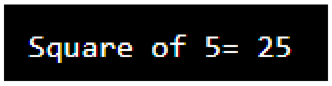
Now, 'value' contains the square value of 5, i.e. 25. This is shown in the program below.

A Python program to create a lambda function that returns a square value of a given number.



```
f = lambda x: x*x
value = f(5)
print('Square of 5=', value)
```

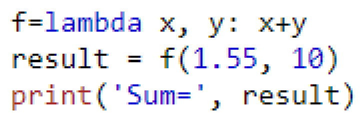
Output:



```
Square of 5= 25
```

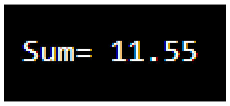
Lambda functions contain only one expression and they return the result implicitly Hence we should not write any return' statement in the lambda functions. Here is a lambda function that calculates the sum of two numbers.

A lambda function to calculate the sum of two numbers.



```
f=lambda x, y: x+y
result = f(1.55, 10)
print('Sum=', result)
```

Output:



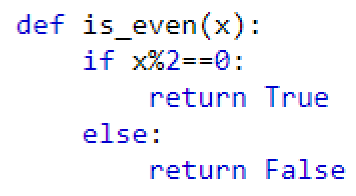
```
Sum= 11.55
```

Using Lambdas with filter() Function

The filter() function is useful to filter out the elements of a sequence depending on the result of a function. We should supply a function and a sequence to the filter() func as:

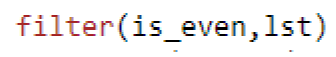
filter (function, sequence)

Here, the ‘function’ represents a function name that may return either True or False; and ‘sequence' represents a list, string or tuple. The ‘function' is applied to every element of the ‘sequence' and when the function returns True, the element is extracted otherwise it is ignored. Before using the filter() function, let's first write a function that tests whether a given number is even or odd.



```
def is_even(x):
    if x%2==0:
        return True
    else:
        return False
```

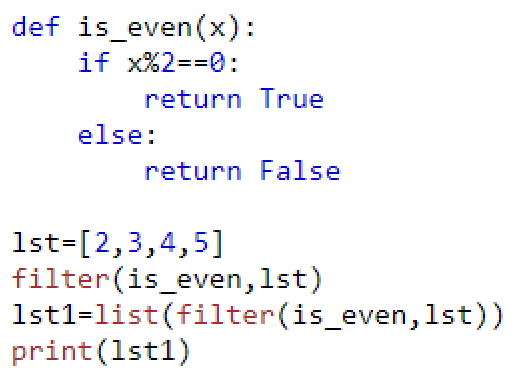
Now, we can use this function inside filter() to test the elements of a list ‘lst' as;



```
filter(is_even,lst)
```

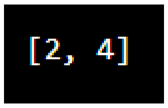
Now, the is_even() function acts on every element of the list ‘lst' and returns only those elements which are even. The resultant element can be stored into another list. This is shown in the program below.

A Python program using filter() to filter out even numbers from a list.



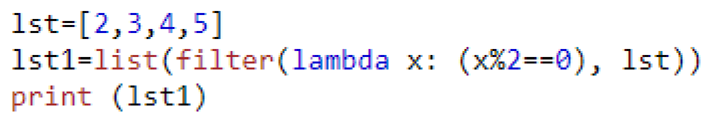
```
def is_even(x):
    if x%2==0:
        return True
    else:
        return False

lst=[2,3,4,5]
filter(is_even,lst)
lst1=list(filter(is_even,lst))
print(lst1)
```

Output:



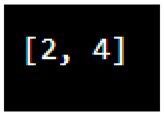
```
[2, 4]
```

Passing the lambda function to filter() function is more elegant. We will rewrite the program using the lambda function.



```
lst=[2,3,4,5]
lst1=list(filter(lambda x: (x%2==0), lst))
print (lst1)
```

Output:



```
[2, 4]
```

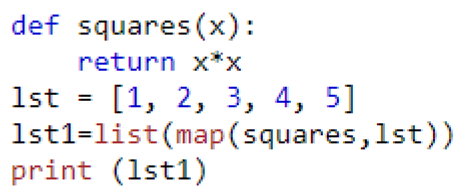
Using Lambdas with map() Function

The map() function is similar to the filter() function but it acts on each element of the sequence and perhaps changes the elements. The format of map() function is:

map(function, sequence)

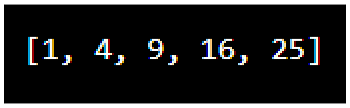
The ‘function' performs a specified operation on all the elements of the sequence and the modified elements are returned which can be stored in another sequence. In the program below, we are using map() function to find squares of elements of a list.

A Python program to find squares of elements in a list.



```
def squares(x):
    return x*x
lst = [1, 2, 3, 4, 5]
lst1=list(map(squares,lst))
print (lst1)
```

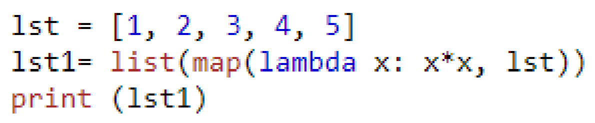
Output:



```
[1, 4, 9, 16, 25]
```

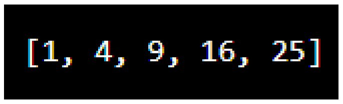
When the same program is rewritten using the lambda function, it becomes more elegant. This is done in the program below.

A lambda function that returns squares of elements in a list.



```
lst = [1, 2, 3, 4, 5]
lst1= list(map(lambda x: x*x, lst))
print (lst1)
```

Output:



```
[1, 4, 9, 16, 25]
```

It is possible to use map() function on more than one list if the lists are of the same length in this case, map() function takes the lists as arguments of the lambda function and does the operation. For example,

map(lambda x, y: x*y, lst1, lst2)

Here, the lambda function has two arguments ‘x' and 'y'. Hence, ‘x' represents ‘lst1' and ‘y' represents ‘lst2'. Since lambda is showing x*y, the respective elements from lst1 and are multiplied and the product is returned.

Using Lambdas with reduce() Function

The reduce() function reduces a sequence of elements to a single value by processing elements according to a function supplied. The reduce() function is uses in the format:

reduce(function, sequence)

For example, we write the reduce() function with a lambda expression, as:

```
lst = [1, 2, 3, 4, 5]
reduce(lambda x, y: x*y, lst)
```

Here, the reduce() function reduces the list to a final value as indicated by the lambda function. The lambda function is taking two arguments and returning their product. Hence, starting from the 0th element of the list lst, the first two elements are multiplied and the product is obtained. Then this product is multiplied with the third element and the product is obtained. Again this product is multiplied with the fourth element and so on. The final product value is returned.

```
from functools import *
lst=[1, 2, 3, 4, 5]
result=reduce(lambda x, y: x*y, lst)
print (result)
```

Output:

```
120
```

Since reduce() function belongs to functools module in Python, we are importing all from that module using the following statement in the above program:

from functools import *

As another example, to calculate the sum of numbers from 1 to 50, we can write reduce() function with a lambda function, as:

```
from functools import *
sum=reduce(lambda a, b: a+b, range(1, 51))
print (sum)
```

Output:

```
1275
```

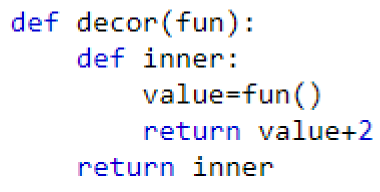
Function Decorators

A decorator is a function that accepts a function as a 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. Decorators concept is a bit confusing but not difficult to understand. The following steps are generally involved in creation of decorators:

1. We should define a decorator function with another function name as a parameter. As an example, let's define a decorator function decor() with ‘fun' as a parameter.

   def decor(fun):

2. 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. As an example, let's write the inner() function in the decor() function. Our assumption is that this inner() function increases the value returned by the function by 2.

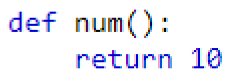


```
def decor(fun):
    def inner:
        value=fun()
        return value+2
    return inner
```

   In the previous code, observe the body of the inner() function. We have accessed the ‘value' returned by the function ‘fun', and added 2 to it and then returned it.

3. Return the inner function that has processed or decorated the value. In our example, in the last statement, we were returning the inner() function using return statement. With this, the decorator is completed.

   The next question is how to use the decorator. Once a decorator is created, it can be used for any function to decorate or process its result. For example, let's take the num() function that returns some value, e.g. 10.



```
def num():
    return 10
```

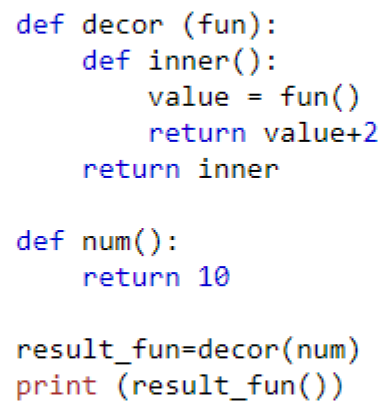
   Now, we should call decor() function by passing num() function name as:

   result_fun=decor (num)

   Now the name 'num' is copied into the parameter of the decor function. Thus, "fun" refers to “num". In the inner() function, ‘value' represents 10 and it is incremented by 2. The returned function ‘inner' is referenced by ‘result_fun'. So, 'result_fun' indicates the resultant function. Call this function and print the result as:
   print (result_fun())

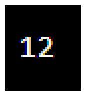
   This will display 12. Thus the value returned by the num() function (i.e. 10) is incremented by 2 by the decor() function. Consider the following program..

A decorator to increase the value of a function by 2.



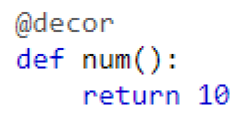
```python
def decor (fun):
    def inner():
        value = fun()
        return value+2
    return inner

def num():
    return 10

result_fun=decor(num)
print (result_fun())
```

Output:



```
12
```

To apply the decorator to any function, we can use the ‘@' symbol and decorator name just above the function definition. For example, to apply our decor() function to the num() function, we can write @decor statement above the function definition as:



```python
@decor
def num():
    return 10
```

It means the decor() function is applied to process or decorate the result of the num function. When we apply the decorator in this way, we need not call the decorator explicitly and pass the function name. Instead, we can call our num function naturally as:

print(num())

So, the symbol is useful to call the associated decorator internally, whenever the function is called. The same program revised version using the fa symbol is shown in the following program.

A Python program to apply a decorator to a function using @ symbol

```
def decor(fun):
    def inner():
        value=fun()
        return value+2
    return inner

@decor
def num():
    return 10
print(num())
```

Output:

```
12
```

We will extend this program where we are going to add one more decorator by the name ‘decor1()' that doubles the value of the function passed to it. That means we want to apply two decorators to the same function num() to decorate its result. Consider the following program:

A Python program to create two decorators.

```
def decor(fun):
    def inner():
        value=fun()
        return value+2
    return inner

def decor1(fun):
    def inner():
        value=fun()
        return value*2
    return inner
def num():
    return 10
result_fun=decor(decor1(num))
print(result_fun())
```

Output:

```
22
```

To apply the decorators to num() function using a symbol, we can rewrite the above program as below.
A Python program to apply two decorators to the same function using @ symbol.

```
def decor(fun):
    def inner():
        value=fun()
        return value+2
    return inner

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

Output:

```
22
```

So, the syntax of decorators is:

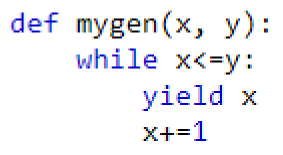
```
@dec1
@dec2
def func(arg1, arg2, ...):
    pass
```

This is equivalent to:

```
def func(arg1, arg2, ...):
    pass
func=dec1(dec2(func))
```

Generators

Generators are functions that return a sequence of values. A generator function is written like an ordinary function but it uses a ‘yield' statement. This statement is useful to return the value. For example, let's write a generator function to return numbers from x to y.



```
def mygen(x, y):
    while x<=y:
        yield x
        x+=1
```

When we call this function by passing 5 and 10 as:

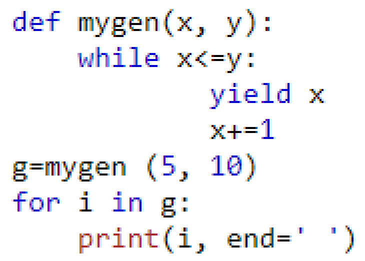
g=mygen(5, 10)

Then the mygen() function returns a generator object that contains a sequence of numbers as returned by yield' statement. So, it refers to the generator object with the sequence of numbers from 5 to 10. We can display the numbers from 5 to 10. We can display the numbers from ‘g’ using a for loop as:

for i in g:
print(i, end=' ')

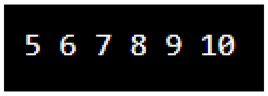
In the following program, we are creating a generator function that generates numbers from x to y and displaying those numbers.

A Python program to create a generator that returns a sequence of numbers from x to y



```
def mygen(x, y):
    while x<=y:
            yield x
            x+=1
g=mygen (5, 10)
for i in g:
    print(i, end=' ')
```

Output:



```
5 6 7 8 9 10
```

Once the generator object is created, we can store the elements of the generator into a list and use the list as we want. For example, to store the numbers of generator ‘g' into a list ‘lst' we can using list() function as:

```
lst = list(g)
```

Now, the list ‘lst' contains the elements:
[5, 6, 7, 8, 9, 10].

If we want to retrieve element by element from a generator object, we can use next() function as:

```
print (next(g))
```

will display the first element in 'g'. When we call the above function the next time, it will display the second element in ‘g'. Thus by repeatedly calling the next() function, we will be able to display all the elements of 'g'. In the following program, a simple generator is created that returns 'A', 'B', 'C’.

```
def mygen():
yield 'A'
yield 'B'
yield 'C'
```

Here, mygen() is returning 'A', 'B', 'C' using yield statements. So, the yield statement returns the elements from a generator function into a generator object. So, when we call mygen() function as:

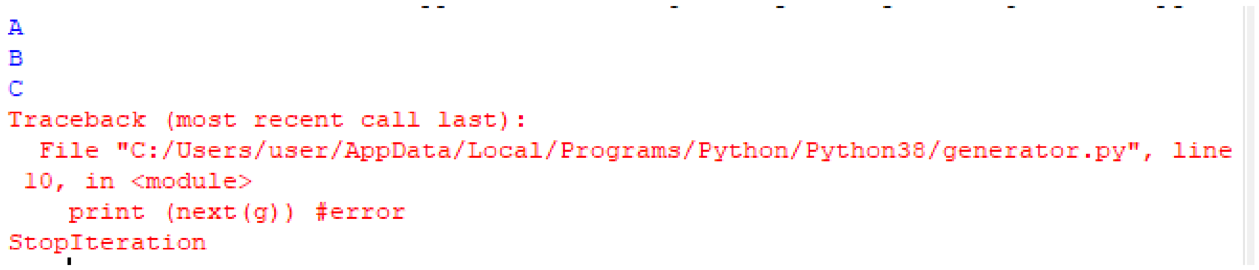
```
g = mygen()
```

The characters 'A',' B ','C' are contained in the generator object 'g'. Using next(g) we can refer to these elements. Consider the following program.

A generator that returns characters from A to C.

```
def mygen ():
    yield 'A'
    yield 'B'
    yield 'C'

g = mygen()
print (next(g)) #display 'A'
print(next (g)) #display 'B'
print(next (g)) #display 'C'
print (next(g)) #error
```

Output:



```
A
B
C
Traceback (most recent call last):
  File "C:/Users/user/AppData/Local/Programs/Python/Python38/generator.py", line
 10, in <module>
    print (next(g)) #error
StopIteration
```

Modules

What are modules in Python?

Modules refer to a file containing Python statements and definitions. A file containing Python code, for example:Pgm.py, is called a module, and its module name would be Pgm.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.We can define our most used functions in a module and import it, instead of copying their definitions into different programs.
Let us create a module.

```
def add(a, b):
   result = a + b
   return result
```

Here, we have defined a function add() inside a module named Pgm. The function takes in two numbers and returns their sum.

How to import modules in Python?

We can import the definitions inside a module to another module or the interactive interpreter in Python. We use the import keyword to do this. To import our previously defined module Pgm, we type the following in the Python prompt.

```
import Pgm
```

This does not import the names of the functions defined in example directly in the current symbol table. It only imports the module name Pgm there.

Using the module name we can access the function using the dot . operator. For example:

```
>>> import Pgm
>>> Pgm.add(4,5.5)
```

Output:

```
9.5
```

Python has tons of standard modules. You can check out the full list of [Python standard modules]() and their use cases. These files are in the Lib directory inside the location where you installed Python.

Standard modules can be imported the same way as we import our user-defined modules. There are various ways to import modules. They are listed below..

Python import statement

We can import a module using the import statement and access the definitions inside it using the dot operator as described above. Here is an example.

```
import math
print("The value of pi is", math.pi)
```

When you run the program, the output will be:

```
The value of pi is 3.141592653589793
```

Import with renaming

We can import a module by renaming it as follows:

```
import math as m
print("The value of pi is", m.pi)
```

Output:

```
The value of pi is 3.141592653589793
```

We have renamed the math module as m. This can save us typing time in some cases.
Note that the name math is not recognized in our scope. Hence, math.pi is invalid, and m.pi is the correct implementation.

Python from...import statement

We can import specific names from a module without importing the module as a whole. Here is an example.

```
from math import pi
print("The value of pi is", pi)
```

Here, we imported only the pi attribute from the math module.
In such cases, we don't use the dot operator. We can also import multiple attributes as follows:

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>>
```

Import all names

We can import all names(definitions) from a module using the following construct:

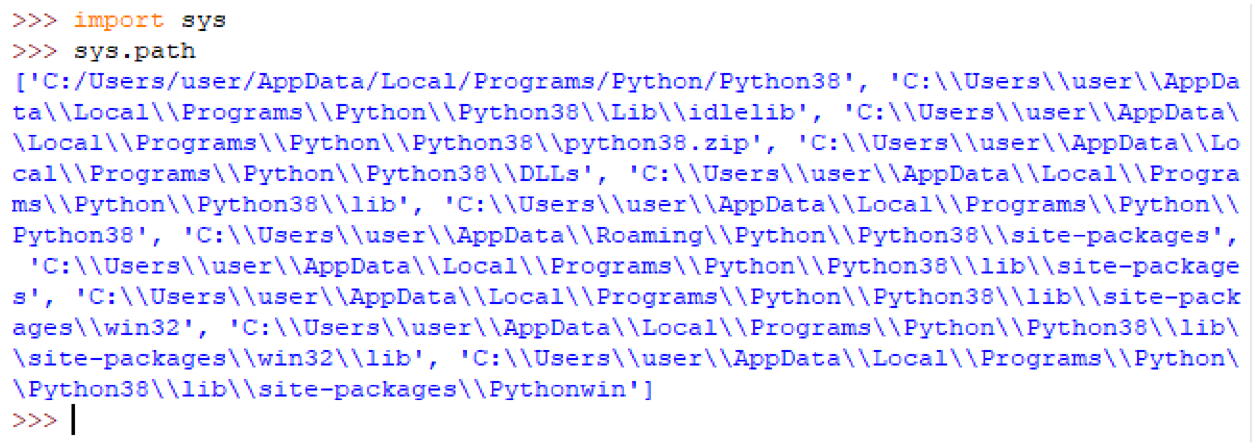
```
from math import *
print("The value of pi is", pi)
```

Here, we have imported all the definitions from the math module. This includes all names visible in our scope except those beginning with an underscore(private definitions).
Importing everything with the asterisk (*) symbol is not a good programming practice. This can lead to duplicate definitions for an identifier. It also hampers the readability of our code.

Python Module Search Path

While importing a module, Python looks at several places. Interpreter first looks for a built-in module. Then(if a built-in module is not found), Python looks into a list of directories defined in sys.path. The search is in this order.

- The current directory.
- PYTHONPATH (an environment variable with a list of directories).
- The installation-dependent default directory.



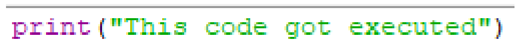
```
>>> import sys
>>> sys.path
['C:/Users/user/AppData/Local/Programs/Python/Python38', 'C:\\Users\\user\\AppDa
ta\\Local\\Programs\\Python\\Python38\\Lib\\idlelib', 'C:\\Users\\user\\AppData\
\Local\\Programs\\Python\\Python38\\python38.zip', 'C:\\Users\\user\\AppData\\Lo
cal\\Programs\\Python\\Python38\\DLLs', 'C:\\Users\\user\\AppData\\Local\\Progra
ms\\Python\\Python38\\lib', 'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\
Python38', 'C:\\Users\\user\\AppData\\Roaming\\Python\\Python38\\site-packages',
 'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\Python38\\lib\\site-package
s', 'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\Python38\\lib\\site-pack
ages\\win32', 'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\Python38\\lib\
\site-packages\\win32\\lib', 'C:\\Users\\user\\AppData\\Local\\Programs\\Python\
\Python38\\lib\\site-packages\\Pythonwin']
>>> |
```

We can add and modify this list to add our own path.

Reloading a module

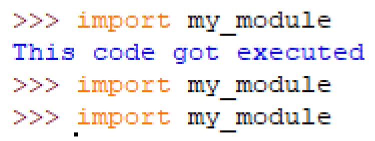
The Python interpreter imports a module only once during a session. This makes things more efficient. Here is an example to show how this works.

Suppose we have the following code in a module named my_module.



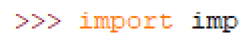
```
print("This code got executed")
```

Now we see the effect of multiple imports.

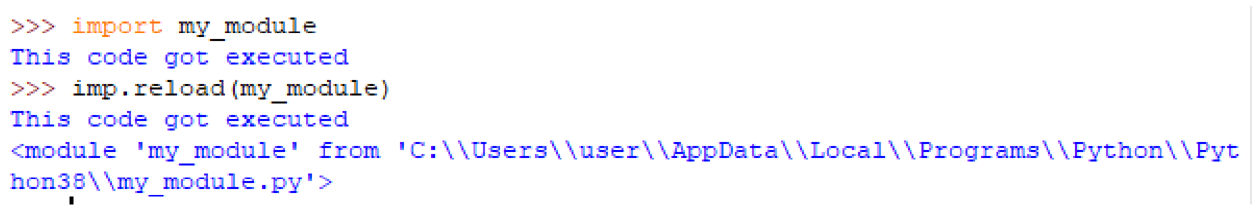


```
>>> import my_module
This code got executed
>>> import my_module
>>> import my_module
```

We can see that our code got executed only once. This goes to say that our module was imported only once.
Now if our module changed during the course of the program, we would have to reload it.One way to do this is to restart the interpreter. But this does not help much.
Python provides a more efficient way of doing this. We can use the reload() function inside the imp module to reload a module. We can do it in the following ways:



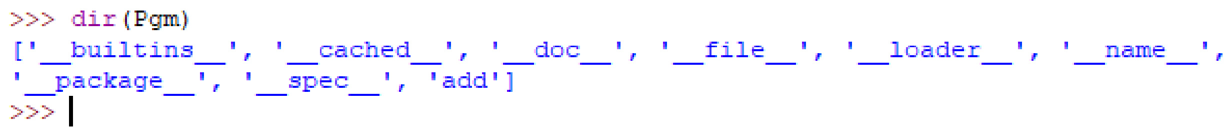
```
>>> import imp
```



```
>>> import my_module
This code got executed
>>> imp.reload(my_module)
This code got executed
<module 'my_module' from 'C:\\Users\\user\\AppData\\Local\\Programs\\Python\\Pyt
hon38\\my_module.py'>
```

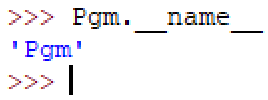
The dir() built-in function

We can use the dir() function to find out names that are defined inside a module.
For example, we have defined a function add() in the module Pgm that we had in the beginning.
We can use dir in Pgm module in the following way:



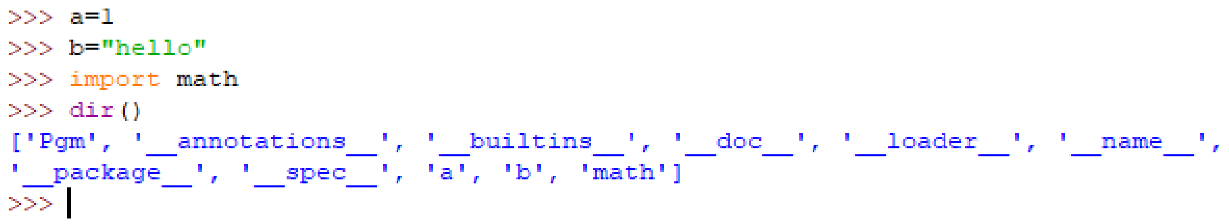
```
>>> dir(Pgm)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'add']
>>> 
```

Here, we can see a sorted list of names (along with add). All other names that begin with an underscore are default Python attributes associated with the module (not user-defined).
For example, the __name__ attribute contains the name of the module.



```
>>> Pgm.__name__
'Pgm'
>>> 
```

All the names defined in our current namespace can be found out using the dir() function without any arguments.



```
>>> a=1
>>> b="hello"
>>> import math
>>> dir()
['Pgm', '__annotations__', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'b', 'math']
>>> 
```

Creating our Own Modules in Python

A module represents a group of classes, methods, functions and variables. While we are developing software, there may be several classes, methods and functions. We should first group them depending on their relationship into various modules and later use these modules in the other programs. It means, when a module is developed, it can be reused in any program that needs that module.

In Python, we have several built-in modules like sys, Io, time etc. Just like these modules, we can also create our own modules and use them whenever we need them. Once a module is created, any programmer in the project team can use that module.Hence, modules will make software development easy and faster.

Now, we will create our own module by the name 'employee' and store the functions da(), hra(), pf() and itax() in that module. Please remember we can also store classes and variables etc. in the same manner in any module. So, please type the following functions and save the file as ‘employee.py’.

```
def da(basic):
        da=basic*80/100
        return da

def hra(basic):
        hra=basic*15/100
        return hra

def pf(basic):
        pf=basic*12/100
        return pf

def itax(gross):
        tax=gross*0.1
        return tax
```

Now, we have our own module by the name “employee.py". This module contains 4 functions which can be used in any program by importing this module. To import the module, we can write:

```
>>> import employee
```

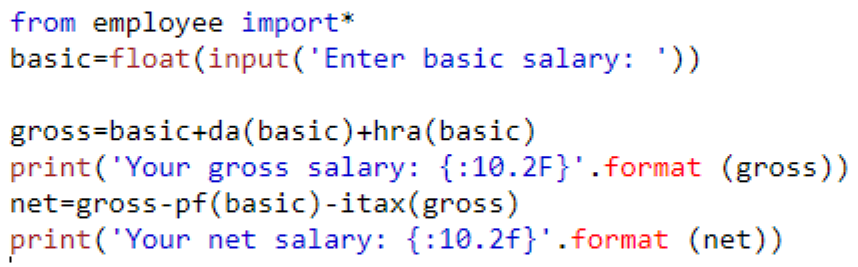
In this case, we have to refer to the functions by adding the module name as:

employee.da(), employee.hra(), employee.pf(), employee.itax(). This is a bit cumbersome and hence we can use another type of import statement as:

```
>>> from employee import *
```

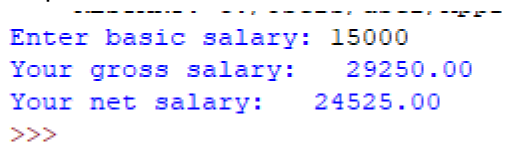
In this case, all the functions of the ‘employee’ module are referenced and hence, we can refer to them without using the module name, simply as: da(), hra(), pf() and itax(). Now see the following program, where we are using the 'employee' module and calculating gross and net salaries of an employee.

A Python program that uses the functions of the employee module and calculates the gross and net salaries of an employee. using employee module to calculate gross and net salaries of an employee



```
from employee import*
basic=float(input('Enter basic salary: '))

gross=basic+da(basic)+hra(basic)
print('Your gross salary: {:10.2F}'.format (gross))
net=gross-pf(basic)-itax(gross)
print('Your net salary: {:10.2f}'.format (net))
```

Output:



```
Enter basic salary: 15000
Your gross salary:   29250.00
Your net salary:   24525.00
>>>
```

The Special Variable_name__

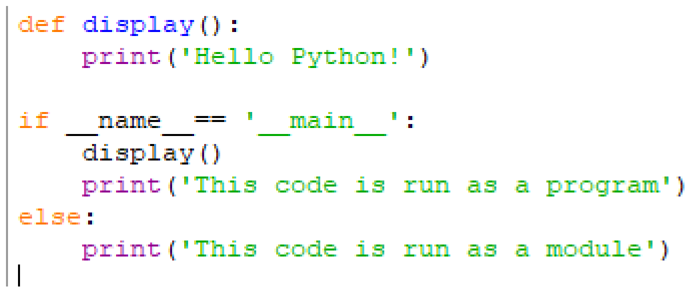
When a program is executed in Python, there is a special variable internally created by the name '__name__'. This variable stores information regarding whether the program is executed as an individual program or as a module. When the program is executed directly, the Python interpreter stores the value '__main__' into this variable. When the program is imported as a module into another program, then the Python interpreter stores the module name into this variable. Thus, by observing the value of the variable '__name__' we can understand how the program is executed.

Let's assume that we have written a program. When this program is run, Python. interpreter stores the value '__main__' into the special variable'__name__'. Hence, we can check if this program is run directly as a program or not as:

```
if_name == '__main__' :
 print('This code is run as a program')
```

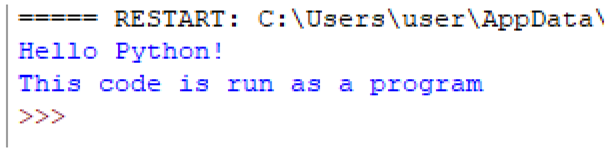
Suppose, if __name__' variable does not contain the value __main_', it represents that this code is run as a module from another external program. To understand this concept, first we will write a program with a display() function that displays some message as 'Hello Python!'. This code is shown in the program below.

A Python program using the special '__name__' variable.



```
def display():
    print('Hello Python!')

if __name__== '__main__':
    display()
    print('This code is run as a program')
else:
    print('This code is run as a module')
```

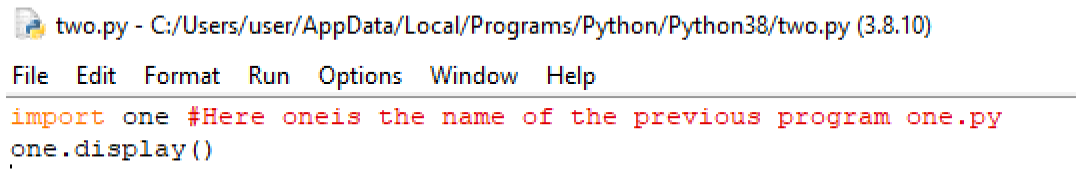
Output:



```
===== RESTART: C:\Users\user\AppData\
Hello Python!
This code is run as a program
>>>
```

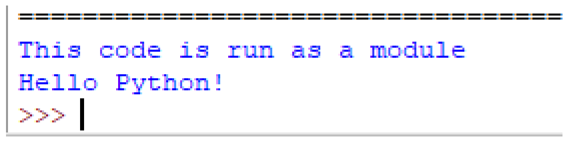
So, the code in the program above is run directly by the Python interpreter and hence it is saying that this code is run as a program. Now, we will write another program (Program 46) where we want to import the above program as a module and call the display() function.

A Python program that imports the previous Python program as a module.


two.py - C:/Users/user/AppData/Local/Programs/Python/Python38/two.py (3.8.10)
File Edit Format Run Options Window Help

```
import one #Here oneis the name of the previous program one.py
one.display()
```

Output:



```
==================================
This code is run as a module
Hello Python!
>>>
```

In the above program, when the first statement: 'import one' is executed by the Python interpreter, it will set the value of the special variable '__name__' as 'one' (i.e. the module name) in one.py program and hence we got the output as: 'This code is run as a module'.