**Functions, Modules**

**Python Method**

1. Method is called by its name, but it is associated to an object(dependent).
2. A method is implicitly passed the object on which it is invoked.
3. It may or may not return any data.
4. A method can operate on the data (instance variables) that is contained by the corresponding class

**Basic Method Structure in Python :**

|  |
| --- |
| class class\_name      def method\_name () :          ......          # method body          ...... |

**User-Defined Method :**

|  |
| --- |
| class ABC :      def method\_abc (self):          print("I am in method\_abc of ABC class. ")    class\_ref = ABC() # object of ABC class  class\_ref.method\_abc() |

Output:

I am in method\_abc of ABC class

**Inbuilt method :**

|  |
| --- |
| import math   ceil\_val = math.ceil(15.25)  print( "Ceiling value of 15.25 is : ", ceil\_val) |

Output:

Ceiling value of 15.25 is : 16

**Functions**

1. Function is block of code that is also **called by its name**. (independent)
2. The function can have different parameters or may not have any at all. If **any data (parameters)** are passed, they are **passed explicitly**.
3. It **may or may not return any data.**
4. Function does not deal with Class and its instance concept.

**Basic Function Structure in Python :**

|  |
| --- |
| def function\_name ( arg1, arg2, ...) :      ......      # function body      ...... |

**User-Defined Function :**

|  |
| --- |
| def Subtract (a, b):      return (a-b)   print( Subtract(10, 12) ) # prints -2   print( Subtract(15, 6) ) # prints 9 |

**Output:**

-2

9

**Inbuilt Function :**

|  |
| --- |
| s = sum([5, 15, 2])  print( s ) # prints 22   mx = max(15, 6)  print( mx ) # prints 15 |

**Output:**

22

15

**Difference between method and function**

1. Simply, function and method both look similar as they perform in almost similar way, but the key difference is the concept of ‘**Class and its Object**‘.
2. Functions can be called **only by its name**, as it is defined independently. But methods c**an’t be called by its name**only, we need to invoke the class by a reference of that class in which it is defined, i.e. method is defined within a class and hence they are dependent on that class.

**Functions:**

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

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

**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]

**Creating a Function**

In Python a function is defined using the def keyword:

**Example**

def my\_function():  
  print("Hello from a function")

**Calling a Function**

To call a function, use the function name followed by parenthesis:

**Example**

def my\_function():  
  print("Hello from a function")  
my\_function()

**Parameters**

Information can be passed to functions as parameter.Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

**Example**

def my\_function(fname):  
  print(fname + " Refsnes")  
my\_function("Emil")  
my\_function("Tobias")  
my\_function("Linus")

**Default Parameter Value**

If we call the function without parameter, it uses the default value:

Example

def my\_function(country = "Norway"):  
  print("I am from " + country)  
  
my\_function("Sweden")  
my\_function("India")  
my\_function()  
my\_function("Brazil")

## Return Values

To let a function return a value, use the return statement:

### Example

def my\_function(x):  
  **return 5 \* x**print(my\_function(3))  
print(my\_function(5))  
print(my\_function(9))

**Types of Functions:**

a) Built-in Functions: Functions that are predefined. We have used many predefined functions in Python.

b) User- Defined: Functions that are created according to the requirements.

**Calling a Function**

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt.

**Example:-**

def printme( str ):

print str

return;

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

**Output:-**

I'm first call to user defined function!

Again second call to the same function

**Pass by reference**

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

**For example −**

#!/usr/bin/python

def changeme( mylist ):

mylist.append([1,2,3,4]);

print "Values inside the function: ", mylist

return

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

**Output:-**

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

#!/usr/bin/python

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assig new reference in mylist

print "Values inside the function: ", mylist

return

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

**Output:-**

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

**Advantages of functions:-**

1. User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
2. If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
3. Programmars working on large project can divide the workload by making different functions.

**Formal and Actual Parameters**

An *identifier* is a name used for a class, a variable, a method, or a parameter. The following definitions are useful:

**formal parameter** — the identifier used in a method to stand for the value that is passed into the method by a caller.

* + For example, amount is a formal parameter of processDeposit

**actual parameter** — the actual value that is passed into the method by a caller.

* + For example, the 200 used when processDeposit is called is an actual parameter.
  + actual parameters are often called **arguments**

When a method is called, the formal parameter is temporarily "bound" to the actual parameter. The method uses the formal parameter to stand for the actual value that the caller wants to be used.

**For example**:-

here the processDeposit method uses the formal parameter amount to stand for the actual value used in the procedure call:

balance = balance + amount ;

***Formal parameter*** -- The parameter defined as part of the function definition, e.g., x in:      
     def cube(**x**):   
        return x\*x\*x   
  ***Actual Parameter*** -- The actual data sent to a function. It's found in the function call, e.g., 7 in      
   result = cube(**7**)   
  for y in:   
     y=5   
    res = cube(**y**)

**The *Anonymous* Functions**

 anonymous function is a [function](https://www.programiz.com/python-programming/function) that is defined without a name.While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called **lambda 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.

**Syntax**

lambda [arg1 [,arg2,.....argn]]:expression

**Example:-**

sum = lambda arg1, arg2: arg1 + arg2;

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )

**Output:-**

Value of total : 30

Value of total : 40

**Example:-**

x = lambda a : a + 10  
print(x(5))

**Example**

A lambda function that multiplies argument a with argument b and print the result:

x = lambda a, b : a \* b  
print(x(5, 6))

**Why Use Lambda Functions?**

The power of lambda is better shown when you use them as an anonymous function inside another function.Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

def myfunc(n):  
  return lambda a : a \* n

Use that function definition to make a function that always doubles the number you send in:

**Example**

def myfunc(n):  
  return lambda a : a \* n  
mydoubler = myfunc(2)  
print(mydoubler(11))

Or, use the same function definition to make a function that always *triples* the number you send in:

**Example**

def myfunc(n):  
  return lambda a : a \* n  
mytripler = myfunc(3)  
print(mytripler(11))

**Scope of Variables**

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python

1)Global variables 2)Local variables

**1) Local Variables:**

Variables declared inside a function body is known as Local Variable. These have a local access thus these variables cannot be accessed outside the function body in which they are declared.

**Example:**

**def** msg():

           a=10

**print** "Value of a is",a

**return**

msg()

**print** a #it will show error since variable is local

**b) Global Variable:**

Variable defined outside the function is called Global Variable. Global variable is accessed all over program thus global variable have widest accessibility.

**Example:**

b=20

def msg():

a=10

print "Value of a is",a

print "Value of b is",b

return

msg()

print b

**Example:-**

total = 0; # This is global variable.

def sum( arg1, arg2 ):

total = arg1 + arg2; # Here total is local variable.

print "Inside the function local total : ", total

return total;

sum( 10, 20 );

print "Outside the function global total : ", total

**Output:-**

Inside the function local total : 30

Outside the function global total : 0

**Modules :-**

**What is Python Module**

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

### **Create a simple Python module**

create a simple calc.py in which we define two functions, one **add** and another **subtract**.

**Example**

# A simple module, calc.py

def add(x, y):

    return (x+y)

def subtract(x, y):

    return (x-y)

**Import Module in Python**

We can import the functions, and classes defined in a module to another module using the [import statement](https://www.geeksforgeeks.org/import-module-python/) in some other Python source **file.**When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches for importing a module. For example, to import the module calc.py, we need to put the following command at the top of the script.

**Syntax of Python Import**

import module

### **Importing modules in Python**

we are importing the **calc** that we created earlier to perform add operation.

**Example:-**

# importing  module calc.py

import calc

print(calc.add(10, 2))

## **The from-importStatement** in Python

Python’s from statement lets you import specific attributes from a module without importing the module as a whole.

**Importing specific attributes from the module**

we are importing specific sqrt and factorial attributes from the math module.

**Example:-**

from math import sqrt, factorial

print(sqrt(16))

print(factorial(6))

**Import all Names**

The \* symbol used with the from import statement is used to import all the names from a module to a current namespace.

**Syntax:-**

from module\_name import \*

**From import \*  Statement**

The use of \* has its advantages and disadvantages. If you know exactly what you will be needing from the module, it is not recommended to use \*, else do so.

**Example:-**

from math import \*

print(sqrt(16))

print(factorial(6))

**creating our own modules :-**

Writing a module is just like writing any other Python file. Modules can contain definitions of functions, classes, and variables that can then be utilized in other Python programs.

**Example:-**

hello.py

**def world**():

print("Hello, World!")

If we run the program on the command line with python hello.py nothing will happen since we have not told the program to do anything.

Let’s create a second file in the same directory called main\_program.py so that we can [import the module](https://www.digitalocean.com/community/tutorials/how-to-import-modules-in-python-3) we just created, and then call the function. This file needs to be in the same directory so that Python knows where to find the module since it’s not a built-in module.

**Example:-**

**main\_program.py**

# Import hello module

import hello

hello.world()

**Output**

Hello, World!

**Advantages:**

1) **Reusability:** Module can be used in some other python code. Hence it provides the facility of code reusability.

2) **Categorization:** Similar type of attributes can be placed in one module.

**working with built-in modules-**

**1)Math module:-**

Mathematical calculations may occasionally be required when dealing with certain fiscal or rigorous scientific tasks. Python has a math module that can handle these complex calculations. Both simple mathematical calculations like addition (+), and subtraction (-), and advanced mathematical calculations like trigonometric operations, and logarithmic operations can be performed by the functions in the math module.

## What is Math Module in Python?

Python has a built-in math module. It is a standard module, so we don't need to install it separately. We only have to import it into the program we want to use. We can import the module, like any other module of Python, using import math to implement the functions to perform mathematical operations

**Example:-**

import math

print(math.sqrt( 9 ))

## Constants in Math Module

The value of numerous constants, including pi and tau, is provided in the math module so that we do not have to remember them. Using these constants eliminates the need to precisely and repeatedly write down the value of each constant. The math module includes the following constants:

1. Euler's Number
2. Tau
3. Infinity
4. Pi
5. Not a Number (NaN)

**1) Euler's Number:-**

The value 2.71828182845 of Euler's number is returned by the math.e constant.

**Syntax:-**

math.e

**Code:-**

import math

print( "The value of Euler's Number is: ", math.e )

**Output:-**

**The value of Euler's Number is: 2.718281828459045**

### 2) Tau

The ratio of a circle's circumference to its radius is known as tau. The value tau returned by the tau constant is 6.283185307179586.

**Syntax:**

math.tau

**Code:-**

import math

print ( "The value of Tau is: ", math.tau )

**Output:-**

The value of Tau is: 6.283185307179586

**3)Infinity**

Infinity refers to anything limitless or never-ending in both directions of the actual number line. Numbers cannot adequately represent it. The math.inf returns positive infinity constant. We can use -math.inf to print negative infinity.

**Syntax**

math.inf

**Code:-**

import math

print( math.inf )

print( -math.inf )

**Output:-**

inf

-inf

**Code**

import math

print( math.inf > 10e109 )

print( -math.inf < -10e109 )

**Output:-**

True

True

### 4)Pi

Pi is known to everyone. It is mathematically represented as either the fraction 22/7 or the decimal number 3.14. math.pi gives the most accurate value of pi.

**Syntax of this is:**

math.pi

**Code**

import math

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

**Code 2**

import math

r = 4

pi\_value = math.pi

print(2 \* pi\_value \* r)

### 5)NaN

The math.nan gives us a floating-point nan (Not a Number) value. This amount is not a valid numeric value. Float("nan") and the nan constant are comparable.

**Code**

import math

print( math.nan )

**Output:-**

nan

**Mathematical Operations with Math Module**

The functions that are required in representation theory and number theory, such as calculating the factorial of an integer, will be covered in this part.

**Calculating the Ceiling and the Floor Value**

The terms "ceiling value" and "floor value" refer to the smallest integral value larger than the number and the largest integral value less than the number, respectively. The ceil() and floor() methods simplify calculating this

**Code**

import math

x = 4.346

print("The ceiling value of 4.346 is : ", end="")

print( math.ceil(x) )

print("The floor value of 4.346 is : ", end="")

print( math.floor(x) )

**Output:**

The ceiling value of 4.346 is : 5

The floor value of 4.346 is : 4

**Calculating the Factorial of the Number:-**

We may determine the factorial of a given integer in a one-liner code by using the math.factorial() function. The Python interpreter will send a message if the given number is not integral.

**Code**

import math

x = 6

print( "The factorial of 6 is : ", math.factorial(x) )

try:

print( "The factorial of 6.5 in: ", math.factorial(6.5) )

except:

print( "Cannot calculate factorial of a non-integral number" )

**Output:**

The factorial of 6 is : 720

Cannot calculate factorial of a non-integral number

**Calculating the Absolute Value**

The method math.fabs() returns the absolute number of the number given to the function.

**Code**

import math

x = -45

print( "The absolute value of -45 is: ", math.fabs(x) )

**Output:**

The absolute value of -45 is: 45.0

**Calculating the Exponential:-**

x to the power of e, often known as the exponential of a number x, is calculated using the exp() function.

**Code**

import math

num1 = 4

num2 = -3

num3 = 0.00

print( f"The exponenetial value of {num1} is: ", math.exp(num1) )

print( f"The exponenetial value of {num2} is: ", math.exp(num2) )

print( f"The exponenetial value of {num3} is: ", math.exp(num3) )

**Output:**

The exponenetial value of 4 is: 54.598150033144236

The exponenetial value of -3 is: 0.049787068367863944

The exponenetial value of 0.0 is: 1.0

**Calculating the Power of a Number**

x\*\*y is computed via the pow() function. This function calculates the value of the power after converting its inputs into floats.

**Code**

import math

x = 4

y = 5

print( f"The value of {x} to the power of {y} is: ", math.pow(x,y) )

**Output:**

The value of 4 to the power of 5 is: 1024.0

**Calculating Sine, Cosine, and Tangent:-**

The values of sine, cosine, and tangent of an angle, which are supplied as an input to the function, are returned by the sin(), cos(), and tan() methods. This function expects a value that is provided in radians.

**Code**

import math

angle = math.pi / 4

print( "The sine of pi/4 is : ", math.sin( angle ) )

print( "The cosine of pi/4 is : ", math.cos( angle ) )

print("The tangent of pi/4 is : ", math.tan( angle ))

**Output:**

The sine of pi/4 is : 0.7071067811865475

The cosine of pi/4 is : 0.7071067811865476

The tangent of pi/4 is : 0.9999999999999999

**The dir( ) Function:-**

A sorted list of strings comprising the identifiers of the functions defined by a module is what the built-in method dir() delivers.

The list includes the names of modules, each specified constants, functions, and methods.

**Code:-**

import math

functions = dir(math)

print( functions )

**Output:-**

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', '

**2) Random Module:-**

Python **Random module** is an in-built module of Python which is used to generate random numbers. These are pseudo-random numbers means these are not truly random. This module can be used to perform random actions such as generating random numbers, print random a value for a list or string, etc.

**Example:-  Printing a random value from a list**

import random

list1 = [1, 2, 3, 4, 5, 6]

print(random.choice(list1)

**Example**: Creating random numbers with seeding value

import random

random.seed(5)

print(random.random())

print(random.random())

## 1)Creating Random Integers

[random.randint()](https://www.geeksforgeeks.org/python-randint-function/) method is used to generate random integers between the given range.

**Syntax :**

randint(start, end)

**Example:**Creating random integers

import random

r1 = random.randint(5, 15)

print("Random number between 5 and 15 is % s" % (r1))

r2 = random.randint(-10, -2)

print("Random number between -10 and -2 is % d" % (r2))

**Output:**

Random number between 5 and 15 is 7

Random number between -10 and -2 is -9

## 2) Creating Random Floats

[random.random()](https://www.geeksforgeeks.org/random-random-function-in-python/) method is used to generate random floats between 0.0 to 1.

**Syntax:**

random.random()

**Example:**

from random import random

print(random())

**3) Selecting Random Elements**

[random.choice()](https://www.geeksforgeeks.org/python-numbers-choice-function/) function is used to return a random item from a list, tuple, or string.

**Syntax:**

random.choice(sequence)

**Example:**Selecting random elements from the list, string, and tuple

import random

list1 = [1, 2, 3, 4, 5, 6]

print(random.choice(list1))

string = "geeks"

print(random.choice(string))

tuple1 = (1, 2, 3, 4, 5)

print(random.choice(tuple1))

## 4) Shuffling List

[random.shuffle()](https://www.geeksforgeeks.org/random-shuffle-function-in-python/) method is used to shuffle a sequence (list). Shuffling means changing the position of the elements of the sequence. Here, the shuffling operation is inplace.

**Syntax:**

random.shuffle(sequence, function)

**Example:**Shuffling a List

import random

sample\_list = [1, 2, 3, 4, 5]

print("Original list : ")

print(sample\_list)

random.shuffle(sample\_list)

print("\nAfter the first shuffle : ")

print(sample\_list)

random.shuffle(sample\_list)

print("\nAfter the second shuffle : ")

print(sample\_list)

## 5) Random Seed:-

We normally use the time of the system to ensure that the software delivers a different output each time we execute it because pseudorandom synthesis is dependent on the preceding number. As a result, we employ seeds.

We can specify a seed to have an initial number using Python's random.seed() function. This seed number determines a random number generator's outcome; therefore, if it stays the same, the outcome will continue to be the same.

## Example:-

import random

random.seed(2)

print('Generating 5 random numbers: ')

print([ random.randint(1, 300) for r in range(6)])

random.seed(2)

print([random.randint(1, 300) for i in range(6)])

**3) Time module:-**

Python time module allows to work with time in Python. It allows functionality like getting the current time, pausing the Program from executing, etc. So before starting with this module we need to import it.

## Importing time module

The time module comes with Python’s standard utility module, so there is no need to install it externally. We can simply import it using the [import statement](https://www.geeksforgeeks.org/import-module-python/).

import time

## What is epoch?

The epoch is the point where the time starts and is platform-dependent. On Windows and most Unix systems, the epoch is January 1, 1970, 00:00:00 (UTC), and leap seconds are not counted towards the time in seconds since the epoch. To check what the epoch is on a given platform we can use [time.gmtime(0)](https://www.geeksforgeeks.org/python-time-gmtime-method/).

**Example:-**

import time

print(time.gmtime(0))

**Example: Current time in seconds since epoch**

import time

curr = time.time()

print("Current time in seconds since epoch =", curr)

**Example: Getting time string from seconds**

import time

curr = time.ctime(1627908313.717886)

print("Current time:", curr)

## time.localtime() method :-

[localtime()](https://www.geeksforgeeks.org/python-time-localtime-method/) method returns the struct\_time object in local time. It takes the number of seconds passed since epoch as an argument. If the seconds parameter is not given then the current time returned by time.time() method is used.

**Example:**Getting local time from epoch

import time

obj = time.localtime(1627987508.6496193)

print(obj)

## time.mktime() method :-

[time.mktime()](https://www.geeksforgeeks.org/python-time-mktime-method/) is the inverse function of time.localtime() which converts the time expressed in seconds since the epoch to a time.struct\_time object in local time.

**Example:**Converting the struct\_time object to seconds since epoch

import time

obj1 = time.gmtime(1627987508.6496193)

time\_sec = time.mktime(obj1)

print("Local time (in seconds):", time\_sec)

## time.gmtime() method:-

[time.gmtime()](https://www.geeksforgeeks.org/python-time-gmtime-method/) is used to convert a time expressed in seconds since the epoch to a time.struct\_time object in UTC in which tm\_isdst attribute is always 0. If the seconds parameter is not given then the current time returned by time.time() method is used.

**Example:**Use of time.gmtime() method

import time

obj = time.gmtime(1627987508.6496193)

print(obj)

## time.strftime() method:-

[time.strftime()](https://www.geeksforgeeks.org/time-strftime-function-in-python/) function converts a tuple or struct\_time representing a time as returned by gmtime() or localtime() to a string as specified by the format argument. If t is not provided, the current time as returned by localtime() is used. The format must be a string. ValueError is raised if any field in t is outside of the allowed range.

**Example:**Converting struct\_time object to a string using strftime() method

from time import gmtime, strftime

s = strftime("%a, %d %b %Y %H:%M:%S",

             gmtime(1627987508.6496193))

print(s)

## time.asctime() method:-

[time.asctime()](https://www.geeksforgeeks.org/python-time-asctime-method/) method is used to convert a tuple or a time.struct\_time object representing a time as returned by time.gmtime() or time.localtime() method to a string of the following form:

Day Mon Date Hour:Min:Sec Year

**Example:**Converting tuple to time.struct\_time object to string

import time

obj = time.gmtime(1627987508.6496193)

time\_str = time.asctime(obj)

print(time\_str)

obj = time.localtime(1627987508.6496193)

time\_str = time.asctime(obj)

print(time\_str)

## time.strptime() method:-

time.strptime() method converts the string representing time to the struct\_time object.

**Example:**Converting string to struct\_time object.

import time

string = "Tue, 03 Aug 2021 10:45:08"

obj = time.strptime(string, "%a, %d %b %Y %H:%M:%S")

print(obj)

**package:-**

We usually organize our files in different folders and subfolders based on some criteria, so that they can be managed easily and efficiently. For example, we keep all our games in a Games folder and we can even subcategorize according to the genre of the game or something like this. The same analogy is followed by the Python package.

A [Python module](https://www.geeksforgeeks.org/python-modules/) may contain several classes, functions, variables, etc. whereas a Python package can contains several module. In simpler terms a package is folder that contains various modules as

**Creating Package**

create a package named mypckg that will contain two modules mod1 and mod2. To create this module follow the below steps –

1)Create a folder named mypckg.

2)Inside this folder create an empty Python file i.e. \_\_init\_\_.py

3)Then create two modules mod1 and mod2 in this folder.

**mod1.py**

def vms():

    print("Welcome to VMS")

**mod2.py**

def sum(a, b):

     return a+b

**The hierarchy of the our package looks like this –**

**mypckg**

|

|

---\_\_init\_\_.py

|

|

---mod1.py

|

|

---mod2.py

**Understanding \_\_init\_\_.py**

\_\_init\_\_.py helps the Python interpreter to recognise the folder as package. It also specifies the resources to be imported from the modules. If the \_\_init\_\_.py is empty this means that all the functions of the modules will be imported. We can also specify the functions from each module to be made available.

**Example:-**

### \_\_init\_\_.py

from .mod1 import vms

from .mod2 import sum

This \_\_init\_\_.py will only allow the vms and sum functions from the mod1 and mod2 modules to be imported.

## Import Modules from a Package

We can import these modules using the [from…import statement](https://www.geeksforgeeks.org/import-module-python/) and the dot(.) operator.

**Syntax:**

import package\_name.module\_name

### Example: Import Module from package

### from mypckg import mod1

from mypckg import mod2

mod1.vms()

res = mod2.sum(1, 2)

print(res)

### Example: Import Specific function from the module

from mypckg.mod1 import vms

from mypckg.mod2 import sum

vms()

res = sum(1, 2)

print(res)