

MODULES

Module is a file containing Python definitions and statements. Modules are imported from other modules using the **import** command.

- i) When module gets imported, Python interpreter searches for module.
- ii) If module is found, it is imported.
- iii) If module is not found, “module not found” will be displayed.

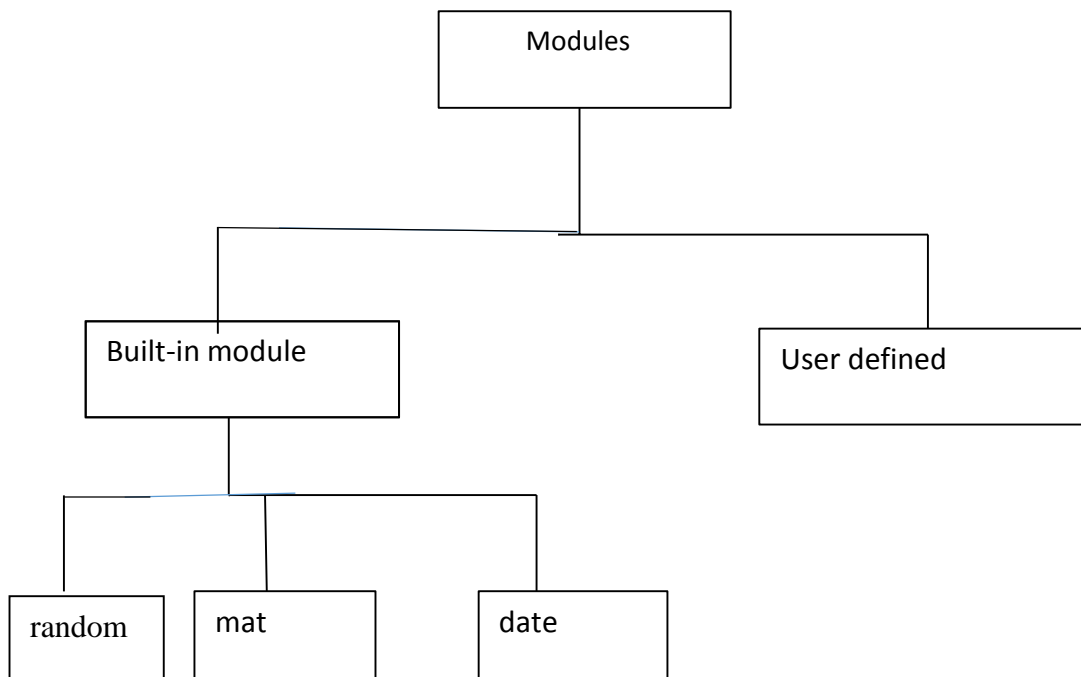


Fig: Types of modules

Built-in modules

Built-in modules are predefined modules, which is already in python standard library.

Python have the following built-in modules.

i) **random**

This module generates random numbers. If we want random integer, we use randint function.

Example:

```
>>>import random
>>>print random . randint(0,5)
1 (or)2 (or) 3 (or)4 (or)5
```

ii) math

This module is used for mathematical calculations. If we want to calculate square root, we use the function sqrt.

Example:

```
>>>import math
>>>math . sqrt(16)
4
```

```
>>>math . factorial(4)
24
```

iii) datetime

This module is used to calculate date and time.

Example:

```
>>>import datetime
>>>datetime . time()
9:15:20
>>>datetime . date()
21.06.2018
```

User Defined modules

To create module, write one or more functions in file, then save it with .py extensions.

i) Create a file

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

ii) Save it as sum.py

iii) Import module

```
>>>import sum
>>>print sum.add(10,20)
```

Module loading and execution:

i) Loading modules

Python modules can be loaded in a number of different ways.

Let's start simple with the math module. Here, we'll load the math module using the import statement and try out some of the functions in the module, such as the square root function sqrt.

```
import math
math.sqrt(81)
9.0
```

Here we have loaded the math module by typing `import math`, which tells Python to read in the functions in the math module and make them available for use. In our example, we see that we can use a function within the math library by typing the name of the module first, a period, and then the name of function we would like to use afterward (e.g., `math.sqrt()`).

ii) Execution module:

Any `.py` file that contains a **module** is essentially also a Python [script](#), and there isn't any reason it can't be executed like one.

Here's `mod.py`, as it was defined earlier:

```
s = "Computers are useless. They can only give you answers."
a = [100, 200, 300]

def printy(arg):
    print(f'arg = {arg}')

class Classy:
    pass
```

This can be run as a script:

```
$ python3 mod.py
```

There are no errors, so it apparently worked. Granted, it's not very interesting. As it is written, it only *defines* objects. It doesn't *do* anything with them, and it doesn't generate any output.

Let's modify the above Python module so it does generate some output when run as a script:

```
s = "Computers are useless. They can only give you answers."
a = [100, 200, 300]

def printy(arg):
    print(f'arg = {arg}')

class Classy:
    pass
```

```
print(s)
print(a)
printy('Good Day Sir!')
x = Classy()
print(x)
```

Now it should be a little more interesting:

```
$ python3 mod.py
```

Computers are useless. They can only give you answers.

```
[100, 200, 300]
```

```
arg = Good Day Sir!
```

```
<__main__.Classy object at 0x10f7a35f8>
```

Unfortunately, now it also generates output when imported as a module:

```
>>>
```

```
>>> import mod
```

Computers are useless. They can only give you answers.

```
[100, 200, 300]
```

```
arg = Good Day Sir!
```

```
<__main__.Classy object at 0x111324748>
```

This is probably not what you want. It isn't usual for a module to generate output when it is imported. Wouldn't it be nice if you could distinguish between when the file is loaded as a module and when it is run as a standalone script? Ask and ye shall receive!

When a .py file is imported as a module, Python sets the special **dunder** variable `__name__` to the name of the module. However, if a file is run as a standalone script, `__name__` [is \(creatively\) set to the string '__main__'](#). Using this fact, you can see which is the case at run-time and change behavior accordingly:

```
s = "Computers are useless. They can only give you answers."
```

```
a = [100, 200, 300]
```

```
def printy(arg):
    print(f'arg = {arg}')
```

```
class Classy:
    pass
```

```
if (__name__ == '__main__'):
    print(s)
    print(a)
    printy('Good Day Sir!')
    x = Classy()
    print(x)
```

Now, if you run as a script, you get output:

```
$ python3 mod.py
Computers are useless. They can only give you answers.
[100, 200, 300]
arg = Good Day Sir!
<__main__.Classy object at 0x100a5d5f8>
```

But if you import as a module, you don't:

```
>>>
>>> import mod
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
 '__spec__', 'help', 'mod']
>>> __name__
'__main__'
>>> mod.__name__
'mod'
```

Modules are often designed with the capability to run as a standalone script for testing the functionality that is contained within the module. This is referred to as [unit testing](#). For example, suppose you have created a module fact.py containing a factorial function, as follows:

```
def fact(n):
    return 1 if n == 1 else n * fact(n-1)

if (__name__ == '__main__'):
    import sys
    if len(sys.argv) > 1:
        print(fact(int(sys.argv[1])))
```

You can treat the file as a module and import fact():

```
>>>
```

```
>>> from fact import fact
```

```
>>> fact(6)
```

```
720
```

But you can also run it as a standalone by passing an integer argument on the command-line for testing:

```
$ python3 fact.py 6
```

```
720
```

Making your own Module

Create Python modules:

To create a Python module, open a Python script, write some statements and functions, and save it with .py extension. Later on, we can call and use these modules anywhere in our program.

Let's create a new module named "MathOperations". This module contains functions to perform addition, subtraction, multiplication, and division.

```
#creating MathOperation module
```

```
#the module provides addition, subtraction, multiplication, and division functions
```

```
#all the functions take two numbers as argument
```

```
#creating addition function
```

```
def addition(num1,num2):
```

```
    return num1+num2
```

```
#creating subtraction function
```

```
def subtraction(num1,num2):
```

```
    return num1-num2
```

```
#creating multiplication function
```

```
def multiplication(num1,num2):
```

```
return num1*num2
```

```
#creating division function
```

```
def division(num1,num2):
```

```
    return num1/num2
```

Now, we can call this module anywhere using the import command, and we can use these functions to perform the related tasks. There is no need to write the code again and again for performing addition, subtraction, multiplication, and division operations.

Call your module:

Let's call this module in our other Python script by using the import command.

```
import MathOperation
```

```
#calling addition function from MathOperation module
```

```
#the function is called by using the module name
```

```
print("The sum is:",MathOperation.addition(10,4))
```

```
#calling subtraction function
```

```
print("The difference is: ",MathOperation.subtraction(100,34))
```

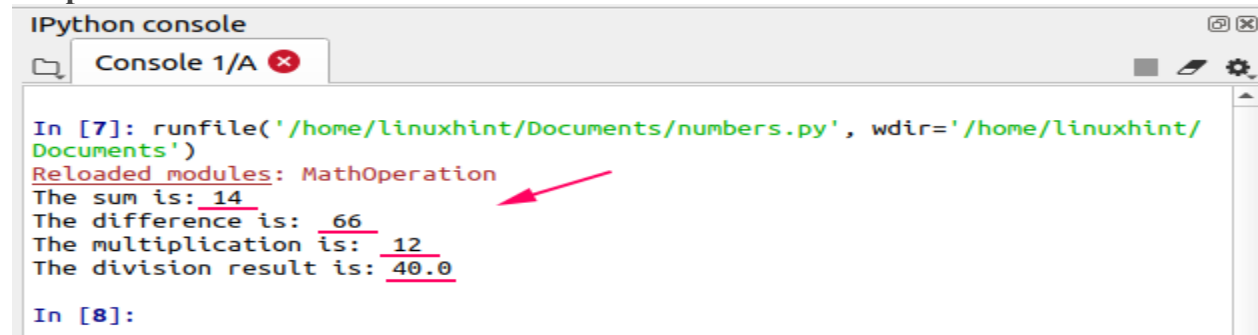
```
#calling multiplication function
```

```
print("The multiplication is: ",MathOperation.multiplication(4,3))
```

```
#calling division function
```

```
print("The division result is:",MathOperation.division(200,5))
```

Output



```
IPython console
Console 1/A x
In [7]: runfile('/home/linuxhint/Documents/numbers.py', wdir='/home/linuxhint/
Documents')
Reloaded modules: MathOperation
The sum is: 14
The difference is: 66
The multiplication is: 12
The division result is: 40.0
In [8]:
```