Python Fundamentals day 8

Today's Agenda

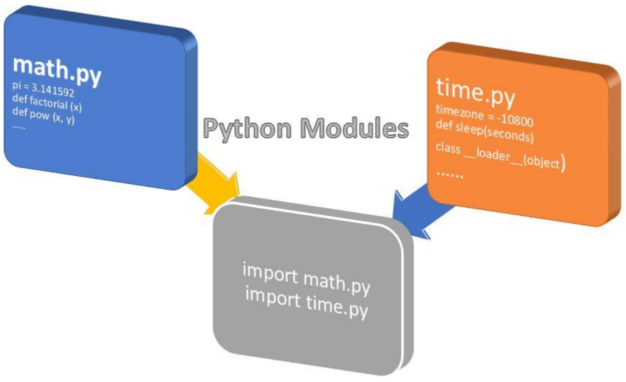
- File, script and module
- Built-in modules
- Documentation string

File, script & module

A file from the hard disk when executed on command line then such a program or a file is called as a **script**.

If we have a single file with numerous set of functions then it is difficult to work with such file. The better way is to group similar set of functions and make a separate file of it and name accordingly. These files are called as **modules**.

Whenever we are in need of certain functions, instead of defining a function again we can just use the function present in particular module by bringing to current file. This process is called as **importing** and is done by using the keyword import.



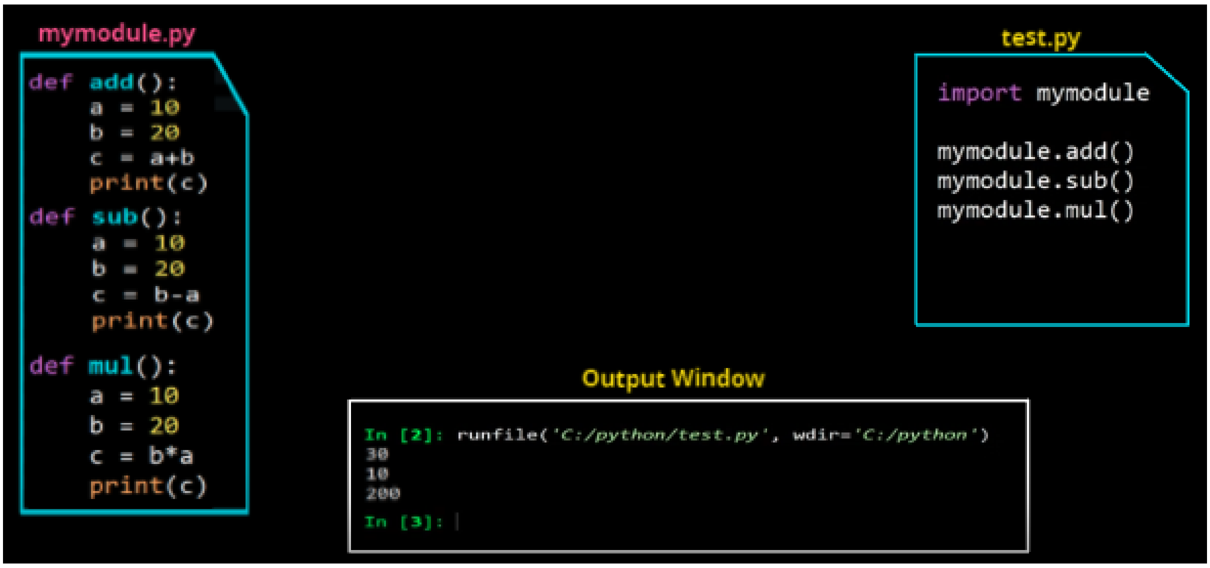
```
math.py
pi = 3.141592
def factorial (x)
def pow (x, y)
.....
```

```
time.py
timezone = -10800
def sleep(seconds)
class __loader__(object)
......
```

```
import math.py
import time.py
```

Importing functions from modules to a file reduces the length of code and certainly reduces the complexity.

Let us now consider an example where we have certain functions in a module called as mymodule and want to use functions present in it, in the python script called as test.py and see how this works.

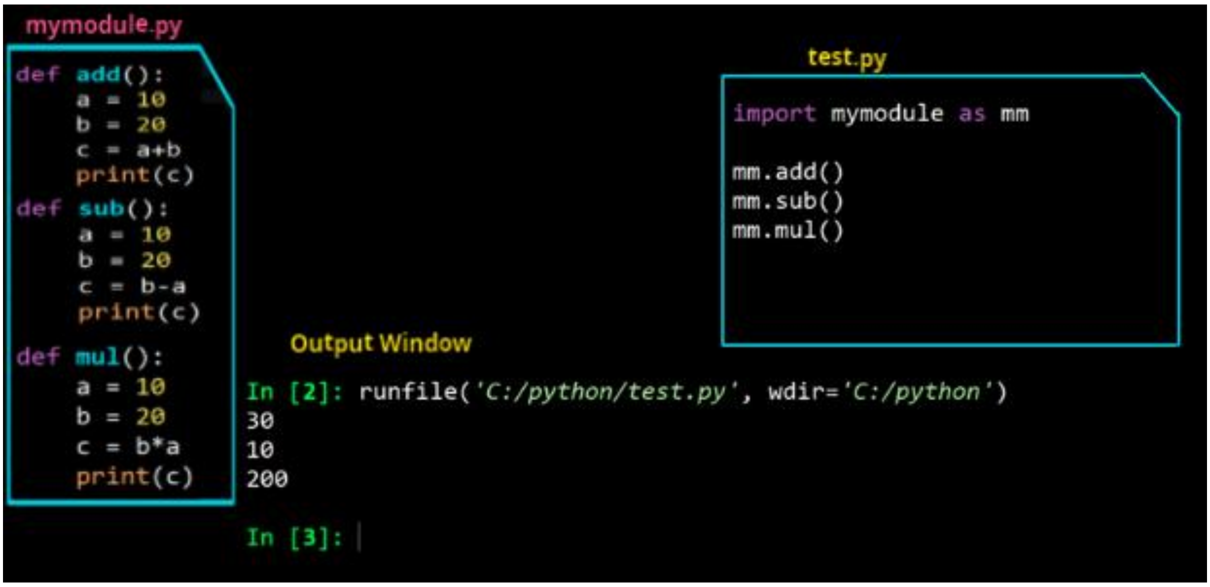


```
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)

def mul():
    a = 10
    b = 20
    c = b*a
    print(c)
```

```
import mymodule

mymodule.add()
mymodule.sub()
mymodule.mul()
```

```
In [2]: runfile('C:/python/test.py', wdir='C:/python')
30
10
200
In [3]: 
```

Instead of calling a module by its name every single time we want to use a function from that module, we can rename it and this is called as aliasing. This can be done by using keyword called as as. Let us see how to do this

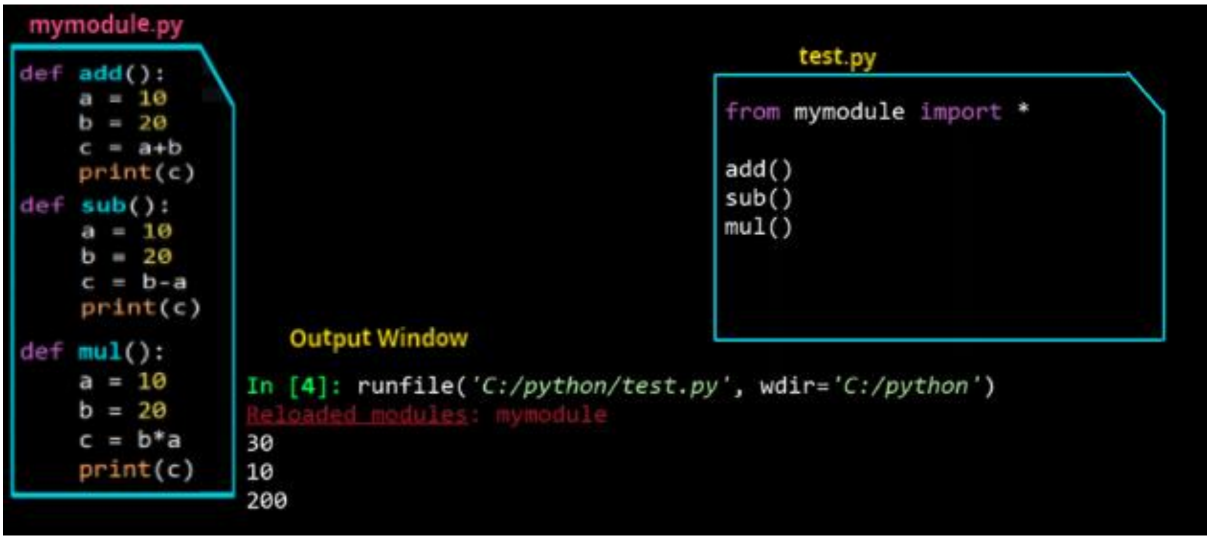


```
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)

def mul():
    a = 10
    b = 20
    c = b*a
    print(c)
```

```
import mymodule as mm

mm.add()
mm.sub()
mm.mul()
```

```
In [2]: runfile('C:/python/test.py', wdir='C:/python')
30
10
200

In [3]: 
```

There is definitely a simpler way for this, which is by using a keyword called as from. Let us see how it makes things easy



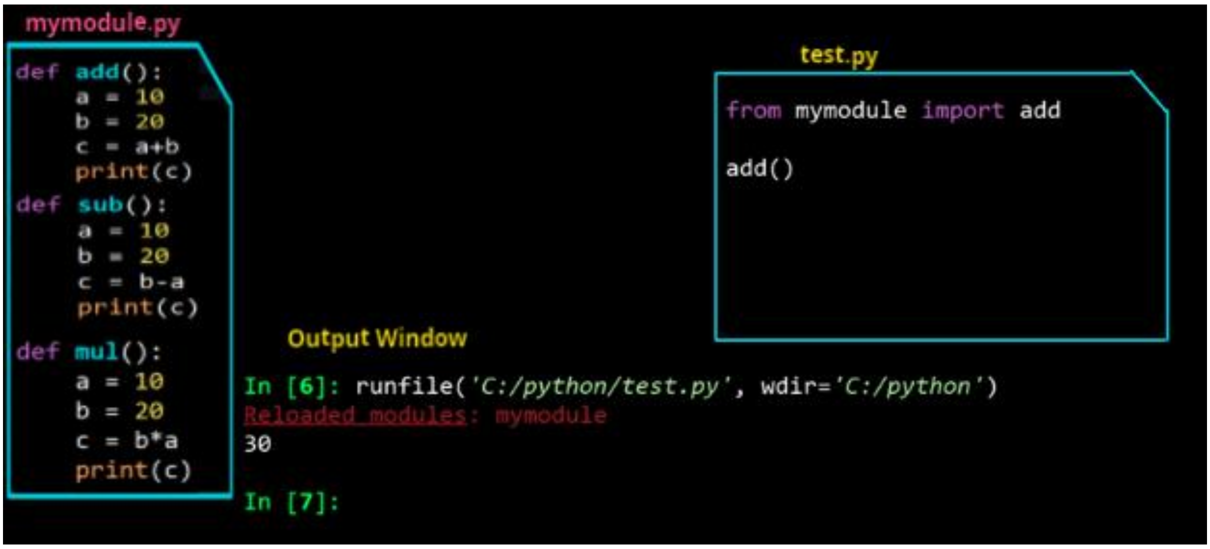
```
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)
```

```
from mymodule import *

add()
sub()
mul()
```

```
In [4]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
30
10
200
```

In above example * means import all the functions present in mymodule to the present file. And we can use the functions just by calling by its name.

If the module contains huge number of functions and we want to use only a couple of functions from it then instead of using * and importing all functions we can just mentions the function which we want to use and import them selectively as shown below

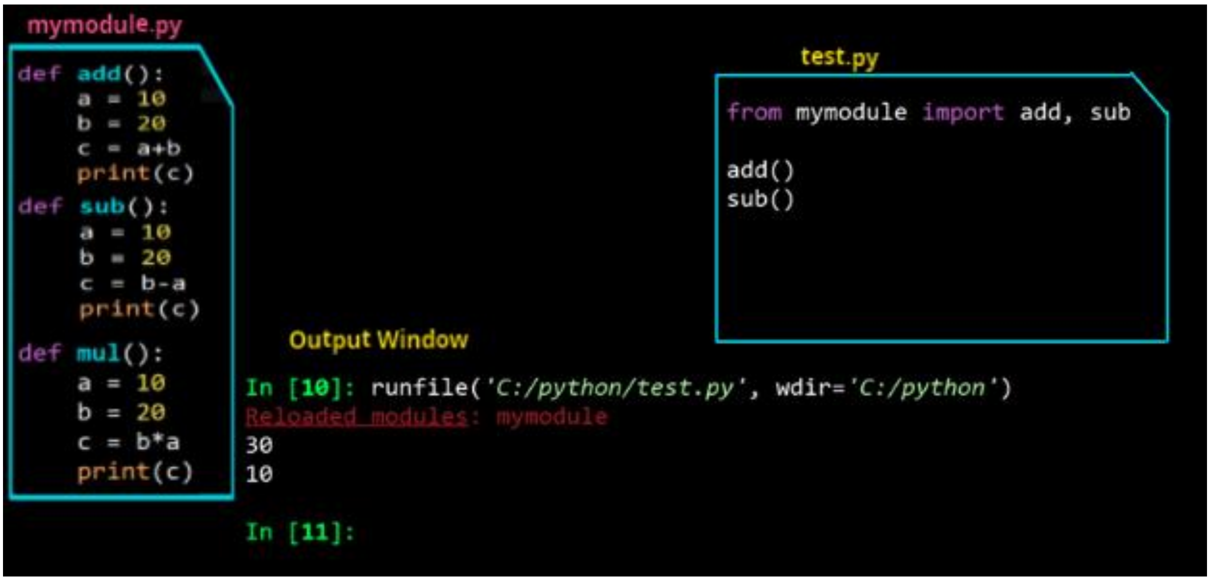


```
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)
```

```
from mymodule import add

add()
```

```
In [6]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
30

In [7]:
```

If you want to import more than one function from the module the all you have to do is



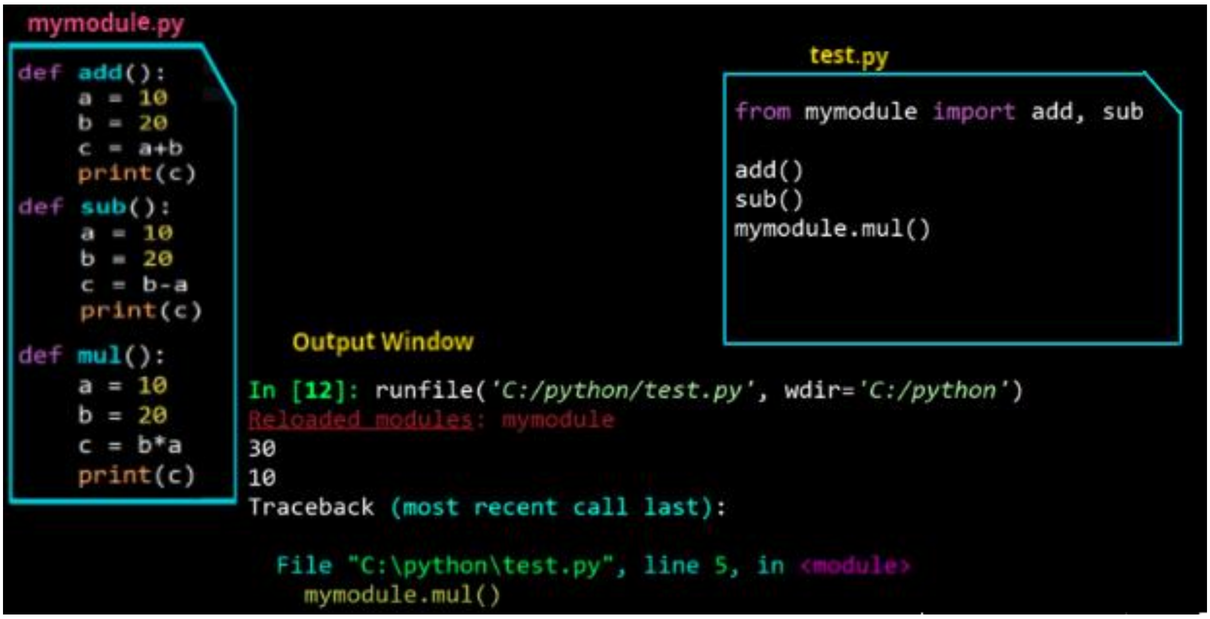
```
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)
```

```
from mymodule import add, sub

add()
sub()
```

```
In [10]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
30
10

In [11]:
```

Note: Knowing that a function is present in the module but calling it without importing, will definitely throw an error saying <function name> is not defined.

And also do not mix these different ways of importing functions, it will give an error. Better follow just one method.



```
def add():
    a = 10
    b = 20
    c = a+b
    print(c)
def sub():
    a = 10
    b = 20
    c = b-a
    print(c)
def mul():
    a = 10
    b = 20
    c = b*a
    print(c)
```

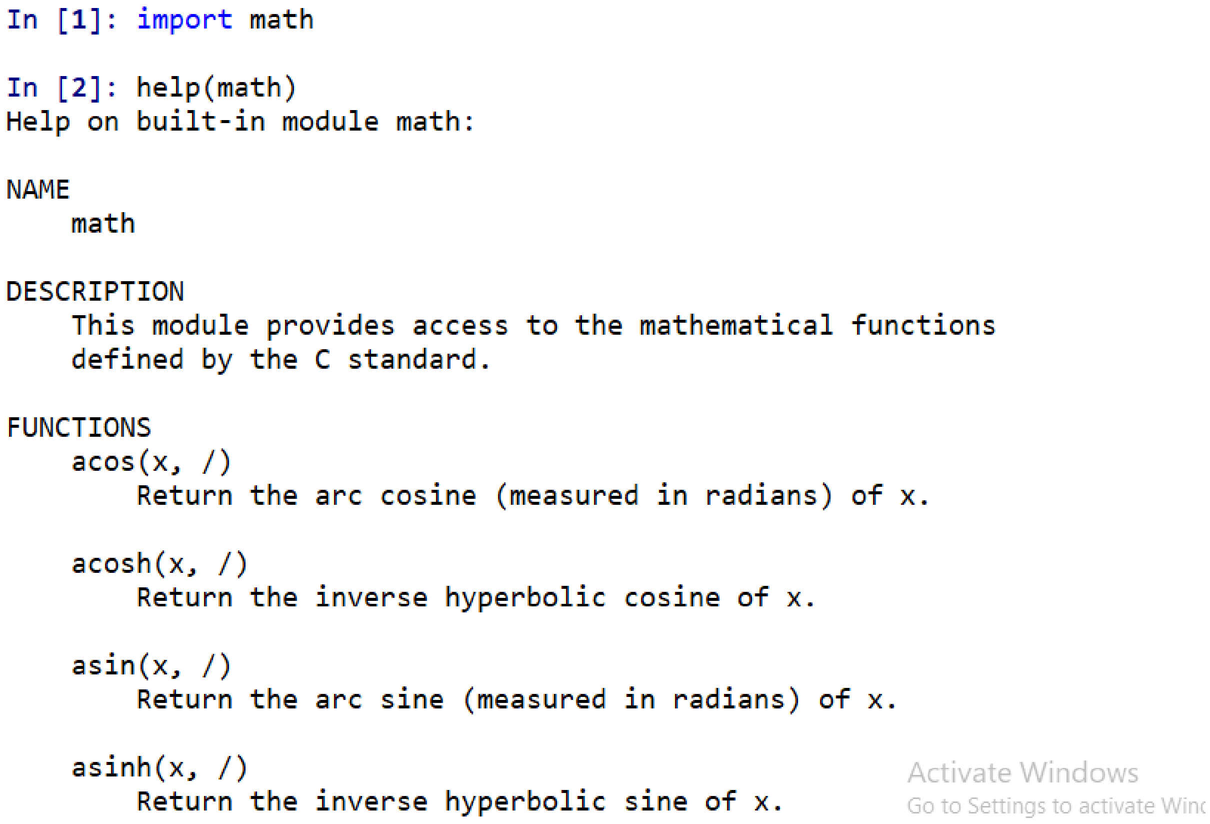
```
from mymodule import add, sub

add()
sub()
mymodule.mul()
```

```
In [12]: runfile('C:/python/test.py', wdir='C:/python')
Reloaded modules: mymodule
30
10
Traceback (most recent call last):

  File "C:\python\test.py", line 5, in <module>
    mymodule.mul()
```

Built-in modules

Whatever we have seen so far are user defined modules, where we have created a module with certain functions in it. But there are some modules which contain numerous functions in it and are already present in python these are called as built-in modules.

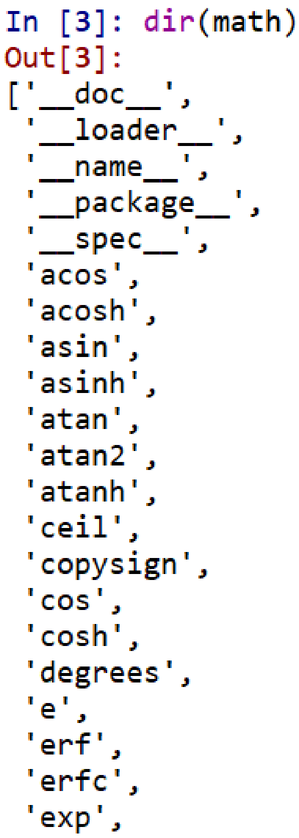
Let us explore one such called as math



```
In [1]: import math

In [2]: help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.
```

help() is a function which will give you complete description about each function present in that module. Above are only few functions present in math module.

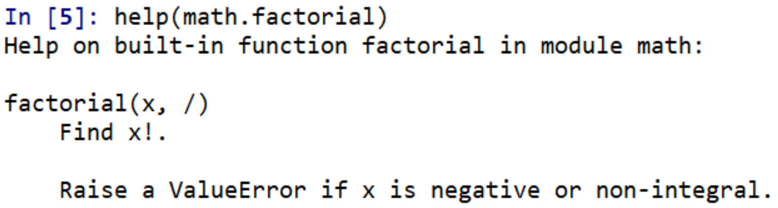
And if you just want to know the different functions present in the module without any description the just use dir() function as shown below



```
In [3]: dir(math)
Out[3]:
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
```

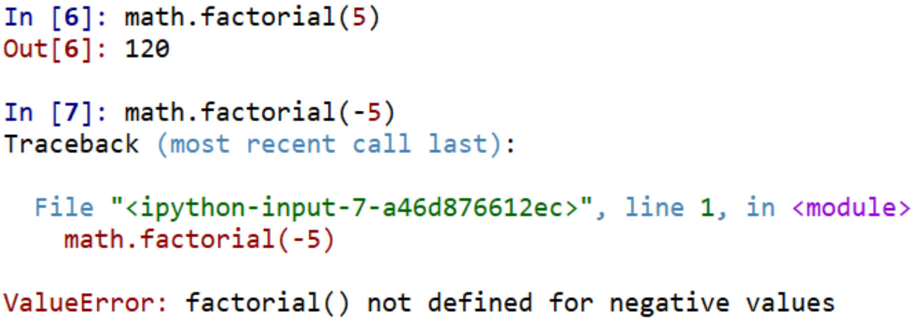
By looking at the square brackets we can say that it returns the list of functions name present in a particular module.

To get the help for a specific function, let us see what to do



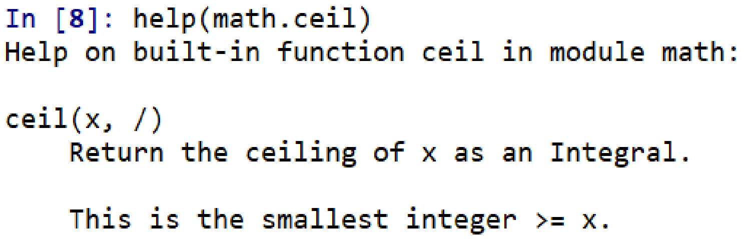
```
In [5]: help(math.factorial)
Help on built-in function factorial in module math:

factorial(x, /)
    Find x!.

    Raise a ValueError if x is negative or non-integral.
```

Let us check if it works as expected



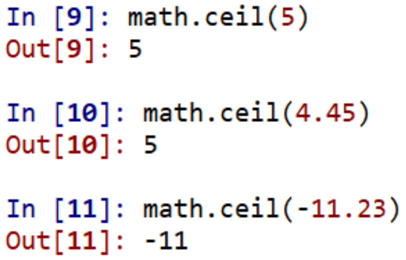
```
In [6]: math.factorial(5)
Out[6]: 120

In [7]: math.factorial(-5)
Traceback (most recent call last):

  File "<ipython-input-7-a46d876612ec>", line 1, in <module>
    math.factorial(-5)

ValueError: factorial() not defined for negative values
```

Let us check some more functions like ceil(), floor()



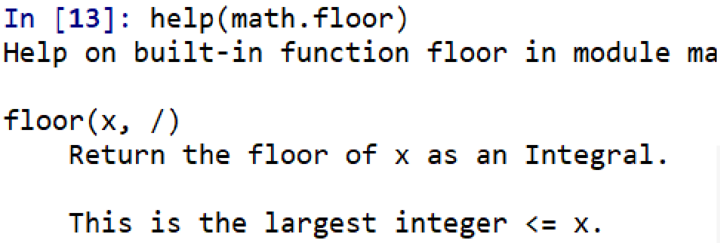
```
In [8]: help(math.ceil)
Help on built-in function ceil in module math:

ceil(x, /)
    Return the ceiling of x as an Integral.

    This is the smallest integer >= x.
```

For example:



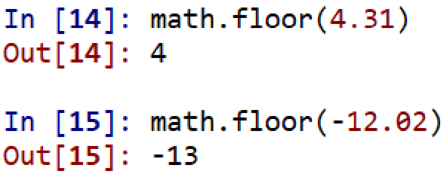
```
In [9]: math.ceil(5)
Out[9]: 5

In [10]: math.ceil(4.45)
Out[10]: 5

In [11]: math.ceil(-11.23)
Out[11]: -11
```

Now let's see what floor() does



```
In [13]: help(math.floor)
Help on built-in function floor in module ma

floor(x, /)
    Return the floor of x as an Integral.

    This is the largest integer <= x.
```

For example:

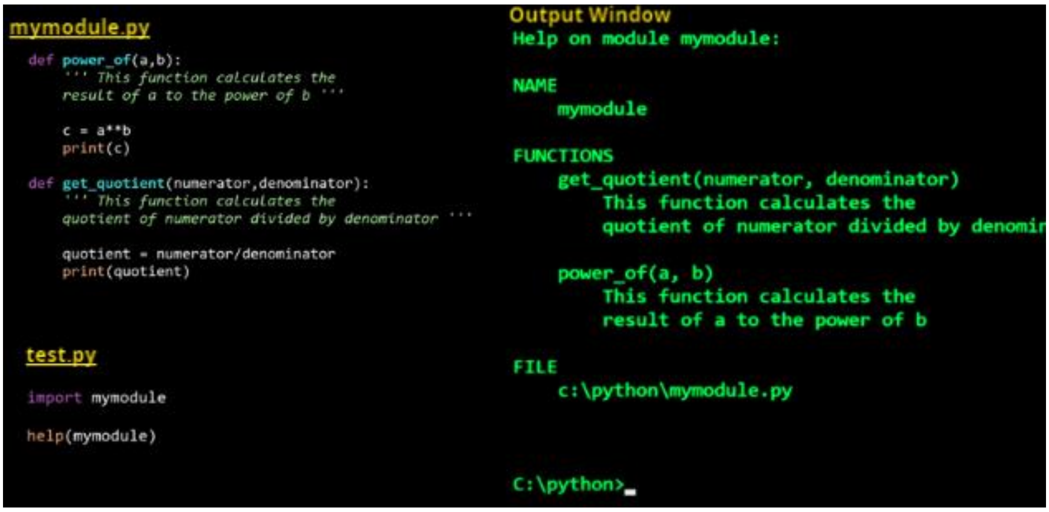


```
In [14]: math.floor(4.31)
Out[14]: 4

In [15]: math.floor(-12.02)
Out[15]: -13
```

Documentation string

Earlier we saw how to create a module, but we have seen that every module has a description so that it's easy for a programmer to know what does a function inside particular module do. The description provided is called as **documentation**. The definition of that function is called **documentation string** which is enclosed within triple quotes.

Let us understand by an example



mymodule.py

```
def power_of(a,b):
    ''' This function calculates the
    result of a to the power of b '''

    c = a**b
    print(c)

def get_quotient(numerator,denominator):
    ''' This function calculates the
    quotient of numerator divided by denominator '''

    quotient = numerator/denominator
    print(quotient)
```

test.py

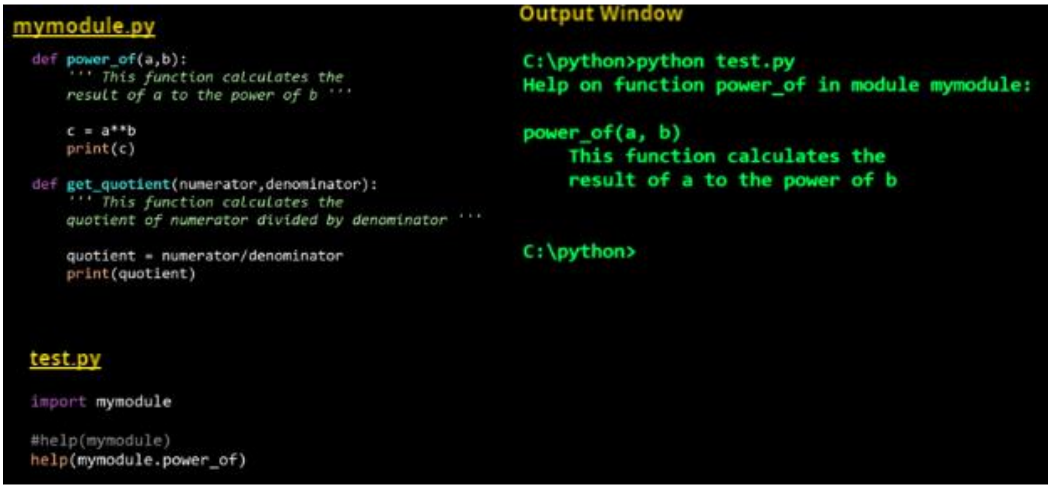
```
import mymodule

help(mymodule)
```

Output Window

```
Help on module mymodule:

NAME
    mymodule

FUNCTIONS
    get_quotient(numerator, denominator)
        This function calculates the
        quotient of numerator divided by denomir

    power_of(a, b)
        This function calculates the
        result of a to the power of b

FILE
    c:\python\mymodule.py


C:\python>_
```

We can certainly see the documentation of all the functions in power_of module.

To see description of selected functions, as said earlier we have to use help(<module_name>.<function_name>) as shown below



mymodule.py

```
def power_of(a,b):
    ''' This function calculates the
    result of a to the power of b '''

    c = a**b
    print(c)

def get_quotient(numerator,denominator):
    ''' This function calculates the
    quotient of numerator divided by denominator '''

    quotient = numerator/denominator
    print(quotient)
```

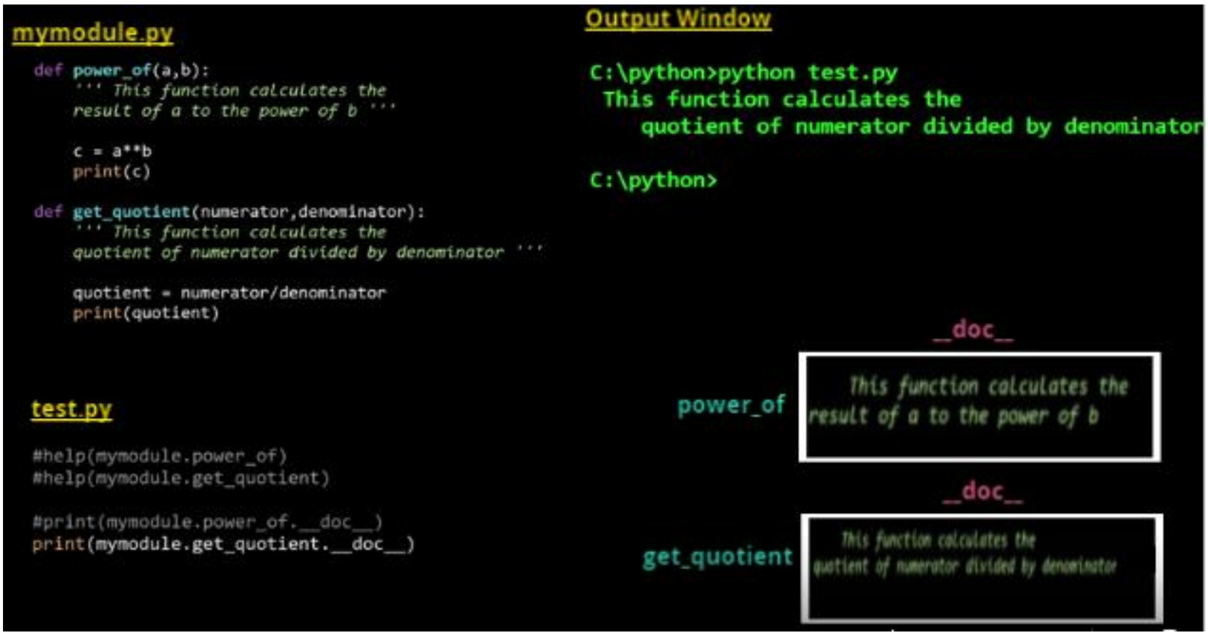
test.py

```
import mymodule

#help(mymodule)
help(mymodule.power_of)
```

Output Window

```
C:\python>python test.py
Help on function power_of in module mymodule:

power_of(a, b)
    This function calculates the
    result of a to the power of b


C:\python>
```

There is also a second way to access the documentation string. Every function has a variable, the name of this variable is __doc__(double underscore doc double underscore)/dunder doc. Let us see how to access it



mymodule.py

```
def power_of(a,b):
    ''' This function calculates the
    result of a to the power of b '''

    c = a**b
    print(c)

def get_quotient(numerator,denominator):
    ''' This function calculates the
    quotient of numerator divided by denominator '''

    quotient = numerator/denominator
    print(quotient)
```

test.py

```
#help(mymodule.power_of)
#help(mymodule.get_quotient)

#print(mymodule.power_of.__doc__)
print(mymodule.get_quotient.__doc__)
```

Output Window

```
C:\python>python test.py
 This function calculates the
    quotient of numerator divided by denominator

C:\python>
```

doc
power_of
This function calculates the result of a to the power of b
doc
get_quotient
This function calculates the quotient of numerator divided by denominator