



PROGRAMMING IN PYTHON

Gavrilut Dragos
Course 7

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py	
<pre>def Sum(x, y) : return x+y</pre>	<pre>import MyModule print (MyModule.Sum(10, 20))</pre>	<div>Output</div> <div>30</div>

Both files `test.py` and `MyModule.py` are located in the same folder.

After the execution of `test.py` a folder with the name `__pycache__` that contains a file called `MyModule.cpython-37.pyc` will appear in the same folder (for Python 3.7) → the version will be different for different versions of Python 3 (pyc = python compiled files)

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py
<pre>def Sum(x, y): return x+y print ("MyModule loaded")</pre>	<pre>import MyModule print (MyModule.Sum(10, 20)) import MyModule</pre>

Loading a module will automatically execute any code (**main code**) that resides in that module.

The main code of a module (code that is written directly and not within a function or a class) will only be executed once (the first time a module is loaded).

Output
MyModule loaded 30

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py
<pre>def Sum(x, y): return x+y print ("MyModule loaded")</pre>	<pre>import MyModule print (MyModule.Sum(10,20)) import MyModule</pre>

What if **MyModule** is not located in the same folder as test.py file ?

Output
<pre>Traceback (most recent call last): File "test.py", line 1, in <module> import sys,MyModule ImportError: No module named 'MyModule'</pre>

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py
<pre>def Sum(x, y): return x+y print ("MyModule loaded")</pre>	<pre>import sys sys.path += ["<folder>"] import MyModule print (MyModule.Sum(10, 20)) import MyModule</pre>

In the above piece of code “<folder>” represents a path to the folder where the file `MyModule.py` resides.

Output

```
MyModule loaded  
30
```

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py
<pre>def Sum(x, y): return x+y print ("MyModule loaded")</pre>	<pre>import MyModule print (dir (MyModule))</pre>
Output	
<pre>['Sum', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']</pre>	

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py
<pre>def Sum(x, y) : return x+y print ("MyModule loaded")</pre>	<pre>import MyModule print (MyModule.__file__) print (MyModule.__name__) print (MyModule.__package__)</pre>

Attributes:

- `__file__` → full path of the file that corresponds to the module (it could be a pyc file as well)
- `__name__` → name of the module (in this example : MyModule)
- `__package__` → name of the package

MODULES

Any Python code (python script) can be used as a module.

Python 3.x

File: *MyModule.py*

```
def Sum(x, y):  
    return x+y  
__doc__ = "Computes the sum of two numbers"
```

Python 3.x

File: *test.py*

```
import MyModule as m  
  
print (m.__doc__)
```

Output

Computes the sum of two numbers

MODULES

Alternatively, the keyword “**help**” can be used as well

Python 3.x

File: MyModule.py

```
def Sum(x, y):  
    return x+y  
__doc__ = "Computes the sum of two numbers"
```

Python 3.x

File: test.py

```
import MyModule as m  
  
help(m)
```

Output

Help on module MyModule:

NAME

MyModule - Computes the sum of two numbers

FUNCTIONS

Sum(x, y)

FILE

...\facultate\python_modules\mymodule.py

MODULES

Alternatively, the keyword “**help**” can be used as well

Python 3.x

File: MyModule.py

```
def Sum(x, y):  
    """returns the sum of x plus y"""  
    return x+y  
__doc__ = "Computes the sum of two numbers"
```

Python 3.x

File: test.py

```
import MyModule as m  
  
help(m)
```

Output

Help on module MyModule:

NAME

MyModule - Computes the sum of two numbers

FUNCTIONS

Sum(x, y)
 returns the sum of x plus y

FILE

e:\documente\facultate\python\2020-2021\mymodule.py

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py
<pre>def Sum(x, y): return x+y print (__name__)</pre>	<pre>import MyModule</pre>
<div>Output</div> <div>__main__</div>	<div>Output</div> <div>MyModule</div>

If a python script is executed directly, the value of `__name__` parameter will be `__main__`.

If it is executed using import, the value of `__name__` parameter will be the name of the module.

MODULES

Any Python code (python script) can be used as a module.

Python 3.x File: MyModule.py	Python 3.x File: test.py
<pre>def Sum(x, y): return x+y if __name__ == "__main__": print("Main code") print("Testing sum(10,20) = ", Sum(10,20)) else: print("Module loaded")</pre>	<pre>import MyModule</pre>
<div>Output</div> <div>Main code Testing sum(10,20) = 30</div>	<div>Output</div> <div>Module loaded</div>

PACKAGES

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a `__init__.py` must exist. That file is an entry point for that package/sub-package.

```
MathOps
├── __init__.py
├── Simple
│   ├── __init__.py
│   ├── Arithmetic.py
│   └── Bits.py
└── Complex
    ├── __init__.py
    └── Series.py
```

PACKAGES

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a `__init__.py` must exist. That file is an entry point for that package/sub-package.

MathOps

`__init__.py`

Simple

`__init__.py`

Arithmetic.py

Bits.py

Complex

`__init__.py`

Series.py

Python 3.x

File: `__init__.py`

```
print ("Package MathOps init")
```

PACKAGES

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a `__init__.py` must exist. That file is an entry point for that package/sub-package.

MathOps

└─ `__init__.py`

└─ Simple

└─ `__init__.py`

└─ Arithmetic.py

└─ Bits.py

└─ Complex

└─ `__init__.py`

└─ Series.py

Python 3.x

File: `__init__.py`

```
print ("Package MathOps.Simple init")
```

PACKAGES

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a `__init__.py` must exist. That file is an entry point for that package/sub-package.

MathOps

└─ `__init__.py`

└─ Simple

└─ `__init__.py`

└─ `Arithmetic.py`

└─ `Bits.py`

Complex

└─ `__init__.py`

└─ `Series.py`

Python 3.x

File: `Arithmetic.py`

```
def Add(x, y) :  
    return x+y
```

```
def Sub(x, y) :  
    return x-y
```


PACKAGES

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a `__init__.py` must exist. That file is an entry point for that package/sub-package.

MathOps

└─ `__init__.py`

└─ Simple

└─ `__init__.py`

└─ `Arithmetic.py`

└─ `Bits.py`

└─ Complex

└─ `__init__.py`

└─ `Series.py`

Python 3.x

File: *Bits.py*

```
def SHL(x, y):  
    return x << y
```

```
def SHR(x, y):  
    return x >> y
```

PACKAGES

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a `__init__.py` must exist. That file is an entry point for that package/sub-package.

MathOps

└─ `__init__.py`

└─ Simple

└─ `__init__.py`

└─ Arithmetic.py

└─ Bits.py

Complex

└─ `__init__.py`

└─ Series.py

Python 3.x

File: `__init__.py`

```
print ("Package MathOps.Complex init")
```

PACKAGES

Python scripts can also be grouped in packages. Packages must be grouped in folder, and in each folder a `__init__.py` must exist. That file is an entry point for that package/sub-package.

MathOps

└─ `__init__.py`

└─ Simple

└─ `__init__.py`

└─ `Arithmetic.py`

└─ `Bits.py`

└─ Complex

└─ `__init__.py`

└─ `Series.py`

Python 3.x

File: *Series.py*

```
def Sum(*p):  
    c = 0  
    for i in p:  
        c += i  
    return c  
  
def Product(*p):  
    c = 1  
    for i in p:  
        c *= i  
    return c
```

PACKAGES

Usage:

Python 3.x

```
import MathOps.Simple.Arithmetic
```

```
print (MathOps.Simple.Arithmetic.Add(2,3))
```

```
from MathOps.Simple import Arithmetic as a
```

```
print (a.Add(2,3))
```

Output

```
Package MathOps init
```

```
Package MathOps.Simple init
```

```
5
```

PACKAGES

Usage:

Python 3.x

```
from MathOps.Simple import *  
  
print (Arithmetic.Add(2,3))  
print (Bits.SHL(2,3))
```

Output

```
Package MathOps init  
Package MathOps.Simple init  
Traceback (most recent call last):  
  File "test.py", line 3, in <module>  
    print (Arithmetic.Add(2,3))  
NameError: name 'Arithmetic' is not defined
```

PACKAGES

To be able to use a syntax similar to “[from <module> import *](#)” a module variable “[__all__](#)” must be defined. That variable will hold a list of all modules that belongs to that package.

MathOps

└─ [__init__.py](#)

└─ Simple

└─ [__init__.py](#)

└─ Arithmetic.py

└─ Bits.py

Complex

└─ [__init__.py](#)

└─ Series.py

Python 3.x

File: [__init__.py](#)

```
print ("Package MathOps.Simple init  
      with __all__ set")  
\_\_all\_\_ = ["Arithmetic", "Bits"]
```

PACKAGES

Usage:

Python 3.x

```
from MathOps.Simple import *  
  
print (Arithmetic.Add(2,3))  
print (Bits.SHL(2,3))
```

Output

```
Package MathOps init  
Package MathOps.Simple init with __all__ set  
5  
16
```

MODULES/PACKAGES

If you want a module and/or package to be available to all the scripts that are executed on that system just copy the module or the entire package folder on the Python search path and you will be able to access it directly. These paths are:

- **Windows:** `<PythonFolder>\Lib`
Exemple: `C:\Python27\Lib` or `C:\Python37\Lib`
- **Linux:** `/usr/lib/<PythonVersion>`
Example: `/usr/lib/python2.7` or `/usr/lib/python3.7)`

MODULES/PACKAGES

Python also has a special library (importlib) that can be used to dynamically import a module.

- `importlib.import_module` (moduleName,package=None) → to import a module
- `importlib.reload` (module) → to reload a module that was already loaded

Python 3.x

File: C:\Python3\Lib\MyModule.py

```
def Sum(x, y) :  
    return x+y  
print ("MyModule loaded")
```

Python 3.x

File: test.py

```
import importlib  
  
m = importlib.import_module("MyModule")  
print (m.Sum(10,20))
```

Output

```
MyModule loaded  
30
```

DYNAMIC CODE

Python has a keyword (`exec`) that can be used to dynamically compile and execute python code.

The format is `exec (code, [global],[local])` where `[global]` and `[local]` represents a list of global and local definition that should be used when executing the code.

Python 3.x

```
exec ("x=100")  
print(x)
```

```
exec ("def num_sum(x,y): return x+y")  
print(num_sum(10,20))
```

```
s = "abcdefg"  
exec ("s2=s.upper()")  
print(s2)
```

Output

100

Output

30

Output

ABCDEFG

DYNAMIC CODE

Because of this keyword, python code can obfuscate or modify itself during runtime.

Python 3.x

```
data = [0x65, 0x66, 0x67, 0x21, 0x54, 0x76, 0x6E, 0x62, 0x29, 0x79,  
        0x2D, 0x7A, 0x2D, 0x7B, 0x2A, 0x3B, 0x0E, 0x0B, 0x0A, 0x73,  
        0x66, 0x75, 0x76, 0x73, 0x6F, 0x21, 0x79, 0x2C, 0x7A, 0x2C,  
        0x7B]  
  
s = ""  
for i in data:  
    s += chr(i-1)  
exec(s)  
print(Suma(1,2,3))
```

Output

6

DYNAMIC CODE

Because of this keyword, python code can obfuscate or modify itself during runtime.

Python 3.x

```
data = [0x65, 0x66, 0x67, 0x21, 0x54, 0x76, 0x6E, 0x62, 0x29, 0x79,  
        0x2D, 0x7A, 0x2D, 0x7B, 0x2A, 0x3B, 0x0E, 0x0B, 0x0A, 0x73,  
        0x66, 0x75, 0x76, 0x73, 0x6F, 0x21, 0x79, 0x2C, 0x7A, 0x2C,  
        0x7B]
```

```
s = ""
```

```
for i in data:
```

```
    s += chr(i)
```

```
exec(s)
```

```
print(Suma(1,2,3))
```

```
def Suma(x,y,z):  
    return x+y+z
```

Output

6

DYNAMIC CODE

Multiple layers of encryption are also possible:

Python 3.x

```
buf =  
[0x74, 0x21, 0x3E, 0x21, 0x23, 0x67, 0x68, 0x69, 0x5D, 0x23, 0x76, 0x2B, 0x64, 0x2F, 0x65,  
0x2C, 0x3D, 0x5D, 0x23, 0x75, 0x68, 0x77, 0x78, 0x75, 0x71, 0x5D, 0x23, 0x64, 0x2E, 0x65, 0  
x23, 0x0E, 0x0B, 0x74, 0x33, 0x21, 0x3E, 0x21, 0x23, 0x23, 0x0E, 0x0B, 0x67, 0x70, 0x73, 0x  
21, 0x64, 0x21, 0x6A, 0x6F, 0x21, 0x74, 0x3B, 0x0E, 0x0B, 0x0A, 0x74, 0x33, 0x21, 0x2C, 0x3  
E, 0x21, 0x64, 0x69, 0x73, 0x29, 0x70, 0x73, 0x65, 0x29, 0x64, 0x2A, 0x2E, 0x33, 0x2A, 0x0E  
, 0x0B, 0x66, 0x79, 0x66, 0x64, 0x29, 0x74, 0x33, 0x2A]  
s = ""  
for i in buf:  
    s += chr(i-1)  
exec(s)  
print(s(10, 20))
```

Output

30

Multiple layers of encryption are also possible:

Python 3.x

```
buf =
[0x74,0x21,0x3E,0x21,0x23,0x67,0x68,0x69,0x5D,0x23,0x76,0x2B,0x64,0x2F,0x65,
0x2C,0x3D,0x5D,0x23,0x75,0x68,0x77,0x78,0x75,0x71,0x5D,0x23,0x64,0x2E,0x65,0
x23,0x0E,0x0B,0x74,0x33,0x21,0x3E,0x21,0x23,0x23,0x0E,0x0B,0x67,0x70,0x73,0x
21,0x64,0x21,0x6A,0x6E,0x21,0x74,0x2B,0x0E,0x0B,0x0A,0x74,0x33,0x21,0x2C,0x3
E,0x21,0x0A,0x0A,0x29,0x64,0x2A,0x2E,0x33,0x2A,0x0E
,0x0B,0x6A,0x6E,0x21,0x74,0x2B,0x0E,0x0B,0x0A,0x74,0x33,0x21,0x2C,0x3E,0x21,0x0A,0x0A,0x29,0x64,0x2A,0x2E,0x33,0x2A,0x0E]

s = "fgh\"u*c.d+<\"tgvwtp\"c-d"
s2 = ""
for c in s:
    s2 += chr(ord(c)-2)
exec(s2)

exec(s)
print(s(10,20))
```

Output

30

Multiple layers of encryption are also possible:

Python 3.x

```

buf = [0x74,0x21,0x3E,0x21,0x23,0x67,0x68,0x69,0x5D,0x23,0x76,0x2B,0x64,0x2F,0x65,
0x2C,0x3D,0x5D,0x23,0x75,0x68,0x77,0x78,0x75,0x71,0x5D,0x23,0x64,0x2E,0x65,0
x23,0x0E,0x0B,0x74,0x33,0x21,0x3E,0x21,0x23,0x23,0x0E,0x0B,0x67,0x70,0x73,0x
21,0x64,0x21,0x6A,0x6E,0x21,0x74,0x2B,0x0E,0x0B,0x0A,0x74,0x33,0x21,0x2C,0x3
E,0x21,0x0A,0x29,0x64,0x2A,0x2E,0x33,0x2A,0x0E
,0x0B,0x6A,0x6E,0x21,0x74,0x2B,0x0E,0x0B,0x0A,0x74,0x33,0x21,0x2C,0x3E,0x21,0x0A]

s = "fgh\"u*c.d+<\"tgvwtp\"c-d"
s2 = ""
for c in s:
    s2 += chr(ord(c)-2)
exec(s2)

def s(a,b): return a+b

exec(s)
print(s(10,20))

```

Output
30

Output

30