Programming in Python

GAVRILUT DRAGOS

COURSE 5

Modules

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

Python 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*

```def Sum(x,y):```
    ```return x+y```
	```import MyModule```
	
	```print (MyModule.Sum(10,20))```

Output
30

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

After the execution of test.py the following things will happen:

o *MyModule.pyc* file will appear in the same folder (Python 2.x)

o A folder with the name __*pycache*__ that contains a file called *MyModule.cpython-35.pyc* will appear in the same folder (Python 3.5) → the version will be different for different versions of Python 3

Modules

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

Python 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*
| ```python
def Sum(x,y):
     return x+y
print ("MyModule loaded")
``` | ```python
import MyModule

print (MyModule.Sum(10,20))
import MyModule
``` |

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 me 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 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*
| ```python
def Sum(x,y):
      return x+y
print ("MyModule loaded")
``` | ```python
import MyModule

print (MyModule.Sum(10,20))
import MyModule
``` |

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

Output

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

Modules

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

Python 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*

``python
def Sum(x,y):
    return x+y
print ("MyModule loaded")
``
	``python
	import sys
	
	sys.path += ["<folder>"]
	
	import MyModule
	
	print (MyModule.Sum(10,20))
	import MyModule
	``

Output
MyModule loaded 30

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

Modules

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

Python 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*

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

Output
Python 2.x ➡ ['Sum', '__builtins__', '__doc__', '__file__', '__name__', '__package__']
Python 3.x ➡ ['Sum', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__']

Modules

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

Python 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*
| ```python
def Sum(x,y):
     return x+y
print ("MyModule loaded")
``` | ```python
import MyModule

print (MyModule.__file__)
print (MyModule.__name__)
print (MyModule.__package__)
``` |

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 (in this example empty string in Python 3.x and None in Python 2.x)

Modules

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

Python 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*
```def Sum(x,y): return x+y print (__name__)```	```import MyModule```

Output
__main__

Output
MyModule

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 2.x / 3.x *File: MyModule.py*	Python 2.x / 3.x *File: test.py*
| ```python
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")
``` | ```python
import MyModule
``` |
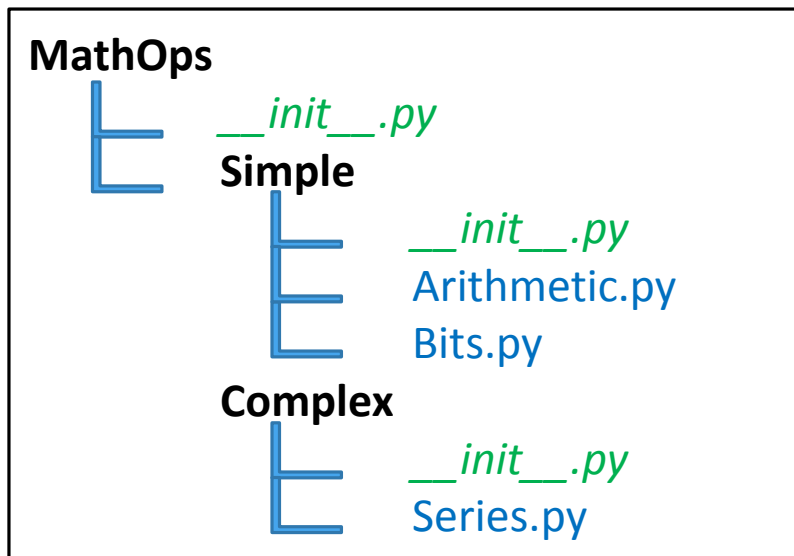
Output

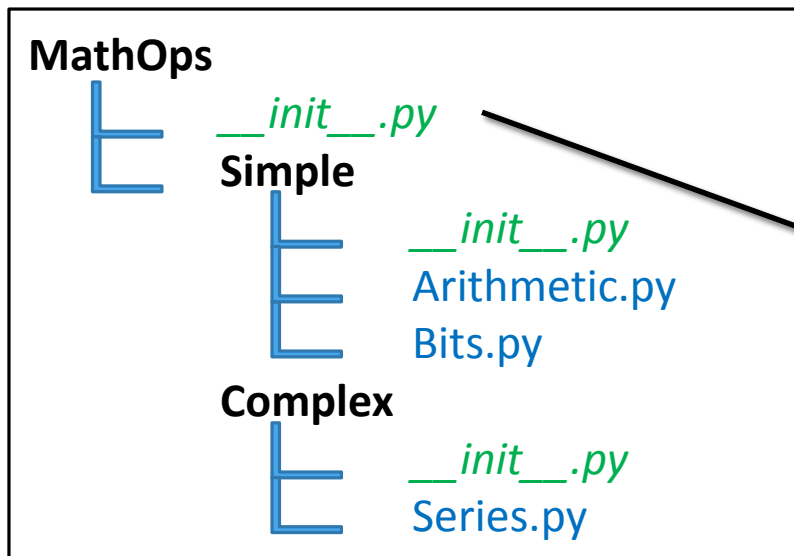Main code
Testing sum(10,20) = 30

Output

Module loaded

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 considered to be an entry point for that package/subpackage.

```
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 considered to be an entry point for that package/subpackage.

MathOps
    __init__.py
        Simple
            __init__.py
            Arithmetic.py
            Bits.py
        Complex
            __init__.py
            Series.py

Python 2.x / 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 considered to be an entry point for that package/subpackage.
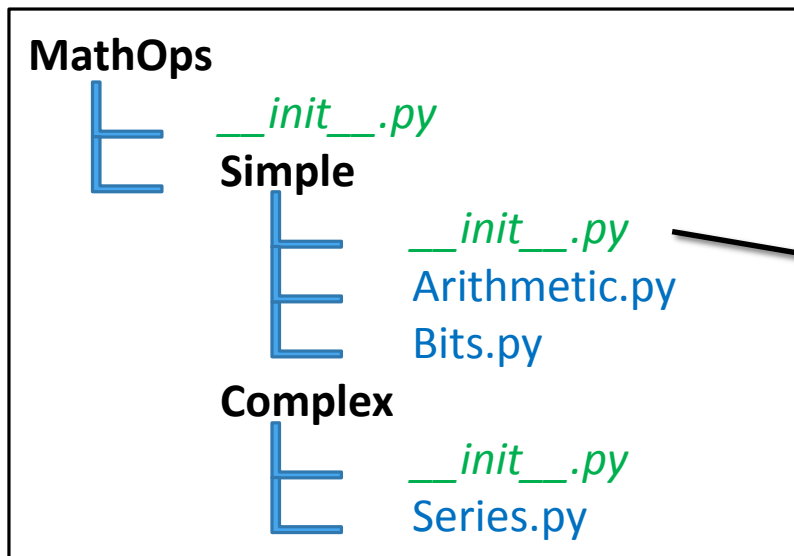
MathOps
   └── *__init__.py*
      └── **Simple**
            └── *__init__.py*
               Arithmetic.py
               Bits.py
      Complex
            └── *__init__.py*
               Series.py

Python 2.x / 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 considered to be an entry point for that package/subpackage.
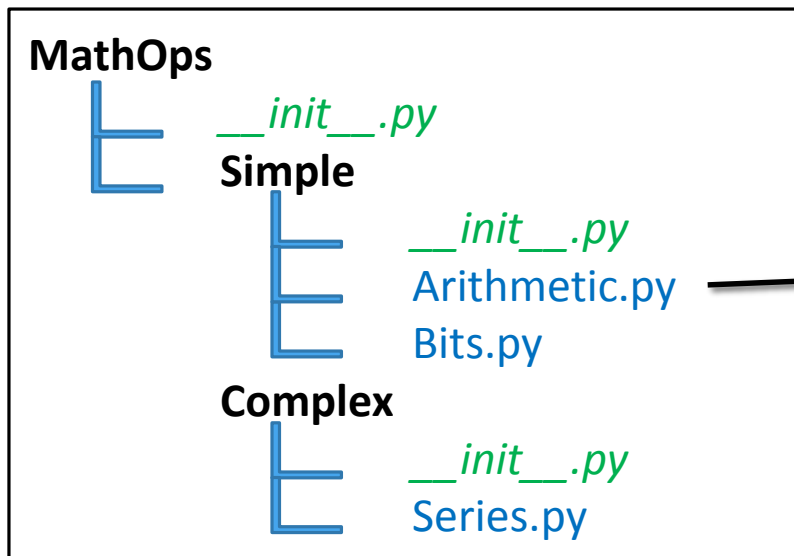
MathOps
- *__init__.py*
- **Simple**
  - *__init__.py*
  - Arithmetic.py
  - Bits.py
- **Complex**
  - *__init__.py*
  - Series.py

Python 2.x / 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 considered to be an entry point for that package/subpackage.
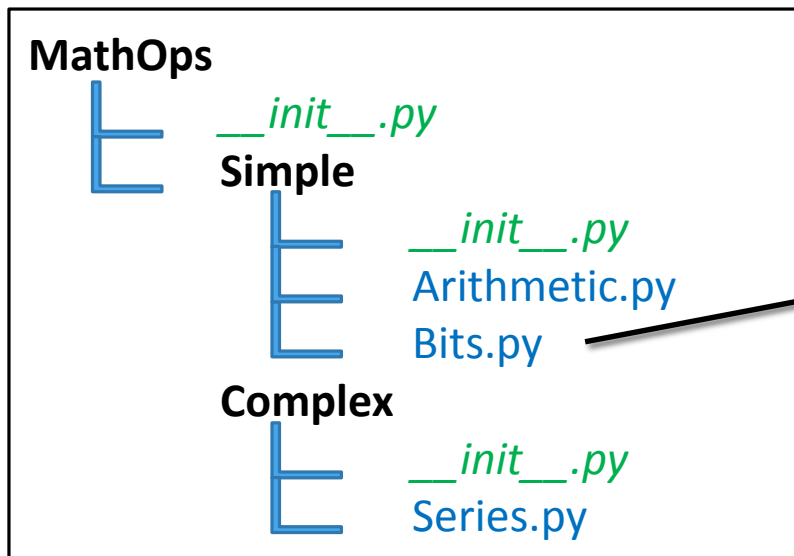
```
MathOps
    └── __init__.py
        Simple
            └── __init__.py
                Arithmetic.py
                Bits.py
        Complex
            └── __init__.py
                Series.py
```

Python 2.x / 3.x
File: Bits.py

```python
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 considered to be an entry point for that package/subpackage.
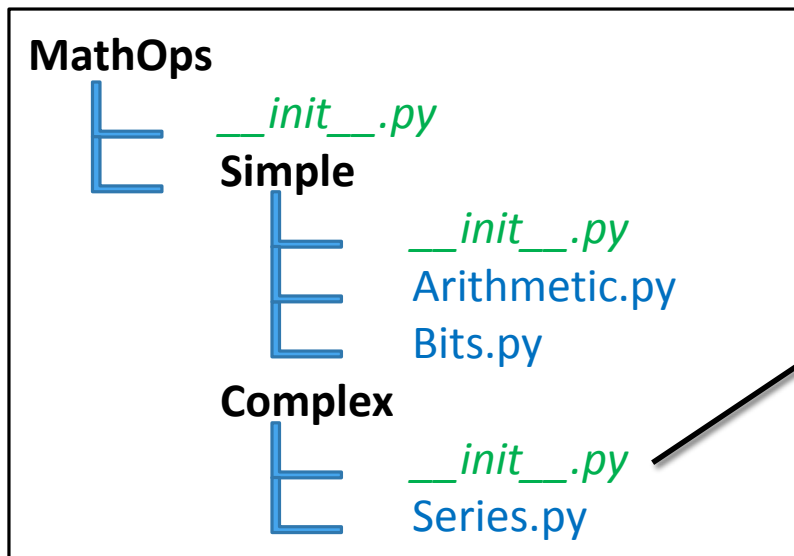
```
MathOps
   └── __init__.py
      └── Simple
             └── __init__.py
                 Arithmetic.py
             └── Bits.py
      Complex
             └── __init__.py
             └── Series.py
```

Python 2.x / 3.x
File: __init__.py

```python
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 considered to be an entry point for that package/subpackage.
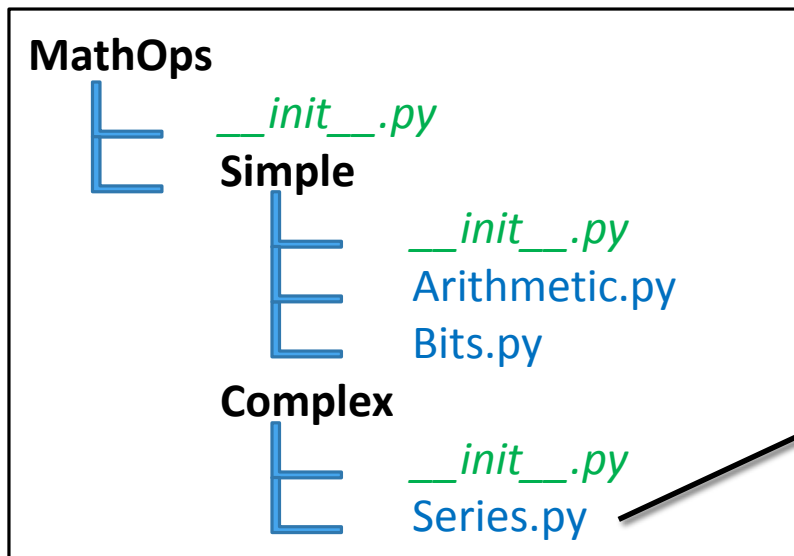
```
MathOps
    └── __init__.py
        └── Simple
                └── __init__.py
                    Arithmetic.py
                    Bits.py
        Complex
                └── __init__.py
                    Series.py
```

Python 2.x / 3.x
File: Series.py

```python
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 2.x / 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))`

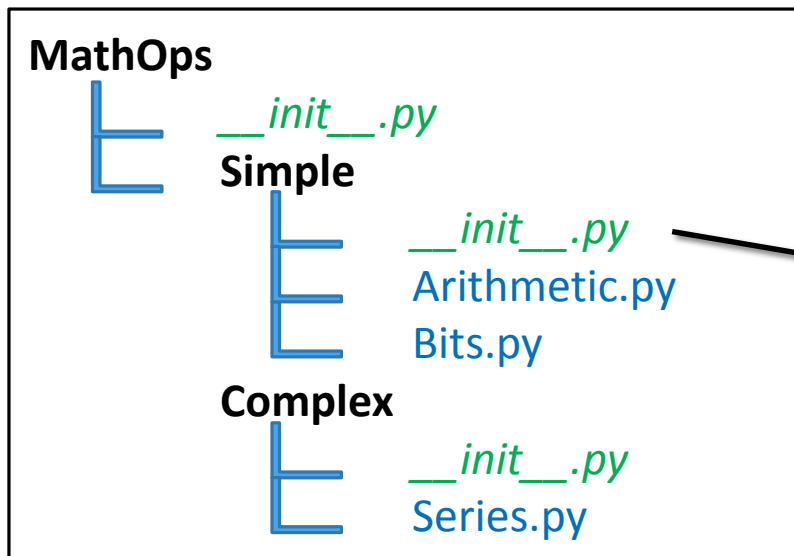| Output |
|---|
| Package MathOps init
Package MathOps.Simple init
5 |

Packages

Usage:

Python 2.x / 3.x

```python
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 "<u>from \<module\> import *</u>" 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 2.x / 3.x
File: __init__.py

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

Packages

Usage:

Python 2.x / 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:\Python35\Lib)
- **Linux:** /usr/lib/\<PythonVersion> (Example: /usr/lib/python2.7 or /usr/lib/python3.5)

Python 2.x File: C:\Python27\Lib\MyModule.py	Python 2.x File: test.py
```python\ndef Sum(x,y):\n return x+y\nprint ("MyModule loaded")\n```	```python\nimport MyModule\nprint (MyModule.Sum(10,20))\nimport MyModule\n```

Output
MyModule loaded 30

Modules/Packages

Python also has a special library (importlib) that can be use 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 2.x *File: C:\Python27\Lib\MyModule.py*	Python 2.x *File: test.py*

```def Sum(x,y):```
```   return x+y```
```print ("MyModule loaded")```
	```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 2.x / 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 can obfuscate or modify itself during runtime.

Python 2.x / 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))
```
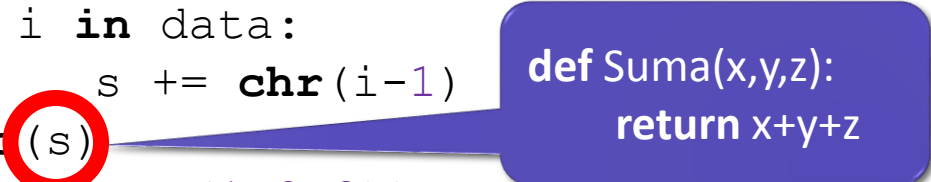
Output
6

Dynamic code

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

Python 2.x / 3.x

```python
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))
```

> **def** Suma(x,y,z):
>   **return** x+y+z

Output

6