Python for Scientists

Part 6 – Modules, Scripts & Distribution

Cyril Desjouy

June, 2016

Updated : January 13, 2019

Modules

From pixabay

Definition:

- `import sys` is a shortcut to `sys = __import__('sys')`
- Importing a module is just creating a module object assigned to a classical variable
- `__import__` function searches for modules in PYTHON PATH

```
» sys = __import__('sys')
» type(sys)
module
» sys.path
['',
 '/(...)/python3.7/site-packages',
 (...)]
```

```
» import sys
» type(sys)
module
» sys.path
['',
 '/(...)/python3.7/site-packages',
 (...)]
```

Definition:

- `import sys` is a shortcut to `sys = __import__('sys')`
- Importing a module is just creating a module object assigned to a classical variable
- `__import__` function searches for modules in PYTHON PATH

```
» sys = __import__('sys')
» type(sys)
module
» sys.path
['',
 '/(...)/python3.7/site-packages',
 (...)]
```

```
» import sys
» type(sys)
module
» sys.path
['',
 '/(...)/python3.7/site-packages',
 (...)]
```

Import custom modules:

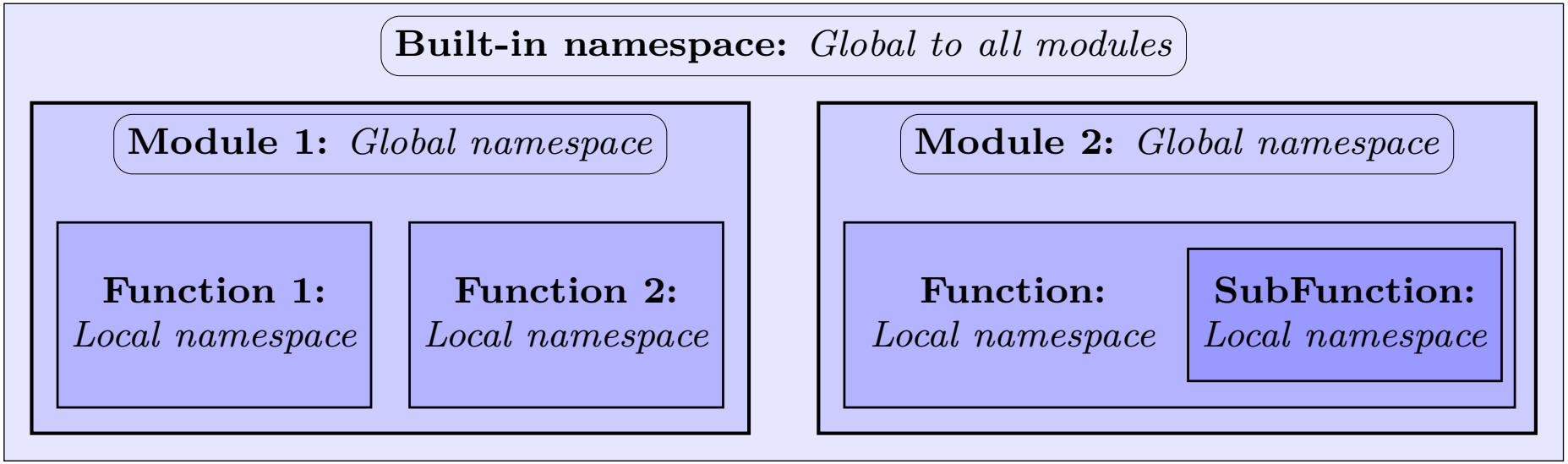
1. Module and script in the same directory
   - ➤ Python includes the current directory to PYTHON PATH
2. Directory of the module in the PYTHON PATH
   - ➤ Module installed with pip/conda or other package manager
   - ➤ Module's path added with the IDE path manager (Spyder, ...)
   - ➤ Module's path added though OS : `export PYTHONPATH=$PYTHONPATH:/MyDir/`
   - ➤ Module's path added though the script : `sys.path.insert(0, "/MyDir/")`
3. None of the two above cited conditions are fullfilled
   - ➤ `ModuleNotFoundError`

Definition :

- The name of an object is the way to access to this object
- A namespace is a set of names providing access to a set of objects
- Different namespaces can co-exist in a single Python interpreter
- Each namespace is isolated from the others
- This isolation ensures that same names in different namespaces don't collide

The namespace hierarchy :

- **Built-in namespace:** Created at startup. Contains all the built-in functions.
- **Module namespace:** Created when a module is imported.
- **Function namespace:** Specific to the current function.



beer.py

```
beers = ['Kro', 'Chouffe', 'Grim']


def serve(name):
    if name in beers:
        print(f"{name} served !")
    else:
        print(f"No {name} here !")


def list_beers(beers=beers):
    print('Beers :')
    print(*beers, sep=', ')
```

script.py

```
import beer

beers = ['Maredsous', 'Guinness']
beer.list_beers(beers)
beer.list_beers(beer.beers)
beer.serve('Chouffe')
```

Outputs:

```
Beers :
Maredsous, Guinness
Beers :
Kro, Chouffe, Grim
Chouffe served !
```

beer.py

```python
beers = ['Kro', 'Chouffe', 'Grim']


def serve(name):
    if name in beers:
        print(f"{name} served !")
    else:
        print(f"No {name} here !")


def list_beers(beers=beers):
    print('Beers :')
    print(*beers, sep=', ')

# Some modules tests !
for beer in beers:
    serve(beer)
```

script.py

```python
import beer

beers = ['Maredsous', 'Guinness']
beer.list_beers(beers)
beer.list_beers(beer.beers)
beer.serve('Chouffe')
```

Outputs:

```
Kro served !
Chouffe served !
Grim served !
Beers :
Maredsous, Guinness
Beers :
Kro, Chouffe, Grim
Chouffe served !
```

The __name__ variable :

- is a variable automatically created when a Python file is interpreted
- contains the name of the current `*.py` file if it has been imported
- contains `"__main__"` if the `*.py` is executed as the main script

mod_name.py

```
print('run mod :', __name__)
```

outputs

```
run mod : __main__
```

script_name.py

```
import mod_name

print('run script: ', __name__)
```

outputs

```
run mod : mod_name
run script:  __main__
```

The __name__ variable :

- is a variable automatically created when a Python file is interpreted
- contains the name of the current *.py file if it has been imported
- contains "__main__" if the *.py is executed as the main script

mod_name.py

```
if __name__ == 'main':
    print('run mod :', __name__)
```

script_name.py

```
import mod_name

print('run script: ', __name__)
```

outputs

```
run mod : __main__
```

outputs

```
run script:  __main__
```

Use of `if __name__ == "__main__":` **?**

- Anything that comes after the `if __name__ == "__main__"` is executed when the script file is explicitly executed
- When the file is imported, the various functions and class definitions will be imported, but the "__main__" script won't be executed !

beer.py

```
beers = ['Kro', 'Chouffe', 'Grim']

def serve(name):
    if name in beers:
        print(f"{name} served !")
    else:
        print(f"No {name} here !")

def list_beers():
    print('Beers :')
    print(*beers, sep=', ')

if __name__ == '__main__':
    for beer in beers:
        serve(beer)
```

script.py

```
import beer

beer.list_beers()
beer.serve('Chouffe')
```

Output from script.py

```
Beers :
Kro, Chouffe, Grim
Chouffe served !
```

Output from beer.py

```
Kro served !
Chouffe served !
Grim served !
```

“Talk is cheap. Show me the code.”

By Linus Torvald

Screenshot by Gamaliel Espinoza Macedo

my_first_python_script.py

```
#!/usr/bin/env python                          -> Needed for script execution
# -*- coding:  utf-8 -*-                       -> If not present, ASCII by default

"""My first awesome Python script!"""          -> The doc of my script/module

import mymodule1 as myshortcut1                -> Modules used in this script
...
import mymoduleN as myshortcutN


def mygreatfct(a):                             -> My function definitions
    """ My great docstring """
    return a


if __name__ == "__main__":                     -> Not necessary, but recommended
    first_line = "Yeah!  It begins"            -> Beginning of my script
    ...
    ...
    ...
    last_line = "Yeah!  It ends"               -> Ending of my script
```

Scripting in Python

Why should I use this thing : `#!/usr/bin/env python` **?**

- If you work on Window, you won't !
- If you work on Mac or Linux, this makes the python script executable
- Note that you should have to adapt the path as a function of your system and of the interpreter you want to use (Python 2.x / Python 3.x)

Why should I use this other thing : `if __name__ == "__main__":` **?**

- `__name__` is a Python variable automatically created by Python
- `__name__` contains the name of the current script if it has been imported
- `__name__` contains `"__main__"` if the script is the main script
- Anything that comes after the `if __name__ == "__main__"` is executed when the script file is explicitly executed
- When the file is imported, the various function and class definition will be imported, but the `"__main__"` script won't be executed !

Distribution

From pixabay

There are several ways to distribute Python code :

- **Packaging :** Creation of a `setup.py` for the installation of your Python code
  - ➤ Extensive guide : Python Packaging
  - ➤ Excellent guide in french : Sam & Max

- **Code freezing:** Create an executable file that contains all your Python code, the libraries used in the code and the Python interpreter
  - ➤ Advantage : The application will run on any system
  - ➤ Disadvantage : The size of the app !

- **Distribution packaging :** To distribute Python code on Linux, creation of a distribution package for Archlinux, Debian, Ubuntu, Fedora... [1]

Whatever you choose, try to:

- publish your source code on dedicated platforms such as github
- package your mature codes and distribute them through Pypi (pip)

[1] The package will not include the Python interpreter. Then, the distribution package will be smaller than freezing the application.

Focus on `cx_freeze`

- There are several freezing tools supporting different features and platforms (`cx_freeze`, `bbfreeze`, `py2exe`, `pyinstaller`, `py2app`, ...)
- `cx_freeze` is multi platform (Win/Linux/OSX)
- `cx_freeze` is compatible with Python 2.x and 3.x
- Using `cx_freeze` is as simple as :

```
cx_freeze MyApp.py
```

- `cx_freeze` will generate an executable adapted to the current OS
- `cx_freeze` can also be used with a `setup.py` file :
  - see here for the official documentation
  - or here for a french tutorial