Python | Main course

Session 10

- Modules
- Packages
- Scripts
- PIP

by Mohammad Amin H.B. Tehrani - Reza Yazdani

www.maktabsharif.ir

Modules

Modules

Intro

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Snapp example:

```python
# users.py: Users module

class User: pass

class Driver(User): pass

class Passenger(User): pass
```

```python
# vehicles.py :Vehicles module

class Vehicle: pass

class Car(Vehicle): pass

class Motor(Vehicle): pass
```

```python
# main.py : main module

import users
import vehicles

def main():
    ...
```

Using a module

'import' Statement:

You can use any Python source file as a module by executing an **import** statement in some other Python source file. The import has the following syntax:

```
import module_name1 [, module_name2, module_name3, ...]
```

```
import dill
import pickle
import re
```

=

```
import dill, pickle, re
```

‘as’ keyword

If the module name is followed by **as,** then the name following as is bound directly to the imported module.

```
import module_name1 as new_module_name
```

```
import dill as DILL
import pickle as P_
import re as regex
```

=

```
import dill as DILL, pickle as P_, re as regex
```

from ... import ...

Python's **from** statement lets you **import** specific attributes from a module into the current namespace. The from...import has the following syntax:

```
from module_name1 import some_variable[, some_function, some_class]
```

```
from math import tan, sin, pow
```

```
from math import tan as tangent, sin as sinus, pow as power
```

It is also possible to import **all names** from a module into the current namespace by using the following import statement

```
from module_name1 import *
```

```
from math import *
```

What's the difference between:
import math
and
from math import *
?

Packages

Packages

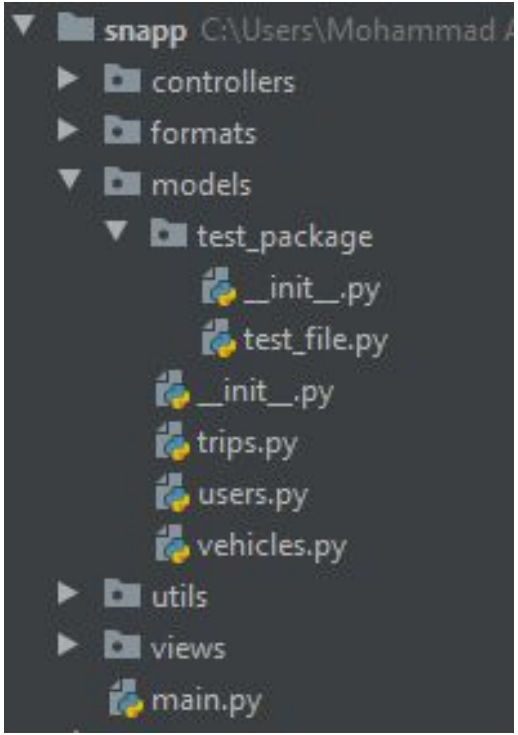
Intro

A **package** is basically a **directory** with Python files and a file with the name **__init__.py .**
This means that every directory inside of the Python path, which contains a file named **__init__.py**, will be treated as a package by Python. It's possible to put several modules into a Package.

Packages are a way of structuring Python's module namespace by using "dotted module names".

Example:

```
from models.test_package.test_file import TestClass
```



Packages

___init___.py

The __init__.py file makes Python treat directories containing it as modules. Furthermore, this is the first file to be loaded in a module, so you can use it to execute code that you want to run each time a module is loaded, or specify the submodules to be exported.

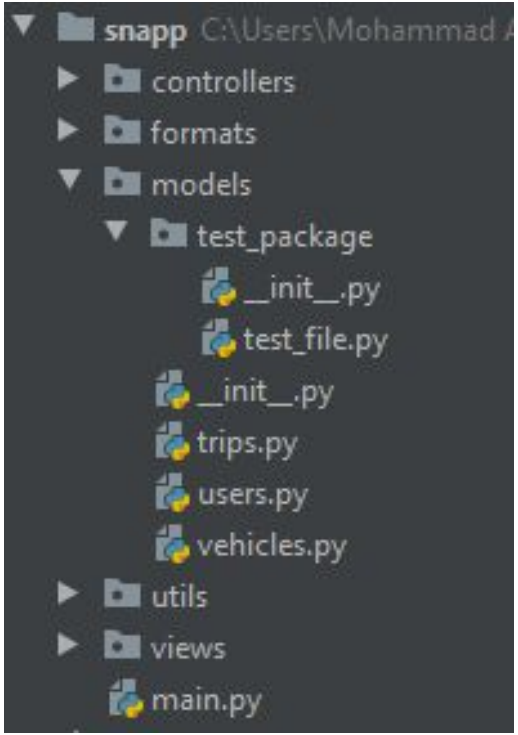
```
# models/__init__py

print(f"\n====== Module {__name__} =======")
print('Path:', __path__)
print('File:',__file__)
print('Name:',__name__)
print('Package:',__package__)
```

```
# main.py

import models
```

Output???



Packages

___init___.py > Example

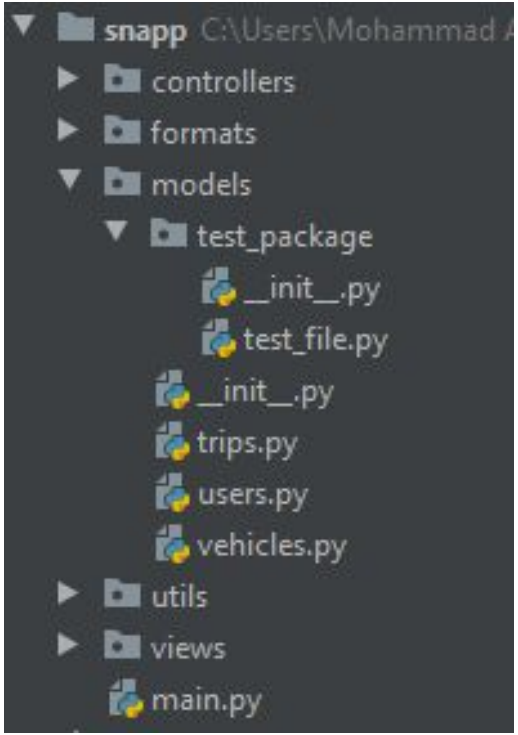
```python
# models/__init__py

print(f"\n====== Module {__name__} =======")
print('Path:', __path__)
print('File:',__file__)
print('Name:',__name__)
print('Package:',__package__)
```

```python
# main.py

import models
```

```
====== Module models =======
Path: ['C:\\Users\\~\\PycharmProjects\\snapp\\models']
File: C:\Users\~\PycharmProjects\snapp\models\__init__.py
Name: models
Package: models
```



Some special variables on Packaging

- **___name___ :** Name of module imported
- **___file___:** Absolute file directory to file imported
- **___package___:** Package name
- **___class___:** Name of class (If class was imported)

UserManager example: packaging

Maktab Sharif

UserManager: Packaging

Use packages and modules for organize your code.

- exceptions.py
- models.py
- menus.py
- main.py
- ...

```
USER MANAGER PROGRAM
     1. Register
     2. Login
Enter option:
```

```
USER MANAGER > REGISTER
>> phone:
>> password:
>> name:
>> email(Optional):
Registered Successfully!
```

```
USER MANAGER > LOGIN
>> phone:
>> password:
ERROR: Invalid password
```

interactive shell

Intro

The interactive shell is between the user and the operating system (e.g. Linux, Unix, Windows or others). Instead of an operating system an interpreter can be used for a programming language like Python as well. The Python interpreter can be used from an interactive shell.

The interactive shell is also interactive in the way that it stands between the commands or actions and their execution. In other words, the shell waits for commands from the user, which it executes and returns the result of the execution. Afterwards, the shell waits for the next input.

Run:

```
python ...
python3 ...
py ...
```

Interactive shell

Example

```
yazdan@MrYazdan:~$ python
Python 3.10.5 (main, Jun  6 2022, 18:49:26) [GCC 12.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 'salam !'
>>> a
'salam !'
>>> import os
>>> os.name
'posix'
>>> exit()
```

Python Scripts

Intro

Scripting languages do not require the compilation step and are rather interpreted.

Python is **scripting**, general-purpose, high-level, and interpreted programming language. It also provides the object-oriented programming approach.

Run python scripts:
`python <python file with .py>`

```
# test.py
from os import listdir

print(listdir())
```

```
yazdan@MrYazdan:~$ python test.py
['docstring_test.py', 'app.py', 'static',
'__pycache__', 'auth', 'menu', 'venv', '.idea',
'templates']
```

___name___

The **__name__** variable (two underscores before and after) is a special Python variable. It gets its value depending on how we execute the containing script. you can import that script as a module in another script.
When you run your script, the **__name__** variable equals **__main__**. When you import the containing script, it will contain the name of the script.

```
# a_module.py

print('Inside a_module.py, name:', __name__)
```

```
# test.py
import a_module

print('Inside test.py, name:', __name__)
```

Run test.py:

```
yazdan@MrYazdan:~$ python3 test.py
Inside a_module.py, name: a_module
Inside test.py, name: __main__
```

Run a_module.py:

```
yazdan@MrYazdan:~$ python3 a_module.py
Inside a_module.py, name: __main__
```

Scripts

___name___ == '___main___'

We can use an **if __name__ == "__main__"** block to allow or prevent parts of code from being run when the modules are imported. When the Python interpreter reads a file, the **__name__** variable is set as **__main__** if the module being run, or as the module's name if it is imported.

```
# a_module.py

print('Inside a_module.py, name:', __name__)

if __name__ == '__main__':
   print('You can see me if you run me!!!')
```

```
# test.py
import a_module

print('Inside test.py, name:', __name__)
```

Run test.py:

```
yazdan@MrYazdan:~$ python3 test.py
Inside a_module.py, name: a_module
Inside test.py, name: __main__
```

Run a_module.py:

```
yazdan@MrYazdan:~$ python3 a_module.py
Inside a_module.py, name: __main__
You can see me if you run me!!!
```

Script

Example: Screen shot

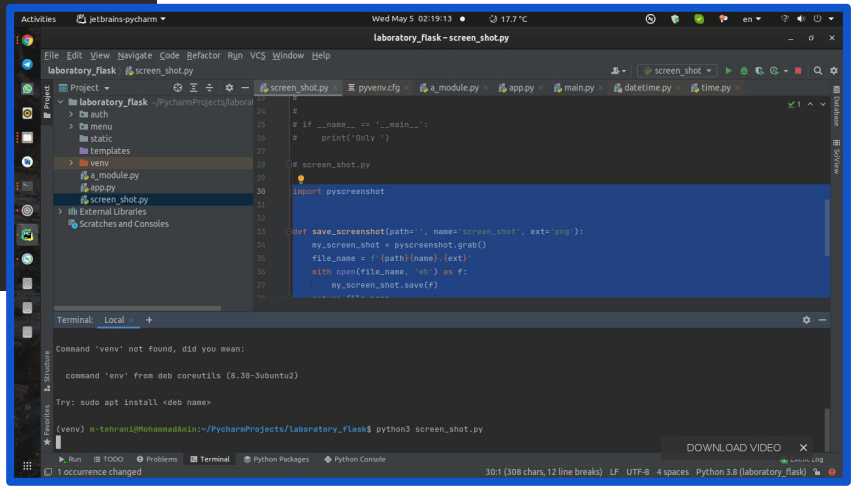
```
import pyscreenshot


def save_screenshot(path='', name='screen_shot',
ext='png'):
    my_screen_shot = pyscreenshot.grab()
    file_name = f'{path}{name}.{ext}'
    with open(file_name, 'wb') as f:
        my_screen_shot.save(f)
    return file_name


if __name__ == '__main__':
    print(save_screenshot())
```

```
yazdan@MrYazdan:~$ python3 screen_shot.py
screen_shot.png
```

Result!



Scripts

python -m ...

You can run library module as a script by using **-m** option.
Now you can run every accessible modules. (from your directory)

Example:

```
yazdan@MrYazdan:~$ python3 -m pip install test
```

```
yazdan@MrYazdan:~$ python3 -m py_compile test.py
```

PIP

PIP

Intro

Python Installs Packages

pip is a package-management system written in Python used to install and manage software packages. It connects to an online repository of public and paid-for private packages, called the Python Package Index.

Syntax

- if set in env variables:

```
pip ...
```

- From python interpreter:

```
python -m pip ...
```

Commands (pip -h)

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.
freeze	Output installed packages in requirements format.
list	List installed packages.
show	Show information about installed packages.
check	Verify installed packages have compatible dependencies.
config	Manage local and global configuration.
search	Search PyPI for packages.
cache	Inspect and manage pip's wheel cache.
wheel	Build wheels from your requirements.
hash	Compute hashes of package archives.
completion	A helper command used for command completion.
debug	Show information useful for debugging.
help	Show help for commands.

Advanced topics

- Dynamic import (__import__ function)
- * PIP freeze & PIP -r
- PIP Wheel
- What is a virtualenv? (Venv)