



Politecnico  
di Torino



# Data Science Lab

Structuring Python projects

Andrea Pasini  
Flavio Giobergia

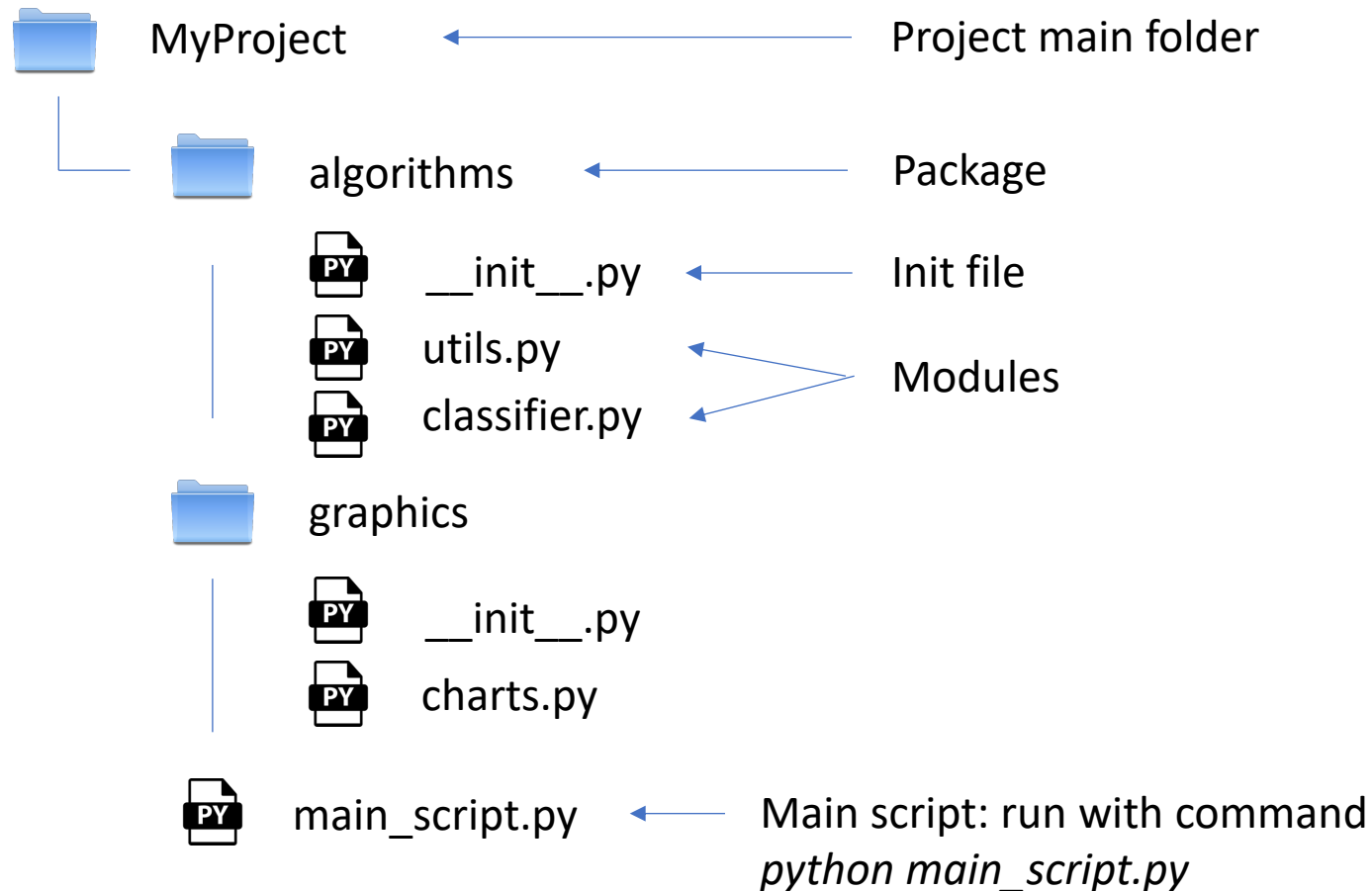
DataBase and Data Mining Group



- The code of a Python project is organized in **packages** and **modules**
- **Package:**
  - Represented with a directory in the file system
  - Collects a set of Python modules
  - Until Python 3.3, a folder used to have to contain a **`__init__.py`** file to be considered a Python package
- **Module**
  - Represented with a Python file (`.py`)
  - Contain **attributes**, **functions** and **class** definitions



## ■ Example: project structure





## ■ Example: content of Python module



graphics



\_\_init\_\_.py



charts.py

Functions/  
classes  
definition

Attribute

Initialization  
operations

charts.py

```
import matplotlib.pyplot as plt
import os

def plot_result():
    ...
    plt.plot(...)
    ...

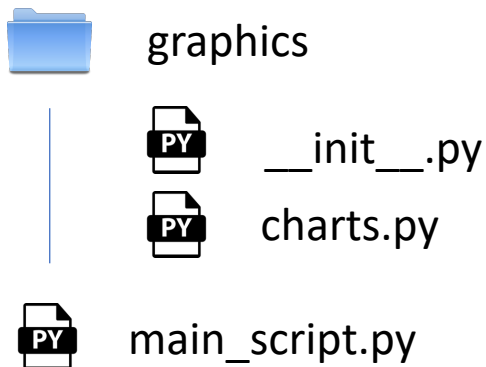
class Visualizer:
    ...

plot_path = "../Data/plots"
if not os.path.exists(plot_path):
    os.makedirs(plot_path)
```



## ■ Importing a module

- To use attributes, functions and classes defined in a module, it must be imported with the **import** command

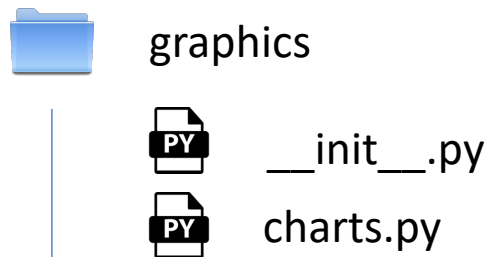


main\_script.py

```
import graphics.charts
```



- **Operations executed when importing a module (e.g. `graphics.charts`)**
  - Execute `__init__.py` file in the package
    - This file can be empty or contain some initialization instructions for the package
  - Execute the content of the module (**`charts.py`**)
    - This will load its attributes/functions/classes and run its initialization code (if any)





## ■ Operations executed when importing a module (e.g. `graphics.charts`)

```
import graphics.charts
```

charts.py

```
...  
def plot_result():  
    ...  
    plot_path = "../Data/plots"  
    if not os.path.exists(plot_path):  
        os.makedirs(plot_path)
```

1) Load a function

2) Load an attribute

3) Initialize directories



## ■ Importing a module

- After import, any attribute, function or class can be used from the module

main\_script.py

```
import graphics.charts
graphics.charts.plot_result()      # Function
print(graphics.charts.plot_path)  # Attribute
```





## ■ Importing the content of a module

- Avoids to write the name of the module when using attributes or functions

main\_script.py

```
from graphics.charts import *  
plot_result()      # Function  
print(plot_path)  # Attribute
```

**Warning**  
Avoid using `import *`  
Modules that use the  
same names will  
overwrite one another



- **Other ways of importing modules**

- Renaming a module

```
import graphics.charts as ch
ch.plot_result()
```

- Importing a single function/attribute

```
from graphics.charts import plot_result
plot_result()
```



- **The main script file(s)**
  - Typically contains a **main function**
  - Can also contain functions and classes as any other **module**
  
  - **2 use cases**
    - Run "python main\_script.py" from terminal
      - Execute the main function
    - Import some content (e.g. a function) of the main script
      - "from main\_script import my\_func"



## ■ The main script file(s)

- Use case 1: "python main\_script.py"
- Python defines a variable called `__name__`
  - Contains the name of the current module
  - "`__main__`" if executed directly
- The if statement is satisfied and `main()` is called

main\_script.py

```
def main():  
    print("This is the main function")  
  
if __name__ == '__main__':  
    main()
```



## ■ The main script file(s)

- Use case 2: import content to another file
  - `main_script.py` is executed by the **import**, but `main()` is not called since `__name__` does not contain `'__main__'`

`main_script.py`

```
def my_function():  
    ... do something ...  
  
...  
  
if __name__ == '__main__':  
    main()
```

`main_script2.py`

```
import main_script  
main_script.my_function()
```