Creating and Working with Modules

CLARUSWAY©
WAY TO REINVENT YOURSELF

Table of Contents

1

Scripts & Modules Initialization

Can you recap the difference between **scripts** and **modules?**

Students, write your response!

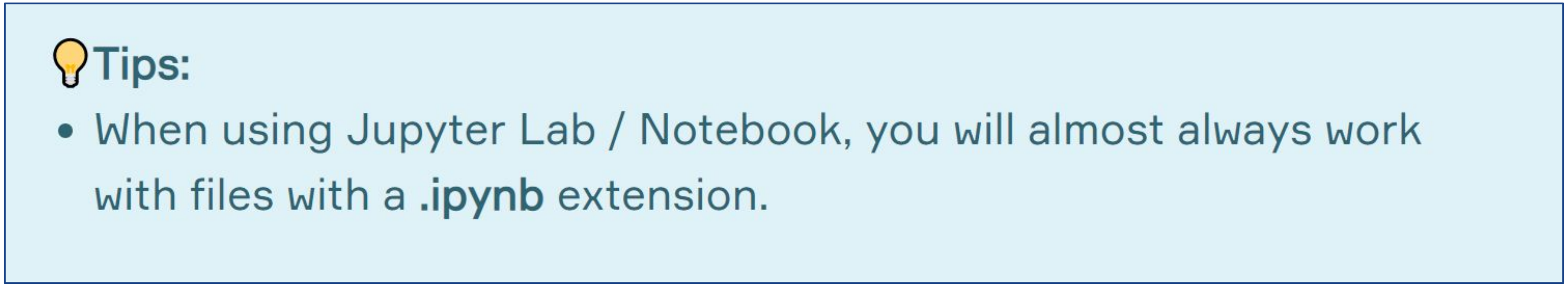
Scripts & Modules Initialization

Recap

- Scripts and modules have essentially identical structures ***in terms of creation*** and are the files with a **.py** extension, containing some Python codes, statements, operations, and functions.

Scripts & Modules Initialization

- In fact, if you're using an advanced IDE/IDLE, such as Jupyter Notebook/Lab (which we are) or Python IDLE, all these issues about the **scripts** and the **modules** don't make much sense. So, these applications have a user-friendly menu on such issues.



💡**Tips:**

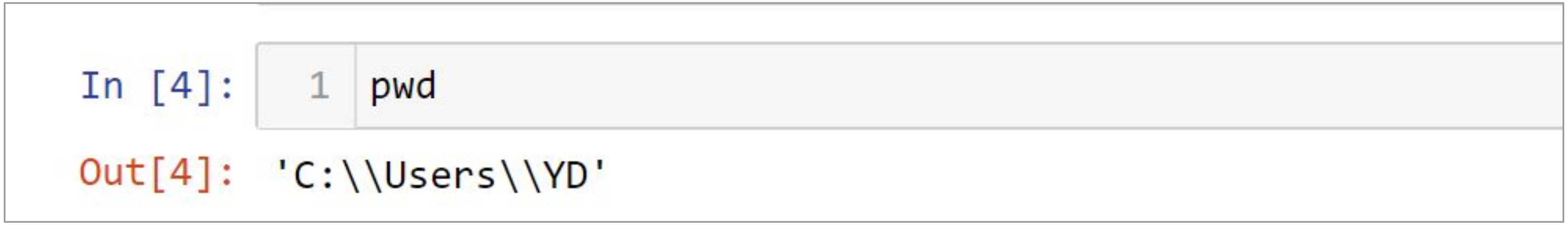
- When using Jupyter Lab / Notebook, you will almost always work with files with a **.ipynb** extension.

Scripts & Modules Initialization

- **Task :**
  - Create a **file** named `my_first` with **.py** extension containing of two simple user-defined *functions* and some *statements*.
  - Use it as a **script** and as a **module**.
  - **Call** some **functions**&**variables** and use it from your module.
  - Display the **docstring** of your module.

Scripts & Modules Initialization

- You can see the current path of your Jupyter using `pwd` command.

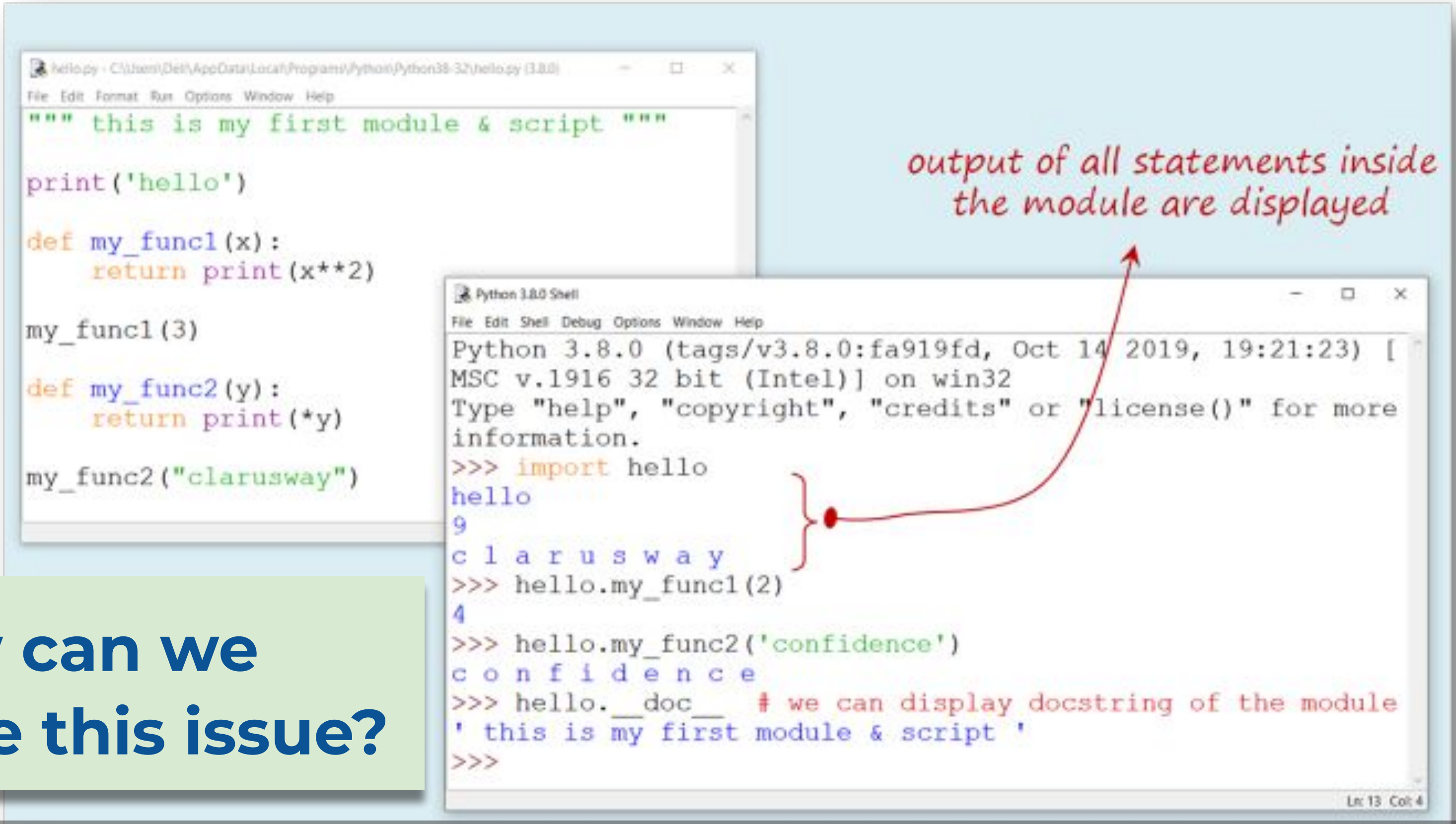


```
In [4]:   pwd
Out[4]: 'C:\\Users\\YD'
```

2

Working with the Modules (*Optional*)

Acting of a Module as a Script optional


hello.py - C:\Users\Dell\AppData\Local\Programs\Python\Python38-32\hello.py (3.8.0)
File Edit Format Run Options Window Help

```
""" this is my first module & script """

print('hello')

def my_func1(x):
    return print(x**2)

my_func1(3)

def my_func2(y):
    return print(*y)

my_func2("clarusway")
```

Python 3.8.0 Shell
File Edit Shell Debug Options Window Help

```
Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [
MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more
information.
>>> import hello
hello
9
c l a r u s w a y
>>> hello.my_func1(2)
4
>>> hello.my_func2('confidence')
c o n f i d e n c e
>>> hello.__doc__    # we can display docstring of the module
' this is my first module & script '
>>>
```

How can we solve this issue?

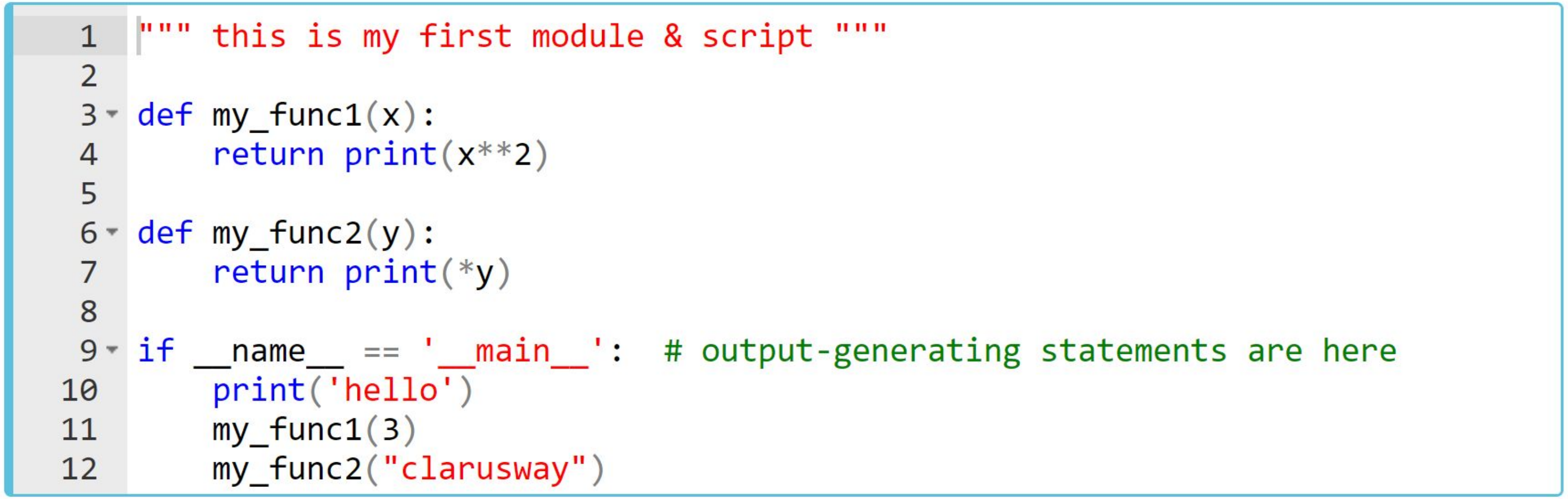
Acting of a Module as a Script **optional**

- As you see, when you want to import this file as a module, it acts as a script for the first importing, which is undesirable. It is not normal for a module to generate output when imported. Then why it happens?
- Well. As a Pythonic rule, when the file you created with **.py** extension is imported as a module, Python sets the specific variable `__name__` to the name of the module. But, if the file is run as a ***script***, variable `__name__` is set to the string value of "`__main__`". So, using this Pythonic rule, we can fix this issue.

`__name__`, "`__main__`" Method optional

- If we collect the output-generating statements which are in our module under if `__name__ == "__main__"` : statement we will solve the problem. Let's do it and see what will happen :

hello.py :

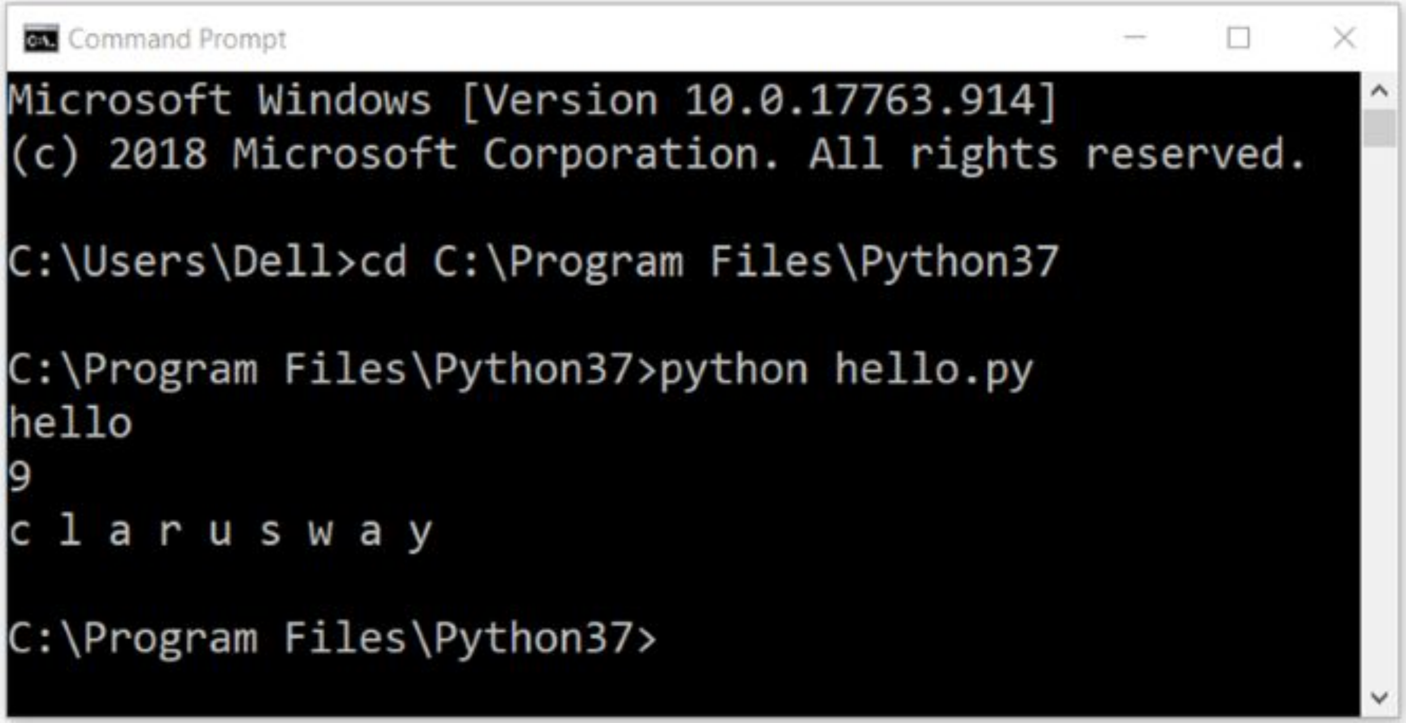


```python
""" this is my first module & script """

def my_func1(x):
    return print(x**2)

def my_func2(y):
    return print(*y)

if __name__ == '__main__':  # output-generating statements are here
    print('hello')
    my_func1(3)
    my_func2("clarusway")
```

Working with the Modules

optional

- Let's run it on the Command Prompt (console) as a ***script*** :



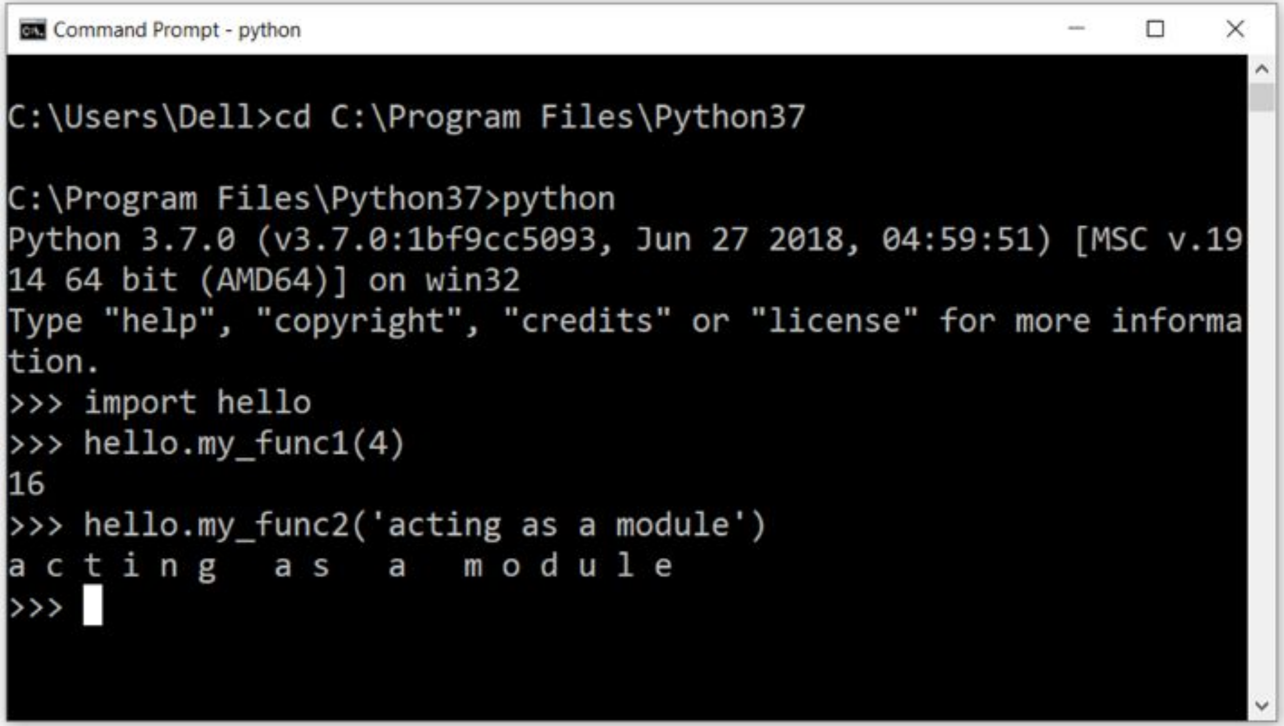
Command Prompt

```
Microsoft Windows [Version 10.0.17763.914]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Dell>cd C:\Program Files\Python37

C:\Program Files\Python37>python hello.py
hello
9
c l a r u s w a y

C:\Program Files\Python37>
```

Working with the Modules

optional

- Let's run it on the Command Prompt (console) as a ***module*** :



```
C:\Users\Dell>cd C:\Program Files\Python37

C:\Program Files\Python37>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.19
14 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>> import hello
>>> hello.my_func1(4)
16
>>> hello.my_func2('acting as a module')
a c t i n g   a s   a   m o d u l e
>>> 
```

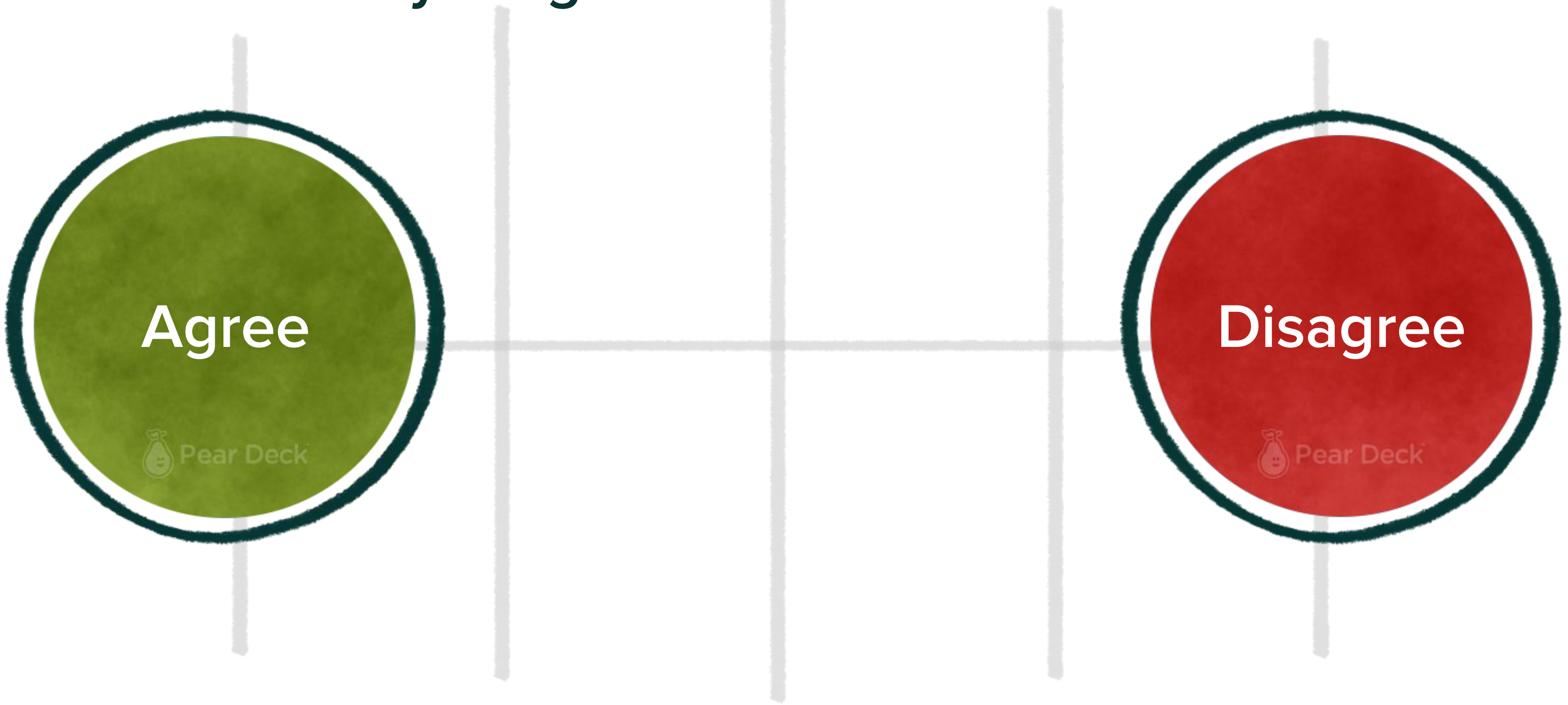
Packages

Table of Contents

1

Package Initialization

I’ve created and load a module on my own and I understood everything about them.



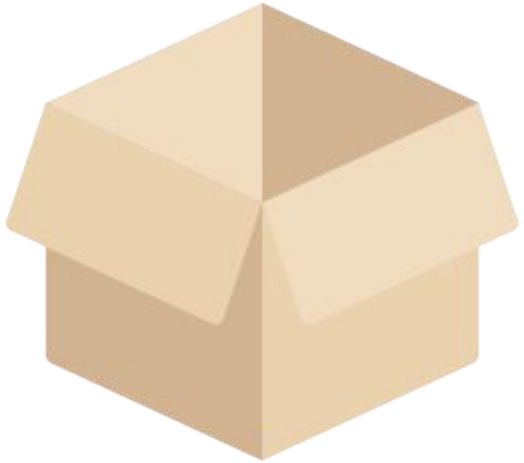
Students, drag the icon!

Package Initialization

- According to the official document of Python,

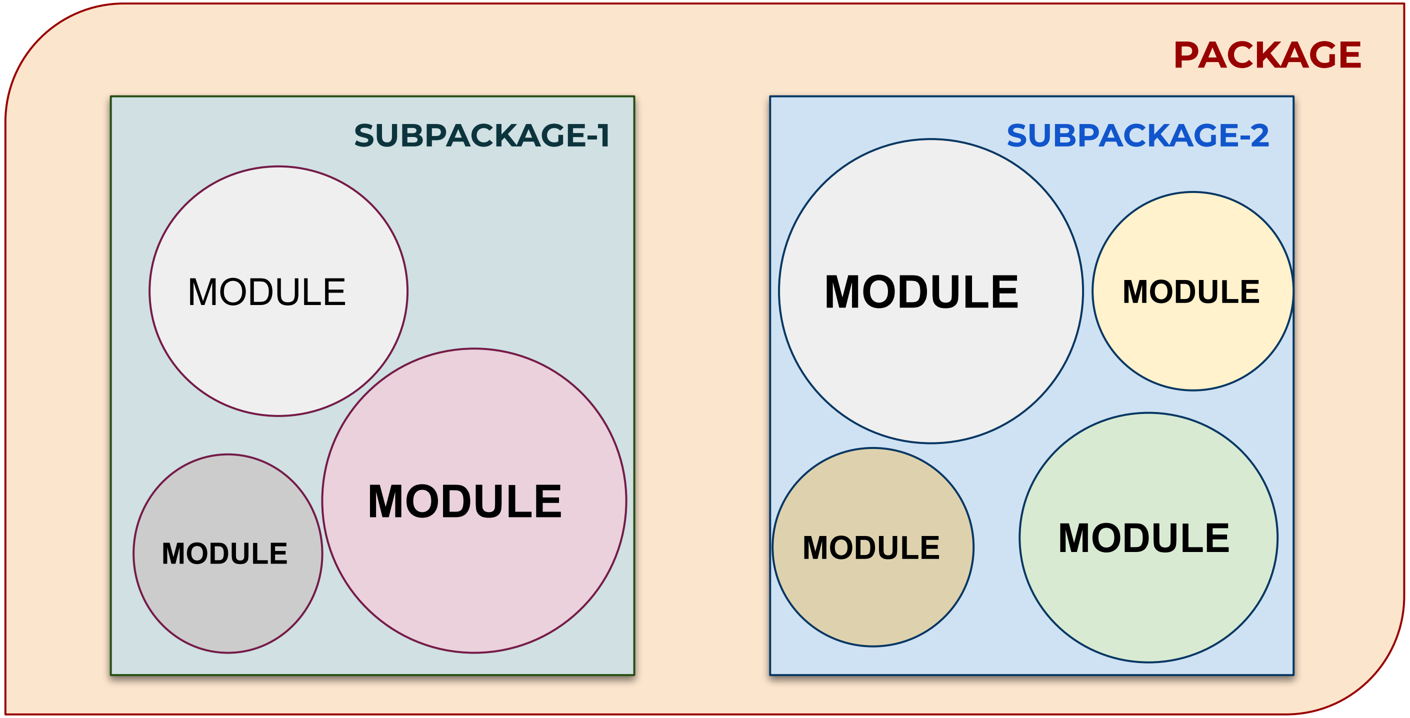
> Packages are basically a way of structuring Python's module namespace by using "**dotted module names**".

- In order to make the modules more systematically organized, we can use **packages**.



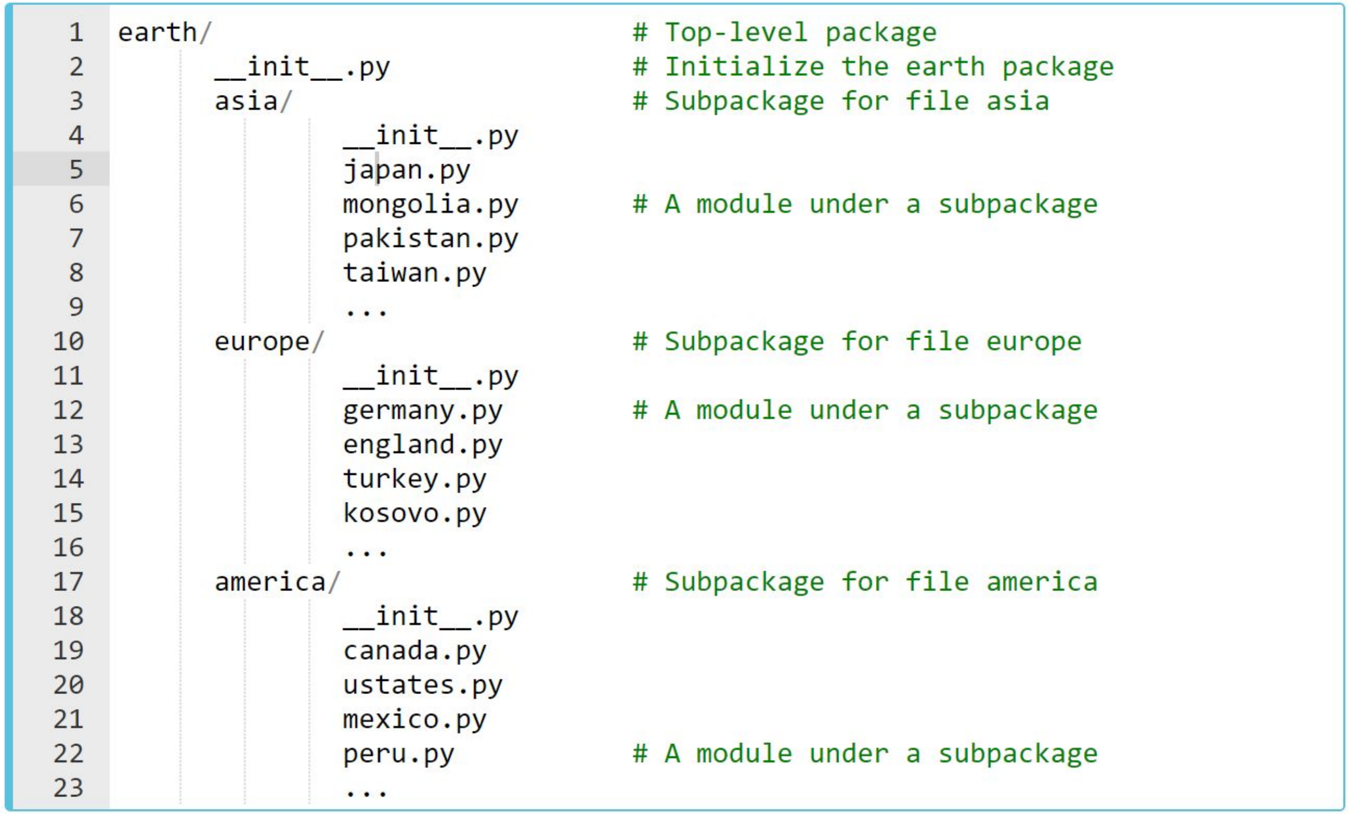
Package Initialization

- A sample diagram of package system of Python.



Package Initialization (review)

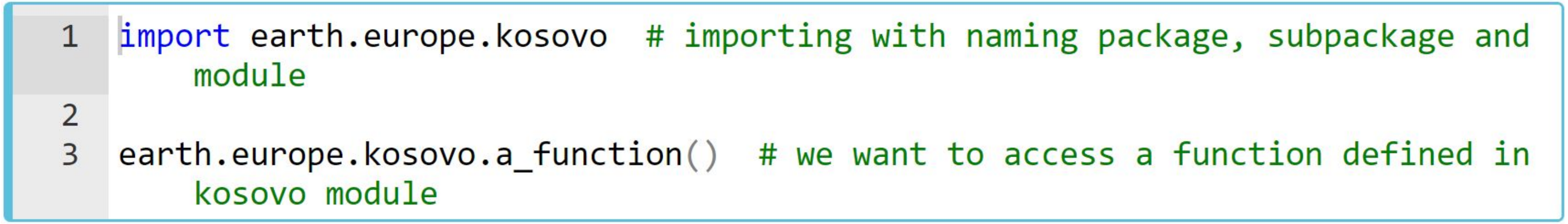
- Examine the basic structure of the package system:



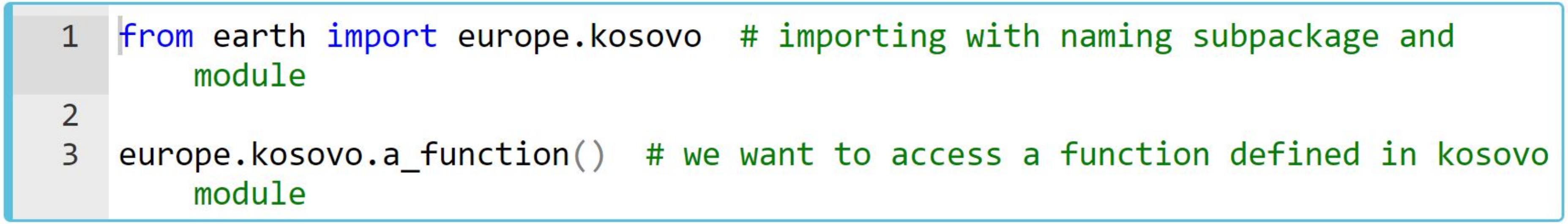
```
earth/                          # Top-level package
      __init__.py               # Initialize the earth package
      asia/                     # Subpackage for file asia
              __init__.py
              japan.py
              mongolia.py       # A module under a subpackage
              pakistan.py
              taiwan.py
              ...
      europe/                   # Subpackage for file europe
              __init__.py
              germany.py        # A module under a subpackage
              england.py
              turkey.py
              kosovo.py
              ...
      america/                  # Subpackage for file america
              __init__.py
              canada.py
              ustates.py
              mexico.py
              peru.py           # A module under a subpackage
              ...
```

Package Initialization (review)

- The hierarchical model of dot notation used to access and work with a module works as follows. The importing syntax which *shows the entire hierarchy* is so-called **absolute importing**.



```python
import earth.europe.kosovo  # importing with naming package, subpackage and module

earth.europe.kosovo.a_function()  # we want to access a function defined in kosovo module
```



```python
from earth import europe.kosovo  # importing with naming subpackage and module

europe.kosovo.a_function()  # we want to access a function defined in kosovo module
```

Package Initialization (review)

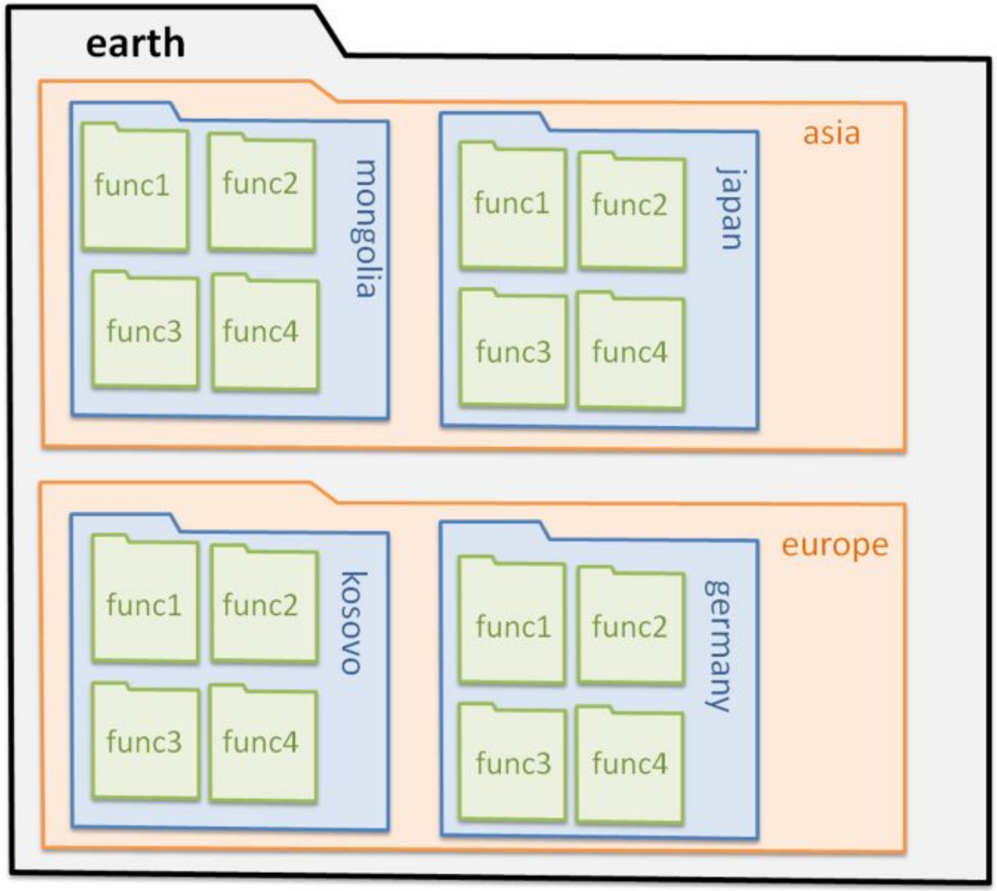
```python
from earth.europe import kosovo  # importing without naming package and subpackage

kosovo.a_function()  # we want to access a function defined in kosovo module
```

```python
from earth.europe.kosovo import a_function  # importing without any naming

a_function()  # we use directly the function's name
```

💡Tips:

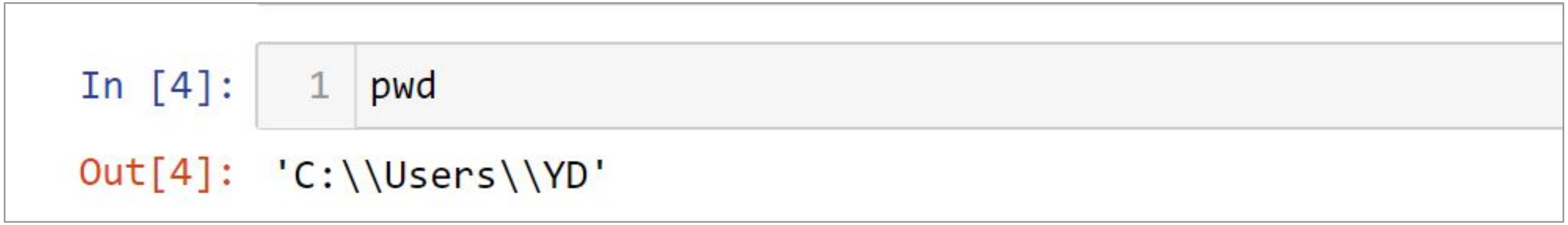
- Which style you should use depends on your needs. But the key point is readability!

Structure of a Package



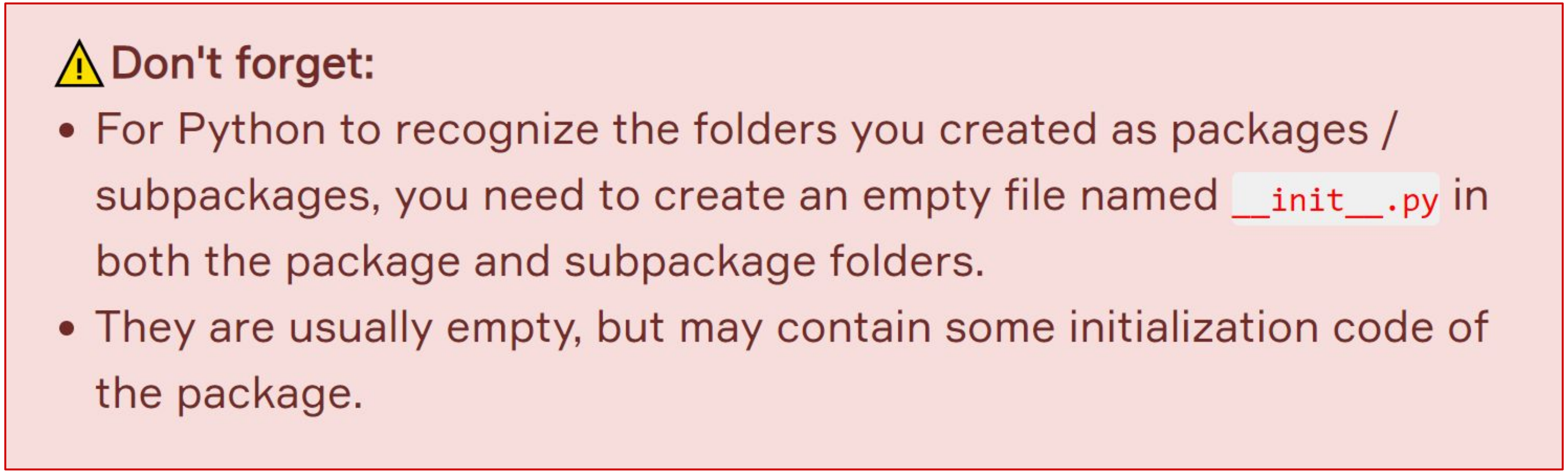
Package Initialization

- You can see the current path of your Jupyter using `pwd` command.



```
In [4]:   pwd
Out[4]: 'C:\\Users\\YD'
```

Package Initialization (review)

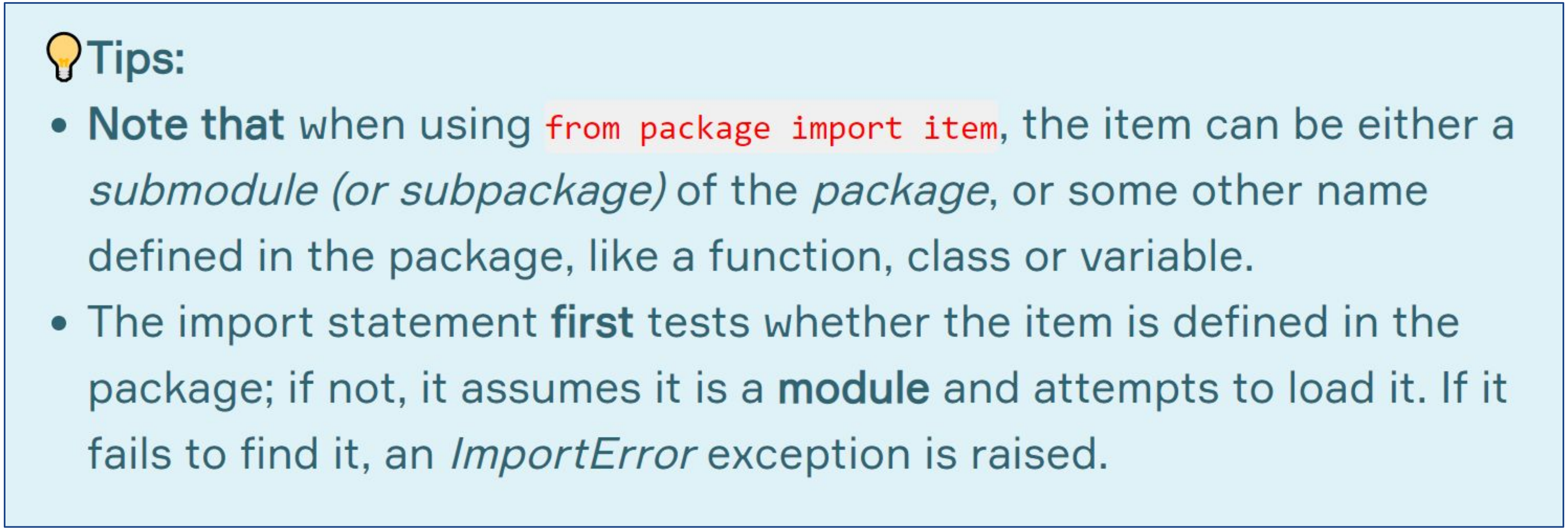


⚠ **Don't forget:**

- For Python to recognize the folders you created as packages / subpackages, you need to create an empty file named `__init__.py` in both the package and subpackage folders.
- They are usually empty, but may contain some initialization code of the package.

- When you need to reorganize your modules with the packaging system, you need to create package/subpackage folders in the directory where Python is installed. Of course, keep in mind that you have to put a file named **__init__.py** in the folders you will create.

Package Initialization (review)

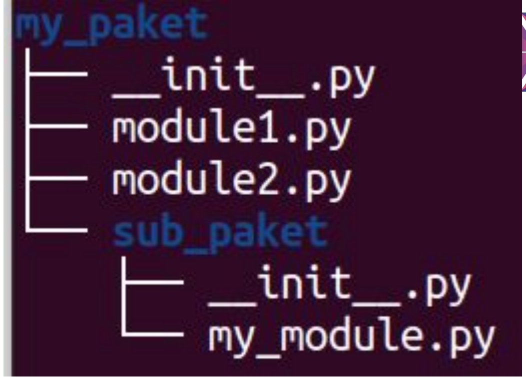


💡Tips:

- **Note that** when using `from package import item`, the item can be either a *submodule (or subpackage)* of the *package*, or some other name defined in the package, like a function, class or variable.
- The import statement **first** tests whether the item is defined in the package; if not, it assumes it is a **module** and attempts to load it. If it fails to find it, an *ImportError* exception is raised.

Package Initialization

- **Task :**
  - Create two **files** named `module1`-`module2` with **.py** extension to be used as a **module** containing of two simple user-defined *functions* each and some *statements*.
  - Create a **package** named `my_paket` containing a **subpackage** named `sub_paket`.
  - **Call** some **functions**&**variables** and use it from your modules using absolute importing methods.



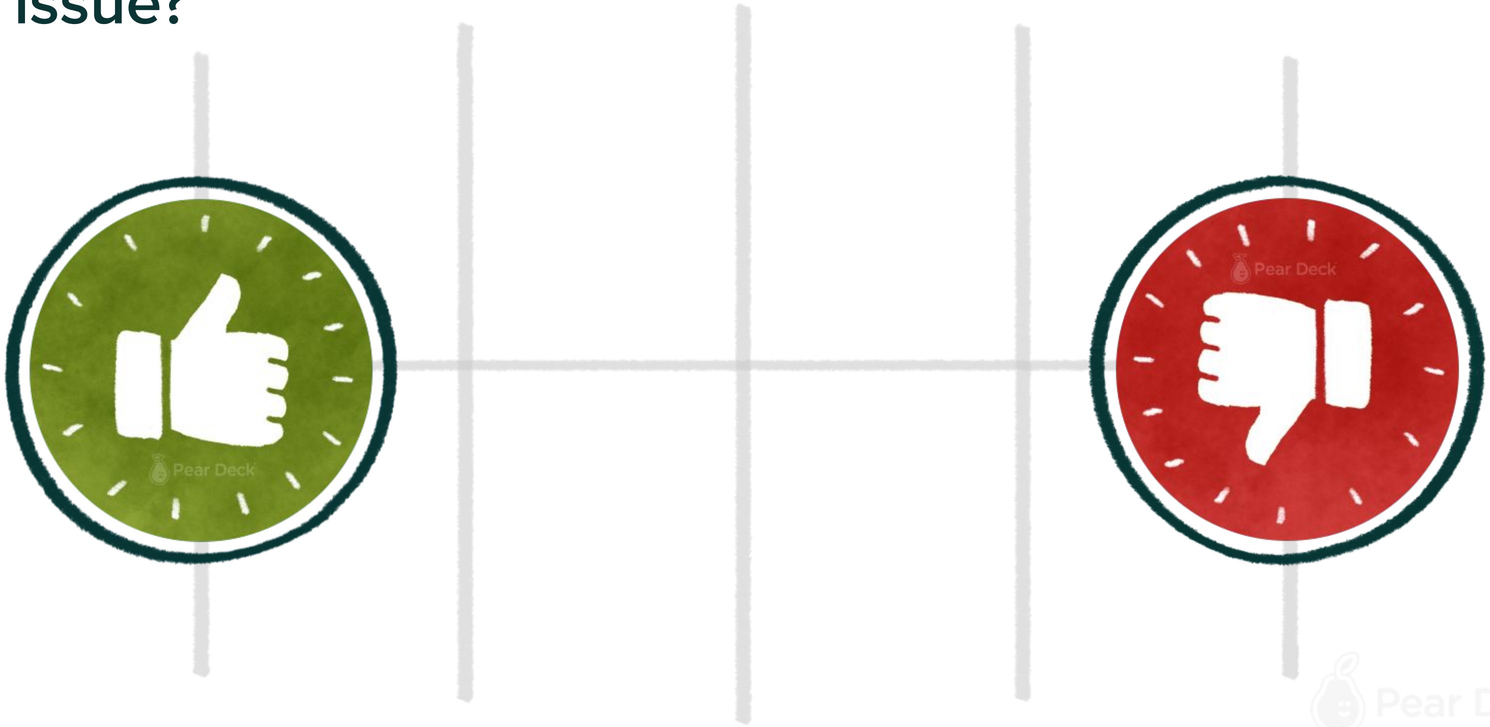
```
my_paket
├── __init__.py
├── module1.py
├── module2.py
└── sub_paket
    ├── __init__.py
    └── my_module.py
```

4

pip - The Package Manager for Python

CLARUSWAY©
WAY TO REINVENT YOURSELF

How was your pre-class preparation? Did you understand `pip` issue?



What is `pip`?

- `pip` is the standard package manager for Python.
- It allows you to install/uninstall, and manage additional packages that are not part of the Python standard modules.

What is `pip`? (review)

- You can use `pip` not only to give additional functionality to the standard library by installing additional packages on your computer, but you can also use it to help you contribute to Python's development by sharing your own projects.
- Now open your command prompt and run the following syntax to make sure that you have pip installed.

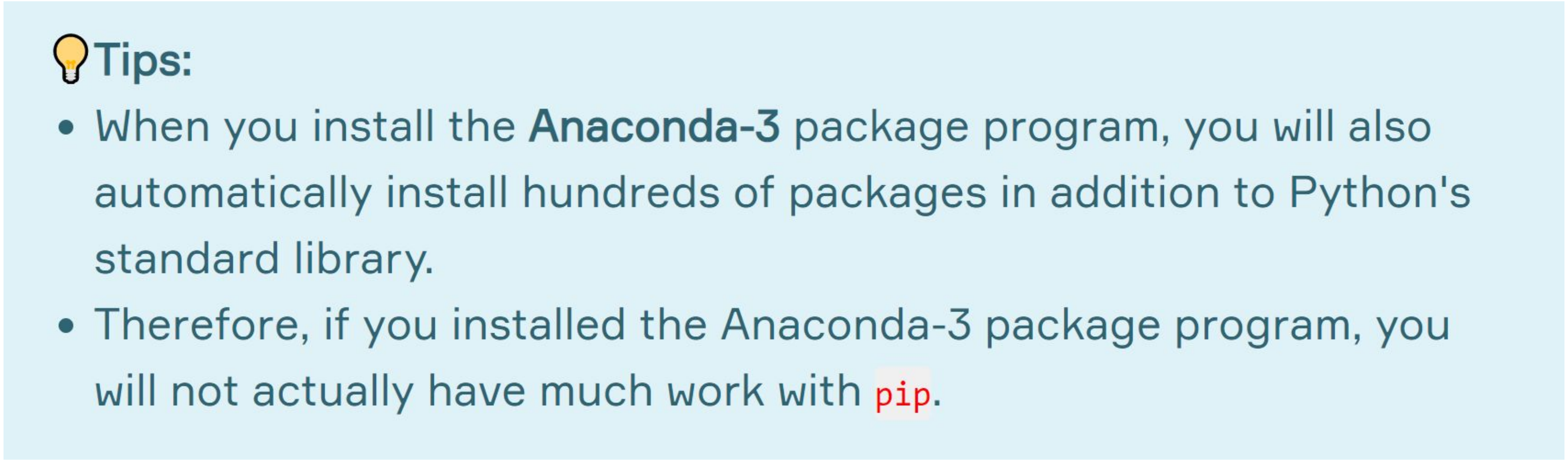
```
pip --version

```

- This code should display your valid pip version which is 19.3.1 currently. The output will be :

```
pip 19.3.1
```

What is `pip`? (review)

- If you have problems with installing or upgrading **`pip`**, you can follow the **official guide** for the best practice.



💡**Tips:**

- When you install the **Anaconda-3** package program, you will also automatically install hundreds of packages in addition to Python's standard library.
- Therefore, if you installed the Anaconda-3 package program, you will not actually have much work with `pip`.

CLARUSWAY©
WAY TO REINVENT YOURSELF

Working with `pip` (review)

```
The formula syntax is : pip command options
```

`install`

- The most common and essential command of `pip` is of course `install`. The most common syntax is :

```
pip install my_package
```

- If you want, you can use this command by adding the version number to the end of the syntax as follows :

```
pip install my_package==3.2.1
```

Working with `pip` (review)

`install`

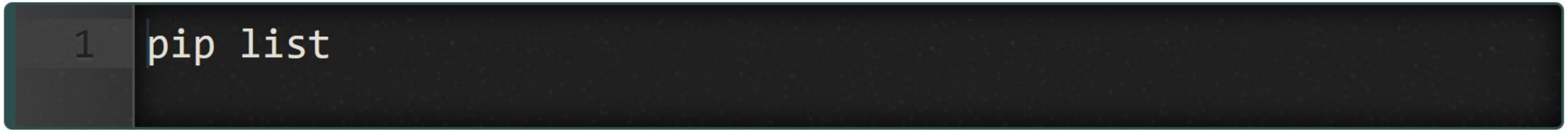
- For the Python's current version you can use the following command. Although Python is **not** actually a ***package***, you can also install it as follows. You do not need to try it because it will be faster if you download and install it from its website.

```
pip install python==3.8.1
```

Working with `pip` (review)

`list`

- Another important command you should learn is **`list`**. It lists all the packages you have installed on your computer in alphabetical order and in two columns.



```
pip list
```

Working with `pip` (review)

`show`

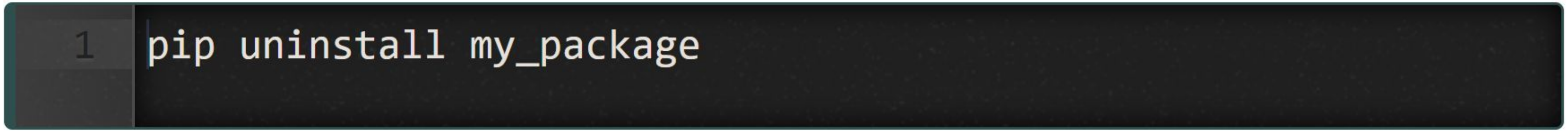
- The other useful command we can mention is `show`.
- It's used to view some information about the packages. These information about a package will be : ***Name, Version, Summary, Home-page, Author, Author-email, License, Location on PC***.

```
pip show my_package
```

Working with `pip` (review)

`uninstall`

- And the last command we want to show you is uninstall. It uninstalls the installed packages from your computer.



```
pip uninstall my_package
```

Working with `pip`

- **Task :** Using `pip` command ;
  - **List** all packages already installed on your device,
  - **Install** **`numpy`** and **`pandas`** packages,
  - **Display** the information of these packages,
  - **List** all packages again that installed on your device.

You don't need to know what *these packages* used for.

THANKS!

Any questions?

You can find me at:

- @joseph
- joseph@clarusway.com