Packages

Table of Contents

1 Package Initialization

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

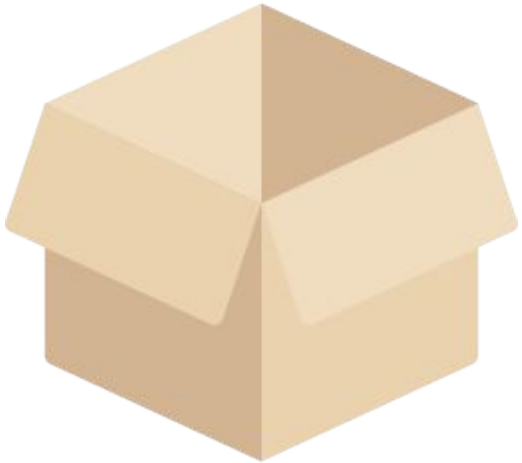
Students, drag the icon!

Package Initialization

- According to the official document of Python,

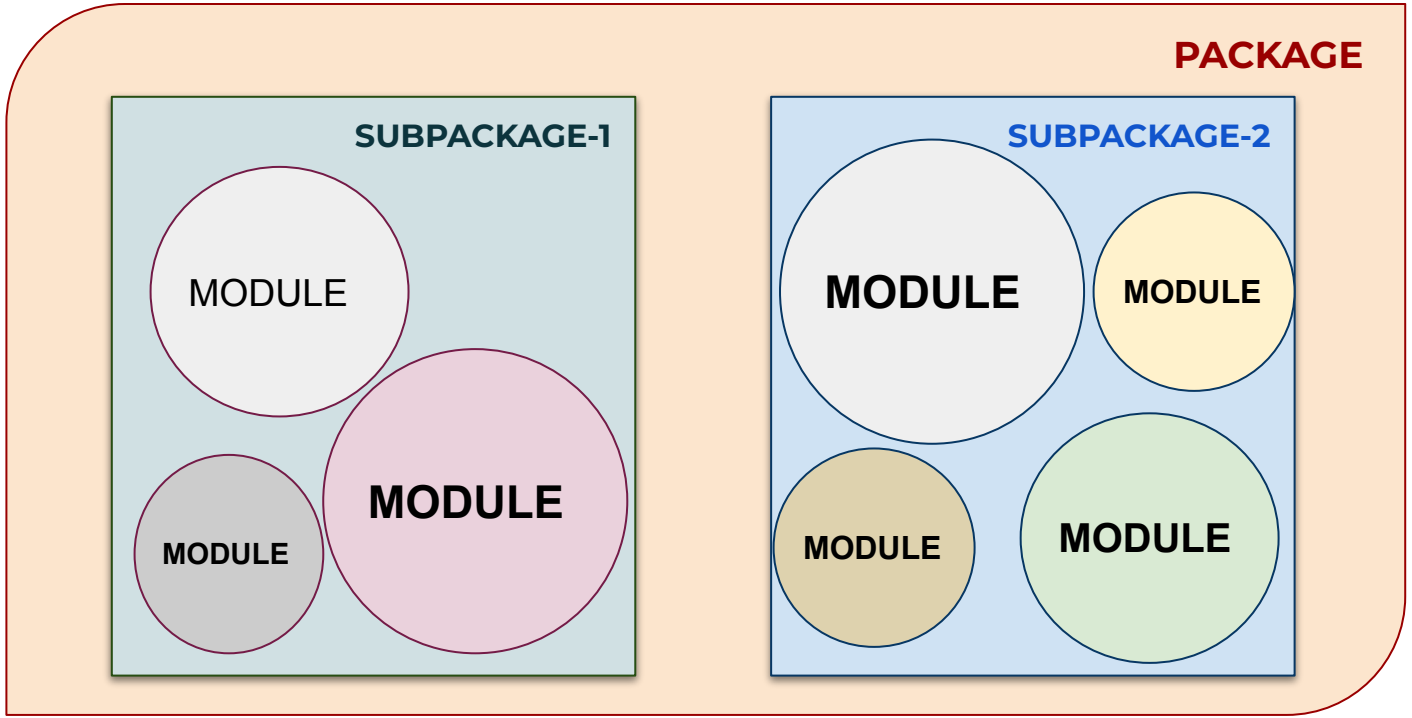
> Packages are basically a way of structuring Python's module namespace by using "**dotted module names**".

- 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**.

```
1  import earth.europe.kosovo  # importing with naming package, subpackage and
       module
2
3  earth.europe.kosovo.a_function()  # we want to access a function defined in
       kosovo module
```

```
1  from earth import europe.kosovo  # importing with naming subpackage and
       module
2
3  europe.kosovo.a_function()  # we want to access a function defined in kosovo
       module
```

Package Initialization (review)

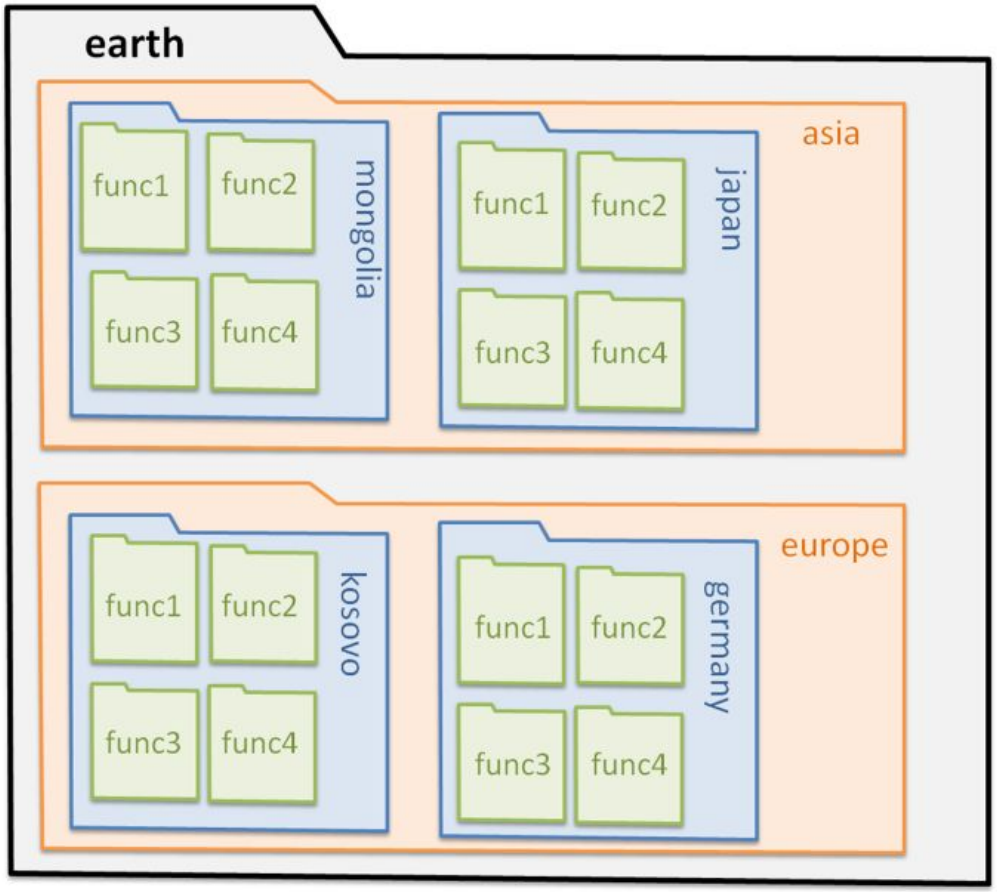
```
1  from earth.europe import kosovo  # importing without naming package and
       subpackage
2
3  kosovo.a_function()  # we want to access a function defined in kosovo module
```

```
1  from earth.europe.kosovo import a_function  # importing without any naming
2
3  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 (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 `my_module1`-`my_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_package` containing a **subpackage** named `my_subpack`.
  - **Call** some **functions**&**variables** and use it from your modules using absolute importing methods.

Package Initialization

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

```
In [4]: 1 pwd

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

3

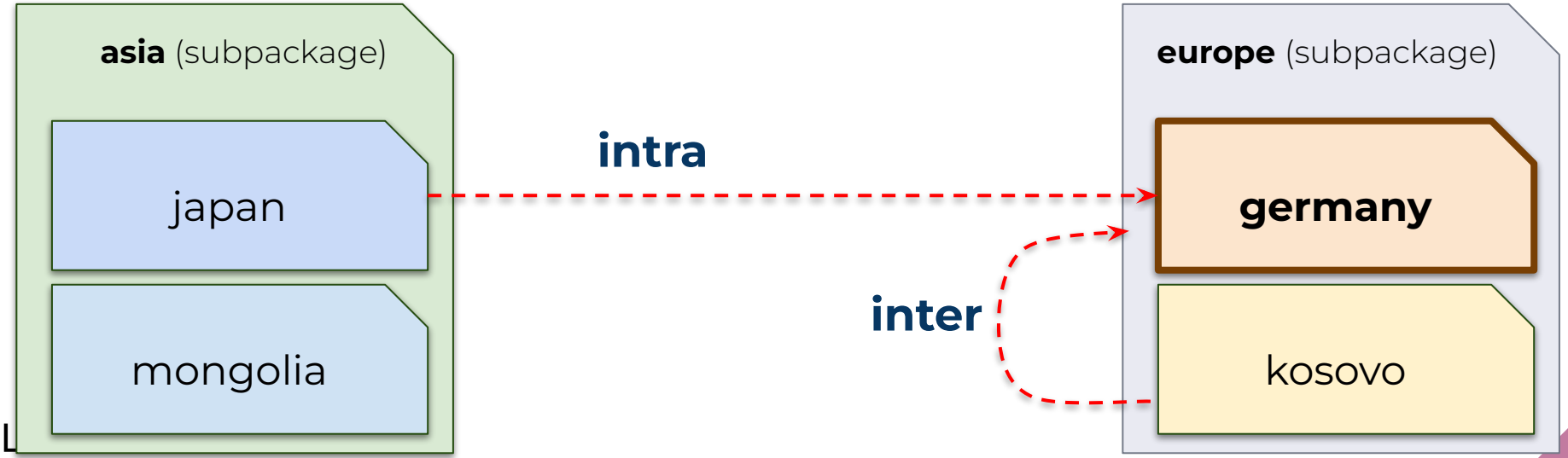
Working with Inter & Intra-Packages/Subpackages

Working with Inter & Intra-Packages/Subpackages

- If you need to work with a **module** defined in another **subpackage**, you can simply **import** it into the **current module** file.
- And of course, you can either **import** and use **another module** from the **current subpackage** you're working on.

Working with Inter & Intra-Packages/Subpackages

- Example : You are working on `earth.europe.germany` in **germany.py** file, you can import and use module '**japan**' from subpackage **asia** by using the **absolute importing** : `from earth.asia import japan`.



Working with Inter & Intra-Packages/Subpackages

- In this case;

  There is another simple way to do the same thing. This style of syntax is so-called **relative imports.**

- Let's see how it works..

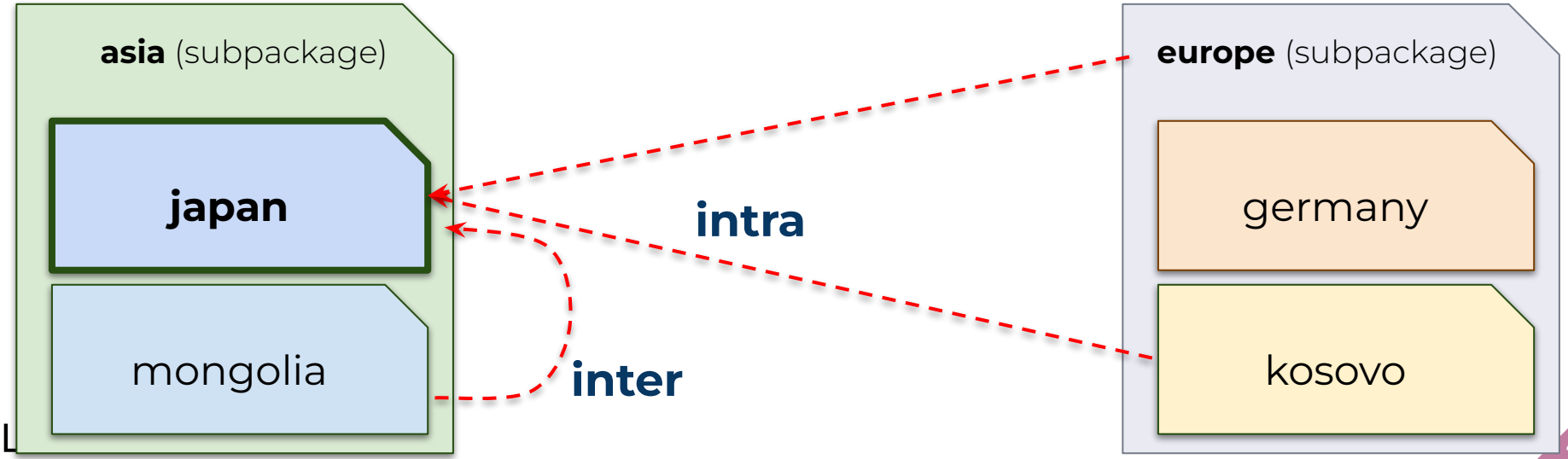
Working with Inter & Intra-Packages/Subpackages

- You can write **relative imports**, with the `from module import name` form of the import statement. These imports use leading **dots** to indicate the **current** and **parent packages** involved in the relative import. Let's see how it works for the module **japan** :

```
1  from . import mongolia  # one dot means addressing to a
       current package/subpackage
2
3  from .. import europe  # two dots mean addressing to a
       parent package/subpackage
4
5  from ..europe import kosovo  # subpackage name comes
       immediately after two dots
```

Working with Inter & Intra-Packages/Subpackages

- You can write **relative imports**, with the `from module import name` form of the import statement. These imports use leading **dots** to indicate the **current** and **parent packages** involved in the relative import. Let's see how it works for the module **japan** :



Working with Inter & Intra-Packages/Subpackages

- Here is another example of the **relative importing**: The modified **germany.py** file is below. We're importing and using module **japan** inside the **germany.py** file :

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

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

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

from ..asia import japan  # we've added importing syntax using two dots
japan.my_func1(6)         # and the name of parent subpackage (asia)

if __name__ == '__main__':
    print('hello')
    my_func1(3)
    my_func2("clarusway")
```

Working with Inter & Intra-Packages/Subpackages

- When we import the module **germany**, what is the output

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

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

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

from ..asia import japan  # we've added importing syntax using two dots
japan.my_func1(6)         # and the name of parent subpackage (asia)

if __name__ == '__main__':
    print('hello')
    my_func1(3)
    my_func2("clarusway")
```

We've called **my_func1** function from **japan** module into **germany** module

module **germany**

```
import earth.europe.germany

```

What is the output? Try to figure out in your mind...

Students, write your response!

Working with Inter & Intra-Packages/Subpackages

- When we import the module germany, we can see the result as follows :

```
1  import earth.europe.germany
2
```

```
1  36
2
```

We've imported **germany** module and **36** is appeared as an output.

Discuss How?

Working with Inter & Intra-Packages/Subpackages

💡**Tips:**

- **Absolute imports** syntax are preferred, as they are usually more readable.
- When dealing with very complex and sophisticated packages, it is preferable to use **relative imports** syntax, since using absolute imports will result in unnecessary redundancy.

4

`pip` - The Package Manager for Python

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`.

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.

```
1  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