Module 04 : Using modules and packages

Content

- Writing and using *modules* for code reuse and distribution
- Installing and using *third-party* Python *packages* with `pip`, the default Python package manager
- Create and setup a project with packages for data analysis in PyCharm

Modular programming: toward higher abstractions

In the previous module, we learn how to write *functions*, our lowest level *modules* (*black box*).

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable units or *modules* (like functions!)

Individual modules can then be cobbled together like building blocks to create new or larger applications.

Breaking large programs into parts

As your program gets *bigger*, you may want to *split it into several files* for easier maintenance and reusability.

Python has a way to put definitions in a file and use them in a script: such a file is called **a module**.

Definitions (variables, functions) from a module can be *imported* into other modules or into the *main* module (entry point of your program)

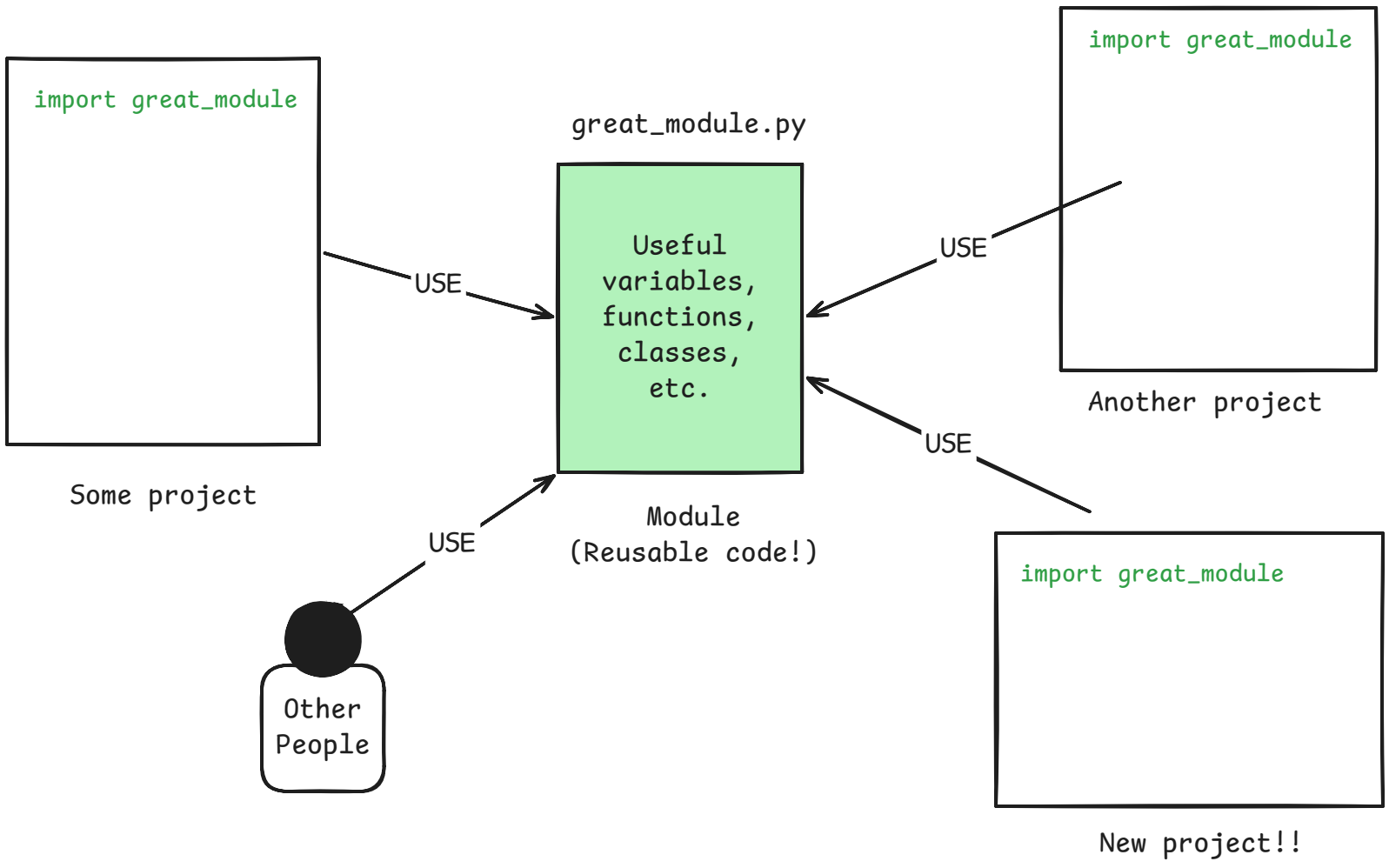
> We already have used built-in modules: `math`, `random`, `datetime`, etc. !

Modularizing advantages

- **Simplicity**: a module typically focuses on one relatively small portion of the problem. A module **does one thing** but does it correctly and completely
- **Reusability**: you can reuse modules in different parts of your application or in another projects. You can share your code with other peolple and use their code too!
- **Maintainability**: modules have boundaries. It's easier to work as a team on separate modules

> Functions, modules and packages are all constructs in Python that promote code modularization!

Share and reuse code across projects



What is a Python module, really?

A Python module is just a **file** containing Python definitions and statements.

The **file name** is **the module name** with the suffix `.py` appended.

Create a module

Create a file that contains Python code and then give the file a name with a `.py` extension. That's it!

For example, suppose you have created a file called `my_module.py` containing the following:

```
s = "An now, for something completely different"
a = [1, 1, 2]

def foo():
    print("function foo called")
```

Use the module, with the import statement

Create another file, `app.py` (your main program file) and import your module (they should be both in the same directory!). To make the module available to your program, use the `import my_module` statement.

To access module *content* (functions, variables, classes, etc.), **you must prefix each call with the module name** using the dot (`.`) operator:

```
import my_module

# Access module content (variables, functions) with the dot operator
print(my_module.s)
print(my_module.a)
my_module.foo()
```

> When the interpreter executes the above `import` statement, it searches for `my_module.py` in a list of directories including the current directory.

Alternatives

Aliasing a module name

```
# Import my_module with an alias
import my_module as mm

# s is accessed under the 'mm' namespace, hence we need the dot operator
print(mm.s)
```

Importing specific items into the global namespace directly

```
from my_module import foo, s

print(s)  # Notice that we don't need to prefix s and foo here
print(foo())
```

Beware of name collisions

Importing functions directly into the global namespace can *silently overwrit*e existing variables or functions with the same name. If you *already* have a `foo()` function into your main program (global namespace) it will overwrite and hide the imported one

```
# Imports foo from my_module directly into the global namespace under the name 'foo'.
from my_module import foo

# Redefines the identifier foo in the same "global scope". The original imported reference is overwritten and lost.
def foo():
    print("foo main program")

# Executes the local foo() function and prints "foo main program"
foo()
```

The last definition in the same scope always wins. Use aliases or namespace imports to keep scope clean.

Many ways to import modules

The `import` statement can take many different forms, shown below:

```
# Import a module
import <module_name>

# Multiple imports on the same line
import <module_name>[, <module_name> ...]

# Import only a few objects (functions) given by the module
from <module_name> import <name(s)>

# Import only a few objects (functions) given by the module under an alternate name (aliases)
from <module_name> import <name> as <alt_name>

# Import the entire module under an alternate name
import <module_name> as <alt_name>
```

Many ways to import modules, a few examples

```python
# Basic import of a single module
import math

# Usage: Access contents using the module name prefix
print(math.sqrt(16))  # Outputs: 4.0

# Multiple imports on the same line
import sys, os

# Usage: Access both modules independently
print("sys version:", sys.version)
print("os name: ", os.name)

# Import specific objects (functions/classes) directly
from datetime import datetime, date

# Usage: Use imported objects directly without module prefix
now = datetime.now()
today = date.today()
print(now, today)

# Import specific objects with alternate names (aliases)
from random import randint as ri

# Usage: Use the function via its alias 'ri' (which is a terrible name by the way!)
dice = ri(1, 6)
print(f"You rolled {dice}!")
```

Packages

A *package* is a *coherent collection of modules*. It's a *folder* containing multiple *modules* and *sub-packages* (recursive structure).

You can create your own packages (outside of the scope of this course) or **install useful third-party packages** provided by open-source community or external *vendors*. These include essential libraries for:

- Data manipulation: pandas
- Numerical computing: numpy
- Data visualization: matplotlib

Installing packages with PyCharm

PyCharm makes it *easy* to install, uninstall and upgrade Python packages.

By default, PyCharm uses pip to manage project packages.

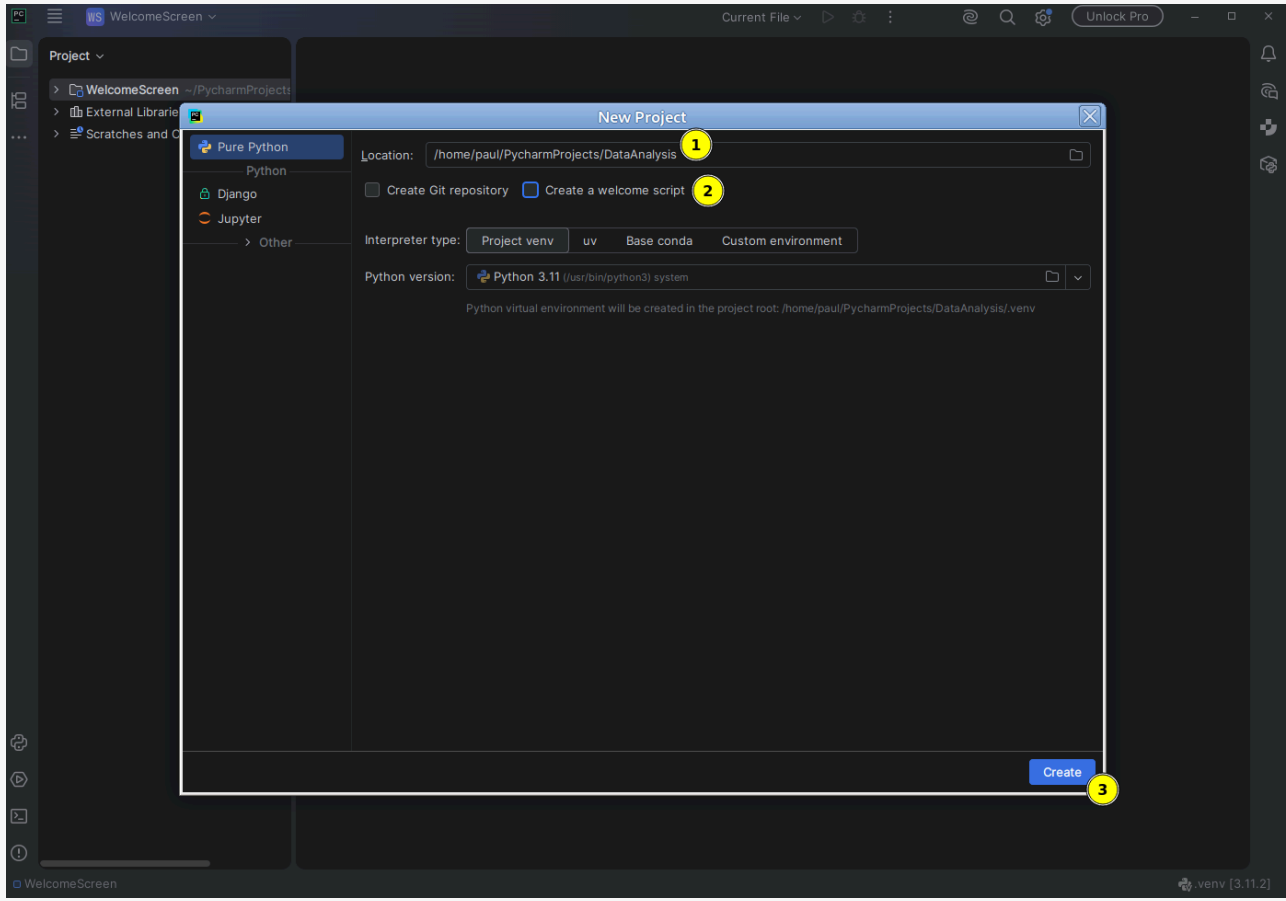
In PyCharm, you can *preview* and *manage* packages in the Python Packages tool window.

Setup a new project

Each *project* has *its own set of packages*.

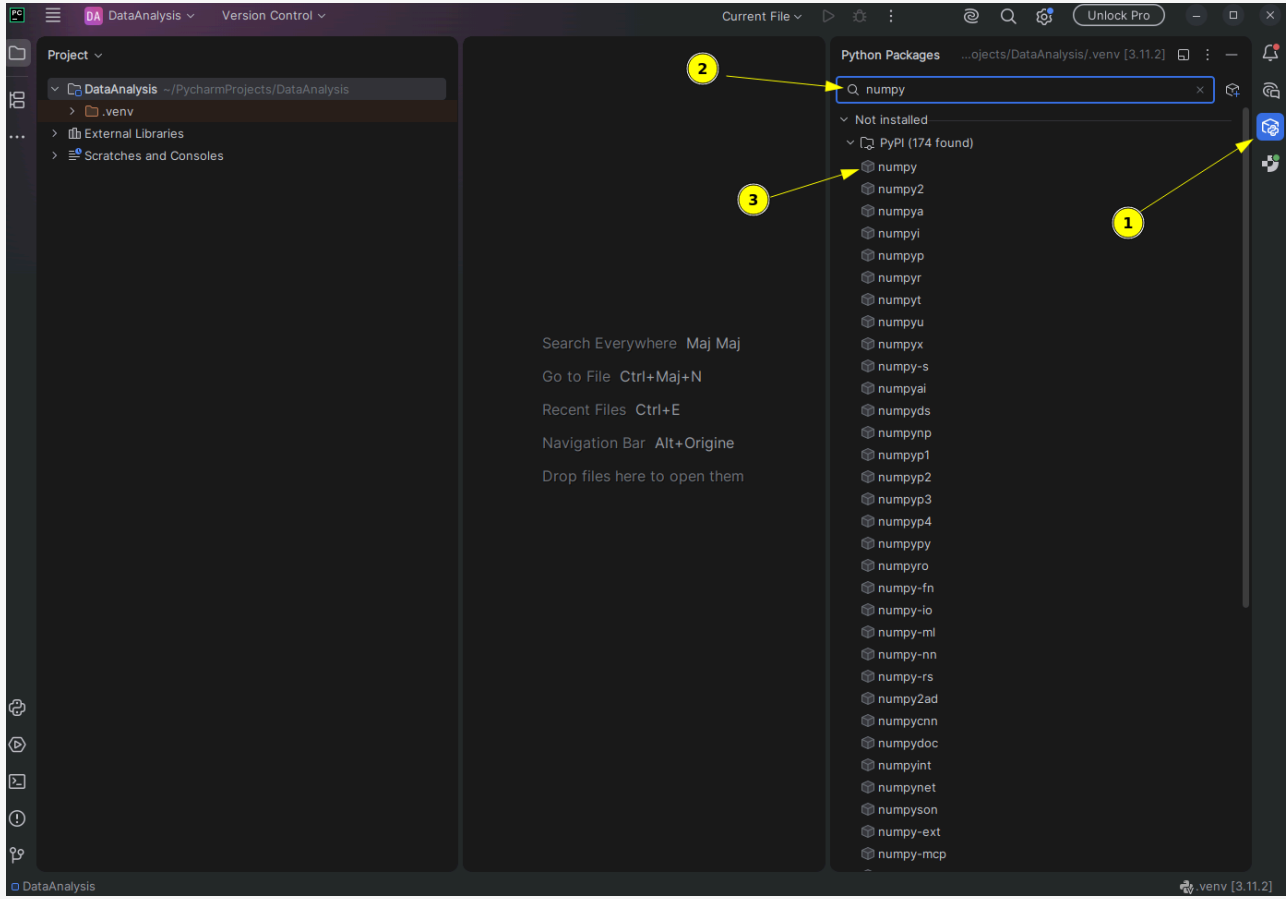
Create a new project (`File > New Project...`) called `DataAnalysis`

For the rest of this course, we are going to work *within this project*!

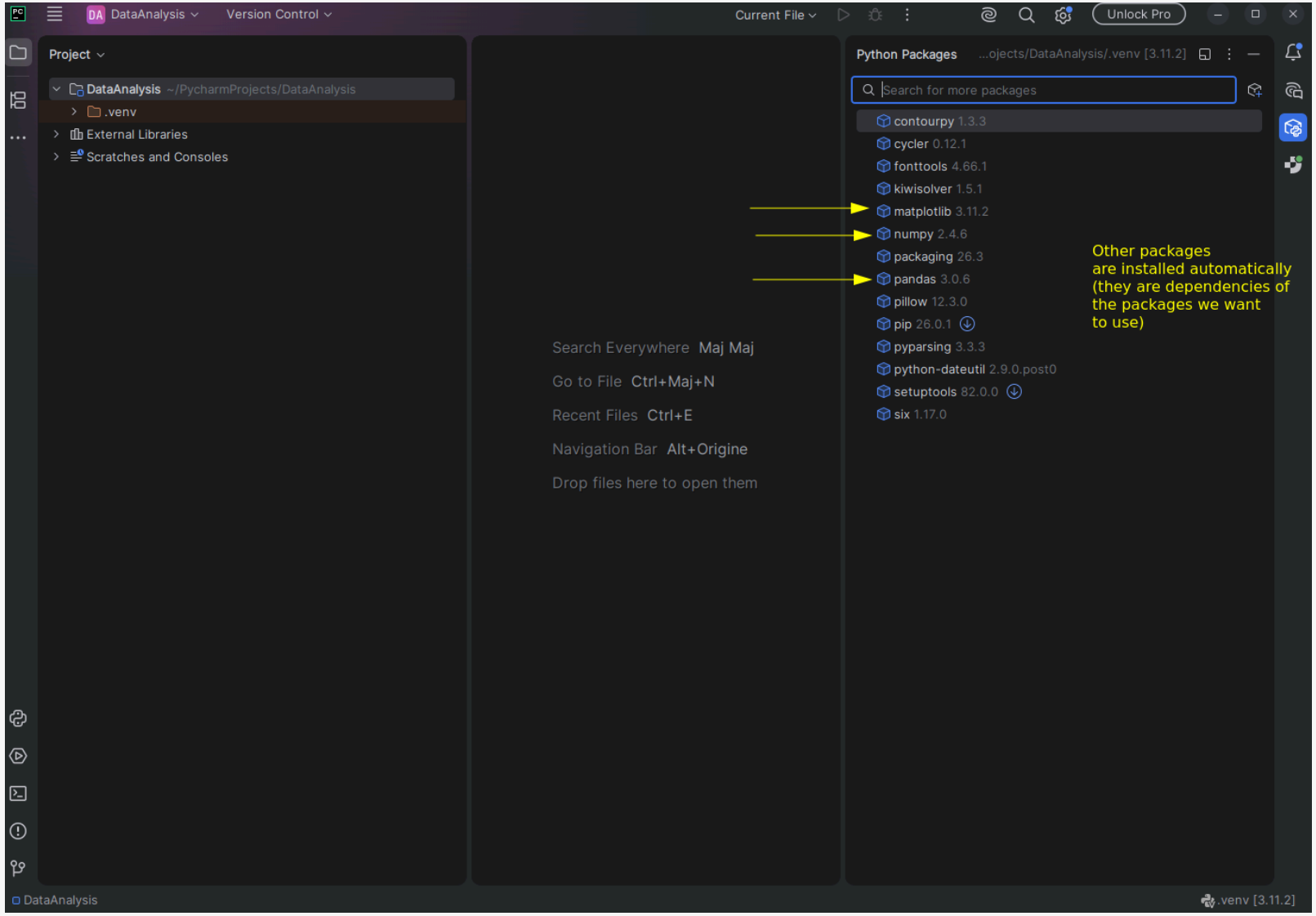


Install packages

1. **Open** the *Python Packages Menu*. You should see only `pip` (the package manager) for now
2. **Search** and **install** the following packages :
    i. `numpy`
    ii. `matplotlib`
    iii. `pandas`



Check all packages are installed



Write a testing program 1/2

1. **Create** a new Python file called `test.py`
2. **Copy/paste** the content below to check everything is working fine:

```
# Test to check everything is OK
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# NumPy: Create numerical data
x = np.linspace(0, 10, 100)
y = np.sin(x)
# Pandas: Store and manipulate data in a DataFrame
df = pd.DataFrame({"X": x, "Sin_X": y})
print("First 5 rows of data:")
print(df.head())
# Matplotlib: Plot the data
plt.plot(df["X"], df["Sin_X"], label="Sin(x)")
plt.title("If you see the sin curve, everything is OK !")
plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
# Display the plot
plt.show()
```

Write a testing program 1/2

3. **Run** the program. **Check** the console. **Check** the visualization window.
4. **Explore** the options. **Save** the plot using SVG format.

Next Module

Go to next module