Working with Modules and Packages in Python

Modules

A **module** is simply a Python file (`.py`) that contains definitions — variables, functions, or classes — that can be reused in other programs.

Modules help in:

- Organizing large programs into smaller, manageable pieces.
- Reusing code across different projects.
- Improving readability and maintainability.

Any `.py` file can act as a module.

Types of Modules

Python provides three main categories of modules:

1. **Built-in modules**: already included with Python (`math` , `sys` , `os` , etc.)
2. **External modules**: need to be installed from PyPI
3. **User-defined modules**: created by user

Importing built-in modules

```
In [ ]: import math
```

```
In [ ]: # Accessing module functions
        print(math.sqrt(16))    # 4.0
        print(math.pi)    # 3.14159...
```

```
4.0
3.141592653589793
```

Aliasing a module

```
In [ ]: import math as m
        print(m.factorial(5))    # 120
```

```
120
```

Importing specific functions or constants

```
In [ ]: from math import sqrt, pi
        print(sqrt(25))
        print(pi)
```

```
5.0
3.141592653589793
```

Importing everything from a module (not recommended, highly discouraged)

```
In [ ]: from math import *
        print(sin(pi/2))    # Works, but can cause name clashes
```

```
1.0
```

Creating and importing user-created modules

Create a file named `myutils.py` that contains the following:

```
# --- myutils.py ---
def greet(name):
    return f"Hello, {name}!"

def is_even(n):
    return n % 2 == 0

pi = 3.1416
g = 9.81
```

Then you can try running:

```
import myutils
print(myutils.greet("John Doe"))
print(is_even(5))
print(myutils.pi)
print(myutils.g)
```

Where Python looks for modules

When you use `import`, Python looks for the module in a list of directories stored in `sys.path`.

You can inspect it:

```
In [ ]: import sys
        print(sys.path)
```

```
['/content', '/env/python', '/usr/lib/python312.zip', '/usr/lib/python3.12', '/us
r/lib/python3.12/lib-dynload', '', '/usr/local/lib/python3.12/dist-packages', '/u
sr/lib/python3/dist-packages', '/usr/local/lib/python3.12/dist-packages/IPython/e
xtensions', '/root/.ipython']
```

You can also place your custom module in any of these directories, or modify `sys.path` at runtime if needed.

Packages

A **package** is a way of organizing related modules into directories.
Each folder that acts as a package contains a special file called `__init__.py`.

Example structure:

```
project/
│
├── mypackage/
│   ├── init.py
│   ├── greetings.py
│   └── math_ops.py
└── main.py
```

- `mypackage` is a **package**
- `greetings.py` and `math_ops.py` are **modules**
- `__init__.py` makes `mypackage` a **package** (it can be empty)

```
# --- greetings.py ---
def say_hello():
    return "Hello from the package!"

# --- math_ops.py ---
def square(n):
    return n ** 2

# --- main.py ---
# Importing from a package
from mypackage import greetings, math_ops

print(greetings.say_hello())
print(math_ops.square(5))
```

You can also initialize something inside `__init__.py` if you want some code to run when the package is imported, or to make importing submodules more convenient.

```
# --- __init__.py ---
from .greetings import say_hello
from .math_ops import square
```

Now we can import directly from the package:

```
# --- main.py ---
from mypackage import say_hello, square

print(say_hello())
print(square(3))
```

Installing and Managing External Packages

Python's **PyPI (Python Package Index)** hosts thousands of third-party modules.
We use `pip` to install, update, or uninstall them.

Some useful commands:

```
# Install a package
```

```
pip install requests

# Upgrade a package
pip install --upgrade requests

# Uninstall a package
pip uninstall requests

# List all installed packages
pip list
```

To install packages from inside a Colab/Jupyter notebooks, one can run:

```
!pip install requests
```

Virtual environments (venv)

A **virtual environment** is an isolated workspace for your Python projects.

It keeps dependencies separate between projects — avoiding conflicts when different projects require different package versions.

They are **highly recommended** for all development work.

```
# Create a virtual environment named "env"
python -m venv env

# Activate it (Windows)
env\Scripts\activate

# Activate it (macOS/Linux)
source env/bin/activate

# Install packages inside the virtual environment
pip install requests

# Deactivate it when done
deactivate
```

Why use virtual environments?

- Avoids version conflicts between projects.
- Keeps system Python clean.
- Makes project setup reproducible (you can share a `requirements.txt` file).

Generating a requirements file:

```
pip freeze > reqs.txt
```

There are third-party tools that streamline the process of creating and managing virtual environments and managing packages in these environments, for example: conda, mamba, micromamba, uv etc.

```
In [1]: import pandas
```

```
In [3]: !pip install pandas -U
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages
(2.2.2)
Collecting pandas
  Downloading pandas-2.3.3-cp312-cp312-manylinux_2_24_x86_64.manylinux_2_28_x86_6
4.whl.metadata (91 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 91.2/91.2 kB 3.2 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.26.0 in /usr/local/lib/python3.12/dist-pa
ckages (from pandas) (2.0.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.1
2/dist-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-pac
kages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-p
ackages (from pandas) (2025.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-package
s (from python-dateutil>=2.8.2->pandas) (1.17.0)
Downloading pandas-2.3.3-cp312-cp312-manylinux_2_24_x86_64.manylinux_2_28_x86_6
4.whl (12.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 12.4/12.4 MB 86.9 MB/s eta 0:00:00
Installing collected packages: pandas
  Attempting uninstall: pandas
    Found existing installation: pandas 2.2.2
    Uninstalling pandas-2.2.2:
      Successfully uninstalled pandas-2.2.2
ERROR: pip's dependency resolver does not currently take into account all the pac
kages that are installed. This behaviour is the source of the following dependenc
y conflicts.
google-colab 1.0.0 requires pandas==2.2.2, but you have pandas 2.3.3 which is inc
ompatible.
cudf-cu12 25.6.0 requires pandas<2.2.4dev0,>=2.0, but you have pandas 2.3.3 which
is incompatible.
dask-cudf-cu12 25.6.0 requires pandas<2.2.4dev0,>=2.0, but you have pandas 2.3.3
which is incompatible.
Successfully installed pandas-2.3.3
```

```
In [ ]:
```