Virtual environments in Python

Sergio Paniego Blanco
@sergiopaniego
sergiopaniegoblanco@gmail.com
https://sergiopaniego.github.io/

Modules

- Files containing Python code, such as functions, classes, or variables.
- Used to organized and reuse code
- Each Python file is considered a module
- Modules contain applied methods
- *Example:* `math` module and `sqrt`() method.

Python Standard Library

- Collection of pre-written modules that come with Python, offering a wide range of functionalities (e.g., file I/O, string manipulation…)
- `import` statement
- Examples:
  - `os`: interaction with the OS.
  - `sys`: access to system-specific parameters and functions (e.g., command-line arguments).
  - `shutil`: high-level file ops like copying, moving…files
  - `pathlib`: object-oriented file path handling
  - `math`: mathematical functions (sqrt(), sin()...)
  - `datetime`: dates and times.
  - `random`: generation of random numbers or choices.

Exercises with modules

ACTIVITY

- Complete an script that:
  - Lists files on a directory using `os` or `pathlib`.
  - Create a text file in that directory and writes information about the system (`sys`).
  - Moves the file to another folder using `shutil`.

Creation of custom module

ACTIVITY

- Complete a module with 2 functions:
  - `list_files_in_directory(path)`
  - `backup_file(file_path, backup_dir)`
- Use this module in another script.

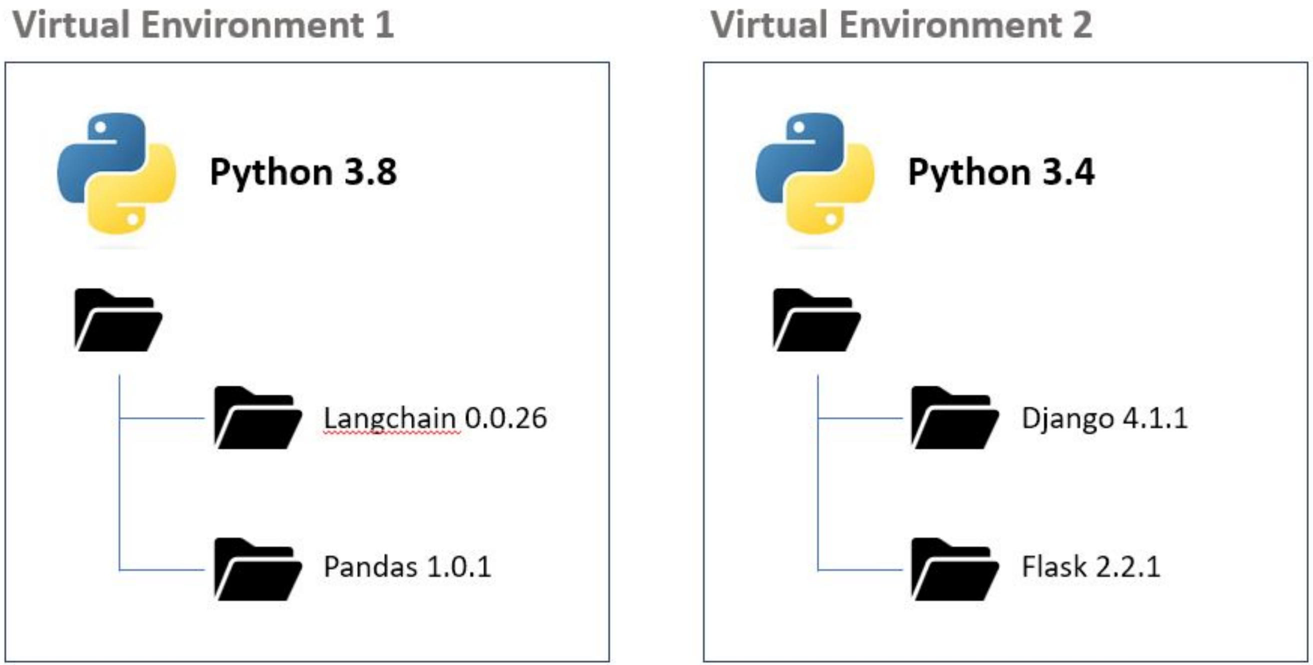
ACTIVITY

- Choose a module from the standard library that hasn’t been used yet (such as `random`, `math`, `datetime`, etc.).
- Research what the module does and create a small program that uses at least one function from that module.

Virtual environments in Python

- Why using virtual environments?
  - Manage project dependencies without conflicts.
- Key points:
  - Isolation: each project can have its own dependencies.
  - Avoid dependency conflicts: different project may require different versions of the same package.
  - Easier collaboration: ensures everyone working on the project is using the same dependencies.
- **Discussion question:** in which scenarios might dependency conflicts arise, and how can virtual environments help resolve them?

Virtual environments in Python



Virtual Environment 1

Python 3.8

Langchain 0.0.26

Pandas 1.0.1

Virtual Environment 2

Python 3.4

Django 4.1.1

Flask 2.2.1

Virtual environments in Python

- Creation and management of virtual environments using `virtualenv.`

```
sudo apt update
sudo apt install python3 python3-pip
python3 -V
pip -V

cd /path/to/your/project
virtualenv venv
python3 -m venv venv
```

- **Activity:** create a virtual environment for the project

Virtual environments in Python

- Activation and deactivation of virtual environments.

```
source venv/bin/activate
deactivate
```

- **Activity:** activate/deactivate the virtualenv and identify when you’re in a virtual environment

Virtual environments in Python

- Installation of packages and dependencies using virtual environments.

```
pip install package_name # pandas
```

- Requirements file (`requirements.txt`)

```
pip freeze > requirements.txt
pip install -r requirements.txt
pip list
pip show package_name
pip uninstall package_name
```

- **Activity:** install a package and create a requirements.txt file so it can be reused.

Virtual environments in Python

- Best practices using virtual environments:
  - Always use a virtual environment for new projects.
  - Keep your `requirements.txt` updated.
  - Avoid installing packages globally unless necessary.
  - Regularly review and clean up unused virtual environments.
- Documentation: encourage documenting dependencies and the virtual environment setup in project `README` files.
- **Activity:** Discuss case studies or examples of projects that suffered from dependency issues and how they could have benefited from virtual environments.

ACTIVITY

- Install the `requests` package in a virtual environment, create a custom module that includes a method using this package, and call that method from a main script.