Environment & Workflow Lab

Python Course · Module 1.5

ElnurBDa

Module 1.5 — Environment & Workflow Lab

This short module is a **hands-on lab** between Module 1 and Module 2.

Goals:

- Set up a **consistent workspace** for the rest of the course
- Practice running Python files from the **terminal**
- Create and activate a **virtual environment (venv)**
- Understand where your scripts and data files live on disk

After this lab, Module 2 (Python basics) will feel much more natural.

Course Folder Layout

Pick a main folder where all course projects will live, for example:

```
/home/elnur/Desktop/idschool/python-course/
├── markdown_files/   # slides
└── projects/         # your code lives here
```

Inside `projects/` you will create one folder per module or mini-project:

```
projects/
├── module1_hello/
├── module2_basics/
├── module3_data/
└── ...
```

Try to keep code, data files, and virtual environments **inside** these project folders.

Terminal Basics Refresher

Open a terminal (Linux/macOS Terminal, PowerShell, etc.) and practice:

```
pwd            # print current directory
ls             # list files (dir on Windows)
cd path/to/dir # change directory
```

Navigation checklist:

- Can you navigate from your home folder to `idschool/python-course/`?
- Can you list the contents of `markdown_files/` and `projects/`?

You should **always know where you are** before running `python3 file.py`.

Creating Your First Project Folder

Let’s create module1_hello:

```
cd /home/elnur/Desktop/idschool/python-course
mkdir -p projects/module1_hello
cd projects/module1_hello
```

Create hello.py in this folder with the following content:

```
print("Hello from Module 1.5!")
```

Run it:

```
python3 hello.py
```

If you see the message, your basic workflow is working.

Virtual Environments — Why Now?

Later modules will install packages like `requests`, Flask, etc.

Without virtual environments:

- Different projects may **fight over versions**
- You may not remember which scripts require which packages

With virtual environments:

- Each project can have its **own dependencies**
- You can safely experiment without breaking other projects

We start using this pattern now so it feels natural later.

Creating and Activating a venv

Inside projects/module1_hello/:

```
python3 -m venv .venv
```

Activate it:

```
source .venv/bin/activate   # Linux/macOS
# or on Windows (PowerShell):
# .venv\Scripts\Activate.ps1
```

Your prompt should now show something like:

```
(.venv) user@machine:~/.../module1_hello$
```

To deactivate later:

```
deactivate
```

Installing a Test Package

While still inside the (.venv):

```
pip install --upgrade pip
pip install requests
pip freeze
```

You should see requests and its dependencies listed.

Optional: save them to requirements.txt:

```
pip freeze > requirements.txt
```

Now this project knows exactly **which versions** it uses.

Using pathlib for Safer Paths

To avoid hard-coded string paths, use pathlib:

```python
from pathlib import Path

BASE_DIR = Path(__file__).parent  # folder containing this file
data_dir = BASE_DIR / "data"

print("Base dir:", BASE_DIR)
print("Data dir:", data_dir)
```

[finished]

```
Base dir: /tmp/nix-shell-61594-0/.presentermXx6S8w
Data dir: /tmp/nix-shell-61594-0/.presentermXx6S8w/data
```

Pattern:

- `Path(__file__).parent` gives you the current project folder
- `BASE_DIR / "subfolder" / "file.txt"` builds paths safely

We will reuse this idea when working with files, logs, and web servers.