What You'll Learn

- **Installation**
  - Install Python 3.13.* correctly on your machine
- **Virtual Environments**
  - Set up and manage virtual environments to isolate projects and avoid package conflicts.
  - Capture and reproduce project environments across machines.
- **Jupyter**
  - Install and use JupyterLab to handle notebooks.
  - Setup Visual Studio Code with Python and Jupyter.
- **Headache Mitigation**
  - Learn about the “Golden Rules” to minimize setup problems.
  - Troubleshoot common installation pitfalls, such as PATH issues, or kernel mismatches.

Chapter 1. Installing Python & Jupyter: A Friction-Free Start

Before you write your first line of code, make sure the ground you stand on is solid.

Every semester, students new to programming waste their early enthusiasm wrestling with broken installs, mismatched package versions, wrong path, etc. Those hours would be better spent learning about Python. While this has been a reality for a long time, in 2025 we no longer have to accept that pain as inevitable. The Python ecosystem now has tools to make writing python code easier and friendlier. Pair it with **JupyterLab**, and the use of **virtual environments**, both topics will be in depth later, and you have a platform that is both beginner-friendly and production-grade.

> **Takeaway** Your very first deliverable in this course is a clean installation that you can *trust*.

The three “Golden Rules” below act like safety rails: they keep your projects cosnisten. Follow them and you’ll avoid 90 % of the headaches beginners usually face. As a cherry on top, it will make your code-base tidy and learning curve smoother.

1.2 The Golden Rules

Rule	Why it matters	What can go wrong if you skip it
Use Python 3.13.*	Everyone in the class will have the *same* features and the newest friendly touches.	Tiny differences in syntax or missing features from utilizing a different version of Python make example code fail for reasons that are hard to spot.
Create a *virtual environment* for each project	Keeps every project’s “ingredients” (packages) in its *own* pantry, so projects don’t mess with each other. Re-creating your setup on a new computer becomes a single-command job.	A virtual environment is like a zip-up tool belt that only your current project can see. It carries its own copy of Python and any add-on libraries you install. When you switch projects, you simply swap belts—no loose tools lying around the shop. Two projects may need different versions of the same package; installing one can silently break the other, leaving beginners stuck.
Choose JupyterLab instead of classic Notebook	JupyterLab gives you tabs (click tab to see possible completions for code you start tyoing), a built-in file browser, a simple debugger, and even a peek at your variables—much closer to what pros use every day.	If you start in classic Notebook and later need these features, migrating notebooks and habits is a hassle.

With these rules in place, you’ll spend your time learning *Python*—not fixing Python.

1.3 Installing Python on Your Platform

macOS (Intel & Apple Silicon)

The power-user way (Homebrew)

Homebrew is a **command-line "app store" for macOS**. It handles downloading, installing, and updating software so you don't have to hunt for installers yourself.

The three commands below do the following, in order:

1. **Install Homebrew itself** – this one-liner fetches and runs Homebrew's official installer.
2. **Install Python 3.13.*** via Homebrew, keeping it isolated from the macOS system Python.
3. **Add Homebrew's install location to your PATH** (Apple-Silicon Macs only) so the terminal can find the new `python3`.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install python
echo 'export PATH="/opt/homebrew/bin:$PATH"' >> ~/.zprofile   # Apple Silicon
```

Always call `python3` and `pip3` to avoid clobbering the system Python that ships with macOS.

Windows 10 / 11

The straightforward way (official installer)

Windows needs a few check-boxes ticked so Python works from any folder and plays nicely with other apps. Here's the smooth path, with a bit more detail on *why* each step matters:

1. **Download the installer** Visit **python.org → Downloads → Windows** and grab the 64-bit **"Windows installer (64-bit)"** for **Python 3.13.2** (the file ends in `.exe`). *Why?* This is the **official** build—secure, up-to-date, and fully compatible with every library you'll meet in this course.
2. **Launch the installer and tick two critical options**
   - First screen: check **"Add Python 3.13 to PATH."** *What it does:* Adds the folder that contains `python.exe` to your **PATH**, a list of places Windows looks for programs. Without this, you'd have to type the full folder path every time you run Python.
   - Same screen: expand **"Customize installation"** (tiny link below the *Install Now* button), then tick **"Use admin privileges when installing `py.exe`."** *What it does:* Installs the *py launcher* system-wide so other tools (IDEs, editors) can always find the correct Python version.
3. **Click *Install*** and let the wizard finish. When it's done, leave "Disable path length limit" *unchecked* unless you know you need it.
4. **Verify the install** Open **Start → Windows Terminal** (or Command Prompt) and run:

   ```
   > python --version
   Python 3.13.*
   ```

   Seeing the version number means Python is ready.

What if Windows can't find Python?

If the terminal replies *"'python' is not recognized…"*:

- **Quick fix:** Rerun the installer and make sure the **PATH** checkbox is selected.
- **Manual fix:** Add these two folders to your *user* PATH:

  ```
  C:\Users\<YourName>\AppData\Local\Programs\Python\Python313\
  C:\Users\<YourName>\AppData\Local\Programs\Python\Python313\Scripts\
  ```

  Then restart the terminal and try `python --version` again.

> **Heads-up – Microsoft Store edition** Windows also offers a "Python" app in the Microsoft Store. It auto-updates and lives in a sandbox, which is handy for quick tests—but it blocks certain C/C++ extensions you'll need by **Week 6** (e.g., `numpy`, `pandas`). For this course, stick with the **python.org installer** above.

Linux

Linux OSes usually ship with a default version of Python installed, but we want Python 3.13.

Use the following three commands to install on apt-based operating systems.

```
sudo add-apt-repository ppa:deadsnakes/ppa
sudo apt update
sudo apt install python3.13
```

1.4 Bootstrapping Your First Virtual Environment

A **virtual environment** is an *isolated* folder that carries its own copy of Python plus any add-on packages you install. Think of it as a self-contained "sand box" where you can experiment freely without spilling paint on your operating system's built-in Python or your other projects.

The four commands below set up that sandbox:

```
python3 -m venv sandbox           # 1 Create a new virtual environment called "sandbox"
source sandbox/bin/activate       # 2 Turn it on (Windows: .\sandbox\Scripts\activate)
pip install --upgrade pip         # 3 Upgrade pip inside the sandbox
```

Step	What it does	Why it matters
1. `python3 -m venv sandbox`	Copies the Python interpreter and standard library into a new folder named **sandbox**.	Keeps project-specific packages separate from the rest of your computer.
2. `activate`	Switches your terminal to *use* that copy of Python. Your prompt changes to **(sandbox)** so you always know where you are.	Prevents "I thought I installed that package, but Python can't find it" moments.
3. `pip install --upgrade pip`	Updates *pip* (Python's package installer) **inside** the virtual environment.	Newer pip versions handle dependencies better and give clearer error messages.

Capturing (and recreating) your sandbox

Once you've added libraries— `pandas` , `matplotlib` , anything—you can *snapshot* the exact versions with:

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

This writes a plain-text file (.txt) listing every package and its version. Feel free to open requirements.txt to see it's content.

The file requirements.txt allows you (or a teammate, or your future self on a new laptop) can rebuild the environment with a single command:

```
python3 -m venv sandbox         # we are creating a fresh environment
source sandbox/bin/activate     # we get into the the sandbox environment.
pip install -r requirements.txt # install (downloads first) the packages defined in the
requirements.txt
```

Deactivate the environment simply with a single command from inside the environment:

```
deactivate
```

Now you have a reproducible workspace—one of the most powerful habits you can form as a new programmer.

1.5 Installing JupyterLab

With your environment activated:

```
pip install jupyterlab
jupyter lab                       # browser opens at http://localhost:8888
```

JupyterLab's sidebar gives you a file browser, terminals, and notebooks in separate tabs. Under the hood, it runs the same **ipykernel** you just installed, so your environment's packages are already available.

> **Tip – Multiple Projects** Register each venv as its own kernel: `python -m ipykernel install --user --name=ds101 --display-name "DS-101"` Now you can switch kernels from the JupyterLab toolbar.

1.6 Why Bring Visual Studio Code into the Mix?

Although we'll begin the course in **JupyterLab**—the de-facto standard for notebooks, modern data-science and AI tooling is quickly converging on a richer, *IDE-style* experience. Editors such as **Windsurf**, **Cursor**, and **GitHub Codespaces** all build on, or embed, **Visual Studio Code (VS Code)** under the hood. Learning VS Code now gives you a head-start with these next-generation tools and prepares you for industry workflows where code, notebooks, and version control live side by side.

VS Code acts as a *control center*: you can open a Jupyter notebook, tweak a Python script, stage a Git commit, and even chat with Copilot—all in one window. That single-pane workflow keeps beginners from bouncing between half a dozen apps and losing context.

Setting It Up

1. **Install VS Code.**
2. In VS Code's Extensions view (`Ctrl Shift X`), add both **Python** and **Jupyter**—each maintained by Microsoft.
3. Hit `Ctrl Shift P` , run **"Python: Select Interpreter,"** and pick the virtual environment you created earlier.
4. Open any `.ipynb` notebook; VS Code auto-detects your environment and wires up the correct kernel.

You now have:

- **IntelliSense** smart autocompletion (no more guessing method names).
- A **Variable Explorer** that shows data frames and arrays live.
- One-click **Git diff** and version-control tools.
- **Copilot or Cursor-style AI suggestions** right inside notebook cells.

1.7 Common Pitfalls & Quick Fixes

Symptom	Likely Cause	Fix
`python` opens the Microsoft Store	Store install hijacked the `python` alias	Uninstall the Store version; reinstall from python.org
`ModuleNotFoundError` in Jupyter but not in terminal	Notebook is running a *different* kernel	From the notebook toolbar choose **Kernel → Select Kernel →** pick your venv
`activate` script blocked in PowerShell	Execution policy too strict	`Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser`

With VS Code in your toolkit you can start in JupyterLab for quick experiments, then graduate to a full IDE seamlessly without any extra steps.

1.8 What's Next

You now have:

- **Python 3.13.*** up and running
- A repeatable **virtual-environment workflow**
- **JupyterLab** launching in your browser
- We have **VS Code** running

Key Takeaways

1. **Installation is a skill.** Master it early and you'll debug code, not tooling.
2. **Isolation prevents heartbreak.** One virtual env per project—no exceptions.
3. **JupyterLab is the modern default.** Classic Notebook is legacy.
4. **Verify everything.** `python --version` , `which python` , and `pip list` are your friends.
5. **Automate the boring parts.** Scripts that recreate environments will save you when you switch machines—or when your future self forgets today's clever hacks.