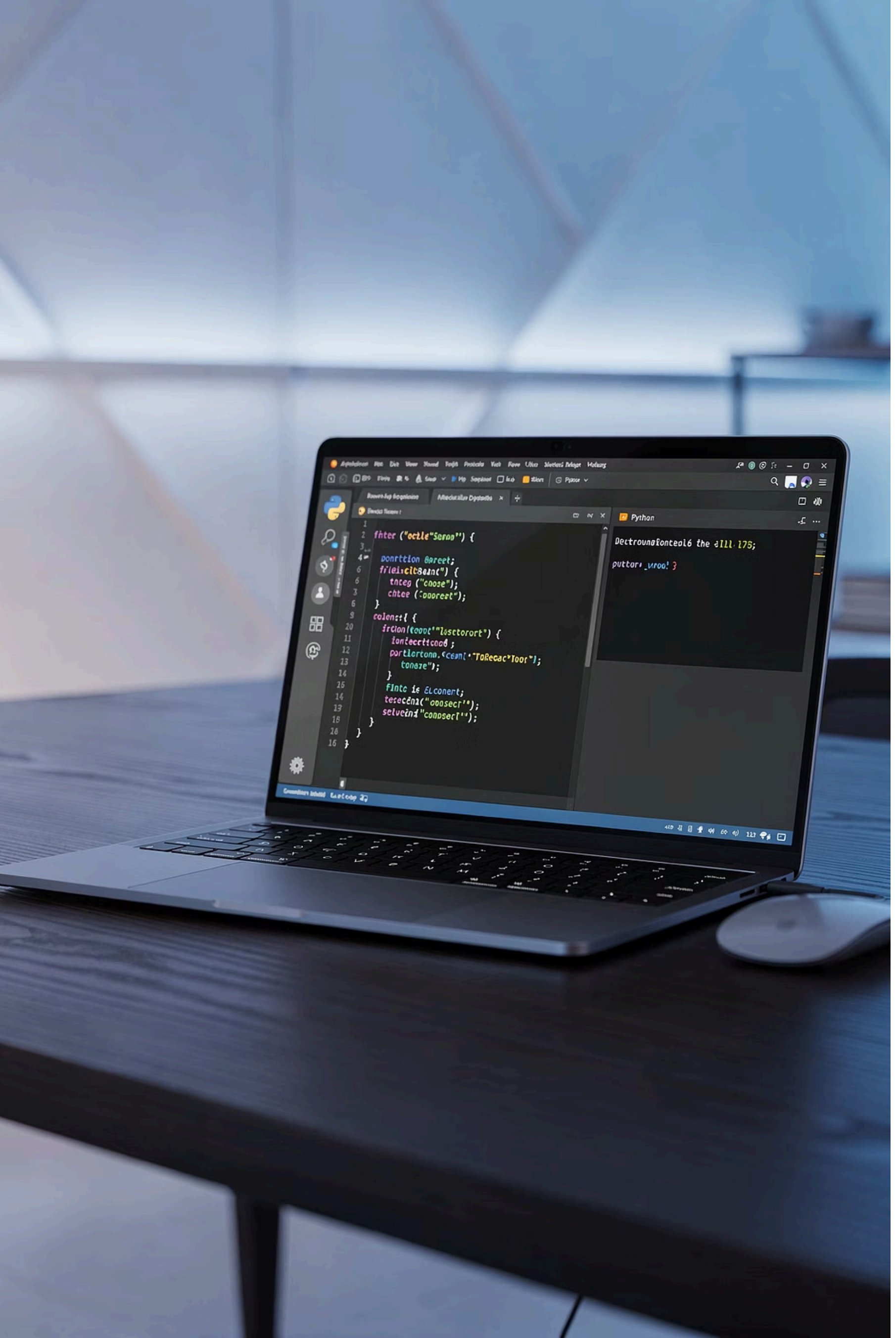

Session 1: Professional Python Setup (venv) + Clean Project Structure

Learn how to build professional Python projects from the ground up with virtual environments, clean structure, and maintainable code practices.

LEARNING OBJECTIVES

What You'll Master Today

By the end of this session, you'll have the fundamental skills needed to set up professional Python projects. These aren't just theoretical concepts—they're the same practices used by development teams at leading technology companies worldwide.

01

Virtual Environments

Create and activate isolated Python environments using venv to keep your projects separate and conflict-free.

02

Dependency Management

Install packages using pip and generate requirements.txt files for reproducible installations across any machine.

03

Project Structure

Follow industry-standard folder organisation with src/, data/, and out/ directories for maintainable codebases.

04

Code Organisation

Separate input/output operations from business logic using well-designed functions with clear responsibilities.

05

Command Line Execution

Run Python projects professionally from the terminal with proper argument handling and execution patterns.

Why Virtual Environments Matter

A virtual environment (venv) creates an isolated Python workspace specifically for one project. Think of it as a self-contained bubble where your project lives independently from every other Python project on your system.

This isolation is crucial for professional development. Without virtual environments, installing packages for one project can break another project that depends on different versions of the same library. It's one of the most common sources of frustration for developers—and one of the easiest problems to prevent.

Prevents Dependency Conflicts

Different projects can use different versions of the same package without interfering with each other.

Ensures Reproducibility

Your project runs identically on any machine by recreating the exact same environment from requirements.txt.

Eliminates Environmental Issues

Say goodbye to "works on my machine" problems—everyone works with identical dependencies.

MACOS / LINUX

Creating Your First Virtual Environment

Setting up a professional Python project on macOS or Linux follows a consistent pattern. Open your terminal and execute these commands in sequence to create a properly isolated development environment.

1

Create Project Directory

```
mkdir session1 && cd session1
```

Creates a new folder called 'session1' and navigates into it in one command.

2

Initialise Virtual Environment

```
python3 -m venv venv
```

Creates a new virtual environment in a folder called 'venv' using Python's built-in venv module.

3

Activate the Environment

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

Activates your virtual environment. Your terminal prompt will change to show (venv) at the beginning.

4

Upgrade pip

```
python -m pip install --upgrade pip
```

Updates pip to the latest version to ensure compatibility with modern packages.

5

Install Initial Packages

```
pip install numpy pandas
```

Installs NumPy and Pandas, two fundamental packages for data analysis work.

6

Freeze Requirements

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

Captures all installed packages and their versions into a requirements file for reproducibility.

Verification Commands

Run these checks to confirm everything is set up correctly:

- **which python** — Should point to your venv folder, not the system Python
- **python --version** — Displays your Python version
- **pip list** — Shows all installed packages in this environment

WINDOWS POWERSHELL

Windows Setup Process

Windows PowerShell requires slightly different syntax for creating and activating virtual environments. The commands follow the same logical flow, but use Windows-specific paths and the 'py' launcher instead of 'python3'.

1

Create and Navigate

```
mkdir session1; cd session1
```

PowerShell uses semicolons to chain commands on a single line.

2

Create Virtual Environment

```
py -m venv venv
```

The 'py' launcher automatically finds your Python installation.

3

Activate Environment

```
.\venv\Scripts\Activate.ps1
```

Windows uses backslashes for paths and a .ps1 script for activation.

4

Upgrade and Install

```
py -m pip install --upgrade pip
pip install numpy pandas
```

After activation, pip commands work identically across all platforms.

5

Save Dependencies

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

Creates your requirements file for project reproducibility.

Windows Verification

Confirm your setup with these PowerShell commands:

- **where python** — Locates your Python executable (should be in venv)
- **py --version** — Shows your Python version
- **pip list** — Lists all packages in the current environment

The Power of requirements.txt

The requirements.txt file is your project's dependency manifest—a simple text file that lists every package your project needs to run. This single file enables any developer to recreate your exact environment in seconds.

Think of it as a recipe. Just as a recipe lists every ingredient needed to bake a cake, requirements.txt lists every package needed to run your code. Share the file, and anyone can reproduce your environment perfectly.

Creating the File

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

Captures a snapshot of all currently installed packages and their exact versions into a text file.

Installing from File

```
pip install -r requirements.txt
```

Reads the requirements file and installs every package listed, recreating the environment identically.

This two-command workflow is fundamental to professional Python development. When you clone a colleague's project or deploy to a server, running `pip install -r requirements.txt` gives you the exact same environment they used during development. No guesswork, no missing packages, no version conflicts.

Professional Project Structure

A well-organised project structure is like a well-organised kitchen—everything has its place, and you can find what you need instantly. This layout is used across the industry because it scales beautifully from small scripts to large applications.

```
session1/
  main.py
  requirements.txt
  README.md
  src/
    io_utils.py
    stats_utils.py
  data/
  out/
```

- **main.py**

  The entry point that orchestrates your programme's execution

- **requirements.txt**

  Lists all package dependencies for reproducibility

- **README.md**

  Documents what your project does and how to use it

- **src/**

  Contains all your source code modules and utility functions

- **data/**

  Stores input files like CSV datasets or configuration files

- **out/**

  Holds generated outputs, reports, and exported results

This structure separates concerns beautifully. Your code lives in src/, your inputs in data/, and your outputs in out/. The root level contains only coordination files. Anyone familiar with Python projects will understand your structure immediately.

Separation of Responsibilities

Professional code divides work into specialised modules, each handling one aspect of the system. This separation makes your code easier to test, debug, and extend. When a bug appears, you know exactly where to look.

io_utils.py

Purpose: Handles all input and output operations

- Reading CSV files and parsing their contents
- Validating data formats and catching errors
- Converting raw strings into appropriate Python types
- Exporting results to files or databases

Keep I/O code isolated so business logic never deals with file formats directly.

stats_utils.py

Purpose: Contains pure computational logic

- Statistical calculations and data transformations
- Business rules and domain-specific logic
- Algorithms that process structured data

These functions receive clean data structures and return results—they never touch files.

main.py

Purpose: Orchestrates the entire workflow

- Calls functions from both utility modules
- Coordinates the flow from input to output
- Prints user-facing messages and results
- Handles command-line arguments and configuration

Think of main.py as the conductor of an orchestra—it doesn't play instruments, it coordinates.

Key Principle: Keep computations independent from file reading. A function that calculates statistics should receive a list or dictionary, not a filename. This makes testing trivial and reuse effortless.

Writing Clean Functions

Clean functions are the building blocks of maintainable code. They do one thing well, have clear names, and make your intentions obvious to any reader—including yourself six months from now.

Follow these minimal but powerful rules when writing functions:

One Clear Responsibility

Each function should do exactly one thing. If you need "and" to describe it, split it into multiple functions.

Convert Types at Boundaries

Handle type conversions and validation in I/O functions. Keep your business logic functions pure and simple.

Keep Functions Short

If a function doesn't fit on your screen, it's probably doing too much. Break it into smaller, well-named pieces.

Example: A Well-Designed Function

```
def total_revenue(rows: list[dict]) -> float:
    """Calculate total revenue from transaction rows.

    Args:
        rows: List of transaction dictionaries with 'quantity' and 'unit_price' keys

    Returns:
        Total revenue as a float
    """
    return sum(r["quantity"] * r["unit_price"] for r in rows)
```

This function has a single, clear purpose. It expects clean data (converted types), performs one calculation, and returns a simple result. The type hints and docstring make its behaviour crystal clear.

Handling Real-World Data Errors

Real-world CSV files are rarely clean. You'll encounter issues that would crash your programme if left unhandled. Professional developers anticipate these problems and build robust error handling into their data pipelines.

Missing Values

Empty cells or fields that should contain data but don't. Decide whether to skip these rows, use default values, or raise an error.

Extra Whitespace

Leading or trailing spaces in strings that break comparisons. Always .strip() string values after reading.

Non-Numeric Values

Text like "N/A" or "-" appearing in columns that should contain numbers. These cause conversion errors if not caught.

Wrong Decimal Separators

European CSVs often use commas (1,50) instead of periods (1.50) for decimals, breaking float conversions.

Error Handling Strategy

Rule of thumb: Decide early whether to fail loudly or skip invalid rows gracefully. For critical data, raise clear error messages that identify the problem. For optional data or large datasets, log warnings and skip bad rows, then report the skip count at the end. Never fail silently—always let the user know something went wrong.

Running Your Programme from the Command Line

Professional Python developers run their code from the terminal, not from an IDE's run button. This approach makes your code more portable, automatable, and production-ready. It's a habit worth forming early.

Basic Execution Pattern

Always run your script from the project root directory, not from within subdirectories. This ensures relative paths work correctly.

```
python main.py
```

When you run this command with your virtual environment activated, Python uses the packages installed in your venv, not your system Python. You can verify this by checking that your terminal prompt shows `(venv)`.

Adding Command-Line Arguments

As your projects grow, you'll want to pass parameters from the command line rather than hardcoding them. Here's a common pattern:

```
python main.py --input data/sales.csv
python main.py --input data/sales.csv --output out/report.txt
```

This flexibility lets you reuse the same code with different input files or configuration options. Your main.py can parse these arguments using the argparse module.

Why CLI Matters

- Works identically on any operating system
- Can be automated with scripts or schedulers
- Integrates into larger workflows easily
- Matches how production systems run

Best Practices

- Always activate your venv first
- Run from the project root
- Use relative paths for file access
- Print clear status messages

Session Summary

You've now learned the foundational practices that separate hobbyist scripts from professional Python projects. These skills form the basis for everything you'll build going forward.

Virtual Environment Mastery

You can create isolated Python environments using venv, activate them correctly on your platform, and understand why this isolation prevents dependency conflicts across projects.

Dependency Management

You know how to install packages with pip, freeze your dependencies into requirements.txt, and reproduce environments on any machine with a single command.

Clean Project Organisation

You can structure projects with separate directories for source code (src/), data inputs (data/), and generated outputs (out/), following industry-standard conventions.

Function Design

You understand how to separate I/O operations from business logic, write functions with single clear responsibilities, and keep computations independent from file handling.

Professional Execution

You can run Python programmes from the command line, pass arguments when needed, and work in a way that matches production workflows.

Next steps: Apply these patterns to your own projects immediately. The best way to internalise these practices is to use them. Start small, stay consistent, and you'll quickly find that professional project setup becomes second nature.