Python Setup Workshop

Jeremy Mikecz
Research Faciliation, Dartmouth Libraries

2026-01-13

Why Program? Why Program in a World of AI?

- understand code to verify AI outputs
- debug when AI gives you broken code
- need practice with computational thinking to break problems down
- control your data and methods (unlike black-box AI tools)
- preserve AGENCY over your research workflow

Think of AI as an assistant, but YOU are still the researcher who needs to understand and validate the work

Why Python?

Why Python?: It's popular

- Most popular language in research
- Massive ecosystem for research: pandas, numpy, scipy, matplotlib, scikit-learn, etc.

Why Python? It's beginner-friendly

- highly readable language (good for beginners)

Python Example

```
numbers = [1, 2, 3, 4, 5]
average = sum(numbers) / len(numbers)
print(f"Average: {average}")
```

```
Average: 3.0
```

R

```r
numbers <- c(1, 2, 3, 4, 5)
average <- mean(numbers)
print(paste("Average:", average))
```

Java

```java
public class AverageCalculator {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};
        double sum = 0;

        for (int number : numbers) {
            sum += number;
        }

        double average = sum / numbers.length;
        System.out.println("Average: " + average);
    }
}
```

C++

```cpp
#include <iostream>
#include <vector>
#include <numeric>

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    double sum = std::accumulate(numbers.begin(), numbers.end(), 0);
    double average = sum / numbers.size();

    std::cout << "Average: " << average << std::endl;
    return 0;
}
```

Javascript

```javascript
const numbers = [1, 2, 3, 4, 5];
const average = numbers.reduce((sum, num) => sum + num, 0) / numbers.length;
console.log(`Average: ${average}`);
```

Code Length Comparison

Language	Lines of Code	Complexity
Python	3	⭐ Simple

Language	Lines of Code	Complexity
R	3	⭐ Simple
JavaScript	3	⭐ Simple
Ruby	3	⭐ Simple
Julia	3	⭐ Simple
PHP	5	⭐⭐ Moderate
Perl	5	⭐⭐ Moderate
Rust	6	⭐⭐⭐ Complex
Scala	7	⭐⭐⭐ Complex
Fortran	8	⭐⭐⭐ Complex
C++	11	⭐⭐⭐⭐ Very Complex
Java	12	⭐⭐⭐⭐ Very Complex
Go	13	⭐⭐⭐ Complex
C	14	⭐⭐⭐⭐ Very Complex

Why Python? Other Reasons

- Strong community support
- Free and open source
- Cross-platform
- Used across disciplines: from digital humanities to computational biology

Problem: Python setup is difficult

Typical Python Setup

1. Install Python
2. install an IDE (Integrated Development Environment): software that combines tools for writing, editing, compiling, and debugging code.
3. Install Python packages to "global" environment on your computer

Resulting Problems

1. IDE can't find Python ('Python is not recognized as a command' errors?)
2. Code that worked on one computer but broke on another? (dependency conflicts)
3. updating one project causes another not to work
4. installing the wrong package causes your Python installation to break
5. Multiple Python versions fighting each other?
6. your project code works differently on different computers (making collaboration difficult)

Common error messages:

- `'python' is not recognized as an internal or external command`
- `ModuleNotFoundError: No module named 'pandas'`
- `Python version mismatch`
- `Permission denied`

This Lesson

Contents

1. Introductory presentation
2. Install: **Visual Studio Code** + **uv** + **Python** (notebook 01)
3. Create Python Project from Scratch within VSC (notebook 02)
    1. set up project folder
    2. write your first markdown document
    3. write your first script
    4. write your first notebook
4. Download and Import Existing Project (notebook 03)

Python Setup

Recommended Python Set Up Approach: UV + VSC

1. **uv**: A new, fast tool that manages everything—Python versions,virtual environments, and packages—all in one place.
2. **Visual Studio Code**: A modern editor that works seamlessly with uv and Python.
3. Project best practices: folder/file organization; file-naming; virtual environment for each project; rigorous documentation
    1. We're building good habits from day one, not adding them later.

Hands-on Setup

Go to 01_setup-instructions (qmd, pdf, or html)

Slides below offer "best practices" recommendations for setting up a Python project on your computer. For workshop participants, we will introduce these as we go. For asynchronous learners, you may want to review these on your own before continuing to notebook 01.

Python / Programming Best Practices

Project Organization

One folder – one project

Project Organization

Folder Organization:

```
my-research-project/
├── data/
│   ├── raw/              # Original, untouched data
│   └── processed/        # Cleaned, transformed data
├── code                  # this may include Python scripts (.py) and coding
notebooks
├── results/
│   ├── figures/          # Plots and visualizations
│   └── tables/           # Output tables/statistics
├── docs/                 # Documentation, notes
├── .gitignore           # Tell git what NOT to track
├── README.md            # Project description and instructions
└── pyproject.toml       # Python dependencies (with uv)
```

Project Organization

Why this structure?

```
+ Separation of concerns: Raw data stays pristine, processed data is separate

+ Findability: Six months from now, you'll know where everything is
+ Collaboration: Others can navigate your project easily

+ Scalability: Works for small and large projects
```

Key principles:

```
+ Keep raw data in data/raw/ and NEVER modify it directly
+ All data processing should be scripted and reproducible
+ Results should be generated by code, not manually created
+ One project = one self-contained folder"
```

File Naming Conventions

Some common conventions:

1. **Lowercase with underscores**: `my-script.py` or `my_script.py`
2. **Be descriptive**: Names should tell you what the file contains or does
3. **NO SPACES**: For file names, you should replace whitespace with hyphens or underscores instead. Spaces cause issues in terminals. **For Python code and folders containing Python code, it is recommended to avoid hyphens too as Python has difficulty importing code from folders or files with hyphens or any special characters besides underscores. Using hyphens for data files is perfectly fine.**
4. **Avoid special characters** with exception of underscore _ whenever possible.
5. **Use number prefixes for sequences**: `01_`, `02_`, `03_` keeps scripts in order

6. **Include dates if relevant**: `YYYY_MM_DD` or `YYYYMMDD` to format sorts chronologically (you may also use hyphens, i.e. `YYYY-MM-DD` for data files and other files or folders that DO NOT contain Python code).
7. **Consistent within a project**: Pick a style and stick with it

For general file management naming conventions see Harvard's guide to file naming.

For Python specific naming conventions (for files and folders, but also for the names of variables, functions, classes, etc. see The Pep 8 Style Guide.

Documentation

Most important document: README.md file. This is where users go to first to learn more about a project.

What goes in a README:

1. project title & overview
2. how to set up / get started
3. how to use
4. data description
5. description of other content
6. how to cite
7. license information

Dependency Conflicts

Problem

- Project A (from 2022) needs pandas version 1.3
- Project B (new project) needs pandas version 2.0

If you just install your Python dependencies to a global environment (meaning they can be accessed anywhere on your personal computer) then your Python setup can only work for one of these packages. For example, if you try to run Project A, but you have pandas 2.0 installed on your computer, then you may receive a `dependency conflict` error.

Solution: Virtual Environments

A **virtual environment** is like a **separate, isolated workspace** for each Python project on your computer.

A virtual environment keeps each project's packages separate and organized, preventing conflicts and making your code easy to share

Think of it as:

- A **project-specific toolbox** that contains only the tools (packages) you need for that project
- A **separate room** for each project, so things don't get mixed up
- A **backpack** you pack differently for each trip/project

Value of Virtual Environments

1. avoid conflicts
2. keep things clean
3. easy sharing
4. protect your system

Visual

```
WITHOUT Virtual Environments:
┌──────────────────────────────┐
│   YOUR COMPUTER              │
│                              │
│  All Projects → One Big      │
│  Messy Pile of Packages 😵   │
│                              │
│  (Conflicts! Confusion!)     │
└──────────────────────────────┘


WITH Virtual Environments:
┌──────────────────────────────┐
│   YOUR COMPUTER              │
│                              │
│  Project A → 📦 Its packages │
│  Project B → 📦 Its packages │
│  Project C → 📦 Its packages │
│                              │
│  (Clean! Organized! 😊)      │
└──────────────────────────────┘
```

Version Control (Git)

IDEs (Integrated Development Environments)

An IDE is your coding workspace. Some examples (Visual Studio Code, Pycharm, Spyder, etc.). Today we are using Visual Studio Code (VSC) because":

- Free and open source
- Excellent Python support
- Built-in git integration
- Integrated terminal
- IntelliSense (code completion)
- Debugging tools
- Works with both .py scripts and Jupyter notebooks
- Huge extension ecosystem

- Works on Windows, Mac, and Linux