Transitioning from Colab to Local Python Development

Lab · CS 203: Software Tools and Techniques for AI

Introduction to Python

IIT Gandhinagar

Part 1: The Motivation

Why move beyond Google Colab?

Why Local Python?

> **Scenario**: You want to build a real-time web app, a game, or a robot controller.
>
> **The Problem**: Google Colab runs in the cloud (browser). It can't easily access your local camera, your robot's motors, or run indefinitely as a web server.

The Solution: Local Python Environment.

Benefits:

- **Persistence**: Files stay on your computer.
- **Performance**: Use your own CPU/GPU (no usage limits).
- **Integration**: Access system hardware (files, camera, sensors).
- **Offline Access**: Code without internet.

Today's Goals

By the end of this lab, you will know how to:

1. Install Python correctly (adding it to PATH).
2. Use the Command Line Interface (CLI) to navigate folders.
3. Run Python scripts locally.
4. Understand how `import` works across files.
5. Use the mysterious `if __name__ == "__main__":` block.
6. Create Virtual Environments to manage libraries.
7. Install packages using `pip`.

Part 2: Installation

Step 1: Install Python

If you haven't already, you need Python installed on your system.

Video Tutorial (Windows & Mac):

> Click to Watch: Python Tutorial for Beginners (Corey Schafer)

IMPORTANT Step for Windows

When installing on Windows:

You **MUST** check the box that says: **"Add Python to PATH"**

Why?

If you don't do this, typing `python` in your terminal will do nothing (or open the Windows Store).

Part 3: The Command Line Interface (CLI)

Talking to your computer with text

CLI Survival Guide

Open your terminal:

- **Windows**: `cmd` (Command Prompt) or PowerShell
- **Mac/Linux**: Terminal

Action	Windows (cmd)	Mac / Linux
Where am I?	`cd` or `chdir`	`pwd`
List files	`dir`	`ls`
Change folder	`cd foldername`	`cd foldername`
Go up one level	`cd ..`	`cd ..`
Make folder	`mkdir name`	`mkdir name`
Clear screen	`cls`	`clear`

Activity: Navigate!

1. Open your terminal.
2. Type `dir` (Win) or `ls` (Mac/Lin) to see where you are.
3. Create a new folder:

```
mkdir lab_demo
```

4. Enter that folder:

```
cd lab_demo
```

5. Check it's empty (run list command again).

Part 4: Running Python Locally

Goodbye, "Play" button

Creating a Script

1. Open a text editor (Notepad, VS Code, Sublime).
2. Write this code:

```
# hello.py
print("Hello from my local computer!")
```

3. Save the file as `hello.py` inside your `lab_demo` folder.

Running the Script

In your terminal (make sure you are inside `lab_demo`):

Windows:

```
python hello.py
```

Mac/Linux (sometimes requires `python3`):

```
python3 hello.py
```

Result: You should see "Hello from my local computer!" printed in the terminal.

Part 5: Modular Code & Imports

Breaking code into pieces

The `import` System

Real software isn't written in one giant file. We split it up.

Scenario:

- `my_module.py`: Contains useful functions.
- `main.py`: Uses those functions.

File 1: `my_module.py`

Create this file in `lab_demo`:

```
def greet(name):
    """
    A simple function to return a greeting message.
    """
    return f"Hello, {name}! Welcome to your first local Python lab."

def add(a, b):
    return a + b
```

File 2: `main.py`

Create this file in `lab_demo`:

```python
import sys
# Import from our local file
from my_module import greet

def main():
    # sys.argv allows us to access command line arguments
    if len(sys.argv) > 1:
        user_name = sys.argv[1]
    else:
        user_name = "Future Engineer"

    message = greet(user_name)
    print(message)

if __name__ == "__main__":
    main()
```

Running the Modular Code

Run `main.py` from the terminal:

```
python main.py
```

Output: Hello, Future Engineer! ...

Now try passing an argument:

```
python main.py Alice
```

Output: Hello, Alice! ...

The Mystery: `if __name__ == "__main__":`

Why do we write this?

```python
if __name__ == "__main__":
    main()
```

- **When you run `python main.py`**:
  Python sets the internal variable `__name__` to `__main__`. The code runs.
- **When you run `import main`** (in another script):
  Python sets `__name__` to `"main"` (the filename). The code **DOES NOT** run automatically.

Rule: This prevents your script from executing immediately when imported as a library.

Part 6: Virtual Environments

Keeping projects separate

The Dependency Hell Problem

- Project A needs `pandas` version 1.0.
- Project B needs `pandas` version 2.0.
- You can't install both globally!

Solution: Virtual Environments (`venv`).
Each project gets its own isolated folder of libraries.

Creating a `venv`

In your terminal (inside `lab_demo`):

```
python -m venv venv
```

(Mac/Linux: `python3 -m venv venv`)

This creates a folder named `venv`.

Activating the `venv`

You must "turn on" the environment before using it.

Windows (Command Prompt):

```
venv\Scripts\activate
```

Windows (PowerShell):

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

Mac/Linux:

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

Success: You will see `(venv)` appear at the start of your command prompt.

Part 7: Pip & Requirements

The Package Installer for Python

Using `pip`

Once your venv is active, you can install libraries safely.

Install a package:

```
pip install requests colorama
```

List installed packages:

```
pip list
```

The `requirements.txt` file

Share your project's dependencies with others.

1. **Create the file** (save current libraries to file):

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

2. **Install from file** (how others setup your code):

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

Lab Activity: Complete Workflow

Task:

1. Navigate to `lab_demo`.
2. Create a virtual environment: `python -m venv venv`.
3. Activate it.
4. Create a `requirements.txt` with:

```
colorama==0.4.6
```

5. Install it: `pip install -r requirements.txt`.
6. Modify `main.py` to use `colorama` (print in color!).
7. Run it!

Questions?

Thankyou for your attention!