Transitioning from Colab to Local Python Development

Lab · CS 203: Software Tools and Techniques for AI

Introduction to Python & Linux

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. Refresh your Python basics.
2. Run Python in different modes (Interactive vs Scripts).
3. Setup and use a Linux environment (WSL/Lab).
4. Master the Command Line Interface (CLI).
5. Work with Intermediate Python concepts (JSON, Pandas, I/O).
6. Create Virtual Environments to manage libraries.
7. Understand Modular Code & Imports.

Part 2: Python Refresher

A quick warm-up

Python Cheat Sheet

> **Resource**: Python Cheat Sheet

Key Concepts to Remember:

- **Variables**: `x = 5`, `name = "Alice"`
- **Lists**: `items = [1, 2, 3]`
- **Loops**: `for i in items: print(i)`
- **Functions**:

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

Part 3: Running Python

Interactive, Scripts, and Cloud

Modes of Execution

Mode	Tool	Use Case	Command
Interactive	Jupyter Notebook	Data Science, Exploration	`jupyter notebook`
Script	Python Interpreter	Production, Automation	`python script.py`
Cloud	Google Colab	Quick start, GPU access	(Browser)
Terminal	Python REPL	Quick math, testing	`python`

1. Interactive Mode: Jupyter Notebook

What is it? A web-based interactive computing platform.

- **Cells**: Code is divided into chunks.
- **State**: Variables persist between cells.
- **Rich Output**: Graphs, tables, and text inline.

How to run locally:

```
pip install notebook
jupyter notebook
```

2. Scripts: The `.py` File

What is it? A plain text file containing Python code.

- **Structure**: Top-to-bottom execution.
- **Usage**: Building apps, servers, tools.
- **Example**:

```
# abc.py
print("Running as a script!")
```

Run it:

```
python abc.py
```

3. Google Colab & The `!` Magic

In Colab (and Jupyter), you can run shell commands using `!`.

Examples:

- `!ls`: List files in the cloud VM.
- `!pip install pandas`: Install libraries.
- `!cd`: **Note** - `!cd` runs in a subshell, so it won't change the persistent directory. Use `%cd` instead.

Part 4: The Linux Environment

The Operating System of AI

Why Linux?

- **Industry Standard**: Most servers and AI clusters run Linux.
- **Tools**: Best support for Docker, Git, and dev tools.
- **Control**: Powerful command line automation.

Your Options:

1. **Lab Computers**: Native Linux installation.
2. **Windows Subsystem for Linux (WSL)**: Run Linux inside Windows.
3. **Mac**: Unix-based (very similar to Linux).

Setup: WSL (Windows Users)

If you are on Windows, **WSL is highly recommended**.

1. Open PowerShell as Administrator.
2. Run: `wsl --install`
3. Restart computer.
4. Open "Ubuntu" from Start Menu.

Now you have a full Linux terminal inside Windows!

Using Linux

The GUI (Graphical User Interface):

- Like Windows/Mac, you have windows, menus, and apps.
- **File Manager**: Browse files visually.
- **Text Editor**: VS Code, Gedit, Sublime.

The Terminal:

- The "Black Box".
- Where the real power lies.

Basic Linux Commands

Command	Purpose	Example
`pwd`	Print Working Directory	`pwd`
`ls`	List files	`ls -l`
`cd`	Change Directory	`cd Folder`
`cat`	View file content	`cat data.txt`
`touch`	Create empty file	`touch new.py`
`mkdir`	Make Directory	`mkdir lab1`

Editing Files: `nano`

A simple terminal-based text editor.

To create/edit a file:

```
nano hello.py
```

Cheatsheet:

- `Ctrl + O`: **Save** (Write Out)
- `Enter`: Confirm filename
- `Ctrl + X`: **Exit**

Installing Software (CLI)

In Linux, we use package managers (like an App Store for terminal).

Ubuntu/Debian/WSL:

```
# Update list of available software
sudo apt update

# Install a tool (e.g., git, python3, tree)
sudo apt install git tree
```

Activity: Linux Survival

1. Open your terminal (WSL or Lab Terminal).
2. Check where you are: `pwd`
3. Create a directory: `mkdir week1_lab`
4. Enter it: `cd week1_lab`
5. Create a file: `touch script.py`
6. Edit it with nano: `nano script.py` → add `print("Hello Linux")`
7. Run it: `python3 script.py`

Part 5: Intermediate Python

Libraries, Data, and I/O

Python Libraries

Python's power comes from its ecosystem.

Installing libraries:

```
pip install pandas numpy
```

Using libraries:

```
import math
print(math.sqrt(16))

import pandas as pd
# pd is an "alias" for pandas
```

JSON (JavaScript Object Notation)

Standard format for data exchange (APIs, config files).

```
import json

# Data dictionary
data = {"name": "Alice", "score": 95}

# Writing to string
json_str = json.dumps(data)

# Reading from string
parsed = json.loads(json_str)
```

Pandas: DataFrames

The Excel of Python.

```
import pandas as pd

# Creating a DataFrame
df = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Age": [25, 30]
})

print(df)

# Reading CSV
# df = pd.read_csv("data.csv")
```

File I/O (Input/Output)

Reading and writing files directly.

```
# Writing
with open("test.txt", "w") as f:
    f.write("Hello World\n")

# Reading
with open("test.txt", "r") as f:
    content = f.read()
    print(content)
```

Using `with` ensures the file is closed automatically.

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:

```
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
```

Part 8: 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:

```
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:

```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! ...

Relative Imports

When files are in subfolders:

```
project/
├── main.py
└── utils/
├── __init__.py
└── helper.py
```

In `main.py`:

```
from utils.helper import some_function
```

Note: Relative imports (using `.`) work best within packages.

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.

Questions?

Thankyou for your attention!