Business Analytics Workshop

Yuan Tian

2025-10-13

Table of contents

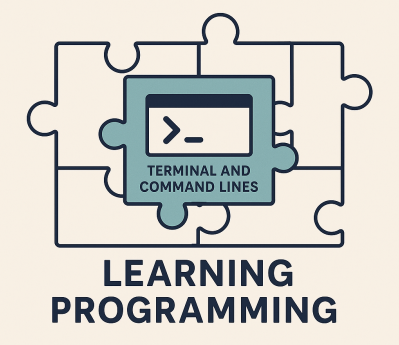
Welcome

Welcome to the Business Analytics Workshop 2025!

Building on prior exposure to programming and modeling fundamentals, this workshop focuses on real-world applications through cutting-edge tools like VS Code, Git, GitHub Copilot, and no-code/low-code AI agent platforms. Participants will engage with case-based exercises and live demonstrations to explore how modern data-driven teams collaborate, code, and build AI-enhanced solutions in agile business environments. This workshop seeks to bridge analytical modeling, modern software practices, and AI development workflows.

1 Command Lines and Terminal

1.1 The Missing Piece



Many students begin your programming journey by writing Python directly in platforms like **Google Colab** - no setup, no files, no terminals. These Jupyter notebook environments make coding easy; however, this convenience often means skipping an important part of programming: **working with real Python files (`.py`) and using the terminal to run and manage code**.

This "missing piece" is **essential** for understanding how programming works in real projects, **servers**, and **production systems**.

> ⚠ Importance of Command Line and Terminal
>
> *If you **skip** learning command line skills or avoid the **terminal**, you'll struggle to work on real-world projects, collaborate effectively with teams, or operate in servers or cloud platforms — where graphical interfaces aren't available. The **terminal** isn't just a tool for experts; it's the foundation for professional workflows in data science and engineering.*

So, before diving deeper into advanced business data workflows, we'll start by filling this gap, and learn the command lines and terminal to navigate files, run Python scripts, and operate in professional computing environments.

What You'll Learn Next

- Main operating systems: Windows, macOS, Linux/Unix

- GUI vs terminal and why terminals matter
- Bash shell basics for programming and data science

1.2 Operating Systems Overview

Most students in this course use **Windows**, which dominates *personal computers* with roughly **70–75%** of the global desktop market. **macOS** holds about **15–20%**, while **Linux** and others make up a small share of *personal use*.

In contrast, the **enterprise, cloud, AI/ML, and high-performance computing (HPC)** worlds are very different. **Linux** and other **Unix-like systems** are the backbone in web servers, cloud computing and supercomputers, making up nearly half of cloud workloads and being the OS for **all top 500 supercomputers**. Popular Unix and Linux systems include:

- **Ubuntu**
- **Debian**
- **Fedora**
- **Red Hat Enterprise Linux (RHEL)**

> **i** macOS is Unix-based
>
> Although **macOS** looks different, it is actually **Unix-based**, meaning the **terminal commands** and **Bash shell** you'll learn in this course work much the same on both macOS and Linux.

- **Servers** Linux holds a 62.7% market share for server operating systems.
  - Web servers: **77–88%** of public web servers run on **Linux or other Unix-like systems**. It is the most used operating system for web servers globally.
- **Cloud computing** Cloud workloads are heavily dependent on **Linux-based** operating systems. As of mid-2025, Linux powers 49.2% of all global cloud workloads.
- **Supercomputers** Linux has a complete monopoly in the supercomputing sector. 100% market share: Since 2017, **100% of the world's top 500 supercomputers have run on Linux**.
- **AI and ML workloads** Linux is the clear leader for AI and ML projects and infrastructure. In mid-2025, 87.8% of machine learning workloads ran on **Linux infrastructure**. Large ML and data science deployments predominantly run on Linux-based or Unix-based servers.
  - Cloud environments: Cloud providers like AWS, Google Cloud (GCP), **Colab**, and Microsoft Azure, which are leading providers for AI services, primarily offer Linux-based instances for running AI and ML tasks.

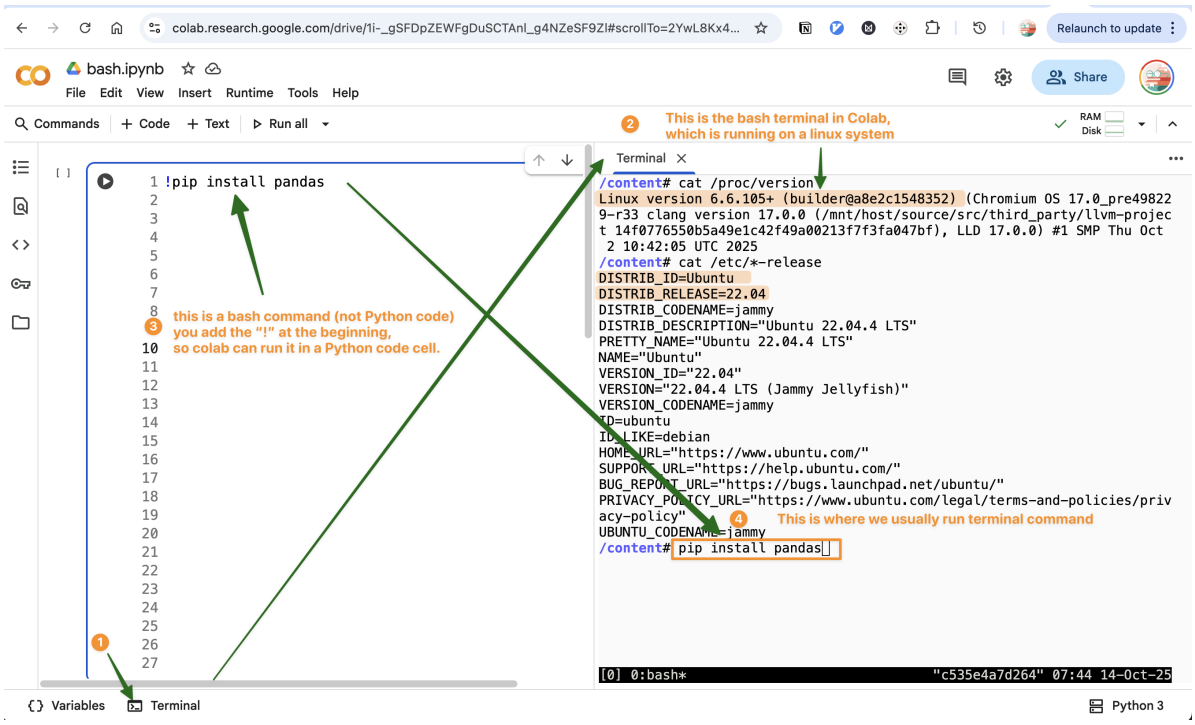
Source: Wikipedia - Usage share of operating systems Azure Official Page, Microsoft Tech Community Update (Feb 2025)

1.2.1 What Operating System does Google Colab use?

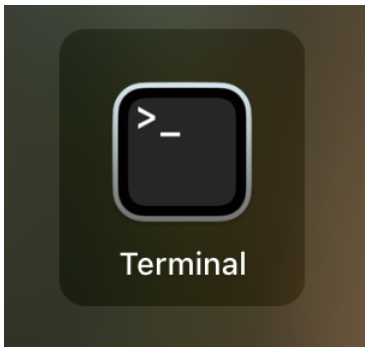
Let's take a look at what is the operating system (OS) that running Google Colab. You can type the following command lines in the Terminal.

```
# Bash
# Display info about the operating system
cat /etc/*-release

# Display the Linux kernel version and build info
cat /proc/version
```

See image below for the outputs.



1.3 What Is a Terminal?



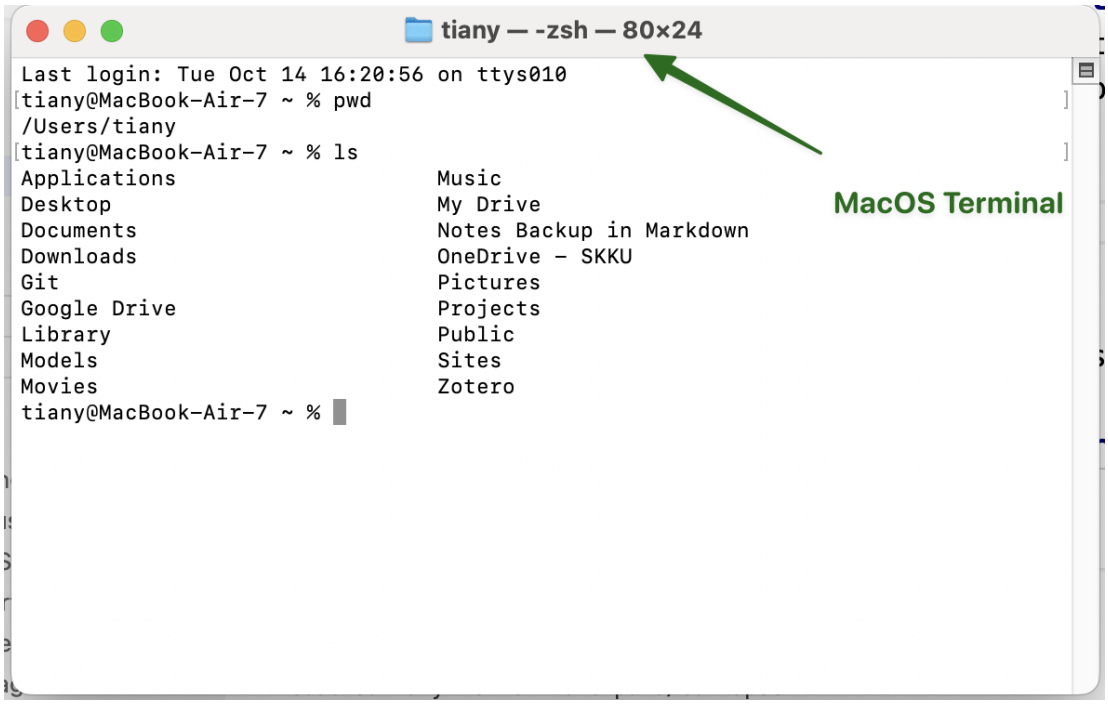
A **terminal** (also called a **command line** or **shell**) is a text-based interface that lets you interact directly with your computer by typing commands.

Before graphical interfaces (with windows, icons, and a mouse) were invented, the **terminal was the primary way users operated computers** — to run programs, manage files, and control hardware.

EVERY operating system includes a terminal app:

- **Windows**:
  - Command Prompt(`cmd`)
  - **PowerShell**
  - or Bash (through Windows Subsystem for Linux)
  - **Linux**: **Bash** is the default shell on most Linux systems (see Colab terminal)
- **macOS**: **Zsh** in Terminal app (based on Unix) is the default terminal in MacOS, see image below.

> **i** Bash vs Zsh
>
> - Both **Bash** and **Zsh** are terminals that interpret your commands, and **they work almost the same**.



The terminal can do almost everything you normally do with a mouse:

- Navigate files and folders
- Run programs or scripts
- Install and manage software
- Connect to remote servers
- Automate repetitive tasks with shell scripts

Data scientists and developers rely on the **terminal** for its speed and automation, especially when working in **cloud environments** like Google Colab, GitHub codescpaces, or on Linux servers.

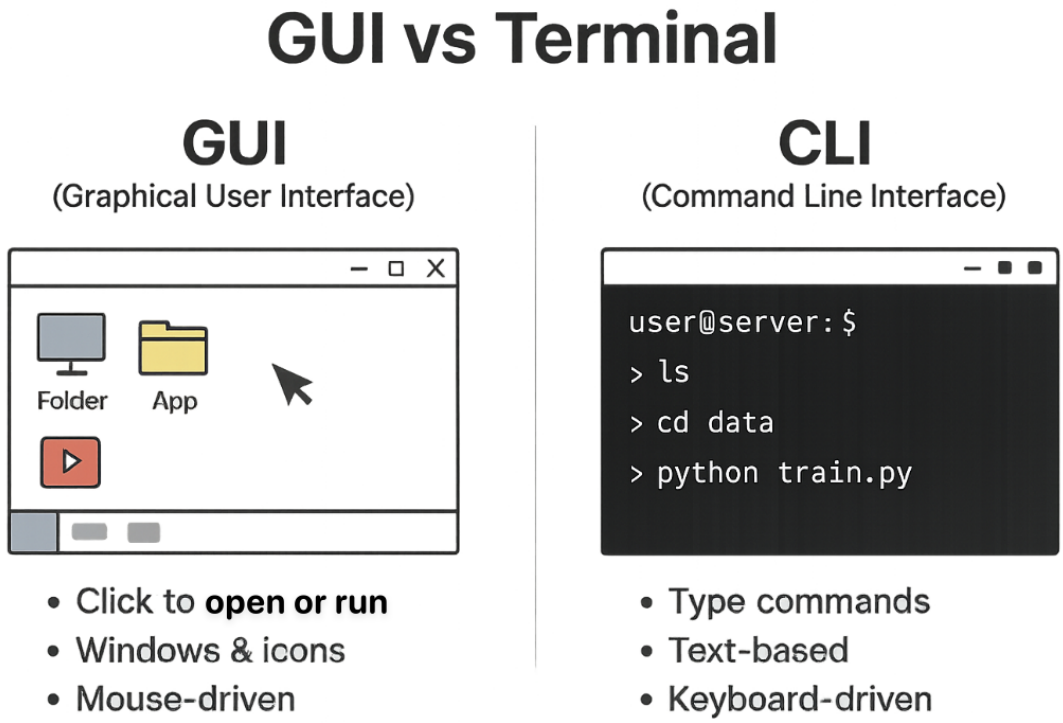
1.4 Why Learn Bash commands and Terminal?

First, data science projects often run on **servers or cloud environments**, not personal laptops which lack the computational power for large-scale training, data processing, or deployment.

These servers — such as AWS EC2, Azure VMs, or Google Cloud Compute instances — usually run Linux or Unix systems and don't include a **graphical user interface (GUI)** by default. — they are managed entirely through the **command line interface (CLI)**. To interact with them efficiently, you use **Bash**, a powerful and widely used command-line shell.

> What is a GUI?
>
> A **Graphical User Interface (UI)** is the visual part of your computer — windows, buttons, and menus you click with the mouse. However, **Linux servers** don't usually have this kind of visual interface.
> Instead, users interact with them through **script commands** typed into a **terminal** such as **bash**.

1.4.1 GUI, CLI, Terminal and Desktop



- **GUI (Graphical User Interface)** – The visual interface you use with a **mouse, icons, and windows**, such as Windows desktop, macOS Finder. GUIs are user-friendly but less efficient for automation or remote access.

- **CLI (Command Line Interface)** – A **text-based interface** where you type commands instead of clicking.
- **Terminal** – The **program that provides access to the CLI**. It's like a window that lets you type commands and see text output, such as Windows PowerShell, macOS Terminal, Linux bash Terminal.
- **Desktop Environment** – The **collection of GUI components** that make up the user's graphical workspace — including the taskbar, file explorer, and app windows; such as Windows Desktop, macOS.

Summary

- The **Terminal** gives you access to the **CLI**, while the **Desktop Environment** provides a **GUI**.
- Both let you control the same computer — one through text, the other through graphics.

1.5 Learning Bash commands in Colab

Mastering Bash is essential. It enables you to write scripts, manage jobs, and execute commands directly on compute servers — a critical skill when working with **large datasets** or **LLM pipelines**.

We are going to learn **basic** bash commands to:

- Navigate and manage files
- Run **Python (.py) scripts** and other programs (e.g., `pip`) directly from the command line
- Work efficiently within **server-based** or **local terminal** environments

We will use Google Colab to learn bash commands in Linux system.

Please create or open a new Colab notebook, and then open the Terminal panel.

1.5.1 File Directory in Google Colab

When you launch a new Colab notebook, the environment starts with a temporary Linux file system that looks like this:

```
/
   bin/
   boot/
   content/
      drive/           ← your Google Drive (if mounted)
      sample_data/     ← sample datasets provided by Colab
      (your files)     ← any files you upload or create
   dev/
   etc/
   home/
      root/
   lib/
   media/
   mnt/
   opt/
   proc/
   root/               ← default home directory if you type `cd ~`
   run/
   sbin/
   srv/
   sys/
   tmp/
   usr/
```

1.5.2 Lab: Linux and bash

- Display info about the operating system.

```
cat /etc/*-release
```

- Display the Linux kernel version and build info.

```
cat /proc/version
```

1.5.3 Lab: Paths, Folders, Directories (`pwd`)

- Print your current working directory (the folder you are "in"). A directory is a folder, directory and folder are the same thing.

```
pwd
#/content
```

Please type `pwd` 5 times and each time say "print working directory".

When to use `pwd`? if you lost in folders and don't know where you are in the directories or folders, `pwd` will tell you where you are.

```
/content# pwd
#/content
```

1.5.4 Lab: List Directory (`ls`)

The `ls` command is used to **list files and folders** in a directory.
Here are some of the most commonly used ones with options (such as `-a`, `-l`)

```
# List files and folders in the current directory
ls

# List **all** files, including hidden ones (those starting with .)
ls -a

# List files in a detailed (**long**) format - shows permissions, owner, size, and date
ls -l

# Combine options: show all files in detailed view
ls -la

# Sort files by modification **time** (newest first)
ls -lt
```

1.5.5 Lab: Change Directory (`cd`)

- `cd sample_data`: go the `sample_data` folder under the current directory.
- `cd ..`: go the parent folder.
- `cd ~`: go to the home folder. In Colab, the home folder is `\root`. If you are lost in a directory and want to start over from a **safe** directory – your home. You can type `cd ~`, and you will be taken to the home directory.

```
# go into the sample_data folder
ls
cd sample_data
ls


# Move up one folder (to the parent directory /content)
```

```
cd ..

# Go back to your "home" folder (/root in Colab)
cd ~
pwd

# To-do: find a way to go back to the /content folder.
```

1.5.6 Lab: Make A Directory (`mkdir`)

```
# Create a new folder named "data" under /content
mkdir data

# Make multiple folders at once
mkdir project scripts results

# Check that they were created
ls
```

1.5.7 Lab: curl

Go to the `"data"` directory, and download a file from the internet. in `curl -O <URL>`, `-O` stands for saving file with the same name as on the web server.

```
# Go to the "data" Directory

cd data

# Download a small sample text file and save it with the same name.
curl -O https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.dat

# List files to confirm it's there
ls
```

1.5.8 Lab: Clear the Screen (`clear`)

```
# Clear the terminal screen
clear
```

1.5.9 Lab: Remove Directory (`rmdir`)

```
# Create an empty folder named "temp_folder"
mkdir temp_folder

# Remove the empty folder
rmdir temp_folder

# Create multiple empty folders and remove them
mkdir folder1 folder2
rmdir folder1 folder2
```

1.5.10 Lab: Making Empty Files (`touch`)

```
# Create an empty file named "notes.txt"
touch notes.txt

# Create multiple files at once
touch a.txt b.txt c.txt

# Verify files were created
ls
```

1.5.11 Lab: Copy a File (`cp`)

```
# Copy a file to a new file
cp notes.txt notes_backup.txt

# Create a folder to copy into
mkdir backup

# Copy a file into a different folder
cp notes.txt backup/

# Check the results
ls backup
```

1.5.12 Lab: Moving/Rename a File (`mv`)

```
# Move a file into a different folder
mv notes_backup.txt backup/
```

```
# Rename a file
mv notes.txt todo.txt

# Verify the changes
ls
```

1.5.13 Lab: Stream a File (`cat`)

```
# Display the contents of a file
cat todo.txt

# To-do: Display the README.md file in the "sample_data" folder:


# Display a system file (try this!)
cat /etc/*-release
```

1.5.14 Lab: Removing a File (`rm`)

```
# Create some temporary files first
touch old.txt temp.txt sample.txt

# Remove a single file
rm old.txt

# Remove multiple files
rm temp.txt sample.txt

# Remove an entire folder and its contents (be careful!)
rm -r backup
```

1.5.15 Lab: Exiting Your Terminal (`exit`)

```
# Exit the current terminal session
exit
```

1.6 Summary Table – Common Bash Commands

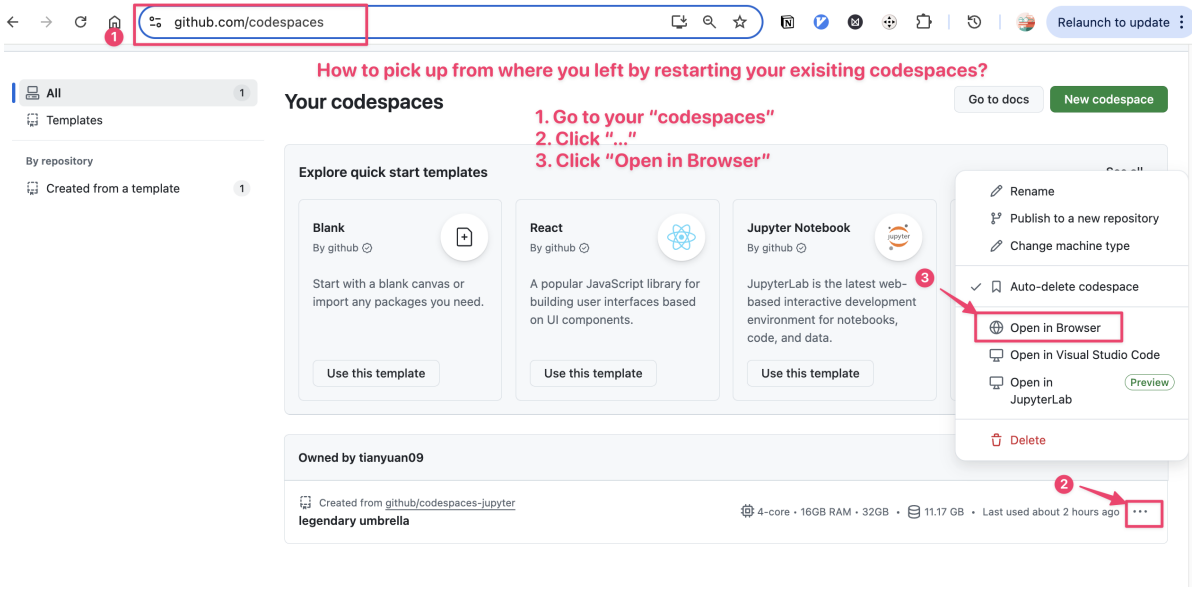
Command	Purpose	Example
`pwd`	Print working directory	`pwd`
`ls`	List files and folders	`ls -la`
`cd`	Change directory	`cd /content`
`mkdir`	Make a new directory	`mkdir data`
`rmdir`	Remove an empty directory	`rmdir temp_folder`
`curl`	Download a file from the internet	`curl -O https://example.com/file.txt`
`touch`	Create an empty file	`touch notes.txt`
`cp`	Copy a file	`cp notes.txt backup/`
`mv`	Move or rename a file	`mv old.txt new.txt`
`cat`	View contents of a file	`cat notes.txt`
`rm`	Remove a file or folder	`rm -r foldername`
`clear`	Clear the screen	`clear`
`exit`	Exit the terminal	`exit`

1.7 Directory Structure in GitHub Codespace Terminal

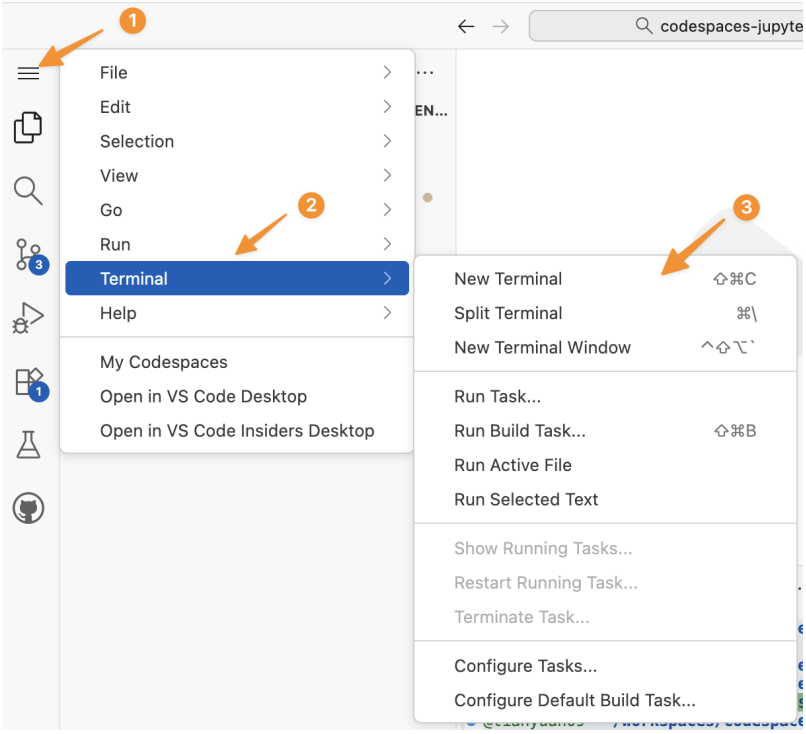
```
/
  bin/
  boot/
  dev/
  etc/
  home/
     codespace/          ← your user home directory if you do `cd ~`
  lib/
  lib64/
  media/
  mnt/
  opt/
  proc/
  root/
  run/
  sbin/
  srv/
  sys/
  tmp/
  usr/
  workspaces/
     /codespaces-jupyter     ← your GitHub repo (default working dir)
```

1.7.1 Bash in VS codespaces

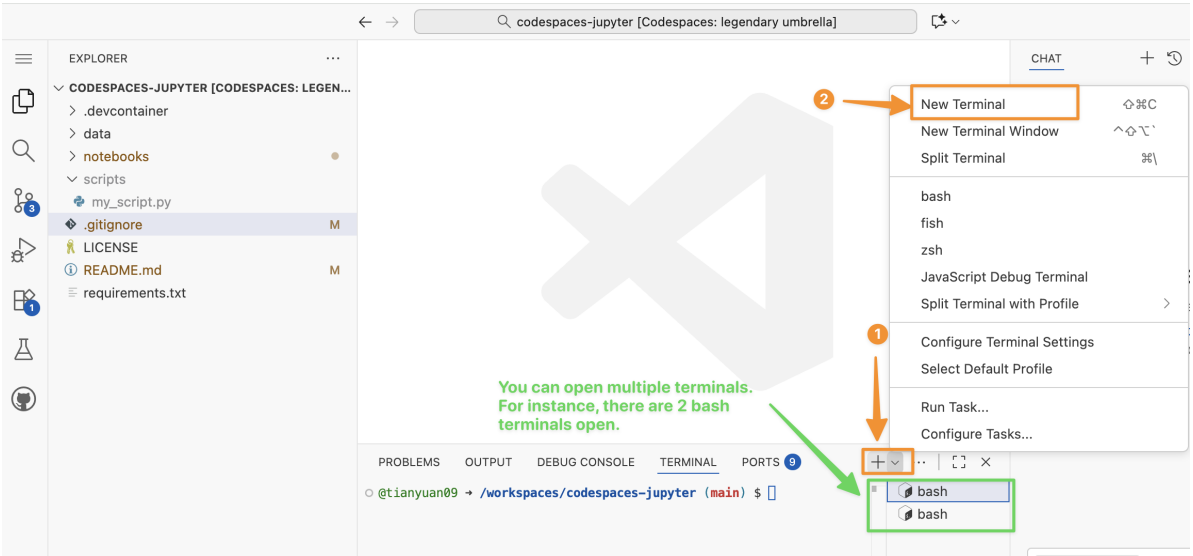
Restart your github codespaces.



Make sure your can see the Terminal panel. If you accidently closed your terminal, you can always start one (or many) following the steps below.



You also can start a second terminal via the Terminal panel.



1.7.2 Lab: create the `scripts` folder

Using the bash terminal to create a `scripts` folder under GitHub default working dir `/workspaces/codespaces-jupyter`, and create an empty `my_script.py` file in the `scripts` folder.

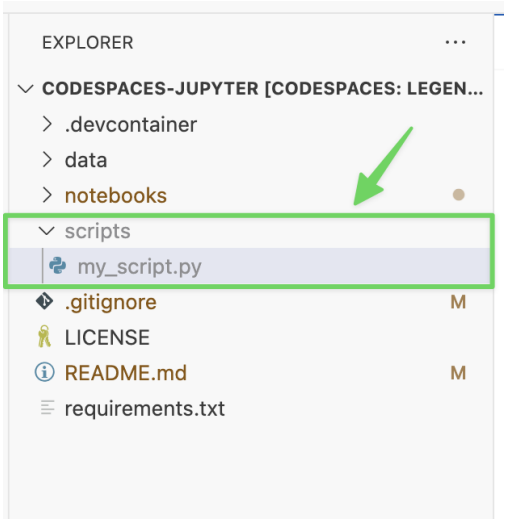
> **Tip**
>
> You may find the commands `pwd`, `mkdir`, `ls`, `cd`, and `touch` helpful for completing this exercise.

Your task is to:

- Verify your current working directory.
- Create a new folder called `scripts` inside the project root.
- List the directory contents to confirm that `scripts` was created successfully.
- Navigate into the `scripts` folder.
- Confirm it is empty.
- Create a new Python file named `my_script.py` inside the `scripts` folder.

Write the Bash commands needed to accomplish each step.

Once you complete the task, your explorer should look like below:



Solution

```
@tianyuan09 /workspaces/codespaces-jupyter (main) $ pwd
/workspaces/codespaces-jupyter
@tianyuan09 /workspaces/codespaces-jupyter (main) $ mkdir scripts
@tianyuan09 /workspaces/codespaces-jupyter (main) $ ls
LICENSE README.md data notebooks requirements.txt scripts
@tianyuan09 /workspaces/codespaces-jupyter (main) $ cd scripts
@tianyuan09 /workspaces/codespaces-jupyter/scripts (main) $ ls
@tianyuan09 /workspaces/codespaces-jupyter/scripts (main) $ touch my_script.py
```

2 Run `.py` scripts with Bash

2.1 Why Run Python Scripts from Terminal?

While interactive environments like **Jupyter notebooks** are great for exploration and prototyping, running **Python scripts (.py files)** from the terminal is essential for:

- **Production workflows**: Automated data processing, model training, and deployment
- **Server environments**: Most servers don't have graphical interfaces
- **Batch processing**: Running scripts on large datasets or multiple files
- **Scheduling**: Using cron jobs or task schedulers to run scripts automatically
- **Command-line arguments**: Passing parameters to scripts dynamically
- **Performance**: Scripts often run faster than notebooks for large tasks

> **i** For Example
>
> Data scientists often submit training jobs via Bash scripts like:
>
> ```
> python train_model.py --epochs 10
> ```
>
> It runs the Python script `train_model.py` to train the model for 10 epochs.

2.2 `.ipynb` Notebook vs `.py` Script Files

Figure 2.1 illustrates the difference between running Python code in `.ipynb` notebook versus in '.py' script file.

Understanding the Terminal

See directory structure in codespaces in the previous chapter.

In the terminal, it always starts with:

```
@tianyuan09   /workspaces/codespaces-jupyter (main) $.
```

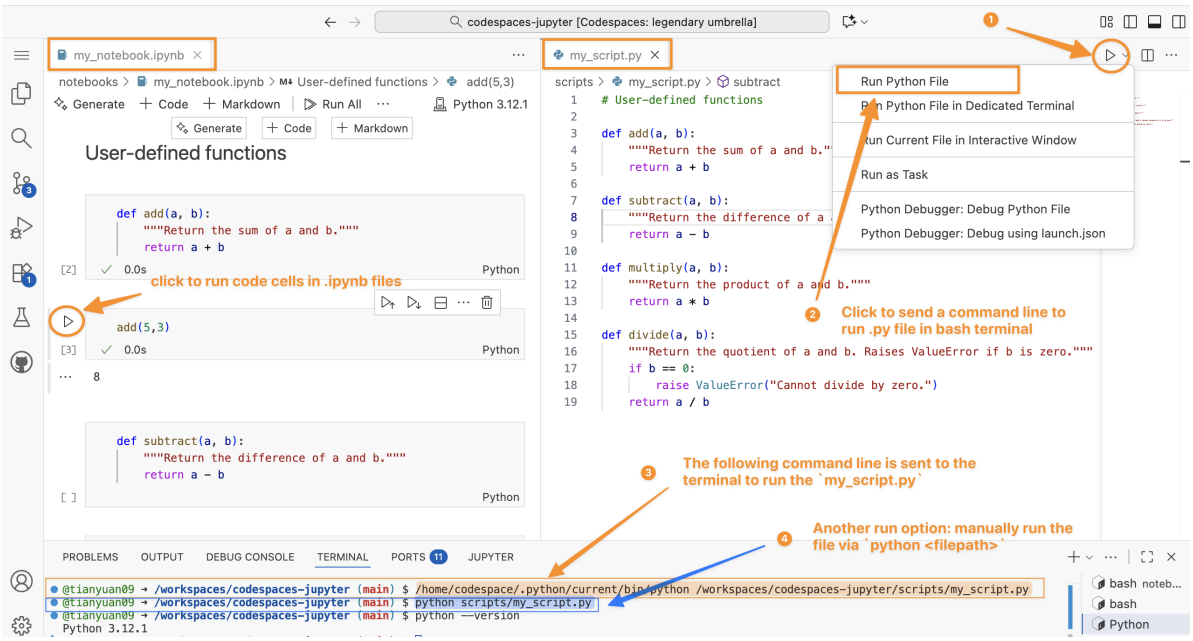


Figure 2.1: Comparison of Jupyter Notebook and Python Script formats

```
@tianyuan09   /workspaces/codespaces-jupyter (main) $

                                                Prompt symbol ($):
                                                 shows the terminal is ready for input

                                          Current working directory:
                                           you're inside the folder "codespaces-jupyter"
                                           located under "/workspaces"

                    Arrow ( ):
                      just a decorative separator in the prompt

  Username (and sometimes host):
   "tianyuan09" - the current user logged into this environment
```

Difference between `.ipynb` Notebook and `.py` Script File.

Feature	`.ipynb` (Jupyter Notebook)	`.py` (Python Script)
Structure	Structured JSON format combining code, text cells in Markdown, and outputs.	Plain text file containing only Python code and comments `#`.

Feature	`.ipynb` (Jupyter Notebook)	`.py` (Python Script)
Execution	Run one cell at a time, showing output immediately below each cell.	Executed all at once using a command like `python my_script.py`, seen in Figure 2.1.
Use Case	Ideal for data analysis, visualization, and teaching due to its interactive nature.	Better for automation, **deployment** of production-ready code.

2.2.1 Lab: Create and run a `.py` script file

First, let's create a `hello.py` under the `scripts` folder.

```
# file path: scripts/hello.py
print("Hello from Python!")
print("This script is running from the terminal.")

# Get current date and time
import datetime
now = datetime.datetime.now()
print(f"Current time: {now}")
```

You can directly use explorer or use bash command Figure 2.2.

Expected output:

```
Hello from Python!
This script is running from the terminal.
Current time: 2025-10-13 10:30:45.123456
```

2.3 Other Python-related Commmands

2.3.1 Checking Your Python Version

```
# Check Python version
python --version
python3 --version
```

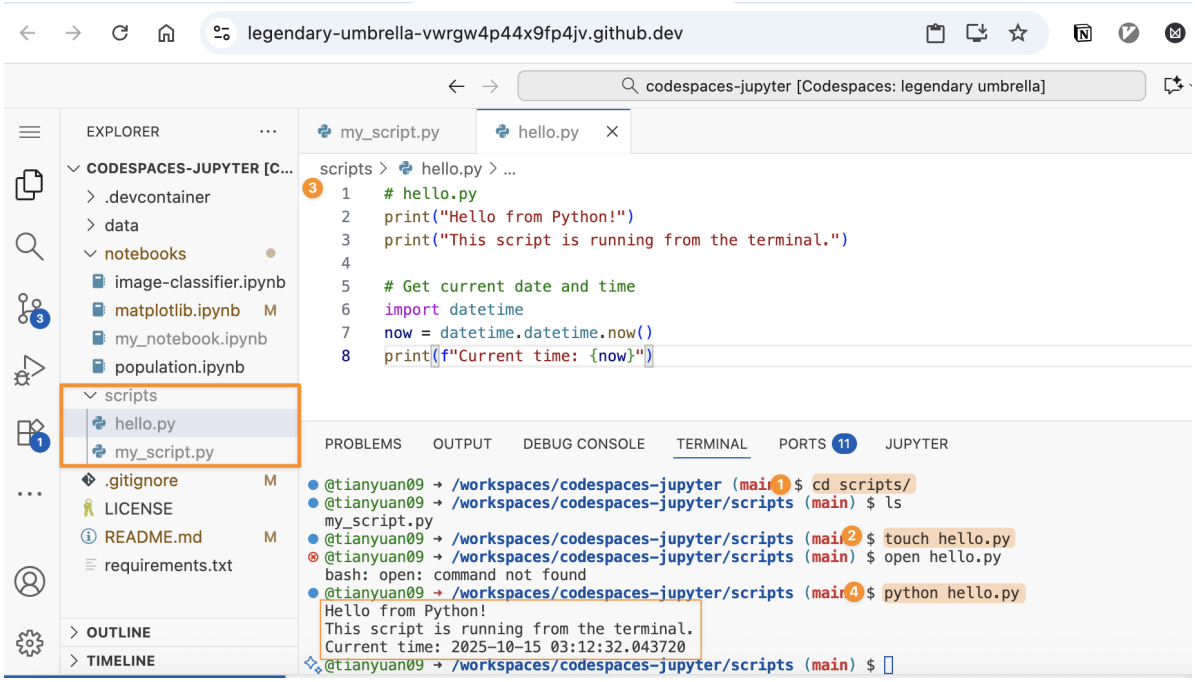


Figure 2.2: Create and run your 1st script file

```
# Check which Python executable you're using
which python
which python3
```

2.3.2 Different Python Commands

Depending on your system setup, you might need to use different commands:

```
# On systems with Python 3 as default
python hello.py

# OR
python3 hello.py

# OR Using specific Python version
python3.9 hello.py
python3.12 hello.py
```

2.3.3 Running `pip` command

> **Tip**
>
> `pip` is a **command-line tool**, **not** Python code. You actually should run `pip install ...` in your terminal.

Although you have run `pip` inside a code cell in Colab with `!`, such as `!pip install pandas`, the notebook actually sends it as a command line to the terminal to execute. It is equivalent to running `pip install pandas` in the terminal, see in Figure 2.3.

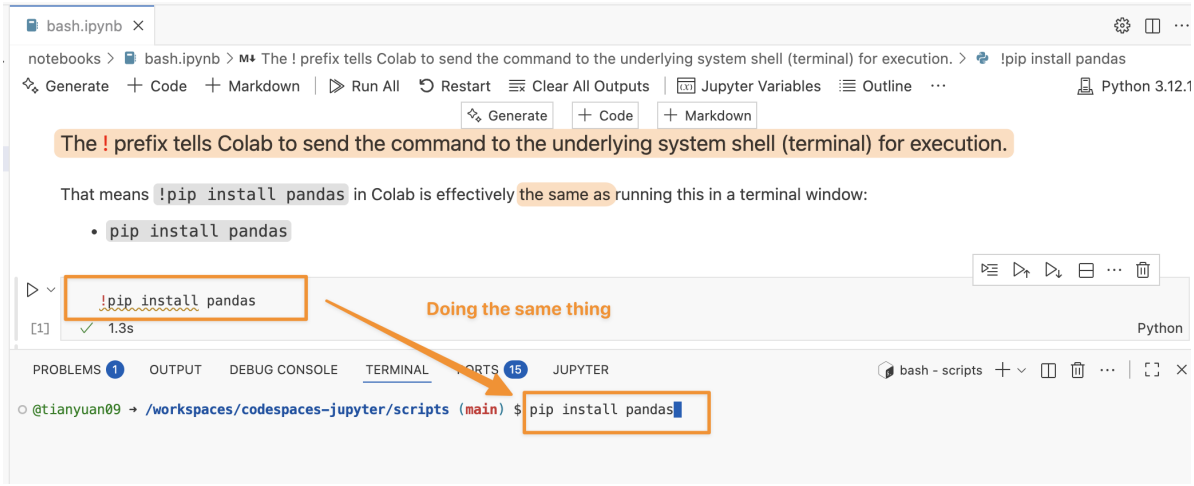


Figure 2.3: `pip` in terminal

> **How to install package for `.py` files?**
>
> In regular `.py` files (non-notebook files), `pip` commands **must be executed in the terminal**, not inside the script itself.
> For example, to install the `streamlit` package, run this in your terminal:
>
> ```
> pip install streamlit
> ```

List all the packages installed:

```
# Check installed packages
pip list
```

Please install streamlit in your codespaces.

2.4 Working with Command-Line Arguments

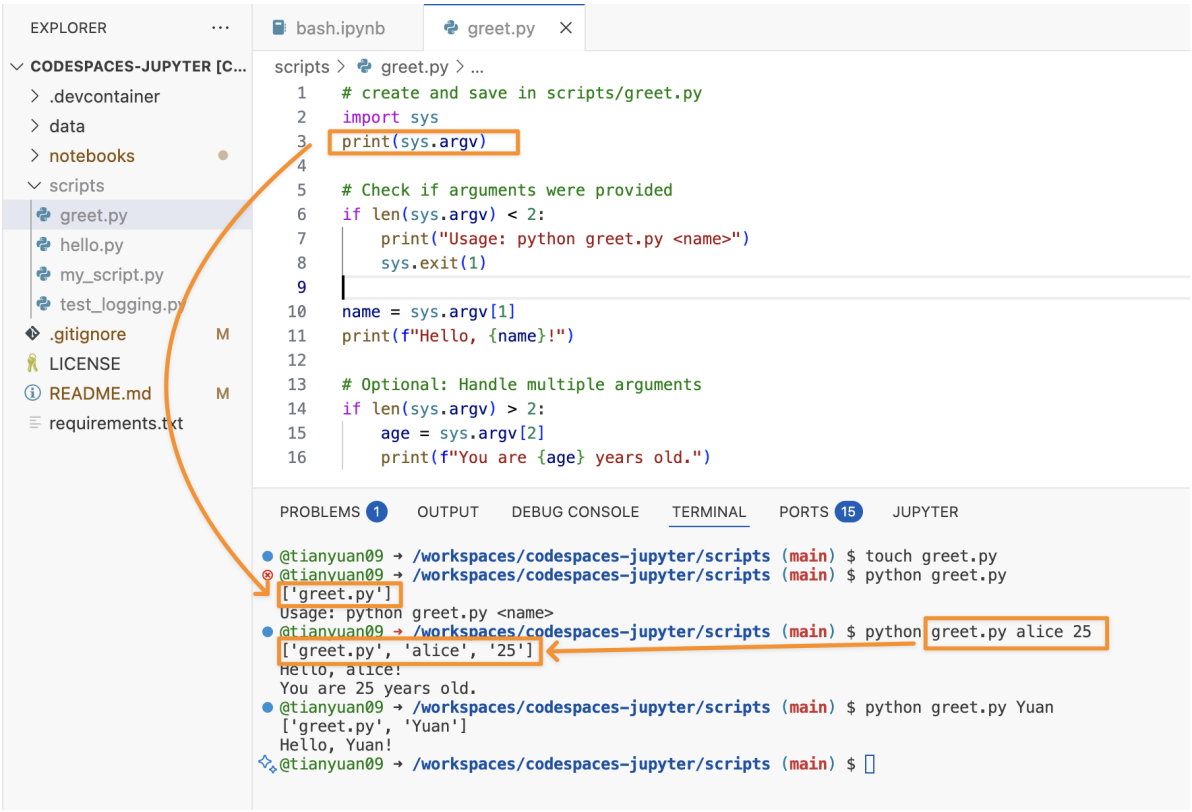
2.4.1 Creating a Script with Arguments

`.py` files with arguments allow the users to provide input directly when running the script. Please work on the following example.

```
# create and save in scripts/greet.py
import sys
print(sys.argv) # sys.argv is a list of string typed after python in the bash

# Check if arguments were provided
if len(sys.argv) < 2:
    print("Usage: python greet.py <name>")
    sys.exit(1)

name = sys.argv[1]
print(f"Hello, {name}!")

# Optional: Handle multiple arguments
if len(sys.argv) > 2:
    age = sys.argv[2]
    print(f"You are {age} years old.")
```

2.4.2 Running with Arguments

```
# Run with one argument
python greet.py Alice

# Run with multiple arguments
python greet.py Bob 25

# Without arguments (will show usage message)
python greet.py
```



2.4.2.1 `sys.argv` — what it really is

`sys.argv` is a **list of strings** containing everything typed after `python` in the terminal.

For example, if you run:

```
python greet.py Alice 25
```

Then in Python:

```
import sys
print(sys.argv)
# output: ['greet.py', 'Alice', '25'] # a list
```

2.4.2.2 Index meaning

Index	Value	Meaning
`sys.argv[0]`	`'greet.py'`	the **name of the script file** being executed

Index	Value	Meaning
`sys.argv[1]`	`'Alice'`	the **first argument** typed after the script name
`sys.argv[2]`	`'25'`	the **second argument** (if given)

2.4.2.3 why `name = sys.argv[1]`?

Because:

- `sys.argv[0]` is always the script name (`greet.py`)
- `sys.argv[1]` is the first real argument that the user provides. In this case, the person's name.

If you used `sys.argv[0]`, it would just say: `Hello, greet.py!`

2.5 Running Scripts in Different Directories.

2.5.1 Absolute Paths

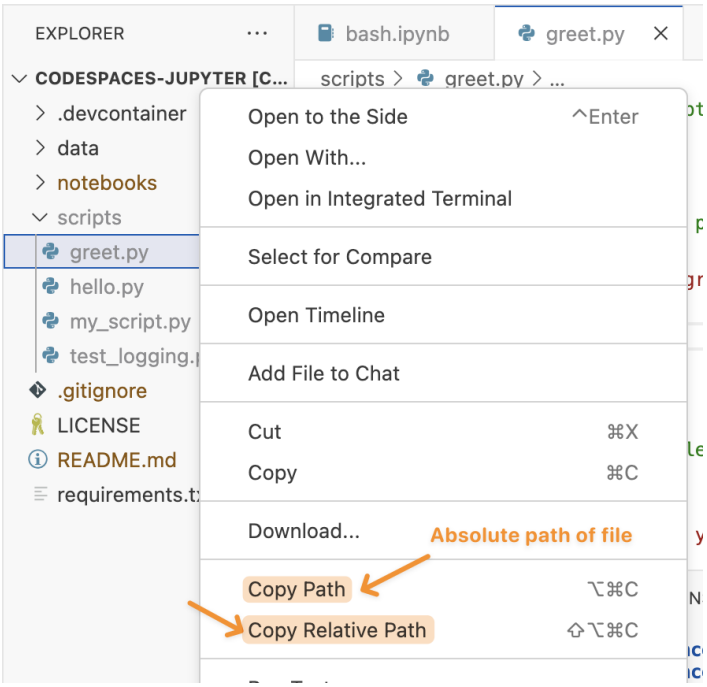
```
# Run script from anywhere using absolute path
python /Users/username/projects/my_script.py
```

`/Users/username/projects/my_script.py` is a absolute file path.

Absolute paths:

- **On macOS/Linux, starts with `/`.**
- **On Windows, starts with a drive letter like `C:/`.**
- In short, it always starts from the root of the filesystem (the top level).

You can find either the relative or aboslute file paths in codespaces:



2.5.2 Relative Paths

```
# Run script in current directory
python ./script.py

# Run script in subdirectory
python scripts/data_analysis.py

# Run script in parent directory
python ../utilities/helper.py
```

The paths above (e.g. `../utilities/helper.py`) are relative paths.

- They **don't start with /** or a drive letter (e.g. `C:/`).
- They may include `.` (current folder) or `..` (parent folder).
- They starts from your current working directory (`pwd`).

2.5.3 Difference in absolute vs relative paths

Starts With	Type	Meaning
/ (Linux/Mac)	Absolute	Starts at root of file system

Starts With	Type	Meaning
`C:\` (Windows)	Absolute	Starts at root of drive
`.` or `..`	Relative	Based on current working directory
No `/` or `C:\`	Relative	Implied to start from current folder

2.5.4 Changing Directories

```
# Navigate to script directory, then run
cd /path/to/scripts
python analysis.py

# Or combine in one line
cd /path/to/scripts && python analysis.py
```

2.6 Make a `.py` file as a module

2.6.1 Python module review

Python modules and packages generally fall into four categories: built-in, standard library, third-party, and user-defined (see in Figure 2.4). So far, you've worked with the first three — modules that come with Python (e.g., `datetime`) or are installed from external sources (e.g., `pandas`).

2.6.2 Create a user-defined Module in `.py` file

In this section, you'll learn how to create your own `.py` module with reuseable functions – one that can be both imported into a Colab notebook and run independently for testing.

Create `my_util.py` in the `scripts` folder.

```
# scripts/my_utils.py

def greet(name):
    """Return a personalized greeting."""
    return f"Hello, {name}!"
```

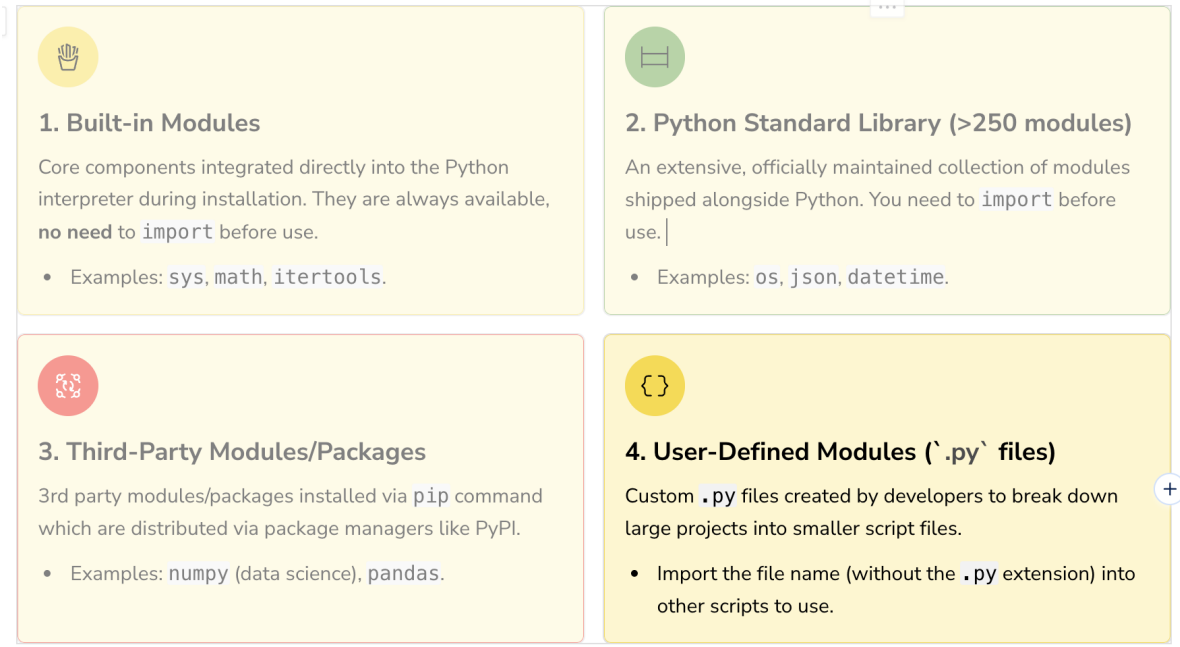
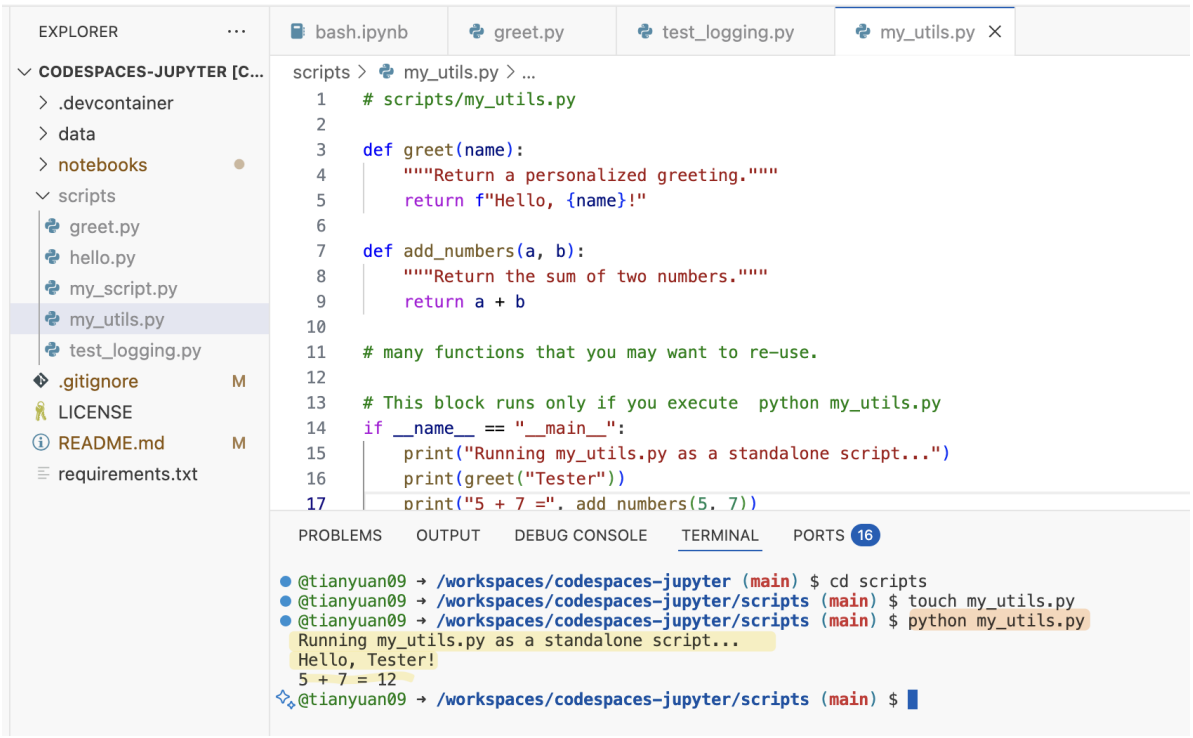


Figure 2.4: Categories of Python Modules/Packages

```python
def add_numbers(a, b):
    """Return the sum of two numbers."""
    return a + b

# many functions that you may want to re-use.

# This block runs only if you execute `python my_utils.py`
# This will make the .py file both run alone or import as a module
if __name__ == "__main__":
    print("Running my_utils.py as a standalone script...")
    print(greet("Tester"))
    print("5 + 7 =", add_numbers(5, 7))
```

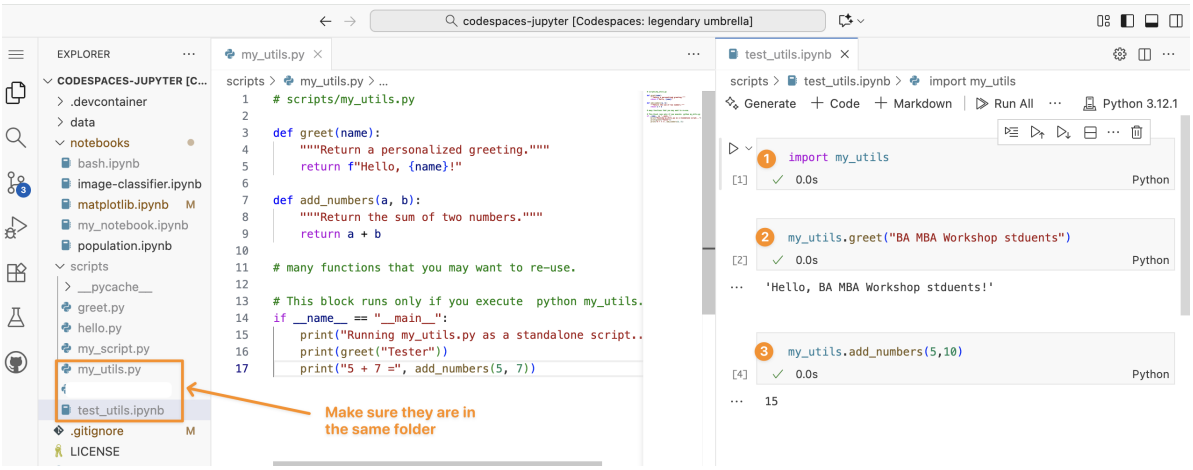
Option 1: Run as a standalone .py file

```
python scripts/my_utils.py
```



Option 2: Run as a user-defined modules with reusable functions

Create a `test_util.ipynb` file in the **same** folder as your `my_utils.py` file. Then, you can do `import my_utils` to use it like any module/package that you have used before.



What does `if __name__ == "__main__"` do?

`__name__` is a special variable that Python automatically sets when a file is run or imported.

- If the file is run directly (e.g., `python my_utils.py`), `__name__` becomes `"__main__"`.
- If the file is imported (e.g., `import my_utils`), `__name__` becomes the module's name (`"my_utils"`).
- `if __name__ == "__main__"` ensures that code inside it runs only when the file is executed directly, not when imported. This allows a `.py` file to act as both:
  - a reusable module (when imported)
  - a standalone script for testing or demos (when run directly).

What if you remove the `if __name__ == "__main__"`: line

If you do that, and then import the module, any code at the bottom of the file will **run automatically during import**, which can **cause unwanted behavior (like printing, running tests, or altering data)**, see Figure 2.5.

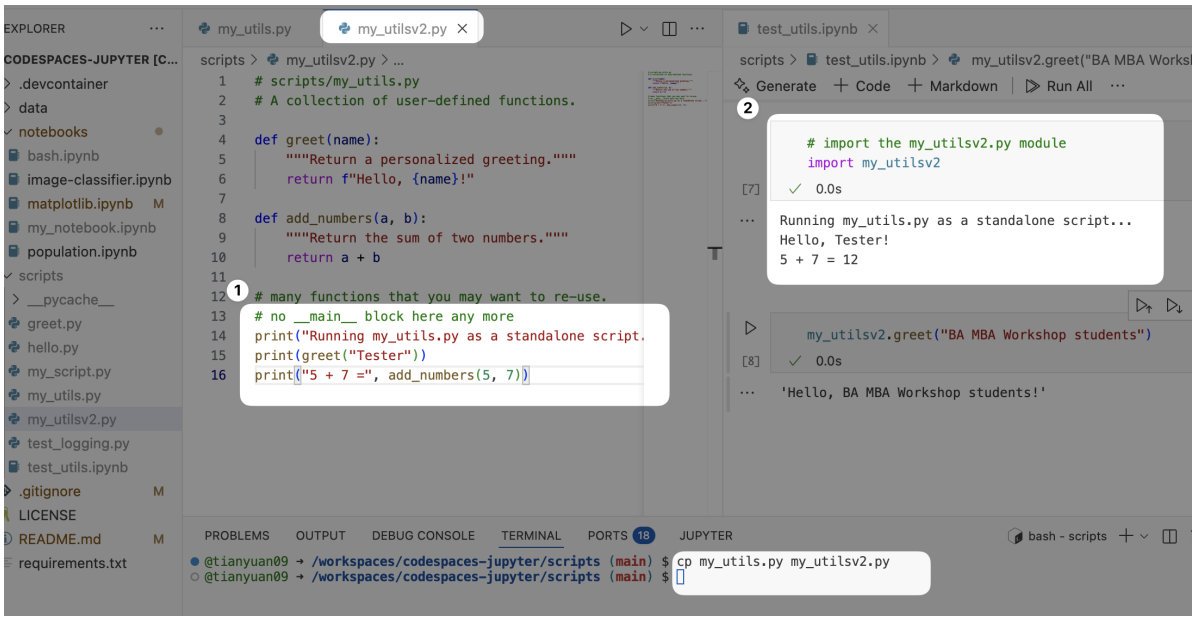


Figure 2.5: Removing `__main__`

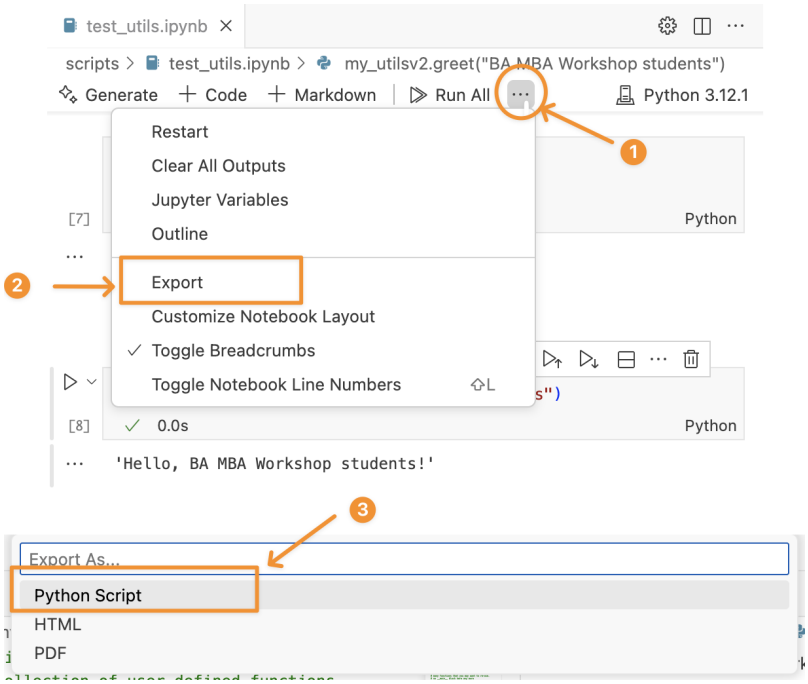
The `if __name__ == "__main__":` line **isn't required** for a module to work, but it helps

keep your code tidy and organized, allowing the file to be used both as a reusable module and as a standalone script with testing.

You also can import a single or multiple functions in a `.py` file.

```
# import the greet(), add_numbers() functions
from my_utils import greet, add_numbers

print(greet("Bob"))           # Output: Hello, Bob!
print(add_numbers(10, 5))     # Output: 10 + 5 = 15
```

2.6.3 Download `.ipynb` file as a `.py` file



3 Develop an app with `streamlit`

3.1 Context and Goal

> **!** Goal
>
> Transition from a prototype `.ipynb` file to a production-ready `.py` file that builds an interactive Streamlit dashboard.

Data scientists often start in Jupyter Notebooks for exploration and analysis. However, real-world applications require deployable, interactive dashboards for sharing insights with others. The goal here is to:

- Refactor your notebook code into modular Python functions.
- Wrap those functions inside an interactive Streamlit interface.
- Understand how to iterate quickly with streamlit run app.py.

3.2 Start from a `.ipynb` file

☐ Download the following files from Canvas course.

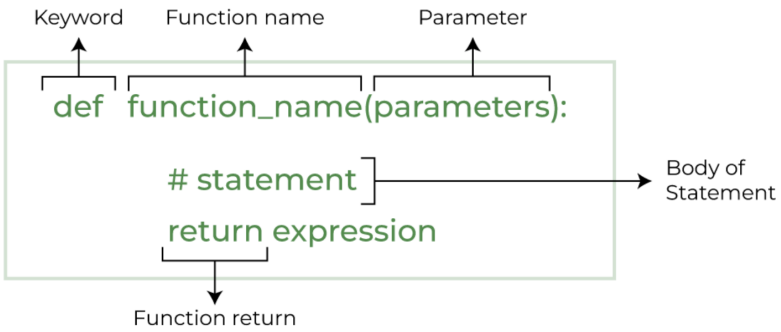
  – `Compute Financial Ratios Notebook.ipynb` – a Jupyter Notebook
  – `sp500_data.csv` – Financial data.
  – `sp500_tickers.csv` – List of tickers.

☐ Create a folder named `streamlit25` in the default directory using bash command.

☐ Drag these three files into this folder in your browser.

3.3 Make it a `.py` file with functions

The syntax of defining a python function.



☐ 1. Read the `Compute Financial Ratios Notebook.ipynb`
☐ 2. Add a z-score metric seen in Figure 3.1.

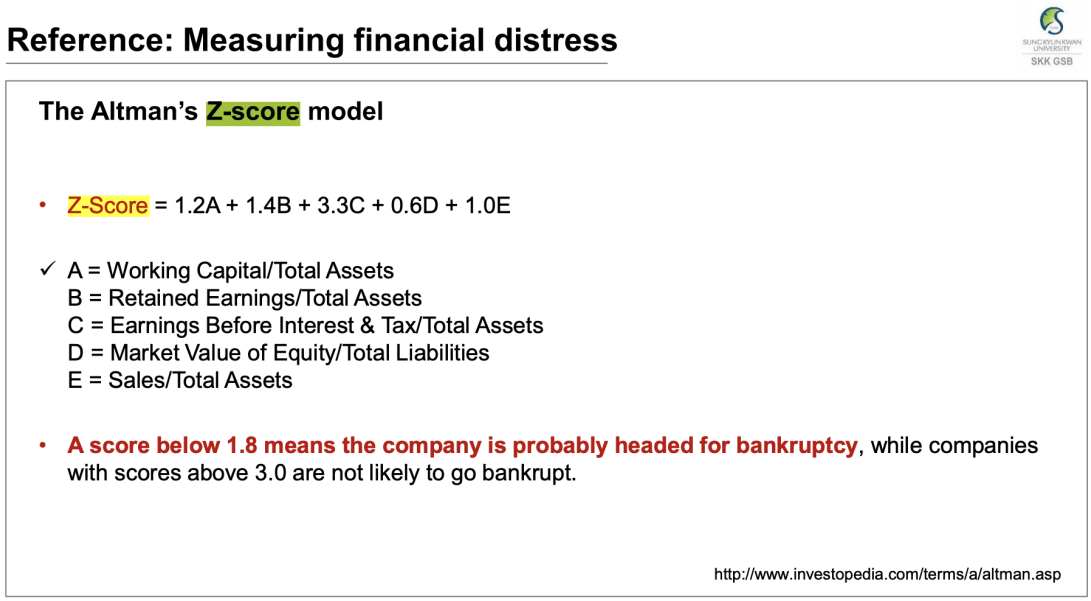


Figure 3.1: Z-score reference

☐ 3. Make a empty `finratios.py` script file inside the same folder using bash command.
☐ 4. Re-write the notebook code into reusable functions inside `finratios.py`

- 'fetch_data_local'
- `fetch_data_local_single_ticker`
- `calculate_metrics`
- `plot_trend`

Good practices:

☐ Each function should do one clear thing (e.g., load data, compute metrics, or plot)

☐ When writing functions, **only use variables that come from the input parameters**. If you need to use something inside the function, then make them input parameters.

> ⚠ Be Careful with Function Inputs
>
> Don't rely on variables that aren't passed into your function.
> If your function can't run using only its inputs, it's not truly reusable.
> **Do this:**
>
> ```
> def compute_margin(revenue, cost):
>     return (revenue - cost) / revenue
> ```
>
> **Not this**:
>
> ```
> total_revenue = 1000
> def compute_margin(cost):
>     return (total_revenue - cost) / total_revenue
>     # works in notebook, breaks in module
> ```

☐ Decide on return values.

☐ Add a `if __name__ == "__main__":` for testing at the bottom of `finratios.py`, such as:

```
if __name__ == "__main__":
    # Simple test cases
    print(compute_pe_ratio(1000000, 50000, 30))
```

3.4 Learn a bit about Streamlit

> 💡 Tip
>
> You don't know what Streamlit can do. Streamlit doesn't know what you want.
> But, we've gotta start *somewhere*.
> So let's learn a bit. Once you know **what it *can* do**,
> you'll finally know **what you *can make it do*** (with AI's help).

> 💡 How learn to use a new Python package?
>
> - What is Streamlit, and where is its API reference and official documentation?
> - What can Streamlit do — what kinds of apps or problems is it best at solving?

- What small examples or experiments can I build to quickly discover and learn its core features?

- **Start with the official docs and examples** — skim the Streamlit docs homepage and gallery to get a mental map of what's possible before diving into code.
- **Learn by doing small experiments** — build micro-apps (e.g., one with a button, one with a chart) to turn reading into muscle memory.
  - E.g., create `app1.py`, `app2.py` files each to test and run some micro streamlit apps.
- **Read code, not just tutorials** — explore community demos or open-source Streamlit apps to see how others structure layouts, manage state, and handle inputs.
- **Iterate with feedback loops** — run streamlit run `app.py` often and tweak one thing at a time; immediate visual feedback accelerates understanding.
- **Reflect and generalize** — after each mini-project, note what patterns repeat (e.g., sidebar widgets, caching, layout control) to build knowledge for future tools.

3.5 Build a Streamlit app

Here is an example. Try to build a dashboard displaying the financial metrics interactively.

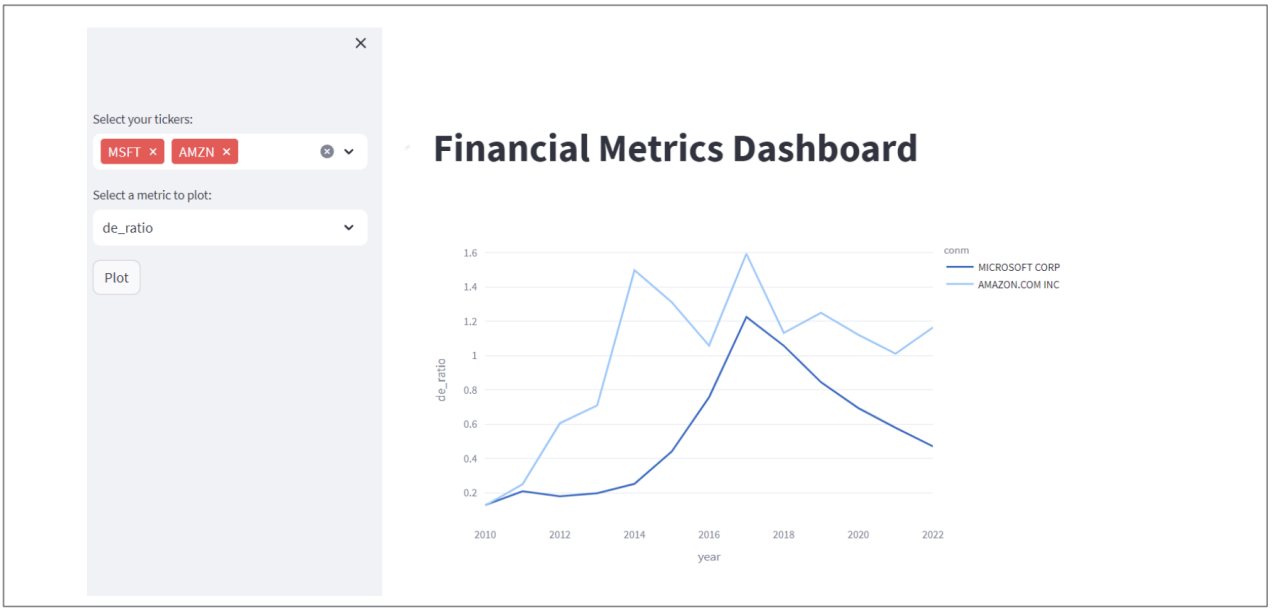


Figure 3.2: An Streamlit Example

3.6 Bonus: Add an AI Feature

Integrate the Google Gemini API to enhance your app with AI-powered capabilities.

- **Explore the API**
  Learn how to use the Google Gemini API and follow the setup guide to install the SDK.
- **Get an API Key**
  You can obtain a free API key using your Google account.
- **Keep It Secure**
  Never share or publish your API key on GitHub or any public platform — treat it like a password. Delete it after use.

References

1. Basic Command Line Crash Course

- Command Line Crash Course - Learn Python the Hard Way, Appendix A

Appendix A: Command Line Crash Course

This appendix is a quick super fast course in using the command line. It is intended to be done rapidly in about a day or two, and not meant to teach you advanced shell usage.