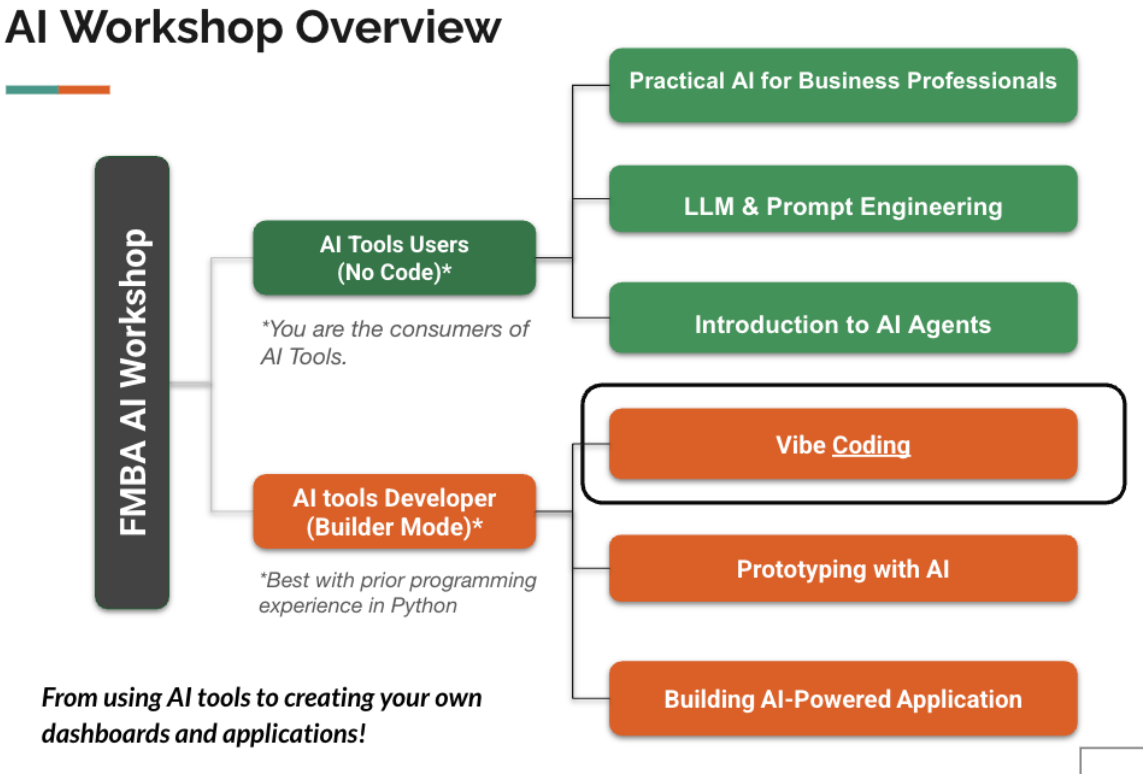
# FMBA AI Workshop Jan 2026

Yuan Tian

2026-01-15

# Table of contents

# Welcome




## Upcoming Sessions

The first 3 sessions on **AI Tools Users** have been completed in December 2025.

Below are the upcoming sessions:

- **Session 4: Vibe Coding** — January 16
- **Session 5: Prototyping with AI** — January 23
- **Session 6: Building AI Powered Applications** — January 30

> ❗ Important
>
> **You must have a GitHub account to participate in the upcoming sessions effectively.**

Please do not cite or distribute without author's permission.

# 1 AI Tools for Coding

## 1.1 AI Tools for Text, Voice and Image Generation

AI tools for text, voice, and image generation have become widespread.

We covered many of them in the first 3 AI workshops.

**Text Generation:** - ChatGPT - Claude - Microsoft Copilot - Google Gemini

**Image Generation:** - DALL-E - Midjourney

**Voice & Synthesis:** - NotebookLM – synthesizes and transforms information into podcasts and voice summaries

## 1.2 AI Tools for Code Generation

However, the world of coding requires a specialized set of tools designed specifically to understand, generate, and debug code for developers, programmers, engineers, etc.

- **GitHub Copilot** – AI pair programmer integrated into IDEs (e.g., VS Code) for real-time code suggestions and completion
- **Claude Code** – Coding-focused AI that analyzes and edits code across entire projects
- **Gemini Code / Gemini CLI** – Google's AI coding agent designed to work directly in the terminal
- **OpenAI Codex** – AI engine that writes, runs, and tests code in controlled development environments
- **Cursor** – AI-powered code editor optimized for UI and end-to-end code generation workflows

Many coding AI tools are accessed through CLIs (command-line interfaces), APIs (application programming interfaces) and professinal development environments, which require basic coding and terminal skills to use.

# 2 What is Vibe Coding

"Vibe coding" is an AI-assisted software development technique introduced by Andrej Karpathy (a co-founder of OpenAI) in February 2025.

The term quickly gained recognition among software developers, AI practitioners, and tech media.

**Reference Reading**

- Wikipedia: Vibe coding
- Google Cloud: What is Vibe Coding?

## 2.1 How It Works

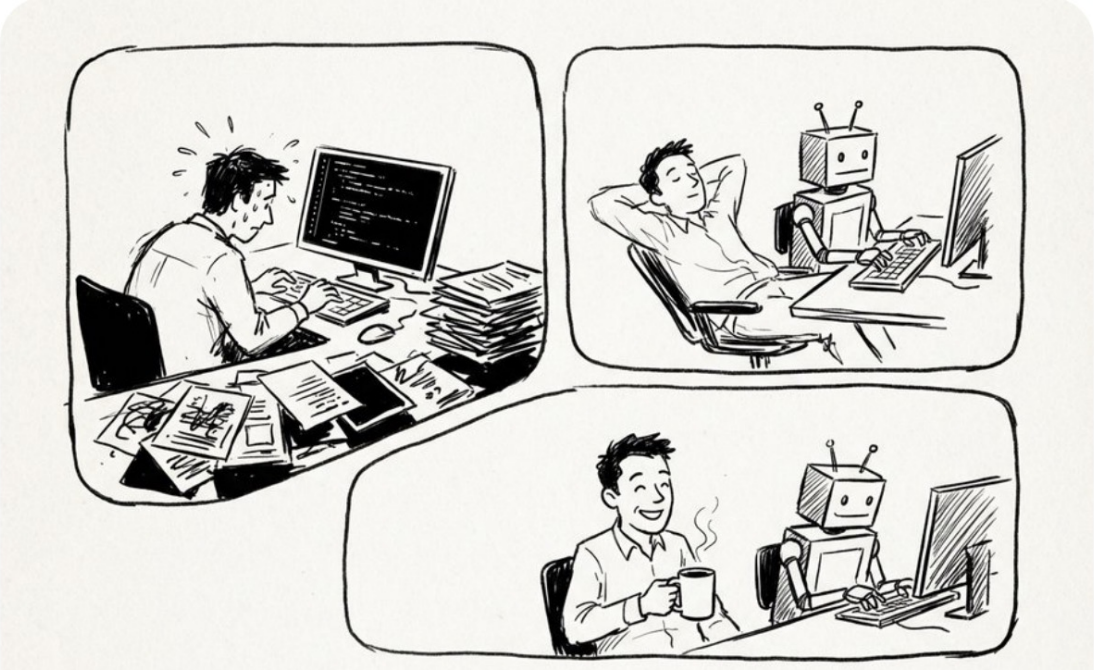
**Traditional Coding:** A human types: `print("Hello World")`.

**AI-Assisted Coding:** A human types half a line, and the AI suggests the rest (auto completion).

**Vibe Coding:** It is a software creation approach where developers describe tasks to a large language model, rely on it to generate code, and evaluate results through execution rather than reviewing or editing the code.

## 2.2 Vibe Coding: An Example

- A human says, "I want an web app that tracks our Q3 sales goals and sends a Slack alert if we drop below 80%."
- The AI then autonomously writes the code, sets up the server, connects the database, and fixes its own bugs.
- The human only need to know how to run the code to test whether it is working as expected.
- If not, the human keeps asking for revisions in natural language instead of inspecting or modifying the code itself.



## 2.3 Is Coding Skills Required?

> **!** Coding Skills Is Still Required
>
> Vibe coding lowers the barrier, but it does not remove the need for technical literacy. Vibe coding reduces how much code you write, not how much responsibility you have.

**Why This Matters:**

- **You must run and test the code** — real apps don't run inside chat; they run in terminals, servers, or browsers
- **You need to validate outcomes** — knowing what "working" looks like requires understanding logs, errors, and outputs
- **AI-generated code can fail silently** — without basic coding knowledge, you won't know whether results are correct or misleading
- **Debugging still happens** — even if AI writes the code, humans must recognize when something breaks and why
- **Security and data risks remain** — you need enough literacy to spot unsafe behaviors (e.g., exposed keys, wrong data access)

## 2.4 Minimal Technical Skills Required for Vibe Coding

- **Run code in professional environments**
  - Use VS Code, GitHub Codespaces, and cloud runtimes
  - Understand how to start, stop, and rerun applications
- **Basic operating system & terminal literacy**
  - Navigate files and folders (Linux basics)
  - Use terminal commands
- **Understand core code artifacts**
  - What is a python script file?
  - Know the difference between .py files and Jupyter notebooks
- **Clear product thinking**
  - Have a concrete idea or prototype in mind before prompting
  - Break ideas into steps the AI can execute
- **Version control with Git (your regret medicine)**
  - Save working versions; roll back when the AI makes mistakes

# 3 Labs: Minimal Technical Skills Required

## 3.1 Lab: GitHub Codespaces

### 3.1.1 Objective

- Use GitHub Codespaces, VS Code, and cloud runtimes.
- Understanding Terminal, Command Line Interface (CLI), and user interface.

### 3.1.2 Task

**Task: Get Started with GitHub Codespaces**

Follow the steps in the document below to learn GitHub Codespaces basics and get comfortable with the professional development environment.

☐ Step-by-step lab: Get Started With GitHub Codespaces

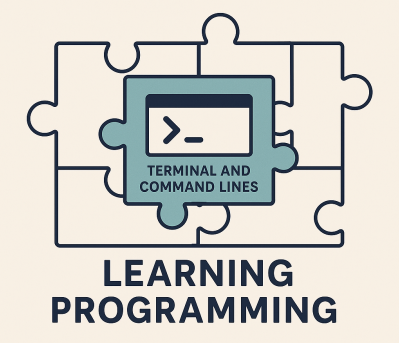
**After completing the lab, answer these questions**:

☐ What is a GitHub Codespace and how does it differ from working on your local machine?
☐ How do you access the terminal in a Codespace?
☐ Where can you find the GitHub Copilot (coding assistant) in Codespaces?

## 3.2 Lab: Basic Operating System & Terminal Literacy

- Navigate files and folders (Linux basics)
- Use terminal commands to start, stop, and rerun applications

### 3.2.1 Command Line and Terminal




Before learning a full programming language like Python, many computer science students first encounter the **terminal and command line**. In practice, the command line acts as a student's *first scripting language*: it teaches how to run programs, navigate files, pass arguments, and control how code executes.

While modern notebook environments (e.g., Google Colab or Jupter Notebooks) make it easy to start Python coding without setup, they often hide this foundational layer. As a result, students may write Python without ever learning how real programs are stored, executed, and managed using `.py` files and the terminal—skills that are essential for real-world development, 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

### 3.2.2 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), and Microsoft Azure 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)

### 3.2.3 What Operating System does GitHub Codespaces use?

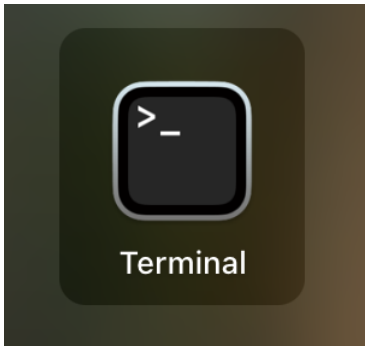
Open the Terminal inside your GitHub Codespace (View → Terminal) and check the OS with the commands below.

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

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

You should see an Ubuntu-based Linux release because GitHub Codespaces runs inside a Linux container.

### 3.2.4 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.

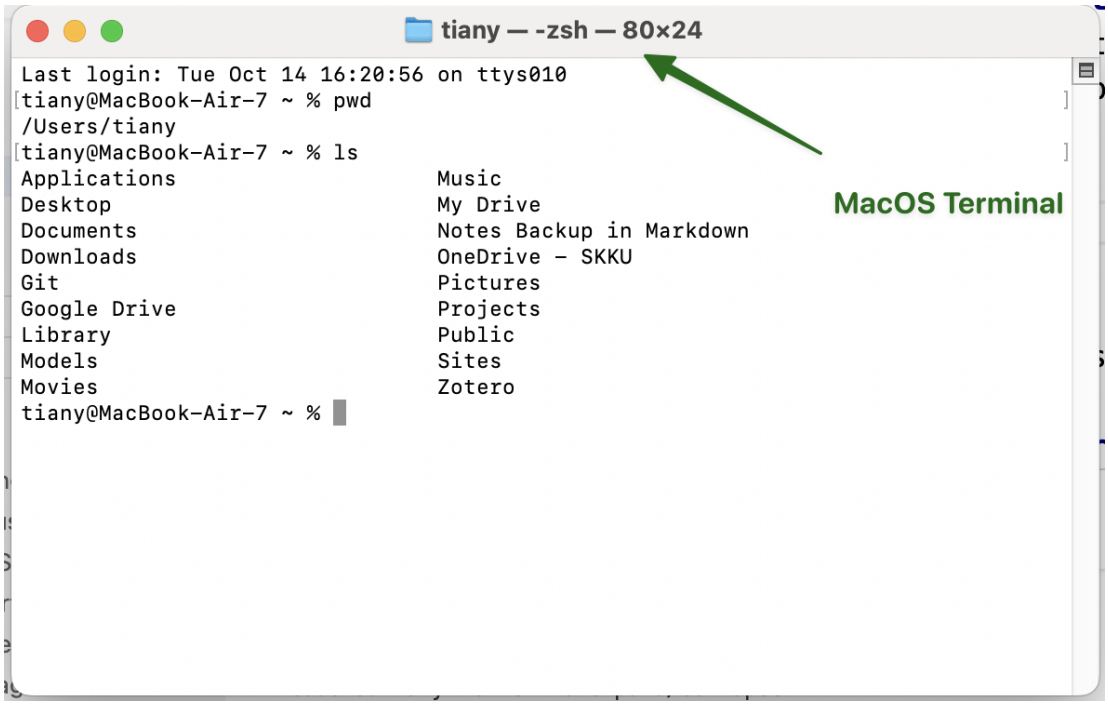
Before graphical interfaces (with windows, icons, and a mouse) were invented, the **terminal was the primary way users operated computers** — to run programs, manage files, and control hardware.

**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
- **macOS**: **Zsh** in Terminal app (based on Unix) is the default terminal in macOS.

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 GitHub Codespaces or on Linux servers.

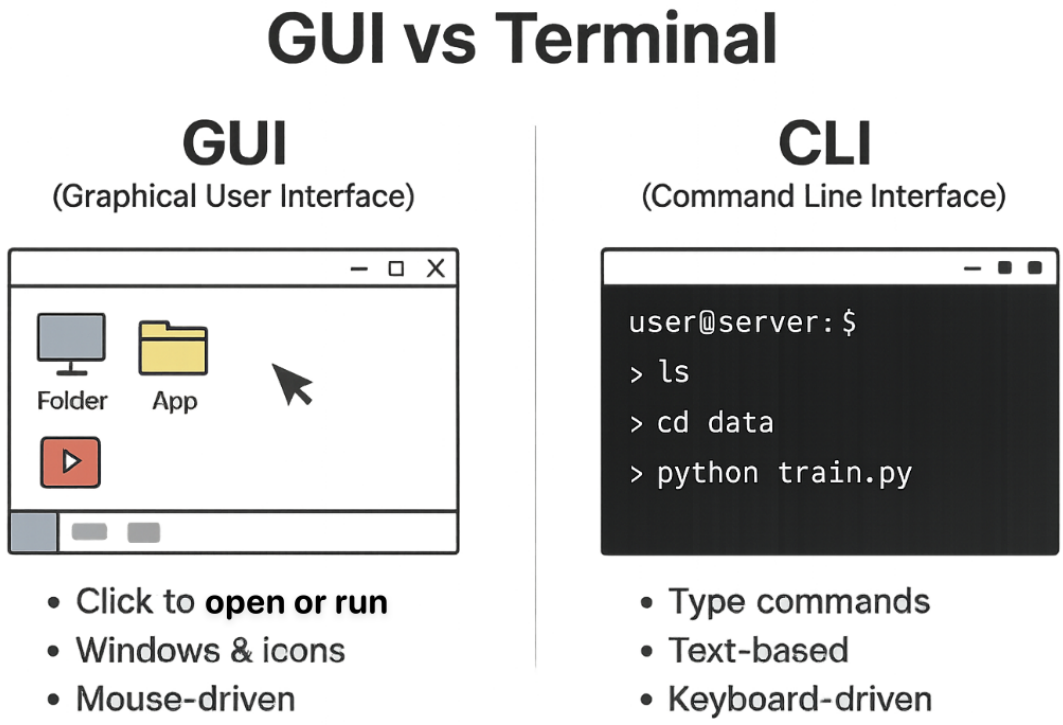
### 3.2.5 Why Learn Bash commands and Terminal?

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**.

### 3.2.6 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

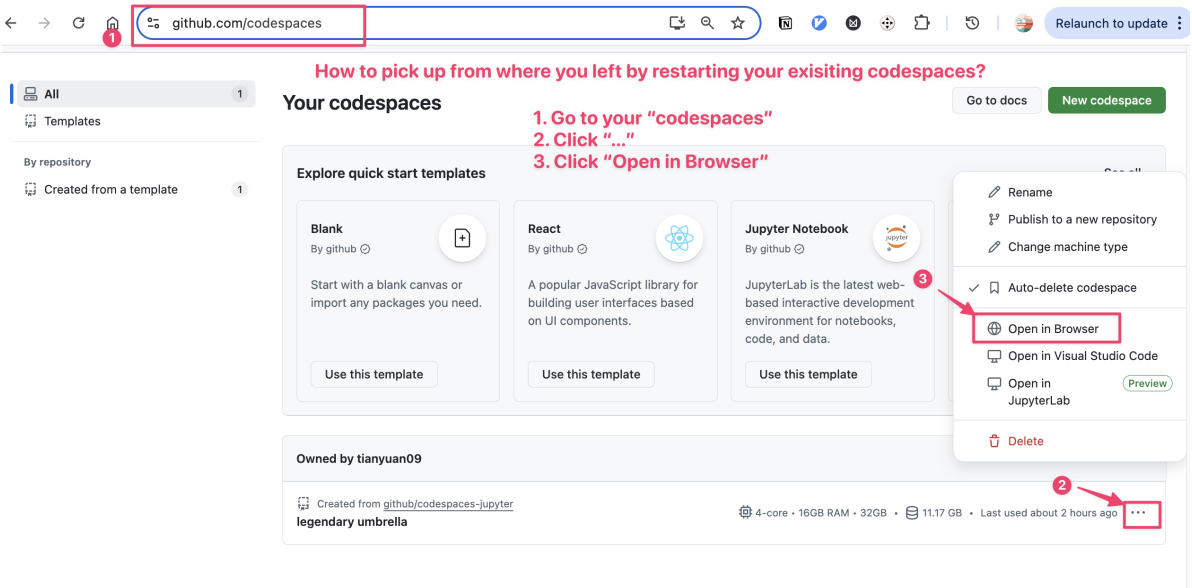
- The **Terminal** gives you access to the **CLI**, while the **Desktop Environment** provides a **GUI**.
- Both let you control the same computer — one through text, the other through graphics.

### 3.2.7 Learning Bash commands in GitHub Codespaces

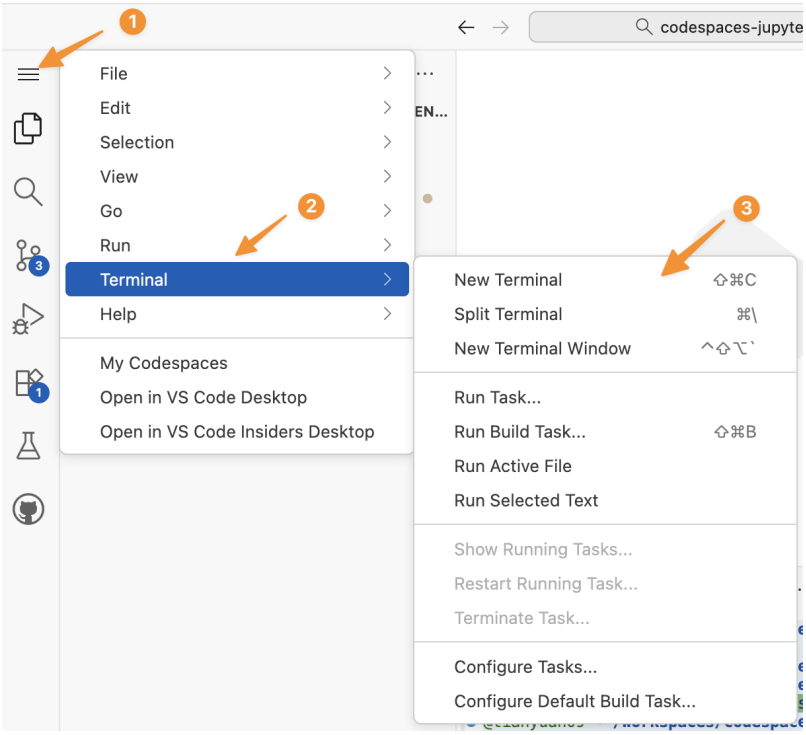
Mastering Bash is essential. It enables you to write scripts, manage jobs, and execute commands directly on compute servers — a critical skill when working with **large datasets** or **LLM pipelines**.

### 3.2.8 Bash in VS Codespaces

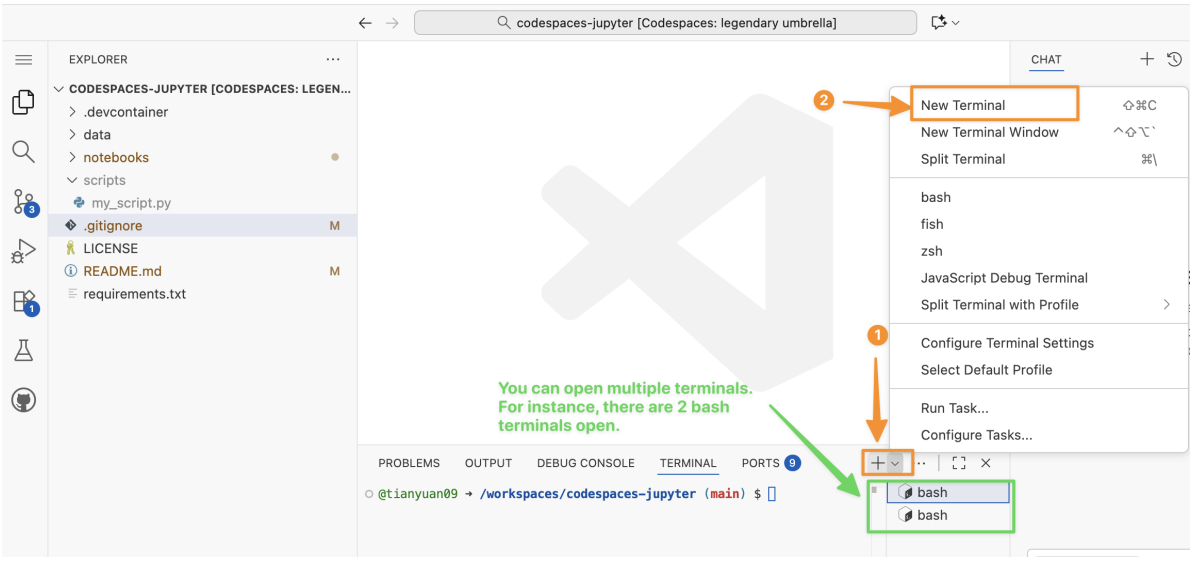
Restart your GitHub Codespaces.




Make sure you 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.




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

- Navigate and manage files
- Run **Python (.py) scripts** directly from the command line
- Work efficiently within **server-based** or **local terminal** environments

Use your GitHub Codespace terminal in VS Code to practice these commands. If you close the terminal, reopen it via **View → Terminal** or the `Ctrl+`` shortcut.

### 3.2.9 Lab: Linux and bash

- Display info about the operating system.

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

- Display the Linux kernel version and build info.

```
cat /proc/version
```

### 3.2.10 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
#/workspaces/codespaces-jupyter
```

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.

### 3.2.11 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)
```

### 3.2.12 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
```

### 3.2.13 Lab: Change Directory (`cd`)

- `cd data`: go the `data` folder under the current directory (create it first if it doesn't exist).
- `cd ..`: go the parent folder.
- `cd ~`: go to the home folder. In Codespaces, the home folder is `/home/codespace`. 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 a data folder under your repo root
ls
cd data
ls
# see the "altantis.csv"

# Move up one folder (to the parent directory /workspaces/codespaces-jupyter)
cd ..

# Go back to your "home" folder (/home/codespace in Codespaces)
cd ~
pwd

# To-do: find a way to go back to the: workspaces/codespaces-jupyter.
# If you failed, simply start a new terminal.
```

### 3.2.14 Lab: Make A Directory (`mkdir`)

```
# From your repo root, create a new folder named "data"
cd /workspaces/codespaces-jupyter
mkdir mydata

# Make multiple folders at once
mkdir project results logs

# Check that they were created
ls
```

### 3.2.15 Lab: Clear the Screen (`clear`)

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

### 3.2.16 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
```

### 3.2.17 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
```

### 3.2.18 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
```

### 3.2.19 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
```

### 3.2.20 Lab: Stream a File (cat)

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

# To-do: Display the README.md file in your repository:
```

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

### 3.2.21 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
```

### 3.2.22 Lab: Exiting Your Terminal (`exit`)

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

### 3.2.23 Summary Table – Common Bash Commands

| Command | Purpose | Example |
|---|---|---|
| `pwd` | Print working directory | `pwd` |
| `ls` | List files and folders | `ls -la` |
| `cd` | Change directory | `cd /workspaces/codespaces-jupyter` |
| `mkdir` | Make a new directory | `mkdir data` |
| `rmdir` | Remove an empty directory | `rmdir temp_folder` |
| `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` |

| Command | Purpose | Example |
|---|---|---|
| `exit` | Exit the terminal | `exit` |

### 3.2.24 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.

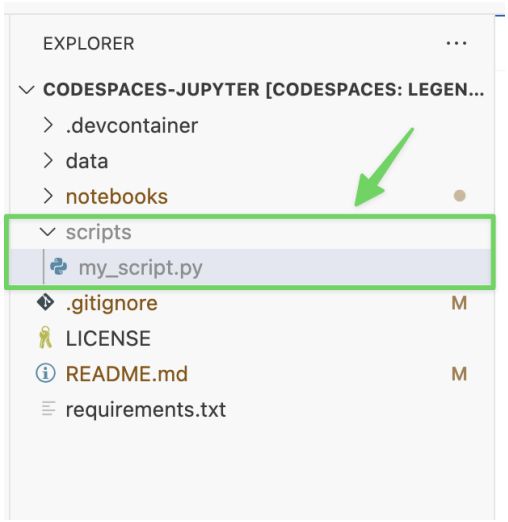
> **Tip**
>
> You may find the commands `pwd`, `mkdir`, `ls`, `cd`, and `touch` helpful for completing this exercise.

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
README.md  _quarto.yml  bash.qmd  chapter0.qmd  chapter1.qmd  chapter2.qmd  chapter3.qmd  c
@tianyuan09  /workspaces/codespaces-jupyter (main) $ cd scripts
@tianyuan09  /workspaces/codespaces-jupyter/scripts (main) $ ls
@tianyuan09  /workspaces/codespaces-jupyter/scripts (main) $ touch my_script.py
```

## 3.3 Lab: Python Script Files Basics

Objectives

- What is a python script file?
- Know the difference between `.py` files and Jupyter notebooks `.ipynb`.

Python is a high-level, general-purpose programming language used for data science, AI/ML, automation, and software development.

- Two common Python file types you will use:
  - **`.py` script files** — plain-text Python source you **run from the terminal** (e.g., `python hello.py`) or import as modules.

– **`.ipynb` Jupyter notebooks** — interactive notebooks that mix code, text, and visuals; executed cell by cell inside VS Code or Jupyter.

### 3.3.1 `.ipynb` Notebook vs `.py` Script Files

Figure 3.1 illustrates the difference between running Python code in `.ipynb` notebook versus in '.py' script file.

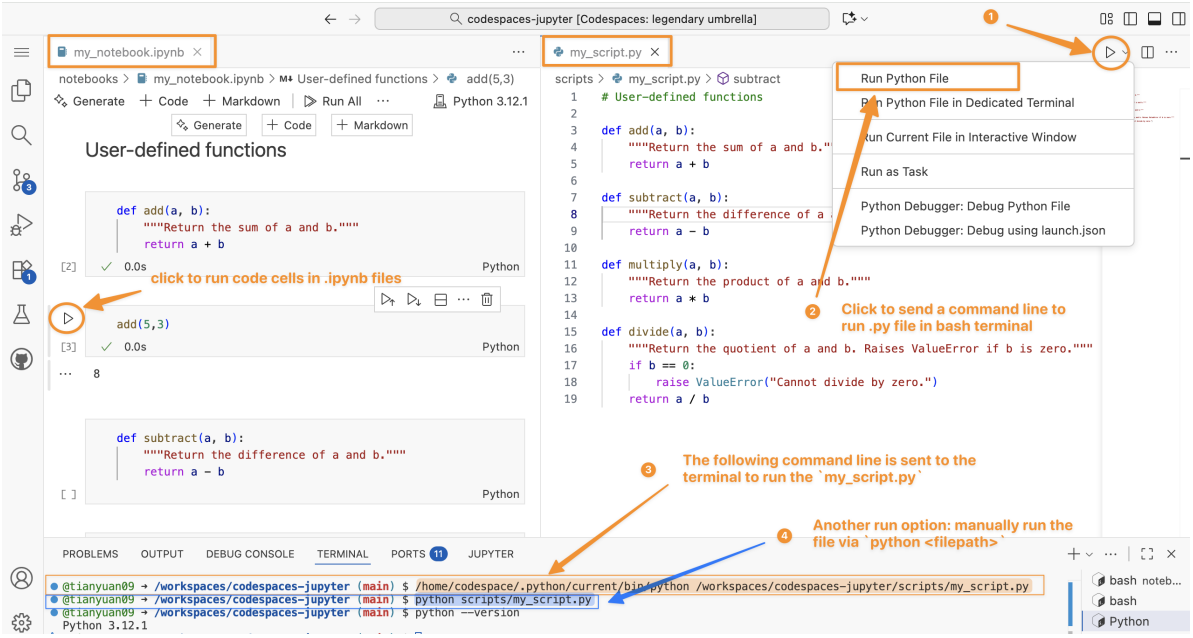



Figure 3.1: Comparison of Jupyter Notebook and Python Script formats

#### Understanding the Terminal

See directory structure in codespaces in the previous chapter.

In the terminal, it always starts with:

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

```
@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 `#`. |
| **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 3.1. |
| **Use Case** | Ideal for data analysis, visualization, and teaching due to its interactive nature. | Better for automation, **deployment** of production-ready code. |

### 3.3.2 Lab: Create and run a `.py` script file

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

```python
# 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 3.2.

**Expected output:**

```
Hello from Python!
This script is running from the terminal.
Current time: 2026-01-13 10:30:45.123456
```

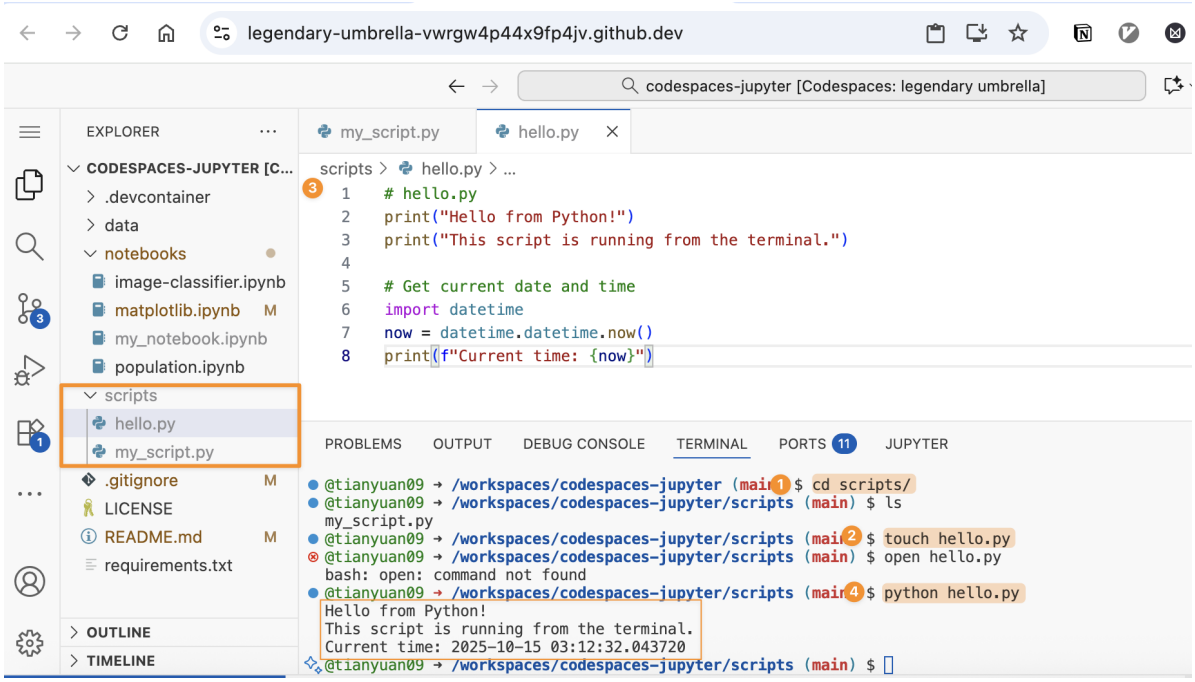



Figure 3.2: Create and run your 1st script file

---

### 3.3.3 Lab: Checking Your Python Version

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

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

### 3.3.4 Lab: 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
```

## 3.4 Running Scripts in Different Directories.

### 3.4.1 Absolute Paths

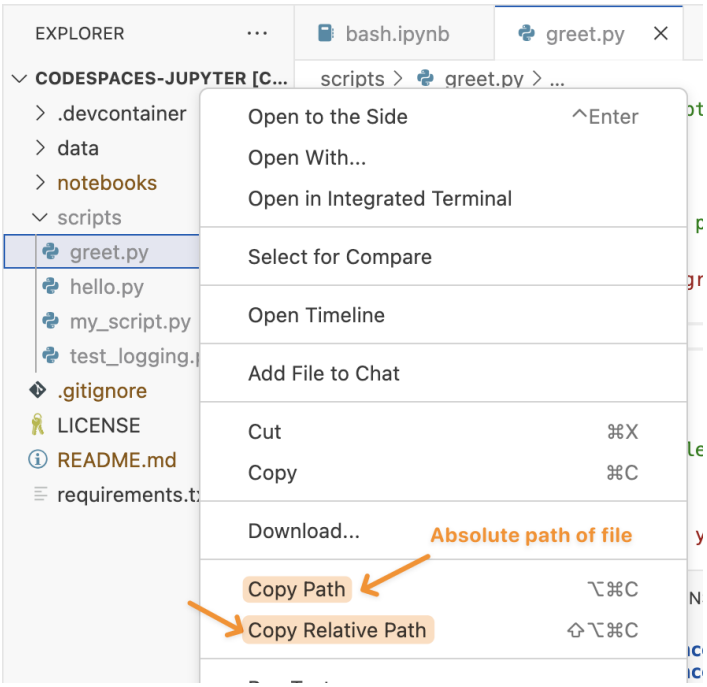
```
# Run script from anywhere using absolute path
python /Users/username/projects/my_script.py
```

`/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:




### 3.4.2 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`).

### 3.4.3 Difference in absolute vs relative paths

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

# 4 Prototyping an app with `streamlit`

## 4.1 Context and Goal

**You are going to prototype an interactive financial dashboard app using Vibe Coding.**

> **!** 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.

You are provided with the some preliminary analyses of financial data in Jupyter notebook. You are tasked to refactor it and **create an Streamlit web application** to present the data in an interactive way.

- ☐ Download the following files from the `downloads` folder in the https://github.com/tianyuan09/fmbaaiworkshop26spring.
  - `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.

## 4.2 The Vibe Coding Setup

You are going to use VS Code with Copilot in Github codespaces to "vibe code" a dashboard web application. Let's cover a few basics before you start.

- Each **Github Codespace** runs on a **Linux virtual machine** that already has tools like **Python, Jupyter Notebook support, various packages, and VS Code (via the web interface)** preinstalled. When you open a Codespace, you're actually using VS Code running in your browser, connected to that remote Linux machine. So:
  - **Codespaces = a Linux system in the cloud + Python + JupyterLab + various Python packages + VS Code (IDE interface)**.

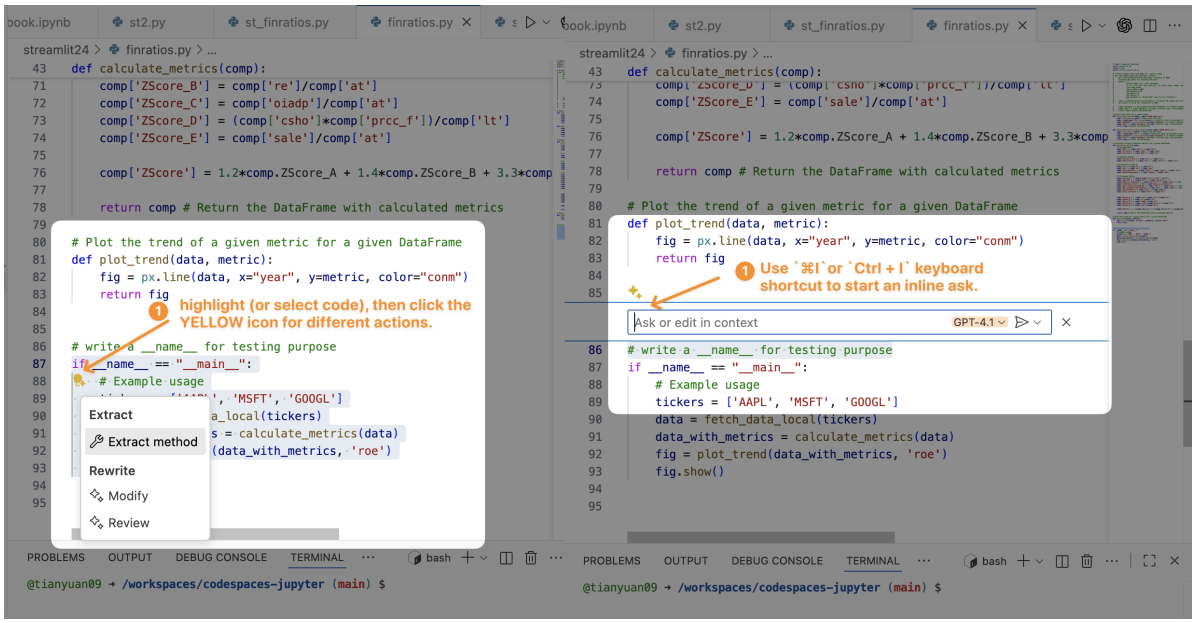
You also can set up the same development environment in your local PC. You will need to install multiple softwares and tools, such as Python, VS Code, Python extension in VS Code, and each python package that you might use (such as `pandas`).

| Environment | What it includes | Notes |
|---|---|---|
| **Codespaces** | Linux + Python + Jupyter + various packages + ***VS Code (web)*** | Cloud-hosted; nothing to install locally; runs VS code interface |
| **Google Colab** | Linux + Python + Jupyter + various packages | Cloud-hosted; focuses on notebook interface |
| **Local PC** | Whatever you install (Python, VS Code, Jupyter) | Runs directly on your machine |

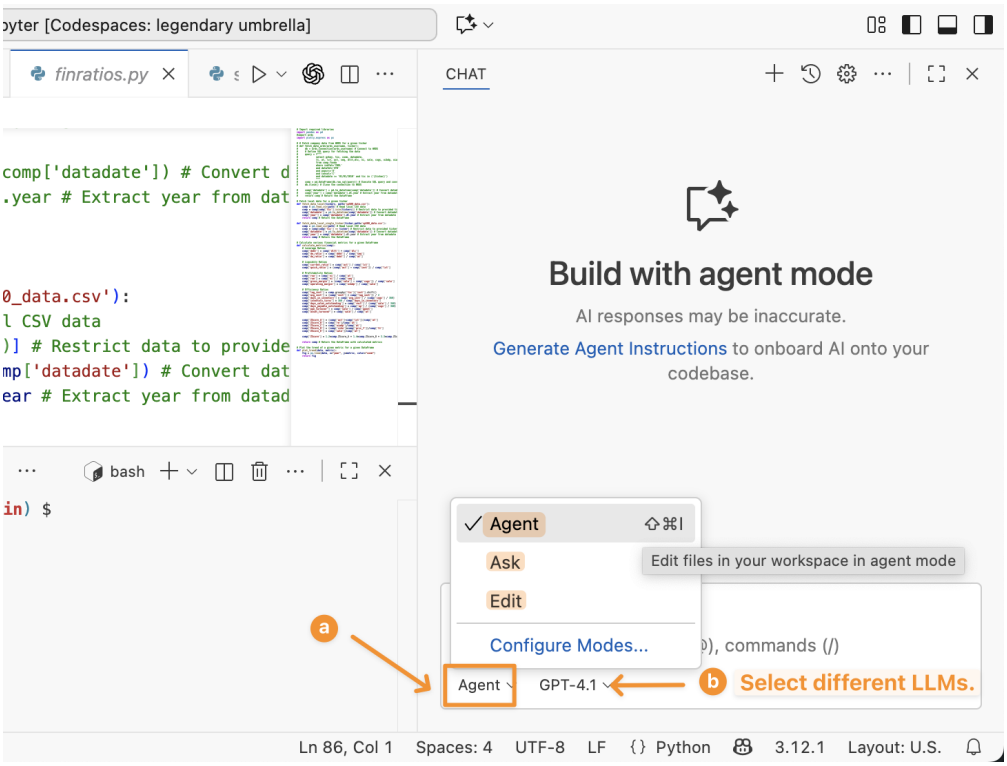
### 4.2.1 GitHub Copilot and Copilot Chat: Ask, Edit and Agent Modes

Let's introduce the features of GitHub Copilot and Copilot Chat. GitHub Copilot comes with four distant modes that you may use:

1. Inline Chat or suggestions: quick in-context code suggestions directly within VS Code using shortcuts `+I` or `Ctrl+I`.




2. **Ask Mode: best for Q&A**. Highlight some code, then ask Copilot questions about its logic, purpose, or braistorm ideas implementation.
3. **Edit Mode**: give you **inline, review-ready code edits** across the files.
4. Agent Mode: an **autonomous mode** where Copilot **analyzes context** (e.g., files in your workspace) and **performs tasks** based on your request.




### 4.2.2 Tips for Using the Copilot Agent Mode

To get the best results in Agent Mode, you can provide additional context or use special commands to guide Copilot about what you want it to do.

> **What is "Context"?**
>
> Context is the information you give Copilot so it understands **what you're working on and can respond more accurately**. In Agent Mode, **context can include the code you've selected**, other `.py` files, or any extra notes you write with `#`.

- Highlight code (e.g., line 57-59 of the `vscodecopilot.qmd` file) in the file will automatically add those lines of code in the context (see Figure 4.1)
- Add context with `#`. Add additional files that you want Copilot to read before completing the task.
- commands with `/`. E.g. `/explain` is a command asking for explanation of the code.

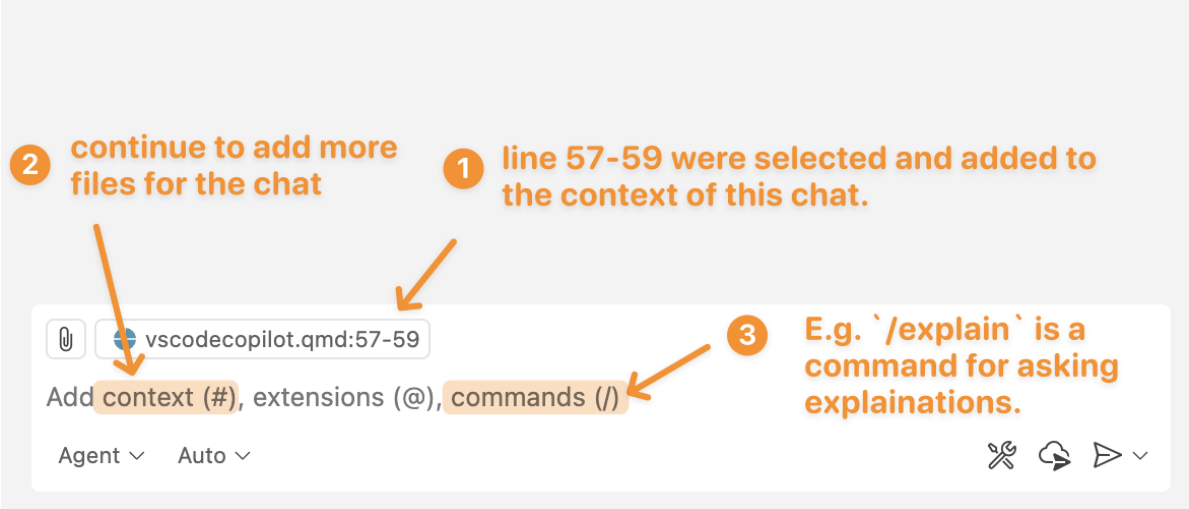



Figure 4.1: Highlight code, or add contexts or commands

### 4.2.3 Vibe-coding vs AI-assisted Coding

Using the **Agent Mode** will allow you to vibe code **without reading or understanding any of the code**. You relied on the AI to produce something that looked right without knowing how it actually worked.

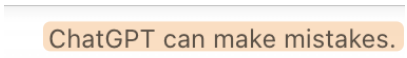
However, **if you read, wrote, or edited the code** while building the app, you were doing **AI-assisted coding** — working with the AI to shape logic, fix bugs, and design structure.

| Aspect | **Vibe Coding** | **AI-Assisted Coding** |
|---|---|---|
| **Definition** | A no-code workflow where you **don't read, write, or edit** code. You simply test the prototype or app to see if it meets your design or intent. | A coding workflow where you **interact with and modify code** using AI tools like Copilot's **Ask**, **Edit**, or **Inline Chat**. |
| **Goal** | Validate the **"vibe"** — check if the **prototype** looks, feels, and behaves as intended. | Build, refine, and ship **production-ready features** with AI support. |
| **User Interaction** | **No direct code manipulation** — focus is on results, not implementation. | **Actively generate, read, edit, debug, and review code** and suggestions with AI |

| Aspect | **Vibe Coding** | **AI-Assisted Coding** |
|---|---|---|
| **AI Tools** | GitHub Copilot **Agent Mode** | GitHub Copilot **Ask Mode, Edit Mode, Inline Chat** |
| **Mindset** | *"I don't care how it's built — does it look and work as intended?"* | *"I'll collaborate with AI to understand, fix, or enhance the code."* |
| **Best For** | Rapid prototyping, early design validation | Full-cycle software or data product development: feature implementation, optimization, and maintenance. |
| **Limitations**. | Ignores the complexity, lacks of consideration over performance, scalability, and maintainability. | Supports real data engineering work but still requires developer understanding and validation |

Both **Vibe-coding** and **AI-assisted coding** have their place in development. Use vibe-coding for quick validation and prototyping, and AI-assisted coding for building robust, production-ready data science solutions.

Although vibe-coding feels exciting, **you can't rely on it from prototype to production, as it often makes mistakes or produces "shit code"** — duplicated logic, buggy and insecure implementations, and tangled features no one dares to maintain. In data science, that might look like a notebook full of hard-coded file paths, random seeds, and global variables — impossible to reproduce or scale.




**Also, don't fool yourself into thinking you're learning to code while working in Agent Mode — you're NOT building the skills yourself.** You can't learn guitar just by watching someone else play, and you can't learn piano by watching performances. **Likewise, you can't truly learn programming just by watching an Agent write code for you**. The craft lies in understanding, experimenting, and making mistakes. That's how you **move from vibe-coding prototypes to engineered, production-ready solutions**.

While **vibe coding** can accelerate prototyping and help non-technical users **validate design ideas**, it often **overlooks many critical aspects** of data engineering — such as scalability, performance, maintainability, and security. Building reliable data science solution or product remains a complex, multi-layered process that requires thoughtful coding, testing, and collaboration between designers, developers, and AI tools.

## 4.3 Lab: Make it a `.py` file with functions

**Let's do some vibe coding!**

- ☐ 1. Open and take a look at the `Compute Financial Ratios Notebook.ipynb`
- ☐ 2. Add a z-score metric seen in Figure 4.2.

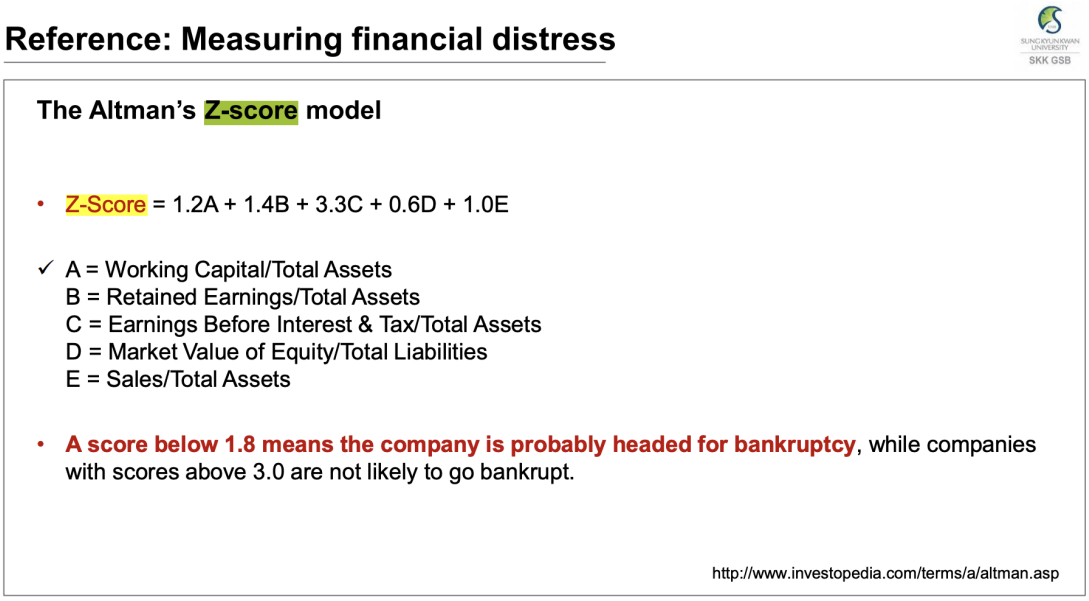



Figure 4.2: Z-score reference

- ☐ 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**:

You can copy and paste the good practice to Copilot when completing the tasks above.

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

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

## 4.4 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.

## 4.5 Prototype a Streamlit app

Here is an example.

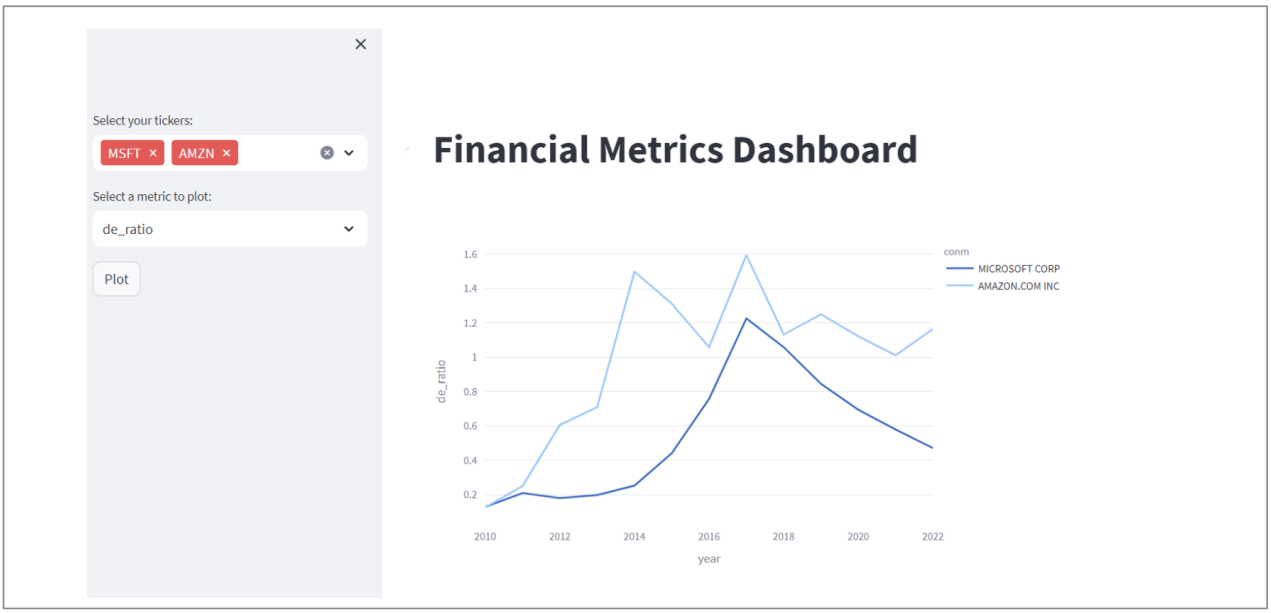



Figure 4.3: An Streamlit Example

☐ Try to build a dashboard displaying the financial metrics interactively.