Prerequisites & Environment Setup

DAIR^3

Jackson State U · U Michigan · UT San Antonio

Juan B. Gutiérrez, Ph.D. Professor of Mathematics
juan.gutierrez3@utsa.edu
The University of Texas at San Antonio
February 12, 2026

Contents

Chapter 1

Environment Setup

These instructions centered on Windows but adaptable for Mac and Linux. We begin with configuring environment variables to securely store API keys for OpenAI, Anthropic, and Groq. The reader is then guided through the installation of Python, emphasizing proper PATH configuration and optional tools like Visual Studio Code (VSC), Git, and Windows Terminal. Additional sections cover the setup of virtual environments both within and outside VSC, the installation of essential Python libraries, and optional tools such as LaTeX and C/C++ compilers. The setup culminates in verifying functionality by cloning a GitHub repository and running a helper script, ensuring the environment supports modular, reproducible project development with minimal dependency conflicts.

1.1 Required: Setting up environment variables for API keys

You will receive an API key for OpenAI and Anthropic during our first session. You can get your own API for free at Groq. You need to create three API key variables following the procedure described below: `OPENAI_API_KEY`, `ANTHROPIC_API_KEY`, `GROQ_API_KEY`. Test the program agentGroq.py, agentGPT.py and/or agentClaude.py from the Github repository described below in this document.

On Windows, execute the following three commands one at a time for each API key. The example for `OPENAI_API_KEY` is as follows:

```
setx OPENAI_API_KEY "[sk-... API key]"
Exit
echo %OPENAI_API_KEY%
```

In the code above, replace `[sk-... API key]` with your actual API key. The first command sets the environment variable permanently for your user account. The second command closes the Command Prompt to ensure the new environment variable is registered. The third command, run in a new Command Prompt window, prints the stored value to verify it was set correctly.

On Mac/Linux execute the following three commands one at a time for each API key. The example for `OPENAI_API_KEY` is as follows:

```
echo "export OPENAI_API_KEY='[sk-... API key]'" >> ~/.zshrc
source ~/.zshrc
echo $OPENAI_API_KEY
```

In the code above, replace `'[sk-... API key]'` with your actual API key.

As reported by Bryan Fowler, if you experience errors trying to add your environmental variables in a Mac/Linux, the most likely problem is ownership of the `~/.zshrc` file. In this case, execute

```
sudo chown $(whoami) ~/.zshrc
```

And try to create the environmental variables again.

As reported by Drew Stephen regarding Mac, "*apparently it doesn't work on the default c shell but switching to the zsh lets it run.*" You might need to close and reopen the terminal window from where you started for changes to become effective.

1.2 Required: Install Git

Install Git for Windows. For Mac, install Git for Mac. Linux also has a version available. In the second screen, select Visual Studio Code as your Git editor. Use all other default options.

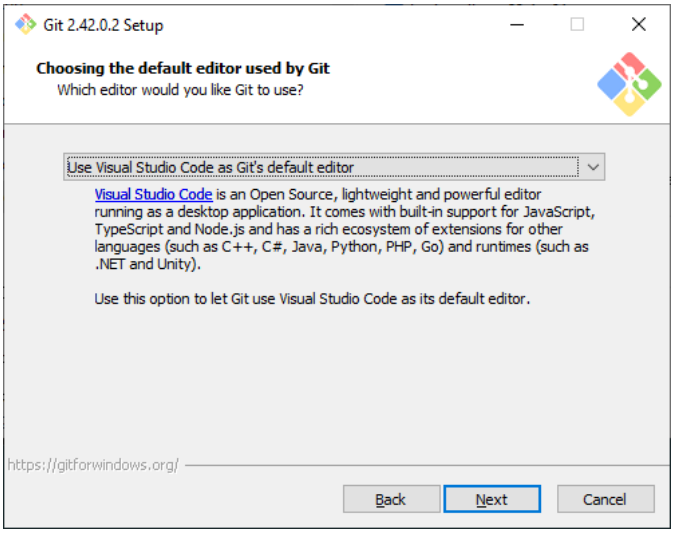

Figure 1.2-1: Git setup screen showing Visual Studio Code as the default editor.

1.3 Required: Open Git Bash (or any terminal)

The command line allows you to control your computer by typing instructions, providing precise access to files and system features. For example, typing `ls` on a Unix-like system such as Linux or macOS lists the files and directories in the current location, such as `ls /home/user/Documents` to display the contents of the Documents folder. On Windows, the `dir` command serves the same function, so `dir C:\Users\Alice\Desktop` shows what is on the Desktop. To move between directories, the `cd` command is used; `cd Downloads` changes the working directory to Downloads, while `cd ..` moves one level up. These commands allow efficient navigation and inspection of the file system through the terminal.

Now that you installed git, you should **go to the folder you want to work in** using command line, and run the following command:

```
git clone https://github.com/DiscursiveNetworks/FOO_QtPy.git
```

1.4 Required: Install Python

https://www.python.org/downloads/. Install any recent Python version. Go to your downloads and double click on the install file to start installation. By default the Python installer for Windows places its executables in the user's AppData directory, so that it doesn't require administrative permissions. This works for most scenarios. If you're the only user on the system, you might want to place Python in a higher-level directory (e.g. `C:\Python` or `/usr/local/bin`) to have a shorter path to the binaries (sometimes you will need that). Depending on your preferences, either select "*Install Now*" or "*Customized installation*" (my preference). Please make sure you select "*Add python.exe to path*"; this will save you a few headaches later.

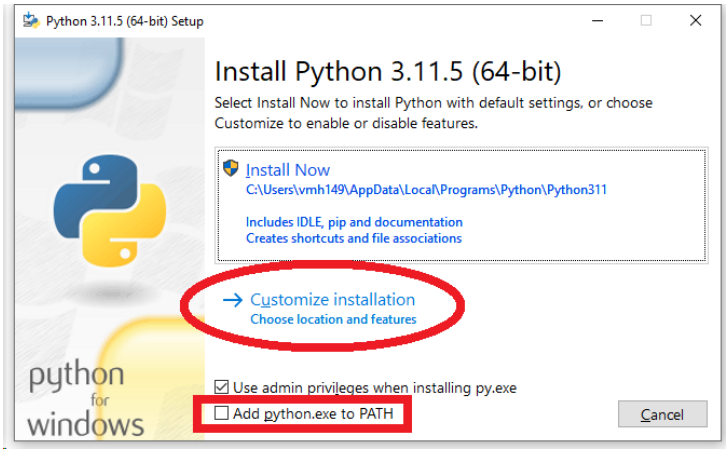

Figure 1.4-2: Python installer showing the "Add python.exe to PATH" checkbox at the bottom.

For PC: If you did not add `python.exe` to the PATH during installation, wait until the installation is complete. Then open File Explorer and right-click on "This PC." Select "Properties" from the context menu. In the System window

that opens, click on "Advanced system settings" in the left-hand sidebar. In the new window, ensure the "Advanced" tab is selected and click the "Environment Variables" button near the bottom. Under the "System variables" section, find and double-click on the variable named "Path." This will open a window where you can add the path to the Python executable manually.

For Mac: If you did not ensure Python was added to your system's PATH during installation, you can manually do so using the following steps on a Mac. After installation is complete, open the Terminal application. Enter the command `which python3` or `which python` to find the path to the installed Python binary. Copy that path. Then open your shell configuration file in a text editor—this will typically be `~/.zshrc` if you are using the Zsh shell (default on newer versions of macOS) or `~/.bash_profile` if you are using Bash. Add the line `export PATH="/path/to/python:$PATH"`, replacing `/path/to/python` with the path you copied. Save the file and close the editor. In the Terminal, run `source ~/.zshrc` or `source ~/.bash_profile` depending on the file you edited, so the new PATH is loaded into your environment.

In the window "Optional Features" select all features

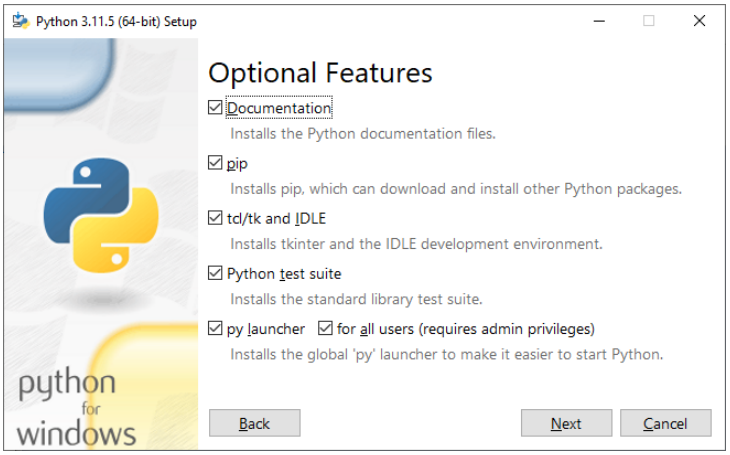

Figure 1.4-3: Python Optional Features screen—select all features.

Select "*Install Python X.XX for all users*" if you can and want. This requires administrative privileges. Select the folder of your choice for the install location.

1.5 Optional: C/C++ compilers

If you'd like to code in C or C++, you have several options, including GCC, the GNU Compiler Collection, but it requires some effort to install the prerequisites. From Microsoft there is Visual Studio Community Edition (VSCE), which encapsulates the complexity of installing multiple compilers (including the .NET framework SDK). I recommend you install VSCE with all "Workloads" in "Web & Cloud" and "Desktop & Mobile" (except Python).

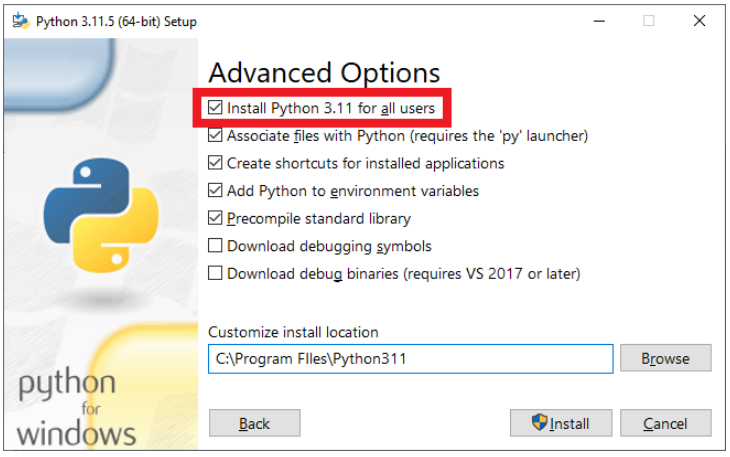

Figure 1.4-4: Python Advanced Options screen showing "Install for all users" selected.

1.6 Optional: LaTeX

If you intend to produce PDF documents using LaTeX, install MiKTeX.

1.7 Strongly recommended: Install Visual Studio Code (VSC)

If you are new to Python, install Visual Studio Code (VSC): There are many options to write Python code, and you could have a favorite one. I must pick one to deliver instruction, and experience has shown me that VSC offers the least amount of friction.

In VSC, there are two main options: "*User Installer*" and "*System Installer*". Use the User Installer if you want to install only for the current user. For all users, use System installer; I prefer this choice because it installs VSC in `C:\Program Files\Microsoft VS Code`.

This little giant has become a darling of coders for many good reasons. It is available for Mac, Linux and Windows. There is an open source identical alternative called VSCodium, available at https://vscodium.com/ I personally use VSCode.

1.8 If you installed VSC: Required extensions

Select the Extensions icon on the left and install the following Visual Studio Code libraries

1. **Required**: Python extension for Visual Studio Code.
2. **Optional**: Also install Jupyter (this also installs Pylance, Jupyter Keymap, Jupyter Notebook Renderers, Jupyter SlideShow). Pay attention to the publisher of the extension; use the ones by Microsoft.

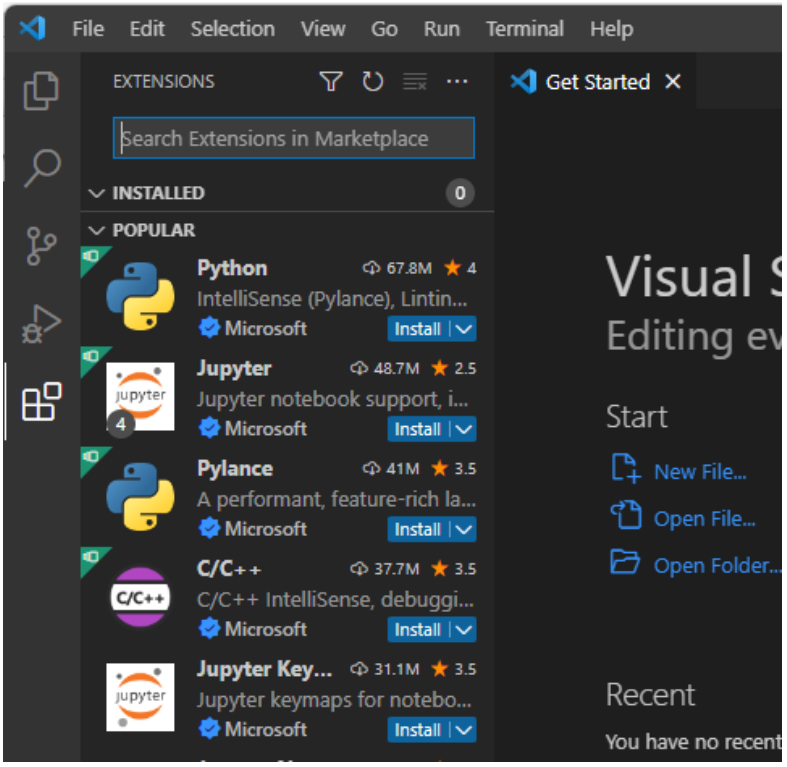

Figure 1.8-5: VS Code Extensions marketplace showing the Python and Jupyter extensions by Microsoft.

3. **Optional**: Install C/C++ for Visual Studio Code, C/C++ Themes & C/C++ Extension Pack.

4. **Optional**: Install the extension "**LaTeX Workshop**" by James Yu. You will never use another LaTeX editor :)

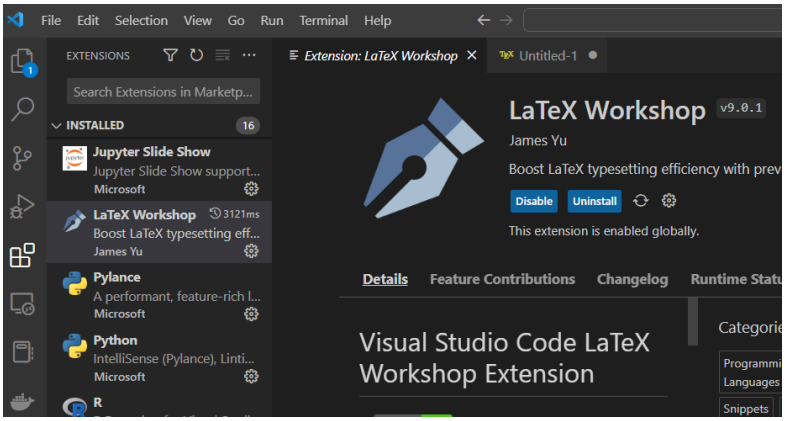

Figure 1.8-6: LaTeX Workshop extension page in VS Code.

1.9 Optional: Windows Terminal

For Windows users, install Windows Terminal. This Microsoft app allows you to use a multi document window with tabs for different command consoles like

PowerShell, Ubuntu, DOS, Git, etc. You will need this if you want to complete the steps related to Linux in the next section of this document.

1.10 If you installed VSC: Required

To start VSC, use the command line tool for your operating system and go to the folder CommandLineLLM. After Step 9, the command will be

```
cd CommandLineLLM
```

Inside this folder, execute the command

```
code .
```

Note the period after the command `code`; it instructs VSC to use the current folder as the working folder.

1.11 Partially Required (this or 1.12): Virtual environments inside VS Code

Create virtual environments *inside* Visual Studio. We will start by creating a virtual environment; the reason for this is that sometimes specific versions of different libraries might be incompatible. You should create an environment for each project you work on.

1. Open Visual Studio. Select the option *File* > *Open Folder*... Choose the folder you want to use for your project.
2. Once you select the folder, you will see the files related to the project you are working on in the Visual Studio File Explorer. Notice that the status bar changes color from purple to blue.
3. Now, hit the keys *SHIFT+CTRL+P*. This will allow you to type in the command palette the following instruction:

   ```
   Python: Create Environment
   ```

4. You will likely see two options: `venv` and `conda`. Choose `venv`.
5. Once you choose `venv`, you will have to pick a Python installation. Pick the one we just installed (this could be the only one you see).
6. Now, go to the menu *View* > *Terminal*.
7. You will see that the command prompt now has the word `(.venv)`. This means that the virtual environment has been installed and is now active. If it has not been activated, you can simply type `activate`. To inactivate the virtual environment, you simply need to type `deactivate`.

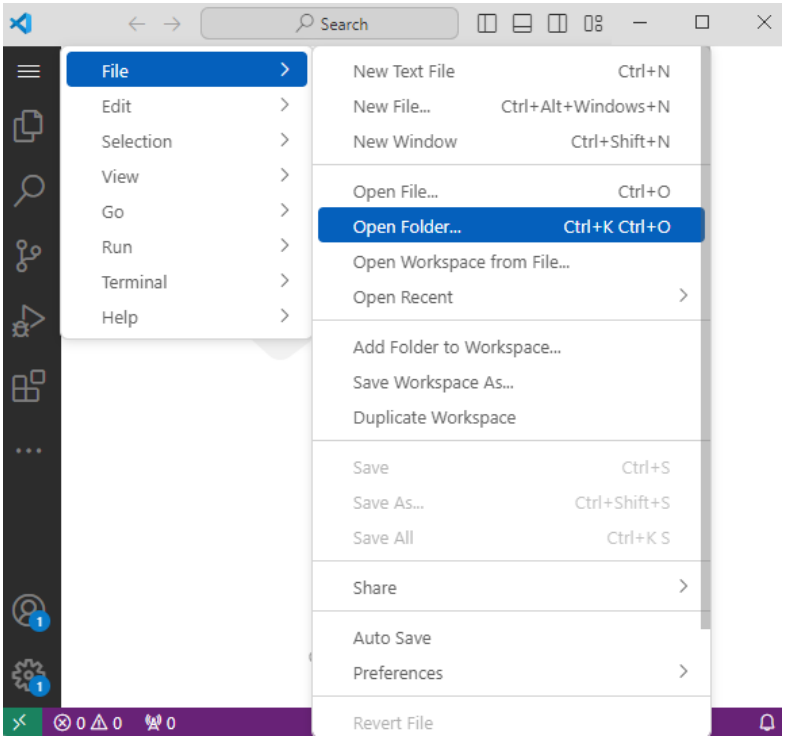

Figure 1.11-7: VS Code File menu showing Open Folder.

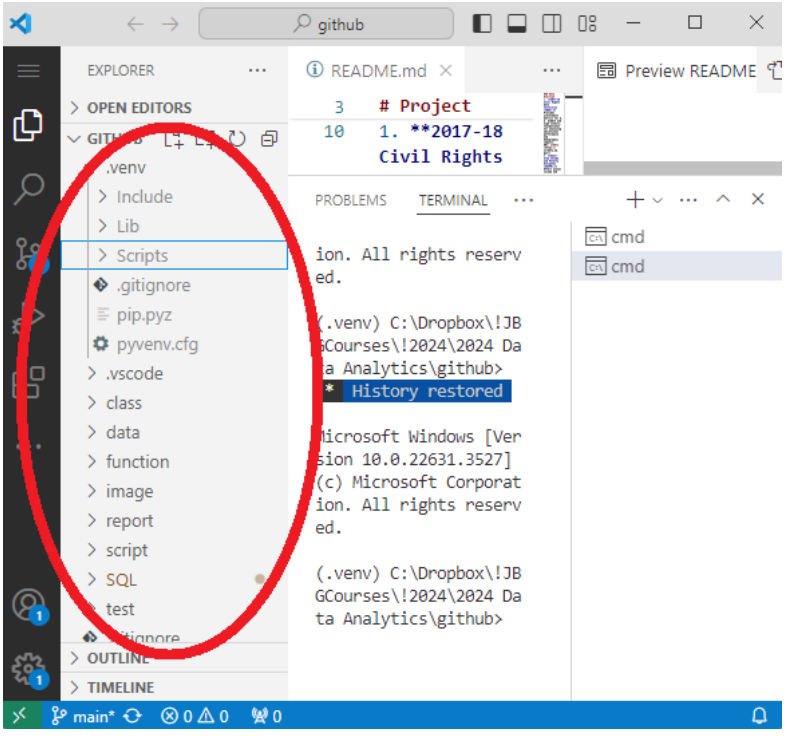

Figure 1.11-8: VS Code with a project open—notice the blue status bar.

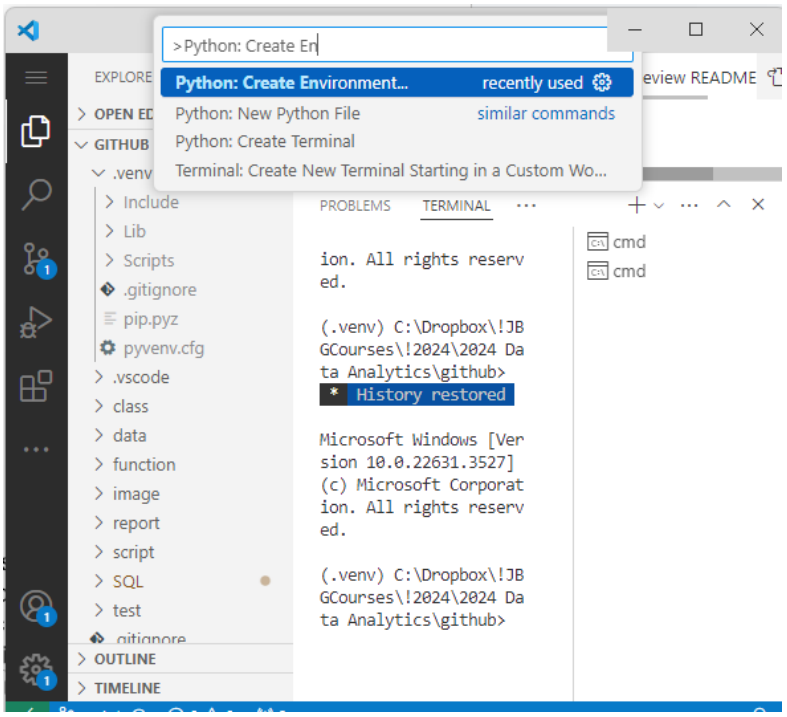

Figure 1.11-9: VS Code Command Palette with "Python: Create Environment" selected.

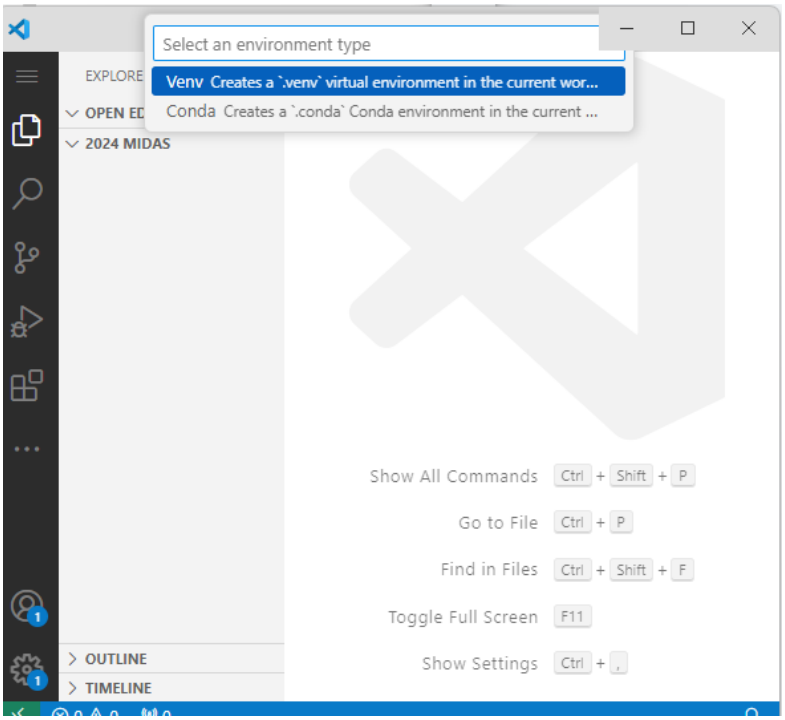

Figure 1.11-10: Selecting venv over conda in VS Code.

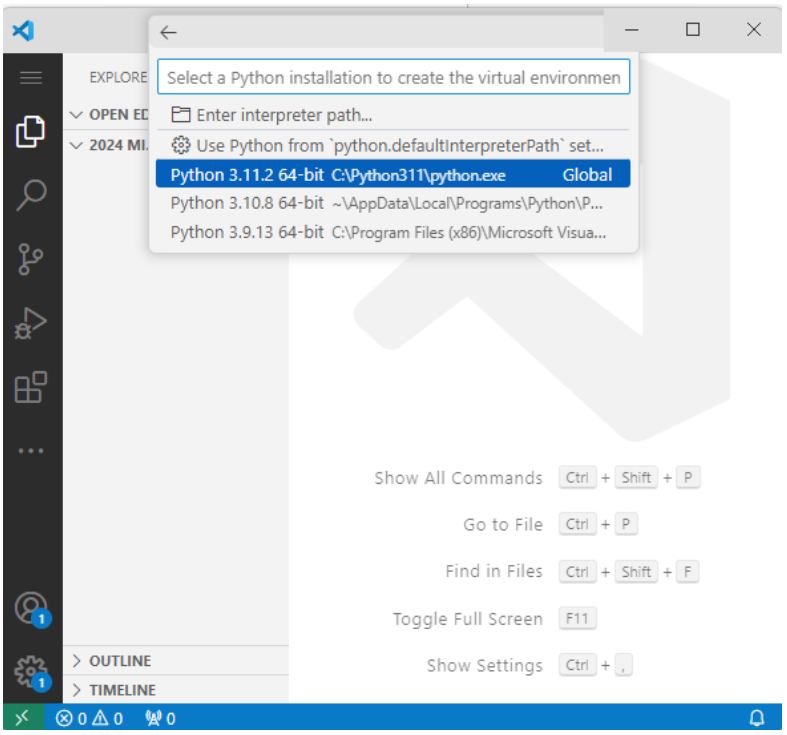

Figure 1.11-11: Selecting the Python installation for the virtual environment.

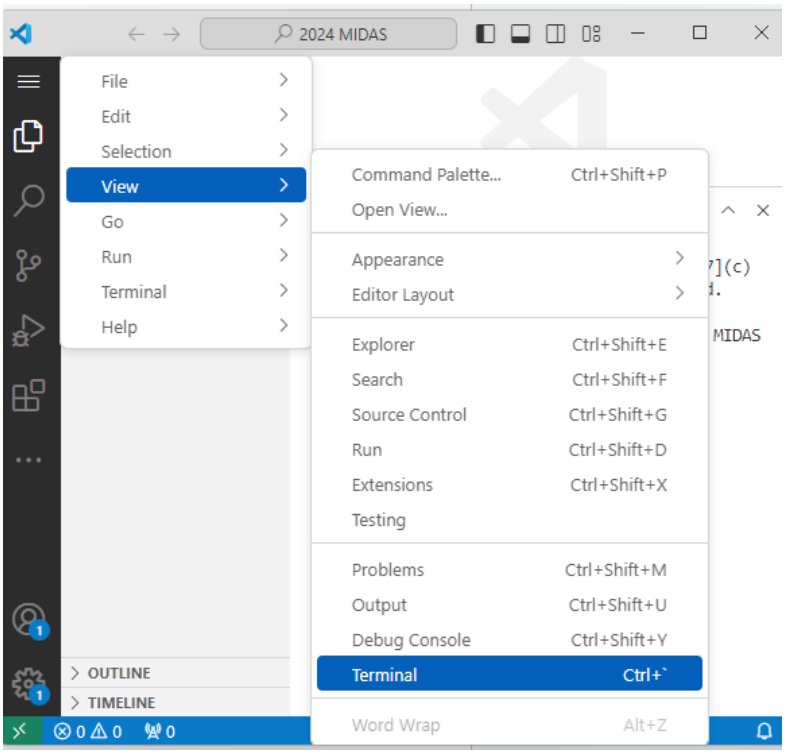

Figure 1.11-12: VS Code View menu showing Terminal option.

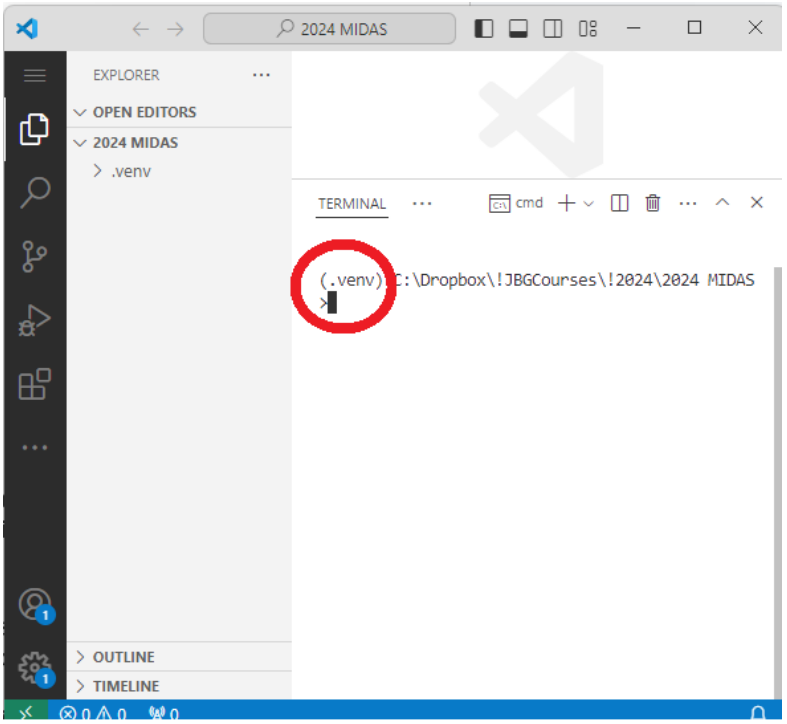

Figure 1.11-13: Terminal showing the `(.venv)` prompt indicating an active virtual environment.

1.12 Partially Required (this or 1.11): Virtual environments outside VS Code

Create virtual environments *outside* Visual Studio. A comprehensive guide is here:
https://biomathematicus.me/working-with-python-virtual-environments-in-visual-studio-code/

Open a terminal window. Now you are ready to install Python packages. We will start by creating a virtual environment; the reason for this is that sometimes specific versions of different libraries might be incompatible. You should create an environment for each project you work on.

1. All related instructions about packages and virtual environments are at:
   https://packaging.python.org/en/latest/tutorials/installing-packages/

2. Check your version of pip. It is the most popular tool for installing Python packages, and the one included with modern versions of Python. If the following command returns a version number, you are good to go. Otherwise, reinstall python (or bootstrap it).

   ```
   python -m pip --version
   ```

3. Update the pip and setuptools

```
python -m pip install --upgrade pip setuptools wheel
```

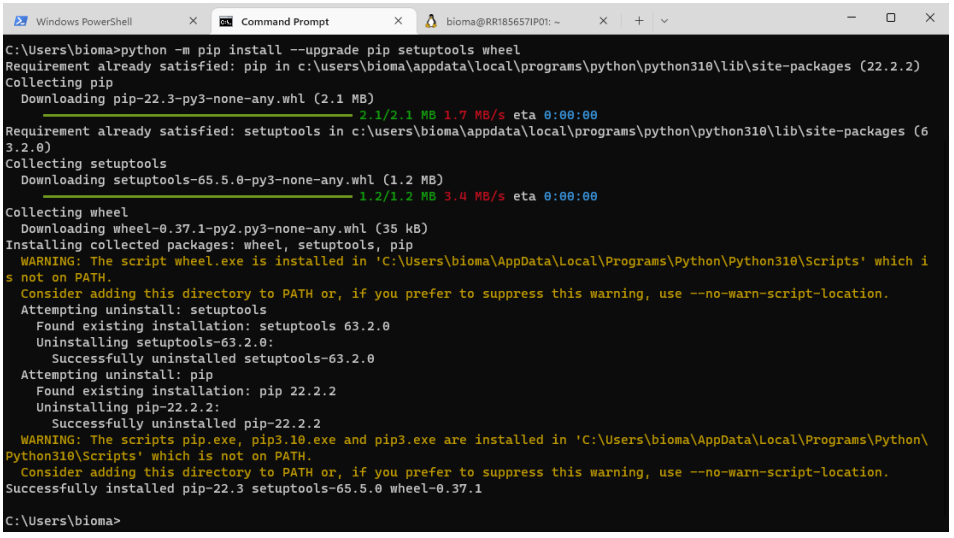

Figure 1.12-14: Terminal output of upgrading pip, setuptools, and wheel.

4. I like to have an easy-to-access folder for virtual environments, thus I always create a folder like `c:\penv` for this purpose. To create this folder, run the following command in the terminal: `mkdir c:\penv`. You can name this folder whatever you like. The importance of this will become evident in step 8. Also importantly, the folder in which you create your virtual environments does not need to be the folder in which your code resides.

5. Now **create** a virtual environment called venn (this is an example for a program about Venn diagrams; you can call it whatever name you prefer):

   ```
   python -m venv c:\penv\venn
   ```

6. To **activate** this environment execute the following command in the terminal in VSC:

   ```
   C:\penv\venn\Scripts\activate
   ```

   Now the command line will show the prompt named as the environment that was activated.

7. To **deactivate** a virtual environment, simply type `deactivate` in the command prompt.

8. The ideal solution is to use the ActivateEnv.bat file described in the appendix (add `c:\venv` to the PATH environment variable). Call it from a .bat or a .sh file as in the following example:

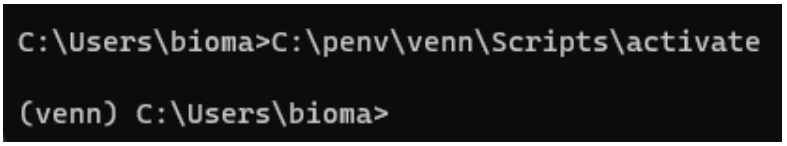

Figure 1.12-15: Command prompt showing the activated virtual environment.

```
ActivateEnv DiNet "C:\Dropbox\!JBGResearch\!!!DiNet\FOO_QtPy"
```

1.13 Troubleshoot

As reported by Lorena Roa de La Cruz, you could see an error when trying to activate a virtual environment. This is due to configuration of permissions. If this happens, run the command:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

After that, run the script `activate`.

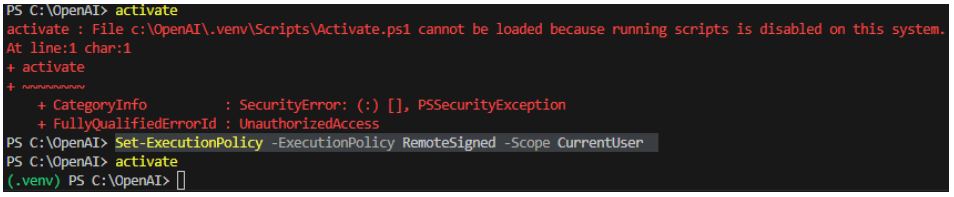

Figure 1.13-16: PowerShell execution policy error and fix: running `Set-ExecutionPolicy` followed by `activate`.

1.14 Required (after 1.11 or 1.12): Install Python libraries

We will install libraries for the python environment for this project environment. The most important libraries are:

- OpenAI
- Anthropic
- Langchain
- PyQt5
- NumPy
- Matplotlib
- SciPy

- Jupyter
- Pandas

For each one of them type the following command: `python -m pip install [library name]`. For example:

```
python -m pip install NumPy
```

Alternatively, if you are using git bash, you can simply type:

```
pip install NumPy
```

Repeat for the libraries listed above. You should see something like this:

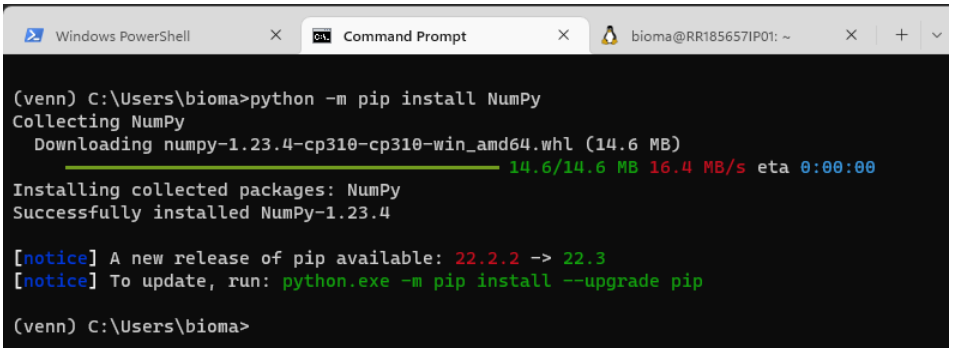

Figure 1.14-17: Terminal output showing a successful NumPy installation.

It is possible that the code fails; the source code repository changes over time. In that case the error will be something like:

```
ModuleNotFoundError: No module named 'library_name'
```

Where `'library_name'` stands for the library name that is causing the error. In that case, simply follow the procedures above to install the missing library.

An important suggestion that comes from experiencing frustration in the past is this: Keep the base environment clean, that is, with minimal libraries. Conflicts between libraries will arise once you install a number of them. This is why it is a good practice to create an environment for each project.

You know your environment is fully functional when you can execute the file agentGroq.py, agentGPT.py and/or agentClaude.py.

1.15 Optional: Jupyter Notebook

Optional: Now, create a Jupyter Notebook.

1. Abundant relevant information regarding Jupyter in VSC is available at https://code.visualstudio.com/docs/datascience/jupyter-notebooks

2. Activate the environment you want to use according to step 1.12.f
3. You need to attach the kernel of your virtual environment. For example, if I have a virtual environment called venn:

```
python -m ipykernel install --user --name=venn
```

4. Execute step 1.12.a
5. In VSC, run the command `Create:New Jupyter Notebook` from the Command Palette (Ctrl+Shift+P) or by creating a new .ipynb file in your workspace.
6. You might receive a Security Alert regarding the firewall. Allow access.

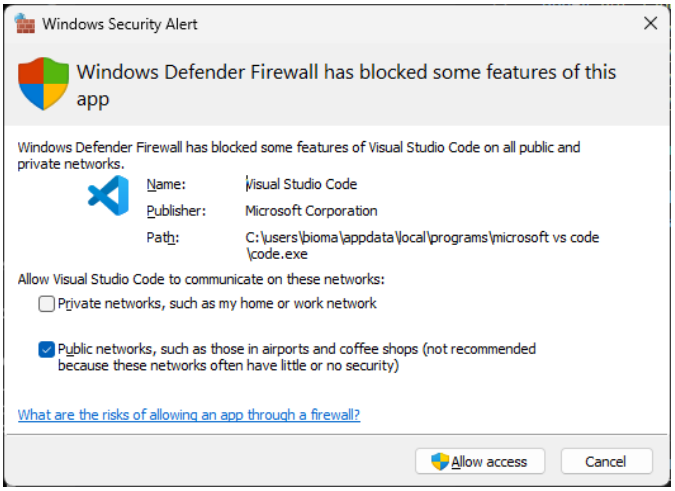

Figure 1.15-18: Windows Defender Firewall alert—allow access for Visual Studio Code.

7. After you enter a command, you might see the following message:

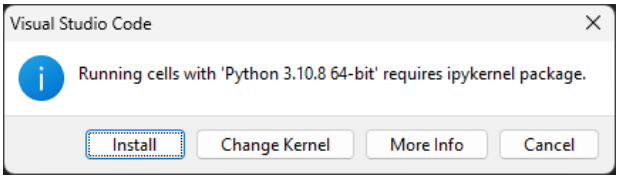

Figure 1.15-19: VS Code prompt to install the ipykernel package—click Install.

Install it.

1.16 Python commands in Terminal

Finally, you can simply enter Python commands in the Terminal. Open a Command Prompt and type the following command:

```
python
```

This will allow you to enter python commands, e.g. `1+1`

APPENDIX: ActivateEnv.bat

ActivateEnv.bat to be placed in `c:\venv` to activate Python environments

```
@echo off
setlocal

:: Read parameters
set "ENV_NAME=%~1"
set "BASE_DIR=%~2"
echo Starting activation of the virtual environment '\%ENV_NAME%'...

:: Validate input
if "%ENV_NAME%"=="" (
    echo [ERROR] Environment name not specified.
    goto :eof
)

if "%BASE_DIR%"=="" (
    echo [ERROR] Base directory not specified.
    goto :eof
)

:: Check if base directory exists
if not exist "%BASE_DIR%" (
    echo [ERROR] Directory not found: %BASE_DIR%
    goto :eof
)

:: Change to working directory
cd /d "%BASE_DIR%"

:: Define the path to the venv under C:\penv
set "VENV_ROOT=C:\penv"
set "VENV_ACTIVATE=%VENV_ROOT%\%ENV_NAME%\Scripts\activate.bat"

:: Check for the activation script
if exist "%VENV_ACTIVATE%" (
    echo calling activation...
    call "%VENV_ACTIVATE%"
    echo activation was called...
) else (
    echo [ERROR] Virtual environment activation script not found:
%VENV_ACTIVATE%
    goto :eof
)
```

```
echo Activation of the virtual environment '\%ENV_NAME%' has ended

echo Launching VSC Starting
code .
echo Launching VSC Ended

endlocal
```

Chapter 2

Introduction to Programming Environments

2.1 Introduction

There is a large number of programming languages. For the purpose of data analytics, we will distinguish two main families of languages: complied vs. scripting languages. They represent two different paradigms of programming language design, each with its own advantages and trade-offs. The key difference lies in when and how the source code you write is translated into machine code that the computer can understand and execute.

In the realm of data analytics, the choice between using a compiled language like C++ and a scripting language like Python often comes down to the specifics of the project at hand, including its complexity, required execution speed, and the need for direct hardware access.

2.1.1 Compiled Languages

Compiled languages are those where the source code is entirely translated into machine code before execution. This translation is done by a compiler, a program that takes the entire source code as an input and generates an executable file. Examples of compiled languages include C++, Java, Rust, Go, and Swift.

Compiled languages offer several important benefits including:

- Performance: Since the code is precompiled into machine code, compiled programs usually run faster and more efficiently than interpreted ones.
- Type checking: Many compiled languages have strict type systems, which catch many type-related errors at compile time, before the code is run.
- Security: The source code is not distributed with the software, which makes it harder for others to reverse engineer the software.

However, compiled languages also have some drawbacks:

- Development speed: The edit-compile-run cycle can be time consuming, particularly for larg software projects.
- Portability: The generated executable is usually specific to a type of processor and operating system. Availability for different operating systems or processors might require specific source code alterations and independent compilations.

Compiled languages like C++ are translated into machine code before execution, making them very fast because they're executed directly by the computer's hardware. This can be a significant advantage when dealing with huge datasets or when computation speed is a critical factor.

However, the syntax of languages like C++ is more complex and it may require more lines of code to achieve the same result as a scripting language. Moreover, C++ lacks many of the built-in data analysis functionalities found in languages such as Python, MATLAB or R, so it's necessary to use external libraries like Eigen or Armadillo for matrix operations and other complex computations.

2.1.2 Scripting Languages

Scripting languages, on the other hand, are typically interpreted from text into machine instructions, line-by-line of code just in time for execution. While this can make scripting languages slower than compiled languages, they're often easier to write and read because of their high-level syntax. Examples of scripting languages include Python, MATLAB, R, JavaScript, Ruby, and PHP.

Scripting languages offer several advantages:

- Ease of use: Scripting languages are usually easier to learn and use. They have less verbose syntax and more built-in functionality.
- Flexibility: Since they are interpreted line-by-line, it's easier to make changes to your code and see their effect immediately without having to recompile.
- Portability: The same script can be run on any system that has the correct interpreter installed, without needing to change the code.

Among the drawbacks of scripting languages there ar issues related to:

- Performance: Interpreted languages are generally slower than compiled languages because of the overhead of interpreting code line-by-line during execution.
- Dependency on the interpreter: The code can only be run on systems that have the correct interpreter installed.

Python is particularly well-suited for data analytics because it has numerous powerful libraries for data analysis (like Pandas, NumPy, and SciPy) and visualization (like Matplotlib and Seaborn). In addition, Python's simple, straightforward

syntax makes it an excellent choice for quick prototyping and exploratory data analysis.

It must be noted that even with all the benefits of Python, the simplified MATLAB syntax for matrix operations facilitates algorithm development since the source code closely resembles mathematical notation.

2.1.3 Comparative Examples

Let us consider a simple example where we want to multiply two matrices A and B. The language examples are listed below in alphabetical oder by the name of the language.

In C++:

```cpp
#include <Eigen/Dense>
#include <iostream>

int main() {
    Eigen::Matrix2f A;
    A << 1, 2,
         3, 4;
    Eigen::Matrix2f B;
    B << 5, 6,
         7, 8;
    std::cout << "The product AB is:\n" << A * B;
}
```

../source/IntroProgMatMul.c

In MATLAB:

```matlab
A = [1, 2; 3, 4];
B = [5, 6; 7, 8];
disp('The product AB is:')
disp(A * B)
```

../source/IntroProgMatMul.m

In R:

```r
A <- matrix(c(1, 3, 2, 4), nrow=2, ncol=2, byrow = TRUE)
B <- matrix(c(5, 7, 6, 8), nrow=2, ncol=2, byrow = TRUE)

print("The product AB is:")
print(A %*% B)
```

../source/IntroProgMatMul.r

In Python:

```python
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print("The product AB is:\n", np.matmul(A, B))
```

../source/IntroProgMatMul.py

From the above example, it's clear that MATLAB code is much more concise and easier to understand than C++. However, if we were dealing with very large matrices or needed to perform this operation many times in a loop, the C++ code might run faster due to being a compiled language.

2.1.4 What is Python?

Python is an *interpreted high-level* programming language for general-purpose computing.

- *Interpreted* means that a program in Python has to be decoded line by line by another program called the Python interpreter and translated into something a computer can understand.
- *High-level* means that there is a strong abstraction of the elements of the hardware of the computer; for instance, memory does not need to be pre-allocated in Python to create a matrix, whereas the C language (a low-level language) requires it. You can think of the level of the language as how close the instructions are to the metal of the machine.
- The process of language interpretation is called *compilation*, that is, the translation of one computer language into another language. This could happen multiple times until the instructions are reduced to a language called *assembler* or *assembly*; this code matches closely the architecture of the computer executing the program. For instance, Java programs are compiled into a language called Java bytecode, which the Java Virtual Machine translates into assembler.

2.1.5 Python Installation

Refer now to the software installation guide provided in a separate document.

Being an interpreted language, it means that Python requires an *interpreter*. This is what you install when you "*install Python*". A Python program is simply a text file that contains instructions the interpreter can understand one line (or block of lines) at a time. Since Python is open source, it has many contributors; this results in a robust ecosystem, but the drawback is that there are numerous software libraries that have to be installed while tracking a complex web of co-dependencies. Hence, there is some difficulty in installing a working Python environment to conduct quantitative analysis.

An easy way to use the Python language is through an *Integrated Development Environment*. There are many IDEs that can execute Python code. Being familiar with more than one environment is important as each has strengths and weaknesses. Furthermore, it is a common occurrence that Python programs appear to fail in one environment but work properly in another; this, of course, is a matter of configuration, but it can be a source of frustration in absence of awareness of the potential sources of conflicts in software. Particularly, you want to know how

to use tools frequently used in software development, and you need exposure to the multiple options you will find in academia, government and industry.

2.1.6 Executing Python Commands

Commands in Python are case-sensitive, that is, you must respect the capitalization of instructions as they appear in the documentation. The Python interpreter offers multiple mechanisms to interact with the environment. The most commonly used is the Python command line. You can start it form a command shell. Simply type `python`. The command line promt will change to `>>>`

You can enter Python commands in the command line prompt. Press *Enter* or *return* to execute commands. Like most other programming languages, Python provides mathematical expressions. The building blocks of expressions are variables, numbers, operators, scripts and functions

The basic operations are listed below.

Symbol	Operation
$+$	Addition
$-$	Subtraction
$\star$	Multiplication
$/$	Division
$\star\star$	Power

Some commands are just like you would type things in a calculator. If you want to refer the result of the command, you need to give it a name, or in other words, assign the result to a variable. The following command assigns the result to the variable s. (Note that Python didn't save the true answer 7/12 but rather a decimal approximation.)

```
>>> s = 1/3 + 1/4
```

If you want to know what the variable s contains, simply type its name in the command prompt:

```
>>> s
0.5833333333333333
```

If a command does not fit on one line, use a backslash (\) followed by Return/Enter to indicate that the statement continues on the next line. For example,

```
>>> s = 1 - 1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 \
... :- 1/8 + 1/9 - 1/10 + 1/11 - 1/12
```

Type s and Enter to see the new value of s.

```
>>> s
0.6532106782106782
```

Use the double star($**$) to raise something to an exponent.

```
>>> 89**2
7921
```

Imaginary numbers can be used with the constant j, which stands for $\sqrt{-1}$ by default.

```
>>> (1j)**2
(-1+0j)
```

Hence, you must be careful in using the variable j when dealing with complex numbers. What do you expect from the following operation? Can you explain what you observe? You might need to complete the entire chapter before you can answer this question. Record your explanation as answer #1.

```
>>> j = 10
>>> j = j*(-1)**(0.5)
>>> j
```

Does the result stored in the variable `j` match your expectation? Explain. Record your explanation as answer #2.

Python uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. Scientific notation uses the letter e or E to specify a power-of-ten scale factor. Imaginary numbers use i as a suffix. (This course does not use complex numbers, but you might generate errors with one.)

Some examples of legal numbers are: 1, -1, 0.0001, 9.87654321, 1.2345e10, 1.2345*10**10, 1j, -2j, 3e5j.

When assigning names to variables or expressions, it is good practice to make the names useful. For example, in the second command above, we gave the result the name rhoSq to imply "rho squared". Observe that you **CAN NOT** have spaces in variable names (i.e. no spaces in names on the left hand side of the equals sign). Giving useful names to your variables or output will help you keep track of what you are doing.

You can use the up and down arrow keys to easily recall and then edit a command with the left and right arrows. Try it. This is the fastest way to repeat Python commands you might have used in the past.

2.1.7 The VSC Python Editor: Executing A Simple Program

Download the file `lab1SimpleInstruction.py`. Open the file in VSC. The following instructions will appear in the editor:

```
1 k=0;
2 for i in range(1,11):
3     k=k+1/10;
4     print(k)
5 print(k==1)
```

You can click the *Run file* button on the toolbar (as shwon in Figure 2.1-1), or press F5. We call all these actions *to execute a program*. The program `lab1SimpleInstruction.py` adds ten times the number 0.1. The comparison of the number 1 against the sum returns a value of *false*, whereas elementary

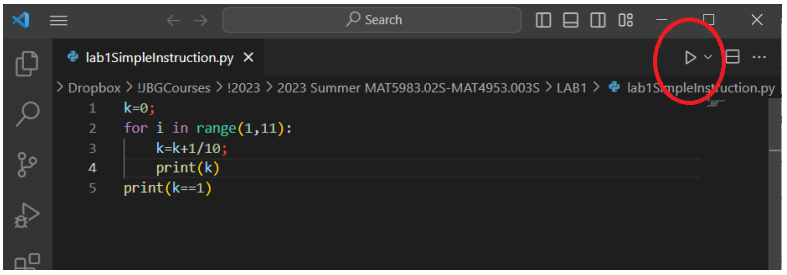

Figure 2.1-1: To execute a Python file in VSC, simply click on the Run button, represented by a triangle pointing to the right. It has been highlighted with a circle.

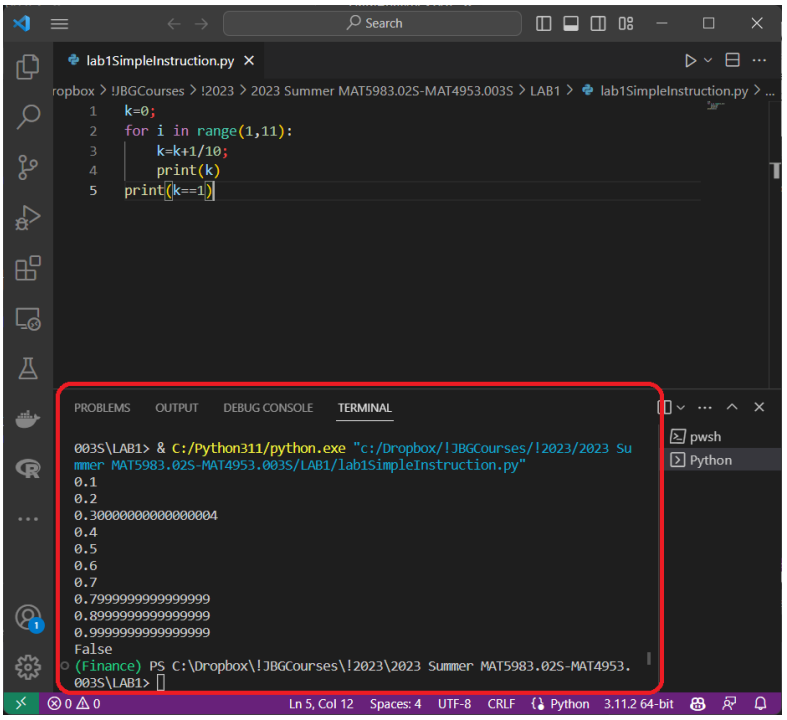

Figure 2.1-2: Executing a python program activates the command line. Results are printed in that panel. The command line is highlighted by the round box.

arithmetic tells us that adding the number 0.1 ten times should result in 1, and the returned value should be *true*. Why is this result false? Record your explanation as answer #3. Once the program is executed, the bottom portion of VSC shows a panel labeled TERMINAL, as well as other tabs labeled PROBLEMS, OUTPUT, and DEBUG CONSOLE, as shown in Figure 2.1-2

Note that you can enter commands in the terminal. For example, type `1+1` and hit enter. Try mathematical operations such as square root, or trigonometric functions. What challenges do you encouter? Record your explanation as answer #4.

In the program `lab1SimpleInstruction.py` all lines of code were executed sequentially, one at a time. This is what we call a **script file**. That is, when you execute a script file, you expect soemthing tangible to happen, whether it is a message on the screen, a file altereed, communication taking place, etc.

There is a way to use source code files to hold code that will be executed

later, without necessarily executing any instruction. At this pointm, open the file `lab1Function.py`. You will see the following instructions:

```
def sum1(n):
    # sum1(n) computes the sum of 1 + 2 + ... + n
    total = 0;
    for i in range(1,n):
        total = total + i
    return total
```

This file defines a function named `sum1(n)`. The command `sum(3)` returns 6 since $1 + 2 + 3 = 6$. It is instructive to see how this code does its work.

- The instruction `def` indicates the beginning of a *function*. Note that all instructions that are executed inside that function have a vertical alignment shifted to the right to indicate that all these commands are subordinate to `def`. This shift in vertical alignment is called indentation. It is part of the syntax of the Python programming language, therefore you must always treat indentations strictly.

- The instruction `# sum1(n)...` is a *comment*. Lines of code that start with the symbol `#` are not executed. The semicolon at the end is optional.

- The instruction `total = 0;` is an *assignment*. This line of code creates the variable `total` and assigns to it the value zero.

- The instruction `for i in range(1,n):` starts a *loop*. The subordinate instructions, identified by indentation, will be repeated n times. Particularly, the instruction `range(1,n)` creates a sequence of integers starting in 1 and ending in n. What is the outcome of `range(10,13)` ?

- The instruction `total = total + i` is an assignment. It is very important to emphasize that this is not an equation; if it were, `total` would cancel and `i` would be zero. What takes place is that for every repetition of the loop, the variable `total` is redefined to contain its previous value plus the value of `i`.

- The instruction `return total` is the output of the function. That is, if somehow we could execute this function like `x = sum1(3)`, the variable `x` would contain the value 6, because `sum1(3)` $= 1 + 2 + 3 = 6$.

Now, we will modify this file. Append at the end a program called `sum2` in which we sum integers squared. The next question has to be: How do we execute these functions?

The command `python [filename]` can be used to execute a Python program from the terminal, or any command line prompt. But if we run the following in the command prompt, nothing happens:

```
python lab1Function.py
```

Why? Record your explanation as answer #5.

Now, add the following line of code to the end of the file, **without indentation**:

```
print(sum1(2))
```

Execute the program. What was the result of `sum1(2)`? Is this what you expected? Why? Record your explanation as answer #6. Compute by hand what the output of `sum1(3)` should be. Now change the last line in the program to read `print(sum1(2))`

What we did was simply to define a function and then use it. Note that if you place the previous line of code at the beginning of the file, the Python interpreter would raise an exception because the function `sum1` is not known to the interpreter yet... keep in mind that the Python interpreter executes one line of code at a time in sequential order (perhaps with branching and repetition, as we will explore later). Therefore, functions must be defined before they can be invoked within a file.

Did the progeam prodice the result you were expecting? In case it did not, enter the following instruction in the terminal: `man range`. This command shows the manual (`man`) for the function `range`. The fact that `range` behaves in certain ways does not force you to abide by that particular design. In fact, you can modify `sum1` so that the outcome corresponds to what a naive user would expect. To achieve this, you will need to change the limit of range inside the function. Implement it and explain. Record your explanation as answer #7.

2.1.8 Executing Functions

In the previous section, we added a single line of code at the end of the program to compute the function we programmed. However, there is a more interesting use case: Create a function that other programs can reuse. In order to use an existing function from a file, we must import that function into the Python environment. We already did something like this when we imported NumPy.

We will now import the function `sum1` from the file `lab1Function.py`. To do this, execute the following command:

```
import lab1function as LAB1
```

This instruction loads into memory a directory of all functions present in that file. Now you can invoke the functions you created. For example,

```
LAB1.sum2(3)
```

It is also possible to load into memory a single function from a file. We can accomplish this by executing

```
from lab1function import sum1
```

In this case, there is no alias associated with the function. To use, you can simply call the function with a parameter, as in `sum2(3)`

Do some research on how to load multiple functions with a single instruction. Explain. Record your explanation as answer #8.

2.2 Introduction to NumPy

Numerical operations can be programmed in Python. A library called `NumPy` encapsulates much needed functionality and has become the *de facto* standard. The basic data element in `NumPy` is a multidimensional array. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a low-level language such as C.

For example, at the Python prompt type in the following command and press Enter.

```
import numpy as np
```

This command loads into memory a reference to the NumPy library. Now you can access all functions in this library by using the prefix `np`; this is an *alias*. You can pick any alias you want, but it is customary to use `np` for NumPy. Now enter the following command:

```
np.sqrt(9)
```

The function `sqrt` computes the square root of a real number. Here is an example, and the resulting values.

```
>>> rho = (1+np.sqrt(5))/2
>>> rhoSq = rho**2
>>> a = rhoSq + rho
>>> a
4.23606797749979
```

However, you cannot use the function `sqrt` with negative numbers. What would you have to do to alloy Python to compute the square root of a negative number and return a complex number? Record your explanation as answer #9.

The library NumPy offers easy access to commonly used constants, such as:

Constant	Value
`np.e`	Returns the number e
`np.pi`	Returns the number π
`np.inf`	Returns the number ∞
`np.nan`	NaN, not-a-number

Expressions use the arithmetic operators and precedence rules you already know by now. There are also functions, such as cosine or sine. You surround the input to the function by parenthesis. With NumPy, we can now access commonly used mathematical functions. Among others:

Function	Operation
`np.abs(x)`	Absolute value $\lvert x \rvert$
`np.sqrt(x)`	Square root $\sqrt{x}$
`np.exp(x)`	Exponential function e^x
`np.log(x)`	$\log(x)$ where x is positive real number
`np.sin(x)`	$\sin(x)$ where x is assumed to be in radians
`np.cos(x)`	$\cos(x)$ where x is assumed to be in radians

The library NumPy has *many* functions. A comprehensive list can be found at `https://numpy.org/doc/stable/reference/`. Now enter the following command:

```
x = np.random.standard_normal(3)
```

There is no output. Why do you think this is the behavior? Record your explanation as answer #10. On the right-hand side of the Spyder window there is a tab called *Variable Explorer*. It will show the value of x. Alternatively, you can enter

```
print(x)
```

Alternatively, you can print the result directly without assigning it to a variable. Try

```
print(np.random.standard_normal(3))
```

Why are the results of this last operation different to the values stored in variable x? Record your explanation as answer #11.

2.2.1 Matrix Operations

The basic data element in NumPy is a multi-dimensional array. Special meaning is sometimes attached to 1-by-1 matrices, which are called scalars, and to matrices with only one row or column, which are called vectors. Python has other ways of storing both numeric and non-numeric data, but in the beginning, it is usually best to think of everything as a matrix.

Row vectors are delimited in notation by square brackets. A comma separates individual numbers in a row vector. Multiple row vectors of the same size are separated by commas, and are delimited by square brackets to define a matrix. It is common practice to use a capital letter when naming matrices in order to distinguish them from a single vector, scalar or variable value.

Now, we will create a vector and a matrix to demonstrate simple matrix operations

Function	Operation
`x = np.array([1,2,3,4])`	Creates the row vector $\mathbf{x} = [1, 2, 3, 4]$
`b = np.array([1,2,3])`	Creates the row vector $\mathbf{b} = [1, 2, 3]$
`A = np.array([[1,2,3],[4,5,6],[7,8,9]])`	Creates the matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$
`B = np.ones([2,3])`	Creates the matrix $A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
`C = np.zeros([2,3])`	Creates the matrix $A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

The following commands demonstrate basic vector manipulation cases:

Operation	Outcome
`x[-1]`	Returns 4, the last element of x
`x[1]`	Returns 2, the second element of x
`x[-2]`	Returns 3, the second to last element of x

The colon (:) operator is notable. In vector notation, it indicates a range of rows or columns. In Python, the first element of a vector has index 0; the second element has index 1, and so on. The colon operator $a : b$ starts at the row or column of index a and covers up to an index less than b. Thus, $A[0 : 2, 0 : 2]$ returns a matrix composed of rows 1 through 2, and columns 1 through 2 with respect to the matrix A.

The following commands demonstrate basic matrix manipulation cases:

Operation	Outcome
`A[1,2]`	Value of 2^{st} row and 3^{nd} column of A
`A[0:2,0:2]`	Returns $A = \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$
`A[1:3,1:3]`	Returns $A = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix}$

The matrix A has three rows and three columns. Then, why does $A[1 : 100, 1 : 100]$ not produce an error? Record your explanation as answer #12.

The size or dimension of a matrix refers to the number of rows and number of columns a matrix has. The `shape` command returns the number of rows and columns of a matrix. Notice that the result is a vector - a one-dimensional matrix with 2 values where the first is the number of rows and the second value is the number of columns. By accessing each element of this vector, we can access each dimension individually.

Using the matrix A of our previous examples,

```
>>> y = A.shape()
>>> y
(3,3)
>>> y[0]
3
>>> y[1]
3
```

The following example will illustrate a subtlety in copying matrices:

```
>>> D = A

array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> D[0,0] = 9
>>> A

array([[9, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Note that by updating a value in D, the corresponding value in A was changed. Why? What would you need to do to prevent an update in D to change A? Record your explanation as answer #13.

The basic matrix operations are listed below, using the matrices in the previous examples.

Operation	Outcome
$A.T$	Transpose of A.
$A.conj().T$	Conjugate transpose of A.
$A@B.T$	Matrix multiplication $A \cdot B$.
$A \star A$	Element-wise multiplication
A/A	Element-wise division
$A \star\star 2$	Element-wise exponentiation

NumPy gives an easy way of solving systems of linear equations, Given the matrix A and the vector $\mathbf{b}$ from the previous examples, the linear system $A\mathbf{x} = \mathbf{b}$ can be solved by invoking `np.linalg.solve` as follows

```
>>> x = np.linalg.solve(A,b)
array([-0.23333333,  0.46666667,  0.1       ])
```

It is easy to use Matrix multiplication $A\mathbf{x}$ to check the answer

```
>>> A@x
array([1., 2., 3.])
```