COMS W3132 Lab 1 Notes

Wednesday, January 31, 8:00 PM-9:15 PM, 480 CSB and Zoom

This is the first COMS W3132 lab of the Spring 2024 semester. The lab will be held in person in CSB 480 and will also be recorded on Zoom for those who could not attend in person. This lab is not mandatory but highly recommended for those relatively new to systematic Python software development.

This lab aims to give you an overview of the most commonly available environments for Python software development. It will cover selected local (running on your computer) and cloud alternatives. It will also teach you about managing Python dependencies, i.e., libraries and modules that are not preinstalled with Python. If you are familiar with these topics, please feel free to skip this lab.

No attendance/participation points will be given in this lab.

1 Choosing a Python Development Environment

1.1 What's an IDE?

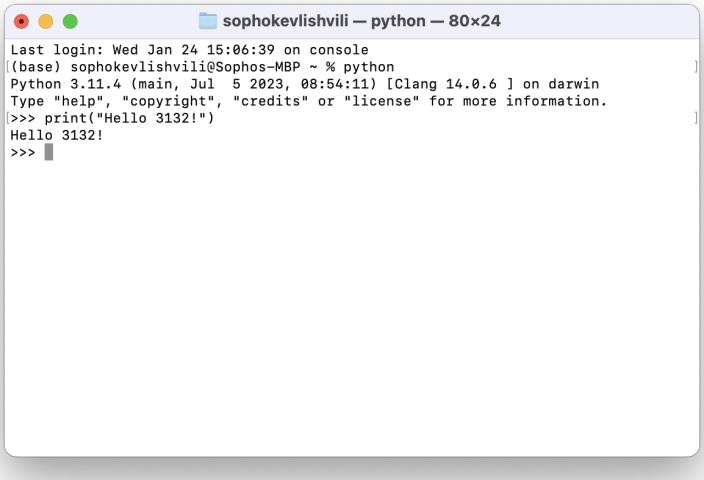
IDE (Integrated Development Environment) is a software application that provides necessary tools and features to make the software development process easier and more efficient. This environment provides tools for effectively writing, debugging, testing, and managing the code. Some of the components that are commonly seen in IDEs to accomplish these tasks include:

1. **Syntax Highlighting:** helps visually distinguish between different elements of the code, e.g. variables, values, comments, brackets, etc.
2. **Autocomplete:** predicts and suggests code snippets as you type to accelerate the coding process.
3. **Debugging tools:** help efficiently find errors in code using breakpoints, step-by-step execution options, variable inspection, error line indication, and etc.
4. **REPL (Read-Eval-Print Loop):** allows running snippets of code interactively while working on a project. This allows code validation or experimentation without the need to create separate files.
5. **Revisions:** some IDEs are integrated with version control systems like Git to allow efficient management of code revisions and version control.
6. **Virtual Environments:** help isolate dependencies and avoid conflicts between different projects. IDEs allow creation of virtual environments for isolation of projects and their dependencies from each other.

1.2 Local Python Development Environments

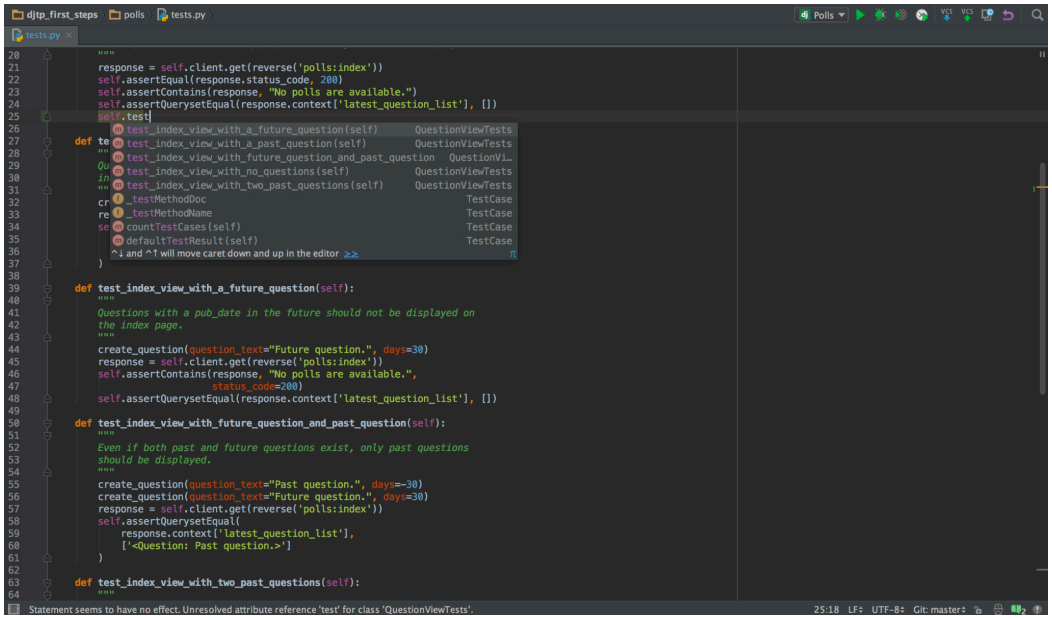
IDLE - Integrated Development and Learning Environment

IDLE is a simple Python IDE that comes bundled with the default Python package. It includes a code editor with features like REPL and syntax highlighting and provides an easy way to quickly run and experience with code. However, it lacks some of the more advanced features found in other IDEs listed below.



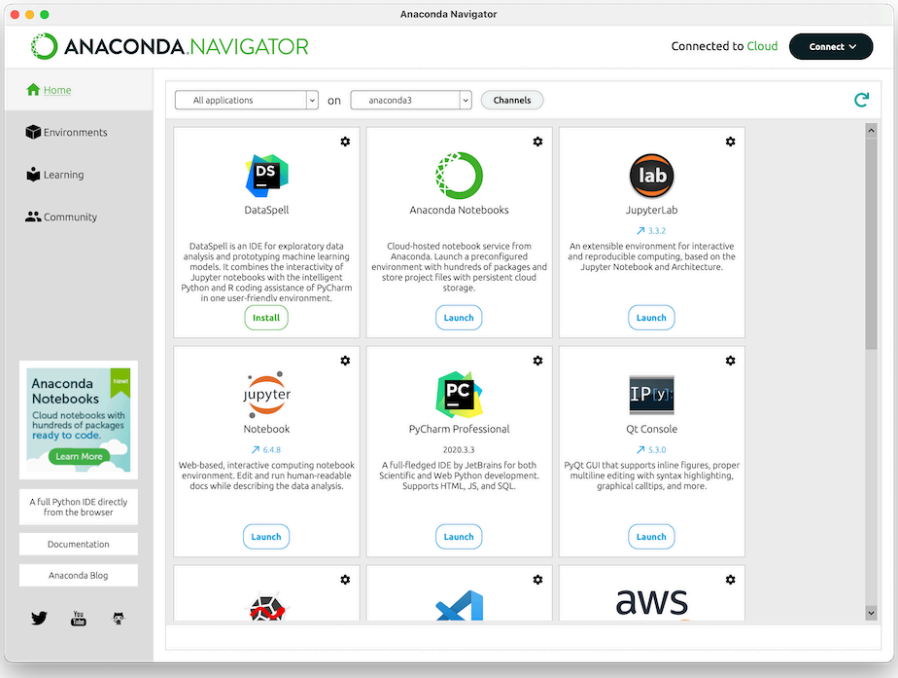
PyCharm

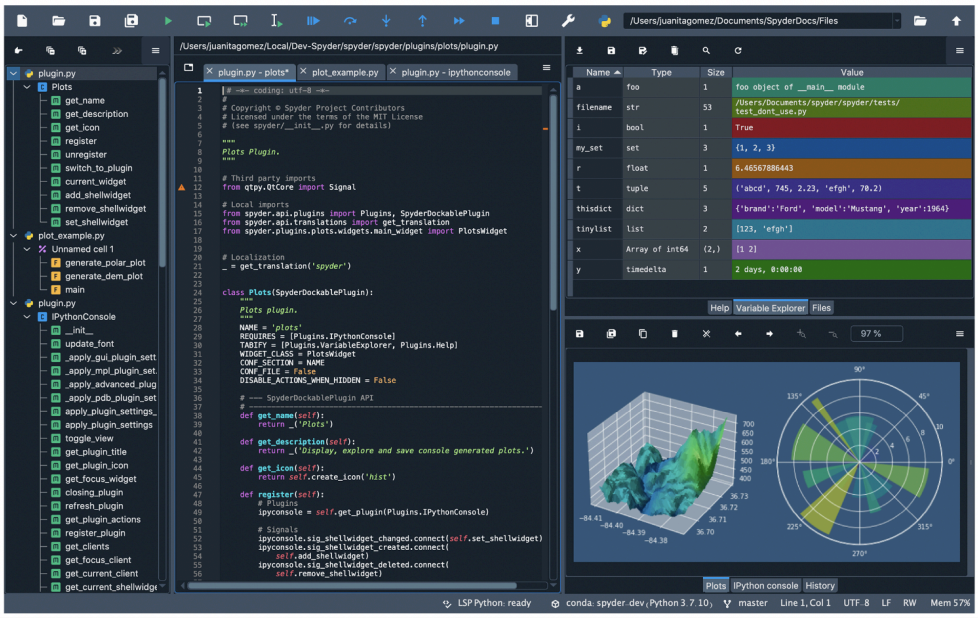
PyCharm offers a wide range of features including code completion, advanced debugging tools, integrated testing support, and built-in version control.



Anaconda / Spyder

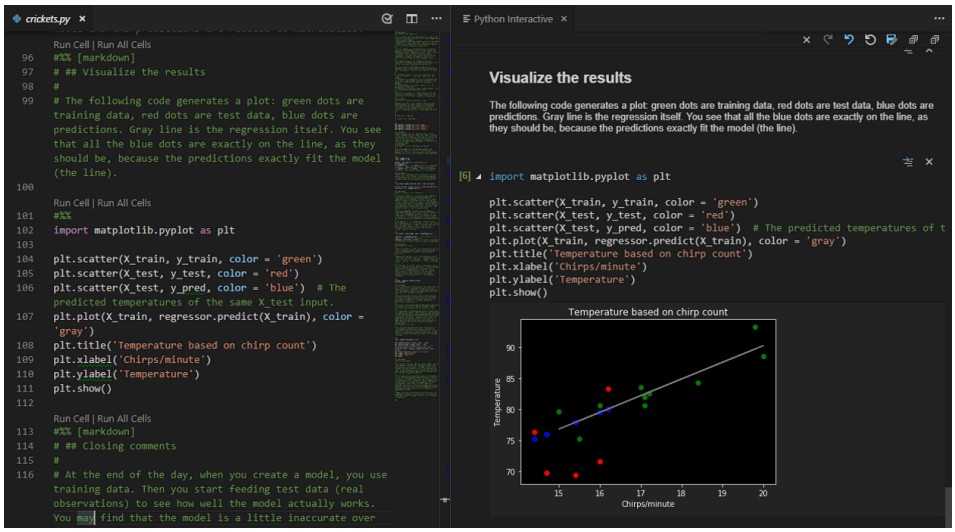
Anaconda comes with a set of data science and machine learning libraries. Spyder is the default IDE that is included in Anaconda. It has an integrated IPython console (REPL), variable explorer, data visualization tools, integrated debugging tools, and etc.





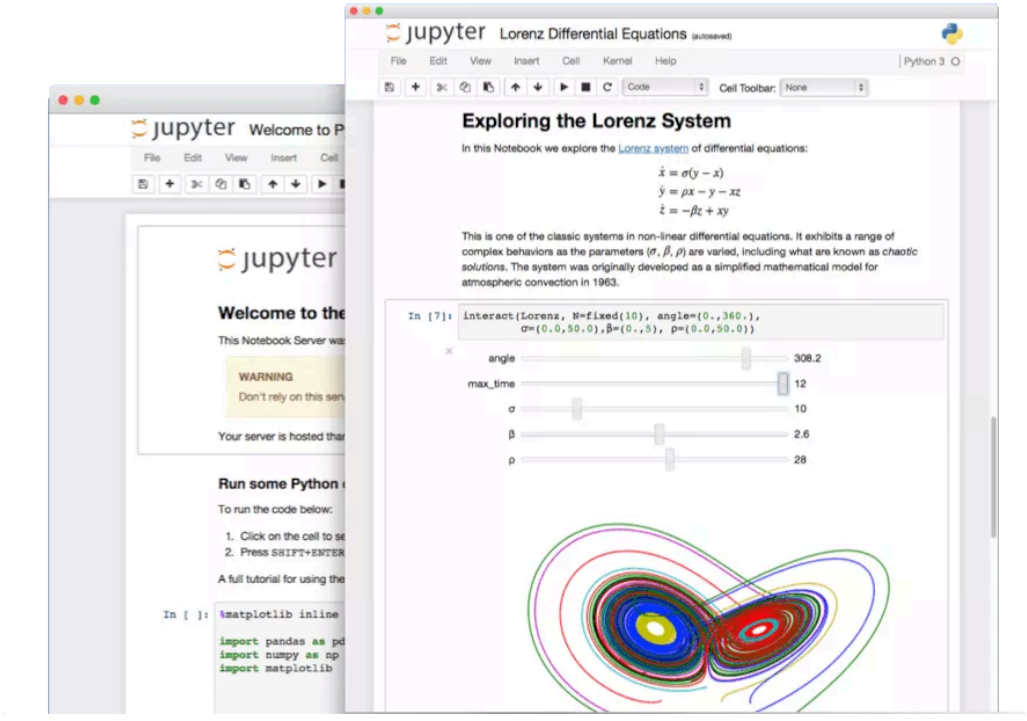
Visual Studio Code

Visual Studio Code (VSCode) is an IDE that allows code development in many languages, including Python. It has a wide variety of features including debugging tools, REPL, version control integration, easy integration with Git, and etc.



Jupyter

Jupyter is an interactive web-based development environment. Jupyter Notebook allows sharing of documents that includes code, markdowns, visualizations, etc.

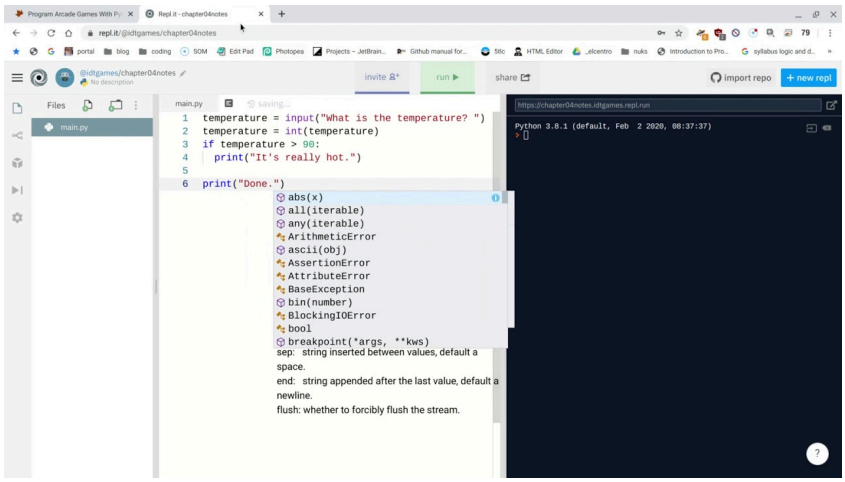


1.3 Cloud Python Development Environments

Cloud Python development environments are online platforms that provide coding environments for Python developers. These can be accessed through a web browser and eliminate the need of installing IDEs and packages locally.

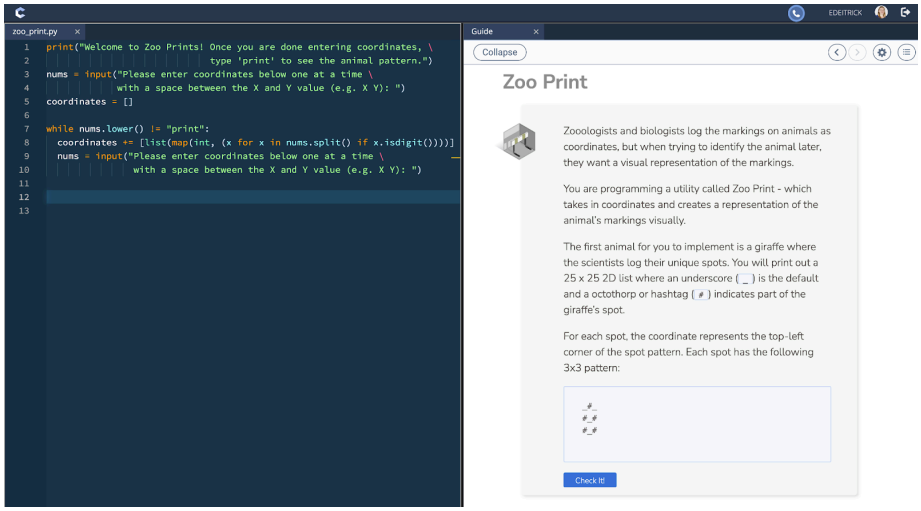
Repl.it

Repl.it is an online coding platform that supports multiple programming languages including Python. It has an interactive coding environment which allows real-time collaboration on a project. It has pre-installed libraries, REPL, syntax highlighting and many other features.



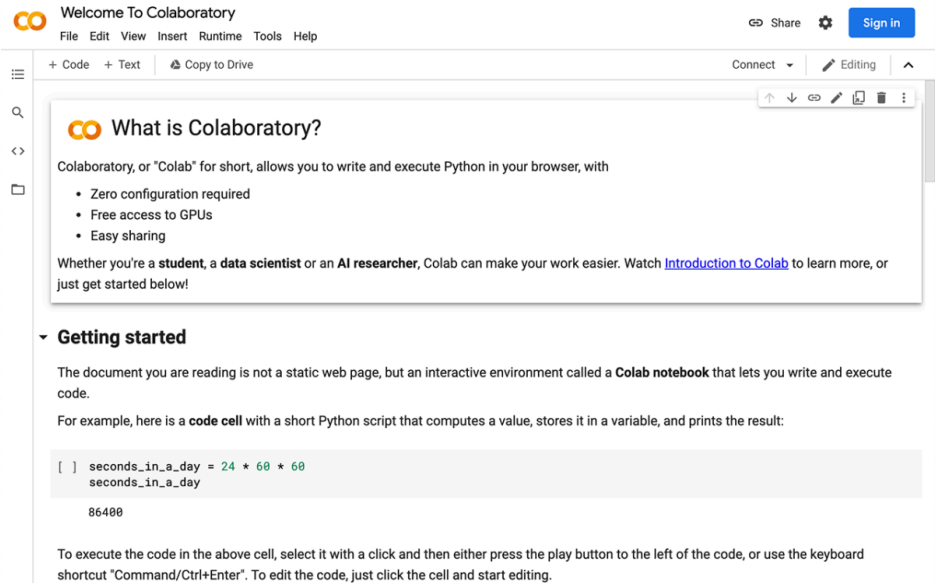
Codio

Codio is a cloud-based development environment which supports multiple programming languages, including Python. It has a customizable interface and allows collaboration on projects. It has some extra features like timelapse of the documents which makes it easier to go back in history and retrieve a previous version of the file.



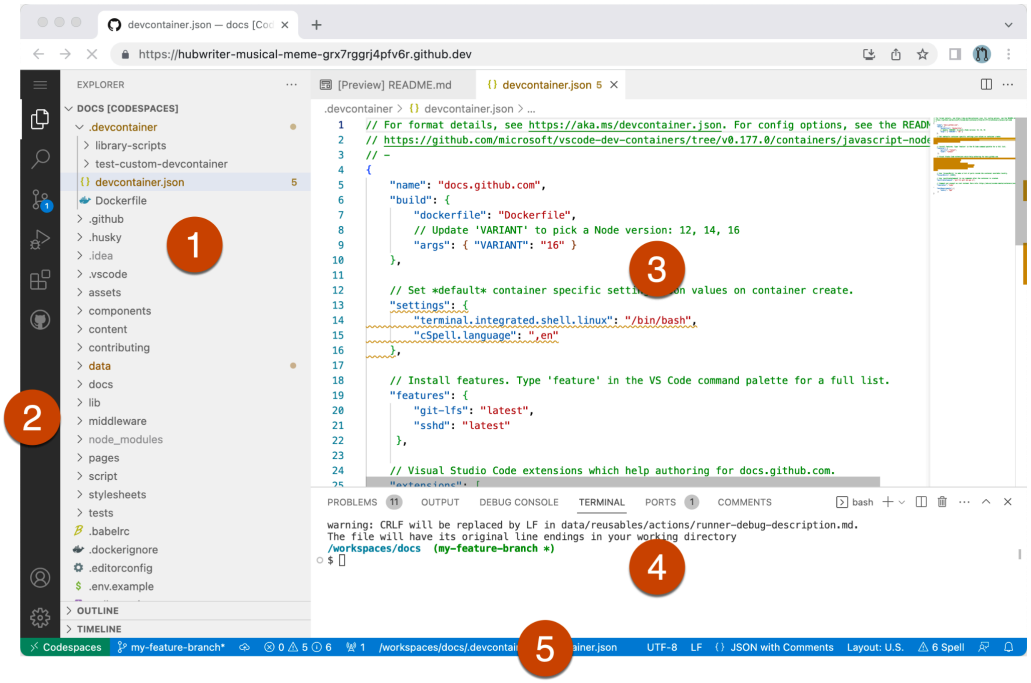
Google Colab

Google Colab (Colaboratory) is a cloud-based Jupyter Notebook environment built by Google. It allows free access to GPU which makes working on Machine Learning and Data Analysis applications easier. It allows collaboration (but not in real time).



Github Codespaces

GitHub Codespaces allows access to development environments within a GitHub repository. You can integrate GitHub Codespaces with VSCode as well. It allows collaboration and easy version control.



2 Managing Python Dependencies

When you install Python, it comes with a limited set of components (libraries and modules). This collection is called the Python standard library. The standard library only includes components that were deemed essential by the Python language developers, i.e., the components that are needed by most Python programs, that enhance the language itself, or improve its portability. The standard library is extensive but does not include everything that Python programmers typically need. When your program needs components not part of the standard library, you must use Python virtual environments[1] and the Python Package Index (PyPI) to get what you need. We will cover both in the following sections.

2.1 Virtual Environments

2.1.1 Motivation

Suppose we want to create a simple Python program to perform matrix multiplication of two matrices a and b. You could implement matrix multiplication yourself by processing individual matrix elements, but that's tedious. Let's instead use a Python library called NumPy. The NumPy library provides a very efficient implementation of common matrix operations such as multiplication. We will create a program in file m.py looking as follows:

```
import numpy

a = [[3, 4, 2],
     [5, 1, 8],
     [3, 1, 9]]

b = [[3, 7, 5],
     [2, 9, 8],
     [1, 5, 8]]

c = numpy.dot(a, b)
print(c)
```

NumPy is not part of the standard library, so when you run the program, you are most likely[2] going to get an error message like this one:

---

[1] The component for managing virtual environments is part of the Python standard library, i.e., every Python installation can create and manage virtual environments.

[2] I am saying **most likely** because some of you may be able to run this program without having to install anything. For example, if you installed Python using Anaconda, you most likely already have the numpy package installed since Anaconda includes more than the standard library. Similarly, you may also have numpy installed system-wide from your previous work with Python. However, if you do a fresh installation of just the Python language, numpy will be missing.

```
$ python m.py
Traceback (most recent call last):
  File "/Users/janakj/m.py", line 1, in <module>
    import numpy
ModuleNotFoundError: No module named 'numpy'
```

Where Python is telling you that it does not know anything about the module numpy. Thus, we first need to install the NumPy package.

There are two ways to install a Python package: system-wide, and in a virtual environment. Installing a Python package system-wide will make it available to all Python programs running on your computer. **This approach is generally discouraged because system-wide packages are difficult to manage.** We will not cover this method here.

2.1.2 Creating a Virtual Environment

We will create a new folder called myenv that will provide a "modified" Python environment for our program. This modified environment will have the NumPy package installed. This folder is called a Python virtual environment. We can ask the Python interpreter to create it for us as follows:

```
$ python -m venv myenv
```

Here, we call the Python interpreter with the optional argument "-m venv"[3] which instructs the Python interpreter to run the venv module from the Python standard library. The last argument "myenv" is the folder name for our virtual environment. The folder must not exist.

WARNING: Do not check virtual environment folders in Git or GitHub! Virtual environments are not meant to be managed in Git because when another developer clones your Git repository with a virtual environment in it, it will not work for them (folder names will be different, they might be running on a different platform, etc.).

2.1.3 Activating a Virtual Environment

Next, we need to *activate* the virtual environment. Activating a virtual environment enables it for the duration of your terminal session. In other words, running the "python" program will look for additional resources in your newly created virtual environment `myenv` instead of the entire system. The activation step depends on your platform. On MacOS and Linux, you can run:

[3] When you run the python program without any argument, it will provide an interactive console where you can run Python functions, perform simple arithmetics, and so on. CS people call this a REPL (from read-eval-print loop). The -m command line argument modifies this behavior. instead of running the REPL, -m venv tells Python to run the venv module from the Python standard library instead. There are several modules in the standard library that can be run this way.

```
$ source myenv/bin/activate
(myenv) janakj@holly ~ $
```

The prefix “(myenv)” indicates that the virtual environment has been activated in my terminal session. Now we can install the NumPy package into the environment using a command called pip (shorthand for Python Package Installer):

```
(myenv) $ pip install numpy
Collecting numpy
  Using cached numpy-1.26.3-cp311-cp311-macosx_11_0_arm64.whl.metadata (115
kB)
Using cached numpy-1.26.3-cp311-cp311-macosx_11_0_arm64.whl (14.0 MB)
Installing collected packages: numpy
Successfully installed numpy-1.26.3

(myenv) $
```

You might be wondering exactly what the pip command does, how it works, and what more it can do. We will cover that in more detail in the following section.

If we rerun our program with the virtual environment activated, it works and will print the result of the matrix multiplication operation:

```
(myenv) $ python m.py
[[19 67 63]
 [25 84 97]
 [20 75 95]]
```

This is how Python virtual environments work. You can create as many virtual environments (folders) as you like. You can install as many packages as you like into each virtual environment. When a virtual environment is activated, everything you install will go into its folder. No system-wide modifications to your computer will be performed.

2.1.4 Deactivating a Virtual Environment

A virtual environment activation is valid for the duration of the terminal session. If you close the terminal window (or log out from the server), the virtual environment will be deactivated. When you open a new terminal window or SSH into your server again, you will have to repeat the activation step. If you wish to deactivate a virtual environment without closing the terminal window or logging out, run the command deactivate:

```
(myenv) $ deactivate
$
```

The command “deactivate” is a shell function that the activation step created internally. Running pip now will install packages system-wide.

2.1.5 Deleting a Virtual Environment

Everything related to a particular virtual environment is stored in its folder. You can simply delete the folder to delete the virtual environment. Make sure that the environment is not active while you are deleting the folder. Nothing bad would happen, but running pip or other programs will not work until you deactivate it (or activate another virtual environment).

2.1.6 Python Program Requirements

Recall the warning about not checking virtual environments into Git. So if we cannot check the virtual environment folder in Git, how do we make it possible for our collaborators or users to recreate the virtual environment so that they could run our program?

We create a plain text file called requirements.txt that will list all the additional packages and their versions installed in our virtual environment. We will use the command “pip freeze” to create the file:

```
(myenv) $ pip freeze > requirements.txt
```

The “> requirements.txt” part redirects the output of the “pip freeze” command to the given file. If you run just “pip freeze”, it will output the list of installed packages and their versions to the terminal:

```
(myenv) $ pip freeze
numpy==1.26.3
```

We see that the virtual environment “myenv” has the version 1.26.3 of the numpy package installed. Since the output contains the version of each package, your collaborators and users will be able to install the precise version of each package you used. If the NumPy developers release a new version of the package in the meantime with a breaking change, your users and collaborators will install a version that is known to work with your program.

If the virtual environment had multiple packages installed, “pip freeze” would output multiple lines with one package-version on each line, for example (the below output is for another virtual environment of mine called “lost”):

```
(lost) $ pip freeze
blinker==1.6.3
certifi==2023.7.22
charset-normalizer==3.3.1
click==8.1.7
Flask==3.0.0
```

```
Flask-Cors==4.0.0
flask-marshmallow==0.15.0
gunicorn==21.2.0
idna==3.4
imageio==2.33.1
itsdangerous==2.1.2
Jinja2==3.1.2
lazy_loader==0.3
lxml==4.9.3
MarkupSafe==2.1.3
marshmallow==3.20.1
networkx==3.2.1
numpy==1.26.1
osm2geojson==0.2.4
packaging==23.2
Pillow==10.1.0
psycopg==3.1.12
psycopg-binary==3.1.12
psycopg-pool==3.1.8
pygml==0.2.2
pyproj==3.6.1
requests==2.31.0
scikit-image==0.22.0
scipy==1.11.4
shapely==2.0.2
tabulate==0.9.0
tifffile==2023.12.9
typing_extensions==4.8.0
urllib3==2.0.7
Werkzeug==3.0.1
```

Suppose we checked the file requirements.txt in Git along with our program m.py. How does somebody recreate the virtual environment for the program to run? The process is fairly straightforward. They first create and activate a new (empty) virtual environment. With the environment activated, they run the command “pip install” with the command line option “-r”:

```
$ python -m venv myenv2
$ source myenv2/bin/activate
(myenv2) $ pip install -r requirements.txt
Collecting numpy==1.26.3 (from -r requirements.txt (line 1))
   Using cached numpy-1.26.3-cp311-cp311-macosx_11_0_arm64.whl.metadata (115
kB)
Using cached numpy-1.26.3-cp311-cp311-macosx_11_0_arm64.whl (14.0 MB)
Installing collected packages: numpy
Successfully installed numpy-1.26.3
```

That's it. Now they have a virtual environment with the same packages as yours and can run the program.

2.1.7 The Virtual Environment Folder (Advanced, Optional)

The curious among you might be wondering how exactly virtual environments work. What is in that virtual environment folder? How does activation work? If you take a closer look at the contents of the folder, you will see three sub-folders and one file:

```
$ ls -al myenv
total 8
drwxr-xr-x    6 janakj  staff   192 Jan 31 15:25 .
drwxr-x---+ 285 janakj  staff  9120 Jan 31 15:44 ..
drwxr-xr-x   13 janakj  staff   416 Jan 31 15:25 bin
drwxr-xr-x    3 janakj  staff    96 Jan 31 15:25 include
drwxr-xr-x    3 janakj  staff    96 Jan 31 15:25 lib
-rw-r--r--    1 janakj  staff   291 Jan 31 15:25 pyvenv.cfg
```

The "bin" sub-folder essentially contains links to your system-wide-installed python and pip programs:

```
$ ls -al myenv/bin
total 80
drwxr-xr-x  13 janakj  staff   416 Jan 31 15:25 .
drwxr-xr-x   6 janakj  staff   192 Jan 31 15:25 ..
-rw-r--r--   1 janakj  staff  9033 Jan 31 15:25 Activate.ps1
-rw-r--r--   1 janakj  staff  1693 Jan 31 15:25 activate
-rw-r--r--   1 janakj  staff   917 Jan 31 15:25 activate.csh
-rw-r--r--   1 janakj  staff  2197 Jan 31 15:25 activate.fish
-rwxr-xr-x   1 janakj  staff   235 Jan 31 15:25 f2py
-rwxr-xr-x   1 janakj  staff   240 Jan 31 15:25 pip
-rwxr-xr-x   1 janakj  staff   240 Jan 31 15:25 pip3
-rwxr-xr-x   1 janakj  staff   240 Jan 31 15:25 pip3.11
lrwxr-xr-x   1 janakj  staff    10 Jan 31 15:25 python -> python3.11
lrwxr-xr-x   1 janakj  staff    10 Jan 31 15:25 python3 -> python3.11
lrwxr-xr-x        1   janakj     staff          44   Jan   31   15:25   python3.11   ->
/opt/homebrew/opt/python@3.11/bin/python3.11
```

Then there is the activation shell script in "bin/activate". When you source the shell script, as we did in the section about activation, the script modifies your PATH and VIRTUAL_ENV environment variables.

The "bin" subfolder is added to your PATH. When you run "python", "pip", or any other command-line tools provided by packages installed in the virtual environment, they will be

searched for in the bin sub-folder first. This guarantees that stuff installed in your activated virtual environment is available to you on your terminal command line.

The VIRTUAL_ENV environment variable, when set, points to the virtual environment's folder. This variable is used by the Python interpreter.

When you run the programs "python" or "pip" with the virtual environment activated, your OS will first find the link in your environment's bin subfolder and run it from there. The Python interpreter is smart enough to detect this, and it will assume you want to use the virtual environment folder. It will then modify its behavior to first search for installed packages in your virtual environment's "lib" subfolder.

The deactivate shell function installed by the "bin/activate" script then reverts your PATH environment variable to the original value and undefines the VIRTUAL_ENV environment variable.

2.2 Working with Python Package Index (PyPI)

Overview

pip is a command-line package installation tool for Python that helps simplify the process of installing and managing packages in Python. pip can download from a variety of sources, but most commonly from PyPI.

PyPI stands for 'Python Package Index' and is the official repository for Python packages where Python developers can publish and distribute their software.

Typically, when installing a package, pip will connect with PyPI, which provides metadata for each package it hosts. pip will use this metadata to examine which packages and versions to install as well as checking and resolving required dependencies. Afterwards, the package distributions will be downloaded and stored in the user's environment in the appropriate directories, with logging to provide additional information.

Installing Packages

To install a new package into our environment, we will use the following command:

```
pip install package_name==version
```

For example, if we want to install pandas, we can run the following command.
Note: not including the version downloads the most recent version of the package.

```
$ pip install pandas
```

If we specifically wanted to install the 2.1.0 version, we can run the following:

```
$ pip install pandas==2.1.0
```

If we wanted to install packages according to a requirements.txt file, we can use the command:

```
$ pip install -r /path/to/requirements.txt
```

Updating Packages

To update a package currently installed, we can use the command:

```
pip install -U package_name
```

For example, to update our pandas package:

```
$ pip install –U pandas
```

Removing Packages

If we want to remove a package from our environment, we can use the command:

```
pip uninstall package_name
```

For example, to remove pandas from our environment:

```
$ pip uninstall pandas
```

Listing Packages

To see what packages are currently installed:

```
pip list
```

Listing Packages + Versions

To see what packages and respective versions are installed:

```
pip freeze
```

For example, if we wanted to generate a requirements.txt file, we can use this command to list all packages and versions of those packages and display the output in file called requirements.txt:

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