



# Lab 3

## IDEs and Venvs

COMS 2132 - Professor Bauer  
TA Session by Darien Moment and Julien Remy

# What's an IDE?

An 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.



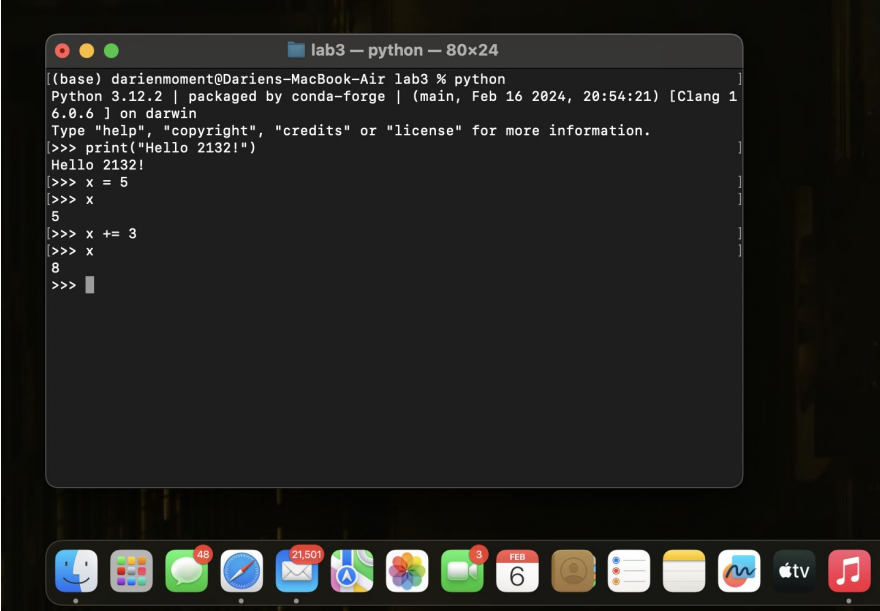


# Why use an IDE?

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.

# IDLE

IDLE is a simple Python IDE that comes bundled with the default Python package. It includes a code editor with features like REPL 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.

A screenshot of a macOS terminal window titled "lab3 — python — 80x24". The terminal shows a Python 3.12.2 prompt. The user enters "python", which shows the version and packaging information. Then, the user enters a series of Python commands: "print('Hello 2132!)", "x = 5", "x", "x += 3", "x", and "8". The output shows "Hello 2132!" and the variable "x" with its value 5, then 8 after the increment. The terminal is set against a dark background with a subtle pattern. The macOS dock is visible at the bottom with various application icons.

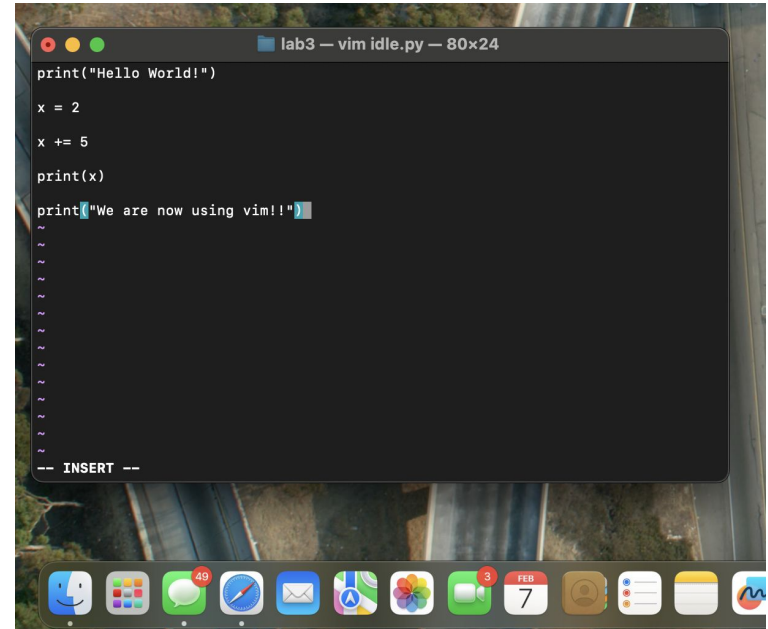
```
lab3 — python — 80x24
(base) darienmoment@Dariens-MacBook-Air lab3 % python
Python 3.12.2 | packaged by conda-forge | (main, Feb 16 2024, 20:54:21) [Clang 16.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello 2132!")
Hello 2132!
>>> x = 5
>>> x
5
>>> x += 3
>>> x
8
>>>
```

# Emacs & Vim

Emacs and Vim are text editors that run in the terminal. Emacs is customizable and includes built-in tools, while Vim is a keyboard-driven editor that lets you edit text efficiently. Both are useful for editing Python code, especially when working on remote computers or using a command-line interface.

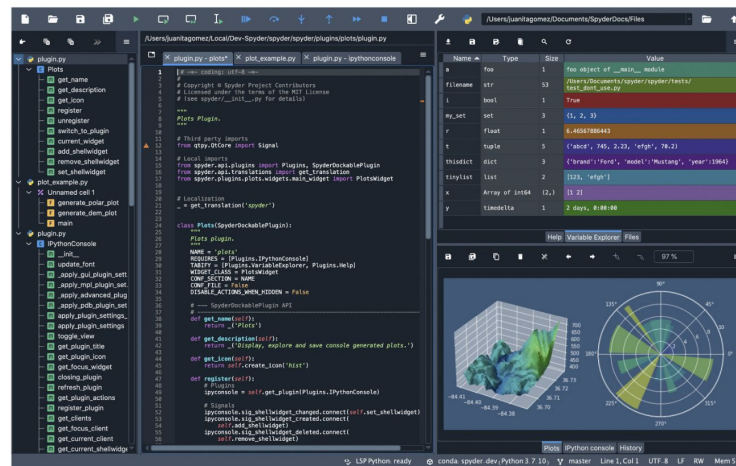
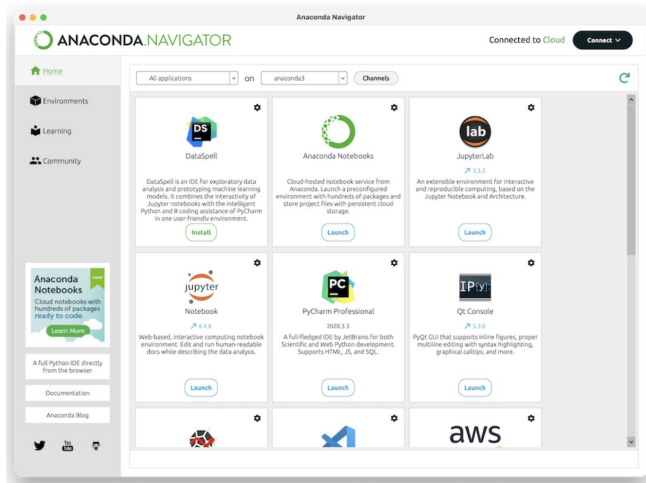
To use vim:

```
$ vim program.py
```

A screenshot of a macOS desktop. In the foreground, a terminal window titled "lab3 - vim idle.py - 80x24" is open. The window has a dark background and shows Python code being edited in Vim. The code includes "print('Hello World!')", "x = 2", "x += 5", "print(x)", and "print('We are now using vim!!')". The cursor is at the end of the last line. Below the code, there are several tilde (~) characters indicating the end of the file, and a "-- INSERT --" prompt. The desktop background is a scenic image of a road and trees. At the bottom, the macOS dock is visible with various application icons, including Finder, Launchpad, Messages, Safari, Mail, Photos, and others. A calendar icon shows the date as February 7th.

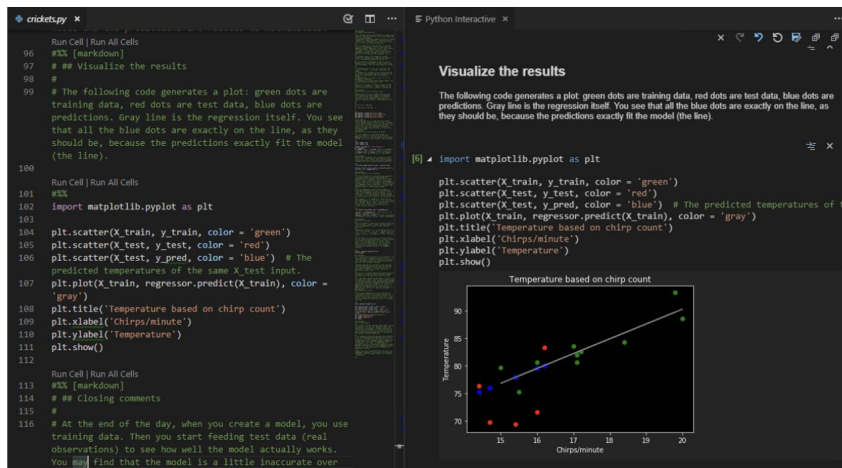
# Anaconda / Spyder

[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

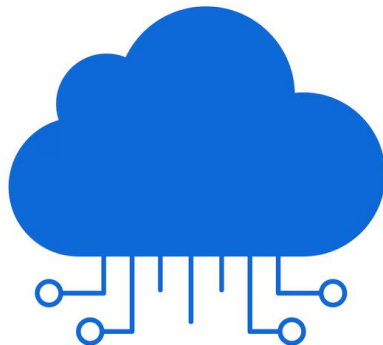
[Visual Studio Code \(VSCode\)](#) is an IDE that allows code development in many languages, including Python. It has a wide variety of features including debugging tools, REPL, version control integration, easy integration with Git, and etc.





# Cloud Python Development Environments

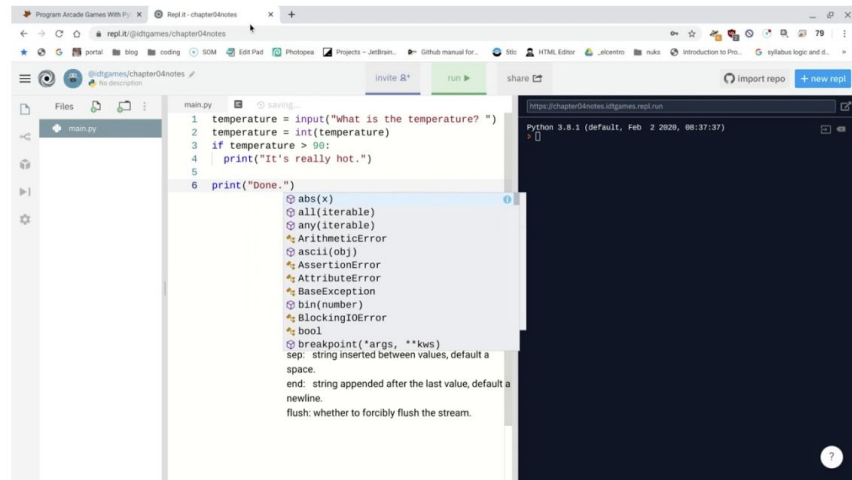
Cloud Python development environments are online platforms that provide coding environments for Python developers. These can be accessed through a web browser and eliminate the need of installing IDEs and packages locally.





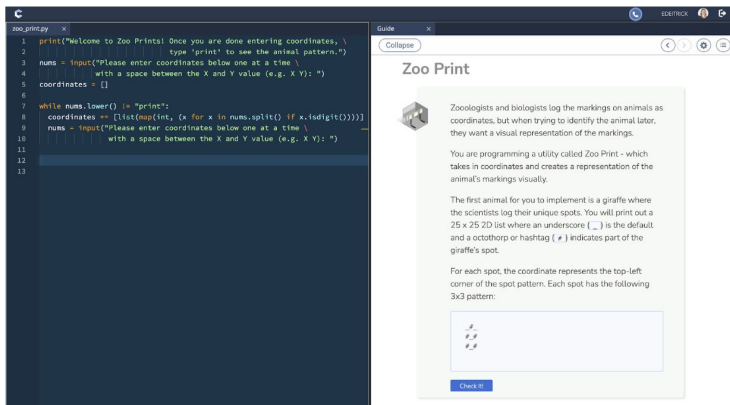
# Repl.it

[Repl.it](https://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.



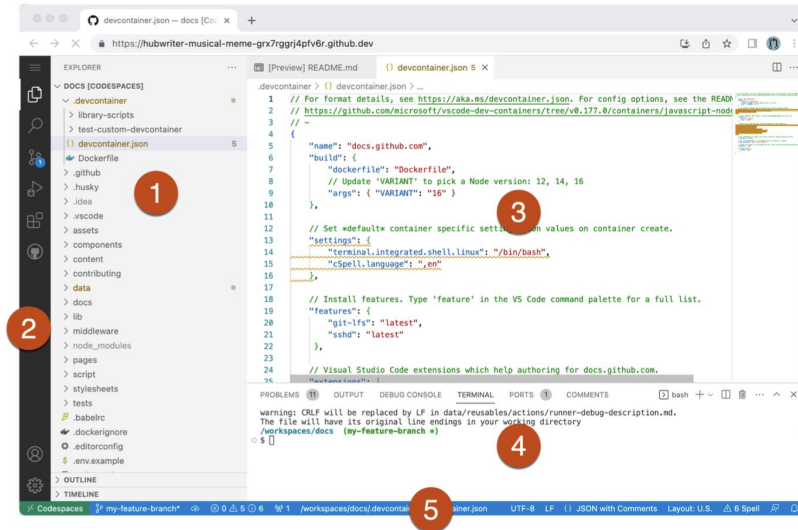
# Codio

[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.



# Github Codespaces

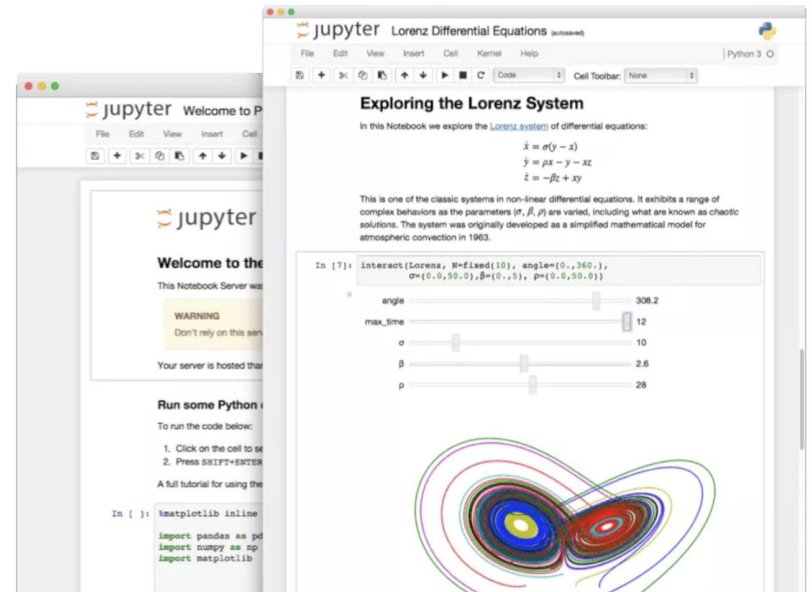
[GitHub Codespaces](#) allows access to development environments within a GitHub repository. You can integrate GitHub Codespaces with VSCode as well. It allows collaboration and easy version control.



# Jupyter

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

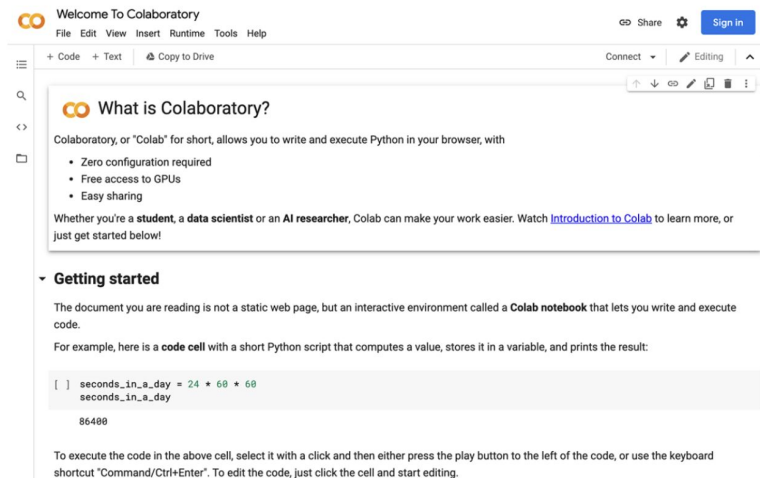
NOTE: Jupyter is an IPython notebook (interactive python) where you can run blocks of code separately. Use Jupyter for data analysis rather than full programs!





# Google Colab

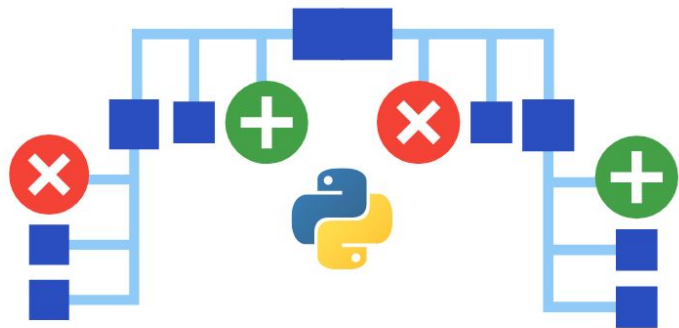
[Google Colab \(Colaboratory\)](#) is a cloud-based Jupyter Notebook environment built by Google. It allows free access to GPU which makes working on Machine Learning and Data Analysis applications easier. It allows collaboration (but not in real time).



# 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 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** and the **Python Package Index (PyPI)** to get what you need.





## Virtual Environment (Venv)

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](#), which 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 going to get an error message like this one:

```
$ python m.py
```

```
Traceback (most recent call last):
```

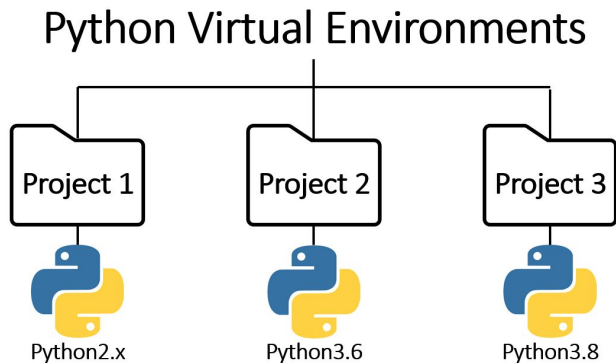
```
  File "/Users/janakj/m.py", line 1, in <module>
```

```
    import numpy
```

```
ModuleNotFoundError: No module named 'numpy'
```

# Venv

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.







## Creating a Virtual Environment

We will create a new folder called myenv that contain a “modified” Python environment for our program. 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
```

#optional argument “-m venv” 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

**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.).**



## Activating a 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.

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

#The prefix “(myenv)” indicates that the virtual environment has been activated in my terminal session



# Installing NumPy

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
```

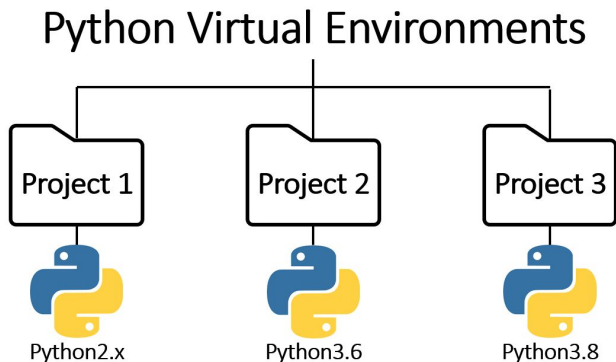
```
Successfully installed numpy-1.26.3
```

```
(myenv) $
```

# Venv

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.





## Deactivating a Venv

A virtual environment activation is valid for the duration of the terminal session. If you close the terminal window, the virtual environment will be deactivated. When you open a new terminal window, 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.
```

## 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).





# Python Program Requirements

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
```



## Recreating a Venv

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?

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
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.





# Working with Python Package Index (PyPI)

**pip** is a command-line package installation tool for Python that helps 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

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
```



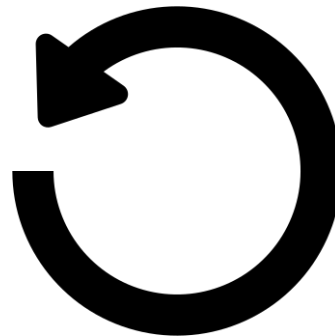
# Updating Packages

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
```



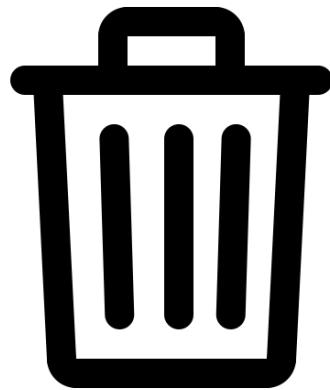
## Removing Packages

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**

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**

