

---



# **Lab 2**

# **IDEs and Venvs**

---

## 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 experiment with code. However, it lacks some of the more advanced features found in other IDEs listed below.



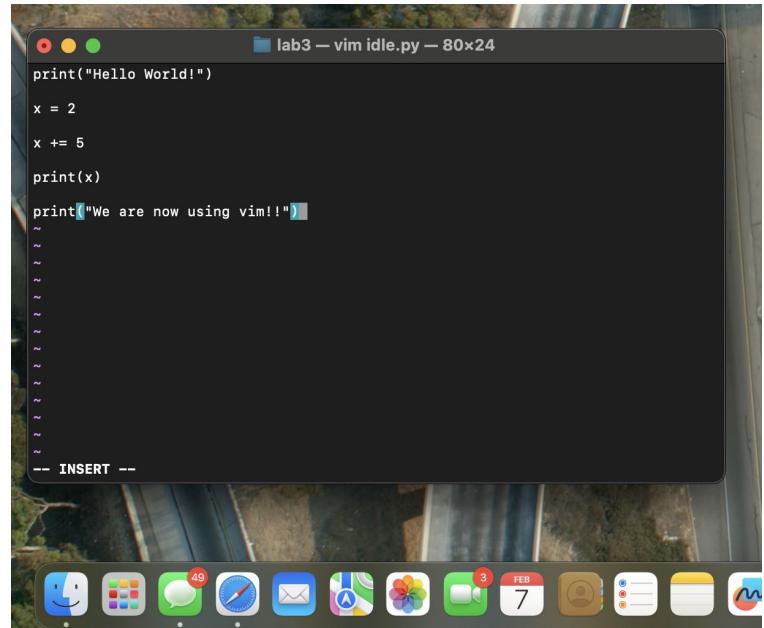
```
(base) darienmoment@Dariens-MacBook-Air lab3 % python
Python 3.12.2 | packaged by conda-forge | (main, Feb 16 2024, 20:54:21) [Clang 1
6.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello 2132!")
Hello 2132!
>>> x = 5
>>> x
5
>>> x += 3
>>> x
8
>>> █
```

# 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.

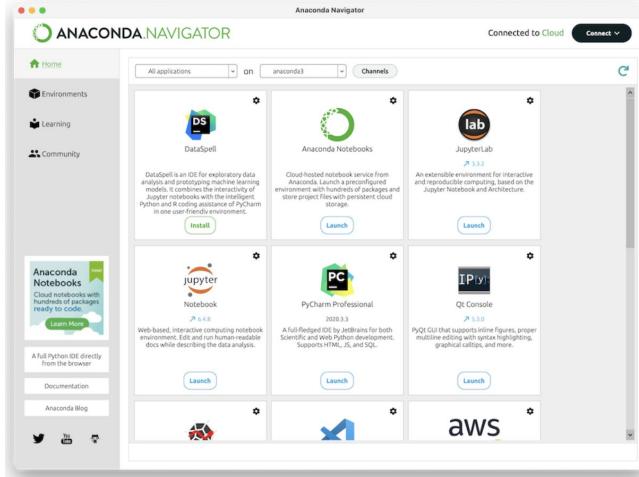
To use vim:

```
$ vim program.py
```



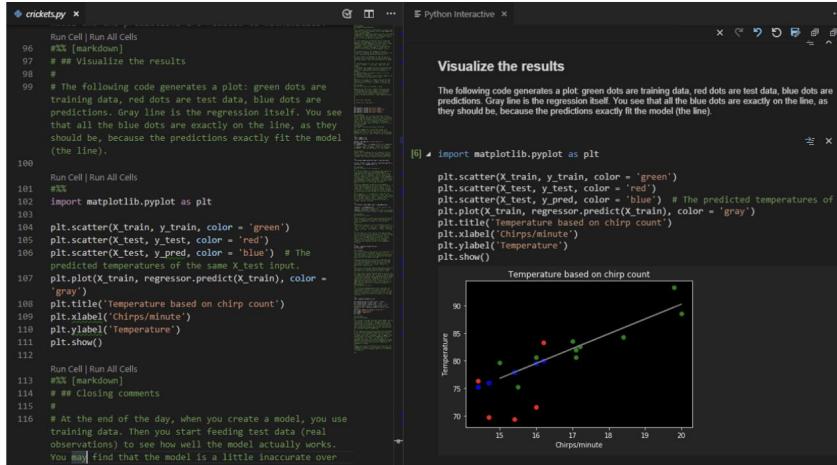
# Anaconda / Spyder

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.

The screenshot shows the Spyder IDE interface. The code editor on the left contains Python code for a "Plots" plugin. The variable explorer in the center shows variables like "foo", "my\_set", "t", "thistd", "tinylist", "x", and "y". The bottom right features a 3D surface plot. The status bar at the bottom indicates "LSP Python ready", "conda spyder.dev/Python 3.7.10", "master", "Line 1, Col 1", "UTF-8", "LF", and "Mem 57%".

# 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.



The screenshot shows the Visual Studio Code interface with two main windows:

- crickets.py**: A code editor window containing Python code for a regression model. The code imports numpy, pandas, and matplotlib.pyplot, then performs data analysis and visualization. It includes comments explaining the steps, such as generating training and test data, and visualizing the results.
- Python Interactive**: A terminal-like window titled "Visualize the results". It displays the output of the code, which includes a scatter plot of Temperature vs Chirps/minute. The plot shows green dots (training data), red dots (test data), blue dots (predicted values), and a gray regression line. The text output explains that the predictions exactly fit the model.

The code in the editor:

```
96 #%% [markdown]
97 # ## Visualize the results
98 #
99 # The following code generates a plot: green dots are
100 # training data, red dots are test data, blue dots are
101 # predictions. Gray line is the regression itself. You see
102 # that all the blue dots are exactly on the line, as they
103 # should be, because the predictions exactly fit the model
104 # (the line).
105 #
106 # At the end of the day, when you create a model, you use
107 # training data. Then you start feeding test data (real
108 # observations) to see how well the model actually works.
109 # You will find that the model is a little inaccurate over
110 #
111 # ## Closing comments
112 #
113 # %% [markdown]
114 # ## Closing comments
115 #
116 # At the end of the day, when you create a model, you use
```

The output in the interactive window:

```
Visualize the results

The following code generates a plot: green dots are training data, red dots are test data, blue dots are predictions. Gray line is the regression itself. You see that all the blue dots are exactly on the line, as they should be, because the predictions exactly fit the model (the line).

[6] > import matplotlib.pyplot as plt

plt.scatter(X_train, y_train, color = 'green')
plt.scatter(X_test, y_test, color = 'red')
plt.scatter(X_test, y_pred, color = 'blue') # The predicted temperatures of t
plt.plot(X_train, regressor.predict(X_train), color = 'gray')
plt.title('Temperature based on chirp count')
plt.xlabel('Chirps/minute')
plt.ylabel('Temperature')
plt.show()
```

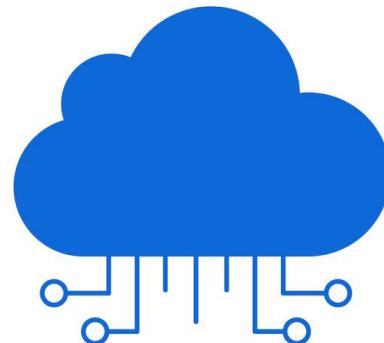
Temperature based on chirp count

Chirps/minute	Temperature
15.0	75.0
15.5	78.0
16.0	72.0
16.5	74.0
17.0	80.0
17.5	82.0
18.0	84.0
18.5	86.0
19.0	88.0
20.0	90.0

---

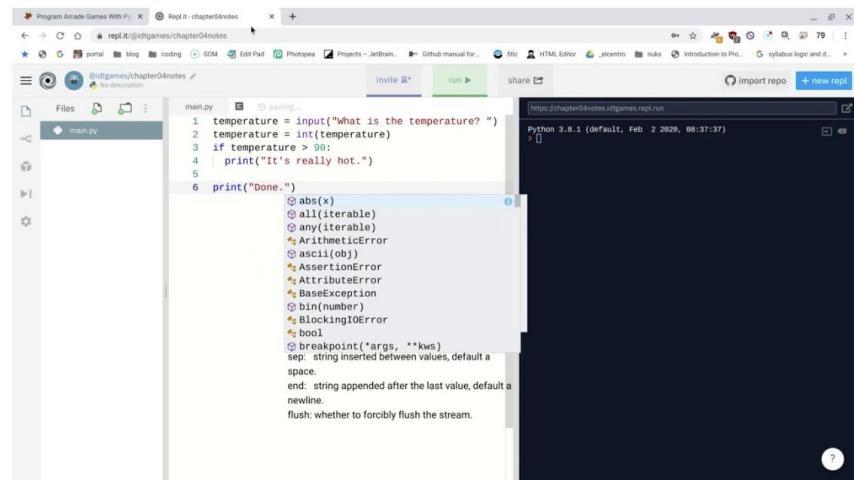
# Cloud Python Development Environments

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.



The screenshot shows a browser-based Python editor interface. On the left, there's a file tree with a single file named 'main.py'. The code in 'main.py' is:

```
1 temperature = input("What is the temperature? ")
2 temperature = float(temperature)
3 if temperature > 90:
4     print("It's really hot.")
5
6 print("Done.")
```

A dropdown code completion menu is open over the final line of code, showing suggestions like 'abs()', 'all()', 'any()', 'ArithmeticError', etc. To the right of the code area is a terminal window showing the command 'Python 3.8.1 (default, Feb 2 2020, 08:37:37)'. Below the terminal is a status bar with the URL 'https://chapter04notes.repl.it/repl' and some other information.

# 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.

The screenshot shows the Codio IDE interface. On the left, there is a code editor window titled "zoo.print.py" containing the following Python code:

```
1  print("Welcome to Zoo Prints! Once you are done entering coordinates, type 'print' to see the animal pattern.")
2
3  num = input("Please enter coordinates below one at a time, with a space between the X and Y value (e.g. X Y); ")
4
5  coordinates = []
6
7  while num.lower() != 'print':
8      coordinates.append([int(x) for x in num.split() if x.isdigit()])
9      num = input("Please enter coordinates below one at a time, with a space between the X and Y value (e.g. X Y); ")
10
11
12
13
```

To the right of the code editor is a "Guide" panel titled "Zoo Print". The guide text provides instructions for creating a visual representation of animal markings:

Zoologists and biologists log the markings on animals as coordinates, but when trying to identify the animal later, they want a visual representation of the markings.

You are programming a utility called Zoo Print - which takes in coordinates and creates a representation of the animal's markings visually.

The first animal for you to implement is a giraffe where the scientists log their unique spots. You will print out a 25 x 25 2D list where an underscore (\_) is the default, and a octothorp or hashtag (#) indicates part of the giraffe's spot.

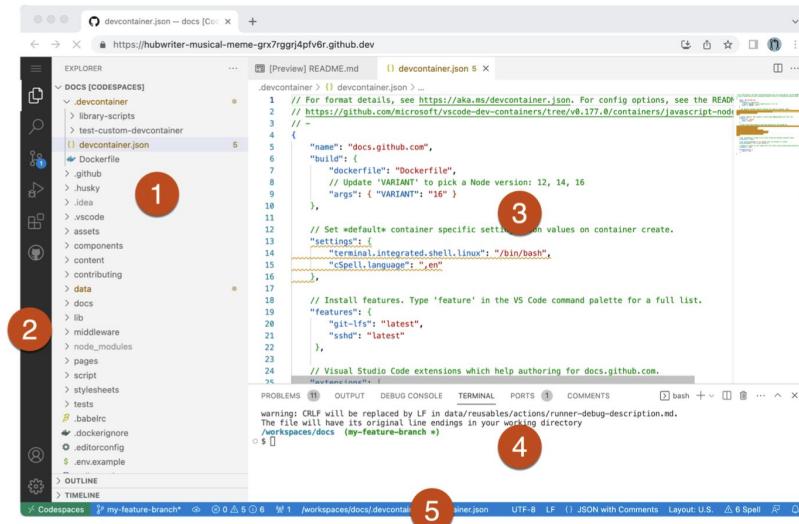
For each spot, the coordinate represents the top-left corner of the spot pattern. Each spot has the following 3x3 pattern:

#	_	_
_	#	_
_	_	#

A "Check It!" button is located at the bottom of the guide panel.

# Github Codespaces

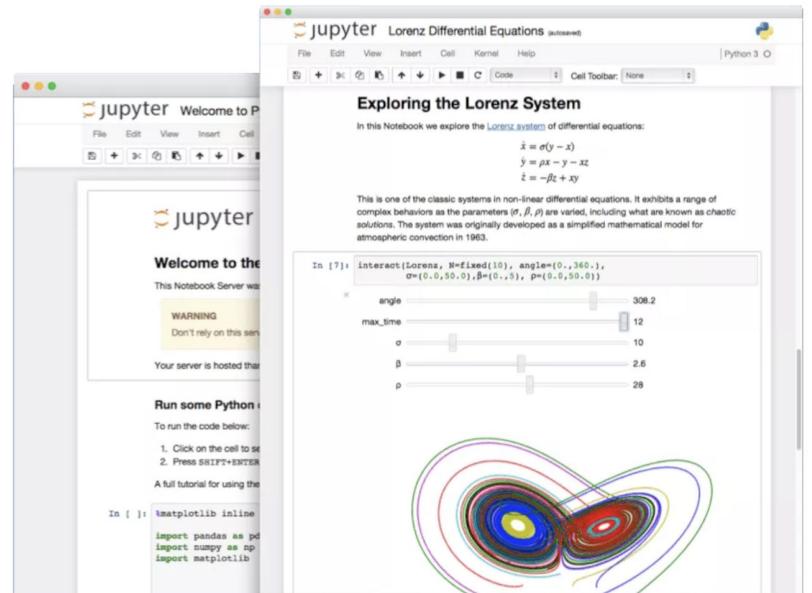
[GitHub Codespaces](#) allows access to development environments within a GitHub repository. You can integrate GitHub Codespaces with VSCode as well. It allows collaboration and easy version control.



# 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).

The screenshot shows the Google Colab interface. At the top, there's a navigation bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. On the right side of the bar are 'Share', 'Sign in', 'Connect', and 'Editing' buttons. Below the bar, there are tabs for '+ Code' and '+ Text', and a 'Copy to Drive' button. The main content area has a title 'What is Colaboratory?' with a sub-section 'Getting started'. The 'Getting started' section contains text about Colab being an interactive environment for writing and executing Python code. It includes a code cell example that calculates the number of seconds in a day:

```
[ ] seconds_in_a_day = 24 * 60 * 60
seconds_in_a_day
```

86400

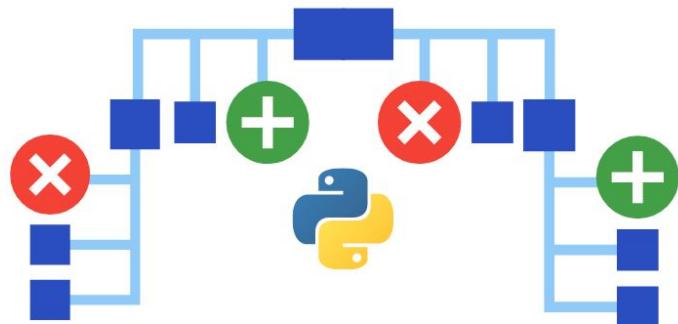
Below the code cell, instructions say: "To execute the code in the above cell, select it with a click and then either press the play button to the left of the code, or use the keyboard shortcut 'Command/Ctrl+Enter'. To edit the code, just click the cell and start editing."

---

# 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)
```

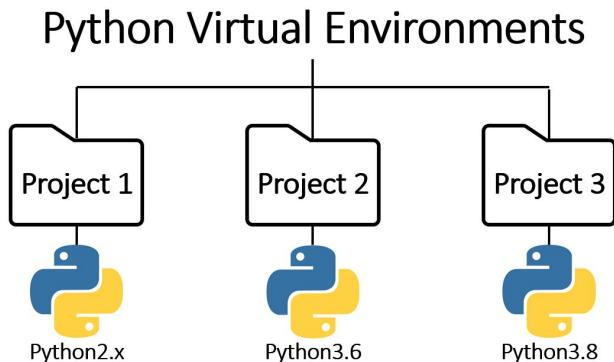
NumPy is not part of the standard library, so when you run the program, you are most likely going to get an error message like this one:

```
$ 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 #The prefix "(myenv)" indicates that the virtual environment has been  
(myenv) janakj@holly ~ $ 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

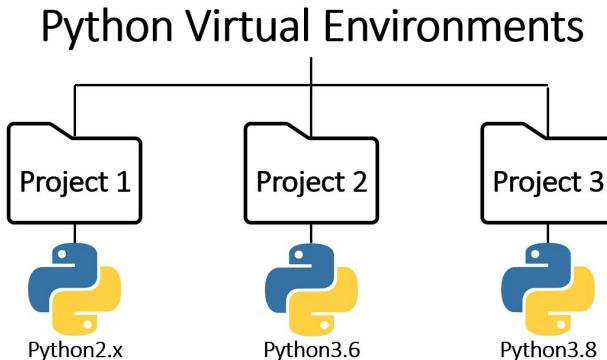
(myenv) $
```

---

# 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
```

---

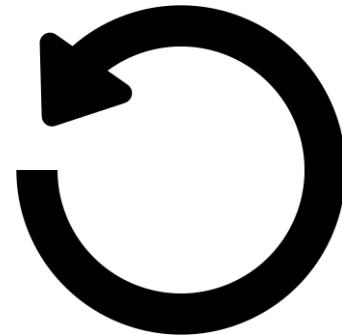
# Updating Packages

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
```



---

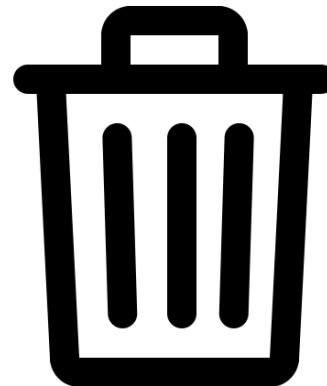
# Removing Packages

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**



---

## Attendance

