Data Visualization with Python

CS230

(Part I)

When we are talking about generating data in python, often it's necessary to visualize that data. We can explore all types of data through various visual representations. While data visualization is somewhat associated with data analytics, keep in mind that data analytics is the exploration of patterns and associated data sets. Nonetheless, think about how much data is out there. It is astronomical!

Humans tend to recognize patterns in our daily lives, and we are constantly looking for visual clues. Patterns help us navigate the world. While you might think that visualizing a dataset is nothing more than a simple visual effect, it is so much more. It enables the viewer to clearly see and analyze the patterns we've produced, while also identifying and differentiating the significance of those data points. In short, a visual representation of our data helps bring these data sets into scope so that they are easier to understand. The alternative: Reading the output line by line!

Two data visualization tools that you should be familiar with are Matplotlib and Plotly. Matplotlib is a mathematical plotting library that can be used to make simple-complex graphs and various plots based on datasets. Plotly is a package that we can employ to create visualizations that are well suited for a multitude of digital devices and offers several interactive features we can use to interact with our visualizations.

Part I

The first part of your project is to become familiar with Matplotlib and Plotly. Although this first part is mostly tutorial-based instruction, it is worth 50 points. You must complete the tutorial and submit any code and graphics that are required. THAT IS: You need to implement each line of code in this tutorial, turn in the scripts and visualizations in Canvas. There are also some extra tasks at the end of the tutorial.

MATPLOTLIB

To create visualizations using Matplotlib, you will first need to install it. Later Python versions standardized the ".pip" module by default, which enables us to download and install Python Packages.

Open up a terminal (Command Prompt) and enter the following command:

```
python -m pip install --user matplotlib
```

**Note – If you are using a different operating system such as macOS, and the installation fails, you can try omitting the "--user" flag. If it still fails, you need to search for a solution on your own. Also, the above command tells Python to install the matplotlib package to the current Python installation (working directory), meaning that if you are using a different version of Python such as Python3, you must use "python3" in your command.*

There are many kinds of graphics we can create using Matplotlib and there are too many to list here. Please look at the different types of visualizations we can create here:

https://matplotlib.org/stable/gallery/index.html

Once you've installed Matplotlib we can begin creating some simple visualizations. For example, lets create a line graph:

1. Create a .py file and call it visualization assignment.

2. Next, we need to import the library and a module called "pyplot". Pyplot is a module which contains various functions and methods that help us create charts and graphs. Therefore, we need to import:

   ```
   import matplotlib.pyplot as plt
   ```

   ***plt is simply an alias for the pyplot module**

3. Let's create a list that stores all prime numbers 2-20.

   ```
   primes = [2, 3, 5, 7, 11, 13, 17, 19]
   ```

4. Next, we need to call matplotlib's subplots() function. This function allows us to plot multiple plots on a single figure. We can provide an arbitrary number of rows and columns, and the function will return a tuple (fig, ax), where fig represents the entire figure (a single figure) and where the variable ax represents a single plot in the figure or an array of axes, thus "ax."

   ```
   fig, ax = plt.subplots()
   ```

5. Next, we plot the primes. We use this method to plot y versus x as lines and/or markers. That is, we are attempting to plot the data in a meaningful way. As you see, we've passed our list "primes" as an argument.

   ```
   ax.plot(primes)
   ```

6. Next, we need to open matplotlib's viewer in order to display our plot. We do so with matplotlib's show() function. Therefore:

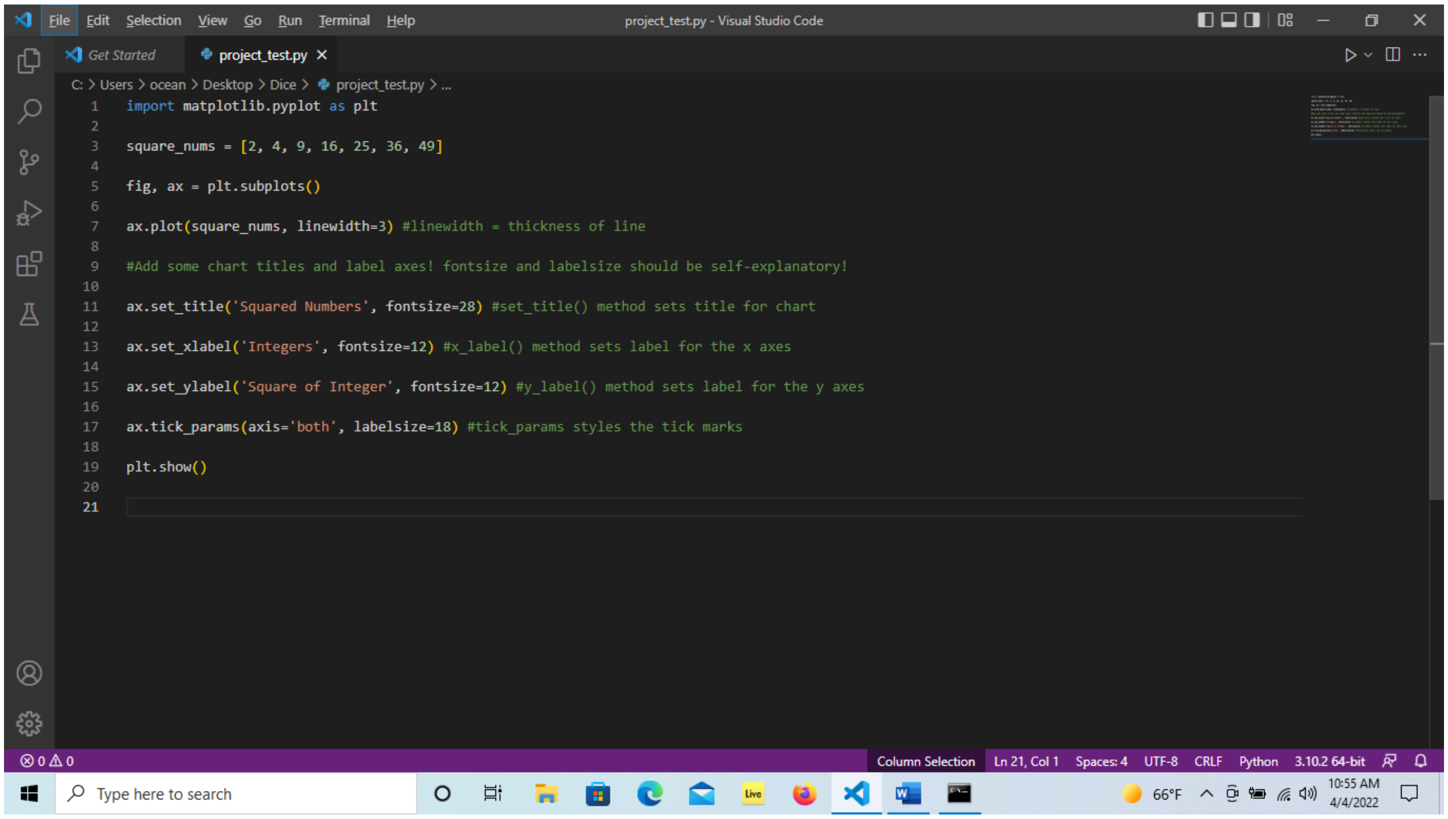
   ```
   plt.show()
   ```

7. As you see, this is an extremely simple plot we've created, but congrats! You've created your first plot using matplotlib. Let's run the complete script trough the interpreter using Visual Studio Code.

   ```
   import matplotlib.pyplot as plt

   primes = [2, 3, 5, 7, 11, 13, 17, 19]

   fig, ax = plt.subplots()

   ax.plot(primes)

   plt.show()
   ```

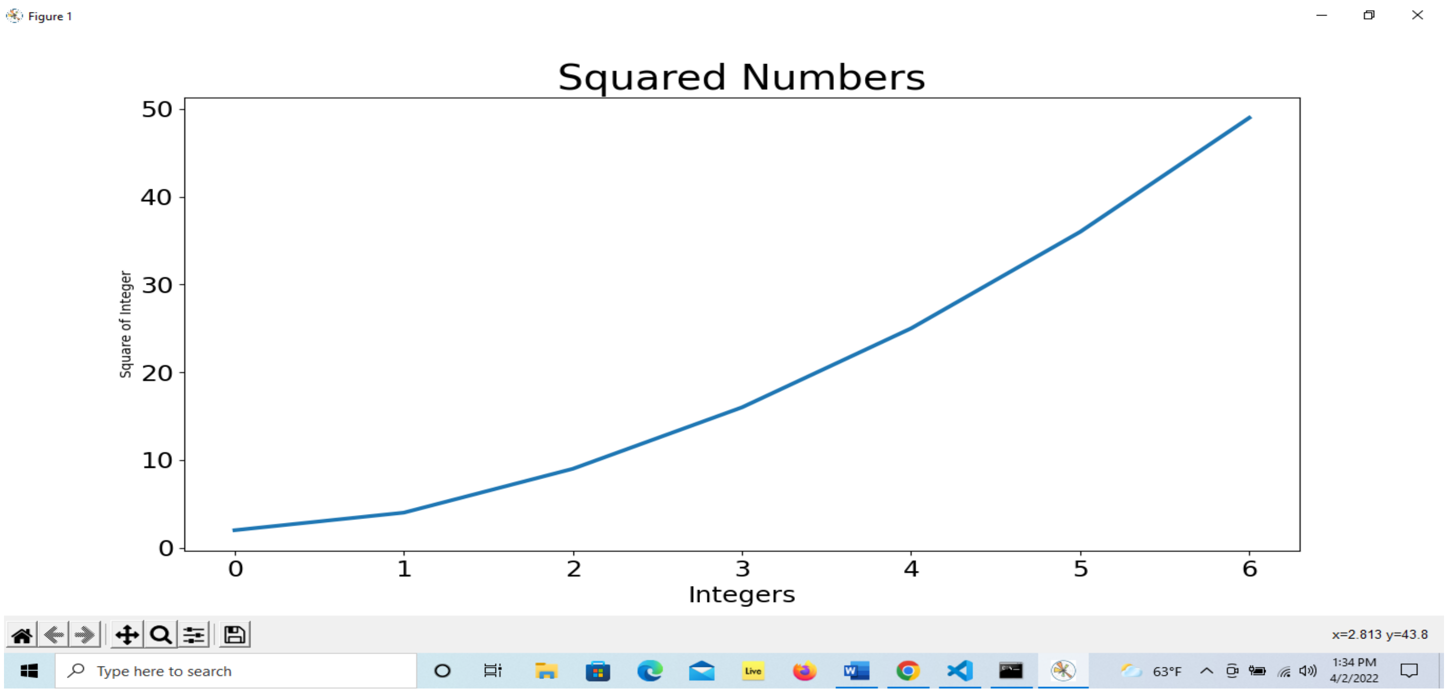
8. The above script produces a basic plot and is meaningless. We have no labels, and you might notice that python has placed some arbitrary values on the x-axis (horizontally). This is because when we provide an argument in the form of a single list or array for example, matplotlib assumes it is a sequence of 'y' values. Therefore, it automatically generates the 'x' values. Python ranges begin at 0. As you can see the x-vector is the same length as 'y' but starts with 0 by default.

9. We can, however, customize our plots to improve readability. For example, we can adjust the linewidth, increase or decrease the font size, and add titles and labels. Let's return to our original script and change a few things. Let's create a list of squares. That is a list containing 2, 4, 9, 16, 25, 36, 49.

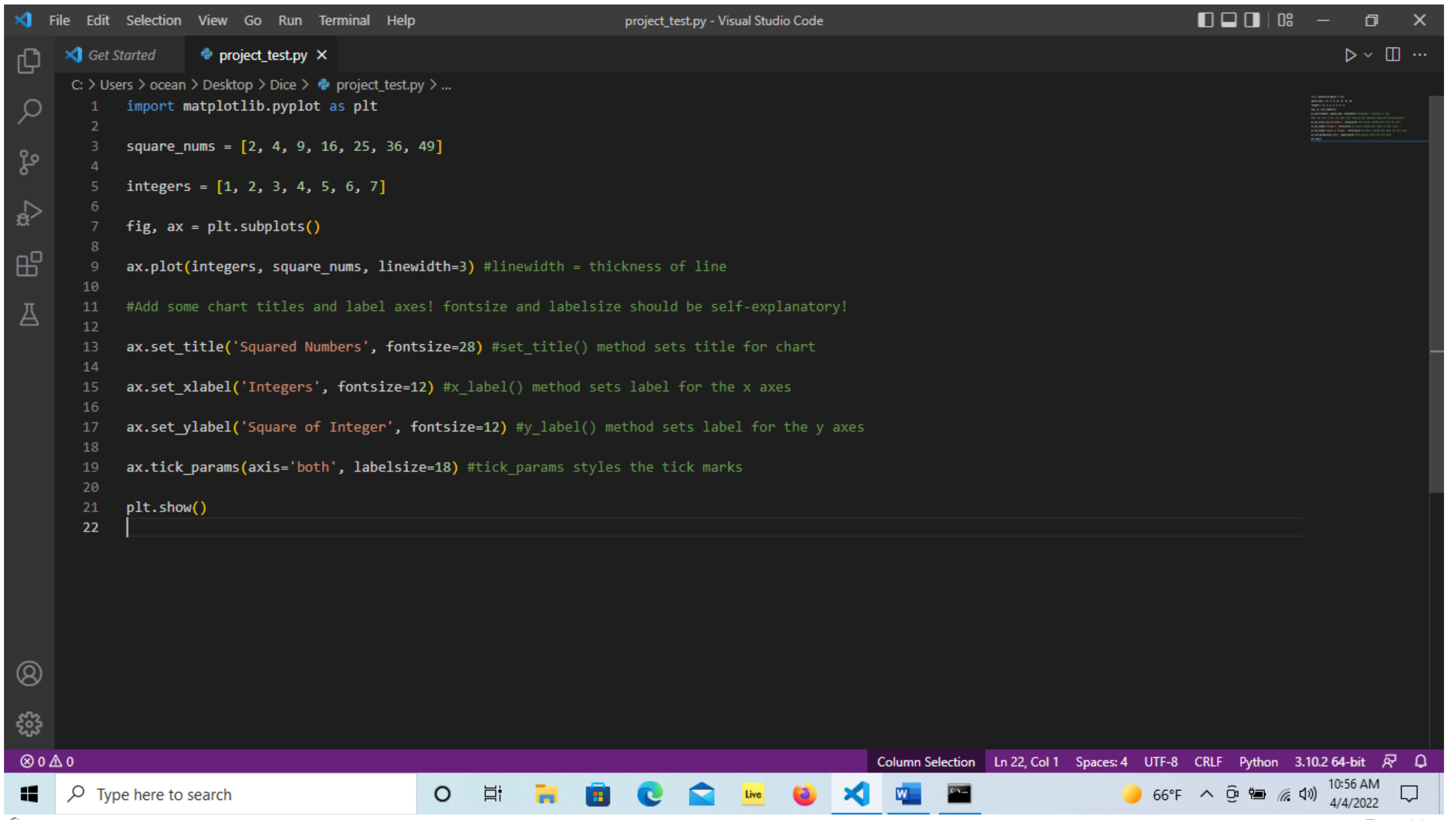


```python
import matplotlib.pyplot as plt

square_nums = [2, 4, 9, 16, 25, 36, 49]

fig, ax = plt.subplots()

ax.plot(square_nums, linewidth=3) #linewidth = thickness of line

#Add some chart titles and label axes! fontsize and labelsize should be self-explanatory!

ax.set_title('Squared Numbers', fontsize=28) #set_title() method sets title for chart

ax.set_xlabel('Integers', fontsize=12) #x_label() method sets label for the x axes

ax.set_ylabel('Square of Integer', fontsize=12) #y_label() method sets label for the y axes

ax.tick_params(axis='both', labelsize=18) #tick_params styles the tick marks

plt.show()

```

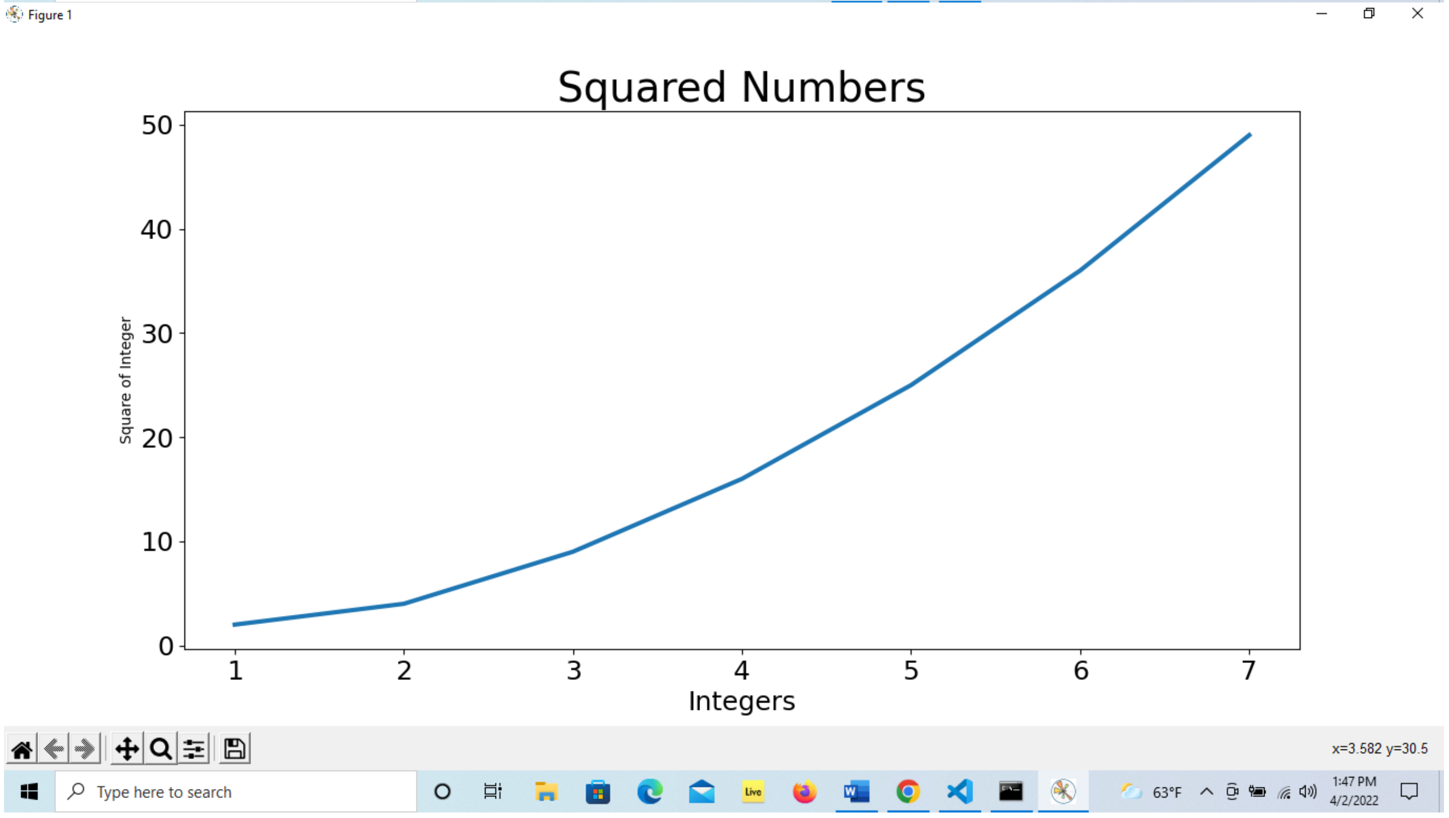
10. Now, when you run this script, it will plot the integers and the squares, but you'll notice that the data has been incorrectly plotted. If you look at the integer 6, you'll notice that the square is 49. Remember, when passing a sequence of numbers, Python assumes that the 'x' coordinate begins with 0.



11. Therefore, it is necessary to override Python's default behavior by passing additional arguments to the plot() method. This time we will pass additional arguments to the plot() method. We will pass some input values (integers) and the square of those integers. Therefore, we need an additional list to store our integers. For example:

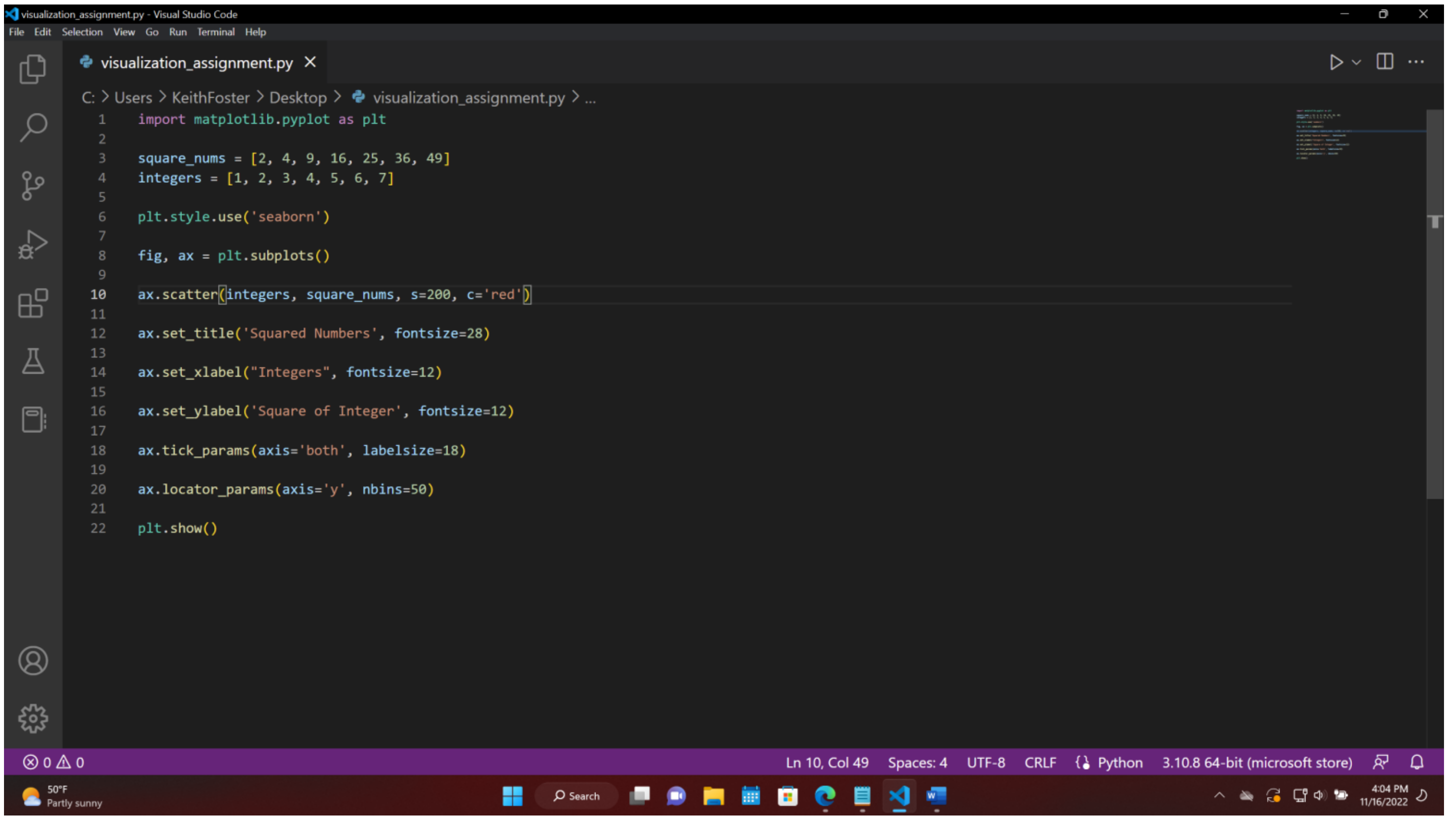


```python
import matplotlib.pyplot as plt

square_nums = [2, 4, 9, 16, 25, 36, 49]

integers = [1, 2, 3, 4, 5, 6, 7]

fig, ax = plt.subplots()

ax.plot(integers, square_nums, linewidth=3) #linewidth = thickness of line

#Add some chart titles and label axes! fontsize and labelsize should be self-explanatory!

ax.set_title('Squared Numbers', fontsize=28) #set_title() method sets title for chart

ax.set_xlabel('Integers', fontsize=12) #x_label() method sets label for the x axes

ax.set_ylabel('Square of Integer', fontsize=12) #y_label() method sets label for the y axes

ax.tick_params(axis='both', labelsize=18) #tick_params styles the tick marks

plt.show()

```

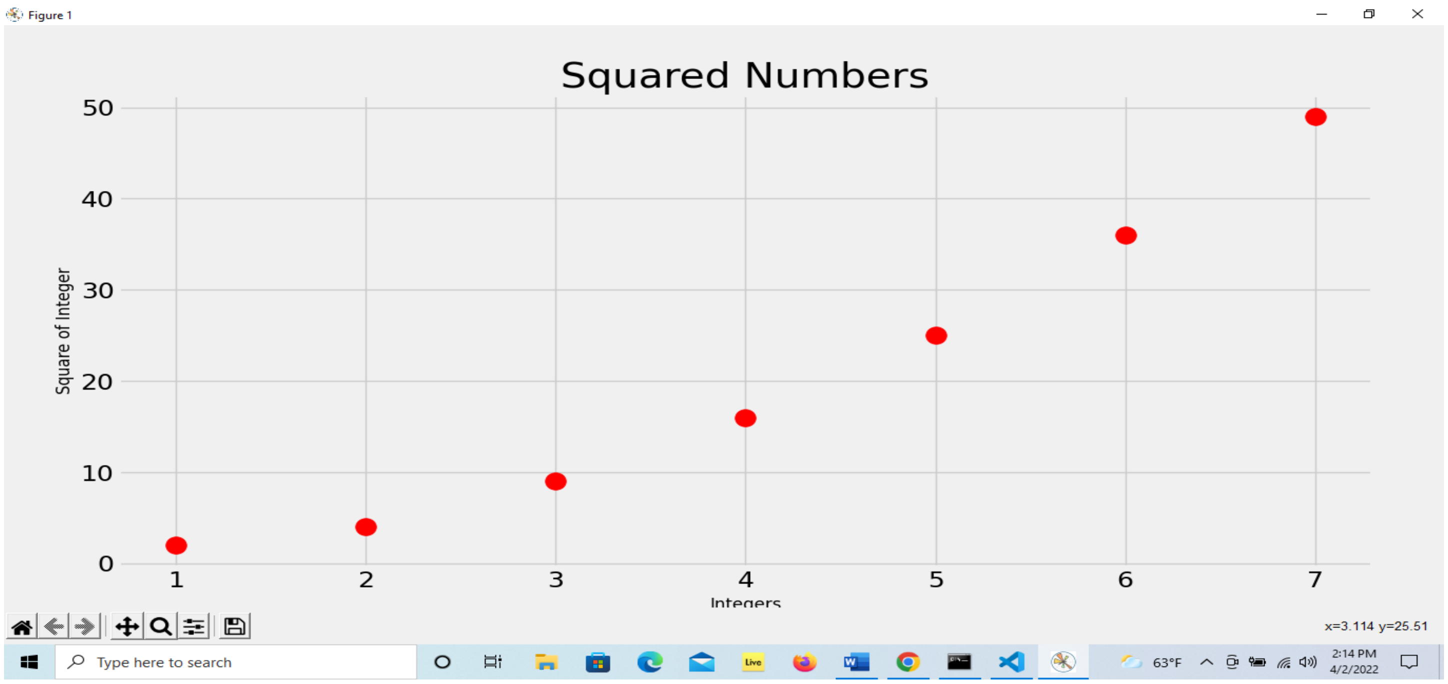


12. You should be able to tell that this chart could still be customized further. Fortunately, matplotlib contains many built-in styles that enable us to format the background, gridlines, and of course our line widths, font sizes, colors, type, and much more. You can refer to the documentation to better understand the different styles we can work with.

13. What if we wanted to visualize a scatter plot of our data? We can simply pass a different plot style and its associated parameters to '.ax.' For example:



```python
import matplotlib.pyplot as plt

square_nums = [2, 4, 9, 16, 25, 36, 49]
integers = [1, 2, 3, 4, 5, 6, 7]

plt.style.use('seaborn')

fig, ax = plt.subplots()

ax.scatter(integers, square_nums, s=200, c='red')

ax.set_title('Squared Numbers', fontsize=28)

ax.set_xlabel("Integers", fontsize=12)

ax.set_ylabel('Square of Integer', fontsize=12)

ax.tick_params(axis='both', labelsize=18)

ax.locator_params(axis='y', nbins=50)

plt.show()
```

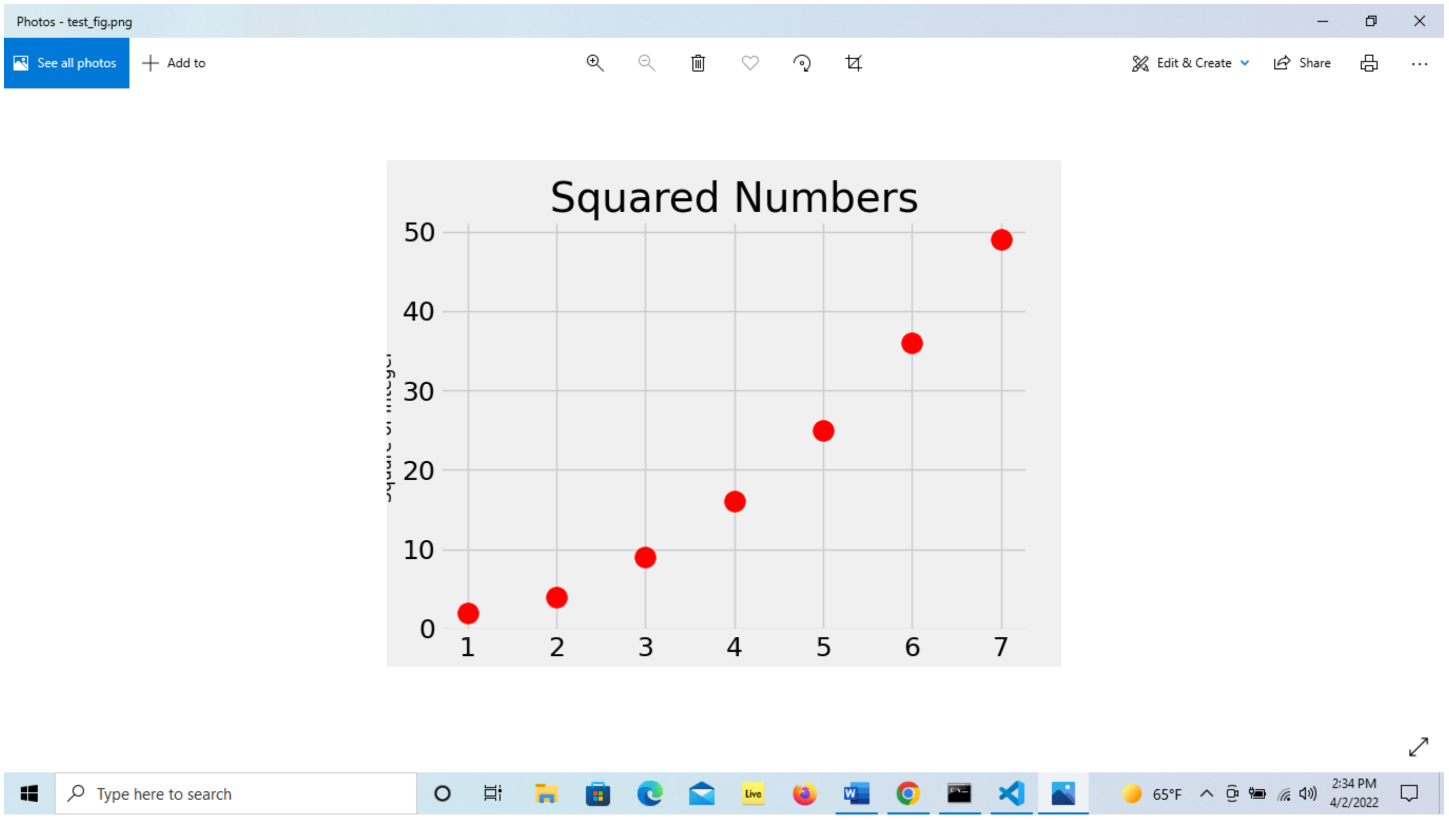


14. We often will want to save our figures and to do so is relatively easy. All we need to do to automatically save to a file is to replace our call to the show() function with a call to the savefig() function. Therefore, we could simply state:

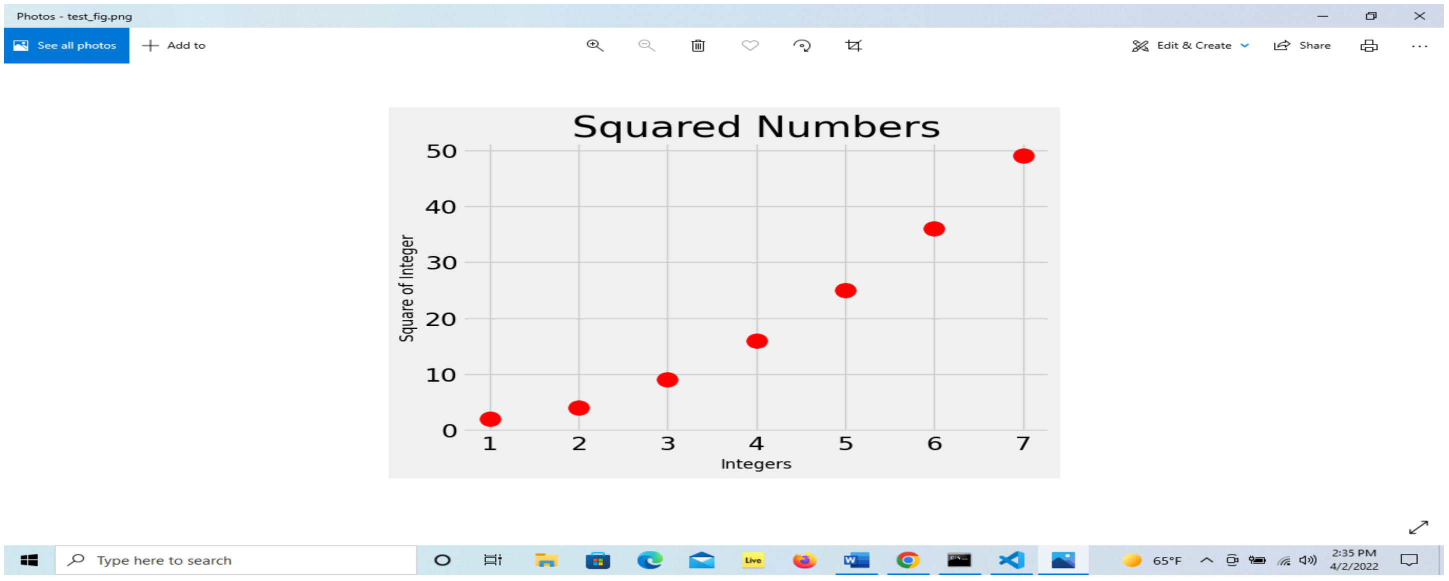
plt.savefig('square_nums.png') #If the file doesn't exist, python will create it!

15. If by chance, after you save the image and open it up and you can't view the labels, you can trim the unnecessary whitespace which is responsible for limiting the room for all of our labels. Therefore, pass a second argument in the savefig() function as: **plt.savefig('square_nums.png', bbox_inches= 'tight').**

Before Trim:

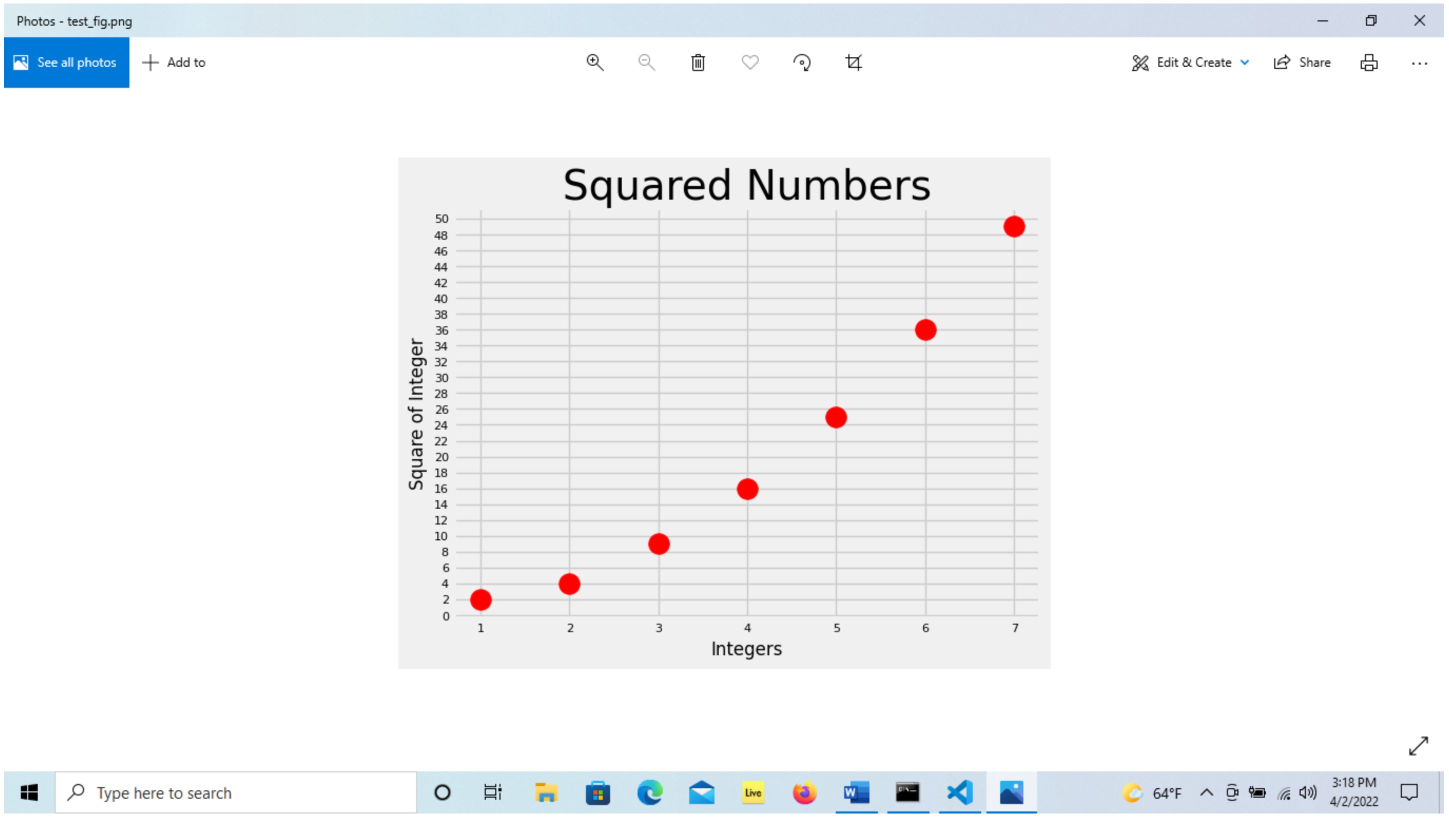


After Trim:



16. You'll notice that the y-axis's ticks are numbered by multiples of 10 (0, 10, 20…..). Perhaps you want to more clearly define where exactly on the plot the prime for each integer lies. There are several solutions to this problem, but one easy solution requires only one line of code. We can call matplotlib's locator_params() function which controls the behavior of Tick Locators. This might be useful when working with smaller datasets like in our case. We can specify which axis we want to work with and pass the appropriate parameters we want to change. In our case, let's just change the nbins for the y-axis. 'nbins' simply means the number of bins our data will be divided into. This is important when working with histograms, but in our case, we are only scattering our values. Therefore, we can say:

```
ax.locator_params(axis='y', nbins=50)
```



There you have it: A basic introduction to creating charts in matplotlib. Of course, this is only a basic introduction, and you should, if interested, refer to the documentation and online tutorials to delve further into this topic. For the scope of this class, I feel this is enough to help you complete your final assignment.

PLOTLY

1. Plotly is a scientific graphing library. We can use Plotly to create interactive, web and publication-quality graphics. Like matplotlib, we can create a variety of basic charts, statistical charts, scientific charts, financial charts, maps, AI and Machine Learning Graphics, Bioinformatic charts, 3D charts, animations, and much more.

2. Plotly is excellent for creating visualizations we wish to display in a browser. Plotly visualizations automatically scale to all types of screen sizes. Furthermore, these visualizations are interactive.

3. Plotly is not built into Python. Therefore, we need to install it, just like we did matplotlib. To do so, open a terminal (command prompt) and type the following script:

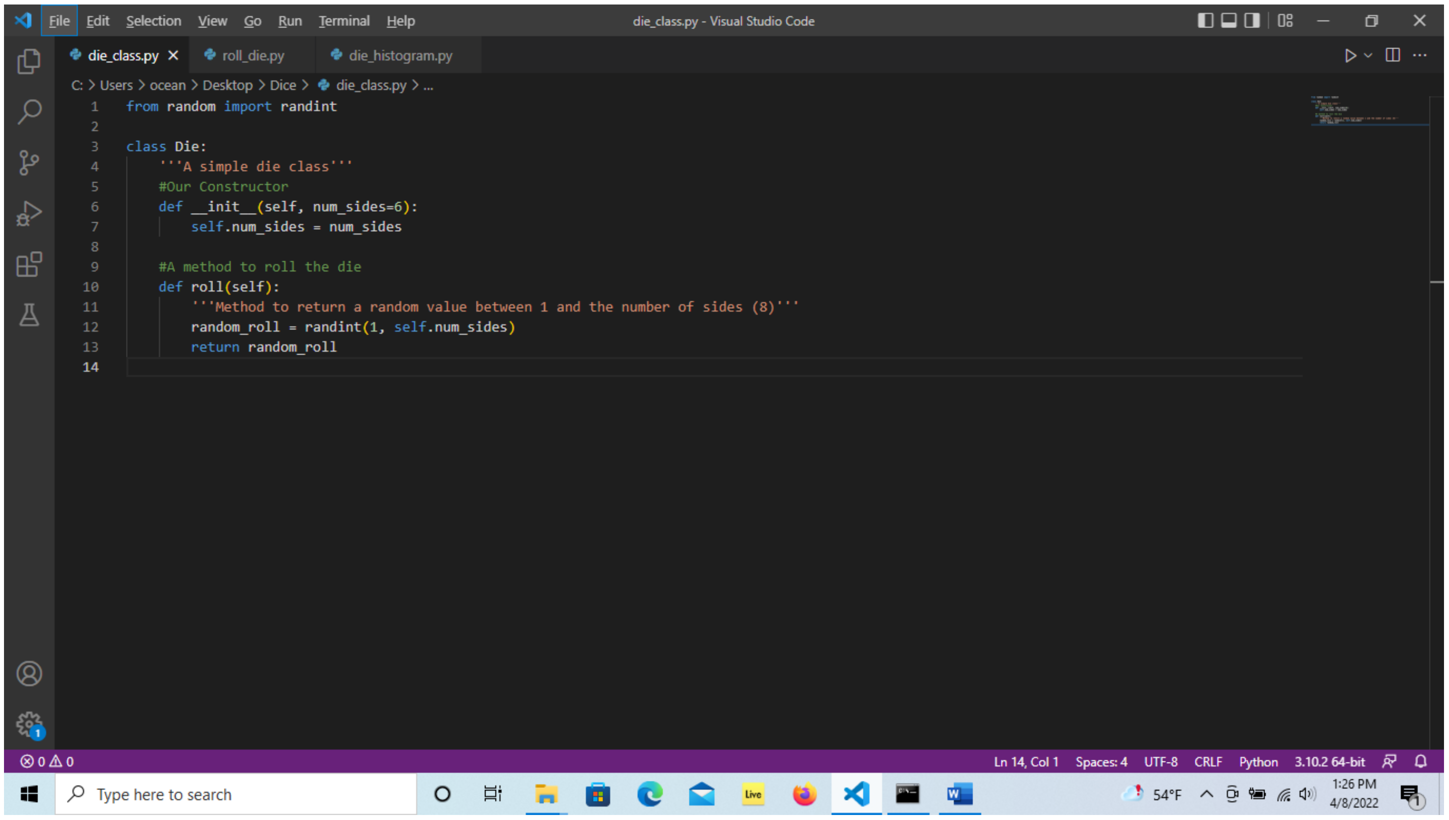
```
python -m pip install --user plotly
```

**Note – If you are using a different operating system such as macOS, and the installation fails, you can try omitting the "--user" flag. If it still fails, you need to search for a solution on your own.*
Also, the above command tells Python to install the plotly library to the current Python installation (working directory), meaning that if you are using a different version of Python such as Python3, you must use "python3" in your command.

4. Older versions of plotly.py included the functionality for creating graphics and figures online as well as offline. Online functionality has since been moved to the Chart Studio Cloud Service. Therefore, we will be exclusively working in offline mode, in that Version 4 and later is "off-line-only." We will be able to generate graphics offline and save them to our local machine. We will use the offline.plot() method which creates a standalone HTML saved locally, in which we can open in a browser.

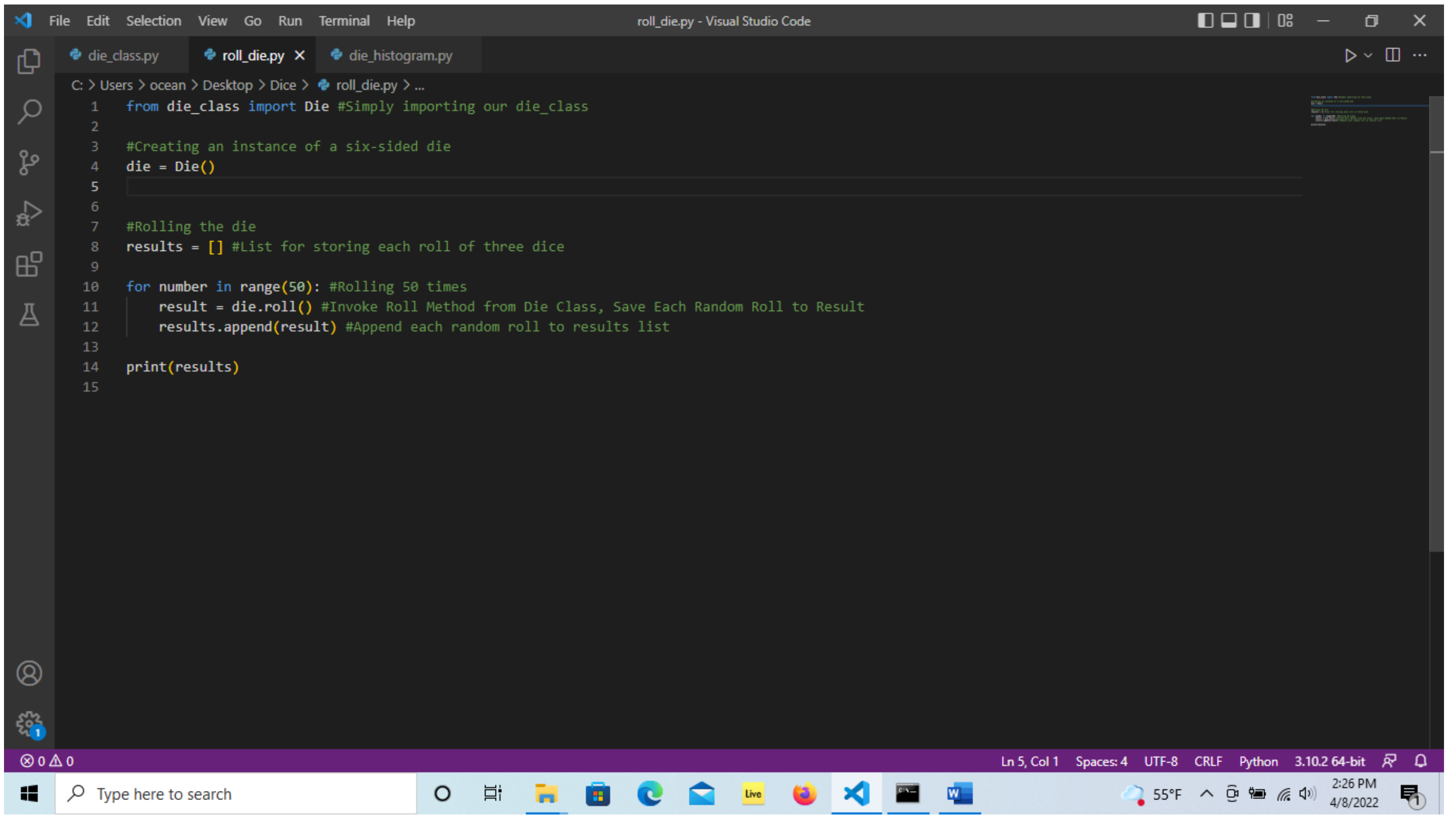
5. You will probably have to take statistics as part of your math curriculum in any CS, CIS major. One certainty is that your professor or instructor will introduce you to the different probabilities associated with rolling one or more dice. It's typically the default to introduce the ideas of probability. You'll learn that throwing a single 'die' results in outcomes that are equally probable. In that, rolling a 1 is just as probable as rolling a 7. However, the probabilities change when rolling two dice because there are more ways to roll some numbers than others. For example, there are six rolls that could result in 7, but there is only one way to throw snake eyes. Therefore, the odds of rolling a 7 is six times more likely to occur that rolling two "1's" and so on and so on. We will use the probability of throwing dice to better understand Plotly.

6. Let's start by rolling one six-sided die! Let's first create a die class with a couple of methods and save it as a Python module we can import later. We will be using Python's built-in random module.

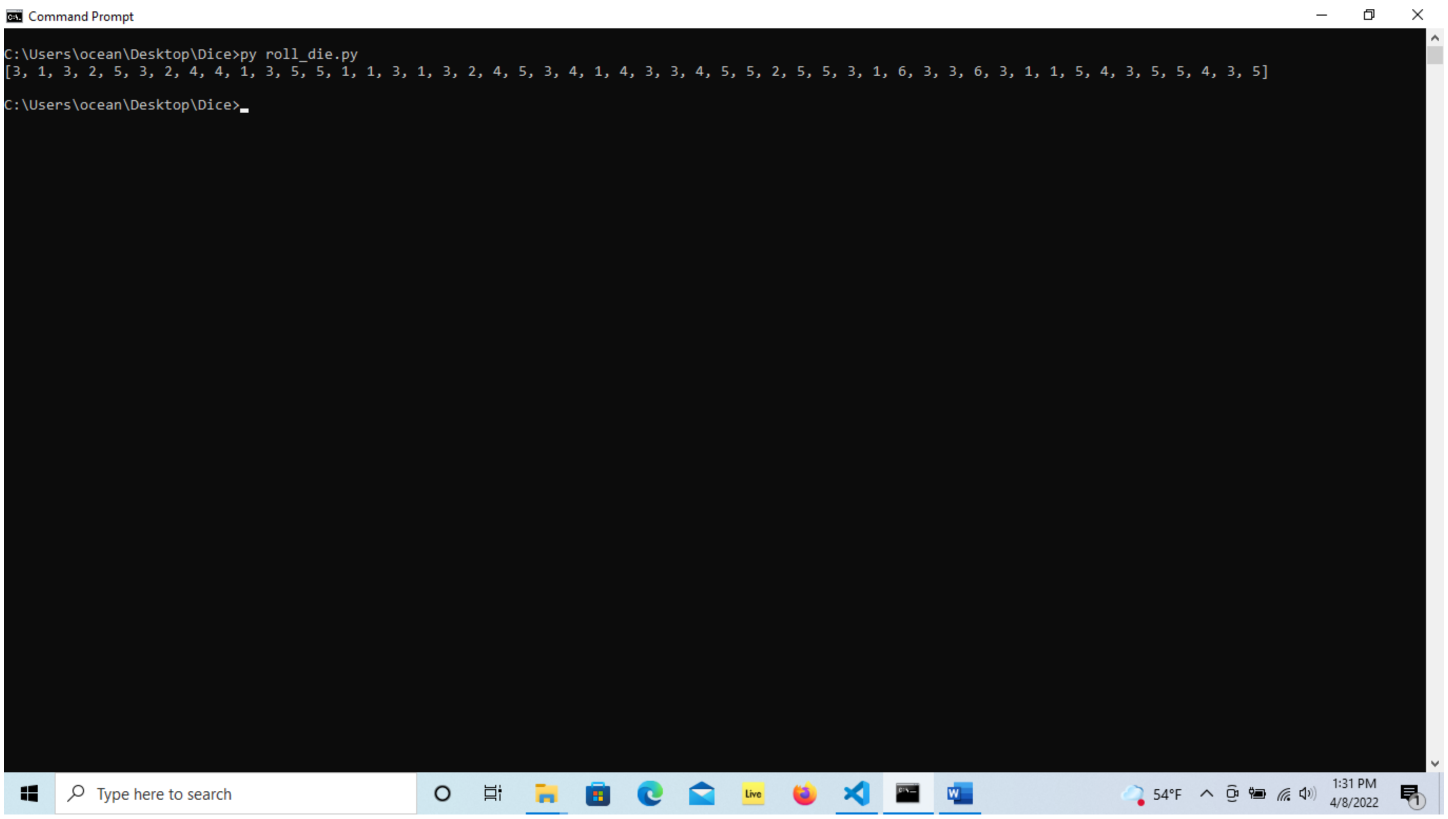

File Edit Selection View Go Run Terminal Help
die_class.py - Visual Studio Code
die_class.py
roll_die.py
die_histogram.py
C: > Users > ocean > Desktop > Dice > die_class.py > ...
Ln 14, Col 1 Spaces: 4 UTF-8 CRLF Python 3.10.2 64-bit

```python
from random import randint

class Die:
    '''A simple die class'''
    #Our Constructor
    def __init__(self, num_sides=6):
        self.num_sides = num_sides

    #A method to roll the die
    def roll(self):
        '''Method to return a random value between 1 and the number of sides (8)'''
        random_roll = randint(1, self.num_sides)
        return random_roll
```

7. We need to first make sure that our class and its corresponding methods are working properly. Therefore, before creating any visualizations, let's create a module called roll_die and import the die_class module. Make sure your classes and modules are stored in the same directory. We want to be sure that our results are reasonable and catch any errors before creating our visualization.

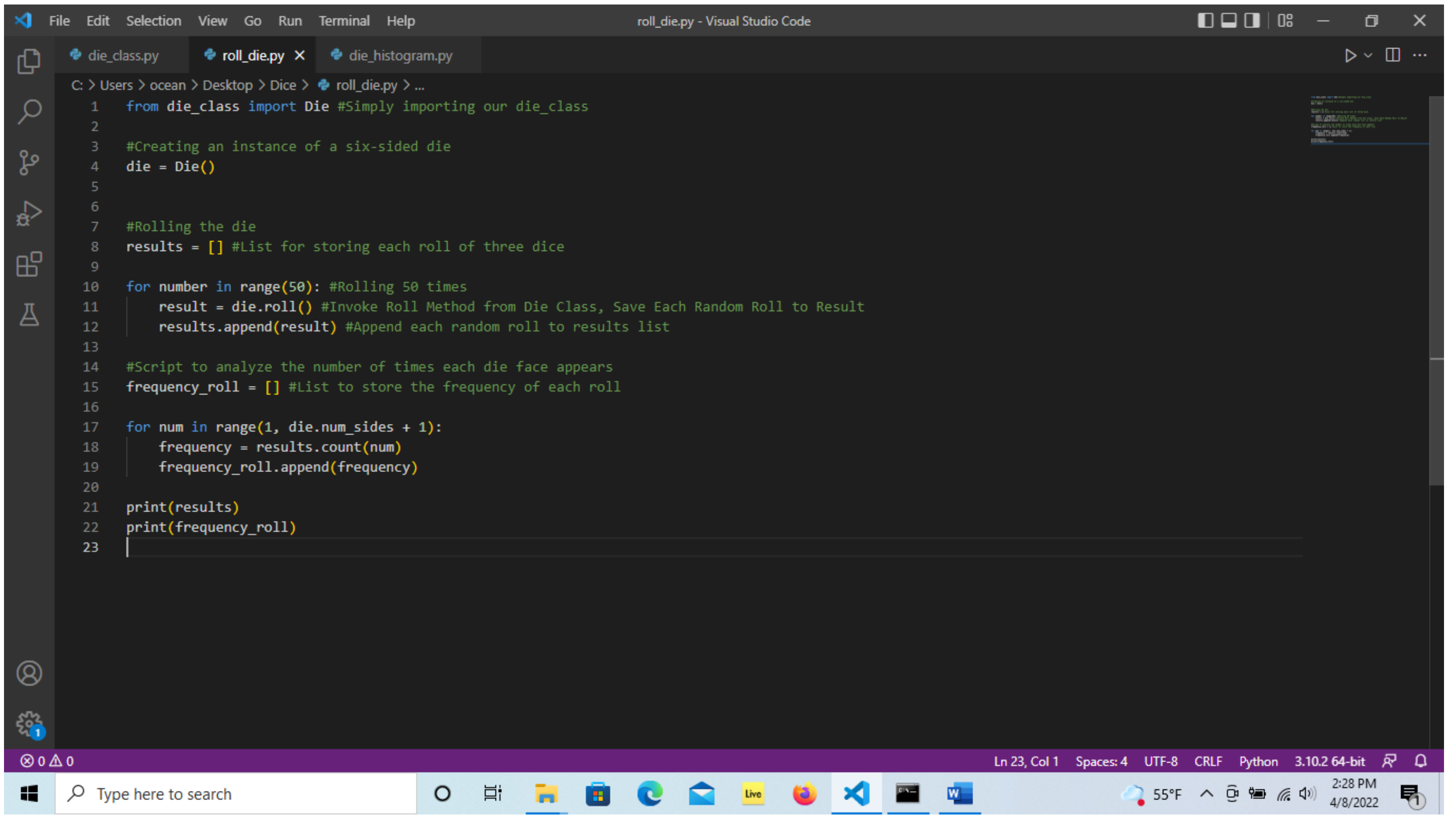


```python
from die_class import Die #Simply importing our die_class

#Creating an instance of a six-sided die
die = Die()


#Rolling the die
results = [] #List for storing each roll of three dice

for number in range(50): #Rolling 50 times
    result = die.roll() #Invoke Roll Method from Die Class, Save Each Random Roll to Result
    results.append(result) #Append each random roll to results list

print(results)
```



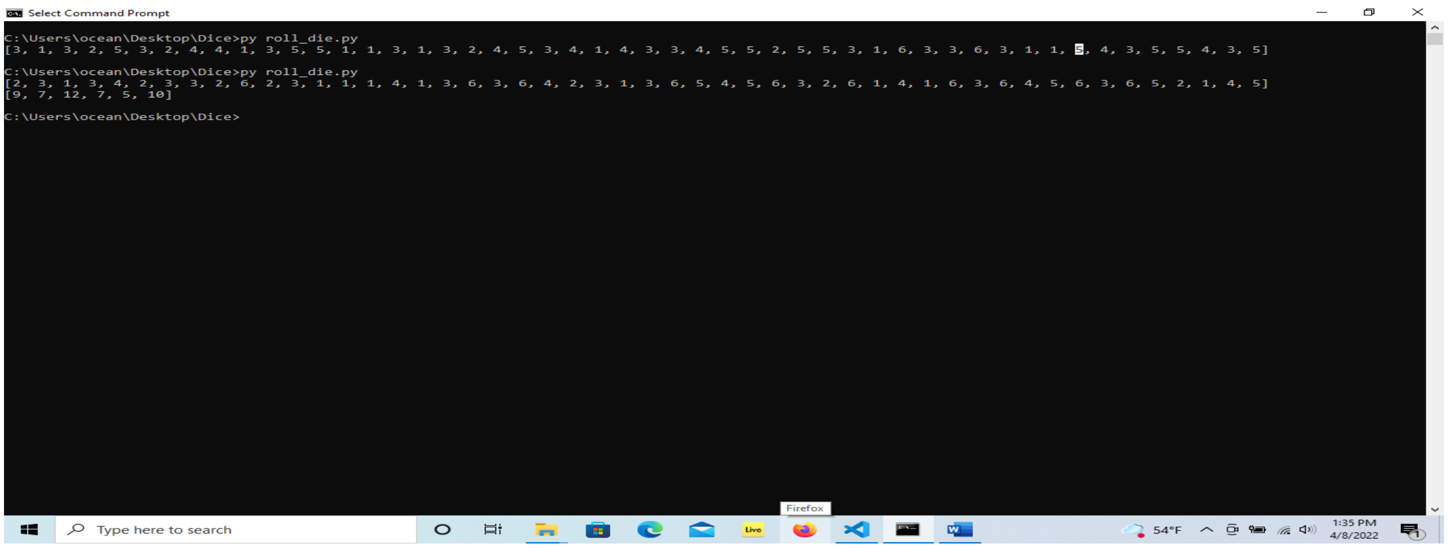
```
C:\Users\ocean\Desktop\Dice>py roll_die.py
[3, 1, 3, 2, 5, 3, 2, 4, 4, 1, 3, 5, 5, 1, 1, 3, 1, 3, 2, 4, 5, 3, 4, 1, 4, 3, 3, 4, 5, 5, 2, 5, 5, 3, 1, 6, 3, 3, 6, 3, 1, 1, 5, 4, 3, 5, 5, 4, 3, 5]

C:\Users\ocean\Desktop\Dice>_
```

8. As you can see after running roll_die.py, everything seems to be working accordingly. We can tell that each random roll produces a number between 1 and 6. We can tell by observing the output that none of our results are out of range and that all possible outcomes can be observed. Now we can determine how many times each die face appears in our results. In the same block of code in our roll_die.py module, let’s amend the script to count the frequency of each roll.



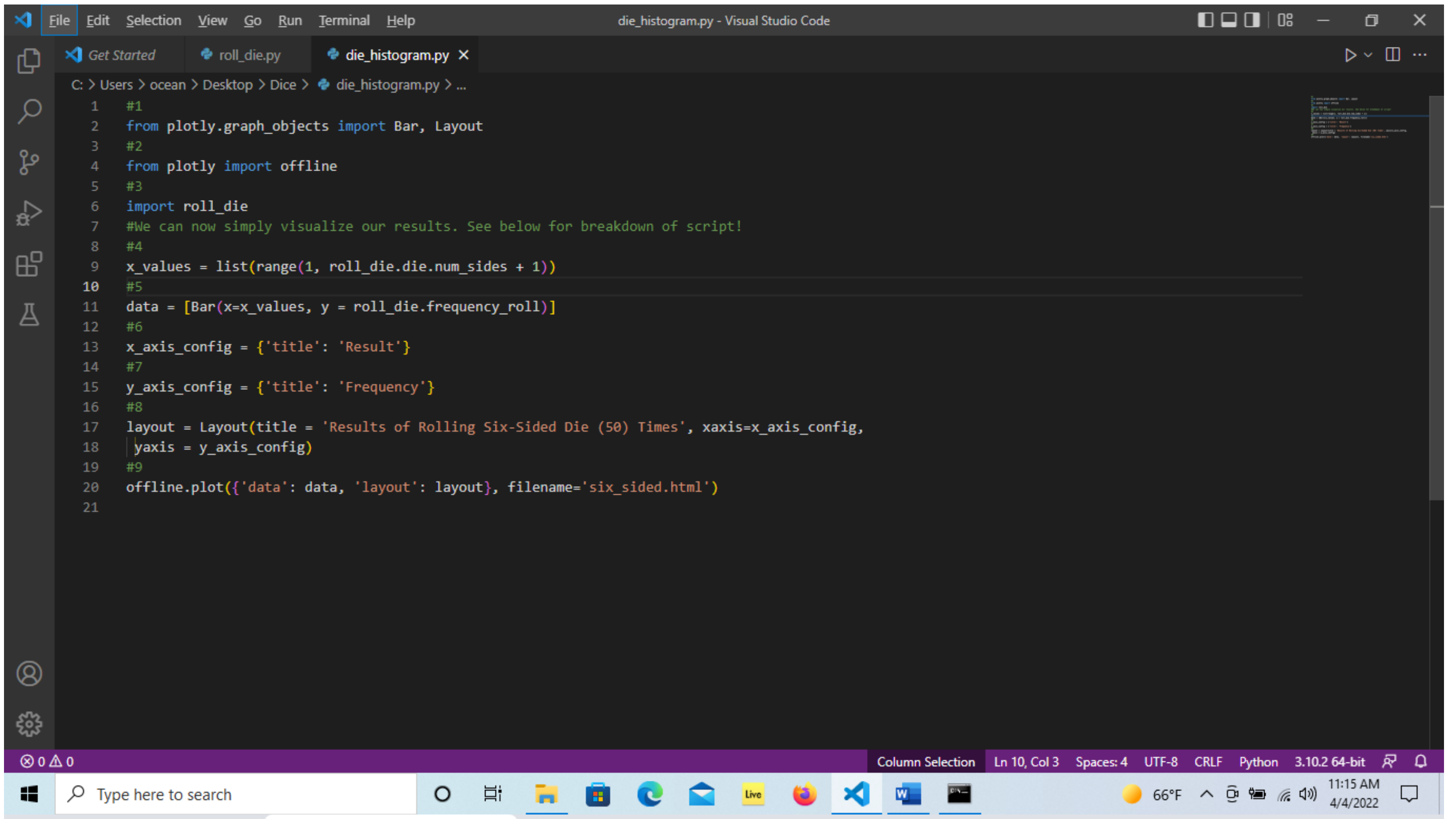
```python
from die_class import Die #Simply importing our die_class

#Creating an instance of a six-sided die
die = Die()


#Rolling the die
results = [] #List for storing each roll of three dice

for number in range(50): #Rolling 50 times
    result = die.roll() #Invoke Roll Method from Die Class, Save Each Random Roll to Result
    results.append(result) #Append each random roll to results list

#Script to analyze the number of times each die face appears
frequency_roll = [] #List to store the frequency of each roll

for num in range(1, die.num_sides + 1):
    frequency = results.count(num)
    frequency_roll.append(frequency)

print(results)
print(frequency_roll)

```

9. As you can see after running roll_die.py again everything seems to be working accordingly. We can tell that each random roll produces a number between 1 and 6. We can tell by observing the output that we’ve produced the number of times each face value was rolled. Because this list is short, you can clearly add up each list item in your head. You’ll notice they all add up to 50 rolls. Each list item represents the die face.

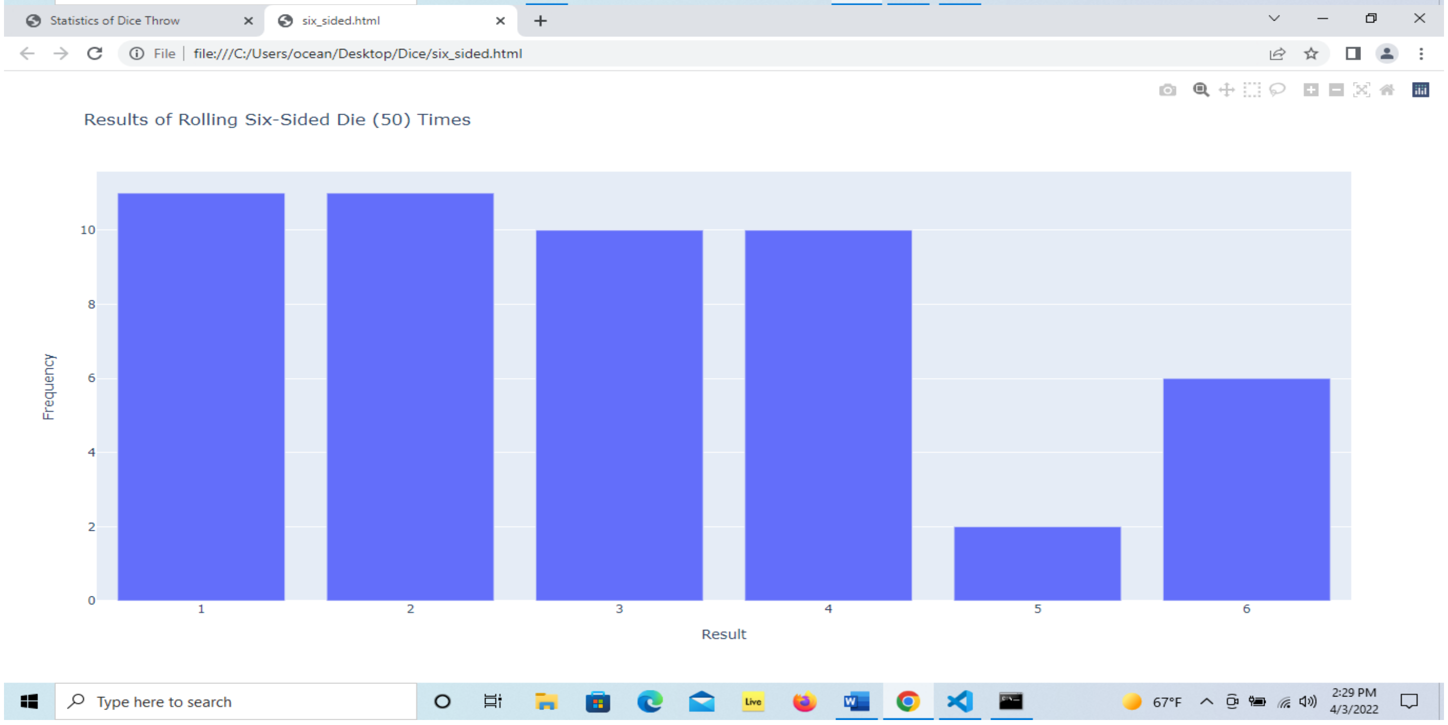


```
C:\Users\ocean\Desktop\Dice>py roll_die.py
[3, 1, 3, 2, 5, 3, 2, 4, 4, 1, 3, 5, 5, 1, 1, 3, 1, 3, 2, 4, 5, 3, 4, 1, 4, 3, 3, 4, 5, 5, 2, 5, 5, 3, 1, 6, 3, 3, 6, 3, 1, 1, 5, 4, 3, 5, 5, 4, 3, 5]

C:\Users\ocean\Desktop\Dice>py roll_die.py
[2, 3, 1, 3, 4, 2, 3, 3, 2, 6, 2, 3, 1, 1, 1, 4, 1, 3, 6, 3, 6, 4, 2, 3, 1, 3, 6, 5, 4, 5, 6, 3, 2, 6, 1, 4, 1, 6, 3, 6, 4, 5, 6, 3, 6, 5, 2, 1, 4, 5]
[9, 7, 12, 7, 5, 10]

C:\Users\ocean\Desktop\Dice>
```

10. We’ve now verified that our modules are working correctly. We can visualize these results and will do so using Plotly to create a histogram (bar chart). This visualization will demonstrate how often certain results are represented in the randomized rolls. We will need to import some graph objects from Plotly, and also be sure to let Python know that we want to import plotly in offline mode. Therefore, we will create a new module in the same folder where die_class.py and roll_die.py modules reside. We’ll call it die_histogram.py. Also, we need to import our roll_die.py module. Remember, in this module, we’ve already created an instance of the Die class, so there is no need to import it. REMEMBER, once we run the program again, it’s going to create a new list of randomized die rolls. Don’t refer to the output above.


File Edit Selection View Go Run Terminal Help
die_histogram.py - Visual Studio Code
Get Started
roll_die.py
die_histogram.py
C: > Users > ocean > Desktop > Dice > die_histogram.py > ...
Column Selection Ln 10, Col 3 Spaces: 4 UTF-8 CRLF Python 3.10.2 64-bit
Type here to search
66°F 11:15 AM 4/4/2022

```python
#1
from plotly.graph_objects import Bar, Layout
#2
from plotly import offline
#3
import roll_die
#We can now simply visualize our results. See below for breakdown of script!
#4
x_values = list(range(1, roll_die.die.num_sides + 1))
#5
data = [Bar(x=x_values, y = roll_die.frequency_roll)]
#6
x_axis_config = {'title': 'Result'}
#7
y_axis_config = {'title': 'Frequency'}
#8
layout = Layout(title = 'Results of Rolling Six-Sided Die (50) Times', xaxis=x_axis_config,
 yaxis = y_axis_config)
#9
offline.plot({'data': data, 'layout': layout}, filename='six_sided.html')
```



11. Let’s break down the above script one line at a time.

1. The module ploty_graph_objs contains each class definition for the object we need to create the plots we want to visualize. Therefore, the Bar Class represents any arbitrary dataset that can be formatted as a Bar Chart. The Bar Class requires that we pass as arguments, x and y values, each as a list. The Layout Class returns an object that specifies the configuration of a graph as well as the layout. Essentially with Layout() we can define the looks of our graph, format the topography and titles just to name a few.

2. As already mentioned, Older versions of plotly.py included the functionality for creating graphics and figures online as well as offline. Online functionality has since been moved to the Chart Studio Cloud Service. Therefore, we will be exclusively working in offline mode, in that Version 4 and later is “off-line-only.” We will be able to generate graphics offline and save them to our local machine. We will use the offline.plot() method which creates a standalone HTML saved locally, in which we can open in a browser.

3. Simply importing our roll_die module!

4. Since we need to create a bar for each possible outcome, we will store these in a list called x_values. Remember on graph, x_values by default run horizontally, while y_values run vertically. Nonetheless, our list of values starts at 1 and ends based on the number of sides of our die. We have 6, but the number of sides could be any number for that matter. It’s important to remember that Plotly will not accept any arguments passed to a range() function by default. This is why we must pass these values to a list.

5. The variable data stores the data set that should be formatted as a bar chart. In our case we are passing the x_values from line 4, as well as the frequency of the rolls. Notice, we’ve used the dot notation. Our object is roll_die and we are telling Python that we require access to a particular property that belongs to this object. In our case it’s the frequency_roll list we created.

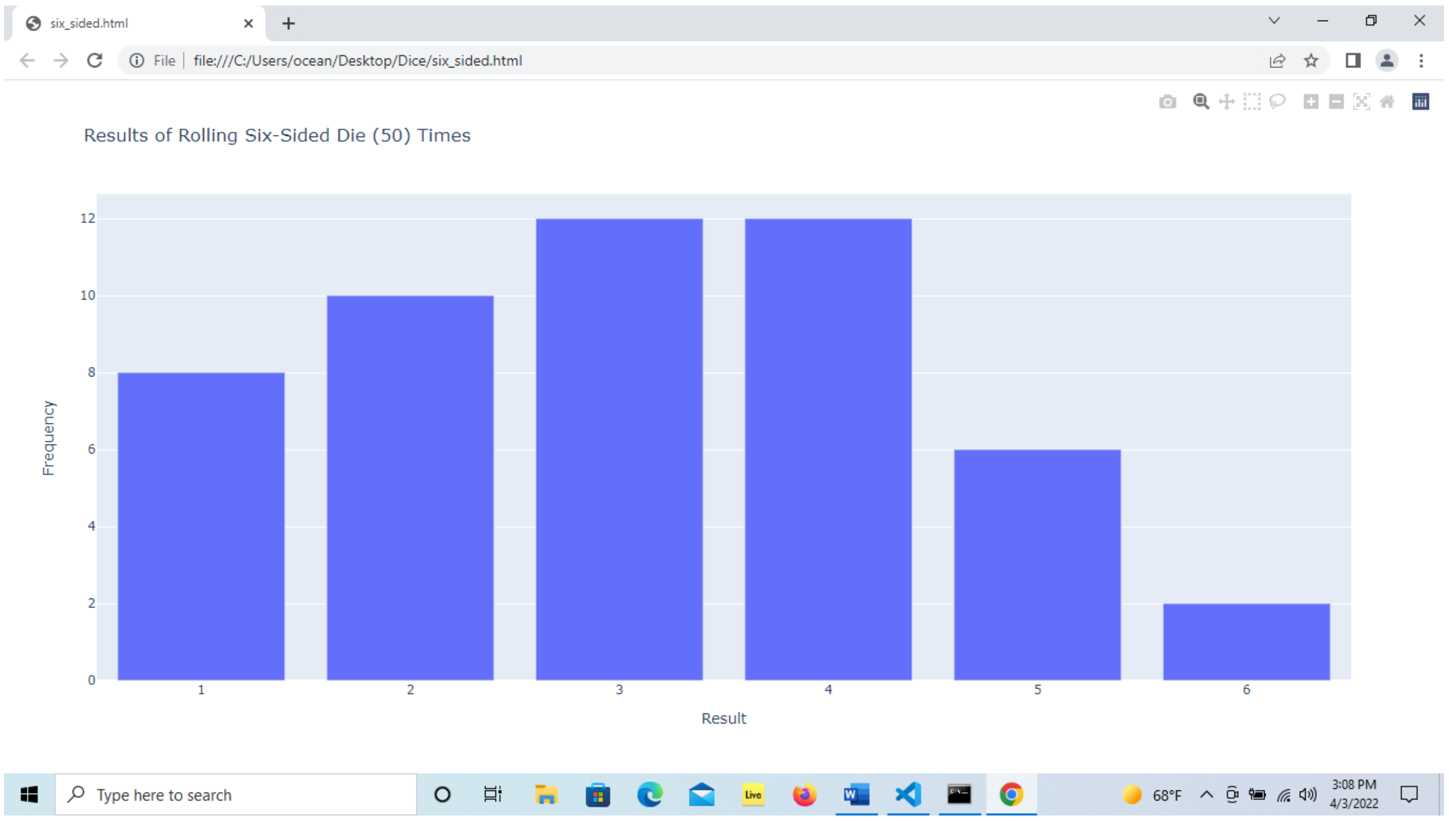
6. We can configure each axis any way we like. We store each configuration option in a dictionary. In our case, we have simply set the title of the x axis.

7. Same as 6 above, be in this case we have simple set the title of the y axis.

8. As already mentioned, we can configure the entire graph based on a variety of different parameters. In our case we are simply creating a title for our graph, and also passing as arguments the x_axis and y_axis configurations we defined in lines 6 and 7.

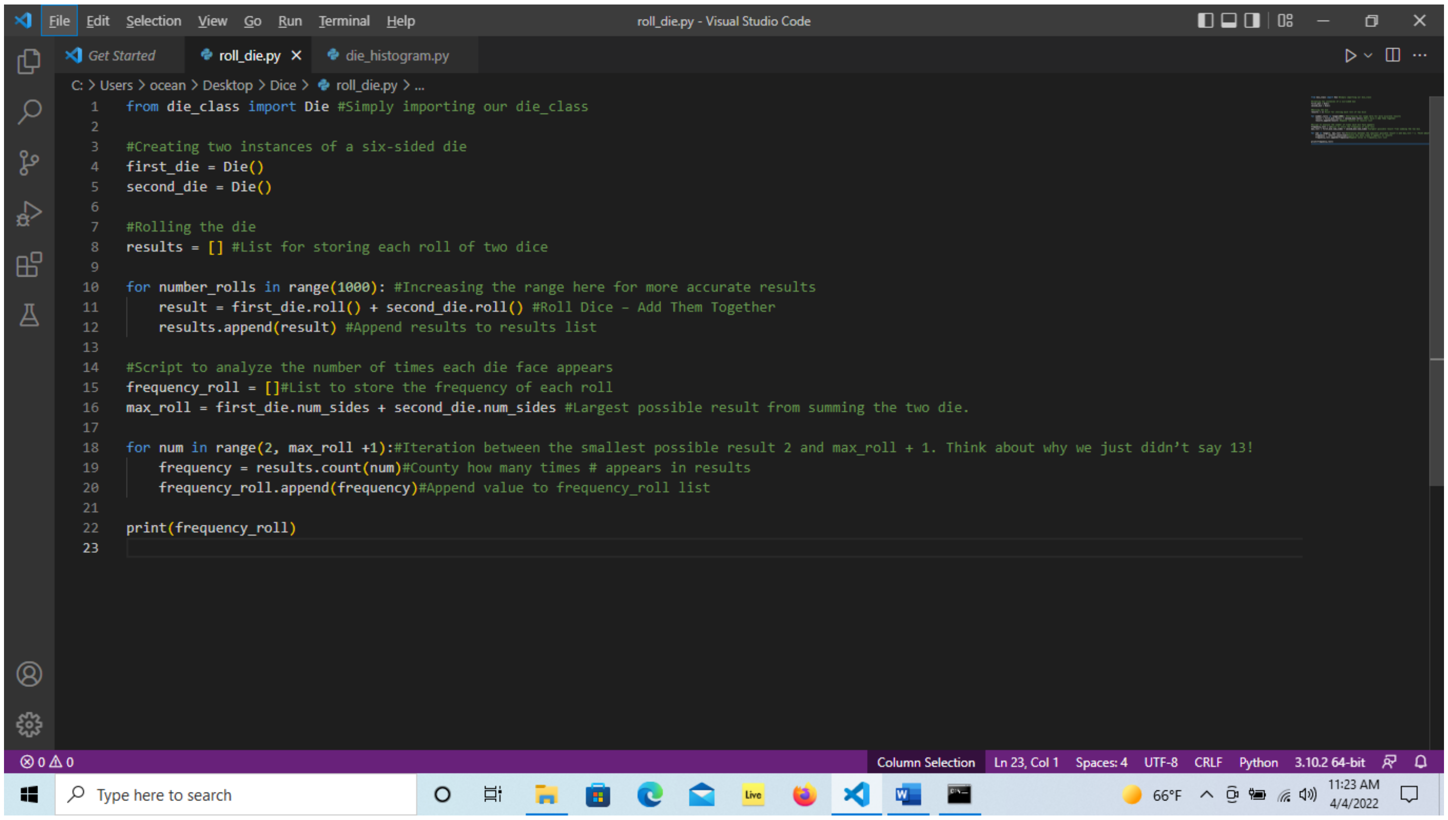
9. Here we are calling the offline.plot() function and it requires that a dictionary be passed containing data and layout objects. We have also created an html file, where we are saving the output. Therefore, our graph is simply saved as a html files called six_sided.html.

12. When we run our die_histogram.py module you might be prompted to choose a browser for opening. Any should work, but if your first option doesn't, try another browser. Also, if this process doesn't open the file automatically, you can open any browser first, and then open the file from the folder you've saved the modules and html file in. Nonetheless, once complete you should see a chart similar to the one below. As I've already mentioned this chart is interactive. Try for yourself to interact with the object.

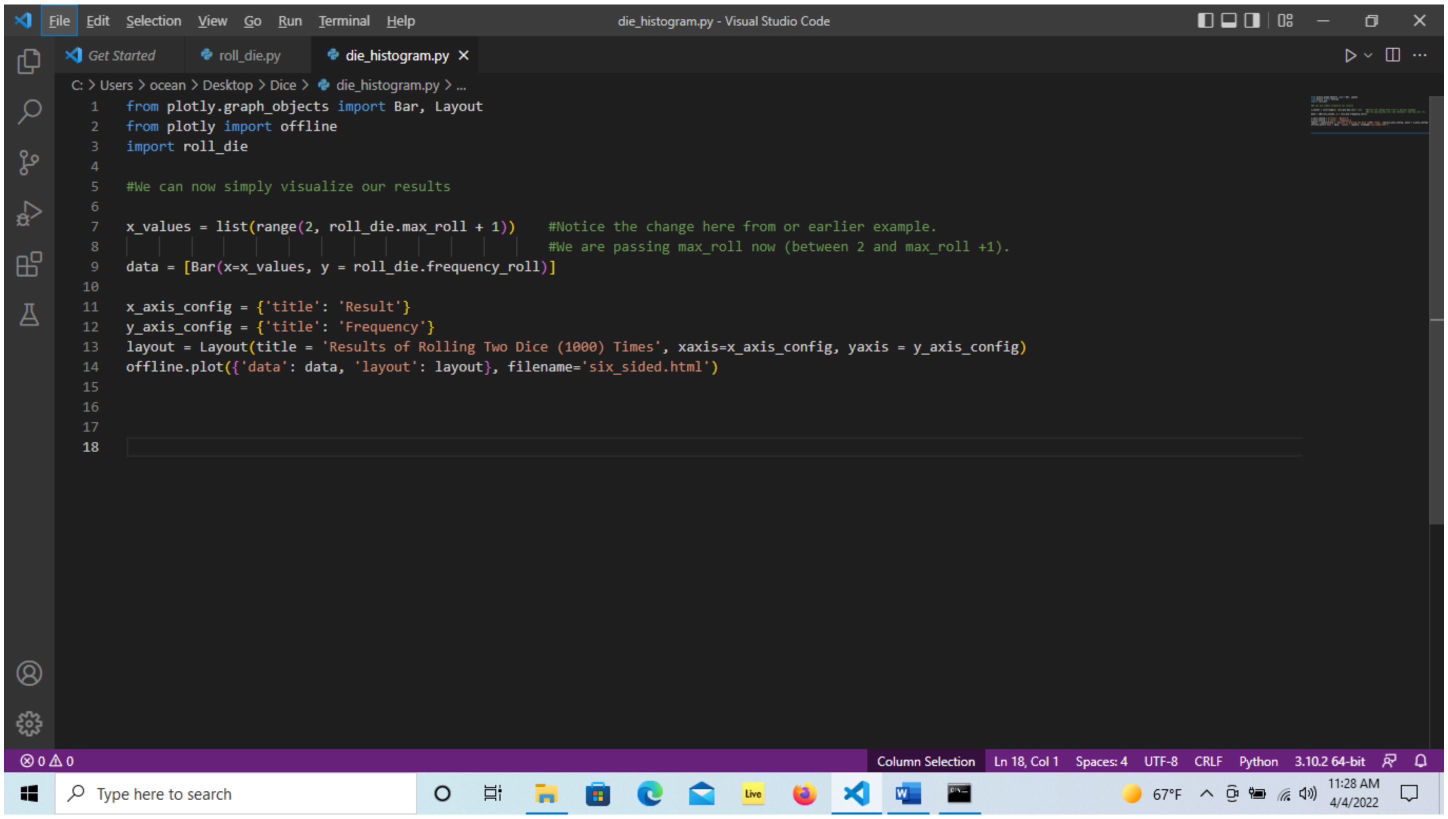


13. Of course, as I mentioned earlier, rolling one die results in a probability that each side will be rolled just as often as the others. However, as you can see in our results this doesn't hold true. This is because a small sample size(number of rolls) will result in a deviation away from the normal distribution. Therefore, try increasing the sample size to say 1000, and run the program again. This time you will clearly see that the probability across all rolls is a more accurate representation of what we would expect from rolling one die multiple times.

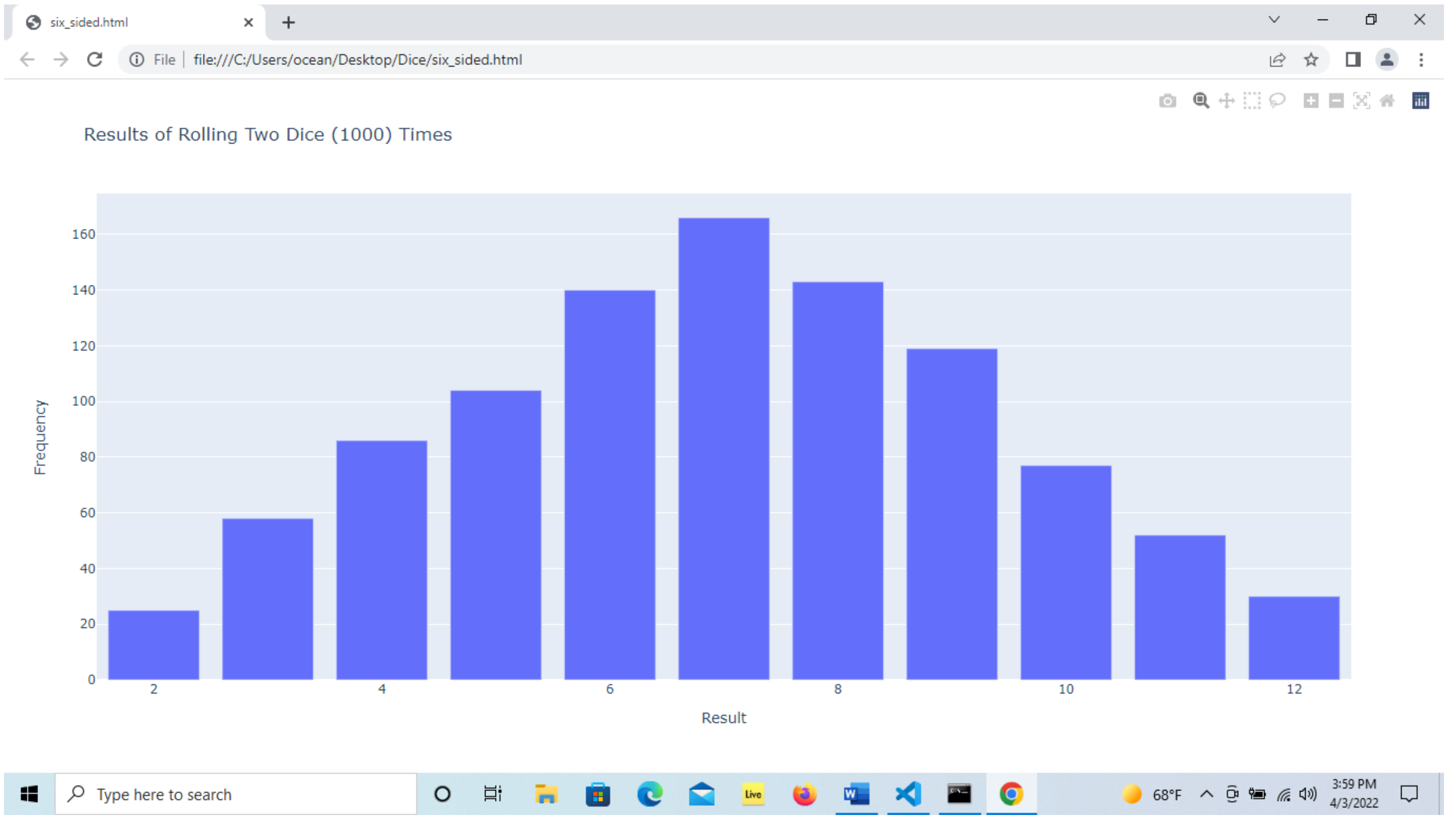
14. What we really want to see is what happens when we roll two dice. Remember there are a certain number of outcomes that will happen more than others (like rolling 7 more often than 2). We don't have to change our code too much but there are some changes we need to make. For example, we need to instantiate an additional Die() object in our roll_die.py module. We will call these dice first_die and second_die. We will also need to edit our code to analyze the results. We will still use the frequency_roll list to store our results with some minor changes in our for loop.



```python
from die_class import Die #Simply importing our die_class

#Creating two instances of a six-sided die
first_die = Die()
second_die = Die()

#Rolling the die
results = [] #List for storing each roll of two dice

for number_rolls in range(1000): #Increasing the range here for more accurate results
    result = first_die.roll() + second_die.roll() #Roll Dice - Add Them Together
    results.append(result) #Append results to results list

#Script to analyze the number of times each die face appears
frequency_roll = []#List to store the frequency of each roll
max_roll = first_die.num_sides + second_die.num_sides #Largest possible result from summing the two die.

for num in range(2, max_roll +1):#Iteration between the smallest possible result 2 and max_roll + 1. Think about why we just didn't say 13!
    frequency = results.count(num)#County how many times # appears in results
    frequency_roll.append(frequency)#Append value to frequency_roll list

print(frequency_roll)
```


File Edit Selection View Go Run Terminal Help
die_histogram.py - Visual Studio Code
Get Started
roll_die.py
die_histogram.py
C: > Users > ocean > Desktop > Dice > die_histogram.py > ...
Column Selection
Ln 18, Col 1
Spaces: 4
UTF-8
CRLF
Python
3.10.2 64-bit
Type here to search
67°F
11:28 AM
4/4/2022

```python
from plotly.graph_objects import Bar, Layout
from plotly import offline
import roll_die

#We can now simply visualize our results

x_values = list(range(2, roll_die.max_roll + 1))    #Notice the change here from or earlier example.
                                                    #We are passing max_roll now (between 2 and max_roll +1).
data = [Bar(x=x_values, y = roll_die.frequency_roll)]

x_axis_config = {'title': 'Result'}
y_axis_config = {'title': 'Frequency'}
layout = Layout(title = 'Results of Rolling Two Dice (1000) Times', xaxis=x_axis_config, yaxis = y_axis_config)
offline.plot({'data': data, 'layout': layout}, filename='six_sided.html')
```



15. Notice how it is more probable to roll a seven than any other combination. Pretty interesting. Also notice the normal distribution (The Bell Curve). You'll learn about this stuff in statistics if you haven't already taken it. Normal distribution is one of the most important concepts in statistics.

There you have it: A basic introduction to creating visualizations with Plotly. Of course, this is only a basic introduction, and you should, if interested, refer to the documentation and online tutorials to delve further into this topic. For the scope of this class, I feel this is enough to help you complete your final assignments.