Welcome to Class

Sunday, October 6, 2019 4:47 PM

Overview

Today's class will introduce students to the basics of Matplotlib, one of the most popular Python plotting libraries in use today

Class Objectives

- Understand Matplotlib's pyplot interface.
- Be able to create line; bar; scatter; and pie charts.
- Be familiar with basic plot configuration options, such as xlim and ylim.

Introduction to Matplotlib

Sunday, October 6, 2019 4:52 PM

Today's lesson will focus in particular on familiarizing students with the basics of a module called PyPlot, which can be used to create simple charts quickly

Open ExponentialChart.ipynb

- The NumPy library is oftentimes used alongside PyPlot. This package contains plenty of built-in methods which allow for simple scientific computing.

```
# Import Numpy for calculations and matplotlib for charting
import numpy as np
import matplotlib.pyplot as plt
```

- **np.arange(start, end, step)** creates a list of numbers from **start** to **end**, where each number in the list is **step** away from the next ones.

```
# Creates a list from 0 to 5 with each step being 0.1 higher than the last
x_axis = np.arange(0, 5, 0.1)
x_axis
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1, 1.2,
       1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 2. , 2.1, 2.2, 2.3, 2.4, 2.5,
       2.6, 2.7, 2.8, 2.9, 3. , 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8,
       3.9, 4. , 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9])
```

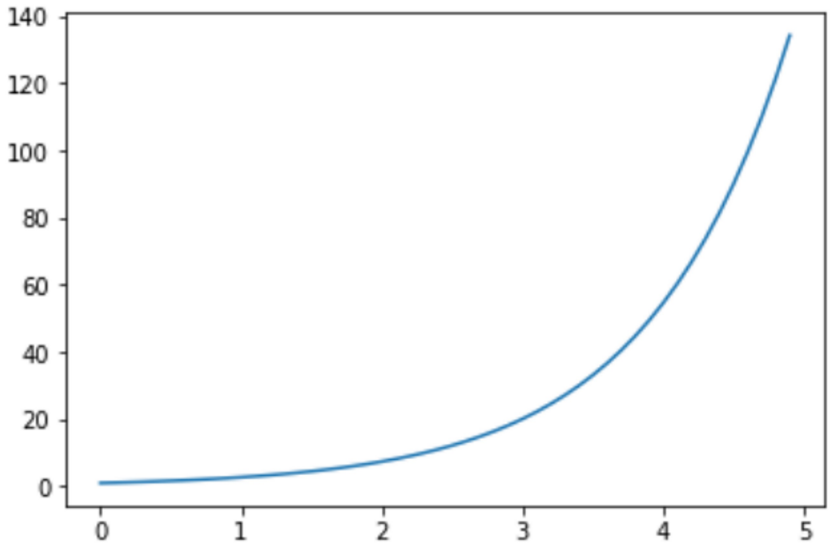
- **e_x** list is being created using a "list comprehension". List comprehensions allow lists to be created using mathematic formulae. This example takes values from the **x_axis** list one at a time, finds the exponential of them, and stores the response within a list.
- **Np.exp()** calculates e^x for each value of the x array (x_axis)
  - e is the mathematical constant called Euler's number (2.718281)

```
# Creates an exponential series of values which we can then chart
e_x = [np.exp(x) for x in x_axis]
e_x
```

```
[1.0,
 1.1051709180756477,
 1.2214027581601699,
 1.3498588075760032,
 1.4918246976412703,
 1.6487212707001282,
 1.822118800390509,
 2.0137527074704766,
 2.225540928492468,
 2.45960311115695,
 2.718281828459045,
 3.0041660239464334,
 3.320116922736548,
 3.6692966676192444,
 4.055199966844675,
 4.4816890703380645,
 4.953032424395115,
 5.473947391727201,
```

- Matplotlib allows users to generate plots by setting one list as the x-axis and another as the y-axis.
- We can do this by calling the **`plt.plot()`**, passing those two lists through as parameters, and then calling **`plt.show()`** afterwards to print the chart to the screen.

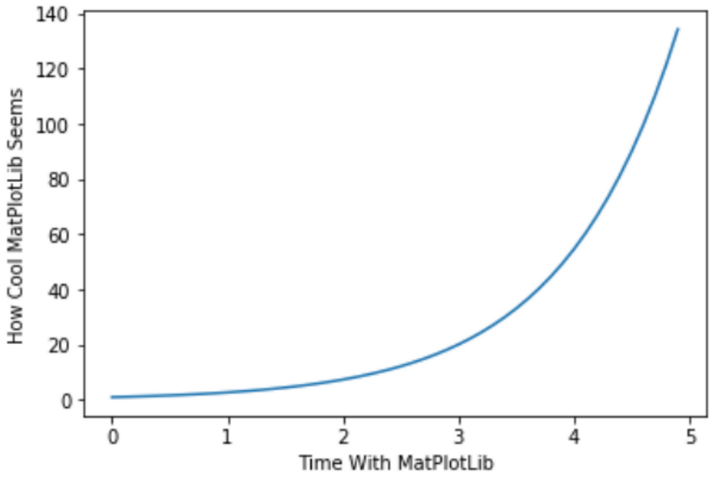
```
# Create a graph based upon the two lists we have created
plt.plot(x_axis, e_x)
plt.show()
```



- Matplotlib handles the details of painting charts to the screen, but the programmer has full control over each stage of the drawing process if they really need it.

- By using **`plt.xlabel()`** and **`plt.ylabel`**, users can add axis titles to their charts

```
# Give our graph axis labels
plt.xlabel("Time With MatPlotLib")
plt.ylabel("How Cool MatPlotLib Seems")

# Have to plot our chart once again as it doesn't stick after being shown
plt.plot(x_axis, e_x)
plt.show()
```



Open SinCos.ipynb

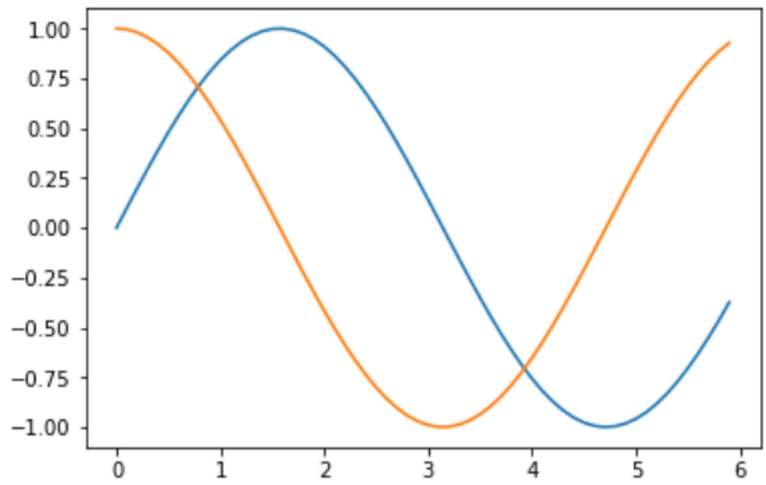
- **`np.arrange()`**, **`np.sin()`**, and **`np.cos()`** are all being used in order to create the lists for the application's charts.
- In order to chart multiple lines on the same chart, it is as simple as calling **`plt.plot()`** two times and providing PyPlot with different values.

```
# Create our x_axis list
x_axis = np.arange(0, 6, 0.1)
```

```
# Creates a list based on the sin of our x_axis values
sin = np.sin(x_axis)
```

```
# Creates a list based on the cos of our x_axis values
cos = np.cos(x_axis)
```

```
# Plot both of these lines so that they will appear on our final chart
plt.plot(x_axis, sin)
plt.plot(x_axis, cos)

plt.show()
```



Configuring Line Plots

Sunday, October 6, 2019 4:53 PM

Matplotlib's basic line plots are somewhat plain, but Matplotlib offers considerable control over the details of our plots' appearances. The easiest method to change the way things look is to use **keyword arguments** to configure the behavior of **plot**.

Open line_config.ipynb

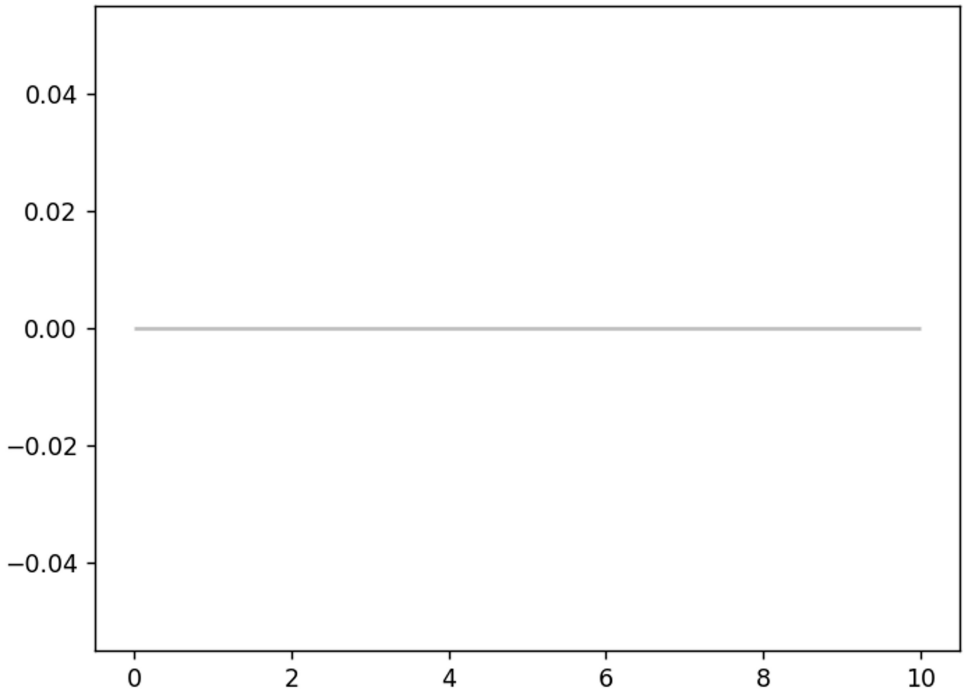
- **`%matplotlib notebook`** makes a plot interactive and it also allows it to be updated after the initial plot.

```
%matplotlib notebook
```

```
# Dependencies
import matplotlib.pyplot as plt
import numpy as np
```

- **`plt.hlines()`** is used to draw a horizontal line. This method takes in three parameters:
    - The Y value across which the line will be drawn,
    - The X value where the line will start
    - The X value where the line will end.
- The transparency of the horizontal line can also be set using the `alpha=` keyword and passing a number between 0 and 1.

```
# Draw a horizontal line with 0.25 transparency
plt.hlines(0, 0, 10, alpha=0.25)
```



- **`plt.plot()`** returns a list of the lines that were added to the plot.
- Calling the `sine_handle,` is a reference to the lines object.
- **`plt.plot()`** can take in more parameters than just the X and Y values for the line being charted.
    - For example
        - markers for a plot can be set using `marker=`
        - color of a plot can be set using `color=`
        - label for a line can be set using `label=`

```
# Assign plots to tuples that stores result of plot

# Each point on the sine chart is marked by a blue circle
sine_handle, = plt.plot(x_axis, sin, marker ='o', color='blue', label="Sine")
# Each point on the cosine chart is marked by a red triangle
cosine_handle, = plt.plot(x_axis, cos, marker='^', color='red', label="Cosine")
```

- The **`plt.legend()`** method allows the user to create a legend for their chart. The **`loc`** argument is used to set the location of the legend on the chart.

- *There is also a **handles** argument which can be used to generate the appropriate entry in the legend

```
# Adds a legend and sets its location to the lower right
plt.legend(loc="lower right")
```

- **`plt.savefig()`** saves a version of the chart to an external file. Simply pass the file path desired as a parameter to save the image

```
# Saves an image of our chart so that we can view it in a folder
plt.savefig("../Images/lineConfig.png")
plt.show()
```

Aesthetics

Sunday, October 6, 2019 4:54 PM

The best plots, like the best code, are easy to read. This does not necessarily mean beautiful. Graphics don't need to be artistic but they should be easy to understand.

- Some ways to improve readability include:
    - Adding labels to the x-axis
    - Adding labels to the y-axis
    - Adding titles to plots
    - Limiting the extent of the plot to bound the plot's data points
    - In some cases adding grids can also help but this is often discouraged in general

Open aesthetics.ipynb

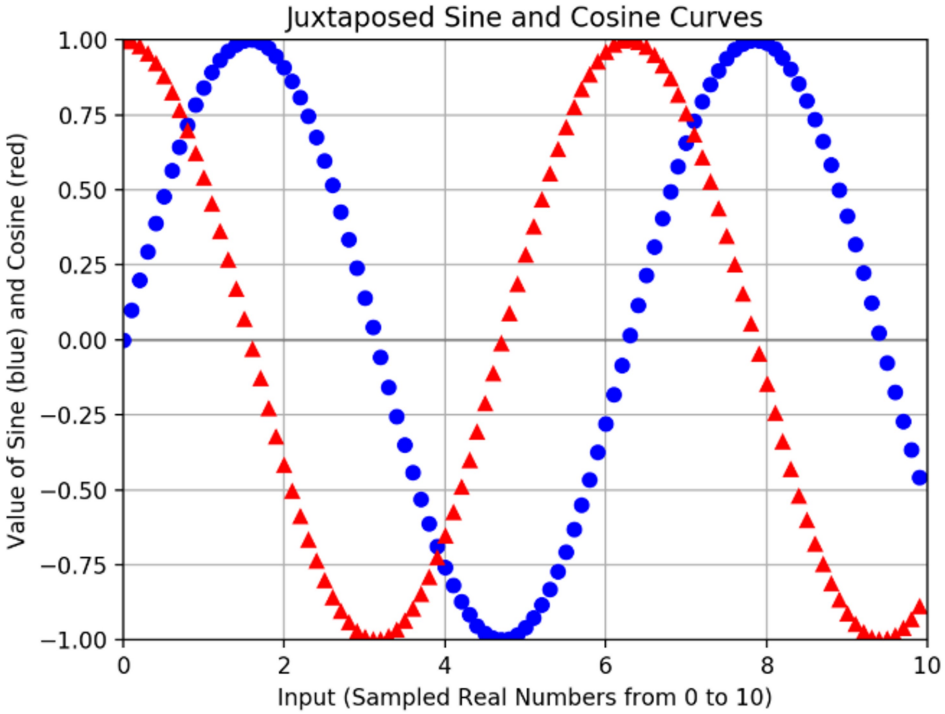
- To use **plt.xlabel()**, **plt.ylabel()**, or **plt.title()** are pass a string into them as a parameter and the labels and title will be drawn onto the chart.

```
# Add labels to the x and y axes
plt.title("Juxtaposed Sine and Cosine Curves")
plt.xlabel("Input (Sampled Real Numbers from 0 to 10)")
plt.ylabel("Value of Sine (blue) and Cosine (red)")
```

- **plt.xlim()** and **plt.ylim()** are used to set where the axes for the chart should begin/end.
- MatplotLib will naturally create charts with a lot of empty space and these methods can help to limit that.

```
# Set a grid on the plot
plt.grid()
```

- **plt.grid()** adds gridlines to the chart.


Juxtaposed Sine and Cosine Curves
Value of Sine (blue) and Cosine (red)
Input (Sampled Real Numbers from 0 to 10)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10

Different Plots

Sunday, October 6, 2019 10:57 PM

- Matplotlib provides a simple interface for producing more than just line plots.
- The most common charts students will generate are line charts, bar charts, pie charts, and scatter plots.
    - **bar charts** are useful for comparing different entities to one another.
    - **pie charts** are suitable for displaying parts of a whole
    - **scatter plots** are good for displaying where points fall with respect to two different factors.
- It’s important to choose the right plot for a given data set.
    - The wrong choice can make the graphic less readable or may even make the data misleading.
    - Some data might lend itself to different plots — some data can be reasonably displayed via bar or pie chart, for instance.

Bar Charts

Sunday, October 6, 2019 4:54 PM

Open bar_chart.ipynb

- When dealing with bar charts, it is necessary to provide the heights of each bar within an array.
- The x-axis will also be an array whose length must equal that of the list of heights.

```
# Create an array that contains the number of users each language has
users = [13000, 26000, 52000, 30000, 9000]
x_axis = np.arange(len(users))
```

- Bar charts are drawn using **plt.bar()**.
- The *align* parameter for **plt.bar()** is center to center.

```
# Tell matplotlib that we will be making a bar chart
# Users is our y axis and x_axis is, of course, our x axis
# We apply align="edge" to ensure our bars line up with our tick marks
plt.bar(x_axis, users, color='r', alpha=0.5, align="center")
```

- An additional aesthetic challenge unique to bar charts is aligning the tick locations on the x-axis and providing textual, rather than numeric, labels.
- The *tick_locations* list created within this application places a tick for each value in the *x_axis*.
- The **plt.xticks()** method sets tick locations and labels of the x-axis

```
# Tell matplotlib where we would like to place each of our x axis headers
tick_locations = [value for value in x_axis]
plt.xticks(tick_locations, ["Java", "C++", "Python", "Ruby", "Clojure"])
```

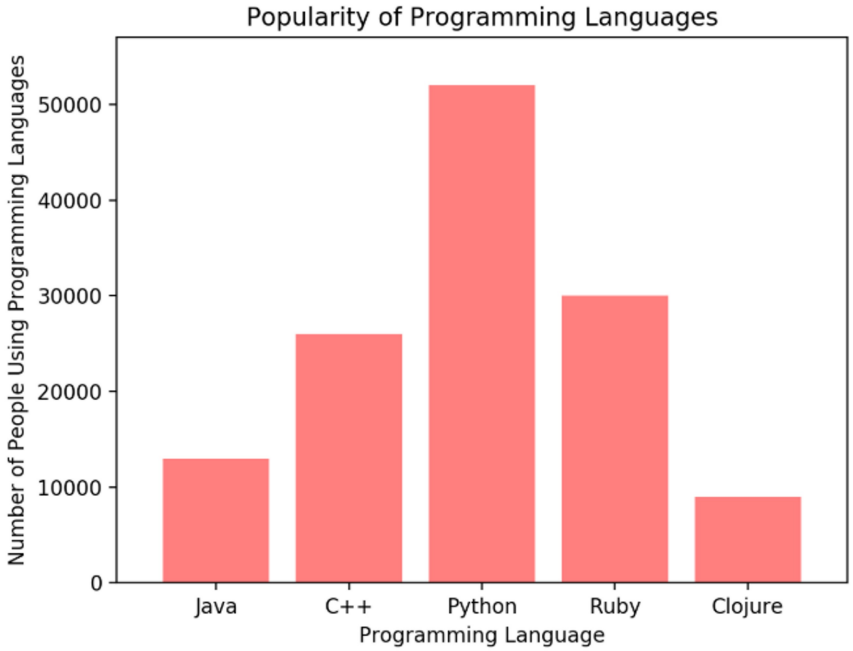
- The **plt.xlim()** and **plt.ylim()** are set so that there is some space between the bars and the edge of the chart. This makes the chart look a little better aesthetically.

```
# Sets the x limits of the current chart
plt.xlim(-0.75, len(x_axis)-0.25)
```

```
(-0.75, 4.75)
```

```
# Sets the y limits of the current chart
plt.ylim(0, max(users)+5000)
```

```
(0, 57000)
```



Pie Charts

Sunday, October 6, 2019 4:55 PM

Open pie_chart.ipynb

- Sizes of each wedge are passed into **plt.pie()** as an array. Lists containing the labels for each wedge and the colors for each wedge are also passed in.

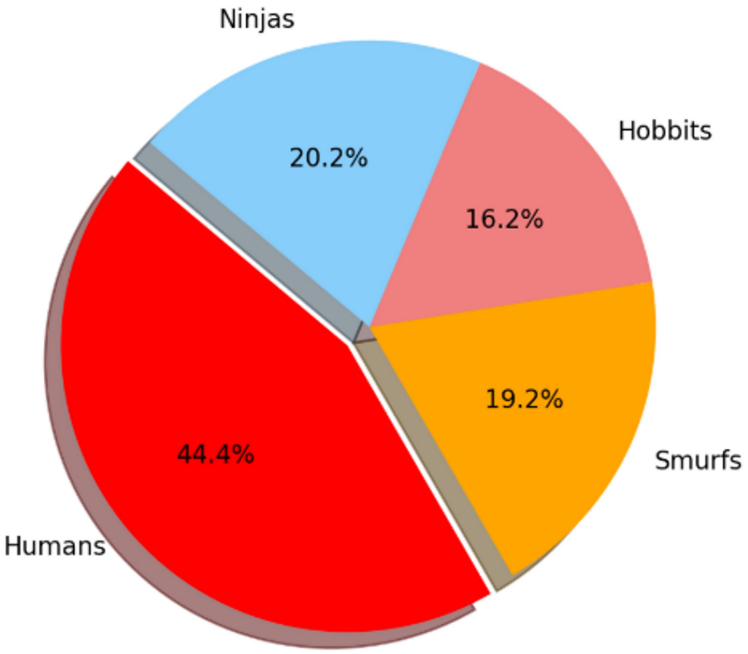
- Pie chart allows the user to choose a wedge to "explode," using the explode option. This will separate one wedge from the rest so that it is easier to examine.

- Inside of the **plt.pie()** method, a parameter of **autopc="%1.1%%"** is being passed. This will automatically convert the values passed into percentages with one decimal place.

```
# Labels for the sections of our pie chart
labels = ["Humans", "Smurfs", "Hobbits", "Ninjas"]

# The values of each section of the pie chart
sizes = [220, 95, 80, 100]

# The colors of each section of the pie chart
colors = ["red", "orange", "lightcoral", "lightskyblue"]

# Tells matplotlib to seperate the "Python" section from the others
explode = (0.1, 0, 0, 0)
```

```
# Creates the pie chart based upon the values above
# Automatically finds the percentages of each part of the pie chart
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct="%1.1f%%", shadow=True, startangle=140)
```

- Matplotlib does not constrain pie charts to be circular — by default, they will be ovals if the window the plot lives in is not a square. This is why `plt.axis("equal")` is being passed


Ninjas
20.2%
Hobbits
16.2%
19.2%
Smurfs
44.4%
Humans

Scatter Plots

Sunday, October 6, 2019 4:56 PM

Open scatter_plot.ipynb

- Generating scatter plots requires the simplest set of methods of all the charts so far. Just take in two sets of data and pass them into **plt.scatter()**

- You can change the size of each dot by passing the **s=<LIST>** parameter. In this case, the values stored within x_axis will determine the size of a dot.

```
# Tells matplotlib that we want to make a scatter plot
# The size of each point on our plot is determined by their x value
plt.scatter(x_axis, data, marker="o", facecolors="red", edgecolors="black",
            s=x_axis, alpha=0.75)
```

```
<IPython.core.display.Javascript object>
```