5. Examples of Visual Analytics in Python

Overview

1. Recap of last time
2. Example application of the week
3. Introduction to Matplotlib
4. Examples and applications

Example application: (ethical) hacking

Last time: reading files

- Last time we saw how plain text files can be read directly into Python using the `open()` function, often in conjunction with the `with` statement.

```
with open('my_file.txt') as input_file:
    text = input_file.read()
```

- Examples of plain common plain text files include TXT, CSV, HTML, JSON and more.

Last time: writing text to a file

- Any text you have in Python, as a string, can be written to a file. To do this, open a file in *write mode* by passing `'w'` to the open function after the new file name.
- If the file does not yet exist, Python will create it for us.

```
my_text = 'Some super important text'

with open('new_file.txt', 'w') as new_file:
    new_file.write(my_text)
```

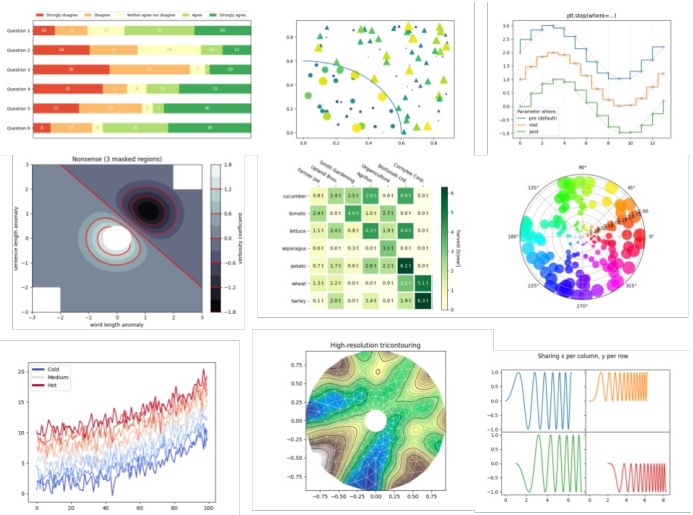
Last time: Pandas

- Pandas is a Python library that is useful for dealing with data in a table format.
- The central object is called the DataFrame, which is essentially a table.
- A new DataFrame can be created directly from Python data, or from a CSV file.

```
import pandas as pd

dataframe = pd.read_csv('my_data.csv')
```

- Once we have our DataFrame, there are many ways to manipulate and do analysis on the data.

New topic: Matplotlib



Matplotlib

- Matplotlib is another Python *library*, like Pandas. Its purpose is to provide flexible and versatile functions for data visualisation and plotting.
- With Matplotlib, you're more or less only limited by your imagination! The plots you can produce are more or less endless.
- The typical way to get access to all the Matplotlib functions is to run the following `import` statement.

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

- In Jupyter, we can also make all plots interactive (pan, zoom etc) by running

```
%matplotlib notebook
```

- Note that this isn't typical Python code - this is a Jupyter-specific command.

Creating a graph

- When creating a new graph, it's usually best to begin by creating a figure. This is done by calling:

```
plt.figure()
```

- This won't do much except create a blank canvas for us to plot onto later.
- If you don't explicitly create a figure it doesn't matter too much - Matplotlib will automatically create one for you when you start to use the plotting functions. However, if you making many figures at once, this will ensure your new plots get put on a new graph.

Line graphs

- One of the most useful types of plot Matplotlib provides is the line plot. This can be accessed via `plt.plot()`.
- This function expects at least two arguments.

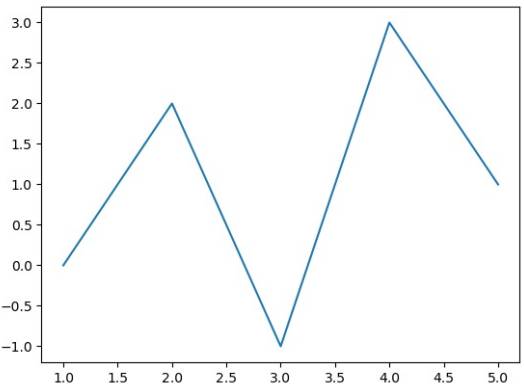
- The first argument should be a list (or other list-like thing such as a pandas DataFrame column) representing the x-coordinates of a set of points.
- The second argument should also be a list (or other list-like thing) representing the y-coordinates of a set of points.

- The function will then plot a line by "joining the dots".
- The only rule is that lists arguments should contain only numbers, and both be the same length.

Line graphs

```
import matplotlib.pyplot as plt

# create two lists to hold the x-y coords
x = [1, 2, 3, 4, 5]
y = [0, 2, -1, 3, 1]

# create a new figure
plt.figure()

# plot the data
plt.plot(x, y)
```

Line graphs

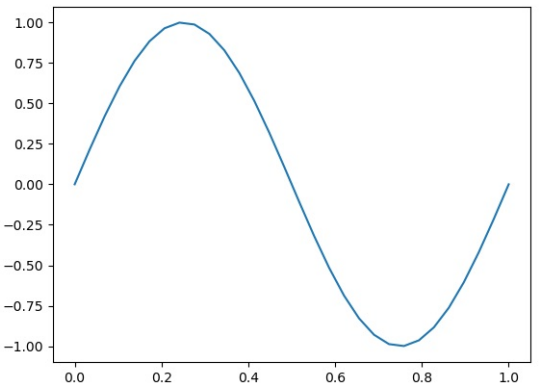


Line graphs

- This creates a "jagged" effect, but a line can be made to appear smooth by using many points.

```
x = [0, 0.034, 0.069, 0.103, 0.138, 0.172, 0.207, 0.241, 0.276, 0.31,
0.345, 0.379, 0.414, 0.448, 0.483, 0.517, 0.552, 0.586, 0.621, 0.655,
0.69 , 0.724, 0.759, 0.793, 0.828, 0.862, 0.897, 0.931, 0.966, 1]

y1 = [ 0,  0.215,  0.42 ,  0.605,  0.762,  0.884, 0.964,  0.999, 0.987,
 0.929,  0.828,  0.688,  0.516,  0.319,  0.108, -0.108, -0.319, -0.516,
-0.688, -0.828, -0.929, -0.987, -0.999, -0.964, -0.884, -0.762, -0.605,
-0.42 , -0.215, 0]

plt.figure()
plt.plot(x, y1)
```

Line graphs

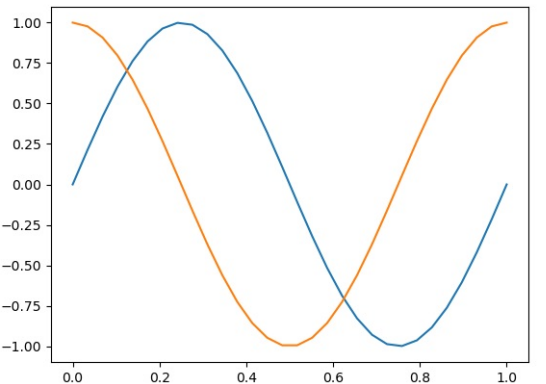


Multiple lines

- We can add more than one line to a single plot. Simply call `plt.plot()` again, without making a new `plt.figure()`.
- The new line will be added in a different colour.

```
y2 = [ 1.,  0.977,  0.908,  0.796,  0.647,  0.468,  0.268,  0.054,
-0.162, -0.37, -0.561, -0.726, -0.857, -0.948, -0.994, -0.994, -0.948,
-0.857, -0.726, -0.561, -0.37 , -0.162,  0.054,  0.268, 0.468,  0.647,
 0.796,  0.908,  0.977,  1]

plt.figure()
plt.plot(x, y1)
plt.plot(x, y2)
```

Multiple lines

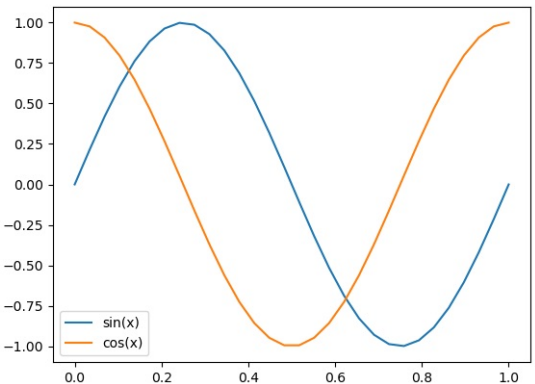


Adding a key

- When there are two or more lines it's often useful to add labels to make the plot easier to interpret.
- This can be achieved by including a `label` argument to `plt.plot()`, plus calling `plt.legend()`.

```
# create a new figure
plt.figure()

# plot lines with labels
plt.plot(x, y1, label='sin(x)')
plt.plot(x, y2, label='cos(x)')

# add a key
plt.legend()
```

Adding a key

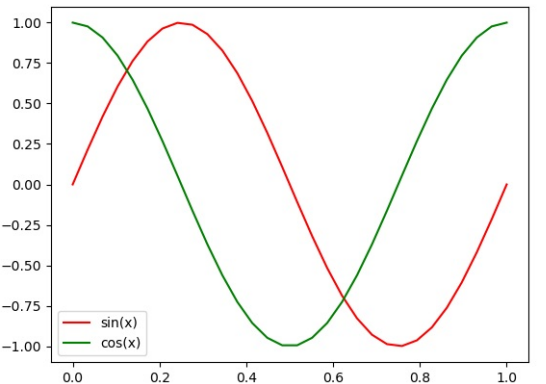


Setting the colour

- By default Matplotlib will plot the first line in blue, the second in orange and so on. However, you can override this default behaviour and specify the colour you would like directly.
- This is done by passing another argument, `color`. (Yes it's sadly the American spelling of colour!)

```
# create a new figure
plt.figure()

# plot lines with labels and colours
plt.plot(x, y1, label='sin(x)', color='red')
plt.plot(x, y2, label='cos(x)', color='blue')

# add a key
plt.legend()
```

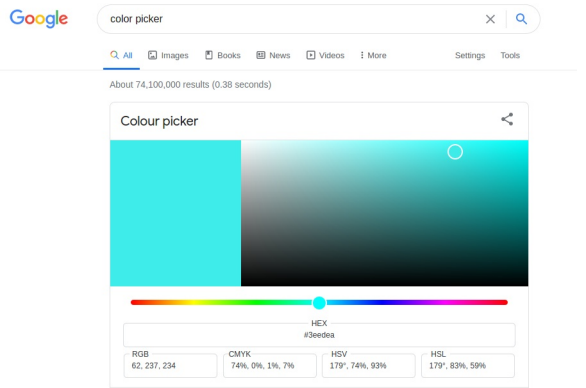
Setting the colour



Setting the colour

Setting the colour

- You can also use hexadecimal colour-codes for even finer control over colours, for example `color='#3eedea'`.

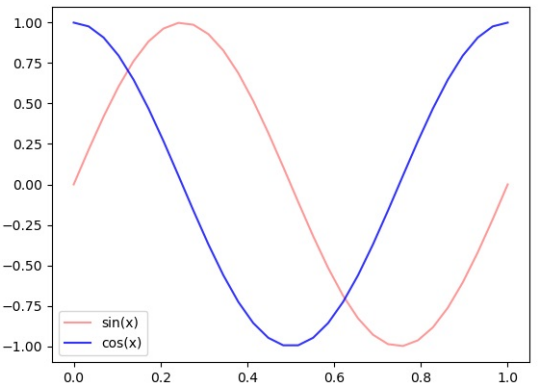


Setting transparency

- When you have many lines on the same graph which may be going over the top of each other, it can be useful to set them somewhat transparent to avoid obscuring any data.
- This can be achieved using the `alpha` argument. Alpha can be any number between 0 and 1, where 0 is totally transparent and 1 is fully opaque.

```
# create a new figure
plt.figure()

# plot lines with labels and colours and transparency
plt.plot(x, y1, label='sin(x)', color='red', alpha=0.4)
plt.plot(x, y2, label='cos(x)', color='blue', alpha=0.8)

# add a key
plt.legend()
```

Setting transparency

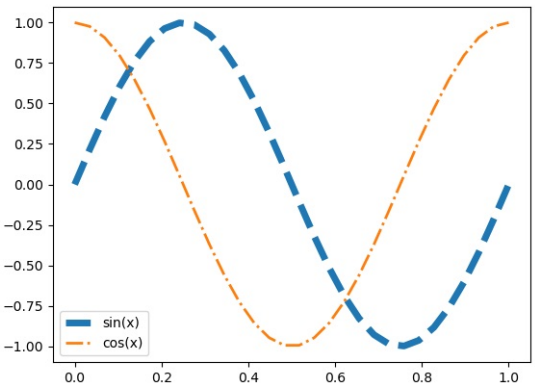


Setting other line properties

- Some other options you might want to set for your line are the line width, and the line style.
- Line width can be changed with the parameter `lw` and can be any number (1 is default).
- Line style can be a range of dashes, dots and mixtures. You can set this by passing one of `'-'`, `'--'`, `'-.'`, or `':'` as a string.

```
# create a new figure
plt.figure()

# plot lines with specific widths and styles
plt.plot(x, y1, label='sin(x)', lw=5, ls='--')
plt.plot(x, y2, label='cos(x)', lw=2, ls='-.')

# add a key
plt.legend()
```

Setting other line properties



Scatter plots

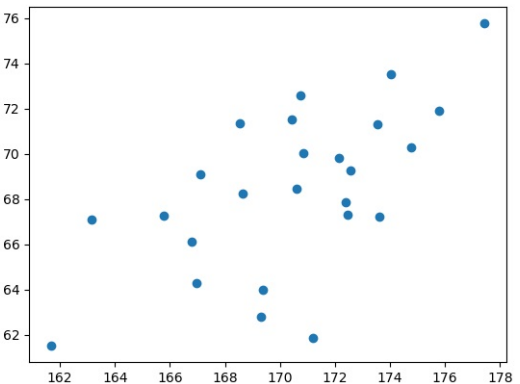
- Scatter plots are for when you want to display a cloud of points, but do not want to connect them.
- They are useful for showing some relationship between two variables, when there is no particular order to the points.
- The function `plt.scatter()` works in much the same way as `plt.plot()`. Again, it needs two arguments first representing the x and y coordinates of some data and can then take a number of optional arguments for extra control.

$$x = [x_1,\ x_2,\ \ldots,\ x_N]$$
$$y = [y_1,\ y_2,\ \ldots,\ y_N]$$
$$\text{points} : (x_1, y_1),\ (x_2, y_2),\ \ldots\ (x_N, y_N)$$

Scatter plots

```
# a list containing the height of 25 people in cm
heights1 = [172.39, 174.76, 170.6 , 167.1 , 173.55, 163.15, 171.21,
161.67, 165.76, 169.31, 169.37, 168.64, 172.47, 170.76, 175.78, 172.55,
170.42, 166.8, 174.03, 166.95, 177.44, 172.14, 170.85, 173.63, 168.55]

# a list containing the weight of the same 25 people in kg
weights1 = [67.86, 70.29, 68.47, 69.1 , 71.32, 67.11, 61.86, 61.52,
67.28, 62.79, 63.99, 68.25, 67.32, 72.6, 71.9 , 69.26, 71.54, 66.12,
73.52, 64.31, 75.8, 69.83, 70.04, 67.23, 71.34]

# create a new figure
plt.figure()

# scatter the points in a cloud
plt.scatter(heights1, weights1)
```

Scatter plots

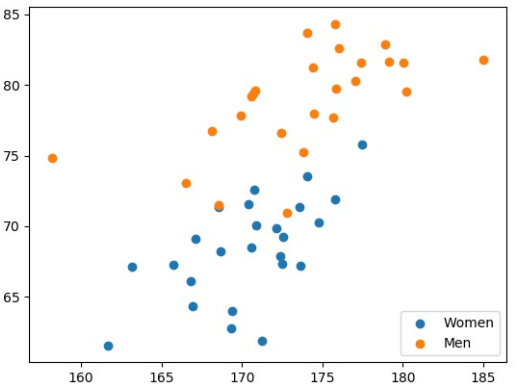


Scatter plots: multiple clouds

- As with line plots, multiple clouds can be added to the same plot.

```
heights2 = [168.14, 172.8 , 170.83, 172.43, 178.89, 170.6, 158.22,
169.91, 174.05, 175.67, 174.48, 177.07, 180.19, 184.98, 175.84, 170.67,
180.05, 179.15, 173.83, 175.77, 177.39, 168.53, 176.04, 166.52, 174.42]

weights2 = [76.77, 70.92, 79.63, 76.61, 82.84, 79.21, 74.82, 77.84,
83.7, 77.69, 77.97, 80.31, 79.5 , 81.75, 79.72, 79.37, 81.57, 81.67,
75.25, 84.29, 81.56, 71.49, 82.59, 73.05, 81.26]

plt.scatter(heights1, weights1, label='Women')
plt.scatter(heights2, weights2, label='Men')

plt.legend()
```

Scatter plots: multiple clouds



Scatter plots: other parameters

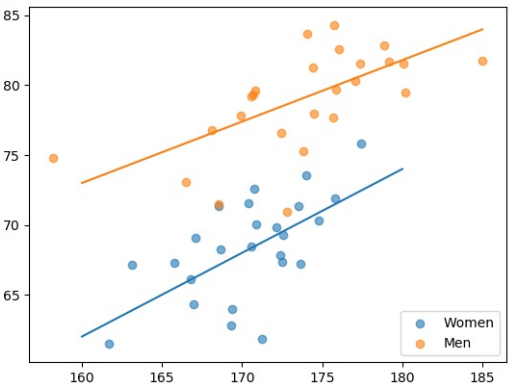
Parameter name	Possible values	Effect
`color`	Any named colour or hex code	Changes the colour of the points
`alpha`	Any number from 0 to 1	Sets the transparency
`label`	Any string	Labels the points when used with `plt.legend()`
`s`	Any number greater than 0	Width of the points in pixels
`marker`	`'x'`, `'o'`, `^` and more	Changes the point from the default circle

Mixed plots

- There's no problem putting line plots and scatter plots onto the same figure.

```
# create a new figure
plt.figure()

# plot two lines
plt.plot([160, 180], [62, 74])
plt.plot([160, 185], [73, 84])

# scatter two clouds with labels and alpha
plt.scatter(height, weight, label='Women', alpha=0.6)
plt.scatter(height2, weight2, label='Men', alpha=0.6)

# add a key
plt.legend()
```

Mixed plots



Adding axis labels and a title

- It's usually a good idea to add labels to the axes so that the plot is more easily interpretable.

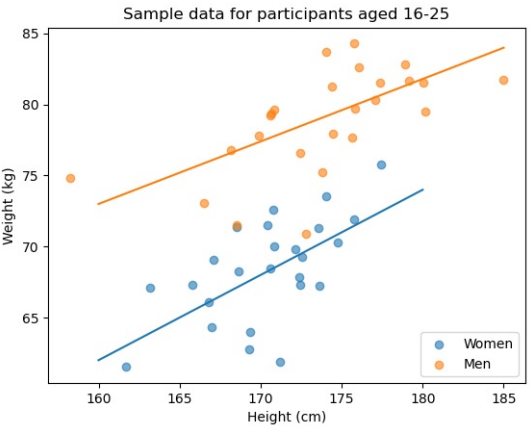
- Sometimes a title is also a good idea, although other times it can clutter up your plot. In general, only include a title if it adds extra information not clear from the plot. E.g. "A graph of height vs weight" is not a great title, if height and weight are already shown on the axes.

- You can add both of these to your Matplotlib plot with simple commands.

Adding axis labels and a title

```
# create new figure
plt.figure()

# scatter two clouds with labels and alpha
plt.scatter(height, weight, label='Women', alpha=0.6)
plt.scatter(height2, weight2, label='Men', alpha=0.6)

# plot two lines
plt.plot([160, 180], [62, 74])
plt.plot([160, 185], [73, 84])

# add a title and axis labels
plt.title('Sample data for participants aged 16-25')
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')

# add a key
plt.legend()
```

Adding axis labels and a title



Plotting with Pandas

```
import pandas as pd

data = pd.read_csv('stocks.csv', index_col='Date', parse_dates=True)
print(data)
```

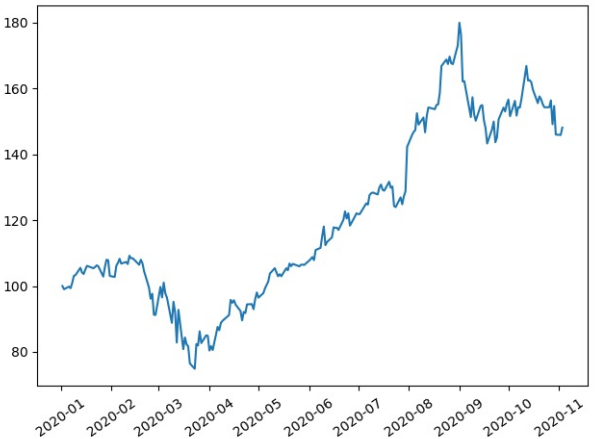
Date	AAPL	AMZN	MSFT	TSLA
2020-01-02	100	100	100	100
2020-01-03	99.0278	98.7861	98.7548	102.963
2020-01-06	99.8169	100.257	99.0101	104.946
2020-01-07	99.3474	100.466	98.1073	109.018
...	...	...	...	...

Plotting with Pandas

- So far we've used lists as the input data for the plotting functions. However, we can also use Pandas columns

```
# create a new figure
plt.figure()

# x: DataFrame index (dates), y: DataFrame column
plt.plot(data.index, data['AAPL'])

# rotate the x-ticks by 35 deg anticlockwise
plt.xticks(rotation=35)

# this makes sure no writing gets cut-off at the bottom
plt.tight_layout()
```

Plotting with Pandas



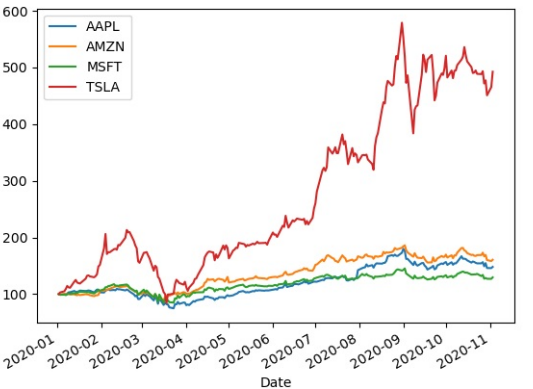
Plotting with Pandas

- Pandas actually has a built-in method `.plot()` which can be applied to a DataFrame. This will automatically:
    1. Plot every column
    2. Use the index as as the x-data.
    3. Add a label for every column

- All of this can be done by hand, but this can make things much faster sometimes.

```
# create a new figure
plt.figure()

# plot all data
data.plot()
```

Plotting with Pandas

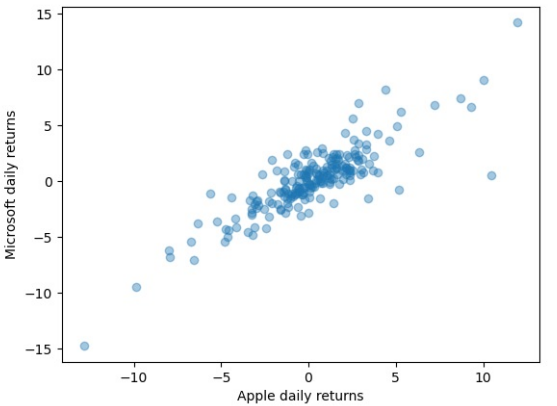


Plotting with Pandas

- Equally, we could scatter two Pandas columns against each other to analyse their correlation.

```
# get a DataFrame containing returns
returns = data.pct_change() * 100

# create a new figure
plt.figure()

# scatter AAPL vs MSFT
plt.scatter(returns['AAPL'], returns['MSFT'], alpha=0.4)

# add axis labels
plt.xlabel('Apple daily returns')
plt.ylabel('Microsoft daily returns')
```

Plotting with Pandas

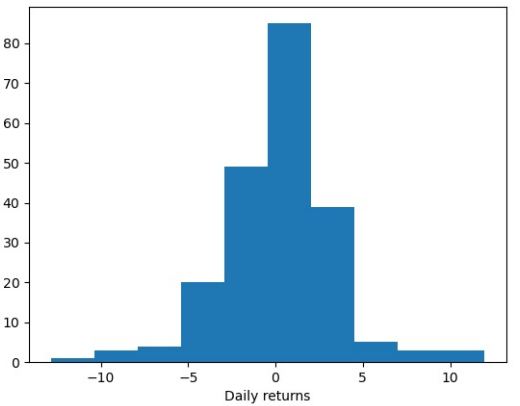


Histograms

- Another useful plot type is histograms. Histograms can be made with the function `plt.hist()`.
- This function requires one argument, which should be any list-like object of numbers.

```
# create new figure
plt.figure()

# plot a histogram
plt.hist(returns['AAPL'])

# add an axis label
plt.xlabel('Daily returns')
```

Histograms

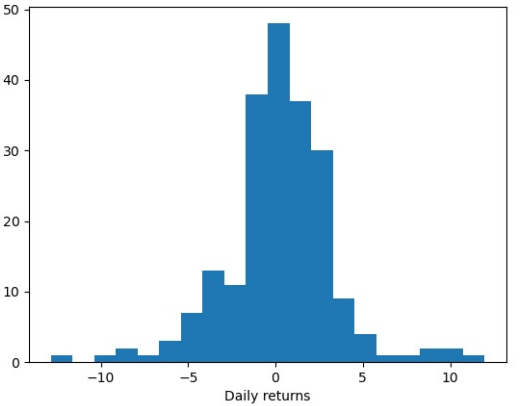


Histograms

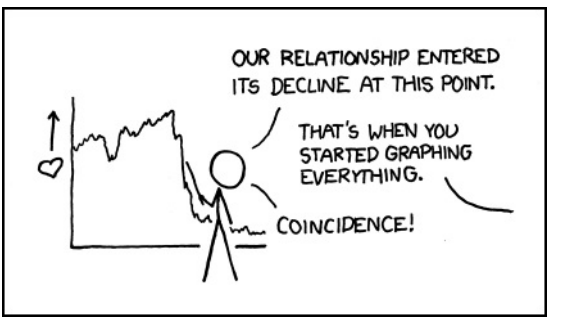
- One useful option for histograms is the number of divisions or *bins*. The default, 10, is often too few.
- The appropriate number of bins will depend on the total number of data points being histogrammed.

```
# create new figure
plt.figure()

# plot a histogram
plt.hist(returns['AAPL'], bins=20)

# add an axis label
plt.xlabel('Daily returns')
```

Histograms



Thanks!


OUR RELATIONSHIP ENTERED ITS DECLINE AT THIS POINT.
THAT'S WHEN YOU STARTED GRAPHING EVERYTHING.
COINCIDENCE!