Analytics

Chapter 3

Analytics

Analytics or data analytics is the computer science discipline of analysing and interpreting data.

Most computer science applications amass a lot of data over time.

For example a medical system could store millions of instances of data in seconds.

The goal of Analytics is to process and transform the data, present the data in a meaningful way and finally interpret the data.

Data Analytics

Terms that are similar:

- Data Analytics
- Data Analysis
- Data Mining
- Data Science

Data

Data is the term used to typically represent raw data, from sources such as a CSV file, a List or data from a database.

It is typically in a form that is of no use to a computer scientist.

Pre-processing is usually the next step.

Pre processing

One of the first things that computer scientists exam are outliers - for example someone could have accidently typed an extra zero when entering an age resulting in 200 and not 20.

A simple method is to remove that value.

Another pre processing method is to sort the data.

This is useful to get the max and min values or getting ready to count frequency.

Information

Information usually refers to gathering or presenting the data for the user.

This typically requires presenting the data visually such as a graph or a figure.

Analysis

Even with information the fine details are not always obvious.

Information is a great aid for presenting your findings, but algorithms can be more accurate when interpreting the data.

Algorithms such as frequency, mean, median, minimum, maximum and mode of data can really add clarity and help interpret the data.

Data - Python Lists

Lists in Python are fundamental in analytics. Even if our data is stored in a 2D list (like a database), most algorithms such as frequency, mean, median and mode can be calculated on a 1D list.

Text Files

Text files are one of the most versatile files for computer scientists.

Example of creating a text file and entering data.

```
file = open(“myFirstTextFile.txt”,”w”)

file.write(“Test text”)

file.close()
```

```
file = open(“myFirstTextFile.txt”,”w”)
```

The first line allows access to the file. If the file does not exist, Python automatically creates it for you.

`File` (on the first line) is just a variable name. The `open` method takes two arguments:

The first argument is a string that is the file name. You also need the file extension. For a text file, this is .txt.

The second argument is what you are doing with the file. There are three options:

“w” - write. This will overwrite any previous contents.

“a” - append. This adds the text to the end of the current contents.

“r” - read. This read the contents of the file as a string.

```
file.write(“Test text”)
```

The second line of code, we pass a string in as an argument to be written to the file.

```
file.close()
```

The third line of code closes the connection to the file.

Python must request to open a file, it must also say when it is complete and release the file. If you don’t then you may get an error when trying to use this file again.

Reading the file

To read in the same file we just created, we could do the following:

```
file = open(“myFirstTextFile.txt”, “r”)

dataIn = file.read()

file.close()

print(dataIn)
```

CSV files

While text files have their uses, they are not the most suitable for storing data, especially large amounts of data.

A CSV file is essentially like a .txt file. It uses the same Unicode/ASCII structure.

CSV stands for “comma separated values”

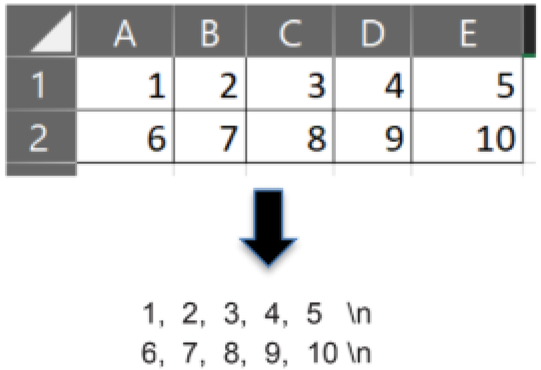
Computer Scientists like this format for the following reasons:

- Quick to read and write to in Python
- Excel has a row limit of 1,048,576 rows
- The file can be used even if you do not have excel on your computer.

CSV Structure

The structure of a CSV file is relatively simple but the format can cause some issues:

- Commas separate elements on a row. These are called delimiters.
- A new line special character “\n” represents the end of a row and the start of a new row.

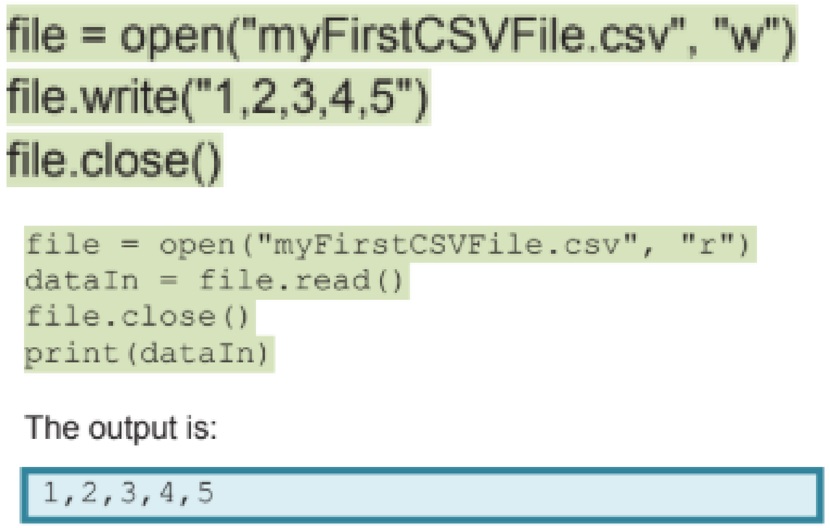


	A	B	C	D	E
1	1	2	3	4	5
2	6	7	8	9	10

```
1,  2,  3,  4,  5   \n
6,  7,  8,  9,  10 \n
```

Reading and Writing to CSV file

Reading and Writing to a csv file can be done the same way as for a text file. You just need to change the file extension.



```
file = open("myFirstCSVFile.csv", "w")
file.write("1,2,3,4,5")
file.close()
```

```
file = open("myFirstCSVFile.csv", "r")
dataIn = file.read()
file.close()
print(dataIn)
```

The output is:

```
1,2,3,4,5
```

The code in the previous slide prints a string for us. This is not really useful if we want to find a total or an average etc.

We know that every element is separated by a comma in a csv file.

We can use this to our advantage and write a loop to go through the string and take out elements with a comma in between them.

This sounds complicated but Python has a method for doing this - it can separate a string into a list based on any specified delimiter (in our case it’s a comma)

```
myList = dataIn.split(",")
print(myList)
```

The output is:

```
[‘1’,’2’,’3’,’4’,’5’]
```

The .split() method takes the delimiter as an argument.

Notice that all the elements in the list are strings.

We can convert each element to an int so that we can do calculations on them. Python has an easy way to do this.

```
myList = dataIn.split(",")

myList = [int(element) for element in myList]

print(myList)
```

The output is (now all integers):

```
[1,2,3,4,5]
```

Pre-Processing and Transforming Data

Pre-Processing allows to apply techniques to ensure the data is of good quality.

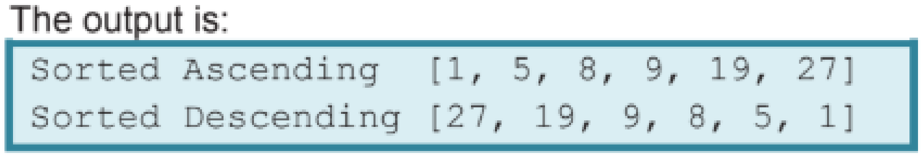
Typical errors in data that require pre-processing are:

- Missing data
- Data out of range
- Non validated data
- Data structure incorrect

Sorting

Sorting is useful for many reasons. If you sort, you will know the largest and the smallest are the first and the last value of the list.

```
myList = [1,19,27,8,5,9]

myList.sort()

print(“Sorted Ascending:” myList)

myList.sort(reverse = True)

print(“Sorted Descending:” myList)
```



The output is:

```
Sorted Ascending  [1, 5, 8, 9, 19, 27]
Sorted Descending [27, 19, 9, 8, 5, 1]
```

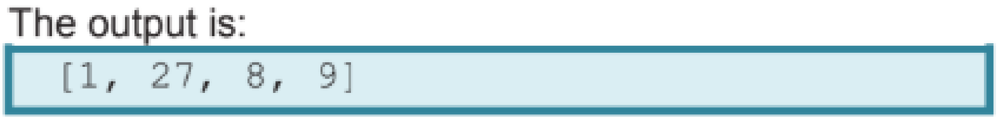
Removing Missing Data

Sometimes data in a list could be identified as an error and may need to be removed.

For example if a list should only contain positive numbers and a negative number is present.

A pre-processing step could be to remove all negative numbers

```
myList = [1,-19,27,8,-5,9]
for item in myList:
        if item<0:
                myList.remove(
print(myList)
```



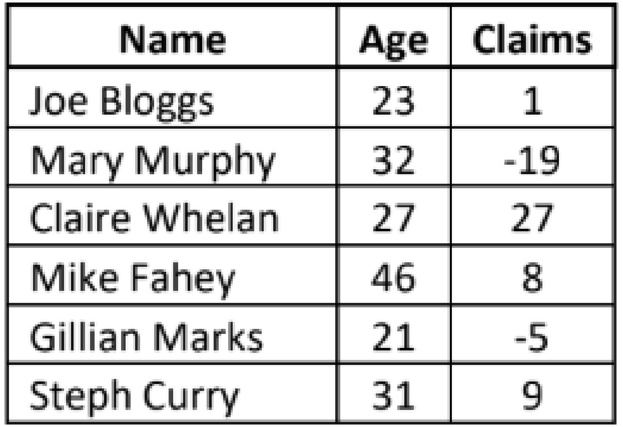
The output is:

```
[1, 27, 8, 9]
```

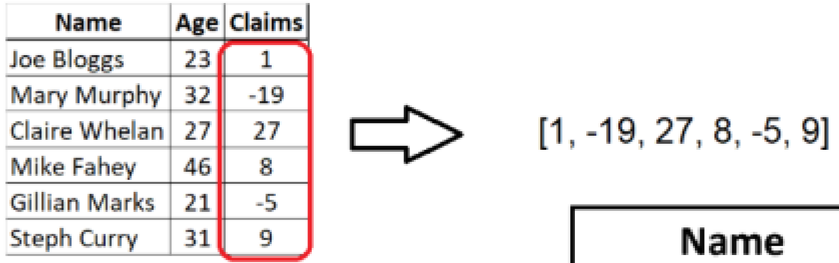
Imputing Missing Data

Imputing data is an option instead of removing missing data.

Sometimes removing data is not an option, especially in a 2D list as there is other data associated with it.

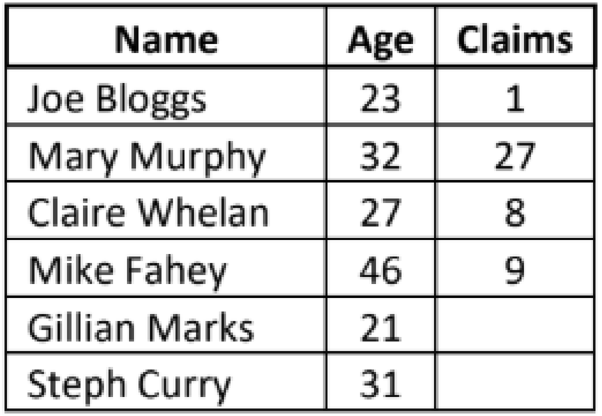


Name	Age	Claims
Joe Bloggs	23	1
Mary Murphy	32	-19
Claire Whelan	27	27
Mike Fahey	46	8
Gillian Marks	21	-5
Steph Curry	31	9



Name	Age	Claims
Joe Bloggs	23	1
Mary Murphy	32	-19
Claire Whelan	27	27
Mike Fahey	46	8
Gillian Marks	21	-5
Steph Curry	31	9

[1, -19, 27, 8, -5, 9]



Name	Age	Claims
Joe Bloggs	23	1
Mary Murphy	32	27
Claire Whelan	27	8
Mike Fahey	46	9
Gillian Marks	21	
Steph Curry	31	

Imputing Missing Data

We can go through a list and replace all values with a number.

Imputing is an algorithm that finds the average of all non missing values and then replaces the missing values with the average of the group.

There are two parts:

First find the average of all non missing values

Second replace all error values with the calculated average

Part A, find the average of all non-missing values:

```
totalNonMissing = 0
countNonMissing = 0
myList = [1, -19, 27, 8, -5, 9]
for item in myList:
    if item > 0:
        totalNonMissing += item
        countNonMissing += 1
averageNonMissing = totalNonMissing /
countNonMissing

print(averageNonMissing)
```

The average value of non-error numbers is:

```
11.25
```

Part B, replace all error values with the recently calculated `averageNonMissing`:

```
for counter in range(len(myList)):
    if myList[counter] < 0:
        myList[counter] = averageNonMissing
print(myList)
```

The imputed list:

```
[1, 11.25, 28, 8, 11.25, 9]
```

Cleaning/Pre-processing Data

In the programs we do, the most common cleaning involves one of the following:

- Remove unwanted characters such as a comma in the middle of a number
- Select only part of the data in a column

Examples of each of these are here:

Remove unwanted characters

Select part of the data in a column

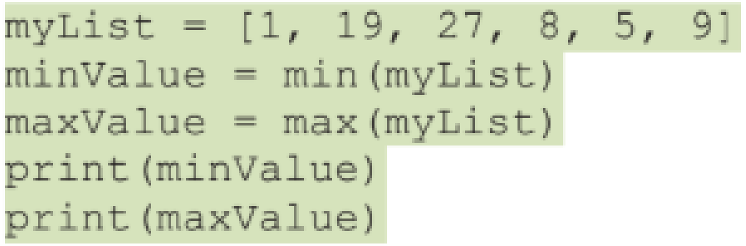
Data Analytics Algorithms

Python is one of the most popular languages for Analytics in 3rd level and industry. One of the main reasons for this is the speed at which you can conduct Analysis such as getting the maximum and minimum values of a list.

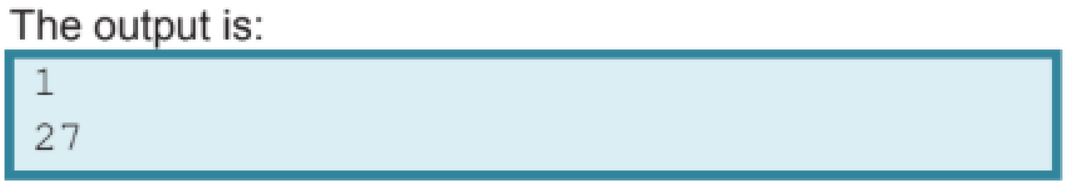
We can also use the statistics library to find the mean, median and mode or we can code these without the use of a library.

Max and Min Values

Method 1: Using Inbuilt functions



```
myList = [1, 19, 27, 8, 5, 9]
minValue = min(myList)
maxValue = max(myList)
print(minValue)
print(maxValue)
```

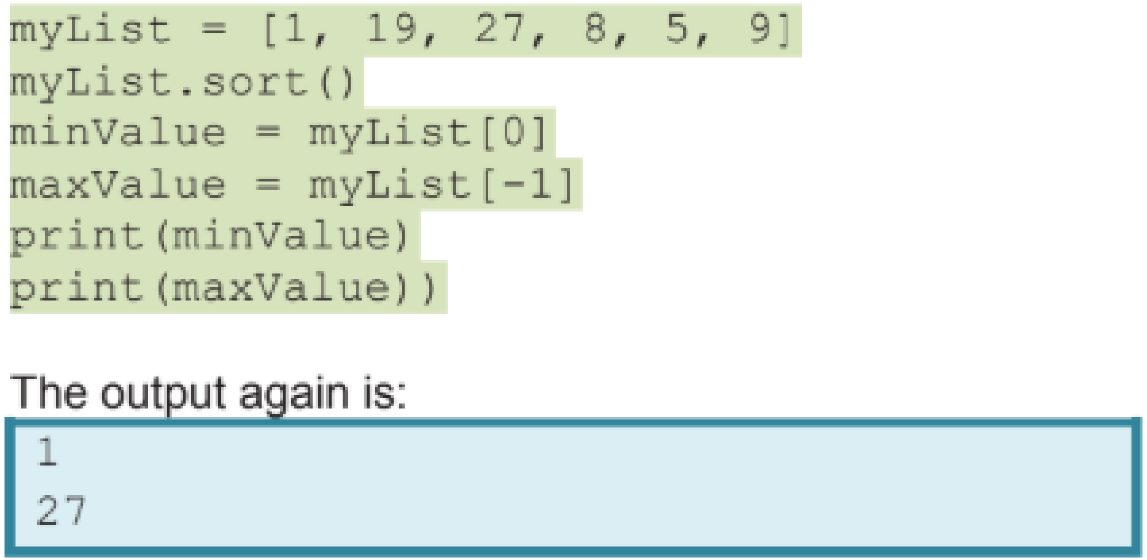


The output is:

```
1
27
```

Max and Min Values

Method 2: Using Sorting and extracting the First and Last Value



```
myList = [1, 19, 27, 8, 5, 9]
myList.sort()
minValue = myList[0]
maxValue = myList[-1]
print(minValue)
print(maxValue))
```

The output again is:

```
1
27
```

Max and Min Values

Method 3: Manually Coding

```
myList = [1, 19, 27, 8, 5, 9]
minValue = myList[0]
maxValue = myList[0]

for item in myList:
    if item < minValue:
        minValue = item
    if item > maxValue:
        maxValue = item

print(minValue)
print(maxValue))
```

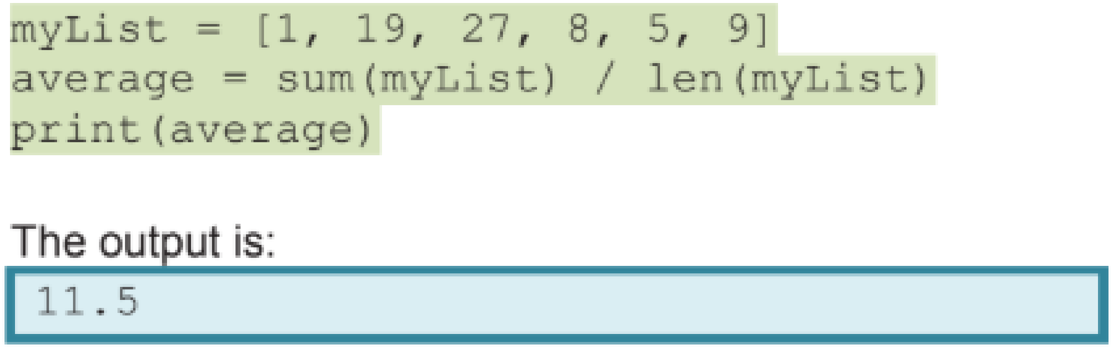
The output again:

```
1
27
```

The algorithm sets the first value in the list to the maximum number. Then you iterate through the list, and if any number is bigger than the current maximum number you replace the maximum number with the new number.

Mean

Method 1: Using sum() and len()



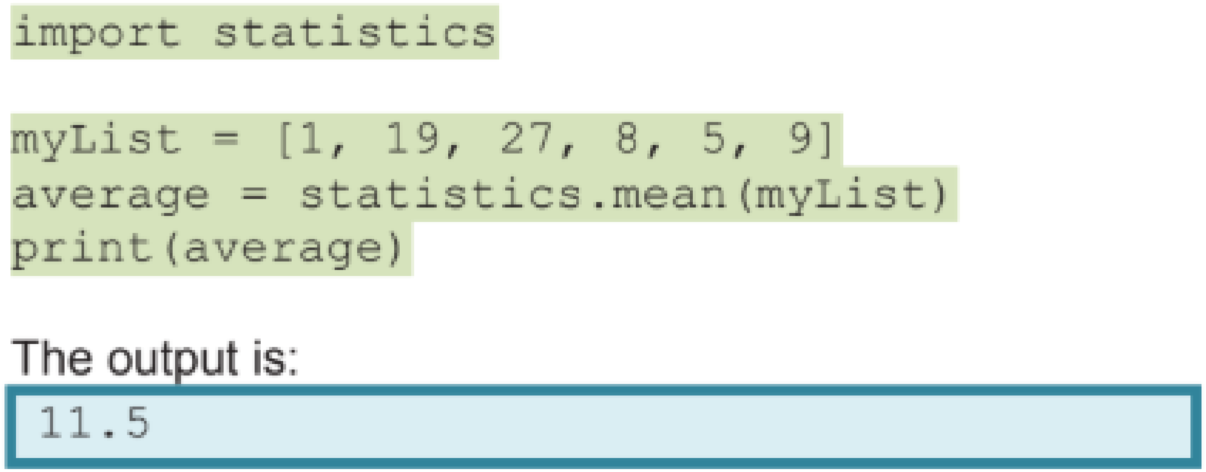
```
myList = [1, 19, 27, 8, 5, 9]
average = sum(myList) / len(myList)
print(average)
```

The output is:

```
11.5
```

Mean

Method 2: Using statistics library



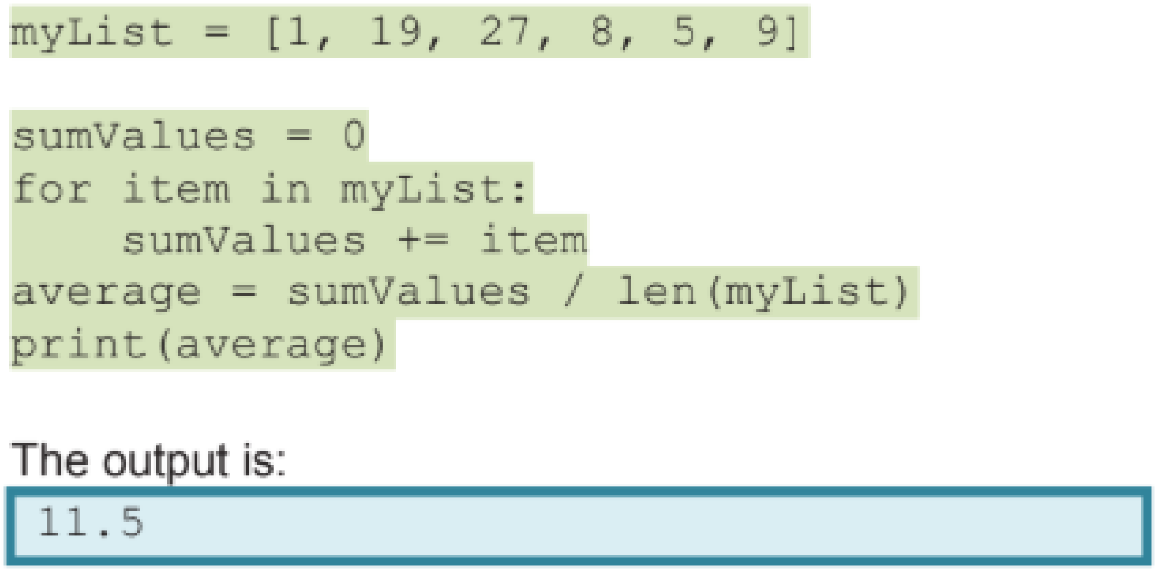
```python
import statistics

myList = [1, 19, 27, 8, 5, 9]
average = statistics.mean(myList)
print(average)
```

The output is:

```
11.5
```

Mean

Method 3: Manually Coding



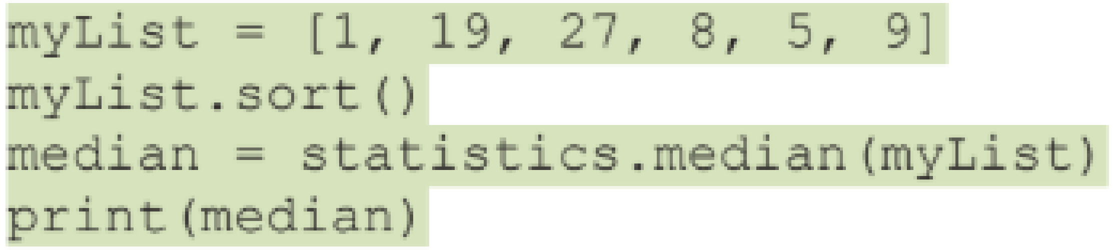
```
myList = [1, 19, 27, 8, 5, 9]

sumValues = 0
for item in myList:
    sumValues += item
average = sumValues / len(myList)
print(average)
```

The output is:

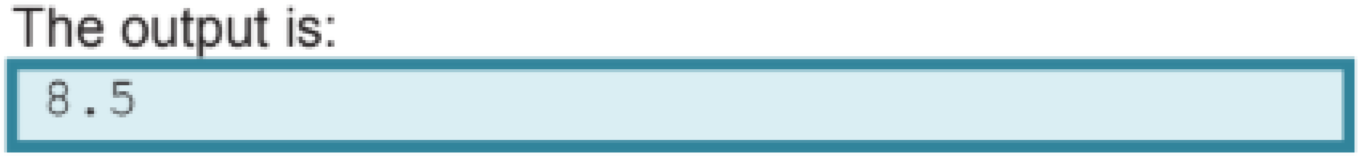
```
11.5
```

Median

Method 1: Using the statistics library



```python
myList = [1, 19, 27, 8, 5, 9]
myList.sort()
median = statistics.median(myList)
print(median)
```

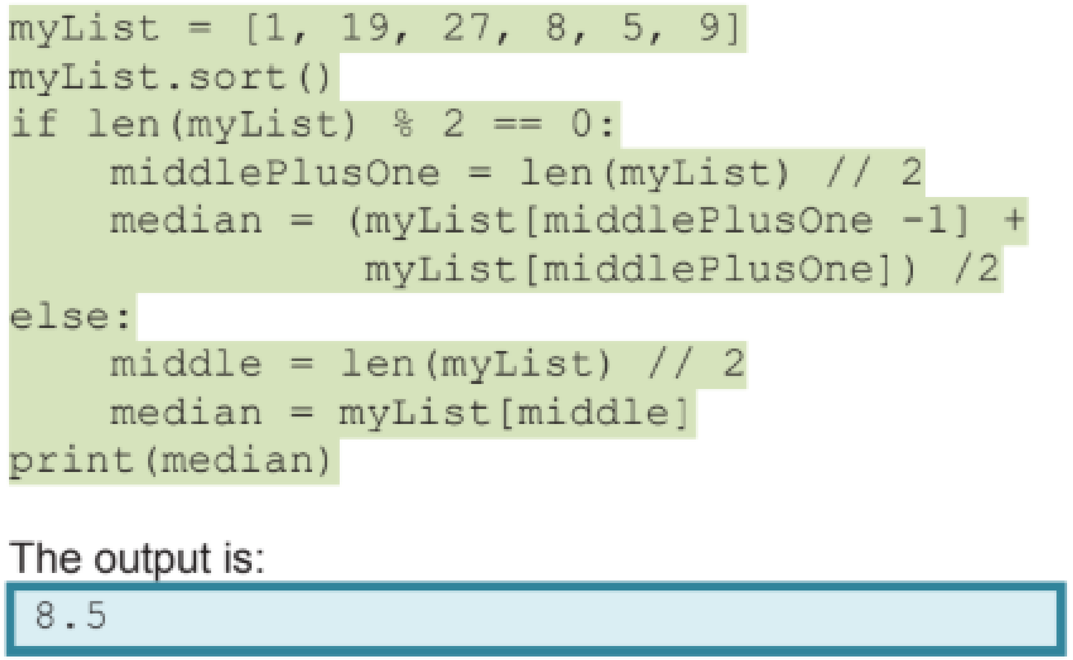


The output is:

```
8.5
```

Median

Method 2: Manually Coding



```
myList = [1, 19, 27, 8, 5, 9]
myList.sort()
if len(myList) % 2 == 0:
    middlePlusOne = len(myList) // 2
    median = (myList[middlePlusOne -1] +
              myList[middlePlusOne]) /2
else:
    middle = len(myList) // 2
    median = myList[middle]
print(median)
```

The output is:

```
8.5
```

Note: The median is the middle number of a sorted list.

If there is an even number of elements in the list then the median is the middle two numbers added together and divided by 2.

Frequency

Frequency is the number of times elements appear in a list, this can be for strings or numbers.

Frequency

```
myList = ["red", "blue", "blue", "red",
"green", "red", "red"]

colourNames = []
colourCounts = []

for item in myList:
    if item not in colourNames:
        colourNames.append(item)
for colour in colourNames:
    total = myList.count(colour)
    colourCounts.append(total)

print(colourCounts)
print(colourNames)
```

Script output:

```
[4, 2, 1]
['red', 'blue', 'green']
```

Creates two empty lists

Loops through myList. If the item is not already in the colourNames list then add it to the list.

Loops through the colourNames list and counts the amount of times each colour is in myList

Mode

Mode is the value that occurs most often in the dataset.

Method 1: Using statistics library

```
import statistics

myList = ["red", "blue", "blue", "red",
"green", "red", "red"]

mode = statistics.mode(myList)
print(mode)
```

Script output:

```
red
```

Mode

Method 2: Manually Coding

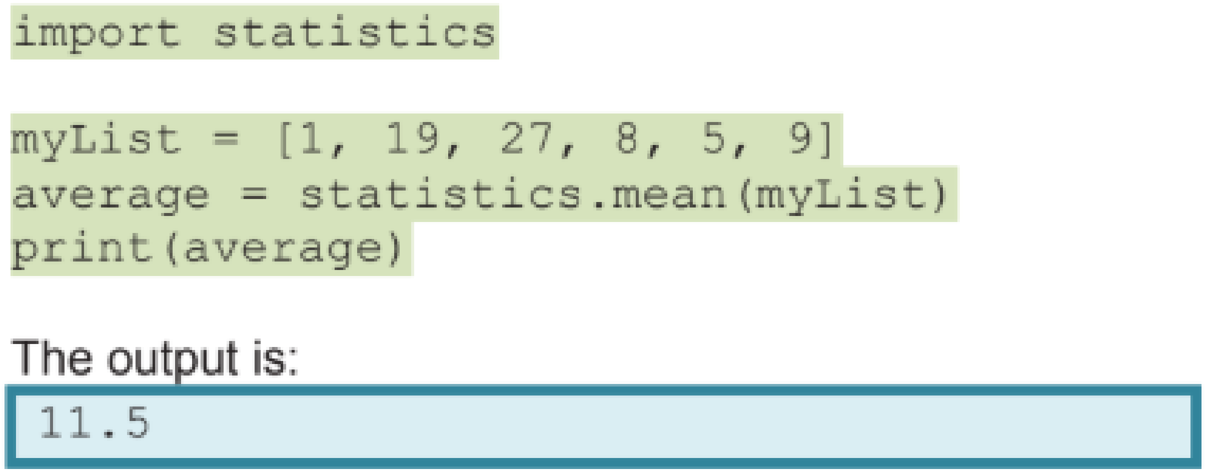
```
colourCounts = [4, 2, 1]
colourNames  = ['red', 'blue', 'green']

maxFreq = max(colourCounts)
maxFreqLoc = colourCounts.index(maxFreq)

mode = colourNames[maxFreqLoc]

print(mode)
```

Script output:

```
red
```

Mean

Method 2: Using statistics library



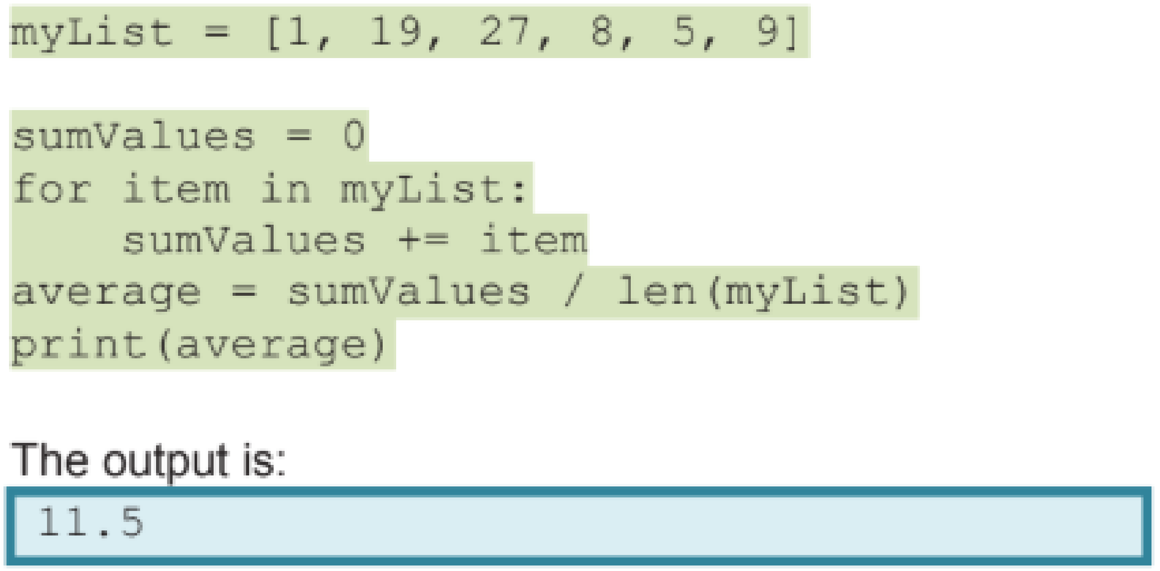
```python
import statistics

myList = [1, 19, 27, 8, 5, 9]
average = statistics.mean(myList)
print(average)
```

The output is:

```
11.5
```

Mean

Method 3: Manually Coding



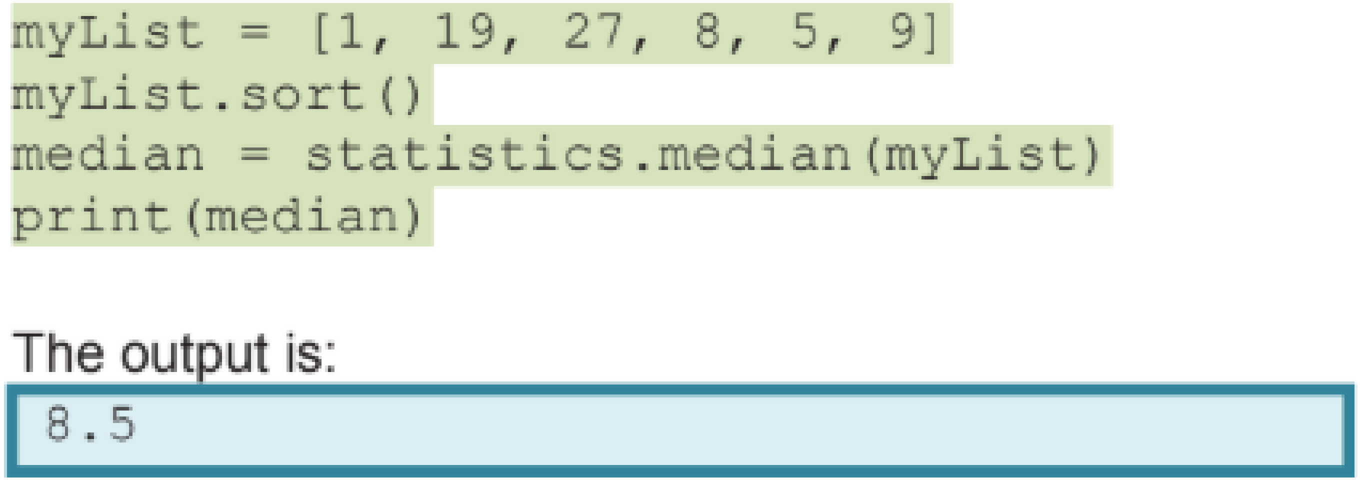
```
myList = [1, 19, 27, 8, 5, 9]

sumValues = 0
for item in myList:
    sumValues += item
average = sumValues / len(myList)
print(average)
```

The output is:

```
11.5
```

Median

Method 1: Using the statistics library



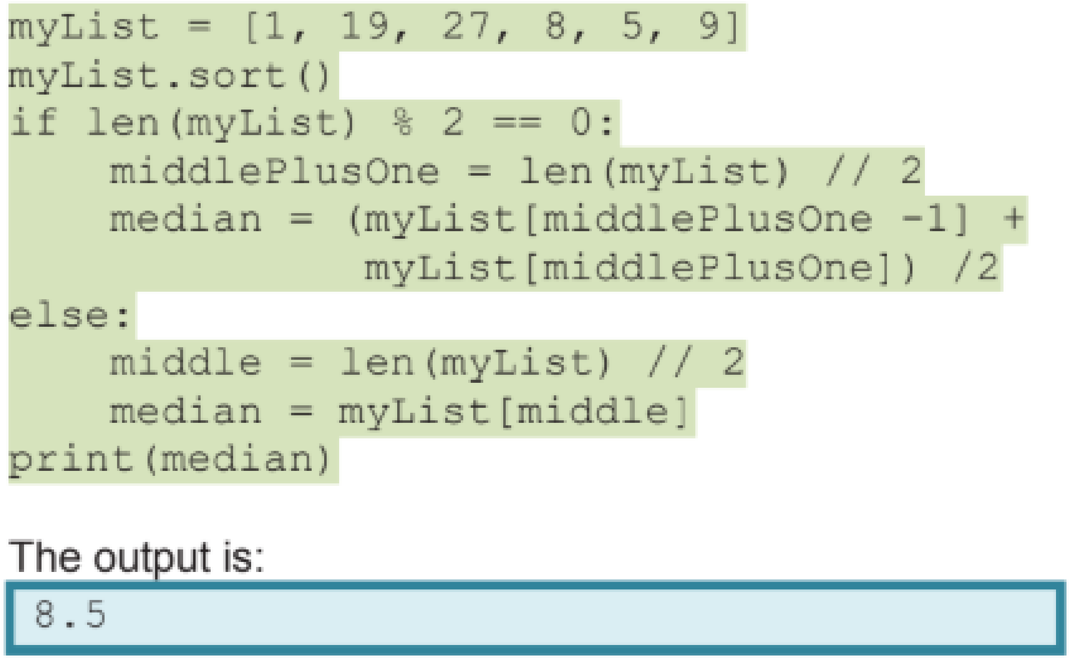
```python
myList = [1, 19, 27, 8, 5, 9]
myList.sort()
median = statistics.median(myList)
print(median)
```

The output is:

```
8.5
```

Median

Method 2: Manually Coding



```
myList = [1, 19, 27, 8, 5, 9]
myList.sort()
if len(myList) % 2 == 0:
    middlePlusOne = len(myList) // 2
    median = (myList[middlePlusOne -1] +
              myList[middlePlusOne]) /2
else:
    middle = len(myList) // 2
    median = myList[middle]
print(median)
```

The output is:

```
8.5
```

Note: The median is the middle number of a sorted list.

If there is an even number of elements in the list then the median is the middle two numbers added together and divided by 2.

Frequency

Frequency is the number of times elements appear in a list, this can be for strings or numbers.

Frequency

```
myList = ["red", "blue", "blue", "red",
"green", "red", "red"]

colourNames = []
colourCounts = []

for item in myList:
    if item not in colourNames:
        colourNames.append(item)
for colour in colourNames:
    total = myList.count(colour)
    colourCounts.append(total)

print(colourCounts)
print(colourNames)
```

Script output:

```
[4, 2, 1]
['red', 'blue', 'green']
```

Creates two empty lists

Loops through myList. If the item is not already in the colourNames list then add it to the list.

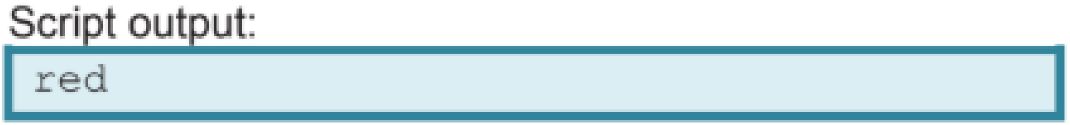
Loops through the colourNames list and counts the amount of times each colour is in myList

Mode

Mode is the value that occurs most often in the dataset.

Method 1: Using statistics library

```
import statistics

myList = ["red", "blue", "blue", "red",
"green", "red", "red"]

mode = statistics.mode(myList)
print(mode)
```



Script output:

```
red
```

Mode

Method 2: Manually Coding

```
colourCounts = [4, 2, 1]
colourNames  = ['red', 'blue', 'green']

maxFreq = max(colourCounts)
maxFreqLoc = colourCounts.index(maxFreq)

mode = colourNames[maxFreqLoc]

print(mode)
```

Script output:

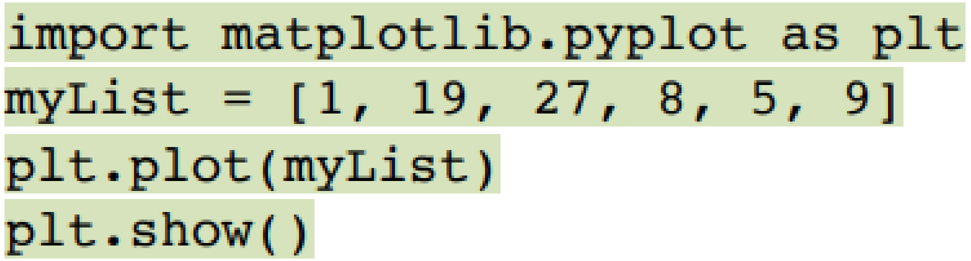
```
red
```

Visualisation/Graphing

It is easy to create visualisations in Python using a module named matplotlib.

We need our data in a list(s) to be able to create a graph.

Plotting graphs

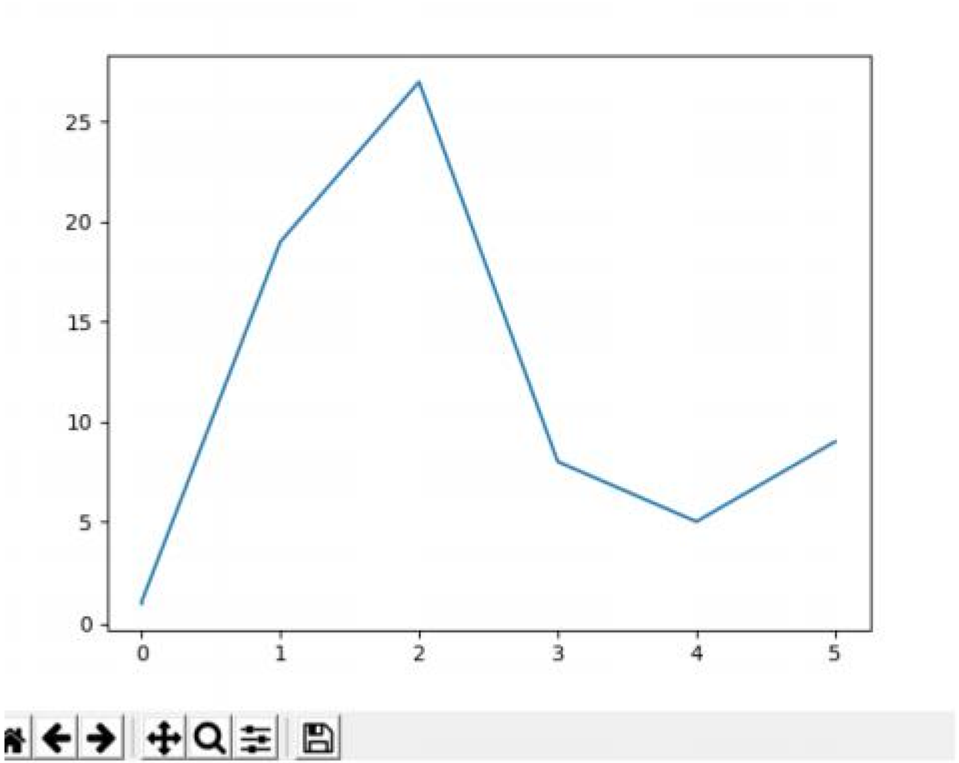


```
import matplotlib.pyplot as plt
myList = [1, 19, 27, 8, 5, 9]
plt.plot(myList)
plt.show()
```

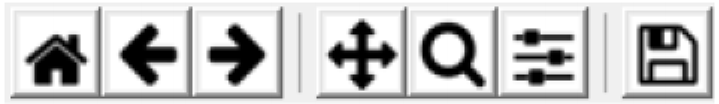
The first line uses an interface pyplot, from the matplotlib module. It is mainly used for interactive plots and simple plot generation, which is what we want.

We import as plt so that we don’t have to type matplotlib.pyplot every time we want to use it. We can just type plt instead.

The output looks like this



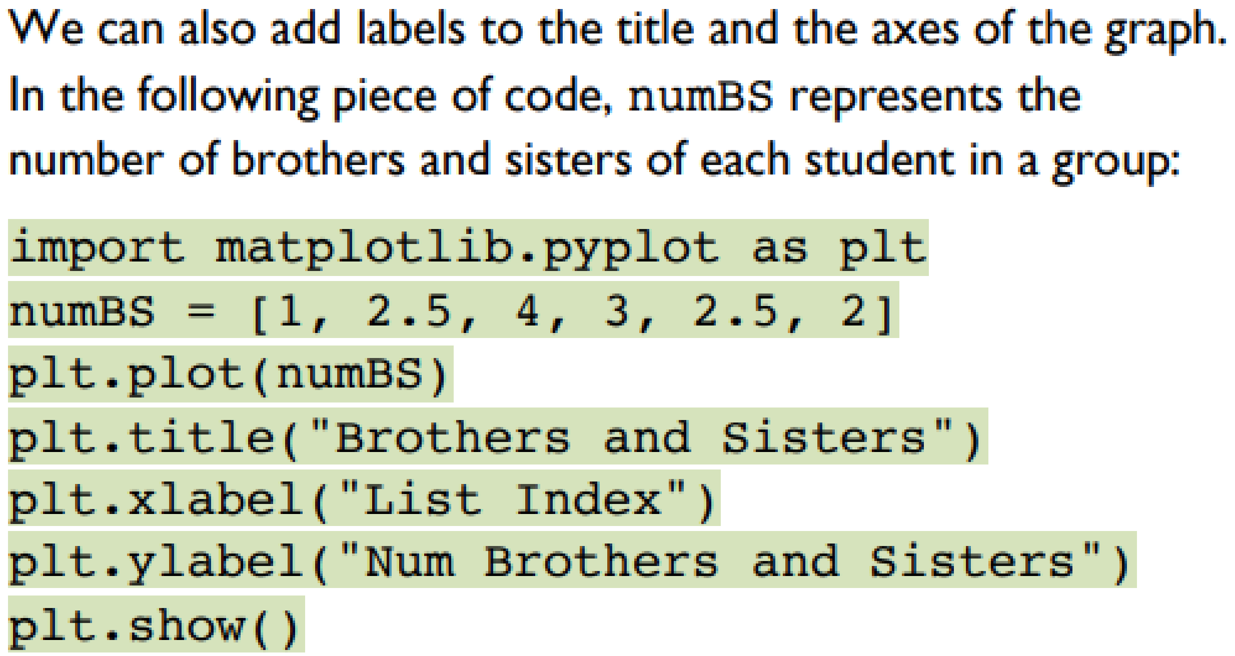
The buttons on the `matplotlib` output are useful.



From left to right:

- The home button resets the view to the original.
- The left and right arrows go back and forward between views, like on a web browser.
- The multi-directional arrows let you grab and move the graph (very useful while zoomed in).
- The magnifying glass, allows you to zoom in on a particular area of the graph, which is useful for detailed graphs like heart rate.
- The sliders allow you to adjust the subplot parameters, such as the location.
- The save icon allows you to save the plot image. This is useful for adding the results to a report or e-portfolio.

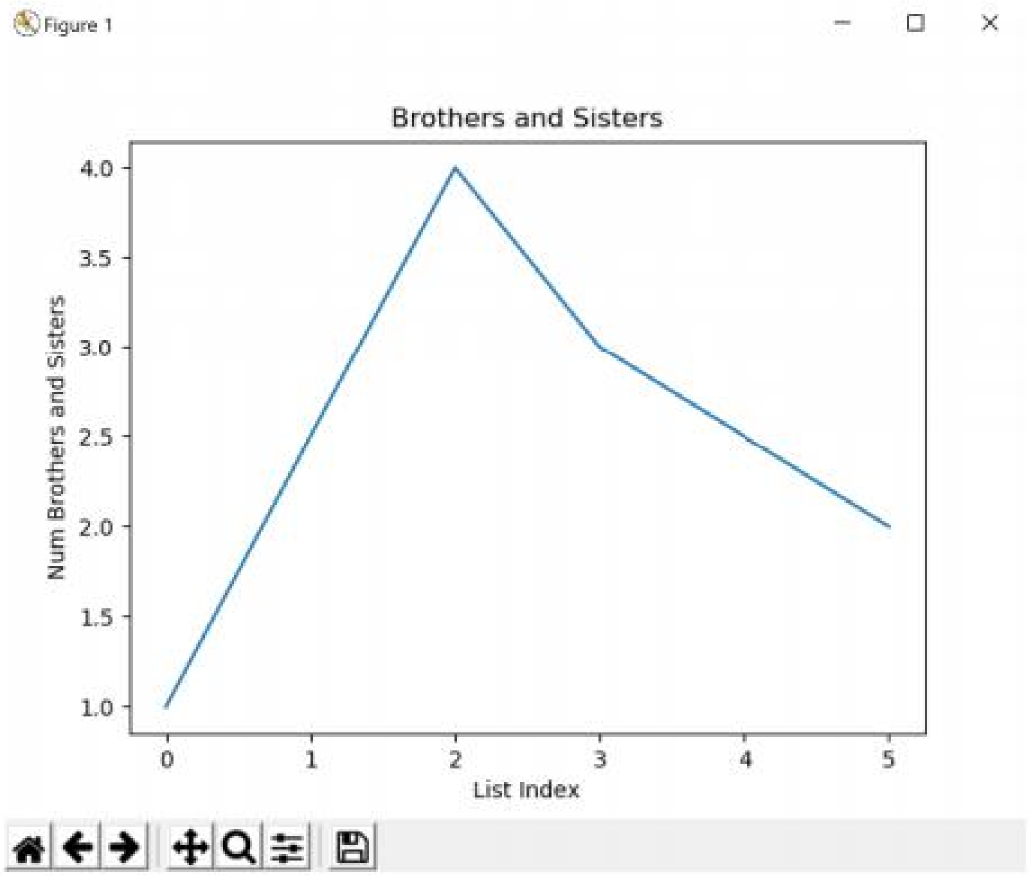
Labelling the graph



We can also add labels to the title and the axes of the graph. In the following piece of code, `numBS` represents the number of brothers and sisters of each student in a group:

```
import matplotlib.pyplot as plt
numBS = [1, 2.5, 4, 3, 2.5, 2]
plt.plot(numBS)
plt.title("Brothers and Sisters")
plt.xlabel("List Index")
plt.ylabel("Num Brothers and Sisters")
plt.show()
```

This is the result:



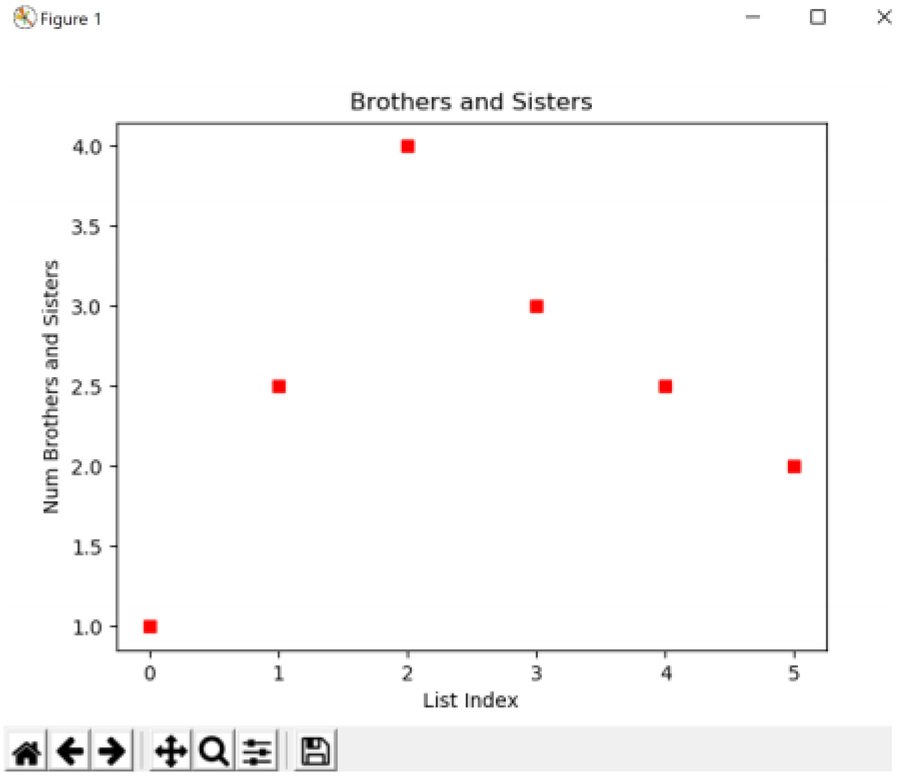
Graphing Options

A line graph may not always be suitable. For example, a line between points suggests that they are linked in some way.

The following code plots points, with no lines.

```
plt.plot(numBS,"rs")
```

Resulting in:



The line-style typically consists of two or three characters, the first represents the colour and the second/third indicates the point-style. The default line-style is `b-`.

The following table gives some more examples of line-styles.

Arguments for line-style	Colour	Point-style
`b-`	Blue	————————
`r--`	Red	----------------
`y-.`	Yellow	-.-.-.-.-.-.-.-
`b:`	Blue	....
`rs`	Red	Squares, no line
`bo`	Blue	Circles, no line

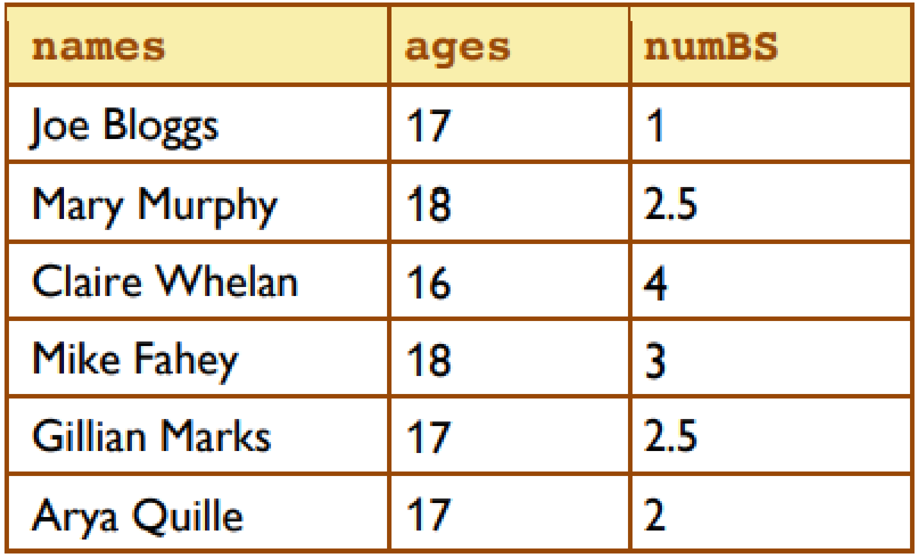
Samples of arguments for line-styles using `matplotlib`

x and y axis

We can show labels on our axis.

Using the following table, we can graph the students and the number of brothers/sisters.

Note: there are two inputs so we need two lists.



names	ages	numBS
Joe Bloggs	17	1
Mary Murphy	18	2.5
Claire Whelan	16	4
Mike Fahey	18	3
Gillian Marks	17	2.5
Arya Quille	17	2

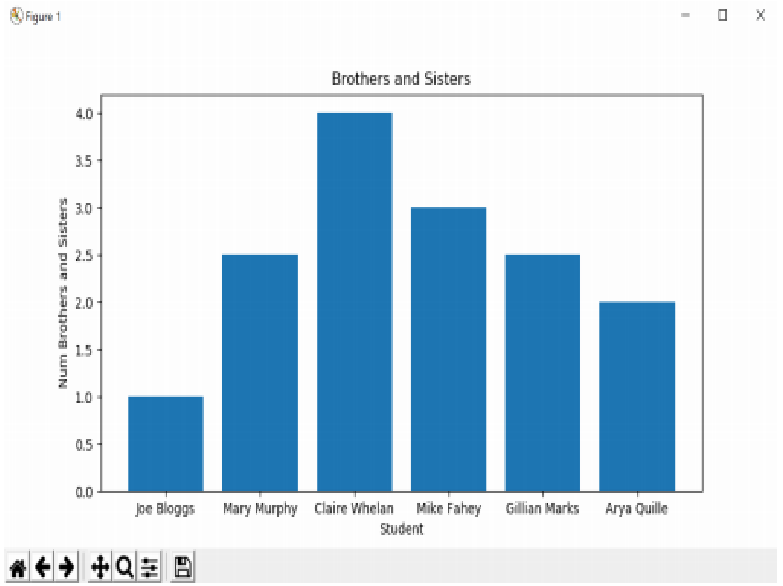
```
import matplotlib.pyplot as plt
numBS = [1, 2.5, 4, 3, 2.5, 2]
names = ["Joe Bloggs", "Mary Murphy",
"Claire Whelan","Mike Fahey",
"Gillian Marks", "Arya Quille"]
plt.plot(names, numBS)
plt.title("Brothers and Sisters")
plt.xlabel("Student")
plt.ylabel("Num Brothers and Sisters")
plt.show()
```

Note: The first argument represents the x-axis and the second argument represents the y-axis

Additional plot types - Bar Chart

The following code outputs a bar chart. Try it:

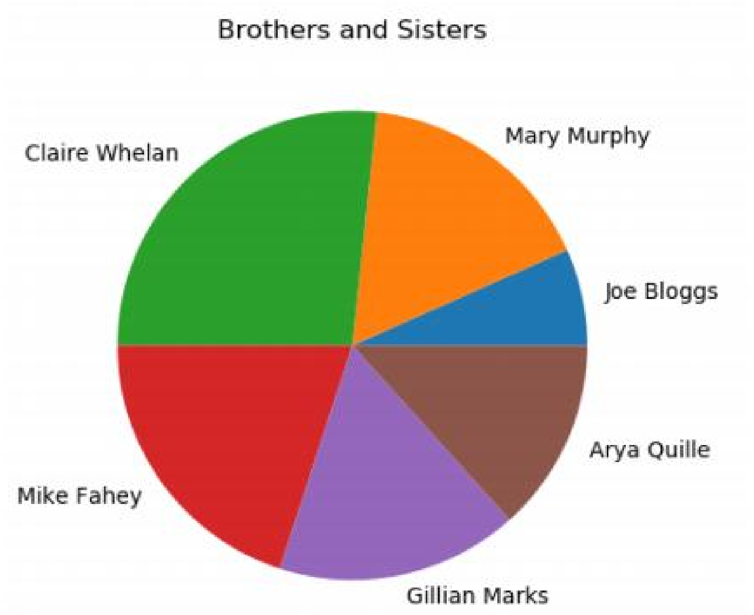
```
import matplotlib.pyplot as plt
numBS = [1, 2.5, 4, 3, 2.5, 2]
names =  ["Joe Bloggs","Mary Murphy",
"Claire Whelan","Mike Fahey",
"Gillian Marks", "Arya Quille"]
plt.bar(names, numBS)
plt.title("Brothers and Sisters")
plt.xlabel("Student")
plt.ylabel("Num Brothers and Sisters")
plt.show()
```



Pie Chart

```
import matplotlib.pyplot as plt
numBS = [1, 2.5, 4, 3, 2.5, 2]
names = ["Joe Bloggs", "Mary Murphy",
"Claire Whelan", "Mike Fahey",
"Gillian Marks", "Arya Quille"]
plt.pie(numBS, labels=names)
plt.title("Brothers and Sisters")
plt.show()
```

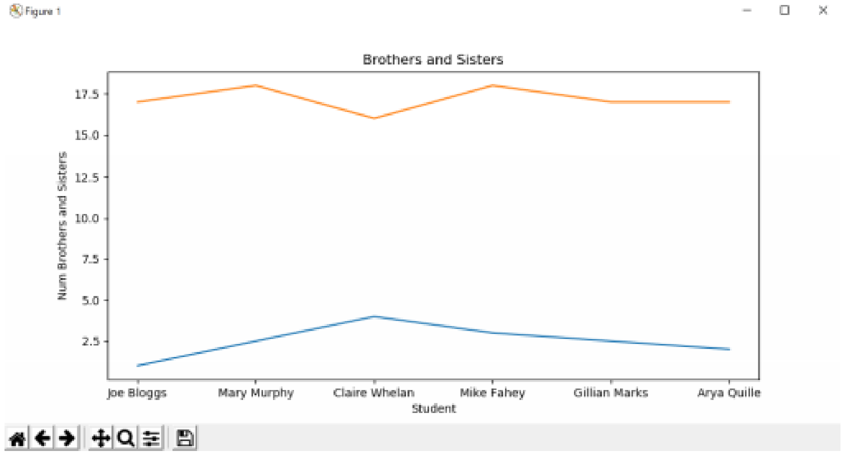
Here is the output:



Displaying two datasets on one graph

```
import matplotlib.pyplot as plt
numBS = [1, 2.5, 4, 3, 2.5, 2]
ages = [17, 18, 16, 18, 17, 17]
names = ["Joe Bloggs", "Mary Murphy",
"Claire Whelan", "Mike Fahey",
"Gillian Marks", "Arya Quille"]
plt.plot(names, numBS)
plt.plot(ages)
plt.title("Brothers and Sisters")
plt.xlabel("Student")
plt.ylabel("Num Brothers and Sisters")
plt.show()
```

Here is the output:



Note that `matplotlib` used the same y-axis for both sets of data. In our example, ages have greater values than those for the number of brothers and sisters. If the scales were significantly different, what issues might arise? How would you address this?

If you had three or four plots on the same graph you might get confused about what each line represents. In the previous example, it is obvious from its values what each line represents but this may not always be the case.

A solution to this problem is to add a legend. A legend takes a list as its argument, where the order of items in the list corresponds to the order of the lists plotted on the y-axis.

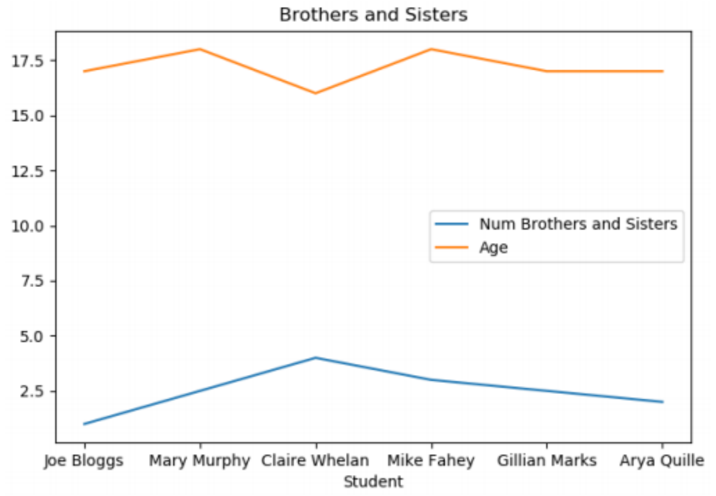
This is the code for adding a legend:

```
plt.legend(["Num Brothers and Sisters",
"Age"])
```

We add the above line to the code as follows:

```
import matplotlib.pyplot as plt
numBS = [1, 2.5, 4, 3, 2.5, 2]
ages = [17, 18, 16, 18, 17, 17]
names = ["Joe Bloggs", "Mary Murphy",
"Claire Whelan", "Mike Fahey",
"Gillian Marks", "Arya Quille"]
plt.plot(names, numBS)
plt.plot(ages)
plt.legend(["Num Brothers and Sisters",
"Age"])
plt.title("Brothers and Sisters")
plt.xlabel("Student")
plt.show()
```

The output now includes a legend:


Brothers and Sisters
Num Brothers and Sisters
Age
Joe Bloggs
Mary Murphy
Claire Whelan
Mike Fahey
Gillian Marks
Arya Quille
Student