SESSION 2.2

Saving and analysing data

WELCOME BACK!!

- Welcome back!!
- In the last few sessions we've been getting you up to speed with the *basics* of python and `pandas`
- We've covered a lot of ground, but we've also been trying to keep it simple.
  - Data types like:
    - `int`, `float`, `str`, `bool`
    - `list`, `tuple`, `dict`
    - `pathlib.Path`, `pandas.DataFrame`
  - functions, methods, and attributes
  - `for` loops

Pandas recap

- We've also been introducing you to the `pandas` module
- We've shown you how to create `DataFrames` from scratch
- How to `read` in data from `.csv` and `.xlsx` files
  - `pd.read_csv()`, `pd.read_excel()`
- We've shown you how to `clean` your data
  - `df.dropna()`, `df.replace()`
- We've shown you how to `examine` your data
  - `df.head()`, `df.tail()`, `df[column].value_counts()`, `df.columns`
- We've shown you how to `slice` your data
  - `df.iloc[]`, `df.loc[]`, `df[]`
  - `df[df['column'] == 'value']`

Pandas recap

- The next thing for you to do is practice ‘recalling’ this stuff so…

It’s time for a quiz!

- You know where to go!
- You know they’re not graded

SET UP FOR THE SESSION

- This session we're going to 'finish' up the introductory python part of these workshops by generating some descriptive statistics, simple plots, and saving output.
- But first things first - we need to import the modules we'll need for this session

```
# import the needed modules in this cell
import pandas as pd # for working with dataframes
import pathlib as pl # for working with file paths
import numpy as np # to let me use the np.nan value in the next cell.
```

SET UP FOR THE SESSION

- Excellent! Modules imported and now we need some data.
- Let's go through the cell below to create the `movies_df` so that we have some data to work with.

SET UP FOR THE SESSION

```
# initialise some list objects that contains our data, this could also be a list of lists, or a dict
director = ['John Carpenter', '', 'Nicolas Winding Refn', 'Matthijs van Heijningen', 'Damien Chazell
names = ['The Thing', 'Blade Runner 2049', 'Drive',  'The Thing', 'Whiplash', 'Arrival', 'No Country
genre = ['Horror', 'Sci-Fi', 'Action', 'Horror', 'Drama', 'Sci-Fi', 'Drama', 'Comedy', 'Comedy', 'Da
year = ['1982', '2017', '2011', '2011', '2014', '2016', '2007', '2004', '2007', '1996']
imdb_score = [82, 80, 78, 62, np.nan, 79, 82, 73, 78, 81]
rt_critics = [82, 88, 93, 34, np.nan, 94, 93, 89, 91, 94]
rt_fans = [92.0, 79.0, 42.0, 82.0, 86.0, 69.0, 89.0, 93.0, 94.0, 79.0]
lead = ['male', 'Male', 'm', 'Female', 'm', 'Male', 'fem', 'Orgre', 'Male', 'Male']



#turning those lists into a dataframe
movies_df = pd.DataFrame(# opening brackets but moving to new line for readability, note the upperca
    list(zip(director, names, genre, year, imdb_score, rt_critics, rt_fans, lead)), #first argument
    columns = ['Director', 'Movie Title', 'Genre', 'Year of Release', 'ImdB Score', 'Rotten Tomatoes
    )# closing the first pair of brackets to complete the function call
movies_df.to_csv('movies_df_2_2.csv', index=False)#saving the dataframe to a csv file
movies_df.columns = [i.lower().replace(' ', '_') for i in movies_df.columns] #capitalising the colum
# cleaning the data a little.
movies_df = movies_df.replace(r'^\s*$', np.nan, regex=True).dropna().reset_index(drop=True) #
#notice how we've 'chained' a bunch of methods together to make all empty cells appear as 'np.nan',
#that one line of code! Imagine if it was a huge data set

movies_df.iloc[:, -1:] = movies_df.iloc[:, -1:].replace(to_replace = ["m"], value = "Male") #Fixing
movies_df.iloc[:, -1:] = movies_df.iloc[:, -1:].replace(to_replace = ["fem"], value = "Female") # fi
```

PH1

```
Index(['director', 'movie_title', 'genre', 'year_of_release', 'imdb_score',
       'rotten_tomatoes_score', 'rotten_tomatoes_fan_score', 'gender_of_lead'],
      dtype='object')
```

SET UP FOR THE SESSION

SAVING STUFF

- One of the key things we might want to do is **save** things.
- You might want to save clean data once you've made it.
  - In this case I've made the dataset somewhat from scratch and maybe I want to be able to save it so that we can send it onto other researchers,
  - Or maybe you have done lots of work cleaning a data set and you want to save it so that you can analyse it later.
- You might have made a table, or a chart, or a plot that you want to save so that you can use it in a report or a presentation.

SAVING DATA

- Saving your data is a good idea
- Fortunately the syntax for saving a dataframe is, at the heart of it, really easy.

```
1 ```{python}
2 #saving a dataframe to a csv file
3 name_of_dataframe.to_csv('path\to\where\you\want\to\save\filename.csv', index=False)
4
5 #saving a dataframe to an excel file
6 name_of_dataframe.to_excel('path\to\where\you\want\to\save\filename.xlsx', index=False)
7 ```
```

SAVING DATA

- So above, we have two examples of saving a `pandas DataFrame`, one to a `.csv` file and one to a `.xlsx` file. Let's just take a second and walk through the actual syntax we used above.
    1. The first thing we do is call the `variable` we want to work on, in this case using the name of our dataframe
    2. We follow this with a fullstop `.` because saving a file is `method`
    3. This is followed by `to_` and then the type of file we're exporting to, so `to_csv` for csvs or `to_excel`
    4. Then we open brackets `()` and we pass the `path` to where we want the file to live A. This can be an `r-string` B. Or it can be a `pl.Path()` object
- But it has to end in `\filename.filetype` (the filetype is the `.csv` or `.xlsx`, also called the `suffix` of the file)

Using an `r-string` or a `Path` is often a matter of personal preference, but we would encourage you to work with `Path` objects more often because they can be a lot more

flexible. Let's take an aside for a few minutes and look at what we can do with `Paths` to make our lives easier.

PATH COMPONENTS

- Just like a `DataFrame` has `attributes` like `.columns` and `.shape`
- A `Path` object also has attributes that can be useful, and these attributes are basically just the components of a file path.
- These `attributes` allow us to pick apart a `path` so that we can use the bits of it we need.
- To get a sense of this, in the cell below we'll create a `Path` to the `csv`

```
1 #make your path object on the next line
2 csv_file = pl.Path(r'../../data/movies_df_2_2.csv')#using the ../../ to move up two directories to t
```

Now that we have an example path, we can can look at some of the attributes.

.name

- The first attribute we're going to look at is the `name` attribute of a `Path` object.
- This is the name of the file or folder that the `path` points to including the file `suffix`.
- This is useful because it allows us to get the name of the file or folder without having to parse the `path` string.

```
1  # printing the file name
2  print(csv_file.name)# note that it is not followed by `()` because it is an attribute not a method
```

```
movies_df_2_2.csv
```

`stem` AND `suffix`

- The `name` attribute is useful, but sometimes we want to get the `stem` of the file (the name without the `suffix`) or the `suffix` of the file (the `.csv` or `.xlsx` at the end of the name)
- Like if we specficially want to find files with the same `stem` but different `suffixes` or if we want to save a new file with the same `stem` but a different `suffix`.

```
1 # printing the stem of the file
2 print(csv_file.stem)
3 # printing the suffix of the file
4 print(csv_file.suffix)
```

```
movies_df_2_2
.csv
```

`stem` AND `suffix`

- Boom.
- You'll notice that the filename also has the `.csv` `suffix`.
- Obviously if it was an excel workbook the extension would be `.xlsx` and if it was a jupyter notebook it would be `.ipynb`.
- All of these `attributes` return `strings` that you can perform all the usual `string operations` on, and we know how much you love those (honestly, it's endearing).
- So you can call `csv_file.stem.upper()` and it will return the `stem` of the file in uppercase.
- Or you can call `csv_file.suffix.replace('.', '')` and it will return the `suffix` of the file without the `.` at the start.

PARENT

- The `parent` attribute of a `Path` is the `folder` one level up from the file or folder that the `path` points to.
- So if the full `path` is
  - `C:\Users\username\Documents\programming_club\data\movies_df_2_2.csv`
- Then the `parent` of the `path` is
  - `C:\Users\username\Documents\programming_club\data`

```
1 # printing the parent of the file
2 print(csv_file.parent)
```

```
../../data
```

`paths` AND `saving`

- Now that we know about some of these `attributes` we can use them to make our lives a little easier when we're saving files.
- For example, we can use the `path` we saved to the `csv` file to save a new file in the same folder as the `csv` file without overwriting our old file.
- We just need to use the parent of the `path` and add a new `.name` (which includes with `suffix`).
- `Pathlib` is really good at this because it allows us to use `/` (forwardslash) to join a `string` onto the end of a `Path` object.

```
1 ```{python}
2 # new file path
3 new_csv_file = csv_file.parent / 'new_filename.csv'
4 ```
```

paths AND saving

- Or what if you wanted to save it into a another folder in the same project directory?
- You could use the `parent` of the `parent`!!

````
```{python}
# new file path
new_csv_file = csv_file.parent.parent / 'processed_data/new_filename.csv'
```
````

paths AND saving

- There's lots you can do with `pathlib` and we've only just scratched the surface.
- My favourite is to use the `pathlib .iterdir()` method to `loop` through all the files in a folder and do something with them.
- `[i for i in csv_file.parent.iterdir()]` will return a list of all the files in the folder that the `csv_file` is in.
- But thats a story for another day.

SAVING DATA

- Your next task is to save the 'cleaned' `movies_df` to a new `csv` file so that you don't lose all the hard work you've done cleaning it up.
- I suggest you save it to a csv with the `df.to_csv()` method.
- Pass the `path` between the brackets and make sure you set `index = False` so that the `index` of the `df` doesn't show up as a column in the exported `csv` file.
- Be really careful with the `path` you pass to the `to_csv()` method, you don't want to overwrite your original data (this is where the `pathlib attributes` can be really useful).
- But if you do overwrite your original data, don't worry, you can always download the file again, but that not always be true of the data you're working with.

DESCRIPTIVE STATISTICS

- Now that we've saved our data, we can start to look at some of the `descriptive statistics` of the data.
- You all know well what descriptive stats are, you've made lots of tables in your time, but we're going to look at how to do this in `pandas`.
- We can look at things like the `mean`, `median`, `mode`, `standard deviation`, `variance`, `range`, `quartiles`, and `percentiles` of the data.
- We can use the `df.describe()` method to get a summary of the data in the `DataFrame` (and with quarto we can render them in apa, but thats for later)

DESCRIPTIVE STATISTICS

- The `df.describe()` method is a really useful method for getting a summary of the data in a `DataFrame`.
- It can be called on a whole `DataFrame` or on a single `column` of a `DataFrame`.
- It returns a `DataFrame` with the `mean`, `standard deviation`, `min`, `max`, `quartiles`, and `count` of the data in the `DataFrame`.
- Think about how useful that is for a second.

.describe()

- Let's take a look at the `describe()` method in action.

```
desc = movies_df.describe()# describe the whole dataframe
desc
```

	imdb_score	rotten_tomatoes_score	rotten_tomatoes_fan_score
count	8.000000	8.000000	8.000000
mean	76.875000	83.750000	80.000000
std	6.685539	20.492159	17.566201
min	62.000000	34.000000	42.000000
25%	76.750000	87.250000	76.500000
50%	78.500000	92.000000	85.500000
75%	81.250000	93.250000	92.250000
max	82.000000	94.000000	94.000000

.describe()

- So you can see that the `describe()` method returns a `DataFrame` with the major `descriptive statistics` for the `numberical` columns in the `DataFrame`.
- It's really easy to clean this up and make it look nice.

```
desc.columns = [i.replace('_', ' ').title() for i in desc.columns]#cleaning up the column names
desc.index = [i.title() for i in desc.index]#cleaning up the index
desc.round(2)#rounding the numbers to 2 decimal places
```

	Imdb Score	Rotten Tomatoes Score	Rotten Tomatoes Fan Score
Count	8.00	8.00	8.00
Mean	76.88	83.75	80.00
Std	6.69	20.49	17.57
Min	62.00	34.00	42.00
25%	76.75	87.25	76.50
50%	78.50	92.00	85.50
75%	81.25	93.25	92.25

	Imdb Score	Rotten Tomatoes Score	Rotten Tomatoes Fan Score
Max	82.00	94.00	94.00

`.describe()`

- We can also just slice the `DataFrame` returned by the `describe()` method to get specific stats

```
1 desc.loc[['Mean', 'Std', 'Min', 'Max']]
```

	Imdb Score	Rotten Tomatoes Score	Rotten Tomatoes Fan Score
Mean	76.875000	83.750000	80.000000
Std	6.685539	20.492159	17.566201
Min	62.000000	34.000000	42.000000
Max	82.000000	94.000000	94.000000

.describe()

- We can also call the `describe()` method on a single `column` of a `DataFrame` to get the `descriptive statistics` for that `column`.

```
1 movies_df['director'].describe()
```

```
count                8
unique               7
top       Coen Brothers
freq                 2
Name: director, dtype: object
```

```
1 movies_df['imdb_score'].describe()
```

```
count     8.000000
mean     76.875000
std       6.685539
min      62.000000
25%      76.750000
50%      78.500000
75%      81.250000
max      82.000000
Name: imdb_score, dtype: float64
```

`.describe()`

- The `describe()` method is really useful for getting a quick summary of the data in a `DataFrame`.
- But what if we want to get the `mode` of the data?
- Or the `quartiles` of the data?
- Or the `percentiles` of the data?
- We can use the `df.mode()`, `df.min()`, `df.max()`, `df.quantile()`, and `df.percentile()` methods to get these stats.
- Let's take a look at these in action.

OTHER STATS

- Just like `.describe()` we can call these methods on a whole `DataFrame` or on a single `column` of a `DataFrame`.
- But we need to specify only numberical columns when we call these methods on a whole `DataFrame`.

```
movies_df[['imdb_score', 'rotten_tomatoes_score']].min()#mode of the imdb_score column
```

```
imdb_score              62.0
rotten_tomatoes_score    34.0
dtype: float64
```

```
movies_df['imdb_score'].mode()#mode of the imdb_score column
```

```
0    78.0
1    82.0
Name: imdb_score, dtype: float64
```

OTHER STATS

- We can use that `syntax` for any individual descriptive stat we might want
- so `.quantile()`, `.percentile()`, `.std()`, `.var()`, `.mean()`, `.median()`, `.mode()`, `.min()`, `.max()`

```
print(movies_df['imdb_score'].quantile(0.25))#first quartile of the imdb_score column
print(movies_df['imdb_score'].quantile(0.75))#third quartile of the imdb_score column
print(movies_df['imdb_score'].mean())#mean of the imdb_score column
```

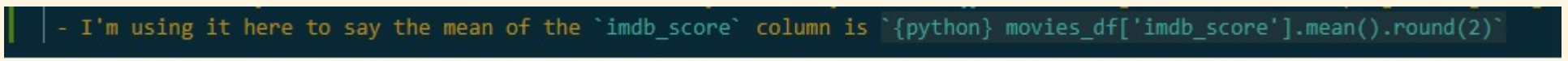
```
76.75
81.25
76.875
```

OTHER STATS

- So you can see that we can get a lot of `descriptive statistics` from a `DataFrame` really easily.
- As an academic writer you can then use these in lots of ways.
- You can not only make tables of these stats, but you can also use them in your writing to describe the data you're working with.
- This is a quarto feature rather than a python feature but you can use `inline code`

INLINE CODE

- the syntax for this is to use `` `{python} df[column].mean()` `` and then you can use the `quarto apa` style to render the output in `apa` style.
- for `R` the syntax is `` `{r} mean(df$column)` ``
- in both cases you need the backtik ‘`’ followed by the curly braces `{}` containing the name of the programming language you’re using and then the code you want to run.
- I’m using it here to say the mean of the `imdb_score` column is 76.88



inlinecode

PLOTS

- plotting is a whole thing in python and R.
- There are genuinely amazing libraries for plotting in both languages.
- And entire books on the subject.
- But we're going to keep it simple and just show you how to make a few basic plots in `pandas`.

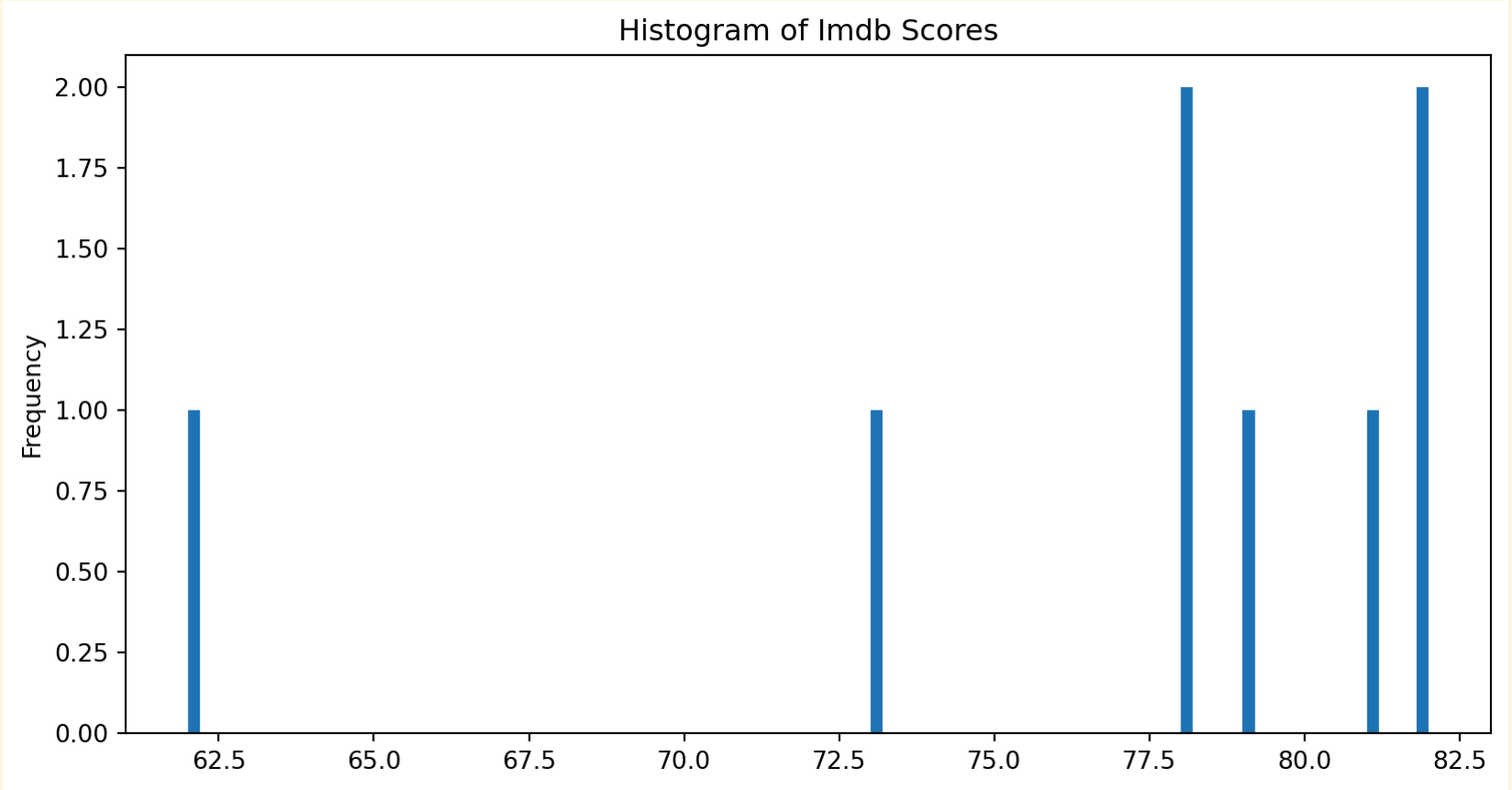
PLOTS

- the main plotting library in python is called `matplotlib` and it's really powerful.
- it has approximately 1 gajillion different ways to plot data…. which is …great… but also a bit overwhelming.
- `pandas` has a `plot()` method that is built on top of `matplotlib` and makes it really easy to make simple plots.
- We can use the `plot()` method to make `line`, `bar`, `scatter`, `hist`, `box`, `density`, `area`, `pie`, `hexbin`, and `kde` plots.
- We can also use the `plot()` method to make `subplots` and `stacked` plots.
- Let's take a look at a `hist` plot of the `imdb_score` column.

PLOTS

- The `plot()` method is really easy to use.
- We just call the `plot()` method on a `DataFrame` or a `column` and pass the `kind` of plot we want to make.
- We can also pass a bunch of other `arguments` to the `plot()` method to customise the plot.
- Let's take a look at a `hist` plot of the `imdb_score` column.

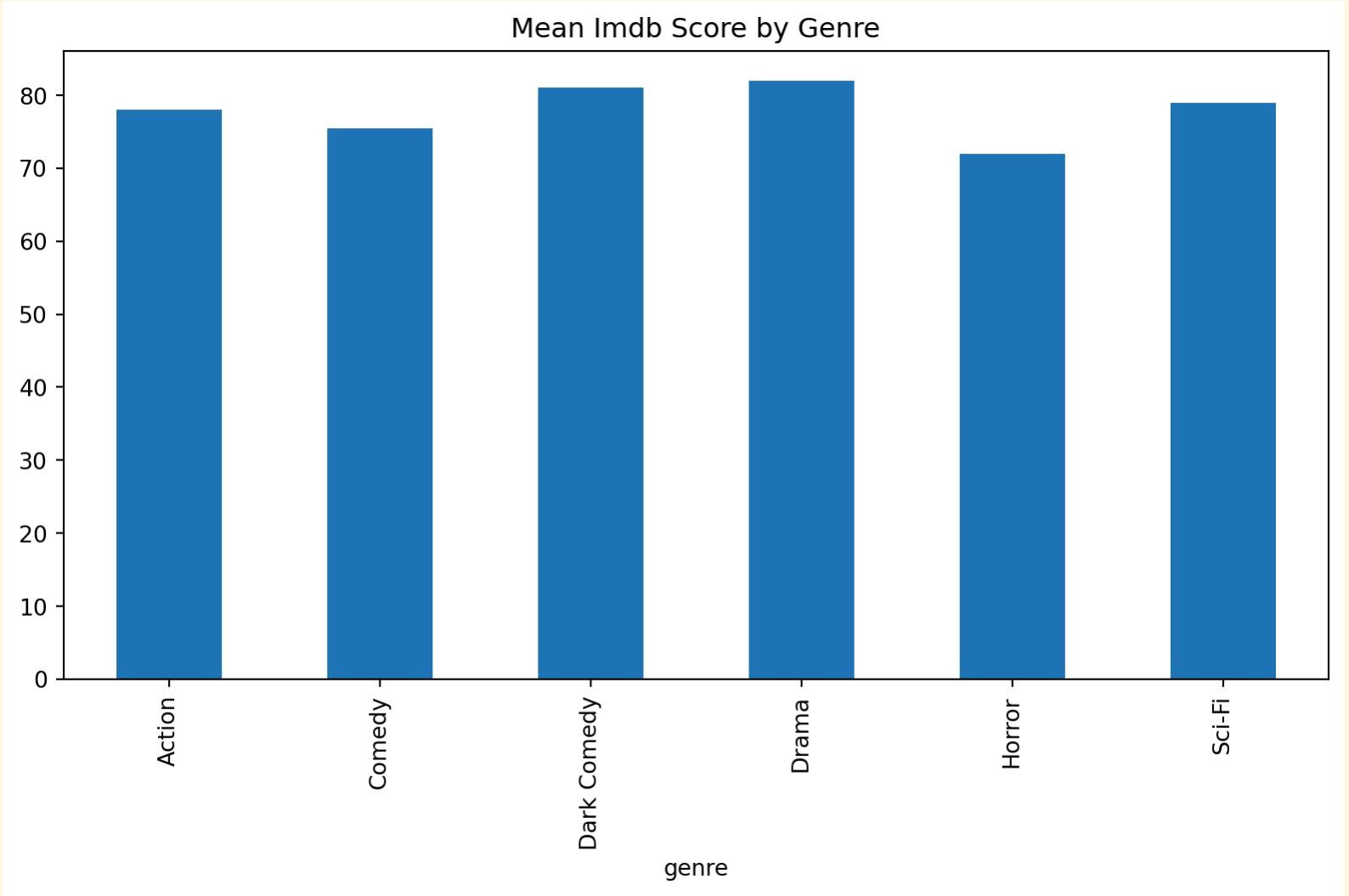
```
1 histogram = movies_df['imdb_score'].plot(kind = 'hist', bins = 100, title = 'Histogram of Imdb Score
```


Histogram of Imdb Scores
Frequency
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
62.5
65.0
67.5
70.0
72.5
75.0
77.5
80.0
82.5

PLOTS

- Ok… that wasn't a great plot.
- There isn't enough data… but you can see that the `plot()` method is really easy to use.
- We can also make a `bar` plot showing the `mean` of the `imdb_score` for each `genre` in the `DataFrame`.

```
1 bar = movies_df.groupby('genre')['imdb_score'].mean().plot(kind = 'bar', title = 'Mean Imdb Score by
```


Mean Imdb Score by Genre
80
70
60
50
40
30
20
10
0
Action
Comedy
Dark Comedy
Drama
Horror
Sci-Fi
genre

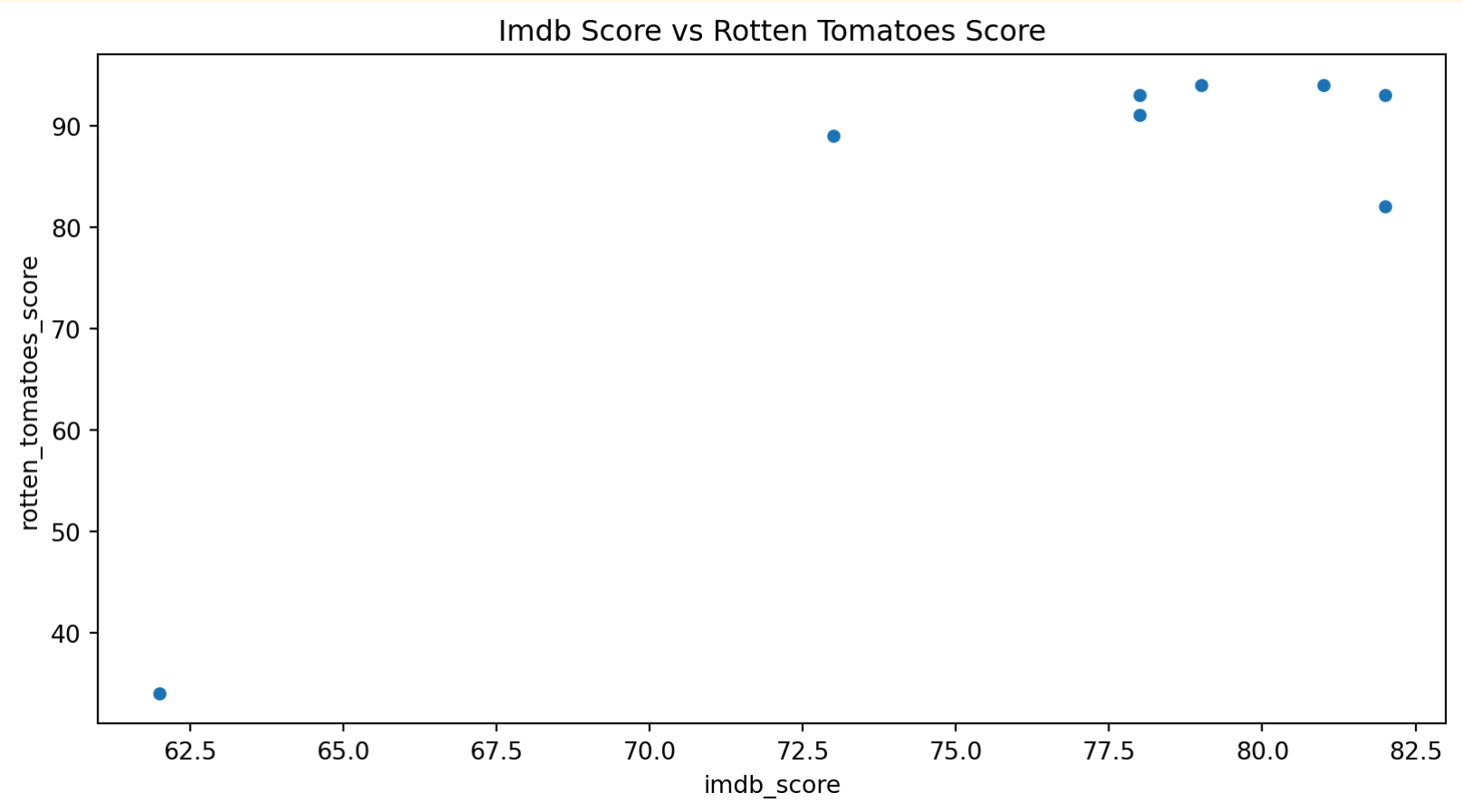
PLOTS

- `movies_df.groupby('genre')['imdb_score'].mean().plot(kind = 'bar', title = 'Mean Imdb Score by Genre')`
- Let's walk through this code a little bit.
  - We're calling the `groupby()` method on the `DataFrame` and passing the `column` we want to group by.
  - We then pass the `column` we want to get the `mean` of.
  - We're then calling the `mean()` method on the `column` we want to get the `mean` of.
  - We're then calling the `plot()` method on the `mean` object and passing the `kind` of plot we want to make.
  - We're also passing a `title` to the `plot()` method to give the plot a title.
- **All as one line of code.**

PLOTS

- We can also make a `scatter` plot of the `imdb_score` and `rotten_tomatoes_score` columns.

```
1 scatter_plot = movies_df.plot(kind = 'scatter', x = 'imdb_score', y = 'rotten_tomatoes_score', title
```



PLOTS

- As you can see, the plot method is really easy to use.
- There's loads of options though and you won't remember them all.
- The `bar`, `scatter`, and `hist` plots are the most common plots you'll use
- Because we generally use them to inspect the data as part of the prep and preliminary analysis of the data.
- If you need a specific plot there are literally 1000s of online video and written tutorials on how to make them.
- Also, if you learn how to make a plot in `matplotlib` you will be really really employable, way beyond academia.

SAVING PLOTS

- Just like we can save a `DataFrame` to a `csv` or `xlsx` file, we can also save a plot to a `png` or `pdf` file.
- We can use the `savefig()` method to save a plot to a file.
- We just pass the `path` to the `savefig()` method and it will save the plot to that `path`.
- Let's take a look at how to save the `scatter` plot we just made.

SAVING PLOTS

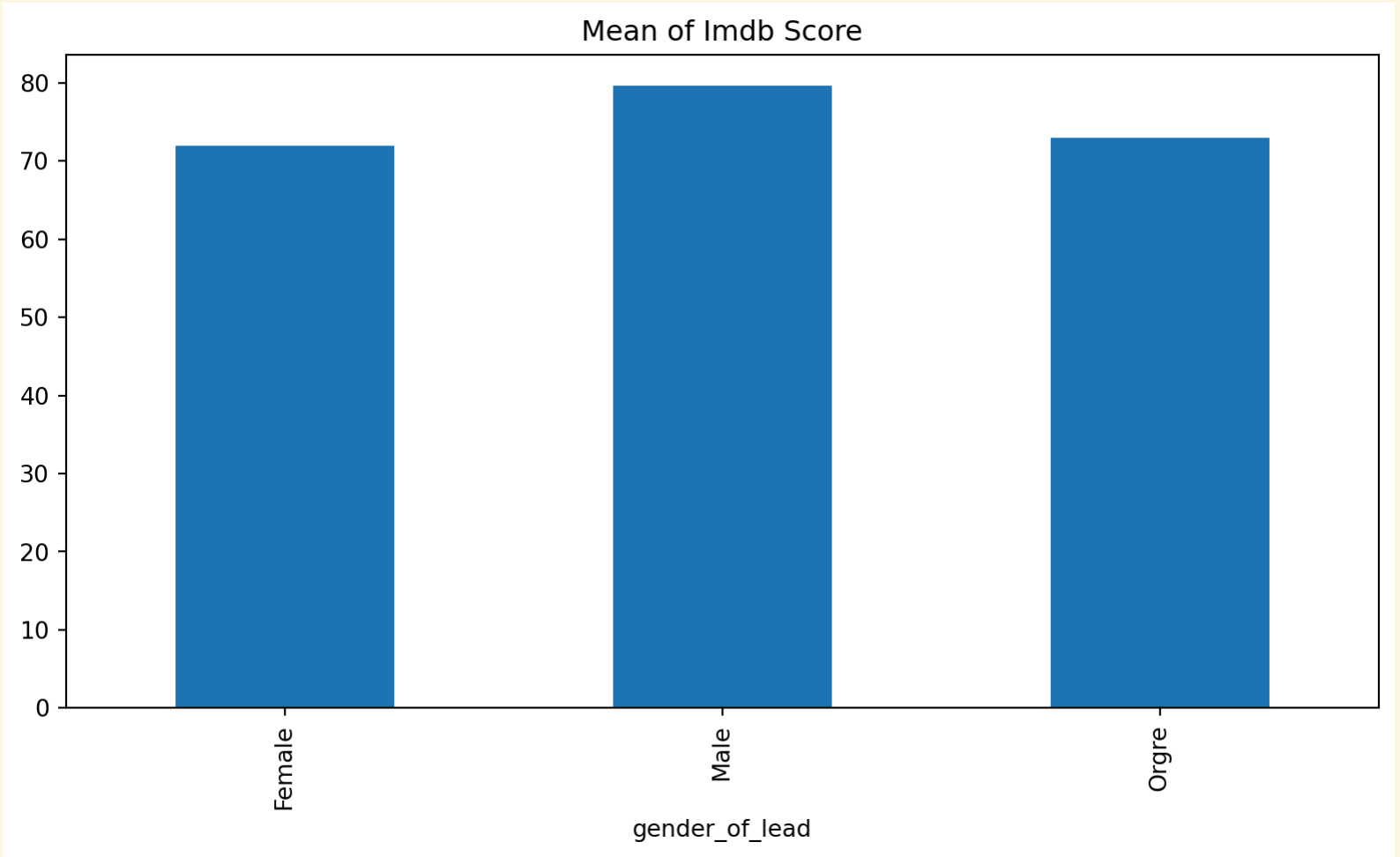
- The `savefig()` method is really easy to use but we have to call it after another method for it to work.
- We need to call the `get_figure()` method on the plot to make it into a `figure` object.
- We can then call the `savefig()` method on the `figure` object and pass the `path` to the `savefig()` method.

```
1 scatter_plot.get_figure().savefig(csv_file.parent / 'scatter_plot.png')
2 scatter_plot
```

```
<Axes: title={'center': 'Imdb Score vs Rotten Tomatoes Score'}, xlabel='imdb_score',
ylabel='rotten_tomatoes_score'>
```

PLOTS PRACTICE

PLOTS PRACTICE



BUT WHAT `if`

- When we're working with data we often want to do something `if` a certain condition is met.
- For example, if the mean, the median, and the mode of a column are all the same then we might use that as one of the tests for normality.
- We can use the `if-elif-else` statement to do this.
- These are really simple, but really powerful tools for controlling the flow of a program.

`if` STATEMENTS

- The `if` statement is the most basic of the `if-elif-else` statements.
- It allows us to execute a block of code if a certain condition is met.
- The syntax is really simple.
- We use the `if` keyword followed by the condition we want to test (using the `boolean` opperators that we covered previously).
- We then open a block of code with a colon `:` and indent the code we want to execute if the condition is met.
- Let's look at a simple example.

```
x = 10 # setting the value of x
if x > 5: # testing if x is greater than 5
    print('x is greater than 5')
else: # if x is not greater than 5
    print('x is not greater than 5')
```

```
x is greater than 5
```

if STATEMENTS

- We can use if statements to test lots of things.
- For example, we can test if the mean of a column is equal to the median of a column.

```
1 if movies_df['imdb_score'].mean() == movies_df['imdb_score'].median():
2     print('The mean and median of the imdb_score column are the same')
3 else:
4     print('The mean and median of the imdb_score column are not the same')
```

```
The mean and median of the imdb_score column are not the same
```


- We can use if statements to test lots of things.
- For example, we can test if the mean of a column is equal to the median of a column.


```
1 if movies_df['imdb_score'].mean() == movies_df['imdb_score'].median():
2     print('The mean and median of the imdb_score column are the same')
3 else:
4     print('The mean and median of the imdb_score column are not the same')
```

```
The mean and median of the imdb_score column are not the same
```


Well if the mean and the median are the same then the data is more likely to be normal.

`if-elif-else` STATEMENTS

- The `if-elif-else` statement is a more complex version of the `if` statement.
- It allows us to test multiple conditions and execute different code depending on which condition is met.
- The syntax is not much more complex than the `if` statement.

```
x = 10 # setting the value of x
if x > 5: # testing if x is greater than 5
    print('x is greater than 5')
elif x < 5: # if x is not greater than 5
    print('x is less than 5')
else: # if x is not greater than 5
    print('x is equal to 5')
```

```
x is greater than 5
```

pandas and if-elif-else

- We could use some `if` logic to do other things with the `DataFrame`
- For example, we could use `if` logic to check the `dtype` (which is an attribute) of a column and then do something depending on the `dtype`.
- if the `dtype` is `float64` we could `describe()` the column, if it's `object` we could `value_counts()` the column, if it's `int64` we could `plot()` the column.

```
1 if movies_df['imdb_score'].dtype == 'float64':
2     print(movies_df['imdb_score'].describe())
3 elif movies_df['imdb_score'].dtype == 'object':
4     print(movies_df['imdb_score'].value_counts())
5 else:
6     movies_df['imdb_score'].plot(kind = 'hist')
```

```
count     8.000000
mean     76.875000
std       6.685539
min      62.000000
25%      76.750000
50%      78.500000
75%      81.250000
```

```
max      82.000000
Name: imdb_score, dtype: float64
```

`if-elif-else` PRACTICE

- In your own notebook you should:
- Use an `if-elif-else` statement to check if the `mean` of the `imdb_score` column is greater than the `median` of the `imdb_score` column.
- If the `mean` is greater than the `median` then print 'The mean is greater than the median', if not then print 'The mean is not greater than the median'.
- Use an `if-elif-else` statement to check if the `dtype` of the `imdb_score` column is `float64`, `object`, or `int64`.
- If the `dtype` is `float64` then `describe()` the column, if it's `object` then `value_counts()` the column, if it's `int64` then `plot()` the column.

SIMPLE ANALYSIS

- While there are a lot of powerful python tools for doing quite complex analyses in python, these involve using other `packages` like `scipy` and `pingouin`.
- To do a `t-test` for example we would use the `ttest_ind()` method from the `scipy.stats` module.
- To 'finish' this session we're just going to look at how you can use the `corr()` method to get the `correlation` between `columns` in a `DataFrame.

correlations

- Correlations are really easy to do in `pandas`.
- We can use the `corr()` method to get the `correlation` between `columns` in a `DataFrame`.
- We can specify the `method` of `correlation` we want to use by passing the `method` to the `corr()` method.
- We can use the `pearson`, `kendall`, and `spearman` methods of `correlation`.
- We have to specify `numeric_only = True` when we call the `corr()` method to get the `correlation` between `numeric columns`.

```
1 movies_df.corr(method = 'pearson', numeric_only=True)
```

	imdb_score	rotten_tomatoes_score	rotten_tomatoe
imdb_score	1.000000	0.876687	-0.032844
rotten_tomatoes_score	0.876687	1.000000	-0.135726

correlations

- You'll notice that the `corr()` method returns a `DataFrame` with the `correlation` between the `numeric columns` in the `DataFrame`.
- We can also check for `specific correlations` by combining slicing and the `corr()` method.
- As if we just wanted to see if the Rotten tomatoes score and the Rotten tomatoes fan score were correlated we could do this.

```
1 movies_df[['rotten_tomatoes_score', 'rotten_tomatoes_fan_score']].corr(method = 'pearson')
```

	rotten_tomatoes_score	rotten_tomatoes_fan_score
rotten_tomatoes_score	1.000000	-0.135726
rotten_tomatoes_fan_score	-0.135726	1.000000

correlations

- We can also use the `corr()` method to get the `correlation` between a `column` and some number of other columns
- If we wanted to store the `correlation` between the two different `rotten tomatoes` scores we could do this.

```
correlation = movies_df['rotten_tomatoes_score'].corr(movies_df['rotten_tomatoes_fan_score'], method

if correlation  > 0.3:
    print(f'The Rotten Tomatoes Score and the Rotten Tomatoes Fan Score are positively correlated, r
elif correlation < -0.3:
    print(f'The Rotten Tomatoes Score and the Rotten Tomatoes Fan Score are negatively correlated, r
else:
    print(f'The Rotten Tomatoes Score and the Rotten Tomatoes Fan Score are not correlated, r = {cor
```

```
The Rotten Tomatoes Score and the Rotten Tomatoes Fan Score are not correlated, r = -0.14
```

correlations

- So you can see that just using `pandas` we can import our data, clean it, slice it.
- We can then do some basic `descriptive statistics`, make some simple `plots`, and check for `correlations`.
- We could also do things like make certain columns into categories, or change the `dtype` of a column, or make a new column based on the values of other columns.
- In your own notebook, play around with correalting the `imdb_score` with the `rotten_tomatoes_score` and the `rotten_tomatoes_fan_score` and then save the `correlation` to a `csv` file.
- You could also make a `scatter` plot of the `imdb_score` and the `rotten_tomatoes_score` and save it to a `png` file.
- Use an `if-elif-else` statement to check if the `correlation` between the `imdb_score` and the `rotten_tomatoes_score` is greater than 0.3, less than -0.3, or between -0.3 and 0.3.

SUMMARY

- We've covered a lot in this session.
- We've looked at how to save data to a `csv` or `xlsx` file.
- We've looked at how to use `pathlib` to make working with `paths` easier.
- We've looked at how to get `descriptive statistics` from a `DataFrame`.
- We've looked at how to make `plots` in `pandas`.
- We've looked at how to save `plots` to a `png` or `pdf` file.
- We've looked at how to use `if-elif-else` statements to control the flow of a program.
- We've looked at how to use `correlations` to check for relationships between `columns` in a `DataFrame`.
- You've done a lot of work in this session, and you should be really proud of yourself.