SESSION 2.1

Make the kind of data you want to see in the world

WELCOME BACK!

- Last sesstion we took a look at importing data into a Pands `DataFrame`, using `.read_csv()` and `Path` objects.
- We also introduced you to the structure of a `DataFrame` through `class methods` like
  - `.head()` and `.tail()` (which display the top and bottom of your `df` respectively)
- and `attributes` like
  - the `df.shape`
  - the `df.columns`
  - and the `df.index`.

THIS SESSION

- This session we're going to look at some options for cleaning a `DataFrames` such as
- `dropping` unneeded `columns` and `rows`
- standardising or `renaming` columns
- as well as some other cool things (like) `slicing` your data, and 'fixing' inconsistent data.

RECAP QUIZ

- You know where the quiz is by now.

THIS SESSION'S DATA

- Make sure the virtual environment is activated
- pip install tabulate

```
from tabulate import tabulate
from IPython.display import Markdown
```

```
import pandas as pd #importing the pandas module
# initialise some list objects that contains our data, this could also be a list of lists, or a dict
import numpy as np #importing a module to allow me to include 'missing data

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]
lead = ['male', 'Male', 'm', 'Female', 'm', 'Male', 'fem', 'Orgre', 'Male', 'Male']
cry = ["No", "No", "No", "No", "No", "Yes", "No", "Yes", "No", "No"]
movies_df = pd.DataFrame(
    list(zip(director, names, genre, year, imdb_score, rt_critics, lead, cry)), # first argument, pa
    columns = ['Participant Name', 'Title of Thing', 'Genre', 'Year of Release', 'ImdB Score', 'Rott
    )# closing the first pair of brackets to complete the function call
```

A CLOSER LOOK AT THE DATA

- You already know how to take the wide view of your data with `head()`, `tail()` and `.shape`, and those things are really useful but, especially with really large `dataframes` or unprocessed (messy) dataframes, we also need to be able to get a more focused view of sections of our data.
- Pandas has a lot of different ways to do this, some of which are interchangeable, and we're going to spend the next little while introducing you to some of them.

RECAP OF SLICING AND INDEXING

- One of the most valuable operations you can perform to get a more focused view of your data is `slicing`.
- Before we learn about slicing, let's recap on indexing (slicing is essentially an extension of indexing).
- Remember that we have an object (`string`, `list`, etc) then each element in that object has a position.
- Indexing enables us to identify the which element is in a given position within that object. Let's say we have a list assigned to the variable, my_list

RECAP OF SLICING AND INDEXING

```
1 my_list = ["apples", "bananas", "cauliflower", "dorritos", "enchiladas", "fajitas", "ginger", "honey
```

- If we want to identify the first, fourth, and last element within my list, then we can use indexing to do that.

```
1 #my_list = ["apples", "bananas", "cauliflower", "dorritos", "enchiladas", "fajitas", "ginger", "hone
2 #           0,          1,             2,             3,           4,           5,         6,         7
3
4 print(my_list[0]) #Remember that when we index with Python, we start at 0.
5 print(my_list[3])
6 print(my_list[7])
```

```
apples
dorritos
honey
```

RECAP OF SLICING AND INDEXING

- Above we used positive indexing (where the index positions we use to identify elements start from 0 and increase from there).
- But we can also use negative indexing, which is really handy when you have a large object, and want to identify the last element within that object, and you do not know how many elements are in that object.

```
1 ```{python}
2 #my_list = ["apples", "bananas", "cauliflower", "dorritos", "enchiladas", "fajitas", "ginger", "hone
3 #           0,          1,            2,            3,           4,           5,         6,          7
4 #          -8,         -7,           -6,           -5,          -4,          -3,        -2,         -1
5
6 print(my_list[-8])
7 print(my_list[-5])
8 print(my_list[-1])
9 ```
```

RECAP OF SLICING AND INDEXING

- While the values we have used to index an element have changed, the output is the same.
- While indexing is useful, it is also limited. Each time we can extract only one particular element within my object. But what if we wanted to extract several elements all at once?

```
my_object[start_value:stop_value]
```

RECAP OF SLICING AND INDEXING

- You may have remember that while the element at the position of the start value is included in our `slice`
- the element at the position of our stop_value is not.
- This is because when we are slicing, our start_value is inclusive, but our stop_value is exclusive.
- If I wanted to include ‘dorritos’ into this smoothie of hell, I need to select one index position further

```
1 print(my_list[0:4])# evil smoothie ingredients
```

```
['apples', 'bananas', 'cauliflower', 'dorritos']
```

RECAP OF SLICING AND INDEXING

- Slicing is a rather apt name - it slices up our object to return a list of elements. We just need to tell Python how we want to slice our cake.
- For example, if I wanted to slice from our second element onwards, I could run the following code

```
1 print(my_list[1:])
```

```
['bananas', 'cauliflower', 'dorritos', 'enchiladas', 'fajitas', 'ginger', 'honey']
```

- By leaving an empty space in place of the stop_value, we are telling Python to begin at the start_value, and then include everything after that within our list.

RECAP OF SLICING AND INDEXING

- If we wanted to Python to start at the beginning of the object, and to return each element within a list up to a certain element, then we can leave an empty_space in our slice syntax

```
1 print(my_list[:4])
```

```
['apples', 'bananas', 'cauliflower', 'dorritos']
```

RECAP OF SLICING AND INDEXING

- Hopefully you can that slicing gives us a lot of flexibility, even if you are probably wondering when such functionality will come in handy. We will get on to that throughout the workshops (including todays one)
- Negative indexing also works with Python, and not only that, but you can also combined positive and negative indexing too.
- We can also use slicing on strings as well as lists.

RECAP OF SLICING AND INDEXING

- Before we move on to slicing a dataframe, there is one last piece of the slicing syntax that you need to know about.
- Not only can we specify a range for Python to slice out, we can specify the `steps` to that slicing:

```
my_object[start_value:stop_value: steps]
```

```
1 #example 1
2 print(my_list[::2])
3
4 #example 2
5 print(my_list[1:5:2])
```

```
['apples', 'cauliflower', 'enchiladas', 'ginger']
['bananas', 'dorritos']
```

RECAP OF SLICING AND INDEXING

- In example 1, `print(my_list[::2])`
- since we have a left a blank space in our start_value and our stop_value, we have asked Python to essentially to span across all the elements within our object.

Okay, let's see what we can do with `slicing` in relatiion to `Dataframes`.

SLICING OUR DATAFRAME

- You've already sliced a dataframe before.
- `.head()` and `.tail()` slice your dataframe to the top or bottom five rows
- We can slice a custom number of `rows` from our `DataFrame`, using the same `syntax` from earlier.

```
1 movies_df[-2:]# Note that I'm not using the '.loc[]'
2 # I'm not directly trying to access the index.
```

	Participant Name	Title of Thing	Genre	Year of Release	ImdB Score	Rotten Tomatoes Score	Gender of Lead	Make Me Cry?
8	Edgar Wright	Hot Fuzz	Comedy	2007	78.0	91.0	Male	No
9	Coen Brothers	Fargo	Dark Comedy	1996	81.0	94.0	Male	No

SLICING OUR DATAFRAME

The syntax for slicing a dataframe goes something like this:

`dataframe[row_start_value:row_end_value:steps]`

The `steps` argument is optional, and if you leave it out, Python will default to 1.

Slicing with this `syntax` returns the rows of a dataframe. It will not select specific columns to display.

SLICING OUR DATAFRAME - COLUMNS

- If we want to select columns then there's a few ways to do that.
- The first way is to pass a list of the columns that you want to select *within* the square brackets we're using for indexing.

```
1 movies_df[["Participant Name", "Title of Thing", "Genre"]].head()# note the double square brackets
```

	Participant Name	Title of Thing	Genre
0	John Carpenter	The Thing	Horror
1		Blade Runner 2049	Sci-Fi
2	Nicolas Winding Refn	Drive	Action
3	Matthijs van Heijningen	The Thing	Horror
4	Damien Chazelle	Whiplash	Drama

SLICING OUR DATAFRAME - COLUMNS

- Another option is to pass a `slice` of the `df.columns` attribute.

```
1 movies_df[movies_df.columns[0:3]].head() #This will return the first three columns of our dataframe
```

	Participant Name	Title of Thing	Genre
0	John Carpenter	The Thing	Horror
1		Blade Runner 2049	Sci-Fi
2	Nicolas Winding Refn	Drive	Action
3	Matthijs van Heijningen	The Thing	Horror
4	Damien Chazelle	Whiplash	Drama

SLICING OUR DATAFRAME - `location` SLICING

- But what if we want to select specific columns, and specific rows?
- There's two ways to do this as well:
  - The first is the `loc` method, which enables us to select rows and columns based on their `labels`
    - You've seen this one before when we talked about `df.index`
  - The second is the `iloc` method, which enables us to select rows and columns based on their `index position`

SLICING OUR DATAFRAME - `location` SLICING

- The `loc` method enables us to select rows and columns based on their labels.
- Remember when we set the `bands_df.index` attribute to be the `Participant Name` column?
- And then used `bands_df.loc["John Lennon"]` to return that row?
- Well the loc method actually lets us do a lot more, but the `syntax` is a little complex.

SLICING OUR DATAFRAME - `location` SLICING

```
1 movies_df.loc[0:2, 'Title of Thing': 'Year of Release']
```

	Title of Thing	Genre	Year of Release
0	The Thing	Horror	1982
1	Blade Runner 2049	Sci-Fi	2017
2	Drive	Action	2011

- Take a look at this code.
- Try to spot the similarities between this code and the code we used to slice our list earlier.
- But also the unexpected change in behaviour!

SLICING OUR DATAFRAME - `location` SLICING

- Essentially, we're passing 2 slices (separated by a comma) to the `loc` method.
- The first slice is for the rows, and the second slice is for the columns.
- And in both instances we're using the `labels` (Note that most of the time the `row labels` will be the numberical index, unless you have set the index, `.set_index()`, to be something else)
- If we want to select all the rows, we can leave the row slice empty.

```
movies_df.loc[:, "Participant Name":"Year of Release"]
```

- If we want to select all the columns, we can leave the column slice empty.

```
movies_df.loc["John Carpenter":"Damien Chazelle", :]
```

SLICING OUR DATAFRAME - `location` SLICING

- The last thing we can do with .loc is to select rows with specific values in a column.

```
movies_df.loc[movies_df["Genre"] == "Sci-Fi"].head() # note the '==' operator
# This is a boolean operation, and will return a dataframe with only the rows that have 'Sci-Fi' in
```

	Participant Name	Title of Thing	Genre	Year of Release	ImdB Score	Rotten Tomatoes Score	Gender of Lead	Make Me Cry?
1		Blade Runner 2049	Sci-Fi	2017	80.0	88.0	Male	No
5	Dennis Villanueve	Arrival	Sci-Fi	2016	79.0	94.0	Male	Yes

SLICING OUR DATAFRAME - `i`

`location` SLICING

- If, instead of using labels, we want to select rows and columns based on their index position, then we can use the `iloc` method.
- The `iloc` method is similar to the `loc` method, but instead of using labels, we use index positions.

```
1 movies_df.iloc[0:2, 1:3]
```

	Title of Thing	Genre
0	The Thing	Horror
1	Blade Runner 2049	Sci-Fi

BOOLEANS

- Before we move on to cleaning our dataframe, we need to talk about `booleans`.
- Booleans are a data type that can only have one of two values: `True` or `False`.
- Booleans are really useful when we want to filter our data based on a condition.
- For example, when we wanted to filter our data to only show the rows where the `Genre` column is equal to `Sci-Fi`, we used `==`.

BOOLEAN OPERATORS

- The `==` operator is a boolean operator that checks whether two values are *equal* (remember that the `=` operator is used to assign a value to a variable).
  - If the values are equal, then the `==` operator returns `True`, otherwise it returns `False`.

```
1 print(5 == '5') #This will return False
2 print(5 == 5) #This will return True
```

```
False
True
```

BOOLEAN OPERATORS

- There's lot's of other boolean operators that you can use to filter your data.
  - `!=` means not equal to
  - `>` means greater than
  - `<` means less than
  - `>=` means greater than or equal to
  - `<=` means less than or equal to
- We can use all these to filter our data based on a condition (as well as to do other things, but we'll get to that later)

FILTERING - PRACTICE

- I know you've seen a lot so far, but I'm going to get you to practice some of the things we've covered so far.
- In a cell in your notebook, use `enumerate` to print the **position and value** of each column in your `df.columns`
- Then use the `loc` method to select the first 5 rows of your dataframe, but only the columns that contain the `Participant Name`, `Title of Thing`, and `Genre` of the movie.
- See if you can use `iloc` to do the same thing.
- Then use the `loc` method to select the rows where the `ImdB Score` is greater than 80.

```
df.loc[df["col_name"] boolean_operator value]
```

DID YOU NOTICE?

- While `slicing strs` and `lists` is *not* inclusive of the `stop_value`
- `loc[] slicing DataFrames` *is* inclusive of the `stop_value`.
- `iloc[] slicing DataFrames` is *not* inclusive of the `stop_value`
- So when you're using labels, you get everything up to and including the `stop_value`
- When you're using index positions, you get everything up to but not including the `stop_value`
- This is annoying to remember, but it's actually better in practice.

CLEANING OUR DATAFRAME

- Now that we've covered slicing, we can start to clean our dataframe.
- The first thing we might want to do is make the columns easier to work with.
- For example, in the existing columns, there are spaces, some words are capitalised, and some are not
- This makes tricky because python can't infer that 'Participant Name' and 'participant name' *mean* the same thing.
- You have to type the column name *exactly* as it appears in the dataframe.
- In your notebook, try calling a column that you know is in your dataframe, but only type in lowercase. What happens?

KEYERROR

```
KeyError: 'participant name'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[2], line 1
----> 1 movies_df['participant name']

File ~/my_projects/ULpsych_programing_club/.wsl_prec/lib/python3.11/site-packages/pandas/core/frame.py:4102,
   4100 if self.columns.nlevels > 1:
...
   3815     #  InvalidIndexError. Otherwise we fall through and re-raise
   3816     #  the TypeError.
   3817     self._check_indexing_error(key)

KeyError: 'participant name'
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

keyerror

KEYERROR

- If you get a `KeyError`, it means that the column you are trying to access does not exist in your dataframe.
- This is because Python is case sensitive, and the column name you have typed does not match the column name in your dataframe.
- This is a common error, and can be really frustrating to debug.
- The best way to avoid this error is to make sure that your column names are consistent, and easy to type.
- We can do this by using the `rename` method with some `str` methods

RENAMING COLUMNS - OPTION 1

- The first way to rename columns is to use the `rename` method.
- The `rename` method takes a dictionary as an argument, where the `keys` are the old column names, and the `values` are the new column names.
- This is a really useful method, because it enables you to rename multiple columns at once.
- So if we just wanted to rename the `Participant Name` column to `Director`, we could do this:

```
1 movies_df.rename(columns = {'Participant Name': 'Director'}, inplace = True)
2 print(movies_df.columns)
```

```
Index(['Director', 'Title of Thing', 'Genre', 'Year of Release', 'ImdB Score',
       'Rotten Tomatoes Score ', 'Gender of Lead', 'Make Me Cry?'],
      dtype='object')
```

RENAMING COLUMNS - OPTION 1

- You can also use the `.rename()` with the for loop as we did in a previous session

```
for i in movies_df.columns:
    movies_df.rename(columns={i:i.lower()}, inplace = True)
print(movies_df.columns)
```

```
Index(['director', 'title of thing', 'genre', 'year of release', 'imdb score',
       'rotten tomatoes score ', 'gender of lead', 'make me cry?'],
      dtype='object')
```

RENAMING COLUMNS - OPTION 2

- Another option is to use what's called a `list comprehension`.
- I. Love. `List. Comprehensions`.
- Basically they're list putting a `for loop` inside a `list`

```
1 movies_df.columns = [i.replace(' ', '_') for i in movies_df.columns]# isn't that awesome!!
2 print(movies_df.columns)
```

```
Index(['director', 'title_of_thing', 'genre', 'year_of_release', 'imdb_score',
       'rotten_tomatoes_score_', 'gender_of_lead', 'make_me_cry?'],
      dtype='object')
```

`list comprehensions`

- `list comprehensions` work like this

```
[do_thing(placeholder) for placeholder in collection]
```

- They can be simply called in place or we can `return` a new list to us.
- We can also call `methods` instead of `functions`

```
[placeholder.do_thing() for placeholder in collection]
```

RENAMEING WITH `list comprehensions`

- We can use also use `list comprehensions` with the `rename` method
- we just combine the `dict(zip(...))` `syntax` we've seen before
- As if we had the BFI_44 in our data set starting at column 10

```
df.rename(columns = dict(zip(df.columns[10:55], [f"bfi_{i}" for i in range(1, 45)])))
```

RENAMING RECAP

- So you can see there's multiple ways to rename columns in a dataframe
  - You can use the `rename` method with a dictionary
  - You can use a `for loop` with the `rename` method
  - You can use a `list comprehension` to rename columns (my favourite way to just apply a `str` method to all the columns)
  - You can use a `list comprehension` to rename columns with a `dict(zip(...))` `syntax`

RENAMING `str methods`

- You've seen the `.upper()`, `.lower()`, and `.title()` `str` methods before
- You also just seen the `.replace()` `str` method
- but there's others like `.strip()` which removes whitespace from the beginning and end of a string (hidden spaces or newlines)
- in your notebook try using each of the renaming strategies we've covered so far to rename the columns in your dataframe
- just use a different string method each time (`.lower(), .strip(), .replace(' ', '_')`)
- And as the most important part **look up** each `str` method online

WORKING WITH OUR DATAFRAME

- Okay, so let's actually get working with the dataframe that we have. The first thing we are going to do is view our dataframe, just to remind ourselves what it looks like.
- Since the dataframe is relatively small, I am going to cheat a little bit and view the entire dataframe, rather than using the head or tail functions.

LET'S VIEW OUR DATAFRAME

	director	title_of_thing	genre	year_of_release	\
0	John Carpenter	The Thing	Horror	1982	
1		Blade Runner 2049	Sci-Fi	2017	
2	Nicolas Winding Refn	Drive	Action	2011	
3	Matthijs van Heijningen	The Thing	Horror	2011	
4	Damien Chazelle	Whiplash	Drama	2014	

	imdb_score	rotten_tomatoes_score	gender_of_lead	make_me_cry?
0	82.0	82.0	male	No
1	80.0	88.0	Male	No
2	78.0	93.0	m	No
3	62.0	34.0	Female	No
4	NaN	NaN	m	No

LET'S VIEW OUR DATAFRAME.

- For the most part, the dataframe looks alright, but it is not perfectly clean.
- We've already cleaned up the column names but:
  - There are missing values (either empty or `np.nan`),
  - there are some inconsistent values ("Male" vs "m" vs "male"),
  - classic data entry mistakes (Orgre when clearly it should be spelled Ogre),
  - and there are columns that we are unlikely to need in our analysis (e.g., Did the movie make me cry or not?).

LET'S VIEW OUR DATAFRAME.

- There also some data points that we may want to add to the dataframe (for example, average review score across ImdB and Rotten Tomatoes, length of the movie).
- Overall, the dataframe is not ready for meaningful statistical analysis.
- For the rest of this workshop, we are going to be showing you some tools that you can use to clean up the dataframe, so that it is ready for a more exhaustive analysis.

MISSING DATA

- Nearly every dataframe that you will work with is likely to contain missing data.
- There are two main types of missing *numerical* data in Python.
- A standard missing value is typically recorded in Python as `NaN`, which refers to a "not a number" value, or a "null" value.
- The first thing we can do is check whether we have standard missing values labelled 'NaN'. We can do this by using the isnull() function.

`.isnull()`

- The `isnull()` function will return our dataframe. It checks each entry our dataframe and asks the question “Is this a NaN” value?
- If there is NaN value, it returns a “True” value (in a *numerical column*).
- We can see from the output on the next slide that we have a couple of missing values in fifth row (index 4) under the columns `ImDb` score and `Rotten Tomatoes Score`
- But if you notice the ‘director’ column also has empty valuse, but because this is not a ‘numeric’ `column`.

.isnull()

```
movies_df.isnull() #This will go through our dataframe and check
#for whether we have any missing numeric values
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tomato
0	False	False	False	False	False	False
1	False	False	False	False	False	False
2	False	False	False	False	False	False
3	False	False	False	False	False	False
4	False	False	False	False	True	True
5	False	False	False	False	False	False
6	False	False	False	False	False	False
7	False	False	False	False	False	False
8	False	False	False	False	False	False
9	False	False	False	False	False	False

`.isnull()`

- We can see from the output above that we have a couple of missing values in fifth row (index 4) under the columns `ImDb` score and `Rotten Tomatoes Score`
- In future sessions, we will show you how to work with `NaN` results (other than deleting them).
- But for now, we do not want to deal with the hassle of having NaN results,
- so we are going to `drop` any row that has a `NaN` result (where isnull() == True)

.dropna()

- We can do this by using the `dropna()` function
- which means, `drop` any row that has a null/NaN result".

```
print(movies_df.shape)
movies_df_cleanv1 = movies_df.dropna()
print(movies_df_cleanv1.shape)
```

```
(10, 8)
(9, 8)
```

`.dropna()`

- We store the resulting dataframe,without the NA results
- We call it `movies_df_cleanv1`.
- This way we still have access to the raw file within this notebook if we need it.
- For example, if we needed to go back and view values from a row we've deleted.

EMPTY BUT NOT EMPTY

- The other type of missing value is a non-standard missing value.
- A non-standard missing value usually occurs when there is a cell that **looks** "empty" to us, but from Python's perspective there is a value there.
- For example, we can see that in the second row, that from our perspective, there is an empty cell under `participant name`. However, from Python's perspective, there is a value there: an empty string `""`.

EMPTY BUT NOT EMPTY

```
1 movies_df_cleanv1.head(3)
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
0	John Carpenter	The Thing	Horror	1982	82.0	82.0
1		Blade Runner 2049	Sci-Fi	2017	80.0	88.0
2	Nicolas Winding Refn	Drive	Action	2011	78.0	93.0

`replace()`

- There are multiple ways to deal with this, but what we are going to do is use `.replace()`
- We'll `replace()` the value there with `np.nan`
- Then remove that row from the dataframe.
- Basically, we convert a non-standard missing value into a missing value.

replace()

```
#Replace cell/field that's entirely space or empty with NaN
movies_df_cleanv1.replace(r'', np.nan, regex=True, inplace=True)
## Hold on a second, what the hell is r'^\s*$'? and what the hell is a regex?
# A regex is short for "regular expression".
# Regex enables us to identify a pattern in text (think about using search function in Word)
# The 'r' indicates that we are using a 'raw string' (remember these from paths).
# The '^\s*$' is regular expression syntax to identify empty strings
print(movies_df_cleanv1.shape)
movies_df_cleanv2 = movies_df_cleanv1.dropna()
print(movies_df_cleanv2.shape)
movies_df_cleanv2.head(2)
```

```
(9, 8)
(8, 8)
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
0	John Carpenter	The Thing	Horror	1982	82.0	82.0
2	Nicolas Winding Refn	Drive	Action	2011	78.0	93.0

`replace()`

- There are more sophisticated ways of handling NaN data.
- But for now, we have a dataframe that has usable (if imperfect) data in each cell.
- Our next steps are
  - to reset the index labels
  - remove columns we don't need
  - to clean up some inconsistent entries in our DataFrame

RESETING THE INDEX

- when we `drop` rows the `index labels` of the `DataFrame` doesn't update.
- you can see this if you view the `DataFrame` after dropping
- `pandas` has a really simple `method` to manage this

```
1 movies_df_cleanv2.reset_index(inplace = True, drop = True)
2 movies_df_cleanv2.head()
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
0	John Carpenter	The Thing	Horror	1982	82.0	82.0
1	Nicolas Winding Refn	Drive	Action	2011	78.0	93.0
2	Matthijs van Heijningen	The Thing	Horror	2011	62.0	34.0

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
3	Dennis Villanueve	Arrival	Sci-Fi	2016	79.0	94.0
4	Coen Brothers	No Country for Old Men	Drama	2007	82.0	93.0

RESETING THE INDEX

- Note that as we have to specify the `inplace = True` and `drop = True` arguments
- This is becase the `.reset_index method` actually saves the 'original' index as a column
- This new column is just called `index` (which is just stupid)
- so specifying these arguments just lets us get on.

DROPPING COLUMNS

- Often, we get data sets that contain variables (`columns`) that we don't need
- As when working with Quatrics data that contains columns like 'Conscent' and 'ResponseIP_Address'
- We can use the `drop()` method

```
movies_df_cleanv2 = movies_df_cleanv2.drop("make_me_cry?", axis = 1) #dropping and assigning to new
for i in movies_df_cleanv2.columns:
    print(i)
```

```
director
title_of_thing
genre
year_of_release
imdb_score
rotten_tomatoes_score
gender_of_lead
```

`.drop()`

`df.drop('label', axis = 1)`

- As you can see the `drop() method` takes 2 `arguments`
  - The label of the thing you want to `drop`
  - The `axis` of the thing you want to `drop`
- The `axis` argument just states whether your dropping rows or columns
  - `axis = 0` relates to `rows`
  - `axis = 0` relates to `columns`

axis

axis image

MORE PRACTICE!

- In you're own notebook create a new cell
- use the `.replace() to replace the empty`strings`with`Nan`
- Drop the "make_me_cry?" column (but put the outcome in a new variable)

INCONSISTENT DATA

- Now that we have columns that we want to work with, let's take a look at our data again to see if there are any inconsistencies.

```
1 movies_df_cleanv2.head()
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
0	John Carpenter	The Thing	Horror	1982	82.0	82.0
1	Nicolas Winding Refn	Drive	Action	2011	78.0	93.0
2	Matthijs van Heijningen	The Thing	Horror	2011	62.0	34.0
3	Dennis Villanueve	Arrival	Sci-Fi	2016	79.0	94.0

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
4	Coen Brothers	No Country for Old Men	Drama	2007	82.0	93.0

INCONSISTENT DATA

- Taking a look at the ‘gender of lead’ column, we can see that there are some inconsistencies in the way that the data is entered.

```
1 movies_df_cleanv2['gender_of_lead'].value_counts()
```

```
gender_of_lead
Male      3
male      1
m         1
Female    1
fem       1
Orgre     1
Name: count, dtype: int64
```

INCONSISTENT DATA

- The `df[column name].value_counts()` method gives us a sense of what values we have in a given column
- We have 3 'Male', 1 'm', 1 'male', 1 'Orgre', 1 'Female', 1 'fem'.
- Just like in SPSS, these would be considered different categories, even though we know that
  - 'Female'
  - 'fem'
- Both mean the same thing
- We can use the `.replace()` method to fix this.

.replace()

```
1 movies_df_cleanv2['gender_of_lead'] = movies_df_cleanv2['gender_of_lead'].replace(to_replace = ["Org
2 print(movies_df_cleanv2['gender_of_lead'].value_counts())
```

```
gender_of_lead
male      3
Male      3
Female    1
fem       1
Name: count, dtype: int64
```

- The syntax for the relevant code is:
- `df[column_name].replace(to_replace = ["Old Value1"...."Old ValueN"], value = "New Value1")`
- note that *instead* of slicing, we've just put the name of the column inside `[]`
- like we were working with a `dict`

We could have also replaced the uppercase `Male`, but we will show another method you use to handle cases where inconsistent values are related to capitalisation.

`.replace()`

- First let's fix the female values
- In your own notebook, use `replace()` to normalise the values for 'male' and 'female'
- Don't forget the `.value_counts()` method will help you see what values you need to clean.

```
1 movies_df_cleanv2['gender_of_lead'] = movies_df_cleanv2['gender_of_lead'].replace(to_replace = ["Fem
```

FIXING CAPITALISATION

- Now let's fix our capitalisation issue.
- You already know that there are lots of `str methods`
- We can use the `str.lower()` method to set all the values to lower.

```
1 movies_df_cleanv2['gender_of_lead'] = movies_df_cleanv2['gender_of_lead'].str.lower()
2 movies_df_cleanv2['gender_of_lead'].value_counts()
```

```
gender_of_lead
male      6
female    2
Name: count, dtype: int64
```

FIXING CAPITALISATION

- There's a few things going on here.
  - 1. We use the `df[column_name]` syntax to select the column
  - 2. We call `.str` to treat everything inside our column as a string (if it can).
  - 3. Then we call the `.lower() str method`.
- There are other functions that we could apply instead of `.lower`.
- For example, we could use `str.upper()` to transform every value into UPPER CASE

FIXING DATA - PRACTICE

- In your own notebook, make the column title case before resetting it to lower case
- (Don't forget to inspect the results with `.head()` or `.value_counts()`
- Use the value counts function on the `'genre'` column and see what type of movie pops up the most often in our dataset.
- Let's say for the sake of our analysis, we do not need to make the distinction between a comedy and a dark comedy
- Using the replace function, change "Dark Comedy" to "Comedy" in our dataset for the `'genre'` column.
- Convert the `'genre'` column to either upper or lower case (dealers choice!)

FIXING DATA

- At this stage
  - the columns are appropriately labelled,
  - there are no missing values,
  - and values are consistent within columns.

Our dataframe is pretty small though right. And there aren't that many movies in there with a female lead, so how can we rectify that?

ADD ROWS

- It is quite easy to concatenate (remember that from the earlier sessions with strings?) now rows onto our data

```
# make a dict of our new rows
new_rows = {'director': ['Ridley Scott', 'Greta Gerwig'],
'title_of_thing': ['Alien', 'Lady Bird'],
'genre': ['Horror', 'Drama'],
"year_of_release": ['1979', '2017'],
"imdB_score": [85, 74],
"rotten_tomatoes_score": [98, 99],
"gender of lead": ["female", "female"]
}
#convert the dict to a DataFrame
df2 = pd.DataFrame(new_rows) #just call the DataFrame class and pass our dict to it
movies_df_cleanv3 = pd.concat([movies_df_cleanv2,df2], ignore_index="True")
movies_df_cleanv3.tail(3)
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
7	Coen Brothers	Fargo	Dark Comedy	1996	81.0	94.0
8	Ridley Scott	Alien	Horror	1979	NaN	98.0

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
9	Greta Gerwig	Lady Bird	Drama	2017	NaN	99.0

ADD ROWS

- It is quite easy to concatenate (remember that from the earlier sessions with strings?) now rows onto our data.

```
# make a dict of our new rows
new_rows = {'director': ['Ridley Scott', 'Greta Gerwig'],
'title_of_thing': ['Alien', 'Lady Bird'],
'genre': ['Horror', 'Drama'],
'year_of_release': ['1979', '2017'],
'imdb_score': [85, 74],
'rotten_tomatoes_score': [98, 99],
'gender_of_lead': ['female', 'female']
}
#convert the dict to a DataFrame
df2 = pd.DataFrame(new_rows) #just call the DataFrame class and pass our dict to it
movies_df_cleanv3 = pd.concat([movies_df_cleanv2, df2], ignore_index=True)
movies_df_cleanv3.tail(3)
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
7	Coen Brothers	Fargo	Dark Comedy	1996	81.0	94.0
8	Ridley Scott	Alien	Horror	1979	85.0	98.0

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
9	Greta Gerwig	Lady Bird	Drama	2017	74.0	99.0

ADD ROWS

- Although there is still a gender imbalance, we are definitely healthier on male/female actor lead.
- How did we achieve that? Well, there's a few things to note:
    1. The first thing is we created a dict object, `new_rows`, containing our column names (dict keys) and column values (dict values).
        - Because we are adding more than one value per column name, we enter the column values as a list.
        - This enables Python to create multiple rows per each column.
    2. The second thing is that we turned these dict into a dataframe, called `df2`.
        - This is because it is much more straightforward to add multiple rows to an existing dataframe if those rows are already in the shape of a dataframe.
    3. The third thing is that we set `ignore_index = True`.
        - The two dataframes, movies_df_cleanv2 and df, will have their own indices.
        - For example, the value in row 1, column 1, in movies_df_cleanv2 is "John Carpenter", whereas the same coordinates in df return "Ridley Scott".
        - To set ignore_index to true simply means to combine the two dataframes under one coordinate system, rather than keeping their own specific coordinate systems. Don't worry if this is still sounding too abstract; you do

ADD ROWS

- The code for that is up on bright space.
- Please copy it into a new cell in your notebook for this session and play around with it

ADD COLUMNS

- At the moment, our columns contain Rotten Tomatoes scores from critics.
- But what if we wanted to add in the Rotten Tomatoes scores from *fans*?
- We can do that fairly easily with pandas dataframes.

```
#first let's create a list that will be converted into a column within our dataframe
rt_fans = [92.0, 79.0, 42.0, 82.0, 86.0, 69.0, 89.0, 93.0, 94.0, 79.0]
#By the way, how in the hell does Shrek 2 only have a 69% on rotten tomatoes?
#What kind of sick joke is that?
#Now let's create a column within our dataframe, and then set its values to rt_fans
movies_df_cleanv3["rotten_tomatoes_fans_scores"] = rt_fans
```

ADD COLUMNS

- We just call the name of the `df` followed by a name for our new column in `[]`
- then we use `=` (assignment not boolean/comparrison) and the name of the collection that we want populate the columns
- `df["name"] = thing`
- could also just be empty `df["empty_col"] = ''`

ADD COLUMNS

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
0	John Carpenter	The Thing	Horror	1982	82	
1	Nicolas Winding Refn	Drive	Action	2011	78	
2	Matthijs van Heijningen	The Thing	Horror	2011	62	

ORGANISE THE COLUMNS

- Nice, we have the fans' opinion in there too for Rotten Tomatoes.
- But ideally, we would want it next to the Rotten Tomatoes Critics scores.
- Let's reorder our dataframe.

```
movies_df_cleanv3.reindex([
  "director", "title_of_thing",
  "genre", "year_of_release",
  "imdB_score", "rotten_tomatoes_score",
  "rotten_tomatoes_fans_scores", "gender_of_lead"], axis = 1)
movies_df_cleanv3.head()
```

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
0	John Carpenter	The Thing	Horror	1982	82.0	82.0
1	Nicolas Winding Refn	Drive	Action	2011	78.0	93.0
2	Matthijs van	The Thing	Horror	2011	62.0	34.0

	director	title_of_thing	genre	year_of_release	imdb_score	rotten_tom
	Heijningen					
3	Dennis Villanueve	Arrival	Sci-Fi	2016	79.0	94.0
4	Coen Brothers	No Country for Old Men	Drama	2007	82.0	93.0

SOME PRACTICE

- With your partner, look up the actors in the table
- Make a list called `oscar` and fill it with “Yes” or “No” for each actor
- Then add this list as a new column called “oscar_winner” to your dataframe
- Then reorganise your data so that ‘gender_of_lead’ and ’oscar_winner are the 3rd and 4th columns

PRACTICE EXERCISES

- Well done. We have covered a lot this week. Do not worry if a lot of these ideas have not sunk in. That's normal.
- This stuff is all about *doing*
- You have to practice recalling it, to get it into your fingertips, like learning an instrument
- In new notebooks, you can practice with the CSV version of the raw data thats up on brightspace.
    1. Save a version of your df that doesn't have any empty cells.
    2. Scan for inconsistent data.
    3. See if you can fix it.
    4. Drop columns you don't need, or use `iloc` to save a version of the data that doesn't have the extra columns in it.

ADDITIONAL PRACTICE EXERCISES

- Additionally, try adding more data to your DataFrame:
  - Google more movies, create dictionaries for each movie, and append them to your DataFrame.
  - Add more columns with relevant data (e.g., box office earnings, runtime).
  - Create a new DataFrame with only the movies released after the year 2000.
  - Calculate and add a new column for the average score between IMDb and Rotten Tomatoes Critics.