SESSION 2.0

loops and dicts and bears (well Pandas)

GOOD MORNING!

- It’s day 2 (but the third day) of the summer school (and today you know why we named it that way)
- We’re going to start off with a recap of yesterday

And as always it’s in the form or a Quiz!!!

RECAP QUIZ

- It’s up on the same place as usual in brightspace
- Take your time with it, chat to your partners
- If you come to a question that doesn’t make sense, then ask me about it! (I could have made an error)

THIS SESSION

- In this session we're going to recap loops a little bit
- We're going to dive a little deeper into `dicts`
- Then we're going to look at the `pandas` library
  - We're going to look at the `DataFrame` object (which is a new `class` of object)
  - We're going to talk about the `shape attribute` of the `DataFrame` object
  - We're going to look at a few `methods` of the `DataFrame` object like `head()`, `tail()` and `describe()`

LOOPS - RECAP

- We've looked at `for` loops so far.
  - These let us `iterate` over a `collection` of items (like a `list`) and do something with each item in the collection

```
1 for item in range(1, 6):
2     t_str = f"Item = {item}"
3     print(t_str)
```

```
Item = 1
Item = 2
Item = 3
Item = 4
Item = 5
```

LOOPS - RECAP

- We've learned that we can use this to, for example, `print` out the items in a `list`
- We could also use this to `sum` the items in a `list` (assuming they are `int`s or `float`s)
- Check out the `+=` operator in the code below, it's really handy

```
t_list = [1, 2, 3, 4, 5]
t_sum = 0
for item in t_list:
    t_sum += item # the += operator is the same as t_sum = t_sum + item
print(t_sum)
```

```
15
```

LOOPS - RECAP

- There are other kinds of loops in Python
  - `while` loops
  - `do while` loops
- They're not really needed for what we're doing in this course, but they're good to know about and you might see them in other code

```
1  t_sum = 0
2  while t_sum < 10: # this will keep going until t_sum is 10 or more
3      t_sum += 1 # this will keep adding 1 to t_sum
4      print(f"the current value of t_sum is {t_sum}")
```

```
the current value of t_sum is 1
the current value of t_sum is 2
the current value of t_sum is 3
the current value of t_sum is 4
the current value of t_sum is 5
the current value of t_sum is 6
the current value of t_sum is 7
the current value of t_sum is 8
the current value of t_sum is 9
the current value of t_sum is 10
```

LOOPS AND INDEXING

- Remember when we spoke about `indexing and slicing` in `lists` and `strings`?
- Theres a few ways that we can combine that with `for loops` to do interesting things
- For example, if we know that we just want to work on the last 4 items in a list:

```
t_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for item in t_list[-4:]: # notice how we're slicing and using negatice indexing here?
    print(f"item is of {type(item)}")
```

```
item is of <class 'int'>
item is of <class 'int'>
item is of <class 'int'>
item is of <class 'int'>
```

DICTS AND INDEXING

- We've seen `dicts` before, they're a way of storing `key-value` pairs
- We can `index` into a `dict` using the `key` to get the `value`
- We can also `index` into a `dict` using the `key` to `set` the `value`

```
1 t_dict = {"name": "John", "age": 30, "city": "New York"}
2 print(t_dict["name"])
3 t_dict["name"] = "Jane"
4 print(t_dict["name"])
```

```
John
Jane
```

DICTS AND INDEXING

- In a sense, we can think of a `dict` as 2 `lists` that are `zipped` together
- The `keys` are one `list` and the `values` are another `list`
- We can get the `keys` and `values` of a `dict` using the `keys()` and `values()` `methods` of the `dict` object

```
1  t_dict = {"name": "John", "age": 30, "city": "New York"}
2  print(t_dict.keys())
3  print(t_dict.values())
```

```
dict_keys(['name', 'age', 'city'])
dict_values(['John', 30, 'New York'])
```

DICTS AND INDEXING AND LOOPS

- which means there are a few ways we can use `for loops` with `dicts`
- We can loop over the `keys()` of a `dict` and get the `values()` of the `dict` using the `key`
- In essence the `keys()` and `values()` `methods` of a `dict` return `lists` that we can loop over

```
1 for key in t_dict.keys():
2     print(f"key is {key} and value is {t_dict[key]}")
```

```
key is name and value is John
key is age and value is 30
key is city and value is New York
```

DICTS AND INDEXING AND LOOPS

- We can also loop over the `items()` of a `dict` and get the `key` and `value` at the same time
- notice in the cell below that we're using the .items() `method`
- and we're using two `variables` in the `for loop` to `unpack` the `key` and `value` from the `dict`
- these are just `placeholders` for the `key` and `value` in the `dict`, but we've given them names that make sense
- we could have called them `a` and `b` if we wanted to, but that would be confusing (sometimes you'll see k, and v or, i and j)

```
1 for key, value in t_dict.items():
2     print(f"key is {key} and value is {value}")
```

```
key is name and value is John
key is age and value is 30
key is city and value is New York
```

DICTS AND ZIPPING

- in fact, thinking about `dicts` as linked `lists` is a good way to think about them
- in python we call this `zipping` two `lists` together (like a `zipper` on a jacket)
- we can `zip` two `lists` together using the `zip() function` which takes two `lists` as `arguments` and returns a `list` of `tuples` (a `tuple` is like a list, but you can't change it once you've made it)

```
1 t_list1 = [1, 2, 3, 4, 5] # temporary list 1
2 t_list2 = ["a", "b", "c", "d", "e"] # temporary list 2
3 #zip the two lists into a dict
4 t_dict = dict(zip(t_list1, t_list2))
5 print(t_dict)
```

```
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
```

DICTS AND ZIPPING

```
#zip the two lists into a dict
t_dict = dict(zip(t_list1, t_list2)) # using the zip() function to zip the two lists together
# inside the dict() function to make a dict from the zipped lists
print(t_dict)
```

```
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
```

DICTS, LISTS, INDEXING AND ZIPPING - YOUR TURN

- In a new notebook (called 'day 2.ipynb'), create two lists
  - one with the `keys` of a `dict` you want to make
  - one with the `values` of a `dict` you want to make
- then `zip` the two lists together into a `dict`
- then use a `for loop` to `print` out the `key` and `value` of the `dict`

QUICK RECAP

- you know know about
  - `strings`,
  - `ints`,
  - `floats`,
  - `lists`,
  - `dicts`,
  - `for loops`,
  - `functions`
  - `methods`
  - `attributes`
- you're not an *expert* with any of them, but you know enough to be dangerous

QUICK RECAP

- There are a couple more basics
  - like `booleans` (True and False)
  - `if/elif/else` statements
- but these will make more sense to you when you start working in context, with actual data
- so now we're going to dive into `pandas` and start working with some real data

PANDAS

- Pandas is the main `module` used for handling SPSS like data.
- It's really powerful and it has a lot of built-in functionality that allows us to import, clean and process data, to run simple analysis, and to produce graphs and charts.
- It can also be extended with other modules to make working in python even more powerful (and fun) but right now we're going to focus on the `DataFrame` because this is the object that will actually contain our data.

THE PANDAS DATATAFRAME

- The `DataFrame` is the main object `type` in the `pandas` library
- It's a bit like a `list` of `dicts` or a `dict` of `lists`
- It's a way of storing `tabular data` in python
- It's a bit like a `spreadsheet` in excel or a `data view` in SPSS

	participant name	band	instrument	song writer	original member
0	'John Lennon'	'Beatles'	'Guitar'	'yes'	'yes'
1	'Morris Day'	'The Time'	'Vocals'	'no'	'yes'
2	'Robert Trujillo'	'metalica'	'Bass'	'no'	'no'

THE PANDAS DATAFRAME

- The above is a simple example of a `DataFrame`,
- as a social science researcher you'll be `reading` your data into a `DataFrame` from files rather than creating them from scratch.
- In the cell below we're going to demonstrate how you might make a `DataFrame` from some `lists`, but you can also make them from `dicts`, and other types of objects as well (The code is on brightspace for you to use in your own notebooks)

THE PANDAS DATAFRAME

```
import pandas as pd #importaing pandas with the nickname pd
# initialise some list objects that contains our data, this could also be a list of lists, or a dict
names = ['John Lennon', 'Morris Day', 'Robert Rtujillo', 'Prince', 'Pete Best', 'Frank Zappa']
bands = ['Beatles', 'The Time', 'metalica', 'The Time', 'Beatles', 'The Mothers']
instruments = ['Guitar', 'Vocals', 'Bass', 'Multi', 'Drums', 'Multi']
writer = ['yes', 'no', 'no', 'yes', 'no', 'yes']
orig = ['yes', 'yes', 'no', 'yes', 'yes', 'yes']
bands_df = pd.DataFrame(# opening brackets but moving to new line for readability, note the uppercas
    list(zip(names,bands, instruments, writer, orig )), #first argument note the comma at the end of
    columns = ['Participant name', 'band', 'instrument', 'song writer', 'original member']# second a
    )# closing the first pair of brackets to complete the function call
```

A NOTE ON `constructors`

- We've actually shown you a couple of examples of `constructors` before
- `int()`, `float()`, `str()`, `list()`, `dict()`, `pl.Path()` and `pd.DataFrame()` are all `constructors`
- a `constructor` is just a special type of `function` that creates an `object` of a certain `type`.
- Take the name of the `type`, put `()` after it, and you pass in the `arguments` that you want to use to create the `object`.

ANOTHER NOTE ON

`constructors`

- What you might note from `pl.Path()` and `pd.DataFrame()` is that both of these `constructor functions` use uppercase letters.
- Most `constructors` do, with the exception of the python basic classes like `list()`, `int()`, `dict()` etc,
- Generally speaking a `constructor` will use uppercase letters at the start of each word in the name.

```
my_data = pd.DataFrame(...) # this is a constructor

data_path = pl.Path(...) # this is a constructor
```

IMPORTING TO PANDAS

`DataFrame`

- As we said above, it's useful to know how to make a `DataFrame` by hand.
- However, you will be `reading` data into a `DataFrame` much more often,
- For example reading in a `.csv` or a `.xlsx` file from qualtrics,
- and pandas has a number of `pd.read...()` `functions` to allow you to do this easily.

IMPORTING TO PANDAS

`DataFrame`

- The `pd.read_csv()` `function` is the most common way to read in a `.csv` file
- The `pd.read_excel()` `function` is the most common way to read in a `.xlsx` file
- The `pd.read_spss()` `function` is the most common way to read in a `.sav` file
- Although `pd.read_spss()` requrires the `pyreadstat` `module` to be installed (because SPSS is closed source and hard to work with)

IMPORTING TO PANDAS

`DataFrame`

- The `pd.read_csv()` `function` is really powerful and has a lot of `arguments` that you can use to `customise` how your data is read in
- For example, you can specify the `delimiter` that separates the columns in your data, you can specify the `header` row, you can specify the `index` column, and you can specify the `column` names
- But, it can also be called really simply, with just the `file path` as an `argument`

IMPORTING TO PANDAS

DataFrame

```
import pathlib as pl # importing the pathlib module with the nickname pl
data_path = pl.Path(r".\data\bands_df.csv") # creating a path object to the location of your data
bands_df = pd.read_csv(data_path) # reading in the data from the path object
bands_df.head(3) # showing the first 5 rows of the data
```

IMPORTING YOUR DATA INTO A PANDAS

DataFrame

- The `read_excel()` `function` is very similar to the `read_csv()` `function`
- It has a lot of the same `arguments` and can be called in the same way

```
#we've already imported pandas as pd and pathlib as pl so no need to do it again
xl_path =  # creating a path object to the location of your data
bands_df_xl = pd.read_excel(pl.Path(r".\data\bands_df.xlsx")) # reading in the data from the path ob
bands_df_xl.head(3) # showing the first 5 rows of the data
```

IMPORTING YOUR DATA INTO A PANDAS

`DataFrame`

- OK it's time for you to try this out for yourself
- In your `day 2` notebook, create a `DataFrame` from a `.csv` file
- the file is called `bands_df.csv` is up on brightspace in the data folder
- Download it, move it into a folder called `data` in the same folder as your notebook,
- then use the `pd.read_csv()` `function` to read it into a `DataFrame`

IMPORTING YOUR DATA INTO A PANDAS

`DataFrame`

- There's lot's of ways to import data, you can even import multiple data files at once
- But for now, we're going to stick with the basics and we're going to start looking at some of the `attributes` and `methods` of the `DataFrame` object
- Don't worry, you'll be practicing other imports later on.

THE ANATOMY OF A `DataFrame`

- One of things thats great about `DataFrames` is that they have a very simple structure which allows us to work with them really easily.
- Too start out with we're going to look at
  - the `head()`,
  - the `tail()`,
  - the `shape`,
  - the `columns`
  - and the `index`.

THE HEAD() METHOD

- The `head()` `method` of a `DataFrame` object is a really useful way to see the first few rows of your data
- It's really useful for checking that you've read your data in correctly
- By default it shows the first 5 rows of your data, but you can pass in a number to show more or fewer rows
- There is also a `tail()` `method` that shows the last few rows of your data

```
1 bands_df.head() # this will show the first 5 rows of the DataFrame
```

	Participant name	band	instrument	song writer	original member
0	John Lennon	Beatles	Guitar	yes	yes
1	Morris Day	The Time	Vocals	no	yes
2	Robert Rtujillo	metalica	Bass	no	no
3	Prince	The Time	Multi	yes	yes
4	Pete Best	Beatles	Drums	no	yes

THE HEAD() METHOD

The *keen-eyed* amongst you might notice that `.head()` prints out the first **5** rows of the dataset, so poor Frank Zappa doesn't get included. **5** is the default value for `.head()`, but we can change that by just putting an `int` between the brackets.

```
1 bands_df.head(6) # this will show the first 5 rows of the DataFrame
```

	Participant name	band	instrument	song writer	original member
0	John Lennon	Beatles	Guitar	yes	yes
1	Morris Day	The Time	Vocals	no	yes
2	Robert Rtujillo	metalica	Bass	no	no
3	Prince	The Time	Multi	yes	yes
4	Pete Best	Beatles	Drums	no	yes
5	Frank Zappa	The Mothers	Multi	yes	yes

THE `.tail()` METHOD

- We're pretty sure you can guess what `.tail()` does... it displays the last *n* rows of a `DataFrame`.

	Participant name	band	instrument	song writer	original member
1	Morris Day	The Time	Vocals	no	yes
2	Robert Rtujillo	metalica	Bass	no	no
3	Prince	The Time	Multi	yes	yes
4	Pete Best	Beatles	Drums	no	yes
5	Frank Zappa	The Mothers	Multi	yes	yes

- There's Frank!
- You can also specify the number of rows with an int between the brackets.

MORE IMPORTING PRACTICE

- In your `day 2` notebook, create a `DataFrame` from the `movies_df.xlsx` file
- The file is up on brightspace in the data folder
- Download it, move it into a folder called `data` in the same folder as your notebook,
- then use the `pd.read_excel()` `function` to read it into a `DataFrame`
- then use the `head()` and `tail()` `methods` to check that you've read it in correctly

ATTRIBUTES

- You already know that `attributes` are like the `properties` of an `object`
- They're like `variables` that are attached to the `object`
- They can be `called` in the same way that you would `call` a `variable`
- We're going to look at the `shape`, `columns`, and `index` `attributes` of the `DataFrame` object

df.shape

- The shape of a df is the height' and the 'width' of the dataset
- which just means the number of rows and columns respectively.
- If we take a look at the shape of bands_df

```
print(bands_df.shape)#note the lack of brackets after 'shape'!
# this will return a tuple with the number of rows and columns in t
```

```
(6, 5)
```

`df.shape`

- We get a pair of values, 6 and 5.
- The first value (6) is the number of rows, which means we have 6 musicians in the dataset.
- The second value (5) is the number of columns in our dataset.
- `shape` `returns` a `list-like` object called `tuple`, which can be indexed like a `list`

```
1 print(f'A total of {bands_df.shape[0]} individuals took part in our
2 # I hope you see why this is useful to you in your life.
```

```
A total of 6 individuals took part in our research
```

`df.shape`

- In the your notebook call `.shape` on the csv `df`.
- (you might need to `print` it out if you're doing anything else in the same cell)
- This will tell you how many columns there are in total, and how many rows we have in the raw data.

```
1 # save the number of participants (using the indexing syntax) as th
2 n = bands_df.shape[0] # you use the movies_df variable name here
```

df.columns

- The `head()` and `tail()` methods and the `shape` attribute give you a highlevel view of you data.
- But when we want to start editing our data to get it ready for analysis we want to be able to see what columns are called and where they are in the dataset.
- Pandas handles this in a really great way with the `.columns attribute`.

```
1 for i in bands_df.columns: #using a for loop to iterate over each column name in the df columns
2     print(i)
```

```
Participant name
band
instrument
song writer
original member
```

`df.columns`

- In short, `.columns` is a `list` of the columns, in the order in which they appear in the `df`.
- It's just a `list` of `strings` (99% of the time, unless someone has made some real errors when making the data)
- and so you already know how to do a lot of things with `.columns`.

df.columns

- For example:
- We can `index` it like any other list

```
1 print(bands_df.columns[0]) # this will return the first column name in the bands data set
```

```
Participant name
```

df.columns

- We can see that the first column in the bands `df` is 'Participant name' (don't forget that python is 0-indexed).
- And you already know that if we want to find out the `index` of each column in our df, we can use a `for-loop` with the `enumerate()` function (like we did in previous examples).
- In your notebook, print out the `index` and `column name` for each column in the `df` using a `for-loop` and the `enumerate()` function.

```
1 for i, c in enumerate(bands_df.columns): #using the enumerate function to get the (i)ndex and the (c
2     print(f'{i} = {c}')
```

```
0 = Participant name
1 = band
2 = instrument
3 = song writer
4 = original member
```

df.columns

- Columns are really easy to work with in pandas, they're just `strings` in a `list`
- So if you wanted to make all the column names 'lowercase' you could do that with a `for-loop` and the `lower()` `method` of a `string`

```
print(bands_df.columns) # printing out the columns of the bands data set
for col in bands_df.columns: # using a for loop to iterate over each column name in the df columns
    bands_df.rename(columns={col: col.lower()}, inplace=True) # using the rename() method to change
print(bands_df.columns) # printing out the columns of the bands data set
```

```
Index(['Participant name', 'band', 'instrument', 'song writer',
       'original member'],
      dtype='object')
Index(['participant name', 'band', 'instrument', 'song writer',
       'original member'],
      dtype='object')
```

`.rename()` METHOD

- The `.rename()` `method` is a handy one to know, especially starting out, but it's not the only way to change column names in a `DataFrame`
- Note that when we call the `.rename()` `method` we're passing in a `dict` with the `old` column name as the `key` and the `new` column name as the `value` (and you know how you can `zip` two `lists` together to make a `dict` right?)
- We're also passing in the `inplace=True` `argument` which tells the `method` to change the `DataFrame` in place, rather than returning a new `DataFrame` with the changes made

```
bands_df.rename(columns=dict(zip(bands_df.columns[0:5]: ['id',
'age', 'gender', 'course'])), inplace=True)
```

df.columns

- In your notebook, make all the column names in the `movies_df` uppercase using a `for-loop` and the `upper()` `method` of a `string`
- the `.rename()` `method` is a handy one to know, especially starting out, but it's not the only way to change column names in a `DataFrame`
- then print out the `columns` to check that you've done it correctly

```
1 ```{python}
2
3 print(movies_df.columns) # printing out the columns of the movies data set
4 for col in movies_df.columns: # using a for loop to iterate over each column name in the df columns
5     movies_df.rename(columns={col: col.upper()}, inplace=True) # using the rename() method to change
6 print(movies_df.columns) # printing out the columns of the movies data set
7 ```
```

df.columns

- Excellent, you now know how to import your data, to get a size of your data, how to view the top and bottom of your dataset, and how to call on the columns of your dataset.
- You can use the name of the column and the same `syntax we used with dicts to just view a single column of your dataset.
- For example, if you wanted to see the `band` column of the `bands_df` you could do this:

```
1 bands_df['band'].head() # calling the 'band' column of the bands data set
```

```
0     Beatles
1    The Time
2    metalica
3    The Time
4     Beatles
Name: band, dtype: object
```

df.columns

- You can also view a `slice` of the `DataFrame` by using the `column` names in the same way you would use `keys` in a `dict`
- For example, if you wanted to see the `band` and `instrument` columns of the `bands_df` you could do this:

```
1 bands_df[['band', 'instrument']].head() # calling the 'band' and 'instrument' columns of the bands d
2 #just note that we've used double square brackets here, this is because we're passing a list of colu
```

	band	instrument
0	Beatles	Guitar
1	The Time	Vocals
2	metalica	Bass
3	The Time	Multi
4	Beatles	Drums

`df.columns`

- In the next session we're going to talk more about cleaning and preparing your data for analysis
- We'll talk a lot more about viewing your data, and about how to work with it in a way that makes sense to you
- But for now, we're going to look at the `index` of the `DataFrame` object

`df.index`

- You've already seen the word `index` when it comes to `lists` and `strings` you know that it means an items position within an object.
- When it comes to `DataFrames` the term `index` is a little different
- Used alone, it refers to the `row` of a particular participant (observation) in your data set.
- In other words it refers to a row's vertical position (reading from top to bottom) in your `df`.

`df.index`

- We can see that John Lennon is as `index` 0 and that Frank Zappa is at `index` 5 (or -1 because it's the last observation in our dataset).

```
1 bands_df
```

	participant name	band	instrument	song writer	original member
0	John Lennon	Beatles	Guitar	yes	yes
1	Morris Day	The Time	Vocals	no	yes
2	Robert Rtujillo	metalica	Bass	no	no
3	Prince	The Time	Multi	yes	yes
4	Pete Best	Beatles	Drums	no	yes
5	Frank Zappa	The Mothers	Multi	yes	yes

df.index

- There are things we can do with the `index`, like set a different column (like maybe 'Participant name') to be the `index`
- Using the `set_index()` `method` of the `DataFrame` object

```
1 # setting the 'Participant name' column as the index of the bands data set
2 new_bands_df = bands_df.copy() # making a copy of the bands data set
3 new_bands_df.set_index('participant name', inplace=True)
4 new_bands_df.head()
```

	band	instrument	song writer	original member
participant name				
John Lennon	Beatles	Guitar	yes	yes
Morris Day	The Time	Vocals	no	yes
Robert Rtujillo	metalica	Bass	no	no
Prince	The Time	Multi	yes	yes
Pete Best	Beatles	Drums	no	yes

df.index

- But this is only useful for very spific types of datasets, which you might work on in future but it's not the norm with the SPSS-like datasets that we normally work with.
- The `.index attribute` of a `DataFrame` isn't a `list` like the `columns attribute`
- It's a `pandas index` object, which is a bit like a `list` but with some extra functionality that makes it really useful for working with `DataFrames`
- However, this means that if we want to select a row by it's `index` we need to use the `.loc[] attribute` of the `DataFrame` object

```
print(bands_df.loc[0]) # selecting the first row of the bands data set
print(new_bands_df.loc['John Lennon']) # selecting the row with the index 'John Lennon' in the new b
```

```
participant name    John Lennon
band                    Beatles
instrument               Guitar
song writer                 yes
original member             yes
Name: 0, dtype: object
band               Beatles
instrument          Guitar
song writer            yes
original member         yes
Name: John Lennon, dtype: object
```

`df.index` - PRACTICE

- That's a lot in this session, so before you go, I want you to do a little practice

1. Create a new notebook called 'data_practice_1.ipynb'
2. Download the `data_practice_.csv` file from brightspace, and put it in the `data`
3. Using `pl.Path()` and `pd.read_csv()` read the data into a `DataFrame`
4. display the head and tail
5. print out each column name, and it's index in the `.columns` list (using a `for-loop` and `enumerate()`)
6. print out the row with the index '4' and the row with the index '-1' using the `.loc[]` `method`
7. make a `.copy()` of the `df` and set 'ResponseId' as the index
8. print out the row with the index 'p_001' and the row with the index 'p_004' using the `.loc[]` `method`

Make a new code cell for each step and take your time working through it