Introduction to Python

Fondren Library

Research Data Services

FLAGS WITH WHILE LOOPS

- Sometimes it makes more sense to create a flag to signal to Python to break the while loop.
- Programs will run while the flag is set to True.

```python
prompt = "Input a message. I will repeat it until you type 'quit'"
message = ""


active = True
while active:
    message = input(prompt)

    if message = 'quit':
        active = False

    else:
        print(message)
```

NOTE: Write out

the example

after class

```python
user_input = ""
while user_input.lower() != "exit":
    user_input = input("Type 'exit' to end the loop: ")
```

a side note on while loops

- while loops make for great techniques when you want to search through data while some condition is true

- let's look at an example

```
number = 1
while number <= 5:
    print(number)
    number += 1
```

```
1
2
3
4
5
```

while loops with a flag

- while loops can react to a flag (like an on-off switch)

```
prompt = "Input a message. I will repeat it until you type 'quit'"
message = ""


active = True
while active:
    message = input(prompt)

    if message = 'quit':
        active = False

    else:
        print(message)
```

aggregating data

- the **groupby** function, like in sql and other programming languages, allows you to create summaries of data in columns

```
df.groupby(['column you want to group'])['column you want to count'].count()
```

```
df.groupby(['Borough'])['Unique Key'].count()
```

```
Borough
BRONX            10925
BROOKLYN         22247
MANHATTAN        13133
QUEENS           18623
STATEN ISLAND     3848
Unspecified        861
Name: Unique Key, dtype: int64
```

chaining

- in python, multiple operations can be chained together using the dot method

```
df.groupby(['Borough'])['Unique Key'].count().sort_values(ascending=False)
```

```
Borough
BROOKLYN         22247
QUEENS           18623
MANHATTAN        13133
BRONX            10925
STATEN ISLAND     3848
Unspecified        861
Name: Unique Key, dtype: int64
```

filtering columns

to select a column in python, we do the following:

- df[‘column1’]

to select multiple columns, we can do:

- df[[‘column1’, ‘column2’, ‘column3’]]

desired columns, or our subset, can be stored in another dataframe:

- df2 = df[[‘column1’, ‘column2’, ‘column3’]]

we can then use our column subset dataframe, df2, to perform an analysis

filtering rows

- df[df['column name'] == 'value']
- df2 = df[df['column name'] == 'value']
- df3 = df[(df['column1'] == 'value1') & (df['column2'] == 'value2')]

filtering rows - exact matching vs. fuzzy matching

- exact matching

```
df[df['Complaint Type'] == 'Noise']
df[df['Complaint Type'] == 'Noise'].count()
```

```
df[df['Complaint Type'].str.contains('Noise')]
```

- fuzzy matching

Import pandas as pd
From fuzzywuzzy import process

Data = {
‘name’ : [‘Alice, ‘Bob’, ‘Charlie’, ‘David’],
‘City’ : [‘New York’, ‘Los Angeles’, ‘Miami’, ‘Chicago’]}

Df = pd.DataFrame(data)

fuzzy matching

Data = {
‘name’ : [‘Alice, ‘Bob’, ‘Charlie’, ‘David’],
‘City’ : [‘New York’, ‘Los Angeles’, ‘Miami’, ‘Chicago’]}

Df = pd.DataFrame(data)

Exact matching

Exact_match = df[df[‘City] == [‘New York, Chicago’, ‘Chicago’]
print(“Exact Matching: “)
print(exact_match)

Output:
Exact Matching:
0 Alice New York
17 Charlie New York

objectives

- to dig deeper into pandas
- to further understand the nuances of real-world data
- to apply pandas to real-world data

more on dataframes

- let’s expand a bit more on our understanding of dataframes
- andom dataframe with some random values

```
import numpy as np
import pandas as pd
from numpy.random import randn
np.random.seed(123)
```

more on dataframes

- let's use a dataframe function to create our dataframe

```
df = pd.DataFrame(randn(5,4), ['A', 'B','C','D','E',], ['W','X','Y','Z'])
```

	W	X	Y	Z
A	-1.085631	0.997345	0.282978	-1.506295
B	-0.578600	1.651437	-2.426679	-0.428913
C	1.265936	-0.866740	-0.678886	-0.094709
D	1.491390	-0.638902	-0.443982	-0.434351
E	2.205930	2.186786	1.004054	0.386186

code and see what we get when we execute

more on dataframes

- dataframes are made up of multiple lists (or series)

```
df['W']
```

```
A   -1.085631
B   -0.578600
C    1.265936
D    1.491390
E    2.205930
Name: W, dtype: float64
```

```
df[['W','X']]
```

- to select multiple columns from our dataframe, we pass in a list of column names

adding new columns in dataframes

- we can create new columns in our dataframe as well

```
df['new'] = df['W'] + df['Y']
```

```
df
```

	W	X	Y	Z	new
A	-1.085631	0.997345	0.282978	-1.506295	-0.802652
B	-0.578600	1.651437	-2.426679	-0.428913	-3.005279
C	1.265936	-0.866740	-0.678886	-0.094709	0.587050
D	1.491390	-0.638902	-0.443982	-0.434351	1.047408
E	2.205930	2.186786	1.004054	0.386186	3.209984

removing columns in dataframes

- always mind the syntax
- **df.drop('new', axis=1, inplace = True)**
- why are these parameters necessary?
- in python, **axis=1** refers to column identification and **axis=0** refers to row identification
- **inplace = True** tells python to modify the existing dataframe (save vs. save as)

creating subsets of original dataframes

- **df2 = df[['W', 'X']]**
- our new dataframe **df2** will now only have two columns
- when would we typically subset?

dropping rows in dataframes

- what do you notice about the syntax below?

```
df.drop('E', axis=0)
```

	W	X	Y	Z	new
A	-1.085631	0.997345	0.282978	-1.506295	-0.802652
B	-0.578600	1.651437	-2.426679	-0.428913	-3.005279
C	1.265936	-0.866740	-0.678886	-0.094709	0.587050
D	1.491390	-0.638902	-0.443982	-0.434351	1.047408

selecting rows in dataframes

- many ways to do this, but the most common and straightforward way is to use **loc** and **iloc**

```
df.loc['C']
```

```
W      1.265936
X     -0.866740
Y     -0.678886
Z     -0.094709
new    0.587050
Name: C, dtype: float64
```

df.iloc[2]

```
df.iloc[2]
```

```
W      1.265936
X     -0.866740
Y     -0.678886
Z     -0.094709
new    0.587050
Name: C, dtype: float64
```

conditional/logic tests on dataframes

```
df > 0
```

	W	X	Y	Z
A	True	True	False	True
B	False	False	True	False
C	False	False	False	False
D	False	False	True	False
E	True	True	False	True

```
df[df>0]
```

	W	X	Y	Z
A	0.737369	1.490732	NaN	1.175829
B	NaN	NaN	0.907105	NaN
C	NaN	NaN	NaN	NaN
D	NaN	NaN	0.927462	NaN
E	0.002846	0.688223	NaN	0.283627

- how might we use this technique to filter?

filering dataframes

- often times, you won't want to filter the entire dataframe
- you might want to only filter based on a specific column

```
df[df['Z']<0]
```

	W	X	Y	Z
B	-1.253881	-0.637752	0.907105	-1.428681
C	-0.140069	-0.861755	-0.255619	-2.798589
D	-1.771533	-0.699877	0.927462	-0.173636

```
df[df['W']>0]
```

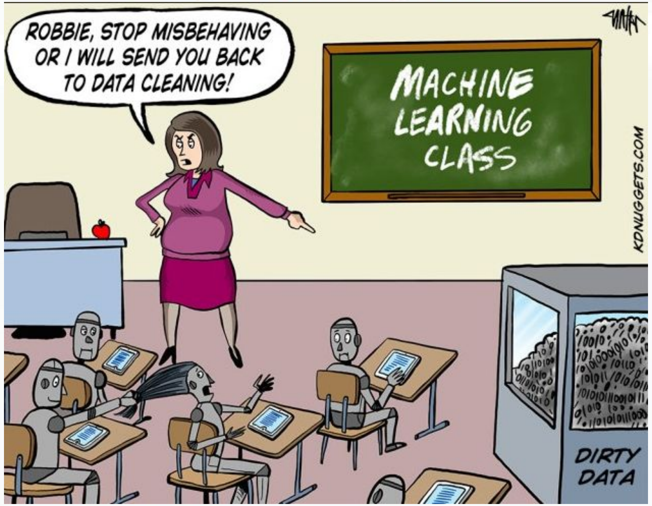
	W	X	Y	Z
A	0.737369	1.490732	-0.935834	1.175829
E	0.002846	0.688223	-0.879536	0.283627

	W	X	Y	Z
A	-1.085631	0.997345	0.282978	-1.506295
B	-0.578600	1.651437	-2.426679	-0.428913
C	1.265936	-0.866740	-0.678886	-0.094709
D	1.491390	-0.638902	-0.443982	-0.434351
E	2.205930	2.186786	1.004054	0.386186

renaming columns in dataframes

- messy data sometimes means messy columns

```
df = df.rename(columns={'Unnamed: 0': 'newName1', 'oldName2': 'newName2'})
# Or rename the existing DataFrame (rather than creating a copy)
df.rename(columns={'oldName1': 'newName1', 'oldName2': 'newName2'}, inplace=True)
```



let’s look at another real-world dataset

- vehicle gas mileage data

```
df = pd.read_csv('https://raw.githubusercontent.com/CunyLaguardiaDataAnalytics/datasets/master/mtcars.csv')
```

- understand the data first (**head**, **describe**, etc.)
- clean the dataset by naming the missing column name
- create a new dataframe that consists of a subset of the original columns (only mpg, hp, wt, and cyl)
- rename the above columns (cylinders = cyl)

objectives

- to dig deeper into pandas
- to further understand the nuances of missing data
- to apply techniques to real-world data

approaches to missing data

- missing data is a part of life
- let’s consider some options along with some pros and cons
- we can drop missing values from the dataset entirely
- we can impute missing values with the mean values of the dataset
- we can use machine learning to impute missing values

missing values

- there are a few techniques to check for missing values
- let's create a sample dataframe with some missing values to work

```
import pandas as pd
import numpy as np
```

```
df = {'A':[1,2,np.nan], 'B':[3,np.nan, np.nan]}
df = pd.DataFrame(df)
```

missing values simple example

- first let's check to see if there are any missing values

	A	B
0	1.0	3.0
1	2.0	NaN
2	NaN	NaN

```
df.isnull()
```

	A	B
0	False	False
1	False	True
2	True	True

- **null** function is one way

missing values simple example

- we can sum up total number of missing values by column

```
df.isnull().sum()
```

```
A    1
B    2
dtype: int64
```

```
df.isnull().sum().sum()
```

```
3
```

- sum up total number of missing values for the entire dataframe

missing values simple example

- let's focus on simply dropping missing values

```
df.dropna()
```

	A	B
0	1.0	3.0

- **df.dropna()** drops all missing values
- **df.dropna(axis=1)** drops na values only from columns that contain na values (if a column doesn't have any missing values, it won't be dropped)

refresher on grouping (groupby)

- remember that grouping allows you to essentially group rows together based on a certain column, and then you can perform some aggregating function on them

Company	Name	Age	Wages	Education.University	Productivity
A	`Wayne	26	50000	1	100
A	Duane	27	70000	1	120
B	William	28	70000	1	120
C	Rafael	32	60000	0	95
A	John	28	50000	0	88
B	Eric	24	70000	1	115
B	James	34	65000	1	100
C	Pablo	30	50000	0	90
C	Tammy	25	55000	1	120

Company	average Age	average Wages	Sum. Education.Unive	average Productivity
A	27	56600	2	102,6
B	28,6	68333	3	111,6
C	29	55000	1	101,6

groupby

- remember with our 311 data, we grouped complaints by borough

```
df.groupby(['Borough'])['Unique Key'].count()
```

```
Borough
BRONX            10925
BROOKLYN         22247
MANHATTAN        13133
QUEENS           18623
STATEN ISLAND     3848
Unspecified        861
Name: Unique Key, dtype: int64
```

groupby

- we could have also grouped complaints by assigned agency

```
df.groupby(['Agency'])['Unique Key'].count()
```

```
Agency
ACS          8
DCA        385
DCAS        26
DEP       5179
DFTA       226
DHS        476
DOB       4150
DOE         44
DOF       1259
DOHMH     1868
DOITT       12
DOT       8208
DPR       3065
DSNY     11262
EDC         61
HPD      11215
HRA        238
NYPD     21188
TAX          6
TLC        761
Name: Unique Key, dtype: int64
```

groupby

- remember that **groupby** works with more than just **.count()**
- you can couple **groupby** with any relevant function (using the dot or chain method)
- for example, if you're looking at salary data by job title, you can use **groupby** with **.mean()** to find average salary for a job title
- if you're looking at salary data, you can use **groupby** with **.max()** to find the highest salary for a given job title, and so on