Introduction to Pandas

- Pandas is designed for manipulating mixtures of data types in *tabular formats* (much like Excel spreadsheets).
- Pandas contains data structures and data manipulation tools designed to make *data cleaning and analysis* fast and easy in Python

Difference between Pandas and NumPy

- Pandas is designed for working with *tabular or heterogeneous data*.
- NumPy, by contrast, is best suited for working with *homogeneous numerical array data*.
- Pandas is popularly used for *data analysis and visualization*.
- NumPy is popularly used for *numerical calculations*.

This Jupyter notebook provides an overview of basic Pandas functionality. For more detailed information, here are a few good resources:

- “Python Data Science Handbook: Essential Tools for Working with Data” by Jake VanderPlas
- “Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython 2nd Edition” by Wes McKinney (the creator of pandas)
- https://realpython.com/python-data-cleaning-numpy-pandas/

Today's Outline:

- Series vs. DataFrame
- Working with Series
- Working with DataFrames
- Importing Data from .csv
- Creating Filters

Using Pandas

We generally import Pandas under the alias `pd`

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

Try

```
pd?
```

to see detailed documentation.

```
pd?
```

```
# Check Pandas Version
pd.__version__
```

```
'1.3.5'
```

Pandas Data Structures: *Series* and *DataFrame*

Series

A Pandas "Series" is a one-dimensional object, like an array.

```
mySeries1 = pd.Series([8, 3 , -6, 7])
mySeries1
```

```
0    8
1    3
2   -6
3    7
dtype: int64
```

- The left column shows the "indices". By default, these will run from 0 to (number of entries - 1).
- The right column shows the "values".

```
# We can extract just the values:
mySeries1.values
```

```
array([ 8,  3, -6,  7])
```

```
# We can also look at the indices:
mySeries1.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

- This is like `range(0, len(mySeries1))`

One useful pandas feature is that we can define custom indices:

```
mySeries2 = pd.Series([8, 3, -6, 7], index = ['c', 'a', 'b', 'xyz'])
mySeries2
```

```
c      8
a      3
b     -6
xyz    7
dtype: int64
```

Take a look at the 3rd row:

```
# We can use the index name:
mySeries2['b']
```

```
-6
```

```
# This is the same as above, but uses the index number
mySeries2[2]
```

```
-6
```

```
# We can create a Series from a python dictionary:
myDict = {'HW1': 90, 'Exam 1': 77, 'Project': 88, 'HW2': 66}

mySeries3 = pd.Series(myDict)
mySeries3
```

```
HW1        90
Exam 1     77
Project    88
HW2        66
dtype: int64
```

- Note that, by default, Pandas sorts by key/index

```
# We can provide an explicit ordering of indices:
assignments = ['HW1', 'HW2', 'HW3', 'Exam 1', 'Project']
mySeries4 = pd.Series(myDict, index = assignments)
mySeries4
```

```
HW1        90.0
HW2        66.0
HW3         NaN
Exam 1     77.0
Project    88.0
dtype: float64
```

- Note that index 'HW3' doesn't appear in myDict. "NaN" stands for "Not a Number"; it represents a null/missing value.

```
pd.isnull(mySeries4)
```

```
HW1        False
HW2        False
HW3         True
Exam 1     False
Project    False
dtype: bool
```

We can also change the indices:

```
mySeries1
```

```
0    8
1    3
2   -6
3    7
dtype: int64
```

```
mySeries1.index = ['a', 'x', 'b', 'z']
mySeries1
```

```
a    8
x    3
b   -6
z    7
dtype: int64
```

Series Indexing

```
mySeries5 = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
mySeries5
```

```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

```
# We can use the row numbers:
mySeries5[1:3]
```

```
b    1.0
c    2.0
dtype: float64
```

```
# We can also use the index labels:
mySeries5['b':'d']
```

```
b    1.0
c    2.0
d    3.0
dtype: float64
```

DataFrame

A Pandas "DataFrame" represents a table of data.

Each column in a Pandas DataFrame can contain a different type of data.

This example comes from Wes McKinney's book.

```
# Suppose we already have some data in the form of a dictionary:
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
```

```
# Convert this to a pandas DataFrame:
frame1 = pd.DataFrame(data)
frame1
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
# Look at the first 5 rows:
frame1.head()
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6

```
# Look at the last 5 rows:
frame1.tail()
```

	state	year	pop
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

While using `head()` or `tail()` function, the default number of elements printed is 5.

This value can be changed by providing an input to the function such as: `head(10)` or `tail(10)`

Checking the column data types

```
# Check the column data types using the dtypes attribute
frame1.dtypes
```

```
state     object
year       int64
pop      float64
dtype: object
```

```
# Use the shape attribute to get the number of rows and columns in your dataframe
frame1.shape
```

```
(6, 3)
```

How would you print just the number of rows in your dataframe?

```
# The info method gives the column datatypes + number of non-null values
frame1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
```

```
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   state   6 non-null      object 
 1   year    6 non-null      int64  
 2   pop     6 non-null      float64
dtypes: float64(1), int64(1), object(1)
memory usage: 272.0+ bytes
```

```
frame1
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
# We can specify the order in which columns are displayed:
pd.DataFrame(data, columns = ['year', 'state', 'pop'])
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

```
# Let's create another dataframe.
# We've added a new column (debt).
# We've also specified the index names.
frame2 = pd.DataFrame(data, columns = ['year', 'state', 'pop', 'debt'],
                      index = ['one', 'two', 'three', 'four', 'five', 'six'])
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN

```
# Assign a scalar value to all rows in a given column:
frame2['debt'] = 16.5
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

```
# Assign the values of a column via a list or array:
frame2['debt'] = np.arange(6)
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	0
two	2001	Ohio	1.7	1
three	2002	Ohio	3.6	2
four	2001	Nevada	2.4	3
five	2002	Nevada	2.9	4
six	2003	Nevada	3.2	5

```
# The following won't work because the list doesn't match the number of rows in frame2:
frame2['debt'] = np.arange(7)
frame2['debt'] = np.arange(3)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-91-edd2e13f43f9> in <module>()
      1 # The following won't work because the list doesn't match the number of rows in
----> 2 frame2['debt'] = np.arange(7)
      3 frame2['debt'] = np.arange(3)

3 frames

/usr/local/lib/python3.7/dist-packages/pandas/core/common.py in require_length_match(dat
    530     if len(data) != len(index):
    531         raise ValueError(
--> 532             "Length of values "
    533             f"({len(data)}) "
    534             "does not match length of index "
```

```
# However, if we assign a pandas Series to a DataFrame column, pandas will fill in the gaps w
val = pd.Series([-1.2, -1.5, -1.7], index = ['two', 'four', 'five'])
frame2['debt'] = val
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

```
# Add a new column:
frame2['eastern'] = frame2.state == 'Ohio'
frame2
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False
six	2003	Nevada	3.2	NaN	False

```
# Remove a column:
```

```
del frame2['eastern']
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

```
# Deleting rows
frame2.drop(['one','two'])
```

	year	state	pop	debt
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

Are rows removed from `frame2`?

```
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

```
frame3=frame2.drop(['one','two'])
frame3
```

	year	state	pop	debt
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

How to create a copy of dataframe?

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

# Convert data to a pandas DataFrame:
frame1 = pd.DataFrame(data)
frame1
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
frame1_copy = frame1
frame1_copy
```

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

```
# Remove a column from the copied dataframe:
del frame1_copy['pop']
```

```
frame1_copy
```

	state	year
0	Ohio	2000
1	Ohio	2001
2	Ohio	2002
3	Nevada	2001
4	Nevada	2002
5	Nevada	2003

```
# Take a look at the original dataframe
frame1
```

	state	year
0	Ohio	2000
1	Ohio	2001
2	Ohio	2002
3	Nevada	2001
4	Nevada	2002
5	Nevada	2003

What do you observe?

DataFrame Indexing

```
# Let's consider frame2
frame2 = pd.DataFrame(data, columns = ['year', 'state', 'pop', 'debt'],
                      index = ['one', 'two', 'three', 'four', 'five', 'six'])
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN

```
# Get a list of all columns:
frame2.columns
```

```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

five	2002	Nevada	2.9	NaN

```
# Retrieving a specific column:
frame2['year']
```

```
one      2000
two      2001
three    2002
four     2001
five     2002
six      2003
Name: year, dtype: int64
```

```
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

```
# Retrieving a specific row:
# a) by row index name, using "loc"
frame2.loc['one']
```

```
year      2000
state     Ohio
pop        1.5
debt       NaN
Name: one, dtype: object
```

```
frame2.loc['one':'four']   # Note that 'four' is included
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN

```
frame2.loc[['one', 'four']]
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
four	2001	Nevada	2.4	NaN

```
# b) by row index ID, using "iloc"
frame2.iloc[0]
```

```
year     2000
state    Ohio
pop       1.5
debt      NaN
Name: one, dtype: object
```

```
frame2.iloc[0:3]   # Note that 'four' is NOT included
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN

```
frame2.iloc[[0, 3]]
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
four	2001	Nevada	2.4	NaN

```
# Select a subset of rows and columns:
frame2.loc['one', ['year', 'pop']]
```

```
year    2000
pop      1.5
Name: one, dtype: object
```

```
frame2.loc['one', 'year':'pop']
```

```
year     2000
state    Ohio
pop       1.5
Name: one, dtype: object
```

```
frame2.loc['one':'three', 'year':'pop']
```

	year	state	pop
one	2000	Ohio	1.5
two	2001	Ohio	1.7
three	2002	Ohio	3.6

Importing Data from .csv

First, suppose we have a .csv file, named "car_financing.csv".

We need to upload file `car_financing.csv` on Colab directory so that it can be imported in the notebook.

To upload `car_financing.csv`, run the following code and upload the file from your computer/laptop.

```
# Uploading example_with_header.csv file
from google.colab import files
uploaded = files.upload()
```

Choose Files car_financing.csv

- **car_financing.csv**(text/csv) - 25802 bytes, last modified: 8/8/2022 - 100% done

```
Saving car_financing.csv to car_financing (1).csv
```

```
# Use "read_csv()"
df = pd.read_csv('car_financing.csv')
df.head()
```

Filtering Data

Filter out the data to only have data `car_type` of 'Toyota Sienna' and `interest_rate` of 0.0702.

```
# Let's first start by looking at the car_type column.
# There is a 'function' called value_counts(). It finds the number of unique rows.
df['car_type'].value_counts()
```

```
VW Golf R         144
Toyota Sienna     120
Toyota Carolla    111
Toyota Corolla     33
Name: car_type, dtype: int64
```

Filter for the `car_type`

```
# Notice that the filter produces a pandas series of True and False values
car_filter = df['car_type'] == 'Toyota Sienna'
car_filter
```

```
0       True
1       True
2       True
3       True
4       True
       ...
403    False
404    False
405    False
406    False
407    False
Name: car_type, Length: 408, dtype: bool
```

```
# Filter dataframe to get a new DataFrame of all columns, but only 'Toyota Sienna' rows.
sienna_df = df[car_filter]
sienna_df['car_type'].value_counts()
```

```
Toyota Sienna    120
Name: car_type, dtype: int64
```

Filter for the `interest_rate`

Comparison Operator	Meaning
<	less than
<=	less than or equal to
>	greater than

Comparison Operator	Meaning
>=	greater than or equal to
==	equal
!=	not equal

```
sienna_df
```

	Month	Starting Balance	Repayment	Interest Paid	Principal Paid	New Balance	term	interest_
0	1	34689.96	687.23	202.93	484.30	34205.66	60	0.
1	2	34205.66	687.23	200.10	487.13	33718.53	60	0.
2	3	33718.53	687.23	197.25	489.98	33228.55	60	0.
3	4	33228.55	687.23	194.38	492.85	32735.70	60	
4	5	32735.70	687.23	191.50	495.73	32239.97	60	0.
...	...	...	...	...	...	...	...	
115	56	3133.83	632.47	9.37	623.10	2510.73	60	0.
116	57	2510.73	632.47	7.51	624.96	1885.77	60	0.

```
df
```

	Month	Starting Balance	Repayment	Interest Paid	Principal Paid	New Balance	term	interest_

```
# Create a filter for a specific interest rate
interest_filter = df['interest_rate'] == 0.0702
interest_filter
```

```
0       True
1       True
2       True
3      False
4       True
       ...
403    False
404    False
405    False
406    False
407    False
Name: interest_rate, Length: 408, dtype: bool
```

```
# Apply the filter
specificInterest_df = df[interest_filter]
# This will be only the rows with the .0702 interest rate. All other rows were dropped.
specificInterest_df
```

	Month	Starting Balance	Repayment	Interest Paid	Principal Paid	New Balance	term	interest_
0	1	34689.96	687.23	202.93	484.30	34205.66	60	0.
1	2	34205.66	687.23	200.10	487.13	33718.53	60	0.
2	3	33718.53	687.23	197.25	489.98	33228.55	60	0.
4	5	32735.70	687.23	191.50	495.73	32239.97	60	0.
5	6	32239.97	687.23	188.60	498.63	31741.34	60	0.
6	7	31741.34	687.23	185.68	501.55	31239.79	60	0.
7	8	31239.79	687.23	182.75	504.48	30735.31	60	0.
8	9	30735.31	687.23	179.80	507.43	30227.88	60	0.
9	10	30227.88	687.23	176.83	510.40	29717.48	60	0.
10	11	29717.48	687.23	173.84	513.39	29204.09	60	0.
11	12	29204.09	687.23	170.84	516.39	28687.70	60	0.
12	13	28687.70	687.23	167.82	519.41	28168.29	60	0.
13	14	28168.29	687.23	164.78	522.45	27645.84	60	0.
14	15	27645.84	687.23	161.72	525.51	27120.33	60	0.
15	16	27120.33	687.23	158.65	528.58	26591.75	60	0.
16	17	26591.75	687.23	155.56	531.67	26060.08	60	0.
17	18	26060.08	687.23	152.45	534.78	25525.30	60	0.
18	19	25525.30	687.23	149.32	537.91	24987.39	60	0.
19	20	24987.39	687.23	146.17	541.06	24446.33	60	0.

20	21	24446.33	687.23	143.01	544.22	23902.11	60	0.
21	22	23902.11	687.23	139.82	547.41	23354.70	60	0.
22	23	23354.70	687.23	136.62	550.61	22804.09	60	0.
23	24	22804.09	687.23	133.40	553.83	22250.26	60	0.
24	25	22250.26	687.23	130.16	557.07	21693.19	60	0.
25	26	21693.19	687.23	126.90	560.33	21132.86	60	0.
26	27	21132.86	687.23	123.62	563.61	20569.25	60	0.
27	28	20569.25	687.23	120.33	566.90	20002.35	60	0.
28	29	20002.35	687.23	117.01	570.22	19432.13	60	0.

Combining Filters

In the previous sections, we created `car_filter` and `interest_filter`. We could do this all at one time.

Bitwise Logic Operator	Meaning
&	and
|	or
~	not

```
# Apply both filters to the DataFrame.
new_df = df[car_filter & interest_filter]
new_df
```

	Month	Starting Balance	Repayment	Interest Paid	Principal Paid	New Balance	term	interest_rate	car_
0	1	34689.96	687.23	202.93	484.30	34205.66	60	0.0702	T Si
1	2	34205.66	687.23	200.10	487.13	33718.53	60	0.0702	T Si
2	3	33718.53	687.23	197.25	489.98	33228.55	60	0.0702	T Si
4	5	32735.70	687.23	191.50	495.73	32239.97	60	0.0702	T Si
5	6	32239.97	687.23	188.60	498.63	31741.34	60	0.0702	T Si
6	7	31741.34	687.23	185.68	501.55	31239.79	60	0.0702	T Si
7	8	31239.79	687.23	182.75	504.48	30735.31	60	0.0702	T Si
8	9	30735.31	687.23	179.80	507.43	30227.88	60	0.0702	T Si
9	10	30227.88	687.23	176.83	510.40	29717.48	60	0.0702	T Si
10	11	29717.48	687.23	173.84	513.39	29204.09	60	0.0702	T Si
11	12	29204.09	687.23	170.84	516.39	28687.70	60	0.0702	T Si
12	13	28687.70	687.23	167.82	519.41	28168.29	60	0.0702	T Si
13	14	28168.29	687.23	164.78	522.45	27645.84	60	0.0702	T Si
14	15	27645.84	687.23	161.72	525.51	27120.33	60	0.0702	T Si
15	16	27120.33	687.23	158.65	528.58	26591.75	60	0.0702	T Si
16	17	26591.75	687.23	155.56	531.67	26060.08	60	0.0702	T Si
17	18	26060.08	687.23	152.45	534.78	25525.30	60	0.0702	T Si
18	19	25525.30	687.23	149.32	537.91	24987.39	60	0.0702	T Si
19	20	24987.39	687.23	146.17	541.06	24446.33	60	0.0702	T Si

20	21	24446.33	687.23	143.01	544.22	23902.11	60	0.0702	T Si
21	22	23902.11	687.23	139.82	547.41	23354.70	60	0.0702	T Si
22	23	23354.70	687.23	136.62	550.61	22804.09	60	0.0702	T Si
23	24	22804.09	687.23	133.40	553.83	22250.26	60	0.0702	T Si
24	25	22250.26	687.23	130.16	557.07	21693.19	60	0.0702	T Si
25	26	21693.19	687.23	126.90	560.33	21132.86	60	0.0702	T Si
26	27	21132.86	687.23	123.62	563.61	20569.25	60	0.0702	T Si
27	28	20569.25	687.23	120.33	566.90	20002.35	60	0.0702	T Si
28	29	20002.35	687.23	117.01	570.22	19432.13	60	0.0702	T Si
29	30	19432.13	687.23	113.67	573.56	18858.57	60	0.0702	T Si
30	31	18858.57	687.23	110.32	576.91	18281.66	60	0.0702	T Si
31	32	18281.66	687.23	106.94	580.29	17701.37	60	0.0702	T Si
32	33	17701.37	687.23	103.55	583.68	17117.69	60	0.0702	T Si
33	34	17117.69	687.23	100.13	587.10	16530.59	60	0.0702	T Si
34	35	16530.59	687.23	96.70	590.53	15940.06	60	0.0702	T Si
35	36	15940.06	687.23	93.24	593.99	15346.07	60	0.0702	T Si
36	37	15346.07	687.23	89.77	597.46	14748.61	60	0.0702	T Si
37	38	14748.61	687.23	86.27	600.96	14147.65	60	0.0702	T Si

0s completed at 1:10 PM