**Data Exploration**

**Practice working with library Pandas**

**What's Pandas for?**

This tool is essentially your data’s home. Through pandas, you get acquainted with your data by cleaning, transforming, and analyzing it.

For example, say you want to explore a dataset stored in a CSV on your computer. Pandas will extract the data from that CSV into a DataFrame — a table, basically — then let you do things like:

* Calculate statistics and answer questions about the data, like
* What's the average, median, max, or min of each column?
* Does column A correlate with column B?
* What does the distribution of data in column C look like?
* Clean the data by doing things like removing missing values and filtering rows or columns by some criteria
* Visualize the data with help from Matplotlib. Plot bars, lines, histograms, bubbles, and more.
* Store the cleaned, transformed data back into a CSV, other file or database

### Install and import

Pandas is an easy package to install. Open up your terminal program (for Mac users) or command line (for PC users) and install it using either of the following commands:

conda install pandas

OR

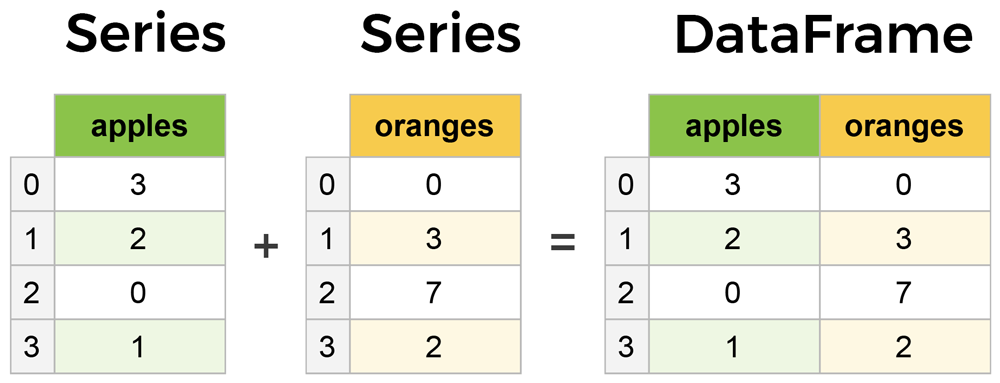
pip install pandas

Now to the basic components of pandas.

## Core components of pandas: Series and DataFrames

The primary two components of pandas are the Series and DataFrame.

A Series is essentially a column, and a DataFrame is a multi-dimensional table made up of a collection of Series.



DataFrames and Series are quite similar in that many operations that you can do with one you can do with the other, such as filling in null values and calculating the mean.

You'll see how these components work when we start working with data below.

To import pandas we usually import it with a shorter name since it's used so much:

**import** **pandas** **as** **pd**

### 1 Creating DataFrames from scratch

Creating DataFrames right in Python is good to know and quite useful when testing new methods and functions you find in the pandas docs.

There are *many* ways to create a DataFrame from scratch, but a great option is to just use a simple dict.

Let's say we have a fruit stand that sells apples and oranges. We want to have a column for each fruit and a row for each customer purchase. To organize this as a dictionary for pandas we could do something like:

**import** **pandas** **as** **pd**

data = {

'apples': [3, 2, 0, 1],

'oranges': [0, 3, 7, 2]

}

And then pass it to the pandas DataFrame constructor:

purchases = pd.DataFrame(data)

purchases

**OUT:**

|  | **apples** | **oranges** |
| --- | --- | --- |
| **0** | 3 | 0 |
| **1** | 2 | 3 |
| **2** | 0 | 7 |
| **3** | 1 | 2 |

**How did that work?**

Each *(key, value)* item in data corresponds to a *column* in the resulting DataFrame.

The **Index** of this DataFrame was given to us on creation as the numbers 0-3, but we could also create our own when we initialize the DataFrame.

Let's have customer names as our index:

purchases = pd.DataFrame(data, index=['June', 'Robert', 'Lily', 'David'])

purchases

**OUT:**

|  | **apples** | **oranges** |
| --- | --- | --- |
| **June** | 3 | 0 |
| **Robert** | 2 | 3 |
| **Lily** | 0 | 7 |
| **David** | 1 | 2 |

So now we could **loc**ate a customer's order by using their name:

purchases.loc['June']

**OUT:**

apples 3

oranges 0

Name: June, dtype: int64

There's more on locating and extracting data from the DataFrame later, but now you should be able to create a DataFrame with any random data to learn on.

Let's move on to some quick methods for creating DataFrames from various other sources.

## 2 How to read in data

It’s quite simple to load data from various file formats into a DataFrame. In the following examples we'll keep using our apples and oranges data, but this time it's coming from various files.

### Reading data from CSVs

Locate the ‘purchases.csv’ from Moodle – Week 2 code files.

With CSV files all you need is a single line to load in the data:

Ensure the correct path I used for the file: (r'''C:\Users\....... \purchases.csv''', delimiter = ",")

df = pd.read\_csv('purchases.csv')

df

**OUT:**

|  | **Unnamed: 0** | **apples** | **oranges** |
| --- | --- | --- | --- |
| **0** | June | 3 | 0 |
| **1** | Robert | 2 | 3 |
| **2** | Lily | 0 | 7 |
| **3** | David | 1 | 2 |

CSVs don't have indexes like our DataFrames, so all we need to do is just designate the index\_col when reading:

df = pd.read\_csv('purchases.csv', index\_col=0)

df

**OUT:**

|  | **apples** | **oranges** |
| --- | --- | --- |
| **June** | 3 | 0 |
| **Robert** | 2 | 3 |
| **Lily** | 0 | 7 |
| **David** | 1 | 2 |

Here we're setting the index to be column zero.

You'll find that most CSVs won't ever have an index column and so usually you don't have to worry about this step.

## 3 Most important DataFrame operations

DataFrames possess hundreds of methods and other operations that are crucial to any analysis. As a beginner, you should know the operations that perform simple transformations of your data and those that provide fundamental statistical analysis.

Let's load in the IMDB movies dataset to begin:

In [2]:

movies\_df = pd.read\_csv("IMDB-Movie-Data.csv", index\_col="Title")

We're loading this dataset from a CSV and designating the movie titles to be our index.

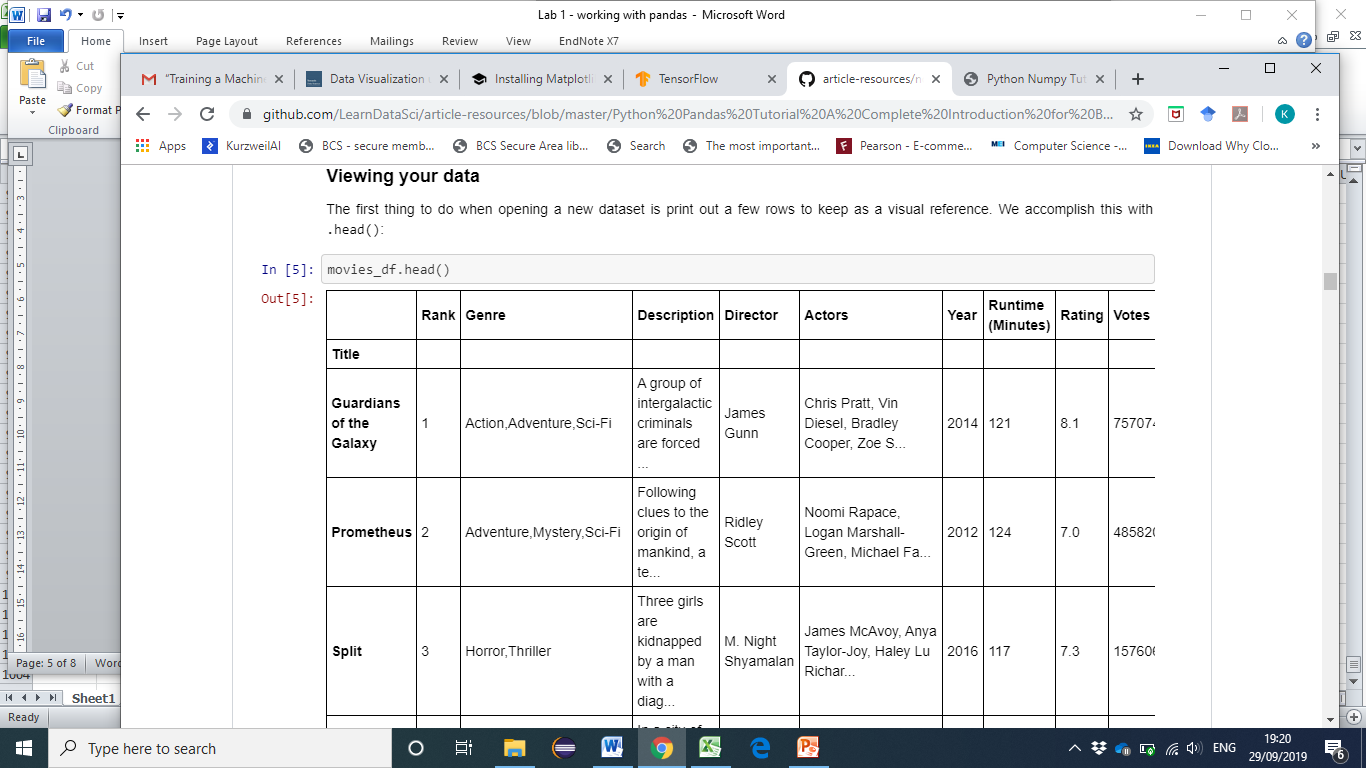
### Viewing your data

The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with .head():

In [5]:

movies\_df.head()

Out[5]:

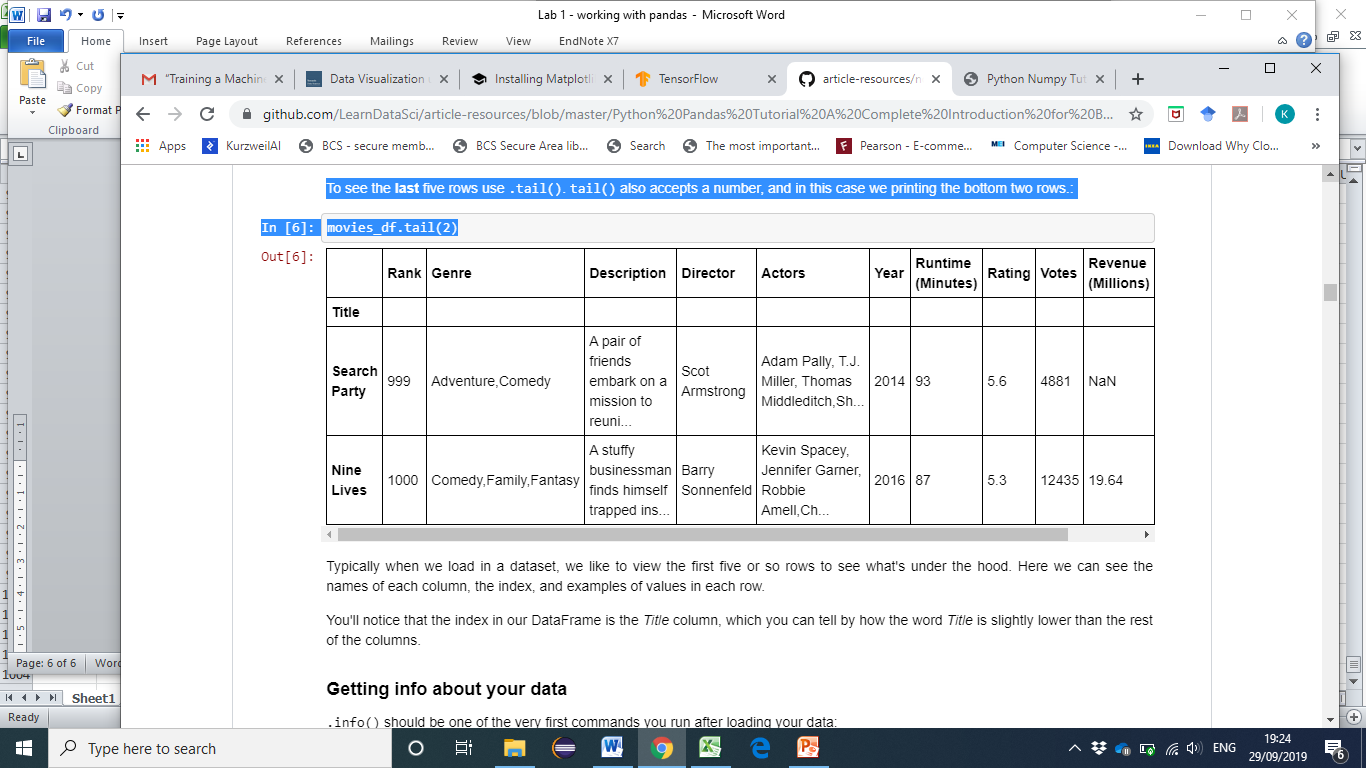


.head() outputs the **first** five rows of your DataFrame by default, but we could also pass a number as well: movies\_df.head(10) would output the top ten rows, for example.

To see the **last** five rows use .tail(). tail() also accepts a number, and in this case we printing the bottom two rows.:

In [6]:

movies\_df.tail(2)



Typically when we load in a dataset, we like to view the first five or so rows to see what's under the hood. Here we can see the names of each column, the index, and examples of values in each row.

You'll notice that the index in our DataFrame is the Title column, which you can tell by how the word Title is slightly lower than the rest of the columns.

### Getting info about your data

.info() should be one of the very first commands you run after loading your data:

movies\_df.info()

<class 'pandas.core.frame.DataFrame'>

Index: 1000 entries, Guardians of the Galaxy to Nine Lives

Data columns (total 11 columns):

Rank 1000 non-null int64

Genre 1000 non-null object

Description 1000 non-null object

Director 1000 non-null object

Actors 1000 non-null object

Year 1000 non-null int64

Runtime (Minutes) 1000 non-null int64

Rating 1000 non-null float64

Votes 1000 non-null int64

Revenue (Millions) 872 non-null float64

Metascore 936 non-null float64

dtypes: float64(3), int64(4), object(4)

memory usage: 93.8+ KB

.info() provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

Notice in our movies dataset we have some obvious missing values in the Revenue and Metascore columns. We'll look at how to handle those in a bit.

Another fast and useful attribute is .shape, which outputs just a tuple of (rows, columns):

In [4]:

movies\_df.shape

Out[4]:

(1000, 11)

Note that .shape has no parentheses and is a simple tuple of format (rows, columns). So we have **1000 rows** and **11 columns** in our movies DataFrame.

You'll be going to .shape a lot when cleaning and transforming data. For example, you might filter some rows based on some criteria and then want to know quickly how many rows were removed.

### 4 Column cleanup

Many times datasets will have verbose column names with symbols, upper and lowercase words, spaces, and typos. To make selecting data by column name easier we can spend a little time cleaning up their names.

Here's how to print the column names of our dataset:

In [86]:

movies\_df.columns

Out[86]:

Index(['Rank', 'Genre', 'Description', 'Director', 'Actors', 'Year',

'Runtime (Minutes)', 'Rating', 'Votes', 'Revenue (Millions)',

'Metascore'],

dtype='object')

Not only does .columns come in handy if you want to rename columns by allowing for simple copy and paste, it's also useful if you need to understand why you are receiving a Key Error when selecting data by column.

We can use the .rename() method to rename certain or all columns via a dict. We don't want parentheses, so let's rename those:

In [87]:

movies\_df.rename(columns={

'Runtime (Minutes)': 'Runtime',

'Revenue (Millions)': 'Revenue\_millions'

}, inplace=**True**)

movies\_df.columns

Out[87]:

Index(['Rank', 'Genre', 'Description', 'Director', 'Actors', 'Year', 'Runtime',

'Rating', 'Votes', 'Revenue\_millions', 'Metascore'],

dtype='object')

But what if we want to lowercase all names? Instead of using .rename() we could also set a list of names to the columns like so:

In [92]:

movies\_df.columns = ['rank', 'genre', 'description', 'director', 'actors', 'year', 'runtime', 'rating', 'votes', 'revenue\_millions', 'metascore']

movies\_df.columns

Out[92]:

Index(['rank', 'genre', 'description', 'director', 'actors', 'year', 'runtime',

'rating', 'votes', 'revenue\_millions', 'metascore'],

dtype='object')

But that's too much work. Instead of just renaming each column manually we can do a list comprehension:

In [93]:

movies\_df.columns = [col.lower() **for** col **in** movies\_df]

movies\_df.columns

Out[93]:

Index(['rank', 'genre', 'description', 'director', 'actors', 'year', 'runtime',

'rating', 'votes', 'revenue\_millions', 'metascore'],

dtype='object')

list (and dict) comprehensions come in handy a lot when working with pandas and data in general.

It's a good idea to lowercase, remove special characters, and replace spaces with underscores if you'll be working with a dataset for some time.

### 5 How to work with missing values (imputation)

When exploring data, you’ll most likely encounter missing or null values, which are essentially placeholders for non-existent values. Most commonly you'll see Python's None or NumPy's np.nan, each of which are handled differently in some situations.

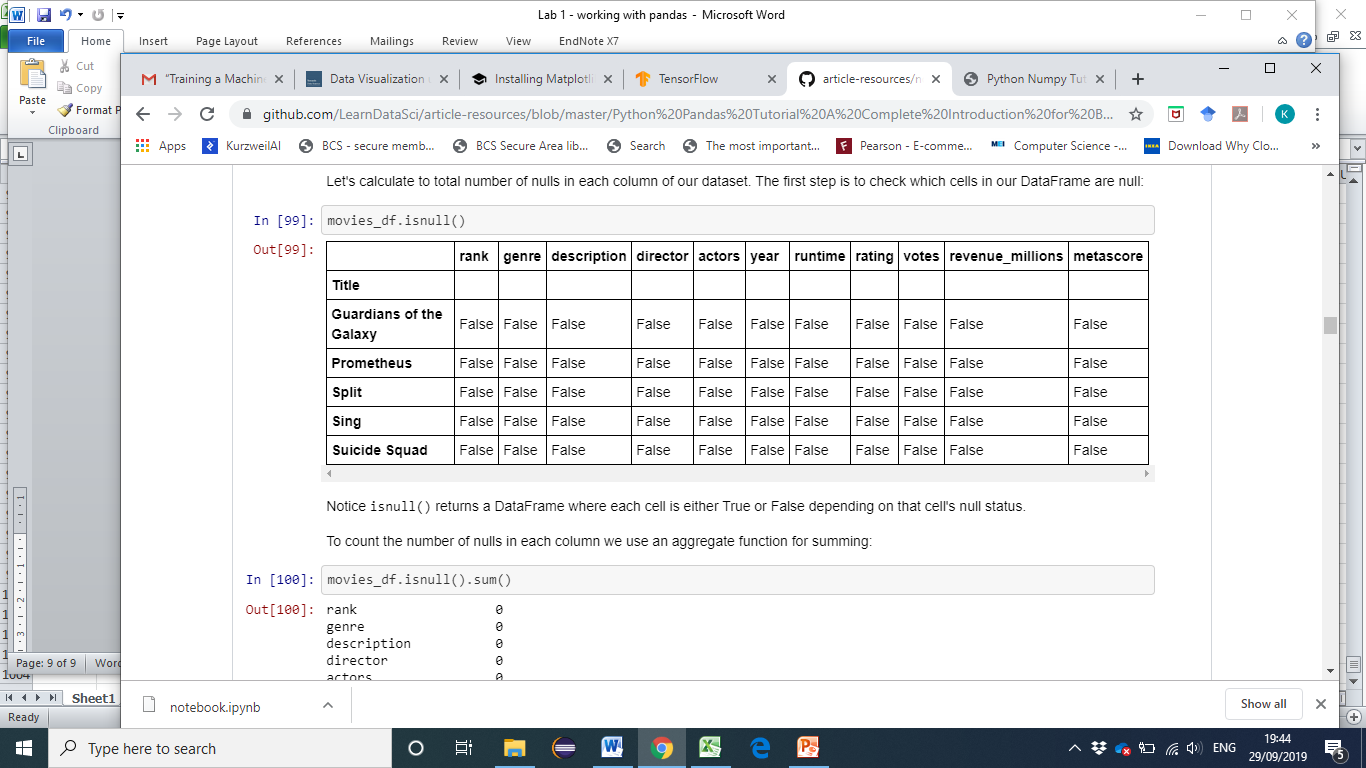
There are two options in dealing with nulls:

1. Get rid of rows or columns with nulls
2. Replace nulls with non-null values, a technique known as **imputation**

Let's calculate to total number of nulls in each column of our dataset. The first step is to check which cells in our DataFrame are null:

In [99]:

movies\_df.isnull()



Notice isnull() returns a DataFrame where each cell is either True or False depending on that cell's null status.

To count the number of nulls in each column we use an aggregate function for summing:

In [100]:

movies\_df.isnull().sum()

Out[100]:

rank 0

genre 0

description 0

director 0

actors 0

year 0

runtime 0

rating 0

votes 0

revenue\_millions 128

metascore 64

dtype: int64

.isnull() just by itself isn't very useful, and is usually used in conjunction with other methods, like sum().

We can see now that our data has **128** missing values for revenue\_millions and **64** missing values for metascore.

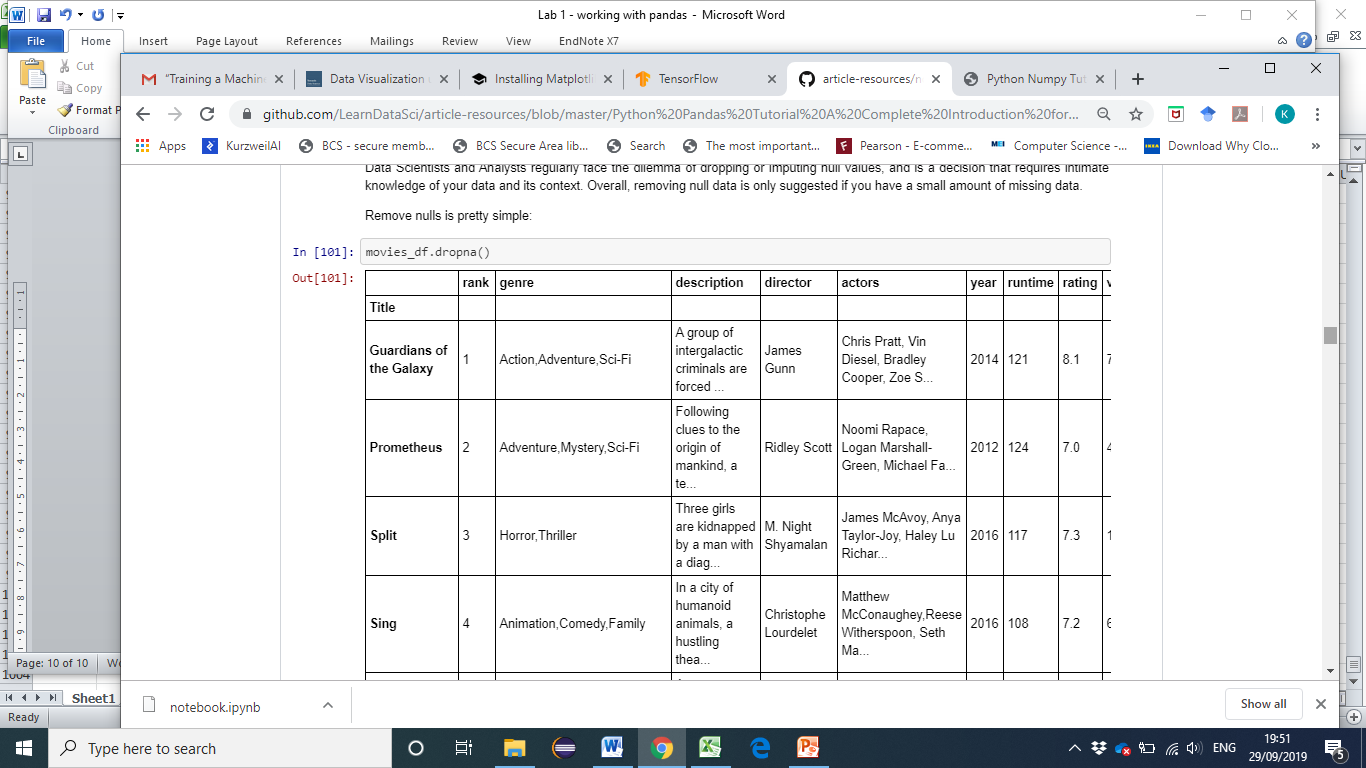
#### 6 Removing null values

Data Scientists and Analysts regularly face the dilemma of dropping or imputing null values, and is a decision that requires intimate knowledge of your data and its context. Overall, removing null data is only suggested if you have a small amount of missing data.

Remove nulls is pretty simple:

In [101]:

movies\_df.dropna()



838 rows × 11 columns

This operation will delete any **row** with at least a single null value, but it will return a new DataFrame without altering the original one. You could specify inplace=True in this method as well.

So in the case of our dataset, this operation would remove 128 rows where revenue\_millions is null and 64 rows where metascore is null. This obviously seems like a waste since there's perfectly good data in the other columns of those dropped rows. That's why we'll look at imputation next.

Other than just dropping rows, you can also drop columns with null values by setting axis=1:

In [102]:

movies\_df.dropna(axis=1)

1000 rows × 9 columns

In our dataset, this operation would drop the revenue\_millions and metascore columns.

**Intuition side note**: What's with this axis=1 parameter?

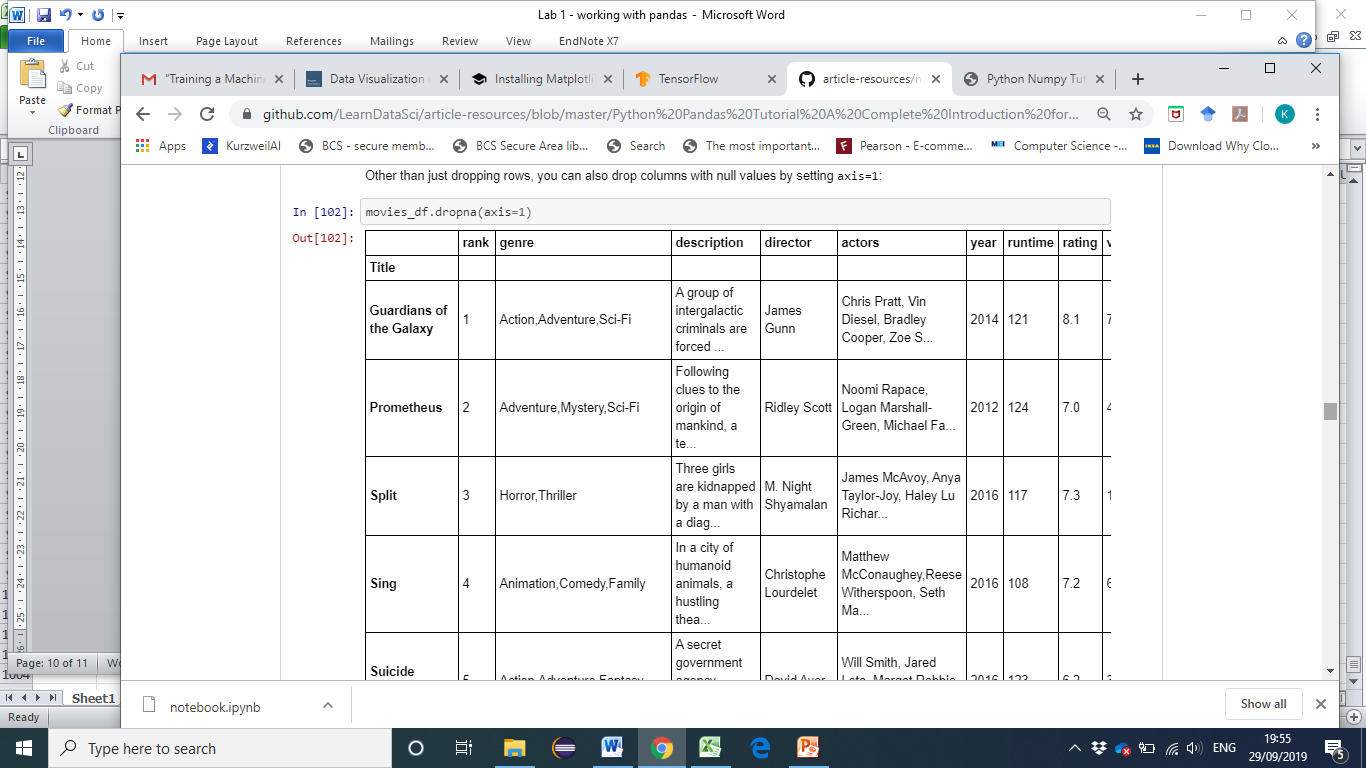
It's not immediately obvious where axis comes from and why you need it to be 1 for it to affect columns. To see why, just look at the .shape output:

In [103]:

movies\_df.shape

Out[103]:

(1000, 11)



As we learned above, this is a tuple that represents the shape of the DataFrame, i.e. 1000 rows and 11 columns. Note that the rows are at index zero of this tuple and columns are at **index one** of this tuple. This is why axis=1 affects columns. This comes from NumPy, and is a great example of why learning NumPy is worth your time.

### Imputation

Imputation is a conventional feature engineering technique used to keep valuable data that have null values.

There may be instances where dropping every row with a null value removes too big a chunk from your dataset, so instead we can impute that null with another value, usually the **mean** or the **median** of that column.

Let's look at imputing the missing values in the revenue\_millions column. First we'll extract that column into its own variable:

In [104]:

revenue = movies\_df['revenue\_millions']

Using square brackets is the general way we select columns in a DataFrame.

If you remember back to when we created DataFrames from scratch, the keys of the dict ended up as column names. Now when we select columns of a DataFrame, we use brackets just like if we were accessing a Python dictionary.

revenue now contains a Series:

In [105]:

revenue.head()

Out[105]:

Title

Guardians of the Galaxy 333.13

Prometheus 126.46

Split 138.12

Sing 270.32

Suicide Squad 325.02

Name: revenue\_millions, dtype: float64

Slightly different formatting than a DataFrame, but we still have our Title index.

We'll impute the missing values of revenue using the mean. Here's the mean value:

In [107]:

revenue\_mean = revenue.mean()

revenue\_mean

Out[107]:

82.95637614678897

With the mean, let's fill the nulls using fillna():

In [108]:

revenue.fillna(revenue\_mean, inplace=**True**)

We have now replaced all nulls in revenue with the mean of the column. Notice that by using inplace=True we have actually affected the original movies\_df:

In [114]:

movies\_df.isnull().sum()

Out[114]:

rank 0

genre 0

description 0

director 0

actors 0

year 0

runtime 0

rating 0

votes 0

revenue\_millions 0

metascore 64

dtype: int64

Imputing an entire column with the same value like this is a basic example. It would be a better idea to try a more granular imputation by Genre or Director.

For example, you would find the mean of the revenue generated in each genre individually and impute the nulls in each genre with that genre's mean.

Let's now look at more ways to examine and understand the dataset.

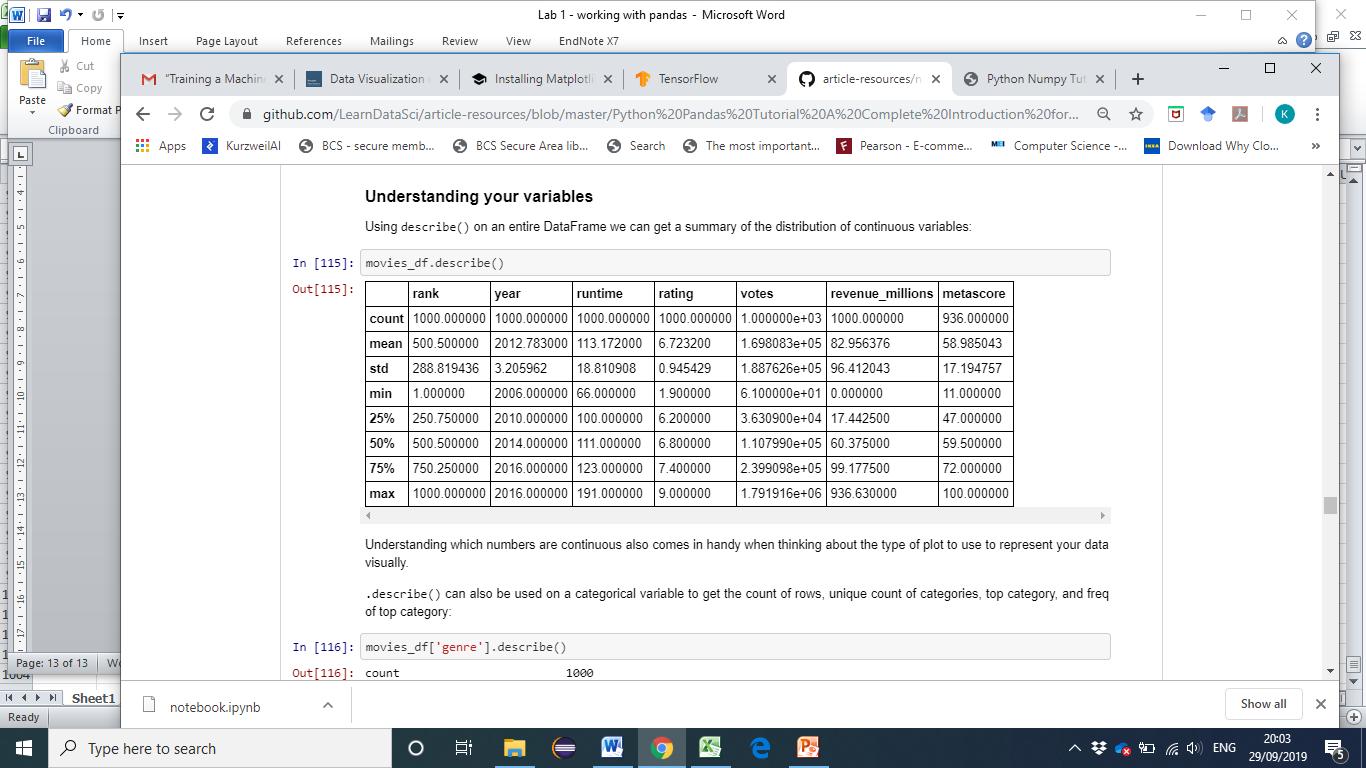
### 8 Understanding your variables

Using describe() on an entire DataFrame we can get a summary of the distribution of continuous variables:

In [115]:

movies\_df.describe()

Out[115]:



Understanding which numbers are continuous also comes in handy when thinking about the type of plot to use to represent your data visually.

.describe() can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

In [116]:

movies\_df['genre'].describe()

Out[116]:

count 1000

unique 207

top Action,Adventure,Sci-Fi

freq 50

Name: genre, dtype: object

This tells us that the genre column has 207 unique values, the top value is Action/Adventure/Sci-Fi, which shows up 50 times (freq).

.value\_counts() can tell us the frequency of all values in a column:

In [119]:

movies\_df['genre'].value\_counts().head(10)

Out[119]:

Action,Adventure,Sci-Fi 50

Drama 48

Comedy,Drama,Romance 35

Comedy 32

Drama,Romance 31

Action,Adventure,Fantasy 27

Comedy,Drama 27

Animation,Adventure,Comedy 27

Comedy,Romance 26

Crime,Drama,Thriller 24

Name: genre, dtype: int64

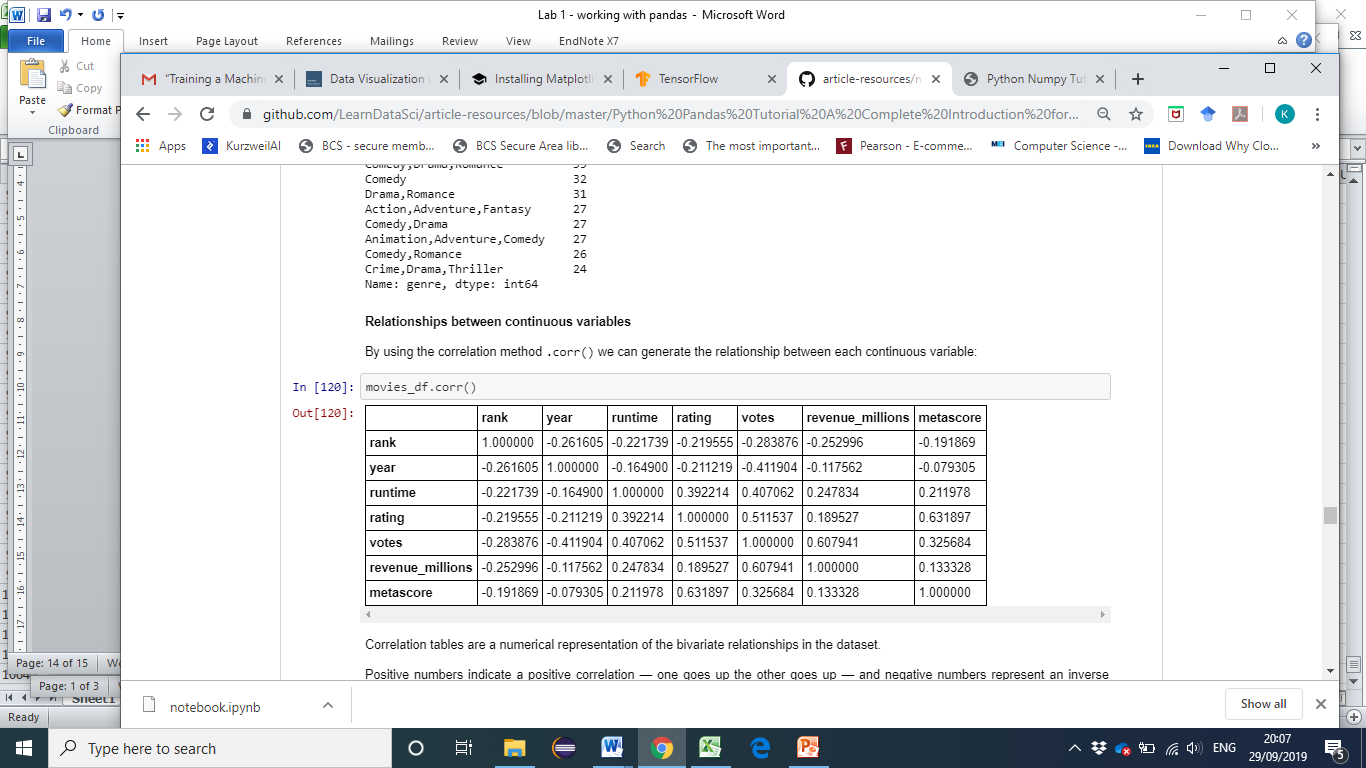
#### Relationships between continuous variables

By using the correlation method .corr() we can generate the relationship between each continuous variable:

In [120]:

movies\_df.corr()

Out[120]:



Correlation tables are a numerical representation of the bivariate relationships in the dataset.

Positive numbers indicate a positive correlation — one goes up the other goes up — and negative numbers represent an inverse correlation — one goes up the other goes down. 1.0 indicates a perfect correlation.

So looking in the first row, first column we see rank has a perfect correlation with itself, which is obvious. On the other hand, the correlation between votes and revenue\_millions is 0.6. A little more interesting.

Examining bivariate relationships comes in handy when you have an outcome or dependent variable in mind and would like to see the features most correlated to the increase or decrease of the outcome. You can visually represent bivariate relationships with scatterplots .

### 9 DataFrame slicing, selecting, extracting

Up until now we've focused on some basic summaries of our data. We've learned about simple column extraction using single brackets, and we imputed null values in a column using fillna(). Below are the other methods of slicing, selecting, and extracting you'll need to use constantly.

It's important to note that, although many methods are the same, DataFrames and Series have different attributes, so you'll need be sure to know which type you are working with or else you will receive attribute errors.

Let's look at working with columns first.

#### By column

You already saw how to extract a column using square brackets like this:

In [125]:

pandas.core.series.Series

type(genre\_col)

Out[125]:

Now we'll look at getting data by rows.

#### By rows

For rows, we have two options:

* .loc - **loc**ates by name
* .iloc- **loc**ates by numerical **i**ndex

Remember that we are still indexed by movie Title, so to use .loc we give it the Title of a movie:

In [128]:

prom = movies\_df.loc["Prometheus"]

prom

Out[128]:

rank 2

genre Adventure,Mystery,Sci-Fi

description Following clues to the origin of mankind, a te...

director Ridley Scott

actors Noomi Rapace, Logan Marshall-Green, Michael Fa...

year 2012

runtime 124

rating 7

votes 485820

revenue\_millions 126.46

metascore 65

Name: Prometheus, dtype: object

On the other hand, with iloc we give it the numerical index of Prometheus:

#### 9 Conditional selections

We’ve gone over how to select columns and rows, but what if we want to make a conditional selection?

For example, what if we want to filter our movies DataFrame to show only films directed by Ridley Scott or films with a rating greater than or equal to 8.0?

To do that, we take a column from the DataFrame and apply a Boolean condition to it. Here's an example of a Boolean condition:

In [134]:

condition = (movies\_df['director'] == "Ridley Scott")

condition.head()

Out[134]:

Title

Guardians of the Galaxy False

Prometheus True

Split False

Sing False

Suicide Squad False

Name: director, dtype: bool

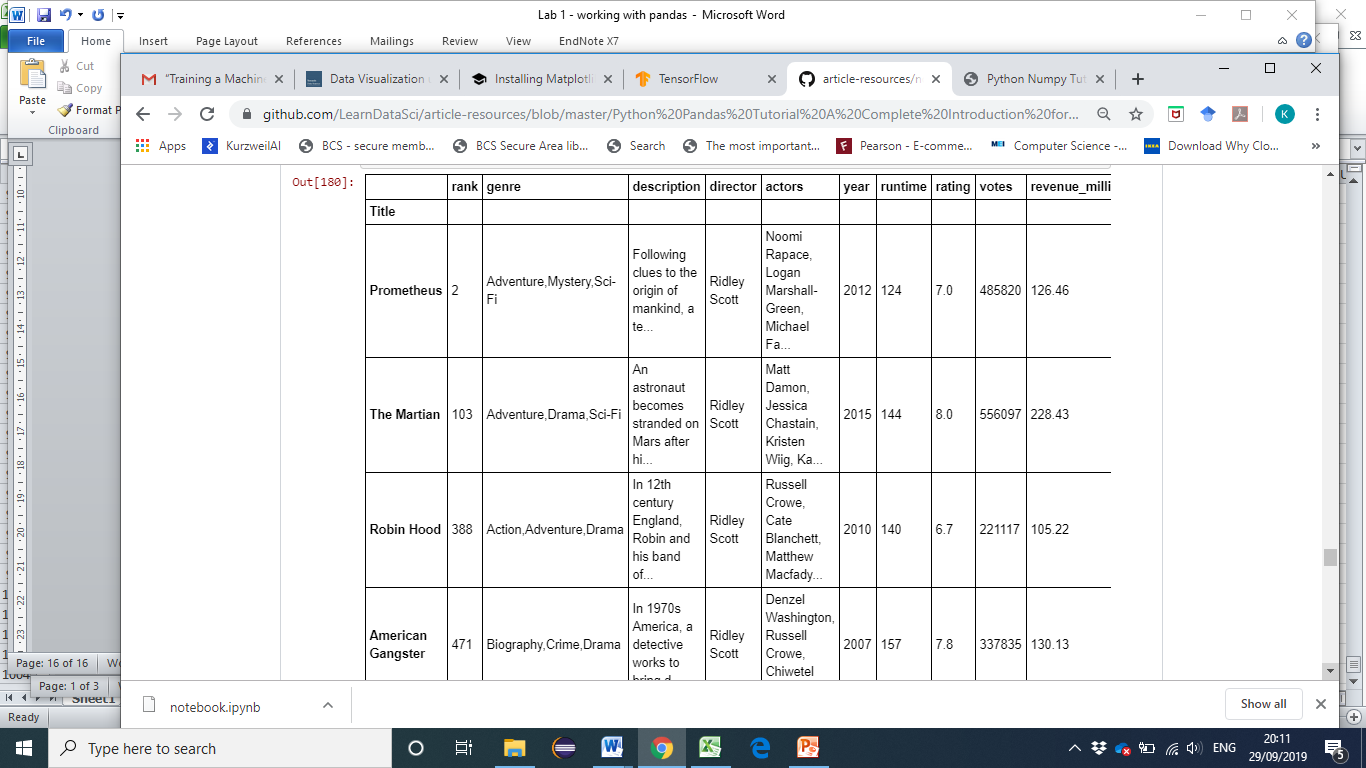
Similar to isnull(), this returns a Series of True and False values: True for films directed by Ridley Scott and False for ones not directed by him.

We want to filter out all movies not directed by Ridley Scott, in other words, we don’t want the False films. To return the rows where that condition is True we have to pass this operation into the DataFrame:

In [180]:

movies\_df[movies\_df['director'] == "Ridley Scott"].head()

Out[180]:



You can get used to looking at these conditionals by reading it like:

Select movies\_df where movies\_df director equals Ridley Scott

Let's look at conditional selections using numerical values by filtering the DataFrame by ratings:

In [137]:

movies\_df[movies\_df['rating'] >= 8.6].head(3)

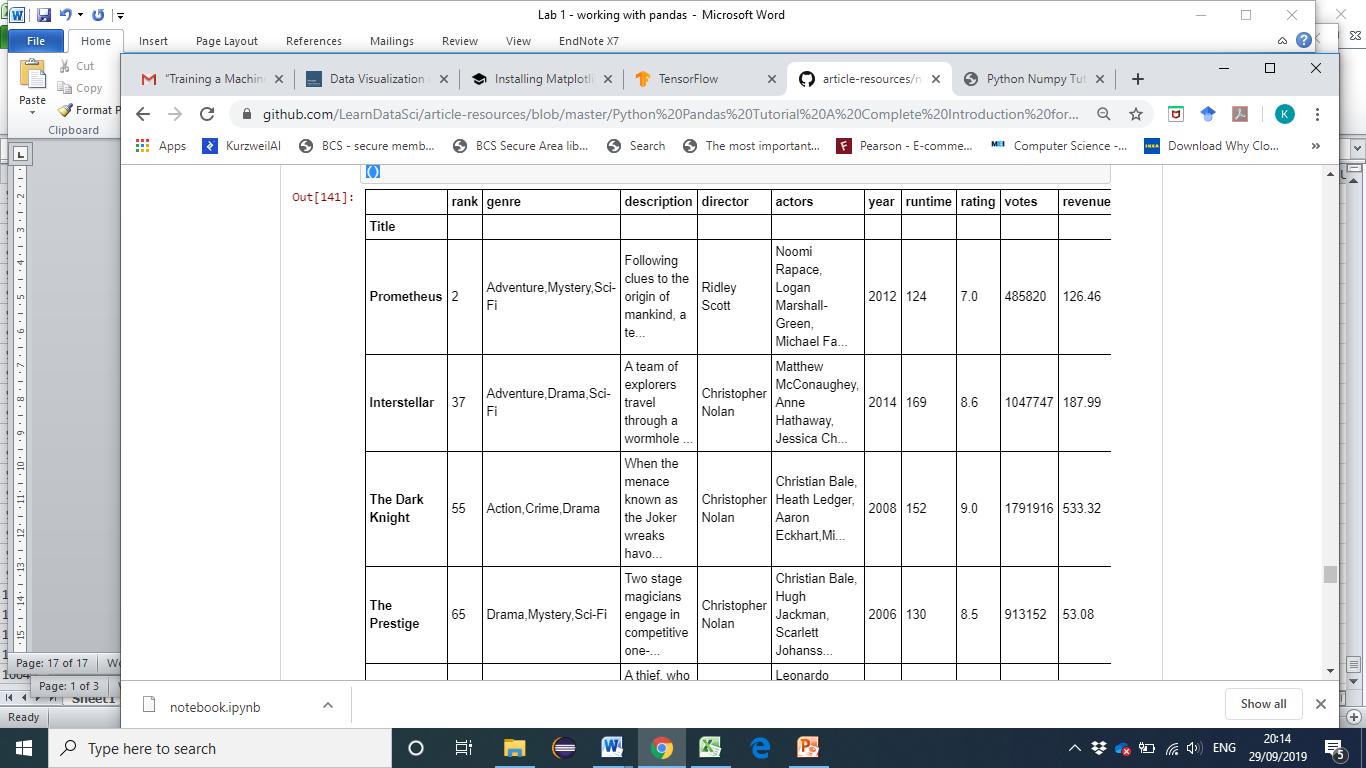


We can make some richer conditionals by using logical operators | for "or" and & for "and".

Let's filter the the DataFrame to show only movies by Christopher Nolan OR Ridley Scott:

In [141]:

movies\_df[(movies\_df['director'] == 'Christopher Nolan') | (movies\_df['director'] == 'Ridley Scott')].head()



We need to make sure to group evaluations with parentheses so Python knows how to evaluate the conditional.

Using the isin() method we could make this more concise though:

In [142]:

movies\_df[movies\_df['director'].isin(['Christopher Nolan', 'Ridley Scott'])].head()



Let's say we want all movies that were released between 2005 and 2010, have a rating above 8.0, but made below the 25th percentile in revenue.

Here's how we could do all of that:

In [149]:

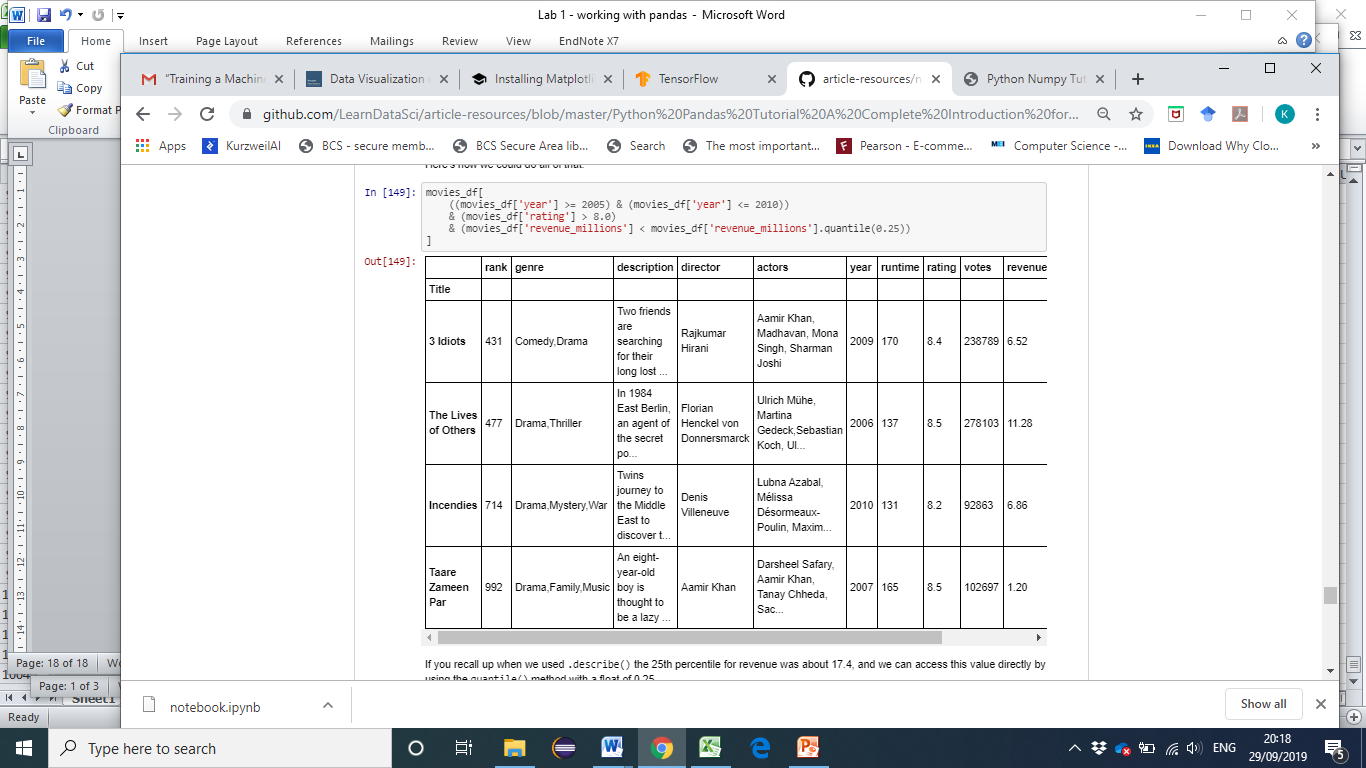
movies\_df[

((movies\_df['year'] >= 2005) & (movies\_df['year'] <= 2010))

& (movies\_df['rating'] > 8.0)

& (movies\_df['revenue\_millions'] < movies\_df['revenue\_millions'].quantile(0.25))

]



## 10 Applying functions

It is possible to iterate over a DataFrame or Series as you would with a list, but doing so — especially on large datasets — is very slow.

An efficient alternative is to apply() a function to the dataset. For example, we could use a function to convert movies with an 8.0 or greater to a string value of "good" and the rest to "bad" and use this transformed values to create a new column.

First we would create a function that, when given a rating, determines if it's good or bad:

In [150]:

**def** rating\_function(x):

**if** x >= 8.0:

**return** "good"

**else**:

**return** "bad"

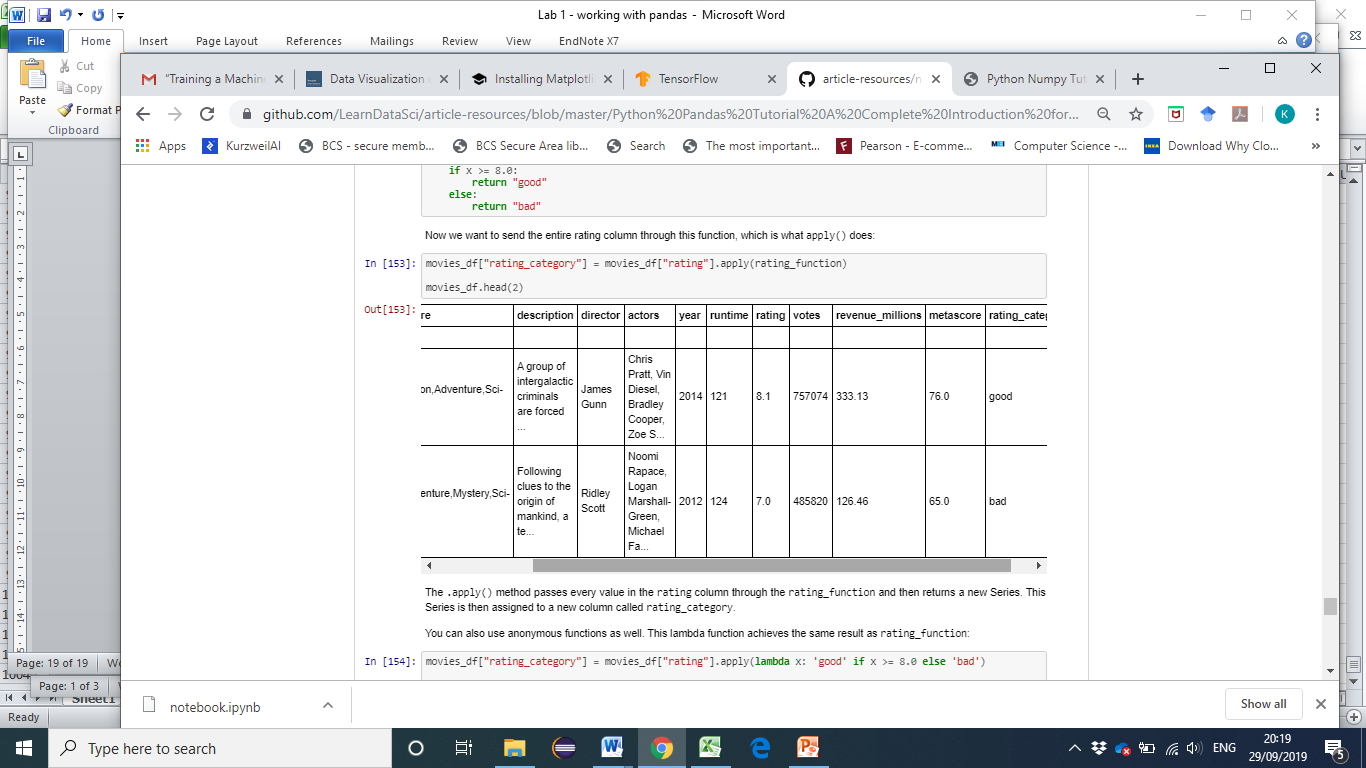
Now we want to send the entire rating column through this function, which is what apply() does:

In [153]:

movies\_df["rating\_category"] = movies\_df["rating"].apply(rating\_function)

movies\_df.head(2)

Out [153]:



The .apply() method passes every value in the rating column through the rating\_function and then returns a new Series. This Series is then assigned to a new column called rating\_category.

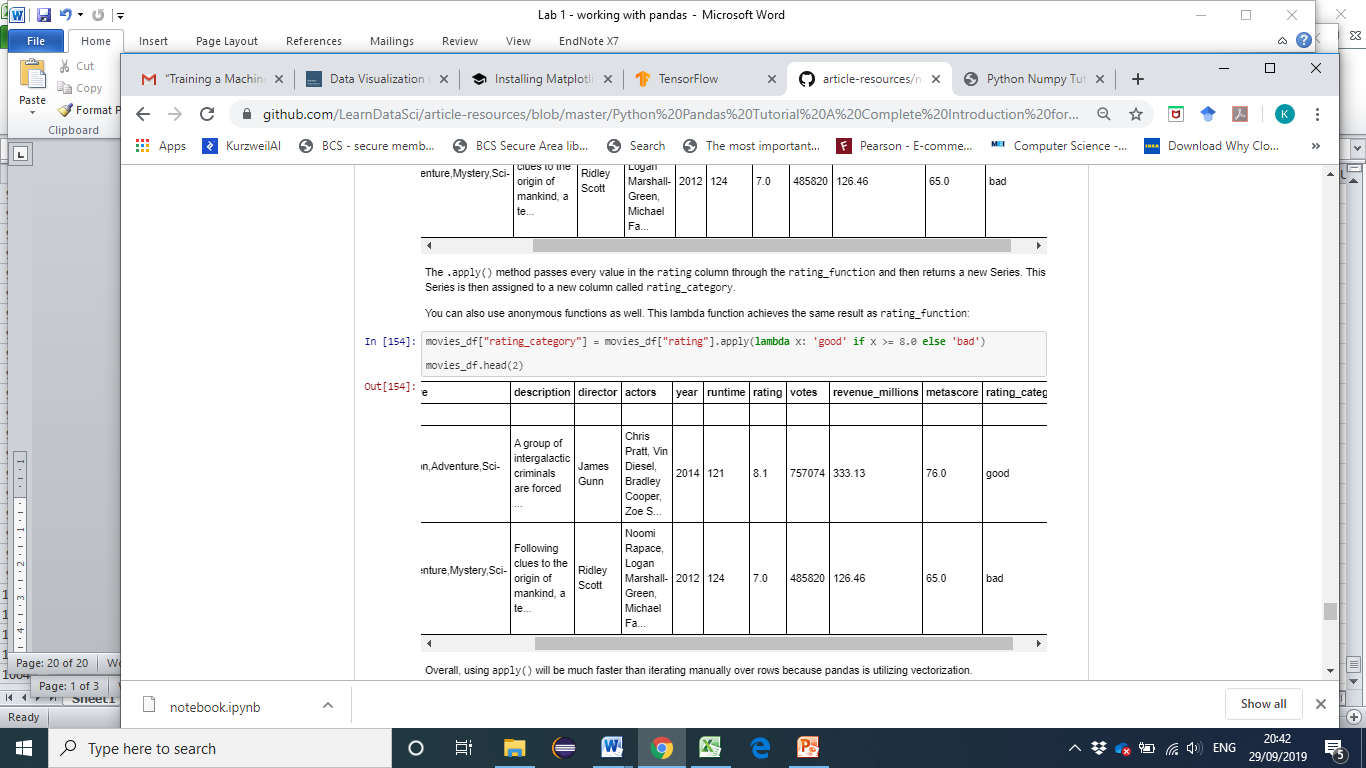
You can also use anonymous functions as well. This lambda function achieves the same result as rating\_function:

In [154]:

movies\_df["rating\_category"] = movies\_df["rating"].apply(**lambda** x: 'good' **if** x >= 8.0 **else** 'bad')

movies\_df.head(2)

Out [154]:



This lab is based on: <https://www.learndatasci.com/tutorials/python-pandas-tutorial-complete-introduction-for-beginners/>