PANDAS Basics

In this lab, we are going to cover some of the most important PANDA operations with a concrete example (IMDB dataset). The first thing that you will need to do is to download the IMDB-Movie-Data.csv from the blackboard and upload it in your Jupyter project folder.



Create a new Jupyter notebook, call it PANDA\_IMDB and we will start our play from here.

## Loading Data

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

**import pandas as pd**

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

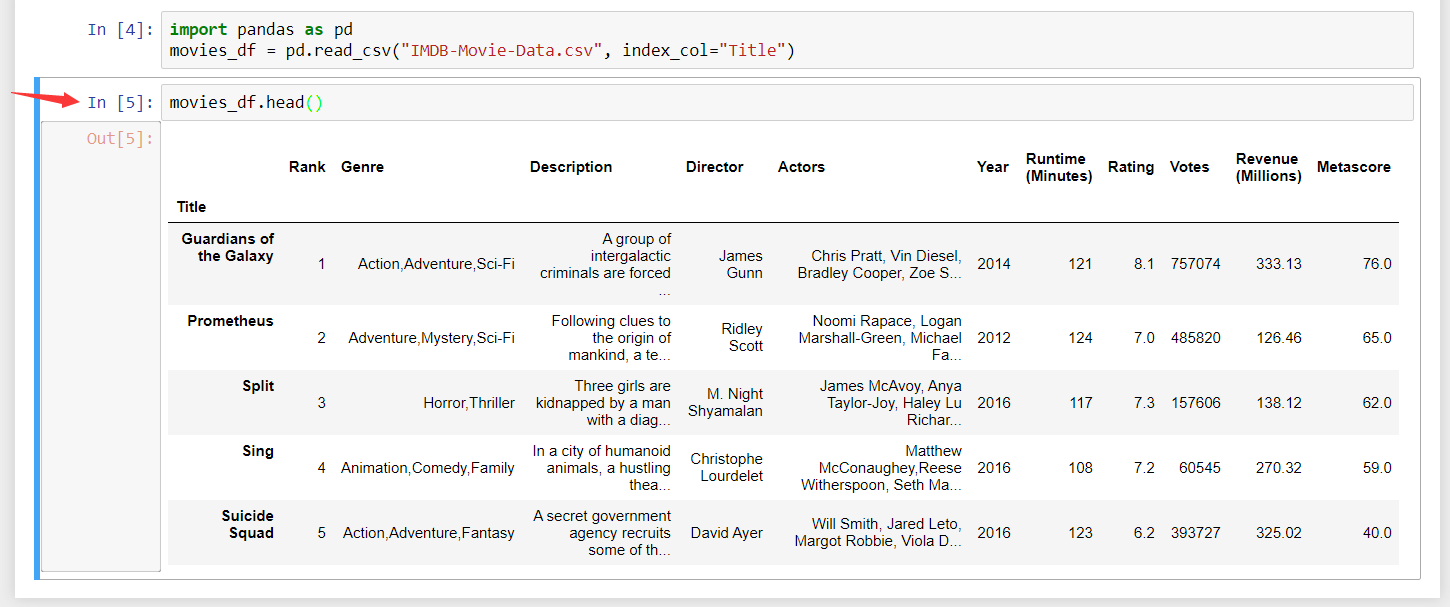
We're loading this dataset from a CSV and designating the movie titles to be our index. You should not see anything in your Jupyter output cell yet as we only read the csv file into the movies\_df object.

## Viewing 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().**

**movies\_df.head()**

Make sure, in your Jupyter notebook, you run the above script in a separate cell



**.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. Try to play with the number yourself.

To see the last five rows use .**tail(). tail()** also accepts a number, and in this case we printing the bottom 8 rows.:

**movies\_df.tail(8)**



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.

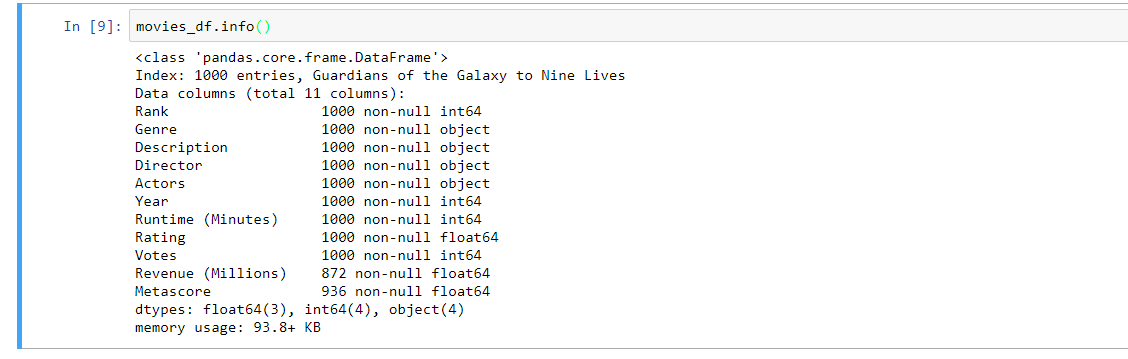
You'll notice that the index in our DataFrame is the Title column, which you can tell by how the word Title is slightly lower than the rest of the columns.

Also, Looking at the data, we already notice that there are some missing values of some columns (features). No worries, you will see how to handle them soon.

## Getting info about your data

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

**movies\_df.info()**



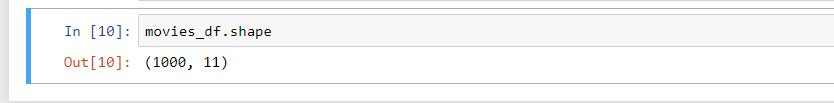
**.info()** provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

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.

Seeing the datatype quickly is actually quite useful. Imagine you just imported some CSVs/JSON and the integers were recorded as strings. You go to do some arithmetic and find an "unsupported operand" Exception because you can't do math with strings. Calling .info() will quickly point out that your column you thought was all integers are actually string objects.

Another fast and useful **attribute (not a method)** is .shape, which outputs just a tuple of (rows, columns):

**movies\_df.shape**



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.

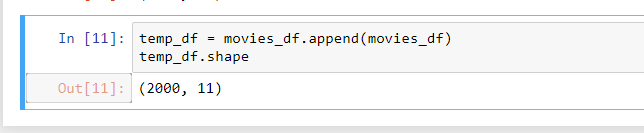
## Handling duplicates

This dataset does not have duplicate rows, but it is always important to verify you aren't aggregating duplicate rows.

To demonstrate, let's simply just double up our movies DataFrame by appending it to itself:

**temp\_df = movies\_df.append(movies\_df)**

**temp\_df.shape**



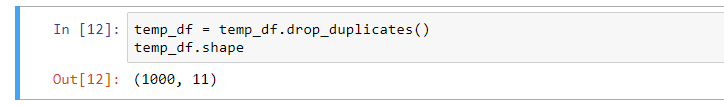
Using append() will return a copy without affecting the original DataFrame. We are capturing this copy in temp so we aren't working with the real data.

Notice call .shape quickly proves our DataFrame rows have doubled.

Now we can try dropping duplicates:

**temp\_df = temp\_df.drop\_duplicates()**

**temp\_df.shape**



Just like append(), the drop\_duplicates() method will also return a copy of your DataFrame, but this time with duplicates removed. Calling .shape confirms we're back to the 1000 rows of our original dataset.

It's a little verbose to keep assigning DataFrames to the same variable like in this example. For this reason, pandas has the inplace keyword argument on many of its methods. Using inplace=True will modify the DataFrame object in place:

**temp\_df.drop\_duplicates(inplace=True)**

Now our temp\_df will have the transformed data automatically.

Another important argument for drop\_duplicates() is **keep**, which has three possible options:

* first: (default) Drop duplicates except for the first occurrence.
* last: Drop duplicates except for the last occurrence.
* False: Drop all duplicates.

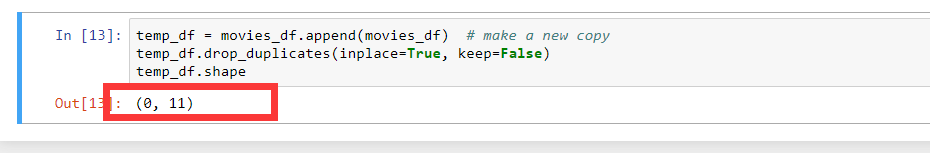
Since we didn't define the keep arugment in the previous example it was defaulted to first. This means that if two rows are the same pandas will drop the second row and keep the first row. Using last has the opposite effect: the first row is dropped.

keep, on the other hand, will drop all duplicates. If two rows are the same then both will be dropped. Watch what happens to temp\_df:

**temp\_df = movies\_df.append(movies\_df) # make a new copy**

**temp\_df.drop\_duplicates(inplace=True, keep=False)**

**temp\_df.shape**



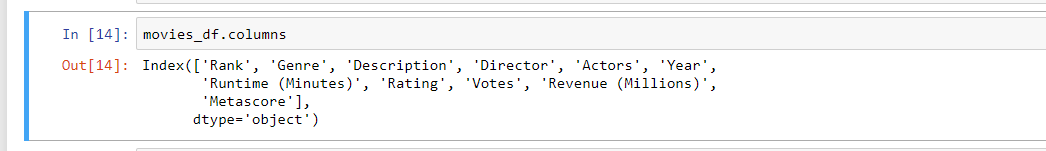
Since all rows were duplicates, keep=False dropped them all resulting in zero rows being left over. If you're wondering why you would want to do this, one reason is that it allows you to locate all duplicates in your dataset. When conditional selections are shown below you'll see how to do that.

## 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:

**movies\_df.columns**



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:

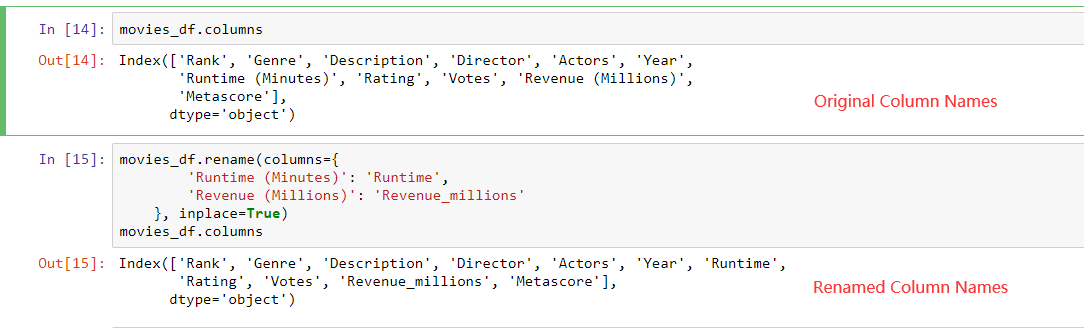
**movies\_df.rename(columns={**

**'Runtime (Minutes)': 'Runtime',**

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

**}, inplace=True)**

**movies\_df.columns**

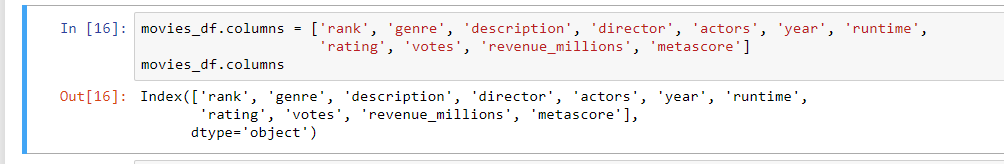


Excellent. But what if we want to lowercase all names? Instead of using .rename() we could also set a list of names to the columns like so:

**movies\_df.columns = ['rank', 'genre', 'description', 'director', 'actors', 'year', 'runtime',**

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

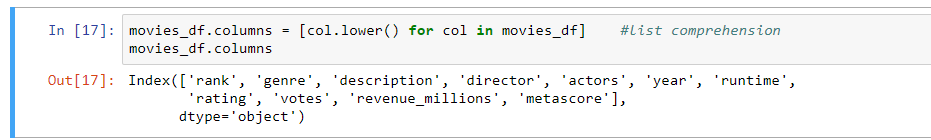
**movies\_df.columns**



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

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

**movies\_df.columns**



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.

## How to work with missing values (dumb approaches)

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:

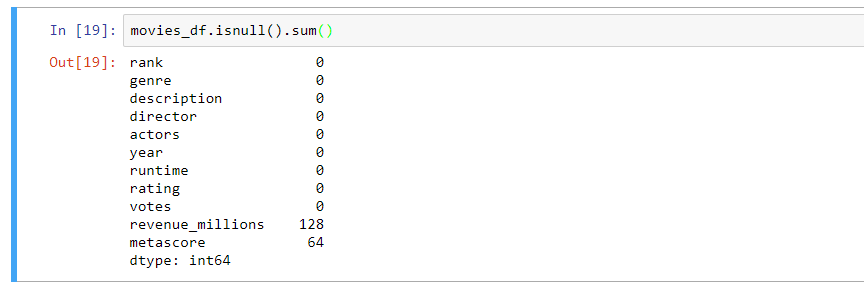
**movies\_df.isnull()**



Notice isnull() returns a new 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:

**movies\_df.isnull().sum()**



.isnull() just by iteself 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.

## 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:

**movies\_df.dropna()**



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 **if and only if** you want the original dataset to be affected.

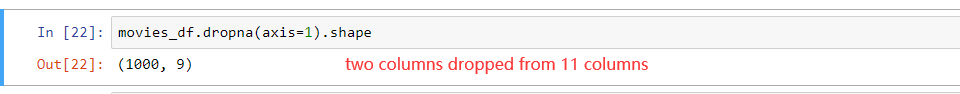
So in the case of our dataset, this operation would remove 128 rows where revenue\_millions is null and 64 rows where metascore is null.

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:

**movies\_df.dropna(axis=1)**

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



*INTUITION*

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:

*movies\_df.shape*

*Out: (1000, 11)*

As we learned above, this is a tuple that represents the shape of the DataFrame, i.e. 1000 rows and 11 columns. N**ote 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. This is generally true for all Numpy vector/matrices based operations.**

# 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. 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.

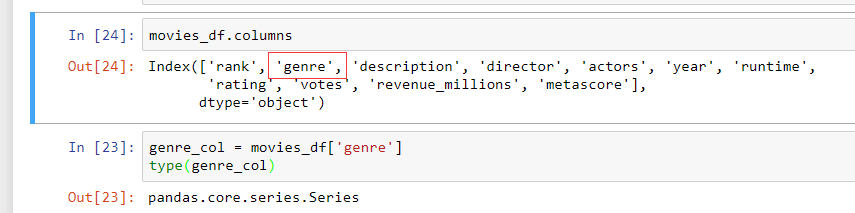
Let's look at working with columns first.

## By column

You can simply select a column using square brackets like this:

**genre\_col = movies\_df['genre']**

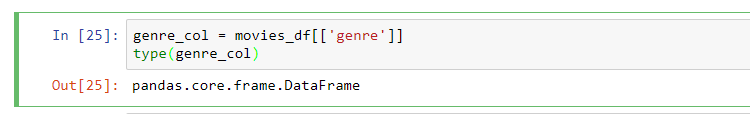
**type(genre\_col)**



This will return a Series. To extract a column as a DataFrame, you need to pass a list of column names. In our case that's just a single column:

**genre\_col = movies\_df[['genre']]**

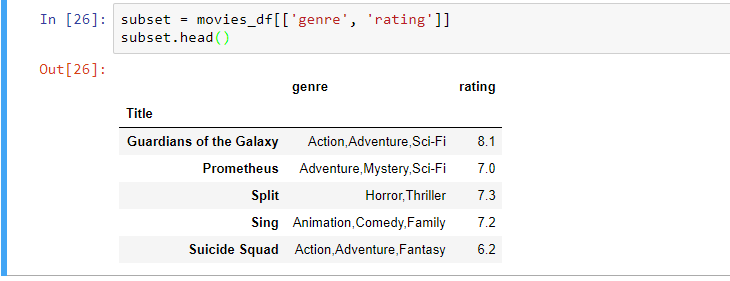
**type(genre\_col)**



Since it's just a list, adding another column name is easy:

**subset = movies\_df[['genre', 'rating']]**

**subset.head()**



## Now we'll look at getting data by rows.

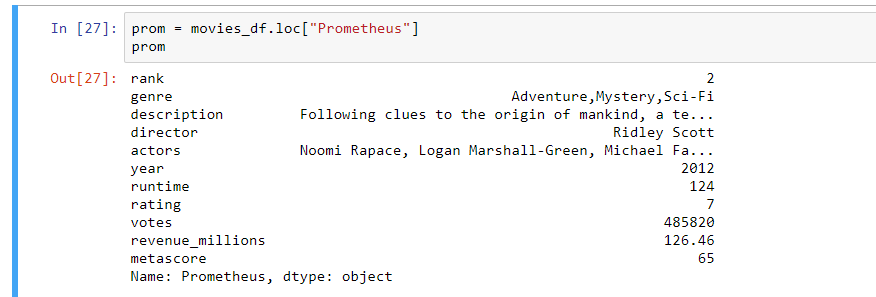
For rows, we have two options:

1. .loc - locates by name
2. .iloc- locates by numerical index

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

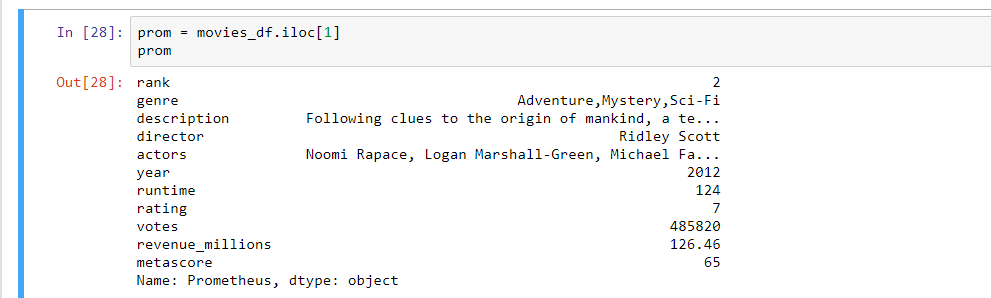
**prom = movies\_df.loc["Prometheus"]**

**prom**



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

**prom = movies\_df.iloc[1]**

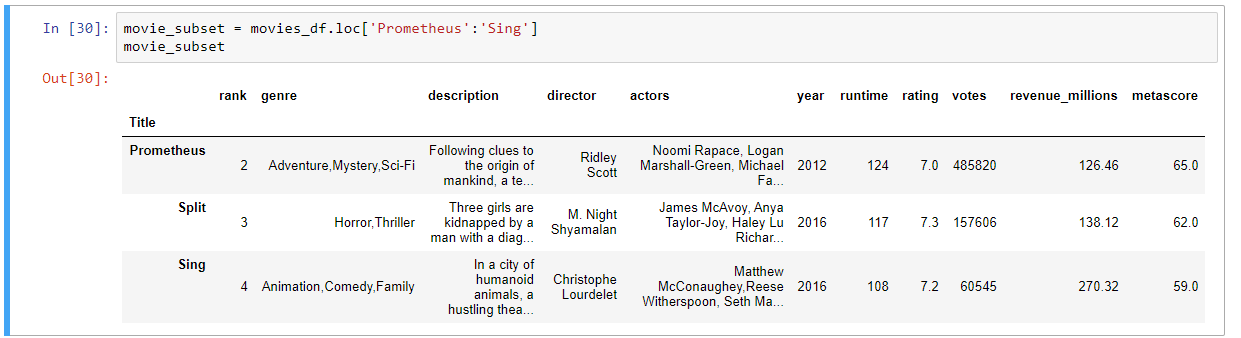


**loc** and **iloc** can be thought of as similar to Python list slicing. To show this even further, let's select multiple rows.

How would you do it with a list? In Python, just slice with brackets like example\_list[1:4]. It's works the same way in pandas:

**movie\_subset = movies\_df.loc['Prometheus':'Sing']**

**movie\_subset**



**movie\_subset = movies\_df.iloc[1:4]**

**movie\_subset**



## Conditional selections

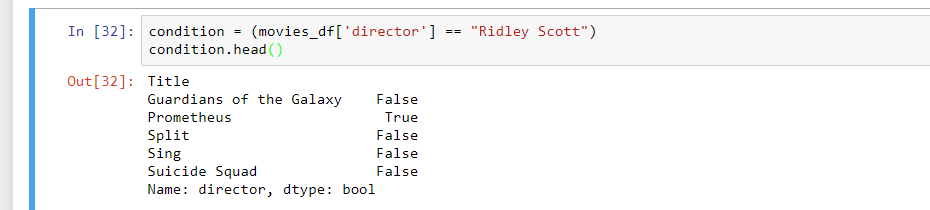
We’ve gone over how to select columns and rows, but what if we want to make a conditional selection?

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:

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

**condition.head()**



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. These are not the actual data records that we want, but are a list of conditional masks that can be used later.

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:

**movies\_df[movies\_df['director'] == "Ridley Scott"]**

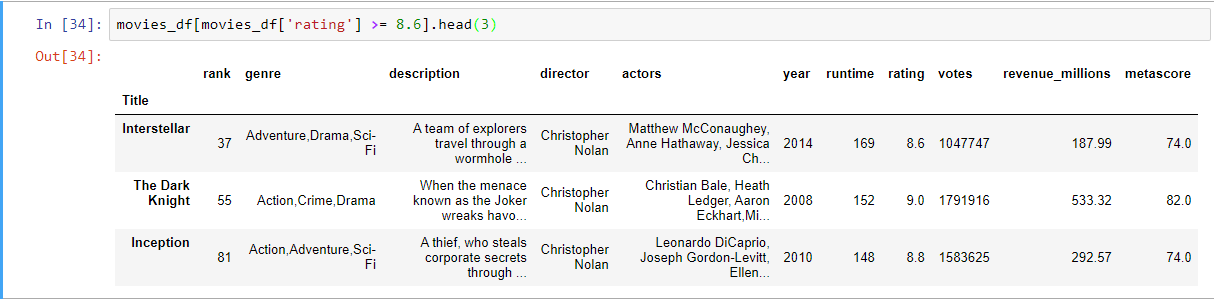


You can get used to looking at these conditionals by reading it like:

**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:

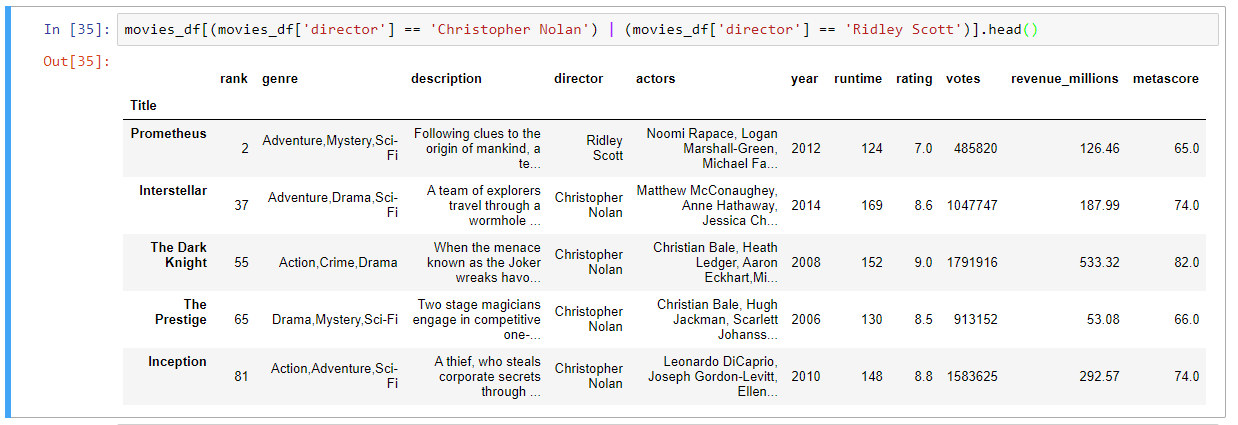
**movies\_df[movies\_df['rating'] >= 8.6].head(3)**



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:

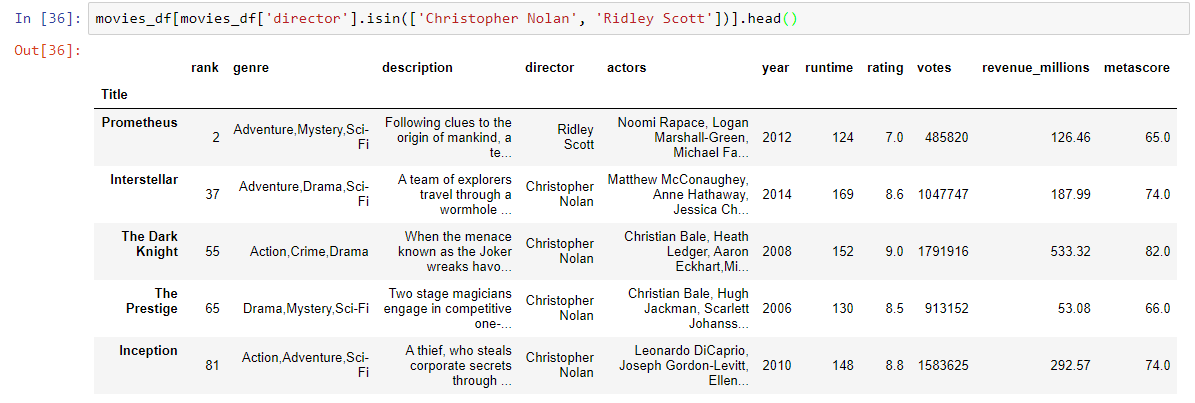
**movies\_df[(movies\_df['director'] == 'Christopher Nolan') | (movies\_df['director'] == 'Ridley Scott')].head()**



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:

**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:

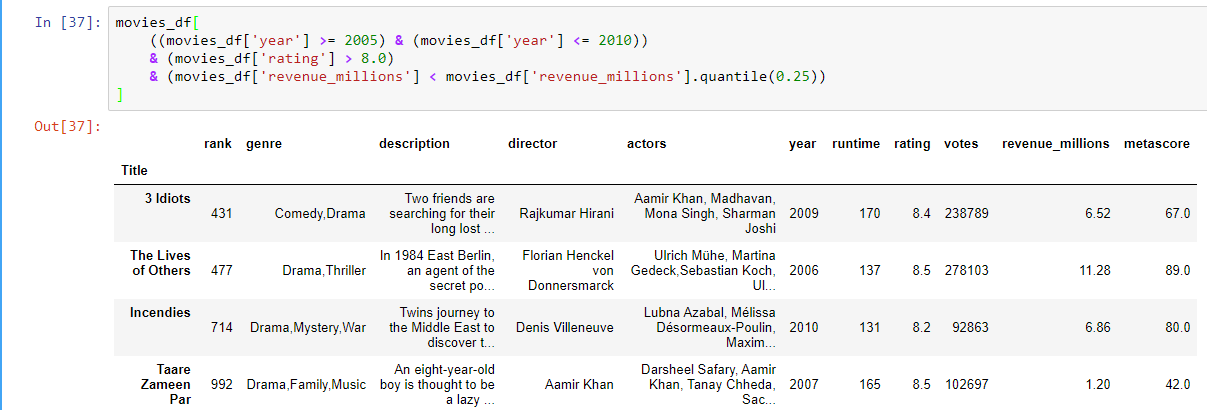
**movies\_df[**

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

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

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

**]**



## 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:

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

**if x >= 8.0:**

**return "good"**

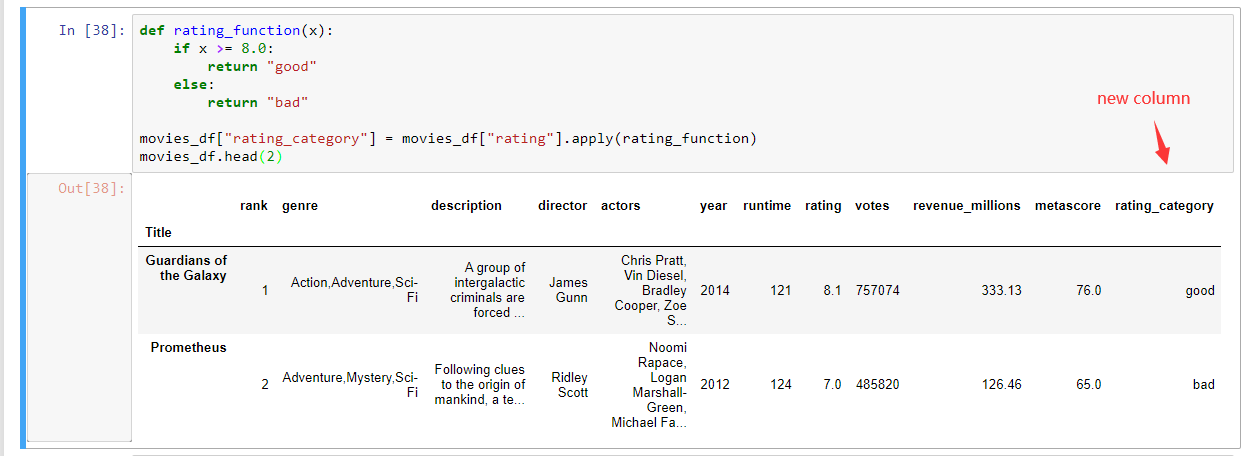
**else:**

**return "bad"**

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

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

**movies\_df.head(2)**



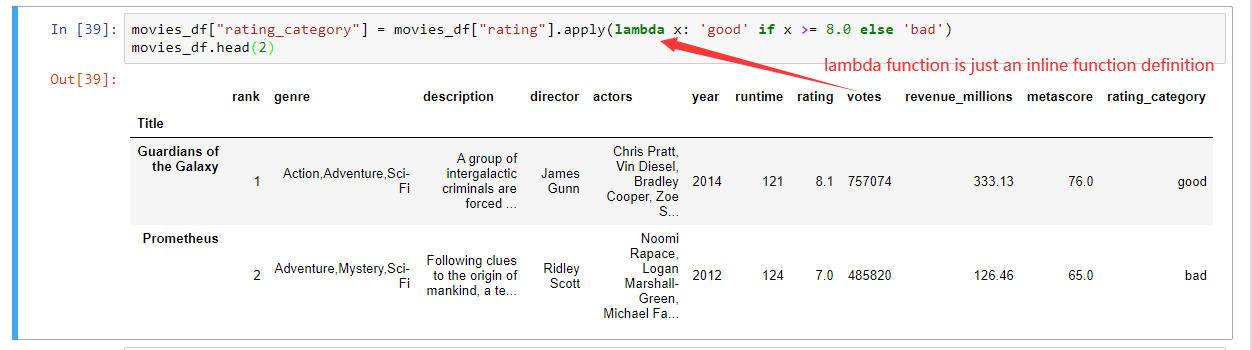
**This example is crucially important when we come to prepare training data for classifications later.**

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:

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

**movies\_df.head(2)**



# Brief Plotting

Another great thing about pandas is that it integrates with Matplotlib, so you get the ability to plot directly off DataFrames and Series. To get started we need to import Matplotlib

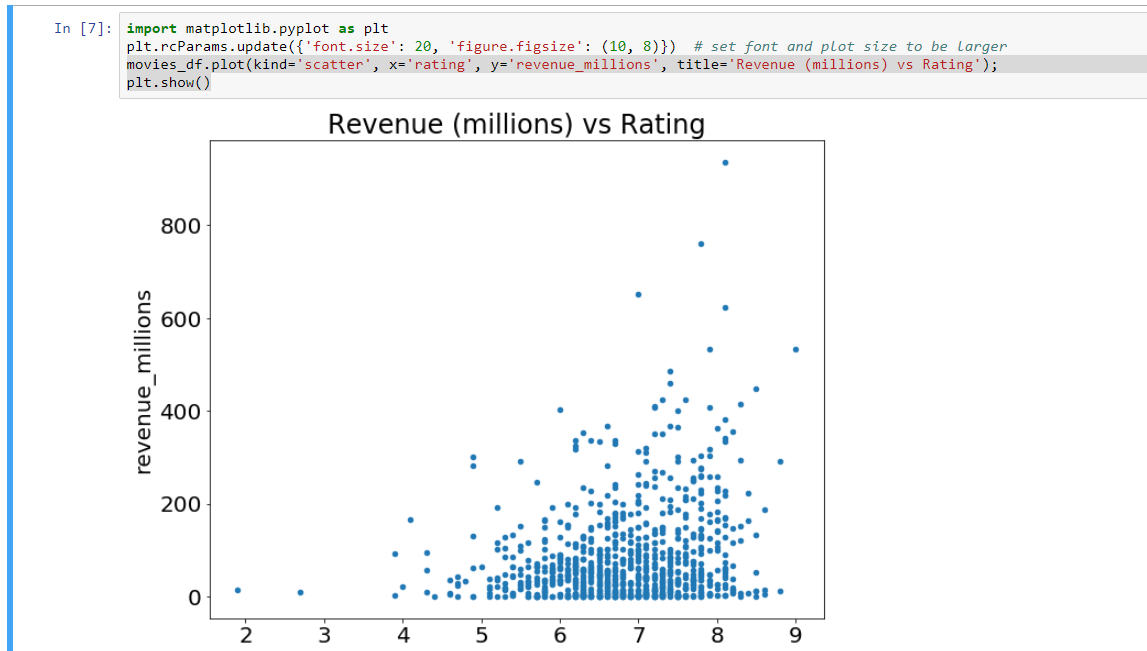
**import matplotlib.pyplot as plt**

**plt.rcParams.update({'font.size': 20, 'figure.figsize': (10, 8)}) # set font and plot size to be larger**

Let's plot the relationship between ratings and revenue. All we need to do is call .plot() on movies\_df with some info about how to construct the plot:

**movies\_df.plot(kind='scatter', x='rating', y='revenue\_millions', title='Revenue (millions) vs Rating');**

**plot.show() #don’t forget to call this, otherwise your chart will not show up**

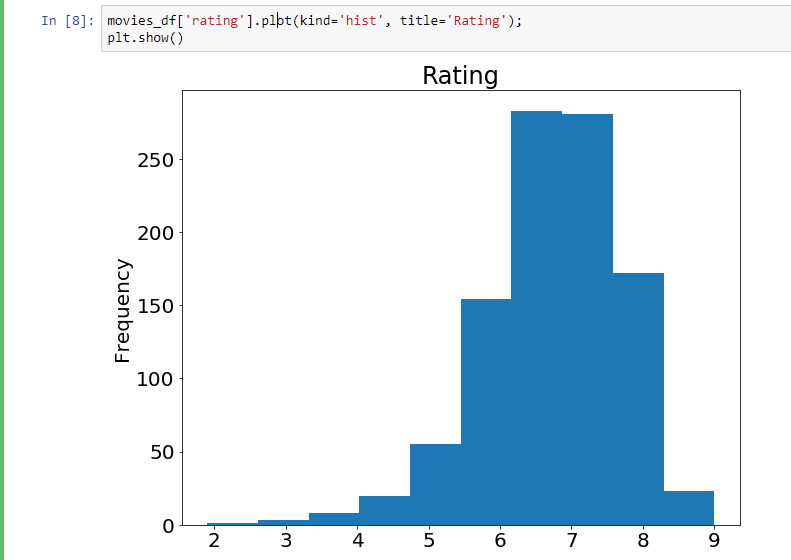


What's with the semicolon? It's not a syntax error, just a way to hide the <matplotlib.axes.\_subplots.AxesSubplot at 0x26613b5cc18> output when plotting in Jupyter notebooks.

If we want to plot a simple Histogram based on a single column, we can call plot on a column:

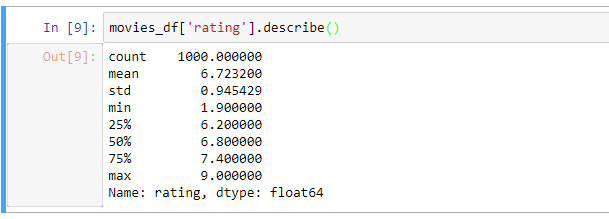
**movies\_df['rating'].plot(kind='hist', title='Rating');**

**plt.show()**



Do you remember the .describe() example at the beginning of this tutorial? Well, there's a graphical representation of the interquartile range, called the Boxplot. Let's recall what describe() gives us on the ratings column:

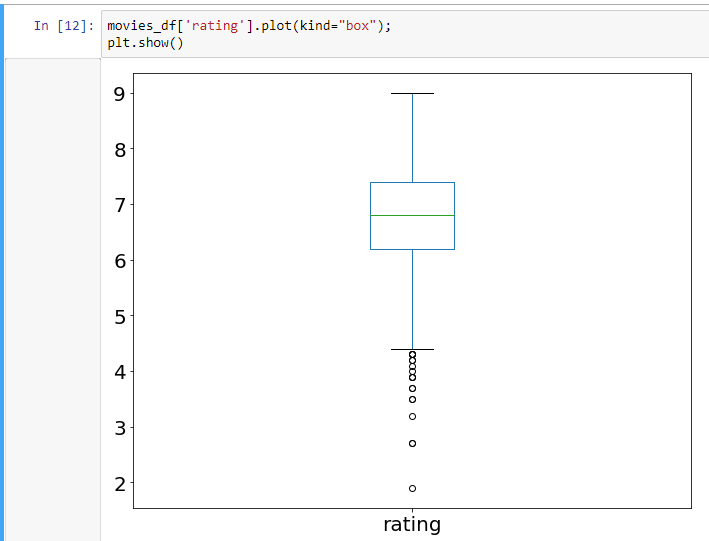
**movies\_df['rating'].describe()**



Using a Boxplot we can visualize this data:

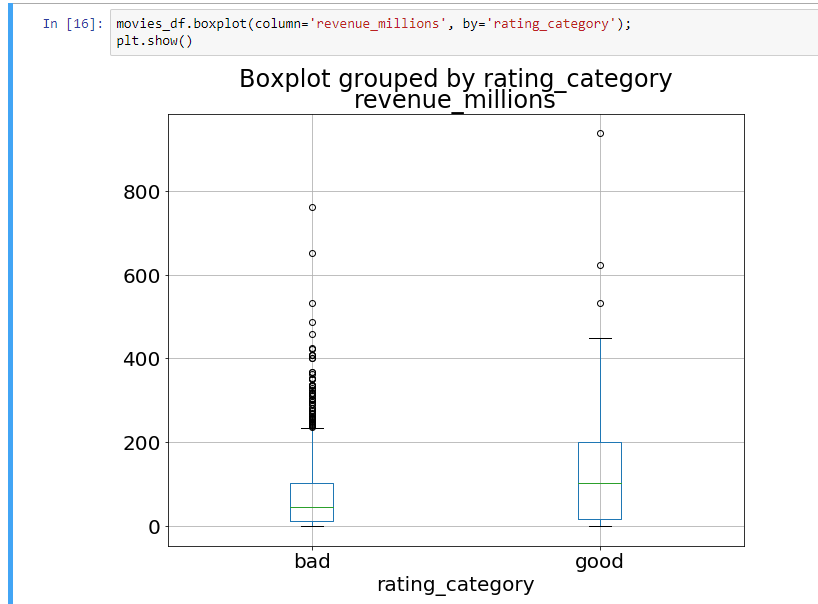
**movies\_df['rating'].plot(kind="box");**

**plt.show()**



By combining categorical and continuous data, we can create a Boxplot of revenue that is grouped by the Rating Category we created above:

**movies\_df.boxplot(column='revenue\_millions', by='rating\_category');**



That's the general idea of plotting with pandas. There's too many plots to mention, so definitely take a look at the plot() [docs here](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html) for more information on what it can do.

## Wrapping up

Exploring, cleaning, transforming, and visualization data with pandas in Python is an essential skill in data science. Just cleaning wrangling data is 80% of your job as a Data Scientist. We will come across more power operations in the following lectures and labs.