Python Libraries and Pandas

Python has emerged as the go-to language in data science, and it is one of the essential skills required in data science. Python libraries for data analysis are designed with their specifications.

Most Popular Python Libraries for Data Analysis

There are many data analysis libraries in Python that are built to perform numerous functions, contain tools, and have methods to manage and analyze data. Each has a particular objective while managing images, textual data, data mining, data visualization, and more.

1. **Pandas**

Pandas is a powerful data manipulation and analysis library for Python. It provides data structures and functions for working with structured data, making it an essential tool for data scientists and analysts.

Key Features:

- Pandas introduces DataFrame, a two-dimensional table-like data structure for easy data handling.
- It offers tools for cleaning and preprocessing data, including handling missing values and duplicate entries.
- You can select, filter, and manipulate data with simple and intuitive syntax.
- Pandas excel at working with time series data and offer robust tools for time-based analysis.
- It integrates well with other data analysis libraries like NumPy and visualization libraries like Matplotlib.

2. **NumPy**

NumPy, short for Numerical Python, is a fundamental library for numerical and mathematical operations in Python. It provides support for large, multi-dimensional arrays and matrices, along with a variety of mathematical functions to operate on them efficiently.

Key Features:

- NumPy's ndarray is a powerful array object that allows efficient element-wise operations and broadcasting.
- It offers a wide range of mathematical functions, including linear algebra, Fourier analysis, and random number generation.
- NumPy is written in C and can be seamlessly integrated with code written in C, C++, and Fortran.
- NumPy allows for element-wise operations on arrays of different shapes, making it convenient for mathematical operations on arrays of varying sizes.
- NumPy efficiently manages memory, making it suitable for handling large datasets.

3. **Matplotlib**

Matplotlib is one of the best python data visualization libraries for generating powerful yet simple visualization. It is a 2-D plotting library that can be used in various ways.

Key Features

- It supports various types of graphical representation, including line graphs, bar graphs, and histograms.
- It can work with the NumPy arrays and border SciPy stack.
- It has a huge number of plots for understanding trends and making correlations.

4. Seaborn

Seaborn is the best python library for data visualization, which offers a variety of visualized patterns. It is designed to work more compatible with Pandas data form and is widely used for statistical visualization.

Key Features

- It performs the necessary mapping and aggregation to form information visuals.
- It is integrated to explore and understand data in a better and more detailed way.
- It offers a high level of crossing point for creating beautiful and informative algebraic graphics.

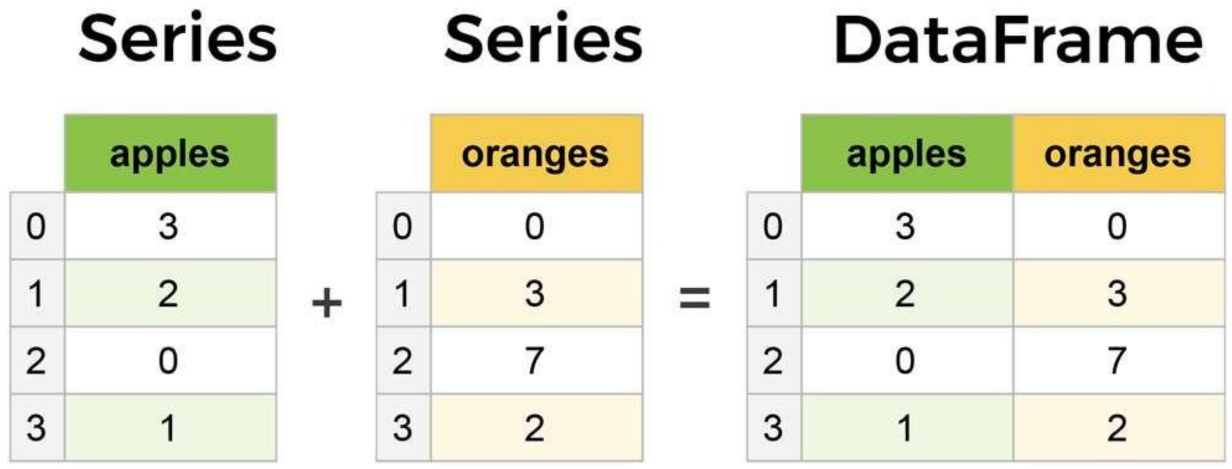
Pandas Library

Pandas is a powerful data manipulation and analysis library. It provides data structures like DataFrames and Series for working with tabular and time-series data.

Basic data structures in pandas

Pandas provides two types of classes for handling data:

1. **Series:** a one-dimensional labeled array holding data of any type, such as integers, strings, Python objects etc.
2. **DataFrame:** a two-dimensional data structure that holds data like a two-dimension array or a table with rows and columns.



Series

	apples
0	3
1	2
2	0
3	1

+

Series

	oranges
0	0
1	3
2	7
3	2

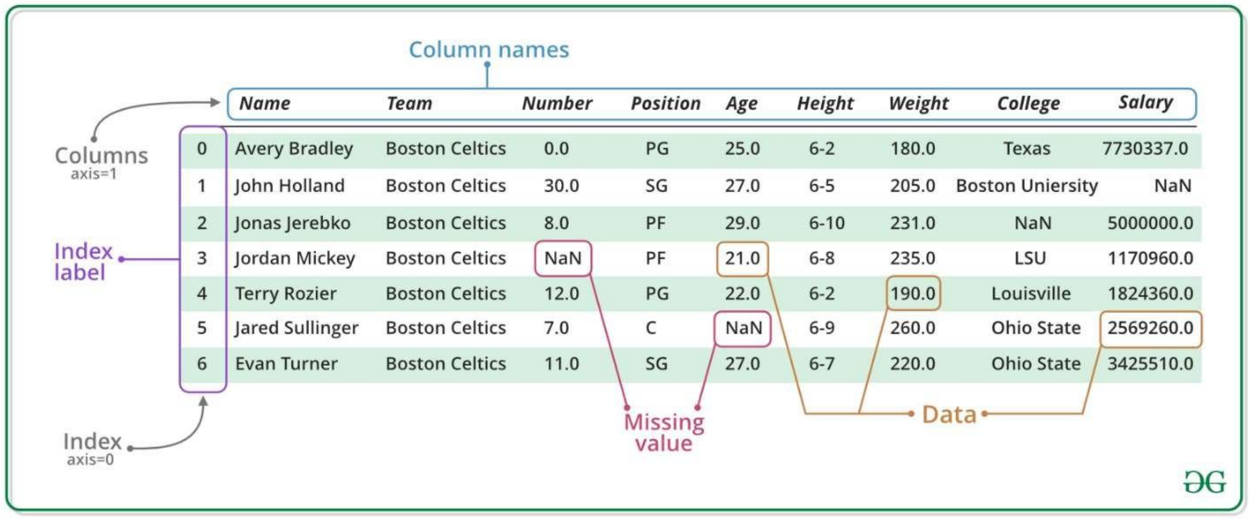
=

DataFrame

	apples	oranges
0	3	0
1	2	3
2	0	7
3	1	2

2.1 DataFrame

A Dataframe is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. In dataframe datasets arrange in rows and columns, we can store any number of datasets in a dataframe. We can perform many operations on these datasets like arithmetic operation, columns/rows selection, columns/rows addition etc.


Column names
Columns
axis=1
Index
label
Index
axis=0
Missing
value
Data

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston Uniersity	NaN
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0	6-8	235.0	LSU	1170960.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0	6-2	190.0	Louisville	1824360.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN	6-9	260.0	Ohio State	2569260.0
6	Evan Turner	Boston Celtics	11.0	SG	27.0	6-7	220.0	Ohio State	3425510.0

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:

Example:

```
data = {
    'apples': [3, 2, 0, 1],
    'oranges': [0, 3, 7, 2]
}

  purchases = pd.DataFrame(data)

  purchases
```

Output:

	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
```

Output:

	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']
```

Output:

```
apples     3
oranges    0
Name: June, dtype: int64
```

2.1.1 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**

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

```
df = pd.read_csv('purchases.csv')

df
```

Output:

	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
```

Output:

	apples	oranges
June	3	0
Robert	2	3
Lily	0	7
David	1	2

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

- **Reading data from Excel files (multiple sheets)**

Reading Excel files with multiple sheets is not that different. You just need to specify one additional argument, sheet_name, where you can either pass a string for the sheet name or an integer for the sheet position (note that Python uses 0-indexing, where the first sheet can be accessed with sheet_name = 0)

```
# Extracting the second sheet since Python uses 0-indexing
df = pd.read_excel('diabetes_multi.xlsx', sheet_name=1)
```

- **Reading data from a SQL database**

If you're working with data from a SQL database you need to first establish a connection using an appropriate Python library, then pass a query to pandas. Here we'll use SQLite to demonstrate.
run this command in your notebook:

```
    !pip install pysqlite3
```

sqlite3 is used to create a connection to a database which we can then use to generate a DataFrame through a SELECT query.

So first we'll make a connection to a SQLite database file:

```
    import sqlite3

    con = sqlite3.connect("database.db")
```

In this SQLite database we have a table called *purchases*, and our index is in a column called "index".

By passing a SELECT query and our con, we can read from the *purchases* table:

```
    df = pd.read_sql_query("SELECT * FROM purchases", con)
```

```
df
```

Output:

	index	apples	oranges
0	June	3	0
1	Robert	2	3
2	Lily	0	7
3	David	1	2

Just like with CSVs, we could pass index_col='index', but we can also set an index after-the-fact:

```
df = df.set_index('index')

df
```

Output:

	apples	oranges
index		
June	3	0
Robert	2	3
Lily	0	7
David	1	2

In fact, we could use set_index() on *any* DataFrame using *any* column at *any* time. Indexing Series and DataFrames is a very common task, and the different ways of doing it is worth remembering.

2.1.2 Converting back to a CSV, Excel or SQL

So, after extensive work on cleaning your data, you’re now ready to save it as a file of your choice. Like the way we read in data, pandas provide intuitive commands to save it:

```
df.to_csv('new_purchases.csv')

df.to_excel('new_purchases.xlsx’)

df.to_sql('new_purchases', con)
```

When we save Excel and CSV files, all we have to input into those functions is our desired filename with the appropriate file extension. With SQL, we’re not creating a new file but instead inserting a new table into the database using our con variable from before.

2.1.3 Most important DataFrame operations

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

```
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():

```
movies_df.head()
```

.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.:

```
movies_df.tail(2)
```

- **Getting info about your data**

```
movies_df.info()
```

Output:

```
<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.

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

```
movies_df.shape
```

Output:

```
(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.

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()
```

Just like append(), the drop_duplicates() method will also return a copy of your DataFrame, but this time with duplicates removed.

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.

False, on the other hand, will drop all duplicates. If two rows are the same then both will be dropped.

```
temp_df = movies_df.append(movies_df)  # make a new copy

temp_df.drop_duplicates(inplace=True, keep=False)
```

- **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
```

Output:

```
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:

Example 1:

```
movies_df.rename(columns={
        'Runtime (Minutes)': 'Runtime',
        'Revenue (Millions)': 'Revenue_millions'
    }, inplace=True)


movies_df.columns
```

Output:

```
Index(['Rank', 'Genre', 'Description', 'Director', 'Actors',
'Year', 'Runtime','Rating', 'Votes', 'Revenue_millions',
'Metascore'],dtype='object')
```

Example 2:

```
    # rename Pandas columns to lower case
    df.columns= df.columns.str.lower()
    df.columns
```

Output:

```
    Index(['column1', 'column2', 'column3'], dtype='object')
```

In addition to upper cases, sometimes column names can have both leading and trailing empty spaces.

Example 3:

```
    df=pd.DataFrame({" C1 ":c1,
                     "C2":c2,
                     "C3 ":c3})
```

Output:

```
    Index([' C1 ', 'C2', 'C3 '], dtype='object')
```

We can use str.strip() function Pandas to strip the leading and trailing white spaces. Here we also convert the column names into lower cases using str.lower() as before.

Example 4:

```
# Column names: remove white spaces and convert to lower case
df.columns= df.columns.str.strip().str.lower()
df.columns
```

Output:

```
Index(['c1', 'c2', 'c3'], dtype='object')
```

- **How to work with missing values**

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:

Example:

```
movies_df.isnull()
```

Output:

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore
Title											
Guardians of the Galaxy	False	False	False	False	False	False	False	False	False	False	False
Prometheus	False	False	False	False	False	False	False	False	False	False	False
Split	False	False	False	False	False	False	False	False	False	False	False
Sing	False	False	False	False	False	False	False	False	False	False	False
Suicide Squad	False	False	False	False	False	False	False	False	False	False	False

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:

Example:

```
movies_df.isnull().sum()
```

Output:

```
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 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.

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

```
movies_df.dropna(axis=1)
```

The 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. 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.

- ## Understanding your variables

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

```
movies_df.describe()
```

Output:

	rank	year	runtime	rating	votes	revenue_millions	metascore
count	1000.000000	1000.000000	1000.000000	1000.000000	1.000000e+03	1000.000000	936.000000
mean	500.500000	2012.783000	113.172000	6.723200	1.698083e+05	82.956376	58.985043
std	288.819436	3.205962	18.810908	0.945429	1.887626e+05	96.412043	17.194757
min	1.000000	2006.000000	66.000000	1.900000	6.100000e+01	0.000000	11.000000
25%	250.750000	2010.000000	100.000000	6.200000	3.630900e+04	17.442500	47.000000
50%	500.500000	2014.000000	111.000000	6.800000	1.107990e+05	60.375000	59.500000
75%	750.250000	2016.000000	123.000000	7.400000	2.399098e+05	99.177500	72.000000
max	1000.000000	2016.000000	191.000000	9.000000	1.791916e+06	936.630000	100.000000

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:

```
movies_df['genre'].describe()
```

Output:

```
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).

```
movies_df['genre'].value_counts().head(10)
```

Output:

```
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:

```
    movies_df.corr()
```

Output:

	rank	year	runtime	rating	votes	revenue_millions	metascore
rank	1.000000	-0.261605	-0.221739	-0.219555	-0.283876	-0.252996	-0.191869
year	-0.261605	1.000000	-0.164900	-0.211219	-0.411904	-0.117562	-0.079305
runtime	-0.221739	-0.164900	1.000000	0.392214	0.407062	0.247834	0.211978
rating	-0.219555	-0.211219	0.392214	1.000000	0.511537	0.189527	0.631897
votes	-0.283876	-0.411904	0.407062	0.511537	1.000000	0.607941	0.325684
revenue_millions	-0.252996	-0.117562	0.247834	0.189527	0.607941	1.000000	0.133328
metascore	-0.191869	-0.079305	0.211978	0.631897	0.325684	0.133328	1.000000

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.

- **DataFrame slicing, selecting, and 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.

1. **By column**

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

```
genre_col = movies_df['genre']

type(genre_col)
```

Output:

```
pandas.core.series.Series
```

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)
pandas.core.frame.DataFrame
```

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

```
subset = movies_df[['genre', 'rating']]

subset.head()
```

Output:

	genre	rating
Title		
Guardians of the Galaxy	Action,Adventure,Sci-Fi	8.1
Prometheus	Adventure,Mystery,Sci-Fi	7.0
Split	Horror,Thriller	7.3

	genre	rating
Title		
Sing	Animation,Comedy,Family	7.2
Suicide Squad	Action,Adventure,Fantasy	6.2

Now we'll look at getting data by rows.

2. **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:

```
    prom = movies_df.loc["Prometheus"]

    prom
```

Output:

```
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
```

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 works the same way in pandas:

```
    movie_subset = movies_df.loc['Prometheus':'Sing']

    movie_subset = movies_df.iloc[1:4]
```

```
movie_subset
```

Output:

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore
Title											
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a te...	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...	2012	124	7.0	485820	126.46	65.0
Split	3	Horror,Thriller	Three girls are kidnapped by a man with a diag...	M. Night Shyamalan	James McAvoy, Anya Taylor-Joy, Haley Lu Richar...	2016	117	7.3	157606	138.12	62.0
Sing	4	Animation,Comedy,Family	In a city of humanoid animals, a hustling thea...	Christophe Lourdelet	Matthew McConaughey,Reese Witherspoon, Seth Ma...	2016	108	7.2	60545	270.32	59.0

One important distinction between using .loc and .iloc to select multiple rows is that .locincludes the movie *Sing* in the result, but when using .iloc we're getting rows 1:4 but the movie at index 4 (*Suicide Squad*) is not included.

Slicing with .iloc follows the same rules as slicing with lists, the object at the index at the end is not included.

- ## 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()
```

Output:

```
Title
Guardians of the Galaxy    False
Prometheus                  True
Split                      False
Sing                       False
Suicide Squad              False
Name: director, dtype: bool
```

conditional selections using numerical values by filtering the DataFrame by ratings:

```
movies_df[movies_df['rating'] >= 8.6].head(3)
```

Output:

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore
Title											
Interstellar	37	Adventure,Drama,Sci-Fi	A team of explorers travel through a wormhole ...	Christopher Nolan	Matthew McConaughey, Anne Hathaway, Jessica Ch...	2014	169	8.6	1047747	187.99	74.0
The Dark Knight	55	Action,Crime,Drama	When the menace known as the Joker wreaks havo...	Christopher Nolan	Christian Bale, Heath Ledger, Aaron Eckhart,Mi...	2008	152	9.0	1791916	533.32	82.0
Inception	81	Action,Adventure,Sci-Fi	A thief, who steals corporate secrets through ...	Christopher Nolan	Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen...	2010	148	8.8	1583625	292.57	74.0

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()
```

Output:

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore
Title											
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a te...	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...	2012	124	7.0	485820	126.46	65.0

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore
Title											
Interstellar	37	Adventure,Drama,Sci-Fi	A team of explorers travel through a wormhole ...	Christopher Nolan	Matthew McConaughey, Anne Hathaway, Jessica Ch...	2014	169	8.6	1047747	187.99	74.0
The Dark Knight	55	Action,Crime,Drama	When the menace known as the Joker wreaks havo...	Christopher Nolan	Christian Bale, Heath Ledger, Aaron Eckhart,Mi...	2008	152	9.0	1791916	533.32	82.0
The Prestige	65	Drama,Mystery,Sci-Fi	Two stage magicians engage in competitive one-...	Christopher Nolan	Christian Bale, Hugh Jackman, Scarlett Johanss...	2006	130	8.5	913152	53.08	66.0
Inception	81	Action,Adventure,Sci-Fi	A thief, who steals corporate secrets through ...	Christopher Nolan	Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen...	2010	148	8.8	1583625	292.57	74.0

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()
```

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore
Title											
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a te...	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...	2012	124	7.0	485820	126.46	65.0
Interstellar	37	Adventure,Drama,Sci-Fi	A team of explorers travel through a wormhole ...	Christopher Nolan	Matthew McConaughey, Anne Hathaway, Jessica Ch...	2014	169	8.6	1047747	187.99	74.0
The Dark Knight	55	Action,Crime,Drama	When the menace known as the Joker wreaks havo...	Christopher Nolan	Christian Bale, Heath Ledger, Aaron Eckhart,Mi...	2008	152	9.0	1791916	533.32	82.0
The Prestige	65	Drama,Mystery,Sci-Fi	Two stage magicians engage in competitive one-...	Christopher Nolan	Christian Bale, Hugh Jackman, Scarlett Johanss...	2006	130	8.5	913152	53.08	66.0
Inception	81	Action,Adventure,Sci-Fi	A thief, who steals corporate secrets through ...	Christopher Nolan	Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen...	2010	148	8.8	1583625	292.57	74.0

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))
]
```

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore
Title											
3 Idiots	431	Comedy,Drama	Two friends are searching for their long lost ...	Rajkumar Hirani	Aamir Khan, Madhavan, Mona Singh, Sharman Joshi	2009	170	8.4	238789	6.52	67.0
The Lives of Others	477	Drama,Thriller	In 1984 East Berlin, an agent of the secret po...	Florian Henckel von Donnersmarck	Ulrich Mühe, Martina Gedeck,Sebastian Koch, Ul...	2006	137	8.5	278103	11.28	89.0
Incendies	714	Drama,Mystery,War	Twins journey to the Middle East to discover t...	Denis Villeneuve	Lubna Azabal, Mélissa Désormeaux-Poulin, Maxim...	2010	131	8.2	92863	6.86	80.0
Taare Zameen Par	992	Drama,Family,Music	An eight-year-old boy is thought to be a lazy ...	Aamir Khan	Darsheel Safary, Aamir Khan, Tanay Chheda, Sac...	2007	165	8.5	102697	1.20	42.0

If you recall up when we used .describe() the 25th percentile for revenue was about 17.4, and we can access this value directly by using the quantile() method with a float of 0.25.

So here we have only four movies that match the criteria.

- **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 these 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)
```

Output:

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore	rating_category
Title												
Guardians of the Galaxy	1	Action,Adventure,Sci-Fi	A group of intergalactic criminals are forced ...	James Gunn	Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...	2014	121	8.1	757074	333.13	76.0	good
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a te...	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...	2012	124	7.0	485820	126.46	65.0	bad

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)
```

Output:

	rank	genre	description	director	actors	year	runtime	rating	votes	revenue_millions	metascore	rating_category
Title												
Guardians of the Galaxy	1	Action,Adventure,Sci-Fi	A group of intergalactic criminals are forced ...	James Gunn	Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...	2014	121	8.1	757074	333.13	76.0	good
Prometheus	2	Adventure,Mystery,Sci-Fi	Following clues to the origin of mankind, a te...	Ridley Scott	Noomi Rapace, Logan Marshall-Green, Michael Fa...	2012	124	7.0	485820	126.46	65.0	bad

Overall, using apply() will be much faster than iterating manually over rows because Pandas is utilizing vectorization.

Reading Material

https://www.w3schools.com/python/pandas/default.asp

https://www.geeksforgeeks.org/introduction-to-pandas-in-python/

https://www.datacamp.com/tutorial/pandas