What is Pandas?

Pandas is a popular Python package for data science, and with good reason: it offers powerful, expressive and flexible data structures that make data manipulation and analysis easy, among many other things. The DataFrame is one of these structures.

[pandas] is derived from the term "panel data", an econometrics term for data sets that include observations over multiple time periods for the same individuals. — Wikipedia

Where Pandas can be used?

- 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

Pandas is built on top of the NumPy package, meaning a lot of the structure of NumPy is used or replicated in Pandas.

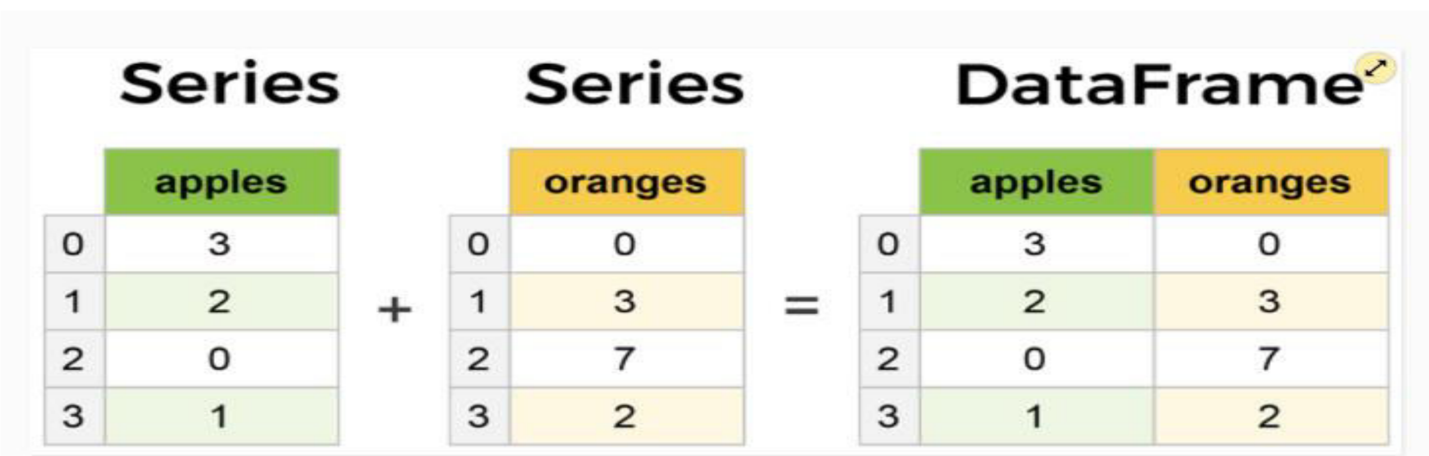
To import pandas we usually import it with a shorter name since it's used so much:

```
In [1]: import pandas as pd
```

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**.



Creating DataFrames:

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:

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

And then pass it to the pandas DataFrame constructor.

In [3]:
```
purchases = pd.DataFrame(data)
purchases
```

Out[3]:

	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:

In [4]:
```
purchases = pd.DataFrame(data, index=['June', 'Robert', 'Lily', 'David'])
purchases
```

Out[4]:

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

So now we could locate a customer's order by using their name:

In [5]:
```
purchases.loc['June']
```

Out[5]:
```
apples     3
oranges    0
Name: June, dtype: int64
```

Reading data from CSVs:

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

In [6]:
```
df = pd.read_csv('/home/saif/LFS/datasets/movies.csv')
df
```

Out[6]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
...	...	...	...

Reading data from JSON:

If you have a JSON file — which is essentially a stored Python dict — pandas can read this just as easily:

In [10]:
```
df = pd.read_json('/home/saif/LFS/datasets/dhoni.json')
df
```

Out[10]:

	title	relase_year	actors	is_awesome	won_oscar
0	The Untold Story	2016	Sushant Singh Rajput	true	false
1	The Untold Story	2016	Kiara Advani	true	false
2	The Untold Story	2016	Disha Patani	true	false
3	The Untold Story	2016	MS Dhoni	true	false

Notes: Pandas will try to figure out how to create a DataFrame by analysing structure of your JSON, and sometimes it doesn't get it right. Often you'll need to set the orient keyword argument depending on the structure, so check out read_json docs about that argument to see which orientation you're using.

Reading data from a MySQL 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.

To connect MySQL using pandas, need to install package 'mysql-connector-python' as below command:
pip install mysql-connector-python

```
In [11]: import mysql.connector as connection
         import pandas as pd
```

```
In [16]: try:
             mydb = connection.connect(host="localhost", database = 'retail_db',
                                       user="root", passwd="Welcome@123", use_pure=True)
             query = "show tables;"
             result_dataFrame = pd.read_sql(query,mydb)
             mydb.close() #close the connection
         except Exception as e:
             mydb.close()
             print(str(e))
```

mysql.connector provides all the database manipulation using python.

Syntax:
connection.connect(host, database, user, password,use_pure)

a) host: provides the hostname of MySQL server. Normally, if we do install in our machine locally then it termed as 'localhost'. Cases like cloud / dedicated third party server provide the IP address there.

b) database: Provides the name of the database to do manipulation.

c) user & password: The credentials to access the database. Normally all database having the credentials set up to make it secure access.

d) use_pure: Symbolize Python implementation

e) ***pandas.read_sql(sql, con):*** Read SQL query or database table into a DataFrame.

- **sql:** SQL query to be executed or a table name.
- **con:** Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable.

The data frame reference holds the result of the SQL query:

```
In [17]: result_dataFrame.head()
```

Out[17]:

	Tables_in_retail_db
0	categories
1	customers
2	departments
3	order_items
4	orders

Exporting dataset to CSV:

We do export the table data to CSV format as well

```
In [24]: result_dataFrame.to_csv('/home/saif/LFS/datasets/pandas_categories.csv')
```

Verify Data:

```
saif@smidsy-technologies:~/LFS/datasets$ head -5 pandas_categories.csv
,category_id,category_department_id,category_name
0,1,2,Football
1,2,2,Soccer
2,3,2,Baseball & Softball
3,4,2,Basketball
```

Reading data from a Data Frame using SQL Query:

Install Package: pip install pandasql

```
In [31]: import pandas as pd
         import pandasql as ps
```

```
In [32]: movies_df = pd.read_csv('/home/saif/LFS/datasets/movies.csv')
```

```
In [35]: ps.sqldf("select * from movies_df")
```

Out[35]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
...	...	...	...

```
In [36]: from pandasql import sqldf
         output = sqldf("select * from movies_df")
         output
```

Out[36]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
...	...	...	...

DataFrame operations:

DataFrames possess hundreds of methods and other operations that are crucial to any analysis.

```
In [3]: movies_df = pd.read_csv('/home/saif/LFS/datasets/movies.csv')
```

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 [4]: movies_df.head()
```

Out[4]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

.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.head(10)
```

Out[6]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy
5	6	Heat (1995)	Action|Crime|Thriller
6	7	Sabrina (1995)	Comedy|Romance
7	8	Tom and Huck (1995)	Adventure|Children
8	9	Sudden Death (1995)	Action
9	10	GoldenEye (1995)	Action|Adventure|Thriller

```
In [7]: movies_df.tail()
```

Out[7]:

	movieId	title	genres
9737	193581	Black Butler: Book of the Atlantic (2017)	Action|Animation|Comedy|Fantasy
9738	193583	No Game No Life: Zero (2017)	Animation|Comedy|Fantasy
9739	193585	Flint (2017)	Drama
9740	193587	Bungo Stray Dogs: Dead Apple (2018)	Action|Animation
9741	193609	Andrew Dice Clay: Dice Rules (1991)	Comedy

movies_db.tail (10) → **Output 10 rows of data**

Getting info about your data:

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

```
In [8]: movies_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   movieId  9742 non-null   int64
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
dtypes: int64(1), object(2)
memory usage: 228.5+ 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):

```
In [9]: movies_df.shape
Out[9]: (9742, 3)
```

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.

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:

```
In [10]: temp_df = movies_df.append(movies_df)
         temp_df.shape
Out[10]: (19484, 3)
```

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:

```
In [11]: temp_df = temp_df.drop_duplicates()
         temp_df.shape
Out[11]: (9742, 3)
```

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 9742 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:

```
In [12]: 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:

```
In [14]: temp_df = movies_df.append(movies_df)  # make a new copy
         temp_df.drop_duplicates(inplace=True, keep=False)
         temp_df.shape

Out[14]: (0, 3)
```

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 Clean-up:

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 [15]: movies_df.columns
Out[15]: Index(['movieId', 'title', 'genres'], 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 [19]: movies_df.rename(columns={
                 'movieId': 'Movie_Id',
                 'title': 'Title',
                 'genres': 'Genres'
             }, inplace=True)


         movies_df.columns
Out[19]: Index(['Movie_Id', 'Title', 'Genres'], dtype='object')
```

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:

```
In [20]: movies_df.columns = ['movied_id', 'title', 'genres']
         movies_df.columns
Out[20]: Index(['movied_id', 'title', 'genres'], dtype='object')
```

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

```
In [22]: movies_df.columns = [col.lower() for col in movies_df]
         movies_df.columns
Out[22]: Index(['movieid', 'title', 'genres'], 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:

- Get rid of rows or columns with nulls
- 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 [23]: movies_df.isnull()
```

Out[23]:

	movieid	title	genres
0	False	False	False
1	False	False	False
2	False	False	False
3	False	False	False
4	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:

```
In [24]: movies_df.isnull().sum()

Out[24]: movieid    0
         title      0
         genres     0
         dtype: int64
```

Removing null values:

Data Scientists and Analysts regularly face the dilemma of dropping or inputting 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 [29]: pd_movies_df = pd.read_csv('/home/saif/LFS/datasets/moviespd.csv')

In [30]: dropNull = pd_movies_df.dropna()

In [33]: dropNull.shape

Out[33]: (9740, 3)
```

```
In [34]: pd_movies_df.shape

Out[34]: (9742, 3)
```

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.

Let's fill the nulls using fillna ( ):

```
In [58]: pd_movies_df['title'].fillna('SAIF', inplace=True)

In [59]: pd_movies_df

Out[59]:
```

	movieId	title	genres
0	1.0	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2.0	Jumanji (1995)	Adventure|Children|Fantasy
2	NaN	Grumpier Old Men (1995)	Comedy|Romance
3	4.0	SAIF	Comedy|Drama|Romance
4	5.0	Father of the Bride Part II (1995)	Comedy

Understanding your variables:

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

```
In [60]: movies_df.describe()
Out[60]:
```

	movieid
count	9742.000000
mean	42200.353623
std	52160.494854
min	1.000000
25%	3248.250000
50%	7300.000000
75%	76232.000000
max	193609.000000

.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 [62]: movies_df['genres'].describe()
Out[62]: count      9742
         unique      951
         top       Drama
         freq       1053
         Name: genres, dtype: object
```

This tells us that the genre column has 951 unique values, the top value is Drama, which shows up 1053 times (freq).

.value_counts() can tell us the frequency of all values in a column:

```
In [64]: movies_df['genres'].value_counts().head(10)
Out[64]: Drama                   1053
         Comedy                   946
         Comedy|Drama             435
         Comedy|Romance           363
         Drama|Romance            349
         Documentary              339
         Comedy|Drama|Romance     276
         Drama|Thriller           168
         Horror                   167
         Horror|Thriller          135
         Name: genres, dtype: int64
```

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 [65]: genre_col = movies_df['genres']
         type(genre_col)

Out[65]: 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:

```
In [66]: genre_col = movies_df[['genres']]
         type(genre_col)

Out[66]: pandas.core.frame.DataFrame
```

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

```
In [68]: subset = movies_df[['genres', 'title']]
         subset.head()

Out[68]:
```

	genres	title
0	Adventure|Animation|Children|Comedy|Fantasy	Toy Story (1995)
1	Adventure|Children|Fantasy	Jumanji (1995)
2	Comedy|Romance	Grumpier Old Men (1995)
3	Comedy|Drama|Romance	Waiting to Exhale (1995)
4	Comedy	Father of the Bride Part II (1995)

By Rows: For rows, we have two options:

- .loc - locates by name
- .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:

```
In [73]: com = movies_df.loc[0]
         com

Out[73]: movieid                                                1
         title                                   Toy Story (1995)
         genres     Adventure|Animation|Children|Comedy|Fantasy
         Name: 0, dtype: object
```

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

```
In [75]: com1 = movies_df.iloc[0]
         com1

Out[75]: movieid                                                1
         title                                   Toy Story (1995)
         genres     Adventure|Animation|Children|Comedy|Fantasy
         Name: 0, 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. In Python, just slice with brackets like example_list[1:3]. It's works the same way in pandas:

```
In [76]: movie_subset = movies_df.iloc[1:3]
         movie_subset

Out[76]:
```

	movieid	title	genres
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance

Conditional selections:

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

```
In [77]: condition = (movies_df['genres'] == "Animation")
         condition.head()

Out[77]: 0    False
         1    False
         2    False
         3    False
         4    False
         Name: genres, dtype: bool
```

```
In [78]: movies_df[movies_df['genres'] == "Animation"]
```

Out[78]:

	movieid	title	genres
6973	66335	Afro Samurai: Resurrection (2009)	Animation
7059	69469	Garfield's Pet Force (2009)	Animation
7195	72603	Merry Madagascar (2009)	Animation
7279	74791	Town Called Panic, A (Panique au village) (2009)	Animation
7439	81018	Illusionist, The (L'illusionniste) (2010)	Animation

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

```
In [79]: movies_df[(movies_df['genres'] == 'Animation') | (movies_df['genres'] == 'comedy')].head()
```

Out[79]:

	movieid	title	genres
6973	66335	Afro Samurai: Resurrection (2009)	Animation
7059	69469	Garfield's Pet Force (2009)	Animation
7195	72603	Merry Madagascar (2009)	Animation
7279	74791	Town Called Panic, A (Panique au village) (2009)	Animation
7439	81018	Illusionist, The (L'illusionniste) (2010)	Animation

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

```
In [80]: movies_df[movies_df['genres'].isin(['Animation', 'Comedy'])].head()
```

Out[80]:

	movieid	title	genres
4	5	Father of the Bride Part II (1995)	Comedy
17	18	Four Rooms (1995)	Comedy
18	19	Ace Ventura: When Nature Calls (1995)	Comedy
58	65	Bio-Dome (1996)	Comedy
61	69	Friday (1995)	Comedy