Pandas

# Introduction: -

# Introduction: -

- **Pandas is a very fast, flexible and powerful data manipulation tool** that is very popular in data science community.
- With the help of pandas, we will be able to do really complex data manipulation with very little code and in record time.
- **Pandas is a python library** and we need to work in python environment for our panda's code to execute. To work with panda's core, we need access to a python interpreter.
- Pandas frequently rely on some other packages such as **numpy** that offers some powerful array manipulation computations.
- The plotting functionality in pandas by default rely on **matplotlib**.
- Pandas in not solo and it is a collection of data science tools.
- Python is a general-purpose programming language that emphasizes code readability and supports many different programming paradigms.
- Our computers only understand binary language 0's and 1's.

# Cloud vs local: -

- So far, we do coding in jupyter notebook environment in anaconda distribution.
- But the anaconda distribution requires a lot of system resources, quite a bit of disk space, plenty of CPU and GPU power as well.
- Now days even better option than this local installation is to use a cloud hosted data science environment.
- And there are many alternatives out there such as AWS offers sage maker, Microsoft azure offers its own cloud hosted jupyter notebook, IBM Watson studio, google Collaboratory etc.
- **Google Collab** is most used for data science. Using **Collab,** we can easily run python code from within our browser without needing to worry about configuration, local installation etc.

- Collab allows us to write and execute Python in your browser, with
    - Zero configuration required
    - Access to GPUs free of charge
    - Easy sharing
- **pd.__version__** → it is used to check the version of pandas.
- Jupyter notebook is simply an interactive computing environment that allow us to create, execute, and share code in a web-based environment.
- To import pandas, we do like **import pandas as pd** in jupyter notebook.

# Numpy: -

- **Numpy is short form of numerical python**. It is one of the most important scientific computing libraries in python.
- It is important to learn numpy because BTS **pandas rely on numpy** for its own data structures.
- The fundamental data structure in **numpy is the n dimensional array**, which is **simply a collection of items** very much similar to a python list.
- Applying operations or transformations to numpy array is much quicker.
- Numpy array offers wide range of mathematical functions that are unavailable in pure python those are known as **universal functions or ufuncs** which we can directly apply on our numpy array.
- Another benefit of working with **numpy is its sheer speed**, these are much more efficient that python list.
- **Numpy stores the data in contiguous blocks of memory**. All the items in the numpy memory are right next to each other.
- While in python lists instead of the data itself, **python lists store pointers** to that data.
- The ability to store the data in contiguous block of memory is result of a **trade-off that within a given numpy array, we could only store one type of data**.
- Both the libraries numpy and pandas go hand in hand with each other.
- We use numpy to create, transform and manipulate tabular data.

# Series at glance: -

# What is a series: -

- **Series** is one of the key data structures in panda's library.
- **Series** are **1D labelled arrays** of any data type. Or we can say series are **sequence of values with associated labels**.
- **pd.Series()** → In order to generate a pandas series we use this function and it will create a pandas series with the list of values and the associated labels. And these values can be anything numbers, Booleans, strings etc.
- Unlike numpy arrays, **panda series supports mixed data types** like **[True, 'say', {'my_mood':100}]**. This would not for numpy arrays because the data type of the elements of the numpy arrays should be same.

# Parameters Vs arguments: -

- We are passing a python list directly in the Series constructor like **pd.Series(students).**
- We are passing the **python list as an argument** to the **data parameter of Series** in pandas and **pd.Series(students)** this is equivalent to saying **pd.Series(data = students).**
- Where **data is the parameter** which are the variables in the method definition. Each method, constructor has its own set of parameters that it accepts.
- **Arguments are the actual values** that we pass in, that we associate with those parameters.
- **pd.Series(data = students)** here the data is the parameter and students is the arguments that we pass to the parameter.

# What's in the data: -

- Python lists, like arrays in other languages is an ordered data structure. Each of the items in the list are associated with an index that specifies its exact location in the object.
- **.equals()** is the method used to compare the content and gives the Boolean output.
- We can also create the series like **pd.Series(23)** or **pd.series("Hello")**.

- Pandas in not critically dependent on labels or implied labels to be provided as inputs. It’s just fall back to integer sequences like 0 when it doesn’t have anything.
- Irrespective of the data we passed in the panda’s series, **it will generate 0 based indices.**

# .dtype attribute: -

- We see that **pandas will automatically infer the dtype** for the series from the data that we provided to the series constructor.
- But we also have the option to specify the type itself.
- We can specify the data type using the **dtype attribute** like **pd.Series(ages, dtype = ‘float’)**. It will give the data in that type only which we specified.
- dtype for a series containing a **string becomes object** automatically. This will always be the case for any series that contains a string.
- **str_series.dtype** → this will give the data type as (‘o’) for strings. Since strings are of variable length and numpy didn’t know how much memory they require in advance.so instead of saving the actual strings they save the pointer or a reference to the string object in the memory.
- Numpy arrays are homogenous, means they contains the same size data.

# Index and RangeIndex: -

- One of the important features of the panda’s series is that the **data aligns automatically by the labels or the index** as it’s known.
- When we don’t specify our own sequence, pandas assign a sequence of integers starting from 0.
- **Index** is a sequence of numbers staring from 0 and ends till 1-length of the series and **Panda series** automatically assigns them.
- But we can get more specific and provide our own labels by using the **index=[]** parameter in series. We can definitely provide a custom index to our series.
- **pd.Series(data=book_list, index=['funny', 'serious and amusing', 'kinda interesting'])** → this is the syntax for giving **labels by ourself using the index=[] parameter**.

- **Keyword arguments** follow the **positional arguments** but vice-versa in not true in the Series data type.
- **RangeIndex** is just a built-in object **that creates a sequence of integers** with fixed differences specified in this step parameter.
- We can also create our own rangeIndex as **pd.RangeIndex(start, stop(exclusive), step)**.

# Series and index names: -

- Series is just **1D sequence of values with just associated axis labels**.
- **Attribute** is like the property/variable of the object e.g. **books_series.size** here the .size will be the attribute and it will give the size of the Series. **Method** is an action/function applied to the object e.g. **list_s.equals(dict_s)** here .equals() is the method.
- Another attribute in python pandas in **.name** and it is pointing to the **none object** in the python.
- We can use the **.name** attribute like **book_series.name = "My favourite books"** and it is **used to give name to the Series** in the output. We use this because later when we learn data frame then the name of the Series will become the column name in the data frame.
- Series can have names and those becomes column names in the data frame later.
- We can also assign name to the Series index like **book_series.index.name = "My books"**.
- **dict(zip(actor_names, actor_ages))** → we can create a dictionary dynamically like this with the help of lists by using the **zip()** method.
- **{name:age for name,age in zip(actor_names, actor_ages)}** → this is how we use **dictionary comprehension** and using this we can transform and iterate dictionaries without using the for or while loops.
- The **zip()** function **returns a zip object**, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.

# The head() and tail() methods: -

- These are the convenient methods for **exploring our data**.
- Sometimes we want to explore the structure of the series without producing all of it or without seeing all the data and that's where the head() and tail() method come in.
- **head()** method is used to get the sneak peak of **first five records** of our series. **int_series.head()** this is how we use the head() method. We can also specify how many records we would like to see like **int_series.head(3)** and it will give us top 3 records.
- **tail()** method is used to get the bottom/last five records of our series. **int_series.tail()** this is how we use the tail method. **int_series.tail(3)**.
- **pd.options.display.min_rows = 40** this function is used to tell pandas to print the minimum of 40 rows every time when we load the Series.

# Extraction by index positions: -

- Here we will talk about **accessing and extracting** values from the Series.
- **from string import ascii_lowercase** this module is used to import all the lowercase letters.
- To access the elements from the **Series we use the [] square brackets** only like we use them in accessing the elements of the list. Alphabets[0] and it will give us the 1st element from the list.
- we can access the elements from the Series like **[start:stop(not inc):step]** like we did in python lists and it is known as **slicing**.
- If our labels are not **the default integer-based labels** and we give labels to the Series by ourself then then also this indexing works the same way. But in addition, we can also extract values with the help of our custom labels.

# Accessing elements by label: -

- We saw how to **access items from a series using index positions**, now we will see if we can do the same when the index is labelled.
- We can access the items from the Series using the label names also like **alphabet_labels['label_A']** and this will give us the corresponding value of that label.

- **iloc[]** is an indexer used for **integer-location-based indexing** of data in a DataFrame. It allows users to select specific rows and columns **by providing integer indices, making it a valuable tool for data manipulation and extraction based on numerical positions** within the DataFrame.
- We can also do slicing using the labels like **labeled_aplha[:'label_C']** here the label_C is included in the output.
- Position and label based indexing is available throught the same [ ] square bracket syntax.

# The add_prefix() and add_suffix() methods: -

- We can directly **add the prefix and suffix to the labels** in the panda's series easily
- To do this we have these functions i.e. **add_prefix()** and **add_suffix().**
- **alphabets.add_prefix('label_')** this is how we use these.
- To add the string to the end of the label we have **add_suffix()** method.
- **alphabets.add_suffix('label_')** this will add the suffix label_ at the end of the label.
- These methods work for both **dataframes as well as series**.
- These methods don't actually modify the alphabet series, instead we get the copies after these methods are applied.

# using Dot notation: -

- The alternative way to extract values is kind of an alias for square **bracket [ ] indexing.**
- **labeled_aplha['label_V']** till now we use this syntax of square bracket to access the values from the Series.
- But whenever we want to access the value for a specific label from a panda Series, we could also just use the **Dot(.) notation** as if we were accessing the attributes of the Series. **We use this notation to extracting using the labels only**.
- We can access the values as **labeled_alpha.label_V** this will give us the same result as the above syntax gives.
- The limitations of this notations are: -
    - We cannot do slicing using this notation.

# Boolean Masks and the .loc[] indexer: -

- We can also extract from the series using **the .loc[] and .iloc[]** attributes or **indexers as they are known**.
- The syntax to use the .loc[] attribute for **accessing the elements/typical way of label based extraction** is **labeled_aplha['label_I':'label_M'] .loc[] is a prototypical way to do label based extraction** whereas the square brackets supports a mix of syntaxes and functions.
- Both **loc[]** and regular Series slicing **using [ ] supports** Boolean arrays or Boolean masks.
- **book_series.loc[[True, True, False]]** this is the concept of Boolean masks. **The length of the mask should be same as the length of the series**.
- It will output the values which has True value corresponding to it and skips the False values.
- If we have thousands of values then we generate the Boolean masks programmatically and then we pass them to the loc[] attribute.
- **labeled_aplha.loc[[True for i in range(26)]]** this is how we use the Boolean mask for large values.
- **Boolean masks** are used to index and select items at scale and it works with **both [ ] and .loc[ ].** They need to be the same length as series.

# Extracting by position with .iloc: -

- The **.iloc[] or .integerloc[]** attribute is similar to **.loc[]** attribute but it is **used for indexing by position** instead of indexing by labels using loc[].
- **labeled_aplha.iloc[0]** this is how the .iloc[] works. Indexing with .iloc[] is also 0 based.
- We can also do slicing using the .iloc[] attribute like **labeled_aplha.iloc[2:5]** this.

# Selecting with .get(): -

- This is the another way to extract values from the series i.e. **.get() method**.
- It is a pretty straight forward method **labeled_aplha.get('label_V')** we simply pass a label in it and get the corresponding value.
- It is very similar to the **.loc[] indexer** and the **[ ] bracket** notation for indexing.

- But the purpose **to use .get()** is some conveniences it offer: -
    - If we try to access the label that doesn’t exist then it doesn’t return anything but NONE because it has a default parameter which is set to NONE by default. We can also specify some custom default value to it.
    - On the other hand **the .loc[] and the [ ] square bracket** method gives the error if the label doesn’t exist.
    - With the same syntax we can access by **index as well as label** also like **labeled_aplha.get(8**), **labeled_aplha.get('label_Vin', default=19).**

# Using callables with .loc and .iloc: -

- Another way to select and extract values from the series is indexing with callables.
- Indexing with callables is used for highly customizable indexing and it works with [ ], .loc, .iloc.
- A callable is just an object that accepts some argument and possibly returns something.
- When indexing with callables in pandas, we built a function that takes a series or dataframe as an input and it will produce some indexing output.
- The valid indexing output are: -
    - A list of labels
    - List of Booleans
    - A slice etc..

# Series Methods and handling: -

# Reading in Data with read_csv(): -

- **CSV → comma separated values** and it is a type of file format which contains values delimited/ separated by commas.
- **CSV** file is just a text file with values separated by commas.
- **pandas.read_csv() →** To bring the CSV content to pandas we use this method.
- **pd.read_csv('D:/Numpy & Pandas/drinks.csv') →** this is how we read the CSV files.
- pd.read_csv('D:/Numpy & Pandas/drinks.csv', **usecols=['country', 'wine_servings']**) → the **usecols** parameter is used to get the particular columns from a CSV file.
- pd.read_csv('D:/Numpy & Pandas/ drinks.csv', usecols=['country', 'wine_servings'], **index_col='country'**) → the **index_col** parameter is used to set the **index name/ labels** for the series.
- The read.csv() method **returns a data frame** regardless the number of columns.
- If we are working with single column data frame then **squeeze that data frame into series** using the .**squeeze()** function.
- alcohol = pd.read_csv('D:/Numpy & Pandas/drinks.csv', usecols=['country', 'wine_servings'], index_col='country')**.squeeze()** → this is how we use the .squeeze() function.

# Series sizing with .size, .shape and len(): -

- Like other python objects, **Series have methods and attributes**. Attributes give us the properties related to the Series whereas methods perform an action.
- **.size →** this attribute will give us the length of the Series/ number of elements in the series (unique + null) values.
- **.shape →** it returns a tuple, **specifying the dimension of the Series** which is by default 1D.
- **len() →** this function is also used to get the size/ length of the Series.

# Unique values and Series Monotonicity: -

- Some of the other Series attributes are: -
  - **.is_unique** → It simply checks of the Series contains the sequence of unique values.
  - **.nunique()** → this method gives the number of unique values in the Series.
  - **.is_monotonic_increasing/ .is_monotonic** → the name monotonicity comes from **ordered theory in maths**. And it means that the function preserves the given order. In pandas Series context, this attribute will return true if the series values are evolving in a given direction e.g. always increasing or always decreasing.
  - **.is_monotonic_decreasing**

# The **count()** method: -

- Pandas has also has a **count()** method and we use the method to count the values of the Series.
- The count() method only give the **count of the values that are NOT NULL/NA**.
- The difference in the values of **count()** and **.size** indicates that our data **contains the NULL values** and confirms **that there are gaps in the data**.
- **NULL indicates the absence of the info**.
- **.hasnans** → this attributes is used to check that Series will contain null data or not and give us the Boolean output.

# Accessing and Counting NAs: -

- NULL or NA indicate the absence of the data.
- **.isnull()/.isna()** → returns Booleans indicating the value corresponding to it is NULL or not.
- **alcohol[alcohol.isnull()]** → this is how we get the countries where the data is missing or NULL or NA.
- This is how we can identify and isolate the Series records that have the null values.
- Booleans in python works as an integer. True = 1 and False = 0. And the **sum()** method is used to count the True and False values in the Series,

- **.isnull().sum()/ .isna().sum() →** this is how we **count the number of NULL values** from the Series using the **sum()** method.
- Another approach to isolate nulls which enables to use the numpy universal function.
- **Ufuncs allows us to conduct vectorized operations** and Vectorization is the process of going from applying computations to each element to running them on the entire collection of them at once.
- **Vectorization** refers to running operations on the entire arrays.
- The numpy **ufunc .isnan attribute** does an element by element check for nulls and returns a sequence of Booleans.
- **alcohol[np.isnan].size →** this is also a way to find the number of NULL values in a Series using numpy ufunc.

# The other side: notnull() and notna(): -

- **.notnull()/ .notna() →** this method returns a series of Booleans indicating whether each record is null or not and the True identifies the NOT NULL.
- The count of **null values** + the count of **not null values** should be equal to the total values in the Series.

# Booleans are literally numbers in python: -

- We treat **True → 1 and False → 0**.
- From python3 the True and False are just keywords for numbers 1 and 0 respectively.
- The Boolean class in python inherits or is a subclass of the integer class.
- Boolean is a type of integer → which further is a type of object.
- **bool.__mro__ →** this is how we check the method resolution order for a particular class.

#Dropping and Filling NAs: -

- If we want to **exclude Nas from our series** that's where the **.dropna()** method comes in.
- It simply **excludes the NAs** and **returns the new series**.
- We use the **dropna** method like **alcohol.dropna()**.
- This method **doesn't change the original series** but it **creates and returns copy of the original series without modifying the original one**.
- If we want to modify the original series then we can use the **inplace=True** parameter in **.dropna()** method like **alcohol.dropna(inplace=True)**.
- Dropping NAs is one way to deal with them. Another approach **is to replace them with something** that we find more meaningful.
- To do this in pandas we rely on **.fillna()** method and the dynamics is similar that a new copy of the series is returned by default.
- We have to specify what value we want the names to be replaced with, this is how we use the **.fillna()** method **alcohol.fillna(100, inplace=False)**.
- Both the methods **.dropna()** and **.fillna()** returns a copy of original series unitll we don't specify **inplace = True** parameter.

# Descriptive statistics: -

- **These are the metrices that allow us to characterize or describe our data**.
- Pandas has a mean method which calculates the average of not nulls using the **.mean()** method and we use it like **alcohol.mean()**.
- The median is the middle most number in the **sorted list of number** and we can also find it using the **.median()** function in pandas like **alcohol.median()** also **alcohol.quantile(.5)** gives the same result as median. The median is the middle quantile.
- To find the inter quantile range we find the difference between the 1st and the 3rd quantile like **iqr = alcohol.quantile(.75) - alcohol.quantile(.25)**.
- We can also find the min and max using the **.min() and .max() function.**
- We can also find the standard deviation using the function **.std()** like **alcohol.std()**.
- In pandas the variance is calculated using **.var()** like **alcohol.var()** which is the square root of standard deviation.

# The describe() method: -

- Almost everything we calculated previously manually can be obtained by using a single method and is enough using the **.describe() method**.
- We use this method like **alcohol.describe()**.
- This method returns a pandas series **containing descriptive statistics for our data**. It gives us a really good numerical sense of what we're working with.
- **alcohol.describe(percentiles=[.79, .19])** we can also give the parameter to the describe() method like this.
- **alcohol.describe(percentiles=[.79, .19], include=float, exclude=object)** we can also give include and exclude parameters to this method which tells what data type we want to include and what to exclude.

# mode() and value_counts(): -

- The **mode()** is one of the **descriptive statistics** in pandas. The **mode gets us the most common item**, in other words the item that occurs most frequently in the collection of values.
- If we have a distribution, the mode is the item that shows up the most frequently.
- The **median** separates the dataset into two equal sized parts and the **mean** is simply the average value which we calculate as **sum of all the observations/count of all observations**.
- We calculate the mode using **.mode()** method like **alcohol.mode()**.
- To know that how many (time/count) the most common value occurring the set we use **.value_count()** method.
- The **.value_count()** gives the count/frequency of each individual value in the series. It provides a sorted series containing unique values and their counts.
- We can also set many parameters inside the **.value_count(sort = True, ascending = False, dropna = True, normalize = False)** like this.
- By setting the **normalize = True**, we will normalize these counts into relative frequencies. So, we don't get the absolute count instead of this **we get relative frequencies**.

# idxmax() and idxmin(): -

- Suppose we want to figure out the name of the country that had the most wine servings in 2010. By using the **.max()** method we get value of max wine serving value but we want to find the corresponding country name.
- This is how we use the **.index()** function to get the max alcohol consuming country **alcohol[alcohol == alcohol.max()].index[0]**.
- **alcohol.idxmax() the .idxmax()** method do the same work like above and it gives the first label with the associated max() value in series. It counts the series alphabetically. Returns the label of the row with minimum value.
- **alcohol.idxmin() the .idxmin()** method do the same work like above and it gives the label with the associated min() value in series. Returns the label of the row with maximum value.
- If there are multiple min/max values, only the 1st label is returned.

# Sorting with sort_values(): -

- When it comes to sorting, we have two options: -
  - **Sort by values**
  - **Sort by index**
- We can sort the values like **alcohol.sort_values(ascending, inplace, na_position, kind)** and it will sort the values in the ascending order and if we want to sort the values in descending order then **alcohol.sort_values(ascending=False)**.
- We can also set the position of NA values like **alcohol.sort_values(ascending=False, na_position = ‘first’)** and this will sort the NA values to the start.
- We can also specify which sorting algorithm we want to use like **alcohol.sort_values(ascending=False, na_position='last', kind='mergesort')** and we can specify other options in it like quicksort, heapsort etc.
- **Sorting the values will return a copy of the series**, it returns a new series where the values are sorted without changing the original data set.
- If we want to change the original series then we have two methods: -
  - Reassignment → **alcohol = alcohol.sort_values()**.
  - Using inplace parameter → **alcohol.sort_values(inplace=True)**.
- Inplace parameter serves the same function of modifying the original series without creating or returning a copy of it.

# nlargest() and nsmallest(): -

- Suppose if we want to know the largest 10 numbers in the data set or the smallest 10 numbers in the dataset then we can do this with the help of these functions.
- We use this functions like **alcohol.nlargest(10)** and **alcohol.nsmallest(10)**.

# Sorting with sort_index(): -

- We can also sort the values based on the index literals like the **sort_values() method**.
- We can sort the series based on the index like **alcohol.sort_index()** and this will sort the series index alphabetically in ascending order.
- We can also set the ascending parameter to sort the series in descending like **alcohol.sort_index(ascending=False)**.
- All the other parameters are similar to the **sort_values()** method.

# Series Arithmatics and fill_value(): -

- One of the greatest features of Pandas data structure **is that the indices align automatically**.
- The **fill_value = 0** parameter guarantee that all unique labels that show up in one or the other series are reflected in the final result.
- **alcohol.add(more_drinks, fill_value=0)** this is how we use the Arithmatic **add()** method in Series to add up the two series by making sure that their content remains unchanged after addition.
- The **fill_value** parameter ensures that not to lose the data due to series misalignment.
- In panda Series we don’t have to worry about aligning the data manually before we operate on it.

# Calculating variance and standard deviation: -

- The variance is the average of squared differences from the mean. The formula of variance is: -

$$\boxed{s^2 = \frac{\sum (x_i - \bar{x})^2}{n - 1}}$$

- This formula is explaining that for each observation, find its difference relative to the mean and then square that difference and find the sum of all of those differences for each observation and at the end divide the sum of squared differences by the number of observations – 1.
- The sample standard deviation is the square root of the variance.

# Cumulative operations: -

- The operations that accumulate the output within the series are: -
  - **.sum()** → it gives us the total of all the values in the series.
  - **.cumsum()** → it is used to calculate the progressive or cumulative sum. These methods skip the NA values in the series in the calculation because there is a parameter **skipna=True** which is by default in these series.
  - **.prod()** → this method is used to calculate the product of the series values.
  - **.cumprod()** → this is used to calculate the cumulative product.
  - **.cummin()** → it is used to find the cumulative min from the Series.
  - **.cummax()** → it is used to find the cumulative max from the series.

# Pairwise differences with diff(): -

- This method is mainly **useful when we are working with time Series**.
- The **diff()** method is used to calculate the discrete difference for pairs of elements in a series so that would be a difference between one element and the another.
- **alcohol.diff(periods=2).head()** → this is how we set the periods parameter and it will calculate the diff with the difference of 2.
- The **.diff()** method find the first discrete element wise difference in a series.

- For eg: -
  - a v1
  - b v2
  - c v3
- .diff(periods=1) → v2 – v1 then v3 – v2
- .diff(periods=-1) → v1 – v2 the v2 – v3
- .diff(periods=2) → v3 – v1 then v4 – v2

# Series iteration: -

- Occasionally we might want to iterate over each element in a series and there is couple of ways to do that.
- We can use the basic python for loop to **iterate over the values of the** Series like: -
  - **for i in mini_alc:**
    print(i)
- To iterate over the pair of indices and values pandas provide much more convenient alternative i.e. **.items() / .iteritems method**. This method will **return the pair of tuples of index with their values** and we can do tuple unpacking here.

# Filtering: filter(), where() and mask(): -

- We can also filter our series like suppose we only want the countries that start with V.
- We can do this by using the filter method like **alcohol.filter(regex='^V')**. We can **filter our series based on the string pattern**. We use regex to define text/string patterns.
- Apart from regex we can also use the **like parameter. alcohol.filter(like='stan')**.
- Using the **.filter()** method, **we filter based on the indices not the values**.
- The **.where()** method helps us efficiently **replace values where a given condition is false** and it works like **alcohol.where(lambda x: x>200, other ='too small')** If the condition is true then it gives us the values and if the condition is false then it will replace the values with the value we have provided in other.

- If we want to get all the values that will be not true the simply flip the condition **inside .where()** method.
- But in some condition flipping the logical condition is not that easy like in multiple if-else statements so for such cases if we are interested in **filtering values when the condition is false** we use the **.mask()** method.
- The **.mask()** method is **also a replacement method** but unlike the .where() method**, it replaces the values where the condition is true**.
- We use the **.mask()** method like **alcohol.mask(lambda x: x> 200).dropna()**.

# Transforming with update(), apply() and map(): -

- When it comes to transformation in pandas, we could really think of two broad categories: -
    - There are those that target couple of specific records (spot transformation)
    - There are those that target the entire series (global transformation)
- So for **Spot transforms** we can always rely on straight assignments by combining the **equals operator (=)** with the **loc[]** and **iloc[]** indexers.
- If we want to modify the couple of records, then it becomes very repetative then for such instances we use the **.update()** method which **modifies the series inplace** from the nonNA values from another series.
- **alcohol.update(pd.Series(data=[200,20], index=['Albania', 'Algeria']))** → this is how we use the update method.
- If we want to apply the transformation to our entire series then we use the **.apply() method**. The **.apply() method** applies the transformations to each and every element in a series like **alcohol.apply(multiply_by_self)**.
- Another approach to apply the global transformation is to use the **.map()** method. This method is technically used for substituting values like **alcohol.map(lambda x: x**2)**.
- For the basic transformations we can use the **.map()** and **.apply()** method interchangeably. But for complex transformations we cannot use the **.map() method**.
- **Z-score** is the number of deviations that a given observation is deviating or being different from the sample mean.
- **z_score = (beers - beers.mean()) / beers.std()** this is how we calculate the Z-score.

- Standard normal distribution is a special type of distribution that has a mean = 0 and standard deviation = 1. Standard normal distribution is always centered at 0 and each value on the horizontal axis corresponds to Z-score which tells us how many standard deviations and observations from the mean.
- Z-score allow us to calculate how much area that specific Z-score is associated.

# **Working with DataFrames: -**

## What is a DataFrame: -

- DataFrame is a **table of data that contains a collection of rows and columns**.
- Unlike Series which are 1D, **DataFrames are 2D and they have labelled indices and columns**.
- **Series extends only in one direction** and when working with Series we need only one piece of information specifically to select any value in this range.
- In order to get information from a DataFrame we need to specify the index position as well as column position like **df.iloc[2,0]**.
- Series has labelled indices only but the **DataFrame has labelled indices as well as columns**.
- **Each column in a DataFrame is a Series**. Think of DataFrame as a collection of Series.
- **Unlike Series, DataFrames could be heterogeneous**. DataFrame themselves don't have a data type.
- Using **df.dtypes** we can get the data type of each of the column from the DataFrame.

## Creating a DataFrame: -

- To build a DataFrame, **we could pass a dictionary** to the pandas DataFrame constructor i.e. **pd.DataFrame({"labels":value})** where each label represents the **name of the column** and each value is the **collection of values** for that column.
- All the lists we pass in the DataFrame dictionary **need to be of equal length**.
- Like the Series constructor, the DataFrame constructor also accepts many attributes.

## The info() method: -

- One of the first things to do when preparing to work with a new data frame is to look at a brief summary of what it consists of, what is contains.

- The **info()** method is used for this purpose. This method only works for a DataFrame.
- If we have very large dataset then for that we have two parameters for the info() method i.e. **info(verbose=False)** → it will ensure that no info. Of a specific column is printed and **info(max_cols=)** → it will get us the specified number of columns only.
- **df.info(memory_usage=False)** → this doesn't give us the memory usage column in the information.
- **df.info(memory_usage='deep')** → it will give us the exact memory usage as it refers to deep introspection.

# Some cleanup: Removing the duplicate index: -

- **.drop('column_name', axis = 1)** →To drop a column, we use this method. The **axis parameter controls what we are removing row or a column** and **axis = 1 means we are referencing a column** and **axis = 0 means we are referencing a row**. This method returns a DataFrame by dropping the duplicate column we specified.
- **.set_index('column_name')** → this method is used to set a duplicate column from a DataFrame as an index column.

# The sample() method: -

- **.sample()** → this method we will use to conveniently take random samples from our dataset. We will **call this method on a DataFrame and it returns back a random record**.
- **nutrition.sample(random_state=12)** → the random_state perimeter is used **if we want to get a fixed random data** from a DataFrame.
- **nutrition.sample(n = 3)** → the n parameter is used for how many random records from a DataFrame we want.
- **nutrition.sample(frac = 0.01)** → the frac parameter is used to get the random fraction of values from the DataFrame.

# Sampling with replacement or weights: -

- The **sample()** method could get fancier.
- **Sampling with replacement** means that whenever we sampled records from our population, we replace that record so that the probability of picking the same item again remains unchanged.

# DataFrame axes: -

- DataFrames having 2D means that the number of attributes that we would need to specify in order to select a specific observation in our DataFrame is 2. We need to **provide the row label as well as the corresponding column label**.
- These are known as **DataFrame Axes**. We can access the dimensions using the **.axes** attribute like **nutrition.axes**.
- This returns a python list of 2 pandas index objects, one for row and one for column.
- **nutrition.index[3]** → we get the index number like this and **nutrition.columns[69]** the column name like this also.
- axes = 0 → means rows, and axes = 1 → means columns.

# Changing the index: -

- DataFrame currently consist of **integer-based index**. **.index** this will give us the indexes in our DataFrame.
- **RangeIndex** in Series is a special case of **Int64Index** but RangeIndex is an optimized alternative. The **RangeIndex** allow us to define the entire sequence using a start, step and an end value.
- **.set_index('column_name')** → this is used to set the index to the name rather that integers. It returns a copy of DataFrame and not modify the original dataset. And if we want to modify the DataFrame we set the **inplace=True**.
- **.set_index('column_name', drop=False)** → the drop = False means that add the column as an index as well as keep that in the column also.

- **nutrition.set_index('folic_acid', drop=False, append=True)** → append = True means that along with the already having index, add this index also so in that way we can have multiple index for a DataFrame.
- **nutrition.set_index('folic_acid', drop=False, append=True, verify_integrity=False)** → the verify_integrity allow us to enforce the uniqueness of our index values. Setting it True checks that the index contains unique values and throw a value error if it contains duplicate values.

# Extracting from DataFrames by label: -

- In Series we have looked at many ways to extract records such as loc[] indexer, iloc[] indexer, [] square bracket indexing, direct attribute access using dot (.) notation, get() method etc.
- All of these and some addition methods will work for DataFrames too.
- **nutrition.loc['Eggplant, raw']** → this is how we extract labels in DataFrames using .loc[] indexer.
- **nutrition.loc[‘row_labels’]['column_labels']** or **nutrition.loc['Eggplant, raw','calories']** → this is how we can access a particular value within a row in a DataFrame.
- **nutrition.loc['Eggplant, raw':'Sherbet, orange', 'calories':'cholesterol']** → we can use slicing to grab range of columns and rows.
- **nutrition.loc[['Raspberries, raw'],['protein', 'vitamin_b6']]** → this is how we can access particular rows and columns.

# DataFrame extraction by position: -

- In Series we have used the **.iloc[] indexer** to extract items by position. We can also use the .iloc[] indexer to extract from DataFrames.
- Just like in the Series, the **.iloc[] indexing is 0 based** for DataFrames too.
- **nutrition.iloc[3]** → this is how we use the iloc[] indexing to extract from DataFrames and it will return a panda Series with all the information.
- **nutrition.iloc[3, :]** → this indicates that we want all the columns for the 4th row.
- **nutrition.iloc[[4,6,9], :]** → If we are interested in non-contiguous records then we have to pass a list of integers in **iloc[] indexer**.

- **nutrition.iloc[rows,column]**
- **nutrition.iloc[[4,6,9], 2]** → this is how we extract 2nd column from a DataFrame along with the rows 4,6,9.
- **nutrition.iloc[[4,6,9], 2:5]** → this is how we extract a multiple columns using slicing.
- Both the **iloc[]** and **loc[]** indexer **supports Boolean masks for DataFrames** too and the syntax is similar to that in the Series.
- **nutrition.iloc[**
  [True if i%2==0 else False for i in range(8789)],
  [True if i%2==0 else False for i in range(75)]
  ] → this is how we use Boolean mask in the DataFrames too
- For single value extraction, pandas have some dedicated attributes which are lot more performant and quicker.

# Single value access with .at and .iat: -

- For single value extraction pandas has **.at** and **.iat** attributes which a **.loc** and **.iloc** indexing attributes.
- **nutrition.at['Nuts, pecans', 'calories']** → if we want to **extract a single value by label** then we can use this.
- **nutrition.iat[1,1]** → if we want to **extract a single value by position** then we can use this.
- Unlike .loc[] and .iloc[] the **.at[] and .iat[] are used for accessing single values only** and **they are faster** as compared to them because of the lack of overhead, they are much-more performant for their isolated use-case.

# The get_loc method: -

- Whenever we are **working with position-based extraction** approaches, we may find ourselves counting columns and rows to figure out the exact integer location we're interested in.
- In case we have a label and a position, how do we get two labels and two positions we will see here.
- The **get_loc('column_name')** helps us get an integer location for a given label/ column_name.

# More cleanup: going numeric: -

- Before we perform any meaningful analysis on the nutrition DataFrame, we need to convert all the columns that contain units, which as a result are strings to numeric columns.
- Strings in pandas are of object type.
- So, for performing any operations on the data inside DataFrame we need to convert the string values to numeric values and we do that in couple of steps.

# The **astype()** method: -

- Pandas has a very useful built-in method that helps us cast a given DataFrame or Series from one data type to another.
- **df.astype(float)** → this is how we use the **astype() method** to convert one data type into another.
- The **astype() method** is also flexible in that instead of targeting the entire DataFrame, it also allows us to modify only a subset of the column.
- **df.astype({'age':int})** → this is how we can change the data type of a particular column using the astype() method.
- But the **astype()** method alone cannot helps us to convert the data type of nutrition DataFrame from string to numeric. We first have to get rid of the string part in the values then we can convert the string to numeric using astype() method.

# DataFrame replace() + a glimpse at regex: -

- The **replace()** method replaces one thing with another.
- **dfm.replace(to_replace='100 g', value=100)** → this is how we use the replace() method **to_replace** one value with another **value**.
- The way we use **replace()** is very nitpicky as it depends on exact string matching.
- There is more flexible way to deal with string replacements that involves **regex (regular expressions)** and these are used to define the text patterns.

- **dfm.replace('\sg','',regex=True)** → this is how we use regex with the replace method to replace a string with some other info.
- **dfm.replace('\sg','',regex=True).astype(float)** → this is how we convert the string values to float values.

# Part I: Collecting the units: -

- It's never the case where we get the perfect quality data right from the start, there will be a lot of cleanup and transformation involved.
- So the first step/methods is to isolate the numeric units in the data set and separate them from strings by using the **.replace(to_replace=' ', value=, regex=True)** method.
- Anything that's not a unit needs to go.
- In regex we can use character sets, a character set allows us to match one of several characters and in regex they are **defined using square brackets [A-Za-Z] and within these brackets** we will specify those range of characters.
- When we apply mode to a series, we get another series containing the most common observation, when we apply mode to the dataframe, we get another dataframe with a single row containing the values that shows up the most frequently in each column.

# The rename() method: -

- This is the method that helps us in the data transformation. Using **rename()** we will be able to essentially change or rename our labels, and using the rename() we can change either the index and the columns. Our dataframe has two labelled dimensions or both of them.
- **df.rename(index={0:'Pratham'})** → this is how we use the rename method to change the index name 0 to pratham.
- **df.rename(columns={'weight':'WEIGHT (KG)'})** → this is how we use the rename method to change the column name.
- The rename() method returns the copy of the dataframe and to change the original dataframe we use the **inplace=True** parameter.

- An alternative approach is to simply pass a dictionary that maps the old labels with the new ones to a parameter called **mapper={}**. Whenever we use mapper, we have to be specific **about which axis** we want the mapper to apply to and by default it is axis = 0.
- **df.rename(mapper={'height':'HEIGHT (ft.)'}, axis = 1)** → this is how we use the mapper parameter, and the catch is, we could **not be able to rename both the column and the index axis at once**.

# DataFrame dropna(): -

- In dataframe the **dropna() method** comes with some additional powers as we have seen in the pandas series.
- Most imp we could specify the axis we want to drop from.
- **df.dropna()** → this is how we use the dropna() method and it will drop all the rows where even a single value contains **null or na** because by default this method looks like **df.dropna(how='any', axis = 0)**.
- The how parameter specifies the condition under which the method applies.
- We can also specify **how = "all"** and it will drop those row that contains all the null values. The how parameter allows only these two values.
- Another parameter that allows us to fine tune this dropping condition is **thresh parameter** and it is a short for threshold and it gives us an opportunity to specifically says how many non NA values we want in a given sequence and it could be a row or a column. So this is a bit of reverse condition of how parameter.
- **df.dropna(thresh=3, axis=0)** → this specifies that we want only those rows that contains atleast 3 NON-NA values in a row.
- **df.dropna(how="all", axis=1)** → this is how we use it with the columns by using axis=1.
- The **dropna()** method returns a copy of the dataframe and to apply the changes to the original data we have to use the **inplace=True** parameter.

# dropna() – with subset: -

- So far the **dropna()** has impacted the entire data frame. We call the method on the dataframe itself and it is applied to all of its rows and columns.

- Here we will introduce the **subset parameter** which gives us a way to restrict dropna() to only a handful of row and column we choose.
- **df.dropna(axis=0, how="any", subset=['gender'])** → this is how we use the subset parameter.
- **df.dropna(axis=1, how="any", subset=[0,2])** → this is how we drop the columns using the subset parameter.

# Part II: Merging units with column names: -

- The goal here is to take the units we isolated before and merge them into the column labels so that we don't completely discard the information.
- The bigger purpose is to get a nutrition dataframe that is all numeric, but at the same time we don't want to lose information on what the relevant units are.
- Our complete data preprocessing steps are: -
  - **Collect units** → isolate the units from each nutrition column label.
  - **Create mapper** → create a dictionary of key:value pairs containing the old labels and new.
  - **Rename df** → rename the column labels of nutrition dataframe
  - **Replace and convert** → replace all the units from the dataframe values and convert values to floats.
- **nutrition.rename(columns=mapper)** → this is how we rename the columns to the new names.

# Filtering in 2D: -

- When working with large data sets, it will be really nice if we were able to quickly slice and dice the labels we are interested in. And to achieve that conveniently we could use a method i.e. **.filter()**.
- It is same as **filter()** method in series but in dataframes it has got some super powers as well as it is used to filter the index as well as the columns.
- **nutrition.filter(like="Octopus",axis=0)** → this is how we use the filter method with the like parameter.
- The **.filter()** method also supports regex.
- **nutrition.filter(regex='[Oo]ctopus', axis=0)** → this is how we can use the filter with the regular expression also.

- We can also filter the data along both the dimensions also.
- The **items parameter** allows us to specify the exact sequence of labels that we want to match.
- **nutrition.filter(regex='[Oo]ctopus', axis=0).filter(items=['cholesterol_mg','serving_size_g','calories'],axis=1)** → this is how we use filter to filter rows and columns together.
- **nutrition.filter(regex='[Oo]ctopus', axis=0).loc[:,['cholesterol_mg','serving_size_g','calories']]** → this is how we can use the loc indexer along with the filter to filter out using the columns.

# Dataframe sorting: -

- Now, as we have the numerical dataframe, we are free to apply all the methods and functions that we have discussed in series.
- One such methods is **.sort()** for sorting the dataframe.
- **nutrition.sort_values(by=['calories'], ascending=False)** → this is how we can sort the values with the help **of by parameter**.
- **nutrition.sort_values(by=['cholesterol_mg','sodium_mg'], ascending=False)** → this is how we can also sort by multiple columns also.
- **nutrition.sort_values(by=['cholesterol_mg','sodium_mg'], ascending=[False, True]).head()** → this is how we can also pass the list of values in the ascending parameter also.

# Using Series between() with Data frames: -

- We've seen the Dataframe extraction by positions and labels across index and column dimensions, but specially working with numerical dataframes there are couple of other really powerful approaches that combine series methods, with dataframe extraction capabilities.
- One of the methods is series **between() method**, which gives us a convenient way to **specify a range of values to select based on a range of values().**
- So, to use this method, we first have to select one dimensional slice from our dataframe, it could be a row or a column.

- **nutrition.calories.between(20,60)** → this is how we use the **between() method** to get a range of values that ranges between the specified range and it returns a Boolean mask series with the size equals to the original series.
- Combination of the series between method and dataframe Boolean indexing is a powerful and a very common technique.

# BONUS – Min, Max, and Idx[MinMax]: -

- Similar to the Min and Max methods in series, we can use them here in the Data frame as well and they can now support an axis parameter as we can apply them on rows as well as on the columns.
- The default is set to find the column wise.
- Axis = 0 → row/index axis = 1 → column.
- **nutrition.max(axis=1)** → this is how we can use the min and max with the dataframe and we can also specify the axis in them and it will give us the minimum and maximum value.
- We can use the **idxmax()** method to identify the label associated with the max value.

# DataFrame nlargest() and nsmallest(): -

- For the purpose of getting the largest X values and their labels, we can also use the methods that we have discussed in the series also and **i.e. .nlargest() and .nsmallest() method**.
- **nutrition.nlargest(10, columns='potassium_mg').potassium_mg** → this is how we can use the .nlargest() method by providing it the column paremeter, which column we want to extract.
- **nutrition.potassium_mg.nlargest(10)** → we can also use it like this also by defining the desired column before the method and here we don’t need to specify the column name inside the .nlargest() method.
- The **.nsmallest()** will give us the first n smallest records from a particular column.
- **nutrition.nsmallest(10, columns=['potassium_mg', 'sodium_mg', 'folate_mcg'])** → we can also pass the list of columns like this.

# Data Frames in Depth: -

# Indexing with Boolean mask: -

- We've seen the indexing with the Boolean mask before.
- The key concept when using the Boolean mask is to best conceived as 2 step process: -
  - Generate a sequence of Booleans.
  - Use Boolean sequence in [ ] or .loc [ ]
- **players[players.market_value > 40]** → this is how we use the Boolean mask for indexing in the dataset.

# More approaches to Boolean masking: -

- Some of the other methods that helps us to generate the Boolean series on fly are: -
  - **players.position.isin(['LB','CB','RB'])** → If we want to extract multiple records based on multiple condition from a column in data set the we use series **.isin([' ', ' '])** method and it will return a Boolean output.
  - **players.market_value.between(40,50, inclusive='neither')** →If we want to extract using the range of values then the series **.between()** method will be used here.
  - More method for creating Boolean sequences are comparison operators like >, <, >=, <= ,!=, ==. But in pandas we have an alternative method to use these comparison operators directly and **that are comparator wrapper methods** like **players.age.le(25).** The wrapper methods **support the fill_value parameter** as well which can be used for replacing the non-NA value whenever an Na value is found. The wrapper methods for corresponding comparison operators are: -
    - < → .lt()
    - <= → .le()
    - > → .gt()
    - >= → .ge()
    - == → .eq()
- The **.unique()** method will get us the list of all the unique records from our dataset.

# Binary operators with Booleans: -

- Binary operators are just like the other operators, but they work on the **binary representation** of the value, on the individual bits.
- So, they allow us to do operations a bit-by-bit basis.
- The 2 that are most useful when working pandas are the binary **AND (&)** and **OR (|)**.
- Think of the OR operators is searching for the True. And the AND operator is always True unless there is a False a single False can result in False.
- When we **combine 2 pandas Series** then the **combination is done label to label** and not based on the order.

# BONUS – XOR and complement binary ops: -

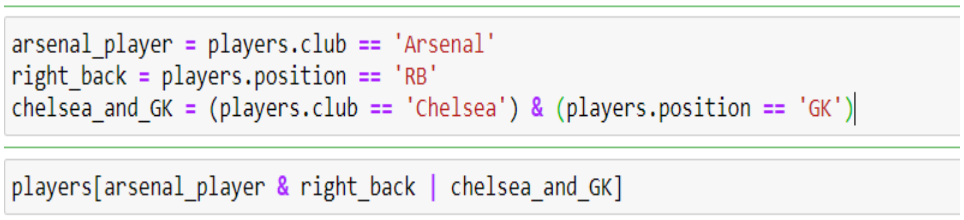
- These are some other binary operation other than **AND(&)** or **OR(|).**
- Starting with we have **XOR(^)** which is exclusive OR. Its only true when the inputs are different and false when they are similar.
- We can also combine them with other operators also.
- Another binary operator is the **2's complement or complement operator** i.e. **~.** It is extremely useful in negating Boolean series or negating all bits.
- This is also known as the binary complement.
- In python **Booleans are integers**. The ~ inverts the bits representing the numbers.
- In pandas we will use the complement binary operator to negate our Boolean conditions. This is very helpful in defining the negative conditions when we are doing dataframe indexing.

# Combining conditions: -

- We can also do indexing using the **multiple conditions**.
- To avoid errors which combining conditions we should wrap our conditions in parenthesis.
- **players[(players.position == 'LB') & (players.age <= 25)]** → this is how we can combine conditions.

# Conditionals as variables: -

- This is how we use conditionals as variables: -



```
arsenal_player = players.club == 'Arsenal'
right_back = players.position == 'RB'
chelsea_and_GK = (players.club == 'Chelsea') & (players.position == 'GK')
```

```
players[arsenal_player & right_back | chelsea_and_GK]
```

# 2D indexing: -

- So far, we have discussed Boolean indexing in the context of extracting rows. This is the default behaviour in pandas.
- But when working with DataFrames we are working with 2-D structure. So, we also have the option of indexing along the other axis as well.
- **players.loc[chelsea_23_under, ["position", "market_value"]]** → this is the 1st method we can indexing using columns also in pandas dataframe.
- **players.columns.str.startswith('p')** → the **.startswith()** is a string method in python that tells us that weather a givens string starts with a particular character or not and return a Boolean result. Then we pass these columns to the loc indexer and it will filter the data based on rows as well as columns.
- To do horizontal indexing we have to make sure that the length of the Boolean series matches the length of the column axis for the underlying dataframe.
- Another way of indexing along the column axis is [ ] chaining **players[chelsea_23_under]['position']** this is the another method of indexing based on columns.

# Fancy Indexing with lookup(): -

- It is another approach for indexing in dataframes. It simple refers to passing multiple labels all at once.
- Here we'll talk about a panda's method that is dedicated to do fancy indexing item selection. The method is called **.lookup()**.
- This method takes a specific row label and column label or a list of them then it returns the values associated with each pair of labels.
- **players.loc[[0,132],('name', 'market_value')]** → this is the basic fancy indexing.
- **players.lookup([0,132], ['name','market_value'])** → this is how we use the **lookup()** method.
- The lookup method is most useful when we already have some collection of labels that we want to use to do dataframe selection.
- **players.set_index('name').loc[names,attributes]** → this is how we use the loc indexer instead of .lookup().

# Sorting by index or column: -

- Previously we explored the **sort_values()** method, which helps us sort our row labels based on different columns that we specify in by parameter.
- But that is not the only way to sort. We occasionally also find the need to sort by the index and this is usually the case for indices that we create.
- A good practice when working with tabular data, especially in the context of databases, is to always maintain an ordered index.
- **players.sort_index()** → To do this we use the pandas **.sort_index()** method to sort the indices.
- **players.sort_index(axis = 1)** → this is how we can also sort the column labels by setting the axis parameter.
- **players.reset_index()** → this method is used to reset all the changes made to the index.

# Sorting vs reordering: -

- The sort_index(), sort_values() methods we have seen is a bit restrictive. But there are only two ways to fundamentally sort using these methods like we either go in ascending or alphabetical order or descending or reverse order.
- If we want to more precisely order the rows or column according to the more specific order that we specify.
- To do this we will use the **.reindex()** method.
- **players_lite.reindex(index=[2,1,3,0], columns = ['age', 'name', 'position','club'])** → this is how we use the **.reindex()** method to easily reorder our data as per our requirement.
- Pandas index object is an **iterable**.
- In python there is a convenient method that returns a new sorted list from a iterable and i.e. **.sorted()**. And **sorted(players.columns)** this is how we use it.
- **players.reindex(index=[2,1,3,0], columns=sorted(players.columns))** → this is how we directly use **.sorted()** with **.reindex()** to sort the columns.
- The **.reindex()** could be used to carve out a slice of the dataframe containing a set of index and column labels in the order that we specify. **players.reindex(index=[2,1,3,0], columns=sorted(players.columns)[:6])** → this is how we get only 1st 6 columns.
- **players.reindex(index=[2,1,3,0], columns=players.columns.sort_values())** → we can also sort the column like this also alternatively.
- **.swapaxes()** → this method is used to swap the axis of the dataframe row → column and column → row. And the **.T attribute** also do the same.

# Identifying dupes: -

- Duplicates are the real problem and all the solutions around duplicated values start with identifying or being able to know what the duplicates in a data set are.
- To find the duplicated record in pandas we use the duplicated method i.e. **.duplicated()**.
- **players[players.duplicated()]** → this is how we can find the duplicate records from a dataframe and it returns a Boolean series.

- The values across all the columns need to be the same for two records to considered duplicates.
- We somehow need to have the ability to define precisely what constitutes a unique value and in pandas we can loosen up this restriction that all columns be compared by specifying a subset of columns that should be considered.
- That is done using the **subset=[‘ ’] parameter**.
- **players[players.duplicated(subset=['club', 'age','position', 'market_value'])]** → this will give us the duplicates by only the column labels mentioned in the subset.
- If we have several repeating values, then how do we distinguish between the original and the duplicate.
- It is in pandas’ default that we treat the first occurrence, the first original occurrence as the original and everything that follows it similar to it is a duplicate.
- We set the **keep=”first”/“last”** inside the **.duplicated()** method, which specifies that consider the first value as the original value like this **players[players.duplicated(subset=['club', 'age','position', 'market_value'], keep='first')].**
- **keep= False** → it indicates that consider all the repeating values as duplicate.
- By default, pandas will compare all the records along all columns for repeating values, but oftentimes we want to have more precise definition and there is only subset of attributes that we really care about so we can customize it by setting the **subset parameter**.
- To identify which one to treat as duplicate and which one to original then we set the **keep parameter** to first, last, False.

# Removing duplicates: -

- What appears to be **duplicated input is not always incorrect or bad data sometimes we want them in our dataframe**. But sometimes due to the duplicates in the dataset we get the wrong calculations so it’s better to remove them.

- **players.drop_duplicates(keep="first")** → To remove the duplicate records we have the **.drop_duplicates()** method which drops the duplicate values and return the copy of the Data frame.
- So, after removing the duplicate values we get the calculation which we are getting wrong previously.
- So these are the methods for excluding the duplicated records from the data frame i.e. **.duplicated(), .drop_duplicates()**.

# Removing dataframe rows: -

- Another approach to remove duplicated records is first identify all the records that we want to remove and then exclude all or some of them separately.
- The more generic drop method to drop a particular method is **.drop(labels=, axis =)**. **players.drop(labels=19, axis=0)** → this is how we drop a row from the dataframe.
- **players.drop(index=19)** → this will also give the same result.
- **players.drop(index=[19,20,21,231])** → we can also pass the list of rows we want to remove in this method.
- This method returns the copy of the DataFrame.

# BONUS - Removing dataframe columns: -

- **players.drop(columns='age')** → this is how we drop the column from a dataframe.
- **players.drop(columns=['age', 'market_value'])** → we can also pass the list of values inside the drop method.
- **players.drop(labels=['age', 'market_value'], axis=1)** → we can also drop the columns like this also.

# BONUS – Another way: pop(): -

- It is the another way of removing columns i.e. **.pop() method**.
- The 3 things to note about this method are: -
    - We didn't get the DataFrame with the data removed as we did with the drop method, but with this method we get the removed column itself in the form of series.
    - **.pop()** works on a single column at a time.
    - This method changes the original dataframe because the inplace is set to True by default in this method.

# BONUS – A sophisticated alternative: -

- Another alternative to drop records from the dataframe is to re-index the dataframe by leaving out several unwanted columns and rows.
- The **.reindex()** method crafts a new dataframe that only reflects the labels that we specify if we re-index the players.
- **players.reindex(index=set(players.index).difference(unwanted_rows), columns = set(players.columns).difference(unwanted_columns))** → this is how we can remove the indexes, columns from the dataframe using the reindex method by calculating the set difference.

# Null values in dataframe: -

- The null values are also known as null markers or NaN i.e. not a number. They simply refer to the gaps in data.
- The **.isna()** method is great approach for labelled 1D sequence of values, in other words series.
- But the **.isna()** method also works on the dataframe also.
- **np.count_nonzero(players.isna())** → this is how we can count the Nas in the dataframes using the numpy **.count_nonzero() method**. It returns a count of the truthful values.
- **players[players.isna().values]** → this is how we get the records with the NAs in the dataframe.
- **players[players.isna().values].drop_duplicates()** → and this is how we can also drop the duplicates if there are any.

# Dropping and filling dataframe NAs: -

- null value usually indicates gap in the data. This could be the information that was lost, not available or simply incorrectly recorded.
- One inclination we could have is to fill or replace these null values with something more meaningful and the **.fillna()** method is useful for that.
- This method returns the copy of the dataframe where the method replaces the null values with the value mentioned in it.
- **players.fillna("NA REMOVED").iloc[[30,192,195]]** → this is how we can use the **.fillna()** method to remove the null values.
- In reality, our needs might be in a bit more sensitive to the context. E.g. the nulls in the market_value should be replaced with a numerical value.
- To be able to do that we'll be able to use the dictionary syntax of the **.fillna()** method.
- **players.fillna({'market_value': 100, 'position': 'RM'}).iloc[[30,192,195]]** → this is how we can also replace the values by passing them in the dictionary.
- The entirely different approach to deal with null values is to simply exclude them and for that we have **.dropna() method**.
- **players.dropna()** → this method will exclude the rows containing the null values. So we are dropping the index labels and the entire dataframe rows are removed if any of the values contains the single null. i.e. the defaults are axis=0 and how = "any".

# BONUS – Methods and axes with fillna(): -

- The filling and dropping methods were not inplace and it just returns the copy of the dataframe so the changes will not affect our dataframe.
- Previously we use the strings or dictionaries in order to replace with the new values that we specified.
- An alternative to that is to use the **method="ffill"** parameter in the **.fillna()** method. This method is **copying forward** the previous non-missing value to a forward field our NA values.
- **players.fillna(method="ffill", axis = 0).loc[[29,30,192,195]]** → this is how we use the method parameter.
- By default, the axis=0 in this method and **when the axis is 0, then the filling happens vertically** so we copy forward from the preceding row record.

- With the **axis = 1** then the filling of the values happens horizontally and we copy forward from the preceding column.
- Think of the axis parameter as aligning or copying either along the index when the axis is 0 or along the columns when the axis is one.
- This filling methods are particularly useful when working with **time series data**.
- The **.uinque()** method is used to find the unique records from the table.

# Calculating Aggregates with agg(): -

- The **.agg()** method will help us in grouping our values along several attributes.
- The **.agg()** will accumulates or aggregates the output of the function and then collapses the entire axis into a single value.
- The **select_dtypes()** method returns a new DataFrame that includes/excludes columns of the specified dtype(s).
- **players.select_dtypes(include=[np.number]).agg('mean')** → this is how we use the agg() function because in newer version of pandas it is strict related to the data types which are involved in the aggregation so first we have to select the columns with the numeric values and then apply the aggregation.
- The comparison operators are not supported between integers and strings.
- To prefilter the column by types, we want to perform aggregation on is by using the **select_dtypes() method**.
- **players.select_dtypes(include=[np.number]).agg(['min', 'max', 'mean'])** → we can also pass the list of aggregate function and it will give us a dataframe where min, max, mean of all the columns are reflected.
- The **.agg()** function reshapes our data.

# Same-shape Transforms: -

- Pandas has also a dedicated method used for applying custom functions to the entire dataframe i.e. **.transform()**. This method guarantees that the shape of the dataframe will not change.
- **players.loc[:,['market_value', 'fpl_value']].transform(lambda x: x * 0.93)** → this is how we use the transform() method.

- To apply string methods directly on the pandas series we use the **.str** keyword between the method name and the applicator like **ser.str.upper()**.

# More flexibility with apply(): -

- Like we use the **.apply()** method in Series, we can also apply the same method in dataframe also.
- Whenever we want to apply a given function to the entire row or column then we use the **.apply() method**.
- The dataframe **.apply()** function gives us the opportunity to apply a function across the entire dataframe, one row or one column at a time.
- To use the .apply() function: -

  -
    ```
    def round_floats(x):
        if x.dtype == np.float64:
            return round(x)
        return x
    ```
  - **players.apply(round_floats)**
- we use the **.apply()** function like this in the dataframe to apply the transformation to the entire dataframe.
- The **.apply()** method is bit more flexible and powerful and it has more general scope as compare to **.transform()** specifically in that it supports aggregations as well as same shape transforms.
- The **.apply()** method first tries to determine whether the function being applied involves some form of aggregation if yes it perform the aggregation and if no then it perform the transformation.
- In all the transformation we can also control how to transformation is being applied i.e. row, column wise by changing the axis parameter.
- **players.select_dtypes(['int64', 'float64']).apply('mean', axis = 1)** → like this we can apply the axis also.
- **players.loc[460,[dtype != object for dtype in players.dtypes]]** → we can also use the list comprehension to dynamically select the columns that we want.

# Element-wise operation with applymap(): -

- The methods we have discussed like **.agg(), .transform(), .apply(),** these operate on entire columns or rows at once.
- **Vectorized operations** apply a given operation or transformation on a set of values all at once. They give us huge performance because they allow us to apply a given operation on a set of values all at once instead of operating them individually.
- **SIMD → single instruction, multiple data** processors allow the processing unit which could be a CPU or GPU to perform the same operation on multiple data points in a single processing cycle.
- Not all the functions could be vectorized. Occasionally we might be working with a function whose logic depends upon accepting and operating on an individual value. Operating one item at a time. And for such cases, pandas offers the **applymap() function**.
- The way we use the .applymap() function is: -

  -
    ```
    from datetime import datetime
    count = 0
    def log_and_transform(x):
      global count
      count += 1
      if count % 100 == 0:
        print(f"Its {datetime.now()} and I just adjusted the {count}th
    value.")
      return x * inflation
    ```
  - **mini_df.applymap(log_and_transform**)

# Setting dataframe values: -

- Sometimes we don’t want the large-scale operations that transforms the entire row or entire column at once.
- Occasionally we just want to spot change our data.
- We could easily do that using the equals assignment operator. First, we select the value and then we use the equals assignment operator to make the change.
- **players.loc[3, 'position'] = 'CM'** like this.

- **.at() and .iat()** should be preferred for single value assignment.
- So, we have 4 values to assign spot values in the dataframe i.e. **loc, iloc, at, iat**.

# The setting with copy warning: -

- The warning that pandas might throw at us when we try to update or set values in our dataframe.
- When we work with the direct assignment then pandas don't know that weather we are working with a copy or a view.
- If we are working with a copy of the dataframe any change that we do to that copy does not change the underlying data.

# View Vs Copy: -

- Whenever we index or extract slices or apply a method to our dataframe, we should think that we are working with a copy or a view.
- **A view** is like a window to the underlying data, it gives us the direct access to the data. If we modify what we see, we end up modifying the dataframe.
- **A copy** is the replica of the what's in the data frame. It's a different distinct object. So, if change the copy we don't affect the underlying data.
- 2-point rule: -
  - Pandas loves to give us copies
  - If we use the loc/iloc or at/iat, we are guaranteed by pandas to get a view.

# Adding dataframe columns: -

- Sometimes we need to expand our dataset by adding the new attributes.
- The easiest way to add a column is to just use the assignment equals operator on square brackets with a label that doesn't exist yet.
- First confirm that the column you want to create should not already exist in the dataframe like **'column_name' in dataframe_name**.
- **players['MVtoFPL'] = 1.**0 → this is how we add new column to the dataframe.
- Another approach to add the column to the dataframe is to use the **.insert()** method.

- **df_mini.insert(0,'nicknames', players_names) →** this is how we insert new column in the dataframe using the insert function. where the player_names are the series of values containing names list. This method allows us to specify where the column should be placed exactly.
- Another method we can use to add the columns in the dataframe is to use the method **.assign().** This method assigns new column to the existing dataframe and returns a new copy.
- **df_mini.assign(career_goals = [12,67, 179, 49]) →** this is how we use the .assign() method to add the new column to the dataframe. With this method we can also add multiple columns at once also.

# Adding dataframe rows: -

- the first method to add the new column to the dataframe is to use the **._append()** method. This method works with Series, dataframes and collection of them.
- We add the rows in the dataframe like: -
  - **cristiano = pd.Series ({**
    'nicknames': 'Cristiano',
    'age': 32,
    'position': 'RW',
    'club': 'Juventus',
    'position_category': 1
    }, name = 4)
  - **df_mini._append(cristiano)**
- We can also add multiple columns in the dataframe like **df_mini._append([player1],[player2])**.
- We can also add a row like this: -
  - other_players = pd.DataFrame({
    'nicknames': ['Gian', 'Lionel'],
    'age': [32, 37],
    'position': ['GK', 'CF'],
    'club': ['Barcelona','Juventus'],
    'position_category': [4,2]
    }, index = [5,6])
  - df_mini._append(other_players)

- These approaches return the copy of the dataframe and doesn't change the original dataframe.
- Adding rows to the dataframe is inefficient, a very expensive operation.

# How are dataframes stored in the memory: -

- Setting rows in the pandas dataframe is very inefficient, weather we are setting with the enlargement or using square brackets or even directly using the append method.
- Each and every time we use these operations the entire dataframe is copied over in memory.
- Pandas under the hood relies on numpy nd array.
- In pandas we are grouping columns based on their data types and then we are storing these chunks together in memory.
- When we add a column then the pandas first determine its types. And then it identifies the block it should be added to and then it modifies that block only and the rest remains unchanged.
- When adding a row, we have to touch each and every block, all of them need to be copied, modified and reassigned back to memory and this is to every row that we append.

Working with multiple data frame: -

Concatenating dataframes: -

- When working with data to address specific analytical questions, oftentimes we need to combine multiple data sources.
- We use the **.concat()** operation to put the dataframes together.
- **pd.concat([ivies, eng])** → this is how we use the **.concat()** operation.
- **pd.concat(dfs).drop_duplicates(subset=['School Name'], keep='first')** → this is how we can drop the duplicates from the concatenated dataframes.

The duplicated index issue: -

- The dataframe we have created above also had a lot of duplicated indices. This is because each dataframe we concatenated has its separated index number as similar to other dataframes.
- This is because the **.concat()** method doesn't discard the original index of the dataframes being concatenated.
- Its technically not invalid or wrong in pandas to have duplicated values in index.
- But as our indexes contains the duplicate values then the slices no longer work.
- So, our preference is when combining the dataframes, is to have atleast one common index with unique values across all the records.
- The **first approach** is to reset index of the dataframe we have using the **reset_index()** method and it restores the 0 based RangeIndex in the dataframe while containing the old index to be a regular column in the new dataframe. **schools.reset_index(drop=True, inplace=True)** → this is how we can reset the index.
- **Another approach** is to discard the index before we do the concatenation. **pd.concat(dfs).drop_duplicates(ignore_index=True, subset=['School Name'])** this is how we can use the **ignore_index** parameter to change the index.

# Enforcing unique indices: -

- Sometimes we want to preserve the indexes of the original dataframe because they can be meaningful sometimes, but at the same time we want them to be unique.
- To ensure that the indices contain the unique values then we use the **verify_integrity parameter** in the concat method that does the unique index check for us. Pandas will throw error if it notices a duplicate index.
- **pd.concat([ivies2,eng2], verify_integrity=True)** → this is how use the verify_integrity parameter.
- This is one way to confirm that the index contains unique values after the concatenation.
- The **verify_integrity** parameter is not exclusive to the concat method only, we can use it with some other methods also.

# BONUS – Creating multiple indices with concat(): -

- Sometimes we want to slightly go in a different direction and instead of completely ignoring the index we might need to add another level of indexing to allow us to partition or easily identify our original dataframes withing the structure of new one that we create.
- **.concat()** method makes easy for us **to create multi-indexes** using the **keys parameter**.
- **pd.concat([ivies, eng], keys=['ivyleague_schools', 'engineering_schools'])** → this is how we create **multi-indexes using the keys parameter**.
- **new_df.loc[('ivyleague_schools',3)]** → this is how we do index in the case of multi-indexes.

# Column axis concatenation: -

- So far we have only use the **.concat()** method to combine the existing dataframes along the row axis.
- If we want to **combine along the column axis** then we only need to change the axis parameter in the **.concat()** method and change it from its default value from 0 to 1.
- **pd.concat([ivies3, eng3], axis=1)** → this is how we use the axis parameter.

#The ._append() method: a special case of concat(): -

- In the previous section we use the **._append()** method to add rows to our existing datasets. The append and concat are actually quite similar.
- The differences between them are: -
  - The ._**append()** method is an instance method and it could only be called on existing dataframe or series, and it returns a new dataframe or series where the data has been combined. It is less flexible as it doesn't do the column wise concatenation. Its axis of operation is fixed to 0.
  - The **concat()** method is available on the main pandas namespace. It is more flexible as compared to the append method.
- We can think of **append as a special case of concat** where the axis is set to 0 by default.

# Concat on different columns: -

- Sometimes we have to concatenate one dataframe with another that has an extra column that we don't really need.
- The default behaviour of pd.concat is to include the set of all unique column names in the resulting dataframe.
- **pd.concat([ivies, eng4], join='inner')** → setting the **join = 'inner'** indicates that only concatenate the columns that are common in all the dataframes and it will exclude the column that is not there in both the dataframe.
- The default join = outer and it adds all the columns in the dataframes. By setting the other dataframe values to NAs.

# The merge method: -

- It is a very powerful technique in pandas that allow us to combine dataframe using techniques and logic very similar to SQL.
- The .**merge() method** in pandas gives us a flexible interface to join various dataframe and series objects.
- How joining and merging is different from concatenation or append: -
  - Think of **.concat()** as an operation that glues together pieces of data into one unified data structure.

    - **Merging** gives a lot more options to flexibly **combine multiple datasets on the basis of the content** that they have in common.
- **.merge()** is lot more sensitive to the content in the dataset.
- **pd.merge(schools, regional_info, on='School Name')** → this is how we merge two datasets by using the common column in both the dataset by specifying it using the **on parameter**.

# The left_on and right_on Params: -

- very often we want to come across the dataframes that we want to merge, but which do not have a key column of the same name.
- The **left_on** and **right_on** parameters helps us to be very explicit about exactly what we want to merge by. In other words what the key column in the each datasets would be.
- **pd.merge(schools, mid_career, left_on='School Name', right_on='school_name')** → this is how we should use the left_on and right_on parameter.

# Inner vs Outer joins: -

- What logic should be applied we merge two datasets.
- The type of join operation in a Pandas context is controlled by **the how parameter** of the **.merge()** method.
- The **default is set to 'inner'** and it returns only the records that are present in both the right and the left object. It returns what's common to both the dataframe. It is similar to intersection.
- **pd.merge(ivies, regional_info, how="inner")** → this is how we use the 'inner' inside the **parameter how**.
- The other option is outer. Think of outer as making sure that all the records from both the data frames are reflected in the resulting merge. It is similar to set union.
- **pd.merge(ivies, regional_info, how="outer")** → this is how we use the 'outer' in the how parameter and it will give us all the records that are present in both the tables.

# Left Vs Right joins: -

- The **.merge()** method also supports two other join methods. Specifically left and right.
- **pd.merge(ivies, regional_info, how="left")** → when doing a left join. We are specifying that we want to preserve the key indices on the left object, merger the data and then discard everything else.
- **pd.merge(ivies, regional_info, how="right")** → when doing a right join. We are specifying that we want to preserve the key indices on the right object, merger the data and then discard everything else.
- If we flip the order of the dataframes that we are merging **then the left join becomes the right join** and vice-versa.

# One-to-One and One-to-Many joins: -

- When working with data in general, when combining multiple datasets, we come across terminology that describes the type of association between different entities in our data.
- **One-to-One** join simply happens when each record in a dataframe is associated with one record in another data set.

# Many-to-Many joins: -

- These are the type of joins that occur when we have duplicates in key columns from both the left and the right object that are being merged.
- The type of output or a join behaviour that multiplies the datasets is called the **Cartesian product**.
- **1 – 1** → both merge objects contain unique values in the respective key. E.g. person <-> DNA
- **1 - M** → one of the merge objects contains non-unique values. E.g. book-pages
- **M – M** → both the merge objects contain non-unique values. The resulting records are repeated M x M times. E.g. book-author.

# Merging by index: -

- In all the above methods we perform joining of the dataframes based on the columns.
- Occasionally we might be interested in joining by index instead and the pandas **.merge()** method supports that.
- **pd.merge(ivies4, regional_info2, right_index=True, left_index=True)** → this is how we merge the dataframes based on index.
- **pd.merge(ivies4, regional_info, left_index=True, right_on='School Name')** → this is how we can merge a combination of index and a column.

# The join() method: -

- For the types of joints that we say in the previous lecture, specifically index on index and column on index, pandas provide a convenient method, simply called .**join()**.
- **ivies4.join(regional_info2)** → this is how we use the **.join()** method.
- The .join() method also supports columns on index joins.
- **.join()** is a convenient method for writing shorter code. Under the hood the **.join()** method is called as the pandas merger method.

Going Multidimensional: -

Index and RangeIndex: -

- In Series we have seen that index is simply a label for each row, for each value and then we switched to working with dataframes and then we introduced an additional dimension, i.e. the columns dimension.
- If we isolate the index for our dataframe, we see that it’s a range index object.
- **Range index** is simply an **immutable or unchangeable** object that represents a series of **increasing or decreasing integers**. The range index in pandas is **further inheriting from the index class**.
- We can think of the index as well as the columns as inheriting from one base index object.
- **Index** is another immutable data structure.
- Oftentimes we find the need to replace these default range index with something meaningful, more meaningful set of labels.
- **tech.set_index('date')** → this is how we set index in our dataframe explicitly which we think should be more meaningful.
- **tech.set_index('date').loc['2019-08-01']** → this is how we can also use indexing with the set_index method.

Creating a multi-Index: -

- Previously we use the **set_index()** method to change the index of the dataframe from the default 0 based RangeIndex to something more meaningful.
- MultiIndex means that a **single index object has more than one level or component** to it. Multi index is also known **as hierarchical index**.
- By using multi-index, we are creating a **hierarchy of relationships** within our data where the information across the two index levels is inseparable from the actual values.
- **tech.set_index(['date', 'name'], inplace=True)** → this is how we make multi index in pandas.

# Multi-index from read_csv: -

- Here we will see more efficient approach to make multi-indexes in our dataframes using the parameter in the read_csv() method and i.e. **index_col=[' ',' '].**
- **pd.read_csv(data_url, index_col=['date', 'name'])** → this is how we can make multi-indexes directly.

# Indexing Hierarchical dataframes: -

- As we have already seen how to create multi-index dataframes now let's understand how to extract values from this data set.
- Multi-index **changes the appearance** of the dataframe, but it also does lot more than that.
- **tech.loc['2014-01-02','GOOGL'].close** → this is the basic way of doing indexing in the dataframe with multi-indexes.
- Another way to do the exactly same thing is to take the advantage of tight coupling we have between name and date field in our dataset.
- We can treat both of them as a single dimension and capture them in a tuple of values.
- **tech.loc[('2014-01-02', 'GOOGL'),'close']** → this is how we use the tuple to capture the records in multi-index.
- The location based indexing **using .iloc** works in the similar way as it does earlier as it is agnostic and doesn't care about the structure of the dataframe.
- **tech.iloc[2,4]** → this is how we use the .iloc[] indexer and it gives us the same results.

# Indexing Ranges and slices: -

- Here we will look at extracting slices of values.
- **tech.loc[(['2015-01-06', '2015-01-07'], ['FB', 'AMZN']),['close', 'volume']]** → this is how we select the specified values from the multi-index dataframe.
- In regular dataframes and series, we have the ability to slice our data structures by specifying a range of values separated by a colon.

- **tech.loc['2017-01-03':'2017-01-31', 'open':'low']** → this is how we can also do slicing in multi-index dataframes.
- If we want to slice a hierarchical index then, we have to use the slice object.
- **tech.loc[(slice('2017-01-03','2017-01-31'), 'GOOGL'), 'open':'low']** → this is how we use the slice() object to get the records for a particular ticker.
- If we need to specify a slice, a range of consecutive values within a multi-index, we need to use the **slice() object**.
- The equivalent of **':'** in the **slice()** object is NONE. **tech.loc[(slice(None), ['FB', 'AMZN']),'open']** → this is how we use the None inside **slice() object** to select all the dates from the dataset.

# BONUS – Use : With pd.IndexSlice: -

- To do slicing using colons ':' we use **the IndexSlice[]** object.
- **tech.loc[pd.IndexSlice[:, ['FB', 'AMZN']], 'high': 'low']** → this is how we can use the **pd.IndexSlice() object** and use the colons with them.
- if we are working with more complex multi indices and we know that we are going to use the **IndexSlice() object** several times, it might be a good idea to assign it to a shorter variable like **i = pd.IndexSlice[]**.
- tech.loc[i['2014-01-06':'2014-01-10', ['FB', 'AMZN']], 'high':'low'] → this is how we can use the **IndexSlice() object**.

# Cross sections with xs(): -

- We have seen couple of indexing approaches for hierarchical dataframes and they are all powerful and flexible, but perhaps not the most intuitive in terms of syntax.
- The **.xs()** method is the subset of the label based indexing what we have seen. This is much straight forward method.
- **.xs()** is a type of label based indexing.
- By default **.xs() drops** the level it is selecting from.
- **tech.xs('FB',level=1, drop_level=False)** → this is how we can use the **xs() method** along with the **level and drop_level** parameters.
- **tech.xs(('2019-01-02','FB'), level=(0,1))** → this is how we can also select from the multi-levels also.

# The Anatomy of Multi-index object: -

- Multi-index is **like a data structure in its own**. It has its own specific attributes, sequence of values and methods.
- The **.names attributes** returns, the list containing the names of each label.
- The **.levels[] attribute** are a list of lists that contains each label value for each of the labels in multi-index.
- The **.nlevels attribute** gives the number of levels we have in our multi-index.
- **tech.index.levels[0]** → it gives us the values in the index with level 0.
- If we want to look at the length of each of our labels, we could use the **.levshape attribute** like this **tech.index.levshape**.
- Multi-indices are not just freely floating sequences of labels, they represent a tight coupling of hierarchies of labels and an **easy way to look at those label combinations** across our index labels is to access the **.values attribute** on the multi index itself like **tech.index.values**.

# Adding another level: -

- **tech.set_index('volume_type', append=True)** → this is how we can add another index to the already existing index without dropping the current indexes by append = True.
- **tech.loc[(slice('2019-01-01', '2019-01-31'), slice(None), 'high'),:]** → this is how we do indexing when the multi-index contains 3 values.

# Shuffling levels: -

- As we add additional levels to the multi-index but if we are not happy with the order in which these levels appear.
- Using **the .swaplevel() method** we can swap the levels in multi-index by passing the values of them.
- **tech.swaplevel(i = 2 , j = 1)** → this is how we use the method to swap the level 2 with level 1.
- **tech.swaplevel('volume_type', 'name')** → we can also use it with the label names also.
- If we want to apply the changes to our dataframe then we have to reapply it to the dataframe as it doesn’t support the inplace parameter.

- In pandas there is more powerful method that works with more than 2 levels at a time is the **.reorder_levels()** method. This allows us to express multi-index all in one go.
- **tech.reorder_levels([2,0,1])** → we use the method like this.

# Removing Multi-Index levels: -

- Pandas recently introduced a method to drop the levels from the multi-index and that is **.droplevel()** method.
- **tech.droplevel(1)** → this is how we use this method, but the dropped level doesn't restore back to the dataframe and it is completely gone.
- The another method is **.reset_index()** which not only allow us to remove a given level, just like **.droplevel()** but it also restores it back to the dataframe.
- **tech.reset_index(1, drop=False)** → we can use it like this and the drop parameter specifies that don't drop the level completely else add it back to the dataframe.
- **tech.droplevel(['volume_type', 'name'])** → this is how we can also drop multiple levels from the multi-index.
- The **.reset_index()** is also helpful in getting us back the default index by resetting all the indices back to the dataframe.

# Multi-Index sort_index(): -

- We can sort our multi-index using the **.sort_index()** method and it will sort the index by default in ascending order.
- If we don't sort our index then while slicing some values from the multi-index it might cause some errors because the values are not sorted.
- In general, make this a practice to always sort the multi-index and the advantages of this is: -
  - Improves retrieval performance, which becomes significant
  - Enable slicing syntax
  - Overall, a good practice when working with tabular data representations, including pandas, excel, SQL etc.
- With the multi-indices we can fine tune the .**sort_index()** method by defining the **level parameter** inside it like **tech.sort_index(level=(0, 2), ascending=[False, True])**.

# More multi-index methods: -

- The multi-index object could also be manipulated as a standalone object since it is a data structure on its own.
- Some of the methods that apply to multi-index only are: -
  - The **.is_lexsorted()/ .is_monotonic_increasing()** → methods are use to check if the multi-index is sorted or not and returns a Boolean value. **tidx.is_monotonic_increasing** this is how we use it.
  - If we want to modify the sort for a given level within our multi-index without touching the dataframe then we can use the **.sortlevel() method. tidx.sortlevel(0, ascending=False, sort_remaining=True)** this is how we can use it or **tidx.sortlevel((0,1,2), ascending=[True, True, False])**.
  - If we want to change the name of the levels in the multi-index to make it more presentable then we can use the **.set_names() method** on the multi-index directly like this **tidx.set_names(['Trading Date', 'Volume Category', 'Ticker'], inplace=True)**.
  - **.to_flat_index()** → Using this method, we can convert the multi-index to a flat non-hierarchical structure.

# Reshaping with stack(): -

- This method is widely used along pivot tables.
- The **.stack() method** is used to take the column axis and pivoted or rotated into the innermost level of index.
- The **.stack()** method takes the innermost column axis and makes it the innermost index level.

# The flipside: unstack(): -

- It does completely the opposite of the **.stack() method**.
- The **unstack() method** takes the innermost level of a multi-index and it takes it back to the column axis.
- The **fill_value = "" parameter** inside the **unstack() method** allows us to replace the NaNs with some other value like **this .unstack(fill_value='-')**.
- We can also specify that on what levels we want to work on in the **unstack() method**. By default, it works on the innermost level.

- **stacked.unstack(level=1)** → this is how we can also specify the level in the unstack() method to specify which level we want as an column in the dataframe.

# BONUS – Creating multilevel columns manually: -

- Previously we end up creating multi-Index columns by chaining 2 unstack calls to our series.

# An easier way: transpose(): -

- An alternate way to create multilevel column axis is by using the **transpose() method**.
- The basic idea here is to combine the **set_index() + transpose()**. The set_index() enables us to create the hierarchical index and then the transpose() turns the dataframe on its head. The columns become the index and the index becomes the columns.
- Suppose we want 2 columns from the dataframe be the 2 levels in the column axis.
- **tech.set_index(['Trading Date', 'Volume Category']).transpose()** → this is how we use the transpose() method to set the 2 columns as 2 level columns in column axis.

# BONUS – What about panels: -

- Hierarchical indices are all about efficiently representing multidimensional data in a two-dimensional data structure, like a data frame.
- Another dimension that is used to represent multi-dimensional data set is the **panel object**. It stores the data in 3D arrays.
- The panels are deprecated and they are no longer part of the modern pandas.
- Prefer **df.MultiIndex** for new projects.

# Group by and Aggregates: -

# Simple aggregation review: -

- The **aggregate functions change the dimension** of the output and converts the dataframe into series of values.
- When we think of aggregation in a dataframe context, it is technically an operation that could be applied in more ways than one.
- Any given **column could be collapsed or aggregated** across all of its rows. Any **row can also be aggregated across** all of its columns also.
- We can control the aggregation by changing the **axis parameter** in the aggregate method like **max(axis = 1)**.

# Conditional aggregates: -

- The aggregate function **by default applies on the entire dataframe**. They applied a function on sequence of values and then return a number. But it was applied across all the rows.
- Oftentimes we need conditional aggregates or rather functions that are applied to a smaller set of values within the dataframe.
- We are still applying the same function, but we want to keep the result from each group separate.
- **sales.sum(numeric_only=True)** → the **numeric_only = True parameter** will give us only the numeric values in the output and discard the string object.
- For calculating the **platform specific sales totals** by first separate the data and then we apply the aggregation function on those smaller dataset like **sales.loc[sales.Platform == "X360"].sum(numeric_only=True)**.
- But this approach is not that efficient and also it requires lot of typing. Also, this doesn’t scale well.

# The split-apply-combine pattern: -

- The approach we have use above for calculating the total sales of some handpicked platforms has a name called as split-apply- combine pattern.

- We split the data, we combine the function and then we combine that intermediate output into a new dataframe. This is the split-apply-combine pattern.
- But we don’t need to do all these steps manually, as the **groupby() function** will do all these steps in one single operation.

# The groupby() method: -

- **sales.groupby('Platform').sum()** → this is how we use the **groupby() method** for calculating the total sum of sales across different Platform.
- All the steps we have used above to calculate the total sales for different Platform can be done in one step using the **groupby() method**.
- We use the **groupby() to apply** the split-apply-combine type of operation all in one go.
- The method splits the dataframe for us by the key that provide.

# The DataFrameGroupBy() object: -

- Think of the dataframe groupby object as the collection of recipes to create the mini datasets and this recipe is waiting to be picked up and executed.
- In computer science, the concept or idea of sitting in the middle knowing exactly what needs to be done, but not yet doing it is called as **lazy evaluation**. Which simply says that no computation or calculation takes place until it is needed or called for.
- As soon as we apply function to the dataframe groupby object, the computation kicks in.
- Applying groupby on the dataframe **returns the dataframe groupby object**.

# Customizing index to group mappings.: -

- Suppose we don’t want to groupby as many unique values as a particular column in the dataframe has. Which is the default behaviour.
- We would much rather consolidate the platforms by type of console. Or we want to groupby using only certain columns.
- This could be very efficiently done by the groupby, if we provide a dictionary mapping index key to the group labels we are interested in.

- **Projection from one column to another** only works if the column we want to groupby is the index column.
- **sales.set_index('Platform').groupby(platform_names).sum()** → this is how we can dynamically change our mappings to project it into different labels without actually effecting the structure of the underlying data.

# BONUS – Series groupby(): -

- So far, we have talked about **groupby in the context of dataframes** but there is nothing about this method or the split-apply-combine pattern that is dataframe specific.
- We can also apply the **groupby()** to the series also.
- **ser.groupby('Genre').mean().sort_values(ascending=False)** → this is how we use the groupby() method with the series also.

# Iterating through groups: -

- To see what is inside the dataframe group by object without applying the aggregate function then it turns out that **we could easily iterate over it** and access each subgroup.
- This is how we can **iterate through dataframes groupby** object: -

  -
    ```
    for name, df in sales.groupby('Platform'):
        print("Subgroup name", name)
        print("------------------------")
        print(df,"\n")
        print("------------------------")
    ```

# Handpicking subgroups: -

- It may actually be quite useful to source an individual group in isolation from all the groups we get by iterating the dataframe groupby object.
- When we select a single column from a dataframe group by object, we go down to a series groupby object.
- One way to extract the subgroup is to convert the entire dataframe object of an iterator like this **dict(iter(sales.groupby('Platform')))** and then we can simply extract from that dictionary.

- **dict(iter(sales.groupby('Platform')))['PS3']** → like this we can extract from the dictionary.
- Another and more efficient approach is by using the **getgroup() method**.
- **sales.groupby('Platform').get_group('PS3')** → this is how we use the **get_group() method** and get a subgroup.

# MultiIndex grouping: -

- Till now we are grouping by only the single key. But now in this section we start grouping by the multiple keys or sets of keys.
- Grouping by single keys will be inadequate if we add another dimension.
- To do grouping by multiple keys we need to pass the list of keys inside the **groupby() method**.
- **studios.groupby(['Genre','Publisher']).sum().sort_values(by="Global_Sales", ascending=False**) → this is how we can pass the multiple keys in the groupby () method.
- The **groupby()** works with the list of keys as well as the single key. And using multiple keys helps us answer very precise questions.
- Whenever we use **groupby() with** multiple keys then the resulting **dataframe contains a multiindex**.

# Fine-tuned aggregates: -

- Now we will look into the aggregation functions and one of the aggregate functions we used in the previous section is **.sum()**.
- We can also use the aggregate functions like **.agg('sum')**.
- Other common aggregate functions are **agg('sum), agg('mean'), agg('median'), agg('mode'), agg('min'), agg('max'), agg('std).**
- In addition, the aggregate function also accepts the numpy ufuncs.
- One of the biggest advantages of using the aggregate function is that **it allows us to apply several functions in one go** because this method accepts the list of python functions as well.
- **studios.groupby(['Genre','Publisher']).agg(['sum', 'count','mean', 'std']).sort_values(by=('Global_Sales', 'sum'), ascending=False)** → this is how we can aggregate multiple functions with the multiindex dataframe.

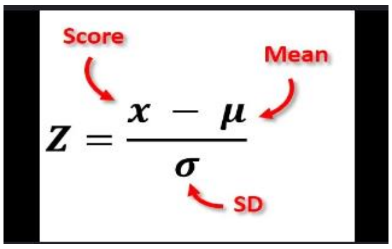
# Named Aggregations: -

- By default, the pandas will assign the aggregate function names as the column labels.
- But if we want something more descriptive then we could always chain on a dataframe **rename() method** to change the name of the dataframe. This method works on both the axis.
- **studios.groupby(['Genre','Publisher']).agg(['sum', 'count','mean', 'std']).rename({'sum': 'total_revenue', 'count':'num_games'}, axis = 1)** → like this we can rename our columns.
- This works great but the groupby() aggregation() combo in pandas also supports the **named aggregations** and this is a special syntax that gives us the optional customizing the output column labels for our aggregates.
- **studios.groupby(['Genre','Publisher']).agg(total_revenue = ('Global_Sales', 'sum'), avg_games = ('Global_Sales', np.mean), revenue_std = ('Global_Sales', 'std'),game_count = ('Global_Sales', 'count'))** → like this we can directly apply names to the columns using the named aggregates.
- Using this method we don’t have multiindex column axis and instead of that we have single column axis.
- We can also apply different aggregation function to different values in a single call to the **.aggregate()** function like this **studios.groupby(['Genre','Publisher']).agg(total_revenue = ('Global_Sales','sum'), average_EU_revenue = ('Global_Sales', 'mean'))**
- We can also pass the dictionary inside the aggregate functions.

# The filter() method: -

- Here we will see how to combine the **groupby() method with the filter() method** in order to exclude records from dataframe based on group level characteristics.
- **games.groupby(['Publisher', 'Genre']).filter(lambda sg: sg['NA_Sales'].sum()>50)** → this is how we use filter() method along with the groupby() to filter the records from the dataframe.

# GroupBy() Transformations: -

- Another class of operations that nicely combine with groupby and those are transforms.
- The transform() method, which is like a filter, also takes a function. It simply applies the transformation in place.
- It gives us access to a whole new array of transforms that could be applied at the subgroup level.
- The standard score concept: -
  - We take any given observation (Xi) – we subtract the mean(u) / divide by the standard deviation(std) and it gives us the sense of how high or low relative to its sample any given observation is.
  - 

- **game_relative.set_index(['Name','Platform']).groupby('Genre').transform(lambda x: (x - x.mean())/ x.std())** → this is how we apply the group by along with the transform() method.
- **aggregate()** in general reduce the dimensions of the dataframe, **filter()** on the other hand simply removes that don't meet the criteria we specify and the **transforms()** changes the values in place without altering the shape without altering or changing the shape of the data frame.

# BONUS – There's also apply(): -

- In addition to the aggregate(), filter() and transform() method, the groupby() productively combine with the generic **apply() method also**.
- And using the **apply() method** we can definitely implement all the functionality that we got from those other methods.
- The paradigm is that → subgroup goes into the apply() method → apply() method applies the function that we specify → then some sort of output comes out and that's something that output could be a dataframe having the same dimension as the subgroup, or it could be a python list or even a single scaler.

- **ps3.groupby('Genre').apply(lambda sg: 'solid' if sg.EU_Sales.sum() > 50 else 'week')** → this is how we use the apply() method with the groupby().
- The **coefficient of variation** is the fancy name of ration of std and the mean i.e**. standard deviation / mean**.
- The function pass to the apply() method takes each of the subgroups as arguments and returns anything really it wants.

# Reshaping with pivots: -

# Pivoting data: -

- Pivots in pandas draw a lot from Excel pivot tables. **Pivots provide a very convenient functional interface** for what could be achieved with more complicated **grouby() and aggregate() methods**.
- **Pivoting is all about reshaping the data**. We take a dataframe or a series and without changing or transforming the content, we present it differently.
- Slicing and dicing the dataset or changing the structure, depending on the question we are trying to answer is a very important skill in data analysis.
- **sat.pivot(index = 'School Name', columns = 'SAT Section', values = 'Score')** → this is how we use pivots to reshape our data with just a single step.

# Undoing Pivots: -

- We can also reset the changes or the structure of the dataframe to remove all the changes that we did when we use pivots.
- **The .stack() method** will take the column axis and rotate it clockwise into the rightmost level of the index axis.
- **pivoted.stack().reset_index()** → like this we can undo the pivoting.
- Another method is to use the **melt() method**. **melt()** is used to change the dataframe to a structure where the measured variables are unpivoted to the row axis.

- **pivoted.reset_index().melt(id_vars='School Name', value_name = ‘Score’)** → this is how we use the melt() method and the id_vars parameter is used to see the identifier parameter.
- Pandas allows duplicates in the index of the dataframe. But the index provided by the **pivot() method** cannot contains the duplicate values given the column configuration specified in the method call.

# The pivot_table(): -

- In the previous section as we try to pivot the dataframe based on the Borough. The dataframe cannot be pivoted as it contains some duplicate columns. The **pivot() method** also don’t support the aggregation. Which would be require here, given our choice of index and column axis.
- To do this we use the **pivot_table() method**. To do aggregation across some column we use this.
- **sat.pivot_table(values='Score', index='Borough', columns='SAT Section', aggfunc=’std’)** → like this we can use the **pivot_table() method**. The **aggfunc parameter** takes what aggregation we have to perform on the dataset and it is by default set to ‘mean’.
- We choose **pivot_table()** whenever we need some data aggregations and calculations that reduce the dimension of our dataset.

# BONUS – The problem with the average percentage: -

- Averages of percentages, equal weighted averages of percentages are very tricky and potentially very misleading.
-

# Replicating Pivot table by groupby(): -

- The pivot_table() method is very great and has a very intuitive interface.
- We can also replicate what we did using the pivot_table() with the default groupby() method also.
- **sat.groupby(['Borough', 'SAT Section']).agg({'Score':'mean'}).unstackck()** → like this we can also achieve what we achieved using the pivot_table().

- All the **pivot_table()** functionality could always be replicated using a combination **of groupby() and aggregate().** Using pivot_table() method results in cleaner or shorter code.

# Adding margins: -

- Here we will see how to add some row or column totals to our pivot_tables().
- The **pivot_table()** gives us an easy way to tag these totals onto our dataframe by simply setting the **margins parameter** to True.
- **sat.pivot_table(values='Score', index='Borough', columns='SAT Section', margins=True)** → this is how we get the totals along all the rows and columns.
- **sat.pivot_table(values='Score', index='Borough', columns='SAT Section', aggfunc='max', margins=True, margins_name='Max')** → this is how we can also change the margin name using the **margins_name parameter**.

# MultiIndex pivot tables: -

- Here we will see how to leverage the syntax offered by **pivot_table()** to create multiindex dataframes with relative ease.
- **sat.pivot_table(values='Score', index=['School Name','Borough'], columns='SAT Section')** → this is how we can make multi-index using the pivot_table() method.
- we can also move the multi-index to index dimension to the column dimension also.
- **sat.pivot_table(values='Score', columns=['School Name','Borough'], index='SAT Section')** → like this.
- Using **pivot_table() method** we can create multiindex dataframes along both the axis.

# Applying multiple functions: -

- Oftentimes it may be useful to be able to apply several aggregation functions at once.

- We can easily achieve this by **passing the list of aggregate functions** to the **aggfunc parameter**.
- **sat.pivot_table(values='Score', index='Borough', columns='SAT Section', aggfunc=[np.min, np.max])** → this is how we do this. We can pass the numpy ufuncs of pass the normal aggregate functions.

# Handling time and date: -

# The python date time module: -

- **datetime** is a module that comes with a **python standard library**. If we have python then we have **access to datetime** but it is **not in our namespace**.
- datetime is used to manipulate dates and times and it comes with a lot of built-in classes.
- To use the datetime module, we have to import it in the notebook using the syntax **from datetime import date** and it will import the date class of the datetime module.
- Using date, we can create date objects that contains date information and a date is characterized by the **year, month and day**.
- We are storing the dates in the datetime object like this **date(2024,6,21)** but not in strings because the datetime objects are easily to manipulate, combined, changed or updated using the methods and attributes that are available for them.
- This unlocks a lot of functionality, which would be really difficult for us to replicate strings. Strings in general are very generic.
- **from datetime import time** we can also use the time class of the datetime module and it stores the time in the format **hour, minute, seconds, microsecond**.
- **time_A.isoformat()** → this will convert the time into isoformat.
- The datetime module also has a datetime class. **from datetime import datetime**. This class combines both the date and the time objects. It contains or stores the information of both the date and the time object.
- **datetime.now()** → this will give us the current date and time.

# Parsing dates from text: -

- The datetime module has a method dedicated to extracting dates out of text and that is **strptime()** it stands for **string parse time**.
- **datetime.strptime('2024-06-22', '%Y-%m-%d')** → this is how we get the dates out of the string by specifying the format in which we want dates.
- The format codes should be specific and we can find the different formats in the official python documentation also.
- The spacing in the format definition is very important because it mirrors, it reflects the spaces in the underlying string.

# Even better: dateutil: -

- It is an easier method for converting dates. This is a module that is external to the standard python library.
- **!pip show python-dateutil** → this is how we can see if the dateutil is present in our notebook or not.
- **!pip install python-dateutil** → this is how we can install the dateutil library in notebook itself.
- We need to import the **parser class** from the **dateutil module** to parse the dates. **from dateutil import parser**.
- Using the **parser()** we can easily convert the text into dates **without needing to define the structure** of the date like we are doing in **strptime().**
- **parser.parse("jan 21st 1910")** → this is how we convert the string to dates using this module. We can even parse complex strings to the dates using this class.

# From datetime to string: -

- We can also create a custom string representation from the datetime object.
- To do this we will use the **strftime() method** from the **datetime module** and this stands for **string format time**.
- **dt.strftime('Year: %Y; Month: %m; Day: %d')** → this is how we can convert the datetime object into string as per our need by specifying the correct format codes.

- **dt.strftime('%c')** → this will give us the complete representation of the date in the string.
- **"My date is {:%c}".format(dt)** → we can also do like this to get the string representation of the date.

# Performant datetimes with numpy: -

- If we need to manipulate large scale datasets of date times, or if we want to operate on massive arrays or dates and times, the pure python approach will prove rather slow.
- To work around this, numpy has created a special data type that encodes and stores the datetime info. much more efficiently and enables data analysts and scientist to conduct large scale vectorized operations on date.
- **np.datetime64('2024-06-21')** --< this is how we can create a numpy datetime object.
- We can change the time unit by rescaling the numpy datetime. E.g. if we want to rescale our datetime **b = np.datetime64(datetime.now())** to have daily time scale the we can do this like **np.datetime64(b, 'D').**
- The numpy datetime has fixed length of 64 bits. But by changing the time unit, we are changing the precision of our dates. and when we lower the precision like from milliseconds to day, we increase the timespan.
- Theres a trade off between precision and span.
- The power of numpy d type approach is that it allows us to perform efficient vectorized operations on collections of dates like: -
    - **dates = np.array([**
      '2019-02-20',
      '2019-06-20',
      '2019-03-23'
      ], dtype = np.datetime64)
    - **dates-10 → this will easily subtract 10 from the array of days.**

# The pandas Timestamp: -

- The **pandas Timestamp()** is a combination of both **python datetime and the numpy datetime64**. The simplicity of python datetime with the performance and rich interface of numpy datetime64

- **pd.Timestamp("4th of July 1776")** → this is how we use the pandas Timestamp().
- **pd.to_datetime('4/7/1776', dayfirst=True)** → using the **to_datetime()** in pandas we can also parse the strings into dates and the **dayfirst parameter** represents that take the first number in the string as the day in the date.
- **Timestamp** is the fundamental building block of all time and date operations in pandas.
- We can also use many methods and attributes on this Timestamp object in pandas like **.day_name(), .days_in_month, .quarter** etc.
- We can also represent the date in a three item tuple containing year, week and day using **.isocalendar()** method.
- It serves as the replacement of the python datetime module and under the hood its based on numpy datetime64.

# Date Parsing and DateTimeIndex: -

- To convert the object type into dates we do this by **casting the date to the numpy datetime64**.
- **brent['Date'] = brent['Date'].astype('datetime64[ns]')** → this is how we can change the data type of the date column from object to datetime64. Here **ns is the nanosecond** resolution and it is the smallest unit of time that could be stored in this object. Second = 1billion nanosecond.
- After converting from object data type to datetime64 the memory_usage also decreases from 360kb to almost 70kb. Hence the datetime is lot more effectively stored in memory than the object type.
- The **DatetimeIndex is a special immutable** pandas data structure that we could think of as a collection of pandas timestamps.

# A Cool shortcut: read_csv() with parse_dates: -

- Another approach of achieving what we did above and we do this during the reading of file as we import the data.
- It turns out that we could identify the index and parse the dates at the moment that we import the data into pandas.
- **parse_dates = True** in **read_csv is a parameter** that indicates that we want to convert our index to try to parse dates from our index.

- **pd.read_csv("D:/STUDIES/Numpy & Pandas /BrentOilPrices.csv",index_col=0, parse_dates=True)** → like this.

# Indexing dates: -

- All the ways through which we can select the data we explore here.
- First → the label based indexing using the **.loc[]** indexer like this **brent.loc['2017-01-03': '2017-01-06']**.
- Second → partial string indexing e.g. **brent.loc['2019-01']** this will give us all days in the month of January. This is called as partial string indexing.
- We can also combine the partial indexes with the slices e.g. **brent.loc['2019-01': '2019-02']**. Also like this **brent.loc['2019-07': '2019-08-15']**.

# DatetimeIndex attribute accessors: -

- In this section we will take a look into few attribute accessors available on datetime indices and that looks like generic properties.
- **brent.index.quarter** → this will give us the quarter of each date.
- **brent.index.week** → gives the week
- **brent.index.month** → gives us the month
- **brent.index.weekofyear** → gives us the week of the year
- **brent.index.day_name()** → gives us the name of day.
- All of these attributes and methods are directly available on our date time index object.
- The **DatetimeIndex objects** are also useful in dynamically creating Boolean masks.
- **brent[(brent.index.is_leap_year == True) & (brent.index.month == 2)].mean()** → the **.is_leap_year()** method is used to check if the given year is a leap year or not and returns a Boolean value.

# Creating Date Ranges: -

- Sometimes we need to construct a date-based index or dataframe manually from scratch.

- The method that is very flexible in generating the date ranges is the **date_range() method**.
- **pd.date_range(start='10 may 2020', end='20 june 2020')** → this is how we generate date ranges and the formatting of the dates can be different in many forms like **pd.date_range('5/10/2020', '20 june 2020')**.
- The date_range() method has many other flexible ways to define and create ranges. For instance, instead of providing a start and an end date, we can also specify the start date and the number of periods we want like **pd.date_range(start="1/20/2020", periods=10)**.
- We can also specify the frequency of dates like **pd.date_range(start="1/20/2020", periods=10, freq='W')**.
- The frequency parameter also supports multiple frequency values.
- The date_range() method is quite useful for creating daytime indices with a fixed freq.

# Shifting Dates with pd.DateOffset: -

- Occasionally we might need to **shift or adjust our date times or date ranges** by a given amount of time and in pandas, this type of transformation could be easily carried out with the help of the **pd.DateOffset object**.
- **pd.DateOffset(days=18)** → think of this object as encapsulating a period of 18 days. It captures the difference of 18 days.
- **dob - pd.DateOffset(days=18)** → this is how we can carry the operation.
- The DateOffset object could not only store dates, we could specify any offset we want, just typing out the temporal parameters we need.
- DateOffset supports many temporal parameters like **pd.DateOffset(days = 4, minute = 10, nanoseconds = 2)**.
- **brent.set_index(brent.index + pd.DateOffset(hours = 18))** → this is how we can use the **DateOffset() method**.
- If we want to perform data Arithmatics then we should really consider the DateOffset**.**

# BONUS – Timedeltas and absolute time: -

- Previously we did some date addition and subtraction by first creating standalone **Dateoffset()** and them combine them with the existing datetimes.
- Here we will explore the **concept of creating object** that stores the time differences using the **Timedelta class**.
- **pd.Timedelta(days = 3, hours = 4) →** this is how we use Timedelta class and it is almost similar to the Dateoffset() method.
- The difference is that the **Timedelta class operates on absolute time** where the **DateOffset() operates on calendar time/ or it is calendar aware**.
- **Timedelta always considers** a day to be 24 hr long. But the **Dateoffset** works depending on the time zone, time of the year. A day may be 23, 24 or 25 hrs long.
- **dst = pd.Timestamp('14 mar 2021', tz = 'US/Eastern') →** the **tz parameter** is used to pass the time zone of the place where we want to apply the timestamp.
- Both the **Timedelta** and **Dateoffset** calculate the differences in time.

# Resampling Timeseries: -

- The process of altering the frequency of a time series is known as resampling.
- Reducing the frequency is also known as **down sampling** like going from month to day.
- In pandas the method used for resampling is the **.resample() method**.
- **brent.resample('M') →** like this we can set the offset alias in which we want to resample our data and we can find a list of them in the pandas documentation.
- The resample() method will return the **DatetimeIndexResampler object**.
- When down sampling time series, the data we go from many data points to fewer and far between. We are reducing the dimension of the data and pandas needs to know what aggregation function will be used to go from these dimension.
- **brent.resample('M').median() →** this is how we do resampling().

- One of the great benefits of having a datetime index is that it will **automatically be treated as the x-axis** by matplotlib which simplifies the plotting.
- The resample() method is collecting a bunch of observations, its applying an aggregation function to reduce them into a single number.
- Whenever we need to change the frequency of our time series data, we use the resample.

# Upsampling and interpolation: -

- Previously we use the **resample() method** to down sample our series using various offset aliases like ‘M’ for monthly or ‘10D’ for 10 days.
- Here we will do the opposite of **resample().** In other words starting with the time series and then increasing its frequency of observation and this is known as **upsampling**.
- In **downsample** we went from many observations to fewer and we do this with the help of some aggregation function.
- When we **upsample**, we face different type of challenge because we are going from few observations to many more and we use interpolation here.
- Now we do the interpolation to fill the gaps in the data or to replace the NA values.
- In linear interpolation to key assumption is that the distances are equal or the items are equally spaced.
- **brent.resample('8H').interpolate(method='linear')** → this is how we do interpolation and fill the gaps for missing values in the datetime series.
- There are many ways of interpolating in pandas other than linear i.e. **spline**.
- If we increase the frequency of observations in a time series beyond the granularity of observations offered. We usually have to specify the type of interpolation we want pandas to apply.

# What about asfreq(): -

- Pandas also support another method that is **quite useful in adjusting the frequency of time series** and that is the **asfrac() method**.
- The **asfrac() method** also has some built-in capabilities that help us fill in the gapes that occurred.

- **brent.asfreq('10D', method='ffill')** → this is how we use the **asfreq** method and use the fill parameter to fill the NA values. It takes the previous number and paste it in the current NA position.
- **brent.asfreq('10D', fill_value = 12)** → this is how we use the **fill_value parameter** to fill the null or NA values.
- The **resample()** and the **asfreq()** method are not similar despite or having many common functionalities.
- Using the asfreq() method we get the series immediately while in the resample() method we get a resampler object.
- The **asfreq()** method resamples the data according to the frequency that we specify, but it merely selects from the data from the entire series.
- The **resample()** method groups the data according to the frequency we specify, and then it opens up those groups to any transformation and aggregation that we are applying.
- If we need to select from our dataframe, from a timeseries select bunch of items according to the frequency that we specify then we use **asfrac() method**.
- If we need to condense the info. we have according to the frequency, we use resample.

# BONUS: Rolling Windows: -

- Another concept when working with time series is this idea of rolling windows.
- Rolling window is just a collection of observations combined with an aggregate function, and its called rolling because it rolls forward.
- **brent.rolling(3).mean()** → this is how we calculate the rolling window.
- Rolling or moving averages are really useful in smoothing out the data and removing the noise from the time series.

Data Formats and I/O: -

Reading JSON: -

- JSON is a text-based interchange format that is widely used in storing and exchanging information between applications that communicate over the internet. A lot of APIs nowadays serves data in the JSON format.
- JSON stands for JavaScript object notation. It is a standardized and language independent format.
- At the top-level JSON is nothing but a collection of key value pairs. The other important structure in JSON is ordered list which is similar to python lists. They just contain the sequence of values separated by commas.
- To read a JSON file we have to write **pd.read_json('path')**. It will give us a dataframe object only. Where each of the key in JOSN will become a column in a dataframe.

Reading HTML: -

- Another data format is HTML i.e. hypertext markup language. Every online website is based on HTML.
- If we inspect a web page then the browser will give us all the HTML that is responsible for generating that web page.
- A lot of info. on the web is available on the internet in the form of HTML tables.
- Pandas has the ability to ready data from the HTML tables directly from the web page.
- So to read the table → first copy the url of the website → to read the HTML in pandas we use the method **pd.read_html('url')** → the data will load into the pandas and from there we get a list of elements and then we simply access the list we want.

# Reading Excel: -

- A excel spreadsheet consist of grid of cells arranged in numbered rows and lettered columns. A excel workbook can have more than one excel worksheet.
- The default format to save excel is **.xls** and it is a proprietary binary format created by Microsoft but newer version of excel support xml-based format and uses the extension **.xslx**.
- To read the data from excel file to pandas we use the method **pd.read_excel(")**.
- To read another worksheet from the same workbook we use the method **sheet_name parameter**. Like this **pd.read_excel("D:/STUDIES/Numpy & Pandas/folks.xlsx", sheet_name='hobbies')**.

# Creating output: The to_* Family of methods: -

- So far, we have only explored the input side of I/O, we have seen how to ready in CSV files, HTML files, JSON files and excel files. And the methods we have used are lot common in terms of syntax.
- But reading is only half part. After we manipulate, analyze, transform and filter the data, we might be interested in exporting the data to one of the other formats.
- **np.random.uniform()** → Draw samples from a uniform distribution.
- For every read method in pandas, there is a corresponding write method that starts with the prefix **to_{name}**.
- **hobbies.to_csv('hobbies.csv', index = False)** → this is how we create a csv file from a dataframe.
- **pd.read_csv('hobbies.csv')** → this is how we again read the data back into the pandas notebook.

# BONUS – Introduction to pickling: -

- Serialization is the process of converting the object into bytes, into a stream of bytes that could be stored in a file or memory or transmitted over a network.

- By serializing only, the python object that we want to share, we make sure that its exact structure and content is captured and converted into a stream of bytes so that we or someone else is able to restore the same object at a later time.
- In python, object serialization is called as pickling and we have a module that comes with the standard library in python.
- To use that we first have the **import pickle**.
- In order to pickle an object, we first have to create a file that's going to hold that object like this **pickle_output = open('my_stock', 'wb').**
- After that put our object into this file like **pickle.dump(googl, pickle_output)**.
- Lastly we make sure that we close the file we open and dumped some data into like **pickle_output.close()**.
- This file is not readable and to bring the file back to python we have to use the **open() function** with the **'rb' mode** like **pickle_input = open('my_stock', 'rb').**
- **pickle.load(pickle_input)** → we use this method to load the stream of bytes back into how we saved it.
- Serialization is also known as marshalling or flattening.

# Pickles in pandas: -

- Pickling is very popular in data science because it allows the analyst to persist complex python objects to store them and then quickly restore them back from the pickle instead of going through the process of creating them from scratch each time.
- We can read pickles in pandas using the **pd.read_pickle() method**.
- We can create the pickle from the dataframe by directly using the **to_pickle('file_name') method**.
- To read the file in the pickle we use the method **read_pickle('file_name') method**.
- We should not load pickles that we receive from untrusted sources as it is possible that it will contain the malicious code that could execute immediately after its unpickled.
- For data-interchange over the web, JSON may be better data interchange format.

# The many other formats: -

- There are many more formats that pandas could directly read in as well as write to.
- To see all the formats go to the I/O tools section in pandas' documentation which contains all the reader and writer functions currently supported by pandas.
- Pandas has the ability to read directly from the SQL databases, including the google's big query.
- **positions.merge(traders, left_on='traderID', right_on='alias')** → this is how we merge the dataframes.
- To download files from the google colaboratory: -
    - **from google.colab import files**
      files.download('traders.csv')

# Regex and text manipulation: -

# Intro: -

- In this section we will see how to extract text information from the text data and transform it using the array of methods, including case operations, splits and more.
- We begin with string manipulation in pure python. The key theme in this section is that pandas borrow heavily from pythons.
- We will also deep dive into regular expressions to handle more advanced string patterns.

# String methods in python: -

- Some of the basic string methods in python includes: -
    - **len()**
    - **.startswith()**
    - **.endswith()**

# Vectorized string operations in pandas: -

- **.str** is the common postfix attribute that allow us to access vectorized string operations. We can use it like **boston.Name.str.len(), boston.Name.str.startswith('A')**.
- Vectorized string methods in pandas usually follows the same naming convention as built-in python strings.
- The vectorized string functions using **.str** simply excludes the null values of NA’s.

# Case operations: -

- Here we will explore the methods that effect the case. Weather the characters or words are in uppercase or lowercase.
- Some the case operations are:-
  - **.title()** → it will capitalize the 1st character of the string
  - **.upper()** → convert the entire string in uppercase.
  - **.lower()** → convert the entire string in lowercase
  - **.swapcase()** → swaps the cases of the string.
  - **.capitalize()** → capitalize the 1st char of the string and convert rest in lowercase.

# Finding characters and words: -

- A popular method that helps us locate substrings or characters within a given text is the **.find() method**.
- **s.find('x')** → this is how we use this method and It will give us the position or index of the character in the string and if it doesn’t find the string then it will return -1.
- **boston.Name.str.find('Andy')** → this is how we use the .find() method with the dataframes also.
- The **.find()** method performs the left to right search and the **.rfind()** method perform the right to left search.

# Strips and whitespaces: -

- Whitespace is the general term that refers to characters that represents vertical or horizontal space. These characters are usually not visible when a string is printed but they do affect the spacing or positioning of the result output.
- To check whether a given character is a whitespace, we could use the **.isspace() method** in both python and pandas.
- Python and pandas offer a very handful or useful method to strip whitespace of text and those are: -
    - **.lstrip()** → removes leading whitespace
    - **.rstrip()** → removes whitespaces from right side
    - **.strip()** → removes whitespaces from both the end

# String splitting and concatenation: -

- Here we will study the method that takes the string or a piece of text and then break it down into smaller strings based on a break point that we specify and that is **.split() method**.
- By default, this method splits the string based on the white spaces.
- We can split on anything we want. **s.split('s')** → this is how we can split based on the character too.
- To extract the element from each item in the series, we could chain a specialized Pandas string method, i.e. **.get()**.
- **boston.Name.str.split(', ').str.get(0)** → this is how we use the .get() method to get the 1st element in the series list.
- To do the opposite of this, means to do the string concatenation pandas has a method i.e. **.cat() method**.
- **boston['M/F'].str.cat(boston.Age.astype('str'), sep='_')** → this is how we use the **.cat()** method for concatenation.

# More split parameters: -

- **boston.Name.str.split(',', expand=True)** → the **expand = True** parameter will end up with a dataframe that has as many columns as there are split substrings as a result of the output.
- **boston[boston.Name.str.split(expand=True).count(axis=1) == 5]** → this is how we can use the expand parameter to get the count of the names who have length == 5.
- **boston.Name.str.split(expand=True, n = 1)** → the **n = 1 parameter** specifies that we only want 1 split to happen.
- Now to incorporate the columns that we get after the expand parameter to our original dataset then **boston[['First_Name','Last_Name']] = boston.Name.str.split(', ', expand=True)** → this is how we can directly append the columns to our original dataframe.

# Slicing Substrings: -

- Slicing is extracting the pieces or slices of text from an existing string.
- In pure python, we can achieve this through the combination of slice() object and the [] square brackets like this **s[slice(0, 7, 1)]**.
- **s[0:7:1]** → this is the shorthand to do slicing in python.
- **boston.Country.str.slice(0,2,1)** → this is how we do slicing in pandas.

# Masking with string methods: -

- Pandas has a dedicated string **.contains() method**, that also **supports regular expressions**.
- So far, we have discussed string methods in the **context of manipulating or changing text**, but lot of these methods in pandas are also exceptionally useful in **quickly filtering and searching for data**.
- The panda's way of doing dataframe filtering is by using Boolean masks.
- **boston.Name.str.contains("Will")** → this is how we use the **.contains() method**.

# BONUS: Parsing Indicators with get_dummies(): -

- This method is used to create **indicators variable** out of **categorical variables** which are stored as text.
- In pandas the operation of converting a categorical variable into dataframe of **binary indicator variables** can be done with a dedicated method in pandas i.e. .**get_dummies() method**.
- We cannot do the conversion inplace because there in no inplace parameter it supports.
- **boston['Years Ran'].str.get_dummies(sep=':')** → this is how we use the .get_dummies() method to separate the categorical data and make them a dataframe.
- **boston.insert(boston.columns.get_loc('Years Ran'), 'Ran 2016', dummies['2016'])** → this is how we use the **.insert() method** to insert a column in the existing dataframe.

# Text replacement: -

- String replacements are very useful in transforming, cleaning and reshaping the text data.
- Here we will focus on string replacement using exact character sequences.
- **boston['M/F'].str.replace('M', 'Male').str.replace('F', 'Female')** → this is how we use the **.replace() method** in pandas to do string replacements. We can also chain the **.replace() method**.
- **boston.Country.str.replace('UsA', 'United States', case=False)** → we can use the case = False parameter to ignore the case while replacing the string.
- **s.replace('text', 'string', 1)** → while replacing regular python strings we can also specify how many replacements we want if there are more that one similar words.

# Introduction to Regex: -

- Regular expressions **allow us to specify the text patterns**, which could be later **used to find, replace or split text** into substring.
- Regex is not python specific, many programming languages implement some version of regex with slight variations.

- We can think of regex as a **domain specific language (DSL**). Where the domain is text manipulation.
- The Regex allow us to create highly complex general string patters, using a concise set to tokens and characters.
- **Meta characters** specify the type of character, we are looking to match. **Quantifiers** specify how many characters of that type we are looking for.
- **Character set** are use to indicate a specific character that we want to match.
- **Anchors** are used to control the word and line boundaries in general.
- In python all the regex functionality is accessed from the **re module** which comes with the python standard library. We can use it by importing it in our notebook like **import re**.

# Pandas str contains(), split() and replace() with regex: -

- In pandas these methods can also supports regex patterns also, which open up another level of functionality and introduces new ways to interact with our data.
- **boston.Name[boston.Name.str.contains(r",\s[Ww]ill$",regex=True)]** → this is how we use the **.str .contains() with the regex also**.
- **boston.Name.str.split(r'\s',expand=True)** → this is how we can also use regex with the **.split()** also.
- **boston['Official Time'].str.replace(r'(\d+):(\d+):(\d+)', r'\1 hours, \2 minutes, and \3 seconds', regex=True)** → this is how we use the regex with the **.replace() method** of string also.
- **boston.insert(boston.columns.get_loc('Official Time')+1, 'Total Time', final_times.total)** → this is how we insert a column directly into the dataframe using the **.insert() method**.