Pandas: Python Libraries for Data Science

Pandas:

Pandas is an open-source Python Library used for high-performance data manipulation and data analysis using its powerful data structures.

- adds data structures and tools designed to work with table-like data

- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.

- Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, organize, manipulate, model, and analyse the data.

- ıntegrated with many other data science & ML Python tools
- Helps you get your data ready for machine learning
- allows handling missing data

What are we going to cover?

- pandas Datatypes
- Importing & exporting data
- Describing data
- Viewing & selecting data
- Manipulating data

pandas Datatypes

Pandas has two main data types:- **Series and DataFrame**.

- Series:-   a 1-dimensional column of data.
- Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 32, 65, …

10	32	65	17	52	61	80	90	89	90

```
import pandas as pd
```

- You can create a Series using pd.Series() and passing it a Python list.

```
cars = pd.Series(["BMW", "Toyota", "Honda"])
print(cars)
# Creating a series of colours
colours = pd.Series(["Blue", "Red", "White"])
colours
```

DataFrame

DataFrame (most common) – A DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

- **Dictionary in python**

Dictionaries are used to store data values in key:value pairs.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

DataFrame

- You can create a DataFrame by using pd.DataFrame() and passing it a Python dictionary.
- Let's use our two Series as the values.
- *# Creating a DataFrame of cars and Colours*

```
car_data = pd.DataFrame({"Car type": cars, "Colour": colours})
print(car_data)
```

Exercises

- Make a Series of different foods. Make a Series of different dollar values (these can be integers) and combine your Series's of foods and dollar values into a DataFrame.

- **Note:** Make sure your two Series are the same size before combining them in a DataFrame.

Importing Data

Read CSV Files

- A simple way to store big data sets is to use CSV files (comma separated values files).
- CSV files contains plain text and is a well know format that can be read by everyone including Pandas.
- pandas allows for easy importing of data like this through functions such as pd.read_csv() and pd.read_excel() (for Microsoft Excel files).
- Data files can be transaction details, patient details, sale information, etc.

Import car sales data

- import pandas as pd
- car_sales = pd.read_csv(“file name with complete path") # takes a filename as string as input
- car_sales will display data in the CSV file and it is exactly a data frame

Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the `head()` method.

The `head()` method returns the headers and a specified number of rows, starting from the top.

```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.head(10))
```

.loc[]

- It is used select data from your Series and DataFrames
- .loc[] refers to index takes an integer as an input and it choses from your Series or DataFrames whichever index matches the number

- Also, we can give a range
- df.loc[3:5]

There is also a `tail()` method for viewing the *last* rows of the DataFrame.

The `tail()` method returns the headers and a specified number of rows, starting from the bottom

Info About the Data

The DataFrames object has a method called `info()`, that gives you more information about the data set.

Example

Print information about the data:

```
df.info()
```

The result tells us there are total number of rows and columns, and the name of each column, with the data type. It also contains no of not null values in each column.

TRY these also

df.sum()- Sum all the values in a numeric column, and append the text/string

df.mean()- Does the mean of only numeric columns

These two works on numeric Series, if applies on DataFrame then considers only numeric column

`describe()` method

It returns description of the data in the DataFrame.

If the DataFrame contains numerical data, the description contains these information for each column:
count - The number of not-empty values.
mean - The average (mean) value.
std - The standard deviation.
min - the minimum value.
25% - The 25% percentile*.
50% - The 50% percentile*.
75% - The 75% percentile*.
max - the maximum value.
*Percentile meaning: how many of the values are less than the given percentile.

Percentiles are used in statistics to give you a number that describes the value that a given percent of the values are lower than.

Example: Let's say we have an array of the ages of all the people that live
in a street.

```
ages =
[5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,
61,31]
```

What is the 75. percentile? The answer is 43,
meaning that 75% of the people are 43 or younger.

max_rows

The number of rows returned is defined in Pandas option settings.

You can check your system's maximum rows with the `pd.options.display.max_rows` statement.

Increase the maximum number of rows to display the entire DataFrame:

-
  ```
  import pandas as pd

  pd.options.display.max_rows = 9999

  df = pd.read_csv('data.csv')

  print(df)
  ```

To view particular column(s)

car_sales = pd.read_csv("car-sales.csv")

car_sales

car_sales['Make’]// This is going to provide “Make” column from car sale data

# Select cars which are made by Toyota

car_sales[car_sales["Make"] == "Toyota"]

# Select cars with over 100,000 on the Odometer

car_sales[car_sales["Odometer (KM)"] > 100000]

Group by a particular column and doing aggregate function

- # Group by the Make column and find the mean of the other columns
- car_sales.groupby(["Make"]).mean()

Plotting of columns

pandas even allows for quick plotting of columns so you can see your data visually.

To plot, we need to import matplotlib.

```
import matplotlib.pyplot as plt
```

We can visualize a column by calling .plot() function (This is only applicable For numeric column(s)

Manipulating data

You can access the string value of a column using .str
Knowing this, we can set a column to lowercase

To display lower case for the Make column of Car_sales

```
car_sales["Make"].str.lower()
```

```
# Set Make column to be lowered
car_sales["Make"] = car_sales["Make"].str.lower()
car_sales.head()
```

Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- Missing values in GroupBy method are excluded
- Many descriptive statistics methods have *skipna* option to control if missing data should be excluded . This value is set to *True* by default

Handling missing values

We have to decide whether how to fill the missing data or remove
the rows which have data missing.

.fillna() is a function which fills missing data.

Missing values are shown by NaN in pandas. This can be considered equivalent to None in python

.fillna() function can be used to fill a column with the average values of the other values in the same column

```
 # inplace is set to False by default
# Fill the Odometer missing values to the mean with inplace=False
car_sales_missing["Odometer"].fillna(car_sales_missing["Odometer"].
mean(), inplace=False)
```

```
# Fill the Odometer missing values to the mean with inplace=True
car_sales_missing["Odometer"].fillna(car_sales_missing["Odometer"].mean(),
inplace=True)
```

In practice, you might not want to fill a column's missing values with the mean, but this example was to show the difference between inplace=False (default) and inplace=True.

Dropping a row having missing data

If we want to remove any rows which had missing data and only
 work with rows which had complete coverage.

It can be done using .dropna() method

This method has inplace=False, means changes will shown and not reassigned in DataFrame

Additional column creation on DataFrame

For example, creating a column called Seats in car-sales file for number of seats.

pandas allows for simple extra column creation on DataFrame’s.

Three common ways for doing this are

- adding a Series,
- Python list or
- by using existing columns.

Exporting data

After you've made a few changes to your data, we may want to export
it and save it so someone else can access the changes.

pandas allows us to export `DataFrame`'s to `.csv` format
using `.to_csv()` or spreadsheet format using `.to_excel()`.

Concatenating Objects

- The **concat** function performs concatenation operations along an axis. Let us create different objects and do concatenation.

```
import pandas as pd
 one = pd.DataFrame({ 'Name': ['Alex', 'Amy', 'Allen’],
'subject_id':['sub1','sub2','sub4'],
 'Marks_scored':[98,90,87]},index=[1,2,3])

two = pd.DataFrame({ 'Name': ['Billy', 'Brian', 'Bran'], 'subject_id':['sub2','sub4','sub3’,],
 'Marks_scored':[89,80,79]}, index=[1,2,3])
 pd.concat([one,two])
```

Concatenating Using append

- A useful shortcut to concat are the append instance methods on Series and DataFrame. These methods actually predated concat.

- The **append** function can take multiple objects as well .