PANDAS

Avijit Bose
Assistant Professor
Department of Computer science & Engineering,
MCKV institute of engineering,Liluah,howrah-711204

Pandas

- Pandas is a python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring and manipulating data.
- Pandas allow us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets and make them readable and relevant.
- Relevant data is very important in data science.

Pandas

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

How to install pandas

- C:\Users\Your Name>pip install pandas

How to import pandas

- import pandas

- mydataset = {
- 'cars': ["BMW", "Volvo", "Ford"],
- 'passings': [3, 7, 2]
- }

- myvar = pandas.DataFrame(mydataset)

- print(myvar)

Checking for pandas version

-
  ```
  import pandas as pd
  print(pd.__version__)
  ```

Pandas Series

Pandas series is like a column of a table

It is a one dimensional array holding data of any type.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

```
print(myvar[0])
```

Creating Labels

- With the index argument, you can name your own labels.

```
import pandas as pd

a = [1, 7, 2]

 myvar = pd.Series(a, index = ["x", "y", "z"])
    print(myvar)
```

When you have created labels, you can access an item by referring to the label.

```
print(myvar["y"])=
```

Key value Objects in Series

- We can also use a key/value object, like a dictionary, when creating a Series.

```
import pandas as pd

calories = {"day1": 420, "day2": 380, "day3": 390}

myvar = pd.Series(calories)

print(myvar)
```

Key value objects in dictionary

- To select only some of the items in the dictionary, use the index argument and specify only the items you want to include in the Series.
- Create a Series using only data from "day1" and "day2":

```
import pandas as pd
    calories = {"day1": 420, "day2": 380, "day3": 390}
    myvar = pd.Series(calories, index = ["day1", "day2"])
   print(myvar)
```

Data Frames

- Data sets in Pandas are usually multi-dimensional tables, called DataFrames.
- Series is like a column, a DataFrame is the whole table.
- Create a DataFrame from two Series:

```
import pandas as pd
     data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
myvar = pd.DataFrame(data)
print(myvar)
```

What is a data frame

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.
- Create a simple Pandas DataFrame:

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df)
```

Locate Row

- As we can see from the result above, the DataFrame is like a table with rows and columns.
- Pandas use the loc attribute to return one or more specified row(s)

Return row 0:

```
#refer to the row index:
print(df.loc[0])
```

Locate row

- Example
- Return row 0 and 1:

```
#use a list of indexes:
print(df.loc[[0, 1]])
```

Locating named indexes

- Named Indexes
- With the index argument, you can name your own indexes.
- Add a list of names to give each row a name:

```
import pandas as pd
     data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
   df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
   print(df)
```

Locating Named indexes

- Use the named index in the loc attribute to return the specified row(s).
- Return "day2":

```
#refer to the named index:
print(df.loc["day2"])
```

Loading files into a data frame

- Load a comma separated file (CSV file) into a DataFrame:
-
  ```
  import pandas as pd
  df = pd.read_csv('data.csv')
  print(df)
  ```

Read CSV Files

- A simple way to store big data sets is to use CSV files (comma separated files).
- CSV files contains plain text and is a well know format that can be read by everyone including Pandas.
- In our examples we will be using a CSV file called 'data.csv'.

Read csv file

- Load the CSV into a DataFrame:

```
import pandas as pd

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

print(df.to_string())
```

Read csv file

- By default, when you print a DataFrame, you will only get the first 5 rows, and the last 5 rows:

```
import pandas as pd

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

print(df) # if data set is huge then only first 5 and last 5
```

Reading JSON File

- Big data sets are often stored, or extracted as JSON.

- JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.

- In our examples we will be using a JSON file called ‘avi.json'.

Reading JSON File

```
import pandas as pd

df = pd.read_json('data.json')

print(df.to_string())
```

Dictionary as JSON

- JSON = Python Dictionary

- JSON objects have the same format as Python dictionaries.

Example

```
import pandas as pd

data = {
  "Duration":{
    "0":60,
    "1":60,
    "2":60,
    "3":45,
    "4":45,
    "5":60
  },
  "Pulse":{
    "0":110,
    "1":117,
    "2":103,
    "3":109,
    "4":117,
    "5":102
  },
```

Example

```
 "Maxpulse":{
   "0":130,
   "1":145,
   "2":135,
   "3":175,
   "4":148,
   "5":127
  },
  "Calories":{
   "0":409,
   "1":479,
   "2":340,
   "3":282,
   "4":406,
   "5":300
  }
}
```

Example

```
df = pd.DataFrame(data)
     print(df)

/* Printing the first 10 rows of data set*/
import pandas as pd
      df = pd.read_csv('data.csv')
      print(df.head(10))

/* Print the first 5 rows of data */
import pandas as pd

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

print(df.head())
```

Printing the data frame

Printing the last 5 rows of data frame

print(df.tail())

/* Info about the data set */

In order to print the information of the data we have to give the command

print(df.info())

The result tells us there are n rows and m columns:

And the name of each column, with the data type:

Concept of Null Values

- The info() method also tells us how many Non-Null values there are present in each column, and in our data set it seems like there are 164 of 169 Non-Null values in the "Calories" column.

- Which means that there are 5 rows with no value at all, in the "Calories" column, for whatever reason.

- Empty values, or Null values, can be bad when analyzing data, and you should consider removing rows with empty values. This is a step towards what is called cleaning data,

Bad data – Data Cleaning

Bad data could be

(a) empty cells

(b) data in wrong format

(c) wrong data

(d) duplicates

Return a new data frame with no empty cells

- Return a new data frame with no empty cells

```
import pandas as pd

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

new_df = df.dropna()

print(new_df.to_string())
```

What is the purpose of dropna

- dropna() method returns a new data frame and will not change the original.
- If we want to change the original DataFrame we use the inplace= True assignment,.

Example

- Remove all rows with NULL values:

```
import pandas as pd

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

df.dropna(inplace = True)

print(df.to_string())
```

Now, the dropna(inplace = True) will NOT return a new DataFrame, but it will remove all rows containg NULL values from the original DataFrame.

Replace empty values

- Another way of dealing with empty cells is to insert a *new* value instead.
- This way you do not have to delete entire rows just because of some empty cells.
- The fillna() method allows us to replace empty cells with a value:

Example

- Replace NULL values with the number 130:

```
import pandas as pd

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

df.fillna(130, inplace = True)
```

Replace Only for a specified column

- To only replace empty values for one column, specify the *column name* for the Data Frame:
- Replace NULL values in the "Calories" columns with the number 130:

```
import pandas as pd

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

df["Calories"].fillna(130, inplace = True)
```

Replace Using Mean,Median,Mode

- A common way to replace empty cells, is to calculate the mean, median or mode value of the column.
- Pandas uses
  the mean() median() and mode() methods to calculate the respective values for a specified column:

Replace Using Mean, Median, or Mode

- Example
- Calculate the MEAN, and replace any empty values with it:

```
import pandas as pd

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

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)
```

Replace Using Mean, Median, or Mode

- Calculate the MEDIAN, and replace any empty values with it:

```
import pandas as pd

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

x = df["Calories"].median()

df["Calories"].fillna(x, inplace = True)
```

Replace Using Mean, Median, or Mode

- Calculate the MODE, and replace any empty values with it:

```
import pandas as pd

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

x = df["Calories"].mode()[0]

df["Calories"].fillna(x, inplace = True)
```

Convert Into a Correct Format

- In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:

Convert Into a Correct Format

- Let's try to convert all cells in the 'Date' column into dates.
- Pandas has a to_datetime() method for this:

```
import pandas as pd

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

df['Date'] = pd.to_datetime(df['Date'])

print(df.to_string())
```

What will be the problem

- As you can see from the result, the date in row 26 where fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value. One way to deal with empty values is simply removing the entire row.
- Remove rows with a NULL value in the "Date" column:
- df.dropna(subset=['Date'], inplace = True)

Pandas- Fixing wrong data

- Wrong data can be anything for example the data in place of 1.78 may be 178.
- Some of the data may be out of range then also the problem needs to be resolved.
- For example we can write something like :-
- df.loc[7,’duration’]=45
- For small data sets we can have values checked but what about large data sets.

Large data sets

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.loc[x, "Duration"] = 120
```

Removing Rows

- Another way of handling wrong data is to remove the rows that contain the wrong data
- Delete rows where duration is higher than 120

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.drop(x, inplace = True)
```

Inplace will make permannt change in df.

Discovering and removing duplicates

- Duplicate rows are rows that have been registered more than one time.
- To discover duplicates we can use the duplicated method.
- The duplicated () method returns a boolean value for each row.
- Returns true for every row that is a duplicate , otherwise false.
- print(df.duplicated())

Removing duplicates

- To remove duplicates we use the drop_duplicates() method
- df.drop_duplicates(inplace = True)
- The (inplace = True) will make sure that the method does NOT return a new DataFrame, but it will remove all duplicates from the original DataFrame.

Pandas data

- A great aspect of the Pandas module is the corr() method.
- The corr() method calculates the relationship between each column in your data set.
- df.corr()

Pandas data

- The Result of the corr() method is a table with a lot of numbers that represents how well the relationship is between two columns.
- The number varies from -1 to 1.
- 1 means that there is a 1 to 1 relationship (a perfect correlation), and for this data set, each time a value went up in the first column, the other one went up as well.
- 0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.
- -0.9 would be just as good relationship as 0.9, but if you increase one value, the other will probably go down.
- 0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

Pandas data

- Perfect Correlation:

We can see that "Duration" and "Duration" got the number 1.000000, which makes sense, each column always has a perfect relationship with itself.

- Good Correlation:

"Duration" and "Calories" got a 0.922721 correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

- Bad Correlation:

"Duration" and "Maxpulse" got a 0.009403 correlation, which is a very bad correlation, meaning that we can not predict the max pulse by just looking at the duration of the work out, and vice versa.