PANDAS

Dimensions	Name	Description
1	Series	1D labeled homogeneously-typed array
2	DataFrame	General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column

DEMO

Note: This was made by the Stackoverflow Developer Dataset.
*import pandas as pd

df = read_csv("file path") # reads the CSV file
df #prints the entire data frame.
df.shape #gives the shape of the data frame (rows, columns)
df.info #gives the complete info about the data frame

pd.set_option("display.max_columns", number of columns) # displays all the columns.
pd.set_option("display.max_rows", number of columns) # displays all the columns.

schema_df = pd.read_csv('file path_schema.csv") #gives what the each column mean
df.head() #displays first 5 rows by default
df.head(10) # displays first 10 rows
df.tail() # displays last 5 rows
df.tail(10) #displays last 10 rows

DATAFRAME AND SERIES

- We can create a data frame with the help of dictionaries.

people = {"First":['Corey', 'jane', 'Jhon']
"Last":['Schafer', 'Doe', 'Doe']
"Email": ['Corey@gmail', "Jane@gmail", Jhon@gmail"]}

- Keys are columns, Values are rows.
- There are two data types in pandas: Series and Data Frames
- **Data Frame**: Generally, It is a 2 Dimensional data structure, In layman terms, it is the data with rows and columns.

df = pd.DataFrame(people) #Creates data frame from the dictionary

- We can access the columns in two ways

df.['email'] #Prints the particular column of the data frame, in this case it is the email column.
type(df.['email']) #prints the data type of the email column

- It prints "pandas.core.series.Series" because it is a series data type.

- **Series**: Its 1 Dimensional array but, in layman terms It is a list of data, but has a lot more functionality than that.

df.email # prints the particular column of the data frame, in this case, email column. (other notation)
df(['last','email']) #This accesses the multiple columns of the data frame, In this case, Lastname and email.
df.columns #prints all the column names as a list

- We can use loc, iloc in order to access the rows

df.iloc(0) #prints the 1st row of the data frame.
df.iloc([0,1]) #prints the 1st, 2nd rows of the data frame.

- We can also access columns using iloc, as the first argument as rows and second argument as columns.
- iloc can take only integer input like 0,1 rows or 0,1(indexes) columns. We cannot use column names here.

df.iloc([0,1], 2) #prints the 2nd column of 1st, 2nd rows of the data frame.

- loc can take both column names and index input as well.

df.loc([0,1], 2) #prints the 2nd column of 1st, 2nd rows of the data frame.
df.loc([0,1], 'email') #prints the 'email' column of 1st, 2nd rows of the data frame.
df.loc([0,1], ['email', 'last']) #prints the 'email','last' column of 1st, 2nd rows of the data frame.

- It prints in the order of columns that we give.

df.shape #prints the shape of the data frame
df['Hobbyist'].value_count() # prints the sum of the total number of hobbyists in Yes/No groups
df.iloc(0:3, Hobbyist: Experiment) #prints the Hobbyist to Experiment columns of 0,1,2 rows

- Experiment column name is included while printing but, 3rd row is not included in rows.

INDEXES

- We can set the indexes whatever we want instead of having default indexes 0,1,2,....

df.set_index('email') #this changes email column in the data set as indexes

- The above line does not affect the original data, original data will have indexes of 0,1,2,...

df.set_index('email', inplace=True) #this changes email column in the data set as indexes

- The above line will change the indexes of the original data set into emails.

df.index #shows all the indexes in the form of a list

- Why change the indexes? Well, a unique data index will be a good identifier.

Now
df.loc[‘Corey@gmail’] # prints the info/row that belongs to Corey

df.reset_index(inplace =True) # Replaces the indexes with default indexes 0,1,2,..

- When there is a unique id in data itself, we don't need special indexes given by the data frame.
- There is another method to set desired index columns

df=pd.read_csv(‘file path’, index_col = “column name”) # changes the default indexes to particular column name

schema_df.sort_index() #sort the data with respect to indexes
schema_df.sort_index(ascending= True) #sort the data in ascending order with respect to indexes
schema_df.sort_index(inplace = True) #sort the data in ascending order with respect to indexes

FILTERING

df[‘last’] == ‘Doe’ #prints the true/ false values for the ‘last’ in data frame containing Doe or not

- If ‘last’ is Doe it will print true, if ‘last’ is not Doe it prints false. This is a filter mask.

df(df[‘last’]==”Doe”) # print the rows that contain the last name as Doe
df(df[‘last’]==”Doe”, ‘email’) # print the emails in the rows that contains last name as Doe

df[‘last’]==”Doe” & df[‘first’]==”John” #prints the rows with last name as Doe and first name as John
df[‘last’]==”Doe” | df[‘first’]==”John” #prints the rows with last name as Doe or first name as John

df.loc(df[‘ConvertedComp’]>70000) #prints salaries greater than 70,000
df.loc(df[‘ConvertedComp’]>70000, [‘Country’, ‘languages’.]) # prints country and languages of the people with salaries greater than 70,000

Countries = [‘united States’, ‘India’,’ United Kingdom’, ‘Germany’, ‘Canada’]
Filt = df[‘country’].isin(Countries)
df.loc(Filt, ‘County’)
#prints countries in the data frame that matches with the countries in the list

df[‘languageworkedwith’]
Filt = df[‘languageworkedwith’].str.contains(‘Python’, na=False) #string method
df.loc(Filt,’languageworkedwith’)
# prints the languageworkedwith column, wherever there is python

Filt #prints the True/False values

UPDATING VALUES

- Let's look at how to update column names.

df= pd.DataFrame(people) #creates data frame of data
df.columns #shows column names

- If we want to update the column name

df.columns = ['first_name','last_name', 'email'] #changes column names

- If we have to change column names into uppercase or lowercase

df.columns= [x.upper for c in df.columns] #changes all column names to uppercase
df.columns= [x.lower for c in df.columns] #changes column names to lowercase

- If want to replace spaces with underscore "_"
- If we use dot "." notation to call the columns names, it doesn't work if column name have spaces

df.columns = df.columns.str(replace(' ', "_") #replaces all spaces with underscores

- If we want to rename a particular columns

df.rename(columns={'first_name':first, 'last_name':last}) #changes column names temporarily
df.rename(columns={'first_name':first, 'last_name':last, inplace = True) #changes column names in data frame as well

- Let's look at how to update values

- Let's say you want to change last name of Jane Doe

df.loc(2) # prints the second row.
df.loc[2]=['Jhon', 'Smith', 'jhonsmith@gmail'] #changes last name and email in second row
Or
df.loc(2, ['last', 'email']) = ['smith', 'jhonsmith@gmailo] #changes last name and email in second row

- Pandas had indexer '.at', it is used to change single value
- We can also use .loc instead , for single values
- Let's say you have a large data and you want to find "john doe" and change the last name, then we apply a filter and we will try to change from there

filt= (df.['email']== 'jhondoe@gmail'] #filters the data that matches with this email, since email will be unique we use that.
df.loc[filt, 'last'] = 'smith' #changes the last name to smith.

- How do you change multiple row values? let’s say you want to change all the email addresses to lowercase, Let's go through all 4 methods.
- 1)apply 2)map 3)applymap 4)replace

df.[‘email’].str.lower() #changes emails to lowercase temporarily
df[‘email’] = df.[‘email’].str.lower() #changes all emails into lowercase in data frame as well

- **1) apply**
  apply is used to call a function on our values. Apply can work on a data frame or a series object. Let's say we want to see the length of all email addresses.

df[‘email’].appyl(len) #calculates length of email addresses individually.

- let's say we want to change all the email addresses into upper case.

```
def upper_email(email):
    return email.upper()
```

df[‘email’].apply(upper_email) # changes all the emails into upper case temporarily not in data frame
df[‘email’] = df[‘email’].apply(upper_email) #changes email addresses permanently in the data frame.

- Let's say you want to apply on data frame, lets see what if we apply “len” function for entire dat frame

df.apply(len) #it applies len function to each and every series of data frame specifically for columns # it shows number of rows in each column

- If we want to see number of columns in each row

df.apply(len, axis =”columns”) #shows number of columns in each row

- apply can only be useful for performing operations on series objects/numbers. We cannot use it for performing operations on every single value when we are considering the entire data frame.

- **2) apply map**
  “apply map” can perform operations on every single value even if we consider the entire data frame.
  It works only on data frames, it doesn’t work on Series objects.

df.applymap(len) #prints the length of each and every value in the data frame.

- Let's say we want to change everything in the data frame into lower case then we can use applymap

df.applymap(str.lower) #changes every value in the data frame into lowercase

- **3) map**
  map method works only on series.It is used for substituting each value in series with another value
  Let’s say we want to change first names of persons in the data frame.

df[‘first’].map(“Corey”: “Chris”, “John”: “Jonny”) # changes Corey to Chris and john to jonny and maps remaining values to Nan

- **4) replace**
  If we want to replace the names of persons, without changing other names to Nan.

df[‘first’].replace (“Corey”: “Chris”, “John”: “Jonny”) # changes Corey to Chris and John to Jonny and remaining names will remain same temporarily
df[‘first’] = df[‘first’].replace (“Corey”: “Chris”, “John”: “Jonny”) #changes permanently

ADD/REMOVE COLUMNS/ROWS

- Let’s see how to update the columns
- Let’s say we want to combine first and last name and create a new column.

df[‘first’]+’ ’+ df[‘last’] # adds both the first and last name
df[‘full_name’] = df[‘first’]+’ ’+ df[‘last’] # creates new column with full_name as column name

- We cannot use dot”.” notation while creating column, because if we use dot notation python thinks you are assigning an attributes on to the data frame not the column
- Now lets look at remove columns, Since we had full name we no need of first and last name columns

df.drop(columns=[‘first’, ‘last’]) #deletes/removes the columns temporarily
df.drop(columns=[‘first’, ‘last’], inplace =True) #removes the columns in dataframe as well

- If we want to split the full name into first and last names.

df[‘full_name’].str.split(‘ ‘) #splits with space as delimiter, 1st word is first_name
df([‘first’, ‘last’])=df[‘full_name’].str.split (‘ ’, expand=True) #creates column names with first, last names

- Now, Let’s look at adding and removing rows.

df.append({‘first’: ‘Tony’}, ignoree_index=True) #appends a row with name Tony in first column and remaining column values will be NaN

- Lets see how to add two data frames

df.append(df2, ignore_index=True) #combines the two data frames with different numbers of rows and columns, the final data frame will be sorted out by default.
df.append(df2, ignore_index=True, sort=False) # combines two data frames without sorting

df= df.append(df2, ignore_index=True, sort=False) #changes permanently

- Let's look at removing rows

df.drop(index=4) #drops the 5th row (index =4)

- If we want to delete the multiple rows with same last name

df.drop(index=df[df[‘last’]==’Doe’].index) # removes all the rows with last name Doe

Or
filt=df[‘last’]==’Doe’
df.drop[index=df[filt].index) # removes all the rows with last name index

SORTING DATA

- Let’s sort the data by last name or descending order

df.sort_values(by= ‘last’) # sorts the data with respect to last names
df.sort_values(by= ‘last’, ascending=False) #sorts with respect to last name in Descending order

- If we want to sort by both first and last names

df.sort_values(by= [‘last’, ‘first’] , ascending=False) # sorts data in descending order with respect to both first and last names temporarily

- If we want to sort last name by descending order and first name is in ascending order.

df.sort_values(by= [‘last’, ‘first’] , ascending=[False, True] , inplace =True) # sorts the data in both ascending and descending order in data frame as well

- If you want to sort the as it was in the beginning

df.sort_index() # sorts data as it was while combined
df.[‘last’].sort_values() #sorts last name values.

- Let’s say you want to see the top 10 largest salaries.

df[‘ConvertedCamp’].nlargest(10) # prints the top 10 highest salaries in the data
df.nlargest(10,’ConvertedCamp’) #shows the entire data frame with top 10 highest salaries
df.nsmallest(10,’ConvertedCamp’) #shows the entire data frame with top 10 smallest salaries

AGGREGATING AND GROUPING

- Aggregation basically means, combining a piece of data into a single result like mean, median etc.
- What might be the salary for developers who answered this survey?
- For this we can calculate the median.

df[‘ConvertedComp’].head(5)
df[‘ConvertedComp’].median() #calculate the median of the particular row.

df.median() # finds the columns with numbers and find the median for all the columns
df.describe #shows all the aggregate values at one place

- Mean is not a good metric to represent the data because it can be heavily affected by outliers. So Median gives somewhat reasonable outcomes.

Count = counts the non-missing rows.

- If you are working on a social media platform, you need to see which social media account was used most

df[‘SocialMedia”] # prints the social media column
df[‘SocialMedia”].value_counts() #shows the number of users of each social media
df[‘SocialMedia”].value_counts(normalize=True) #shows output in percentages

- “Group by” is a combination of splitting the object, applying the function, combining the results

df[‘Country’].value_counts #which place prints majority of the survey took place
country_group = df.groupby[‘Country’]. #groups the data by country names
country _group.get_group[‘UnitedStates’] #shows the data that belong to United States

- Most popular social media sites in United States

filt=df[‘Country’]==’UnitedStates’
df.loc(filt)[‘SocialMedia’].value_counts()#prints the most popular social media in united states with number of users
country_group[‘SocialMedia’].value_counts() #prints countrywise popular Social Media
country_group[‘SocialMedia’].value_counts(),loc[‘United States’] #prints the most popular social media in united states with number of users

- If the filter and this was doing the same , why do we need “groupby”? It removes the need for a filter. Instead of applying filter over and over we can use “groupby”

country_group[‘ConvertedComp’].median() #prints the median salaries with respect to all the countries
country_group[‘ConvertedComp’].median(.loc[‘Germany’]) #prints the median salary of the data
country_group[‘Converted Comp’].agg([‘median’,’mean’]) #prints the mean and median of the salaries with respect to countries.
country_group[‘Converted Comp’].agg([‘median’,’mean’]).loc[‘Canda’] #prints mean and median salaries of the Canada state

- How many people in the country know to use Python?

filt = df[‘Country’] ==’India’
df.loc[filt][‘LanguageWorkedWith’] #prints the languageworkedwith column with India
df.loc[filt][‘LanguageWorkedWith’].str.contains(‘Python’) #prints True or False values whether people know python or not.

- Sum function can also work on boolean functions, It considers 1 for True and 0 for False.

df.loc[filt][‘LanguageWorkedWith’].str.contains(‘Python’).sum() #gives the total number of people know the python as language

pd.concat() #concatenates two series

CLEANING DATA
Casting Data types and handling missing values

df.dropna() #drops the rows with missing values
df.dropna(axis= ‘index’, how= ‘any’) #default arguments of dropna
df.replace(‘NA’, np.nan,inplace =True) # replaces all the NA values to NaN values so that it can be easy to deal with missing values.

- The above drops the rows that have null values, if we replace ‘index’ with columns it will drop the columns with null values.
- If we change”any’ to ‘all’ then it will drop the rows/columns that are filled with null values.
- If we want to drop only a particular null rows then

df.dropna(axis= ‘index’, how= ‘any’, subset=[‘email’]) #drops the rows that has null values in email column
df.dropna(axis= ‘index’, how= ‘any’, subset=[‘last’ , ‘email’]) #drops the rows that has null values in email column and in last name column
df.isna() # prints True or False, if the data has Na values it prints True at that position and False if not.
df.fillna(0) # Replaces all the NA values with zero.(useful when working with Numerical data)
df[‘age’] =df[‘age’].astype(int) #Converts the age columns into numbers.

- By default all the columns in data frame are Objects/Strings
- astype() doesn’t work on columns with Null values, you have to convert null values to some number or you have to convert the column into ‘float’ data type.

df[‘age’] =df[‘age’].astype(float) #converts age column into float