Learning_Pandas_Part_4_GroupBy

June 20, 2021

0.0.1 Prepared by Abhishek Kumar

0.0.2 https://www.linkedin.com/in/abhishekkumar-0311/

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
# To get multiple outputs in the same cell

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

%matplotlib inline
```

[3]:
```
# Setup : DataFrame creation

salary = [['1','Abhishek Kumar','AIML', 'Machine Learning Engineer','M', 'Y',␣
 ↪'04051990', 1121000],
          ['2','Arjun Kumar','DM', 'Tech Lead','M', 'Y', '09031992', 109000],
          ['3','Vivek Raj','DM', 'Devops Engineer','M', 'N', np.NaN , 827000],
          ['4','Mika Singh','DM', 'Data Analyst','F', 'Y', '15101991',  np.NaN],
          ['5','Anusha Yenduri','AIML', 'Data Scientist','F', 'Y', '01011989', ␣
 ↪921000],
          ['6','Ritesh Srivastava','AIML', 'Data Engineer','M', 'Y', np.NaN,␣
 ↪785000]]

columns_name=['Emp_Id','Emp_Name','Department','Role','Gender', 'WFH Status',␣
 ↪'DOB', 'Salary']

emp_df = pd.DataFrame(salary,columns=columns_name)
emp_df
```

[3]:

	Emp_Id	Emp_Name	Department	Role	Gender	\
0	1	Abhishek Kumar	AIML	Machine Learning Engineer	M	
1	2	Arjun Kumar	DM	Tech Lead	M	
2	3	Vivek Raj	DM	Devops Engineer	M	
3	4	Mika Singh	DM	Data Analyst	F	

```
4      5     Anusha Yenduri       AIML             Data Scientist      F
5      6  Ritesh Srivastava       AIML              Data Engineer      M

  WFH Status       DOB     Salary
0          Y  04051990  1121000.0
1          Y  09031992   109000.0
2          N       NaN   827000.0
3          Y  15101991        NaN
4          Y  01011989   921000.0
5          Y       NaN   785000.0
```

1 1. Group By: Split-Apply-Combine

i. df.groupby()
ii. .apply() , .agg(), .filter()
iii.

[4]:
```
emp_df_1 = emp_df.copy()
emp_df_1
```

[4]:
```
  Emp_Id           Emp_Name Department                        Role Gender  \
0      1     Abhishek Kumar       AIML  Machine Learning Engineer      M
1      2        Arjun Kumar         DM                   Tech Lead      M
2      3          Vivek Raj         DM            Devops Engineer      M
3      4         Mika Singh         DM               Data Analyst      F
4      5     Anusha Yenduri       AIML             Data Scientist      F
5      6  Ritesh Srivastava       AIML              Data Engineer      M

  WFH Status       DOB     Salary
0          Y  04051990  1121000.0
1          Y  09031992   109000.0
2          N       NaN   827000.0
3          Y  15101991        NaN
4          Y  01011989   921000.0
5          Y       NaN   785000.0
```

[5]:
```
grouped_1 = emp_df_1.groupby('Department')
grouped_1
```

[5]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000192D03CD820>
```

1.1 1.1 Meta Methods

Meta methods are less concerned with the original object on which .groupby() is called.
Mainly provide high-level information such as the number of groups and indices of those groups

[6]:
```
grouped_1.groups
```

[6]:
```
{'AIML': [0, 4, 5], 'DM': [1, 2, 3]}
```

[7]:
```
grouped_1.get_group('DM')
```

[7]:
```
   Emp_Id     Emp_Name Department             Role Gender WFH Status       DOB  \
1       2  Arjun Kumar         DM        Tech Lead      M          Y  09031992
2       3    Vivek Raj         DM  Devops Engineer      M          N       NaN
3       4   Mika Singh         DM     Data Analyst      F          Y  15101991

     Salary
1  109000.0
2  827000.0
3       NaN
```

[8]:
```
grouped_1.indices
```

[8]:
```
{'AIML': array([0, 4, 5], dtype=int64), 'DM': array([1, 2, 3], dtype=int64)}
```

[9]:
```
grouped_1.ndim
```

[9]:
```
2
```

[10]:
```
grouped_1.ngroups
```

[10]:
```
2
```

[11]:
```
# Assign this to a new variable. This will assign a number to each group
grouped_1.ngroup()
```

[11]:
```
0    0
1    1
2    1
3    1
4    0
5    0
dtype: int64
```

[12]:
```
grouped_1.dtypes
```

[12]:
```
            Emp_Id Emp_Name    Role  Gender WFH Status     DOB   Salary
Department
AIML        object   object  object  object     object  object  float64
DM          object   object  object  object     object  object  float64
```

[13]:
```
#for i in range(2):
#    grouped_1.__iter__()
```

[14]:
```
grouped_1.size()
```

[14]:
```
Department
AIML    3
DM      3
dtype: int64
```

[15]:
```
len(grouped_1)
```

[15]:
```
2
```

1.2 1.2 Filter Methods

Filter methods return a subset of the original DataFrame.
Most common is .filter() to drop entire groups based on some comparative statistic about that g
There are a number of methods that exclude particular rows from each group.

- **https://stackoverflow.com/questions/55583246/what-is-different-between-groupby-first-groupby-nth-groupby-head-when-as-index**

[16]:
```
grouped_2 = emp_df_1.groupby('Department')
grouped_2
```

[16]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000192D0442B80>
```

1.3 first/last

This will return the **first/last non-null value** within each group. Oddly enough it will not skip None, though this can be made possible with the kwarg dropna=True. As a result, **you may return values for columns that were part of different rows originally:**

[17]:
```
grouped_2.first()
```

[17]:
```
            Emp_Id        Emp_Name                        Role Gender  \
Department
AIML             1  Abhishek Kumar  Machine Learning Engineer      M
DM               2     Arjun Kumar                  Tech Lead      M

            WFH Status       DOB     Salary
Department
AIML                Y  04051990  1121000.0
DM                  Y  09031992   109000.0
```

[18]:
```
grouped_2.last()
```

[18]:
```
            Emp_Id           Emp_Name           Role Gender WFH Status  \
Department
AIML             6  Ritesh Srivastava  Data Engineer      M          Y
```

```
DM              4         Mika Singh   Data Analyst      F          Y

                 DOB    Salary
Department
AIML        01011989  785000.0
DM          15101991  827000.0
```

1.4 head(n)/tail(n)

Returns the **top/bottom n rows** within a group. **Values remain bound within rows**. If you give it an n that is more than the number of rows, it returns all rows in that group without complaining:

[19]:
```
grouped_2.head(2)
```

[19]:
```
   Emp_Id        Emp_Name Department                        Role Gender  \
0       1  Abhishek Kumar       AIML  Machine Learning Engineer      M
1       2     Arjun Kumar         DM                   Tech Lead      M
2       3       Vivek Raj         DM            Devops Engineer      M
4       5  Anusha Yenduri       AIML             Data Scientist      F

  WFH Status       DOB     Salary
0          Y  04051990  1121000.0
1          Y  09031992   109000.0
2          N       NaN   827000.0
4          Y  01011989   921000.0
```

[20]:
```
grouped_2.tail(1)
```

[20]:
```
   Emp_Id           Emp_Name Department           Role Gender WFH Status  \
3       4         Mika Singh         DM   Data Analyst      F          Y
5       6  Ritesh Srivastava       AIML  Data Engineer      M          Y

        DOB    Salary
3  15101991       NaN
5       NaN  785000.0
```

1.5 nth

- GroupBy.nth(n, dropna=None)[source]
  - Take the nth row from each group if n is an int, or a subset of rows if n is a list of ints.
  - If dropna, will take the nth non-null row, dropna is either 'all' or 'any'; this is equivalent to calling dropna(how=dropna) before the groupby.

This takes the nth row, so again **values remain bound within the row**. `.nth(0)` is the same as .head(1), though they have different uses. For instance, if you need the 0th and 2nd row, that's difficult to do with `.head()`, but easy with `.nth([0,2])`. Also it's fair easier to write `.head(10)` than `.nth(list(range(10)))`.

[21]:
```
# Take the nth row from each group if n is an int, or a subset of rows if n is␣
↪a list of ints.

grouped_2.nth(2)

grouped_2.nth([0,2])
```

[21]:
```
            Emp_Id           Emp_Name           Role Gender WFH Status  \
Department
AIML             6  Ritesh Srivastava  Data Engineer      M          Y
DM               4         Mika Singh   Data Analyst      F          Y

                  DOB    Salary
Department
AIML              NaN  785000.0
DM           15101991       NaN
```

[21]:
```
            Emp_Id           Emp_Name                        Role Gender  \
Department
AIML             1     Abhishek Kumar  Machine Learning Engineer      M
AIML             6  Ritesh Srivastava              Data Engineer      M
DM               2        Arjun Kumar                  Tech Lead      M
DM               4         Mika Singh               Data Analyst      F

            WFH Status       DOB     Salary
Department
AIML                 Y  04051990  1121000.0
AIML                 Y       NaN   785000.0
DM                   Y  09031992   109000.0
DM                   Y  15101991        NaN
```

- **nth also supports dropping rows with any null-values, so you can use it to return the first row without any null-values, unlike `.head()`**

[23]:
```
# grouped_2.nth([0,2], dropna='any')
```

[ ]:
```
# ![image.png](attachment:image.png)
```

[24]:
```
# we are selecting the 0th and 2nd rows, not rows whose indices equal 0 and 2.

grouped_2.take([0,2])
```

[24]:
```
              Emp_Id           Emp_Name                        Role Gender  \
Department
AIML       0       1     Abhishek Kumar  Machine Learning Engineer      M
           5       6  Ritesh Srivastava              Data Engineer      M
DM         1       2        Arjun Kumar                  Tech Lead      M
```

```
            3      4          Mika Singh                Data Analyst      F

              WFH Status       DOB      Salary
Department
AIML          0          Y  04051990  1121000.0
              5          Y       NaN   785000.0
DM            1          Y  09031992   109000.0
              3          Y  15101991        NaN
```

1.5.1 Selecting group based on the condition that applies on the whole group

[ ]:
```
grouped_1 = emp_df_1.groupby('Department', as_index=False)
grouped_1

# The argument of filter must be a function that, applied to the group as a␣
↪whole, returns True or False.

grouped_1.filter(lambda x: max(x['Salary']) >= 1121000.0)
```

[ ]:
```
# The argument of filter must be a function that, applied to the group as a␣
↪whole, returns True or False.

grouped_1.filter(lambda x: min(x['Emp_Name'].str.len()) >= 10)
```

[ ]:
```
# The argument of filter must be a function that, applied to the group as a␣
↪whole, returns True or False.

grouped_2.filter(lambda x: sum(x['Salary']) >= 950000)
```

[ ]:
```

```

1.6 1.3 Aggregation Methods

- .agg()

  Aggregation methods (also called reduction methods) "smush" many data points into an aggregated statistic about those data points. An example is to take the sum, mean, or median of 10 numbers, where the result is just a single number.

[ ]:
```
grouped_3 = emp_df_1.groupby('Department')
grouped_3
```

[ ]:
```
# grouped_3.agg(np.sum)

grouped_3.agg('sum')
```

[ ]:
```
grouped_3.agg('mean')
```

1.6.1 + Applying multiple functions at once

[ ]:
```
x= grouped_3.agg(['max','mean', 'min'])
x
```

1.6.2 - End

1.6.3 + Analysing the aggregated result dataframe

[ ]:
```
x.ndim
```

[ ]:
```
x.size
```

[ ]:
```
x.shape
```

[ ]:
```
len(x)
```

[ ]:
```
x.iloc[:,2:]
```

[ ]:
```
x.columns
x.columns[0]
```

[ ]:
```
x.index
x.index[0]
```

1.6.4 - End

[ ]:
```
# as_index = False does not create the groupby columns as Indexes

grouped_3a = emp_df_1.groupby(['Department','Gender'], as_index = False)
grouped_3a
```

[ ]:
```
grouped_3a.agg('sum')
grouped_3a['Salary'].agg(['sum'])
```

[ ]:
```
# We can also use the reset_index DataFrame function to achieve the same result␣
↪as the column names are stored in the resulting MultiIndex

emp_df_1.groupby(['Department','Gender']).sum().reset_index()
```

[ ]:
```
grouped_3a.size()
grouped_3a.size().reset_index()
```

[ ]:
```
grouped_3a.describe()
```

[ ]:
```
grouped_3a.aggregate('count')
grouped_3a.count()
```

```
grouped_3a.agg(lambda x: x.count())
```

[ ]:
```
grouped_3a['Salary'].aggregate('count')
grouped_3a['Salary'].count()
grouped_3a['Salary'].agg(lambda x: x.count())
```

Note: The aggregating functions above will exclude NA values.

1.6.5 Renaming column labels

i. .rename()
ii. Named Aggregation

i. .rename()

[ ]:
```
grouped_3b = emp_df_1.groupby(['Department','Gender'])
grouped_3b
```

[ ]:
```
grouped_3b.agg(['min','max','mean'])
grouped_3b.agg(['min','max','mean']).rename(columns = { 'min' : 'Least', 'max':␣
↪'Most', 'mean':'Avg'})
```

ii. NamedAggregation

To support column-specific aggregation with control over the output column names, pandas accept

i. The keywords are the output column names

ii. The values are tuples whose first element is the column to select and the second element is

iii. Pandas provides the pandas.NamedAgg namedtuple with the fields ['column', 'aggfunc'] to ma

[ ]:
```
# Named Tuple

grouped_3b.agg( Max_Sal = pd.NamedAgg( column = 'Salary' , aggfunc = 'max'),
                Min_Sal = pd.NamedAgg( column = 'Salary' , aggfunc = 'min'),
                Avg_Sal = pd.NamedAgg( column = 'Salary' , aggfunc = 'mean'))


# Plain Tuple
# Also, the index is reset here.

grouped_3b.agg( Max_Sal = pd.NamedAgg( 'Salary' ,  'max'),
                Min_Id = pd.NamedAgg( 'Emp_Id' ,  'min'),
                Avg_Sal = pd.NamedAgg( 'Salary' ,  'mean')).reset_index()
```

1.6.6 Applying different functions to DataFrame columns

By passing a dict to aggregate we can apply a different aggregation to the columns of a DataFra

[ ]:
```
grouped_3b.agg({ 'Salary' : lambda x: np.std(x, ddof=1)})

# index on Groupby columns is also reset.
grouped_3b.agg({ 'Salary' : 'mean', 'Role' : 'sum'}).reset_index()
```

1.7 1.4 Transformation

- .transform()

  Transformation methods return a DataFrame with the same shape and indices as the original, but with different values. With both aggregation & filter methods, the resulting DataFrame will commonly be smaller in size than the input DF. This is not true of a transformation, which transforms individual values themselves but retains d shape of the original DataFrame.

[ ]:
```
grouped_3c = emp_df_1.groupby(['Department'])
grouped_3c.count()
```

[ ]:
```
# Here i have not created a new column
# But a new column can be created
emp_df_1
transformed = grouped_3c.transform(lambda x : x.fillna(x.mean()))
transformed
```

[ ]:
```
# Using transform to get boolean values and then passing this boolean value to␣
 ↪the dataframe to get the correct record
# NOT WORKING AS EXPECTED

emp_df_1['MaxSalary'] = grouped_1['Salary'].transform('max')
emp_df_1

emp_df_1['SumSalary'] = grouped_1['Salary'].transform('sum')
emp_df_1

emp_df_1['PctSalary'] = emp_df_1['Salary']/emp_df_1['SumSalary'] * 100
emp_df_1

# emp_df_1['PctSalary_2'] = grouped_1['Salary'].transform(lambda x : x.sum)
# emp_df_1
```

[ ]:
```
grouped_trans = transformed.groupby(level=0)
grouped_trans.count()
```

1.7.1 + Window and resample operations

i. rolling()

ii. expanding()
iii. resample()

[ ]:
```
df_re = pd.DataFrame({'A': [1] * 10 + [5] * 10,
                      'B': np.arange(20)})
df_re.head()
df_re.tail()
```

[ ]:
```
# This will apply the rolling() method on the samples of the column B based on␣
↪the groups of column A.

df_re.groupby('A').rolling(4).B.sum()
```

[ ]:
```
# The expanding() method will accumulate a given operation (sum() in the␣
↪example) for all the members of each particular group.

df_re.groupby('A').expanding().B.sum()
```

[ ]:
```
# ReSampling is not yet covered...
```

1.8 Iteration 2

[ ]:
```
df1 = pd.DataFrame({'id': [1,2],
                    'name': ['a','b'],
                    'prem1' : [100,280],
                    'prem2' : [np.NaN,180],
                    'prem3' : [300,np.NaN],
                    'disc1' : [20,40],
                    'disc2' : [np.NaN,30],
                    'disc3' : [50,np.NaN],})
df1
```

[ ]:
```
df1_melted = pd.wide_to_long(df1, i=['id','name'], j='month',␣
↪stubnames=['prem','disc'])
df_long = df1_melted.reset_index()
```

[ ]:
```
df_long
```

[ ]:
```
# Returns min value for each columns within each group

df_long.groupby('id').min()
```

[ ]:
```
# Returns max value for each columns within each group

df_long.groupby('id').max()
```

1.8.1 FIRST and LAST returns the non-null value

[ ]:
```
df_long.groupby('id').first()
```

[ ]:
```
df_long.groupby('id').last()
```

1.8.2 HEAD() and TAIL() - returns the actual head(n) and tail(n) records

[ ]:
```
df_long.groupby('id').head(2)
```

[ ]:
```
df_long.groupby('id').tail(1)
```

[ ]:
```
df_long2 = df_long.sort_values(['id','prem'])
```

[ ]:
```
df_long2.groupby('id').head(2)
```

[ ]:
```
df_long2.groupby('id').tail(1)
```

1.8.3 Another way to get the first and last row is to find the INDEX of MIN or MAX value of a columns and use that index to filter out records

- idxmin() and idxmax()

[ ]:
```
### Here, idxmax() finds the indices of the rows with max value within groups,
### and .loc() filters the rows using those indices :

df_long2.loc[df_long2.groupby(["id"])["prem"].idxmax()]
df_long2.loc[df_long2.groupby(["id"])["prem"].idxmin()]
```

1.9 TRANSFORM

https://pbpython.com/pandas_transform.html

1.9.1 Creating a FLAG , indicating the MAX or MIN value

[ ]:
```
df_long['flag'] = df_long.groupby('id')['prem'].transform(lambda x : x == x.
↪max())
df_long
```

1.9.2 Using transform to perform filtering of rows

- Transform will help to create a new column or a flag
- Based on the new flag, we will filter out rows

1.9.3 Examples

- 1. Simple Scenario :

  - Selecting rows with the highest / max / lowest / min values : This can be achieved using sorting by sort_values() and head() and tail()

- 2. Not straighforward Scenario :
  - But incase of scenarios, wherein, the selection criteria is not straightforward like MIN/MAX, instead like MEAN or PCT.
    * Then we need to first find the mean or pct within each group and find the rows which satisfy those condition.

[ ]:
```
# Simple scenario
# This is handled using SORT_VALUES() and HEAD()

df_long.sort_values(['id','prem'], ascending=[True, False], inplace = True)
df_long.groupby('id').head(1)
```

[ ]:
```
df_long[df_long.groupby('id')['prem'].transform(lambda x : x == x.max())]
```

[ ]:
```
# Complex scenario

df_long[df_long.groupby('id')['prem'].transform(lambda x : x <= x.mean())]
```

[ ]:
```

```

1.9.4 Alternate way :

1.9.5 Transform creates a new variable , without changing the shape of the dataframe.

- It does not filter any records. (But can be used to filter record, by passing the BOOLEAN Value created within transform() to the original dataframe.)
  - See the above example
- In case of any requirement of creating a FLAG , indicating the MAX or MIN value , the new column can be checked for equality using ==

[ ]:
```
df_long['flag'] = df_long.groupby('id')['prem'].transform('max')
df_long
```

[ ]:
```
df_long['flag'] = df_long['prem'] == df_long.groupby('id')['prem'].
↪transform('max')
df_long
```

https://www.analyticsvidhya.com/blog/2020/03/understanding-transform-function-python/

[ ]:
```

```

1.10 Creating running totals with cumsum( )

[ ]:
```
d = {"salesperson":["Nico", "Carlos", "Juan", "Nico", "Nico", "Juan", "Maria",␣
 ↪"Carlos"], "item":[10, 120, 130, 200, 300, 550, 12.3, 200]}
df = pd.DataFrame(d)
df

df["running_total"] = df["item"].cumsum()
df["running_total_by_person"] = df.groupby("salesperson")["item"].cumsum()
df
```

1.11 Calculate running count with groups using cumcount() + 1

[ ]:
```
d = {"salesperson":["Nico", "Carlos", "Juan", "Nico", "Nico", "Juan", "Maria",␣
 ↪"Carlos"], "item":["Car", "Truck", "Car", "Truck", "cAr", "Car", "Truck",␣
 ↪"Moto"]}
df = pd.DataFrame(d)
df

# Fixing columns
df["salesperson"] = df["salesperson"].str.title()
df["item"] = df["item"].str.title()

df["count_by_person"] = df.groupby("salesperson").cumcount() + 1
df["count_by_item"] = df.groupby("item").cumcount() + 1
df["count_by_both"] = df.groupby(["salesperson","item"]).cumcount() + 1
df
```

[ ]:
```
# Creating a new dataframe
emp_df3 = emp_df.copy()
```

[ ]:
```
emp_df3.groupby('Department').first()
emp_df3.groupby('Department').head(1)
```

[ ]:
```
emp_df3.groupby('Department').last()
emp_df3.groupby('Department').tail(1)
```

[ ]:
```
emp_df3.sort_values(['Department','Emp_Name'], ascending=True).
 ↪groupby('Department').last()
emp_df3.sort_values(['Department','Emp_Name'], ascending=False).
 ↪groupby('Department').tail(1)
```

[ ]:
```
emp_df3.sort_values(['Department','Salary'], ascending=False).
 ↪groupby('Department').last()
emp_df3.sort_values(['Department','Salary'], ascending=False).
 ↪groupby('Department').tail(1)
```

1.12 To generate ranking within each group

- method = 'first' / 'dense' / 'min' / 'max' / 'average'
- ascending = True/False
- pct = True

1.12.1 Example 1

[ ]:
```
emp_df3.dtypes
emp_df3['Salary'] = emp_df3['Salary'].astype('float')
```

[ ]:
```
# Rank() does not work when rank is done on NON-Numeric column
emp_df3['default_rank2'] = emp_df3.groupby('Department')[['Salary']].
 ↪rank(ascending=False)
emp_df3
```

[ ]:
```
emp_df3['default_rank'] = emp_df3['Salary'].rank()
emp_df3
```

1.12.2 Example 2

[ ]:
```
data = {'close_date': ["2012-08-01", "2012-08-01", "2012-08-01", "2012-08-02",␣
 ↪"2012-08-03", "2012-08-04", "2012-08-05", "2012-08-07"],
        'seller_name': ["Lara", "Julia", "Julia", "Emily", "Julia", "Lara",␣
 ↪"Julia", "Julia"]
       }
df = pd.DataFrame(data)
```

[ ]:
```
df['close_date'] = pd.to_datetime(df['close_date'])
```

[ ]:
```
df['rank_seller_by_close_date'] = df.groupby('seller_name')['close_date'].
 ↪rank(method='first')
```

[ ]:
```

```

1.13 Other functions

[ ]:
```
emp_df3['default_rank3'] = emp_df3.groupby('Department')['default_rank'].bfill()
emp_df3
```

[ ]:
```
emp_df3.sort_values(['Department','Salary'], ascending=True).
 ↪groupby('Department')['Salary'].nth(0).to_frame().reset_index()
# emp_df3
```

[ ]:
```
emp_df3.groupby('Department')['Role'].unique()
```

[ ]:
```
emp_df3.groupby('Department')['Role'].nunique()
```

```
[ ]: ods = emp_df3.groupby('Department', as_index = False)
     ods['Role'].count()
```

```
[ ]: emp_df3.groupby('Department', as_index = False)['Role'].size()
```

```
[ ]: emp_df3.groupby('Department')['Role'].describe()
```

```
[ ]: emp_df3.groupby('Department')['Gender'].value_counts()
```

```
[ ]: emp_df3.groupby('Department')['Salary'].nlargest()
```

```
[ ]: emp_df3.groupby('Department')['Salary'].nsmallest()
```

```
[ ]: emp_df3.groupby('Department')['Salary'].sum()
```

```
[ ]: # as_index helps to create a dataframe
     emp_df3.groupby('Department', as_index=False)['Salary'].min()
```

```
[ ]: emp_df3.groupby('Department')['Salary'].max()
```

```
[ ]: emp_df3.groupby('Department')['Salary'].mean()
```

```
[ ]: emp_df3
```

1.14 Cumulative sum within each group using CUMSUM( )

```
[ ]: emp_df3['Salary'].fillna(0, inplace=True)
     emp_df3
```

```
[ ]: emp_df3['cum_sal'] = emp_df3.groupby('Department')['Salary'].cumsum()
     emp_df3
```

1.15 To generate a sequential rownumber using CUMCOUNT() + 1

```
[ ]: emp_df3['Count'] = emp_df3.sort_values(['Department','Emp_Name'],␣
      ↪ascending=True).groupby('Department')['Emp_Name'].cumcount()+1
     emp_df3.sort_values(['Department','Emp_Name'], ascending=True, inplace=True)
     emp_df3
```

Alternate way, not effective

```
[ ]: tmp = emp_df3.groupby('Department')['Emp_Name'].cumcount().reset_index()
     tmp
     tmp.rename(columns={tmp.columns[-1]:'new'},inplace=True)
     tmp
```

[ ]:
```
emp_df3 = pd.merge(emp_df3,tmp, left_index=True, right_index=True).
↪drop('index', axis=1)
emp_df3
```

1.16 LAG (+n) / LEAD (-n) functionality

1.17 To retrieve previous (+n) /ahead (-n) values using SHIFT(n / -n)

- shift(n) : LAG
- shift(-n) : LEAD

[ ]:
```
emp_df3.sort_values(['Department','Salary'],inplace=True)
emp_df3['PrevSal'] = emp_df3.groupby('Department')['Salary'].shift(1)
emp_df3
```

1.18 Retain the last filled value to fill the NaN cells

1.19 Using FILLNA(method = 'bfill' / 'ffill')

1.19.1 bfill - backward fill : Go Backward and fill the empty cell

1.19.2 ffill - forward fill : Go Forward and fill the empty cell

[ ]:
```
emp_df3
```

[ ]:
```
emp_df3.loc[emp_df3.Emp_Id.isin( ['4','6']), 'PrevSal'] = np.NaN
emp_df3.sort_values(['Department','Emp_Name'], inplace = True)
emp_df3
```

[ ]:
```
emp_df3['ForwardFilledPrevSal'] = emp_df3.groupby('Department')['PrevSal'].
↪fillna(method = 'ffill')
emp_df3
```

[ ]:
```
emp_df3['BackwardFilledPrevSal'] = emp_df3.groupby('Department')['PrevSal'].
↪fillna(method = 'bfill')
emp_df3
```

1.20 filling-missing-values-by-mean-in-each-group

[ ]:
```
emp_df3['MeanFilledPrevSal'] = emp_df3.
↪groupby('Department')['BackwardFilledPrevSal'].transform(lambda x: x.
↪fillna(x.mean()))
emp_df3
```

1.20.1 References:

1. Pandas Documentation

2. **Real Python**

3. **TDS - Window Functions**

[ ]: