2.2-Pandas Transformations & Aggregations

KeytoDataScience.com

Documentation of pandas: https://pandas.pydata.org/docs/reference/frame.html

Table of Contents

1 Transforming Data

1.1 Inspecting a DataFrame

When you get a new DataFrame to work with, the first thing you need to do is explore it and see what it contains. There are several useful methods and attributes for this.

- **.head()** returns the first few rows (the "head" of the DataFrame).
- **.info()** shows information on each of the columns, such as the data type and number of missing values.
- **.shape** returns the number of rows and columns of the DataFrame.
- **.describe()** calculates a few summary statistics for each column.

In [3]:

```
import pandas as pd
df = pd.read_csv('adult.csv')
```

```python
df.head()
```

Out[3]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Female

In [4]:

```python
type(df['age'])
```

Out[4]:

```
pandas.core.series.Series
```

In [5]:

```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 48842 entries, 0 to 48841
Data columns (total 15 columns):
 #   Column           Non-Null Count  Dtype 
---  ------           --------------  ----- 
 0   age              48842 non-null  int64 
 1   workclass        48842 non-null  object
 2   fnlwgt           48842 non-null  int64 
 3   education        48842 non-null  object
 4   educational-num  48842 non-null  int64 
 5   marital-status   48842 non-null  object
 6   occupation       48842 non-null  object
 7   relationship     48842 non-null  object
 8   race             48842 non-null  object
 9   gender           48842 non-null  object
 10  capital-gain     48842 non-null  int64 
 11  capital-loss     48842 non-null  int64 
 12  hours-per-week   48842 non-null  int64 
 13  native-country   48842 non-null  object
 14  income           48842 non-null  object
dtypes: int64(6), object(9)
memory usage: 5.6+ MB
```

In [6]:

```python
df.shape
```

```
(48842, 15)
```

Out[6]:

In [7]:

```
df.describe()
```

Out[7]:

	age	fnlwgt	educational-num	capital-gain	capital-loss	hours-per-week
count	48842.000000	4.884200e+04	48842.000000	48842.000000	48842.000000	48842.000000
mean	38.643585	1.896641e+05	10.078089	1079.067626	87.502314	40.422382
std	13.710510	1.056040e+05	2.570973	7452.019058	403.004552	12.391444
min	17.000000	1.228500e+04	1.000000	0.000000	0.000000	1.000000
25%	28.000000	1.175505e+05	9.000000	0.000000	0.000000	40.000000
50%	37.000000	1.781445e+05	10.000000	0.000000	0.000000	40.000000
75%	48.000000	2.376420e+05	12.000000	0.000000	0.000000	45.000000
max	90.000000	1.490400e+06	16.000000	99999.000000	4356.000000	99.000000

In [8]:

```
df.describe()['fnlwgt'].astype('int64')
```

Out[8]:

```
count      48842
mean      189664
std       105604
min        12285
25%       117550
50%       178144
75%       237642
max      1490400
Name: fnlwgt, dtype: int64
```

↥ back to top

1.2 Parts of a DataFrame

To better understand DataFrame objects, it's useful to know that they consist of three components, stored as attributes:

- **.values:** A two-dimensional NumPy array of values.
- **.columns:** An index of columns: the column names.
- **.index:** An index for the rows: either row numbers or row names.

In [9]:

```
array = df.values
array
```

Out[9]:

```
array([[25, 'Private', 226802, ..., 40, 'United-States', '<=50K'],
       [38, 'Private', 89814, ..., 50, 'United-States', '<=50K'],
       [28, 'Local-gov', 336951, ..., 40, 'United-States', '>50K'],
       ...,
       [58, 'Private', 151910, ..., 40, 'United-States', '<=50K'],
       [22, 'Private', 201490, ..., 20, 'United-States', '<=50K'],
       [52, 'Self-emp-inc', 287927, ..., 40, 'United-States', '>50K']],
      dtype=object)
```

In [10]:
```
df.columns
```

Out[10]:
```
Index(['age', 'workclass', 'fnlwgt', 'education', 'educational-num',
       'marital-status', 'occupation', 'relationship', 'race', 'gender',
       'capital-gain', 'capital-loss', 'hours-per-week', 'native-country',
       'income'],
      dtype='object')
```

In [11]:
```
df.index
```

Out[11]:
```
RangeIndex(start=0, stop=48842, step=1)
```

↥ back to top

1.3 Sorting rows

Finding interesting bits of data in a DataFrame is often easier if you change the order of the rows. You can sort the rows by passing a column name to **.sort_values()**.

In cases where rows have the same value (this is common if you sort on a categorical variable), you may wish to break the ties by sorting on another column. You can sort on multiple columns in this way by passing a list of column names.

Sort on	Syntax
one column	`df.sort_values("col1")`
multiple columns	`df.sort_values(["col1","col2"])`

By combining .sort_values() with .head(), you can answer questions in the form, **"What are the top cases where...?"**.

In [12]:
```
df.sort_values(["hours-per-week","age"], ascending=False).head()
```

Out[12]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	ge
8427	90	Federal-gov	311184	Masters	14	Divorced	Prof-specialty	Not-in-family	White	I
31637	90	Private	90523	HS-grad	9	Widowed	Transport-moving	Unmarried	White	I
8677	73	Self-emp-not-inc	228899	7th-8th	4	Never-married	Adm-clerical	Not-in-family	White	Fe
32885	73	Self-emp-not-inc	102510	7th-8th	4	Married-civ-spouse	Farming-fishing	Husband	White	I
36278	72	Private	268861	7th-8th	4	Widowed	Other-service	Not-in-family	White	Fe

↥ back to top

1.4 Subsetting columns

When working with data, you may not need all of the variables in your dataset. Square-brackets ([]) can be used to select only the columns that matter to you in an order that makes sense to you.

- To select only "col_a" of the DataFrame df, use
  - `df["col_a"]`
- To select "col_a" and "col_b" of df, use
  - `df[["col_a", "col_b"]]`

In [13]:

```
df[["age","occupation","relationship"]].head(n=10)
```

Out[13]:

	age	occupation	relationship
0	25	Machine-op-inspct	Own-child
1	38	Farming-fishing	Husband
2	28	Protective-serv	Husband
3	44	Machine-op-inspct	Husband
4	18	?	Own-child
5	34	Other-service	Not-in-family
6	29	?	Unmarried
7	63	Prof-specialty	Husband
8	24	Other-service	Unmarried
9	55	Craft-repair	Husband

A large part of data science is about finding which bits of your dataset are interesting. One of the simplest techniques for this is to find a subset of rows that match some criteria. This is sometimes known as filtering rows or selecting rows.

There are many ways to subset a DataFrame, perhaps the most common is to use relational operators to return True or False for each row, then pass that inside square brackets.

- dogs[dogs["height_cm"] > 60]
- dogs[dogs["color"] == "tan"]

You can filter for multiple conditions at once by using the "logical and" operator, &.

- dogs[(dogs["height_cm"] > 60) & (dogs["col_b"] == "tan")]

In [14]:

```
df[(df["hours-per-week"]> 90) & (df["age"]>70)]
```

Out[14]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	ge
8427	90	Federal-gov	311184	Masters	14	Divorced	Prof-specialty	Not-in-family	White	I
8677	73	Self-emp-not-inc	228899	7th-8th	4	Never-married	Adm-clerical	Not-in-family	White	Fe
31637	90	Private	90523	HS-grad	9	Widowed	Transport-moving	Unmarried	White	I
32885	73	Self-emp-not-inc	102510	7th-8th	4	Married-civ-spouse	Farming-fishing	Husband	White	I
36278	72	Private	268861	7th-8th	4	Widowed	Other-service	Not-in-family	White	Fe

↥ back to top

1.5 Subsetting rows by categorical variables

Subsetting data based on a categorical variable often involves using the "or" operator (|) to select rows from multiple categories. This can get tedious when you want all states in one of three different regions,

for example. Instead, use the .isin() method, which will allow you to tackle this problem by writing one condition instead of three separate ones.

- colors = ["brown", "black", "tan"]
- condition = dogs["color"].isin(colors)
- dogs[condition]

In [15]:

```
print(df["occupation"].unique())
```

```
['Machine-op-inspct' 'Farming-fishing' 'Protective-serv' '?'
 'Other-service' 'Prof-specialty' 'Craft-repair' 'Adm-clerical'
 'Exec-managerial' 'Tech-support' 'Sales' 'Priv-house-serv'
 'Transport-moving' 'Handlers-cleaners' 'Armed-Forces']
```

In [19]:

```
lst = ['Tech-support','Sales','Armed-Forces']
condition = df["occupation"].isin(lst)
```

In [20]:

```
df[condition]
```

Out[20]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	g

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race
20	34	Private	107914	Bachelors	13	Married-civ-spouse	Tech-support	Husband	White
32	24	Self-emp-not-inc	188274	Bachelors	13	Never-married	Sales	Not-in-family	White
41	44	Self-emp-inc	120277	Assoc-voc	11	Married-civ-spouse	Sales	Husband	White
61	39	Private	118429	Some-college	10	Divorced	Sales	Not-in-family	White
69	43	Private	102606	HS-grad	9	Married-civ-spouse	Sales	Husband	White
...	...	...	...	...	...	...	...	...	...
48810	29	Private	125976	HS-grad	9	Separated	Sales	Unmarried	White
48827	37	Private	198216	Assoc-acdm	12	Divorced	Tech-support	Not-in-family	White
48833	43	Private	84661	Assoc-voc	11	Married-civ-spouse	Sales	Husband	White
48834	32	Private	116138	Masters	14	Never-married	Tech-support	Not-in-family	Asian-Pac-Islander
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White

6965 rows × 15 columns

↥ back to top

1.6 Adding new columns

You aren't stuck with just the data you are given. Instead, you can add new columns to a DataFrame. This has many names, such as transforming, mutating, and feature engineering.

You can create new columns from scratch, but it is also common to derive them from other columns, for example, by adding columns together, or by changing their units.

In [14]:

```
df["hours-per-day"] = df["hours-per-week"]/5
df.head()
```

Out[14]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Female

↥ back to top

1.7 Operations

There are lots of operations with pandas that will be really useful to you, but don't fall into any distinct category.

In [15]:

```
import pandas as pd
df = pd.DataFrame({'col1':[1,2,3,4],'col2':[444,555,666,444],'col3':['abc','def','ghi',
df.head()
```

Out[15]:

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

In [16]:

```
print(df['col2'].nunique())
print('--------------------------')
print(df['col2'].unique())
print('--------------------------')
print(df['col2'].value_counts())
```

```
3
--------------------------
[444 555 666]
--------------------------
444    2
555    1
```

```
666    1
Name: col2, dtype: int64
```

In [17]:
```
df['col1'].sum()
```

Out[17]:
```
10
```

In [18]:
```
df['col3'].apply(len)
```

Out[18]:
```
0    3
1    3
2    3
3    3
Name: col3, dtype: int64
```

In [19]:
```
# Applying functions using lambda
df['times2'] = df['col1'].apply(lambda x: x**2)
df.head()
```

Out[19]:

	col1	col2	col3	times2
0	1	444	abc	1
1	2	555	def	4
2	3	666	ghi	9
3	4	444	xyz	16

In [20]:
```
# Check if column 'col3' starts with 'a'
df['col3'].apply(lambda x: 1 if x.startswith('a') else 0)
```

Out[20]:
```
0    1
1    0
2    0
3    0
Name: col3, dtype: int64
```

In [21]:
```
df['original'] = df.apply(lambda rec: rec.times2/rec.col1, axis=1)
df.head()
```

Out[21]:

	col1	col2	col3	times2	original
0	1	444	abc	1	1.0
1	2	555	def	4	2.0
2	3	666	ghi	9	3.0
3	4	444	xyz	16	4.0

↥ back to top

2 Aggregating Data

2.1 Mean and median

Summary statistics are exactly what they sound like - they summarize many numbers in one statistic. For example, mean, median, minimum, maximum, and standard deviation are summary statistics. Calculating summary statistics allows you to get a better sense of your data, even if there's a lot of it.

In [22]:
```
df = pd.read_csv('adult.csv')
df['age'].describe()
```

Out[22]:
```
count    48842.000000
mean        38.643585
std         13.710510
min         17.000000
25%         28.000000
50%         37.000000
75%         48.000000
max         90.000000
Name: age, dtype: float64
```

In [23]:
```
print(df['age'].mean())
print(df['age'].median())
print('---------------------')
print(df['age'].describe().loc['mean'])
print(df['age'].describe()['50%'])
```

```
38.64358543876172
37.0
---------------------
38.64358543876172
37.0
```

↥ back to top

2.2 Summarizing dates

Summary statistics can also be calculated on date columns which have values with the data type datetime64. Some summary statistics — like mean — don't make a ton of sense on dates, but others are super helpful, for example minimum and maximum, which allow you to see what time range your data covers.

In [24]:
```
time_df = pd.DataFrame({"Date":['7/3/2020','7/4/2020','7/5/2020','7/6/2020','7/7/2020']
time_df["Date"] = pd.to_datetime(time_df["Date"], infer_datetime_format=True)
print(time_df["Date"].min())
print(time_df["Date"].max())
```

```
2020-07-03 00:00:00
2020-07-07 00:00:00
```

↥ back to top

2.3 Efficient summaries

While pandas and NumPy have tons of functions, sometimes you may need a different function to summarize your data.

The .agg() method allows you to apply your own custom functions to a DataFrame, as well as apply functions to more than one column of a DataFrame at once, making your aggregations super efficient.

In the custom function for this exercise, "IQR" is short for inter-quartile range, which is the 75th percentile minus the 25th percentile. It's an alternative to standard deviation that is helpful if your data contains outliers.

In [25]:

```
# A custom IQR function
def iqr(column):
    return column.quantile(0.75) - column.quantile(0.25)

# Print IQR of the age column
print(df['age'].agg(iqr))

print('---------------------')
print(df['age'].describe()['75%'] - df['age'].describe()['25%'])
```

```
20.0
---------------------
20.0
```

↥ back to top

2.4 Cumulative statistics

Cumulative statistics can also be helpful in tracking summary statistics over time. In this exercise, you'll calculate the cumulative sum and cumulative max of a department's weekly sales, which will allow you to identify what the total sales were so far as well as what the highest weekly sales were so far.

In [26]:

```
# Sort df by age
sorted_df = df.sort_values('age')

# Get the cumulative sum of hours-per-week
sorted_df['cum_hours_per_week'] = sorted_df['hours-per-week'].cumsum()

# Get the cumulative max of hours_per_week
sorted_df['cum_max_hours_per_week'] = sorted_df['hours-per-week'].cummax()

# See the columns you calculated
print(sorted_df[["age", "hours-per-week", "cum_hours_per_week", "cum_max_hours_per_week
```

```
       age  hours-per-week  cum_hours_per_week  cum_max_hours_per_week
32598   17              26                  26                      26
29817   17              35                  61                      35
36580   17              15                  76                      35
26409   17              12                  88                      35
19520   17              40                 128                      40
```

↥ back to top

2.5 Dropping duplicates

Removing duplicates is an essential skill to get accurate counts, because often you don't want to count the same thing multiple times.

In [23]:

```
"""
If you want to have the changes applied to the same dataframe, keep inplace=True. You d

df.drop_duplicates(subset="native-country", inplace=True)
"""

new_df = df.drop_duplicates(subset="native-country", inplace=False)

new_df.head()
```

Out[23]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gend
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Ma
19	40	Private	85019	Doctorate	16	Married-civ-spouse	Prof-specialty	Husband	Asian-Pac-Islander	Ma
23	25	Private	220931	Bachelors	13	Never-married	Prof-specialty	Not-in-family	White	Ma
37	22	Private	248446	5th-6th	3	Never-married	Priv-house-serv	Not-in-family	White	Ma
46	39	Private	290208	7th-8th	4	Married-civ-spouse	Craft-repair	Husband	White	Ma

In [24]:

```
# Drop duplicate on multiple columns
new_df = df.drop_duplicates(subset=["workclass", "occupation"], keep='first')

new_df.head()
```

Out[24]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Female
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male

↥ back to top

2.6 Counting categorical variables

Counting is a great way to get an overview of your data and to spot curiosities that you might not notice otherwise.

In [29]:
```
df["workclass"].value_counts(sort=True)
```

Out[29]:
```
Private             33906
Self-emp-not-inc     3862
Local-gov            3136
?                    2799
State-gov            1981
Self-emp-inc         1695
Federal-gov          1432
Without-pay            21
Never-worked           10
Name: workclass, dtype: int64
```

In [30]:
```
df["workclass"].value_counts(normalize=True)
```

Out[30]:
```
Private             0.694198
Self-emp-not-inc    0.079071
Local-gov           0.064207
?                   0.057307
State-gov           0.040559
Self-emp-inc        0.034704
Federal-gov         0.029319
Without-pay         0.000430
Never-worked        0.000205
Name: workclass, dtype: float64
```

↥ back to top

2.7 Calculations with .groupby()

The groupby method allows you to group rows of data together and call aggregate functions

In [31]:
```
import pandas as pd
# Create dataframe
data = {'Company':['GOOG','GOOG','GOOG','MSFT','MSFT','MSFT','FB','FB','FB'],
       'Person':['Sam','Charlie','Ana','Amy','Vanessa','Paul','Carl','Sarah','Sean'],
       'Sales':[200,120,50,340,124,150,243,350,450]}
```

In [32]:
```
df = pd.DataFrame(data)
```

In [33]:
```
df
```

Out[33]:

	Company	Person	Sales
0	GOOG	Sam	200
1	GOOG	Charlie	120
2	GOOG	Ana	50
3	MSFT	Amy	340
4	MSFT	Vanessa	124
5	MSFT	Paul	150
6	FB	Carl	243
7	FB	Sarah	350
8	FB	Sean	450

Now you can use the .groupby() method to group rows together based off of a column name.

For instance let's group by Company. This will create a DataFrameGroupBy object:

In [34]:
```
df.groupby('Company')
```

Out[34]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000025101A59470>
```

You can save this object as a new variable:

In [35]:
```
by_comp = df.groupby("Company")
```

And then call aggregate methods off the object:

In [36]:
```
by_comp['Sales'].mean()
```

Out[36]:
```
Company
FB      347.666667
GOOG    123.333333
MSFT    204.666667
Name: Sales, dtype: float64
```

In [37]:
```
df.groupby('Company')['Sales'].mean()
```

Out[37]:
```
Company
FB      347.666667
GOOG    123.333333
MSFT    204.666667
Name: Sales, dtype: float64
```

More examples of aggregate methods:

In [38]:
```
df.groupby('Company')['Sales'].std()
```

Out[38]:
```
Company
FB      103.519724
GOOG     75.055535
MSFT    117.920877
Name: Sales, dtype: float64
```

In [39]:
```
df.groupby('Company')['Sales','Person'].min()
```

Out[39]:

	Sales	Person
Company		
FB	243	Carl
GOOG	50	Ana
MSFT	124	Amy

In [40]:
```
df.groupby('Company')['Sales','Person'].max()
```

Out[40]:

	Sales	Person
Company		
FB	450	Sean
GOOG	200	Sam
MSFT	340	Vanessa

In [41]:
```
df.loc[df.groupby('Company')['Sales'].idxmin()]
```

Out[41]:

	Company	Person	Sales
6	FB	Carl	243
2	GOOG	Ana	50
4	MSFT	Vanessa	124

In [42]:
```
df.groupby('Company')['Sales','Person'].count()
```

Out[42]:

	Sales	Person
Company		
FB	3	3
GOOG	3	3
MSFT	3	3

In [43]:
```
df.groupby('Company')['Sales'].describe()
```

Out[43]:

	count	mean	std	min	25%	50%	75%	max
Company								
FB	3.0	347.666667	103.519724	243.0	296.5	350.0	400.0	450.0
GOOG	3.0	123.333333	75.055535	50.0	85.0	120.0	160.0	200.0
MSFT	3.0	204.666667	117.920877	124.0	137.0	150.0	245.0	340.0

In [44]:

```
df.groupby('Company')['Sales'].describe().transpose()
```

Out[44]:

Company	FB	GOOG	MSFT
count	3.000000	3.000000	3.000000
mean	347.666667	123.333333	204.666667
std	103.519724	75.055535	117.920877
min	243.000000	50.000000	124.000000
25%	296.500000	85.000000	137.000000
50%	350.000000	120.000000	150.000000
75%	400.000000	160.000000	245.000000
max	450.000000	200.000000	340.000000

In [45]:

```
df.groupby('Company')['Sales'].describe().transpose()['GOOG']
```

Out[45]:

```
count      3.000000
mean     123.333333
std       75.055535
min       50.000000
25%       85.000000
50%      120.000000
75%      160.000000
max      200.000000
Name: GOOG, dtype: float64
```

In [46]:

```
### Multiple grouped statistics
df.groupby('Company').agg([min,max])
```

Out[46]:

	Person		Sales	
	min	max	min	max
Company				
FB	Carl	Sean	243	450
GOOG	Ana	Sam	50	200
MSFT	Amy	Vanessa	124	340

In [47]:
```
df.groupby('Company').agg(
    {
        'Person':['min'],
     'Sales':['min', 'max']
    })
```

Out[47]:

	Person	Sales	
	min	min	max
Company			
FB	Carl	243	450
GOOG	Ana	50	200
MSFT	Amy	124	340

In [48]:
```
df.groupby(['Company','Person'])['Sales'].mean()
```

Out[48]:
```
Company  Person
FB       Carl       243
         Sarah      350
         Sean       450
GOOG     Ana         50
         Charlie    120
         Sam        200
MSFT     Amy        340
         Paul       150
         Vanessa    124
Name: Sales, dtype: int64
```

In [49]:
```
df.groupby(['Company'])['Person','Sales'].min()
```

Out[49]:

	Person	Sales
Company		
FB	Carl	243
GOOG	Ana	50
MSFT	Amy	124

↥ back to top

2.8 Pivot Tables

In [50]:
```
# df.groupby('Company')['Sales'].mean()

df.pivot_table(values="Sales", index="Company", aggfunc='mean')
```

Out[50]:

	Sales
Company	
FB	347.666667

	Sales
Company	
GOOG	123.333333
MSFT	204.666667

In [51]:

```
import numpy as np
df.pivot_table(values="Sales", index="Company", aggfunc='min')
```

Out[51]:

	Sales
Company	
FB	243
GOOG	50
MSFT	124

In [52]:

```
df.pivot_table(values="Sales", index="Company", aggfunc=[np.mean,np.median])
```

Out[52]:

	mean	median
	Sales	Sales
Company		
FB	347.666667	350
GOOG	123.333333	120
MSFT	204.666667	150

In [53]:

```
# df.groupby(['Company','Person'])['Sales'].mean()

df.pivot_table(values="Sales", index="Company", columns="Person")
```

Out[53]:

Person	Amy	Ana	Carl	Charlie	Paul	Sam	Sarah	Sean	Vanessa
Company									
FB	NaN	NaN	243.0	NaN	NaN	NaN	350.0	450.0	NaN
GOOG	NaN	50.0	NaN	120.0	NaN	200.0	NaN	NaN	NaN
MSFT	340.0	NaN	NaN	NaN	150.0	NaN	NaN	NaN	124.0

In [54]:

```
df.pivot_table(values="Sales", index="Company", columns="Person", fill_value=0)
```

Out[54]:

Person	Amy	Ana	Carl	Charlie	Paul	Sam	Sarah	Sean	Vanessa
Company									

Person	Amy	Ana	Carl	Charlie	Paul	Sam	Sarah	Sean	Vanessa
Company									
FB	0	0	243	0	0	0	350	450	0
GOOG	0	50	0	120	0	200	0	0	0
MSFT	340	0	0	0	150	0	0	0	124

In [55]:

```
df.pivot_table(values="Sales", index="Company", columns="Person", fill_value=0, margins
```

Out[55]:

Person	Amy	Ana	Carl	Charlie	Paul	Sam	Sarah	Sean	Vanessa	All
Company										
FB	0	0	243	0	0	0	350	450	0	347.666667
GOOG	0	50	0	120	0	200	0	0	0	123.333333
MSFT	340	0	0	0	150	0	0	0	124	204.666667
All	340	50	243	120	150	200	350	450	124	225.000000

↥ back to top

Great Job!
