Module 5: MultiIndex

- Enables you to store and manipulate data with an arbitrary number of dimensions in lower-dimensional data structures like Series (1d) and DataFrame (2d).
- Categorize the data more effectively.

In [1]:

```
import pandas as pd
bigmac = pd.read_csv("data/bigmac.csv", parse_dates=["Date"])
bigmac.head()
```

Out[1]:

	Date	Country	Price in US Dollars
0	2016-01-01	Argentina	2.39
1	2016-01-01	Australia	3.74
2	2016-01-01	Brazil	3.35
3	2016-01-01	Britain	4.22
4	2016-01-01	Canada	4.14

Summarisation on this dataset

- No null values for every column.
- The Date column is converted into datetime type by using **parameter parse_dates**. We can do operations such as add date, subtract date and many more on this column
- All column types are precisely according to their data types.

1. Creta a MultiIndex DataFrame with .set_index() method

- As we have learned previously, the first parameter becomes the index. We can also pass a list of columns as its argument.
- If we do that, the outer column should have lesser unique values. In this case, column Date has only **12 unique values**.

In [2]:

```
bigmac["Date"].nunique()
```

Out[2]:

```
12
```

In [3]:

```
bigmac["Country"].nunique()
```

Out[3]:

58

In [4]:

```
bigmac.set_index(["Date", "Country"], inplace=True)
bigmac
```

Out[4]:

		Price in US Dollars
Date	**Country**	
	Argentina	2.39
	Australia	3.74
2016-01-01	Brazil	3.35
	Britain	4.22
	Canada	4.14
...	...	...
	Turkey	3.83
	UAE	2.99
2010-01-01	Ukraine	1.83
	United States	3.58
	Uruguay	3.32

652 rows × 1 columns

In [5]:

```
bigmac.sort_index(inplace=True)
```

In [6]:

```
bigmac.head()
```

Out[6]:

		Price in US Dollars
Date	**Country**	
	Argentina	1.84
	Australia	3.98
2010-01-01	Brazil	4.76
	Britain	3.67
	Canada	3.97

Another way we can set the MultiIndex is by using the **index_col parameter** in **read_csv()** method.

The output is the same, but we can complete this in one single line.

In [7]:

```
bigmac = pd.read_csv("data/bigmac.csv", index_col=["Date", "Country"], parse_dates=["Date"]
bigmac.sort_index(inplace=True)
bigmac.head(10)
```

Out[7]:

		Price in US Dollars
Date	**Country**	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97
	Chile	3.18
	China	1.83
	Colombia	3.91
	Costa Rica	3.52
	Czech Republic	3.71

As you can see, the column is automatically categorized into Date and Country. Since Date is first in the list, then the first column will be Date.

We can use type() built-in function to check our DataFrame index.

In [8]:

```
type(bigmac.index)
```

Out[8]:

```
pandas.core.indexes.multi.MultiIndex
```

In [9]:

```
bigmac.index.names
```

Out[9]:

```
FrozenList(['Date', 'Country'])
```

To extract information on every row, we have to provide two data from Date and Country.

In [10]:

```
bigmac.index[0]
```

Out[10]:

```
(Timestamp('2010-01-01 00:00:00'), 'Argentina')
```

In [11]:

```
bigmac.loc["2016-01-01", "Italy"]
```

Out[11]:

```
Price in US Dollars    4.3
Name: (2016-01-01 00:00:00, Italy), dtype: float64
```

2. get_levels_values() method

Returns an index of values for requested level, equal to the length of the index.

Parameters

- level : level is either the integer position of the level in the MultiIndex or the name of the level.

For this dataset, 0 is referring to the Date column, which is the first level of the MultiIndex DataFrame. Meanwhile, 1 refers to the 2nd level which is the Country column.

In [12]:

```
bigmac.index.get_level_values(level = 0)
```

Out[12]:

```
DatetimeIndex(['2010-01-01', '2010-01-01', '2010-01-01', '2010-01-01',
               '2010-01-01', '2010-01-01', '2010-01-01', '2010-01-01',
               '2010-01-01', '2010-01-01',
               ...
               '2016-01-01', '2016-01-01', '2016-01-01', '2016-01-01',
               '2016-01-01', '2016-01-01', '2016-01-01', '2016-01-01',
               '2016-01-01', '2016-01-01'],
              dtype='datetime64[ns]', name='Date', length=652, freq=None)
```

In [13]:

```
bigmac.index.get_level_values(level = 1)
```

Out[13]:

```
Index(['Argentina', 'Australia', 'Brazil', 'Britain', 'Canada', 'Chile',
       'China', 'Colombia', 'Costa Rica', 'Czech Republic',
       ...
       'Switzerland', 'Taiwan', 'Thailand', 'Turkey', 'UAE', 'Ukraine',
       'United States', 'Uruguay', 'Venezuela', 'Vietnam'],
      dtype='object', name='Country', length=652)
```

3. set_names() method on MultiIndex

Set Index or MultiIndex name.

Able to set new names partially and by level.

Hint : Another approach you can use is to rename the columns first before setting them as index.

In [14]:

```
bigmac.index.set_names(["Day", "Location"], inplace=True)
bigmac.head()
```

Out[14]:

		Price in US Dollars
Day	**Location**	
	Argentina	1.84
	Australia	3.98
2010-01-01	Brazil	4.76
	Britain	3.67
	Canada	3.97

In [15]:

```
bigmac.index.set_names(["Date", "Country"], inplace=True)
```

4. sort_index() Method

sort_index() method enables us to sort object accordingly.

For example, we can sort the Date in ascending order and Country in Descending order. This can be done by passing a list of Boolean to the parameter.

There are other parameters that we can use.

Reference : https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_index.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_index.html)

In [16]:

```
bigmac.sort_index(ascending=[True, False])
```

Out[16]:

		Price in US Dollars
Date	**Country**	
	Uruguay	3.32
	United States	3.58
2010-01-01	Ukraine	1.83
	UAE	2.99
	Turkey	3.83
...	...	...
	Brazil	3.35
	Belgium	4.25
2016-01-01	Austria	3.76
	Australia	3.74
	Argentina	2.39

652 rows × 1 columns

5. Extract rows from MultiIndex DataFrame

In the previous module, loc and iloc are used to extract data by rows. For MultiIndex, loc and iloc can also be used but we need to pass additional data.

In [17]:

```
bigmac.head(3)
```

Out[17]:

		Price in US Dollars
Date	**Country**	
	Argentina	1.84
2010-01-01	Australia	3.98
	Brazil	4.76

Since the DataFrame is using two indexes, we need to use two arguments to extract specific information. Those two arguments need to be in tuple format.

In [18]:

```
bigmac.loc[("2010-01-01","Argentina"),"Price in US Dollars"]
```

Out[18]:

```
Date        Country
2010-01-01  Argentina    1.84
Name: Price in US Dollars, dtype: float64
```

In [19]:

```
type(("2010-01-01","Argentina"))
```

Out[19]:

```
tuple
```

Here is an example of how we can extract data from Price in US Dollar column at a specific date. Since Country is also an index, by default it will appear as well.

In [20]:

```
bigmac.loc[("2010-01-01"),"Price in US Dollars"]
```

Out[20]:

```
Date        Country
2010-01-01  Argentina         1.84
            Australia         3.98
            Brazil            4.76
            Britain           3.67
            Canada            3.97
            Chile             3.18
            China             1.83
            Colombia          3.91
            Costa Rica        3.52
            Czech Republic    3.71
            Denmark           5.99
            Egypt             2.38
            Euro area         4.84
            Hong Kong         1.91
            Hungary           3.86
            Indonesia         2.24
            Israel            3.99
            Japan             3.50
            Latvia            3.09
            Lithuania         2.87
            Malaysia          2.08
            Mexico            2.50
            New Zealand       3.61
            Norway            7.02
            Pakistan          2.42
            Peru              2.81
            Philippines       2.21
            Poland            2.86
            Russia            2.34
            Saudi Arabia      2.67
            Singapore         3.19
            South Africa      2.46
            South Korea       2.98
            Sri Lanka         1.83
            Sweden            5.51
            Switzerland       6.30
            Taiwan            2.36
            Thailand          2.11
            Turkey            3.83
            UAE               2.99
            Ukraine           1.83
            United States     3.58
            Uruguay           3.32
Name: Price in US Dollars, dtype: float64
```

6. transpose() method

Pandas DataFrame **.transpose()** function transposes index and columns of the DataFrame. It reflects the DataFrame over its main diagonal by writing rows as columns and vice-versa.

In [21]:

```
bigmac
```

Out[21]:

		Price in US Dollars
Date	**Country**	
	Argentina	1.84
	Australia	3.98
2010-01-01	Brazil	4.76
	Britain	3.67
	Canada	3.97
...	...	...
	Ukraine	1.54
	United States	4.93
2016-01-01	Uruguay	3.74
	Venezuela	0.66
	Vietnam	2.67

652 rows × 1 columns

Bigmac DataFrame has 652 rows and one column. After applying the transpose() method, the row and column size changed.

In [22]:

```
bigmac.transpose()
```

Out[22]:

Date	**2010-01-01**									
Country	**Argentina**	**Australia**	**Brazil**	**Britain**	**Canada**	**Chile**	**China**	**Colombia**	**Costa Rica**	**Czech Republic**
Price in US Dollars	1.84	3.98	4.76	3.67	3.97	3.18	1.83	3.91	3.52	3.71

1 rows × 652 columns

Here is another example.

In [23]:

```
# Creating the DataFrame
df = pd.DataFrame({"A":[12, 4, 5, None, 1],
                   "B":[7, 2, 54, 3, None],
                   "C":[20, 16, 11, 3, 8],
                   "D":[14, 3, None, 2, 6]})
df
```

Out[23]:

	A	B	C	D
0	12.0	7.0	20	14.0
1	4.0	2.0	16	3.0
2	5.0	54.0	11	NaN
3	NaN	3.0	3	2.0
4	1.0	NaN	8	6.0

In [24]:

```
df.transpose()
```

Out[24]:

	0	1	2	3	4
A	12.0	4.0	5.0	NaN	1.0
B	7.0	2.0	54.0	3.0	NaN
C	20.0	16.0	11.0	3.0	8.0
D	14.0	3.0	NaN	2.0	6.0

7. swaplevel() method

Swap index from one level to another.

If we have 3 to 4 levels MultiIndex, then, i and j refer to the index level. Swap levels i and j in a MultiIndex on a particular axis.

Referance : https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.swaplevel.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.swaplevel.html)

In [25]:

```
bigmac.head()
```

Out[25]:

		Price in US Dollars
Date	**Country**	
2010-01-01	Argentina	1.84
	Australia	3.98
	Brazil	4.76
	Britain	3.67
	Canada	3.97

After applying swaplevel() method, the first index and second index swap with each other.

In [26]:

```
bigmac.swaplevel(i = "Date", j ="Country")
```

Out[26]:

		Price in US Dollars
Country	**Date**	
Argentina	2010-01-01	1.84
Australia	2010-01-01	3.98
Brazil	2010-01-01	4.76
Britain	2010-01-01	3.67
Canada	2010-01-01	3.97
...	...	...
Ukraine	2016-01-01	1.54
United States	2016-01-01	4.93
Uruguay	2016-01-01	3.74
Venezuela	2016-01-01	0.66
Vietnam	2016-01-01	2.67

652 rows × 1 columns

However, it is not practical to assign Country to be the first index as it has more unique values than Date column.

8. stack() method

Returns a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataFrame.

- change Pandas DataFrame into Pandas Series
- All the columns become the row

In [27]:

```
bigmac.stack()
```

Out[27]:

```
Date        Country
2010-01-01  Argentina      Price in US Dollars    1.84
            Australia      Price in US Dollars    3.98
            Brazil         Price in US Dollars    4.76
            Britain        Price in US Dollars    3.67
            Canada         Price in US Dollars    3.97
                                                  ...
2016-01-01  Ukraine        Price in US Dollars    1.54
            United States  Price in US Dollars    4.93
            Uruguay        Price in US Dollars    3.74
            Venezuela      Price in US Dollars    0.66
            Vietnam        Price in US Dollars    2.67
Length: 652, dtype: float64
```

Using method chaining, we can apply **to_frame()** to turn the Series into DataFrame.

In [28]:

```
bigmac.stack().to_frame()
```

Out[28]:

			0
Date	**Country**		
	Argentina	Price in US Dollars	1.84
	Australia	Price in US Dollars	3.98
2010-01-01	Brazil	Price in US Dollars	4.76
	Britain	Price in US Dollars	3.67
	Canada	Price in US Dollars	3.97
...	...	...	...
	Ukraine	Price in US Dollars	1.54
	United States	Price in US Dollars	4.93
2016-01-01	Uruguay	Price in US Dollars	3.74
	Venezuela	Price in US Dollars	0.66
	Vietnam	Price in US Dollars	2.67

652 rows × 1 columns

Here's another example using a different dataset named worldstats. In this dataset, we assign country and year as Index. We are left with two columns which are Population and GDP. Applying stack() method will change these remaining columns to rows.

In [29]:

```
world = pd.read_csv("data/worldstats.csv", index_col=["country", "year"])
world.head()
```

Out[29]:

		Population	GDP
country	**year**		
	2015	392022276.0	2.530102e+12
	2014	384222592.0	2.873600e+12
Arab World	2013	376504253.0	2.846994e+12
	2012	368802611.0	2.773270e+12
	2011	361031820.0	2.497945e+12

In [30]:

```
world = world.stack().to_frame()
world
```

Out[30]:

			0
country	**year**		
	2015	Population	3.920223e+08
		GDP	2.530102e+12
Arab World	2014	Population	3.842226e+08
		GDP	2.873600e+12
	2013	Population	3.765043e+08
...	...	...	...
	1962	GDP	1.117602e+09
	1961	Population	3.876638e+06
Zimbabwe		GDP	1.096647e+09
	1960	Population	3.752390e+06
		GDP	1.052990e+09

22422 rows × 1 columns

In [31]:

```
world.loc[("Arab World"), 2015]
```

```
C:\Users\Ismail\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: Perform
anceWarning: indexing past lexsort depth may impact performance.
  """Entry point for launching an IPython kernel.
```

Out[31]:

	0
Population	3.920223e+08
GDP	2.530102e+12

9. unstack() method

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.

- It is the opposite of stack() method.
- It changes rows into columns.

In [32]:

```
world.head()
```

Out[32]:

			0
country	**year**		
Arab World	2015	Population	3.920223e+08
		GDP	2.530102e+12
	2014	Population	3.842226e+08
		GDP	2.873600e+12
	2013	Population	3.765043e+08

One time unstack() method.

In [33]:

```
world.unstack()
```

Out[33]:

		0	
		Population	GDP
country	year		
	1960	8994793.0	5.377778e+08
	1961	9164945.0	5.488889e+08
Afghanistan	1962	9343772.0	5.466667e+08
	1963	9531555.0	7.511112e+08
	1964	9728645.0	8.000000e+08
...	...	...	...
	2011	14255592.0	1.095623e+10
	2012	14565482.0	1.239272e+10
Zimbabwe	2013	14898092.0	1.349023e+10
	2014	15245855.0	1.419691e+10
	2015	15602751.0	1.389294e+10

11211 rows × 2 columns

Two times unstack() method.

In [34]:

```
world.unstack().unstack()
```

Out[34]:

	0					
	Population					
year	1960	1961	1962	1963	1964	1965
country						
Afghanistan	8.994793e+06	9.164945e+06	9.343772e+06	9.531555e+06	9.728645e+06	9.935358e+
Albania	NaN	NaN	NaN	NaN	NaN	Na
Algeria	1.112489e+07	1.140486e+07	1.169015e+07	1.198513e+07	1.229597e+07	1.262695e+
Andorra	NaN	NaN	NaN	NaN	NaN	Na
Angola	NaN	NaN	NaN	NaN	NaN	Na
...	...	...	...	...	...	
West Bank and Gaza	NaN	NaN	NaN	NaN	NaN	Na
World	3.035056e+09	3.076121e+09	3.129064e+09	3.193947e+09	3.259355e+09	3.326054e+
Yemen, Rep.	NaN	NaN	NaN	NaN	NaN	Na
Zambia	3.049586e+06	3.142848e+06	3.240664e+06	3.342894e+06	3.449266e+06	3.559687e+
Zimbabwe	3.752390e+06	3.876638e+06	4.006262e+06	4.140804e+06	4.279561e+06	4.422132e+

252 rows × 112 columns

Three times unstack() method.

Since the index is not MultiIndex, the output will be a Series (the analog of stack when the columns are not a MultiIndex). The levels involved will automatically get sorted.

In [35]:

```
world.unstack().unstack().unstack()
```

Out[35]:

```
               year  country
0  Population  1960  Afghanistan           8.994793e+06
                     Albania                        NaN
                     Algeria               1.112489e+07
                     Andorra                        NaN
                     Angola                         NaN
                                                 ...
   GDP         2015  West Bank and Gaza    1.267740e+10
                     World                 7.343364e+13
                     Yemen, Rep.                    NaN
                     Zambia                2.120156e+10
                     Zimbabwe              1.389294e+10
Length: 28224, dtype: float64
```

Parameter in unstack() method

- **level** : Level(s) of index to unstack, can pass the level name.

In [36]:

```
world.head()
```

Out[36]:

			0
country	**year**		
Arab World	2015	Population	3.920223e+08
		GDP	2.530102e+12
	2014	Population	3.842226e+08
		GDP	2.873600e+12
	2013	Population	3.765043e+08

In [37]:

```
world.unstack(level='country')
```

Out[37]:

		0					
	country	Afghanistan	Albania	Algeria	Andorra	Angola	Antigu Barbu
year							
1960	Population	8.994793e+06	NaN	1.112489e+07	NaN	NaN	
	GDP	5.377778e+08	NaN	2.723638e+09	NaN	NaN	
1961	Population	9.164945e+06	NaN	1.140486e+07	NaN	NaN	
	GDP	5.488889e+08	NaN	2.434767e+09	NaN	NaN	
1962	Population	9.343772e+06	NaN	1.169015e+07	NaN	NaN	
...	...	...	...	...	...	...	
2013	GDP	2.004633e+10	1.278103e+10	2.097035e+11	3.249101e+09	1.249121e+11	1.200
2014	Population	3.162751e+07	2.893654e+06	3.893433e+07	NaN	2.422752e+07	9.090
	GDP	2.005019e+10	1.327796e+10	2.135185e+11	NaN	1.267751e+11	1.220
2015	Population	3.252656e+07	2.889167e+06	3.966652e+07	NaN	2.502197e+07	9.181
	GDP	1.919944e+10	1.145560e+10	1.668386e+11	NaN	1.026431e+11	1.297

112 rows × 252 columns

We can also use index position to choose which column we want to unstack. Index count starts with 0. Hence, the index for Country column is 0, followed by 1 for Year, 2 for Population and so on.

In [38]:

```
world.unstack(level=2)
```

Out[38]:

		0	
		Population	GDP
country	year		
	1960	8994793.0	5.377778e+08
	1961	9164945.0	5.488889e+08
Afghanistan	1962	9343772.0	5.466667e+08
	1963	9531555.0	7.511112e+08
	1964	9728645.0	8.000000e+08
...	...	...	...
	2011	14255592.0	1.095623e+10
	2012	14565482.0	1.239272e+10
Zimbabwe	2013	14898092.0	1.349023e+10
	2014	15245855.0	1.419691e+10
	2015	15602751.0	1.389294e+10

11211 rows × 2 columns

We can also provide a list to the level parameter. Notice that different sequence numbers in the list will prompt different outputs.

In [39]:

```
# Unstack year first, then unstack country column
world.unstack(level=[1,0])
```

Out[39]:

	0					
year	2015	2014	2013	2012	2011	2010
country	Arab World	Arab World	Arab World	Arab World	Arab World	Arab Worl
Population	3.920223e+08	3.842226e+08	3.765043e+08	3.688026e+08	3.610318e+08	3.531122e+0
GDP	2.530102e+12	2.873600e+12	2.846994e+12	2.773270e+12	2.497945e+12	2.103825e+1

2 rows × 11211 columns

In [40]:

```
# Unstack country first, then unstack year column.
world.unstack(level=[0,1])
```

Out[40]:

	0					
country	Arab World					
year	2015	2014	2013	2012	2011	201
Population	3.920223e+08	3.842226e+08	3.765043e+08	3.688026e+08	3.610318e+08	3.531122e+0
GDP	2.530102e+12	2.873600e+12	2.846994e+12	2.773270e+12	2.497945e+12	2.103825e+1

2 rows × 11211 columns

- **fill_value parameter** will handle all the NaN/Null values that appear in the output table. In this example, we replace the NaN values to 0.

In [41]:

```
world.unstack().unstack(fill_value=0)
```

Out[41]:

	0					
	Population					
year	1960	1961	1962	1963	1964	1965
country						
Afghanistan	8.994793e+06	9.164945e+06	9.343772e+06	9.531555e+06	9.728645e+06	9.935358e+
Albania	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+
Algeria	1.112489e+07	1.140486e+07	1.169015e+07	1.198513e+07	1.229597e+07	1.262695e+
Andorra	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+
Angola	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+
...	...	...	...	...	...	
West Bank and Gaza	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+
World	3.035056e+09	3.076121e+09	3.129064e+09	3.193947e+09	3.259355e+09	3.326054e+
Yemen, Rep.	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+
Zambia	3.049586e+06	3.142848e+06	3.240664e+06	3.342894e+06	3.449266e+06	3.559687e+
Zimbabwe	3.752390e+06	3.876638e+06	4.006262e+06	4.140804e+06	4.279561e+06	4.422132e+

252 rows × 112 columns

10. pivot() method

Reshape data (produce a “pivot” table) based on column values.

Before we apply the method, let's understand the dataset first.

In [42]:

```
sales = pd.read_csv("data/salesmen.csv", parse_dates=["Date"])
sales.head(3)
```

Out[42]:

	Date	Salesman	Revenue
0	2016-01-01	Bob	7172
1	2016-01-02	Bob	6362
2	2016-01-03	Bob	5982

In [43]:

```
len(sales)
```

Out[43]:

```
1830
```

In [44]:

```
sales["Salesman"].value_counts()
```

Out[44]:

```
Bob       366
Jeb       366
Dave      366
Ronald    366
Oscar     366
Name: Salesman, dtype: int64
```

Based on the dataset, we can understand that

- There only 5 unique salesmen.
- There are 1830 rows in total.

There is a better way to visualize the dataset in table. A good table makes understanding the data much easier.

From the table below, we can identify who has the highest revenue for each day. This method allows us to understand the table from different angles/perspectives.

In [45]:

```
sales.pivot(index="Date", columns="Salesman", values="Revenue")
```

Out[45]:

Salesman	Bob	Dave	Jeb	Oscar	Ronald
Date					
2016-01-01	7172	1864	4430	5250	2639
2016-01-02	6362	8278	8026	8661	4951
2016-01-03	5982	4226	5188	7075	2703
2016-01-04	7917	3868	3144	2524	4258
2016-01-05	7837	2287	938	2793	7771
...	...	...	...	...	...
2016-12-27	2045	2843	6666	835	2981
2016-12-28	100	8888	1243	3073	6129
2016-12-29	4115	9490	3498	6424	7662
2016-12-30	2577	3594	8858	7088	2570
2016-12-31	3845	6830	9717	8408	2619

366 rows × 5 columns

11. pivot_table method

Create a spreadsheet-style pivot table as a DataFrame.

parameters

- values : column to aggregate the columns that we use to calculate the values from.
- index : new Dataframe will be based on the category oft his column.
- aggfunc : by default, it is set as mean but we can change it to sum, max, min, etc.

In [46]:

```
sales.head(3)
```

Out[46]:

	Date	Salesman	Revenue
0	2016-01-01	Bob	7172
1	2016-01-02	Bob	6362
2	2016-01-03	Bob	5982

In [47]:

```
sales.pivot_table(values="Revenue", index="Salesman", aggfunc="mean")
```

Out[47]:

Salesman	Revenue
Bob	4992.292350
Dave	5079.407104
Jeb	5241.579235
Oscar	4857.319672
Ronald	4992.109290

Using groupby() method, we can also get the same result.

In [48]:

```
sales.groupby("Salesman")["Revenue"].mean().to_frame()
```

Out[48]:

Salesman	Revenue
Bob	4992.292350
Dave	5079.407104
Jeb	5241.579235
Oscar	4857.319672
Ronald	4992.109290

Let's try pivotting the new table based on Revenue column

In [49]:

```
foods = pd.read_csv("data/foods.csv")
foods.head(3)
```

Out[49]:

	First Name	Gender	City	Frequency	Item	Spend
0	Wanda	Female	Stamford	Weekly	Burger	15.66
1	Eric	Male	Stamford	Daily	Chalupa	10.56
2	Charles	Male	New York	Never	Sushi	42.14

- **index = ["Gender" , "Item"]** to categorise the spending into genders and items.
- **aggfunc = "mean"** means we want to know the mean value of the value column (average spending of an item by each gender).
- **values = "Spend"** is the column which the aggregate function will apply to.

In [50]:

```
foods.pivot_table(index=["Gender", "Item"], values="Spend", aggfunc="mean")
```

Out[50]:

		Spend
Gender	**Item**	
Female	Burger	49.930488
	Burrito	50.092000
	Chalupa	54.635000
	Donut	49.926316
	Ice Cream	49.788519
	Sushi	50.355699
Male	Burger	49.613919
	Burrito	48.344819
	Chalupa	49.186761
	Donut	43.649565
	Ice Cream	51.096000
	Sushi	55.614384

The table below displays the maximum spending for each item by gender. The data is also categorized into the 3 cities which are New York, Philadelphia and Stanford.

In [51]:

```
df = foods.pivot_table(index=["Gender", "Item"], values="Spend", aggfunc="max", columns="Ci
df
```

Out[51]:

	City	New York	Philadelphia	Stamford
Gender	**Item**			
Female	Burger	98.96	97.79	85.06
	Burrito	92.25	96.79	99.21
	Chalupa	98.43	99.29	98.78
	Donut	95.63	96.52	91.75
	Ice Cream	97.83	88.14	97.44
	Sushi	99.51	99.02	95.43
Male	Burger	90.32	99.68	97.20
	Burrito	98.04	93.27	95.07
	Chalupa	96.44	98.40	99.87
	Donut	86.70	93.12	99.26
	Ice Cream	97.65	99.24	99.17
	Sushi	93.85	97.12	98.48

12. melt() method

Opposite of pivot() method. This method condenses the columns and creates more rows.

In [52]:

```
df = foods.pivot_table(index=["Item"], values="Spend", aggfunc="max", columns="City")
df
```

Out[52]:

City	New York	Philadelphia	Stamford
Item			
Burger	98.96	99.68	97.20
Burrito	98.04	96.79	99.21
Chalupa	98.43	99.29	99.87
Donut	95.63	96.52	99.26
Ice Cream	97.83	99.24	99.17
Sushi	99.51	99.02	98.48

First, we have to reset the index. The index for DataFrame above is labelled with Item.

In [53]:

```
df.reset_index(inplace=True)
df
```

Out[53]:

City	Item	New York	Philadelphia	Stamford
0	Burger	98.96	99.68	97.20
1	Burrito	98.04	96.79	99.21
2	Chalupa	98.43	99.29	99.87
3	Donut	95.63	96.52	99.26
4	Ice Cream	97.83	99.24	99.17
5	Sushi	99.51	99.02	98.48

After applying the .melt() method, we can see that the three columns of City became one.

Parameter:

- **id_vars** : the variables that we don't want to change.

Reference : https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.melt.html)

In [54]:

```
df.melt(id_vars="Item")
```

Out[54]:

	Item	City	value
0	Burger	New York	98.96
1	Burrito	New York	98.04
2	Chalupa	New York	98.43
3	Donut	New York	95.63
4	Ice Cream	New York	97.83
5	Sushi	New York	99.51
6	Burger	Philadelphia	99.68
7	Burrito	Philadelphia	96.79
8	Chalupa	Philadelphia	99.29
9	Donut	Philadelphia	96.52
10	Ice Cream	Philadelphia	99.24
11	Sushi	Philadelphia	99.02
12	Burger	Stamford	97.20
13	Burrito	Stamford	99.21
14	Chalupa	Stamford	99.87
15	Donut	Stamford	99.26
16	Ice Cream	Stamford	99.17
17	Sushi	Stamford	98.48

Do you remember about stack() and unstack() methods? We can actually generate almost similar table like the previous one by using stack() method.

In [55]:

```
df = foods.pivot_table(index=["Item"], values="Spend", aggfunc="max", columns="City")
df
```

Out[55]:

City	New York	Philadelphia	Stamford
Item			
Burger	98.96	99.68	97.20
Burrito	98.04	96.79	99.21
Chalupa	98.43	99.29	99.87
Donut	95.63	96.52	99.26
Ice Cream	97.83	99.24	99.17
Sushi	99.51	99.02	98.48

In [56]:

```
df.stack().to_frame().sort_values("City")
```

Out[56]:

		0
Item	City	
Burger	New York	98.96
Sushi	New York	99.51
Burrito	New York	98.04
Chalupa	New York	98.43
Donut	New York	95.63
Ice Cream	New York	97.83
Burger	Philadelphia	99.68
Burrito	Philadelphia	96.79
Chalupa	Philadelphia	99.29
Sushi	Philadelphia	99.02
Donut	Philadelphia	96.52
Ice Cream	Philadelphia	99.24
	Stamford	99.17
Chalupa	Stamford	99.87
Burrito	Stamford	99.21
Burger	Stamford	97.20
Donut	Stamford	99.26
Sushi	Stamford	98.48