Some Pre-Concepts:

Before starting going through functions and implementations, I would like to emphasize the importance of Understanding the Axis and the Inplace parameter. 1) Understanding “Axis”

A DataFrame object has two axes: “axis 0” and “axis 1”:

- axis 0: Wherever you see this -> it represents rows
- axis 1: Wherever you see this -> it represents columns

2) Understanding “Inplace”

- Understanding the “inplace” parameter can help us a lot of time and memory!
- When inplace = False -> which is the default, then the operation is performed and it returns a copy of the object. You then need to save it to something.

```
temp=df.set_index(‘CustomerId’)# here by Default inplace = False
temp
# While , When inplace = True -> the data is modified in place, which means it will return nothing and the dataframe is now updated.
```

```
df.set_index(‘CustomerId’,inplace=True)
df
```

Data Manipulation with Pandas

Pandas is the most widely used library of python for data science. It is incredibly helpful in manipulating the data so that you can derive better insights and build great machine learning models.

Table of Contents

Loading dataset

***In this notebook use the Big Mart Sales Data. You can download the data from : https://www.kaggle.com/brijbhushannanda1979/bigmart-sales-data?select=Test.csv (https://www.kaggle.com/brijbhushannanda1979/bigmart-sales-data?select=Test.csv)

In [1]:

```python
import pandas as pd
import numpy as np

# read the dataset
data_BM = pd.read_csv('bigmart_data.csv')
# drop the null values
data_BM = data_BM.dropna(how="any")
# view the top results
data_BM.head()
```

Out[1]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Out
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	OUT049	1999	Medium	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009	Medium	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	OUT049	1999	Medium	
4	NCD19	8.930	Low Fat	0.000000	Household	53.8614	OUT013	1987	High	
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	OUT018	2009	Medium	

1. Sorting dataframes

Pandas data frame has two useful functions

- **sort_values()**: to sort pandas data frame by one or more columns
- **sort_index()**: to sort pandas data frame by row index

Each of these functions come with numerous options, like sorting the data frame in specific order (ascending or descending), sorting in place, sorting with missing values, sorting by specific algorithm etc.

Suppose you want to sort the dataframe by "Outlet_Establishment_Year" then you will use **sort_values**

In [2]:

```
# sort by year
sorted_data = data_BM.sort_values(by='Outlet_Establishment_Year')
# print sorted data
sorted_data[:5]
```

Out[2]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size
2812	FDR60	14.30	Low Fat	0.130307	Baking Goods	75.7328	OUT013	1987	High
5938	NCJ06	20.10	Low Fat	0.034624	Household	118.9782	OUT013	1987	High
3867	FDY38	13.60	Regular	0.119077	Dairy	231.2300	OUT013	1987	High
1307	FDB37	20.25	Regular	0.022922	Baking Goods	240.7538	OUT013	1987	High
5930	NCA18	10.10	Low Fat	0.056031	Household	115.1492	OUT013	1987	High

- Now `sort_values` takes multiple options like:
  - `ascending` : The default sorting order is ascending, when you pass False here then it sorts in descending order.
  - `inplace` : whether to do inplace sorting or not

In [3]:
```
# sort in place and descending order
data_BM.sort_values(by='Outlet_Establishment_Year', ascending=False, inplace=True)
data_BM[:5]
```

Out[3]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size
2825	FDL16	12.85	Low Fat	0.169139	Frozen Foods	46.4060	OUT018	2009	Medium
7389	NCD42	16.50	Low Fat	0.012689	Health and Hygiene	39.7506	OUT018	2009	Medium
2165	DRJ39	20.25	Low Fat	0.036474	Dairy	218.3482	OUT018	2009	Medium
2162	FDR60	14.30	Low Fat	0.130946	Baking Goods	76.7328	OUT018	2009	Medium
2158	FDM58	16.85	Regular	0.080015	Snack Foods	111.8544	OUT018	2009	Medium

You might want to sort a data frame based on the values of multiple columns. We can specify the columns we want to sort by as a list in the argument for sort_values().

In [4]:

```
# read the dataset
data_BM = pd.read_csv('bigmart_data.csv')
# drop the null values
data_BM = data_BM.dropna(how="any")

# sort by multiple columns
data_BM.sort_values(by=['Outlet_Establishment_Year', 'Item_Outlet_Sales'], ascending=False)[:5]
```

Out[4]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size
43	FDC02	21.35	Low Fat	0.069103	Canned	259.9278	OUT018	2009	Medium
2803	FDU51	20.20	Regular	0.096907	Meat	175.5028	OUT018	2009	Medium
641	FDY51	12.50	Low Fat	0.081465	Meat	220.7798	OUT018	2009	Medium
2282	NCX30	16.70	Low Fat	0.026729	Household	248.4776	OUT018	2009	Medium
2887	FDR25	17.00	Regular	0.140090	Canned	265.1884	OUT018	2009	Medium

- Note that when sorting by multiple columns, pandas sort_value() uses the first variable first and second variable next.
- We can see the difference by switching the order of column names in the list.

In [5]:
```
# changed the order of columns
data_BM.sort_values(by=['Item_Outlet_Sales', 'Outlet_Establishment_Year'], ascending=False, inplace=True)
data_BM[:5]
```

Out[5]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size
4888	FDF39	14.850	Regular	0.019495	Dairy	261.2910	OUT013	1987	High
4289	NCM05	6.825	Low Fat	0.059847	Health and Hygiene	262.5226	OUT046	1997	Small
6409	FDA21	13.650	Low Fat	0.035931	Snack Foods	184.4924	OUT013	1987	High
4991	NCQ53	17.600	Low Fat	0.018905	Health and Hygiene	234.6590	OUT046	1997	Small
5752	FDI15	13.800	Low Fat	0.141326	Dairy	265.0884	OUT035	2004	Small

- We can use **sort_index()** to sort pandas dataframe to sort by row index or names.
- In this example, row index are numbers and in the earlier example we sorted data frame by 'Item_Outlet_Sales', 'Outlet_Establishment_Year' and therefore the row index are jumbled up.
- We can sort by row index (with inplace=True option) and retrieve the original dataframe.

In [6]:
```
# sort by index
data_BM.sort_index(inplace=True)
data_BM[:5]
```

Out[6]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Out
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	OUT049	1999	Medium	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009	Medium	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	OUT049	1999	Medium	
4	NCD19	8.930	Low Fat	0.000000	Household	53.8614	OUT013	1987	High	
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	OUT018	2009	Medium	

2. Merging dataframes

- Joining and merging DataFrames is the core process to start with data analysis and machine learning tasks.
- It is one of the toolkits which every Data Analyst or Data Scientist should master because in almost all the cases data comes from multiple source and files.
- Pandas has two useful functions for merging dataframes:
  - **concat()**
  - **merge()**

Creating dummy data

In [7]:
```
# create dummy data
df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                   index=[0, 1, 2, 3])


df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                   index=[4, 5, 6, 7])


df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                   index=[8, 9, 10, 11])
```

a. concat() for combining dataframes

- Suppose you have the following three dataframes: df1, df2 and df3 and you want to combine them **"row-wise"** so that they become a single dataframe like the given image:
- You can use **concat()** here. You will have to pass the names of the DataFrames in a list as the argument to the concat().

In [8]:
```
# combine dataframes
result = pd.concat([df1, df2, df3])
result
```

Out[8]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

- pandas also provides you with an option to label the DataFrames, after the concatenation, with a key so that you may know which data came from which DataFrame.
- You can achieve the same by passing additional argument **keys** specifying the label names of the DataFrames in a list.

In [9]:
```python
# combine dataframes
result = pd.concat([df1, df2, df3], keys=['x', 'y', 'z'])
result
```

Out[9]:

		A	B	C	D
x	0	A0	B0	C0	D0
	1	A1	B1	C1	D1
	2	A2	B2	C2	D2
	3	A3	B3	C3	D3
y	4	A4	B4	C4	D4
	5	A5	B5	C5	D5
	6	A6	B6	C6	D6
	7	A7	B7	C7	D7
z	8	A8	B8	C8	D8
	9	A9	B9	C9	D9
	10	A10	B10	C10	D10
	11	A11	B11	C11	D11

- Mentioning the keys also makes it easy to retrieve data corresponding to a particular DataFrame.
- You can retrieve the data of DataFrame df2 which had the label `y` by using the `loc` method.

In [10]:
```
# get second dataframe
result.loc['y']
```

Out[10]:

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

- When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following three ways:
  - Take the union of them all, `join='outer'` . This is the default option as it results in zero information loss.
  - Take the intersection, `join='inner'` .
  - Use a specific index, as passed to the `join_axes` argument.
- Here is an example of each of these methods. First, the default `join='outer'` behavior:

In [11]:
```
df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
                    'D': ['D2', 'D3', 'D6', 'D7'],
                    'F': ['F2', 'F3', 'F6', 'F7']},
                   index=[2, 3, 6, 7])


result = pd.concat([df1, df4], axis=1, sort=False)
result
```

Out[11]:

	A	B	C	D	B	D	F
0	A0	B0	C0	D0	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3
6	NaN	NaN	NaN	NaN	B6	D6	F6
7	NaN	NaN	NaN	NaN	B7	D7	F7

- Here is the same thing with `join='inner'` :

In [12]:
```
result = pd.concat([df1, df4], axis=1, join='inner')
result
```

Out[12]:

	A	B	C	D	B	D	F
2	A2	B2	C2	D2	B2	D2	F2
3	A3	B3	C3	D3	B3	D3	F3

b. merge() for combining dataframes using SQL like joins

- Another ubiquitous operation related to DataFrames is the merging operation.
- Two DataFrames might hold different kinds of information about the same entity and linked by some common feature/column.
- We can use **merge()** to combine such dataframes in pandas.

Creating dummy data

In [13]:
```python
# create dummy data
df_a = pd.DataFrame({
        'subject_id': ['1', '2', '3', '4', '5'],
        'first_name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
        'last_name': ['Anderson', 'Ackerman', 'Ali', 'Aoni', 'Atiches']})

df_b = pd.DataFrame({
        'subject_id': ['4', '5', '6', '7', '8'],
        'first_name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
        'last_name': ['Bonder', 'Black', 'Balwner', 'Brice', 'Btisan']})

df_c = pd.DataFrame({
        'subject_id': ['1', '2', '3', '4', '5', '7', '8', '9', '10', '11'],
        'test_id': [51, 15, 15, 61, 16, 14, 15, 1, 61, 16]})
```

Now these are our dataframes:

- Let's start with a basic join, we want to combine `df_a` with `df_c` based on the `subject_id` column.

In [14]:
```python
pd.merge(df_a, df_c, on='subject_id')
```

Out[14]:

	subject_id	first_name	last_name	test_id
0	1	Alex	Anderson	51
1	2	Amy	Ackerman	15
2	3	Allen	Ali	15
3	4	Alice	Aoni	61
4	5	Ayoung	Atiches	16

- Now that we have done a basic join, let's get into **some commmon SQL joins.**

Merge with outer join

- “Full outer join produces the set of all records in Table A and Table B, with matching records from both sides where available. If there is no match, the missing side will contain null.”

In [15]:
```
1 pd.merge(df_a, df_b, on='subject_id', how='outer')
```

Out[15]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	1	Alex	Anderson	NaN	NaN
1	2	Amy	Ackerman	NaN	NaN
2	3	Allen	Ali	NaN	NaN
3	4	Alice	Aoni	Billy	Bonder
4	5	Ayoung	Atiches	Brian	Black
5	6	NaN	NaN	Bran	Balwner
6	7	NaN	NaN	Bryce	Brice
7	8	NaN	NaN	Betty	Btisan

Merge with inner join

- “Inner join produces only the set of records that match in both Table A and Table B.”

In [16]:
```
1 pd.merge(df_a, df_b, on='subject_id', how='inner')
```

Out[16]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	4	Alice	Aoni	Billy	Bonder
1	5	Ayoung	Atiches	Brian	Black

Merge with right join

- “Right outer join produces a complete set of records from Table B, with the matching records (where available) in Table A. If there is no match, the left side will contain null.”

In [17]:
```
pd.merge(df_a, df_b, on='subject_id', how='right')
```

Out[17]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	4	Alice	Aoni	Billy	Bonder
1	5	Ayoung	Atiches	Brian	Black
2	6	NaN	NaN	Bran	Balwner
3	7	NaN	NaN	Bryce	Brice
4	8	NaN	NaN	Betty	Btisan

Merge with left join

- “Left outer join produces a complete set of records from Table A, with the matching records (where available) in Table B. If there is no match, the right side will contain null.”

In [18]:
```
pd.merge(df_a, df_b, on='subject_id', how='left')
```

Out[18]:

	subject_id	first_name_x	last_name_x	first_name_y	last_name_y
0	1	Alex	Anderson	NaN	NaN
1	2	Amy	Ackerman	NaN	NaN
2	3	Allen	Ali	NaN	NaN
3	4	Alice	Aoni	Billy	Bonder
4	5	Ayoung	Atiches	Brian	Black

Merge OR Concat : Which to use when?

1. After learning both of the functions in detail, chances are that you might be confused which to use when.
2. One major difference is that `merge()` is used to combine dataframes on the basis of values of **common columns**. While `concat()` is used to **append dataframes** one below the other (or sideways, depending on whether the axis option is set to 0 or 1).
3. Exact usage depends upon the kind of data you have and analysis you want to perform.

3. Apply Function

- Apply function can be used to perform pre-processing/ data -manupulation on both rows and columns.
- It is faster method than simple using a for loop over dataframe.
- Almost everytime i need to itrate over a dataframe or it's rows/columns. I will think of using the `apply`
- It is used in feature engineering code

In [19]:

```python
# accessing row wise

data_BM.apply(lambda x:x)
```

Out[19]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	OUT049	1999	Medium
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009	Medium
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	OUT049	1999	Medium
4	NCD19	8.930	Low Fat	0.000000	Household	53.8614	OUT013	1987	High
5	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	OUT018	2009	Medium
...	...	...	...	...	...	...	...	...	...
8517	FDF53	20.750	reg	0.083607	Frozen Foods	178.8318	OUT046	1997	Small
8518	FDF22	6.865	Low Fat	0.056783	Snack Foods	214.5218	OUT013	1987	High
8520	NCJ29	10.600	Low Fat	0.035186	Health and Hygiene	85.1224	OUT035	2004	Small
8521	FDN46	7.210	Regular	0.145221	Snack Foods	103.1332	OUT018	2009	Medium
8522	DRG01	14.800	Low Fat	0.044878	Soft Drinks	75.4670	OUT046	1997	Small

4650 rows × 12 columns

In [20]:

```
# access first row

data_BM.apply(lambda x:x[0])
```

Out[20]:

```
Item_Identifier                             FDA15
Item_Weight                                   9.3
Item_Fat_Content                          Low Fat
Item_Visibility                         0.0160473
Item_Type                                   Dairy
Item_MRP                                  249.809
Outlet_Identifier                          OUT049
Outlet_Establishment_Year                    1999
Outlet_Size                                Medium
Outlet_Location_Type                       Tier 1
Outlet_Type                     Supermarket Type1
Item_Outlet_Sales                         3735.14
dtype: object
```

In [21]:

```
# accessing first column by index

data_BM.apply(lambda x:x[0],axis=1)
```

Out[21]:

```
0       FDA15
1       DRC01
2       FDN15
4       NCD19
5       FDP36
        ...
8517    FDF53
8518    FDF22
8520    NCJ29
8521    FDN46
8522    DRG01
Length: 4650, dtype: object
```

In [22]:
```
# accessing by column name

data_BM.apply(lambda x:x["Item_Weight"],axis=1)
```

Out[22]:
```
0        9.300
1        5.920
2       17.500
4        8.930
5       10.395
         ...
8517    20.750
8518     6.865
8520    10.600
8521     7.210
8522    14.800
Length: 4650, dtype: float64
```

- we can also apply to implement a **condtion** individually on every row and column of our dataframe.
- SUppose i want to clio Item_MRP to 200 and not consider any value greater than that.

In [23]:
```
# before clipping

data_BM["Item_MRP"][:5]
```

Out[23]:
```
0    249.8092
1     48.2692
2    141.6180
4     53.8614
5     51.4008
Name: Item_MRP, dtype: float64
```

In [24]:
```
# clip price if it is greater than 200

def clip_price(price):
    if price > 200:
        price = 200
    return price
```

In [25]:
```
# after clipping

data_BM["Item_MRP"].apply(lambda x:clip_price(x))[:5]
```

Out[25]:
```
0    200.0000
1     48.2692
2    141.6180
4     53.8614
5     51.4008
Name: Item_MRP, dtype: float64
```

- Suppose i want to label encode Outlet_Location_Type as 0 ,1,2 for Tier 1,Tier2,Tier 3 city. so my logic would be :

In [26]:
```
# label encode city type

def label_encode(city):
    if city== "Tier 1":
        label = 0
    elif city== "Tier 2":
        label = 1
    else:
        label = 2
    return label

```

- Now i will use `apply` to operate `label_encode` logic on evry row of the Outlet_Location_Type column

In [27]:
```
# before label encoding

data_BM["Outlet_Location_Type"][:5]
```

Out[27]:
```
0    Tier 1
1    Tier 3
2    Tier 1
4    Tier 3
5    Tier 3
Name: Outlet_Location_Type, dtype: object
```

In [28]:
```
# after label encoding

data_BM["Outlet_Location_Type"].apply(lambda x : label_encode(x))[:5]
```

Out[28]:
```
0    0
1    2
2    0
4    2
5    2
Name: Outlet_Location_Type, dtype: int64
```

Types of Aggregations in Pandas

An essential piece of analysis of large data is efficient summarization: computing aggregations like sum(), mean(), median(), min(), and max(), in which a single number gives insight into the nature of a potentially large dataset.

explore aggregations in Pandas namely the following functions:

1. Crosstab
2. Groupby
3. Pivot Table

In [29]:

```python
import pandas as pd
import numpy as np

data = pd.read_csv("bigmart_data.csv")

# droping the null values

data = data.dropna(how="any")

# reset index after dropping

data = data.reset_index(drop= True)

#view data

data.head()
```

Out[29]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_Type	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Out
0	FDA15	9.300	Low Fat	0.016047	Dairy	249.8092	OUT049	1999	Medium	
1	DRC01	5.920	Regular	0.019278	Soft Drinks	48.2692	OUT018	2009	Medium	
2	FDN15	17.500	Low Fat	0.016760	Meat	141.6180	OUT049	1999	Medium	
3	NCD19	8.930	Low Fat	0.000000	Household	53.8614	OUT013	1987	High	
4	FDP36	10.395	Regular	0.000000	Baking Goods	51.4008	OUT018	2009	Medium	

1. Aggregating data

- after looking at the data , there are few questions poped-up . I will answer it using 1. groupby,crosstab, pivotable.

a. What is the mean price for each item type? :groupby

-Groupby is basically taking data and grouping it into bucket.

In [30]:

```python
# group price based on item type

group_price  =  data.groupby("Item_Type")

# display first few rows

group_price.first()
```

Out[30]:

	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	O
Item_Type									
Baking Goods	FDP36	10.395	Regular	0.000000	51.4008	OUT018	2009	Medium	
Breads	FDW11	12.600	Low Fat	0.048981	61.9194	OUT018	2009	Medium	
Breakfast	FDP49	9.000	Regular	0.069089	56.3614	OUT046	1997	Small	
Canned	FDC02	21.350	Low Fat	0.069103	259.9278	OUT018	2009	Medium	
Dairy	FDA15	9.300	Low Fat	0.016047	249.8092	OUT049	1999	Medium	
Frozen Foods	FDR28	13.850	Regular	0.025896	165.0210	OUT046	1997	Small	
Fruits and Vegetables	FDY07	11.800	Low Fat	0.000000	45.5402	OUT049	1999	Medium	
Hard Drinks	DRJ59	11.650	low fat	0.019356	39.1164	OUT013	1987	High	
Health and Hygiene	NCB42	11.800	Low Fat	0.008596	115.3492	OUT018	2009	Medium	
Household	NCD19	8.930	Low Fat	0.000000	53.8614	OUT013	1987	High	
Meat	FDN15	17.500	Low Fat	0.016760	141.6180	OUT049	1999	Medium	
Others	NCM43	14.500	Low Fat	0.019472	164.8210	OUT035	2004	Small	

Item_Type	Item_Identifier	Item_Weight	Item_Fat_Content	Item_Visibility	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	O
Seafood	FDH21	10.395	Low Fat	0.031274	160.0604	OUT049	1999	Medium	
Snack Foods	FDO10	13.650	Regular	0.012741	57.6588	OUT013	1987	High	
Soft Drinks	DRC01	5.920	Regular	0.019278	48.2692	OUT018	2009	Medium	
Starchy Foods	FDB11	16.000	Low Fat	0.060837	226.8404	OUT035	2004	Small	

- it has group the rows by it's item type by showing as index
- next step would be to calculate the mean of Item_MRP

In [31]:
```
# mean price by its item

group_price.Item_MRP.mean()
```

Out[31]:
```
Item_Type
Baking Goods             125.795653
Breads                   141.300639
Breakfast                134.090683
Canned                   138.551179
Dairy                    149.481471
Frozen Foods             140.095830
Fruits and Vegetables    145.418257
Hard Drinks              140.102908
Health and Hygiene       131.437324
Household                149.884244
Meat                     140.279344
Others                   137.640870
Seafood                  146.595782
Snack Foods              147.569955
Soft Drinks              130.910182
Starchy Foods            151.256747
Name: Item_MRP, dtype: float64
```

groupby using multiple columns

In [32]:

```
multi_group = data[:10].groupby(["Item_Type","Item_Fat_Content"])
multi_group.first()
```

Out[32]:

Item_Type	Item_Fat_Content	Item_Identifier	Item_Weight	Item_Visibility	Item_MRP	Outlet_Identifier	Outlet_Establishment_Year	Outlet_Size	Outlet_
Baking Goods	**Regular**	FDP36	10.395	0.000000	51.4008	OUT018	2009	Medium	
Dairy	**Low Fat**	FDA15	9.300	0.016047	249.8092	OUT049	1999	Medium	
	Regular	FDA03	18.500	0.045464	144.1102	OUT046	1997	Small	
Fruits and Vegetables	**Low Fat**	FDY07	11.800	0.000000	45.5402	OUT049	1999	Medium	
	Regular	FDX32	15.100	0.100014	145.4786	OUT049	1999	Medium	
Household	**Low Fat**	NCD19	8.930	0.000000	53.8614	OUT013	1987	High	
Meat	**Low Fat**	FDN15	17.500	0.016760	141.6180	OUT049	1999	Medium	
Snack Foods	**Regular**	FDO10	13.650	0.012741	57.6588	OUT013	1987	High	
Soft Drinks	**Regular**	DRC01	5.920	0.019278	48.2692	OUT018	2009	Medium	

b. How are outlet sizes distributed based on the city type? : crosstab

- will tier 1 city have bigger outlet size?
- here, `crosstab` does is , it builts the cross tabulation table ,that shows the frequency with which certain group of data appear.
- For example , in this case, "Outlet_Location_Type" is expected to affect the "Outlet_size" significantly.

In [33]:
```python
# generate crosstab of outlet size and outlet_location type

pd.crosstab(data["Outlet_Size"],data["Outlet_Location_Type"],margins=True)
```

Out[33]:

Outlet_Location_Type Outlet_Size	Tier 1	Tier 2	Tier 3	All
High	0	0	932	932
Medium	930	0	928	1858
Small	930	930	0	1860
All	1860	930	1860	4650

- tier 3 is highest outlet_size , and the highest outlet size is not present in tier1 amd tier 2 cities.
- and 50% of medium size Outlet are present only on tier 1 and tier 3 cities

c. How are the sales changing per year? : pivottable

- pivot_table requires a data and an index parameter
- data is the Pandas dataframe you pass to the function
- index is the feature that allows you to group your data. The index feature will appear as an index in the resultant table

In [34]:
```
# crating pivottable

pd.pivot_table(data,index=["Outlet_Establishment_Year"],values="Item_Outlet_Sales")
```

Out[34]:

	Item_Outlet_Sales
Outlet_Establishment_Year	
1987	2298.995256
1997	2277.844267
1999	2348.354635
2004	2438.841866
2009	1995.498739

- the mean of the each year is shown
- Multiple colums using pivot table

In [35]:
```
pd.pivot_table(data,index=["Outlet_Establishment_Year","Outlet_Location_Type","Outlet_Size"],values="Item_Outlet_Sal
```

Out[35]:

			Item_Outlet_Sales
Outlet_Establishment_Year	**Outlet_Location_Type**	**Outlet_Size**	
1987	**Tier 3**	**High**	2298.995256
1997	**Tier 1**	**Small**	2277.844267
1999	**Tier 1**	**Medium**	2348.354635
2004	**Tier 2**	**Small**	2438.841866
2009	**Tier 3**	**Medium**	1995.498739

- This makes it easier to see that Tier 1 cities have good sales irrespective of year and outlet size.
- Tier 2 & Tier 3 cities dominate during the later year. This might mean both are performing beteer or we gave less data of later

In [36]:
```
# performing multiple agg. like mean, median,min,max etc in pivot using aggfunc parameter

pd.pivot_table(data,index=["Outlet_Establishment_Year","Outlet_Location_Type","Outlet_Size"],values="Item_Outlet_Sal
```

Out[36]:

			mean	median	min	max	
			Item_Outlet_Sales	Item_Outlet_Sales	Item_Outlet_Sales	Item_Outlet_Sales	Item_Outlet_
Outlet_Establishment_Year	Outlet_Location_Type	Outlet_Size					
1987	Tier 3	High	2298.995256	2050.6640	73.2380	10256.6490	1533.5
1997	Tier 1	Small	2277.844267	1945.8005	101.8674	9779.9362	1488.4
1999	Tier 1	Medium	2348.354635	1966.1074	111.8544	7646.0472	1513.2
2004	Tier 2	Small	2438.841866	2109.2544	113.8518	8479.6288	1538.5
2009	Tier 3	Medium	1995.498739	1655.1788	69.2432	6768.5228	1375.9

***Note : For detail study follow full documentation in Pandas.*