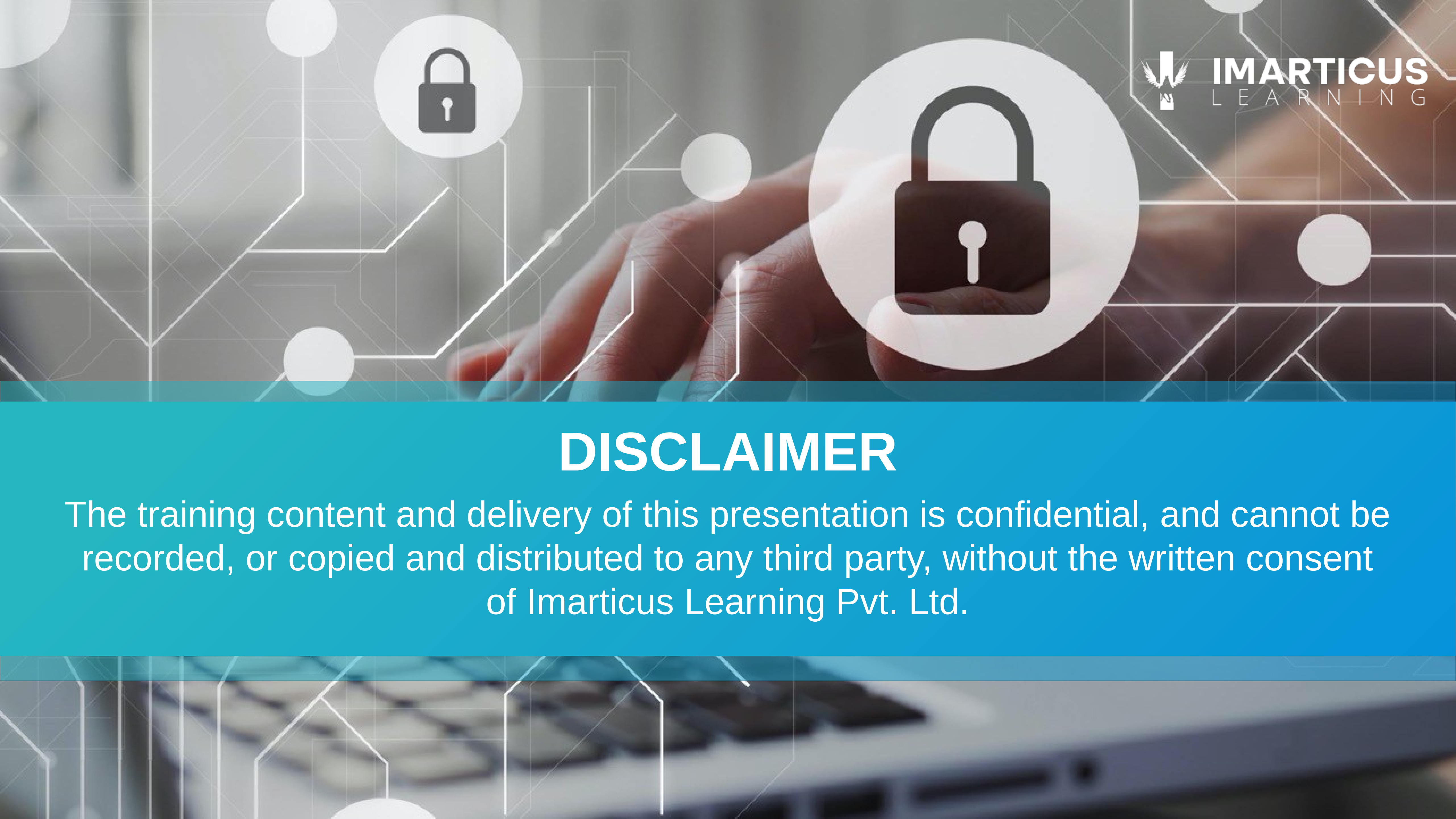
Python Programming

DataFrame Manipulation



IMARTICUS LEARNING

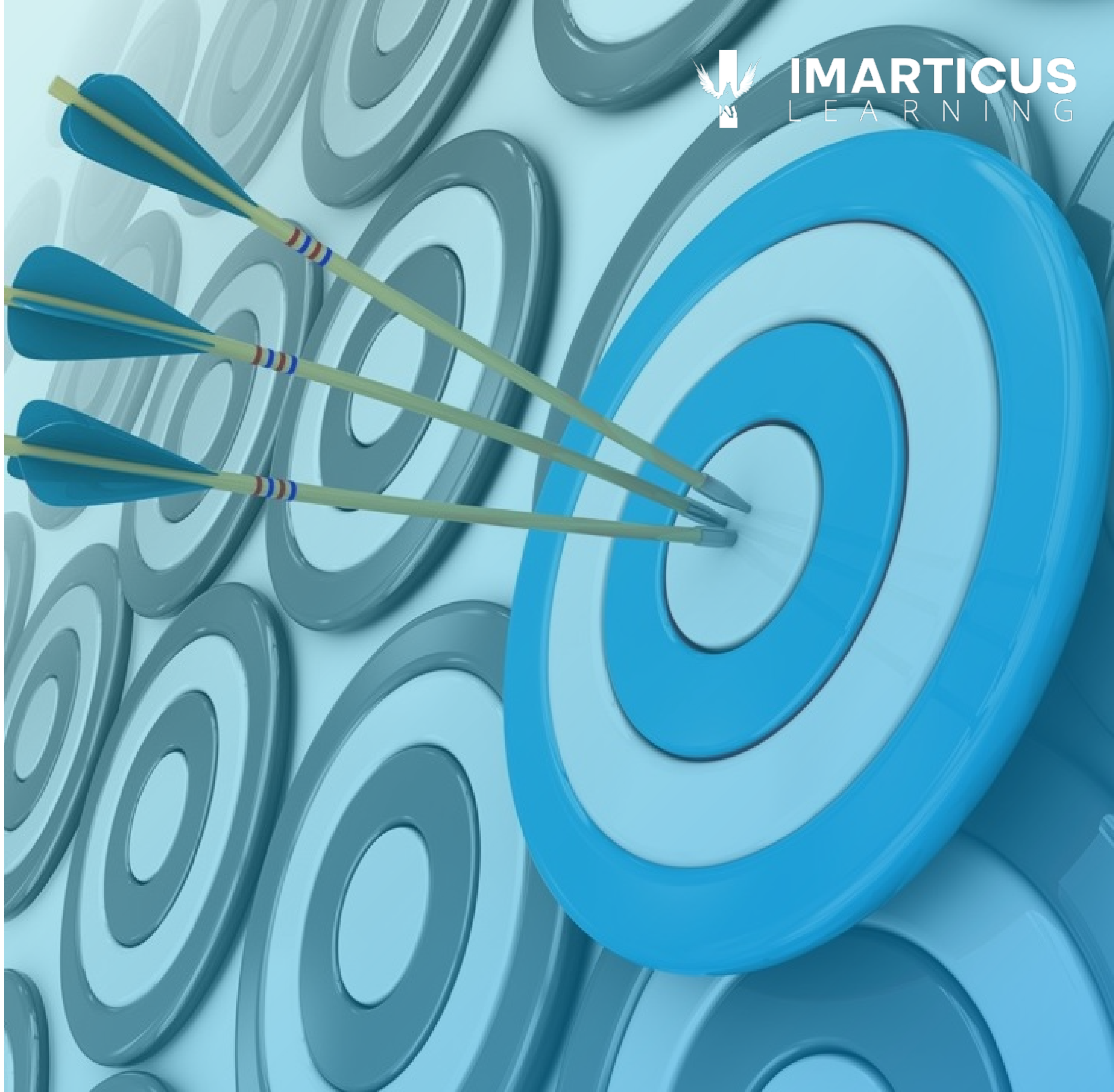
DISCLAIMER

The training content and delivery of this presentation is confidential, and cannot be recorded, or copied and distributed to any third party, without the written consent of Imarticus Learning Pvt. Ltd.

LEARNING OBJECTIVES

At the end of this session, you will learn:

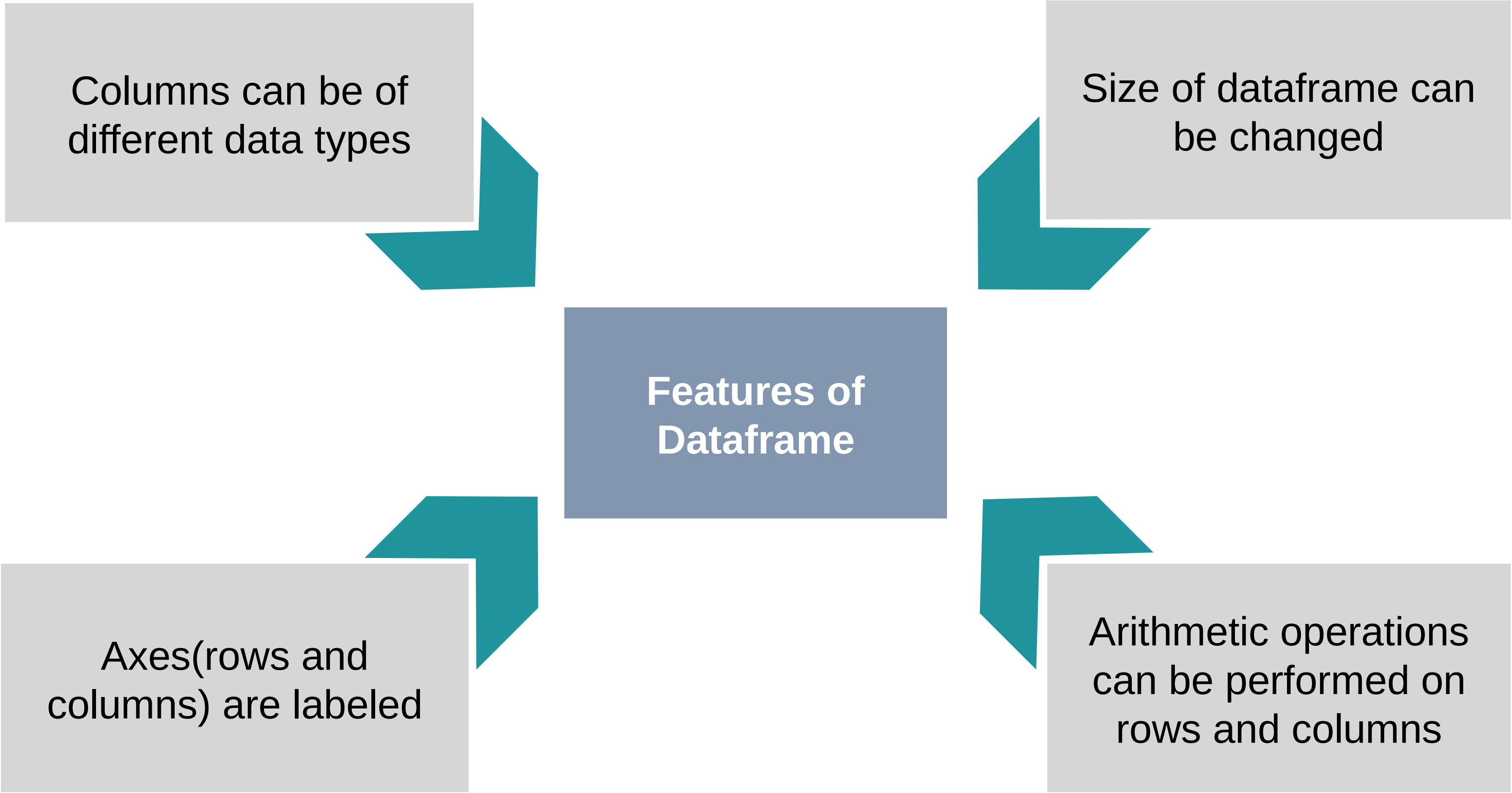
- Pandas DataFrame - Introduction
- DataFrame Creation
- Reading Data from Various Files
- Understanding Data
- Accessing DataFrame elements using Indexing
- Dataframe Sorting & Ranking
- Dataframe Concatenation
- Dataframe Joins & Merge
- Reshaping Dataframe
- Pivot Tables & Cross Tables
- Dataframe Operations
- Checking Duplicates
- Dropping Rows and Columns
- Replacing Values
- Grouping Dataframe
- Missing Value Analysis & Treatment



Pandas DataFrame - Introduction

INTRODUCTION TO DATAFRAMES

A DataFrame is a two dimensional data structure where the data is arranged in a tabular format in rows and columns



Features of Dataframe

Columns can be of different data types

Size of dataframe can be changed

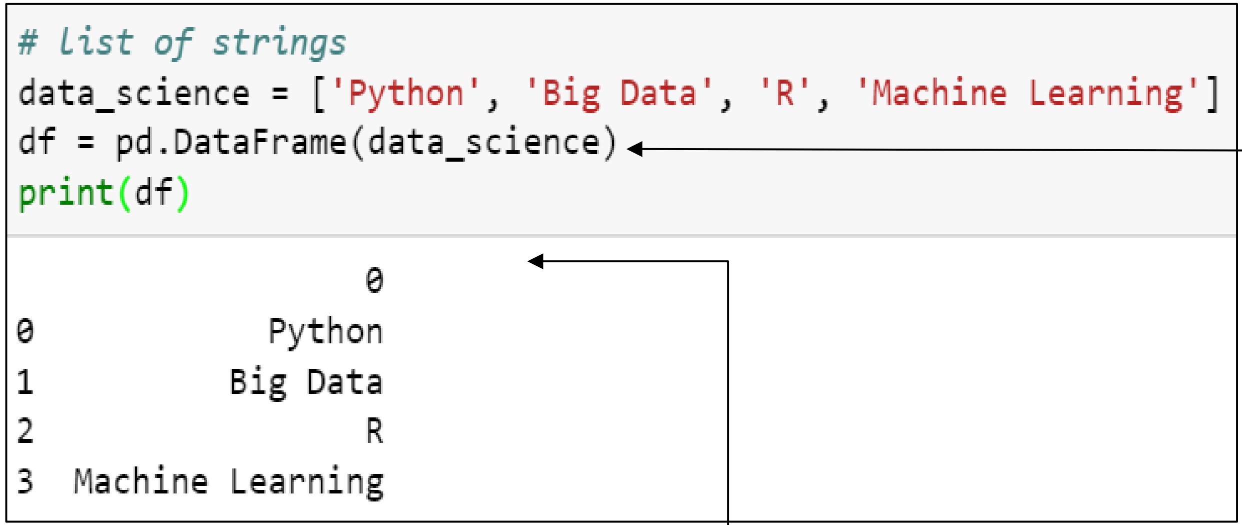
Axes(rows and columns) are labeled

Arithmetic operations can be performed on rows and columns

DataFrame Creation

CREATING A DATAFRAME

- Creating a dataframe from a list



```
# list of strings
data_science = ['Python', 'Big Data', 'R', 'Machine Learning']
df = pd.DataFrame(data_science)
print(df)
```

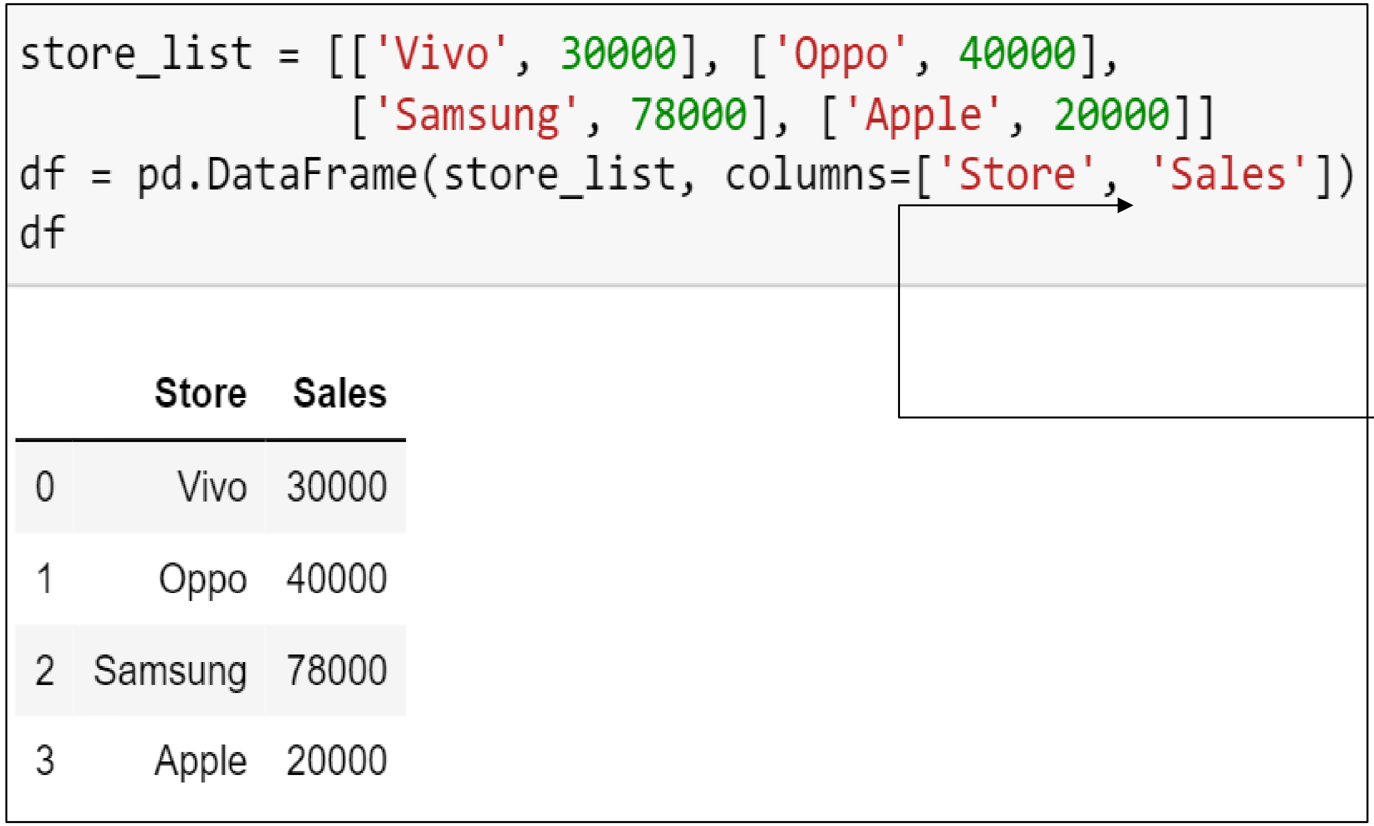
```
                  0
0            Python
1          Big Data
2                 R
3  Machine Learning
```

Pass a list as a column in 'df'

As no column name is passed, by default it returns '0' as column name

CREATING A DATAFRAME

- Creating a dataframe from a list of list



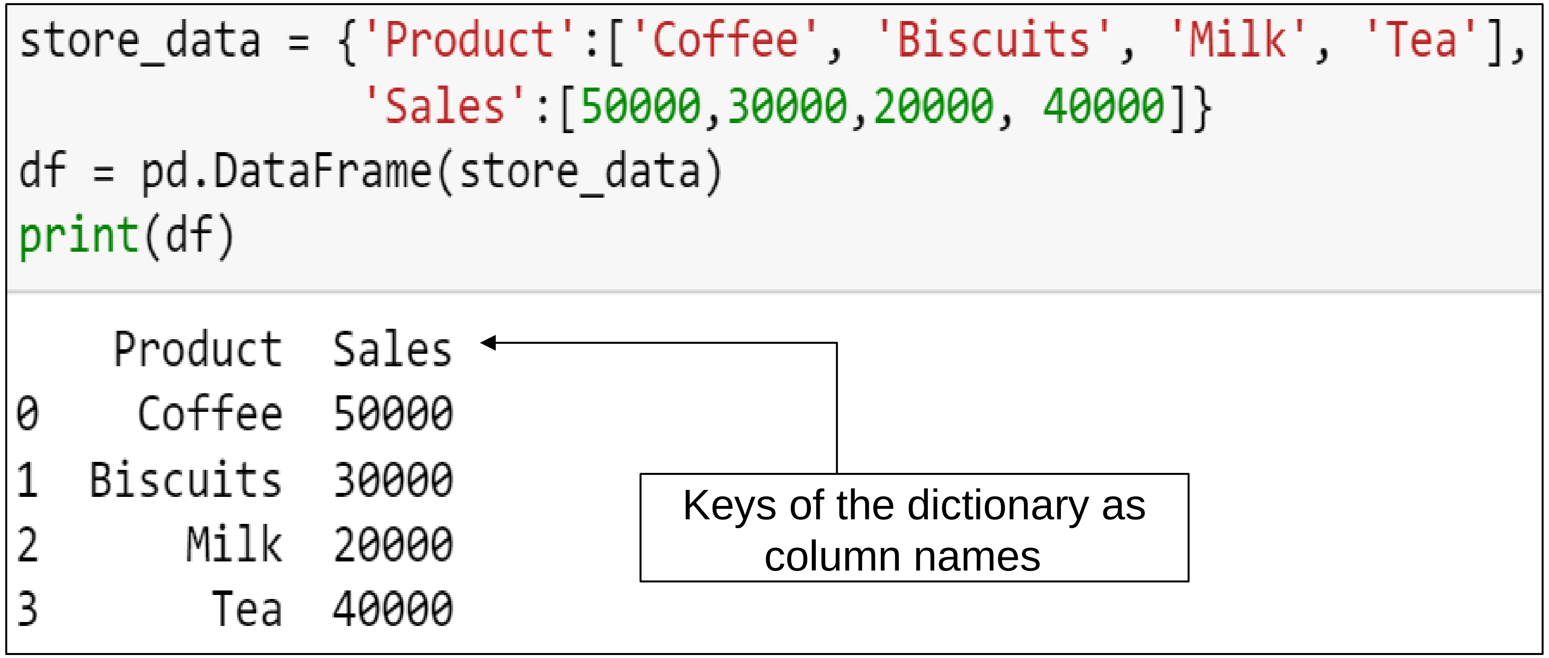
```python
store_list = [['Vivo', 30000], ['Oppo', 40000],
              ['Samsung', 78000], ['Apple', 20000]]
df = pd.DataFrame(store_list, columns=['Store', 'Sales'])
df
```

	Store	Sales
0	Vivo	30000
1	Oppo	40000
2	Samsung	78000
3	Apple	20000

Pass the list of column names

CREATING A DATAFRAME

- Creating a dataframe from a dictionary



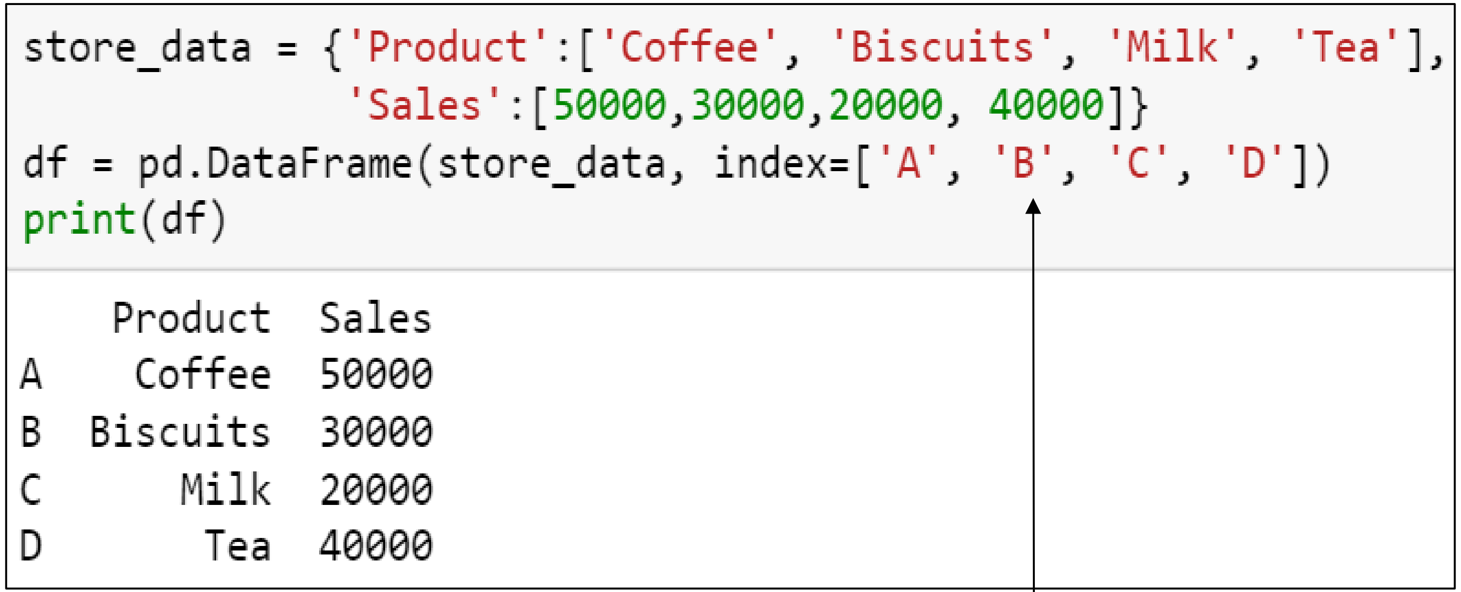
```
store_data = {'Product':['Coffee', 'Biscuits', 'Milk', 'Tea'],
              'Sales':[50000,30000,20000, 40000]}
df = pd.DataFrame(store_data)
print(df)
```

```
    Product  Sales
0    Coffee  50000
1  Biscuits  30000
2      Milk  20000
3       Tea  40000
```

Keys of the dictionary as column names

CREATING A DATAFRAME

- Creating a dataframe with index from a dictionary



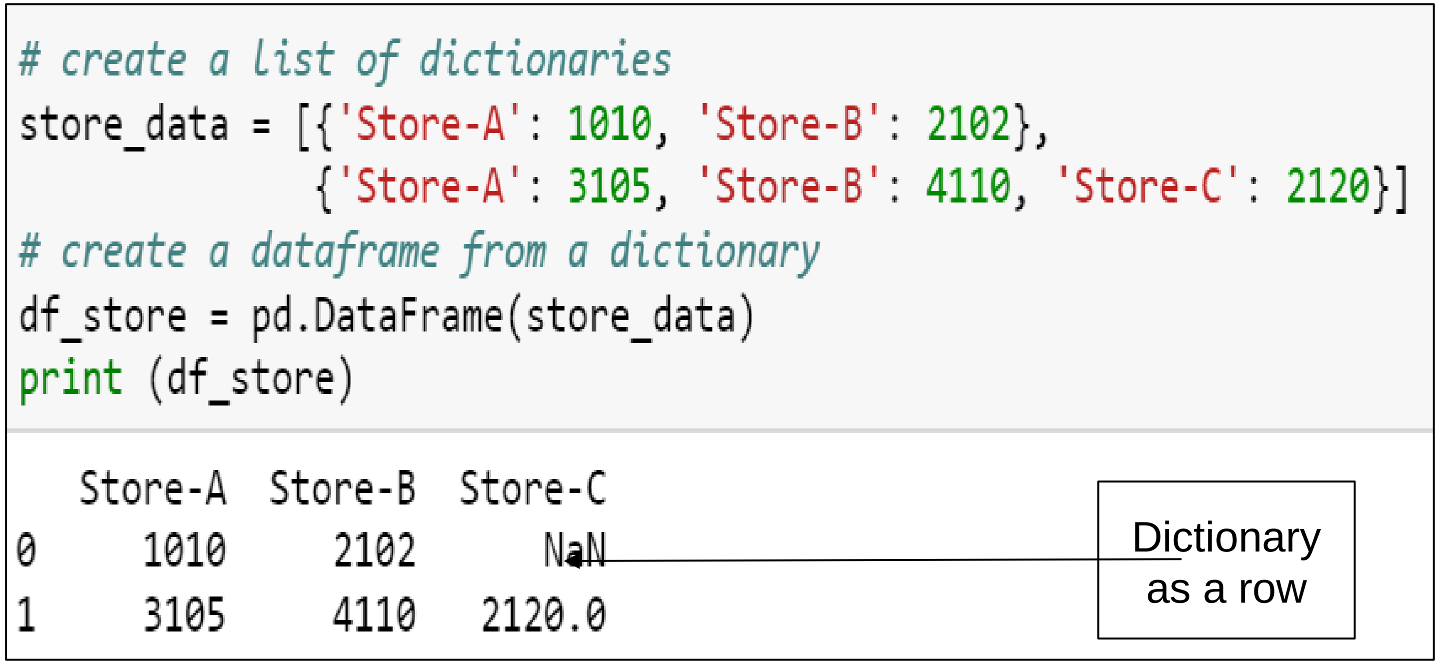
```
store_data = {'Product':['Coffee', 'Biscuits', 'Milk', 'Tea'],
              'Sales':[50000,30000,20000, 40000]}
df = pd.DataFrame(store_data, index=['A', 'B', 'C', 'D'])
print(df)
```

```
    Product  Sales
A    Coffee  50000
B  Biscuits  30000
C      Milk  20000
D       Tea  40000
```

Pass a list of index

CREATING A DATAFRAME

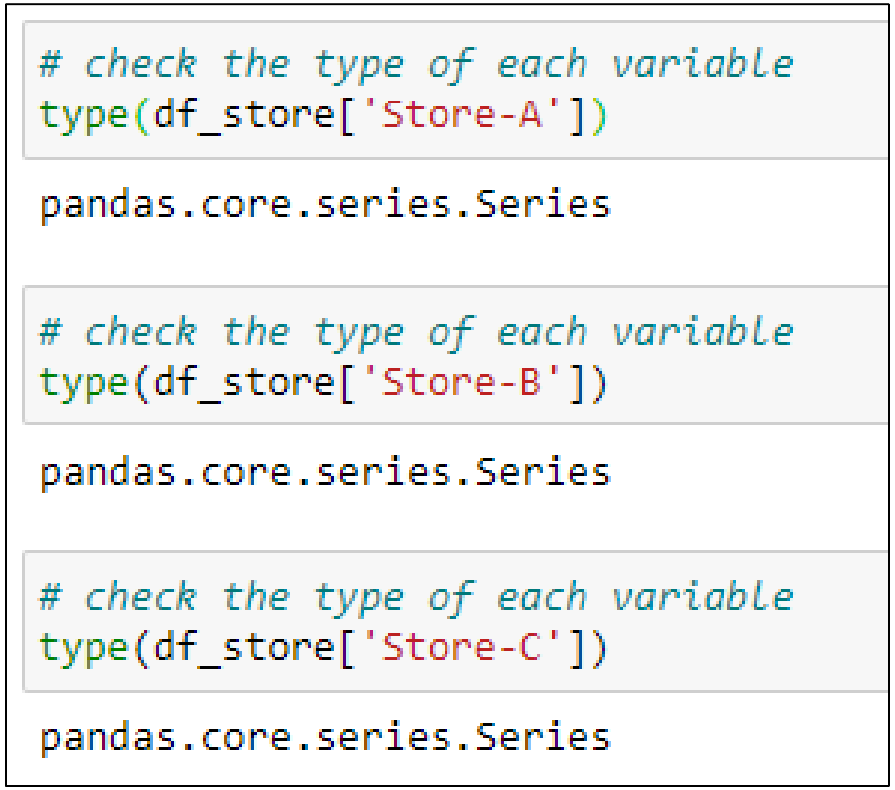
- Creating a dataframe with a list of dictionaries



```
# create a list of dictionaries
store_data = [{'Store-A': 1010, 'Store-B': 2102},
              {'Store-A': 3105, 'Store-B': 4110, 'Store-C': 2120}]
# create a dataframe from a dictionary
df_store = pd.DataFrame(store_data)
print (df_store)
```

```
   Store-A  Store-B  Store-C
0     1010     2102      NaN
1     3105     4110   2120.0
```

Dictionary as a row



```
# check the type of each variable
type(df_store['Store-A'])
```

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

```
# check the type of each variable
type(df_store['Store-B'])
```

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

```
# check the type of each variable
type(df_store['Store-C'])
```

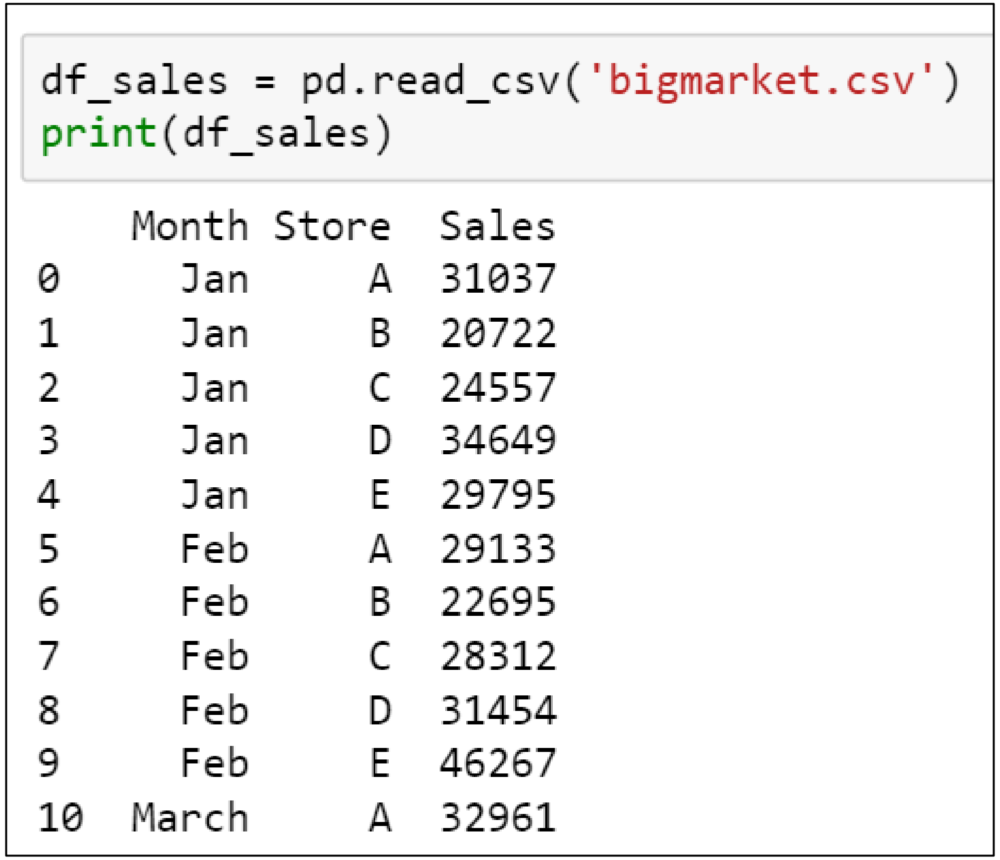
```
pandas.core.series.Series
```

Note that every column of the data frame is a pandas Series.

Reading Data from Various Files

READING DATA FROM CSV FILE

- Use the pandas read_csv() method to read the data from a csv file

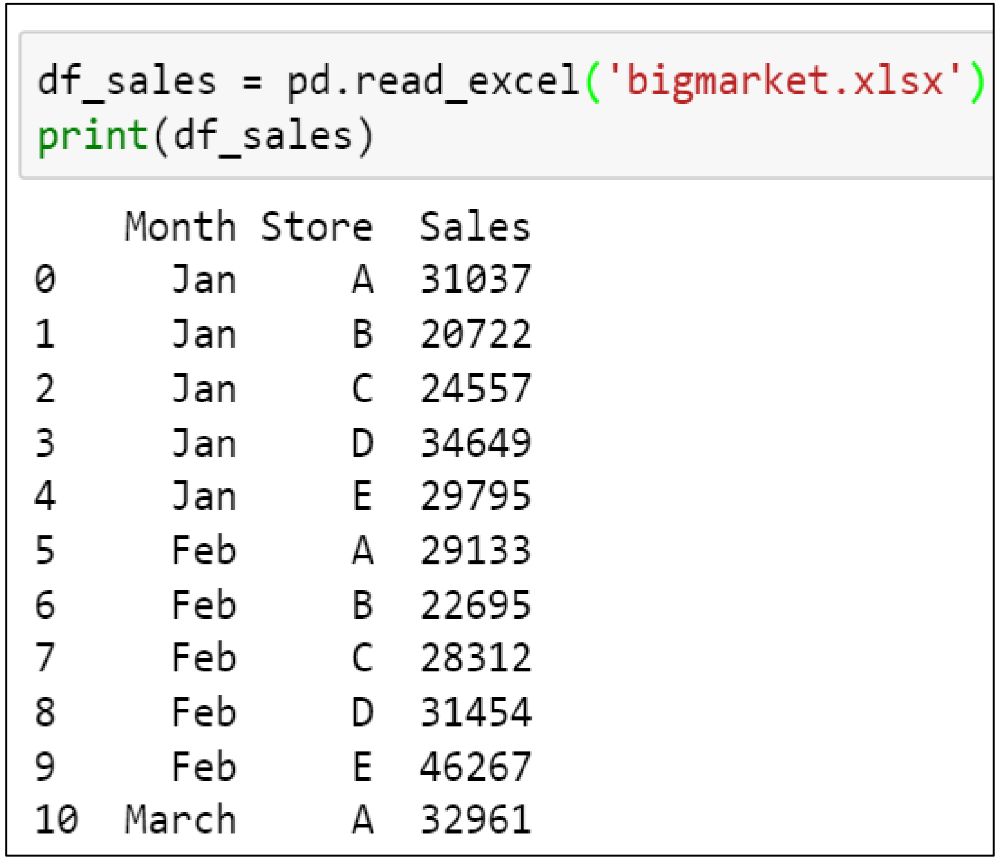


```
df_sales = pd.read_csv('bigmarket.csv')
print(df_sales)
```

```
    Month Store  Sales
0     Jan     A  31037
1     Jan     B  20722
2     Jan     C  24557
3     Jan     D  34649
4     Jan     E  29795
5     Feb     A  29133
6     Feb     B  22695
7     Feb     C  28312
8     Feb     D  31454
9     Feb     E  46267
10  March     A  32961
```

READING DATA FROM XLSL FILE

- Use the pandas read_excel() method to read the data from xlsx file

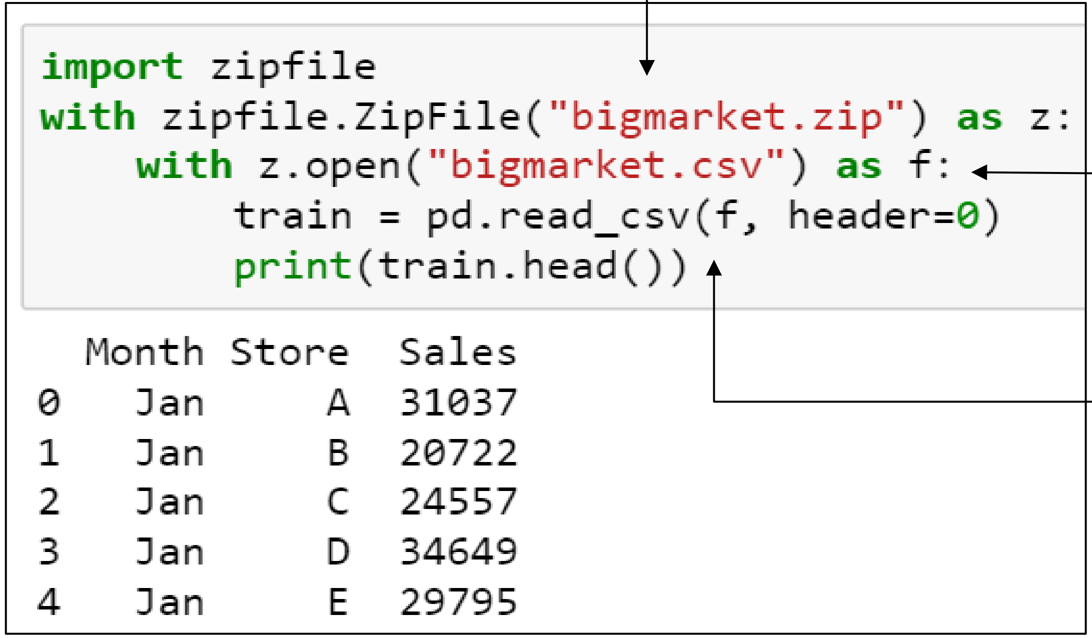


```
df_sales = pd.read_excel('bigmarket.xlsx')
print(df_sales)
```

```
    Month Store  Sales
0     Jan     A  31037
1     Jan     B  20722
2     Jan     C  24557
3     Jan     D  34649
4     Jan     E  29795
5     Feb     A  29133
6     Feb     B  22695
7     Feb     C  28312
8     Feb     D  31454
9     Feb     E  46267
10  March     A  32961
```

READING DATA FROM ZIP FILE

- If you have a csv file inside a zip file:



```
import zipfile
with zipfile.ZipFile("bigmarket.zip") as z:
    with z.open("bigmarket.csv") as f:
        train = pd.read_csv(f, header=0)
        print(train.head())
```

```
  Month Store  Sales
0   Jan     A  31037
1   Jan     B  20722
2   Jan     C  24557
3   Jan     D  34649
4   Jan     E  29795
```

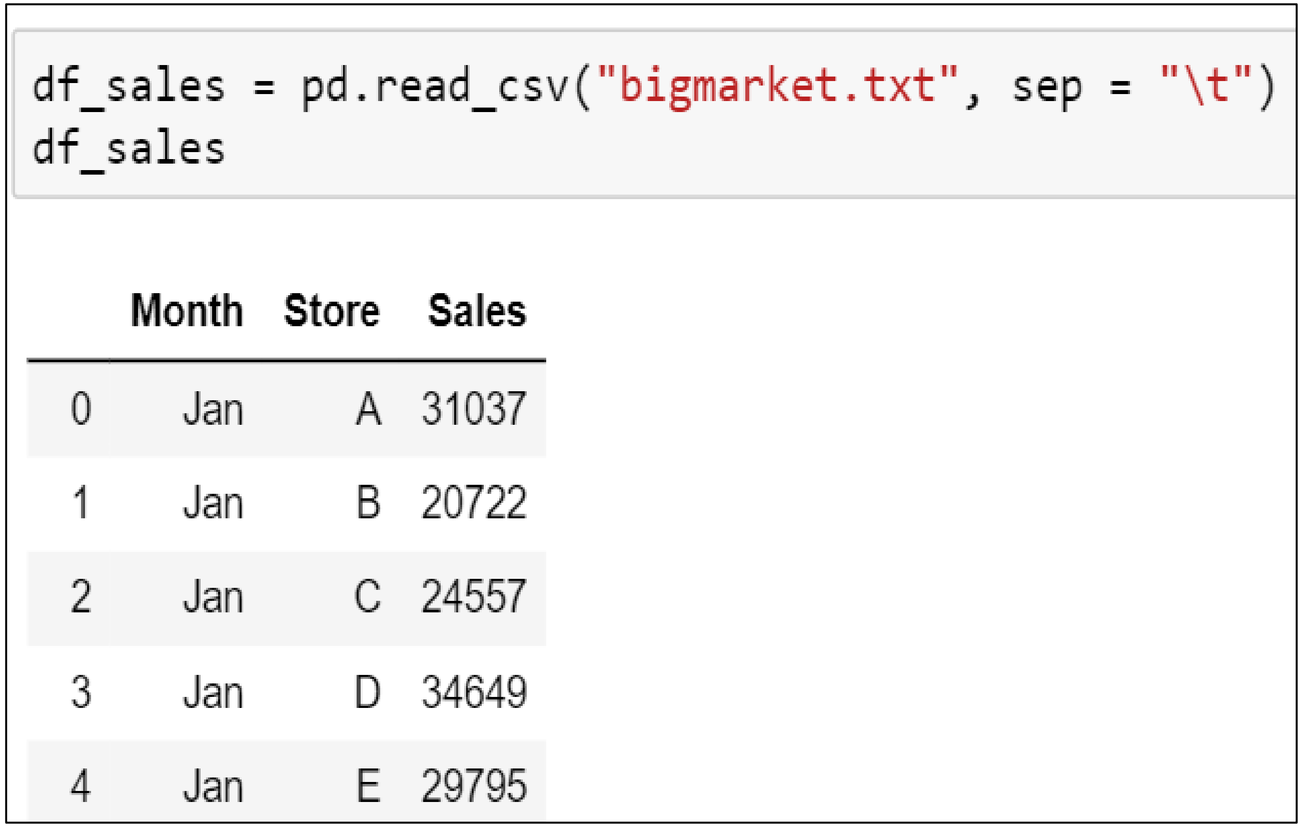
Read the zip file

Open csv file inside the zip file

Read the csv file

READING DATA FROM TEXT FILE

- You can use the pandas read_csv() method to read the data from text file

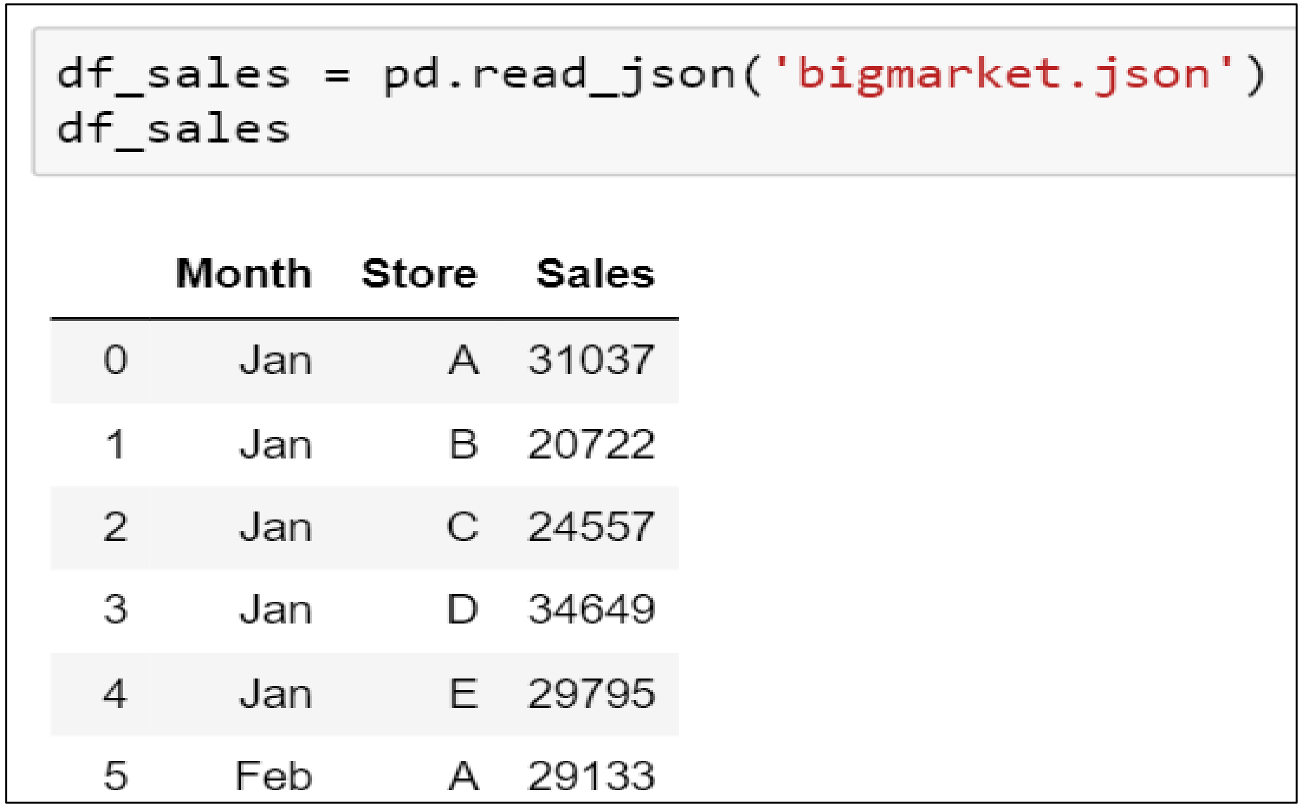


```
df_sales = pd.read_csv("bigmarket.txt", sep = "\t")
df_sales
```

	Month	Store	Sales
0	Jan	A	31037
1	Jan	B	20722
2	Jan	C	24557
3	Jan	D	34649
4	Jan	E	29795

READING DATA FROM JSON FILE

- Use the pandas read_json() method to read the data from json file

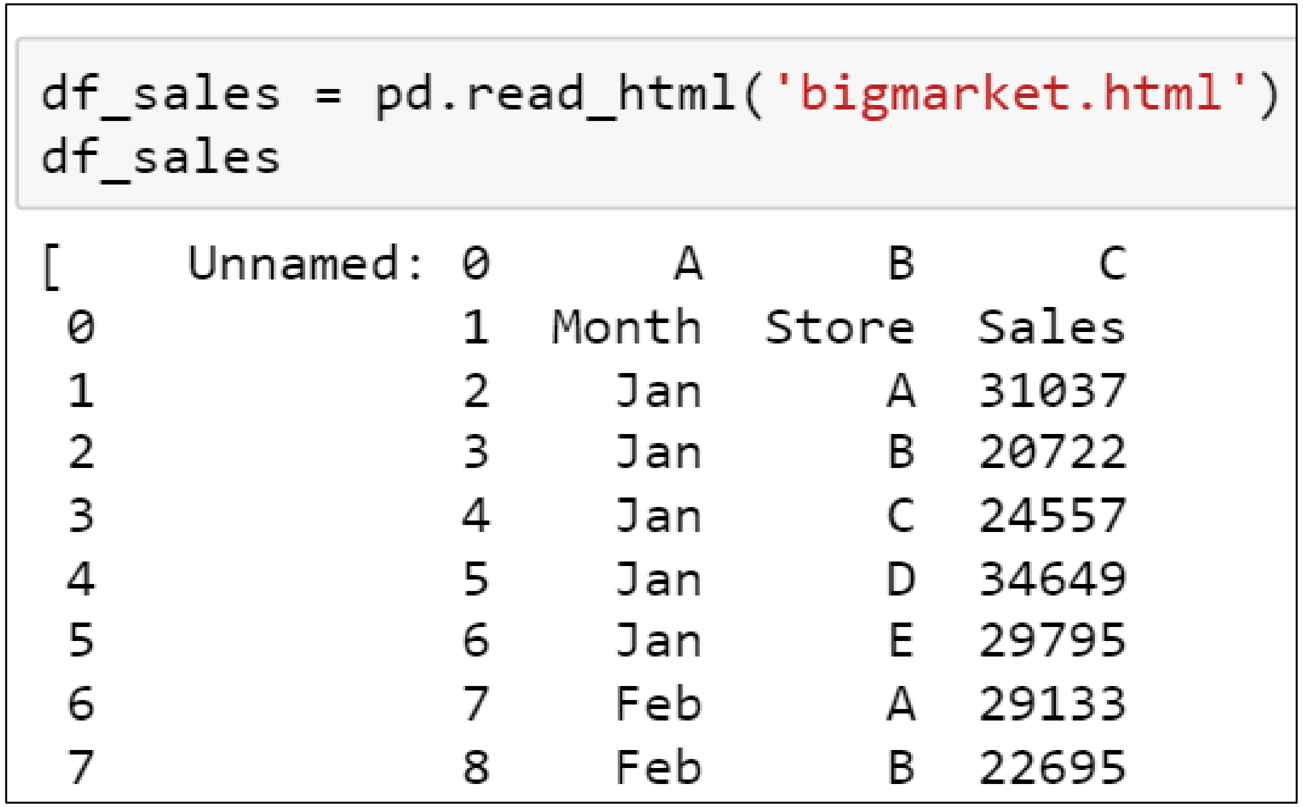


```
df_sales = pd.read_json('bigmarket.json')
df_sales
```

	Month	Store	Sales
0	Jan	A	31037
1	Jan	B	20722
2	Jan	C	24557
3	Jan	D	34649
4	Jan	E	29795
5	Feb	A	29133

READING DATA FROM HTML FILE

- Use the pandas read_html() method to read the data from html file



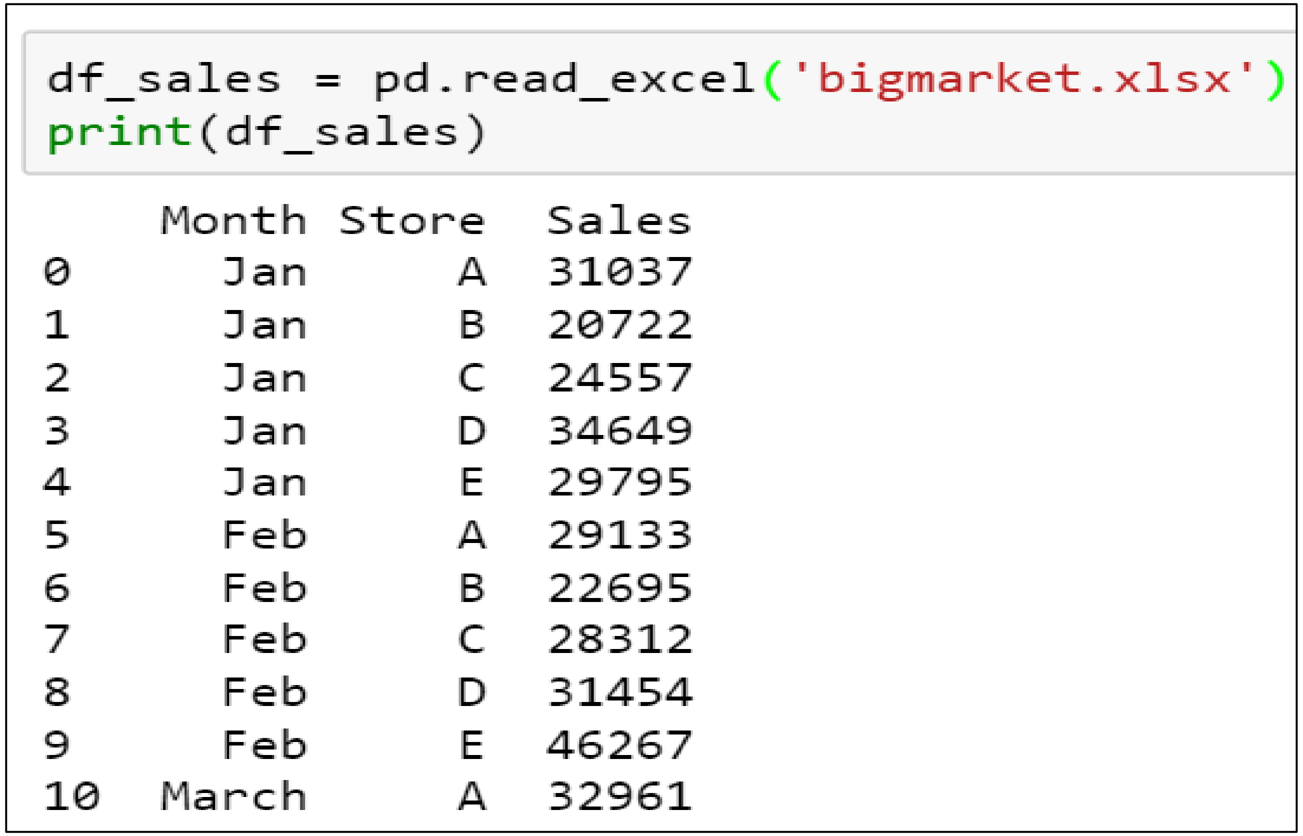
```
df_sales = pd.read_html('bigmarket.html')
df_sales
```

```
[    Unnamed: 0      A      B      C
 0            1  Month  Store  Sales
 1            2    Jan      A  31037
 2            3    Jan      B  20722
 3            4    Jan      C  24557
 4            5    Jan      D  34649
 5            6    Jan      E  29795
 6            7    Feb      A  29133
 7            8    Feb      B  22695
```

Understanding Data

READ THE DATA FROM XLSX FILE

- We use the following DataFrame for the study:

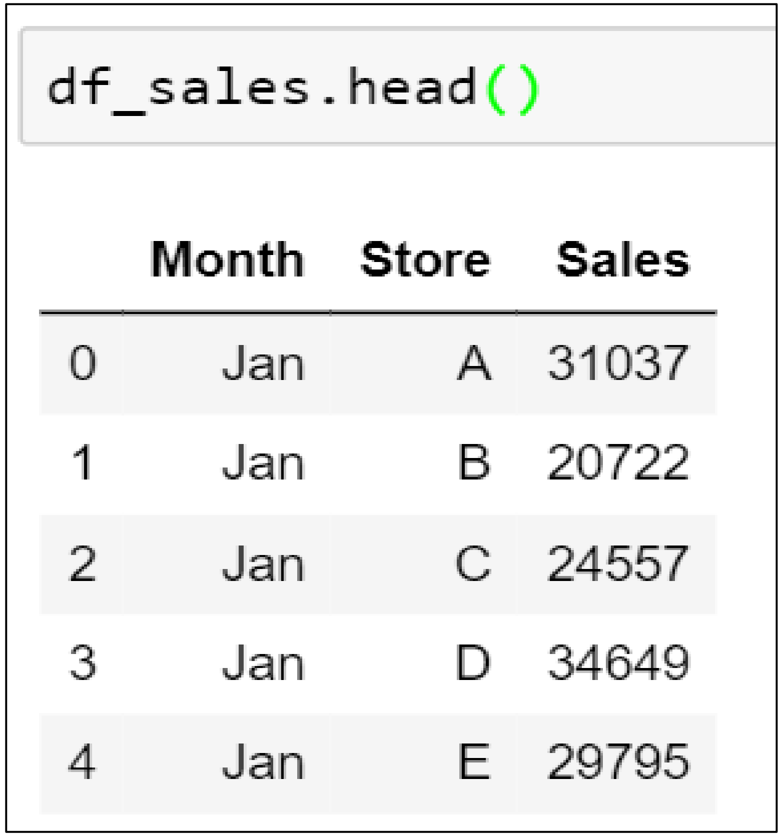


```python
df_sales = pd.read_excel('bigmarket.xlsx')
print(df_sales)
```

```
    Month Store  Sales
0     Jan     A  31037
1     Jan     B  20722
2     Jan     C  24557
3     Jan     D  34649
4     Jan     E  29795
5     Feb     A  29133
6     Feb     B  22695
7     Feb     C  28312
8     Feb     D  31454
9     Feb     E  46267
10  March     A  32961
```

DISPLAY THE TOP FIVE ROWS

- The pandas head() method displays the first five rows of the data

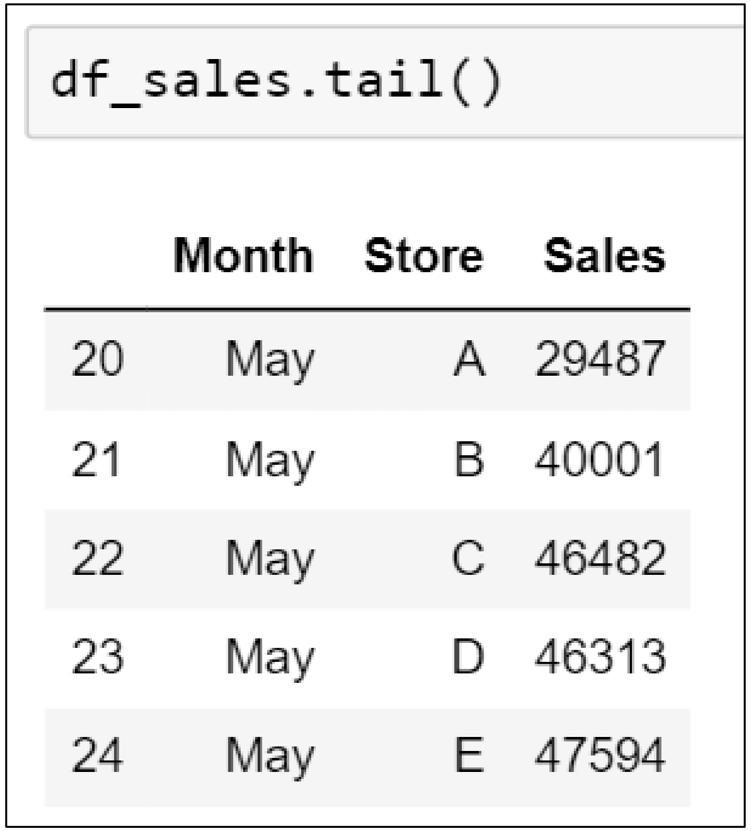


```
df_sales.head()
```

	Month	Store	Sales
0	Jan	A	31037
1	Jan	B	20722
2	Jan	C	24557
3	Jan	D	34649
4	Jan	E	29795

DISPLAY THE BOTTOM FIVE ROWS

- The pandas tail() method displays the last five rows of the data

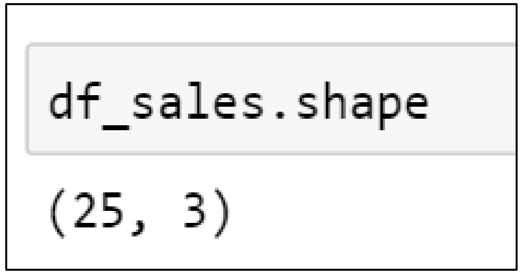


```
df_sales.tail()
```

	Month	Store	Sales
20	May	A	29487
21	May	B	40001
22	May	C	46482
23	May	D	46313
24	May	E	47594

UNDERSTANDING DATA

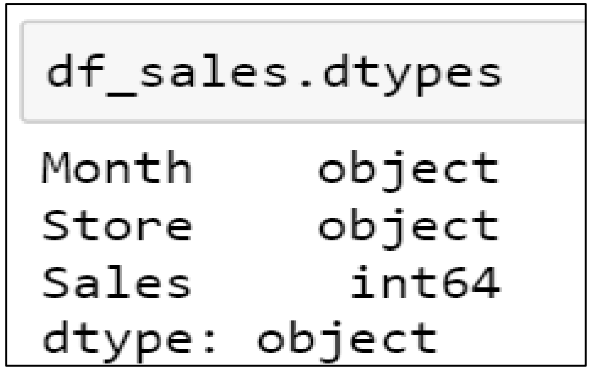
- Get the number of observations and number of columns of the data using the shape attribute



```
df_sales.shape
```

```
(25, 3)
```

- Get the data type of each variable in the data using the dtypes attribute

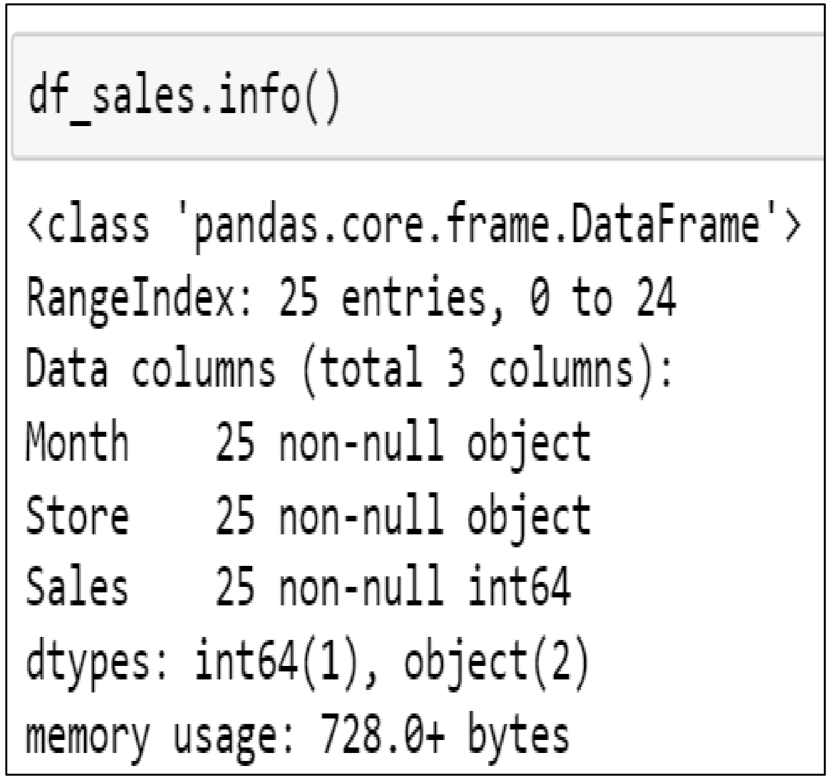


```
df_sales.dtypes
```

```
Month     object
Store     object
Sales      int64
dtype: object
```

UNDERSTANDING DATA

- The pandas info() method prints the information about the shape, data type and null values in the data
- Here, ‘df_sales’ has 3 variables with 25 non-null observations in each
- Of all the three variables, two are categorical (‘Month’ and ‘Store’) and one is numeric (Sales)



```
df_sales.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25 entries, 0 to 24
Data columns (total 3 columns):
Month     25 non-null object
Store     25 non-null object
Sales     25 non-null int64
dtypes: int64(1), object(2)
memory usage: 728.0+ bytes
```

Accessing DataFrame Elements using Indexing

ACCESSING DATAFRAME ELEMENTS USING INDEXING

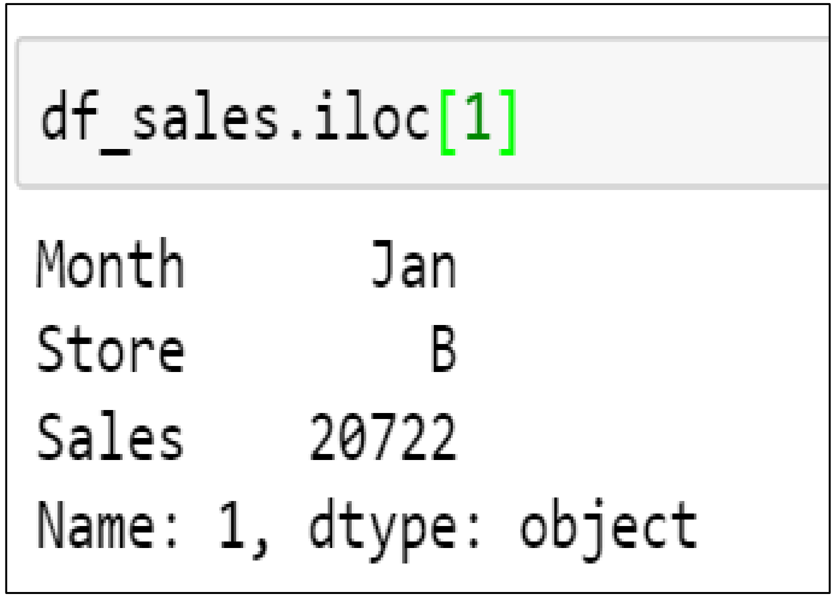
Indexing is often required in DataFrame to retrieve information

The .iloc[], the .loc[] or some conditions can be used to retrieve the elements

.iloc[]	Retrieves the rows and columns by position
.loc[]	Retrieves the elements by the column or row name

ACCESSING DATAFRAME ELEMENTS USING INDEXING

- Retrieve the 2nd row from ‘df_sales’ by using the .iloc[]

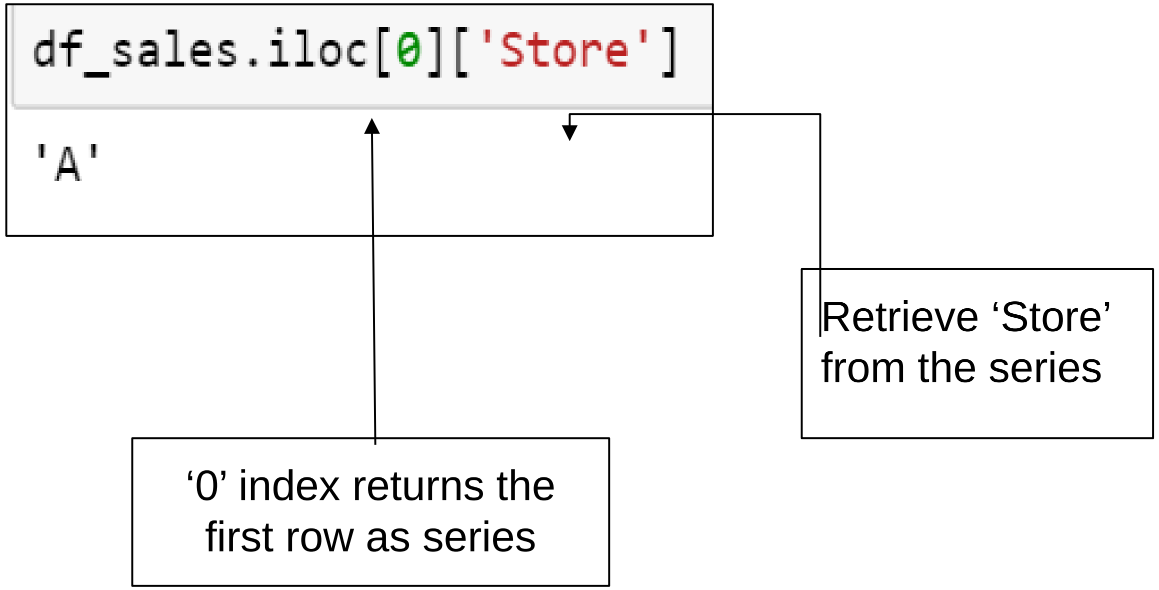


```
df_sales.iloc[1]
```

```
Month        Jan
Store          B
Sales      20722
Name: 1, dtype: object
```

ACCESSING DATAFRAME ELEMENTS USING INDEXING

- Retrieve the name of the first store from the ‘df_sales’ dataframe using .iloc[]



```
df_sales.iloc[0]['Store']
```

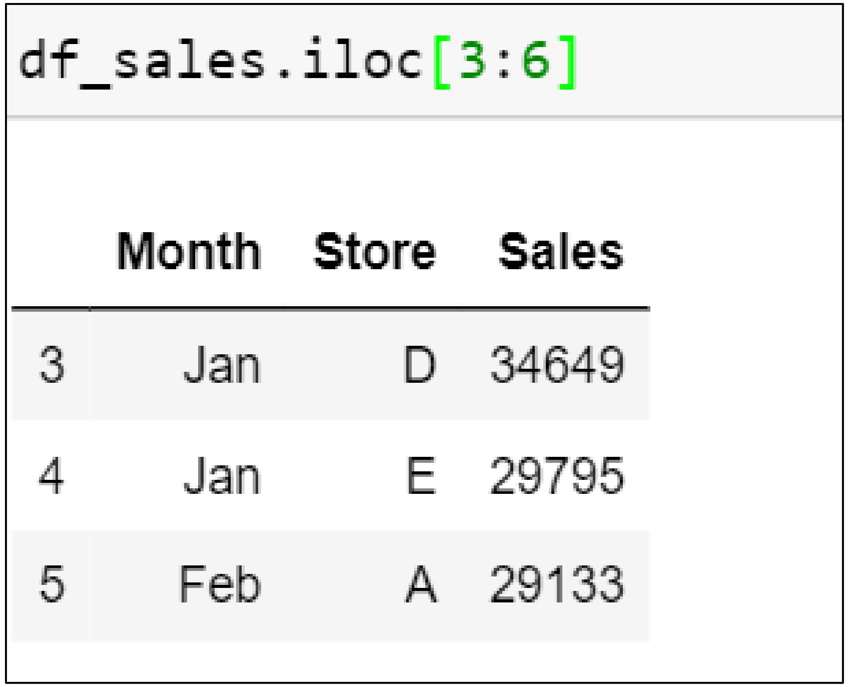
```
'A'
```

Retrieve ‘Store’ from the series

‘0’ index returns the first row as series

ACCESSING DATAFRAME ELEMENTS USING INDEXING

- Retrieve the 4th, 5th, and 6th row in the DataFrame using the .iloc[]

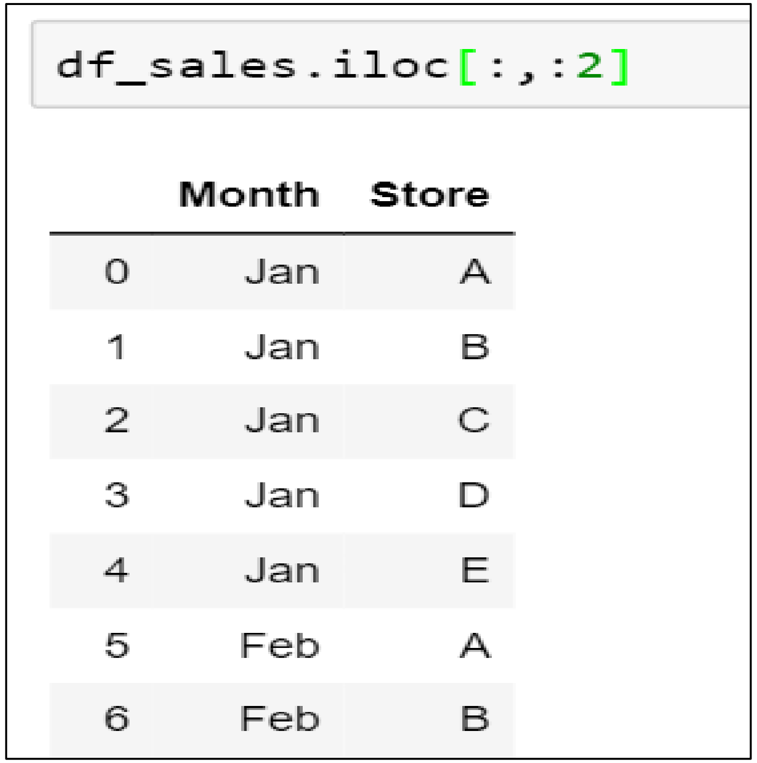


```
df_sales.iloc[3:6]
```

	Month	Store	Sales
3	Jan	D	34649
4	Jan	E	29795
5	Feb	A	29133

ACCESSING DATAFRAME ELEMENTS USING INDEXING

- Select first two columns by using the position of the columns

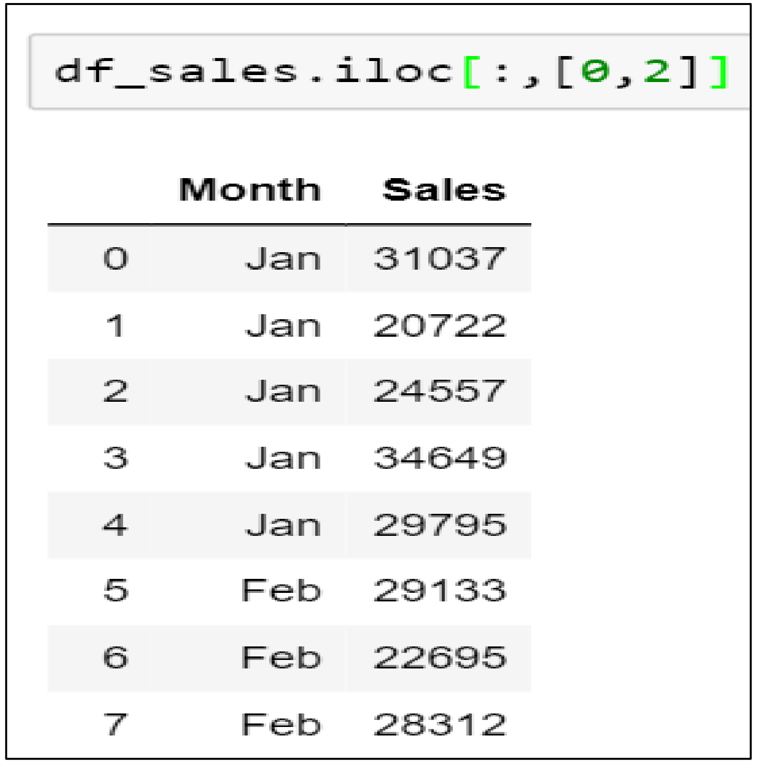


```
df_sales.iloc[:,:2]
```

```
   Month  Store
0    Jan      A
1    Jan      B
2    Jan      C
3    Jan      D
4    Jan      E
5    Feb      A
6    Feb      B
```

ACCESSING DATAFRAME ELEMENTS USING INDEXING

- Find the number of sales of corresponding store (check previous slide) for each month using the .iloc[]

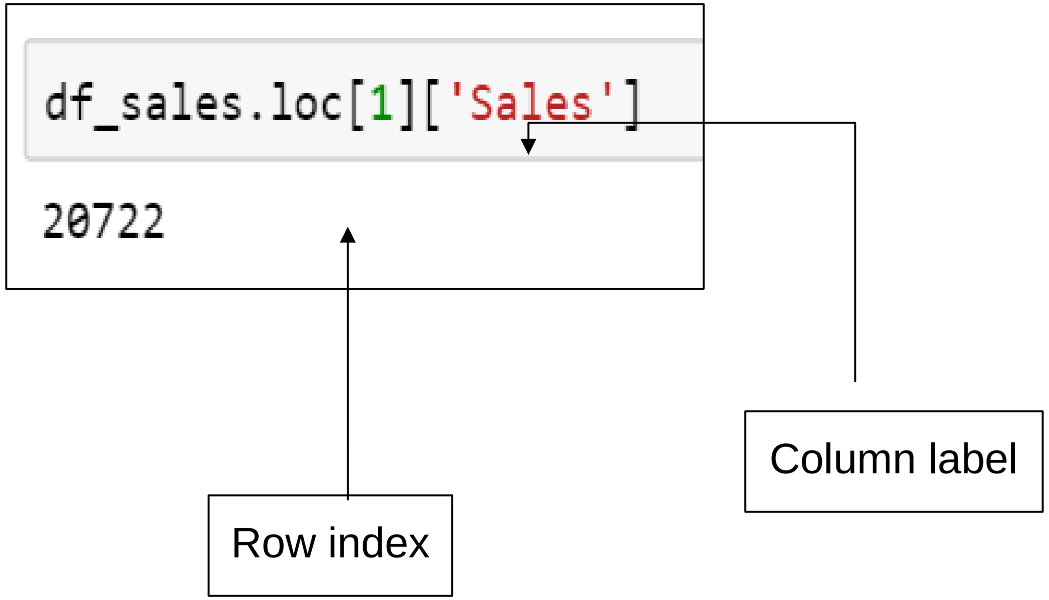


```
df_sales.iloc[:,[0,2]]
```

	Month	Sales
0	Jan	31037
1	Jan	20722
2	Jan	24557
3	Jan	34649
4	Jan	29795
5	Feb	29133
6	Feb	22695
7	Feb	28312

ACCESSING DATAFRAME ELEMENTS USING INDEXING

- The .loc[] selects the data by the label of the rows and column
- Retrieve the sales of the second store using the .loc[]



```
df_sales.loc[1]['Sales']
```

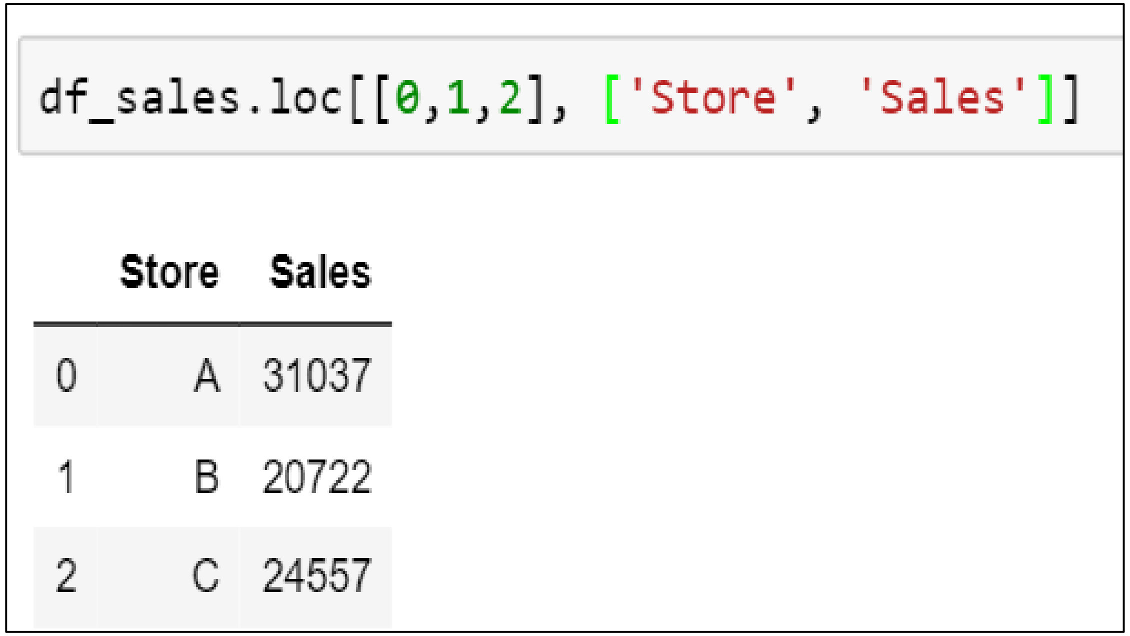
```
20722
```

Column label

Row index

ACCESSING DATAFRAME ELEMENTS USING INDEXING

- Retrieve the columns ‘Store’ and ‘Sales’ for first three stores

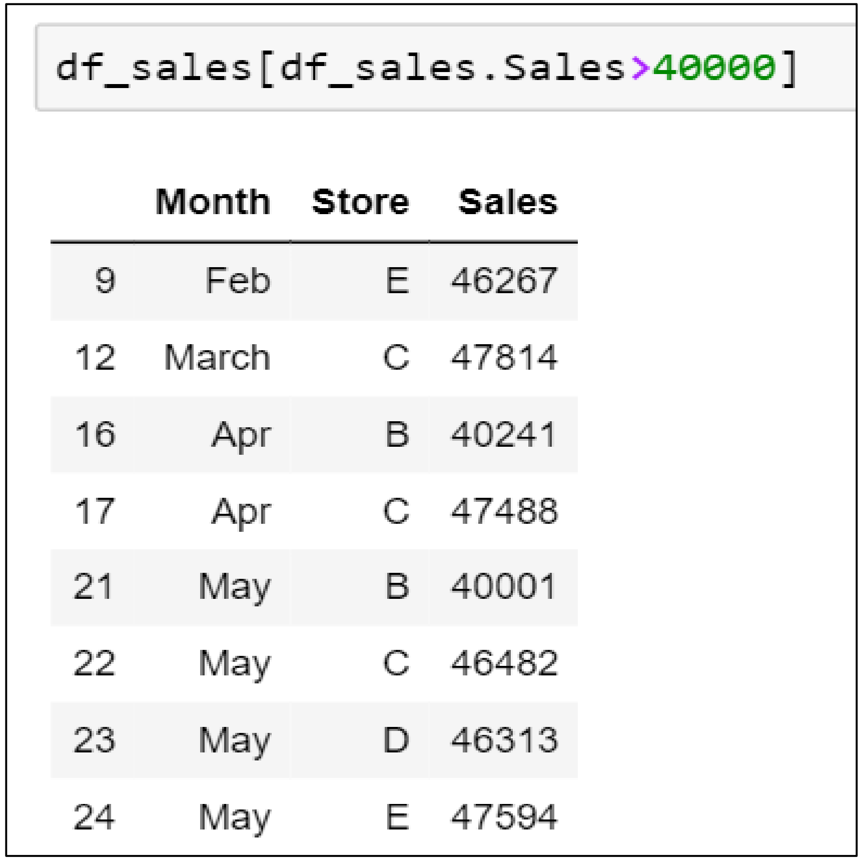


```
df_sales.loc[[0,1,2], ['Store', 'Sales']]
```

	Store	Sales
0	A	31037
1	B	20722
2	C	24557

ACCESSING DATAFRAME ELEMENTS USING CONDITIONS

- Retrieve the information of the stores whose sales is more than 40000

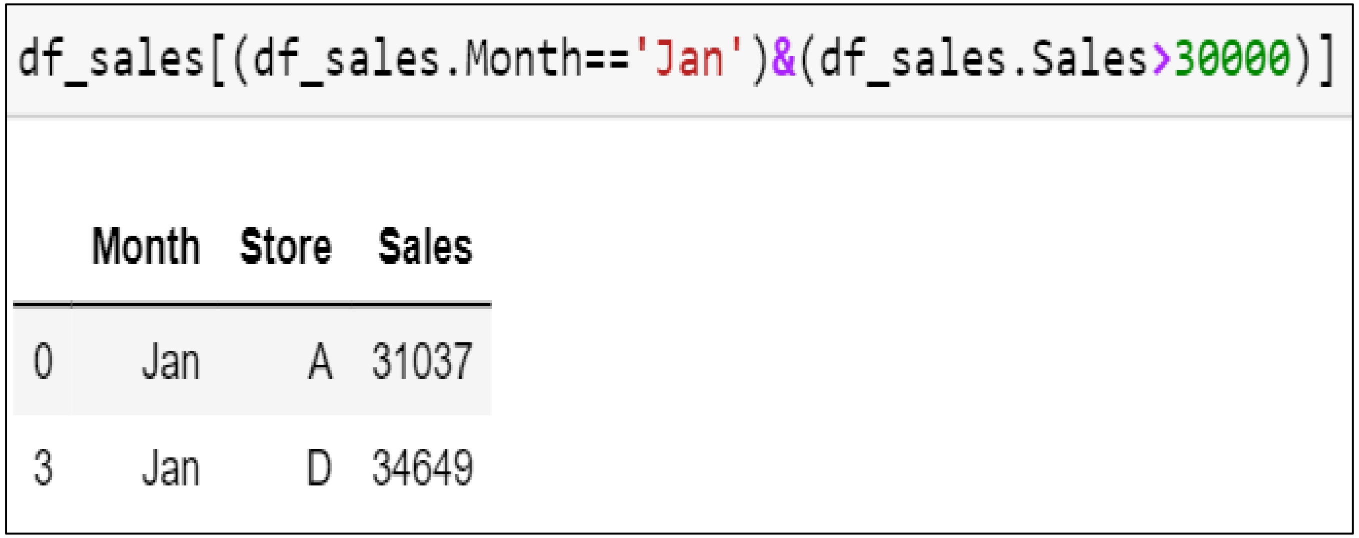


```
df_sales[df_sales.Sales>40000]
```

	Month	Store	Sales
9	Feb	E	46267
12	March	C	47814
16	Apr	B	40241
17	Apr	C	47488
21	May	B	40001
22	May	C	46482
23	May	D	46313
24	May	E	47594

ACCESSING DATAFRAME ELEMENTS USING CONDITIONS

- Retrieve the rows where the sales of the store was greater than 30000 in the month of January



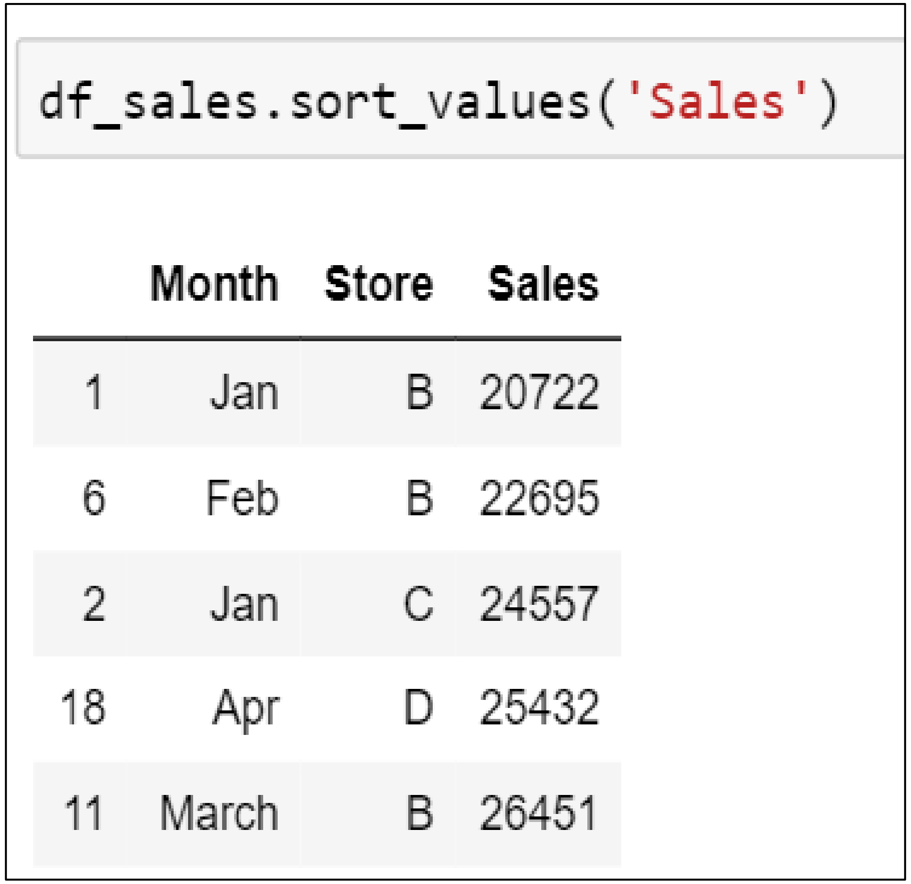
```
df_sales[(df_sales.Month=='Jan')&(df_sales.Sales>30000)]
```

```
   Month  Store  Sales
0    Jan      A  31037
3    Jan      D  34649
```

DataFrame Sorting

DATAFRAME SORTING

- Sort the rows in ascending order of the column ‘Sales’

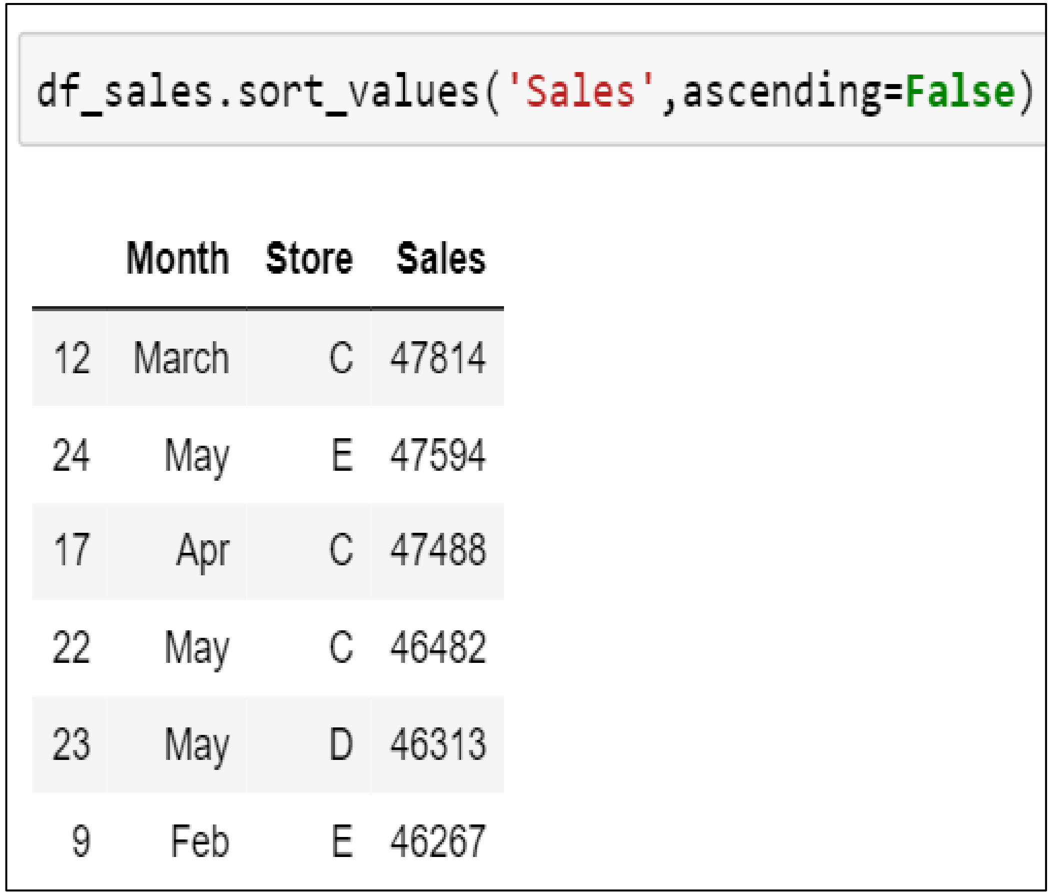


```
df_sales.sort_values('Sales')
```

	Month	Store	Sales
1	Jan	B	20722
6	Feb	B	22695
2	Jan	C	24557
18	Apr	D	25432
11	March	B	26451

DATAFRAME SORTING

- Sorting rows such that values in the column ‘Sales’ are in the descending order



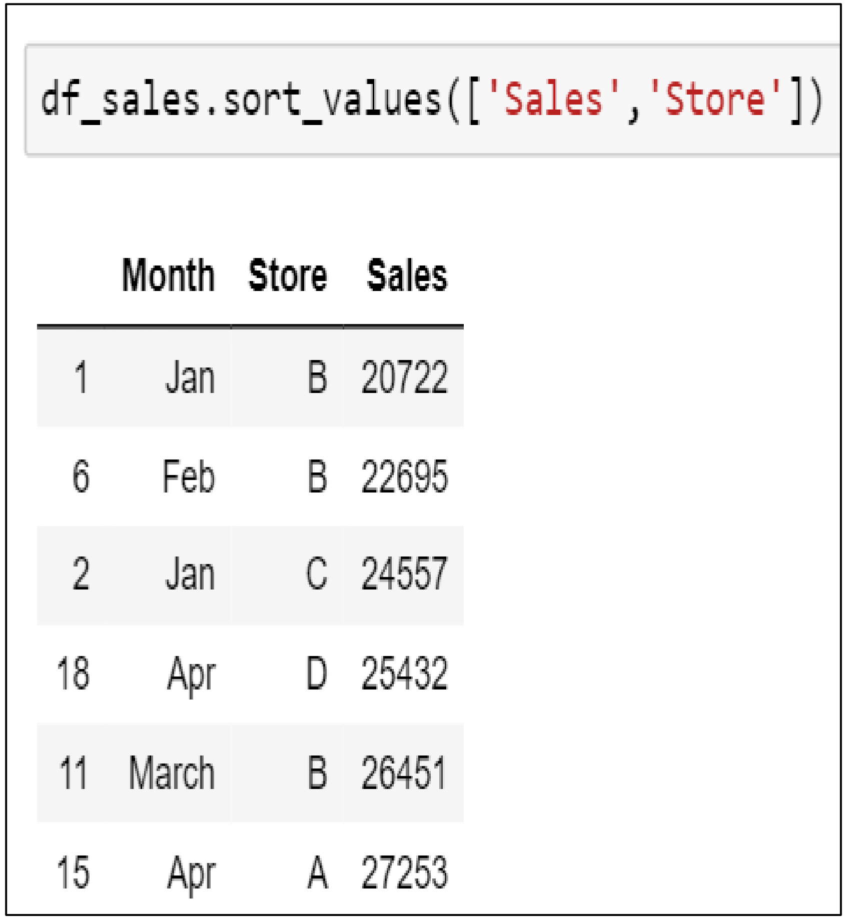
```
df_sales.sort_values('Sales',ascending=False)
```

	Month	Store	Sales
12	March	C	47814
24	May	E	47594
17	Apr	C	47488
22	May	C	46482
23	May	D	46313
9	Feb	E	46267

DATAFRAME SORTING

- While sorting the DataFrame by multiple columns, the .sort_values() first sorts the first passed variable and then the next variable

- In this case, the function first sorts the variable ‘Sales’ and then the variable ‘Store’

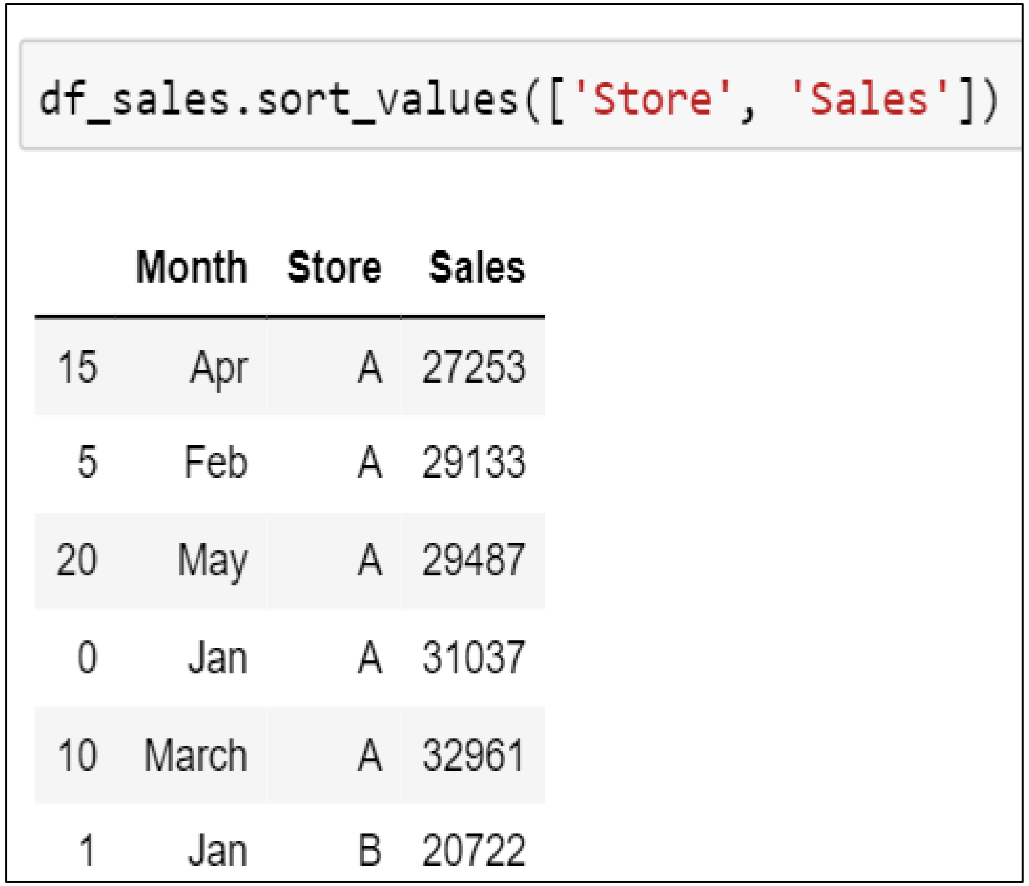


```
df_sales.sort_values(['Sales','Store'])
```

	Month	Store	Sales
1	Jan	B	20722
6	Feb	B	22695
2	Jan	C	24557
18	Apr	D	25432
11	March	B	26451
15	Apr	A	27253

SORT THE DATAFRAME

- Sort the DataFrame by values in the columns ‘Store’ and ‘Sales’

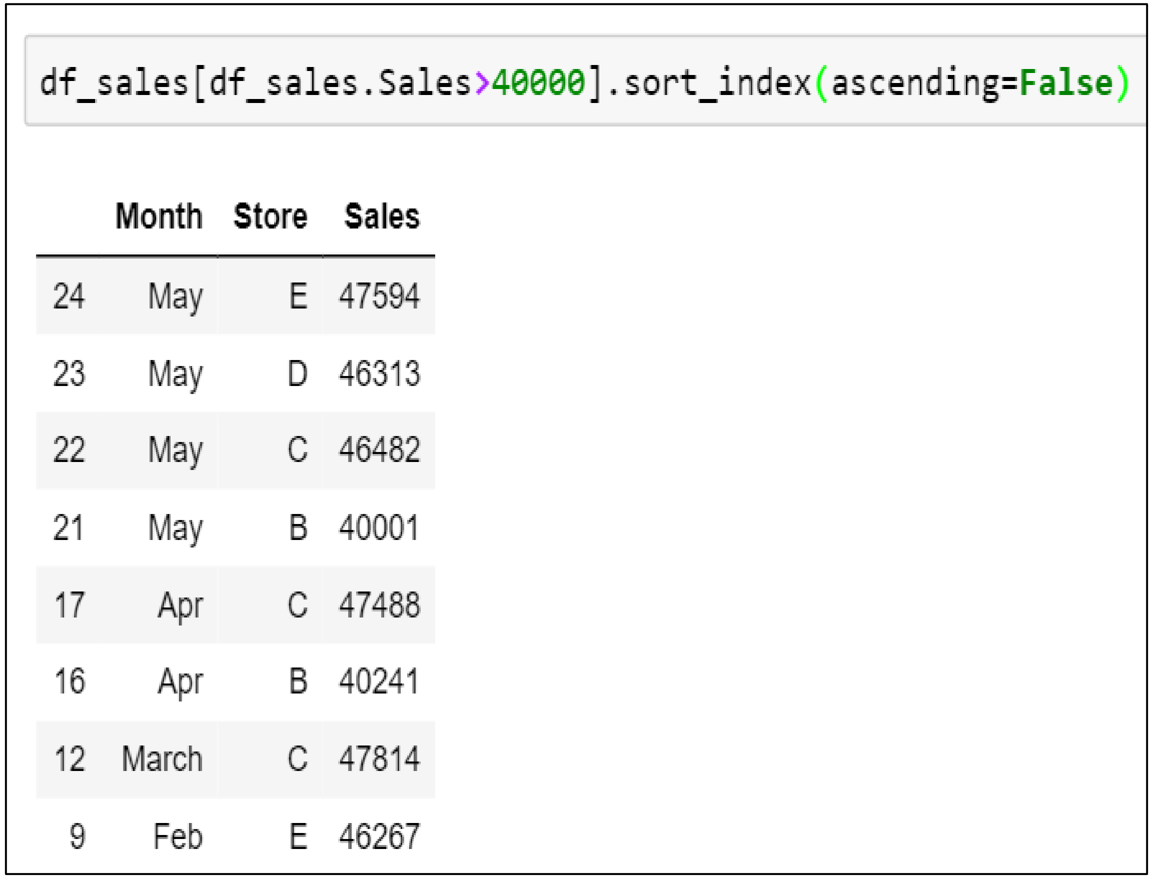


```
df_sales.sort_values(['Store', 'Sales'])
```

	Month	Store	Sales
15	Apr	A	27253
5	Feb	A	29133
20	May	A	29487
0	Jan	A	31037
10	March	A	32961
1	Jan	B	20722

SORT THE DATAFRAME

- Get all the rows where Sales > 40000 and then sort all the rows by index using the sort_index() method



```
df_sales[df_sales.Sales>40000].sort_index(ascending=False)
```

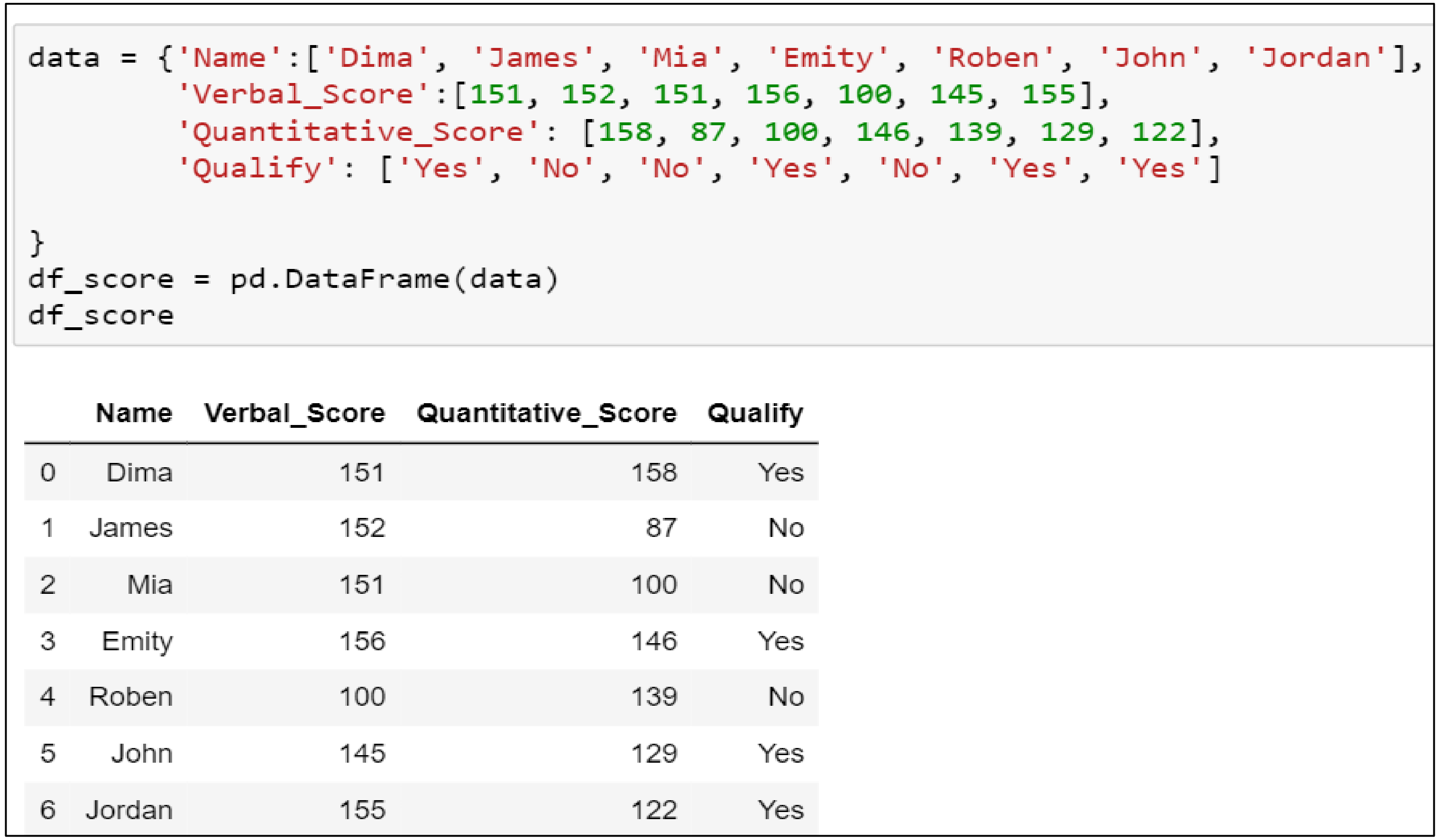
	Month	Store	Sales
24	May	E	47594
23	May	D	46313
22	May	C	46482
21	May	B	40001
17	Apr	C	47488
16	Apr	B	40241
12	March	C	47814
9	Feb	E	46267

Ranking in DataFrame

RANKING IN DATAFRAME

Example

Create a DataFrame of seven students as shown below:

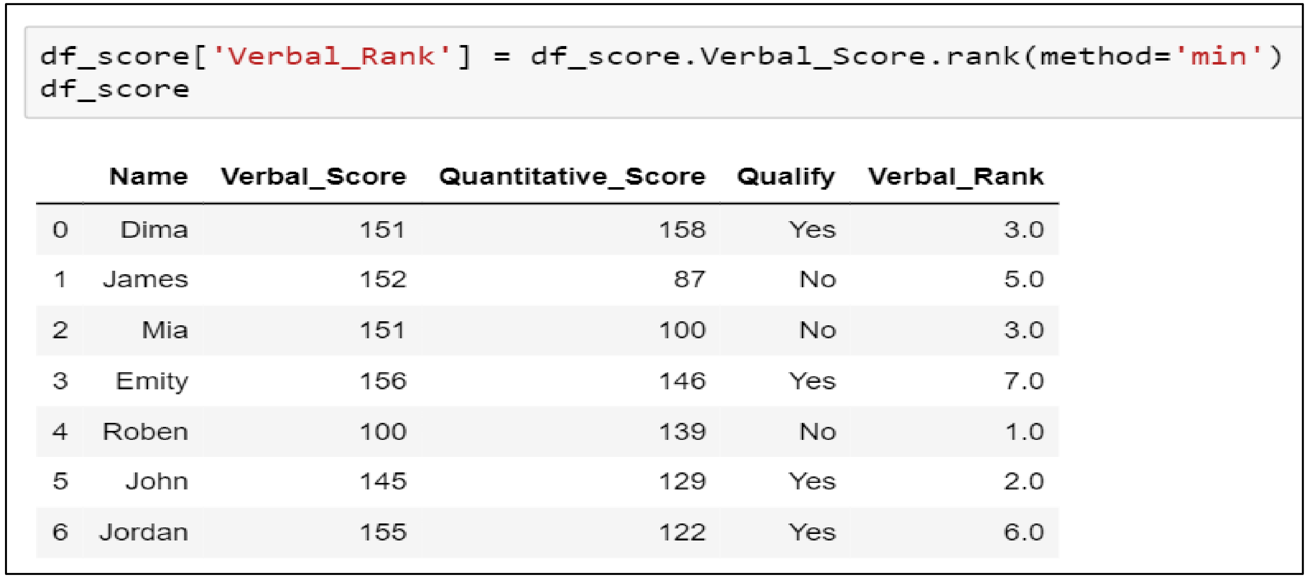


```
data = {'Name':['Dima', 'James', 'Mia', 'Emity', 'Roben', 'John', 'Jordan'],
        'Verbal_Score':[151, 152, 151, 156, 100, 145, 155],
        'Quantitative_Score': [158, 87, 100, 146, 139, 129, 122],
        'Qualify': ['Yes', 'No', 'No', 'Yes', 'No', 'Yes', 'Yes']

}
df_score = pd.DataFrame(data)
df_score
```

	Name	Verbal_Score	Quantitative_Score	Qualify
0	Dima	151	158	Yes
1	James	152	87	No
2	Mia	151	100	No
3	Emity	156	146	Yes
4	Roben	100	139	No
5	John	145	129	Yes
6	Jordan	155	122	Yes

RANKING IN DATAFRAME

- Rank the DataFrame by values in the column ‘Verbal_Score’ using the parameter, method = ‘min’
- If the Verbal_Score is same for two or more observations, then the ‘min’ method assigns the minimum rank to all the equal scores
- Here it assigned the rank ‘3’ to the Verbal_Score = 151

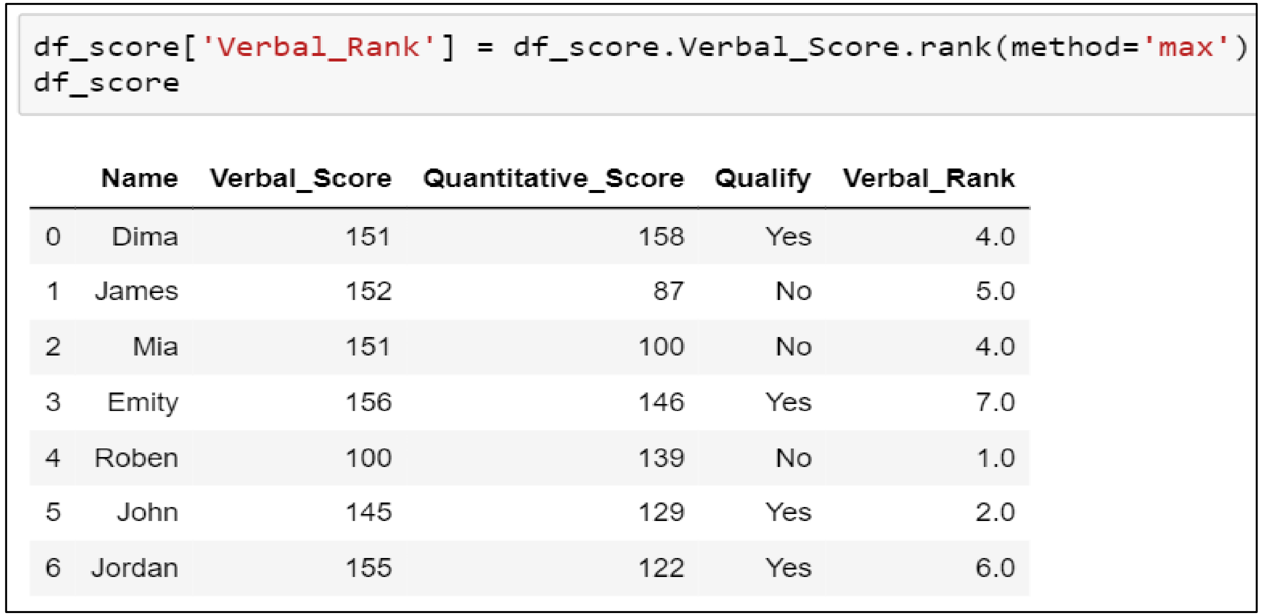


```
df_score['Verbal_Rank'] = df_score.Verbal_Score.rank(method='min')
df_score
```

	Name	Verbal_Score	Quantitative_Score	Qualify	Verbal_Rank
0	Dima	151	158	Yes	3.0
1	James	152	87	No	5.0
2	Mia	151	100	No	3.0
3	Emity	156	146	Yes	7.0
4	Roben	100	139	No	1.0
5	John	145	129	Yes	2.0
6	Jordan	155	122	Yes	6.0

RANKING IN DATAFRAME

- Rank the DataFrame by values in the column ‘Verbal_Score’ using the parameter, method = ‘max’
- If the Verbal_Score is same for two or more observations, then the ‘max’ method assigns the maximum rank to all the equal scores
- Here it assigned the rank ‘4’ to the Verbal_Score = 151

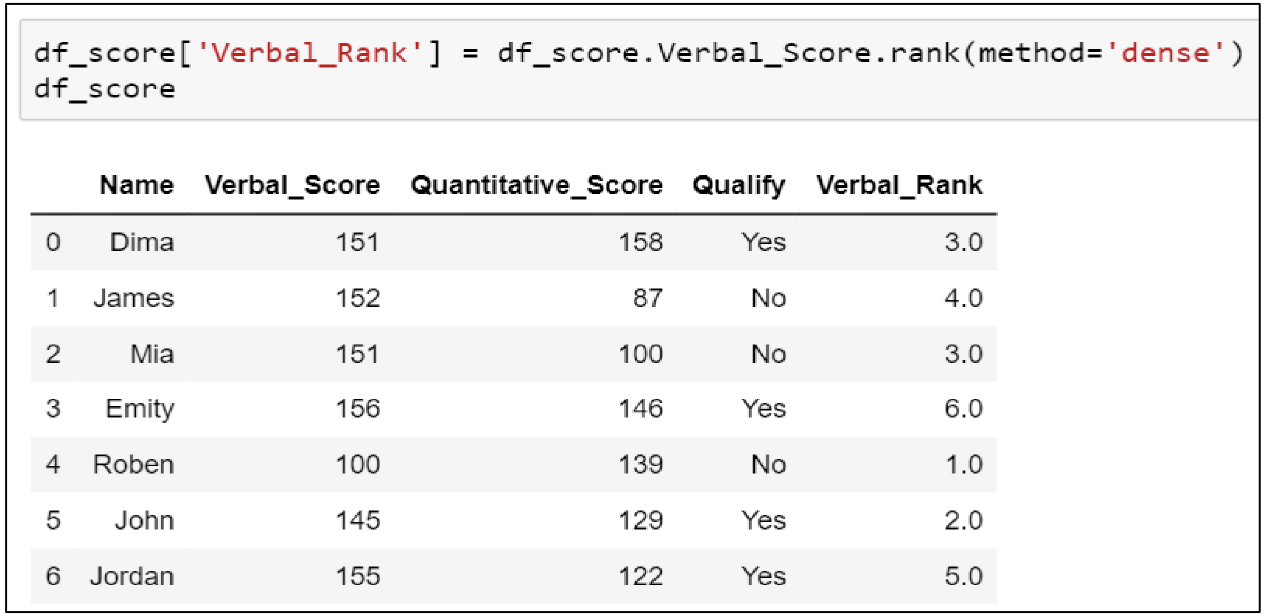


```
df_score['Verbal_Rank'] = df_score.Verbal_Score.rank(method='max')
df_score
```

	Name	Verbal_Score	Quantitative_Score	Qualify	Verbal_Rank
0	Dima	151	158	Yes	4.0
1	James	152	87	No	5.0
2	Mia	151	100	No	4.0
3	Emity	156	146	Yes	7.0
4	Roben	100	139	No	1.0
5	John	145	129	Yes	2.0
6	Jordan	155	122	Yes	6.0

RANKING IN DATAFRAME

- Rank the DataFrame by values in the column ‘Verbal_Score’ using the parameter, method = ‘dense’
- This method does not skip a rank, like the ‘min’ and ‘max’ method
- Here, it assigned the rank ‘3’ to Verbal_Score = 151, and ‘4’ to next greater Verbal_Score = 152

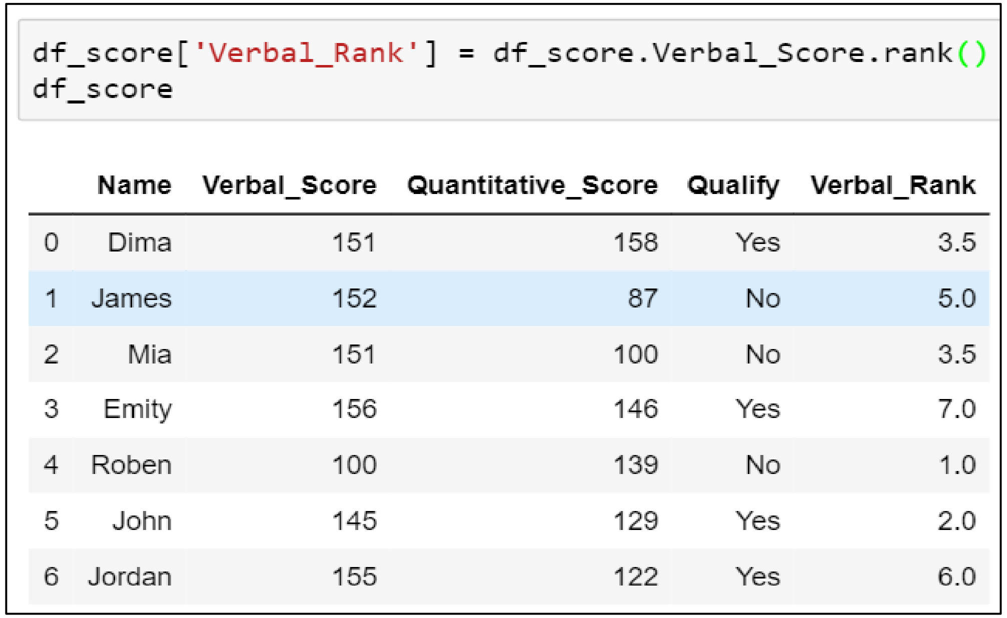


```
df_score['Verbal_Rank'] = df_score.Verbal_Score.rank(method='dense')
df_score
```

	Name	Verbal_Score	Quantitative_Score	Qualify	Verbal_Rank
0	Dima	151	158	Yes	3.0
1	James	152	87	No	4.0
2	Mia	151	100	No	3.0
3	Emity	156	146	Yes	6.0
4	Roben	100	139	No	1.0
5	John	145	129	Yes	2.0
6	Jordan	155	122	Yes	5.0

RANKING IN DATAFRAME

- Rank the DataFrame by values in the column ‘Verbal_Score ’ in descending order
- By default, the method is ‘average’ in the .rank(), and it assigns the average rank to the equal values
- Here, it assigned the rank ‘3.5’ to the same Verbal_Score = 151



```
df_score['Verbal_Rank'] = df_score.Verbal_Score.rank()
df_score
```

	Name	Verbal_Score	Quantitative_Score	Qualify	Verbal_Rank
0	Dima	151	158	Yes	3.5
1	James	152	87	No	5.0
2	Mia	151	100	No	3.5
3	Emity	156	146	Yes	7.0
4	Roben	100	139	No	1.0
5	John	145	129	Yes	2.0
6	Jordan	155	122	Yes	6.0

DataFrame Concatenation

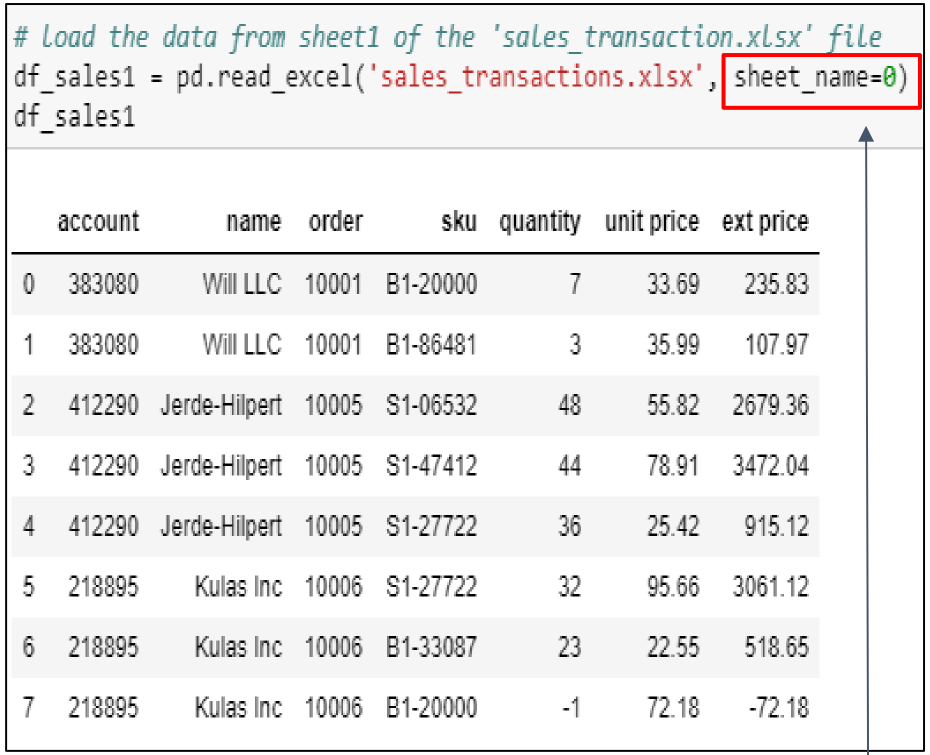
DATAFRAME CONCATENATION

Pandas DataFrames can be concatenated vertically (column-wise) and horizontally (row-wise)

The concat() and append() methods are used to concatenate the DataFrames

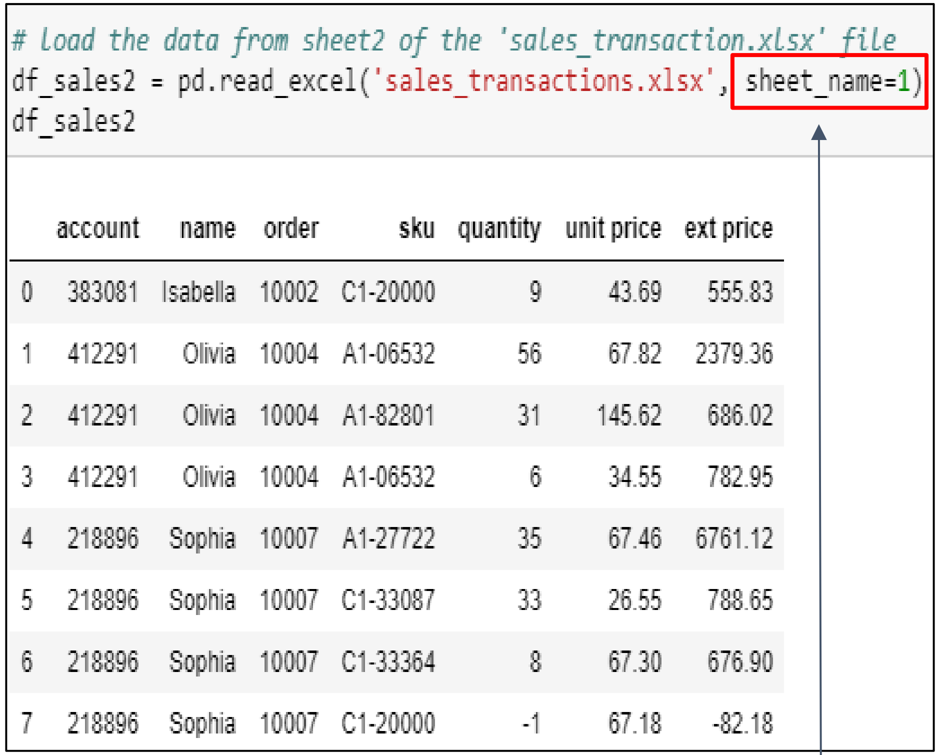
DATAFRAME CONCATENATION

- Use the following DataFrames for the study



```python
# load the data from sheet1 of the 'sales_transaction.xlsx' file
df_sales1 = pd.read_excel('sales_transactions.xlsx', sheet_name=0)
df_sales1
```

	account	name	order	sku	quantity	unit price	ext price
0	383080	Will LLC	10001	B1-20000	7	33.69	235.83
1	383080	Will LLC	10001	B1-86481	3	35.99	107.97
2	412290	Jerde-Hilpert	10005	S1-06532	48	55.82	2679.36
3	412290	Jerde-Hilpert	10005	S1-47412	44	78.91	3472.04
4	412290	Jerde-Hilpert	10005	S1-27722	36	25.42	915.12
5	218895	Kulas Inc	10006	S1-27722	32	95.66	3061.12
6	218895	Kulas Inc	10006	B1-33087	23	22.55	518.65
7	218895	Kulas Inc	10006	B1-20000	-1	72.18	-72.18



```python
# load the data from sheet2 of the 'sales_transaction.xlsx' file
df_sales2 = pd.read_excel('sales_transactions.xlsx', sheet_name=1)
df_sales2
```

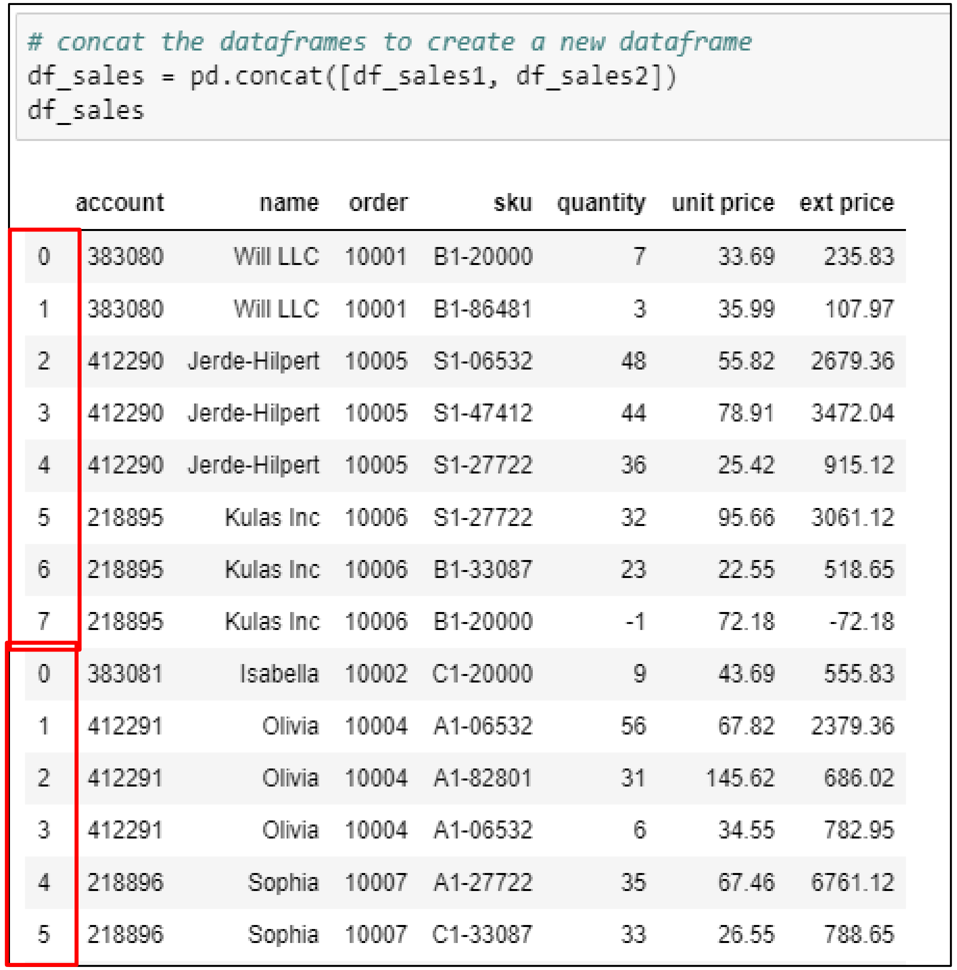
	account	name	order	sku	quantity	unit price	ext price
0	383081	Isabella	10002	C1-20000	9	43.69	555.83
1	412291	Olivia	10004	A1-06532	56	67.82	2379.36
2	412291	Olivia	10004	A1-82801	31	145.62	686.02
3	412291	Olivia	10004	A1-06532	6	34.55	782.95
4	218896	Sophia	10007	A1-27722	35	67.46	6761.12
5	218896	Sophia	10007	C1-33087	33	26.55	788.65
6	218896	Sophia	10007	C1-33364	8	67.30	676.90
7	218896	Sophia	10007	C1-20000	-1	67.18	-82.18

Sales details of company A

Sales details of company B

DATAFRAME CONCATENATION

- We concatenate the two DataFrames using concat() method
- By default, the concat() method concatenates along the axis = 0 (vertically)
- The concatenation is in the order they are passed in the function
- The index numbers of the concatenated DataFrame are of the actual DataFrames



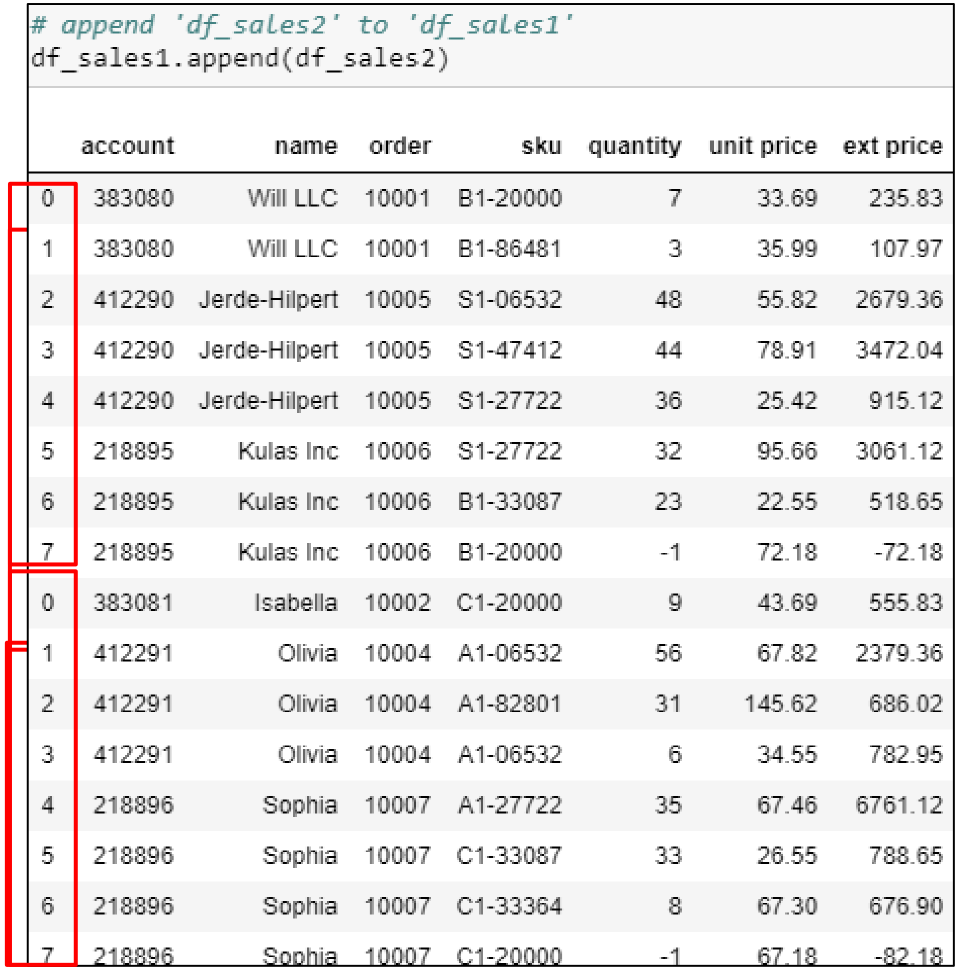
```
# concat the dataframes to create a new dataframe
df_sales = pd.concat([df_sales1, df_sales2])
df_sales
```

	account	name	order	sku	quantity	unit price	ext price
0	383080	Will LLC	10001	B1-20000	7	33.69	235.83
1	383080	Will LLC	10001	B1-86481	3	35.99	107.97
2	412290	Jerde-Hilpert	10005	S1-06532	48	55.82	2679.36
3	412290	Jerde-Hilpert	10005	S1-47412	44	78.91	3472.04
4	412290	Jerde-Hilpert	10005	S1-27722	36	25.42	915.12
5	218895	Kulas Inc	10006	S1-27722	32	95.66	3061.12
6	218895	Kulas Inc	10006	B1-33087	23	22.55	518.65
7	218895	Kulas Inc	10006	B1-20000	-1	72.18	-72.18
0	383081	Isabella	10002	C1-20000	9	43.69	555.83
1	412291	Olivia	10004	A1-06532	56	67.82	2379.36
2	412291	Olivia	10004	A1-82801	31	145.62	686.02
3	412291	Olivia	10004	A1-06532	6	34.55	782.95
4	218896	Sophia	10007	A1-27722	35	67.46	6761.12
5	218896	Sophia	10007	C1-33087	33	26.55	788.65

DATAFRAME CONCATENATION

- The append() method is used append a DataFrame with another

- We append the customers data of company ‘B’ to data of company ‘A’

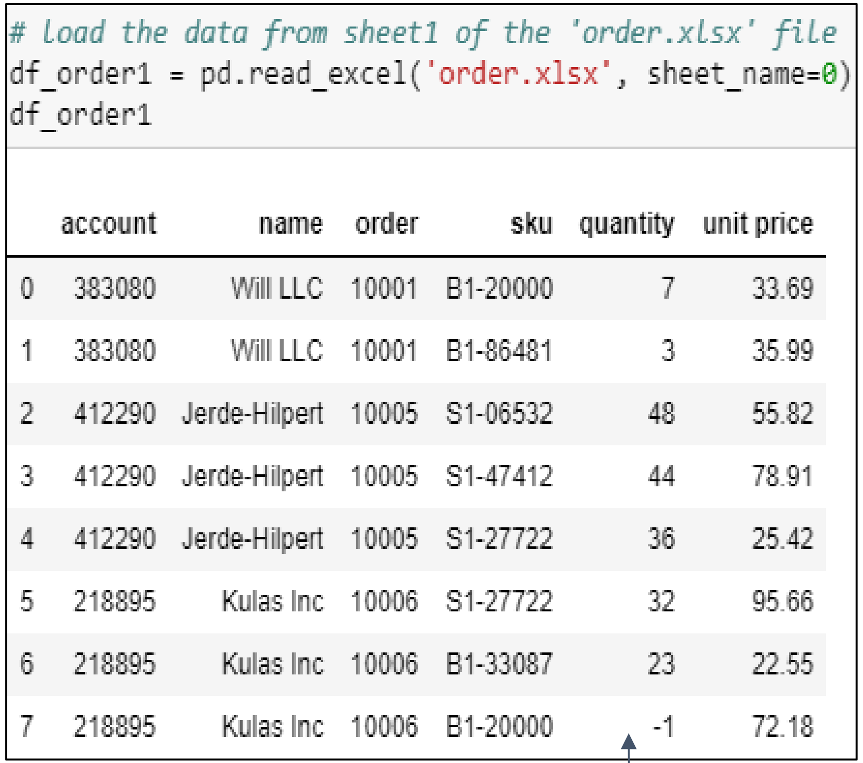


```
# append 'df_sales2' to 'df_sales1'
df_sales1.append(df_sales2)
```

	account	name	order	sku	quantity	unit price	ext price
0	383080	Will LLC	10001	B1-20000	7	33.69	235.83
1	383080	Will LLC	10001	B1-86481	3	35.99	107.97
2	412290	Jerde-Hilpert	10005	S1-06532	48	55.82	2679.36
3	412290	Jerde-Hilpert	10005	S1-47412	44	78.91	3472.04
4	412290	Jerde-Hilpert	10005	S1-27722	36	25.42	915.12
5	218895	Kulas Inc	10006	S1-27722	32	95.66	3061.12
6	218895	Kulas Inc	10006	B1-33087	23	22.55	518.65
7	218895	Kulas Inc	10006	B1-20000	-1	72.18	-72.18
0	383081	Isabella	10002	C1-20000	9	43.69	555.83
1	412291	Olivia	10004	A1-06532	56	67.82	2379.36
2	412291	Olivia	10004	A1-82801	31	145.62	686.02
3	412291	Olivia	10004	A1-06532	6	34.55	782.95
4	218896	Sophia	10007	A1-27722	35	67.46	6761.12
5	218896	Sophia	10007	C1-33087	33	26.55	788.65
6	218896	Sophia	10007	C1-33364	8	67.30	676.90
7	218896	Sophia	10007	C1-20000	-1	67.18	-82.18

DATAFRAME CONCATENATION

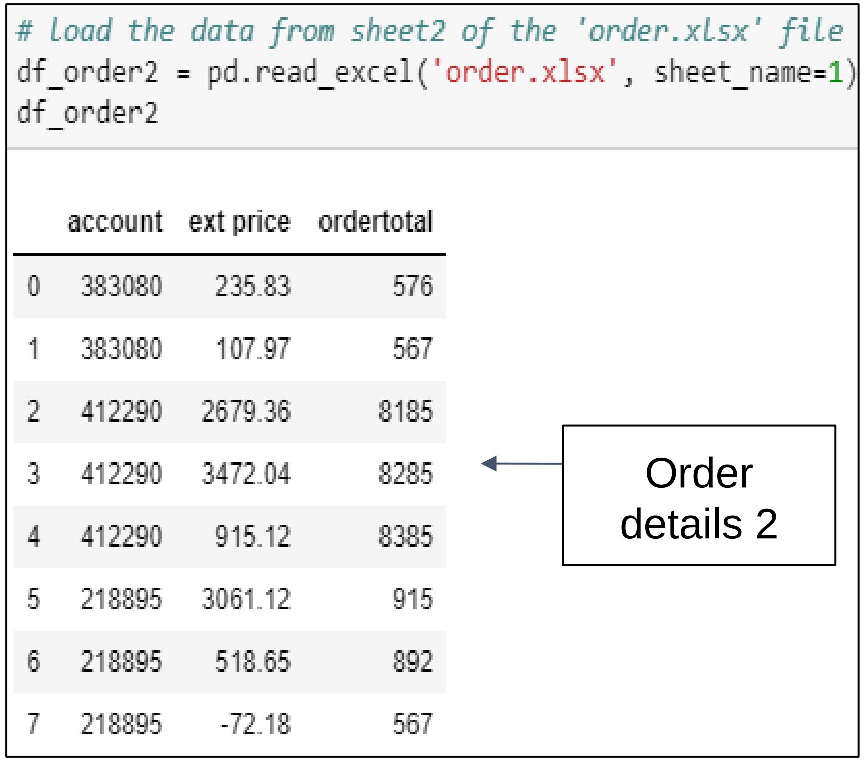
- Use the following DataFrames for the study



```python
# load the data from sheet1 of the 'order.xlsx' file
df_order1 = pd.read_excel('order.xlsx', sheet_name=0)
df_order1
```

	account	name	order	sku	quantity	unit price
0	383080	Will LLC	10001	B1-20000	7	33.69
1	383080	Will LLC	10001	B1-86481	3	35.99
2	412290	Jerde-Hilpert	10005	S1-06532	48	55.82
3	412290	Jerde-Hilpert	10005	S1-47412	44	78.91
4	412290	Jerde-Hilpert	10005	S1-27722	36	25.42
5	218895	Kulas Inc	10006	S1-27722	32	95.66
6	218895	Kulas Inc	10006	B1-33087	23	22.55
7	218895	Kulas Inc	10006	B1-20000	-1	72.18

Order details 1



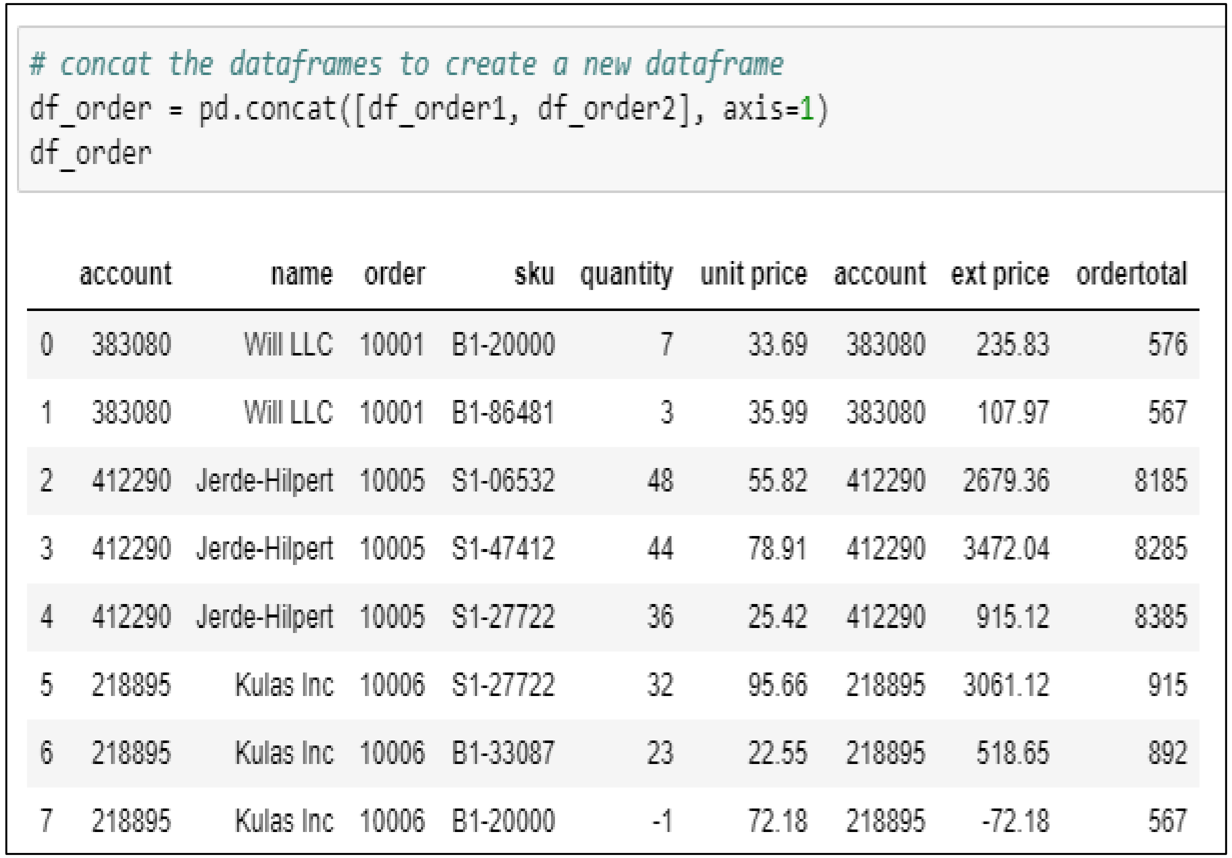
```python
# load the data from sheet2 of the 'order.xlsx' file
df_order2 = pd.read_excel('order.xlsx', sheet_name=1)
df_order2
```

	account	ext price	ordertotal
0	383080	235.83	576
1	383080	107.97	567
2	412290	2679.36	8185
3	412290	3472.04	8285
4	412290	915.12	8385
5	218895	3061.12	915
6	218895	518.65	892
7	218895	-72.18	567

Order details 2

DATAFRAME CONCATENATION

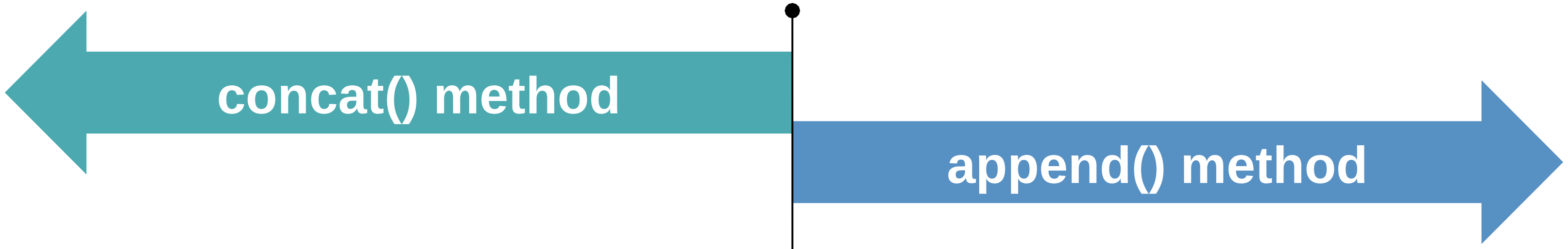
- The parameter, ‘axis = 1’ concatenates the DataFrames horizontally
- The concatenation is in the order they are passed in the function



```
# concat the dataframes to create a new dataframe
df_order = pd.concat([df_order1, df_order2], axis=1)
df_order
```

	account	name	order	sku	quantity	unit price	account	ext price	ordertotal
0	383080	Will LLC	10001	B1-20000	7	33.69	383080	235.83	576
1	383080	Will LLC	10001	B1-86481	3	35.99	383080	107.97	567
2	412290	Jerde-Hilpert	10005	S1-06532	48	55.82	412290	2679.36	8185
3	412290	Jerde-Hilpert	10005	S1-47412	44	78.91	412290	3472.04	8285
4	412290	Jerde-Hilpert	10005	S1-27722	36	25.42	412290	915.12	8385
5	218895	Kulas Inc	10006	S1-27722	32	95.66	218895	3061.12	915
6	218895	Kulas Inc	10006	B1-33087	23	22.55	218895	518.65	892
7	218895	Kulas Inc	10006	B1-20000	-1	72.18	218895	-72.18	567

APPEND vs CONCAT



concat() method

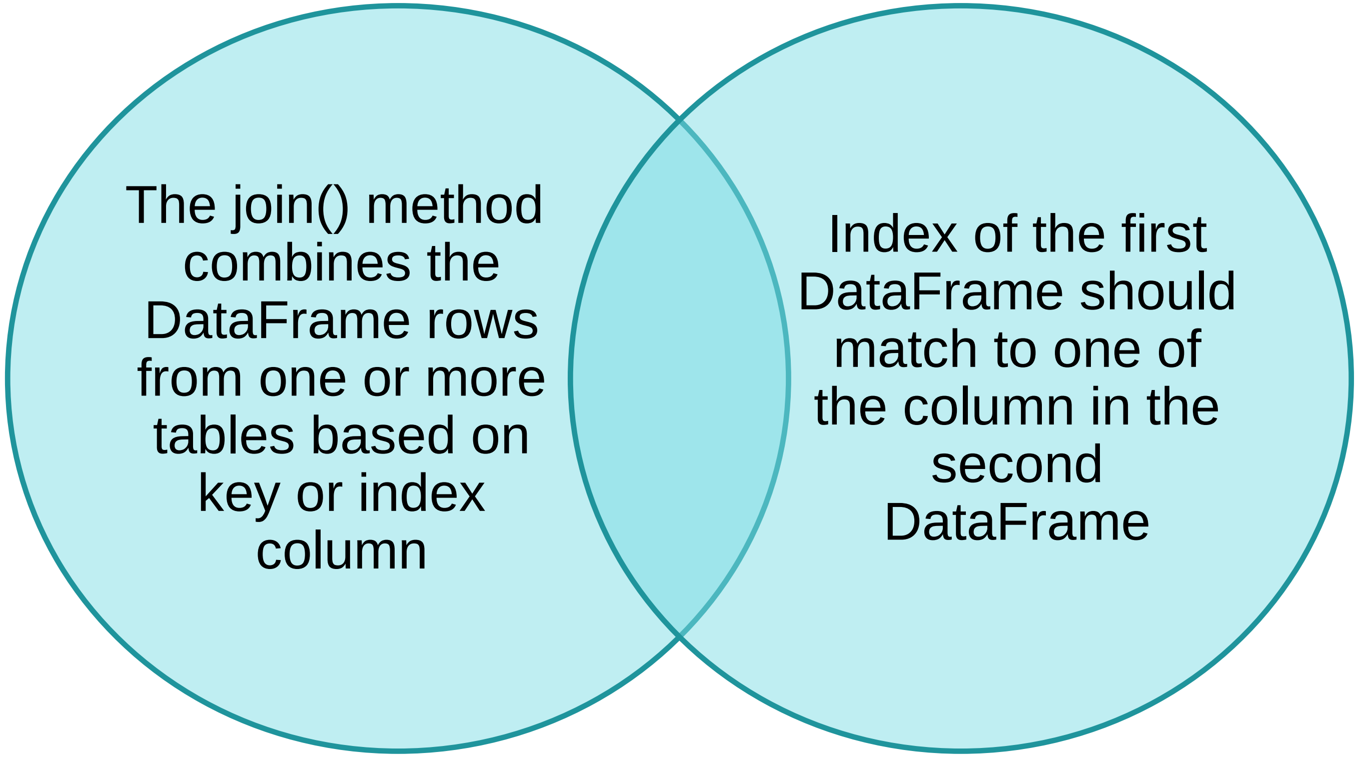
append() method

- Concatenates multiple DataFrames simultaneously

- Returns an error if we try to concatenate more than two DataFrames simultaneously

DataFrame Joins

DATAFRAMES JOINS

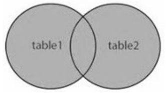
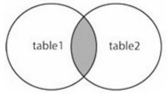
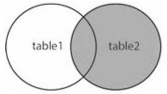
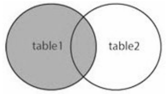


The join() method combines the DataFrame rows from one or more tables based on key or index column

Index of the first DataFrame should match to one of the column in the second DataFrame

TYPES OF JOINS

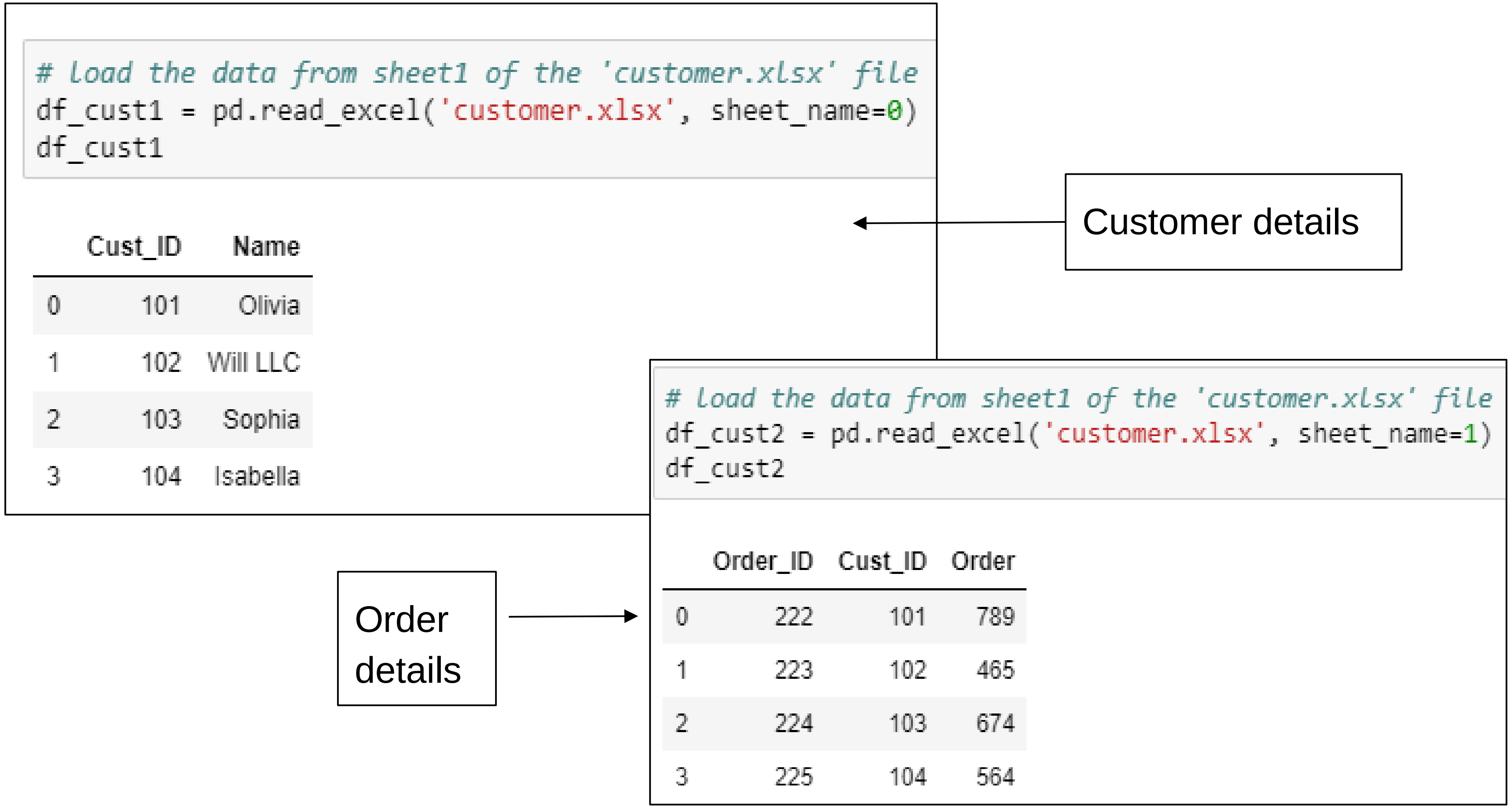
The join types can be specified using the parameter, ‘how’

how = ‘Type’	Description	
outer	Use union of index (or column) observed in both DataFrames	
inner	Use intersection of index (or column) observed in both DataFrames	
right	Use only the index found in the right DataFrame	
left	Use only the index (or column) found in the left DataFrame	

If the type is not specified, by default it is ‘left’

DATAFRAME JOINS

- Consider the following DataFrames for the study:



```
# load the data from sheet1 of the 'customer.xlsx' file
df_cust1 = pd.read_excel('customer.xlsx', sheet_name=0)
df_cust1
```

```
   Cust_ID      Name
0      101    Olivia
1      102  Will LLC
2      103    Sophia
3      104  Isabella
```

Customer details

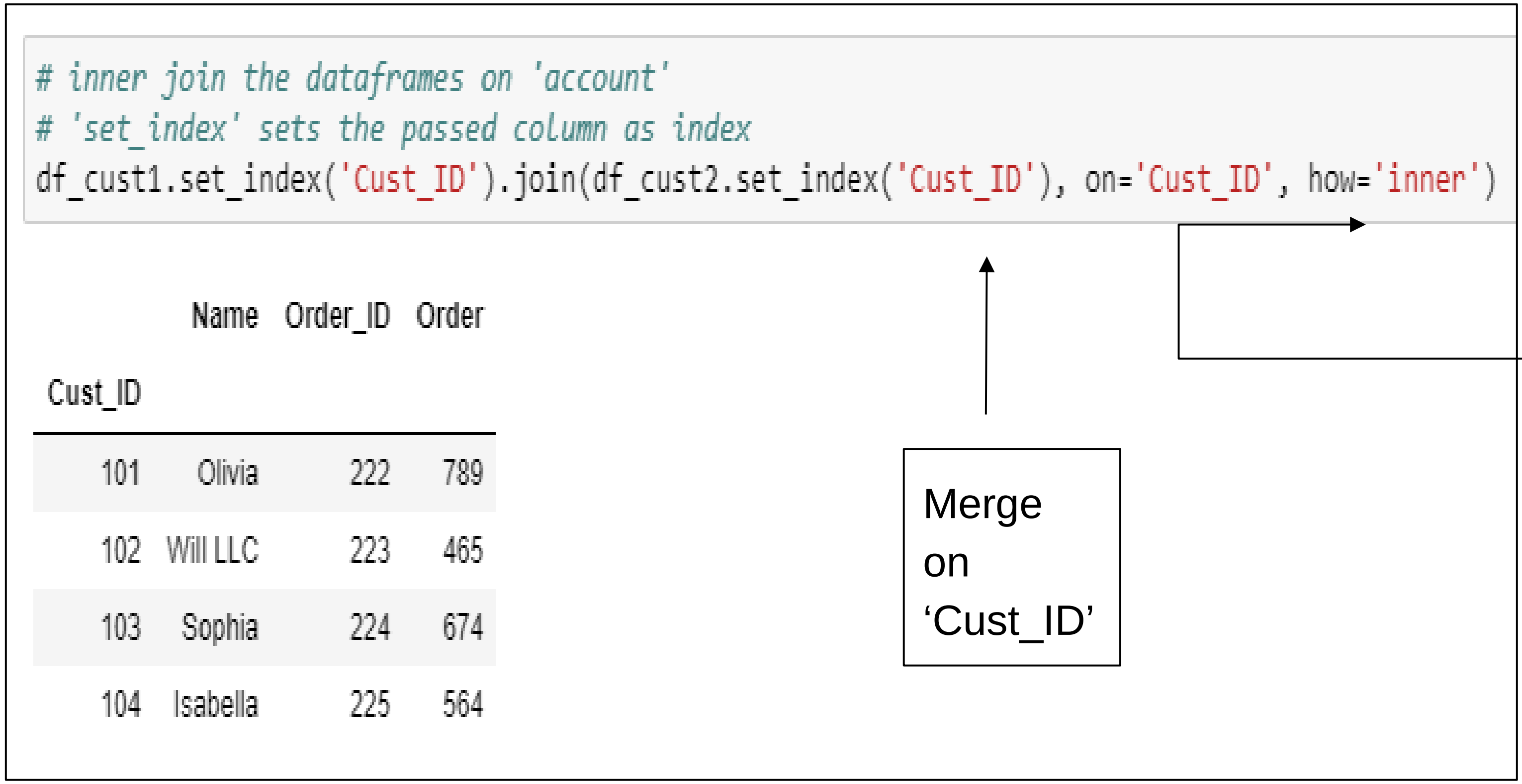
```
# load the data from sheet1 of the 'customer.xlsx' file
df_cust2 = pd.read_excel('customer.xlsx', sheet_name=1)
df_cust2
```

```
   Order_ID  Cust_ID  Order
0       222      101    789
1       223      102    465
2       224      103    674
3       225      104    564
```

Order details

DATAFRAME JOINS - INNER JOINS

- Join the DataFrames to get the order details along with the customer information



```
# inner join the dataframes on 'account'
# 'set_index' sets the passed column as index
df_cust1.set_index('Cust_ID').join(df_cust2.set_index('Cust_ID'), on='Cust_ID', how='inner')
```

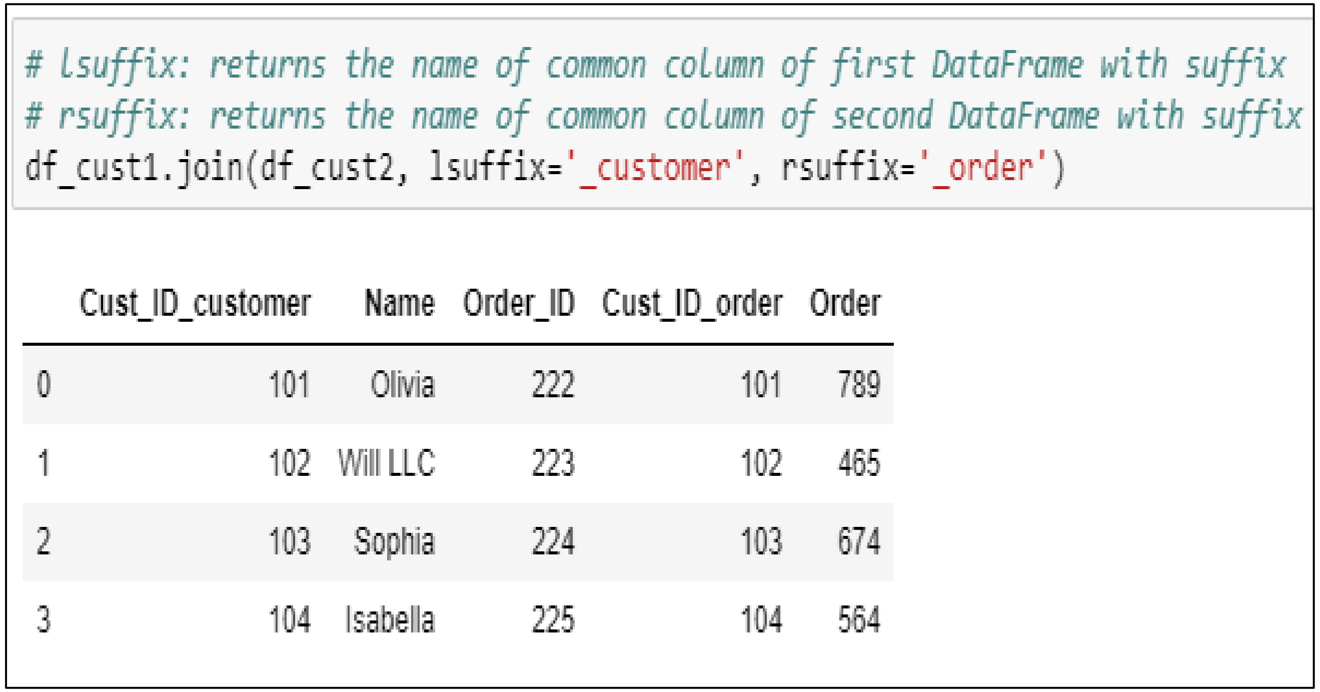
```
           Name  Order_ID  Order
Cust_ID
    101  Olivia       222    789
    102 Will LLC      223    465
    103  Sophia       224    674
    104 Isabella      225    564
```

Merge on 'Cust_ID'

Merge includes the common IDs in both the DataFrames

DATAFRAME JOINS USING INDEX

- Resultant DataFrame includes rows from both the DataFrames with same index as of ‘df_cust1’



```
# lsuffix: returns the name of common column of first DataFrame with suffix
# rsuffix: returns the name of common column of second DataFrame with suffix
df_cust1.join(df_cust2, lsuffix='_customer', rsuffix='_order')
```

	Cust_ID_customer	Name	Order_ID	Cust_ID_order	Order
0	101	Olivia	222	101	789
1	102	Will LLC	223	102	465
2	103	Sophia	224	103	674
3	104	Isabella	225	104	564

DataFrame Merge

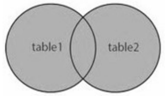
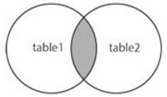
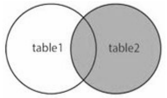
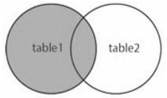
DATAFRAME MERGE

The merge() method concatenates the DataFrames based on one or more keys

If the column for join is not specified, the merge() method uses the overlapping column names as the keys

TYPES OF MERGE

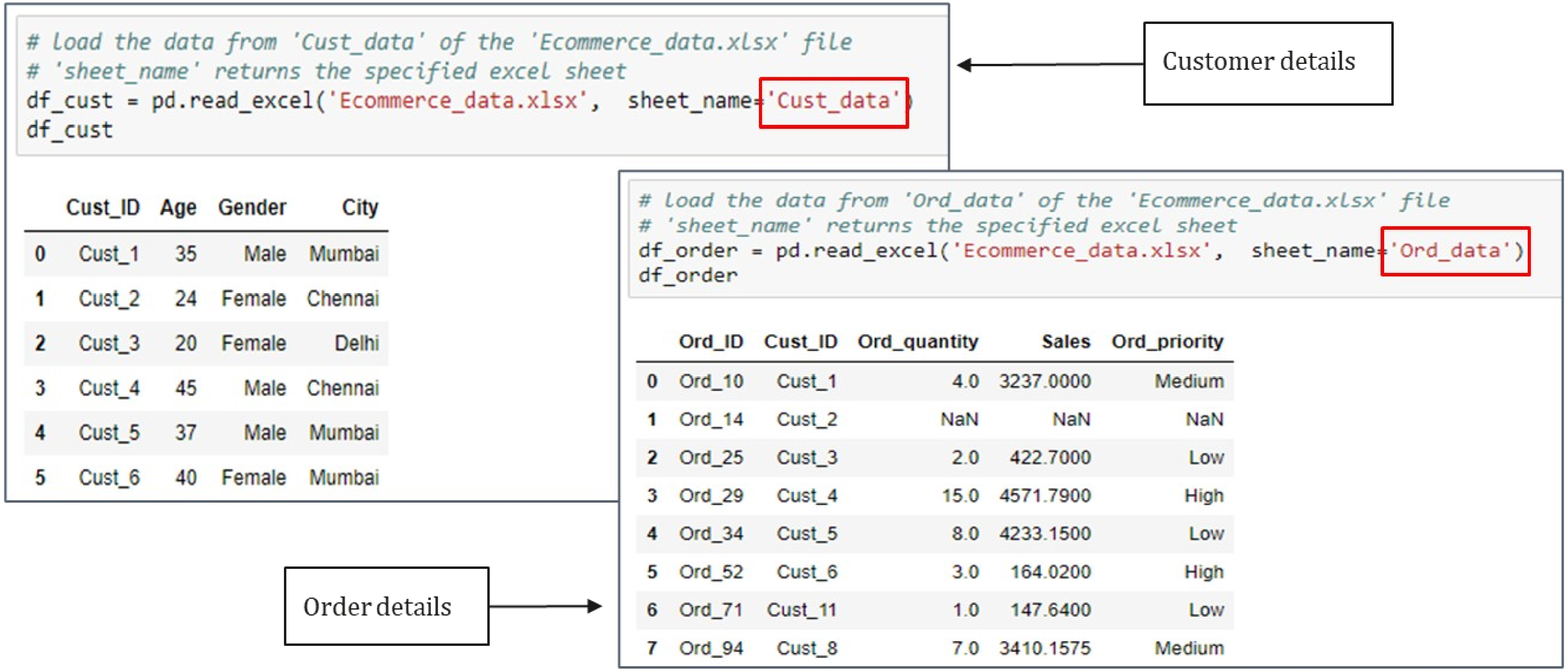
The merge types can be specified using the parameter, ‘how’

how = ‘Type’	Description	
outer	Use union of keys observed in both DataFrames	
inner	Use intersection of keys observed in both DataFrames	
right	Use only the keys found in the right DataFrame	
left	Use only the keys found in the left DataFrame	

By default the type is ‘inner’

DATAFRAME MERGE

- Use the following DataFrames for the study:



Customer details

```
# load the data from 'Cust_data' of the 'Ecommerce_data.xlsx' file
# 'sheet_name' returns the specified excel sheet
df_cust = pd.read_excel('Ecommerce_data.xlsx',  sheet_name='Cust_data')
df_cust
```

	Cust_ID	Age	Gender	City
0	Cust_1	35	Male	Mumbai
1	Cust_2	24	Female	Chennai
2	Cust_3	20	Female	Delhi
3	Cust_4	45	Male	Chennai
4	Cust_5	37	Male	Mumbai
5	Cust_6	40	Female	Mumbai

Order details

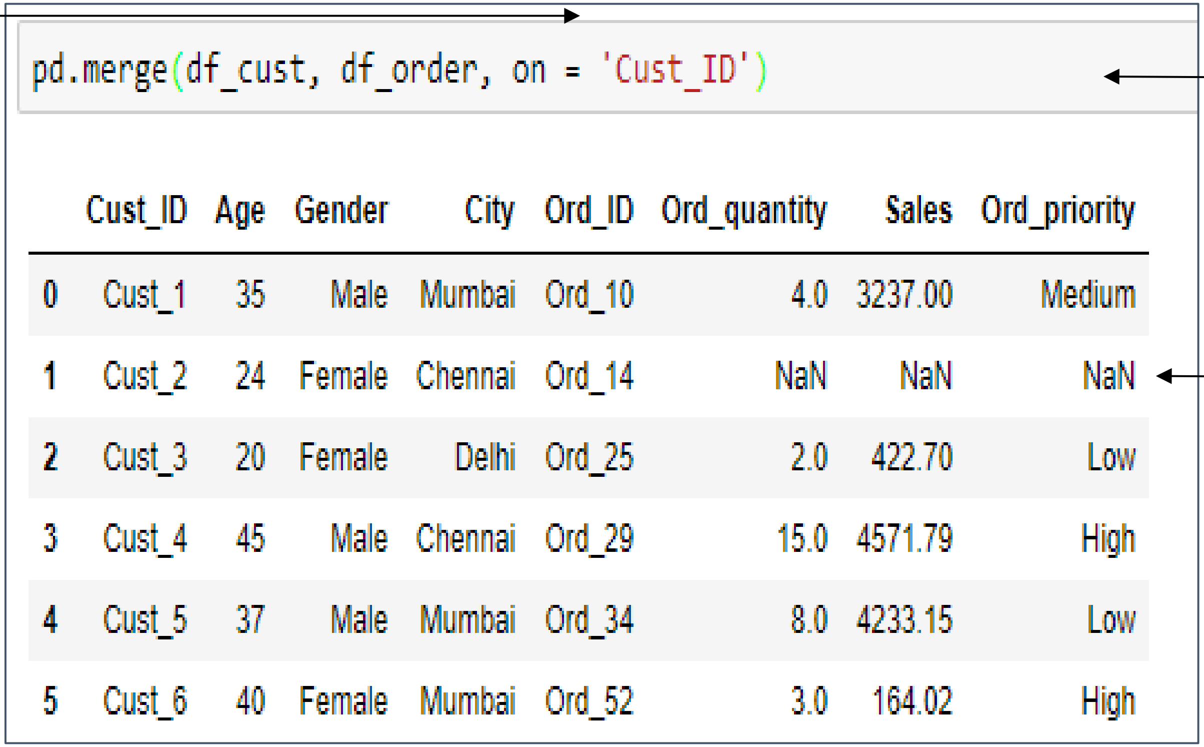
```
# load the data from 'Ord_data' of the 'Ecommerce_data.xlsx' file
# 'sheet_name' returns the specified excel sheet
df_order = pd.read_excel('Ecommerce_data.xlsx',  sheet_name='Ord_data')
df_order
```

	Ord_ID	Cust_ID	Ord_quantity	Sales	Ord_priority
0	Ord_10	Cust_1	4.0	3237.0000	Medium
1	Ord_14	Cust_2	NaN	NaN	NaN
2	Ord_25	Cust_3	2.0	422.7000	Low
3	Ord_29	Cust_4	15.0	4571.7900	High
4	Ord_34	Cust_5	8.0	4233.1500	Low
5	Ord_52	Cust_6	3.0	164.0200	High
6	Ord_71	Cust_11	1.0	147.6400	Low
7	Ord_94	Cust_8	7.0	3410.1575	Medium

DATAFRAME MERGE – INNER MERGE

- Merging DataFrames on common customer IDs

Merge on ‘Cust_ID’



```python
pd.merge(df_cust, df_order, on = 'Cust_ID')
```

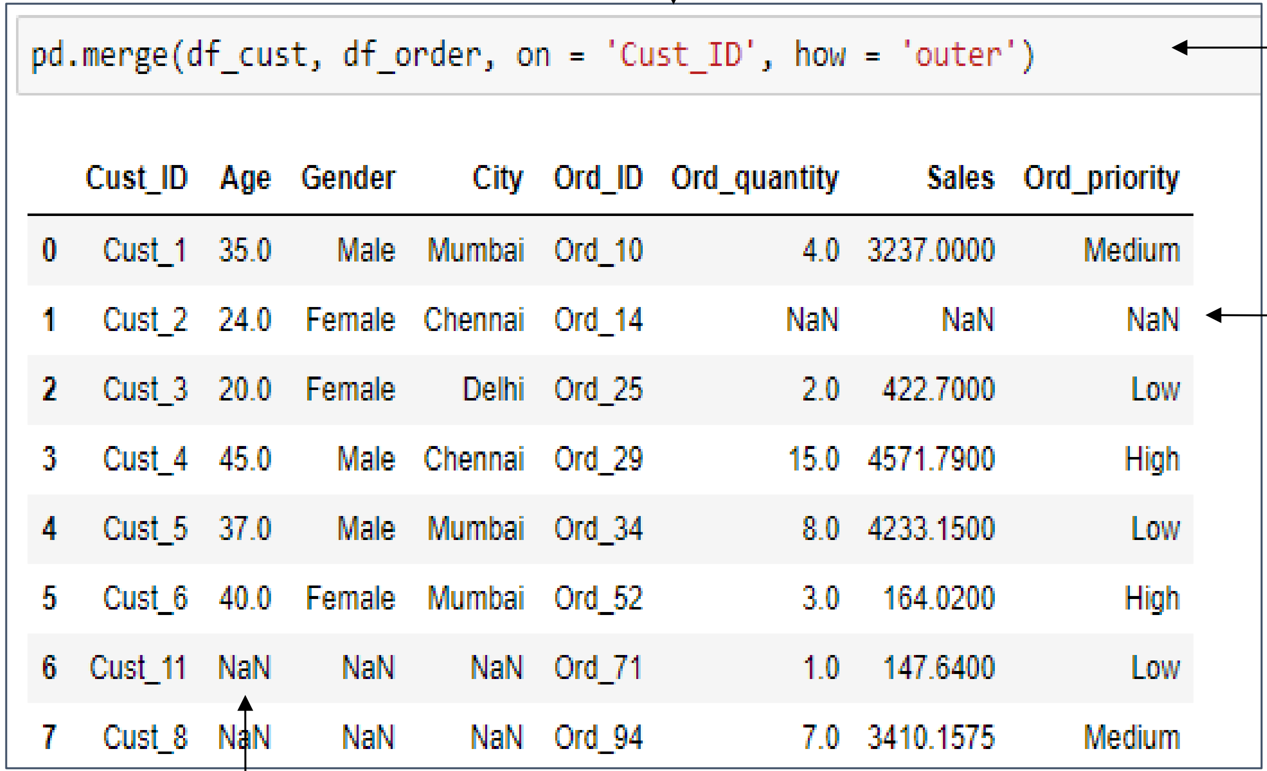
	Cust_ID	Age	Gender	City	Ord_ID	Ord_quantity	Sales	Ord_priority
0	Cust_1	35	Male	Mumbai	Ord_10	4.0	3237.00	Medium
1	Cust_2	24	Female	Chennai	Ord_14	NaN	NaN	NaN
2	Cust_3	20	Female	Delhi	Ord_25	2.0	422.70	Low
3	Cust_4	45	Male	Chennai	Ord_29	15.0	4571.79	High
4	Cust_5	37	Male	Mumbai	Ord_34	8.0	4233.15	Low
5	Cust_6	40	Female	Mumbai	Ord_52	3.0	164.02	High

Merge includes the common IDs in both the DataFrames

NaNs are printed where order details are not available

DATAFRAME MERGE – OUTER MERGE

Merge on ‘Cust_ID’



```
pd.merge(df_cust, df_order, on = 'Cust_ID', how = 'outer')
```

	Cust_ID	Age	Gender	City	Ord_ID	Ord_quantity	Sales	Ord_priority
0	Cust_1	35.0	Male	Mumbai	Ord_10	4.0	3237.0000	Medium
1	Cust_2	24.0	Female	Chennai	Ord_14	NaN	NaN	NaN
2	Cust_3	20.0	Female	Delhi	Ord_25	2.0	422.7000	Low
3	Cust_4	45.0	Male	Chennai	Ord_29	15.0	4571.7900	High
4	Cust_5	37.0	Male	Mumbai	Ord_34	8.0	4233.1500	Low
5	Cust_6	40.0	Female	Mumbai	Ord_52	3.0	164.0200	High
6	Cust_11	NaN	NaN	NaN	Ord_71	1.0	147.6400	Low
7	Cust_8	NaN	NaN	NaN	Ord_94	7.0	3410.1575	Medium

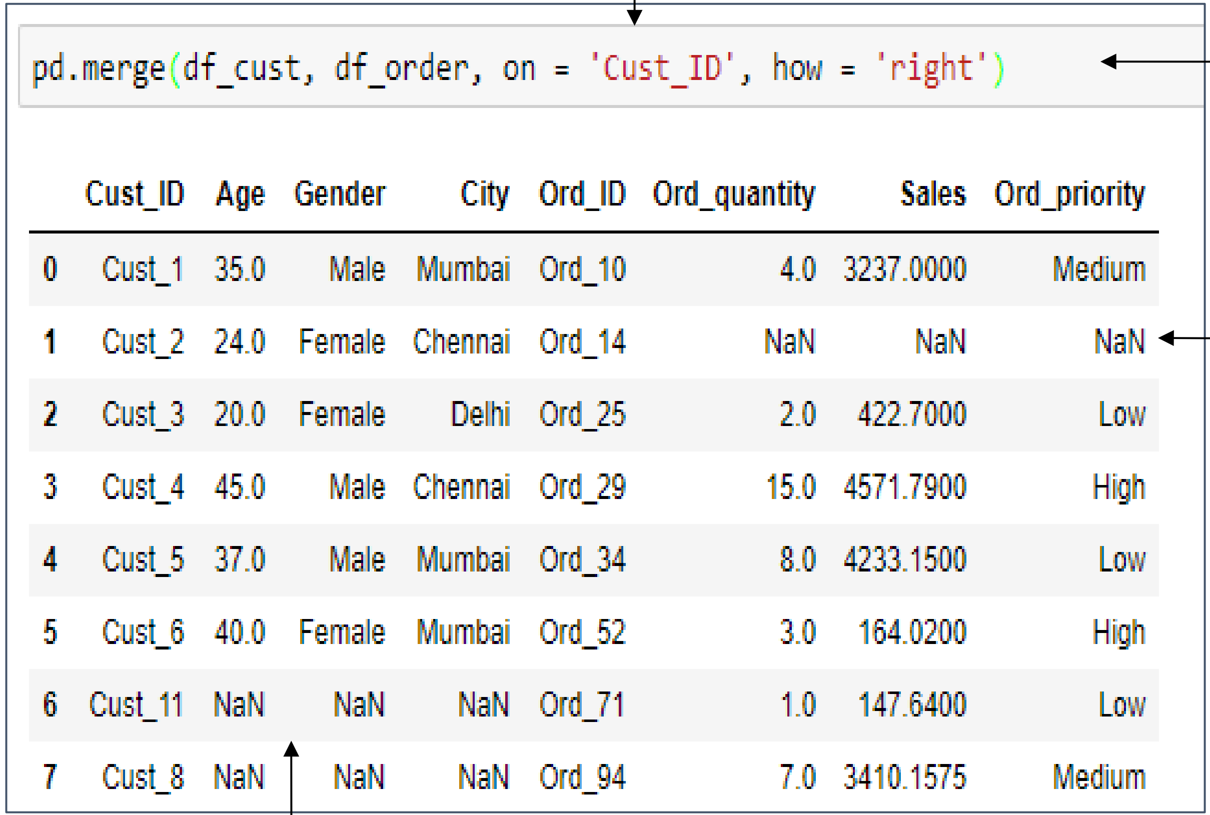
Outer merge includes the IDs in both DataFrames

NaNs are printed where order details are not available

NaNs are printed where customer details are not available

DATAFRAME MERGE – RIGHT MERGE

Merge on ‘Cust_ID’



```
pd.merge(df_cust, df_order, on = 'Cust_ID', how = 'right')
```

	Cust_ID	Age	Gender	City	Ord_ID	Ord_quantity	Sales	Ord_priority
0	Cust_1	35.0	Male	Mumbai	Ord_10	4.0	3237.0000	Medium
1	Cust_2	24.0	Female	Chennai	Ord_14	NaN	NaN	NaN
2	Cust_3	20.0	Female	Delhi	Ord_25	2.0	422.7000	Low
3	Cust_4	45.0	Male	Chennai	Ord_29	15.0	4571.7900	High
4	Cust_5	37.0	Male	Mumbai	Ord_34	8.0	4233.1500	Low
5	Cust_6	40.0	Female	Mumbai	Ord_52	3.0	164.0200	High
6	Cust_11	NaN	NaN	NaN	Ord_71	1.0	147.6400	Low
7	Cust_8	NaN	NaN	NaN	Ord_94	7.0	3410.1575	Medium

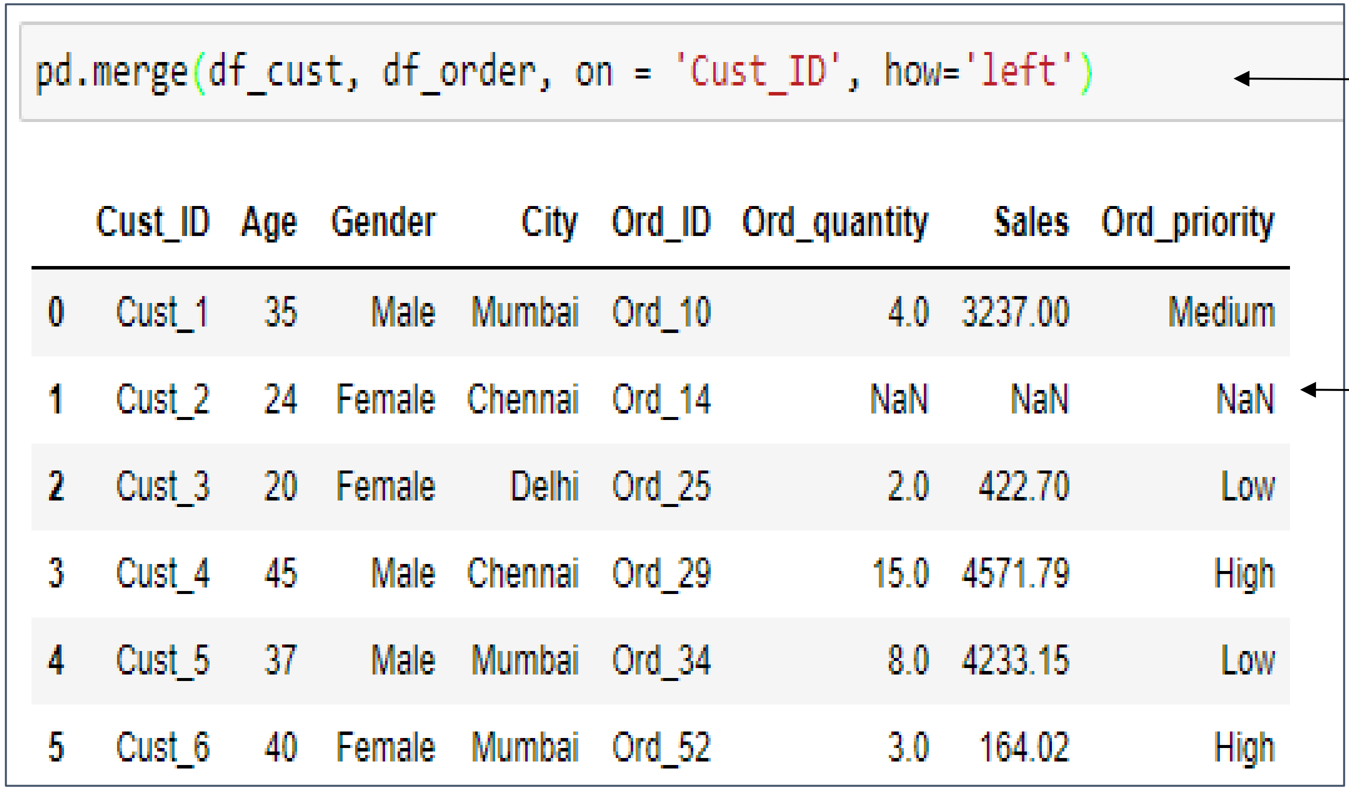
Merge includes all the IDs in ‘df_order’

NaNs are printed where order details are not available

NaNs are printed where customer details are not available

DATAFRAME MERGE – LEFT MERGE

Merge on ‘Cust_ID’



```
pd.merge(df_cust, df_order, on = 'Cust_ID', how='left')
```

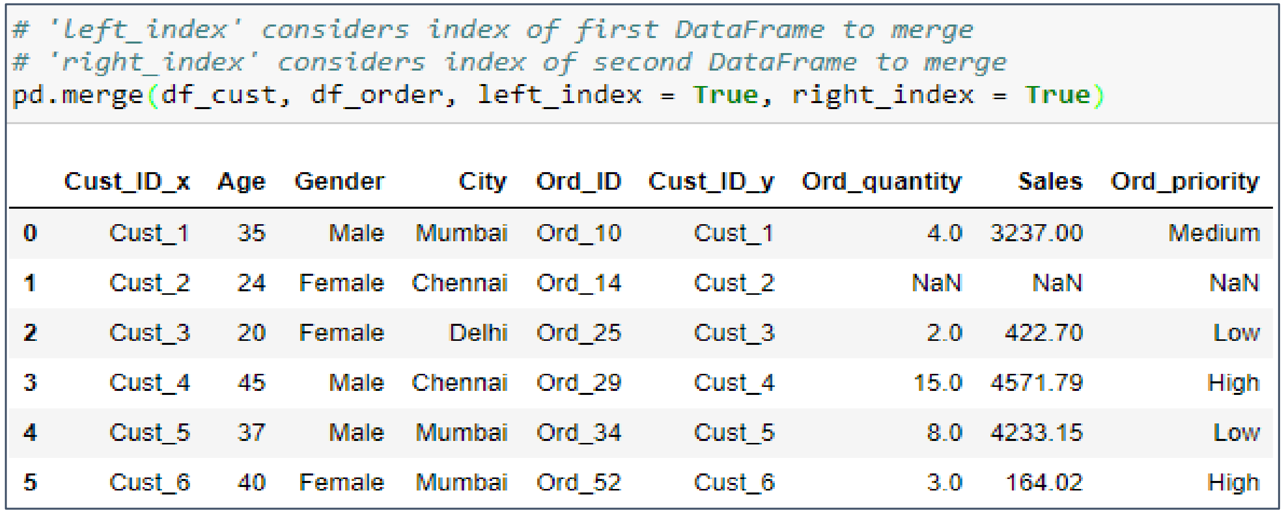
	Cust_ID	Age	Gender	City	Ord_ID	Ord_quantity	Sales	Ord_priority
0	Cust_1	35	Male	Mumbai	Ord_10	4.0	3237.00	Medium
1	Cust_2	24	Female	Chennai	Ord_14	NaN	NaN	NaN
2	Cust_3	20	Female	Delhi	Ord_25	2.0	422.70	Low
3	Cust_4	45	Male	Chennai	Ord_29	15.0	4571.79	High
4	Cust_5	37	Male	Mumbai	Ord_34	8.0	4233.15	Low
5	Cust_6	40	Female	Mumbai	Ord_52	3.0	164.02	High

Merge includes all the IDs in ‘df_cust’

NaNs are printed where order details are not available

DATAFRAME MERGE USING INDEX

- Merged DataFrame has the number of rows equal to that of the minimum of both the DataFrames
- It includes rows from both DataFrames having same index
- This method is useful, only if the record has same index in both the DataFrames



```
# 'left_index' considers index of first DataFrame to merge
# 'right_index' considers index of second DataFrame to merge
pd.merge(df_cust, df_order, left_index = True, right_index = True)
```

	Cust_ID_x	Age	Gender	City	Ord_ID	Cust_ID_y	Ord_quantity	Sales	Ord_priority
0	Cust_1	35	Male	Mumbai	Ord_10	Cust_1	4.0	3237.00	Medium
1	Cust_2	24	Female	Chennai	Ord_14	Cust_2	NaN	NaN	NaN
2	Cust_3	20	Female	Delhi	Ord_25	Cust_3	2.0	422.70	Low
3	Cust_4	45	Male	Chennai	Ord_29	Cust_4	15.0	4571.79	High
4	Cust_5	37	Male	Mumbai	Ord_34	Cust_5	8.0	4233.15	Low
5	Cust_6	40	Female	Mumbai	Ord_52	Cust_6	3.0	164.02	High

MERGE vs JOIN

Merge

- Joins one or more columns of the second DataFrame
- By default, performs ‘inner’ merge
- Returns error if one tries to merge more than two DataFrames simultaneously

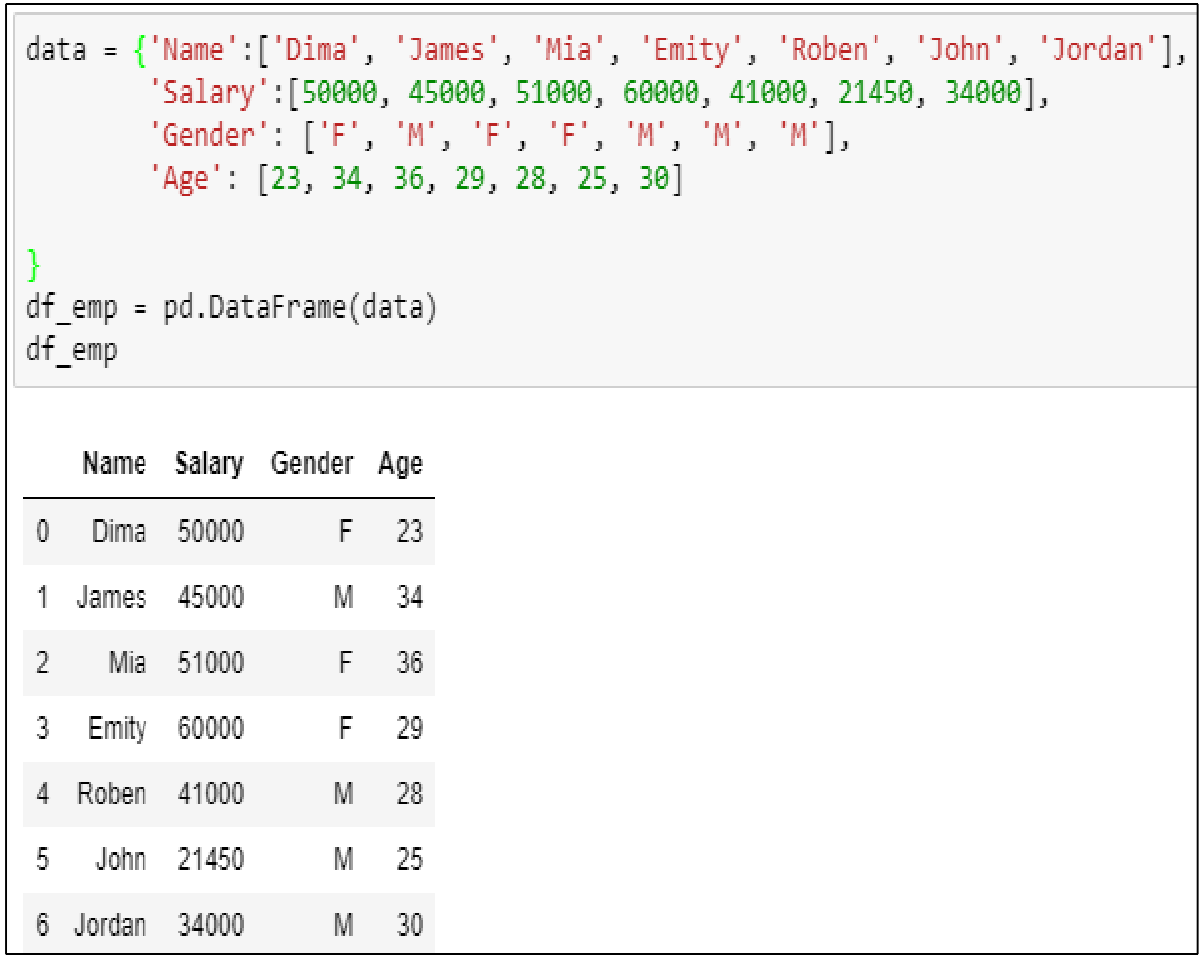
Join

- Joins by the index of the second DataFrame
- By default, performs ‘Left’ join
- Joins multiple DataFrames by index

Reshaping DataFrame

RESHAPING DATAFRAME

- Use the following DataFrames for the study:



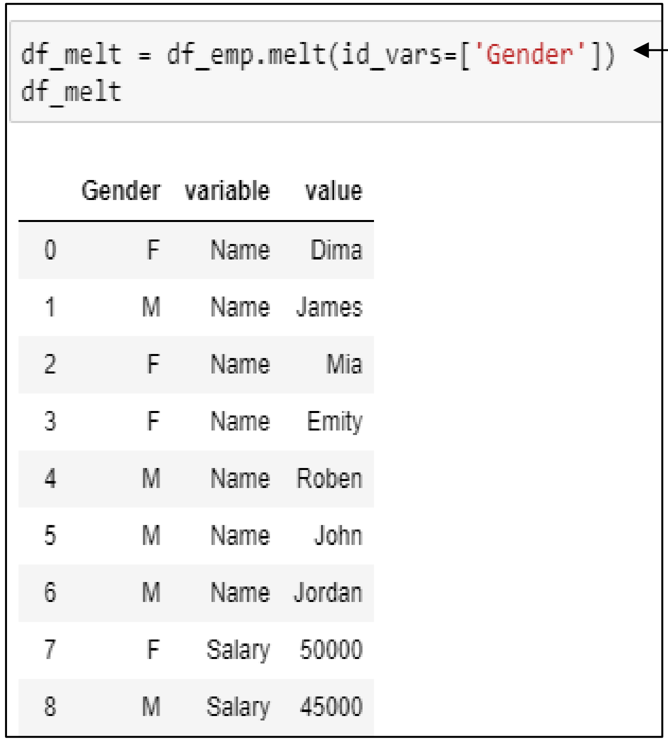
```
data = {'Name':['Dima', 'James', 'Mia', 'Emity', 'Roben', 'John', 'Jordan'],
        'Salary':[50000, 45000, 51000, 60000, 41000, 21450, 34000],
        'Gender': ['F', 'M', 'F', 'F', 'M', 'M', 'M'],
        'Age': [23, 34, 36, 29, 28, 25, 30]

}
df_emp = pd.DataFrame(data)
df_emp
```

	Name	Salary	Gender	Age
0	Dima	50000	F	23
1	James	45000	M	34
2	Mia	51000	F	36
3	Emity	60000	F	29
4	Roben	41000	M	28
5	John	21450	M	25
6	Jordan	34000	M	30

RESHAPING DATAFRAME

- The melt() method is used to change the DataFrame format from wide to long

- The column ‘variable’ contains all the columns except the identifiers and ‘value’ contains the values of corresponding column



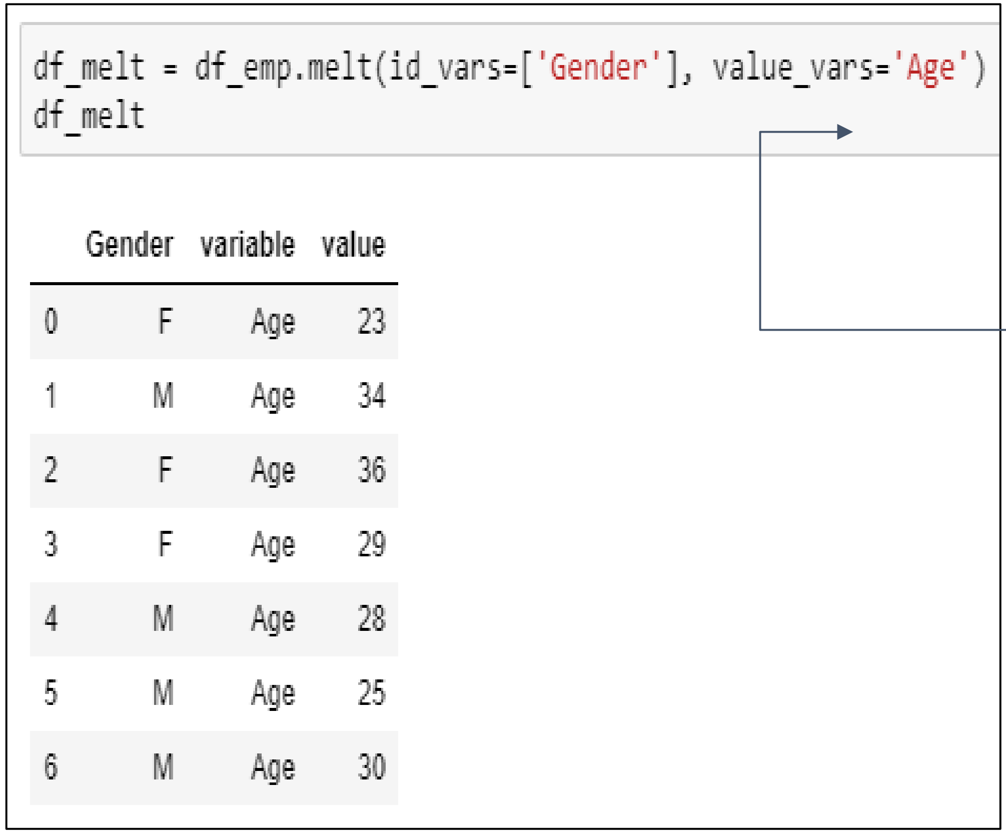
```
df_melt = df_emp.melt(id_vars=['Gender'])
df_melt
```

	Gender	variable	value
0	F	Name	Dima
1	M	Name	James
2	F	Name	Mia
3	F	Name	Emily
4	M	Name	Roben
5	M	Name	John
6	M	Name	Jordan
7	F	Salary	50000
8	M	Salary	45000

You can pass list of columns as identifiers

RESHAPING DATAFRAME

- Assign the variables to the parameter, ‘value_vars’ to get the corresponding values for specified identifiers



```
df_melt = df_emp.melt(id_vars=['Gender'], value_vars='Age')
df_melt
```

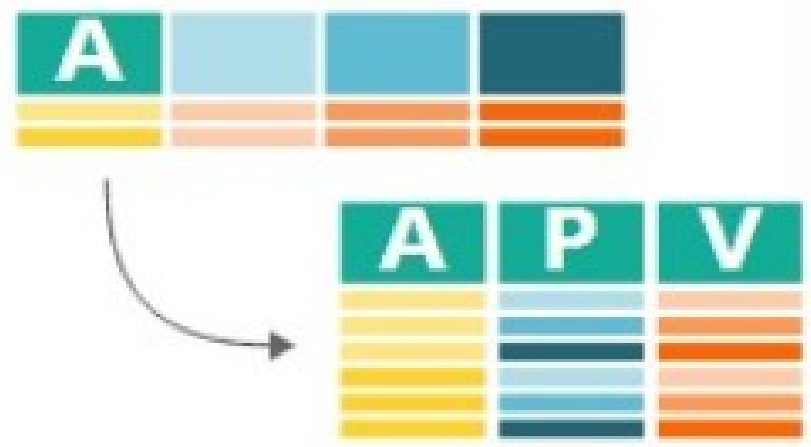
```
   Gender  variable  value
0       F       Age     23
1       M       Age     34
2       F       Age     36
3       F       Age     29
4       M       Age     28
5       M       Age     25
6       M       Age     30
```

Pass the column names to return the corresponding values

Pivot Tables

PIVOT TABLE

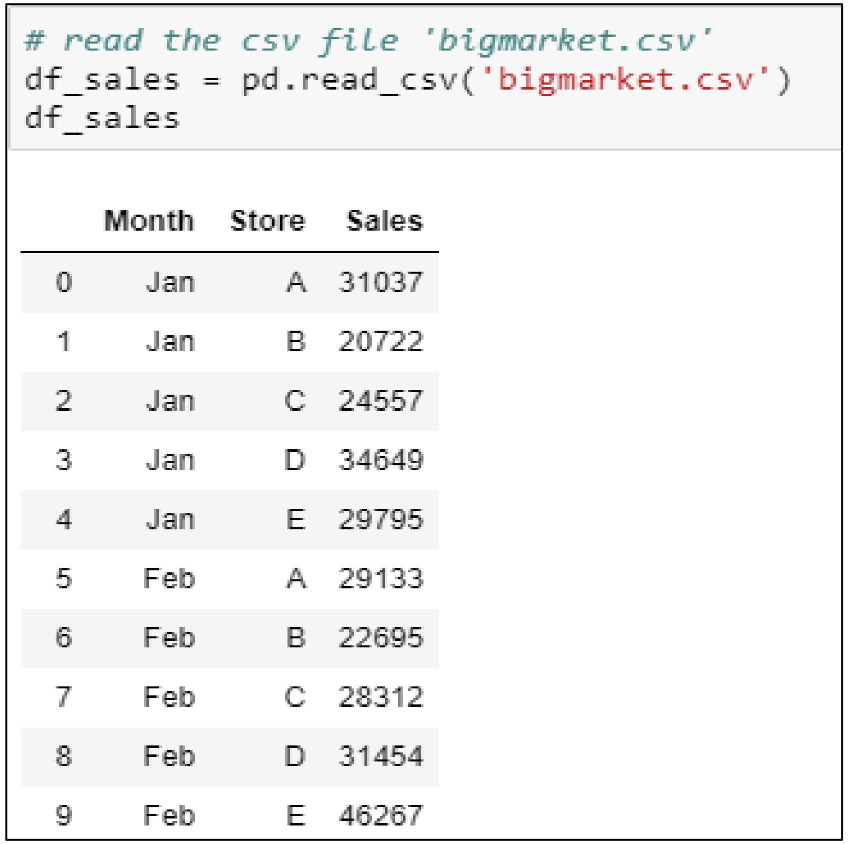
DataFrame like structure



Used to display the data for specified columns and index

READ THE DATAFRAME

- Use the following DataFrame to create a pivot table

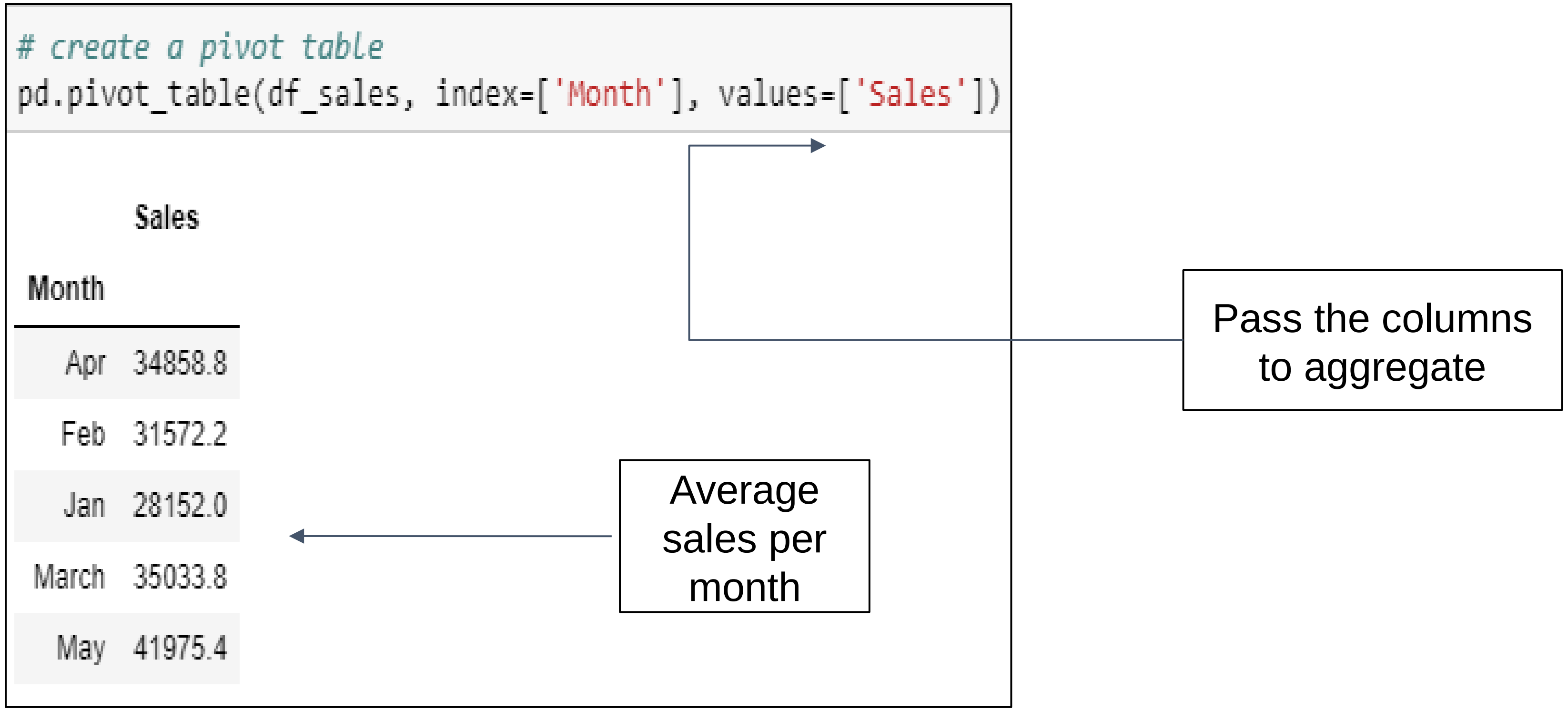


```
# read the csv file 'bigmarket.csv'
df_sales = pd.read_csv('bigmarket.csv')
df_sales
```

	Month	Store	Sales
0	Jan	A	31037
1	Jan	B	20722
2	Jan	C	24557
3	Jan	D	34649
4	Jan	E	29795
5	Feb	A	29133
6	Feb	B	22695
7	Feb	C	28312
8	Feb	D	31454
9	Feb	E	46267

CREATE A PIVOTAL TABLE

- The pivot_table() method generates a pivot table for the given index
- By default, the aggregate function is ‘mean’, which aggregates the columns passed in the parameter, ‘values’



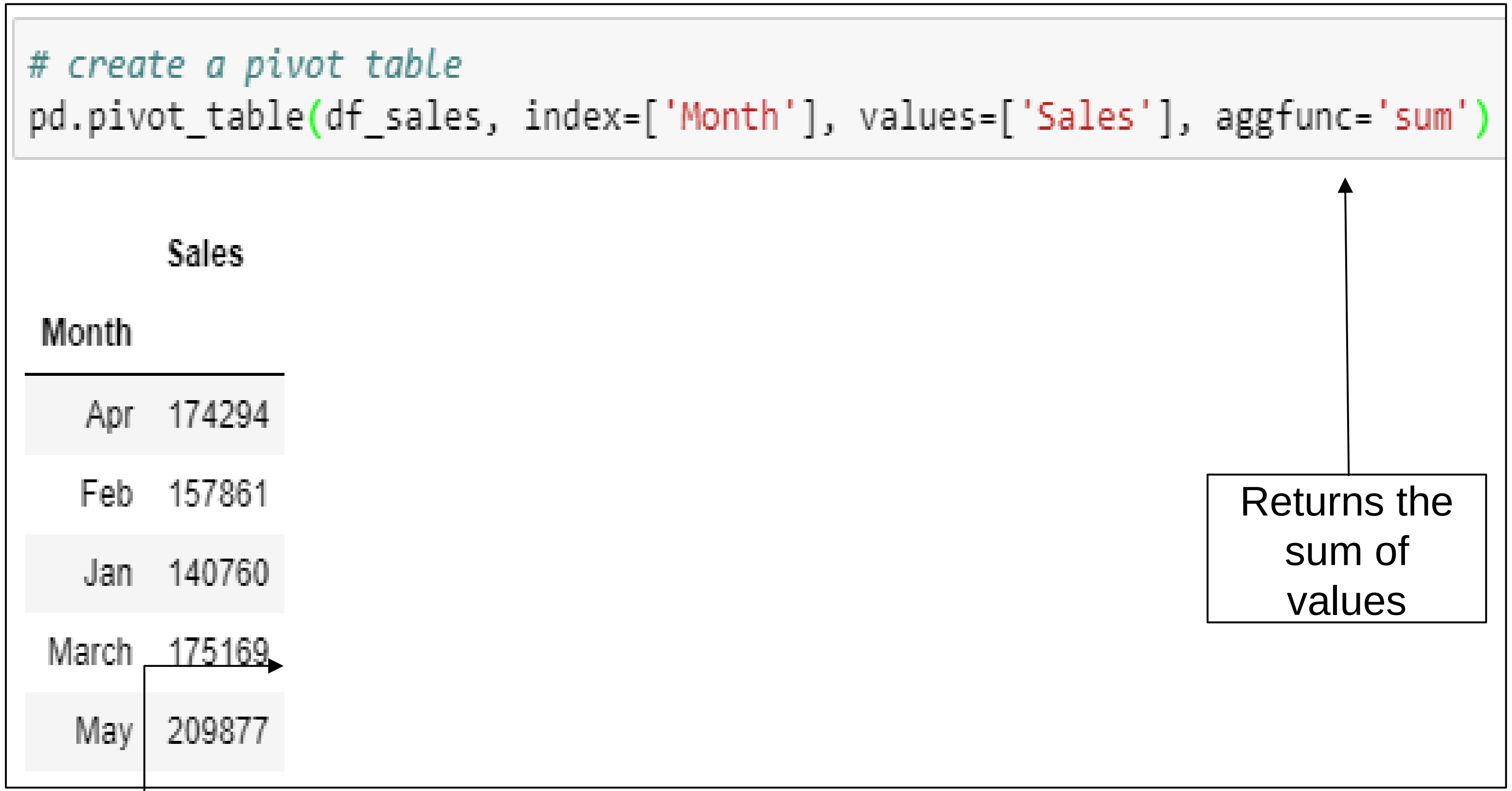
```
# create a pivot table
pd.pivot_table(df_sales, index=['Month'], values=['Sales'])
```

```
         Sales
Month
  Apr  34858.8
  Feb  31572.2
  Jan  28152.0
March  35033.8
  May  41975.4
```

Pass the columns to aggregate

Average sales per month

CREATE A PIVOTAL TABLE



```python
# create a pivot table
pd.pivot_table(df_sales, index=['Month'], values=['Sales'], aggfunc='sum')
```

	Sales
Month	
Apr	174294
Feb	157861
Jan	140760
March	175169
May	209877

Returns the sum of values

Sum of sales per month

Cross Tables

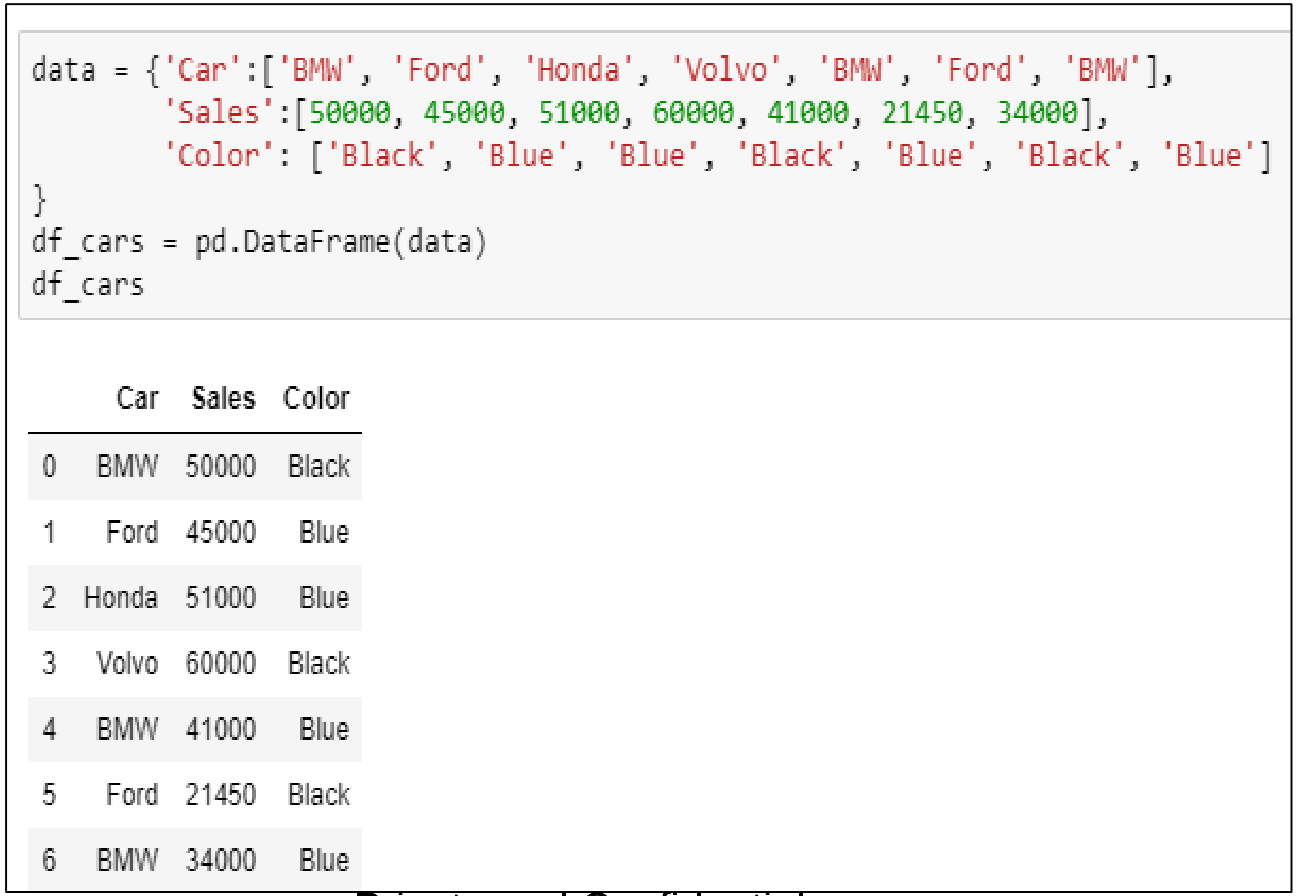
CROSS TABLES

Similar to pivot tables

Computes a cross tabulation of two or more factors

CROSS TABLES

- Use the following DataFrame to create a cross table

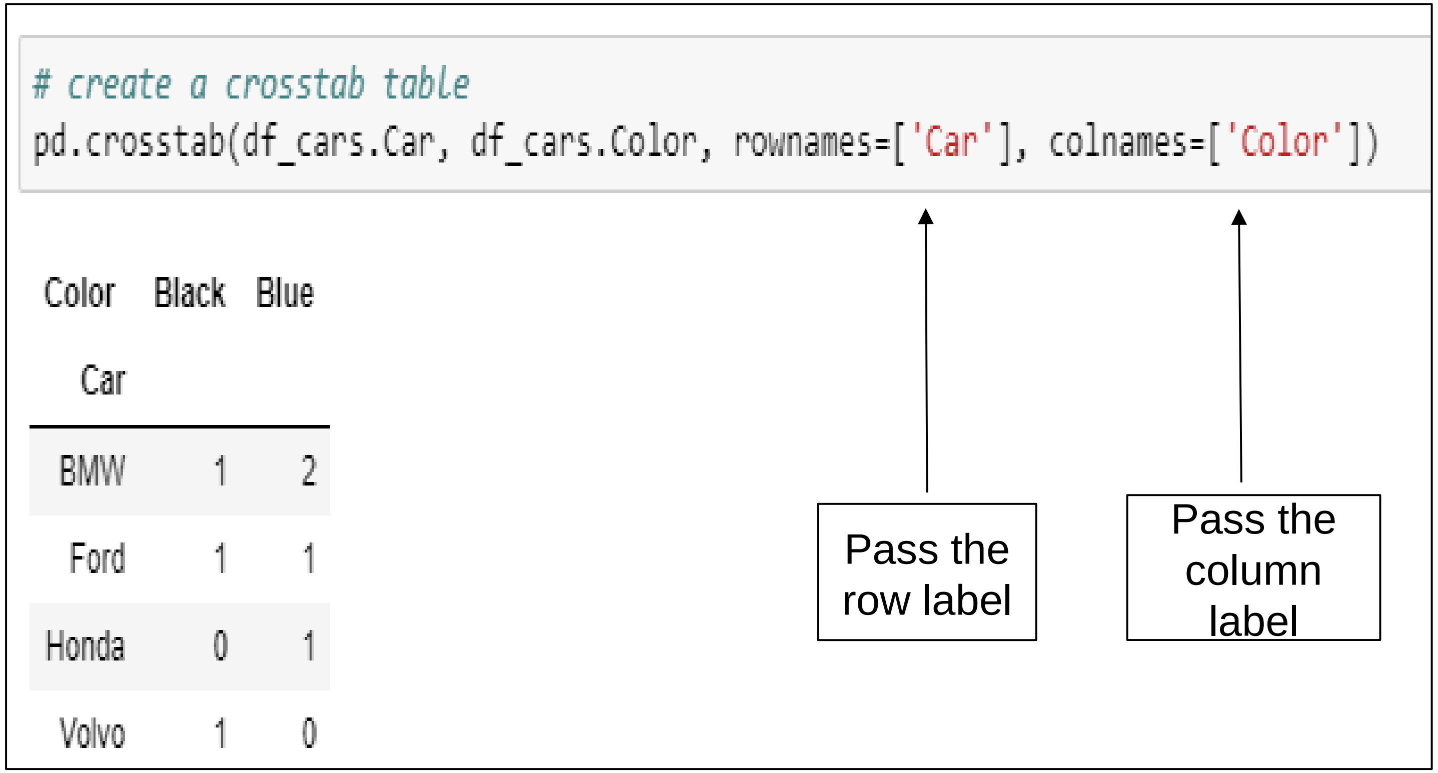


```
data = {'Car':['BMW', 'Ford', 'Honda', 'Volvo', 'BMW', 'Ford', 'BMW'],
        'Sales':[50000, 45000, 51000, 60000, 41000, 21450, 34000],
        'Color': ['Black', 'Blue', 'Blue', 'Black', 'Blue', 'Black', 'Blue']
}
df_cars = pd.DataFrame(data)
df_cars
```

	Car	Sales	Color
0	BMW	50000	Black
1	Ford	45000	Blue
2	Honda	51000	Blue
3	Volvo	60000	Black
4	BMW	41000	Blue
5	Ford	21450	Black
6	BMW	34000	Blue

CREATING CROSS TABLES

- Find the color-wise car count using the crosstab() method



```
# create a crosstab table
pd.crosstab(df_cars.Car, df_cars.Color, rownames=['Car'], colnames=['Color'])
```

```
Color  Black  Blue
  Car
  BMW      1     2
 Ford      1     1
Honda      0     1
Volvo      1     0
```

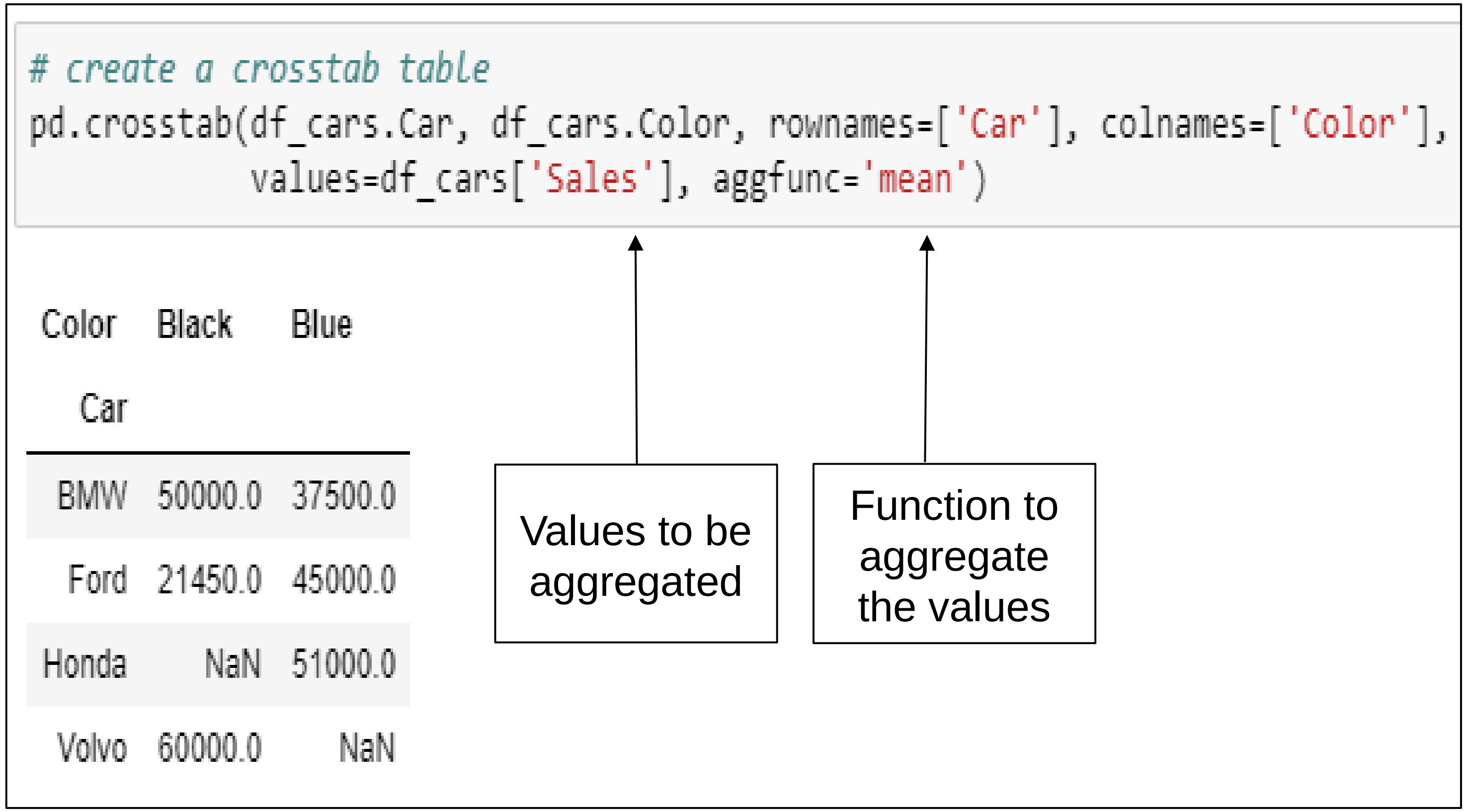
Pass the row label

Pass the column label

By default, the crosstab() method returns the frequency table of the variables

CREATING CROSS TABLES

- Find the color-wise distribution of sales for different cars



```
# create a crosstab table
pd.crosstab(df_cars.Car, df_cars.Color, rownames=['Car'], colnames=['Color'],
            values=df_cars['Sales'], aggfunc='mean')
```

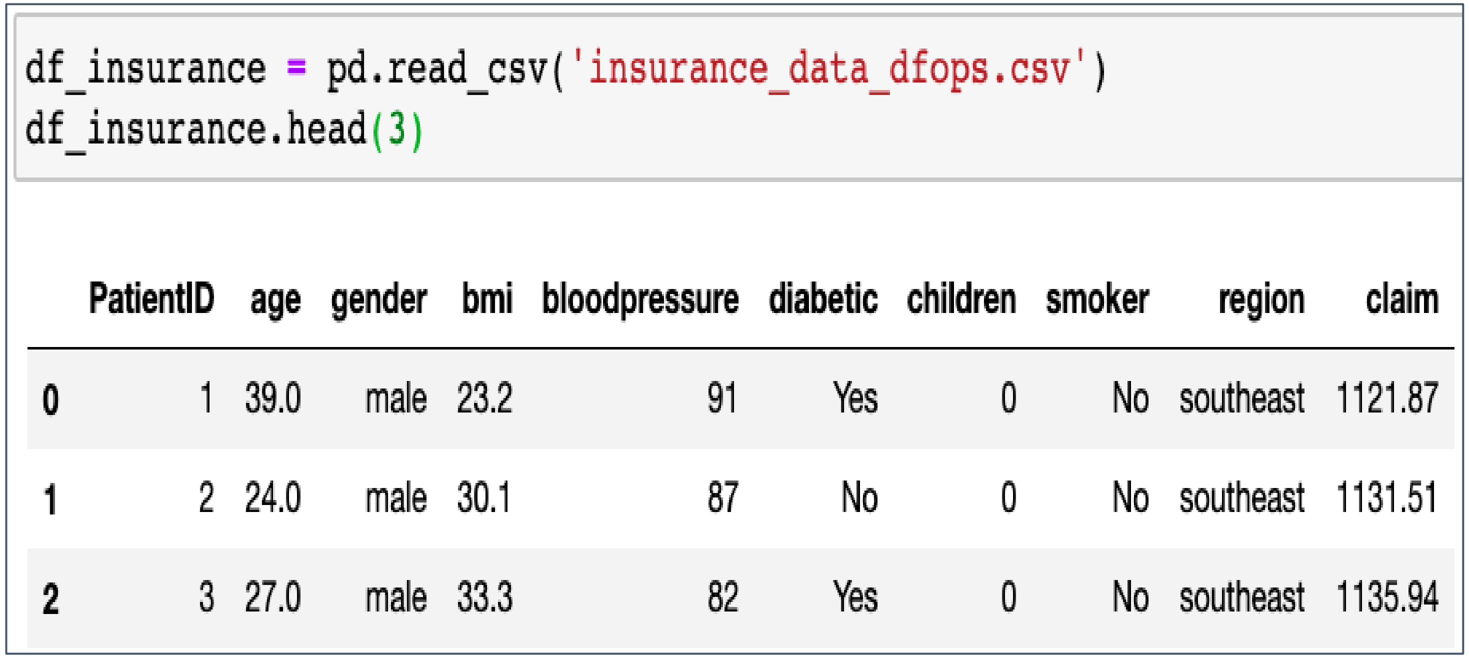
Color	Black	Blue
Car		
BMW	50000.0	37500.0
Ford	21450.0	45000.0
Honda	NaN	51000.0
Volvo	60000.0	NaN

Values to be aggregated

Function to aggregate the values

DataFrame Operations

READ THE DATA



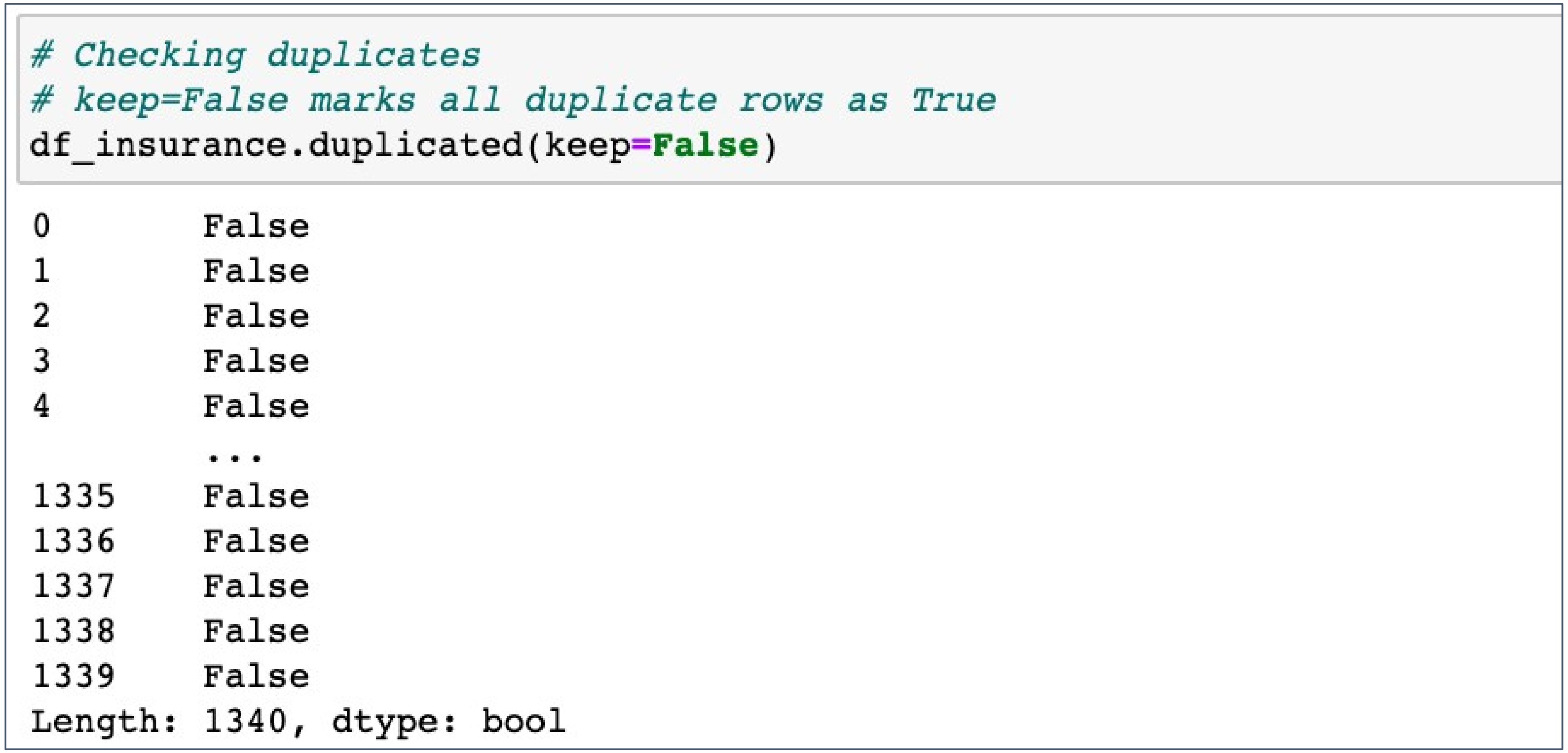
```
df_insurance = pd.read_csv('insurance_data_dfops.csv')
df_insurance.head(3)
```

	PatientID	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
0	1	39.0	male	23.2	91	Yes	0	No	southeast	1121.87
1	2	24.0	male	30.1	87	No	0	No	southeast	1131.51
2	3	27.0	male	33.3	82	Yes	0	No	southeast	1135.94

Checking Duplicates

CHECKING DUPLICATES

- Check the duplicate observations using the duplicated() method

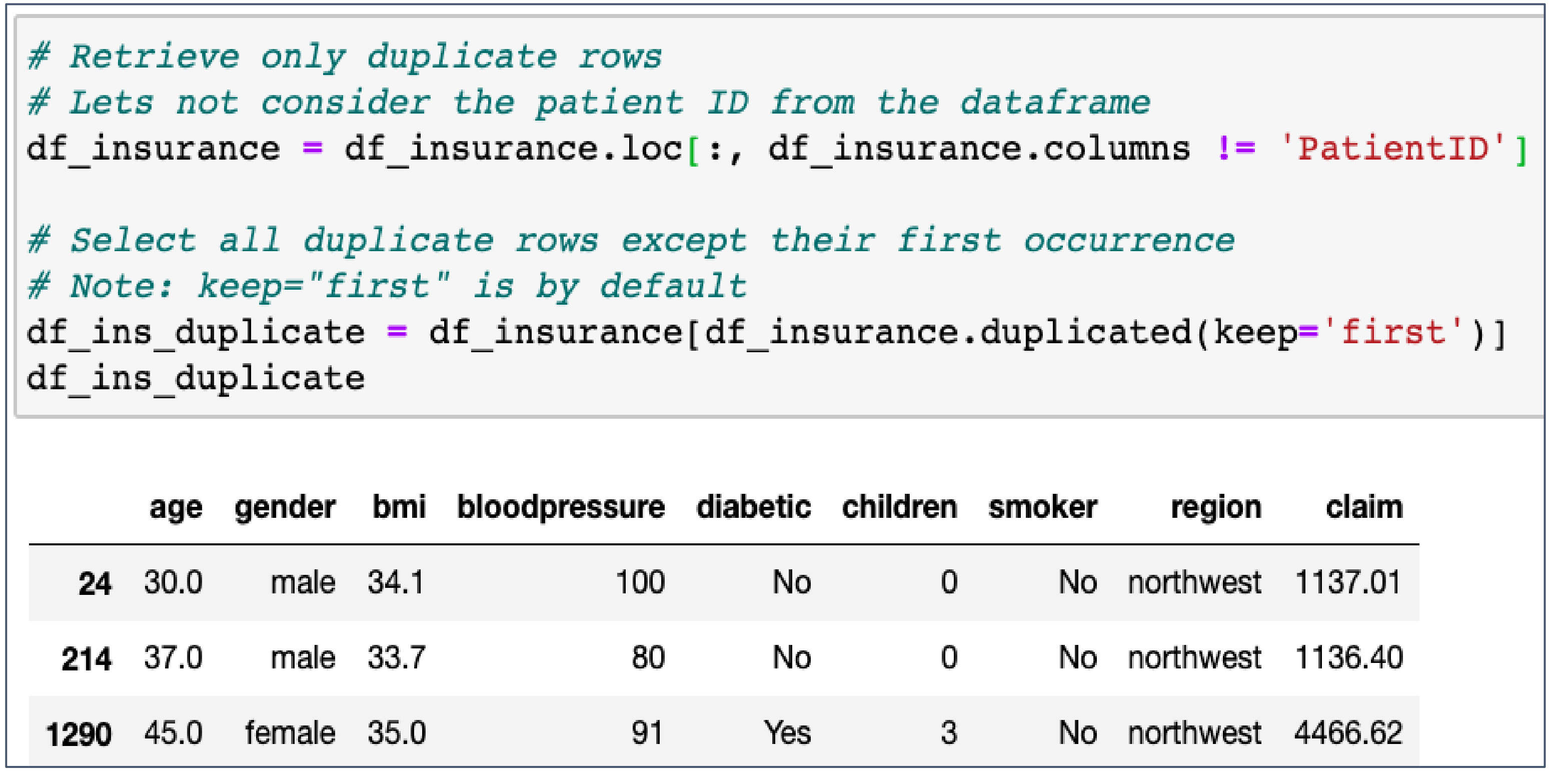


```
# Checking duplicates
# keep=False marks all duplicate rows as True
df_insurance.duplicated(keep=False)
```

```
0       False
1       False
2       False
3       False
4       False
        ...
1335    False
1336    False
1337    False
1338    False
1339    False
Length: 1340, dtype: bool
```

CHECKING DUPLICATES

- Find duplicate rows based on all columns
- The parameter, keep=”first”, will select all duplicate rows except their 1st occurence

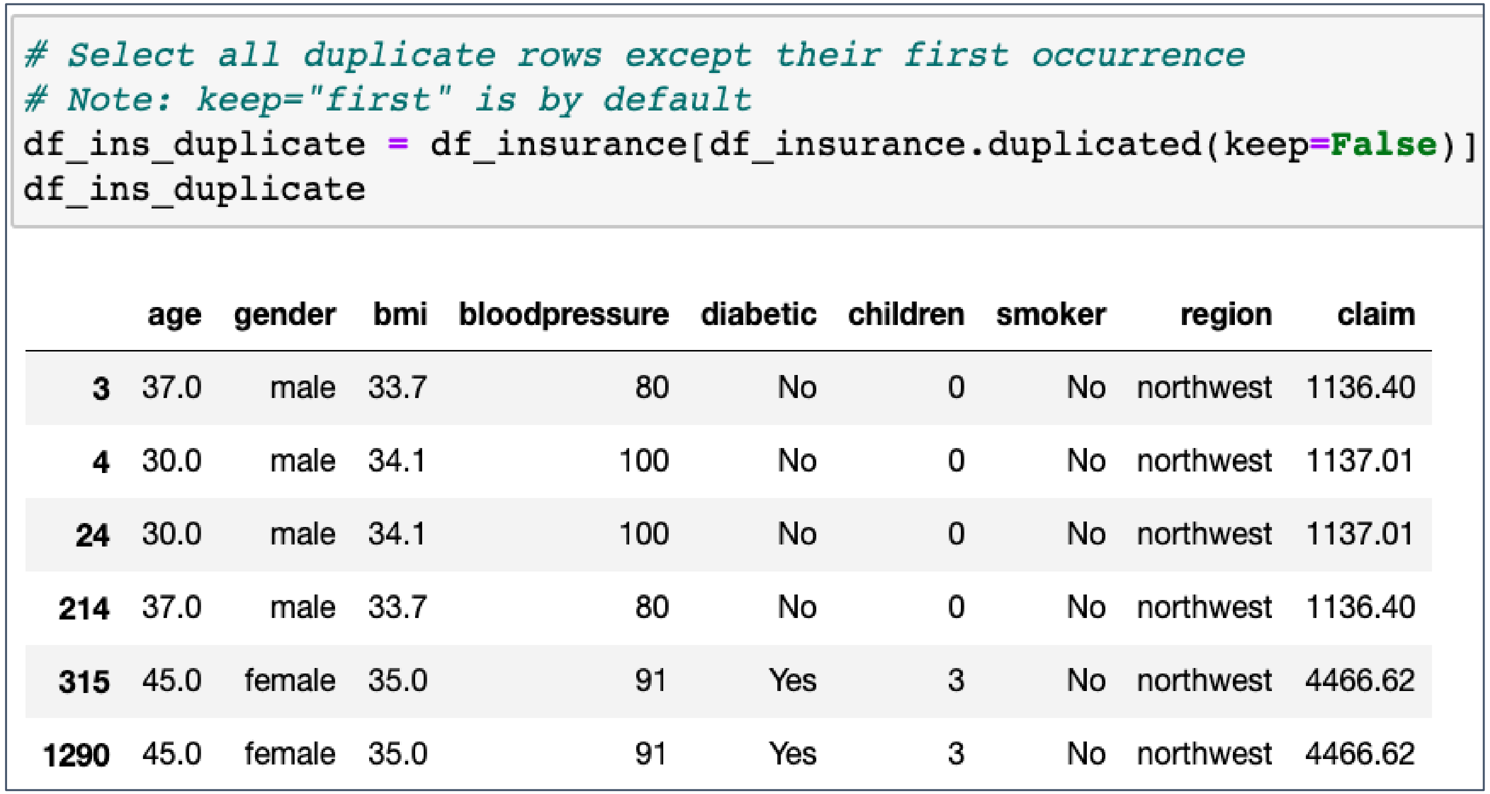


```
# Retrieve only duplicate rows
# Lets not consider the patient ID from the dataframe
df_insurance = df_insurance.loc[:, df_insurance.columns != 'PatientID']

# Select all duplicate rows except their first occurrence
# Note: keep="first" is by default
df_ins_duplicate = df_insurance[df_insurance.duplicated(keep='first')]
df_ins_duplicate
```

	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
24	30.0	male	34.1	100	No	0	No	northwest	1137.01
214	37.0	male	33.7	80	No	0	No	northwest	1136.40
1290	45.0	female	35.0	91	Yes	3	No	northwest	4466.62

CHECKING DUPLICATES

- Find duplicate rows based on all columns
- The parameter, keep=False, will select all duplicate rows

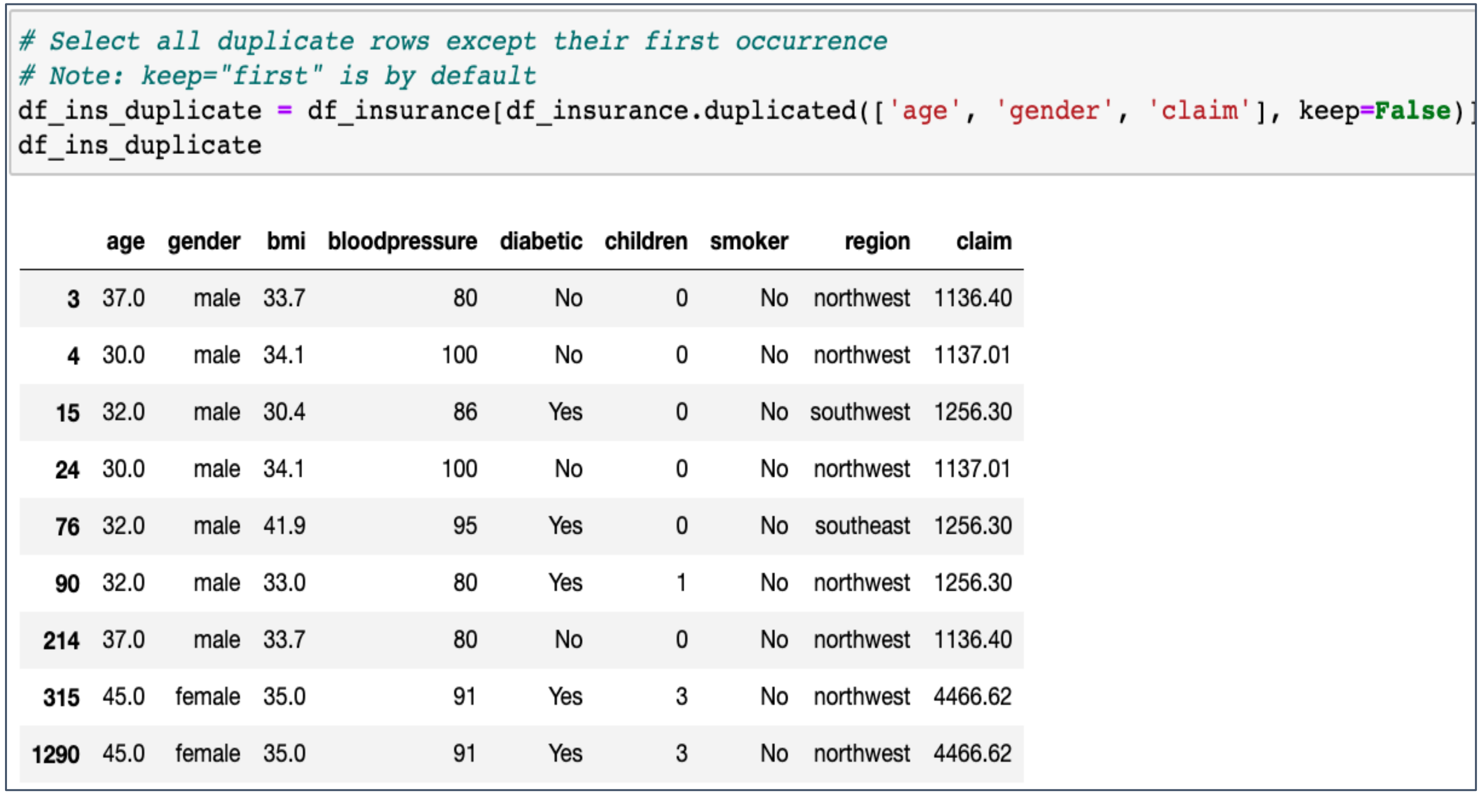


```
# Select all duplicate rows except their first occurrence
# Note: keep="first" is by default
df_ins_duplicate = df_insurance[df_insurance.duplicated(keep=False)]
df_ins_duplicate
```

	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
3	37.0	male	33.7	80	No	0	No	northwest	1136.40
4	30.0	male	34.1	100	No	0	No	northwest	1137.01
24	30.0	male	34.1	100	No	0	No	northwest	1137.01
214	37.0	male	33.7	80	No	0	No	northwest	1136.40
315	45.0	female	35.0	91	Yes	3	No	northwest	4466.62
1290	45.0	female	35.0	91	Yes	3	No	northwest	4466.62

CHECKING DUPLICATES

- Find duplicate rows based on selected columns
- The parameter, keep=False, will select all duplicate rows

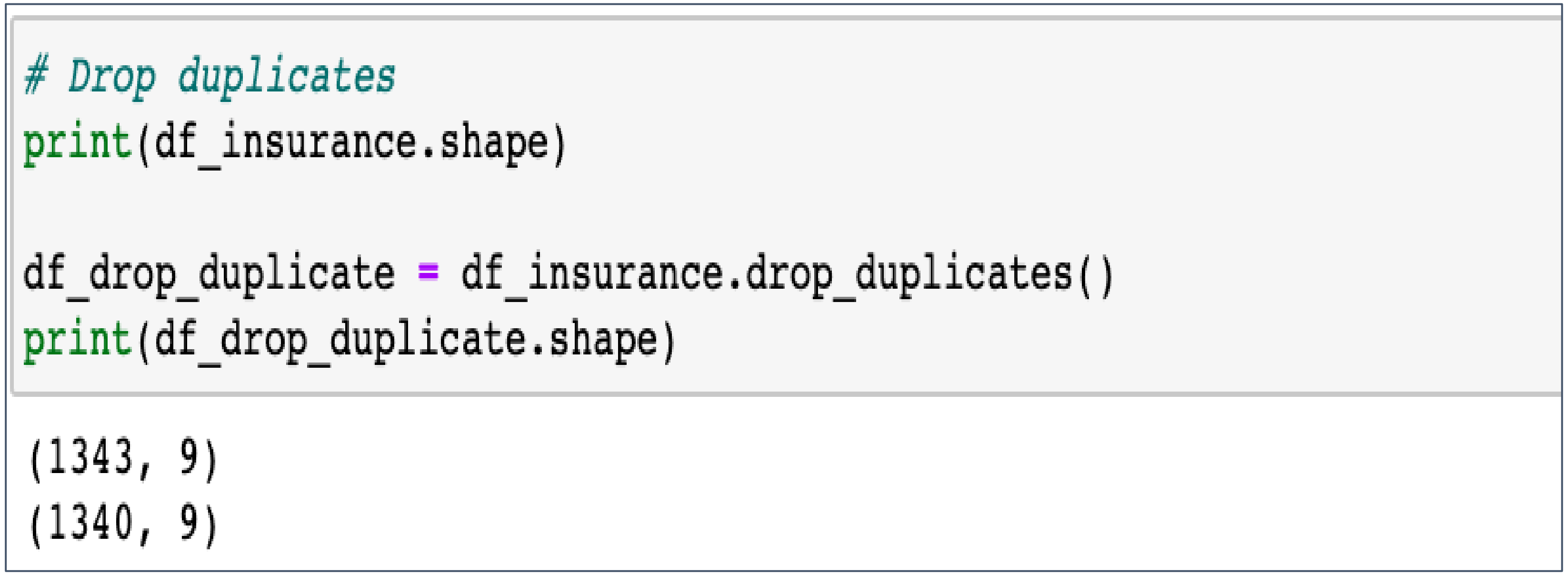


```
# Select all duplicate rows except their first occurrence
# Note: keep="first" is by default
df_ins_duplicate = df_insurance[df_insurance.duplicated(['age', 'gender', 'claim'], keep=False)]
df_ins_duplicate
```

	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
3	37.0	male	33.7	80	No	0	No	northwest	1136.40
4	30.0	male	34.1	100	No	0	No	northwest	1137.01
15	32.0	male	30.4	86	Yes	0	No	southwest	1256.30
24	30.0	male	34.1	100	No	0	No	northwest	1137.01
76	32.0	male	41.9	95	Yes	0	No	southeast	1256.30
90	32.0	male	33.0	80	Yes	1	No	northwest	1256.30
214	37.0	male	33.7	80	No	0	No	northwest	1136.40
315	45.0	female	35.0	91	Yes	3	No	northwest	4466.62
1290	45.0	female	35.0	91	Yes	3	No	northwest	4466.62

DROP DUPLICATES

- Use the drop_duplicates() method to drop all duplicate rows where all columns match

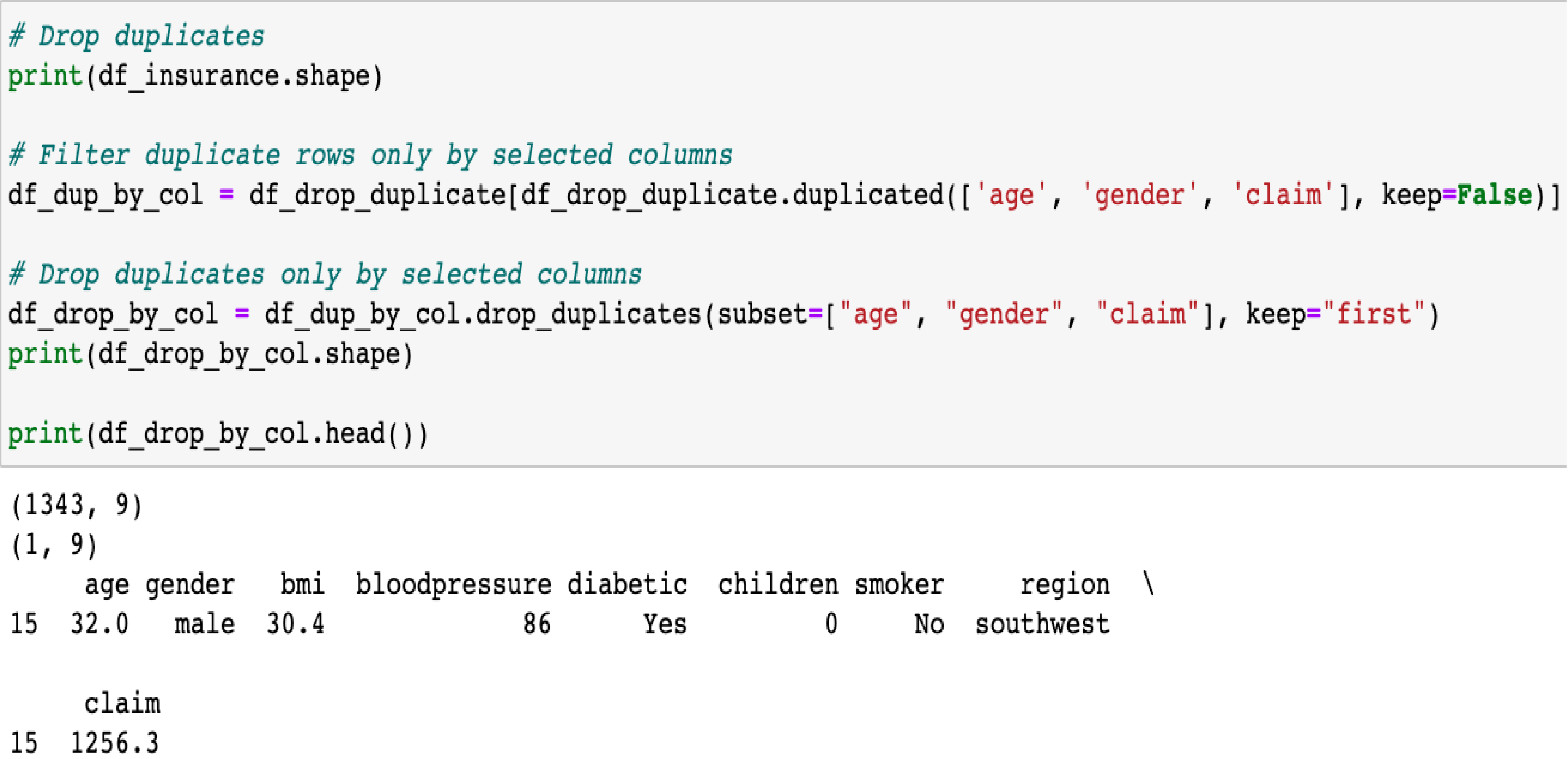


```
# Drop duplicates
print(df_insurance.shape)

df_drop_duplicate = df_insurance.drop_duplicates()
print(df_drop_duplicate.shape)
```

```
(1343, 9)
(1340, 9)
```

DROP DUPLICATES

- Use the drop_duplicates() method to drop all duplicate rows based on individual or list of columns

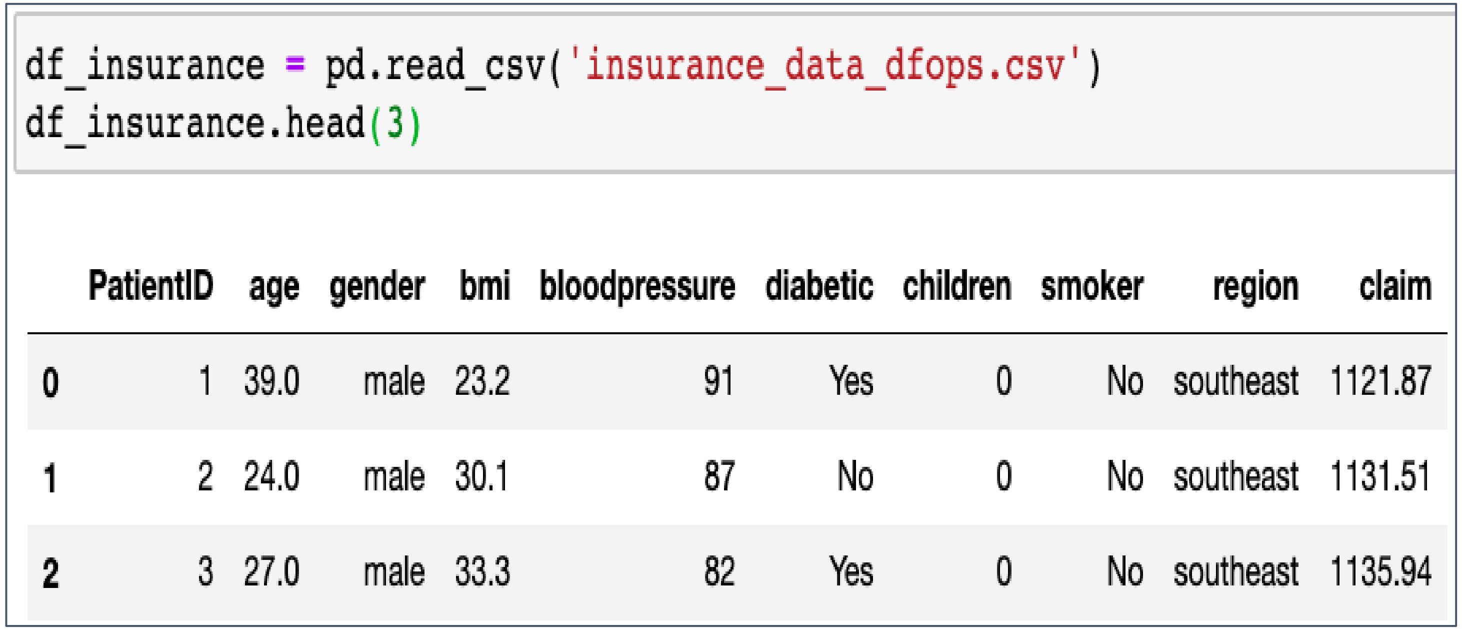


```
# Drop duplicates
print(df_insurance.shape)

# Filter duplicate rows only by selected columns
df_dup_by_col = df_drop_duplicate[df_drop_duplicate.duplicated(['age', 'gender', 'claim'], keep=False)]

# Drop duplicates only by selected columns
df_drop_by_col = df_dup_by_col.drop_duplicates(subset=["age", "gender", "claim"], keep="first")
print(df_drop_by_col.shape)

print(df_drop_by_col.head())
```

```
(1343, 9)
(1, 9)
     age gender   bmi  bloodpressure diabetic  children smoker     region  \
15  32.0   male  30.4             86      Yes         0     No  southwest

     claim
15  1256.3
```

Dropping Rows and Columns

READ THA DATA



```python
df_insurance = pd.read_csv('insurance_data_dfops.csv')
df_insurance.head(3)
```

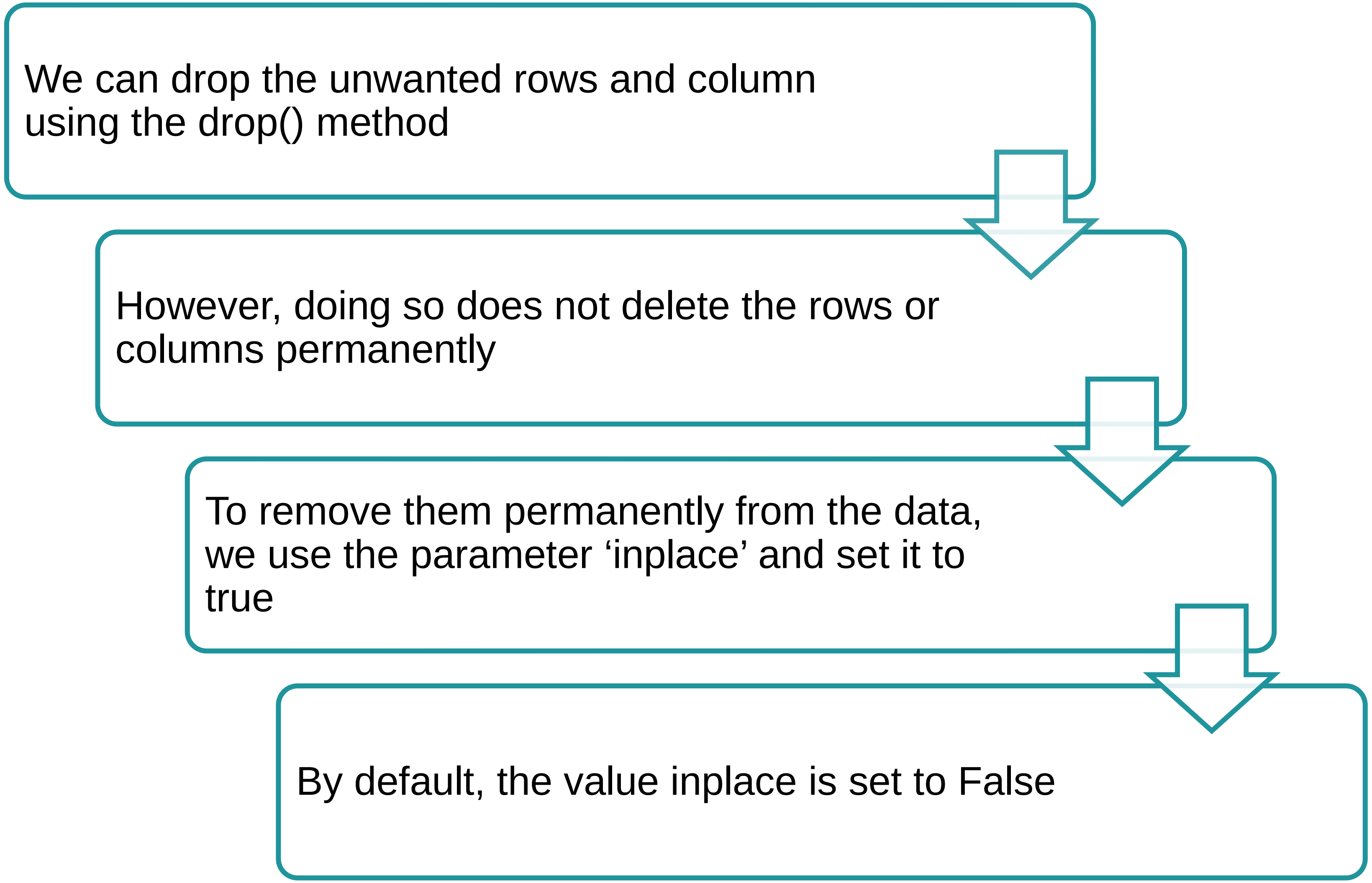
	PatientID	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
0	1	39.0	male	23.2	91	Yes	0	No	southeast	1121.87
1	2	24.0	male	30.1	87	No	0	No	southeast	1131.51
2	3	27.0	male	33.3	82	Yes	0	No	southeast	1135.94

DROPPING ROWS AND COLUMNS FROM DATAFRAME

The drop() method is used to drop the rows and columns that are not required for the analysis

There are scenarios where we need to drop certain rows and/or columns which have missing values, or are redundant with respect to our analysis

UNDERSTANDING THE ‘INPLACE’ PARAMETER



We can drop the unwanted rows and column using the drop() method

However, doing so does not delete the rows or columns permanently

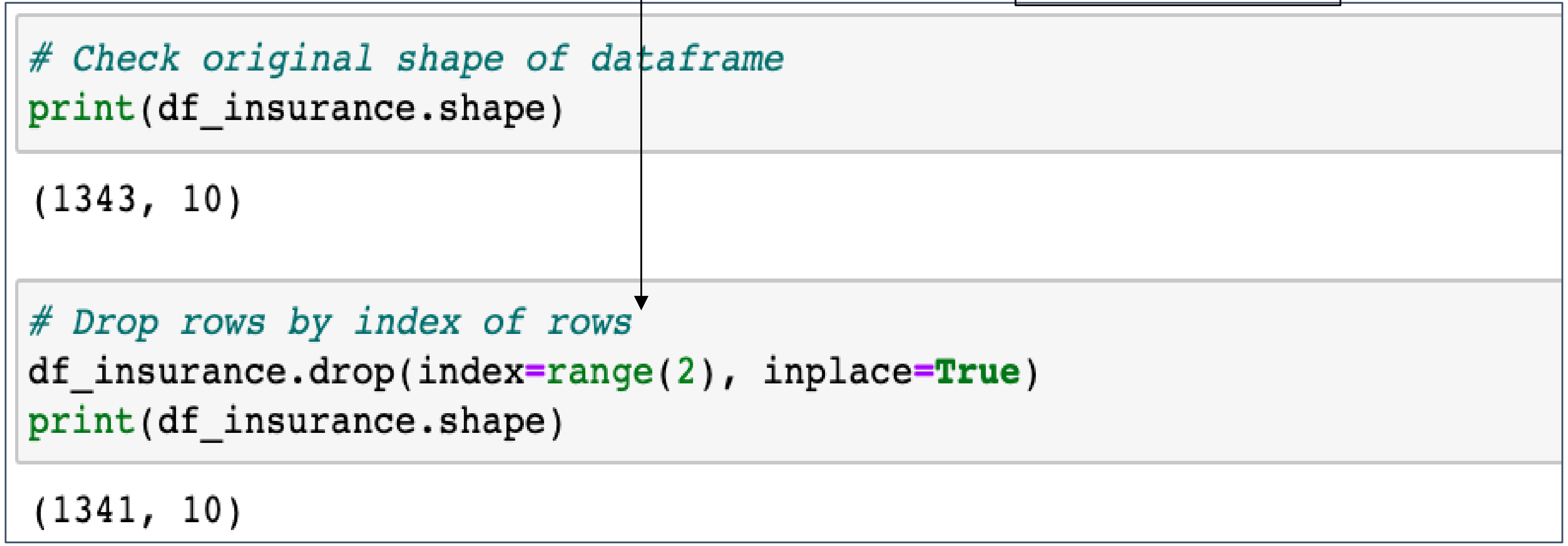
To remove them permanently from the data, we use the parameter ‘inplace’ and set it to true

By default, the value inplace is set to False

DROPPING ROWS

- The drop() method is used to drop the rows with index values
- Here ‘range(2)’ is used to drop the first two rows

Pass the row indices to ‘index’



```
# Check original shape of dataframe
print(df_insurance.shape)
```

```
(1343, 10)
```

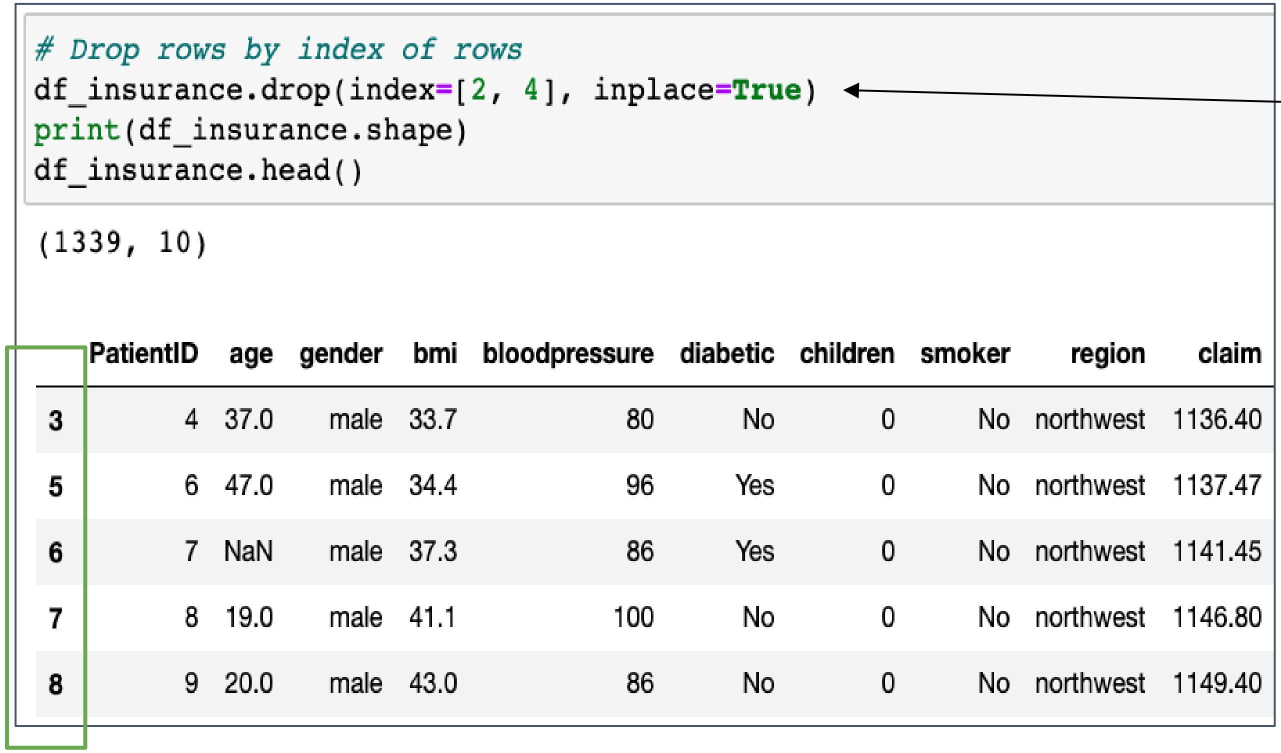
```
# Drop rows by index of rows
df_insurance.drop(index=range(2), inplace=True)
print(df_insurance.shape)
```

```
(1341, 10)
```

Note: The rows with index 0 & 1 get removed. The index for the remaining rows remain unchanged

DROPPING ROWS

- Here index=[2, 4] is used to drop the rows with index 2 & 4



```
# Drop rows by index of rows
df_insurance.drop(index=[2, 4], inplace=True)
print(df_insurance.shape)
df_insurance.head()
```

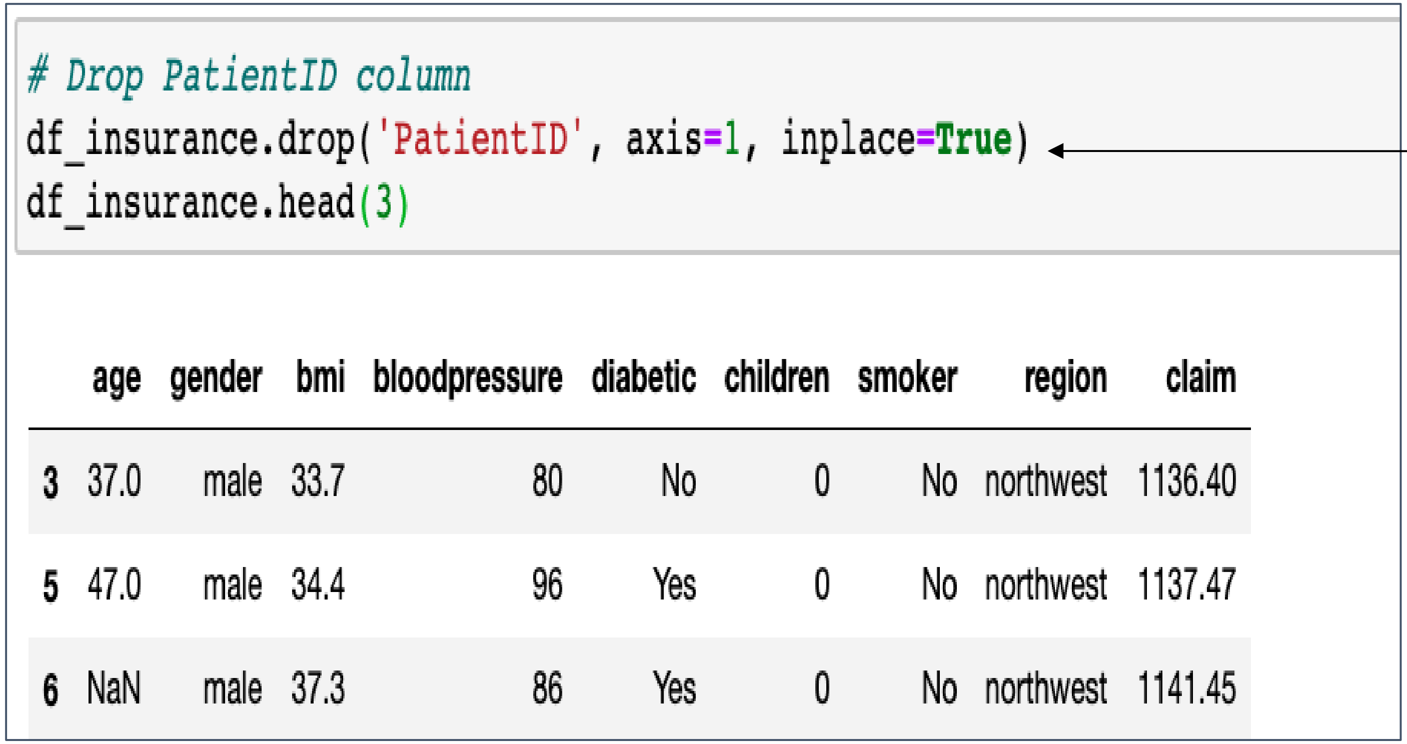
```
(1339, 10)
```

	PatientID	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
3	4	37.0	male	33.7	80	No	0	No	northwest	1136.40
5	6	47.0	male	34.4	96	Yes	0	No	northwest	1137.47
6	7	NaN	male	37.3	86	Yes	0	No	northwest	1141.45
7	8	19.0	male	41.1	100	No	0	No	northwest	1146.80
8	9	20.0	male	43.0	86	No	0	No	northwest	1149.40

Pass the list of row indices to drop the rows

DROPPING COLUMNS

- Drop columns by column name



```python
# Drop PatientID column
df_insurance.drop('PatientID', axis=1, inplace=True)
df_insurance.head(3)
```

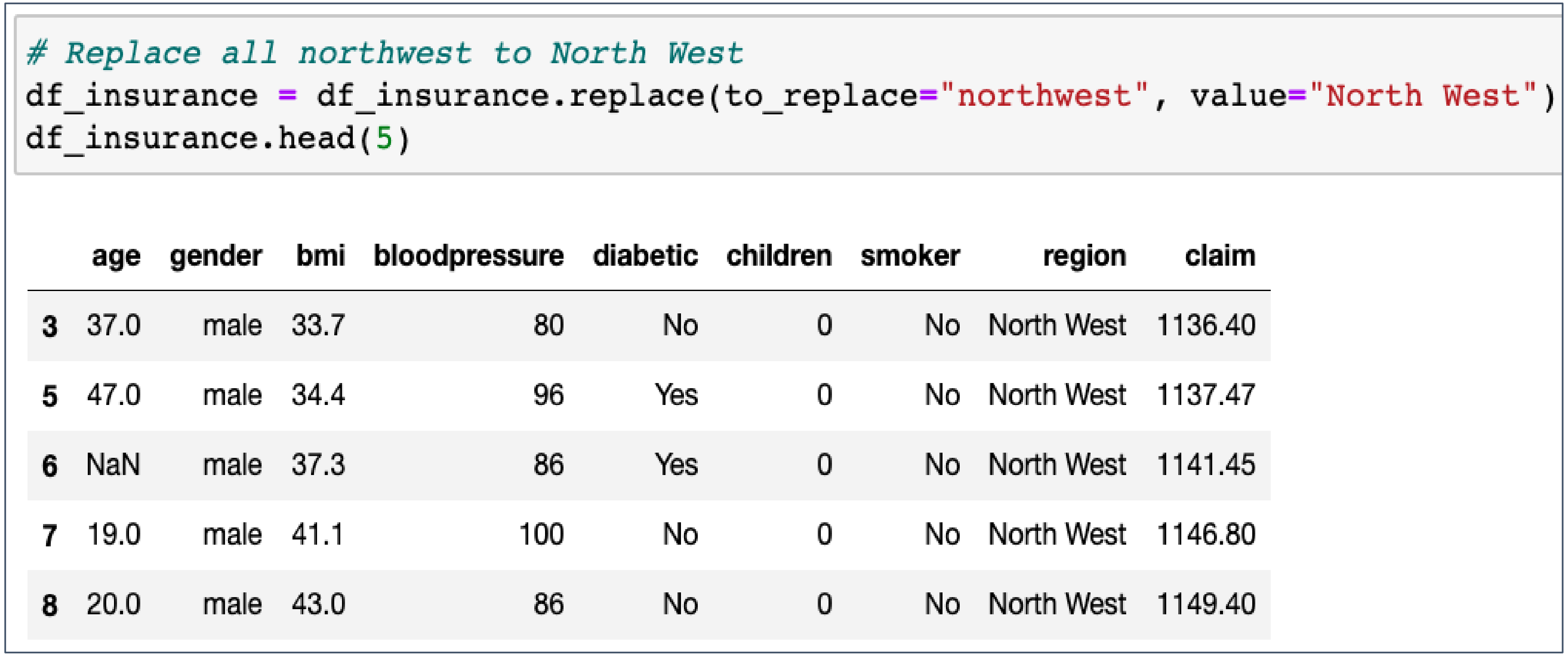
	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
3	37.0	male	33.7	80	No	0	No	northwest	1136.40
5	47.0	male	34.4	96	Yes	0	No	northwest	1137.47
6	NaN	male	37.3	86	Yes	0	No	northwest	1141.45

Pass the column name while setting axis = 1 to drop the column by name

Replacing Values

REPLACE THE VALUES

- The replace() method is used to replace the values in the DataFrame
- Note: No column name is passed

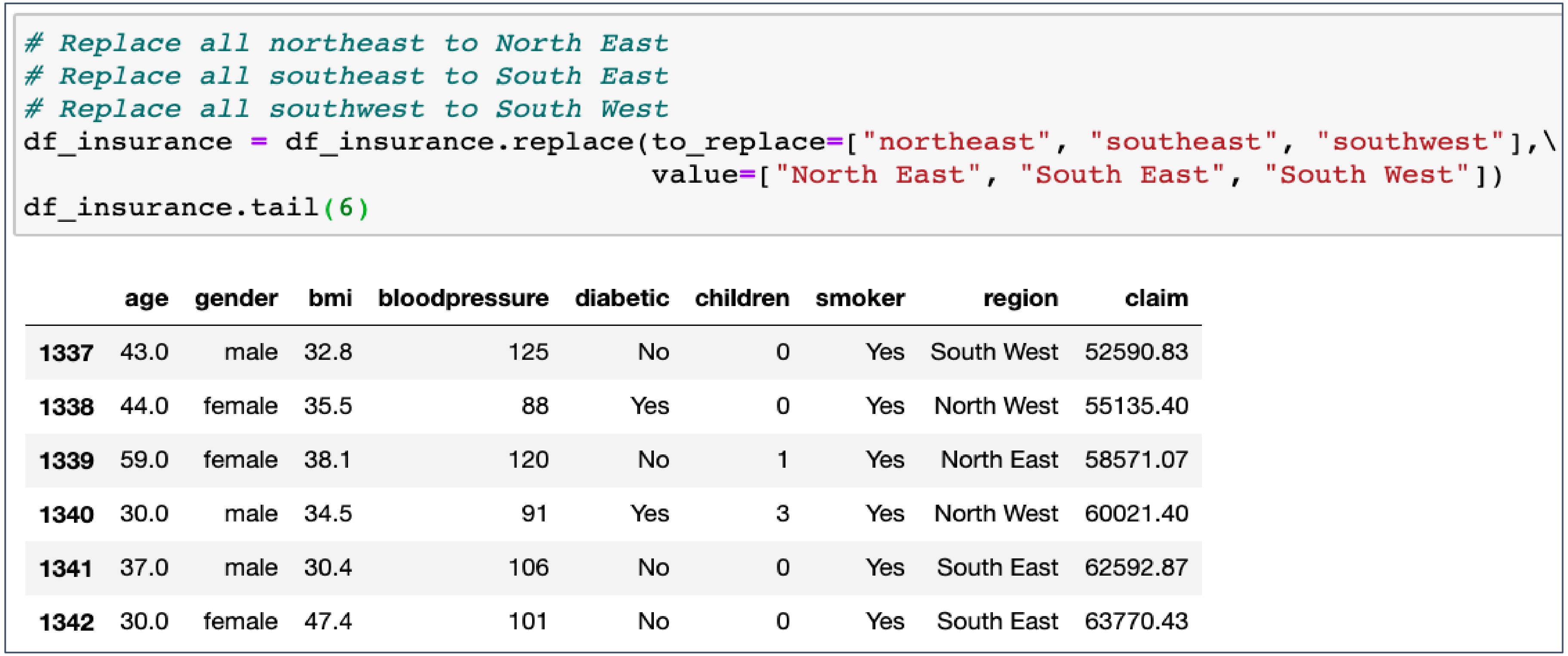


```
# Replace all northwest to North West
df_insurance = df_insurance.replace(to_replace="northwest", value="North West")
df_insurance.head(5)
```

	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
3	37.0	male	33.7	80	No	0	No	North West	1136.40
5	47.0	male	34.4	96	Yes	0	No	North West	1137.47
6	NaN	male	37.3	86	Yes	0	No	North West	1141.45
7	19.0	male	41.1	100	No	0	No	North West	1146.80
8	20.0	male	43.0	86	No	0	No	North West	1149.40

REPLACE THE VALUES

- The replace() method is used to replace the values in the DataFrame
- Note: No column name is passed

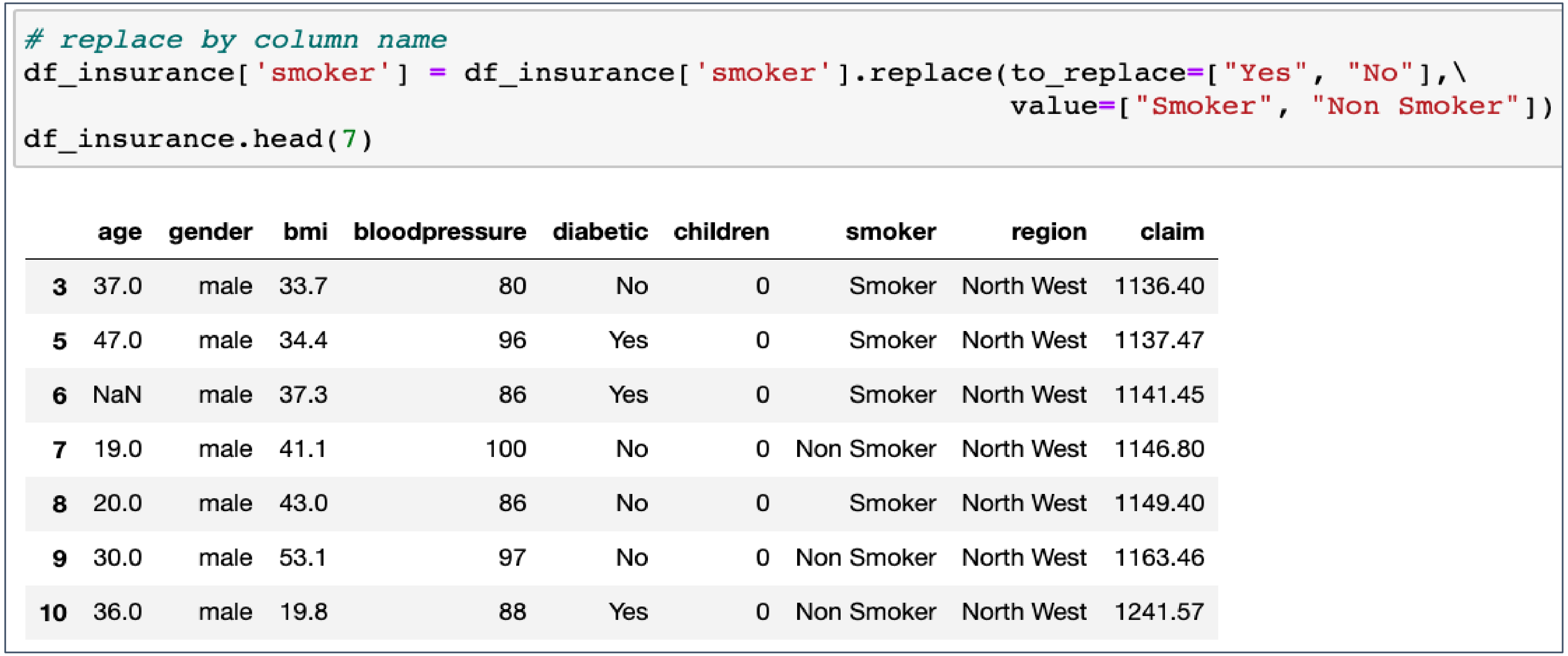


```
# Replace all northeast to North East
# Replace all southeast to South East
# Replace all southwest to South West
df_insurance = df_insurance.replace(to_replace=["northeast", "southeast", "southwest"],\
                                    value=["North East", "South East", "South West"])
df_insurance.tail(6)
```

	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
1337	43.0	male	32.8	125	No	0	Yes	South West	52590.83
1338	44.0	female	35.5	88	Yes	0	Yes	North West	55135.40
1339	59.0	female	38.1	120	No	1	Yes	North East	58571.07
1340	30.0	male	34.5	91	Yes	3	Yes	North West	60021.40
1341	37.0	male	30.4	106	No	0	Yes	South East	62592.87
1342	30.0	female	47.4	101	No	0	Yes	South East	63770.43

REPLACE THE VALUES

- The replace() method is used to replace the values in the DataFrame
- Values replaced of a specific column

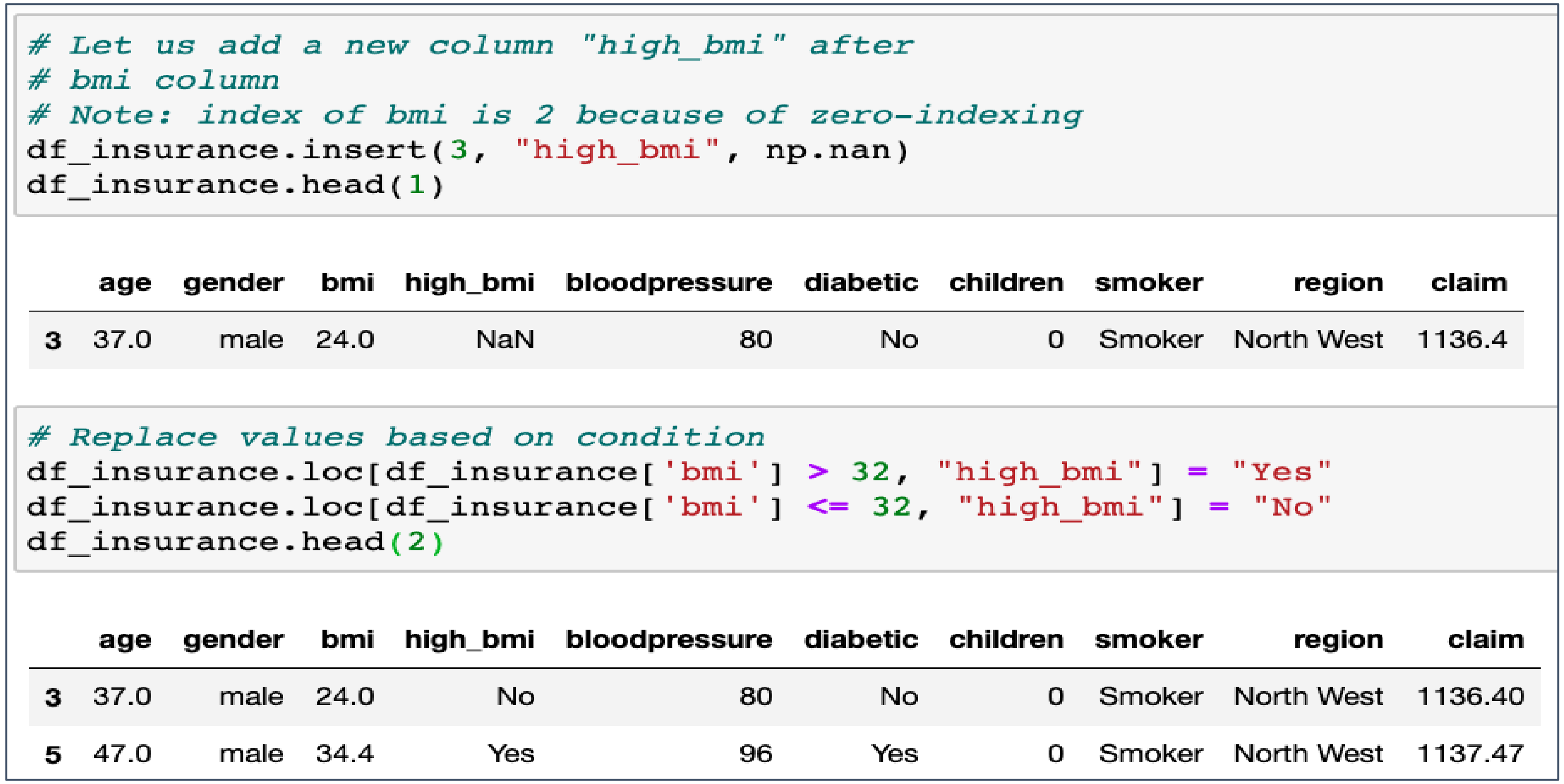


```
# replace by column name
df_insurance['smoker'] = df_insurance['smoker'].replace(to_replace=["Yes", "No"],\
                                                        value=["Smoker", "Non Smoker"])
df_insurance.head(7)
```

	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
3	37.0	male	33.7	80	No	0	Smoker	North West	1136.40
5	47.0	male	34.4	96	Yes	0	Smoker	North West	1137.47
6	NaN	male	37.3	86	Yes	0	Smoker	North West	1141.45
7	19.0	male	41.1	100	No	0	Non Smoker	North West	1146.80
8	20.0	male	43.0	86	No	0	Smoker	North West	1149.40
9	30.0	male	53.1	97	No	0	Non Smoker	North West	1163.46
10	36.0	male	19.8	88	Yes	0	Non Smoker	North West	1241.57

REPLACE THE VALUES BY CONDITION USING INDEX

- Values replaced of a specific column by condition



```
# Let us add a new column "high_bmi" after
# bmi column
# Note: index of bmi is 2 because of zero-indexing
df_insurance.insert(3, "high_bmi", np.nan)
df_insurance.head(1)
```

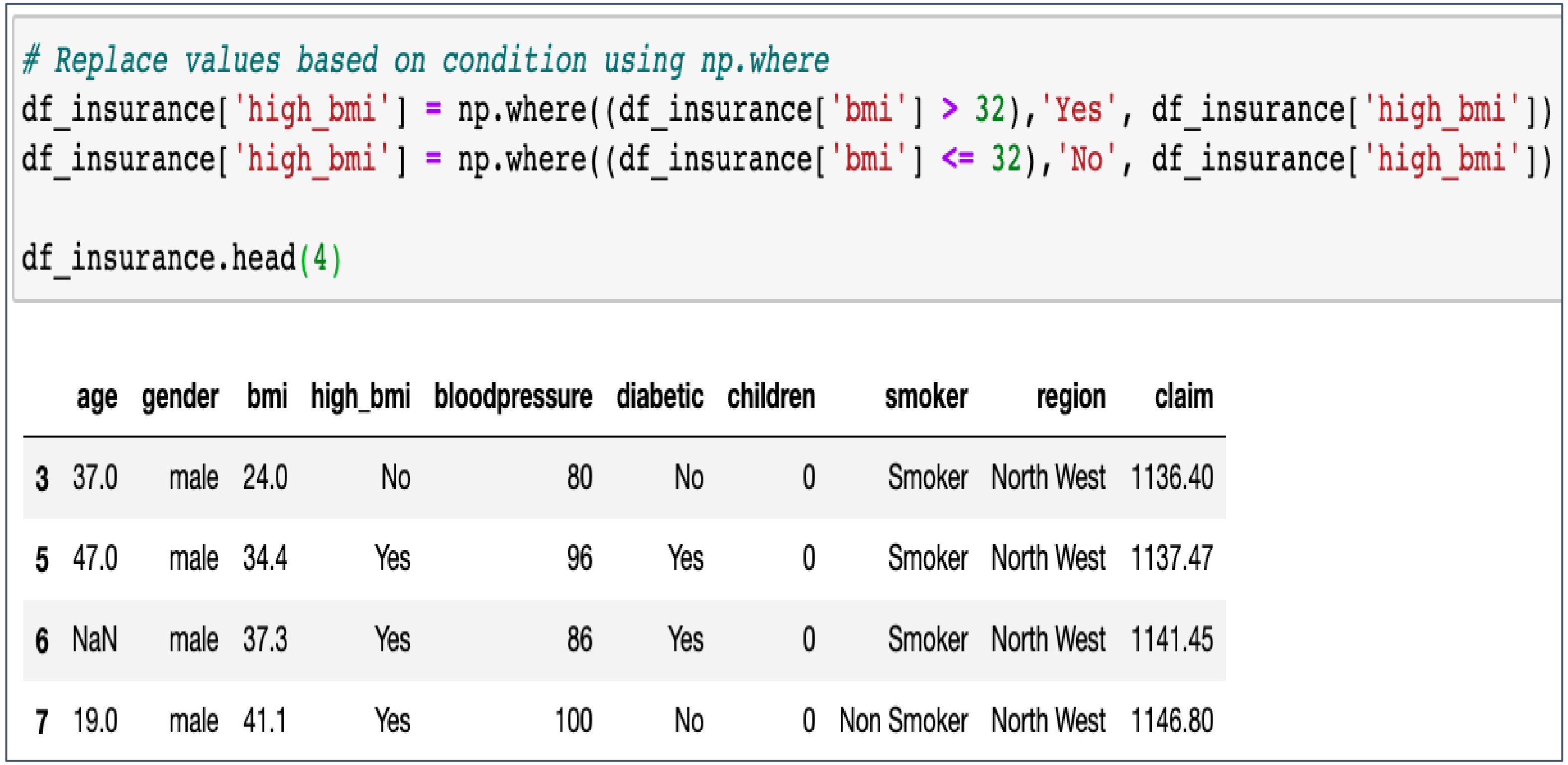
```
    age  gender   bmi  high_bmi  bloodpressure  diabetic  children  smoker       region   claim
3  37.0    male  24.0       NaN             80        No         0  Smoker  North West  1136.4
```

```
# Replace values based on condition
df_insurance.loc[df_insurance['bmi'] > 32, "high_bmi"] = "Yes"
df_insurance.loc[df_insurance['bmi'] <= 32, "high_bmi"] = "No"
df_insurance.head(2)
```

```
    age  gender   bmi  high_bmi  bloodpressure  diabetic  children  smoker       region    claim
3  37.0    male  24.0        No             80        No         0  Smoker  North West  1136.40
5  47.0    male  34.4       Yes             96       Yes         0  Smoker  North West  1137.47
```

REPLACE THE VALUES BY CONDITION USING NUMPY

- Values replaced of a specific column by condition

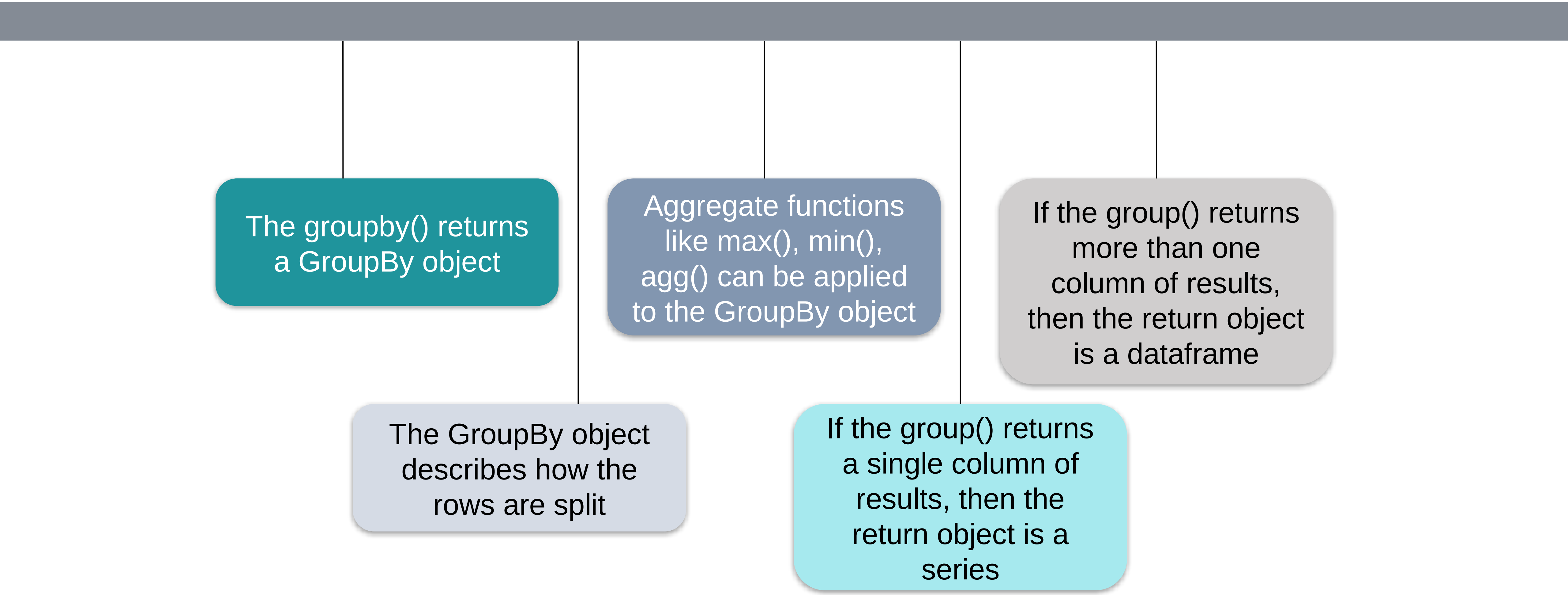


```
# Replace values based on condition using np.where
df_insurance['high_bmi'] = np.where((df_insurance['bmi'] > 32),'Yes', df_insurance['high_bmi'])
df_insurance['high_bmi'] = np.where((df_insurance['bmi'] <= 32),'No', df_insurance['high_bmi'])

df_insurance.head(4)
```

	age	gender	bmi	high_bmi	bloodpressure	diabetic	children	smoker	region	claim
3	37.0	male	24.0	No	80	No	0	Smoker	North West	1136.40
5	47.0	male	34.4	Yes	96	Yes	0	Smoker	North West	1137.47
6	NaN	male	37.3	Yes	86	Yes	0	Smoker	North West	1141.45
7	19.0	male	41.1	Yes	100	No	0	Non Smoker	North West	1146.80

Grouping Dataframe

GROUPING DATAFRAME



The groupby() returns a GroupBy object

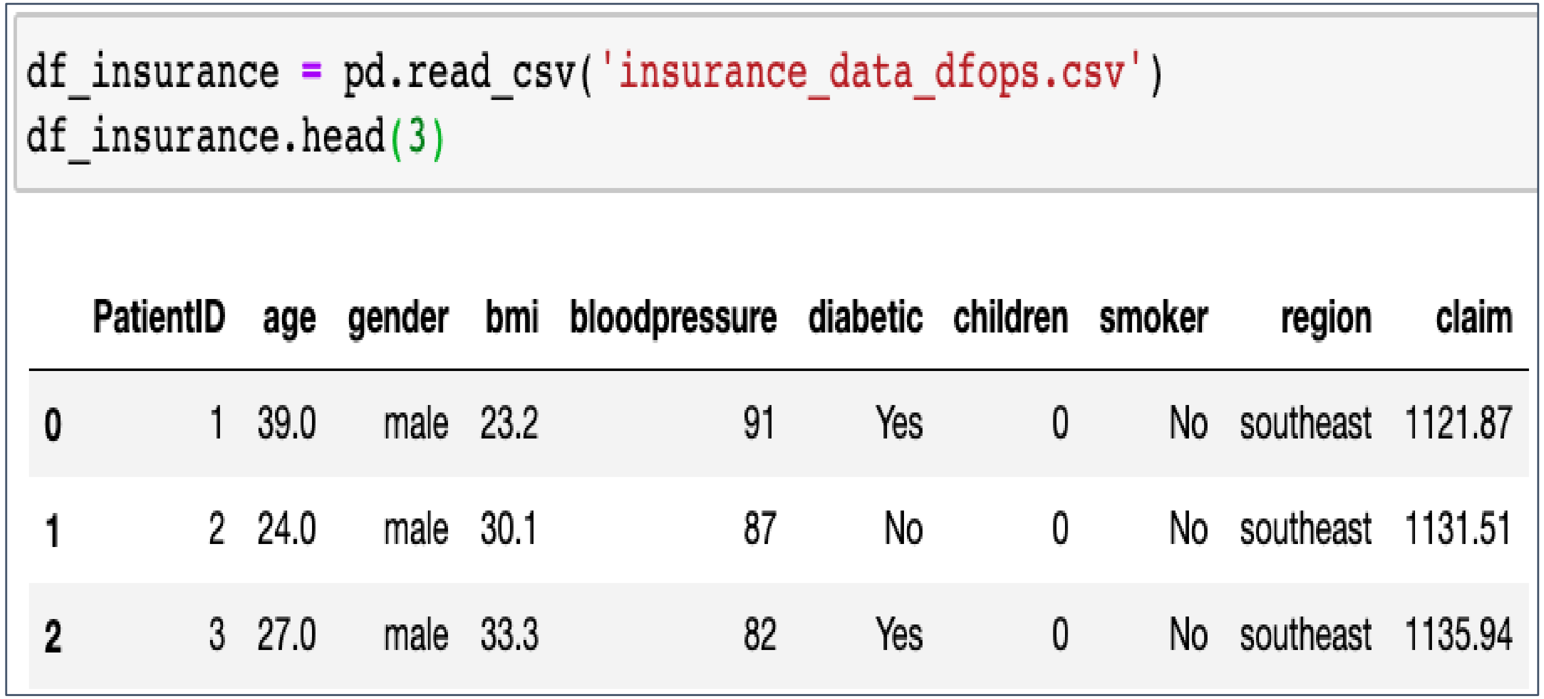
The GroupBy object describes how the rows are split

Aggregate functions like max(), min(), agg() can be applied to the GroupBy object

If the group() returns a single column of results, then the return object is a series

If the group() returns more than one column of results, then the return object is a dataframe

READ THE DATA

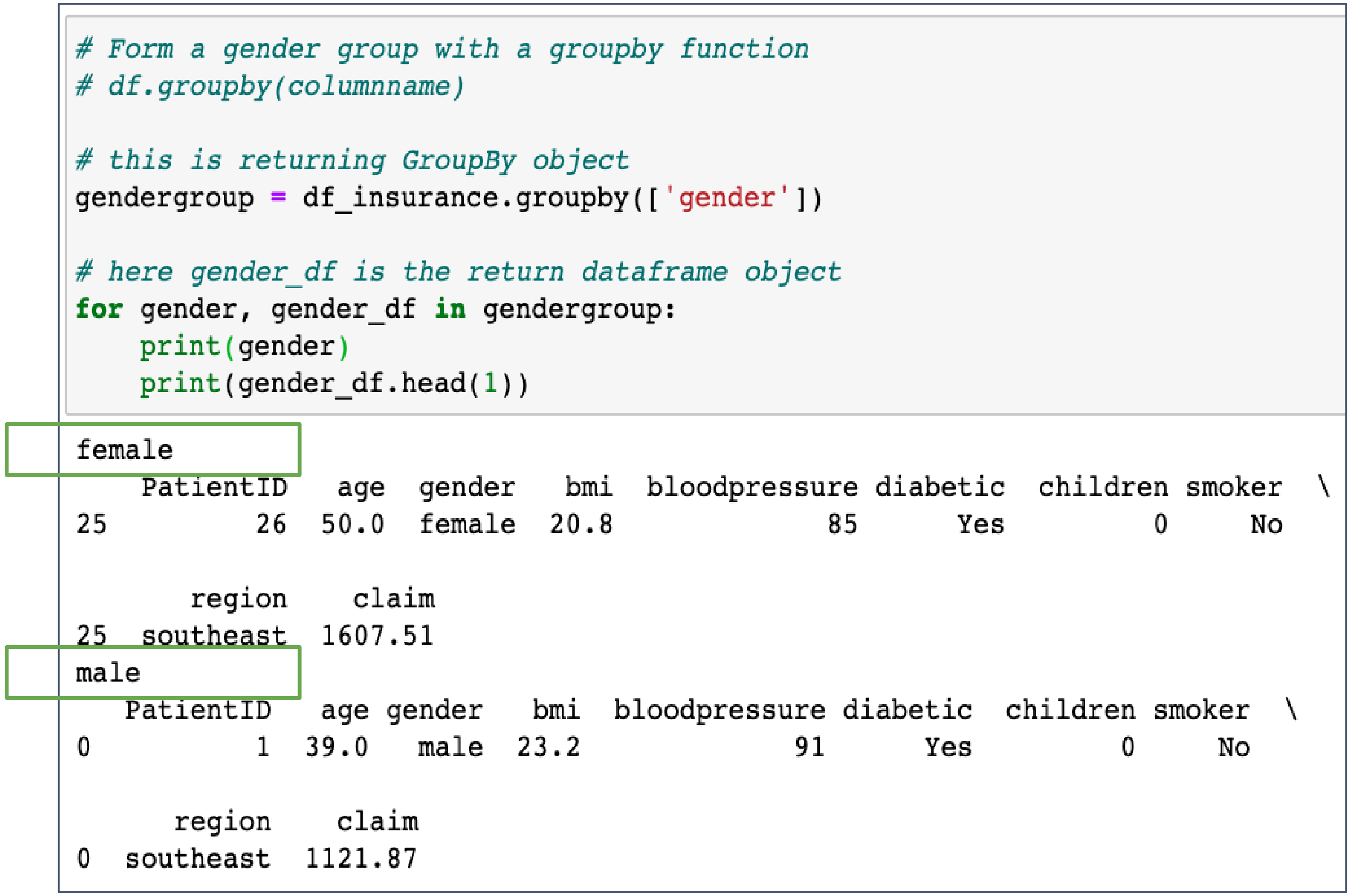


```
df_insurance = pd.read_csv('insurance_data_dfops.csv')
df_insurance.head(3)
```

	PatientID	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
0	1	39.0	male	23.2	91	Yes	0	No	southeast	1121.87
1	2	24.0	male	30.1	87	No	0	No	southeast	1131.51
2	3	27.0	male	33.3	82	Yes	0	No	southeast	1135.94

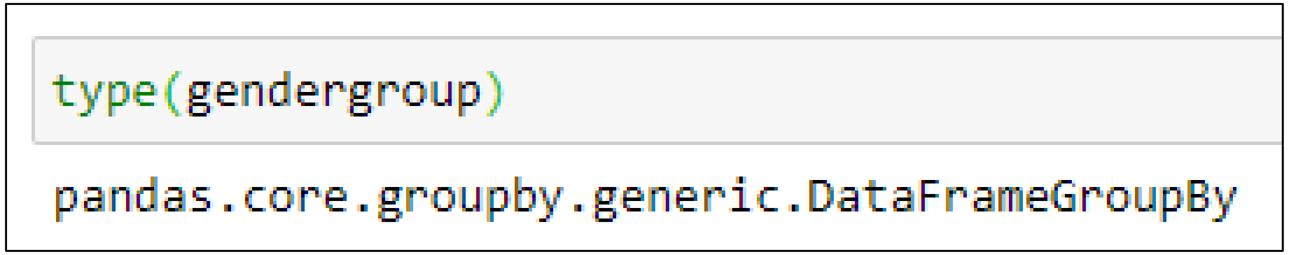
GROUPING DATAFRAME

- Use groupby() method to group the dataframe by the specific column(s)



```
# Form a gender group with a groupby function
# df.groupby(columnname)

# this is returning GroupBy object
gendergroup = df_insurance.groupby(['gender'])

# here gender_df is the return dataframe object
for gender, gender_df in gendergroup:
    print(gender)
    print(gender_df.head(1))
```

```
female
    PatientID   age  gender   bmi  bloodpressure diabetic  children smoker  \
25         26  50.0  female  20.8             85      Yes         0     No

      region    claim
25  southeast  1607.51
male
   PatientID   age gender   bmi  bloodpressure diabetic  children smoker  \
0          1  39.0   male  23.2             91      Yes         0     No

     region    claim
0  southeast  1121.87
```



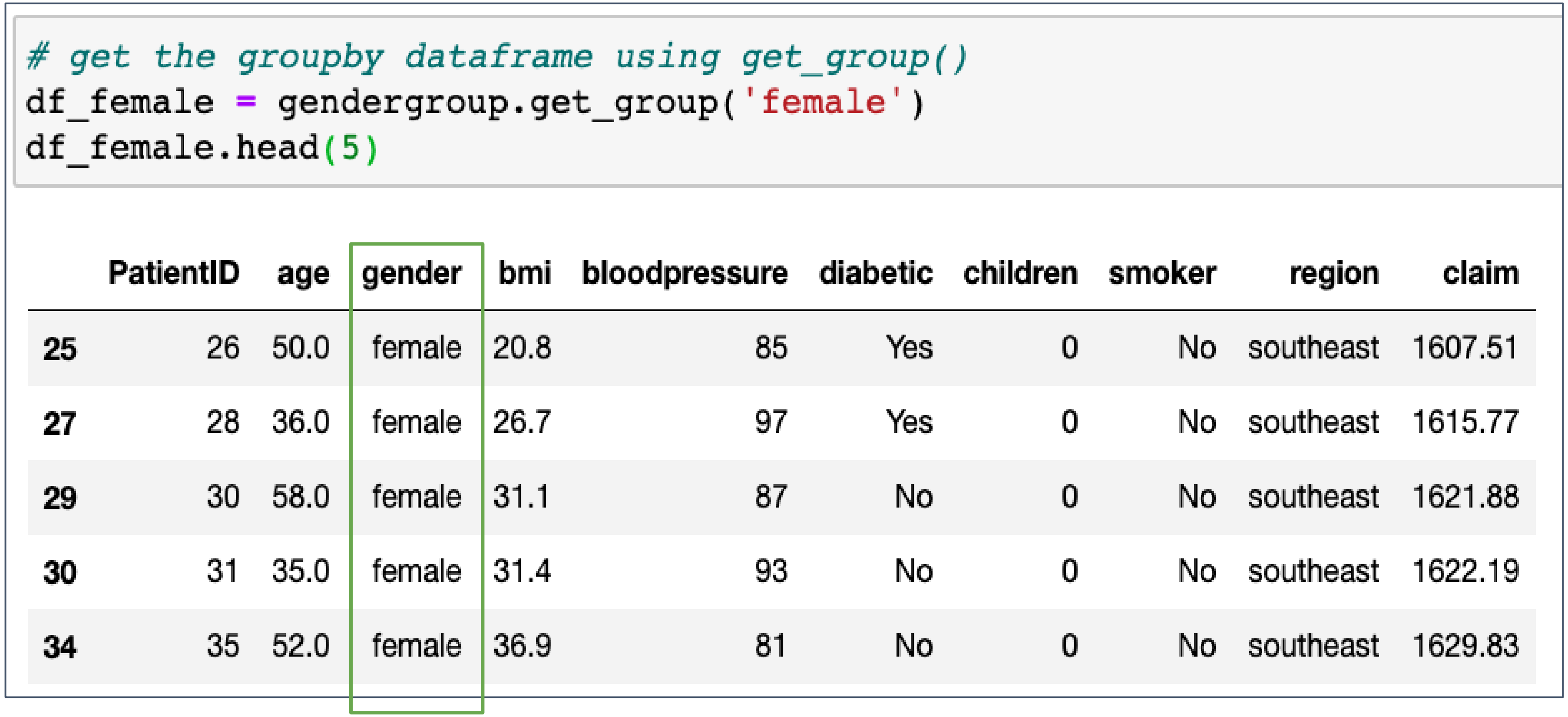
```
type(gendergroup)
```

```
pandas.core.groupby.generic.DataFrameGroupBy
```

The groupby() applied on a pandas DataFrame returns a DataFrameGroupBy object

GET GROUPBY DATAFRAME

- Internally a groupby dataframe will split the data by groups
- Get the groupby dataframe object using get_group()

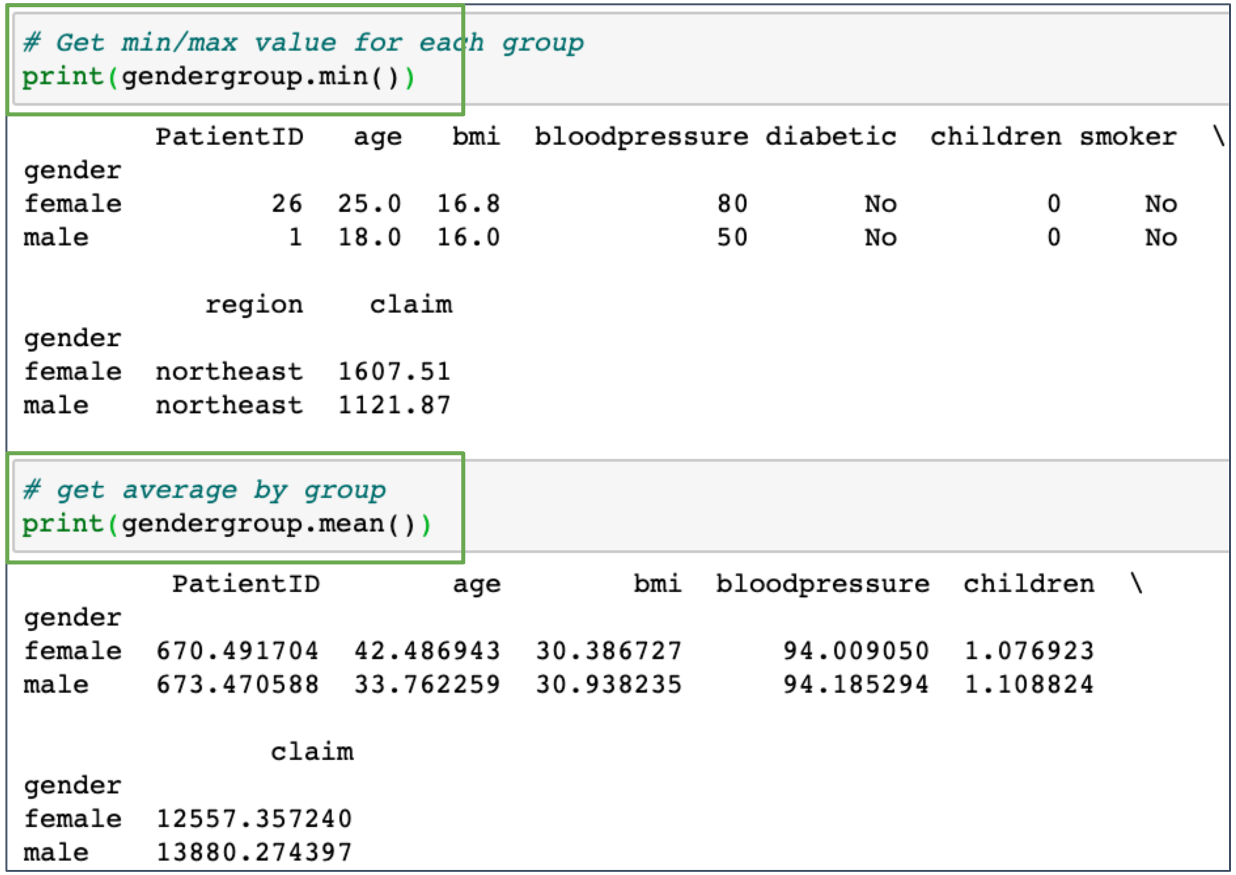


```
# get the groupby dataframe using get_group()
df_female = gendergroup.get_group('female')
df_female.head(5)
```

	PatientID	age	gender	bmi	bloodpressure	diabetic	children	smoker	region	claim
25	26	50.0	female	20.8	85	Yes	0	No	southeast	1607.51
27	28	36.0	female	26.7	97	Yes	0	No	southeast	1615.77
29	30	58.0	female	31.1	87	No	0	No	southeast	1621.88
30	31	35.0	female	31.4	93	No	0	No	southeast	1622.19
34	35	52.0	female	36.9	81	No	0	No	southeast	1629.83

FUNCTIONS ON GROUPBY OBJECT

- Internally a groupby dataframe will split the data by groups
- Aggregate functions by groupby object



```
# Get min/max value for each group
print(gendergroup.min())
```

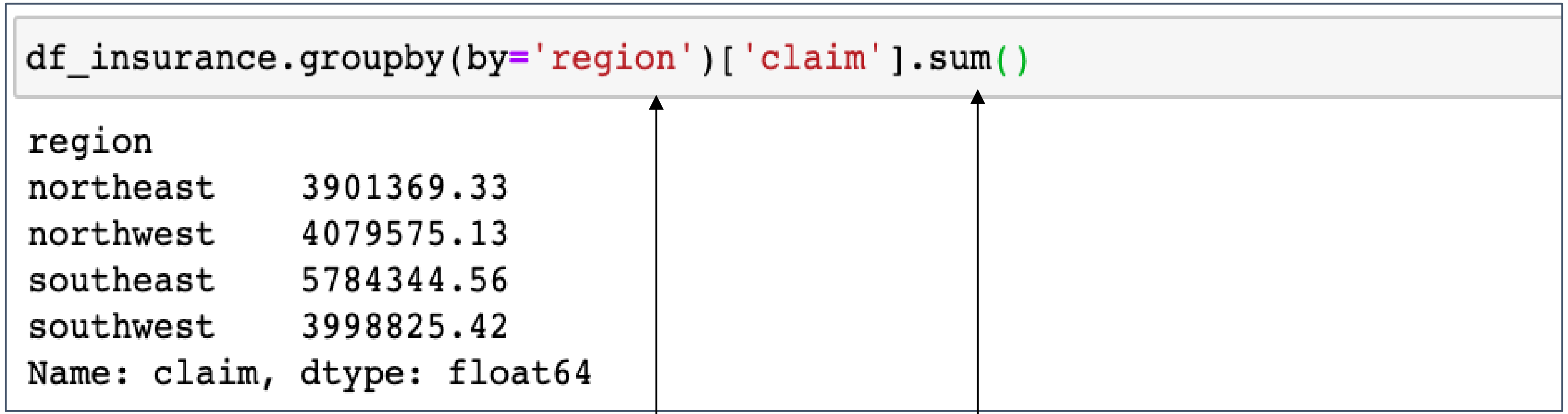
```
        PatientID   age   bmi  bloodpressure diabetic  children smoker  \
gender
female         26  25.0  16.8             80       No         0     No
male            1  18.0  16.0             50       No         0     No

           region    claim
gender
female  northeast  1607.51
male    northeast  1121.87
```

```
# get average by group
print(gendergroup.mean())
```

```
         PatientID        age        bmi  bloodpressure  children  \
gender
female  670.491704  42.486943  30.386727      94.009050  1.076923
male    673.470588  33.762259  30.938235      94.185294  1.108824

              claim
gender
female  12557.357240
male    13880.274397
```

FUNCTIONS ON GROUPBY OBJECT AT RUNTIME

- Showcasing groupby() function that creates a groupby object at runtime
- Aggregate functions return result by groups



```
df_insurance.groupby(by='region')['claim'].sum()
```

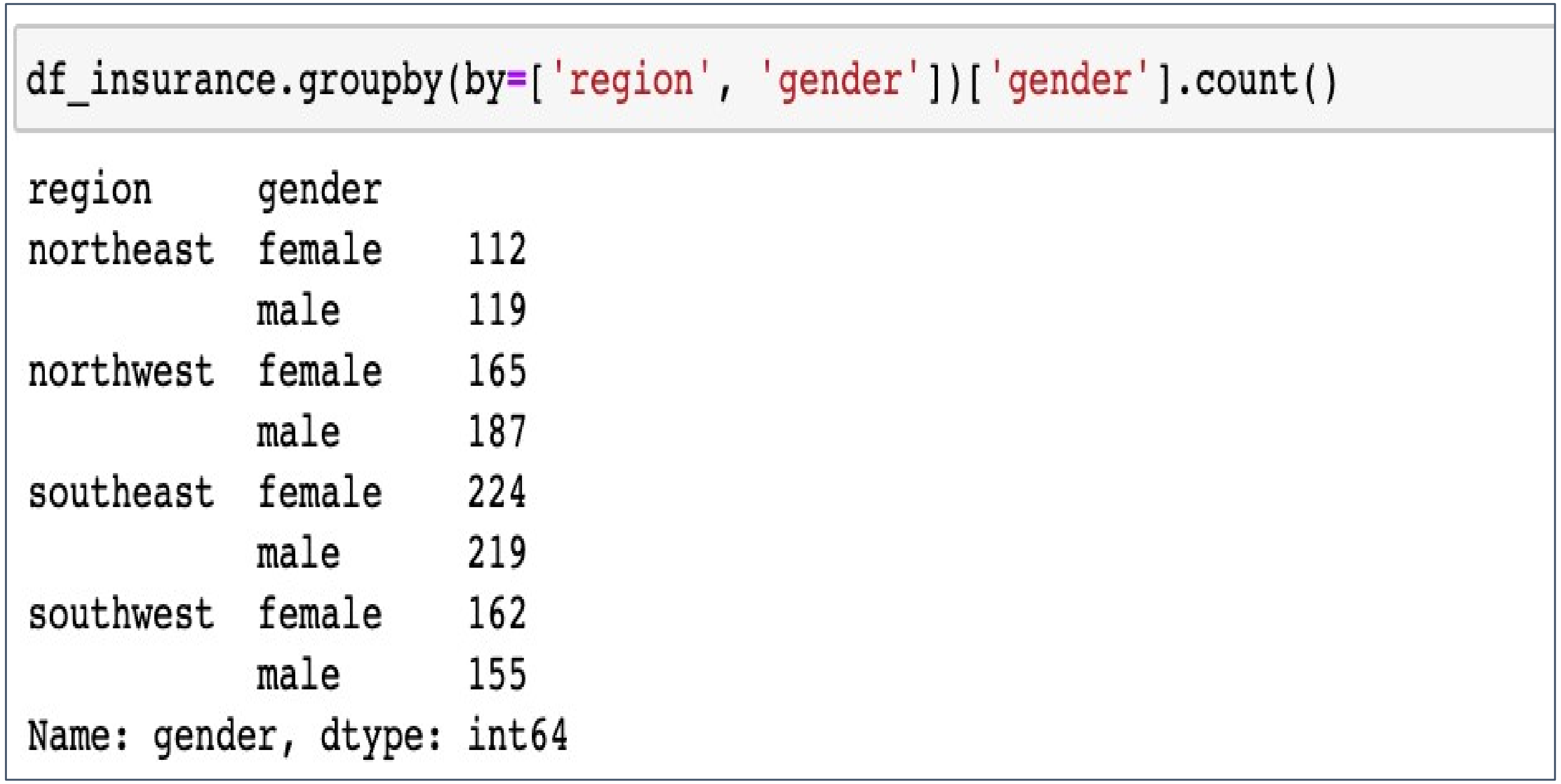
```
region
northeast    3901369.33
northwest    4079575.13
southeast    5784344.56
southwest    3998825.42
Name: claim, dtype: float64
```

Add the claims for each region

Group the data by ‘region’

FUNCTIONS ON GROUPBY OBJECT AT RUNTIME

- Get the number of male & female for each region

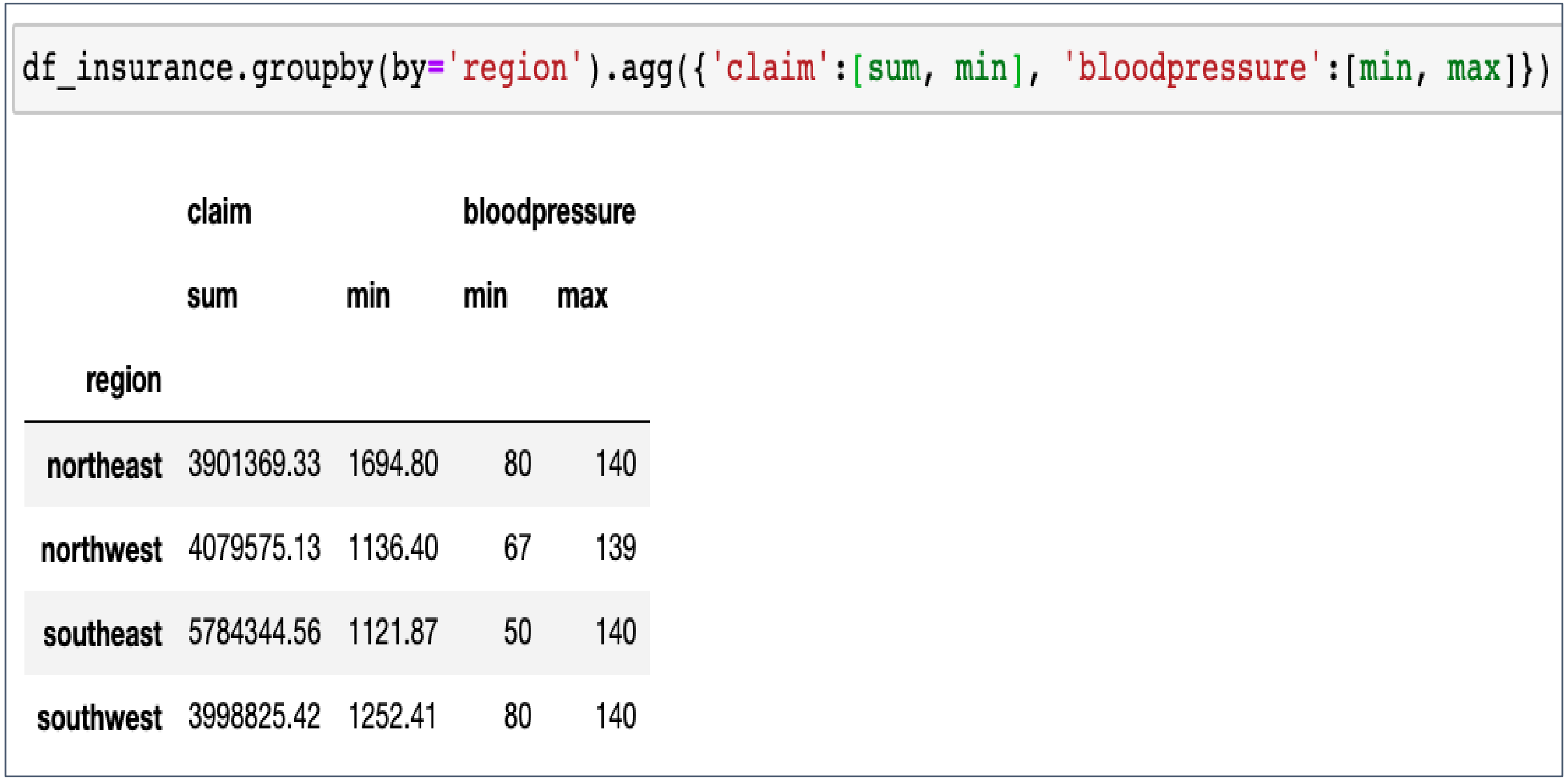


```
df_insurance.groupby(by=['region', 'gender'])['gender'].count()
```

```
region     gender
northeast  female    112
           male      119
northwest  female    165
           male      187
southeast  female    224
           male      219
southwest  female    162
           male      155
Name: gender, dtype: int64
```

AGGREGATES ON MULTIPLE COLUMNS

- Calculating sum & min on ‘claim’ while calculating min & max on ‘bloodpressure’



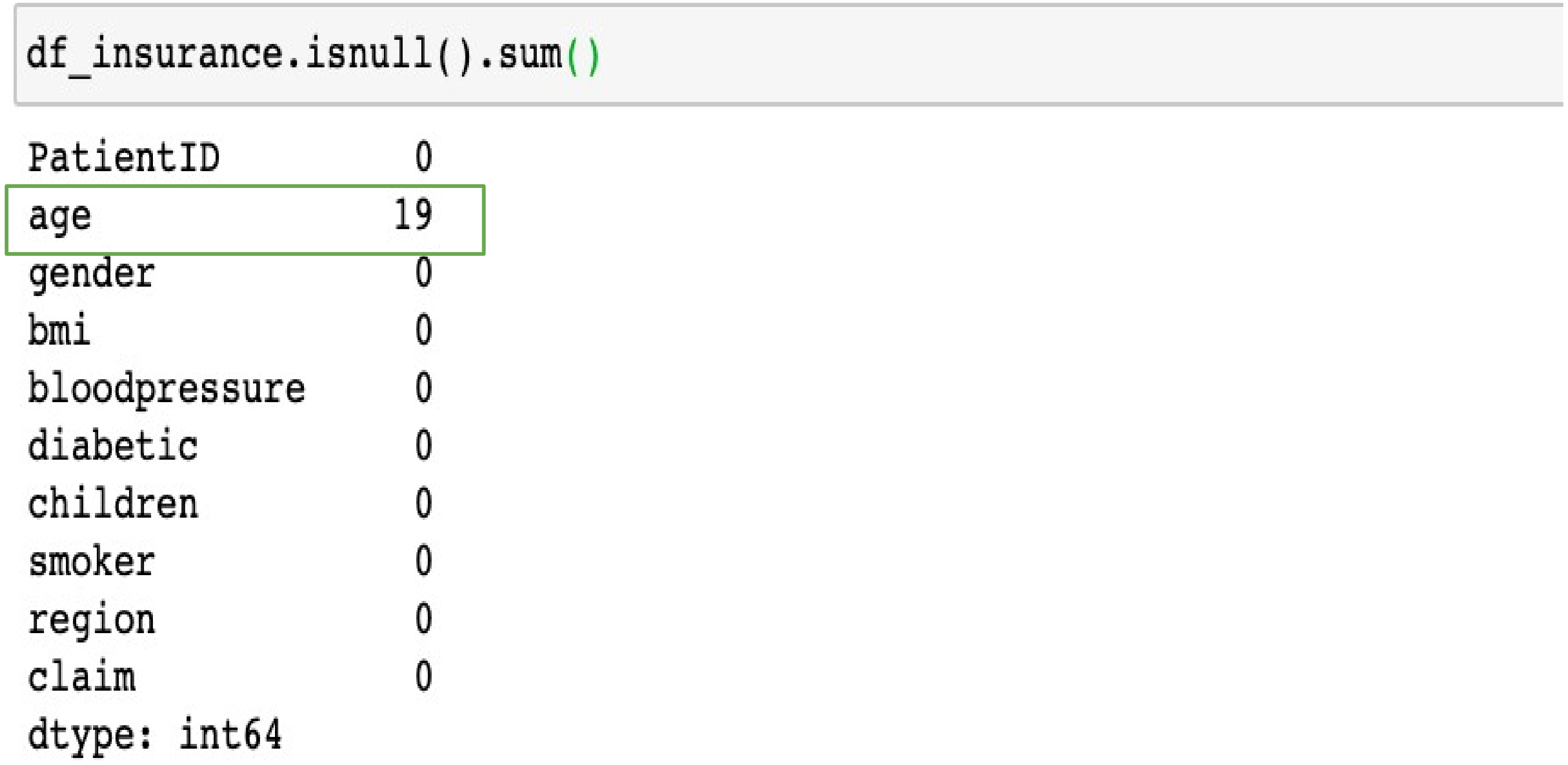
```
df_insurance.groupby(by='region').agg({'claim':[sum, min], 'bloodpressure':[min, max]})
```

	claim		bloodpressure	
	sum	min	min	max
region				
northeast	3901369.33	1694.80	80	140
northwest	4079575.13	1136.40	67	139
southeast	5784344.56	1121.87	50	140
southwest	3998825.42	1252.41	80	140

Missing Values Analysis & Treatment

DOES MISSING VALUES EXISTS

- Check if there are missing value in any columns

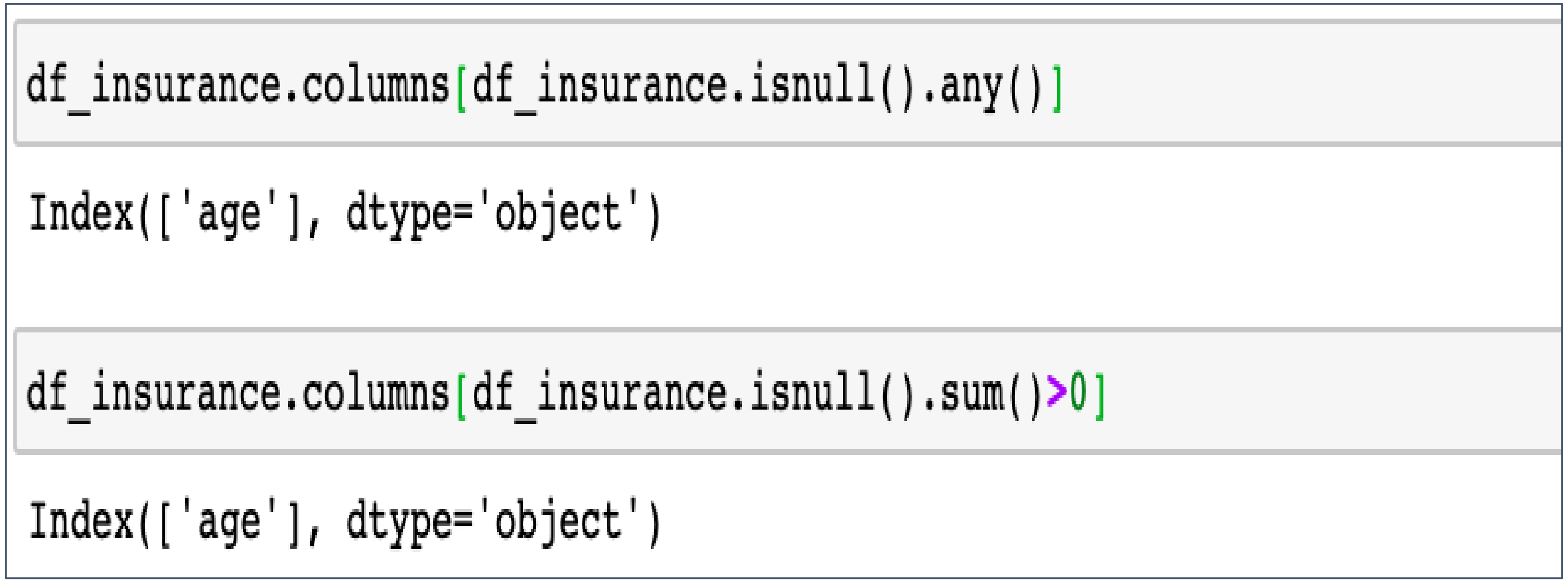


```
df_insurance.isnull().sum()
```

```
PatientID        0
age             19
gender           0
bmi              0
bloodpressure    0
diabetic         0
children         0
smoker           0
region           0
claim            0
dtype: int64
```

DOES MISSING VALUES EXISTS

- Check only columns which have missing values



```
df_insurance.columns[df_insurance.isnull().any()]
```

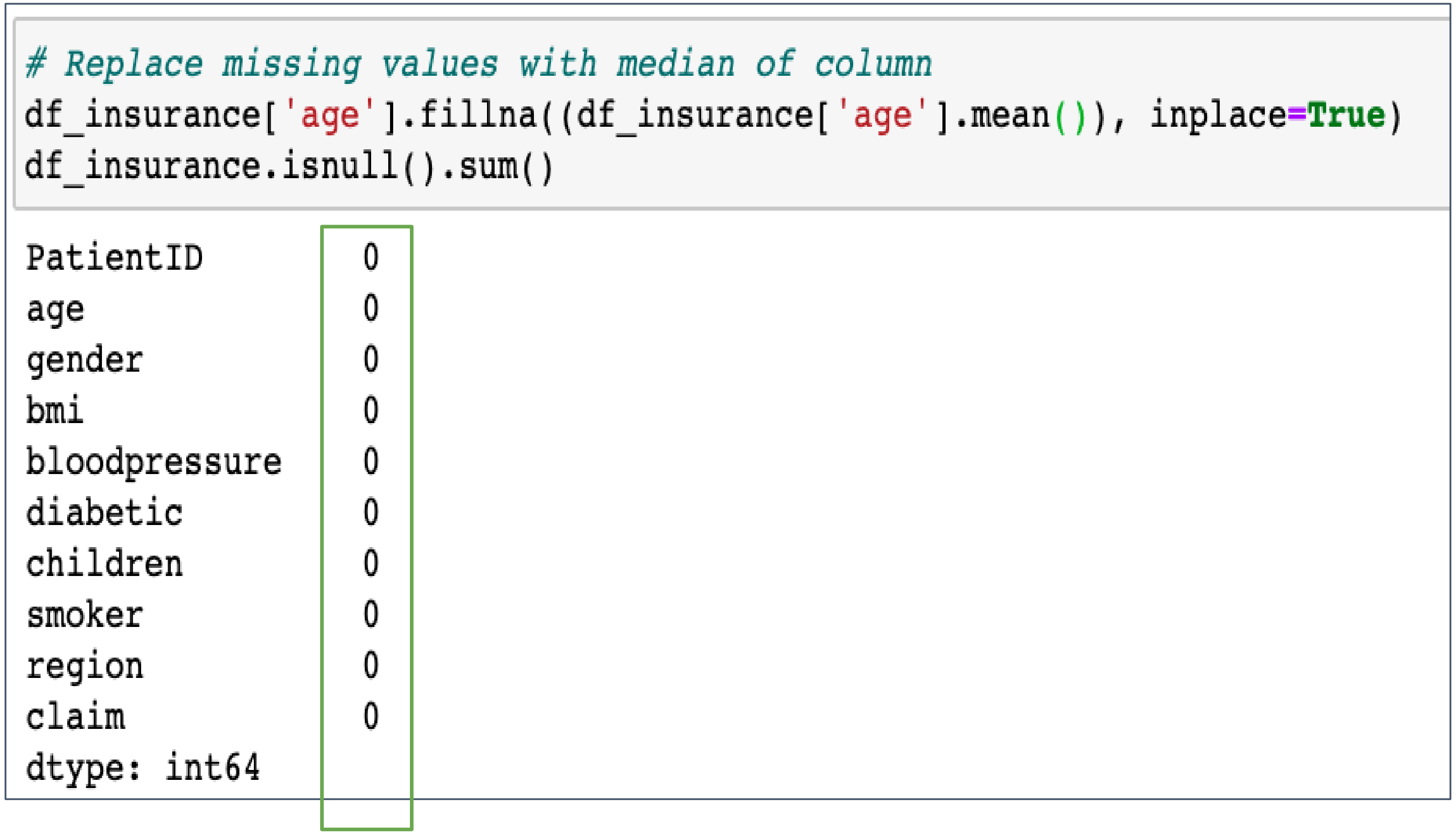
```
Index(['age'], dtype='object')
```

```
df_insurance.columns[df_insurance.isnull().sum()>0]
```

```
Index(['age'], dtype='object')
```

FILLING IN THE MISSING VALUES

- Filling in the missing values of a numeric variable with the mean



```
# Replace missing values with median of column
df_insurance['age'].fillna((df_insurance['age'].mean()), inplace=True)
df_insurance.isnull().sum()
```

```
PatientID        0
age              0
gender           0
bmi              0
bloodpressure    0
diabetic         0
children         0
smoker           0
region           0
claim            0
dtype: int64
```

We're committed to empower you to be
#FutureReady
through powerful training solutions.

IMARTICUS
LEARNING

250+
Corporate Clients

30,000+
Learners Trained

25000+
Learners Placed

We build the workforce of the future.

FREE WEBINARS

Sign up for free webinars with industry experts every fortnight!

Enroll Now!

Contact us today!
https://imarticus.org/