# PANDAS DATA STRUCTURES

# Data Structures in Pandas

- Pandas has two main data structures;
  - DataFrame, which is two dimensional
  - Series, which is one dimensional

*DataFrame*

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **5** | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| **6** | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| **7** | Basic 4 | 130 | 3 | 25 | 37.038562 |
| **8** | Bran Chex | 90 | 2 | 25 | 49.120253 |
| **9** | Bran Flakes | 90 | 3 | 25 | 53.313813 |

```
0      70
1     120
2      70
3      50
4     110
5     110
6     110
7     130
8      90
9      90
Name: calories, dtype: int64
```

*Series*

# What is Pandas DataFrame

- Pandas provides a two dimensional data structure called DataFrame.
- A row is represented by row labels, also called index, which may be numerical or strings.
- A column is represented by column labels which may be numerical or strings.
- Following DataFrame contains 10 rows (0-9) and and 5 columns (name, calories, protein, vitamins, rating)

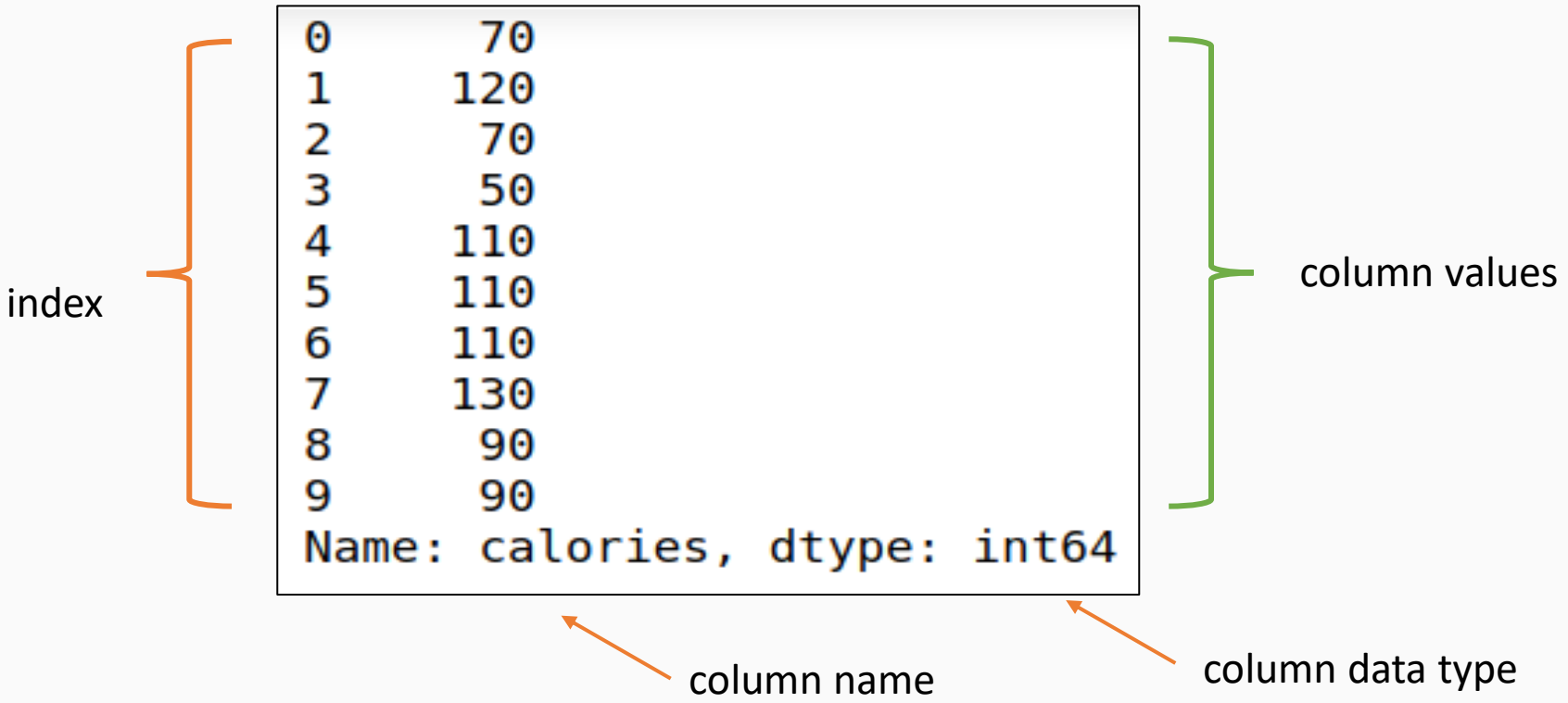
index

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |
| 5 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 6 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 7 | Basic 4 | 130 | 3 | 25 | 37.038562 |
| 8 | Bran Chex | 90 | 2 | 25 | 49.120253 |
| 9 | Bran Flakes | 90 | 3 | 25 | 53.313813 |

column labels

# What is Pandas Series

- A Series in Pandas is a one dimensional data structure.
- It consists of a single row or column.
- Following Series contains 10 rows (0-9) and and 1 column called calories.



# DataFrame and Series

- A Pandas DataFrame is just a collection of one or more Series.
- The Series in the previous example was extracted from the DataFrame.

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |
| 5 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 6 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 7 | Basic 4 | 130 | 3 | 25 | 37.038562 |
| 8 | Bran Chex | 90 | 2 | 25 | 49.120253 |
| 9 | Bran Flakes | 90 | 3 | 25 | 53.313813 |

*DataFrame*

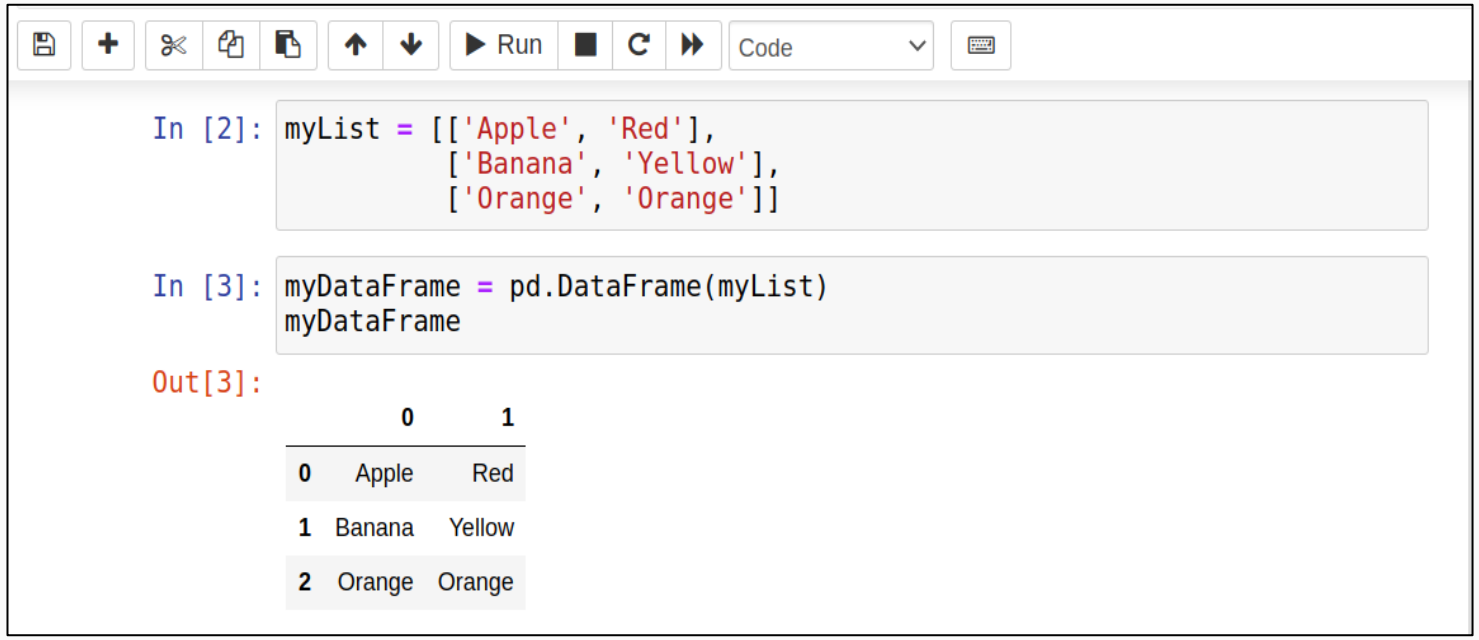
```
0     70
1    120
2     70
3     50
4    110
5    110
6    110
7    130
8     90
9     90
Name: calories, dtype: int64
```

*Series*

*Identical*

# Creating a DataFrame Using Lists (1/3)

- We can create a DataFrame using lists.
- We pass the list as an argument to the pandas.DataFrame() function which returns us a DataFrame.
- Pandas automatically assigns numerical row labels to each row of the DataFrame.
- Since, we did not provide column labels, Pandas automatically assigned numerical column labels to each column as well.




| | 0 | 1 |
|---|---|---|
| 0 | Apple | Red |
| 1 | Banana | Yellow |
| 2 | Orange | Orange |

# Creating a DataFrame Using Lists (2/3)

- Let's create another DataFrame using the same list, but this time with custom column labels.
- Pandas.DataFrame() takes another optional argument called 'columns' which takes a list of custom column names to be set as columns' labels.

```
In [2]: myList = [['Apple', 'Red'],
                  ['Banana', 'Yellow'],
                  ['Orange', 'Orange']]

In [4]: myDataFrame = pd.DataFrame(myList, columns=['Fruit', 'Color'])
        myDataFrame
```

Out[4]:

| | Fruit | Color |
|---|---|---|
| 0 | Apple | Red |
| 1 | Banana | Yellow |
| 2 | Orange | Orange |

# Creating a DataFrame Using Lists (3/3)

- As we know that a NumPy Array is similar to a Python list with added functionality, we can also convert a NumPy Array to a DataFrame using the same method.

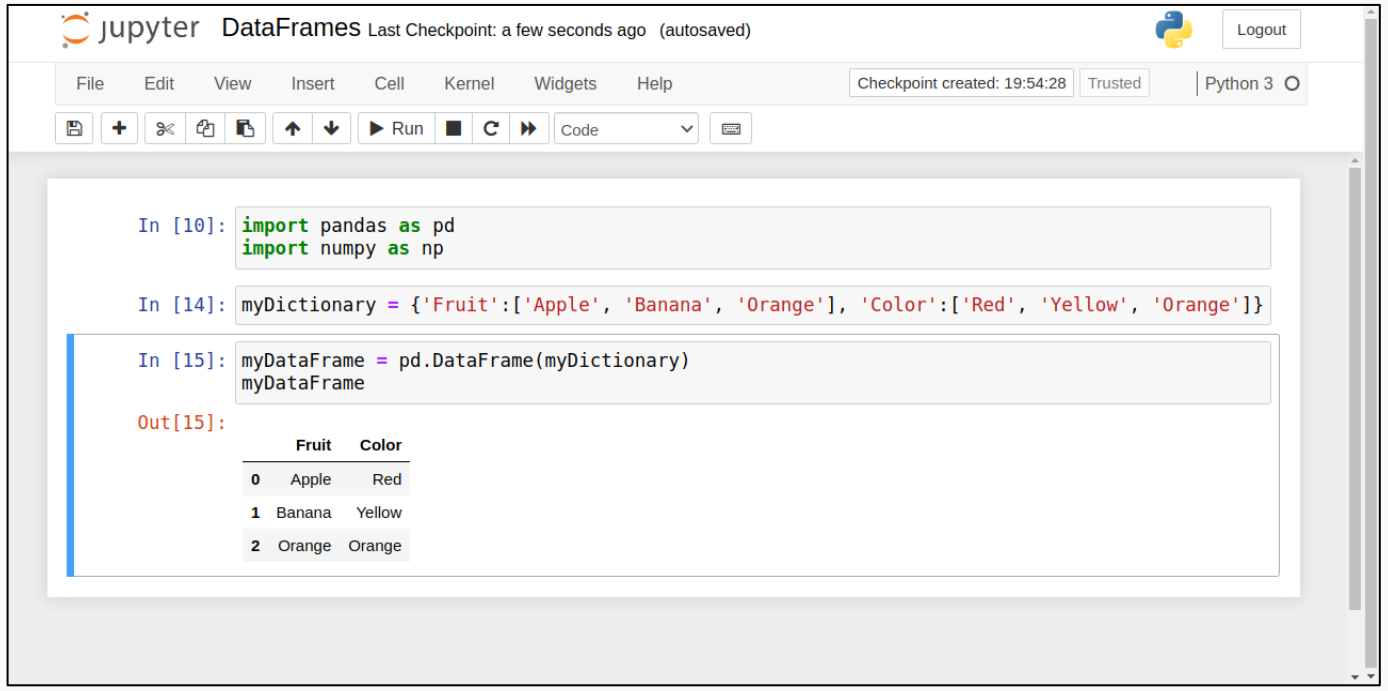
```
In [9]: myList = np.array([[0, 1],
                           [2, 3],
                           [4, 5]])

In [10]: myDataFrame = pd.DataFrame(myList, columns=['even', 'odd'])
         myDataFrame
```

Out[10]:

| | even | odd |
|---|---|---|
| **0** | 0 | 1 |
| **1** | 2 | 3 |
| **2** | 4 | 5 |

# Creating a DataFrame Using Dictionary

- We can also pass a dictionary to the pandas.DataFrame() function to create a DataFrame.
- Each key of the array should have a list of one or more values associated with it.
- The keys of the dictionary become column labels.
- Pandas automatically assigns numerical row labels to each row of the DataFrame.



# Loading csv File as a DataFrame

- We can also load a csv (comma separated values) file as a DataFrame in Pandas using the pandas.read_csv() function.
- Each value of the first row of the csv file becomes a column label.
- Pandas automatically assigns numerical row labels to each row of the DataFrame.

```
In [11]: df = pd.read_csv('cereals.csv')
         df
```

Out[11]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |
| 5 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 6 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 7 | Basic 4 | 130 | 3 | 25 | 37.038562 |
| 8 | Bran Chex | 90 | 2 | 25 | 49.120253 |
| 9 | Bran Flakes | 90 | 3 | 25 | 53.313813 |

# Changing the Index Column

- We can set one of the existing columns as the new index column of the DataFrame using .set_index() function.

```
In [13]: df
```

Out[13]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |

```
In [14]: df.set_index('name')
```

Out[14]:

| | calories | protein | vitamins | rating |
|---|---|---|---|---|
| **name** | | | | |
| **100% Bran** | 70 | 4 | 25 | 68.402973 |
| **100% Natural Bran** | 120 | 3 | 0 | 33.983679 |
| **All-Bran** | 70 | 4 | 25 | 59.425505 |
| **All-Bran with Extra Fiber** | 50 | 4 | 25 | 93.704912 |
| **Almond Delight** | 110 | 2 | 25 | 34.384843 |

# Inplace (1/2)

- Remember that most of the functions in Pandas do not change the original DataFrame.
- In the previous section we changed the index column of our DataFrame. If we print our DataFrame again, we see that the original DataFrame is unchanged.

```
In [15]: df
```

Out[15]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |

# Inplace (2/2)

- We can use the inplace argument to make changes to the original DataFrame.
- In the following example we use the .set_index() function to change the index of our DataFrame, and set inplace = True.
- As shown in the figure, our original DataFrame has been changed.

```
In [16]: df.set_index('name', inplace=True)

In [17]: df
```

Out[17]:

| name | calories | protein | vitamins | rating |
|---|---|---|---|---|
| 100% Bran | 70 | 4 | 25 | 68.402973 |
| 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| All-Bran | 70 | 4 | 25 | 59.425505 |
| All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| Almond Delight | 110 | 2 | 25 | 34.384843 |

# Examining the Data (1/2)

## *head()*

- head() function gives us the **first** 5 rows of the DataFrame/Series by default.
- To get more rows, we can pass the desired number as an argument to the head() function.

```
In [20]: df.head(7)
```

Out[20]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **5** | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| **6** | Apple Jacks | 110 | 2 | 25 | 33.174094 |

# Examining the Data (2/2)

## *tail()*

- tail() function gives us the **last** 5 rows of the DataFrame/Series by default.
- To get more rows, we can pass the desired number as an argument to the tail() function.

```
In [22]: df.tail(7)
```

Out[22]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **5** | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| **6** | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| **7** | Basic 4 | 130 | 3 | 25 | 37.038562 |
| **8** | Bran Chex | 90 | 2 | 25 | 49.120253 |
| **9** | Bran Flakes | 90 | 3 | 25 | 53.313813 |

# Statistical Summary

- We can use the describe() function to get a quick statistical summary of each column of the DataFrame.

```
In [25]: df.describe()
```

Out[25]:

| | calories | protein | vitamins | rating |
|---|---|---|---|---|
| **count** | 5.000000 | 5.000000 | 5.00000 | 5.000000 |
| **mean** | 84.000000 | 3.400000 | 20.00000 | 57.980382 |
| **std** | 29.664794 | 0.894427 | 11.18034 | 25.097570 |
| **min** | 50.000000 | 2.000000 | 0.00000 | 33.983679 |
| **25%** | 70.000000 | 3.000000 | 25.00000 | 34.384843 |
| **50%** | 70.000000 | 4.000000 | 25.00000 | 59.425505 |
| **75%** | 110.000000 | 4.000000 | 25.00000 | 68.402973 |
| **max** | 120.000000 | 4.000000 | 25.00000 | 93.704912 |

# [ ] Operator for Row Slicing (1/2)

- We can use the brackets ([ ]) operator to slice rows of the DataFrame.
- We pass a start index (inclusive) and an end index (exclusive) to the bracket operator ([ ]) to slice the rows of the DataFrame.

```
In [26]: df[1:4]
```

Out[26]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |

# [ ] Operator for Row Slicing (2/2)

- Remember that [ ] operator works on row position and not row labels.
- For example, in the following case row labels are strings. But we pass positions of the rows that we want to slice.

```
In [30]: df
```

Out[30]:

| name | calories | protein | vitamins | rating |
|---|---|---|---|---|
| 100% Bran | 70 | 4 | 25 | 68.402973 |
| 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| All-Bran | 70 | 4 | 25 | 59.425505 |
| All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| Almond Delight | 110 | 2 | 25 | 34.384843 |

```
In [31]: df[1:4]
```

Out[31]:

| name | calories | protein | vitamins | rating |
|---|---|---|---|---|
| 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| All-Bran | 70 | 4 | 25 | 59.425505 |
| All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |

# Quiz Time

1. Consider the given DataFrame called df. Which of the following will give us rows 5-10 of the DataFrame?
   a) df[5:10]
   b) df[5:11]
   c) df[4:10]
   d) df[4:11]

# Quiz Time

1. Consider the given DataFrame called df. Which of the following will give us rows 5-10 of the DataFrame?
   a) df[5:10]
   b) df[5:11]
   c) df[4:10]
   d) df[4:11]

# [ ] Operator for Column Indexing

- We can also use the brackets ([ ]) operator to index column of the DataFrame.
- Indexing a single column returns a Series.
- Indexing a list of columns returns a DataFrame.
- Remember that for indexing columns, we pass their labels to the [ ] operator and not their positions.

```
In [34]: df[['name', 'rating']]
```

Out[34]:

| | name | rating |
|---|---|---|
| 0 | 100% Bran | 68.402973 |
| 1 | 100% Natural Bran | 33.983679 |
| 2 | All-Bran | 59.425505 |
| 3 | All-Bran with Extra Fiber | 93.704912 |
| 4 | Almond Delight | 34.384843 |

# Boolean List

- We can also pass a list of booleans to the [ ] operator.
- We get all the rows of the DataFrame for which the corresponding element in the list is True.
- Rows of the DataFrame for which the corresponding element in the list is False are ignored.
- Note: Original DataFrame remains unchanged.

```
In [71]: df
```

Out[71]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |

```
In [75]: thirdRow = [False, False, True, False, False]
         df[thirdRow]
```

Out[75]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |

# Filtering Rows (1/3)

- We can also use the [ ] operator to apply conditions on one or more columns of the DataFrame.
- Rows of the DataFrame which satisfy those conditions are filtered out.

```
In [36]: condition = df['calories'] > 70
         df[condition]
```

Out[36]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |

```
In [37]: df[ df['calories'] > 70]
```

Out[37]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |

# Filtering Rows (2/3)

## *and (&)*

- We can also group conditions using the and operator.
- Symbol for and in pandas is &. It works the same way as and in Python.
- Note: Each condition should be in parentheses.

```
In [38]: df[ (df['calories'] > 70) & (df['protein'] < 4)]
```

Out[38]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |

# Filtering Rows (3/3)

## *or (|)*

- We can also group conditions using the or operator.
- Symbol for and in pandas is | It works the same way as or in Python.
- Note: Each condition should be in parentheses.

```
In [39]: df[ (df['calories'] > 70) | (df['protein'] > 3)]
```

Out[39]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |

# loc (1/4)

## *Indexing*

- loc is used to index/slice a group of rows and columns based on their labels.
- The first argument is the row label and the second argument is the column label.
- In the following example we index the first row and the first column.

```
In [79]: df
```

Out[79]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |

```
In [88]: df.loc[0, 'name']
Out[88]: '100% Bran'
```

# loc (2/4)

## *Indexing*

- If we pass a list of row and column labels, we get a DataFrame.
- In the following example, we index first row and first column, but we pass the labels as lists. We get a DataFrame.

```
In [79]: df
```

Out[79]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |

```
In [89]: df.loc[[0], ['name']]
```

Out[89]:

| | name |
|---|---|
| 0 | 100% Bran |

# loc (3/4)

## *Slicing*

- We can also slice rows and/or columns using the loc method.
- Both the start and stop index of a slice with loc are inclusive.
- In the following example, we slice the first 5 rows and the first 3 columns of the DataFrame. The result is a DataFrame.

```
In [41]: df
```

Out[41]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |
| 5 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 6 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 7 | Basic 4 | 130 | 3 | 25 | 37.038562 |
| 8 | Bran Chex | 90 | 2 | 25 | 49.120253 |
| 9 | Bran Flakes | 90 | 3 | 25 | 53.313813 |

```
In [42]: df.loc[0:4, 'name':'protein']
```

Out[42]:

| | name | calories | protein |
|---|---|---|---|
| 0 | 100% Bran | 70 | 4 |
| 1 | 100% Natural Bran | 120 | 3 |
| 2 | All-Bran | 70 | 4 |
| 3 | All-Bran with Extra Fiber | 50 | 4 |
| 4 | Almond Delight | 110 | 2 |

# loc (4/4)

## *Indexing and Slicing*

- We can index and slice simultaneously as well.
- In the following example we index rows and slice columns. The opposite is also possible.

```
In [43]: df.loc[[5, 8], 'name':'protein']
```

Out[43]:

| | name | calories | protein |
|---|---|---|---|
| **5** | Apple Cinnamon Cheerios | 110 | 2 |
| **8** | Bran Chex | 90 | 2 |

# Quiz Time

1. Consider the given DataFrame called df. What will be the result of the following loc command?

   df.loc[[0, 1], [‘name’]]

   a) DataFrame
   b) Series
   c) Cell

# Quiz Time

1. Consider the given DataFrame called df. What will be the result of the following loc command?

   df.loc[[0, 1], [‘name’]]

   a) DataFrame
   b) Series
   c) Cell

# iloc (1/4)

## *Indexing*

- iloc is used to index/slice a group of rows and columns.
- Iloc takes row and column positions as arguments and not their labels.
- The first argument is the row position and the second argument is the column position.
- In the following example we index the $10^{th}$ row and the third column. The result is a Series.

```
In [41]: df
```

Out[41]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |
| 5 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 6 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 7 | Basic 4 | 130 | 3 | 25 | 37.038562 |
| 8 | Bran Chex | 90 | 2 | 25 | 49.120253 |
| 9 | Bran Flakes | 90 | 3 | 25 | 53.313813 |

```
In [44]: df.iloc[9, 2]

Out[44]: 3
```

# iloc (2/4)

## *Indexing*

- If we pass a lsit of row and column positions, we get a DataFrame.
- In the following example, we index 10th row and third column, but we pass the positions as lists. We get a DataFrame.

```
In [45]: df.iloc[[9], [2]]
```

Out[45]:

| | protein |
|---|---|
| 9 | 3 |

# iloc (3/4)

## *Slicing*

- We can also slice rows and/or columns using the iloc method.
- We provide row and column positions for slicing using iloc.
- The start index of a slice with iloc is inclusive. However, the end index is exclusive.
- In the following example, we slice the first 5 rows and the first 3 columns of the DataFrame. The result is a DataFrame.

```
In [46]: df.iloc[0:5, 0:3]
```

Out[46]:

| | name | calories | protein |
|---|---|---|---|
| **0** | 100% Bran | 70 | 4 |
| **1** | 100% Natural Bran | 120 | 3 |
| **2** | All-Bran | 70 | 4 |
| **3** | All-Bran with Extra Fiber | 50 | 4 |
| **4** | Almond Delight | 110 | 2 |

# iloc (4/4)

## *Indexing and Slicing*

- We can index and slice simultaneously as well.
- In the following example we index rows and slice columns. The opposite is also possible.

```
In [47]: df.iloc[[0, 2, 4], 0:3]
```

Out[47]:

| | name | calories | protein |
|---|---|---|---|
| **0** | 100% Bran | 70 | 4 |
| **2** | All-Bran | 70 | 4 |
| **4** | Almond Delight | 110 | 2 |

# Quiz Time

1. Consider the given DataFrame called df. What will be the result of the following iloc command?

   df.loc[[0, 2], [2]]

   a) DataFrame
   b) Series
   c) Cell

1. The stop index in iloc slice is inclusive. Is this statement True or False?

   a) True
   b) False

# Quiz Time

1. Consider the given DataFrame called df. What will be the result of the following iloc command?

   df.loc[[0, 2], [2]]

   a) DataFrame
   b) Series
   c) Cell

1. The stop index in iloc slice is inclusive. Is this statement True or False?

   a) True
   b) False

# Adding and Deleting Rows and Columns (1/4)

## *Adding Rows*

- We can add more rows to our DataFrame using the loc method.
- If the row label does not exist, a new row with the specified label will be added at the end of the row.

```
In [24]: df
```

Out[24]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |

```
In [68]: df.loc[6] = ['Trix', 110, 1, 25, 27.753301]
         df
```

Out[68]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |
| 6 | Trix | 110 | 1 | 25 | 27.753301 |

# Adding and Deleting Rows and Columns (2/4)

## *Deleting Rows*

- We can delete rows from the DataFrame using drop() function by specifying axis=0 for rows.
- Provide the labels of the rows to be deleted as argument to the drop() function.
- Don’t forget to use inplace=True, otherwise the original DataFrame will remain unchanged.

```
In [69]: df.drop(2, axis=0, inplace=True)

In [70]: df
```

Out[70]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **6** | Trix | 110 | 1 | 25 | 27.753301 |

# Adding and Deleting Rows and Columns (3/4)

## *Adding Columns*

- To add a column to the DataFrame, we use the same notation as adding a key, value pair to a dictionary.
- Instead of the key, we provide column name in the square brackets, and then provide a list of values for that column.
- If no column with the given name exists, a new column with the specified name and values will be added to the DataFrame.

```
In [71]: df['My Column'] = ['A', 'B', 'C', 'D', 'E']
         df
```

Out[71]:

| | name | calories | protein | vitamins | rating | My Column |
|---|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 | A |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 | B |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 | C |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 | D |
| **6** | Trix | 110 | 1 | 25 | 27.753301 | E |

# Adding and Deleting Rows and Columns (4/4)

## *Deleting Columns*

- We can also delete columns of the DataFrame using drop() function by specifying axis=1 for columns.
- Provide the column names to be deleted as argument to the drop() function.
- Don’t forget to use inplace=True, otherwise the original DataFrame will remain unchanged.

```
In [72]: df.drop('My Column', axis=1, inplace=True)

In [73]: df
```

Out[73]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **6** | Trix | 110 | 1 | 25 | 27.753301 |

# Sorting Values (1/2)

## *Ascending*

- We can sort the values of a DataFrame with respect to  a column using the sort_values() function, which sorts the values in ascending order by default.
- If the values of the column are alphabets, the are sorted alphabetically.
- If the values of the column are numbers, they are sorted numerically.

```
In [74]: df.sort_values(by='calories')
```

Out[74]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **6** | Trix | 110 | 1 | 25 | 27.753301 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |

# Sorting Values (2/2)

## *Descending*

- To sort the values in descending order, we set ascending = False in the sort_values() function.

```
In [75]: df.sort_values(by='calories', ascending=False)
```

Out[75]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **6** | Trix | 110 | 1 | 25 | 27.753301 |
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |

# Exporting and Saving Pandas DataFrame

- To export a DataFrame as a csv file, use to_csv() function.
- If a file with the specified filename exists, it will be modified. Otherwise, a new file with the specified filename will be created.
- If you do not want to store index column in the csv file, you can set index_label=False in the to_csv() function.

```
In [76]: df.to_csv('myFile.csv', index_label=False)

In [77]: newDf = pd.read_csv('myFile.csv')
         newDf
```

Out[77]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **3** | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| **4** | Almond Delight | 110 | 2 | 25 | 34.384843 |
| **6** | Trix | 110 | 1 | 25 | 27.753301 |

# Concatenating DataFrames (1/3)

- We can concatenate two or more DataFrames together using pandas.concat() function.

First Data Frame

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |

Second Data Frame

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 5 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 6 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 7 | Basic 4 | 130 | 3 | 25 | 37.038562 |
| 8 | Bran Chex | 90 | 2 | 25 | 49.120253 |
| 9 | Bran Flakes | 90 | 3 | 25 | 53.313813 |
| 10 | Cap'n'Crunch | 120 | 1 | 25 | 18.042851 |

Resultant Data Frame

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |
| 3 | All-Bran with Extra Fiber | 50 | 4 | 25 | 93.704912 |
| 4 | Almond Delight | 110 | 2 | 25 | 34.384843 |
| 5 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 6 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 7 | Basic 4 | 130 | 3 | 25 | 37.038562 |
| 8 | Bran Chex | 90 | 2 | 25 | 49.120253 |
| 9 | Bran Flakes | 90 | 3 | 25 | 53.313813 |
| 10 | Cap'n'Crunch | 120 | 1 | 25 | 18.042851 |

# Concatenating DataFrames (2/3)

- We can also concatenate two or more DataFrames side-by-side each other.

First Data Frame

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| **1** | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| **2** | Basic 4 | 130 | 3 | 25 | 37.038562 |

Second Data Frame

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 |

| | name | calories | protein | vitamins | rating | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 100% Bran | 70 | 4 | 25 | 68.402973 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| **1** | 100% Natural Bran | 120 | 3 | 0 | 33.983679 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| **2** | All-Bran | 70 | 4 | 25 | 59.425505 | Basic 4 | 130 | 3 | 25 | 37.038562 |

Resultant Data Frame

# Concatenating DataFrames (3/3)

- To join two or more DataFrames side-by-side, use axis = 1 in the pandas.concat() function.

```
In [26]: df
```

Out[26]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 |

```
In [36]: df2
```

Out[36]:

| | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|
| 0 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 1 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 2 | Basic 4 | 130 | 3 | 25 | 37.038562 |

```
In [37]: pd.concat([df, df2], axis=1)
```

Out[37]:

| | name | calories | protein | vitamins | rating | name | calories | protein | vitamins | rating |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 100% Bran | 70 | 4 | 25 | 68.402973 | Apple Cinnamon Cheerios | 110 | 2 | 25 | 29.509541 |
| 1 | 100% Natural Bran | 120 | 3 | 0 | 33.983679 | Apple Jacks | 110 | 2 | 25 | 33.174094 |
| 2 | All-Bran | 70 | 4 | 25 | 59.425505 | Basic 4 | 130 | 3 | 25 | 37.038562 |

# groupby() (1/7)

- groupby() function is used to group DataFrame based on Series.
  - The DataFrame is splitted into groups.
  - An aggregate function is applied to each column of the splitted DataFrame.
  - Results are combined together.
- Consider the following DataFrame.

| | Gender | Score |
|---|---|---|
| 0 | female | 85 |
| 1 | male | 88 |
| 2 | female | 95 |
| 3 | male | 80 |

# groupby() (2/7)

- The ‘Gender’ column contains two values, male and female.
- Let’s split our DataFrame into two parts based on ‘Gender’ column;
  - First part will contain the rows where Gender = male
  - Second part will contain the rows where Gender = female

| | Gender | Score |
|---|---|---|
| **0** | female | 85 |
| **2** | female | 95 |

| | Gender | Score |
|---|---|---|
| **1** | male | 88 |
| **3** | male | 80 |

# groupby() (3/7)

- If we find the mean score of both the genders, this is what we get.

| Gender | Score |
|---|---|
| female | 90 |

| Gender | Score |
|---|---|
| male | 84 |

# groupby() (4/7)

- Let’s combine the two results together. This is what we get.

| Gender | Score |
|---|---|
| female | 90 |
| male | 84 |

# groupby() (5/7)

- The groupby() function works exactly the same way, except that it makes things easier for us.
- In the given example, we group our DataFrame on the basis of ‘Gender’ column, and then apply the aggregate function mean() on it.

```
In [55]: df
```

Out[55]:

| | Gender | Score |
|---|---|---|
| 0 | female | 85 |
| 1 | male | 88 |
| 2 | female | 95 |
| 3 | male | 80 |

```
In [56]: df.groupby(x['Gender']).mean()
```

Out[56]:

| | Score |
|---|---|
| Gender | |
| female | 90 |
| male | 84 |

# groupby() (6/7)

- Note that aggregate functions are applied automatically on all the columns of the DataFrame except the one used to group the DataFrame.

```
In [72]: df
```

Out[72]:

| | Gender | Math | English |
|---|---|---|---|
| 0 | female | 85 | 80 |
| 1 | male | 88 | 88 |
| 2 | female | 95 | 92 |
| 3 | male | 80 | 95 |

```
In [73]: df.groupby(x['Gender']).mean()
```

Out[73]:

| Gender | Math | English |
|---|---|---|
| female | 90.0 | 86.0 |
| male | 84.0 | 91.5 |

# groupby() (7/7)

- The common aggregate functions are;
  - mean()
  - sum()
  - max()
  - min()
  - median()
  - count()
  - std() (standard deviation)

# Resources

- https://www.geeksforgeeks.org/python-pandas-dataframe/?ref=lbp
- https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html
- https://www.w3schools.com/python/pandas/pandas_dataframes.asp