Data Statistics

Denaco Analytics

The recent success of machine learning algorithms is partly due to the huge amounts of data that we have available to train our algorithms on. However, when it comes to data, quantity is not the only thing that matters, the quality of your data is just as important. It often happens that large datasets don’t come ready to be fed into your learning algorithms. More often than not, large datasets will often have missing values, outliers, incorrect values, etc… Having data with a lot of missing or bad values, for example, is not going to allow your machine learning algorithms to perform well. Therefore, one very important step in machine learning is to look at your data first and make sure it is well suited for your training algorithm by doing some basic data analysis. This is where Pandas come in. Pandas Series and DataFrames are designed for fast data analysis and manipulation, as well as being flexible and easy to use. Below are just a few features that makes Pandas an excellent package for data analysis:

1. Allows the use of labels for rows and columns
2. Can calculate rolling statistics on time series data
3. Easy handling of NaN values
4. Is able to load data of different formats into DataFrames
5. Can join and merge different datasets together
6. It integrates with NumPy and Matplotlib

For these and other reasons, Pandas DataFrames have become one of the most commonly used Pandas object for data analysis in Python.

Creating Pandas Series

Pandas Series

A Pandas series is a one-dimensional array-like object that can hold many data types, such as numbers or strings, and has an option to provide axis labels. Difference between NumPy ndarrays and Pandas Series

1. One of the main differences between Pandas Series and NumPy ndarrays is that you can assign an index label to each element in the Pandas Series. In other words, you can name the indices of your Pandas Series anything you want.
2. Another big difference between Pandas Series and NumPy ndarrays is that Pandas Series can hold data of different data types.

Let's start by importing Pandas into Python. It has become a convention to import Pandas as `pd`, therefore, you can import Pandas by typing the following command in your Jupyter notebook:

```
import pandas as pd
```

Let's begin by creating a Pandas Series. You can create Pandas Series by using the command `pd.Series(data, index)`, where index is a list of `index` labels. Let's use a Pandas Series to store a grocery list. We will use the food items as index labels and the quantity we need to buy of each item as our data.

Example 1 - Create a Series

In [3]:
```
# We import Pandas as pd into Python
import pandas as pd

# We create a Pandas Series that stores a grocery list
groceries = pd.Series(data = [30, 6, 'Yes', 'No'], index = ['eggs', 'apples', 'milk', 'bread'])

# We display the Groceries Pandas Series
groceries
```

Out[3]:
```
eggs       30
apples      6
milk      Yes
bread      No
dtype: object
```

We see that Pandas Series are displayed with the indices in the first column and the data in the second column. Notice that the data is not indexed 0 to 3 but rather it is indexed with the names of the food we put in, namely eggs, apples, etc... Also, notice that the data in our Pandas Series has both integers and strings.

Just like NumPy ndarrays, Pandas Series have attributes that allow us to get information from the series in an easy way. Let's see some of them:

Example 2 - Print attributes - shape, ndim,and size

In [4]:
```
# We print some information about Groceries
print('Groceries has shape:', groceries.shape)
print('Groceries has dimension:', groceries.ndim)
print('Groceries has a total of', groceries.size, 'elements')
```

```
Groceries has shape: (4,)
Groceries has dimension: 1
Groceries has a total of 4 elements
```

We can also print the index labels and the data of the Pandas Series separately. This is useful if you don't happen to know what the index labels of the Pandas Series are.

Example 3 - Print attributes - values, and index

In [3]:
```
# We print the index and data of Groceries
print('The data in Groceries is:', groceries.values)
print('The index of Groceries is:', groceries.index)
```

```
The data in Groceries is: [30 6 'Yes' 'No']
The index of Groceries is: Index(['eggs', 'apples', 'milk', 'bread'], dtype='object')
```

If you are dealing with a very large Pandas Series and if you are not sure whether an index label exists, you can check by using the in command

Example 4 - Check if an index is available in the given Series

In [6]:
```
# We check whether bananas is a food item (an index) in Groceries
x = 'bananas' in groceries

# We check whether bread is a food item (an index) in Groceries
y = 'bread' in groceries

# We print the results
print('Is bananas an index label in Groceries:', x)
print('Is bread an index label in Groceries:', y)
```

```
Is bananas an index label in Groceries: False
Is bread an index label in Groceries: True
```

Now let's look at how we can access or modify elements in a Pandas Series. One great advantage of Pandas Series is that it allows us to access data in many different ways. Elements can be accessed using index labels or numerical indices inside square brackets, [ ], similar to how we access elements in NumPy ndarrays. Since we can use numerical indices, we can use both positive and negative integers to access data from the beginning or from the end of the Series, respectively. Since we can access elements in various ways, in order to remove any ambiguity to whether we are referring to an index label or numerical index, Pandas Series have two attributes, `.loc` and `.iloc` to explicitly state what we mean. The attribute `.loc` stands for location and it is used to explicitly state that we are using a labeled index. Similarly, the attribute `.iloc` stands for integer location and it is used to explicitly state that we are using a numerical index. Let's see some examples:

Example 1. Access elements using index labels

In [7]:
```
# We access elements in Groceries using index labels:

# We use a single index label
print('How many eggs do we need to buy:', groceries['eggs'])
print()

# we can access multiple index labels
print('Do we need milk and bread:\n', groceries[['milk', 'bread']])
print()

# we use loc to access multiple index labels
print('How many eggs and apples do we need to buy:\n', groceries.loc[['eggs', 'apples']])
print()

# We access elements in Groceries using numerical indices:

# we use multiple numerical indices
print('How many eggs and apples do we need to buy:\n',  groceries[[0, 1]])
print()

# We use a negative numerical index
print('Do we need bread:\n', groceries[[-1]])
print()

# We use a single numerical index
print('How many eggs do we need to buy:', groceries[0])
print()
# we use iloc to access multiple numerical indices
print('Do we need milk and bread:\n', groceries.iloc[[2, 3]])
```

```
How many eggs do we need to buy: 30

Do we need milk and bread:
 milk     Yes
bread     No
dtype: object

How many eggs and apples do we need to buy:
 eggs      30
apples     6
dtype: object

How many eggs and apples do we need to buy:
 eggs      30
apples     6
dtype: object

Do we need bread:
 bread    No
dtype: object

How many eggs do we need to buy: 30

Do we need milk and bread:
 milk     Yes
bread     No
dtype: object
```

In [8]:

```python
# Example 2. Mutate elements using index labels

# We display the original grocery list
print('Original Grocery List:\n', groceries)

# We change the number of eggs to 2
groceries['eggs'] = 2

# We display the changed grocery list
print()
print('Modified Grocery List:\n', groceries)
```

```
Original Grocery List:
 eggs       30
apples      6
milk      Yes
bread      No
dtype: object

Modified Grocery List:
 eggs        2
apples      6
milk      Yes
bread      No
dtype: object
```

We can also delete items from a Pandas Series by using the `.drop()` method. The `Series.drop(label)` method removes the given label from the given Series. We should note that the `Series.drop(label)` method drops elements from the Series out-of-place, meaning that it doesn't change the original Series being modified. Let's see how this works:

Example 3. Delete elements out-of-place using `drop()`

In [9]:

```python
# We display the original grocery list
print('Original Grocery List:\n', groceries)

# We remove apples from our grocery list. The drop function removes elements out of place
print()
print('We remove apples (out of place):\n', groceries.drop('apples'))

# When we remove elements out of place the original Series remains intact. To see this
# we display our grocery list again
print()
print('Grocery List after removing apples out of place:\n', groceries)
```

```
Original Grocery List:
 eggs        2
apples      6
milk      Yes
bread      No
dtype: object

We remove apples (out of place):
 eggs       2
milk     Yes
bread     No
dtype: object

Grocery List after removing apples out of place:
 eggs        2
apples      6
milk      Yes
bread      No
dtype: object
```

Arithmetic Operations on Pandas Series

In [10]:

```
# We create a Pandas Series that stores a grocery list of just fruits
fruits= pd.Series(data = [10, 6, 3,], index = ['apples', 'oranges', 'bananas'])

# We display the fruits Pandas Series
fruits
```

Out[10]:

```
apples     10
oranges     6
bananas     3
dtype: int64
```

In [11]:

```
# Example 1. Element-wise basic arithmetic operations

# We print fruits for reference
print('Original grocery list of fruits:\n ', fruits)

# We perform basic element-wise operations using arithmetic symbols
print()
print('fruits + 2:\n', fruits + 2) # We add 2 to each item in fruits
print()
print('fruits - 2:\n', fruits - 2) # We subtract 2 to each item in fruits
print()
print('fruits * 2:\n', fruits * 2) # We multiply each item in fruits by 2
print()
print('fruits / 2:\n', fruits / 2) # We divide each item in fruits by 2
print()
```

```
Original grocery list of fruits:
  apples     10
oranges     6
bananas     3
dtype: int64

fruits + 2:
 apples     12
oranges     8
bananas     5
dtype: int64

fruits - 2:
 apples     8
oranges    4
bananas    1
dtype: int64

fruits * 2:
 apples     20
oranges    12
bananas     6
dtype: int64

fruits / 2:
 apples     5.0
oranges    3.0
bananas    1.5
dtype: float64
```

In [13]:

```
# Example 2. Use mathematical functions from NumPy to operate on Series

# We import NumPy as np to be able to use the mathematical functions
import numpy as np

# We print fruits for reference
print('Original grocery list of fruits:\n', fruits)
```

```python
# We apply different mathematical functions to all elements of fruits
print()
print('EXP(X) = \n', np.exp(fruits))
print()
print('SQRT(X) =\n', np.sqrt(fruits))
print()
print('POW(X,2) =\n',np.power(fruits,2)) # We raise all elements of fruits to the power of 2
```

```
Original grocery list of fruits:
 apples     10
oranges     6
bananas     3
dtype: int64

EXP(X) =
 apples     22026.465795
oranges      403.428793
bananas       20.085537
dtype: float64

SQRT(X) =
 apples     3.162278
oranges    2.449490
bananas    1.732051
dtype: float64

POW(X,2) =
 apples     100
oranges     36
bananas      9
dtype: int64
```

In [14]:
```python
# Example 3. Perform arithmetic operations on selected elements

# We print fruits for reference
print('Original grocery list of fruits:\n ', fruits)
print()

# We add 2 only to the bananas
print('Amount of bananas + 2 = ', fruits['bananas'] + 2)
print()

# We subtract 2 from apples
print('Amount of apples - 2 = ', fruits.iloc[0] - 2)
print()

# We multiply apples and oranges by 2
print('We double the amount of apples and oranges:\n', fruits[['apples', 'oranges']] * 2)
print()

# We divide apples and oranges by 2
print('We half the amount of apples and oranges:\n', fruits.loc[['apples', 'oranges']] / 2)
```

```
Original grocery list of fruits:
  apples     10
oranges     6
bananas     3
dtype: int64

Amount of bananas + 2 =  5

Amount of apples - 2 =  8

We double the amount of apples and oranges:
 apples     20
oranges    12
dtype: int64

We half the amount of apples and oranges:
 apples     5.0
oranges    3.0
dtype: float64
```

In [15]:
```python
#Example 4. Perform multiplication on a Series having integer and string elements

# We multiply our grocery list by 2
groceries * 2
```

Out[15]:
```
eggs            4
apples         12
milk       YesYes
bread        NoNo
dtype: object
```

As we can see, in this case, since we multiplied by `2`, Pandas doubles the data of each item including the strings. Pandas can do this because the multiplication operation `*` is defined both for numbers and strings. If you were to apply an operation that was valid for numbers but not strings, say for instance, `/` you will get an error. So when you have mixed data types in your Pandas Series make sure the arithmetic operations are valid on all the data types of your elements.

Creating Pandas DataFrames

Pandas DataFrames are two-dimensional data structures with labeled rows and columns, that can hold many data types. If you are familiar with Excel, you can think of Pandas DataFrames as being similar to a spreadsheet. We can create Pandas DataFrames manually or by loading data from a file. In this lesson, we will start by learning how to create Pandas DataFrames manually from dictionaries, and later we will see how we can load data into a DataFrame from a data file.

Create a DataFrame manually

We will start by creating a DataFrame manually from a dictionary of Pandas Series. It is a two-step process:

1. The first step is to create the dictionary of Pandas Series.
2. After the dictionary is created we can then pass the dictionary to the `pd.DataFrame()` function.

We will create a dictionary that contains items purchased by two people, Alice and Bob, on an online store. The Pandas Series will use the price of the items purchased as data, and the purchased items will be used as the index labels to the Pandas Series. Let's see how this done in code:

In [16]:
```
# We import Pandas as pd into Python
import pandas as pd

# We create a dictionary of Pandas Series
items = {'Bob' : pd.Series(data = [245, 25, 55], index = ['bike', 'pants', 'watch']),
         'Alice' : pd.Series(data = [40, 110, 500, 45], index = ['book', 'glasses', 'bike', 'pants'])}

# We print the type of items to see that it is a dictionary
print(type(items))
```

```
<class 'dict'>
```

Now that we have a dictionary, we are ready to create a DataFrame by passing it to the `pd.DataFrame()` function. We will create a DataFrame that could represent the shopping carts of various users, in this case we have only two users, Alice and Bob.

Example 1. Create a DataFrame using a dictionary of Series.

In [17]:
```
# We create a Pandas DataFrame by passing it a dictionary of Pandas Series
shopping_carts = pd.DataFrame(items)

# We display the DataFrame
shopping_carts
```

Out[17]:

	Bob	Alice
bike	245.0	500.0
book	NaN	40.0
glasses	NaN	110.0
pants	25.0	45.0
watch	55.0	NaN

1. We see that DataFrames are displayed in tabular form, much like an Excel spreadsheet, with the labels of rows and columns in bold.
2. Also, notice that the row labels of the DataFrame are built from the union of the index labels of the two Pandas Series we used to construct the dictionary. And the column labels of the DataFrame are taken from the keys of the dictionary.
3. Another thing to notice is that the columns are arranged alphabetically and not in the order given in the dictionary. We will see later that this won't happen when we load data into a DataFrame from a data file.
4. The last thing we want to point out is that we see some `NaN` values appear in the DataFrame. `NaN` stands for Not a Number, and is Pandas way of indicating that it doesn't have a value for that particular row and column index. For example, if we look at the column of Alice, we see that it has NaN in the watch index. You can see why this is the case by looking at the dictionary we created at the beginning. We clearly see that the dictionary has no item for Alice labeled watches. So whenever a DataFrame is created, if a particular column doesn't have values for a particular row index, Pandas will put a NaN value there.
   A. If we were to feed this data into a machine learning algorithm we will have to remove these `NaN` values first. In a later lesson, we will learn how to deal with `NaN` values and clean our data. For now, we will leave these values in our DataFrame.

In the example above, we created a Pandas DataFrame from a dictionary of Pandas Series that had clearly defined indexes. If we don't provide index labels to the Pandas Series, Pandas will use numerical row indexes when it creates the DataFrame. Let's see an example:

Example 2. DataFrame assigns the numerical row indexes by default.

In [18]:
```
# We create a dictionary of Pandas Series without indexes
```

```
data = {'Bob' : pd.Series([245, 25, 55]),
        'Alice' : pd.Series([40, 110, 500, 45])}

# We create a DataFrame
df = pd.DataFrame(data)

# We display the DataFrame
df
```

Out[18]:

	Bob	Alice
0	245.0	40
1	25.0	110
2	55.0	500
3	NaN	45

We can see that Pandas indexes the rows of the DataFrame starting from 0, just like NumPy indexes ndarrays.

Now, just like with Pandas Series we can also extract information from DataFrames using attributes. Let's print some information from our `shopping_carts` DataFrame

Example 3. Demonstrate a few attributes of DataFrame

In [19]:
```
# We print some information about shopping_carts
print('shopping_carts has shape:', shopping_carts.shape)
print('shopping_carts has dimension:', shopping_carts.ndim)
print('shopping_carts has a total of:', shopping_carts.size, 'elements')
print()
print('The data in shopping_carts is:\n', shopping_carts.values)
print()
print('The row index in shopping_carts is:', shopping_carts.index)
print()
print('The column index in shopping_carts is:', shopping_carts.columns)
```

```
shopping_carts has shape: (5, 2)
shopping_carts has dimension: 2
shopping_carts has a total of: 10 elements

The data in shopping_carts is:
 [[245. 500.]
 [ nan  40.]
 [ nan 110.]
 [ 25.  45.]
 [ 55.  nan]]

The row index in shopping_carts is: Index(['bike', 'book', 'glasses', 'pants', 'watch'], dtype='object')

The column index in shopping_carts is: Index(['Bob', 'Alice'], dtype='object')
```

When creating the `shopping_carts` DataFrame we passed the entire dictionary to the `pd.DataFrame()` function. However, there might be cases when you are only interested in a subset of the data. Pandas allows us to select which data we want to put into our DataFrame by means of the keywords `columns` and `index`. Let's see some examples:

In [20]:
```
# We Create a DataFrame that only has Bob's data
bob_shopping_cart = pd.DataFrame(items, columns=['Bob'])

# We display bob_shopping_cart
bob_shopping_cart
```

Out[20]:

	Bob
bike	245
pants	25
watch	55

In [21]:
```
# Example 4. Selecting specific rows of a DataFrame

# We Create a DataFrame that only has selected items for both Alice and Bob
sel_shopping_cart = pd.DataFrame(items, index = ['pants', 'book'])

# We display sel_shopping_cart
sel_shopping_cart
```

Out[21]:

	Bob	Alice
pants	25.0	45
book	NaN	40

In [26]:
```
#Example 5. Selecting specific columns of a DataFrame
```

```
# We Create a DataFrame that only has selected items for Alice
alice_sel_shopping_cart = pd.DataFrame(items, index = ['glasses', 'bike'], columns = ['Alice'])

# We display alice_sel_shopping_cart
alice_sel_shopping_cart
```

Out[26]:

	Alice
glasses	110
bike	500

You can also manually create DataFrames from a dictionary of lists (arrays). The procedure is the same as before, we start by creating the dictionary and then passing the dictionary to the `pd.DataFrame()` function. In this case, however, all the lists (arrays) in the dictionary must be of the same length. Let' see an example:

Example 6. Create a DataFrame using a dictionary of lists

In [27]:

```
# We create a dictionary of lists (arrays)
data = {'Integers' : [1,2,3],
        'Floats' : [4.5, 8.2, 9.6]}

# We create a DataFrame
df = pd.DataFrame(data)

# We display the DataFrame
df
```

Out[27]:

	Integers	Floats
0	1	4.5
1	2	8.2
2	3	9.6

In [28]:

```
#Example 7. Create a DataFrame using a dictionary of lists, and custom row-indexes (labels)

# We create a dictionary of lists (arrays)
data = {'Integers' : [1,2,3],
        'Floats' : [4.5, 8.2, 9.6]}

# We create a DataFrame and provide the row index
df = pd.DataFrame(data, index = ['label 1', 'label 2', 'label 3'])

# We display the DataFrame
df
```

Out[28]:

	Integers	Floats
label 1	1	4.5
label 2	2	8.2
label 3	3	9.6

In [29]:

```
#Example 8. Create a DataFrame using a of list of dictionaries

# We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35},
          {'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]

# We create a DataFrame
store_items = pd.DataFrame(items2)

# We display the DataFrame
store_items
```

Out[29]:

	bikes	pants	watches	glasses
0	20	30	35	NaN
1	15	5	10	50.0

In [30]:

```
#Example 9. Create a DataFrame using a of list of dictionaries, and custom row-indexes (labels)

# We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35},
          {'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5}]

# We create a DataFrame  and provide the row index
store_items = pd.DataFrame(items2, index = ['store 1', 'store 2'])

# We display the DataFrame
store_items
```

Out[30]:

	bikes	pants	watches	glasses
store 1	20	30	35	NaN
store 2	15	5	10	50.0

Accessing Elements in Pandas DataFrames

In [31]:

```
#Example 1. Access elements using labels

# We print the store_items DataFrame
print(store_items)

# We access rows, columns and elements using labels
print()
print('How many bikes are in each store:\n', store_items[['bikes']])
print()
print('How many bikes and pants are in each store:\n', store_items[['bikes', 'pants']])
print()
print('What items are in Store 1:\n', store_items.loc[['store 1']])
print()
print('How many bikes are in Store 2:', store_items['bikes']['store 2'])
```

```
         bikes  pants  watches  glasses
store 1     20     30       35      NaN
store 2     15      5       10     50.0

How many bikes are in each store:
          bikes
store 1     20
store 2     15

How many bikes and pants are in each store:
          bikes  pants
store 1     20     30
store 2     15      5

What items are in Store 1:
          bikes  pants  watches  glasses
store 1     20     30       35      NaN

How many bikes are in Store 2: 15
```

In [32]:

```
#Example 2. Add a column to an existing DataFrame

# We add a new column named shirts to our store_items DataFrame indicating the number of
# shirts in stock at each store. We will put 15 shirts in store 1 and 2 shirts in store 2
store_items['shirts'] = [15,2]

# We display the modified DataFrame
store_items
```

Out[32]:

	bikes	pants	watches	glasses	shirts
store 1	20	30	35	NaN	15
store 2	15	5	10	50.0	2

In [33]:

```
#Example 3. Add a new column based on the arithmetic operation between existing columns of a DataFrame

# We make a new column called suits by adding the number of shirts and pants
store_items['suits'] = store_items['pants'] + store_items['shirts']

# We display the modified DataFrame
store_items
```

Out[33]:

	bikes	pants	watches	glasses	shirts	suits
store 1	20	30	35	NaN	15	45
store 2	15	5	10	50.0	2	7

In [34]:

```
#Example 4 a. Create a row to be added to the DataFrame

# We create a dictionary from a list of Python dictionaries that will contain the number of different items at
new_items = [{'bikes': 20, 'pants': 30, 'watches': 35, 'glasses': 4}]

# We create new DataFrame with the new_items and provide and index labeled store 3
new_store = pd.DataFrame(new_items, index = ['store 3'])

# We display the items at the new store
new_store
```

Out[34]:

	bikes	pants	watches	glasses
store 3	20	30	35	4

In [37]:
```
#Example 4 b. Append the row to the DataFrame

# We append store 3 to our store_items DataFrame
store_items = store_items.append(new_store)

# We display the modified DataFrame
store_items
```

```
C:\Users\Isaac\AppData\Local\Temp\ipykernel_816\1402328061.py:4: FutureWarning: The frame.append method is depr
ecated and will be removed from pandas in a future version. Use pandas.concat instead.
  store_items = store_items.append(new_store)
```

Out[37]:

	bikes	pants	watches	glasses	shirts	suits
store 1	20	30	35	NaN	15.0	45.0
store 2	15	5	10	50.0	2.0	7.0
store 3	20	30	35	4.0	NaN	NaN
store 3	20	30	35	4.0	NaN	NaN

In [41]:
```
#Example 6. Add new column at a specific location

# We insert a new column with label shoes right before the column with numerical index 4
store_items.insert(4, 'shoes', [8,5,0])

# we display the modified DataFrame
store_items
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_816\3392358348.py in <module>
      2 
      3 # We insert a new column with label shoes right before the column with numerical index 4
----> 4 store_items.insert(4, 'shoes', [8,5,0])
      5 
      6 # we display the modified DataFrame

~\anaconda3\lib\site-packages\pandas\core\frame.py in insert(self, loc, column, value, allow_duplicates)
   4445             raise TypeError("loc must be int")
   4446 
-> 4447         value = self._sanitize_column(value)
   4448         self._mgr.insert(loc, column, value)
   4449 

~\anaconda3\lib\site-packages\pandas\core\frame.py in _sanitize_column(self, value)
   4536 
   4537         if is_list_like(value):
-> 4538             com.require_length_match(value, self.index)
   4539         return sanitize_array(value, self.index, copy=True, allow_2d=True)
   4540 

~\anaconda3\lib\site-packages\pandas\core\common.py in require_length_match(data, index)
    555     """
    556     if len(data) != len(index):
--> 557         raise ValueError(
    558             "Length of values "
    559             f"({len(data)}) "

ValueError: Length of values (3) does not match length of index (4)
```

In [43]:
```
#Example 8. Delete multiple columns from a DataFrame

# We remove the watches and shoes columns
store_items = store_items.drop(['watches', 'pants'], axis = 1)

# we display the modified DataFrame
store_items
```

Out[43]:

	bikes	glasses	shirts	suits
store 1	20	NaN	15.0	45.0
store 2	15	50.0	2.0	7.0
store 3	20	4.0	NaN	NaN
store 3	20	4.0	NaN	NaN

In [44]:
```
#Example 9. Delete rows from a DataFrame

# We remove the store 2 and store 1 rows
store_items = store_items.drop(['store 2', 'store 1'], axis = 0)
```

```
# we display the modified DataFrame
store_items
```

Out[44]:

	bikes	glasses	shirts	suits
store 3	20	4.0	NaN	NaN
store 3	20	4.0	NaN	NaN

In [45]:

```
#Example 10. Modify the column label

# We change the column label bikes to hats
store_items = store_items.rename(columns = {'bikes': 'hats'})

# we display the modified DataFrame
store_items
```

Out[45]:

	hats	glasses	shirts	suits
store 3	20	4.0	NaN	NaN
store 3	20	4.0	NaN	NaN

In [46]:

```
#Example 11. Modify the row label

# We change the row label from store 3 to last store
store_items = store_items.rename(index = {'store 3': 'last store'})

# we display the modified DataFrame
store_items
```

Out[46]:

	hats	glasses	shirts	suits
last store	20	4.0	NaN	NaN
last store	20	4.0	NaN	NaN

In [48]:

```
#Example 12. Use existing column values as row-index

# We change the row index to be the data in the pants column
store_items = store_items.set_index('hats')

# we display the modified DataFrame
store_items
```

Out[48]:

	glasses	shirts	suits
hats			
20	4.0	NaN	NaN
20	4.0	NaN	NaN

Dealing with NaN

As mentioned earlier, before we can begin training our learning algorithms with large datasets, we usually need to clean the data first. This means we need to have a method for detecting and correcting errors in our data. While any given dataset can have many types of bad data, such as outliers or incorrect values, the type of bad data we encounter almost always is missing values. As we saw earlier, Pandas assigns `NaN` values to missing data. In this lesson we will learn how to detect and deal with `NaN` values.

We will begin by creating a DataFrame with some `NaN` values in it.

Example 1. Create a DataFrame

In [55]:

```
# We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35, 'shirts': 15, 'shoes':8, 'suits':45},
{'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5, 'shirts': 2, 'shoes':5, 'suits':7},
{'bikes': 20, 'pants': 30, 'watches': 35, 'glasses': 4, 'shoes':10}]

# We create a DataFrame  and provide the row index
store_items = pd.DataFrame(items2, index = ['store 1', 'store 2', 'store 3'])

# We display the DataFrame
store_items
```

Out[55]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	NaN
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	NaN	10	NaN	4.0

We can clearly see that the DataFrame we created has 3 NaN values: one in store 1 and two in store 3. However, in cases where we

We can clearly see that the DataFrame we created has 3 NaN values: one in store 1 and two in store 3. However, in cases where we load very large datasets into a DataFrame, possibly with millions of items, the number of `NaN` values is not easily visualized. For these cases, we can use a combination of methods to count the number of `NaN` values in our data. The following example combines the `.isnull()` and the `sum()` methods to count the number of NaN values in our DataFrame.

Example 2 a. Count the total NaN values

In [29]:

```python
# We count the number of NaN values in store_items
x =  store_items.isnull().sum().sum()

# We print x
print('Number of NaN values in our DataFrame:', x)
```

```
Number of NaN values in our DataFrame: 3
```

In [30]:

```python
# Example 2 c. Count NaN down the column.
x =  store_items.isnull().sum()

x
```

Out[30]:

```
bikes      0
pants      0
watches    0
shirts     1
shoes      0
suits      1
glasses    1
dtype: int64
```

In [31]:

```python
# We count the number of NaN values in store_items
x =  store_items.isnull()

x
```

Out[31]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	False	False	False	False	False	False	True
store 2	False	False	False	False	False	False	False
store 3	False	False	False	True	False	True	False

In the above example, the `.isnull()` method returns a Boolean DataFrame of the same size as store_items and indicates with `True` the elements that have `NaN` values and with `False` the elements that are not. Let's see an example:

Example 2 b. Return boolean True/False for each element if it is a NaN

In [32]:

```python
store_items.isnull()
```

Out[32]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	False	False	False	False	False	False	True
store 2	False	False	False	False	False	False	False
store 3	False	False	False	True	False	True	False

Instead of counting the number of `NaN` values we can also do the opposite, we can count the number of `non-NaN` values. We can do this by using the `.count()` method as shown below:

Example 3. Count the total non-NaN values

In [33]:

```python
# We print the number of non-NaN values in our DataFrame
print()
print('Number of non-NaN values in the columns of our DataFrame:\n', store_items.count())
```

```
Number of non-NaN values in the columns of our DataFrame:
 bikes      3
pants      3
watches    3
shirts     2
shoes      3
suits      2
glasses    2
dtype: int64
```

Eliminating NaN Values

Now that we learned how to know if our dataset has any `NaN` values in it, the next step is to decide what to do with them. In general, we have two options, we can either delete or replace the NaN values. In the following examples, we will show you how to do both.

We will start by learning how to eliminate rows or columns from our DataFrame that contain any NaN values. The `.dropna(axis)` method eliminates any rows with `NaN` values when `axis = 0` is used and will eliminate any columns with NaN values when `axis =`

`1` is used.

> **Tip:** Remember, you learned that you can read **axis = 0** as **down** and **axis = 1** as **across** the given Numpy ndarray or Pandas dataframe

Let's see some examples.

Example 4. Drop rows having NaN values

In [41]:
```python
# We drop any rows with NaN values
store_items.dropna(axis=0)
```

Out[41]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 2	15	5	10	2.0	5	7.0	50.0

Example 5. Drop columns having NaN values

In [42]:
```python
# We drop any columns with NaN values
store_items.dropna(axis=1)
```

Out[42]:

	bikes	pants	watches	shoes
store 1	20	30	35	8
store 2	15	5	10	5
store 3	20	30	35	10

Substituting NaN Values

Now, instead of eliminating `NaN` values, we can replace them with suitable values. We could choose for example to replace all `NaN` values with the value 0. We can do this by using the `.fillna()` method as shown below.

Example 6. Replace NaN with 0

In [45]:
```python
# We replace all NaN values with 0
store_items.fillna(0)
```

Out[45]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	0.0
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	0.0	10	0.0	4.0

Example 7. Forward fill NaN values down (axis = 0) the dataframe

In [49]:
```python
# We replace NaN values with the previous value in the column
store_items.fillna(method = 'ffill', axis = 0)
```

Out[49]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	NaN
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	2.0	10	7.0	4.0

Notice that the two `NaN` values in store 3 have been replaced with previous values in their columns. However, notice that the `NaN` value in store 1 didn't get replaced. That's because there are no previous values in this column, since the `NaN` value is the first value in that column. However, if we do forward fill using the previous row values, this won't happen. Let's take a look:

Example 8. Forward fill NaN values across (axis = 1) the dataframe

In [47]:
```python
# We replace NaN values with the previous value in the row
store_items.fillna(method = 'ffill', axis = 1)
```

Out[47]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20.0	30.0	35.0	15.0	8.0	45.0	45.0
store 2	15.0	5.0	10.0	2.0	5.0	7.0	50.0
store 3	20.0	30.0	35.0	35.0	10.0	10.0	4.0

Example 9. Backward fill NaN values down (axis = 0) the dataframe

In [50]:
```python
# We replace NaN values with the next value in the column
```

```
In [58]: # We replace NaN values with the next value in the column
store_items.fillna(method = 'backfill', axis = 0)
```

Out[58]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	50.0
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	NaN	10	NaN	4.0

Notice! Notice that the **.fillna()** method replaces (fills) the **NaN** values out of place. This means that the original DataFrame is not modified. You can always replace the **NaN** values in place by setting the keyword **inplace = True** inside the **fillna()** function.

```
In [59]: store_items
```

Out[59]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	NaN
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	NaN	10	NaN	4.0

```
In [60]: # We replace NaN values with the next value in the column
store_items.fillna(0, inplace = True)
```

```
In [61]: store_items
```

Out[61]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	0.0
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	0.0	10	0.0	4.0

```
In [63]: # We create a list of Python dictionaries
items2 = [{'bikes': 20, 'pants': 30, 'watches': 35, 'shirts': 15, 'shoes':8, 'suits':45},
{'watches': 10, 'glasses': 50, 'bikes': 15, 'pants':5, 'shirts': 2, 'shoes':5, 'suits':7},
{'bikes': 20, 'pants': 30, 'watches': 35, 'glasses': 4, 'shoes':10}]

# We create a DataFrame  and provide the row index
store_items = pd.DataFrame(items2, index = ['store 1', 'store 2', 'store 3'])

# We display the DataFrame
store_items
```

Out[63]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	NaN
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	NaN	10	NaN	4.0

We can also choose to replace `NaN` values by using different interpolation methods. For example, the `.interpolate(method = 'linear', axis)` method will use linear interpolation to replace `NaN` values using the values along the given axis. Let's see some examples:

Example 11. Interpolate (estimate) NaN values down (axis = 0) the dataframe

```
In [64]: # We replace NaN values by using linear interpolation using column values
store_items.interpolate(method = 'linear', axis = 0)
```

Out[64]:

	bikes	pants	watches	shirts	shoes	suits	glasses
store 1	20	30	35	15.0	8	45.0	NaN
store 2	15	5	10	2.0	5	7.0	50.0
store 3	20	30	35	2.0	10	7.0	4.0

```
In [88]: import pandas as pd
import numpy as np

# DO NOT CHANGE THE VARIABLE NAMES

# Set the precision of our dataframes to one decimal place.
pd.set_option('display.precision', 1)

# Create a Pandas DataFrame that contains the ratings some users have given to a series of books.
# The ratings given are in the range from 1 to 5, with 5 being the best score.
# The names of the books, the corresponding authors, and the ratings of each user are given below:
```

```
books = pd.Series(data = ['Great Expectations', 'Of Mice and Men', 'Romeo and Juliet', 'The Time Machine', 'Ali
authors = pd.Series(data = ['Charles Dickens', 'John Steinbeck', 'William Shakespeare', ' H. G. Wells', 'Lewis 

# User ratings are in the order of the book titles mentioned above
# If a user has not rated all books, Pandas will automatically consider the missing values as NaN.
# If a user has mentioned `np.nan` value, then also it means that the user has not yet rated that book.
user_1 = pd.Series(data = [3.2, np.nan ,2.5])
user_2 = pd.Series(data = [5., 1.3, 4.0, 3.8])
user_3 = pd.Series(data = [2.0, 2.3, np.nan, 4])
user_4 = pd.Series(data = [4, 3.5, 4, 5, 4.2])


# Use the data above to create a Pandas DataFrame that has the following column
# labels: 'Author', 'Book Title', 'User 1', 'User 2', 'User 3', 'User 4'.
# Let Pandas automatically assign numerical row indices to the DataFrame.

# TO DO: Create a dictionary with the data given above

dat = {'Book Title': books,
       'Author' : authors,
       'User 1' : user_1,
       'User_2' : user_2,
       'User_3' : user_3,
       'User_4' : user_4}


# TO DO: Create a Pandas DataFrame using the dictionary created above
book_ratings = pd.DataFrame(dat)

# TO DO:
# If you created the dictionary correctly you should have a Pandas DataFrame
# that has column labels:
# 'Author', 'Book Title', 'User 1', 'User 2', 'User 3', 'User 4'
# and row indices 0 through 4.

# Now replace all the NaN values in your DataFrame with the average rating in
# each column. Replace the NaN values in place.
# HINT: Use the `pandas.DataFrame.fillna(value, inplace = True)` function for substituting the NaN values.
# Write your code below:

book_ratings.fillna(book_ratings.mean(numeric_only=True), inplace = True)

book_ratings
```

Out[88]:

	Book Title	Author	User 1	User_2	User_3	User_4
0	Great Expectations	Charles Dickens	3.2	5.0	2.0	4.0
1	Of Mice and Men	John Steinbeck	2.9	1.3	2.3	3.5
2	Romeo and Juliet	William Shakespeare	2.5	4.0	2.8	4.0
3	The Time Machine	H. G. Wells	2.9	3.8	4.0	5.0
4	Alice in Wonderland	Lewis Carroll	2.9	3.5	2.8	4.2

In [89]:

```
pwd
```

Out[89]:

```
'C:\\Users\\Isaac'
```

In machine learning you will most likely use databases from many sources to train your learning algorithms. Pandas allows us to load databases of different formats into DataFrames. One of the most popular data formats used to store databases is csv. CSV stands for Comma Separated Values and offers a simple format to store data. We can load CSV files into Pandas DataFrames using the `pd.read_csv()` function. Let's load Google stock data into a Pandas DataFrame. The GOOG.csv file contains Google stock data from 8/19/2004 till 10/13/2017 taken from Yahoo Finance.

Example 1. Load the data from a .csv file.

In [91]:

```
# We load Google stock data in a DataFrame
Google_stock = pd.read_csv('./GOOG.csv')

# We print some information about Google_stock
print('Google_stock is of type:', type(Google_stock))
print('Google_stock has shape:', Google_stock.shape)
```

```
Google_stock is of type: <class 'pandas.core.frame.DataFrame'>
Google_stock has shape: (3313, 7)
```

In [92]:

```
Google_stock
```

Out[92]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2004-08-19	49.7	51.7	47.7	49.8	49.8	44994500
1	2004-08-20	50.2	54.2	49.9	53.8	53.8	23005800
2	2004-08-23	55.0	56.4	54.2	54.3	54.3	18393200
3	2004-08-24	55.3	55.4	51.5	52.1	52.1	15361800
4	2004-08-25	52.1	53.7	51.6	52.7	52.7	9257400
...	...	...	...	...	...	...	...
3308	2017-10-09	980.0	985.4	976.1	977.0	977.0	891400
3309	2017-10-10	980.0	981.6	966.1	972.6	972.6	968400
3310	2017-10-11	973.7	990.7	972.2	989.2	989.2	1693300
3311	2017-10-12	987.5	994.1	985.0	987.8	987.8	1262400
3312	2017-10-13	992.0	997.2	989.0	989.7	989.7	1157700

3313 rows × 7 columns

In [93]:

```
#Example 3. Look at the first 5 rows of the DataFrame

Google_stock.head()
```

Out[93]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2004-08-19	49.7	51.7	47.7	49.8	49.8	44994500
1	2004-08-20	50.2	54.2	49.9	53.8	53.8	23005800
2	2004-08-23	55.0	56.4	54.2	54.3	54.3	18393200
3	2004-08-24	55.3	55.4	51.5	52.1	52.1	15361800
4	2004-08-25	52.1	53.7	51.6	52.7	52.7	9257400

In [96]:

```
#Example 4. Look at the last 5 rows of the DataFrame

Google_stock.tail()
```

Out[96]:

	Date	Open	High	Low	Close	Adj Close	Volume
3308	2017-10-09	980.0	985.4	976.1	977.0	977.0	891400
3309	2017-10-10	980.0	981.6	966.1	972.6	972.6	968400
3310	2017-10-11	973.7	990.7	972.2	989.2	989.2	1693300
3311	2017-10-12	987.5	994.1	985.0	987.8	987.8	1262400
3312	2017-10-13	992.0	997.2	989.0	989.7	989.7	1157700

We can also optionally use `.head(N)` or `.tail(N)` to display the first and last `N` rows of data, respectively.

Let's do a quick check to see whether we have any `NaN` values in our dataset. To do this, we will use the `.isnull()` method followed by the `.any()` method to check whether any of the columns contain NaN values.

Example 5. Check if any column contains a NaN. Returns a boolean for each column label.

In [103…

```
Google_stock.isnull().any()
```

Out[103]:

```
Date         False
Open         False
High         False
Low          False
Close        False
Adj Close    False
Volume       False
dtype: bool
```

We see that we have no `NaN` values.

When dealing with large datasets, it is often useful to get statistical information from them. Pandas provides the `.describe()` method to get descriptive statistics on each column of the DataFrame. Let's see how this works:

Example 6. See the descriptive statistics of the DataFrame

In [104…

```
# We get descriptive statistics on our stock data
Google_stock.describe()
```

Out[104]:

	Open	High	Low	Close	Adj Close	Volume
count	3313.0	3313.0	3313.0	3313.0	3313.0	3.3e+03
mean	380.2	383.5	376.5	380.1	380.1	8.0e+06
std	223.8	225.0	222.5	223.9	223.9	8.4e+06
min	49.3	50.5	47.7	49.7	49.7	7.9e+03
25%	226.6	228.4	224.0	226.4	226.4	2.6e+06
50%	293.3	295.4	289.9	293.0	293.0	5.3e+06
75%	536.7	540.0	532.4	536.7	536.7	1.1e+07
max	992.0	997.2	989.0	989.7	989.7	8.3e+07

Example 7. See the descriptive statistics of one of the columns of the DataFrame

In [107…

```
# We get descriptive statistics on a single column of our DataFrame
Google_stock['Adj Close'].describe()
```

Out[107]:

```
count    3313.0
mean      380.1
std       223.9
min        49.7
25%       226.4
50%       293.0
75%       536.7
max       989.7
Name: Adj Close, dtype: float64
```

Similarly, you can also look at one statistic by using one of the many statistical functions Pandas provides. Let's look at some examples:

Example 8. Statistical operations - Min, Max, and Mean

In [114…

```
# We print information about our DataFrame
print()
print('Maximum values of each column:\n', Google_stock.max())
print()
print('Minimum Close value:', Google_stock['Close'].min())
print()
print('Average value of each column:\n', Google_stock.mean())
```

```
Maximum values of each column:
 Date         2017-10-13
Open              992.0
High              997.2
Low               989.0
Close             989.7
Adj Close         989.7
Volume         82768100
dtype: object

Minimum Close value: 49.681866

Average value of each column:
 Open         3.8e+02
High         3.8e+02
Low          3.8e+02
Close        3.8e+02
Adj Close    3.8e+02
Volume       8.0e+06
dtype: float64
C:\Users\Isaac\AppData\Local\Temp\ipykernel_10164\2578638969.py:7: FutureWarning: Dropping of nuisance columns
in DataFrame reductions (with 'numeric_only=None') is deprecated; in a future version this will raise TypeError
.  Select only valid columns before calling the reduction.
  print('Average value of each column:\n', Google_stock.mean())
```

Another important statistical measure is data correlation. Data correlation can tell us, for example, if the data in different columns are correlated. We can use the `.corr()` method to get the correlation between different columns, as shown below:

Example 9. Statistical operation - Correlation

In [117…

```
Google_stock.corr()
```

Out[117]:

	Open	High	Low	Close	Adj Close	Volume
Open	1.0	1.0	1.0	1.0	1.0	-0.6
High	1.0	1.0	1.0	1.0	1.0	-0.6
Low	1.0	1.0	1.0	1.0	1.0	-0.6
Close	1.0	1.0	1.0	1.0	1.0	-0.6
Adj Close	1.0	1.0	1.0	1.0	1.0	-0.6
Volume	-0.6	-0.6	-0.6	-0.6	-0.6	1.0

A correlation value of 1 tells us there is a high correlation and a correlation of 0 tells us that the data is not correlated at all.

`groupby()` method

We will end this Introduction to Pandas by taking a look at the `.groupby()` method. The `.groupby()` method allows us to group data in different ways. Let's see how we can group data to get different types of information. For the next examples, we are going to load fake data about a fictitious company.

In [119…

```
data = pd.read_csv('fake-company.csv')
data
```

Out[119]:

	Year	Name	Department	Age	Salary
0	1990	Alice	HR	25	50000
1	1990	Bob	RD	30	48000
2	1990	Charlie	Admin	45	55000
3	1991	Dakota	HR	26	52000
4	1991	Elsa	RD	31	50000
5	1991	Frank	Admin	46	60000
6	1992	Grace	Admin	27	60000
7	1992	Hoffman	RD	32	52000
8	1992	Inaar	Admin	28	62000

Example 10. Demonstrate groupby() and sum() method

Let's calculate how much money the company spent on salaries each year. To do this, we will group the data by Year using the `.groupby()` method and then we will add up the salaries of all the employees by using the `.sum()` method.

In [130…

```
# We display the total amount of money spent in salaries each year
data.groupby('Year')['Salary'].sum()
```

Out[130]:

```
Year
1990    153000
1991    162000
1992    174000
Name: Salary, dtype: int64
```

Example 11. Demonstrate `groupby()` and `mean()` method

Now, let's suppose I want to know what was the average salary for each year. In this case, we will group the data by Year using the `.groupby()` method, just as we did before, and then we use the `.mean()` method to get the average salary. Let's see how this works

In [131…

```
# We display the average salary per year
data.groupby('Year')['Salary'].mean()
```

Out[131]:

```
Year
1990    51000.0
1991    54000.0
1992    58000.0
Name: Salary, dtype: float64
```

In [132…

```
# We display the total salary each employee received in all the years they worked for the company
data.groupby('Name')['Salary'].sum()
```

Out[132]:
```
Name
Alice      50000
Bob        48000
Charlie    55000
Dakota     52000
Elsa       50000
Frank      60000
Grace      60000
Hoffman    52000
Inaar      62000
Name: Salary, dtype: int64
```

Example 13. Demonstrate `groupby()` on two columns

Now let's see what was the salary distribution per department per year. In this case, we will group the data by Year and by Department using the `.groupby()` method and then we will add up the salaries for each department. Let's see the result

In [136…
```python
# We display the salary distribution per department per year.
data.groupby(['Year', 'Department'])['Salary'].sum()
```

Out[136]:
```
Year  Department
1990  Admin          55000
      HR             50000
      RD             48000
1991  Admin          60000
      HR             52000
      RD             50000
1992  Admin         122000
      RD             52000
Name: Salary, dtype: int64
```

Glossary

Below is the summary of all the functions and methods that you learned in this lesson:

Category: Initialization and Utility

Function/Method	Description
`pandas.read_csv(relative_path_to_file)`	Reads a comma-separated values (csv) file present at `relative_path_to_file` and loads it as a DataFrame
`pandas.DataFrame(data)`	Returns a 2-D heterogeneous tabular data. Note: There are other optional arguments as well that you can use to create a dataframe.
`pandas.Series(data, index)`	Returns 1-D ndarray with axis labels
`pandas.Series.shape` `pandas.DataFrame.shape`	Returns a tuple representing the dimensions
`pandas.Series.ndim` `pandas.DataFrame.ndim`	Returns the number of the dimensions (rank). It will return 1 in case of a Series
`pandas.Series.size` `pandas.DataFrame.size`	Returns the number of elements
`pandas.Series.values`	Returns the data available in the Series
`pandas.Series.index`	Returns the indexes available in the Series
`pandas.DataFrame.isnull()`	Returns a same sized object having True for NaN elements and False otherwise
`pandas.DataFrame.count(axis)`	Returns the count of non-NaN values along the given axis. If axis=0, it will count down the dataframe, meaning column-wise count of non-NaN values.
`pandas.DataFrame.head([n])`	Return the first *n* rows from the dataframe. By default, n=5.
`pandas.DataFrame.tail([n])`	Return the last *n* rows from the dataframe. By default, n=5. Supports negative indexing as well.
`pandas.DataFrame.describe()`	Generate the descriptive statistics, such as, count, mean, std deviation, min, and max.
`pandas.DataFrame.min()`	Returns the minimum of the values along the given axis.
`pandas.DataFrame.max()`	Returns the maximum of the values along the given axis.
`pandas.DataFrame. mean()`	Returns the mean of the values along the given axis.

`pandas.DataFrame.corr()`	Compute pairwise correlation of columns, excluding NA/null values.
`pandas.DataFrame.rolling(windows)`	Provide rolling window calculation, such as `pandas.DataFrame.rolling(15).mean()` for rolling mean over window size of 15.
`pandas.DataFrame.loc[label]`	Access a group of rows and columns by label(s)
`pandas.DataFrame.groupby(mapping_function)`	Groups the dataframe using a given mapper function or or by a Series of columns.

Category: Manipulation

Function/Method	Description
`pandas.Series.drop(index)`	Drops the element positioned at the given index(es)
`pandas.DataFrame.drop(labels)`	Drop specified labels (entire columns or rows) from the dataframe.
`pandas.DataFrame.pop(item)`	Return the item and drop it from the frame. If not found, then raise a KeyError.
`pandas.DataFrame.insert(location, column, values)`	Insert column having given values into DataFrame at specified location.
`pandas.DataFrame.rename(dictionary-like)`	Rename label(s) (columns or row-indexes) as mentioned in the `dictionary-like`
`pandas.DataFrame.set_index(keys)`	Set the DataFrame's row-indexes using existing column-values.
`pandas.DataFrame.dropna(axis)`	Remove rows (if axis=0) or columns (if axis=1) that contain missing values.
`pandas.DataFrame.fillna(value, method, axis)`	Replace NaN values with the specified value along the given axis, and using the given method ('backfill', 'bfill', 'pad', 'ffill', None)
`pandas.DataFrame.interpolate(method, axis)`	Replace the NaN values with the estimated value calculated using the given method along the given axis.

Project

In [ ]:

```

```