upGrad

Summary
Python for Data Science

NumPy Basics

NumPy is a library used in scientific computing and data analysis. It stands for 'Numerical Python'. The most basic object in NumPy is **ndarray** or **array**, which is an n-dimensional array.

Before using NumPy, you have to import it.
The NumPy library can be imported as shown in the code snippet given below.

```
import numPy as np
```

Now, we will create an array using the NumPy library.

```
array_1d = np.array([2, 4, 5, 6, 7, 9])
print(array_1d)
print(type(array_1d))
```

Output:

```
[2 4 5 6 7 9]
<class 'numpy.ndarray'>
```

Now, we will create a two-dimensional (2-D) array.

```
array_2d = np.array([[2, 3, 4], [5, 8, 7]])
print(array_2d)
print(type(array_2d))
```

Output:

```
[[2 3 4]
 [5 8 7]]
<class 'numpy.ndarray'>
```

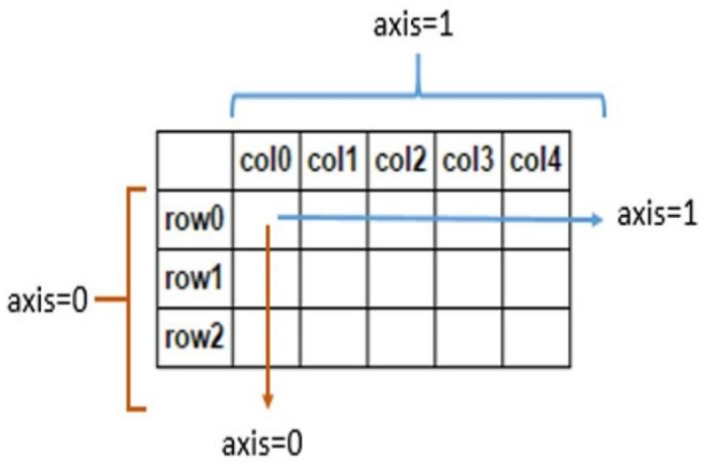
The advantages of NumPy are as follows:

1. You can write vectorized code on NumPy arrays, and not on lists, which is convenient to read and write, and is concise as well.
2. NumPy is much faster than the standard Python ways in performing computations.

In NumPy, dimensions are called **axes**. In the 2-D array given above, there are two axes having two and three elements respectively.

In NumPy terminology, for 2-D arrays:

- axis = 0 refers to rows, and
- axis = 1 refers to columns.



Now, to calculate product element-wise of all the elements in the list, we will use a lambda function as follows:

```
list_1 = [3, 6, 7, 5]
list_2 = [4, 5, 1, 7]

# the list way to do it: map a function to the two lists
product_list = list(map(lambda x, y: x*y, list_1, list_2))
print(product_list)
```

Output:

```
[12, 30, 7, 35]
```

Now, we will use the NumPy library to compute the product.

```
# The numpy array way to do it: simply multiply the two arrays
array_1 = np.array(list_1)
array_2 = np.array(list_2)
```

```
array_3 = array_1*array_2
print(array_3)
print(type(array_3))
```

Output:

```
[12 30  7 35]
<class 'numpy.ndarray'>
```

```
# Square a list
list_1 = [3, 6, 7, 5]
list_squared = [i**2 for i in list_1]

# Square a numpy array
array_1 = np.array(list_1)
array_squared = array_1**2

print(list_squared)
print(array_squared)
```

Output:

```
[9, 36, 49, 25]
[ 9 36 49 25]
```

From the illustration provided above, we can conclude that squaring a NumPy array is easier than squaring a list, which requires a loop, whereas squaring a NumPy array does not require any loop.

Creating a NumPy Array

There are multiple ways to create a NumPy array. The most common way is to use np.array, which is shown in the code snippet given below.

```
# Convert lists or tuples to arrays using np.array()
# Note that np.array(2, 5, 6, 7) will throw an error - you need to
pass a list or a tuple
array_from_list = np.array([2, 5, 6, 7])
array_from_tuple = np.array((4, 5, 8, 9))

print(array_from_list)
print(array_from_tuple)
```

Output:

```
[2 5 6 7]
[4 5 8 9]
```

The second most common way to create a NumPy array is to initialise an array. You can do this only when you know the size of the array beforehand.

Other commonly used functions are as follows:

- np.ones():
  - ❖ It is used to create an array of 1s.
- np.zeros():
  - ❖ It is used to create an array of 0s.
- np.random.random():
  - ❖ It is used to create an array of random numbers between 0 and 1.
- np.arange():
  - ❖ It is used to create an array with increments of fixed step size.
- np.linspace():
  - ❖ It is used to create an array of a fixed length when the step size is unknown.

The code snippet given below shows how to initialise 5x3 unit matrix.

```
# Notice that, by default, numpy creates data type = float64
# Can provide dtype explicitly using dtype
np.ones((5, 3), dtype = np.int)
```

Output:

```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

.
The code snippet given below shows how to initialise 4x1 zero matrix.

```
# Creating array of zeros
np.zeros(4, dtype = np.int)
```

Output:

```
array([0, 0, 0, 0])
```

The code snippet given below shows how to initialise 3x4 random numbers matrix.

```
# Array of random numbers
np.random.random([3, 4])
```

Output:

```
array([[0.21067268, 0.20062275, 0.9481293 , 0.37904343],
       [0.28643457, 0.26614814, 0.43219753, 0.63020881],
       [0.36568786, 0.37602622, 0.85852183, 0.29602912]])
```

There is one more initialisation function, np.arange(), which is equivalent to the range() function.

The code to initialise 5x1 matrix with multiples of 5 less than 100 is shown in the code snippet given below.

```
# From 10 to 100 with a step of 5
numbers = np.arange(10, 100, 5)
print(numbers)
```

Output:

```
[10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]
```

Sometimes, you only know the length of the array, and not the step size, so we use the linspace function to initialise the matrix.

```
# np.linspace()
# Sometimes, you know the length of the array, not the step size

# Array of length 25 between 15 and 18
np.linspace(15, 18, 25)
```

Output:

```
array([15.   , 15.125, 15.25 , 15.375, 15.5  , 15.625, 15.75 ,
15.875,
       16.   , 16.125, 16.25 , 16.375, 16.5  , 16.625, 16.75 ,
16.875,
       17.   , 17.125, 17.25 , 17.375, 17.5  , 17.625, 17.75 ,
17.875,
       18.   ])
```

A few more NumPy functions that you can use to create special NumPy arrays are as follows:

- np.full(): This function is used to create a constant array of any number ‘n’.

```
# Creating a 4 x 3 array of 7s using np.full()
# The default data type here is int only
np.full((4,3), 7)
```

Output:

```
array([[7, 7, 7],
       [7, 7, 7],
       [7, 7, 7],
       [7, 7, 7]])
```

- np.tile(): This function is used to create a new array by repeating an existing array for a particular number of times.

```
# Given an array, np.tile() creates a new array by repeating the
given array for any number of times that you want
# The default data type her is int only
arr = ([0, 1, 2])
np.tile(arr, 3)
```

Output:

```
array([0, 1, 2, 0, 1, 2, 0, 1, 2])
```

- np.eye(): This function is used to create an identity matrix of any dimension.

```
# Create a 3 x 3 identity matrix using np.eye()
# The default data type here is float. So if we want integer
values, we need to specify the dtype to be int
np.eye(3, dtype = int)
```

Output:

```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

- np.randint(): This function is used to create a random array of integers within a particular range.

```
# Create a 4 x 4 random array of integers ranging from 0 to 9
np.random.randint(0, 10, (4,4))
```

Output:

```
array([[2, 8, 9, 6],
       [7, 0, 5, 2],
       [6, 3, 8, 8],
       [9, 9, 1, 9]])
```

Structure and Content of Arrays

It is recommended to inspect NumPy arrays, especially while working with large arrays. Some attributes of NumPy arrays are as follows:

- shape: It determines the number of rows and columns in the array.
- dtype: It determines the data type (int, float, etc.).

- ndim: It determines the number of dimensions (or axes).
- itemsize: It determines the memory used by each element of an array in bytes.

The above function discussed to know attributes of the array are shown in the code snippet given below.

```
# Initialising a random 1000 x 300 array
rand_array = np.random.random((1000, 300))
# Inspecting shape, dtype, ndim and itemsize
print("Shape: {}".format(rand_array.shape))
print("dtype: {}".format(rand_array.dtype))
print("Dimensions: {}".format(rand_array.ndim))
print("Item size: {}".format(rand_array.itemsize))
```

Output:

```
Shape: (1000, 300)
dtype: float64
Dimensions: 2
Item size: 8
```

With the help of the illustration provided above, we understood what these attributes do, where shape determines the number of rows and columns in an array, dtype determines the data type, ndim determines the number of dimensions, and itemsize determines how much memory is used by each element of the array.

Subset, Slice, Index, and Iterate Through Arrays

-> For **one-dimensional arrays**, indexing, slicing, etc. are **similar to Python lists**, where indexing starts at 0.

Some of the examples of slicing are discussed in the following code snippet.

```
array_1d = np.arange(10)
# Third element
print(array_1d[2])

# Specific elements
# Notice that array[2, 5, 6] will throw an error, you need to provide
the indices as a list
print(array_1d[[2, 5, 6]])

# Slice third element onwards
print(array_1d[2:])

# Slice first three elements
print(array_1d[:3])

# Slice third to seventh elements
print(array_1d[2:7])

# Subset starting 0 at increment of 2
print(array_1d[0::2])
```

Output:

```
2
[2 5 6]
[2 3 4 5 6 7 8 9]
[0 1 2]
[2 3 4 5 6]
[0 2 4 6 8]
```

With this, you learnt how to slice a NumPy array and find an element at a particular index or find elements from starting to ending indices.

Multidimensional Arrays

-> **Multidimensional arrays** are indexed using as many indices as the number of dimensions or axes. For instance, to index a 2-D array, you need two indices: array[x, y].

Some slicing of multidimensional arrays in Python is shown in the code snippet given below.

```
# Creating a 2-D array
array_2d = np.array([[2, 5, 7, 5], [4, 6, 8, 10], [10, 12, 15, 19]])
# Third row second column
print(array_2d[2, 1])
# Slicing the second row, and all columns
# Notice that the resultant is itself a 1-D array
print(array_2d[1, :])
# Slicing all rows and the third column
print(array_2d[:, 2])
# Slicing all rows and the first three columns
print(array_2d[:, :3])
```

Output:

```
12
[ 4  6  8 10]
[ 7  8 15]
[[ 2  5  7]
 [ 4  6  8]
 [10 12 15]]
```

Iterating over 2-D arrays is done with respect to the first axis (which is the row, and the second axis is the column).

```
# Iterating over 2-D arrays
for row in array_2d:
    print(row)
```

Output:

```
[2 5 7 5]
[ 4  6  8 10]
[10 12 15 19]
```

Computation Times in NumPy and Standard Python Lists

You learnt that the key advantages of NumPy are convenience and speed of computation.

You will often work with extremely large data sets; thus it is important for you to understand how much computation time (and memory) you can save using NumPy as compared with standard Python lists.

Now, let's compare the computation times of arrays and lists for calculating the element-wise product of numbers as shown in code snippet given below.

```
## Comparing time taken for computation
list_1 = [i for i in range(1000000)]
list_2 = [j**2 for j in range(1000000)]

# list multiplication
import time

# store start time, time after computation, and take the difference
t0 = time.time()
product_list = list(map(lambda x, y: x*y, list_1, list_2))
t1 = time.time()
list_time = t1 - t0
print(t1-t0)
# numpy array
array_1 = np.array(list_1)
array_2 = np.array(list_2)
t0 = time.time()
array_3 = array_1*array_2
t1 = time.time()
numpy_time = t1 - t0
print(t1-t0)
print("The ratio of time taken is {}".format(list_time/numpy_time))
```

Output:

```
0.13900446891784668
0.005043983459472656
The ratio of time taken is 27.558470410285498
```

Basic Operations on NumPy Arrays

➔ Reshaping Arrays

Reshaping is done using the reshape() function.

Example of reshaping an array is shown below :

```
import numpy as np
# Reshape a 1-D array to a 3 x 4 array
some_array = np.arange(0, 12).reshape(3, 4)
print(some_array)
```

Output:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

➔ Stacking and Splitting Arrays

Stacking: np.hstack() and np.vstack()

Stacking is done using the np.hstack() and np.vstack() methods. For horizontal stacking, the number of rows should be the same, whereas for vertical stacking, the number of columns should be the same.

We will see an example of using np.vstack() method in the code snippet given below:

```
# Creating two arrays
array_1 = np.arange(12).reshape(3, 4)
array_2 = np.arange(20).reshape(5, 4)
print(array_1)
print("\n")
print(array_2)
# vstack
# Note that np.vstack(a, b) throws an error - you need to pass the
arrays as a list
np.vstack((array_1, array_2))
```

Output:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]

array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

Operations on Arrays

➔ Basic Mathematical Operations

NumPy provides almost all the basic math functions, such as exp, sin, cos, log and sqrt. One function is applied to each element of an array.

```
# Basic mathematical operations
a = np.arange(1, 20)
# sin, cos, exp, log
print(np.sin(a))
print(np.cos(a))
print(np.exp(a))
print(np.log(a))
```

Output:

```
[ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427 -0.2794155
  0.6569866   0.98935825  0.41211849 -0.54402111 -0.99999021 -0.53657292
  0.42016704  0.99060736  0.65028784 -0.28790332 -0.96139749 -0.75098725
  0.14987721]
[ 0.54030231 -0.41614684 -0.9899925  -0.65364362  0.28366219  0.96017029
  0.75390225 -0.14550003 -0.91113026 -0.83907153  0.0044257   0.84385396
  0.90744678  0.13673722 -0.75968791 -0.95765948 -0.27516334  0.66031671
  0.98870462]
[2.71828183e+00 7.38905610e+00 2.00855369e+01 5.45981500e+01
 1.48413159e+02 4.03428793e+02 1.09663316e+03 2.98095799e+03
 8.10308393e+03 2.20264658e+04 5.98741417e+04 1.62754791e+05
 4.42413392e+05 1.20260428e+06 3.26901737e+06 8.88611052e+06
 2.41549528e+07 6.56599691e+07 1.78482301e+08]
[0.         0.69314718 1.09861229 1.38629436 1.60943791 1.79175947
 1.94591015 2.07944154 2.19722458 2.30258509 2.39789527 2.48490665
 2.56494936 2.63905733 2.7080502  2.77258872 2.83321334 2.89037176
 2.94443898]
```

➔ User-Defined Functions

You can also apply your own functions on arrays. For example, you can apply the x/(x+1) function to each element of an array.

One way to do this is by looping through the array, which is the non-NumPy way. If you want to write **vectorized code**, then you can vectorise the function that you want, and then apply it on the array. Also, NumPy provides the np.vectorise() method to vectorise functions.Let's take a look at both the ways.

```
# Basic mathematical operations
a = np.arange(1, 20)
print(a)
# The non-numpy way, not recommended
a_list = [x/(x+1) for x in a]
print(a_list)
# The numpy way: vectorize the function, then apply it
f = np.vectorize(lambda x: x/(x+1))
print(f(a))
# Apply function on a 2-d array: Applied to each element
```

```
b = np.linspace(1, 100, 10)
print(f(b))
```

Output:

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

[0.5, 0.6666666666666666, 0.75, 0.8, 0.8333333333333334,
0.8571428571428571, 0.875, 0.8888888888888888, 0.9,
0.9090909090909091, 0.9166666666666666, 0.9230769230769231,
0.9285714285714286, 0.9333333333333333, 0.9375, 0.9411764705882353,
0.9444444444444444, 0.9473684210526315, 0.95]

[0.5, 0.66666667, 0.75, 0.8, 0.83333333,0.85714286, 0.875, 0.88888889,
0.9, 0.90909091,0.91666667, 0.92307692, 0.92857143, 0.93333333,
0.9375,0.94117647, 0.94444444, 0.94736842, 0.95]

[0.5, 0.92307692, 0.95833333, 0.97142857, 0.97826087,0.98245614,
0.98529412, 0.98734177, 0.98888889, 0.99009901]
```

np.vectorise() also gives you the advantage to vectorise the function once, and then apply it as many times as needed.

Basic Linear Algebra Operations

Basic Linear Algebra Operations

NumPy provides the np.linalg package to apply common linear algebra operations, which are as follows:

- np.linalg.inv: It is used to compute the inverse of a matrix.
- np.linalg.det: It is used to compute the determinant of a matrix.
- np.linalg.eig: It is used to compute the eigenvalues and eigenvectors of a matrix.

Also, you can multiply matrices using np.dot(a, b).

Now we will apply what we have learned about the Linear Algebra operations as shown in the code below.

```
# Creating arrays
a = np.arange(1, 10).reshape(3, 3)
```

```
b= np.arange(1, 13).reshape(3, 4)
print(a)
print(b)
# Inverse
print(np.linalg.inv(a))
# Determinant
print(np.linalg.det(a))
# Eigenvalues and eigenvectors
print(np.linalg.eig(a))
# Multiply matrices
print(np.dot(a, b))
```

Output:

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

[[ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]
 [-6.30503948e+15  1.26100790e+16 -6.30503948e+15]
 [ 3.15251974e+15 -6.30503948e+15  3.15251974e+15]]

-9.51619735392994e-16

(array([ 1.61168440e+01, -1.11684397e+00, -9.75918483e-16]),
array([[-0.23197069, -0.78583024,  0.40824829],[-0.52532209,
-0.08675134, -0.81649658], [-0.8186735 ,  0.61232756,
0.40824829]]))

[[ 38  44  50  56]
 [ 83  98 113 128]
 [128 152 176 200]]
```

Pandas Library

Pandas is a library built specifically for data analysis and created using NumPy. You will be using Pandas extensively for performing data manipulation or visualisation, building machine learning models, etc.

The two main data structures in Pandas are Series and Dataframes. The default way to store data is by using data frames; thus, manipulating data frames quickly is probably the most important skill for data analysis.

Pandas Series: A series is similar to a 1-D NumPy array, and contains scalar values of the same type (numeric, character, date/time, etc.). A data frame is simply a table, where each column is a Pandas series.

➔ **Creating Pandas Series**

You can create Pandas series from array-like objects using pd.Series(). Each element in the series has an index, which starts from 0.

```
# import pandas, pd is an alias
import pandas as pd
import numpy as np

# Creating a numeric pandas series
s = pd.Series([2, 4, 5, 6, 9])
print(s)
print(type(s))
```

Output:

```
0    2
1    4
2    5
3    6
4    9
dtype: int64
<class 'pandas.core.series.Series'>
```

➔ Indexing Series

Indexing series is the same as 1-D NumPy arrays, where index starts from 0.

Indexing pandas series as shown below in the snippet.

```
# Indexing pandas series: Same as indexing 1-d numpy arrays or lists
# accessing the fourth element
s[3]
# accessing elements starting index = 2 till the end
s[2:]
```

Output:

```
2    5
3    6
4    9
dtype: int64
```

➔ Explicitly Specifying Indices

You might have noticed that while creating a series, Pandas automatically indexes it from 0 to (n-1), where n is the number of rows. However, you can also explicitly set the index yourself using the 'index' argument while creating the series using pd.Series().

```
# You can also give the index as a sequence or use functions to specify
the index
# But always make sure that the number of elements in the index list is
equal to the number of elements specified in the series
pd.Series(np.array(range(0,10))**2, index = range(0,10))
```

Output:

```
0      0
1      1
2      4
3      9
4     16
5     25
6     36
7     49
8     64
9     81
dtype: int32
```

➔ Pandas Data Frame

Dataframe is the most widely used data structure in data analysis. It is a table with rows and columns, where rows have an index and columns have meaningful names.

➔ **Creating Data Frames From Dictionaries**

Various ways of creating data frames include using dictionaries, JSON objects and CSV files, and reading from text files.

```
# keys become column names
df = pd.Dataframe({'name': ['Vinay', 'Kushal', 'Aman', 'Saif'],
                   'age': [22, 25, 24, 28],
                    'occupation': ['engineer', 'doctor', 'data analyst',
'teacher']})
df
```

Output:

	name	age	occupation
0	Vinay	22	engineer
1	Kushal	25	doctor
2	Aman	24	data analyst
3	Saif	28	teacher

➔ **Importing CSV Files as Pandas Dataframes**

You can read CSV files as shown in code snippet below:

```
# reading a CSV file as a dataframe
market_df = pd.read_csv("global_sales_data/market_fact.csv")
market_df.head() #head() gives first five rows of dataset.
```

Output:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.54
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	14.30	0.37
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38

```
# Looking at the datatypes of each column
market_df.info()
# Note that each column is basically a pandas Series of length
8399
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8399 entries, 0 to 8398
Data columns (total 10 columns):
Ord_id                 8399 non-null object
Prod_id                8399 non-null object
Ship_id                8399 non-null object
Cust_id                8399 non-null object
Sales                  8399 non-null float64
Discount               8399 non-null float64
Order_Quantity         8399 non-null int64
Profit                 8399 non-null float64
Shipping_Cost          8399 non-null float64
Product_Base_Margin    8336 non-null float64
dtypes: float64(5), int64(1), object(4)
memory usage: 656.2+ KB
```

```
# Describe gives you a summary of all the numeric columns in the
dataset
market_df.describe()
```

Output:

	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
count	8399.000000	8399.000000	8399.000000	8399.000000	8399.000000	8336.000000
mean	1775.878179	0.049671	25.571735	181.184424	12.838557	0.512513
std	3585.050525	0.031823	14.481071	1196.653371	17.264052	0.135589
min	2.240000	0.000000	1.000000	-14140.700000	0.490000	0.350000
25%	143.195000	0.020000	13.000000	-83.315000	3.300000	0.380000
50%	449.420000	0.050000	26.000000	-1.500000	6.070000	0.520000
75%	1709.320000	0.080000	38.000000	162.750000	13.990000	0.590000
max	89061.050000	0.250000	50.000000	27220.690000	164.730000	0.850000

upGrad

➔ Sorting Data Frames

You can sort data frames in the following two ways:

1) By indices

2) By values

Firstly we will sort Dataframes by indices as shown in code snippet below:

```
# Sorting by index
# axis = 0 indicates that you want to sort rows (use axis=1 for
columns)
market_df.sort_index(axis = 0, ascending = False).head()
```

Output:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
Ord_999	Prod_15	SHP_1383	Cust_361	5661.0800	0.00	33	1055.47	30.00	0.62
Ord_998	Prod_8	SHP_1380	Cust_372	750.6600	0.00	33	120.05	4.00	0.60
Ord_998	Prod_5	SHP_1382	Cust_372	2149.3700	0.03	42	217.87	19.99	0.55
Ord_998	Prod_8	SHP_1381	Cust_372	254.3200	0.01	8	-117.39	6.50	0.79
Ord_997	Prod_14	SHP_1379	Cust_365	28761.5200	0.04	8	285.11	24.49	0.37

Secondly, we will sort Dataframes by Values as shown in code snippet below.:

```
# Sorting by values

# Sorting in decreasing order of Sales we use an attribute called
ascending which is boolean for descending we use false and true
for ascending.
market_df.sort_values(by='Sales',ascending=false).head()
```

Output:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
Ord_3084	Prod_17	SHP_4279	Cust_1151	89061.05	0.00	13	27220.69	24.49	0.39
Ord_2338	Prod_17	SHP_3207	Cust_932	45923.76	0.07	7	102.61	24.49	0.39
Ord_3875	Prod_17	SHP_5370	Cust_1351	41343.21	0.09	8	3852.19	24.49	0.39
Ord_2373	Prod_14	SHP_3259	Cust_942	33367.85	0.01	9	3992.52	24.49	0.37
Ord_4614	Prod_14	SHP_6423	Cust_1571	29884.60	0.05	49	12748.86	24.49	0.44

Indexing and Selecting Data

➔ Selecting Rows

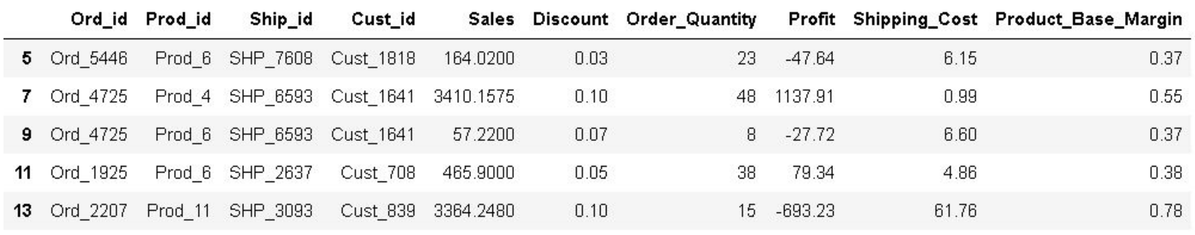
Selecting rows in data frames is similar to indexing in NumPy arrays, which you looked at. The syntax `df[start_index:end_index]` will subset rows according to the start and end indices.

Selecting the rows from indices 2 to 6 shown in the code snippet below.

```
import numpy as np
import pandas as pd

market_df = pd.read_csv("../global_sales_data/market_fact.csv")
# Selecting the rows from indices 2 to 6
market_df[2:7]
```

Output:



	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.0200	0.03	23	-47.64	6.15	0.37
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	0.99	0.55
9	Ord_4725	Prod_6	SHP_6593	Cust_1641	57.2200	0.07	8	-27.72	6.60	0.37
11	Ord_1925	Prod_6	SHP_2637	Cust_708	465.9000	0.05	38	79.34	4.86	0.38
13	Ord_2207	Prod_11	SHP_3093	Cust_839	3364.2480	0.10	15	-693.23	61.76	0.78

➔ Selecting Columns

The two simple ways to select a single column from a dataframe are as follows:

- `df['column_nam` `df['column_name']`
- `df.column_name`

```
# Using df['column']
sales = market_df['Sales']
sales.head()
```

Output:

```
0      136.81
1       42.27
2     4701.69
3     2337.89
4     4233.15
Name: Sales, dtype: float64
```

➔ **Selecting Multiple Columns**

You can select multiple columns by passing the list of column names inside the [ ]: df[['column_1', 'column_2', 'column_n']].

```
# Select Cust_id, Sales and Profit:
market_df[['Cust_id', 'Sales', 'Profit']].head()
```

Output:

	Cust_id	Sales	Profit
0	Cust_1818	136.81	-30.51
1	Cust_1818	42.27	4.56
2	Cust_1818	4701.69	1148.90
3	Cust_1818	2337.89	729.34
4	Cust_1818	4233.15	1219.87

➔ **Merge and Append**

In this section, you will merge and concatenate multiple Dataframes. Merging is one of the most common operations you will do since data often comes in various files.

- Merge multiple data frames using common columns/keys using pd.merge().
- Concatenate data frames using pd.concat().

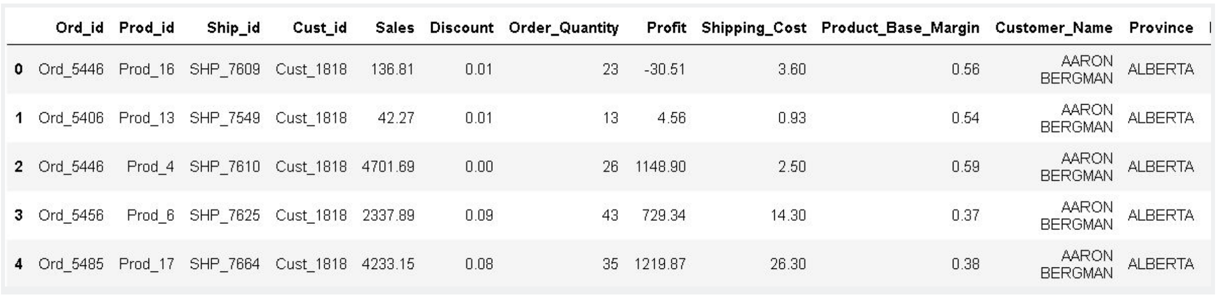
Now we merge Dataframes by using pd.merge() method as shown below :

```
# loading libraries and reading the data
import numpy as np
import pandas as pd

market_df = pd.read_csv("./global_sales_data/market_fact.csv")
```

```
customer_df = pd.read_csv("./global_sales_data/cust_dimen.csv")
product_df = pd.read_csv("./global_sales_data/prod_dimen.csv")
shipping_df = pd.read_csv("./global_sales_data/shipping_dimen.csv")
orders_df = pd.read_csv("./global_sales_data/orders_dimen.csv")
# Merging the dataframes
# Note that Cust_id is the common column/key, which is provided to the
'on' argument
# how = 'inner' makes sure that only the customer ids present in both
dfs are included in the result
df_1 = pd.merge(market_df, customer_df, how='inner', on='Cust_id')
df_1.head()
```

Output:



	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	Customer_Name	Province
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	AARON BERGMAN	ALBERTA
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	0.93	0.54	AARON BERGMAN	ALBERTA
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	AARON BERGMAN	ALBERTA
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	14.30	0.37	AARON BERGMAN	ALBERTA
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	26.30	0.38	AARON BERGMAN	ALBERTA

➔ **Concatenate Data Frames**

Concatenation is much more straightforward than merging. It is used when you have dataframes with the same columns and want to append them (pile one on top of the other), or when you have the same rows and want to append them side by side.

```
# dataframes having the same columns
df1 = pd.Dataframe({'Name': ['Aman', 'Joy', 'Rashmi', 'Saif'],
                    'Age': ['34', '31', '22', '33'],
                    'Gender': ['M', 'M', 'F', 'M']}
                  )

df2 = pd.Dataframe({'Name': ['Akhil', 'Asha', 'Preeti'],
                    'Age': ['31', '22', '23'],
                    'Gender': ['M', 'F', 'F']}
                  )
# To concatenate them, one on top of the other, you can use
pd.concat
# The first argument is a sequence (list) of dataframes
# axis = 0 indicates that we want to concat along the row axis
pd.concat([df1, df2], axis = 0)
```

Output:

```
     Name  Age  Gender
0    Aman   34       M
1     Joy   31       M
2  Rashmi   22       F
3    Saif   33       M
0   Akhil   31       M
1    Asha   22       F
2  Preeti   23       F
```

➔ Concatenating Dataframes Having Same Rows

You may also have dataframes with the same rows but different columns (and no common columns). In this case, you may want to concatenate them side by side.

We will concatenate the two dataFrames by the method shown below in the snippet.

```
df1 = pd.Dataframe({'Name': ['Aman', 'Joy', 'Rashmi', 
'Saif'],
                    'Age': ['34', '31', '22', '33'],
                    'Gender': ['M', 'M', 'F', 'M']}
                  )
df2 = pd.Dataframe({'School': ['RK Public', 'JSP', 'Carmel 
Convent', 'St. Paul'],
                    'Graduation Marks': ['84', '89', '76', 
'91']}
                  )
# To join the two dataframes, use axis = 1 to indicate 
joining along the columns axis
# The join is possible because the corresponding rows have 
the same indices
pd.concat([df1, df2], axis = 1)
```

Output:

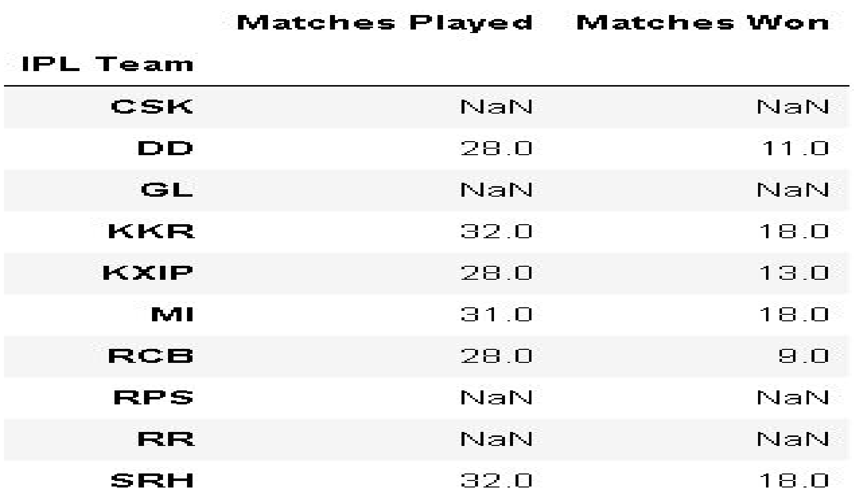
	Name	Age	Gender	School	Graduation Marks
0	Aman	34	M	RK Public	84
1	Joy	31	M	JSP	89
2	Rashmi	22	F	Carmel Convent	76
3	Saif	33	M	St. Paul	91

➔ Performing Arithmetic Operations on More Than One Data Frames

Now we will perform some operations on Dataframes.

```
# Teamwise stats for IPL 2018
IPL_2018 = pd.Dataframe({'IPL Team': ['CSK', 'SRH', 'KKR', 'RR',
'MI', 'RCB', 'KXIP', 'DD'],
                         'Matches Played': [16, 17, 16, 15, 14,
14, 14, 14],
                         'Matches Won': [11, 10, 9, 7, 6, 6, 6,
5]}
                       )
# Set the 'IPL Team' column as the index to perform arithmetic
operations on the other rows using the team as reference
IPL_2018.set_index('IPL Team', inplace = True)
# Similarly, we have the stats for IPL 2017
IPL_2017 = pd.Dataframe({'IPL Team': ['MI', 'RPS', 'KKR', 'SRH',
'KXIP', 'DD', 'GL', 'RCB'],
                         'Matches Played': [17, 16, 16, 15, 14,
14, 14, 14],
                         'Matches Won': [12, 10, 9, 8, 7, 6, 4,
3]}
                       )
IPL_2017.set_index('IPL Team', inplace = True)
# Simply add the two DFs using the add opearator
Total = IPL_2018 + IPL_2017
Total
```

Output:



IPL Team	Matches Played	Matches Won
CSK	NaN	NaN
DD	28.0	11.0
GL	NaN	NaN
KKR	32.0	18.0
KXIP	28.0	13.0
MI	31.0	18.0
RCB	28.0	9.0
RPS	NaN	NaN
RR	NaN	NaN
SRH	32.0	18.0

You might have noticed that there are a lot of Not a Number (NaN) values. This is because some teams that played in IPL 2017 were not present in IPL 2018. In addition, there were new teams present in IPL 2018. We can handle these NaN values using df.add() instead of the simple add operator. Let's learn how.

➔ Grouping and Summarising Data Frames

Grouping and aggregation are some of the most frequently used operations in data analysis, especially while performing exploratory data analysis (EDA), where comparing summary statistics across groups of data is common.

For example, in the retail sales data that you are working with, you may want to compare the average sales of various regions, or the total profit of two customer segments.

Grouping analysis can consist of the following three parts:

1. **Splitting** the data into groups (For example, groups of customer segments and product categories)
2. **Applying** a function to each group (For example, mean or total sales of each customer segment)
3. **Combining** the results into a data structure showing the summary statistics

First, we will merge all the Dataframes, so that we have all the data in one master_df as shown in the code snippet below:

```
# Loading libraries and files
import numpy as np
import pandas as pd
market_df = pd.read_csv("../global_sales_data/market_fact.csv")
customer_df = pd.read_csv("../global_sales_data/cust_dimen.csv")
product_df = pd.read_csv("../global_sales_data/prod_dimen.csv")
shipping_df = pd.read_csv("../global_sales_data/shipping_dimen.csv")
orders_df = pd.read_csv("../global_sales_data/orders_dimen.csv")
# Merging the dataframes one by one
df_1 = pd.merge(market_df, customer_df, how='inner', on='Cust_id')
df_2 = pd.merge(df_1, product_df, how='inner', on='Prod_id')
df_3 = pd.merge(df_2, shipping_df, how='inner', on='Ship_id')
master_df = pd.merge(df_3, orders_df, how='inner', on='Ord_id')
master_df.head()
```

Output:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	...	Region	Customer_Segment
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	...	WEST	CORPORATE
1	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	...	WEST	CORPORATE
2	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.02	0.03	23	-47.64	6.15	0.37	...	WEST	CORPORATE
3	Ord_2978	Prod_16	SHP_4112	Cust_1088	305.05	0.04	27	23.12	3.37	0.57	...	ONTARIO	HOME OFFICE
4	Ord_5484	Prod_16	SHP_7663	Cust_1820	322.82	0.05	35	-17.58	3.98	0.56	...	WEST	CONSUMER

5 rows × 22 columns

_Margin	...	Region	Customer_Segment	Product_Category	Product_Sub_Category	Order_ID_x	Ship_Mode	Ship_Date	Order_ID_y	Order_Date	Order_Priority
0.56	...	WEST	CORPORATE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	36262	REGULAR AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED
0.59	...	WEST	CORPORATE	TECHNOLOGY	TELEPHONES AND COMMUNICATION	36262	EXPRESS AIR	27-07-2010	36262	27-07-2010	NOT SPECIFIED
0.37	...	WEST	CORPORATE	OFFICE SUPPLIES	PAPER	36262	EXPRESS AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED
0.57	...	ONTARIO	HOME OFFICE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	37863	REGULAR AIR	26-02-2011	37863	24-02-2011	HIGH
0.56	...	WEST	CONSUMER	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	53026	REGULAR AIR	03-03-2012	53026	26-02-2012	LOW

Step 1: Grouping Using df.groupby()

```
# Which customer segments are the least profitable?
# Step 1. Grouping: First, we will group the dataframe by customer segments
df_by_segment = master_df.groupby('Customer_Segment')
df_by_segment
```

Output:

```
<pandas.core.groupby.DataframeGroupBy object at 0x1046be710>
```

Step 2: Applying a Function

```
# We can choose aggregate functions such as sum, mean, median,
etc.
df_by_segment['Profit'].sum()
```

Output:

```
Customer_Segment
CONSUMER          287959.94
CORPORATE         599746.00
HOME OFFICE       318354.03
SMALL BUSINESS    315708.01
Name: Profit, dtype: float64
```

Step 3: Combining Results Into Data Structures

We will combine results in data structure as shown in code snippet below

```
# Converting to a df
pd.Dataframe(df_by_segment['Profit'].sum())
# Let's go through some more examples
# E.g.: Which product categories are the least profitable?

# 1. Group by product category
by_product_cat = master_df.groupby('Product_Category')
# E.g.: Which product categories and sub-categories are the least
profitable?
# 1. Group by category and sub-category
by_product_cat_subcat = master_df.groupby(['Product_Category',
'Product_Sub_Category'])
by_product_cat_subcat['Profit'].mean()
# E.g.: Which product categories and sub-categories are the least
profitable?
# 1. Group by category and sub-category
by_product_cat_subcat = master_df.groupby(['Product_Category',
'Product_Sub_Category'])
by_product_cat_subcat['Profit'].mean()
```

Output:

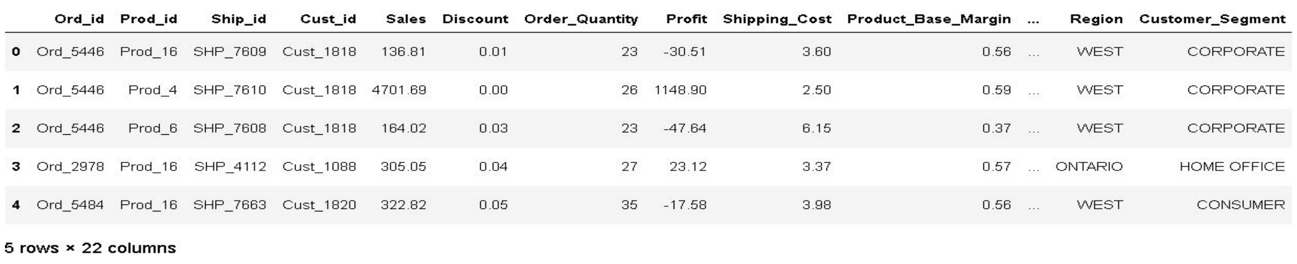
```
Product_Category  Product_Sub_Category
FURNITURE         BOOKCASES                          -177.683228
                  CHAIRS & CHAIRMATS                  387.693601
                  OFFICE FURNISHINGS                  127.446612
                  TABLES                             -274.411357
OFFICE SUPPLIES   APPLIANCES                          223.866498
                  BINDERS AND BINDER ACCESSORIES      335.970918
                  ENVELOPES                           195.864228
                  LABELS                               47.490174
                  PAPER                                36.949551
                  PENS & ART SUPPLIES                  11.950679
                  RUBBER BANDS                         -0.573575
                  SCISSORS, RULERS AND TRIMMERS       -54.161458
                  STORAGE & ORGANIZATION               12.205403
TECHNOLOGY        COMPUTER PERIPHERALS                124.389815
                  COPIERS AND FAX                    1923.695287
                  OFFICE MACHINES                     913.094748
                  TELEPHONES AND COMMUNICATION        358.948607
Name: Profit, dtype: float64
```

➔ **Lambda Function and Pivot Tables**

Let's first read all the files and create a master_df as shown in the code snippet below.

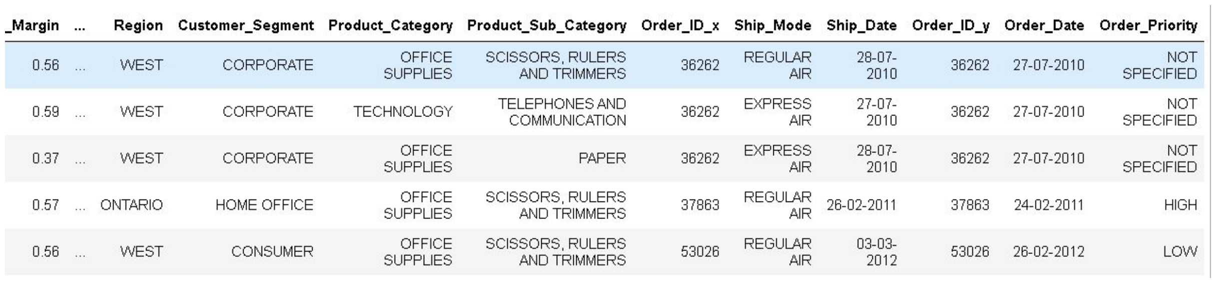
```
# Loading libraries and files
import numpy as np
import pandas as pd
market_df =
pd.read_csv("../global_sales_data/market_fact.csv")
customer_df =
pd.read_csv("../global_sales_data/cust_dimen.csv")
product_df =
pd.read_csv("../global_sales_data/prod_dimen.csv")
shipping_df =
pd.read_csv("../global_sales_data/shipping_dimen.csv")
orders_df =
pd.read_csv("../global_sales_data/orders_dimen.csv")
# Merging the dataframes to create a master_df
df_1 = pd.merge(market_df, customer_df, how='inner',
on='Cust_id')
df_2 = pd.merge(df_1, product_df, how='inner', on='Prod_id')
df_3 = pd.merge(df_2, shipping_df, how='inner', on='Ship_id')
master_df = pd.merge(df_3, orders_df, how='inner',
on='Ord_id')
master_df.head()
```

Output:



	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	...	Region	Customer_Segment
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	...	WEST	CORPORATE
1	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	...	WEST	CORPORATE
2	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.02	0.03	23	-47.64	6.15	0.37	...	WEST	CORPORATE
3	Ord_2978	Prod_16	SHP_4112	Cust_1088	305.05	0.04	27	23.12	3.37	0.57	...	ONTARIO	HOME OFFICE
4	Ord_5484	Prod_16	SHP_7663	Cust_1820	322.82	0.05	35	-17.58	3.98	0.56	...	WEST	CONSUMER

5 rows × 22 columns

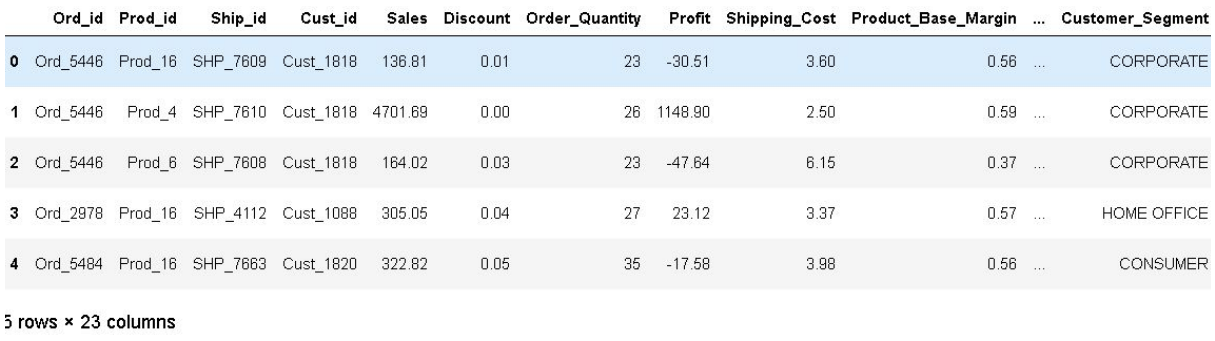


_Margin	...	Region	Customer_Segment	Product_Category	Product_Sub_Category	Order_ID_x	Ship_Mode	Ship_Date	Order_ID_y	Order_Date	Order_Priority
0.56	...	WEST	CORPORATE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	36262	REGULAR AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED
0.59	...	WEST	CORPORATE	TECHNOLOGY	TELEPHONES AND COMMUNICATION	36262	EXPRESS AIR	27-07-2010	36262	27-07-2010	NOT SPECIFIED
0.37	...	WEST	CORPORATE	OFFICE SUPPLIES	PAPER	36262	EXPRESS AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED
0.57	...	ONTARIO	HOME OFFICE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	37863	REGULAR AIR	26-02-2011	37863	24-02-2011	HIGH
0.56	...	WEST	CONSUMER	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	53026	REGULAR AIR	03-03-2012	53026	26-02-2012	LOW

Lambda Functions

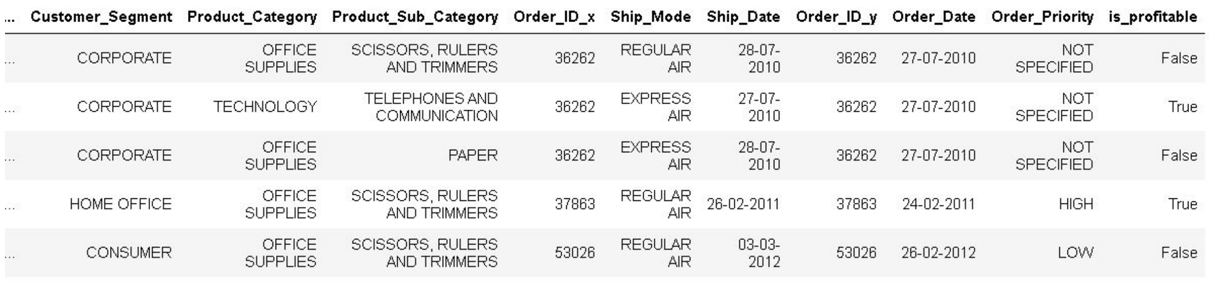
```
# Create a function to be applied
def is_positive(x):
    return x > 0
# Create a new column
master_df['is_profitable'] =
master_df['Profit'].apply(is_positive)
master_df.head()
                              #Or
# Create a new column using a lambda function
master_df['is_profitable'] = master_df['Profit'].apply(lambda x: x
> 0)
master_df.head()
```

Output:



	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_Margin	...	Customer_Segment
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	3.60	0.56	...	CORPORATE
1	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	2.50	0.59	...	CORPORATE
2	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.02	0.03	23	-47.64	6.15	0.37	...	CORPORATE
3	Ord_2978	Prod_16	SHP_4112	Cust_1088	305.05	0.04	27	23.12	3.37	0.57	...	HOME OFFICE
4	Ord_5484	Prod_16	SHP_7663	Cust_1820	322.82	0.05	35	-17.58	3.98	0.56	...	CONSUMER

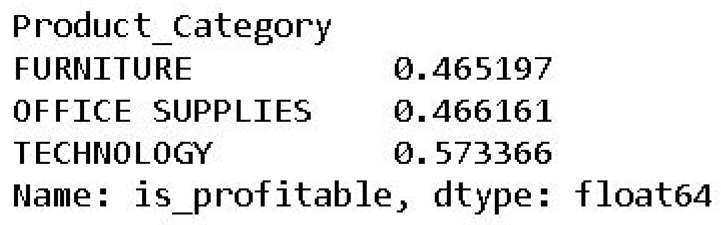
5 rows × 23 columns



...	Customer_Segment	Product_Category	Product_Sub_Category	Order_ID_x	Ship_Mode	Ship_Date	Order_ID_y	Order_Date	Order_Priority	is_profitable
...	CORPORATE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	36262	REGULAR AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED	False
...	CORPORATE	TECHNOLOGY	TELEPHONES AND COMMUNICATION	36262	EXPRESS AIR	27-07-2010	36262	27-07-2010	NOT SPECIFIED	True
...	CORPORATE	OFFICE SUPPLIES	PAPER	36262	EXPRESS AIR	28-07-2010	36262	27-07-2010	NOT SPECIFIED	False
...	HOME OFFICE	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	37863	REGULAR AIR	26-02-2011	37863	24-02-2011	HIGH	True
...	CONSUMER	OFFICE SUPPLIES	SCISSORS, RULERS AND TRIMMERS	53026	REGULAR AIR	03-03-2012	53026	26-02-2012	LOW	False

```
# Comparing percentage of profitable orders across product categories
by_category = master_df.groupby('Product_Category')
by_category.is_profitable.mean()
```

Output:



```
Product_Category
FURNITURE           0.465197
OFFICE SUPPLIES     0.466161
TECHNOLOGY          0.573366
Name: is_profitable, dtype: float64
```

Pivot Tables

You can use Pandas pivot tables as an alternative to groupby(). They provide Excel-like functionalities to create aggregate tables.
The general syntax is pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', ...)
where,

- ‘data’ is the data frame,
- ‘values’ contain the column to aggregate within the dataset,
- ‘index’ is the row in the pivot table,
- ‘columns’ contain the columns that you want in the pivot table, and
- ‘aggfunc’ is the aggregate function.

We will now see the data under the Product_Category column as shown below:

```
master_df.pivot_table(columns = 'Product_Category')
```

Output:

```
                         Sales
Customer_Segment
      CONSUMER     1857.859965
     CORPORATE     1787.680389
   HOME OFFICE     1754.312931
SMALL BUSINESS     1698.124841
```

We will compute mean of numeric across categories as shown in code snippet below

```
# Computes the mean of all numeric columns across categories
# Notice that the means of Order_IDs are meaningless
master_df.pivot_table(columns = 'Product_Category')
```

Output:

```
   Product_Category     FURNITURE  OFFICE SUPPLIES    TECHNOLOGY
           Discount      0.049287         0.050230      0.048746
         Order_ID_x  30128.711717     30128.122560  29464.891525
         Order_ID_y  30128.711717     30128.122560  29464.891525
     Order_Quantity     25.709977        25.656833     25.266344
Product_Base_Margin      0.598555         0.461270      0.556305
             Profit     68.116531       112.369544    429.208668
              Sales   3003.822820       814.048178   2897.941008
      Shipping_Cost     30.883811         7.829829      8.954886
      is_profitable      0.465197         0.466161      0.573366
     profit_per_qty     -3.607020         1.736175    -52.274216
```

Getting and Cleaning Data

➔ Getting Data From Text Files

The easiest way to read delimited files is by using pd.read_csv(file path, sep, header) and specifying a separator (delimiter), and code for the same is shown below.

```
import numpy as np
import pandas as pd

# reading companies file: throws an error
# companies = pd.read_csv("companies.txt", sep="\t")
# Using encoding = "ISO-8859-1"
companies = pd.read_csv("companies.txt", sep="\t", encoding =
"ISO-8859-1")
companies.head()
```

Output:

	permalink	name	homepage_url	category_list	status	country_code	state_code	region	city	founded_at
0	/Organization/-Fame	#fame	http://livfame.com	Media	operating	IND	16	Mumbai	Mumbai	NaN
1	/Organization/-Qounter	:Qounter	http://www.qounter.com	Application Platforms|Real Time|Social Network...	operating	USA	DE	DE - Other	Delaware City	04-09-2014
2	/Organization/-The-One-Of-Them-Inc-	(THE) ONE of THEM,Inc.	http://oneofthem.jp	Apps|Games|Mobile	operating	NaN	NaN	NaN	NaN	NaN
3	/Organization/0-6-Com	0-6.com	http://www.0-6.com	Curated Web	operating	CHN	22	Beijing	Beijing	01-01-2007
4	/Organization/004-Technologies	004 Technologies	http://004gmbh.de/en/004-interact	Software	operating	USA	IL	Springfield, Illinois	Champaign	01-01-2010

➔ Getting Data From a Relational Database

There are many libraries to connect MySQL and Python, such as PyMySQL and MySQLdb. All of these libraries follow the procedure mentioned below to connect to MySQL:

- Create a connection object between MySQL and Python.
- Create a cursor object (You can use the cursor to open and close the connection.).
- Execute the SQL query.
- Retrieve results of the query using methods such as fetchone() and fetchall().

```
import pymysql

# create a connection object 'conn'
conn = pymysql.connect(host="localhost", # your host, localhost
for your local machine
                     user="root", # your username, usually "root"
for localhost
                      passwd="yourpassword", # your password
                      db="world") # name of the data base; world
comes inbuilt with mysql

# create a cursor object c
c = conn.cursor()

# execute a query using c.execute
c.execute("select * from city;")

# getting the first row of data as a tuple
all_rows = c.fetchall()

# to get only the first row, use c.fetchone() instead
```

➔ Getting Data From Websites

Web scraping refers to the art of programmatically getting data from the internet. One of the best features of Python is that it makes it easy to scrape websites.

In Python 3, the most popular library for web scraping is BeautifulSoup. To use BeautifulSoup, you need the requests module, which connects to a given URL and fetches data from it (in HTML format). A web page is HTML code, and the main use of BeautifulSoup is that it helps you parse HTML easily.

Note: The discussion on HTML syntax is beyond the scope of this module. However, an extremely basic HTML experience should be enough to understand web scraping.

- Use Case - Fetching Mobile App Reviews From Google Play Store

Suppose you want to understand why people install and uninstall mobile apps, and why they like or dislike certain apps. An extremely rich source of app reviews data is Google Play Store, where people write their feedback about an app.

The reviews of the Facebook Messenger app are given in the link provided below:
https://play.google.com/store/apps/details?id=com.facebook.orca&hl=en

We will scrape the reviews of the Facebook Messenger app, i.e., get them into Python, and then, you can perform some interesting analyses on the same.

- Parsing HTML Data Using BeautifulSoup and Requests Module.

To use BeautifulSoup, you need to install it using pip install beautifulsoup4, and load the module bs4 using import bs4. You also need to install the requests module using pip install requests.

The general procedure to get data from websites is as follows:

1. Use requests to connect to a URL and get data from it.
2. Create a BeautifulSoup object.
3. Get attributes of the BeautifulSoup object (i.e., the HTML elements that you want).

Code for web scraping reviews from Google Play Store as shown in code snippet below:

```
# Import useful libraries and classes.
from urllib.request import urlopen as uReq
from bs4 import BeautifulSoup as soup
import io
#html upload
my_url=
"https://play.google.com/store/apps/details?id=com.facebook.orca&hl=en&s
howAllReviews=true"
uClient= uReq(my_url)
page_html= uClient.read()
uClient.close()
page_soup=soup(page_html,"html.parser")
containers=page_soup.findAll("div",{"class":"d15Mdf bAhLNe"})
print(len(containers))
container=containers[0]
name=(container.findAll("span",{"class":"X43Kjb"}))
print(name[0].text)
review=(container.findAll("div",{"class":"UD7Dzf"}))
print(review[0].text)
date=(container.findAll("span",{"class":"p2TkOb"}))
print(date[0].text)
filename="review.csv"
f=open(filename,"w",encoding='utf-8')
headers="Name,Date,Review\n"
f.write(headers)
```

```
for container in containers:
    name_1=container.findAll("span",{"class":"X43Kjb"})
    name_final=name_1[0].text;
    date_1=(container.findAll("span",{"class":"p2TkOb"}))
    date_final=date_1[0].text
    review_1=(container.findAll("div",{"class":"UD7Dzf"}))
    review_final=review_1[0].text;

print(name_final+","+date_final.replace(",","")+","+review_final.replace
(",","")+"\n")

f.write(name_final+","+date_final.replace(",","")+","+review_final.repla
ce(",","")+"\n")
f.close()
```

➔ Getting Data From APIs

Application programming interfaces or APIs are created by companies and organisations to provide restricted access to data. It is extremely common to get data from APIs for data analysis. For example, you can get financial data (stock prices, etc.), social media data (Facebook, Twitter, etc., provide APIs), weather data, data on healthcare, music, food and drinks, and data from almost every domain.

Apart from being rich sources of data, the other reasons to use APIs are as follows:

- When the data is getting updated in real-time: If you use downloaded CSV files, then you have to download data manually and update your analysis multiple times. However, through APIs, you can automate the process of getting real-time data.
- Easy access to structured and verified data: Even though websites can be scraped manually, APIs can directly provide good-quality data in a structured format.
- Access to restricted data: You cannot scrape all websites easily, as web scraping is usually considered illegal (For example, Facebook, financial data, etc.). APIs are the only way to access restricted data.

Example Use Case: Google Maps Geocoding API

https://maps.googleapis.com/maps/api/geocode/json?address=UpGrad,+Nishuvi+building,+Anne+Besant+Road,+Worli,+Mumbai&key=YOUR_API_KEY

Geocoding is a Three-step process, which is as follows:

- Join the words in the address using a plus sign and convert them into a form words+in+the+address.
- Connect to the URL by appending the address and the API key.
- Get a response from the API and convert it into a Python object (here, a dictionary).

Now we will find out latitude and longitude of upGrad's Mumbai office location so the code for the following is shown below:

```
import numpy as np
import pandas as pd

# Need requests to connect to the URL, json to convert JSON to dict
import requests, json
import pprint

# joining words in the address by a "+"
add = "UpGrad, Nishuvi building, Anne Besant Road, Worli, Mumbai"
split_address = add.split(" ")
address = "+".join(split_address)
print(address)

api_key = "AIzaSyBXrK8md7uaOcpRpaluEGZAtdXS4pcI5xo"

url =
"https://maps.googleapis.com/maps/api/geocode/json?address={0}&key={1}".
format(address, api_key)
r = requests.get(url)

# The r.text attribute contains the text in the response object
print(type(r.text))
print(r.text)
# converting the json object to a dict using json.loads()
r_dict = json.loads(r.text)

# the pretty printing library pprint makes it easy to read large
dictionaries
pprint.pprint(r_dict)
# The dict has two main keys - status and results
r_dict.keys()
pprint.pprint(r_dict['results'])
lat = r_dict['results'][0]['geometry']['location']['lat']
lng = r_dict['results'][0]['geometry']['location']['lng']

print((lat, lng))
# Input to the fn: Address in standard human-readable form
# Output: Tuple (lat, lng)

api_key = "AIzaSyBXrK8md7uaOcpRpaluEGZAtdXS4pcI5xo"


def address_to_latlong(address):
    # convert address to the form x+y+z
    split_address = address.split(" ")
    address = "+".join(split_address)

    # pass the address to the URL
    url =
"https://maps.googleapis.com/maps/api/geocode/json?address={0}&key={1}".
format(address, api_key)

    # connect to the URL, get response and convert to dict
    r = requests.get(url)
    r_dict = json.loads(r.text)
    lat = r_dict['results'][0]['geometry']['location']['lat']
    lng = r_dict['results'][0]['geometry']['location']['lng']

    return (lat, lng)


# getting some coordinates
print(address_to_latlong("UpGrad, Nishuvi Building, Worli, Mumbai"))
print(address_to_latlong("IIIT Bangalore, Electronic City, Bangalore"))

# Importing addresses file
add = pd.read_csv("addresses.txt", sep="\t", header = None)
add.head()

# renaming the column
add = add.rename(columns={0:'address'})
add.head()
add.head()['address'].apply(address_to_latlong)
```

To summarise, the steps involved in the procedure for getting lat/long coordinates from an address are as follows:

- Convert the address into a suitable format and connect to the Google Maps URL using your key.
- Get a response from the API and convert it into a dictionary using json.loads(r.text).

- Get the lat/long coordinates using lat = r_dict['results'][0]['geometry']['location']['lat'] and analogous for longitude.

➔ Getting Data From PDFs

Reading PDF files is not as straightforward as reading text or delimited files, since PDFs often contain images, tables, etc. PDFs are mainly designed to be human readable, and thus, you need special libraries to read them in Python (or any other programming language).

There are some excellent libraries in Python. We will use PyPDF2 to read PDFs in Python, since it is easy to use and works with most types of PDFs.

Note that Python will only be able to read text from PDFs, and not from images, tables, etc. (which is possible using other specialised libraries).

You can install PyPDF2 using pip install PyPDF2.

For this illustration, we will read a PDF of the book *Animal Farm* written by George Orwell.

Code for the same is as follows:

```
import PyPDF2

# reading the pdf file
pdf_object = open('animal_farm.pdf', 'rb')
pdf_reader = PyPDF2.PdfFileReader(pdf_object)

# Number of pages in the PDF file
print(pdf_reader.numPages)

# get a certain page's text
page_object = pdf_reader.getPage(5)

# Extract text from the page_object
print(page_object.extractText())
```

➔ Handling Missing Data

There are various reasons for missing data such as human errors during data entry, and non-availability at the end of the user (for instance, DOB of certain people). Most often, the reasons are simply unknown.

In Python, missing data is represented using either of the two objects: NaN (Not a Number) or NULL. We will not look at the differences between these two objects and how Python stores them internally. But, we will learn about the different ways to identify and treat missing values in Pandas data frames.

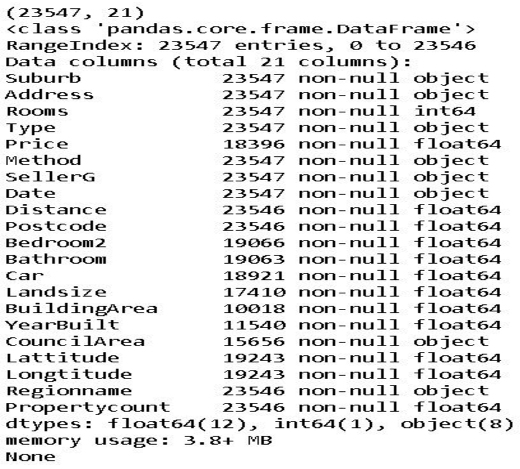
The four main methods to identify and treat missing data are as follows:

- isnull(): This method indicates the presence of missing values and returns a boolean.
- notnull(): This method is opposite of the isnull() method and returns a boolean.
- dropna(): This method drops the missing values from a data frame and returns the remaining values.
- fillna(): This method fills (or imputes) the missing values with a specified value.

To start with missing data we will read the csv file by importing Pandas and NumPy libraries.

```python
import numpy as np
import pandas as pd
df = pd.read_csv("melbourne.csv")
print(df.shape)
print(df.info())
```

Output:



```
(23547, 21)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 23547 entries, 0 to 23546
Data columns (total 21 columns):
Suburb                 23547 non-null object
Address                23547 non-null object
Rooms                  23547 non-null int64
Type                   23547 non-null object
Price                  18396 non-null float64
Method                 23547 non-null object
SellerG                23547 non-null object
Date                   23547 non-null object
Distance               23546 non-null float64
Postcode               23546 non-null float64
Bedroom2               19066 non-null float64
Bathroom               19063 non-null float64
Car                    18921 non-null float64
Landsize               17410 non-null float64
BuildingArea           10018 non-null float64
YearBuilt              11540 non-null float64
CouncilArea            15656 non-null object
Lattitude              19243 non-null float64
Longtitude             19243 non-null float64
Regionname             23546 non-null object
Propertycount          23546 non-null float64
dtypes: float64(12), int64(1), object(8)
memory usage: 3.8+ MB
None
```

➔ Identifying Missing Data

The isnull() and notnull() methods are the most common ways of identifying missing values.

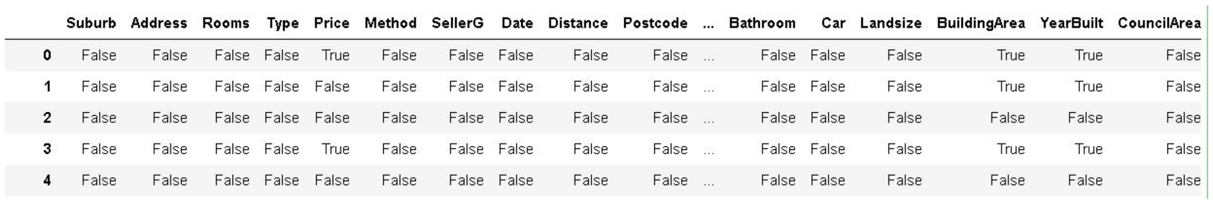
While handling missing data, you first need to identify the rows and columns containing missing values, count the number of missing values, and then decide how you want to treat them.

You must treat **missing values in each column separately**, rather than implementing a single solution (for example, replacing NaNs by the mean of a column) for all columns.

isnull() returns a boolean (True/False), which can be used to find the rows or columns containing missing values. The code for the isnull() method is shown below.

```
# isnull()
df.isnull()
```

Output:



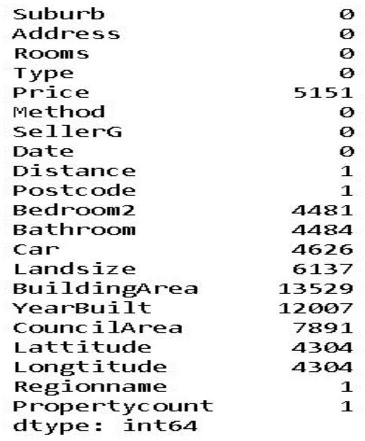
	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	...	Bathroom	Car	Landsize	BuildingArea	YearBuilt	CouncilArea
0	False	False	False	False	True	False	False	False	False	False	...	False	False	False	True	True	False
1	False	False	False	False	False	False	False	False	False	False	...	False	False	False	True	True	False
2	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False
3	False	False	False	False	True	False	False	False	False	False	...	False	False	False	True	True	False
4	False	False	False	False	False	False	False	False	False	False	...	False	False	False	False	False	False

➔ Identifying Missing Data in Columns

Let's first compute the total number of missing values in the data frame. You can calculate the number of missing values in each column by using df.isnull().sum().

```
# summing up the missing values (column-wise)
df.isnull().sum()
```

Output:



```
Suburb               0
Address              0
Rooms                0
Type                 0
Price             5151
Method               0
SellerG              0
Date                 0
Distance             1
Postcode             1
Bedroom2          4481
Bathroom          4484
Car               4626
Landsize          6137
BuildingArea     13529
YearBuilt        12007
CouncilArea       7891
Lattitude         4304
Longtitude        4304
Regionname           1
Propertycount        1
dtype: int64
```

➔ Treating Missing Data

Let's now treat missing values in columns. Take a look at the number of NaNs in each column once again, but this time as the *percentage of missing values in each column*. Observe that we will calculate the number of rows by len(df.index).

```
# summing up the missing values (column-wise)
```

```
round(100*(df.isnull().sum()/len(df.index)), 2)
```

Output:

```
Suburb            0.00
Address           0.00
Rooms             0.00
Type              0.00
Price            21.88
Method            0.00
SellerG           0.00
Date              0.00
Distance          0.00
Postcode          0.00
Bedroom2         19.03
Bathroom         19.04
Car              19.65
Landsize         26.06
BuildingArea     57.46
YearBuilt        50.99
CouncilArea      33.51
Lattitude        18.28
Longtitude       18.28
Regionname        0.00
Propertycount     0.00
dtype: float64
```

Observe that the columns have around 22%, 19%, 26%, 57%, etc., of missing values. When dealing with a column, you have two simple choices: either **delete** or **retain the column.** If you retain the column, then you have to treat (i.e., delete or impute) the rows having missing values.

If you delete the missing rows, then you lose data. And if you impute, then you introduce bias.

Apart from the number of missing values, the decision to delete or retain a variable depends on various other factors, which are as follows:

- The analysis task at hand
- The usefulness of the variable (based on your understanding of the problem)
- The total size of available data (If you have enough, then you can afford to throw away some of it.)
-

Suppose we want to build a (linear regression) model to predict the house prices in Melbourne. Now, even though the variable Price has about 22% of missing values, you cannot drop the variable, since that is what you want to predict.

Similarly, you would expect some other variables such as Bedroom2, Bathroom and Landsize to be the important predictors of Price, and thus, you cannot remove those columns.

We will drop the columns which have more missing values such as Building Area, YearBuilt and Council Area so the code for this is shown below.

```
# removing the three columns
df = df.drop('BuildingArea', axis=1)
df = df.drop('YearBuilt', axis=1)
df = df.drop('CouncilArea', axis=1)

round(100*(df.isnull().sum()/len(df.index)), 2)
```

Output:

```
Suburb            0.00
Address           0.00
Rooms             0.00
Type              0.00
Price            21.88
Method            0.00
SellerG           0.00
Date              0.00
Distance          0.00
Postcode          0.00
Bedroom2         19.03
Bathroom         19.04
Car              19.65
Landsize         26.06
Lattitude        18.28
Longtitude       18.28
Regionname        0.00
Propertycount     0.00
dtype: float64
```

➔ Treating Missing Data in Rows

Now, we need to either delete or impute the missing values. First, let's see whether or not any rows have a significant number of missing values. If so, we can drop those rows, and then decide to delete or impute the rest.

After dropping three columns, we now have 18 columns to work with. To inspect rows with missing values, let's take a look at the rows having more than five missing values.

```
# retaining the rows having <= 5 NaNs
df = df[df.isnull().sum(axis=1) <= 5]

# look at the summary again
round(100*(df.isnull().sum()/len(df.index)), 2)
```

Output:

```
Suburb            0.00
Address           0.00
Rooms             0.00
Type              0.00
Price            21.71
Method            0.00
SellerG           0.00
Date              0.00
Distance          0.00
Postcode          0.00
Bedroom2          1.05
Bathroom          1.07
Car               1.81
Landsize          9.65
Lattitude         0.13
Longtitude        0.13
Regionname        0.00
Propertycount     0.00
dtype: float64
```

Observe that we now have removed most of the rows where multiple columns (Bedroom2, Bathroom, Landsize) were missing.

Now, we still have around 21% of missing values in the Price column and 9% in the Landsize column. Since the Price column still contains a lot of missing data (and imputing 21% of values of a variable you want to predict will introduce heavy bias), it is not sensible to impute those values.

Thus, let's remove the missing rows from the Price column as well. Notice that you can use np.isnan(df['column']) to filter out the corresponding rows, and then use a ~ to discard the values satisfying the condition.

```
# removing NaN Price rows
df = df[~np.isnan(df['Price'])]

round(100*(df.isnull().sum()/len(df.index)), 2)
```

Output:

```
Suburb           0.00
Address          0.00
Rooms            0.00
Type             0.00
Price            0.00
Method           0.00
SellerG          0.00
Date             0.00
Distance         0.00
Postcode         0.00
Bedroom2         1.05
Bathroom         1.07
Car              1.76
Landsize         9.83
Lattitude        0.15
Longtitude       0.15
Regionname       0.00
Propertycount    0.00
dtype: float64
```

Now, you have Land Size as the only variable with a significant number of missing values. Let's consider this variable and try imputing the NaNs.

The decision (whether and how to impute) will depend upon the distribution of the variable. For example, if the variable is such that all the observations lie in a short range (say between 800 sq. ft to 820 sq.ft), then you can make a call to impute the missing values by something similar to the mean or median Land Size.

So, we have to check the data repeatedly to confirm that there is no missing data or wrong data in the dataset, and if it is present, then we have to remove the missing data and perform the operations.