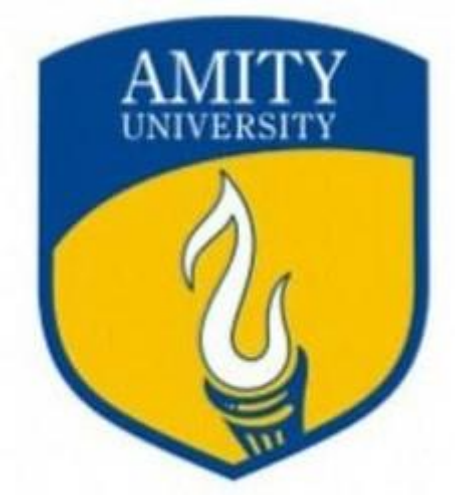
FUNDAMENTALS OF DATA ANALYTICS

AIML301

Practical file – 2022



AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY
AMITY UNIVERSITY, UTTAR PRADESH

Submitted To:

Dr. Krishna Kant Singh

Submitted By:

Manya Tuli

A2305220091

5CSE2X

TABLE OF CONTENTS

EXPERIMENT- 1

AIM:

Python environment set up and essentials

SOFTWARE USED:

Python 3.7.3, Jupyter Notebook 6.0.1, Pandas 0.25.1

THEORY AND SOURCE CODE

1. PYTHON ENVIRONMENT SETUP

The following links can be used to download Python and Jupyter Notebook:

Python: https://www.python.org/downloads/ (https://www.python.org/downloads/)

Jupyter Notebook: https://jupyter.org/install (https://jupyter.org/install)

2. PYTHON ESSENTIALS

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics.

Variable Names

1. Permissible characters: letters(Upper and Lower), numbers and undescore(_)
2. It should start with either letter or underscore
3. It is case sensitive

In [ ]:

```
x = 1.0  # Declaration
```

In [ ]:

```
x = 1
```

In [4]:

```
x = 'Hello world'
```

Standard Data Types

String, float, int, long, bool

Dynamic references -> Unlike many compiled languages, such as Java and C++, object references in Python have no type associated with them.

In [5]:

```
a = 5
type(a)   # int
```

Out[5]:

```
int
```

In [6]:

```
a = 1.0
type(a)   # Float
```

Out[6]:

```
float
```

In [7]:

```
a = True
type(a)  # Boolean
```

Out[7]:

```
bool
```

In [8]:

```
a = 'Hello world'
type(a)  # String  -- Str
```

Out[8]:

```
str
```

String

** For string we can use either single quotes ' or double quotes " **

In [11]:

```
temp_string = 'Hello World'   # Single quote
print(temp_string)
```

```
Hello World
```

In [10]:

```
temp_string = "Hello World" # Double quote
print(temp_string)
```

```
Hello World
```

Above both statements are fine. This is particularly useful, when you have ' or " in the data itself.

In [ ]:

```
single_quote_string = "'Wow!' Great place"    # Single quote is part of the string.
print(a)
```

In [ ]:

```
double_quote_string = '"Wow!" Great place'    # Double quote is part of the string.
print(a)
```

But what if you have both ' and " in the string and/or if you can mulitple lines?

We could use triple quotes: ''' or """

In [14]:

```
multiline_string = """This is a
multiline
string.
"""

print(multiline_string)
```

```
This is a
multiline
string.
```

Printing

In [15]:

```
print_format_string = "World's population is more than %.2f %s."
print_format_string % (7.5, 'billion')
```

Out[15]:

```
"World's population is more than 7.50 billion."
```

In [17]:

```
# Another format
course_name = 'Python'
course_level = 'basic'

print('Welcome to {} course. This is a {} level course.' \
              .format(course_name,course_level))
```

```
Welcome to Python course. This is a basic level course.
```

Binary operators: +, -, /, **, //

In [19]:

```
a = 10
b = 2
```

In [20]:

```
# Addition
print(a + b)
```

```
12
```

In [22]:

```
# Subtraction
print(a - b)
```

```
8
```

In [23]:

```
# Modulus or Remainder
a%b
```

Out[23]:

```
0
```

In [27]:

```
# Quotient
a//b
```

Out[27]:

```
5
```

In [24]:

```
# Power - 10 ^ 2 i.e. square of 10
a**b
```

Out[24]:

```
100
```

In [28]:

```
# Comparison
a > b
```

Out[28]:

```
True
```

In [25]:

```
a == b
```

Out[25]:

```
False
```

Control flow

Condition -- if, elif, and else

In [29]:

```
a = -3
b = -5
```

In [31]:

```
# Note down the colon and Indentation, not braces

if a > 0 and b < 0:     # Logical operator AND
    print('a is positive and b is negative')
elif a > 0 and b > 0 :
    print('Both a and b are postive')
elif a < 0 and b < 0:
    print('Both a and b are negative')
else:
    print('a is negative and b is postive')
```

```
Both a and b are negative
```

In [32]:

```
# Loops
# For loop is discussed below.
```

In [33]:

```
# While loop
i = 0
while i < 10:
    print(i)
    i=i+1
```

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

Data Structures

Python has data structures which are created to store the data in particular optimized format.

1. Tuple

** A tuple is a one-dimensional, fixed-length, immutable sequence of Python objects. **

In [34]:

```
first_tuple = 1,2,3
print(first_tuple)
```

```
(1, 2, 3)
```

In [36]:

```
first_tuple[0] = 1    # Tuples are immutable sequence
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-36-0f5074d673a4> in <module>()
----> 1 first_tuple[0] = 1    # Tuples are immutable sequence

TypeError: 'tuple' object does not support item assignment
```

In [37]:

```
# Unpacking tuples

unpack_tuple = (4, 5, 6)
first_element, second_element, third_element = unpack_tuple
```

In [38]:

```
print(first_element)
print(second_element)
print(third_element)
```

```
4
5
6
```

2. List

In [1]:

```
first_list = [0,1,2,3,4,5,6,7,8,9]
print(first_list)
```

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

In [2]:

```
first_list.append(10)
first_list
```

Out[2]:

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

In [ ]:

```
first_list.insert(1, 1.5)
first_list
```

Out[4]:

```
[0, 1, 2, 1.5, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [45]:

```
first_list.pop(1)
first_list
```

Out[45]:

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

Accessing and slicing

Python uses zero-based position index so the first element is at 0th position.

In [46]:

```
first_list[3:7]   # First index is inclusive and last index is exclusive
```

Out[46]:

```
[3, 4, 5, 6]
```

In [47]:

```
# First five elements
first_list[:5]
```

Out[47]:

```
[0, 1, 2, 3, 4]
```

In [48]:

```
# ALl items from 5th element
first_list[5:]
```

Out[48]:

```
[5, 6, 7, 8, 9, 10]
```

In [49]:

```
# Last 4 elements
first_list[-4:]
```

Out[49]:

```
[7, 8, 9, 10]
```

In [50]:

```
# The third parameter mentions to fetch the nth value (3rd parameter).
first_list[::2]
```

Out[50]:

```
[0, 2, 4, 6, 8, 10]
```

In [51]:

```
# One of the ways of reversing a list
first_list[::-1]
```

Out[51]:

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

3. Strings

In [53]:

```
# Strings are a sequence of characters and therefore can be treated like other sequences

str_list = 'hello world'
list(str_list)[:3]
```

Out[53]:

```
['h', 'e', 'l']
```

Sequence Functions

Range function - list of integers from start to stop with step size

In [55]:

```
for i in range(2,10,1):   # Consecutive numbers from 2 to 9
    print(i)
```

```
2
3
4
5
6
7
8
9
```

In [56]:

```
# enumerate - useful if we want to retain the index while iterating

list_numbers = range(120, 130)
for i, value in enumerate(list_numbers):
    print(i,value)
```

```
0 120
1 121
2 122
3 123
4 124
5 125
6 126
7 127
8 128
9 129
```

In [58]:

```
# zip “pairs” up the elements of a number of lists, tuples, or other sequences, to creat

seq1 = [1,2,3]
seq2 = ['model1', 'model2', 'model3']
```

In [59]:

```
list(zip(seq1, seq2))
```

Out[59]:

```
[(1, 'model1'), (2, 'model2'), (3, 'model3')]
```

4. Dict - This structure stores key-value pairs.

Most commonly used for data analysis. Similar to a Hash Table

In [63]:

```
temp_dict = {1: 'hello', 2: 'world'}
temp_dict
```

Out[63]:

```
{1: 'hello', 2: 'world'}
```

5. Set - An unordered collection of unique elements

Most commonly used when we have to find out the unique values in a list.

In [61]:

```
set([1,2,3,4,4,5,5,6,6,6,6,7])
```

Out[61]:

```
{1, 2, 3, 4, 5, 6, 7}
```

Comprehensions

List, Set, and Dict Comprehensions.

They allow you to concisely form a new list by filtering the elements of a collection and transforming the elements passing the filter in one concise expression.

In [66]:

```
# verbose
temp_list = [1,2,3,4]

multiple_two = []
for number in temp_list:
    multiple_two.append(number*2)

print(multiple_two)
```

```
[2, 4, 6, 8]
```

In [69]:

```
# List comprehension
[number*2 for number in temp_list]
```

Out[69]:

```
[2, 4, 6, 8]
```

Functions -- makes the code modular and reusable.

In [ ]:

```
def squaring_number(x):
    y = x**2
    return y
```

In [ ]:

```
print(squaring_number(10))
```

In [ ]:

```
# Most of times we might end up with a function returning multiple values

def squaring_number(x):
    y = x**2
    return x,y   # returning a tuple
```

In [ ]:

```
print(squaring_number(10))
```

Lambda functions

Anonymous/Lambda functions - Simple function with single statement. We dont need to name the function, hence the name anonymous.

In [71]:

```
list(map(lambda x: x**2, [1,2,3,4]))  # Here we see that we dont need to name the square
```

Out[71]:

```
[1, 4, 9, 16]
```

EXPERIMENT- 2

AIM:

Create Pandas, Series and Data Frame from various inputs

SOFTWARE:

Python 3.7.3, Jupyter Notebook 6.0.1, Pandas 0.25.1

THEORY AND SOURCE CODE

PANDAS

Pandas is a library built using NumPy specifically for data analysis. You'll be using Pandas heavily for data manipulation, visualisation, building machine learning models, etc.

There are two main data structures in Pandas - Series and Dataframes. The default way to store data is dataframes, and thus manipulating dataframes quickly is probably the most important skill set for data analysis.

Source: *https://pandas.pydata.org/pandas-docs/stable/overview.html (https://pandas.pydata.org/pandas-docs/stable/overview.html)*

This experiment covers:

1. The pandas Series (similar to a numpy array)
    - Creating a pandas series
    - Indexing series
2. Dataframes
    - Creating dataframes from dictionaries
    - Importing CSV data files as pandas dataframes
    - Reading and summarising dataframes
    - Sorting dataframes

1. PANDAS SERIES

A series is similar to a 1-D numpy array, and contains scalar values of the same type (numeric, character, datetime etc.). A dataframe is simply a table where each column is a pandas series.

Creating Pandas Series

Series are one-dimensional array-like structures, though unlike numpy arrays, they often contain non-numeric data (characters, dates, time, booleans etc.)

We can create pandas series from array-like objects using `pd.Series()` .

In [1]:

```
# 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))
```

```
0    2
1    4
2    5
3    6
4    9
dtype: int64
```

Note that each element in the Series has an index, and the index starts at 0 as usual.

In [26]:

```
# creating a series of characters
# notice that the 'dtype' here is 'object'
char_series = pd.Series(['a', 'b', 'af'])
char_series
```

Out[26]:

```
0     a
1     b
2    af
dtype: object
```

In [27]:

```
# creating a series of type datetime
date_series = pd.date_range(start = '11-09-2017', end = '12-12-2017')
date_series
```

Out[27]:

```
DatetimeIndex(['2017-11-09', '2017-11-10', '2017-11-11', '2017-11-12',
               '2017-11-13', '2017-11-14', '2017-11-15', '2017-11-16',
               '2017-11-17', '2017-11-18', '2017-11-19', '2017-11-20',
               '2017-11-21', '2017-11-22', '2017-11-23', '2017-11-24',
               '2017-11-25', '2017-11-26', '2017-11-27', '2017-11-28',
               '2017-11-29', '2017-11-30', '2017-12-01', '2017-12-02',
               '2017-12-03', '2017-12-04', '2017-12-05', '2017-12-06',
               '2017-12-07', '2017-12-08', '2017-12-09', '2017-12-10',
               '2017-12-11', '2017-12-12'],
              dtype='datetime64[ns]', freq='D')
```

Indexing Series

Indexing series is exactly same as 1-D numpy arrays - index starts at 0.

In [28]:

```
# 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:]
```

Out[28]:

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

In [29]:

```
# accessing the second and the fourth elements
# note that s[1, 3] will not work, you need to pass the indices [1, 3] as a list inside
s[[0,2,4]]
```

Out[29]:

```
0    2
2    5
4    9
dtype: int64
```

Explicitly specifying indices

While creating a series, Pandas automatically indexes it from 0 to (n-1), n being the number of rows. But if we want, we can also explicitly set the index ourselves, using the 'index' argument while creating the series using `pd.Series()`

In [30]:

```
# Indexing explicitly
pd.Series([0, 5, 2], index = ['a', 'b', 'c'])
```

Out[30]:

```
a    0
b    5
c    2
dtype: int64
```

In [31]:

```
# 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 nu

pd.Series(np.array(range(0,10))**2, index = range(0,10))
```

Out[31]:

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

Usually, you will work with Series only as a part of dataframes

2. PANDAS DATAFRAME

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

Creating dataframes from dictionaries

There are various ways of creating dataframes, such as creating them from dictionaries, JSON objects, reading from txt, CSV files, etc.

In [8]:

```
# keys become column names
df = pd.DataFrame({'name': ['Vinay', 'Kushal', 'Aman', 'Rahul'],
                   'age': [22, 25, 24, 28],
                    'occupation': ['engineer', 'doctor', 'data analyst', 'teacher'],
                    'random': [1, 2, 3, 4]})
df
```

Out[8]:

	name	age	occupation	random
0	Vinay	22	engineer	1
1	Kushal	25	doctor	2
2	Aman	24	data analyst	3
3	Rahul	28	teacher	4

In [19]:

```
df.iloc[2:4,2]
```

Out[19]:

```
2    data analyst
3         teacher
Name: occupation, dtype: object
```

In [18]:

```
df.loc[2:4,'occupation']
```

Out[18]:

```
2    data analyst
3         teacher
Name: occupation, dtype: object
```

In [14]:

```
df2=df.drop(['name','occupation'],axis=1)
```

In [15]:

```
df2
```

Out[15]:

	age	random
0	22	1
1	25	2
2	24	3
3	28	4

In [16]:

```
df2[df2<24]=0
```

In [17]:

```
df2
```

Out[17]:

	age	random
0	0	0
1	25	0
2	24	0
3	28	0

Importing CSV data files as pandas dataframes

For the upcoming exercises, we will use a dataset of a retail store having details about the orders placed, customers, product details, sales, profits etc.

In [33]:

```
# reading a CSV file as a dataframe
market_df = pd.read_csv(r"C:\programming languages\Data science\market_fact.csv")
```

Usually, dataframes are imported as CSV files, but sometimes it is more convenient to convert dictionaries into dataframes. For e.g. when the raw data is in a JSON format (which is not uncommon), you can easily convert it into a dictionary, and then into a dataframe.

You will learn how to convert JSON objects to dataframes later.

Reading and Summarising Dataframes

After importing a dataframe, let's see its structure, shape, meanings of rows and columns etc. Fut to look at summary statistics - such as mean, percentiles etc.

In [34]:

```
# Looking at the top and bottom entries of dataframes
market_df.head()
```

Out[34]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

In [35]:

```
market_df.tail()
```

Out[35]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	S
8394	Ord_5353	Prod_4	SHP_7479	Cust_1798	2841.4395	0.08	28	374.63	
8395	Ord_5411	Prod_6	SHP_7555	Cust_1798	127.1600	0.10	20	-74.03	
8396	Ord_5388	Prod_6	SHP_7524	Cust_1798	243.0500	0.02	39	-70.85	
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.34	
8398	Ord_5459	Prod_6	SHP_7628	Cust_1798	603.6900	0.00	47	131.39	

Here, each row represents an order placed at a retail store. Notice the index associated with each row - starts at 0 and ends at 8398, implying that there were 8399 orders placed.

In [36]:

```
# Looking at the datatypes of each column
market_df.info()

# Note that each column is basically a pandas Series of length 8399
# The ID columns are 'objects', i.e. they are being read as characters
# The rest are numeric (floats or int)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8399 entries, 0 to 8398
Data columns (total 10 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Ord_id               8399 non-null   object 
 1   Prod_id              8399 non-null   object 
 2   Ship_id              8399 non-null   object 
 3   Cust_id              8399 non-null   object 
 4   Sales                8399 non-null   float64
 5   Discount             8399 non-null   float64
 6   Order_Quantity       8399 non-null   int64  
 7   Profit               8399 non-null   float64
 8   Shipping_Cost        8399 non-null   float64
 9   Product_Base_Margin  8336 non-null   float64
dtypes: float64(5), int64(1), object(4)
memory usage: 656.3+ KB
```

In [37]:

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

Out[37]:

	Sales	Discount	Order_Quantity	Profit	Shipping_Cost	Product_Base_
count	8399.000000	8399.000000	8399.000000	8399.000000	8399.000000	8336.
mean	1775.878179	0.049671	25.571735	181.184424	12.838557	0.
std	3585.050525	0.031823	14.481071	1196.653371	17.264052	0.
min	2.240000	0.000000	1.000000	-14140.700000	0.490000	0.
25%	143.195000	0.020000	13.000000	-83.315000	3.300000	0.
50%	449.420000	0.050000	26.000000	-1.500000	6.070000	0.
75%	1709.320000	0.080000	38.000000	162.750000	13.990000	0.
max	89061.050000	0.250000	50.000000	27220.690000	164.730000	0.

In [38]:

```
# Column names
market_df.columns
```

Out[38]:

```
Index(['Ord_id', 'Prod_id', 'Ship_id', 'Cust_id', 'Sales', 'Discount',
       'Order_Quantity', 'Profit', 'Shipping_Cost', 'Product_Base_Margi
n'],
      dtype='object')
```

In [39]:

```
# The number of rows and columns
market_df.shape
```

Out[39]:

```
(8399, 10)
```

In [40]:

```
# You can extract the values of a dataframe as a numpy array using df.values
market_df.values
```

Out[40]:

```
array([['Ord_5446', 'Prod_16', 'SHP_7609', ..., -30.51, 3.6, 0.56],
       ['Ord_5406', 'Prod_13', 'SHP_7549', ..., 4.56, 0.93, 0.54],
       ['Ord_5446', 'Prod_4', 'SHP_7610', ..., 1148.9, 2.5, 0.59],
       ...,
       ['Ord_5388', 'Prod_6', 'SHP_7524', ..., -70.85, 5.35, 0.4],
       ['Ord_5348', 'Prod_15', 'SHP_7469', ..., 565.34, 30.0, 0.62],
       ['Ord_5459', 'Prod_6', 'SHP_7628', ..., 131.39, 4.86, 0.38]],
      dtype=object)
```

Indices

An important concept in pandas dataframes is that of *row indices*. By default, each row is assigned indices starting from 0, and are represented at the left side of the dataframe.

In [41]:

```
market_df.head()
```

Out[41]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

Now, arbitrary numeric indices are difficult to read and work with. Thus, you may want to change the indices of the df to something more meanigful.

Let's change the index to Ord_id (unique id of each order), so that you can select rows using the order ids directly.

In [42]:

```
# Setting index to Ord_id
market_df.set_index('Ord_id', inplace = True)
market_df.head()
```

Out[42]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping
Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

Having meaningful row labels as indices helps you to select (subset) dataframes easily. You will study selecting dataframes in the next section.

Sorting dataframes

You can sort dataframes in two ways - 1) by the indices and 2) by the values.

In [43]:

```
# 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)
```

Out[43]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping
Ord_999	Prod_15	SHP_1383	Cust_361	5661.08	0.00	33	1055.47	
Ord_998	Prod_8	SHP_1380	Cust_372	750.66	0.00	33	120.05	
Ord_998	Prod_5	SHP_1382	Cust_372	2149.37	0.03	42	217.87	
Ord_998	Prod_8	SHP_1381	Cust_372	254.32	0.01	8	-117.39	
Ord_997	Prod_14	SHP_1379	Cust_365	28761.52	0.04	8	285.11	
...	...	...	...	...	...	...	...	
Ord_1001	Prod_5	SHP_1385	Cust_374	1981.26	0.07	49	100.80	
Ord_1000	Prod_6	SHP_1384	Cust_373	334.71	0.01	25	31.74	
Ord_100	Prod_8	SHP_138	Cust_58	121.12	0.10	3	-118.82	
Ord_10	Prod_3	SHP_13	Cust_10	80.61	0.02	15	-4.72	
Ord_1	Prod_1	SHP_1	Cust_1	261.54	0.04	6	-213.25	

8399 rows × 9 columns

In [44]:

```
# Sorting by values

# Sorting in increasing order of Sales
market_df.sort_values(by='Sales').head()
```

Out[44]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_Cos
Ord_704	Prod_7	SHP_964	Cust_242	2.24	0.01	1	-1.97	0.7
Ord_149	Prod_3	SHP_7028	Cust_1712	3.20	0.09	1	-3.16	1.4
Ord_4270	Prod_7	SHP_5959	Cust_1450	3.23	0.06	2	-2.73	0.7
Ord_4755	Prod_13	SHP_6628	Cust_1579	3.41	0.06	1	-1.78	0.7
Ord_2252	Prod_3	SHP_3064	Cust_881	3.42	0.05	1	-2.91	1.4

In [45]:

```
# Sorting in decreasing order of Shipping_Cost
market_df.sort_values(by='Shipping_Cost', ascending = False).head()
```

Out[45]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shippi
Ord_1751	Prod_15	SHP_2426	Cust_597	14740.510	0.00	46	3407.73	
Ord_839	Prod_11	SHP_1361	Cust_364	12689.870	0.04	44	-169.23	
Ord_1741	Prod_11	SHP_2411	Cust_595	15168.820	0.02	26	-1096.78	
Ord_417	Prod_11	SHP_561	Cust_156	20333.816	0.02	45	-1430.45	
Ord_1581	Prod_15	SHP_2184	Cust_519	2573.920	0.07	17	117.23	

In [46]:

```
# Sorting by more than two columns

# Sorting in ascending order of Sales for each Product
market_df.sort_values(by=['Prod_id', 'Sales'], ascending = False)
```

Out[46]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping_
Ord_2197	Prod_9	SHP_2994	Cust_827	7522.80	0.04	48	3187.37	
Ord_4356	Prod_9	SHP_6074	Cust_1481	6831.72	0.01	41	3081.02	
Ord_262	Prod_9	SHP_358	Cust_66	6553.45	0.03	39	2969.81	
Ord_4059	Prod_9	SHP_5660	Cust_1378	5587.20	0.05	36	2254.16	
Ord_2973	Prod_9	SHP_6073	Cust_1480	5410.95	0.09	36	2077.91	
...	...	...	...	...	...	...	...	
Ord_3713	Prod_1	SHP_5145	Cust_1307	27.83	0.09	2	-22.14	
Ord_2746	Prod_1	SHP_3767	Cust_1030	22.61	0.03	1	-8.40	
Ord_439	Prod_1	SHP_587	Cust_136	18.73	0.05	1	-6.68	
Ord_2314	Prod_1	SHP_3171	Cust_899	18.16	0.03	1	-7.25	
Ord_286	Prod_1	SHP_387	Cust_90	18.15	0.04	1	-7.26	

8399 rows × 9 columns

In [48]:

```
market_df.columns
```

Out[48]:

```
Index(['Prod_id', 'Ship_id', 'Cust_id', 'Sales', 'Discount', 'Order_Quanti
ty',
       'Profit', 'Shipping_Cost', 'Product_Base_Margin'],
      dtype='object')
```

In [50]:

```
market_df[['Prod_id','Ship_id']]
```

Out[50]:

Ord_id	Prod_id	Ship_id
Ord_5446	Prod_16	SHP_7609
Ord_5406	Prod_13	SHP_7549
Ord_5446	Prod_4	SHP_7610
Ord_5456	Prod_6	SHP_7625
Ord_5485	Prod_17	SHP_7664
...	...	...
Ord_5353	Prod_4	SHP_7479
Ord_5411	Prod_6	SHP_7555
Ord_5388	Prod_6	SHP_7524
Ord_5348	Prod_15	SHP_7469
Ord_5459	Prod_6	SHP_7628

8399 rows × 2 columns

EXPERIMENT- 3

AIM:

Indexing and Selecting data using Pandas

SOFTWARE USED:

Python 3.7.3, Jupyter Notebook 6.0.1, Pandas 0.25.1

THEORY & SOURCE CODE

INDEXING AND SELECTING DATA

Objectives:

- Select rows from a dataframe
- Select columns from a dataframe
- Select subsets of dataframes

1. Selecting Rows

Selecting rows in dataframes is similar to the indexing you have seen in numpy arrays. The syntax `df[start_index:end_index]` will subset rows according to the start and end indices.

In [3]:

```
import numpy as np
import pandas as pd

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

Out[3]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

By default, pandas assigns integer labels to the rows, starting at 0.

In [2]:

```
# Selecting the rows from indices 2 to 6
market_df[2:7]
```

Out[2]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.02	0.03	23	-47.64	
6	Ord_31	Prod_12	SHP_41	Cust_26	14.76	0.01	5	1.32	

In [4]:

```
# Selecting alternate rows starting from index = 5
market_df[5::2].head()
```

Out[4]:

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

2. Selecting Columns

There are two simple ways to select a single column from a dataframe - `df['column_name']` and `df.column_name` .

In [5]:

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

Out[5]:

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

In [6]:

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

Out[6]:

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

In [7]:

```
# Notice that in both these cases, the resultant is a Series object
print(type(market_df['Sales']))
print(type(market_df.Sales))
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.series.Series'>
```

Selecting Multiple Columns

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

For instance, to select only the columns Cust_id, Sales and Profit:

In [7]:

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

Out[7]:

	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

The output is itself a dataframe.

In [8]:

```
type(market_df[['Cust_id', 'Sales', 'Profit']])
```

Out[8]:

```
pandas.core.frame.DataFrame
```

In [9]:

```
# Similarly, if you select one column using double square brackets,
# you'll get a df, not Series

type(market_df[['Sales']])
```

Out[9]:

```
pandas.core.frame.DataFrame
```

3. Selecting Subsets of Dataframes

The selecting rows and columns has been implemented using the following ways:

- Selecting rows: `df[start:stop]`
- Selecting columns: `df['column']` or `df.column` or `df[['col_x', 'col_y']]`
  - `df['column']` or `df.column` return a series
  - `df[['col_x', 'col_y']]` returns a dataframe

But pandas does not prefer this way of indexing dataframes, since it has some ambiguity.

In [ ]:

```
# Trying to select the third row: Throws an error
market_df[2]
```

Pandas throws an error because it is confused whether the `[2]` is an *index* or a *label*.

In [12]:

```
# Changing the row indices to Ord_id
market_df.set_index('Ord_id').head()
```

Out[12]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping
Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

Now imagine you had a column with entries `[2, 4, 7, 8 ...]`, and you set that as the index. What should `df[2]` return? The second row, or the row with the index value = 2?

Considering an example from this dataset, we will assign the `Order_Quantity` column as the index.

In [13]:

```
market_df.set_index('Order_Quantity').head()
```

Out[13]:

Order_Quantity	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Profit	Shipping
23	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	-30.51	
13	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	4.56	
26	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	1148.90	
43	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	729.34	
35	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	1219.87	

Now, what should `df[13]` return - the 14th row, or the row with index label 13 (i.e. the second row)?

Because of this and similar other ambiguities, pandas provides **explicit ways** to subset dataframes - position based indexing and label based indexing.

EXPERIMENT - 4

AIM:

Position and Label based indexing using Pandas

SOFTWARE USED:

Python 3.9.6, Jupyter Notebook 6.4.6

THEORY AND SOURCE CODE

Position and Label Based Indexing: df.iloc and df.loc

df.loc: loc gets rows (and/or columns) with particular labels. df.iloc: iloc gets rows (and/or columns) at integer locations.

INDEXING DATAFRAMES

They are since they are more explicit (and less ambiguous).

There are two main ways of indexing dataframes:

1. Position based indexing using df.iloc
2. Label based indexing using df.loc

Using both the methods, we will do the following indexing operations on a dataframe:

- Selecting single elements/cells
- Selecting single and multiple rows
- Selecting single and multiple columns
- Selecting multiple rows and columns

In [1]:

```
# 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")
market_df.head()
```

Out[1]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

1. Position (Integer) Based Indexing

Pandas provides the `df.iloc` functionality to index dataframes **using integer indices**.

As mentioned in the documentation, the inputs x, y to `df.iloc[x, y]` can be:

- An integer, e.g. `3`
- A list or array of integers, e.g. `[3, 7, 8]`
- An integer range, i.e. `3:8`
- A boolean array

In [ ]:

```
help(pd.DataFrame.iloc)
```

Note that simply writing `df[2, 4]` will throw an error, since pandas gets confused whether the 2 is an integer index (the third row), or is it a row with label = 2.

On the other hand, `df.iloc[2, 4]` tells pandas explicitly that it should assume **integer indices**.

In [2]:

```
# Selecting a single element
# Here 2, 4 corresponds to the third row and fifth column (Sales)

market_df.iloc[2, 4]
```

Out[2]:

```
4701.69
```

In [3]:

```
# Selecting a single row, and all columns
# Select the 6th row, with label (and index) = 5
market_df.iloc[5]
```

Out[3]:

```
Ord_id                 Ord_5446
Prod_id                  Prod_6
Ship_id                SHP_7608
Cust_id               Cust_1818
Sales                    164.02
Discount                   0.03
Order_Quantity               23
Profit                    -47.64
Shipping_Cost              6.15
Product_Base_Margin        0.37
Name: 5, dtype: object
```

In [4]:

```
# The above is equivalent to this
# The ":" indicates "all rows/columns"
market_df.iloc[5, :]

# equivalent to market_df.iloc[5, ]
```

Out[4]:

```
Ord_id                 Ord_5446
Prod_id                  Prod_6
Ship_id                SHP_7608
Cust_id               Cust_1818
Sales                    164.02
Discount                   0.03
Order_Quantity               23
Profit                    -47.64
Shipping_Cost              6.15
Product_Base_Margin        0.37
Name: 5, dtype: object
```

In [5]:

```
# Select multiple rows using a list of indices
market_df.iloc[[3, 7, 8]]
```

Out[5]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.34	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84	

In [6]:

```
# Equivalently, you can use:
market_df.iloc[[3, 7, 8],:]

# same as market_df.iloc[[3, 7, 8], ]
```

Out[6]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.34	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84	

In [7]:

```
# Selecting rows using a range of integer indices
# Notice that 4 is included, 8 is not
market_df.iloc[4:8]
```

Out[7]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87	
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.0200	0.03	23	-47.64	
6	Ord_31	Prod_12	SHP_41	Cust_26	14.7600	0.01	5	1.32	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	

In [7]:

```
# or equivalently
market_df.iloc[4:8, :]

# or market_df.iloc[4:8, ]
```

Out[7]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87	
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.0200	0.03	23	-47.64	
6	Ord_31	Prod_12	SHP_41	Cust_26	14.7600	0.01	5	1.32	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	

In [9]:

```
# Selecting a single column
# Notice that the column index starts at 0, and 2 represents the third column (Cust_id)
market_df.iloc[:, 2]
```

Out[9]:

```
0       SHP_7609
1       SHP_7549
2       SHP_7610
3       SHP_7625
4       SHP_7664
5       SHP_7608
6         SHP_41
7       SHP_6593
8       SHP_6593
9       SHP_6593
10      SHP_6615
11      SHP_2637
12      SHP_4112
13      SHP_3093
14      SHP_3006
15      SHP_3114
16      SHP_3122
17      SHP_6228
18      SHP_6171
19      SHP_1378
20      SHP_1378
21      SHP_1378
22      SHP_1377
23      SHP_1378
24      SHP_3525
25      SHP_3204
26      SHP_3367
27      SHP_3300
28      SHP_3527
29      SHP_3395
          ...
8369    SHP_5031
8370    SHP_3690
8371    SHP_3591
8372    SHP_3806
8373    SHP_3560
8374    SHP_3637
8375    SHP_3806
8376    SHP_3590
8377    SHP_3729
8378    SHP_3705
8379    SHP_3730
8380    SHP_3807
8381    SHP_3691
8382    SHP_3636
8383    SHP_3731
8384    SHP_6435
8385    SHP_2527
8386    SHP_3189
8387    SHP_3019
8388    SHP_6165
8389    SHP_6192
8390    SHP_7594
```

```
8391    SHP_7594
8392    SHP_7519
8393    SHP_7470
8394    SHP_7479
8395    SHP_7555
8396    SHP_7524
8397    SHP_7469
8398    SHP_7628
Name: Ship_id, Length: 8399, dtype: object
```

In [8]:

```
# Selecting multiple columns
market_df.iloc[:, 3:8]
```

Out[8]:

	Cust_id	Sales	Discount	Order_Quantity	Profit
0	Cust_1818	136.8100	0.01	23	-30.51
1	Cust_1818	42.2700	0.01	13	4.56
2	Cust_1818	4701.6900	0.00	26	1148.90
3	Cust_1818	2337.8900	0.09	43	729.34
4	Cust_1818	4233.1500	0.08	35	1219.87
...	...	...	...	...	...
8394	Cust_1798	2841.4395	0.08	28	374.63
8395	Cust_1798	127.1600	0.10	20	-74.03
8396	Cust_1798	243.0500	0.02	39	-70.85
8397	Cust_1798	3872.8700	0.03	23	565.34
8398	Cust_1798	603.6900	0.00	47	131.39

8399 rows × 5 columns

In [9]:

```
# Selecting multiple rows and columns
market_df.iloc[3:6, 2:5]
```

Out[9]:

	Ship_id	Cust_id	Sales
3	SHP_7625	Cust_1818	2337.89
4	SHP_7664	Cust_1818	4233.15
5	SHP_7608	Cust_1818	164.02

In [13]:

```
# Using booleans
# This selects the rows corresponding to True
market_df.iloc[[True, True, False, True, True, False, True]]
```

Out[13]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	
6	Ord_31	Prod_12	SHP_41	Cust_26	14.76	0.01	5	1.32	

`df.iloc[x, y]` uses integer indices starting at 0.

The other common way of indexing is the **label based** indexing, which uses `df.loc[]` .

2. Label Based Indexing

Pandas provides the `df.loc[]` functionality to index dataframes **using labels**.

The inputs x, y to df.loc[x, y] can be:

- A single label, e.g. `'3'` or `'row_index'`
- A list or array of labels, e.g. `['3', '7', '8']`
- A range of labels, where `row_x` and `row_y` **both are included**, i.e. `'row_x':'row_y'`
- A boolean array

In [14]:

```
help(pd.DataFrame.loc)
```

```
Help on property:

    Purely label-location based indexer for selection by label.

    ``.loc[]`` is primarily label based, but may also be used with a
    boolean array.

    Allowed inputs are:

    - A single label, e.g. ``5`` or ``'a'``, (note that ``5`` is
      interpreted as a *label* of the index, and **never** as an
      integer position along the index).
    - A list or array of labels, e.g. ``['a', 'b', 'c']``.
    - A slice object with labels, e.g. ``'a':'f'`` (note that contrary
      to usual python slices, **both** the start and the stop are include
d!).
    - A boolean array.
    - A ``callable`` function with one argument (the calling Series, DataF
rame
      or Panel) and that returns valid output for indexing (one of the abo
ve)

    ``.loc`` will raise a ``KeyError`` when the items are not found.

    See more at :ref:`Selection by Label <indexing.label>`
```

In [18]:

```
# Selecting a single element
# Select row label = 2 and column label = 'Sales
market_df.loc[2, 'Sales']
```

Out[18]:

```
4701.69
```

In [9]:

```
# Selecting a single row using a single label
# df.loc reads 5 as a label, not index
market_df.loc[0]
```

Out[9]:

```
Ord_id                 Ord_5446
Prod_id                 Prod_16
Ship_id                SHP_7609
Cust_id               Cust_1818
Sales                    136.81
Discount                   0.01
Order_Quantity               23
Profit                    -30.51
Shipping_Cost                3.6
Product_Base_Margin         0.56
Name: 0, dtype: object
```

In [10]:

```
# or equivalently
market_df.loc[5, :]

# or market_df.loc[5, ]
```

Out[10]:

```
Ord_id                 Ord_5446
Prod_id                  Prod_6
Ship_id                SHP_7608
Cust_id               Cust_1818
Sales                    164.02
Discount                   0.03
Order_Quantity               23
Profit                    -47.64
Shipping_Cost                6.15
Product_Base_Margin         0.37
Name: 5, dtype: object
```

In [11]:

```
# Select multiple rows using a list of row labels
market_df.loc[[3, 7, 8]]
```

Out[11]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.34	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84	

In [12]:

```
# Or equivalently
market_df.loc[[3, 7, 8], :]
```

Out[12]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.34	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84	

In [18]:

```
# Selecting rows using a range of labels
# Notice that with df.loc, both 4 and 8 are included, unlike with df.iloc
# This is an important difference between iloc and loc
market_df.loc[4:8]
```

Out[18]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87	
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.0200	0.03	23	-47.64	
6	Ord_31	Prod_12	SHP_41	Cust_26	14.7600	0.01	5	1.32	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84	

In [16]:

```
# Or equivalently
market_df.loc[4:8, ]
```

Out[16]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87	
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.0200	0.03	23	-47.64	
6	Ord_31	Prod_12	SHP_41	Cust_26	14.7600	0.01	5	1.32	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84	

In [19]:

```
# Or equivalently
market_df.loc[4:8, :]
```

Out[19]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shi
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87	
5	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.0200	0.03	23	-47.64	
6	Ord_31	Prod_12	SHP_41	Cust_26	14.7600	0.01	5	1.32	
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91	
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84	

In [20]:

```
# The use of label based indexing will be more clear when we have custom row indices
# Let's change the indices to Ord_id
market_df.set_index('Ord_id', inplace = True)
market_df.head()
```

Out[20]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping
Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

In [22]:

```
# Select Ord_id = Ord_5406 and some columns
market_df.loc['Ord_5406', ['Sales', 'Profit', 'Cust_id']]
```

Out[22]:

```
Sales          42.27
Profit          4.56
Cust_id    Cust_1818
Name: Ord_5406, dtype: object
```

In [23]:

```
# Select multiple orders using labels, and some columns
market_df.loc[['Ord_5406', 'Ord_5446', 'Ord_5485'], 'Sales':'Profit']
```

Out[23]:

Ord_id	Sales	Discount	Order_Quantity	Profit
Ord_5406	42.27	0.01	13	4.56
Ord_5446	136.81	0.01	23	-30.51
Ord_5446	4701.69	0.00	26	1148.90
Ord_5446	164.02	0.03	23	-47.64
Ord_5485	4233.15	0.08	35	1219.87

In [26]:

```
# Using booleans
# This selects the rows corresponding to True
market_df.loc[[True, True, False, True, True, False, True]]
```

Out[26]:

Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping
Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	
Ord_31	Prod_12	SHP_41	Cust_26	14.76	0.01	5	1.32	

EXPERIMENT - 5

AIM:

Slicing and Dicing using Pandas

SOFTWARE USED:

Python 3.9.6, Jupyter Notebook 6.4.6

THEORY AND SOURCE CODE

SLICING AND DICING DATAFRAMES

In [1]:

```
# loading libraries and reading the data
import numpy as np
import pandas as pd
df = pd.read_csv("global_sales_data/market_fact.csv")
df.head()
```

Out[1]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

Subsetting Rows Based on Conditions

Often, you want to select rows which satisfy some given conditions. For e.g., select all the orders where the `Sales > 3000`, or all the orders where `2000 < Sales < 3000` and `Profit < 100`.

Arguably, the best way to do these operations is using `df.loc[]`, since `df.iloc[]` would require you to remember the integer column indices, which is tedious.

In [2]:

```
# Select all rows where Sales > 3000
# First, we get a boolean array where True corresponds to rows having Sales > 3000
df['Sales'] > 3000
```

Out[2]:

```
0       False
1       False
2        True
3       False
4        True
        ...
8394    False
8395    False
8396    False
8397     True
8398    False
Name: Sales, Length: 8399, dtype: bool
```

In [3]:

```
# Then, we pass this boolean array inside df.loc
df.loc[df.Sales > 3000]
```

Out[3]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.6900	0.00	26	1148.90
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91
10	Ord_4743	Prod_2	SHP_6615	Cust_1641	4072.0100	0.01	43	1675.98
13	Ord_2207	Prod_11	SHP_3093	Cust_839	3364.2480	0.10	15	-693.23
...	...	...	...	...	...	...	...	...
8366	Ord_3593	Prod_3	SHP_4974	Cust_1274	12073.0600	0.03	39	5081.87
8367	Ord_3593	Prod_15	SHP_4975	Cust_1274	6685.0500	0.09	25	1653.60
8371	Ord_2624	Prod_4	SHP_3591	Cust_1006	4924.1350	0.07	28	1049.54
8383	Ord_2722	Prod_1	SHP_3731	Cust_1006	3508.3300	0.04	21	-546.98
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.34

1359 rows × 10 columns

In [5]:

```
# An alternative to df.Sales is df['Sales]
# You may want to put the : to indicate that you want all columns
# It is more explicit
df.loc[df['Sales'] > 3000,:]
```

Out[5]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Prof
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.6900	0.00	26	1148.9
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.8
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.9
10	Ord_4743	Prod_2	SHP_6615	Cust_1641	4072.0100	0.01	43	1675.9
13	Ord_2207	Prod_11	SHP_3093	Cust_839	3364.2480	0.10	15	-693.2
17	Ord_4471	Prod_15	SHP_6228	Cust_1521	13255.9300	0.02	25	4089.2
22	Ord_996	Prod_5	SHP_1377	Cust_371	3202.2500	0.09	44	991.2
28	Ord_2573	Prod_4	SHP_3527	Cust_931	3594.7435	0.05	38	1016.9
40	Ord_5035	Prod_15	SHP_7024	Cust_1710	4917.6900	0.02	42	126.3
57	Ord_4546	Prod_1	SHP_6327	Cust_1474	5208.7800	0.05	34	1547.7
64	Ord_4475	Prod_4	SHP_6234	Cust_1474	7640.2250	0.07	46	2027.6
79	Ord_5188	Prod_14	SHP_7252	Cust_1749	5177.4000	0.06	12	287.5
82	Ord_5156	Prod_4	SHP_7207	Cust_1749	8374.1320	0.06	48	2568.1
84	Ord_5232	Prod_17	SHP_7306	Cust_1758	21366.5100	0.00	3	-11984.4
107	Ord_250	Prod_15	SHP_346	Cust_45	8901.7800	0.04	31	2795.3
121	Ord_5423	Prod_17	SHP_7577	Cust_1800	13070.2000	0.07	4	-6923.6
124	Ord_2432	Prod_4	SHP_3339	Cust_933	3883.4715	0.10	42	707.1
133	Ord_2432	Prod_3	SHP_3338	Cust_933	15337.5800	0.10	30	6670.4
148	Ord_1649	Prod_2	SHP_2277	Cust_498	3005.7400	0.02	48	1053.2
163	Ord_4646	Prod_15	SHP_6475	Cust_1603	4083.1900	0.07	43	-1049.8
164	Ord_4646	Prod_1	SHP_6476	Cust_1603	4902.3800	0.05	32	1438.4
165	Ord_4702	Prod_3	SHP_6560	Cust_1603	11823.5200	0.10	34	4592.7
168	Ord_2973	Prod_4	SHP_4102	Cust_1081	4671.1495	0.07	28	947.3
169	Ord_3009	Prod_15	SHP_4168	Cust_1081	5718.8500	0.09	49	-2426.5
170	Ord_2973	Prod_4	SHP_4102	Cust_1480	6264.1855	0.01	34	1312.0
171	Ord_2973	Prod_9	SHP_6073	Cust_1480	5410.9500	0.09	36	2077.9
183	Ord_5157	Prod_15	SHP_7208	Cust_1753	5149.0600	0.03	27	605.4
185	Ord_5153	Prod_11	SHP_7202	Cust_1753	3457.5600	0.02	35	-365.4
189	Ord_2831	Prod_3	SHP_3893	Cust_1036	18092.6600	0.09	36	7917.7
191	Ord_2791	Prod_11	SHP_3838	Cust_1036	10351.0100	0.08	19	-1331.5
...	...	...	...	...	...	...	...	

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Prof
8125	Ord_4515	Prod_1	SHP_6285	Cust_1546	4976.6000	0.06	32	601.8
8137	Ord_2267	Prod_2	SHP_3089	Cust_890	6608.2400	0.09	35	2164.6
8141	Ord_3368	Prod_15	SHP_4668	Cust_1190	3227.3800	0.05	26	192.3
8143	Ord_3265	Prod_1	SHP_4530	Cust_1190	10330.9400	0.07	25	2914.5
8151	Ord_3214	Prod_2	SHP_4456	Cust_1187	8316.7600	0.05	30	2108.8
8158	Ord_2868	Prod_15	SHP_3950	Cust_1059	5405.4400	0.07	18	604.4
8207	Ord_3433	Prod_8	SHP_4759	Cust_1180	3601.0700	0.04	50	174.8
8210	Ord_2110	Prod_17	SHP_2885	Cust_802	4502.2600	0.09	38	1272.1
8211	Ord_2144	Prod_1	SHP_2925	Cust_802	5318.8900	0.06	48	525.1
8217	Ord_3359	Prod_10	SHP_7245	Cust_1762	28389.1400	0.07	33	7132.1
8219	Ord_5200	Prod_4	SHP_7266	Cust_1762	4815.8620	0.08	47	1316.7
8235	Ord_759	Prod_5	SHP_1036	Cust_245	8549.0400	0.03	42	2861.0
8240	Ord_793	Prod_11	SHP_1090	Cust_245	4452.6500	0.09	30	-265.1
8249	Ord_3470	Prod_17	SHP_4805	Cust_1203	5615.4000	0.07	46	1807.1
8263	Ord_4954	Prod_15	SHP_6913	Cust_1694	7987.4300	0.03	49	1304.9
8283	Ord_1741	Prod_11	SHP_2411	Cust_595	15168.8200	0.02	26	-1096.7
8286	Ord_1749	Prod_1	SHP_2424	Cust_595	5176.2700	0.01	38	-743.4
8292	Ord_1765	Prod_14	SHP_2446	Cust_595	14647.2600	0.07	25	5485.1
8298	Ord_2172	Prod_1	SHP_2960	Cust_787	6648.5800	0.08	19	-555.9
8308	Ord_2659	Prod_8	SHP_3638	Cust_1016	4744.6400	0.04	46	1033.3
8317	Ord_2770	Prod_3	SHP_3802	Cust_1016	7535.9600	0.08	46	2745.8
8329	Ord_2160	Prod_15	SHP_2944	Cust_810	3636.3700	0.06	33	-176.9
8332	Ord_2101	Prod_11	SHP_2875	Cust_790	10714.7800	0.00	41	-627.6
8335	Ord_2070	Prod_11	SHP_2833	Cust_784	3905.7500	0.01	21	55.3
8343	Ord_2178	Prod_4	SHP_2968	Cust_785	6030.5800	0.09	39	1197.8
8366	Ord_3593	Prod_3	SHP_4974	Cust_1274	12073.0600	0.03	39	5081.8
8367	Ord_3593	Prod_15	SHP_4975	Cust_1274	6685.0500	0.09	25	1653.6
8371	Ord_2624	Prod_4	SHP_3591	Cust_1006	4924.1350	0.07	28	1049.5
8383	Ord_2722	Prod_1	SHP_3731	Cust_1006	3508.3300	0.04	21	-546.9
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.3

1359 rows × 10 columns

In [4]:

```
# We combine multiple conditions using the & operator
# E.g. all orders having 2000 < Sales < 3000 and Profit > 100
df.loc[(df.Sales > 2000) & (df.Sales < 3000) & (df.Profit > 100), :]
```

Out[4]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	S
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.34	
81	Ord_5205	Prod_4	SHP_7274	Cust_1749	2546.5235	0.09	26	210.00	
109	Ord_139	Prod_17	SHP_186	Cust_45	2671.2100	0.06	14	636.18	
110	Ord_239	Prod_4	SHP_332	Cust_45	2157.3085	0.00	38	519.25	
141	Ord_1673	Prod_17	SHP_2314	Cust_498	2027.5500	0.04	14	537.40	
...	...	...	...	...	...	...	...	...	
8338	Ord_2107	Prod_2	SHP_2882	Cust_785	2409.9600	0.07	32	575.10	
8350	Ord_3570	Prod_4	SHP_4942	Cust_1266	2094.9780	0.06	44	697.29	
8354	Ord_3592	Prod_4	SHP_4973	Cust_1266	2614.3705	0.07	25	384.01	
8381	Ord_2696	Prod_4	SHP_3691	Cust_1006	2836.0505	0.01	25	561.13	
8394	Ord_5353	Prod_4	SHP_7479	Cust_1798	2841.4395	0.08	28	374.63	

328 rows × 10 columns

In [7]:

```
# The 'OR' operator is represented by a | (Note that 'or' doesn't work with pandas)
# E.g. all orders having 2000 < Sales  OR Profit > 100
df.loc[(df.Sales > 2000) | (df.Profit > 100), :]
```

Out[7]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Prof
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.6900	0.00	26	1148.9
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.3
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.8
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.9
10	Ord_4743	Prod_2	SHP_6615	Cust_1641	4072.0100	0.01	43	1675.9
13	Ord_2207	Prod_11	SHP_3093	Cust_839	3364.2480	0.10	15	-693.2
16	Ord_2282	Prod_9	SHP_3122	Cust_839	443.4600	0.06	30	193.1
17	Ord_4471	Prod_15	SHP_6228	Cust_1521	13255.9300	0.02	25	4089.2
22	Ord_996	Prod_5	SHP_1377	Cust_371	3202.2500	0.09	44	991.2
27	Ord_2405	Prod_9	SHP_3300	Cust_931	1062.6900	0.01	28	401.8
28	Ord_2573	Prod_4	SHP_3527	Cust_931	3594.7435	0.05	38	1016.9
37	Ord_5040	Prod_2	SHP_7032	Cust_1710	1637.7800	0.08	38	461.6
38	Ord_5035	Prod_12	SHP_7025	Cust_1710	416.8000	0.01	32	223.3
39	Ord_5045	Prod_5	SHP_7040	Cust_1710	1286.8700	0.05	48	384.3
40	Ord_5035	Prod_15	SHP_7024	Cust_1710	4917.6900	0.02	42	126.3
47	Ord_4659	Prod_6	SHP_6493	Cust_1579	1451.5900	0.06	26	435.1
54	Ord_4682	Prod_4	SHP_6531	Cust_1579	1909.0065	0.01	32	344.1
56	Ord_4410	Prod_5	SHP_6146	Cust_1474	605.7700	0.05	42	129.3
57	Ord_4546	Prod_1	SHP_6327	Cust_1474	5208.7800	0.05	34	1547.7
62	Ord_4410	Prod_6	SHP_6146	Cust_1474	1480.9100	0.00	44	489.1
64	Ord_4475	Prod_4	SHP_6234	Cust_1474	7640.2250	0.07	46	2027.6
70	Ord_4607	Prod_2	SHP_6408	Cust_1474	1463.4200	0.08	26	350.8
77	Ord_5213	Prod_13	SHP_7284	Cust_1749	1633.3700	0.08	50	144.6
79	Ord_5188	Prod_14	SHP_7252	Cust_1749	5177.4000	0.06	12	287.5
81	Ord_5205	Prod_4	SHP_7274	Cust_1749	2546.5235	0.09	26	210.0
82	Ord_5156	Prod_4	SHP_7207	Cust_1749	8374.1320	0.06	48	2568.1
84	Ord_5232	Prod_17	SHP_7306	Cust_1758	21366.5100	0.00	3	-11984.4
88	Ord_5172	Prod_4	SHP_7229	Cust_1758	1239.4445	0.07	22	165.5
98	Ord_5481	Prod_15	SHP_7659	Cust_1820	2700.7800	0.06	29	-793.3
102	Ord_1557	Prod_2	SHP_2152	Cust_565	1736.5300	0.10	46	457.0
...	...	...	...	...	...	...	...	
8328	Ord_2169	Prod_6	SHP_2956	Cust_810	1321.5500	0.10	25	448.2

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Prof
8329	Ord_2160	Prod_15	SHP_2944	Cust_810	3636.3700	0.06	33	-176.9
8331	Ord_2148	Prod_10	SHP_2929	Cust_810	2966.1300	0.08	31	-1456.3
8332	Ord_2101	Prod_11	SHP_2875	Cust_790	10714.7800	0.00	41	-627.6
8334	Ord_2124	Prod_3	SHP_2900	Cust_790	926.8500	0.09	34	382.9
8335	Ord_2070	Prod_11	SHP_2833	Cust_784	3905.7500	0.01	21	55.3
8338	Ord_2107	Prod_2	SHP_2882	Cust_785	2409.9600	0.07	32	575.1
8342	Ord_2119	Prod_2	SHP_2895	Cust_785	1623.0900	0.02	28	245.8
8343	Ord_2178	Prod_4	SHP_2968	Cust_785	6030.5800	0.09	39	1197.8
8344	Ord_2194	Prod_5	SHP_2991	Cust_785	525.7800	0.03	20	180.6
8347	Ord_3534	Prod_4	SHP_4894	Cust_1266	1735.3515	0.08	31	258.6
8350	Ord_3570	Prod_4	SHP_4942	Cust_1266	2094.9780	0.06	44	697.2
8351	Ord_3584	Prod_15	SHP_4961	Cust_1266	2502.6700	0.10	9	-198.8
8354	Ord_3592	Prod_4	SHP_4973	Cust_1266	2614.3705	0.07	25	384.0
8358	Ord_3582	Prod_6	SHP_4959	Cust_1266	539.6600	0.00	26	147.0
8359	Ord_3543	Prod_3	SHP_4906	Cust_1266	514.2200	0.02	17	187.3
8360	Ord_3639	Prod_10	SHP_5038	Cust_1266	2173.2600	0.00	35	-465.6
8362	Ord_3593	Prod_5	SHP_4975	Cust_1274	1175.5300	0.03	18	257.5
8366	Ord_3593	Prod_3	SHP_4974	Cust_1274	12073.0600	0.03	39	5081.8
8367	Ord_3593	Prod_15	SHP_4975	Cust_1274	6685.0500	0.09	25	1653.6
8369	Ord_3633	Prod_3	SHP_5031	Cust_1274	1169.2600	0.02	41	515.6
8371	Ord_2624	Prod_4	SHP_3591	Cust_1006	4924.1350	0.07	28	1049.5
8375	Ord_2772	Prod_3	SHP_3806	Cust_1006	1413.8200	0.10	47	226.5
8378	Ord_2706	Prod_2	SHP_3705	Cust_1006	1361.9100	0.05	20	312.5
8381	Ord_2696	Prod_4	SHP_3691	Cust_1006	2836.0505	0.01	25	561.1
8383	Ord_2722	Prod_1	SHP_3731	Cust_1006	3508.3300	0.04	21	-546.9
8385	Ord_1833	Prod_3	SHP_2527	Cust_637	611.1600	0.04	46	100.2
8394	Ord_5353	Prod_4	SHP_7479	Cust_1798	2841.4395	0.08	28	374.6
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.3
8398	Ord_5459	Prod_6	SHP_7628	Cust_1798	603.6900	0.00	47	131.3

3009 rows × 10 columns

In [5]:

```
# E.g. all orders having 2000 < Sales < 3000 and Profit > 100
# Also, this time, you only need the Cust_id, Sales and Profit columns
df.loc[(df.Sales > 2000) & (df.Sales < 3000) & (df.Profit > 100), ['Cust_id', 'Sales',
```

Out[5]:

	Cust_id	Sales	Profit
3	Cust_1818	2337.8900	729.34
81	Cust_1749	2546.5235	210.00
109	Cust_45	2671.2100	636.18
110	Cust_45	2157.3085	519.25
141	Cust_498	2027.5500	537.40
...	...	...	...
8338	Cust_785	2409.9600	575.10
8350	Cust_1266	2094.9780	697.29
8354	Cust_1266	2614.3705	384.01
8381	Cust_1006	2836.0505	561.13
8394	Cust_1798	2841.4395	374.63

328 rows × 3 columns

In [7]:

```
# You can use the == and != operators
df.loc[(df.Sales == 4233.15), :]
df.loc[(df.Sales != 1000), :]
```

Out[7]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.8100	0.01	23	-30.51
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.2700	0.01	13	4.56
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.6900	0.00	26	1148.90
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.8900	0.09	43	729.34
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87
...	...	...	...	...	...	...	...	...
8394	Ord_5353	Prod_4	SHP_7479	Cust_1798	2841.4395	0.08	28	374.63
8395	Ord_5411	Prod_6	SHP_7555	Cust_1798	127.1600	0.10	20	-74.03
8396	Ord_5388	Prod_6	SHP_7524	Cust_1798	243.0500	0.02	39	-70.85
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.34
8398	Ord_5459	Prod_6	SHP_7628	Cust_1798	603.6900	0.00	47	131.39

8399 rows × 10 columns

In [8]:

```
# You may want to select rows whose column value is in an iterable
# For instance, say a colleague gives you a list of customer_ids from a certain region

customers_in_bangalore = ['Cust_1798', 'Cust_1519', 'Cust_637', 'Cust_851']

# To get all the orders from these customers, use the isin() function
# It returns a boolean, which you can use to select rows
df.loc[df['Cust_id'].isin(customers_in_bangalore), :]
```

Out[8]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit
8385	Ord_1833	Prod_3	SHP_2527	Cust_637	611.1600	0.04	46	100.22
8386	Ord_2324	Prod_7	SHP_3189	Cust_851	121.8700	0.07	39	11.32
8387	Ord_2220	Prod_3	SHP_3019	Cust_851	41.0600	0.04	4	-16.39
8388	Ord_4424	Prod_1	SHP_6165	Cust_1519	994.0400	0.03	10	-335.06
8389	Ord_4444	Prod_13	SHP_6192	Cust_1519	159.4100	0.00	44	34.68
8390	Ord_5435	Prod_16	SHP_7594	Cust_1798	316.9900	0.04	47	-276.54
8391	Ord_5435	Prod_4	SHP_7594	Cust_1798	1991.8985	0.07	20	88.36
8392	Ord_5384	Prod_9	SHP_7519	Cust_1798	181.5000	0.08	43	-6.24
8393	Ord_5348	Prod_8	SHP_7470	Cust_1798	356.7200	0.07	9	12.61
8394	Ord_5353	Prod_4	SHP_7479	Cust_1798	2841.4395	0.08	28	374.63
8395	Ord_5411	Prod_6	SHP_7555	Cust_1798	127.1600	0.10	20	-74.03
8396	Ord_5388	Prod_6	SHP_7524	Cust_1798	243.0500	0.02	39	-70.85
8397	Ord_5348	Prod_15	SHP_7469	Cust_1798	3872.8700	0.03	23	565.34
8398	Ord_5459	Prod_6	SHP_7628	Cust_1798	603.6900	0.00	47	131.39

EXPERIMENT - 6

AIM:

Merging and concatenating Data Frames using Pandas

SOFTWARE USED:

Python 3.9.6, Jupyter Notebook 6.4.6

THEORY AND SOURCE CODE

MERGE AND CONCAT DATAFRAMES

Merging is one of the most common operations you will do, since data often comes in various files.

Using the sales data of a retail store spread across multiple files, we carry out:

- Merge multiple dataframes using common columns/keys using `pd.merge()`
- Concatenate dataframes using `pd.concat()`

In [2]:

```
# 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")
```

1. Merging Dataframes Using `pd.merge()`

There are five data files:

1. The `market_fact` table contains the sales data of each order
2. The other 4 files are called 'dimension tables/files' and contain metadata about customers, products, shipping details, order details etc.

If you are familiar with star schemas and data warehouse designs, you will note that we have one fact table and four dimension tables.

In [3]:

```
# Already familiar with market data: Each row is an order
market_df.head()
```

Out[3]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

In [4]:

```
# Customer dimension table: Each row contains metadata about customers
customer_df.head()
```

Out[4]:

	Customer_Name	Province	Region	Customer_Segment	Cust_id
0	MUHAMMED MACINTYRE	NUNAVUT	NUNAVUT	SMALL BUSINESS	Cust_1
1	BARRY FRENCH	NUNAVUT	NUNAVUT	CONSUMER	Cust_2
2	CLAY ROZENDAL	NUNAVUT	NUNAVUT	CORPORATE	Cust_3
3	CARLOS SOLTERO	NUNAVUT	NUNAVUT	CONSUMER	Cust_4
4	CARL JACKSON	NUNAVUT	NUNAVUT	CORPORATE	Cust_5

In [6]:

```
# Product dimension table
product_df.head()
```

Out[6]:

	Product_Category	Product_Sub_Category	Prod_id
0	OFFICE SUPPLIES	STORAGE & ORGANIZATION	Prod_1
1	OFFICE SUPPLIES	APPLIANCES	Prod_2
2	OFFICE SUPPLIES	BINDERS AND BINDER ACCESSORIES	Prod_3
3	TECHNOLOGY	TELEPHONES AND COMMUNICATION	Prod_4
4	FURNITURE	OFFICE FURNISHINGS	Prod_5

In [5]:

```
# Shipping metadata
shipping_df.head()
```

Out[5]:

	Order_ID	Ship_Mode	Ship_Date	Ship_id
0	3	REGULAR AIR	20-10-2010	SHP_1
1	293	DELIVERY TRUCK	02-10-2012	SHP_2
2	293	REGULAR AIR	03-10-2012	SHP_3
3	483	REGULAR AIR	12-07-2011	SHP_4
4	515	REGULAR AIR	30-08-2010	SHP_5

In [8]:

```
# Orders dimension table
orders_df.head()
```

Out[8]:

	Order_ID	Order_Date	Order_Priority	Ord_id
0	3	13-10-2010	LOW	Ord_1
1	293	01-10-2012	HIGH	Ord_2
2	483	10-07-2011	HIGH	Ord_3
3	515	28-08-2010	NOT SPECIFIED	Ord_4
4	613	17-06-2011	HIGH	Ord_5

In [6]:

```
# 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 t
df_1 = pd.merge(market_df, customer_df, how='inner', on='Cust_id')
df_1.head()
```

Out[6]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

In [7]:

```
df1= pd.merge(market_df, customer_df, how='inner', on='Cust_id')
df1.head()
```

Out[7]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5406	Prod_13	SHP_7549	Cust_1818	42.27	0.01	13	4.56	
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
3	Ord_5456	Prod_6	SHP_7625	Cust_1818	2337.89	0.09	43	729.34	
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.15	0.08	35	1219.87	

In [8]:

```
# Now, you can subset the orders made by customers from 'Corporate' segment
df_1.loc[df_1['Customer_Segment'] == 'HOME OFFICE', :]
```

Out[8]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91
8	Ord_4725	Prod_13	SHP_6593	Cust_1641	162.0000	0.01	33	45.84
9	Ord_4725	Prod_6	SHP_6593	Cust_1641	57.2200	0.07	8	-27.72
10	Ord_4743	Prod_2	SHP_6615	Cust_1641	4072.0100	0.01	43	1675.98
11	Ord_1925	Prod_6	SHP_2637	Cust_708	465.9000	0.05	38	79.34
...	...	...	...	...	...	...	...	...
8365	Ord_3588	Prod_3	SHP_4968	Cust_1274	107.6300	0.00	15	-29.89
8366	Ord_3593	Prod_3	SHP_4974	Cust_1274	12073.0600	0.03	39	5081.87
8367	Ord_3593	Prod_15	SHP_4975	Cust_1274	6685.0500	0.09	25	1653.60
8368	Ord_3574	Prod_6	SHP_4947	Cust_1274	438.6600	0.01	41	58.86
8369	Ord_3633	Prod_3	SHP_5031	Cust_1274	1169.2600	0.02	41	515.62

2032 rows × 14 columns

In [10]:

```
df_1.loc[(df_1['Customer_Segment']=='HOME OFFICE') &(df_1['Profit']>500),:]
df_1.loc[(df_1['Profit']> 500) & (df_1['Customer_Name']=='ALEX AVILA'),:]
```

Out[10]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shippi
237	Ord_597	Prod_4	SHP_816	Cust_209	4283.2350	0.10	44	676.13	
239	Ord_597	Prod_4	SHP_816	Cust_209	4374.6865	0.05	43	973.16	

In [11]:

```
df_1.loc[(df_1['Sales'] >= 300) & (df_1['Profit']>1000),:]
```

Out[11]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit
2	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.6900	0.00	26	1148.90
4	Ord_5485	Prod_17	SHP_7664	Cust_1818	4233.1500	0.08	35	1219.87
7	Ord_4725	Prod_4	SHP_6593	Cust_1641	3410.1575	0.10	48	1137.91
10	Ord_4743	Prod_2	SHP_6615	Cust_1641	4072.0100	0.01	43	1675.98
17	Ord_4471	Prod_15	SHP_6228	Cust_1521	13255.9300	0.02	25	4089.27
...	...	...	...	...	...	...	...	...
8317	Ord_2770	Prod_3	SHP_3802	Cust_1016	7535.9600	0.08	46	2745.87
8343	Ord_2178	Prod_4	SHP_2968	Cust_785	6030.5800	0.09	39	1197.86
8366	Ord_3593	Prod_3	SHP_4974	Cust_1274	12073.0600	0.03	39	5081.87
8367	Ord_3593	Prod_15	SHP_4975	Cust_1274	6685.0500	0.09	25	1653.60
8371	Ord_2624	Prod_4	SHP_3591	Cust_1006	4924.1350	0.07	28	1049.54

678 rows × 14 columns

In [12]:

```
# Example 2: Select all orders from product category = office supplies and from the corp
# We now need to merge the product_df

df_2 = pd.merge(df_1, product_df, how='inner', on='Prod_id')
df_2.head()
```

Out[12]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipping
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_2978	Prod_16	SHP_4112	Cust_1088	305.05	0.04	27	23.12	
2	Ord_5484	Prod_16	SHP_7663	Cust_1820	322.82	0.05	35	-17.58	
3	Ord_3730	Prod_16	SHP_5175	Cust_1314	459.08	0.04	34	61.57	
4	Ord_4143	Prod_16	SHP_5771	Cust_1417	207.21	0.06	24	-78.64	

In [13]:

```
# Select all orders from product category = office supplies and from the corporate segme
df_2.loc[(df_2['Product_Category']=='OFFICE SUPPLIES') & (df_2['Customer_Segment']=='COR
```

Out[13]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Sh
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
3	Ord_3730	Prod_16	SHP_5175	Cust_1314	459.08	0.04	34	61.57	
7	Ord_4506	Prod_16	SHP_6273	Cust_1544	92.02	0.07	9	-24.88	
9	Ord_1551	Prod_16	SHP_2145	Cust_531	184.77	0.00	29	-71.96	
11	Ord_1429	Prod_16	SHP_1976	Cust_510	539.06	0.05	42	-123.07	
...	...	...	...	...	...	...	...	...	
7545	Ord_4629	Prod_1	SHP_6447	Cust_1587	848.19	0.06	25	120.02	
7546	Ord_4604	Prod_1	SHP_6403	Cust_1522	234.24	0.09	24	-151.80	
7551	Ord_3543	Prod_1	SHP_4905	Cust_1266	1184.11	0.07	6	-145.07	
7552	Ord_2722	Prod_1	SHP_3731	Cust_1006	3508.33	0.04	21	-546.98	
7553	Ord_4424	Prod_1	SHP_6165	Cust_1519	994.04	0.03	10	-335.06	

1680 rows × 16 columns

Merging Dimension Tables - `shipping_df` and `orders_df` to create a `master_df` and perform indexing using any column in the master dataframe.

In [14]:

```
# Merging shipping_df
df_3 = pd.merge(df_2, shipping_df, how='inner', on='Ship_id')
df_3.shape
```

Out[14]:

```
(8399, 19)
```

In [15]:

```
# Merging the orders table to create a master df
master_df = pd.merge(df_3, orders_df, how='inner', on='Ord_id')
master_df.shape
master_df.head()
```

Out[15]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit	Shipp
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51	
1	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90	
2	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.02	0.03	23	-47.64	
3	Ord_2978	Prod_16	SHP_4112	Cust_1088	305.05	0.04	27	23.12	
4	Ord_5484	Prod_16	SHP_7663	Cust_1820	322.82	0.05	35	-17.58	

5 rows × 22 columns

One can use joins with the argument `how = 'left' / 'right' / 'outer'` to merge dataframe.

2. Concatenating Dataframes

Concatenation is much more straightforward than merging. It is used when you have dataframes having the same columns and want to append them, or having the same rows and want to append them side-by-side.

2.1 Concatenating Dataframes Having the Same columns

In [16]:

```
# 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']}
                   )
df1
```

Out[16]:

	Name	Age	Gender
0	Aman	34	M
1	Joy	31	M
2	Rashmi	22	F
3	Saif	33	M

In [17]:

```
df2
```

Out[17]:

	Name	Age	Gender
0	Akhil	31	M
1	Asha	22	F
2	Preeti	23	F

In [18]:

```
# 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)
```

Out[18]:

	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

In [19]:

```
# A useful and intuitive alternative to concat along the rows is the append() function
# It concatenates along the rows
df1.append(df2)
```

```
C:\Users\Admin\AppData\Local\Temp\ipykernel_15036\132225679.py:3: FutureWa
rning: The frame.append method is deprecated and will be removed from pand
as in a future version. Use pandas.concat instead.
  df1.append(df2)
```

Out[19]:

	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

2.2 Concatenating Dataframes Having the Same Rows

For dataframes having values same for rows we concat them side by side

In [23]:

```
df1 = pd.DataFrame({'Name': ['Aman', 'Joy', 'Rashmi', 'Saif'],
                    'Age': ['34', '31', '22', '33'],
                    'Gender': ['M', 'M', 'F', 'M']}
                   )
df1
```

Out[23]:

	Name	Age	Gender
0	Aman	34	M
1	Joy	31	M
2	Rashmi	22	F
3	Saif	33	M

In [22]:

```
df2 = pd.DataFrame({'School': ['RK Public', 'JSP', 'Carmel Convent', 'St. Paul'],
                    'Graduation Marks': ['84', '89', '76', '91']}
                   )
df2
```

Out[22]:

	School	Graduation Marks
0	RK Public	84
1	JSP	89
2	Carmel Convent	76
3	St. Paul	91

In [24]:

```
# 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)
```

Out[24]:

	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

`pd.concat()` method can also be used to merge dataframes using common keys. But, we have used the `pd.merge()` method for database-style merging and `pd.concat()` for appending dataframes having no common columns.

3. Performing Arithmetic Operations on two or more dataframes

We can also perform simple arithmetic operations on two or more dataframes. Below are the stats for IPL 2018 and 2017.

In [25]:

```
# Teamwise stats for IPL 2018
IPL_2018 = pd.DataFrame({'IPL Team': ['CSK', 'SRH', 'KKR', 'RR', 'MI', 'RCB', 'KXIP', 'D
                         '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 r
IPL_2018.set_index('IPL Team', inplace = True)
IPL_2018
```

Out[25]:

IPL Team	Matches Played	Matches Won
CSK	16	11
SRH	17	10
KKR	16	9
RR	15	7
MI	14	6
RCB	14	6
KXIP	14	6
DD	14	5

In [26]:

```
# Similarly, we have the stats for IPL 2017
IPL_2017 = pd.DataFrame({'IPL Team': ['MI', 'RPS', 'KKR', 'SRH', 'KXIP', 'DD', 'GL', 'RC
                         '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)
IPL_2017
```

Out[26]:

IPL Team	Matches Played	Matches Won
MI	17	12
RPS	16	10
KKR	16	9
SRH	15	8
KXIP	14	7
DD	14	6
GL	14	4
RCB	14	3

In [27]:

```
# Simply add the two DFs using the add opearator

Total = IPL_2018 + IPL_2017
Total
```

Out[27]:

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

On adding the two dataframes, the resultant dataframes have many NAN values. This is mainly because the absence of these teams in IPL2018 but presence in IPL 2017. This can be handled by using `df.add()`

instead of the simple add operator.

In [28]:

```
# The fill_value argument inside the df.add() function replaces all the NaN values in th
Total = IPL_2018.add(IPL_2017, fill_value = 0)
Total
```

Out[28]:

IPL Team	Matches Played	Matches Won
CSK	16.0	11.0
DD	28.0	11.0
GL	14.0	4.0
KKR	32.0	18.0
KXIP	28.0	13.0
MI	31.0	18.0
RCB	28.0	9.0
RPS	16.0	10.0
RR	15.0	7.0
SRH	32.0	18.0

Resultant dataframe is sorted by the index, i.e. in alphabetic order.

In [29]:

```
# Creating a new column - 'Win Percentage'

Total['Win Percentage'] = Total['Matches Won']/Total['Matches Played']
Total
```

Out[29]:

IPL Team	Matches Played	Matches Won	Win Percentage
CSK	16.0	11.0	0.687500
DD	28.0	11.0	0.392857
GL	14.0	4.0	0.285714
KKR	32.0	18.0	0.562500
KXIP	28.0	13.0	0.464286
MI	31.0	18.0	0.580645
RCB	28.0	9.0	0.321429
RPS	16.0	10.0	0.625000
RR	15.0	7.0	0.466667
SRH	32.0	18.0	0.562500

In [30]:

```
# Sorting to determine the teams with most number of wins. If the number of wins of two

Total.sort_values(by = (['Matches Won', 'Win Percentage']), ascending = False)
```

Out[30]:

IPL Team	Matches Played	Matches Won	Win Percentage
MI	31.0	18.0	0.580645
KKR	32.0	18.0	0.562500
SRH	32.0	18.0	0.562500
KXIP	28.0	13.0	0.464286
CSK	16.0	11.0	0.687500
DD	28.0	11.0	0.392857
RPS	16.0	10.0	0.625000
RCB	28.0	9.0	0.321429
RR	15.0	7.0	0.466667
GL	14.0	4.0	0.285714

A list of all the mathematical functions that you can use to perform operations on two or more dataframes.

- add() : +
- sub() : -
- mul() : *
- div() : /
- floordiv() : //
- mod() : %
- pow() : **

EXPERIMENT - 7

AIM:

Grouping and summarizing DataFrames using Pandas.

SOFTWARE USED:

Python 3.9.6, Jupyter Notebook 6.4.6

THEORY AND SOURCE CODE

GROUPING AND AGGREGATION

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

It is mainly used for comparing the averages or the features of a dataset i.e. the columns of a dataframe.

Grouping analysis can be thought of as having three parts:

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

In [3]:

```
# 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")
```

To identify areas of business where the heavy losses needs to be calculated. To take steps, one should answer the following questions:

- Which customer segments are the least profitable?
- Which product categories and sub-categories are the least profitable?
- Customers in which geographic region cause the most losses?
- Etc.

In [5]:

```
# Merging the dataframes one by one into a single resultant dataframe 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
```

Out[5]:

	Ord_id	Prod_id	Ship_id	Cust_id	Sales	Discount	Order_Quantity	Profit
0	Ord_5446	Prod_16	SHP_7609	Cust_1818	136.81	0.01	23	-30.51
1	Ord_5446	Prod_4	SHP_7610	Cust_1818	4701.69	0.00	26	1148.90
2	Ord_5446	Prod_6	SHP_7608	Cust_1818	164.02	0.03	23	-47.64
3	Ord_2978	Prod_16	SHP_4112	Cust_1088	305.05	0.04	27	23.12
4	Ord_5484	Prod_16	SHP_7663	Cust_1820	322.82	0.05	35	-17.58
...	...	...	...	...	...	...	...	...
8394	Ord_5018	Prod_14	SHP_7001	Cust_1696	7325.63	0.04	38	1899.23
8395	Ord_669	Prod_14	SHP_916	Cust_224	20872.16	0.03	29	-4437.91
8396	Ord_508	Prod_14	SHP_686	Cust_186	19109.61	0.10	40	-379.29
8397	Ord_3721	Prod_14	SHP_5162	Cust_1309	614.14	0.09	3	-735.27
8398	Ord_825	Prod_14	SHP_1132	Cust_247	27663.92	0.05	8	-391.92

8399 rows × 22 columns

1. Grouping using `df.groupby()`

Grouping with respect to the categorical data to get as many subsets as possible of the categorical data

In [4]:

```
# Which customer segments are the least profitable?
# Grouping the dataframe by customer segments
df_by_segment = master_df.groupby('Customer_Segment')
df_by_segment
#It returns a DataFrameGroupBy object
```

Out[4]:

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000194E77FDBA8
>
```

2. Applying a Function

After grouping, you apply a function to a **numeric variable**, such as `mean(Sales)`, `sum(Profit)`, etc.

In [4]:

```
# Step 2. Applying a function
# We can choose aggregate functions such as sum, mean, median, etc.
df_by_segment['Profit'].sum()
```

Out[4]:

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

So this tells us that profits are the least in the CONSUMER segment, and highest in the CORPORATE segment.

In [6]:

```
# For better readability, you may want to sort the summarised series:
df_by_segment.Profit.sum().sort_values(ascending = False)
```

Out[6]:

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

3. Combining the results into a Data Structure

In [7]:

```
# Converting to a df
pd.DataFrame(df_by_segment['Profit'].sum())
```

Out[7]:

Customer_Segment	Profit
CONSUMER	287959.94
CORPORATE	599746.00
HOME OFFICE	318354.03
SMALL BUSINESS	315708.01

In [8]:

```
# 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')
```

In [9]:

```
# 2. This time, let's compare average profits
# Apply mean() on Profit
by_product_cat['Profit'].mean()
```

Out[9]:

```
Product_Category
FURNITURE           68.116607
OFFICE SUPPLIES    112.369074
TECHNOLOGY         429.207516
Name: Profit, dtype: float64
```

FURNITURE is the least profitable, TECHNOLOGY the most. Let's see which product sub-cetgories within FURNITURE are less profitable.

In [10]:

```
# 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()
```

Out[10]:

```
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
```

Thus, within FURNITURE, TABLES are the least profitable, followed by BOOKCASES.

In [11]:

```
# Recall the df.describe() method?
# To apply multiple functions simultaneously, you can use the describe() function on the
by_product_cat['Profit'].describe()
```

Out[11]:

	count	mean	std	min	25%	50%	75%	
Product_Category								
FURNITURE	1724.0	68.116607	1112.923257	-11053.60	-281.3550	-14.250	187.1600	8
OFFICE SUPPLIES	4610.0	112.369074	744.617939	-2175.09	-57.0225	-3.845	56.9475	11
TECHNOLOGY	2065.0	429.207516	1863.208375	-14140.70	-88.9400	66.220	561.1300	27

In [12]:

```
# Some other summary functions to apply on groups
by_product_cat['Profit'].count()
```

Out[12]:

```
Product_Category
FURNITURE          1724
OFFICE SUPPLIES    4610
TECHNOLOGY         2065
Name: Profit, dtype: int64
```

In [13]:

```
by_product_cat['Profit'].min()
```

Out[13]:

```
Product_Category
FURNITURE         -11053.60
OFFICE SUPPLIES    -2175.09
TECHNOLOGY        -14140.70
Name: Profit, dtype: float64
```

In [14]:

```
# To identify the least profitable customers are present in  which geographical area?
master_df.groupby('Region').Profit.mean()
```

Out[14]:

```
Region
ATLANTIC                 221.259870
NORTHWEST TERRITORIES    255.464670
NUNAVUT                   35.963418
ONTARIO                  189.960865
PRARIE                   188.253294
QUEBEC                   179.803649
WEST                     149.175595
YUKON                    136.253155
Name: Profit, dtype: float64
```

In [15]:

```
# Note that the resulting object is a Series, thus you can perform vectorised computatio

# E.g. Calculate the Sales across each region as a percentage of total Sales
# You can divide the entire series by a number (total sales) easily
(master_df.groupby('Region').Sales.sum() / sum(master_df['Sales']))*100
```

Out[15]:

```
Region
ATLANTIC                 13.504305
NORTHWEST TERRITORIES     5.369193
NUNAVUT                   0.780233
ONTARIO                  20.536970
PRARIE                   19.022396
QUEBEC                   10.124936
WEST                     24.119372
YUKON                     6.542595
Name: Sales, dtype: float64
```

The regions which comprise of about 64% of the sales -

- ONTARIO
- WEST
- PRARIE

EXPERIMENT- 8

AIM:

Implementation of Simple Linear Regression

RESOURCES USED:

Python 3.10.5, Jupyter Notebook

THEORY AND SOURCE CODE

Linear Regression is a machine learning algorithm based on supervised learning. It performs a regression task. Regression models a target prediction value based on independent variables. It is mostly used for finding out the relationship between variables and forecasting. Different regression models differ based on – the kind of relationship between dependent and independent variables they are considering, and the number of independent variables getting used.

1. Importing Libraries

In [ ]:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Importing CSV

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

In [ ]:

```
data=pd.read_csv('/content/advertising.csv')
```

3. Visualizing CSV

In [ ]:

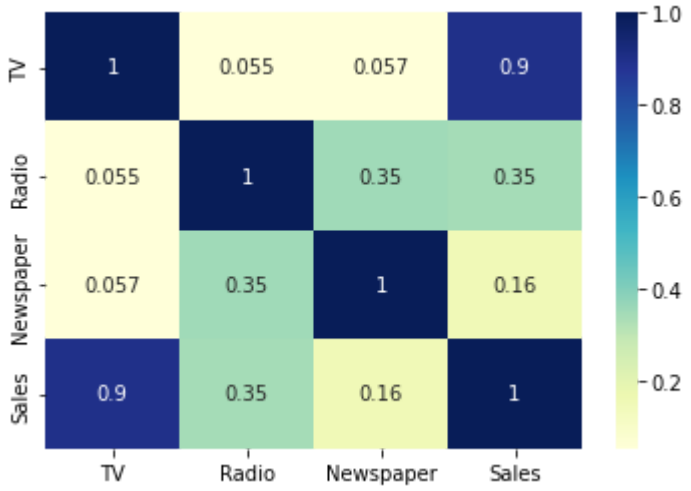
```
data.corr()
```

Out[21]:

	TV	Radio	Newspaper	Sales
TV	1.000000	0.054809	0.056648	0.901208
Radio	0.054809	1.000000	0.354104	0.349631
Newspaper	0.056648	0.354104	1.000000	0.157960
Sales	0.901208	0.349631	0.157960	1.000000

In [ ]:

```
sns.heatmap(data.corr(), cmap="YlGnBu", annot = True)
plt.show()
```

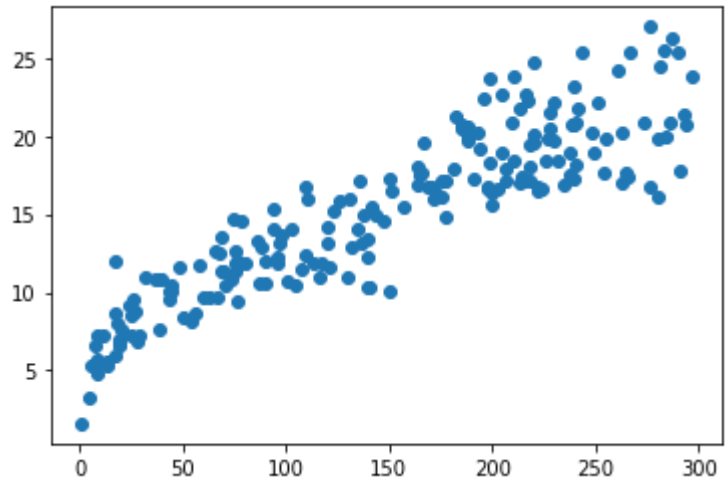


In [ ]:

```
plt.scatter(data.TV,data.Sales)
```

Out[25]:

```
<matplotlib.collections.PathCollection at 0x7f131f23d750>
```

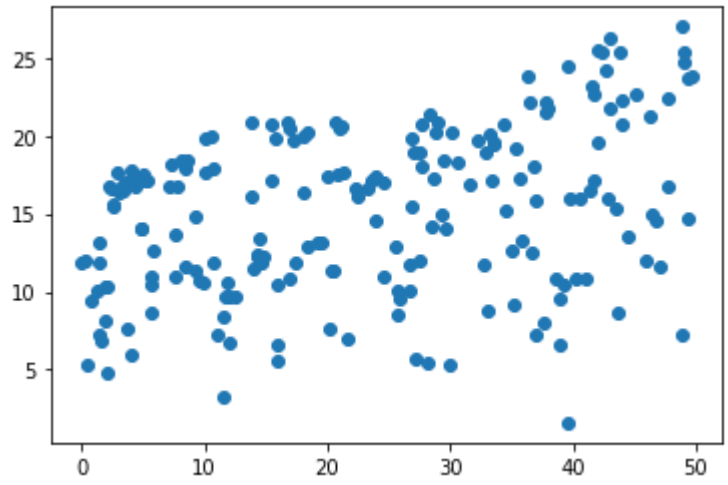


In [ ]:

```
plt.scatter(data.Radio,data.Sales)
```

Out[26]:

```
<matplotlib.collections.PathCollection at 0x7f131f12ed10>
```

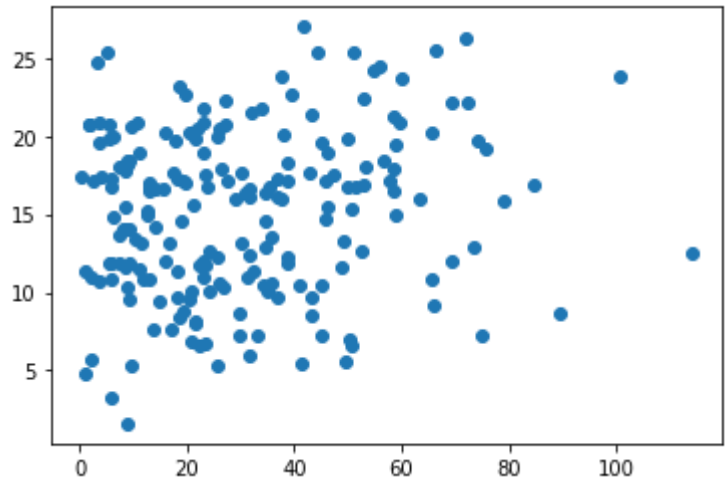


In [ ]:

```
plt.scatter(data.Newspaper,data.Sales)
```

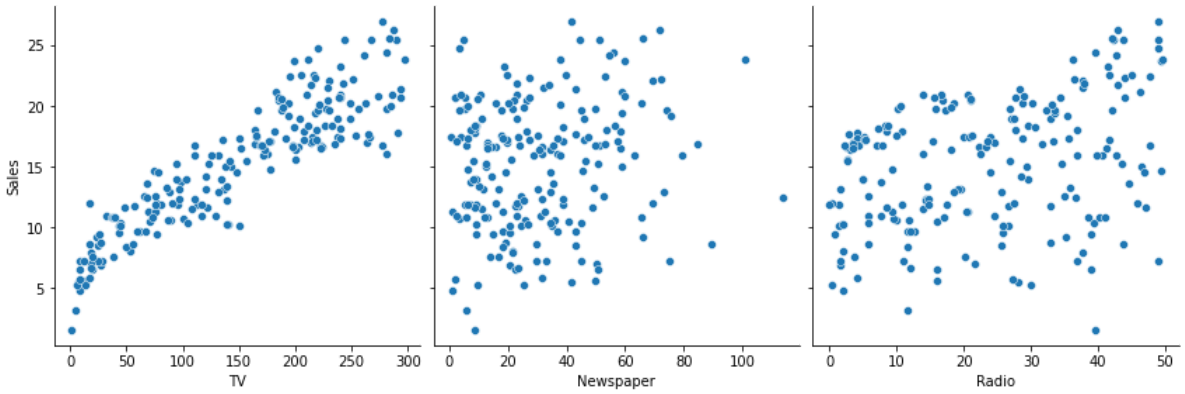
Out[27]:

```
<matplotlib.collections.PathCollection at 0x7f131f047450>
```



In [ ]:

```
sns.pairplot(data, x_vars=['TV', 'Newspaper', 'Radio'], y_vars='Sales',height=4, aspect=
plt.show()
```



4. Selecting Regression variables

In [ ]:

```
y=data.loc[:,['Sales']]
X=data.loc[:,['TV']]
```

In [ ]:

```
X.head()
```

Out[45]:

	TV
0	230.1
1	44.5
2	17.2
3	151.5
4	180.8

In [ ]:

```
y.head()
```

Out[46]:

	Sales
0	22.1
1	10.4
2	12.0
3	16.5
4	17.9

4. Splitting Dataset(70-30)

In [ ]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.7, test_size =
```

In [ ]:

```
X_train.head()
```

Out[48]:

	TV
74	213.4
3	151.5
185	205.0
26	142.9
90	134.3

In [ ]:

```
X_test.head()
```

Out[49]:

	TV
126	7.8
104	238.2
99	135.2
92	217.7
111	241.7

In [ ]:

```
y_train.head()
```

Out[50]:

	Sales
74	17.0
3	16.5
185	22.6
26	15.0
90	14.0

In [ ]:

```
y_test.head()
```

Out[51]:

	Sales
126	6.6
104	20.7
99	17.2
92	19.4
111	21.8

5. Performing statistical analysis and making prediction

In [ ]:

```
import statsmodels.api as sm
# Add a constant to get an intercept
X_train_sm = sm.add_constant(X_train)
# Fit the resgression line using 'OLS'
lr = sm.OLS(y_train, X_train_sm).fit()
```

In [ ]:

```
print(lr.params)
```

```
const    6.948683

TV       0.054546

dtype: float64
```

In [ ]:

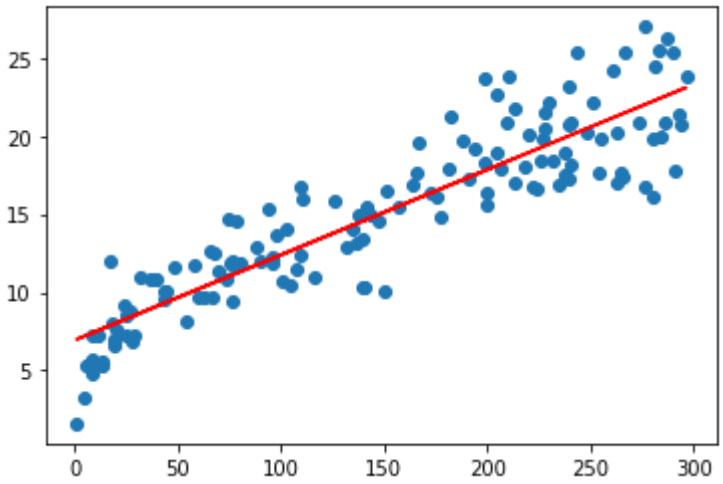
```
print(lr.summary())
```

```
                            OLS Regression Results

==============================================================================
=======

Dep. Variable:                  Sales   R-squared:
0.816

Model:                            OLS   Adj. R-squared:
0.814

Method:                 Least Squares   F-statistic:
611.2

Date:                Thu, 29 Sep 2022   Prob (F-statistic):
1.52e-52

Time:                        10:58:20   Log-Likelihood:
-321.12

No. Observations:                 140   AIC:
646.2

Df Residuals:                     138   BIC:
652.1

Df Model:                           1

Covariance Type:            nonrobust

==============================================================================
=======

                 coef    std err          t      P>|t|      [0.025
0.975]

------------------------------------------------------------------------------
-------

const          6.9487      0.385     18.068      0.000       6.188
7.709

TV             0.0545      0.002     24.722      0.000       0.050
0.059

==============================================================================
=======

Omnibus:                        0.027   Durbin-Watson:
2.196

Prob(Omnibus):                  0.987   Jarque-Bera (JB):
0.150

Skew:                          -0.006   Prob(JB):
0.928
```

```
Kurtosis:                       2.840   Cond. No.
328.

==============================================================================
=======



Notes:

[1] Standard Errors assume that the covariance matrix of the errors is
correctly specified.
```

In [ ]:

```
plt.scatter(X_train, y_train)
plt.plot(X_train, 6.9487 + 0.0545*X_train, 'r')
plt.show()
```



In [ ]:

```
input1=int(input("Enter the variable: "))
```

```
Enter the variable: 55
```

In [ ]:

```
sales_pred=6.9487 + 0.0545*50
```

In [ ]:

```
print(sales_pred)
```

```
16.4862
```

EXPERIMENT- 9

AIM:

Implementation of Multiple Linear Regression (Housing case study)

RESOURCES USED:

Python 3.9.6, Jupyter Notebook 6.4.6

THEORY AND SOURCE CODE

In [1]:

```
#importing libraries

import pandas as pd
import numpy as np
import seaborn as sns
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt;
import random
```

In [2]:

```
# readind the csv file into a DataFrame
df=pd.read_csv("Housing.csv")
```

EXPLORATORY DATA ANALYSIS

In [3]:

```
df.head()
```

Out[3]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterhe
0	13300000	7420	4	2	3	yes	no	no	
1	12250000	8960	4	4	4	yes	no	no	
2	12250000	9960	3	2	2	yes	no	yes	
3	12215000	7500	4	2	2	yes	no	yes	
4	11410000	7420	4	1	2	yes	yes	yes	

In [4]:

```
df.shape
```

Out[4]:

```
(545, 13)
```

In [5]:

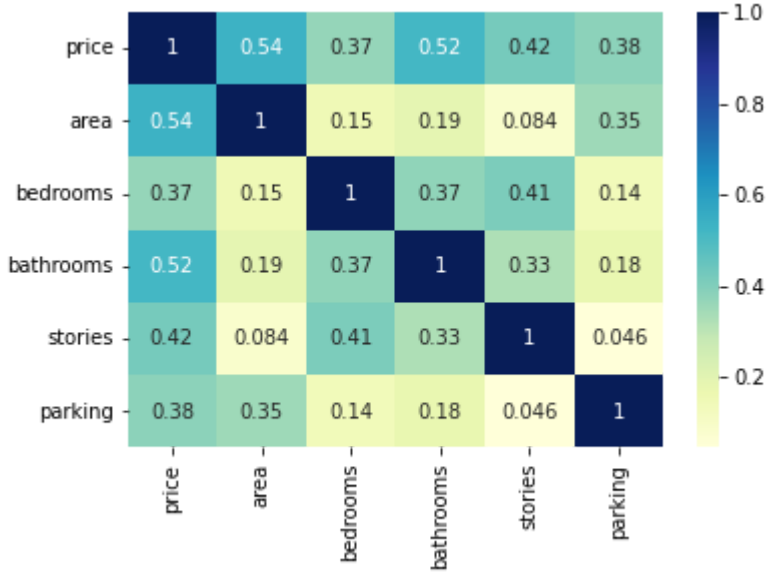
```
df.describe()
```

Out[5]:

	price	area	bedrooms	bathrooms	stories	parking
count	5.450000e+02	545.000000	545.000000	545.000000	545.000000	545.000000
mean	4.766729e+06	5150.541284	2.965138	1.286239	1.805505	0.693578
std	1.870440e+06	2170.141023	0.738064	0.502470	0.867492	0.861586
min	1.750000e+06	1650.000000	1.000000	1.000000	1.000000	0.000000
25%	3.430000e+06	3600.000000	2.000000	1.000000	1.000000	0.000000
50%	4.340000e+06	4600.000000	3.000000	1.000000	2.000000	0.000000
75%	5.740000e+06	6360.000000	3.000000	2.000000	2.000000	1.000000
max	1.330000e+07	16200.000000	6.000000	4.000000	4.000000	3.000000

In [6]:

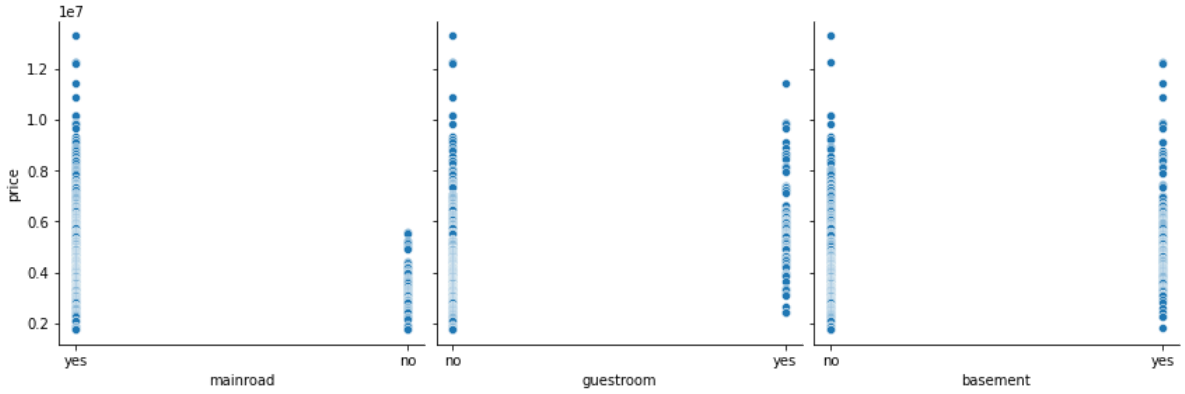
```
sns.heatmap(df.corr(), cmap="YlGnBu", annot = True)
plt.show()
```



In [7]:

```
sns.pairplot(df, x_vars=['mainroad', 'guestroom', 'basement'], y_vars='price',size=4, as
plt.show()
```

```
c:\users\manya\python\lib\site-packages\seaborn\axisgrid.py:2076: UserWarn
ing: The `size` parameter has been renamed to `height`; please update your
code.
  warnings.warn(msg, UserWarning)
```

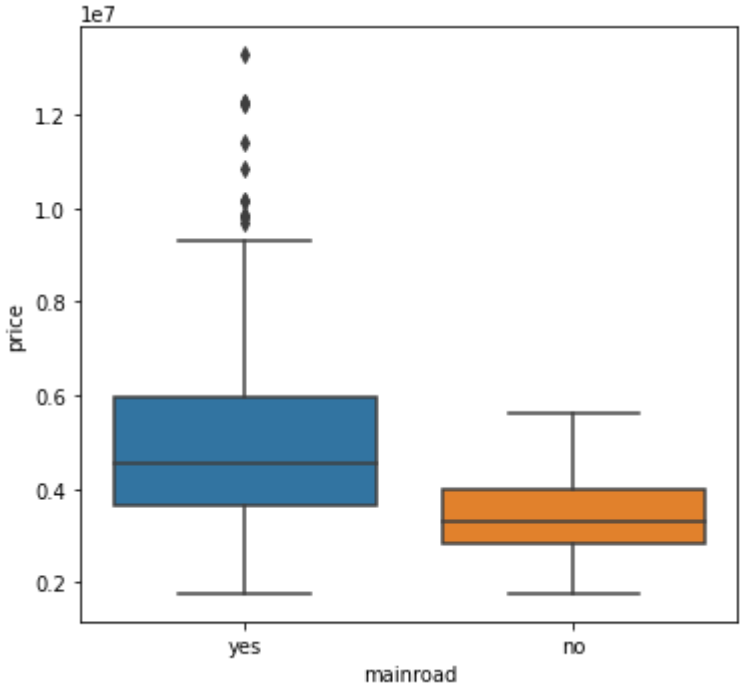


In [8]:

```
plt.figure(figsize=(20,12))
plt.subplot(2,3,1)
sns.boxplot(x='mainroad',y='price',data=df)
```

Out[8]:

```
<AxesSubplot:xlabel='mainroad', ylabel='price'>
```



DATA PREPARATION

In [9]:

```
def binary_map(x):
    return x.map({'yes': 1, "no": 0})
```

In [10]:

```
#binary_map()
```

In [11]:

```
df.head()
```

Out[11]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterhe
0	13300000	7420	4	2	3	yes	no	no	
1	12250000	8960	4	4	4	yes	no	no	
2	12250000	9960	3	2	2	yes	no	yes	
3	12215000	7500	4	2	2	yes	no	yes	
4	11410000	7420	4	1	2	yes	yes	yes	

In [12]:

```
varlist =  ['mainroad', 'guestroom', 'basement', 'hotwaterheating', 'airconditioning',

# Defining the map function
def binary_map(x):
    return x.map({'yes': 1, "no": 0})

# Applying the function to the housing list
df[varlist] = df[varlist].apply(binary_map)
```

In [13]:

```
df.head()
```

Out[13]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterhe
0	13300000	7420	4	2	3	1	0	0	
1	12250000	8960	4	4	4	1	0	0	
2	12250000	9960	3	2	2	1	0	1	
3	12215000	7500	4	2	2	1	0	1	
4	11410000	7420	4	1	2	1	1	1	

In [14]:

```
# Get the dummy variables for the feature 'furnishingstatus' and store it in a new varia
status = pd.get_dummies(df['furnishingstatus'])
```

In [15]:

```
print(status)
```

```
     furnished  semi-furnished  unfurnished
0            1               0            0
1            1               0            0
2            0               1            0
3            1               0            0
4            1               0            0
..         ...             ...          ...
540          0               0            1
541          0               1            0
542          0               0            1
543          1               0            0
544          0               0            1

[545 rows x 3 columns]
```

In [16]:

```
status = pd.get_dummies(df['furnishingstatus'], drop_first = True)
```

In [17]:

```
status.head()
```

Out[17]:

	semi-furnished	unfurnished
0	0	0
1	0	0
2	1	0
3	0	0
4	0	0

In [18]:

```
df=pd.concat([df,status],axis =1)
df.head()
```

Out[18]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterhe
0	13300000	7420	4	2	3	1	0	0	
1	12250000	8960	4	4	4	1	0	0	
2	12250000	9960	3	2	2	1	0	1	
3	12215000	7500	4	2	2	1	0	1	
4	11410000	7420	4	1	2	1	1	1	

In [19]:

```
df.drop(labels=['furnishingstatus'],inplace=True,axis=1)
```

In [20]:

```
df.head()
```

Out[20]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterhe
0	13300000	7420	4	2	3	1	0	0	
1	12250000	8960	4	4	4	1	0	0	
2	12250000	9960	3	2	2	1	0	1	
3	12215000	7500	4	2	2	1	0	1	
4	11410000	7420	4	1	2	1	1	1	

CREATING THE REGRESSION MODEL

In [21]:

```
from sklearn.model_selection import train_test_split

# We specify this so that the train and test data set always have the same rows, respect
np.random.seed(0)
df_train, df_test = train_test_split(df, train_size = 0.7, test_size = 0.3, random_state
```

In [22]:

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
num_vars = ['area', 'bedrooms', 'bathrooms', 'stories', 'parking','price']

scaler.fit_transform(df_train[num_vars])
```

Out[22]:

```
array([[0.15522703, 0.4       , 0.        , 0.        , 0.33333333,
        0.16969697],
       [0.40337909, 0.4       , 0.5       , 0.33333333, 0.33333333,
        0.61515152],
       [0.1156283 , 0.4       , 0.5       , 0.        , 0.        ,
        0.32121212],
       ...,
       [0.13938754, 0.2       , 0.        , 0.33333333, 0.33333333,
        0.17575758],
       [0.36642027, 0.4       , 0.5       , 0.66666667, 0.        ,
        0.42424242],
       [0.51601549, 0.2       , 0.        , 0.        , 0.        ,
        0.06060606]])
```

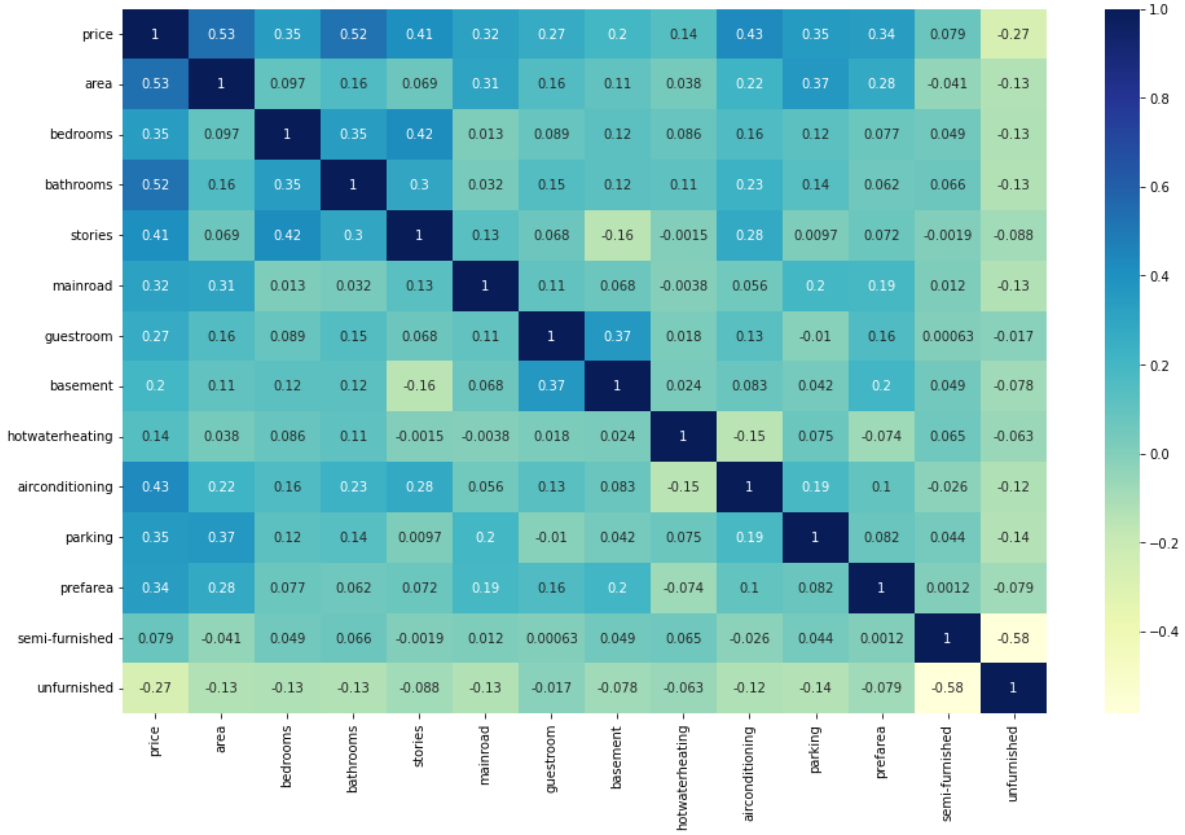
In [23]:

```
df_train[num_vars] = scaler.fit_transform(df_train[num_vars])
```

```
c:\users\manya\python\lib\site-packages\pandas\core\frame.py:3673: Setting
WithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-doc
s/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://
pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-
view-versus-a-copy)
  self[col] = igetitem(value, i)
```

In [24]:

```
plt.figure(figsize = (16, 10))
sns.heatmap(df_train.corr(), annot = True, cmap="YlGnBu")
plt.show()
```

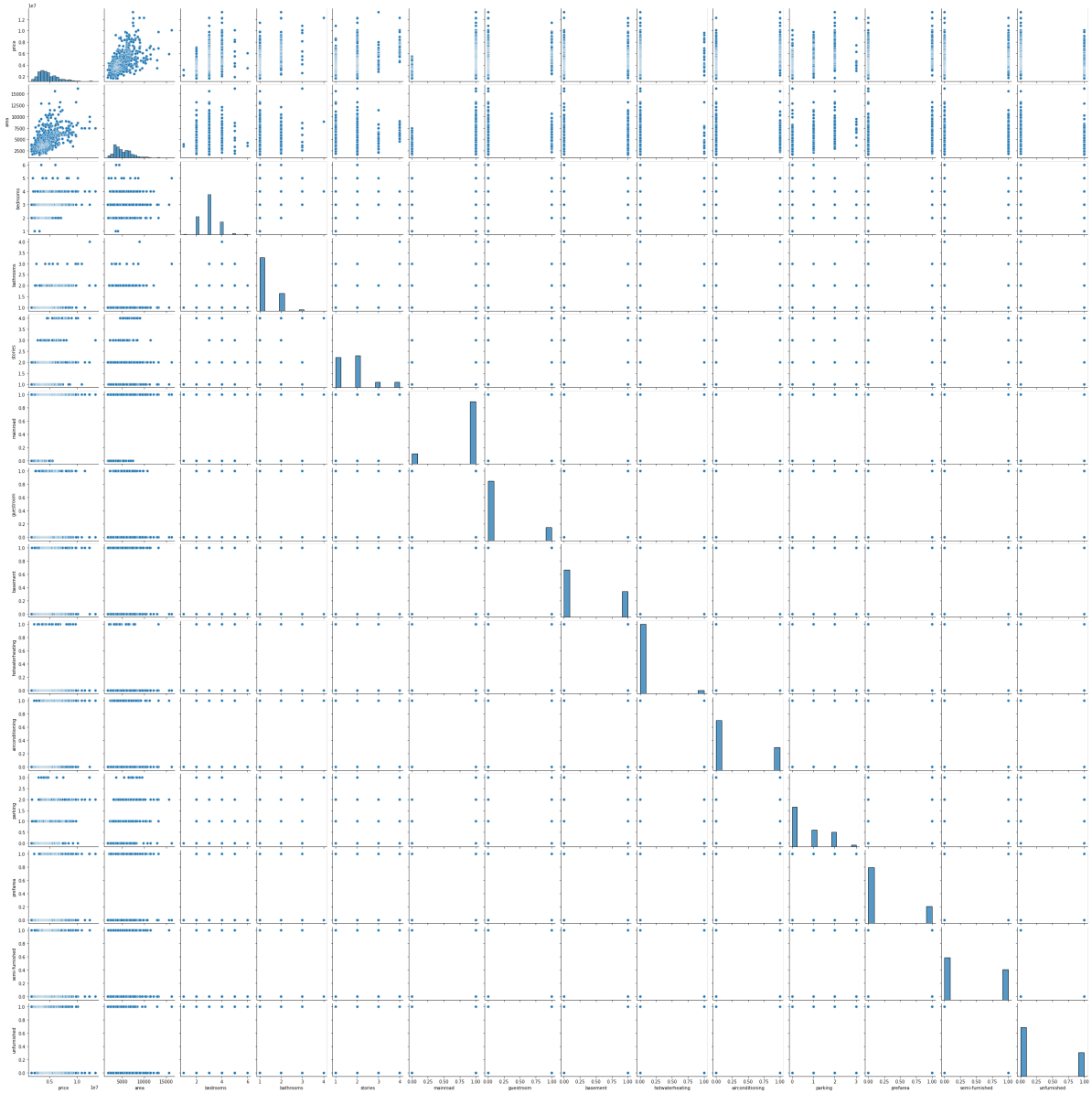


In [25]:

```
sns.pairplot(data=df)
```

Out[25]:

```
<seaborn.axisgrid.PairGrid at 0x1e4cc53ddf0>
```



In [26]:

```
t = df.pop('area')
```

In [27]:

```
t
```

Out[27]:

```
0      7420
1      8960
2      9960
3      7500
4      7420
       ...
540    3000
541    2400
542    3620
543    2910
544    3850
Name: area, Length: 545, dtype: int64
```

In [28]:

```
#was done agian that why not working see whatapp filw sent to someone where u will find

y_train = df_train.pop("price")
x_train = df_train
y_test = df_test.pop("price")
x_test = df_test
```

In [29]:

```
import statsmodels.api as sm

# Add a constant
X_train_lm = sm.add_constant(x_train[['area']])

# Create a first fitted model
lr = sm.OLS(y_train, X_train_lm).fit()
```

In [30]:

```
lr.summary()
```

Out[30]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.283
Model:	OLS	**Adj. R-squared:**	0.281
Method:	Least Squares	**F-statistic:**	149.6
Date:	Tue, 08 Nov 2022	**Prob (F-statistic):**	3.15e-29
Time:	12:53:52	**Log-Likelihood:**	227.23
No. Observations:	381	**AIC:**	-450.5
Df Residuals:	379	**BIC:**	-442.6
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.1269	0.013	9.853	0.000	0.102	0.152
area	0.4622	0.038	12.232	0.000	0.388	0.536

Omnibus:	67.313	**Durbin-Watson:**	2.018
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	143.063
Skew:	0.925	**Prob(JB):**	8.59e-32
Kurtosis:	5.365	**Cond. No.**	5.99

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [31]:

```
x_train_ba = sm.add_constant(x_train[['area','bathrooms']])
lr_ab = sm.OLS(y_train,x_train_ba).fit()
lr_ab.summary()
```

Out[31]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.480
Model:	OLS	**Adj. R-squared:**	0.477
Method:	Least Squares	**F-statistic:**	174.1
Date:	Tue, 08 Nov 2022	**Prob (F-statistic):**	2.51e-54
Time:	12:53:52	**Log-Likelihood:**	288.24
No. Observations:	381	**AIC:**	-570.5
Df Residuals:	378	**BIC:**	-558.6
Df Model:	2		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.1046	0.011	9.384	0.000	0.083	0.127
area	0.3984	0.033	12.192	0.000	0.334	0.463
bathrooms	0.2984	0.025	11.945	0.000	0.249	0.347

Omnibus:	62.839	**Durbin-Watson:**	2.157
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	168.790
Skew:	0.784	**Prob(JB):**	2.23e-37
Kurtosis:	5.859	**Cond. No.**	6.17

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

In [32]:

```
# Check for the VIF values of the feature variables.
from statsmodels.stats.outliers_influence import variance_inflation_factor
# Create a dataframe that will contain the names of all the feature variables and their
vif = pd.DataFrame()
vif['Features'] = x_train.columns
vif['VIF'] = [variance_inflation_factor(x_train.values, i) for i in range(x_train.shape[
vif['VIF'] = round(vif['VIF'], 2)
vif = vif.sort_values(by = "VIF", ascending = False)
vif
```

Out[32]:

	Features	VIF
1	bedrooms	7.33
4	mainroad	6.02
0	area	4.67
3	stories	2.70
11	semi-furnished	2.19
9	parking	2.12
6	basement	2.02
12	unfurnished	1.82
8	airconditioning	1.77
2	bathrooms	1.67
10	prefarea	1.51
5	guestroom	1.47
7	hotwaterheating	1.14

EXPERIMENT - 10

AIM:

Data Visualization in Python using Matplotlib. (Any data)

RESOURCES USED:

Python 3.9.6, Jupyter Notebook 6.4.6

THEORY AND SOURCE CODE

Data visualization is a field in data analysis that deals with visual representation of data. It graphically plots data and is an effective way to communicate inferences from data.

Using data visualization, we can get a visual summary of our data. With pictures, maps and graphs, the human mind has an easier time processing and understanding any given data. Data visualization plays a significant role in the representation of both small and large data sets, but it is especially useful when we have large data sets, in which it is impossible to see all of our data, let alone process and understand it manually.

Data Visualization in Python Python offers several plotting libraries, namely Matplotlib, Seaborn and many other such data visualization packages with different features for creating informative, customized, and appealing plots to present data in the most simple and effective way.

Matplotlib and Seaborn Matplotlib and Seaborn are python libraries that are used for data visualization. They have inbuilt modules for plotting different graphs. While Matplotlib is used to embed graphs into applications, Seaborn is primarily used for statistical graphs.

In [2]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [3]:
```
df=pd.read_csv("Housing.csv")
df.corr()
```

Out[3]:

	price	area	bedrooms	bathrooms	stories	parking
price	1.000000	0.535997	0.366494	0.517545	0.420712	0.384394
area	0.535997	1.000000	0.151858	0.193820	0.083996	0.352980
bedrooms	0.366494	0.151858	1.000000	0.373930	0.408564	0.139270
bathrooms	0.517545	0.193820	0.373930	1.000000	0.326165	0.177496
stories	0.420712	0.083996	0.408564	0.326165	1.000000	0.045547
parking	0.384394	0.352980	0.139270	0.177496	0.045547	1.000000

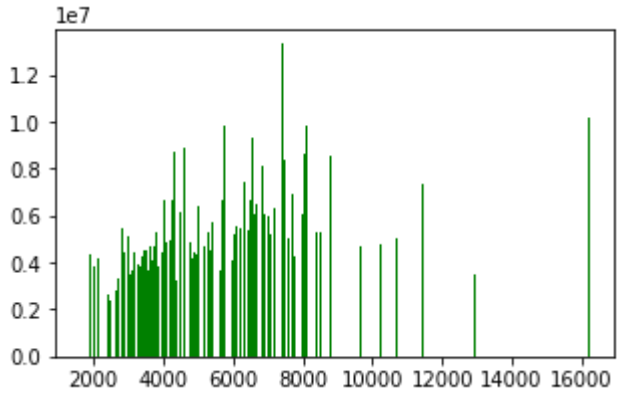
In [4]:
```
df.head()
```

Out[4]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterheating	airco
0	13300000	7420	4	2	3	yes	no	no	no	
1	12250000	8960	4	4	4	yes	no	no	no	
2	12250000	9960	3	2	2	yes	no	yes	no	
3	12215000	7500	4	2	2	yes	no	yes	no	
4	11410000	7420	4	1	2	yes	yes	yes	no	

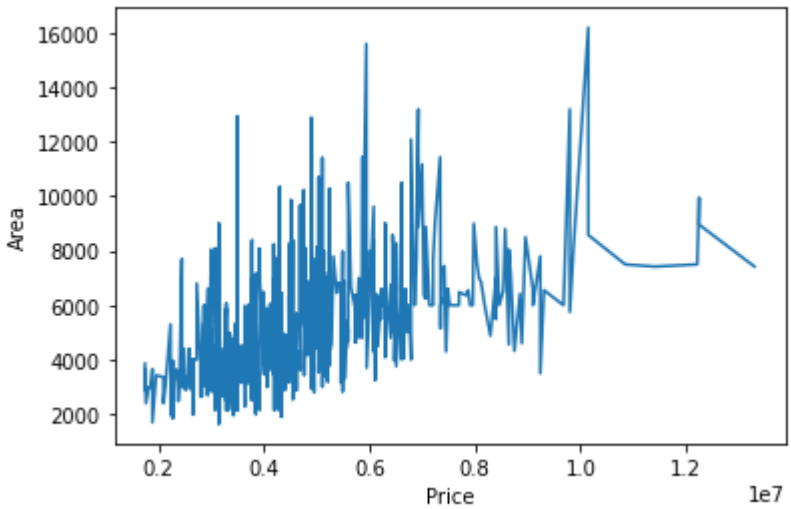
BAR PLOT

In [10]:
```
plt.figure(figsize=(5,3))
plt.bar(df["area"], df["price"], color = 'g', width = 20, label = "area")
plt.show()
```



LINE PLOT

In [36]:
```
plt.figure(figsize=(6,4))
plt.plot(df["price"], df["area"])
plt.xlabel("Price")
plt.ylabel("Area")
plt.show()
```



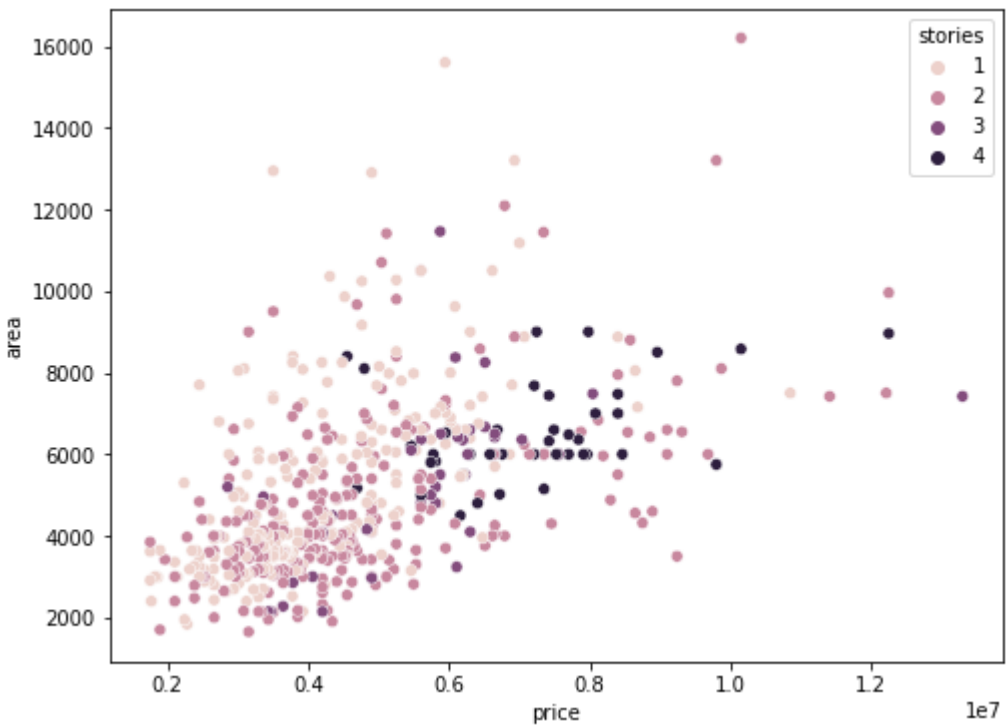
HEATMAP

In [39]:
```
plt.figure(figsize=(7,5))
sns.heatmap(df.corr(), cmap="YlGnBu", annot = True)
plt.show()
```

SCATTER PLOT

In [41]:
```
plt.figure(figsize=(8,6))
sns.scatterplot( df['price'], df['area'], hue =df["stories"])
plt.show()
```

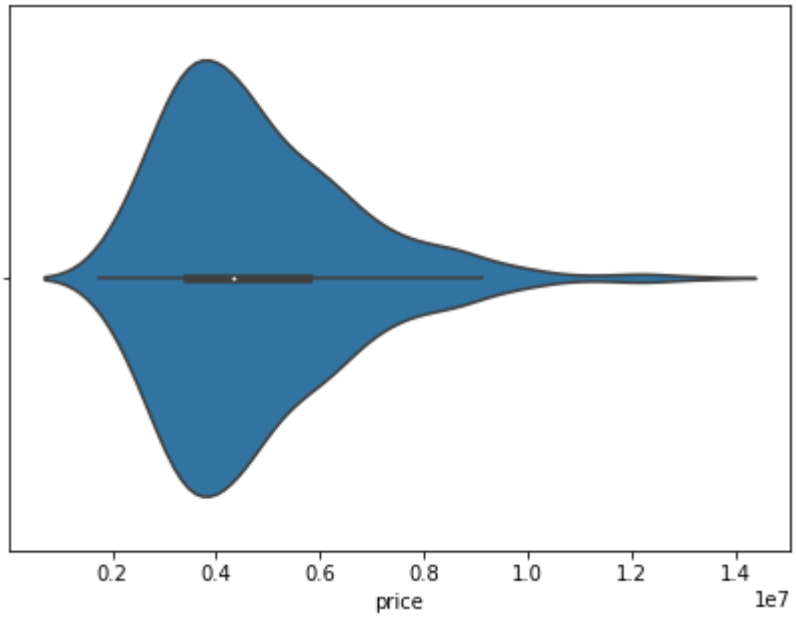
```
c:\users\manya\python\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: P
ass the following variables as keyword args: x, y. From version 0.12, the only vali
d positional argument will be `data`, and passing other arguments without an explic
it keyword will result in an error or misinterpretation.
  warnings.warn(
```



VIOLIN PLOT

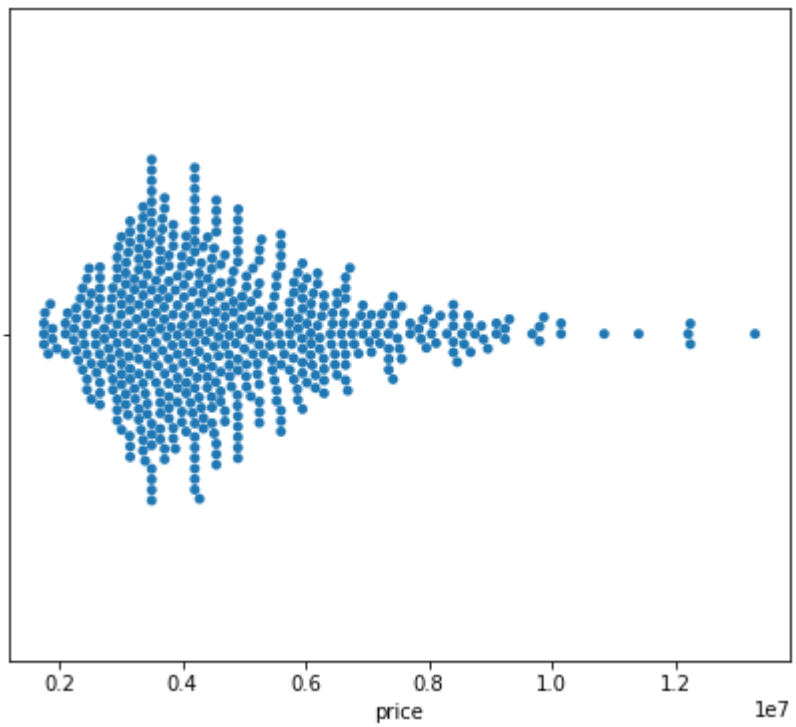
In [42]:
```
plt.figure(figsize=(7,5))
sns.violinplot(df['price'])
plt.show()
```

```
c:\users\manya\python\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: P
ass the following variable as a keyword arg: x. From version 0.12, the only valid p
ositional argument will be `data`, and passing other arguments without an explicit
keyword will result in an error or misinterpretation.
  warnings.warn(
```



SWARM PLOT

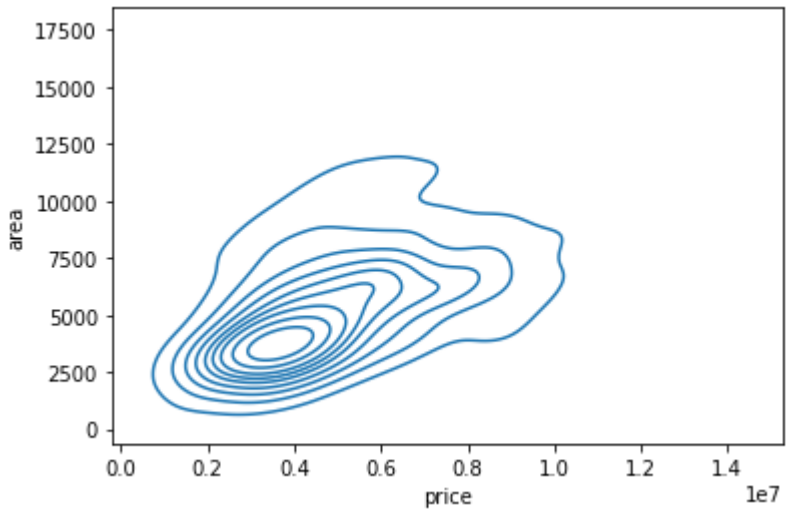
In [44]:
```
plt.figure(figsize=(7,6))
sns.swarmplot(x = df["price"])
plt.show()
```



KDE PLOT

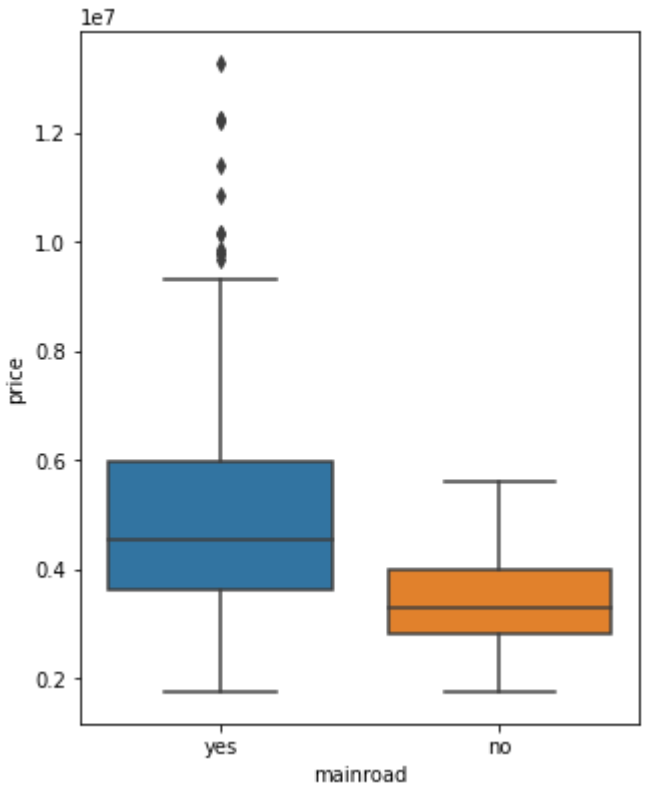
In [47]:
```
plt.figure(figsize=(6,4))
sns.kdeplot( df['price'], df['area'])
plt.show()
```

```
c:\users\manya\python\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: P
ass the following variable as a keyword arg: y. From version 0.12, the only valid p
ositional argument will be `data`, and passing other arguments without an explicit
keyword will result in an error or misinterpretation.
  warnings.warn(
```



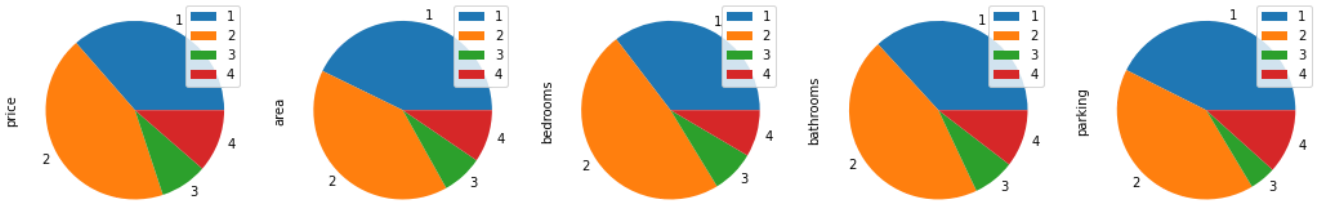
BOX PLOT

In [51]:
```
plt.figure(figsize=(17,14))
plt.subplot(2,3,1)
sns.boxplot(x = 'mainroad', y = 'price', data = df)
plt.show()
```



BAR PLOT

In [75]:
```
dataframe=df
dataframe.groupby(['stories']).sum().plot(kind='pie',subplots=True,figsize=(18, 18))
plt.show()
```



PAIR PLOT

In [77]:
```
sns.pairplot(data=df,height=1)
plt.show()
```