WEEK 1

Reading txt File

```
data <- read.table(file = "F:/lab/airquality.txt", header = TRUE) # Equivalent

> head(airquality)

  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

Import text from URL

```
> url <- "http://courses.washington.edu/b517/Datasets/string.txt"

> data <- read.table(url, header = TRUE)

> head(data)

   x       y
1 10 34.7081
2 12 34.5034
3 14 36.5656
4 16 38.3125
5 18 42.5441
6 20 43.7210
```

Read a CSV from a URL

```
> # importing Data
> data <- read.csv('https://www.stats.govt.nz/assets/Uploads\
+ /Annual-enterprise-survey/Annual-enterprise-survey-2020-\
+ financial-year-provisional/Download-data/annual-enterprise\
+ -survey-2020-financial-year-provisional-csv.csv')
>
> # display top 5 row
> head(data)
   x      y
1 10 34.7081
2 12 34.5034
3 14 36.5656
4 16 38.3125
5 18 42.5441
```

Read a CSV from a URL

```
 > data <- read.csv("https://www.stats.govt.nz/large-datasets/csv-files-for-download/")
> head(data)
                                        X..DOCTYPE.html.
1                                      <!--[if !IE]><!-->
2                                       <html lang=en-NZ>
3                                        <!--<![endif]-->
4 <!--[if lt IE 9 ]><html lang=en-NZ class=ie ie8 lt-ie9><![endif]-->
```

```
5        <!--[if IE 9 ]><html lang=en-NZ class=ie ie9><![endif]-->
6                <head profile=http://www.w3.org/2005/10/profile>
```

Reading Excel File from URL

```
> xlsx_example <- readxl_example("datasets.xlsx")
> read_excel(xlsx_example)

# A tibble: 150 × 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl> <chr>
 1         5.1         3.5          1.4         0.2 setosa
 2         4.9         3            1.4         0.2 setosa
 3         4.7         3.2          1.3         0.2 setosa
 4         4.6         3.1          1.5         0.2 setosa
 5         5           3.6          1.4         0.2 setosa
 6         5.4         3.9          1.7         0.4 setosa
 7         4.6         3.4          1.4         0.3 setosa
 8         5           3.4          1.5         0.2 setosa
 9         4.4         2.9          1.4         0.2 setosa
10         4.9         3.1          1.5         0.1 setosa
# ℹ 140 more rows
# ℹ Use `print(n = ...)` to see more rows

> excel_sheets(xlsx_example)
```

```
[1] "iris"     "mtcars"  "chickwts" "quakes"

> read_excel(xls_example, sheet = 4)

# A tibble: 1,000 × 5
     lat  long depth  mag stations
   <dbl> <dbl> <dbl> <dbl>    <dbl>
 1 -20.4  182.   562   4.8       41
 2 -20.6  181.   650   4.2       15
 3 -26    184.    42   5.4       43
 4 -18.0  182.   626   4.1       19
 5 -20.4  182.   649   4         11
 6 -19.7  184.   195   4         12
 7 -11.7  166.    82   4.8       43
 8 -28.1  182.   194   4.4       15
 9 -28.7  182.   211   4.7       35
10 -17.5  180.   622   4.3       19
# ℹ 990 more rows
# ℹ Use `print(n = ...)` to see more rows
```

10_NumPy

April 17, 2020

[1]:
```
height = [1.73, 1.68, 1.71, 1.89, 1.79]
```

[2]:
```
height
```

[2]:
```
[1.73, 1.68, 1.71, 1.89, 1.79]
```

[3]:
```
weight = [65.4, 59.2, 63.6, 88.4, 68.7]
```

[4]:
```
weight
```

[4]:
```
[65.4, 59.2, 63.6, 88.4, 68.7]
```

[5]:
```
weight / height ** 2
```

```
↪---------------------------------------------------------------------------

    TypeError                                 Traceback (most recent call␣
↪last)

    <ipython-input-5-6a4c0c70e3b9> in <module>
  ----> 1 weight / height ** 2


    TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

Solution: NumPy (short for Numerical Python): Provide much more efficient storage and data operations as the size grows. - Alternative to Python List: NumPy Array, - Calculations over entire arrays - Easy and Fast

[6]:
```
import numpy as np
```

[7]:
```
np.__version__
```

[7]:
```
'1.18.1'
```

[8]:
```python
np_height = np.array(height)
```

[9]:
```python
np_height
```

[9] :
```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

[10] :
```python
type(np_height)
```

[10] :
```
numpy.ndarray
```

[11] :
```python
np_weight = np.array(weight)
```

[12] :
```python
np_weight
```

[12] :
```
array([65.4, 59.2, 63.6, 88.4, 68.7])
```

[13] :
```python
bmi = np_weight / np_height ** 2
```

[14] :
```python
bmi
```

[14] :
```
array([21.85171573, 20.97505669, 21.75028214, 24.7473475 , 21.44127836])
```

We'll cover a few categories of basic array manipulations here:

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

0.1 NumPy Array Attributes

First let's discuss some useful array attributes. We'll start by defining three random arrays, a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will *seed* with a set value in order to ensure that the same random arrays are generated each time this code is run:

[15] :
```python
np.random.seed(1) # seed for reproducibility

x1 = np.random.randint(10, size=6)  # One-dimensional array
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array
```

[16] :
```python
x1
```

[16] :
```
array([5, 8, 9, 5, 0, 0])
```

[17]:
```python
x2 #3 rows, 4 columns
```

[17]:
```
array([[1, 7, 6, 9],
       [2, 4, 5, 2],
       [4, 2, 4, 7]])
```

[18]:
```python
x3 #4 rows, 5 columns and 3 items
```

[18]:
```
array([[[7, 9, 1, 7, 0],
        [6, 9, 9, 7, 6],
        [9, 1, 0, 1, 8],
        [8, 3, 9, 8, 7]],

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

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

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

[19]:
```python
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Another useful attribute is the `dtype`, the data type of the array.

[20]:
```python
print("dtype:", x3.dtype)
```

```
dtype: int32
```

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array:

[21]:
```python
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes") # nbytes = itemsize * size
```

```
itemsize: 4 bytes
nbytes: 240 bytes
```

In general, we expect that `nbytes` is equal to `itemsize` times `size`.

0.2 Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

[22] :
```
x1
```

[22] : array([5, 8, 9, 5, 0, 0])

[23] :
```
x1[0]
```

[23]: 5

[24] :
```
x1[4]
```

[24]: 0

To index from the end of the array, you can use negative indices:

[25] :
```
x1[-1] # refers the last element
```

[25]: 0

[26] :
```
x1[-2]
```

[26]: 0

Values can also be modified using any of the above index notation:

[27] :
```
x2
```

[27] :
```
array([[1, 7, 6, 9],
       [2, 4, 5, 2],
       [4, 2, 4, 7]])
```

[28] :
```
x2[1,2]
```

[28]: 5

[29] :
```
x2[0, 0] = 15
x2
```

[29] :
```
array([[15, 7,  6,  9],
       [ 2,  4,  5,  2],
       [ 4,  2,  4,  7]])
```

[30] :
```
x2[1,1] = 12
x2
```

[30]: array([[15, 7, 6, 9],
       [2, 12, 5, 2],
       [4, 2, 4, 7]])

0.3 Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (:) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

x[start:stop:step]

If any of these are unspecified, they default to the values start=0, stop=*size of dimension*, step=1. We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

0.3.1 One-dimensional subarrays

[31]:
```
x = np.arange(10)
x
```

[31]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

[32]:
```
x[:5]  # first five elements
```

[32]: array([0, 1, 2, 3, 4])

[33]:
```
x[5:]  # elements after index 5
```

[33]: array([5, 6, 7, 8, 9])

[34]:
```
x[4:7]  # middle sub-array
```

[34]: array([4, 5, 6])

[35]:
```
x[::2]  # every other element
```

[35]: array([0, 2, 4, 6, 8])

[36]:
```
x[1::2]  # every other element, starting at index 1
```

[36]: array([1, 3, 5, 7, 9])

A potentially confusing case is when the step value is negative. In this case, the defaults for start and stop are swapped. This becomes a convenient way to reverse an array:

[37]:
```
x[::-1]  # all elements, reversed
```

[37]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

[38] :
```python
x[5::-1]  # reversed every other from index 5
```

[38] :
```
array([5, 4, 3, 2, 1, 0])
```

0.3.2 Multi-dimensional subarrays

Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

[39] :
```python
x2
```

[39] :
```
array([[15,  7,  6,  9],
       [ 2, 12,  5,  2],
       [ 4,  2,  4,  7]])
```

[40] :
```python
x2[1,:] # second row
```

[40] :
```
array([ 2, 12,  5,  2])
```

[41] :
```python
x2[1:,1:3]
```

[41] :
```
array([[12,  5],
       [ 2,  4]])
```

[42] :
```python
x2[:,1:3]
```

[42] :
```
array([[ 7,  6],
       [12,  5],
       [ 2,  4]])
```

[43] :
```python
x2[0:2, :2]   # two rows, two columns
```

[43] :
```
array([[15,  7],
       [ 2, 12]])
```

[44] :
```python
x2[:3, :]   # all rows, every other column
```

[44] :
```
array([[15,  7,  6,  9],
       [ 2, 12,  5,  2],
       [ 4,  2,  4,  7]])
```

[45] :
```python
x2[:, 1::2]
```

[45] :
```
array([[ 7,  9],
       [12,  2],
       [ 2,  7]])
```

Finally, subarray dimensions can even be reversed together:

[46] :
```
x2
```

[46] :
```
array([[15,  7,  6,  9],
       [ 2, 12,  5,  2],
       [ 4,  2,  4,  7]])
```

[47] :
```
x2[::-1, ::-1] # all rows and columns reversed
```

[47] :
```
array([[ 7,  4,  2,  4],
       [ 2,  5, 12,  2],
       [ 9,  6,  7, 15]])
```

[48] :
```
x3
```

[48]:
```
array([[[7, 9, 1, 7, 0],
        [6, 9, 9, 7, 6],
        [9, 1, 0, 1, 8],
        [8, 3, 9, 8, 7]],

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

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

[49] :
```
x3[0,0:2,:] # element no, row, column
```

[49] :
```
array([[7, 9, 1, 7, 0],
       [6, 9, 9, 7, 6]])
```

[50] :
```
x3[2,0:2,0::2]
```

[50] :
```
array([[9, 6, 3],
       [7, 4, 9]])
```

Accessing array rows and columns One commonly needed routine is accessing of single rows or columns of an array. This can be done by combining indexing and slicing, using an empty slice marked by a single colon (:):

[51] :
```
print(x2[:, 0])  # first column of x2
```

```
[15  2  4]
```

[52] :
```python
print(x2[0, :])    # first row of x2
```

```
[15  7  6  9]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

[53] :
```python
print(x2[0]) # equivalent to x2[0, :]
```

```
[15  7  6  9]
```

0.4 Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing this is with the `reshape` method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following:

[54] :
```python
grid = np.arange(1, 10).reshape((3, 3)) #converting 1D array to 2D
print(grid)
```

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

Note that for this to work, the size of the initial array must match the size of the reshaped array.

Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. This can be done with the `reshape` method, or more easily done by making use of the `newaxis` keyword within a slice operation:

[55] :
```python
x = np.array([1, 2, 3])
# row vector via reshape
x.reshape((1, 3))
```

[55]: `array([[1, 2, 3]])`

[56]:
```python
# row vector via newaxis
x[np.newaxis, :]
```

[56]: `array([[1, 2, 3]])`

[57]:
```python
x[:, np.newaxis] # reshaping on column
```

[57]:
```
array([[1],
       [2],
       [3]])
```

[58]:
```python
# column vector via reshape
x.reshape((1, 3))
```

[58]: `array([[1, 2, 3]])`

0.5 Array Concatenation and Splitting

All of the preceding routines worked on single arrays. It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays. We'll take a look at those operations here.

0.5.1 Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, is primarily accomplished using the routines `np.concatenate`, `np.vstack`, and `np.hstack`. `np.concatenate` takes a tuple or list of arrays as its first argument, as we can see here:

[59]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

[59]:
```
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once:

[60]:
```
z = np.array([99, 99, 99])
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

It can also be used for two-dimensional arrays:

[61]:
```
grid = np.array([[1, 2, 3],[4, 5, 6]])
grid
```

[61]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[62]:
```
# concatenate along the first axis
np.concatenate([grid, grid])
```

[62]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

[63]:
```
# concatenate along the second axis (zero-indexed)
np.concatenate([grid, grid], axis=1)
```

[63]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

0.5.2 Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:

[64]:
```
x = np.array([1, 2, 3, 99, 99, 3, 2, 1])
x1, x2= np.split(x, [2])
print(x1, x2)
```

```
[1 2] [ 3 99 99 3  2  1]
```

[65]:
```
x = np.array([1, 2, 3, 99, 99, 3, 2, 1])
x1,x2, x3 = np.split(x, [2, 4])
print(x1, x2, x3)
```

```
[1 2] [ 3 99] [99  3  2  1]
```

Notice that N split-points, leads to $N + 1$ subarrays. The related functions np.hsplit and np.vsplit are similar: - Split an array into multiple sub-arrays vertically (row-wise). - Split an array into multiple sub-arrays horizontally (column-wise)

[66]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

[66]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[67]:
```
upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

[68]:
```
left, right = np.hsplit(grid, [2])
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
```

```
[10 11]
[14 15]]
```

[69] :

```
np.hsplit?
```

11_Pandas

October 31, 2019

1 1. What is Pandas

Pandas is a Python package providing fast, exible, and expressive data structures. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python (http://pandas.pydata.org/)

Pandas builds on top of Numpy to ease managing heterogeneous data sets.

1.1 1.1 Data Handled by Pandas

Pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns (comparable to EXCEL, R or relational Databases)
- Time series data
- Matrix data(homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets.

1.2 1.2 Feature Overview

- Easy handling of missing data (represented as NaN)
- Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
- Automatic and explicit data alignment
- Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both ag- gregating and transforming data
- Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
- Intuitive merging and joining data sets
- Flexible reshaping and pivoting of data sets
- Hierarchical labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading and storing data
- Time series-specific functionality

2 2. Pandas Data Structures

Pandas is build around two data structures

- `Series` represent 1 dimensional datasets as subclass of Numpy's ndarray

- DataFrame represent 2 dimensional data sets as list of Series

  For all data structures, labels/indices can be defined per row and column.
  Data alignment is intrinsict, i.e. the link between labels and data will not be broken.
  Series: * Homogeneous data * Size Immutable * Values of Data Mutable
  Data Frames: * Heterogeneous data * Size Mutable * Data Mutable

2.1 2.1. Series

Series is a one-dimensional labeled array capable of holding any data type (integers, strings, oating point numbers,Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call:

Series(data, index=index)

data may be a dict, a numpy.ndarray or a sclar value
A series can be created using various inputs like

- Array
- Dict
- Scalar value or constant

2.1.1 2.1.1 Creating a series from ndarray

[1]:
```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print (s)
```

```
0    a
1    b
2    c
3    d
dtype: object
```

[2]:
```
data = np.array(['a','b','c','d'])
s = pd.Series(data, index=[100,101,102,103])
print (s)
```

```
100    a
101    b
102    c
103    d
dtype: object
```

2.1.2 2.1.2 Creating a Series from dict

A dict can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If index is passed, the values in data corresponding to the labels in the index will be pulled out.

[3]:
```
data = {'a' : 0, 'b' : 1, 'c' : 2}
s = pd.Series(data)
print (s)
```

```
a    0
b    1
c    2
dtype: int64
```

Dictionary keys are used to construct index.

[4]:
```
data1 = {'a' : 0., 'b' : 1., 'c' : 2.}
s1 = pd.Series(data1,index=['b','c','d','a'])
print (s1)
```

```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

Index order is persisted and the missing element is filled with NaN (Not a Number).

2.1.3 2.1.3 Creating a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of index

[5]:
```
s = pd.Series(5, index=[0, 1, 2, 3])
print (s)
```

```
0    5
1    5
2    5
3    5
dtype: int64
```

[6]:
```
#show the index
s.index
```

[6]:
```
Int64Index([0, 1, 2, 3], dtype='int64')
```

[7]:
```
#show the value
s.values
```

[7]:
```
array([5, 5, 5, 5])
```

2.1.4 2.1.4 Series Indexing

Accessing elements in a series can be either done via the number or the index

[8]:
```
s = pd.Series([1,2,3,4,5], index = ['a','b','c','d','e'])

#retrieve a single element
s['c']
```

[8]: 3

[9]:
```
#retrieve multiple elements
s[['a','c','d']]
```

[9]:
```
a    1
c    3
d    4
dtype: int64
```

[10]:
```
s.at['a'] # value at index 'a'
```

[10]: 1

2.2 2.2. DataFrame: a Series of Series

The pandas DataFrame is a 2 dimensional labeled data structure with columns of potentially different types. Similar to * a spreadsheet * relational database table * a dictionary of series

Creating DataFrame's

A pandas DataFrame can be created using various inputs like

- Lists
- Dict
- Series
- Numpy ndarrays
- Another DataFrame

2.2.1 2.2.1 Create a DataFrame from Lists

[11]:
```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
df
```

[11]:

	0
0	1
1	2
2	3
3	4
4	5

[12]:
```
data = [['Ramesh',10],['Himesh',12],['Suresh',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
```

```
df
```

[12]:

	Name	Age
0	Ramesh	10
1	Himesh	12
2	Suresh	13

[13]:

```
data = [['Ramesh',10],['Himesh',12],['Suesh',13]]
            ataFrame(data,columns=['Name','Age'],
                      dtype=float)
df
```

[13]:

	Name	Age
0	Ramesh	10.0
1	Himesh	12.0
2	Suesh	13.0

2.2.2 2.2.2 Create a DataFrame from Dict of ndarrays / Lists

All the ndarrays must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where n is the array length.

[14]:

```
data = {'Name':['Nitesh','Ramesh','Rajesh','Nilesh'],
        'Age':[28,34,29,45]}
df = pd.DataFrame(data)
df
```

[14]:

	Name	Age
0	Nitesh	28
1	Ramesh	34
2	Rajesh	29
3	Nilesh	45

[15]:

```
data = {'Name':['Ramesh', 'Rajesh', 'Nitesh', 'Nilesh'],
        'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
df
```

[15]:

	Name	Age
rank1	Ramesh	28
rank2	Rajesh	34
rank3	Nitesh	29
rank4	Nilesh	42

2.2.3 2.2.3 Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

[16]:
```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
df
```

[16]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

[17]:
```
data = [{'a':1, 'b':2},{'a':5, 'b':10, 'c':20}]
df = pd.DataFrame(data, index=['first','second']) # passing row indices
df
```

[17]:

	a	b	c
first	1	2	NaN
second	5	10	20.0

[18]:
```
data = [{'a':1, 'b':2},{'a':5, 'b':10, 'c':20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'],
                   columns=['a', 'b'])

#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'],
                   columns=['a', 'b1'])


print (df1)
print()
print (df2)
```

```
        a   b
first   1   2
second  5  10

        a  b1
first   1 NaN
second  5 NaN
```

2.2.4 2.2.4 Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

[19]:
```
d = {'one' : pd.Series([1, 2, 3], index=['a','b','c']),
     'two' : pd.Series([1, 2, 3, 4], index=['a','b','c','d'])}
df = pd.DataFrame(d)
df
```

[19]:

	one	two
a	1.0	1
b	2.0	2

```
c   3.0   3
d   NaN   4
```

2.2.5 2.2.5 Column selection, addition, deletion

[20]:
```
d = {'one' : pd.Series([1, 2, 3],
                       index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4],
                       index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
df['one']
```

[20]:
```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

[21]:
```
# Adding a new column to an existing DF object
# with column label by passing new series
print ("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],
                      index=['a','b','c'])
print (df)
```

```
Adding a new column by passing as Series:
   one  two  three
a  1.0    1   10.0
b  2.0    2   20.0
c  3.0    3   30.0
d  NaN    4    NaN
```

[22]:
```
# Adding a new column using the existing columns
df['four']=df['one']+df['three']
print (df)
```

```
   one  two  three  four
a  1.0    1   10.0  11.0
b  2.0    2   20.0  22.0
c  3.0    3   30.0  33.0
d  NaN    4    NaN   NaN
```

[23]:
```
# deleting a column using del function


del df['one']
df
```

[23] :

	two	three	four
a	1	10.0	11.0
b	2	20.0	22.0
c	3	30.0	33.0
d	4	NaN	NaN

[24] :
```
# Deleting another column using POP function
df.pop('two')
df
```

[24] :

	three	four
a	10.0	11.0
b	20.0	22.0
c	30.0	33.0
d	NaN	NaN

2.2.6 2.2.5 Row Selection, Addition, and Deletion

Selection by Row Label

Rows can be selected by passing row label to a loc function.

[25] :
```
#import pandas as pd
d = {'one' : pd.Series([1, 2, 3],
                       index=['a', 'b', 'c']),
     'two' : pd.Series([1, 2, 3, 4],
                       index=['a', 'b', 'c', 'd'])}


df = pd.DataFrame(d)
print(df)

print("\n Accessing row having label 'b':")
print (df.loc['b'])
```

```
   one  two
a  1.0    1
b  2.0    2
c  3.0    3
d  NaN    4

 Accessing row having label 'b':
one    2.0
two    2.0
Name: b, dtype: float64
```

Selection by integer location

Rows can be selected by passing integer location to an iloc function.

[26] :
```
print (df.iloc[1])
```

```
one    2.0
```

```
two      2.0
Name: b, dtype: float64
```

Slice Rows

Multiple rows can be selected using ' : ' operator.

[27]:
```
print (df[2:4])
```

```
   one  two
c  3.0    3
d  NaN    4
```

Addition of Rows

Add new rows to a DataFrame using the append function. This function will append the rows at the end.

[28]:
```
df1 = pd.DataFrame([[1, 2], [3, 4]], columns=['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns=['a','b'])


df3 = df1.append(df2)
print (df3)
```

```
   a  b
0  1  2
1  3  4
0  5  6
1  7  8
```

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

[29]:
```
# Drop rows with label 0
df = df3.drop(0)

print (df)
```

```
   a  b
1  3  4
1  7  8
```

2.3 3 Basic Functionality

[30]:
```
#import pandas as pd

#Create a Dictionary of series
d={'Name':pd.Series(['Ramesh','Suresh','Rajesh']),
   'Age':pd.Series([25,26,25]),
   'Rating':pd.Series([4.23,3.24,3.98])}
```

[31]:
```python
#Create a DataFrame
df = pd.DataFrame(d)
df
```

[31]:

	Name	Age	Rating
0	Ramesh	25	4.23
1	Suresh	26	3.24
2	Rajesh	25	3.98

T (Transpose)

Returns the transpose of the DataFrame. The rows and columns will interchange.

[32]:
```python
print ("The transpose of the data series is:", )
print (df.T)
```

```
The transpose of the data series is:
             0       1       2
Name    Ramesh  Suresh  Rajesh
Age         25      26      25
Rating    4.23    3.24    3.98
```

axes

Returns the list of row axis labels and column axis labels.

[33]:
```python
print ("Row axis labels and column axis labels are:")
print (df.axes)
```

```
Row axis labels and column axis labels are:
[RangeIndex(start=0, stop=3, step=1), Index(['Name', 'Age', 'Rating'],
dtype='object')]
```

dtypes

Returns the data type of each column.

[34]:
```python
print ("The data types of each column are:")
print (df.dtypes)
```

```
The data types of each column are:
Name       object
Age         int64
Rating    float64
dtype: object
```

ndim

Returns the number of dimensions of the object. By definition, DataFrame is a 2D object.

[35]:
```python
print ("The dimension of the object is:", df.ndim)
```

```
The dimension of the object is: 2
```

shape

Returns a tuple (a,b), where a represents the number of rows and b represents the number of columns.

[36]:
```python
print ("The shape of the object is:",df.shape)
```

```
The shape of the object is: (3, 3)
```

size

Returns the number of elements in the DataFrame.

[37]:
```python
print ("The total no. of elements:",df.size )
```

```
The total no. of elements: 9
```

values

Returns the actual data in the DataFrame as an NDarray.

[38]:
```python
print ("The actual data in our data frame is:")
print (df.values)
```

```
The actual data in our data frame is:
[['Ramesh' 25 4.23]
 ['Suresh' 26 3.24]
 ['Rajesh' 25 3.98]]
```

Head & Tail

To view a small sample of a DataFrame object, use the head() and tail() methods. - head() returns the first n rows (observe the index values). - tail() returns the last few rows

The default number of elements to display is 5, but you may pass a custom number.

[39]:
```python
# first few rows of the data frame
print (df.head())
```

```
     Name  Age  Rating
0  Ramesh   25    4.23
1  Suresh   26    3.24
2  Rajesh   25    3.98
```

[40]:
```python
# first 2 rows of the data frame
print (df.head(2))
```

```
     Name  Age  Rating
0  Ramesh   25    4.23
1  Suresh   26    3.24
```

[41]:
```python
# last few rows of the data frame
print (df.tail())
```

```
     Name  Age  Rating
0  Ramesh   25    4.23
1  Suresh   26    3.24
2  Rajesh   25    3.98
```

3 4. Descriptive Statistics

Descriptive Statistics sumarizes the underlying distribution of data values through statistical values like mean, variance etc.

3.0.1 Basic Functions

Function
Description
count
Number of non-null observations
sum
Sum of values
mean
Mean of values
mad
Mean absolute deviation
median
Arithmetic median of values
min
Minimum
max
Maximum
mode
Mode
abs
Absolute Value
prod
Product of values
std
Unbiased standard deviation
var
Unbiased variance
skew
Unbiased skewness (3rd moment)
kurt
Unbiased kurtosis (4th moment)
quantile
Sample quantile (value at %)
cumsum
Cumulative sum
cumprod
Cumulative product
cummax
Cumulative maximum
cummin
Cumulative minimum

3.0.2 4.1 sum()

Returns the sum of the values for the requested axis. By default, axis is index (axis=0).

[42]:
```
df
```

[42]:

	Name	Age	Rating
0	Ramesh	25	4.23
1	Suresh	26	3.24
2	Rajesh	25	3.98

[43]:
```
print (df.sum()) # axis = 0
```

```
Name       RameshSureshRajesh
Age                        76
Rating                  11.45
dtype: object
```

Each individual column is added individually (Strings are appended).

[44]:
```
print (df.sum(1)) # axis = 1, adds columns
```

```
0    29.23
1    29.24
2    28.98
dtype: float64
```

3.0.3 4.2 mean()

Returns the average value

[45]:
```
print (df.mean())
```

```
Age       25.333333
Rating     3.816667
dtype: float64
```

3.0.4 4.3 std()

Returns the Bressel standard deviation of the numerical columns.

[46]:
```
print (df.std())
```

```
Age       0.577350
Rating    0.514814
dtype: float64
```

[47]:
```
df.min()
```

[47]:
```
Name      Rajesh
Age           25
Rating      3.24
```

```
dtype: object
```

[48]:
```
df.max()
```

[48]:
```
Name       Suresh
Age            26
Rating       4.23
dtype: object
```

3.0.5 4.4 Summarizing Data

The describe() function computes a summary of statistics pertaining to the DataFrame columns.

[49]:
```
print(df)
print (df.describe())
```

```
     Name  Age  Rating
0  Ramesh   25    4.23
1  Suresh   26    3.24
2  Rajesh   25    3.98

             Age    Rating
count   3.000000  3.000000
mean   25.333333  3.816667
std     0.577350  0.514814
min    25.000000  3.240000
25%    25.000000  3.610000
50%    25.000000  3.980000
75%    25.500000  4.105000
max    26.000000  4.230000
```

This function gives the mean, std and IQR values. And, function excludes the character columns and given summary about numeric columns.

'include' is the argument which is used to pass necessary information regarding what columns need to be considered for summarizing. Takes the list of values; by default, 'number'.

- object Summarizes String columns
- number Summarizes Numeric columns
- all Summarizes all columns together (Should not pass it as a list value)

[50]:
```
print (df.describe(include=['object']))
```

```
          Name
count        3
unique       3
top     Ramesh
freq         1
```

[73]:
```
print (df.describe(include='all'))
```

3.1 5. Input/Output Tools

The Pandas I/O API is a set of top level reader functions accessed like pd.read_csv() that generally return a pandas object. * read_csv * read_excel * read_hdf * read_sql * read_json * read_msgpack (experimental) * read_html * read_gbq (experimental) * read_stata * read_clipboard * read_pickle

The corresponding writer functions are object methods that are accessed like df.to_csv(). * to_csv * to_excel * to_hdf * to_sql * to_json * to_msgpack (experimental) * to_html * to_gbq (experimental) * to_stata * to_clipboard * to_pickle

3.1.1 5.1 Loading the Weather Data from the CSV

In this example we load the weather datafrom the data directory ("data_data.csv")

[54]:
```
#! executes a shell command
#!ls data
```

[55]:
```
df = pd.read_csv("data/weather_data.csv")
print (df)
```

```
          outlook  temperature  humidity  windy play
0    1      sunny           85        85  False   no
1    2      sunny           80        90   True   no
2    3   overcast           83        86  False  yes
3    4      rainy           70        96  False  yes
4    5      rainy           68        80  False  yes
5    6      rainy           65        70   True   no
6    7   overcast           64        65   True  yes
```

[56]:
```
pd.read_csv?
```

[57]:
```
df.to_csv('temp1.csv')
```

3.1.2 5.2 Excel data

[ ]:
```
#use help to see the parameters
#pd.read_excel?
```

[58]:
```
import pandas as pd
df_out = pd.DataFrame([('Ramesh', 25), ('Rajesh', 20), ('Kamesh', 35)],
↪columns=['Name', 'Age'])
```

[59]:
```
df_out
```

[59]:

	Name	Age
0	Ramesh	25
1	Rajesh	20
2	Kamesh	35

[60]:
```
df_out.to_excel('tmp.xlsx', index=False)
```

[61]:
```
pd.read_excel('tmp.xlsx')
```

[61]:

	Name	Age
0	Ramesh	25
1	Rajesh	20
2	Kamesh	35

3.1.3 5.3 Sqlite data

[62]:
```
import sqlite3 as lite
import sys

con = lite.connect('employee.db')
```

[63]:
```
with con:
    cur = con.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS csdept(eid INTEGER PRIMARY KEY,␣
↪ename TEXT, esalary INT)")
    cur.execute("INSERT INTO csdept VALUES(101,'Ramesh',25000)")
    cur.execute("INSERT INTO csdept VALUES(102,'Suresh', 6500)")

    cur.execute("INSERT INTO csdept VALUES(103,'Naresh', 45000)")
    cur.execute("INSERT INTO csdept VALUES(104,'Mahesh', 60000)")
```

[64]:
```
q="select * from csdept"
df = pd.read_sql_query(q,con)
print(df)
```

```
   eid   ename esalary
0  101  Ramesh   25000
1  102  Suresh    6500
2  103  Naresh   45000
3  104  Mahesh   60000
```

[65]:
```
query = "SELECT ename FROM csdept WHERE esalary > 20000;"

df = pd.read_sql_query(query,con)

for i in df['ename']:
    print(i)
```

```
Ramesh
Naresh
Mahesh
```

[66]:
```
con.close()
```

3.1.4 5.4 JSON Data

JSON (JavaScript Object Notation) is a popular data format used for representing structured data. It's common to transmit and receive data between a server and web application in JSON format.
JSON is built on two structures:

- A collection of name/value pairs. This is realized as an object, record, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. This is realized as an array, vector, list, or sequence.

[67]:
```
# creating a data frame
df = pd.DataFrame([['a', 'b'], ['c', 'd']],
                  index=['row 1', 'row 2'],
                  columns=['col 1', 'col 2'])
```

[68]:
```
df
```

[68]:

	col 1	col 2
row 1	a	b
row 2	c	d

[69]:
```
df.to_json?
```

Convert the object to a JSON string:

[70]:
```
df.to_json(orient='split')
'{"columns":["col 1","col 2"], "index":["row 1","row 2"], "data":
 ↪[["a","b"],["c","d"]]}'
```

[70]: '{"columns":["col 1","col 2"], "index":["row 1","row 2"], "data":[["a","b"],["c","d"]]}'

Encoding/decoding a Dataframe using 'split' formatted JSON

[71]:
```
pd.read_json?
```

[72]:
```
pd.read_json(_,orient='split') #input function
```

[72]:

	col 1	col 2
row 1	a	b
row 2	c	d

3.2 Resources:

- Book : Python for Data Analysis
- SQLite Tutorial :
  - http://www.sqlitetutorial.net,
  - https://www.sqlite.org/lang.html

Merge

```
import pandas as pd
# Create two sample DataFrames
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'ID': [2, 3, 4],
    'Age': [25, 30, 22]
})

# Merge DataFrames based on the 'ID' column
merged_df = pd.merge(df1, df2, on='ID', how='inner')

print("Merged DataFrame:")
print(merged_df)
```

OUTPUT:

```
Merged DataFrame:
   ID     Name  Age
0   2      Bob   25
1   3  Charlie   30
```

Combining DataFrame

```
import pandas as pd

# Create two sample DataFrames
df1 = pd.DataFrame({
```

```
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'ID': [4, 5, 6],
    'Name': ['David', 'Eva', 'Frank']
})

# Concatenate DataFrames along rows
concatenated_df = pd.concat([df1, df2], ignore_index=True)

print("Concatenated DataFrame along rows:")
print(concatenated_df)
```

OUTPUT:

```
Concatenated DataFrame along rows:
   ID     Name
0   1    Alice
1   2      Bob
2   3  Charlie
3   4    David
4   5      Eva
5   6    Frank
```

```
# Create two sample DataFrames
df1 = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})
```

```
df2 = pd.DataFrame({
    'Age': [25, 30, 22],
    'City': ['New York', 'San Francisco', 'Seattle']
})

# Concatenate DataFrames along columns
concatenated_df = pd.concat([df1, df2], axis=1)

print("Concatenated DataFrame along columns:")
print(concatenated_df)
```

OUTPUT:

```
Concatenated DataFrame along columns:
   ID     Name  Age           City
0   1    Alice   25       New York
1   2      Bob   30  San Francisco
2   3  Charlie   22        Seattle
```

Pivoting

```
import pandas as pd

# Create a sample DataFrame
data = {
    'Date': ['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02'],
    'Category': ['A', 'B', 'A', 'B'],
    'Sales': [100, 150, 200, 250]
}

df = pd.DataFrame(data)
```

```
# Display the original DataFrame
print("Original DataFrame:")
print(df)
```

OUTPUT:

```
Original DataFrame:
         Date Category  Sales
0  2023-01-01        A    100
1  2023-01-01        B    150
2  2023-01-02        A    200
3  2023-01-02        B    250
```

```
# Pivot the DataFrame
pivot_df = df.pivot(index='Date', columns='Category', values='Sales')

# Display the pivoted DataFrame
print("\nPivoted DataFrame:")
print(pivot_df)
```

OUTPUT:

```
Pivoted DataFrame:
Category      A    B
Date
2023-01-01  100  150
2023-01-02  200  250
```

Duplicates

```
import pandas as pd
# Create a sample DataFrame with duplicate values in the 'Name' column
data = {
    'ID': [1, 2, 3, 4, 5],
    'Name': ['Alice', 'Bob', 'Charlie', 'Bob', 'Alice'],
```

```
    'Age': [25, 30, 22, 30, 25]
}

df = pd.DataFrame(data)

# Display the original DataFrame
print("Original DataFrame:")
print(df)
```

OUTPUT:

```
Original DataFrame:
   ID     Name  Age
0   1    Alice   25
1   2      Bob   30
2   3  Charlie   22
3   4      Bob   30
4   5    Alice   25
```

```
# Identify duplicate rows based on the 'Name' column
duplicates = df.duplicated(subset='Name')

# Display the rows that are duplicates based on the 'Name' column
print("\nDuplicate Rows based on 'Name':")
print(df[duplicates])
```

OUTPUT:

```
Duplicate Rows based on 'Name':
   ID   Name  Age
3   4    Bob   30
4   5  Alice   25
```

```
# Drop duplicate rows based on the 'Name' column
```

df_no_duplicates = df.drop_duplicates(subset='Name')

# Display the DataFrame without duplicates based on the 'Name' column
print("\nDataFrame without Duplicates based on 'Name':")
print(df_no_duplicates)

OUTPUT:

```
DataFrame without Duplicates based on 'Name':
   ID     Name  Age
0   1    Alice   25
1   2      Bob   30
2   3  Charlie   22
```

Mapping

import pandas as pd

Create a sample DataFrame

data = {

  'Numbers': [1, 2, 3, 4, 5]

}

df = pd.DataFrame(data)

Display the original DataFrame

print("Original DataFrame:")

print(df)

OUTPUT:

```
Original DataFrame:
   Numbers
0        1
1        2
2        3
3        4
4        5
```

```
# Define a function to square a number
def square(x):
    return x ** 2

# Use the apply function to apply the square function to the 'Numbers' column
df['Squared'] = df['Numbers'].apply(square)

# Display the DataFrame with squared numbers
print("\nDataFrame with Squared Numbers:")
print(df)
```

OUTPUT:

```
DataFrame with Squared Numbers:
   Numbers  Squared
0        1        1
1        2        4
2        3        9
3        4       16
4        5       25
```

```
import pandas as pd

# Create a sample DataFrame with numerical data
data = {
    'Scores': [75, 82, 95, 68, 60, 90, 78, 88, 72, 85]
}

df = pd.DataFrame(data)

# Define bin edges and labels
bins = [0, 60, 70, 80, 90, 100]
```

labels = ['F', 'D', 'C', 'B', 'A']

Create a new column 'Grade' by binning the 'Scores' column

df['Grade'] = pd.cut(df['Scores'], bins=bins, labels=labels, right=False)

Display the original DataFrame and the DataFrame with the 'Grade' column

print("Original DataFrame:")

print(df)

OUTPUT:

```
Original DataFrame:
   Scores Grade
0      75     C
1      82     B
2      95     A
3      68     D
4      60     D
5      90     A
6      78     C
7      88     B
8      72     C
9      85     B
```

Outlier

import pandas as pd

from scipy.stats import zscore

Create a sample DataFrame with numerical data

data = {

  'Values': [10, 15, 12, 18, 22, 8, 25, 30, 5, 35, 40]

}

```
df = pd.DataFrame(data)

# Calculate Z-scores for each value in the 'Values' column
z_scores = zscore(df['Values'])

# Define a threshold for identifying outliers (e.g., Z-score greater than 3 or less than -3)
threshold = 3

# Identify outliers based on the threshold
outliers = (z_scores > threshold) | (z_scores < -threshold)

# Add a new column 'Is_Outlier' to the DataFrame indicating whether each value is an outlier
df['Is_Outlier'] = outliers
# Display the original DataFrame and the DataFrame with the 'Is_Outlier' column
print("Original DataFrame:")
print(df)
```

OUTPUT:

```
Original DataFrame:
    Values  Is_Outlier
0       10       False
1       15       False
2       12       False
3       18       False
4       22       False
5        8       False
6       25       False
7       30       False
8        5       False
9       35       False
10      40       False
```

WEEK-7

Working with Data PART-III

Group by on Data Frame

```
import pandas as pd

# Create a sample DataFrame
data = {
    'Category': ['A', 'B', 'A', 'B', 'A', 'B'],
    'Value': [10, 15, 20, 25, 30, 35]
}

df = pd.DataFrame(data)

# Group by 'Category' and calculate the mean for each group
grouped_df = df.groupby('Category').mean()

# Display the original DataFrame and the result of the groupby operation
print("Original DataFrame:")
print(df)
```

OUTPUT:

```
Original DataFrame:
  Category  Value
0        A     10
1        B     15
2        A     20
3        B     25
4        A     30
5        B     35
```

Splitting Applying and Combining

Splitting: The groupby('Category') operation is used to split the DataFrame into groups based on the 'Category' column.

Applying: The ['Sales'].sum() operation is applied to each group to calculate the sum of sales for each category.

Combining: The result is a Pandas Series that represents the total sales for each category.

```
import pandas as pd

# Create a sample DataFrame

data = {

    'Category': ['A', 'B', 'A', 'B', 'A', 'B'],

    'Sales': [100, 150, 200, 120, 180, 220]

}

df = pd.DataFrame(data)

# Step 1: Splitting - Group by 'Category'

grouped_df = df.groupby('Category')

# Step 2: Applying - Calculate the sum for each group

sum_per_category = grouped_df['Sales'].sum()

# Step 3: Combining - Display the result

print("Total Sales per Category:")

print(sum_per_category)
```

OUTPUT:

```
Total Sales per Category:
Category
A    480
B    490
Name: Sales, dtype: int64
```

Calculate multiple aggregations at once

result = grouped_df['Sales'].agg(['sum', 'mean', 'count'])

Display the result

print("Aggregations per Category:")

print(result)

OUTPUT:

```
Aggregations per Category:
          sum        mean  count
Category
A         480  160.000000      3
B         490  163.333333      3
```

Cross Tabulation

pd.crosstab() function to perform cross-tabulation on a DataFrame. Cross-tabulation is a way to summarize and analyze the relationship between two or more categorical variables.

import pandas as pd

Create a sample DataFrame

data = {

'Category': ['A', 'B', 'A', 'B', 'A', 'B'],

'Color': ['Red', 'Blue', 'Red', 'Green', 'Blue', 'Red']

}

df = pd.DataFrame(data)

Perform cross-tabulation

cross_tab = pd.crosstab(df['Category'], df['Color'])

Display the cross-tabulation result

print("Cross-Tabulation:")

print(cross_tab)

OUTPUT:

```
Cross-Tabulation:
Color      Blue  Green  Red
Category
A             1      0    2
B             1      1    1
```

Specify additional parameters for the crosstab function

cross_tab = pd.crosstab(df['Category'], df['Color'], margins=True, margins_name='Total')

Display the modified cross-tabulation result

print("Modified Cross-Tabulation:")

print(cross_tab)

OUTPUT:

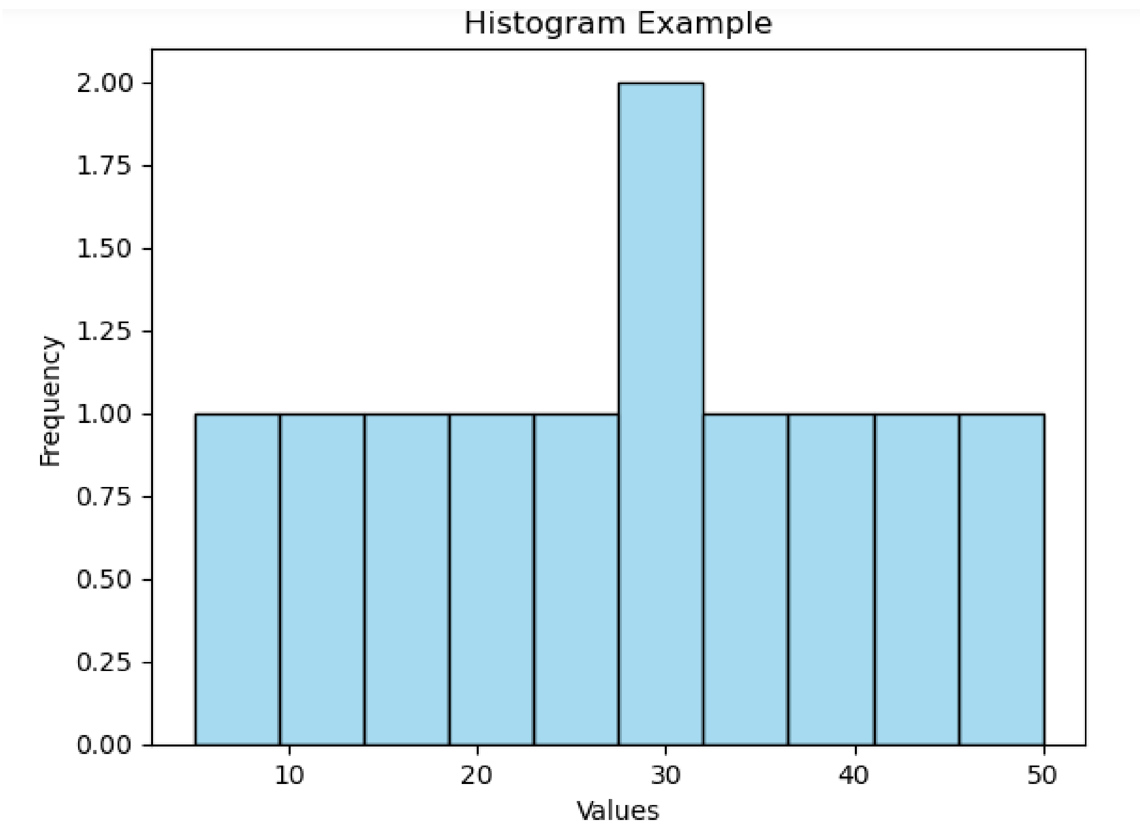
```
Modified Cross-Tabulation:
Color      Blue  Green  Red  Total
Category
A             1      0    2      3
B             1      1    1      3
Total         2      1    3      6
```

WEEK-8   DATA VISUALIZATION

Installing Seaborn

```
import seaborn as sns
import matplotlib.pyplot as plt
data = [5, 10, 15, 20, 25, 30, 30, 35, 40, 45, 50]
sns.histplot(data, bins=10, kde=False, color='skyblue')
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram Example')
plt.show()
```

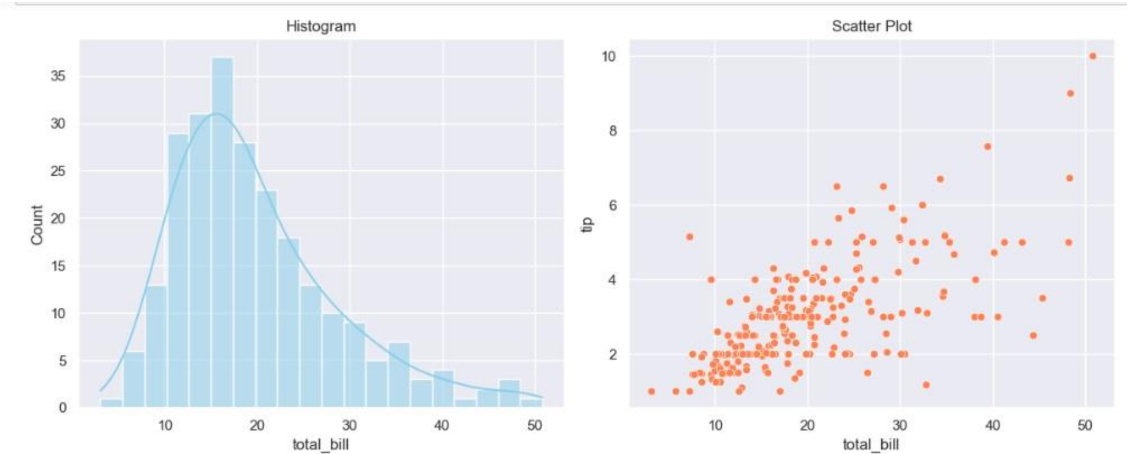
OUTPUT:



Combining Plot Styles

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

# Set the style to 'darkgrid'
sns.set(style='darkgrid')

# Sample data
data = sns.load_dataset('tips')

# Create a figure with two subplots
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 5))

# Plot a histogram on the first subplot
sns.histplot(data['total_bill'], bins=20, kde=True, color='skyblue',
ax=axes[0])
axes[0].set_title('Histogram')

# Plot a scatter plot on the second subplot
sns.scatterplot(x='total_bill', y='tip', data=data, color='coral', ax=axes[1])
axes[1].set_title('Scatter Plot')

# Adjust layout
plt.tight_layout()

# Show the combined plot
```

```
plt.show()
```

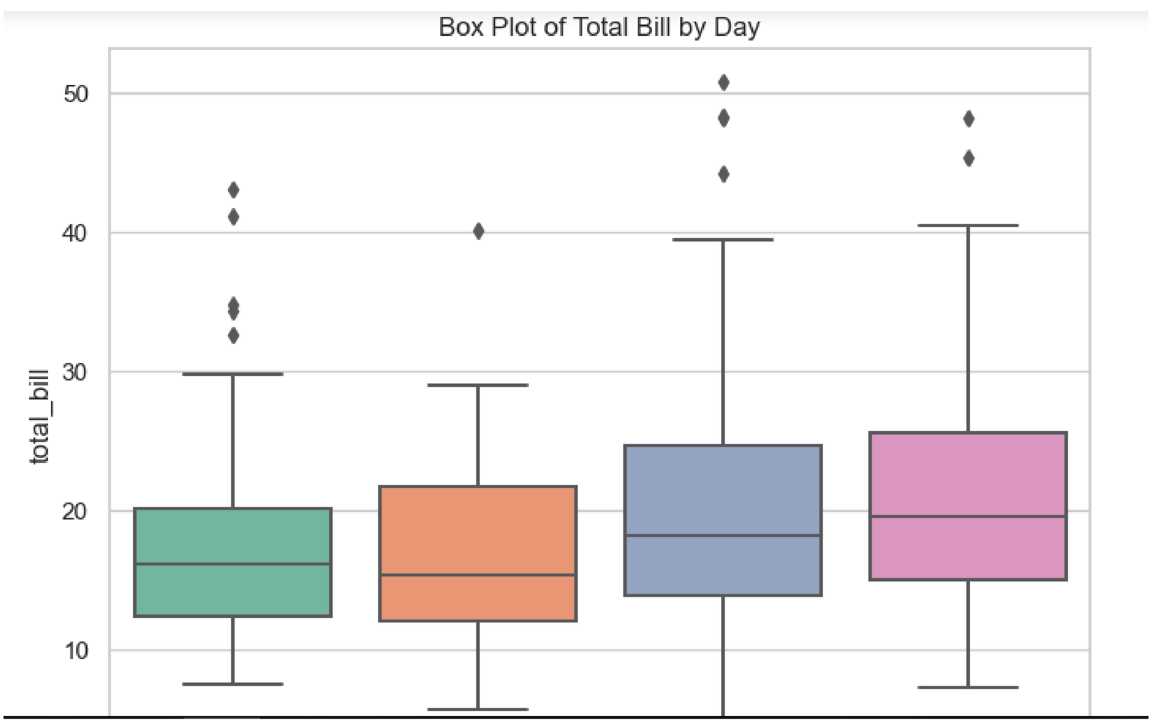


WEEK-9

Box and Violin Plots

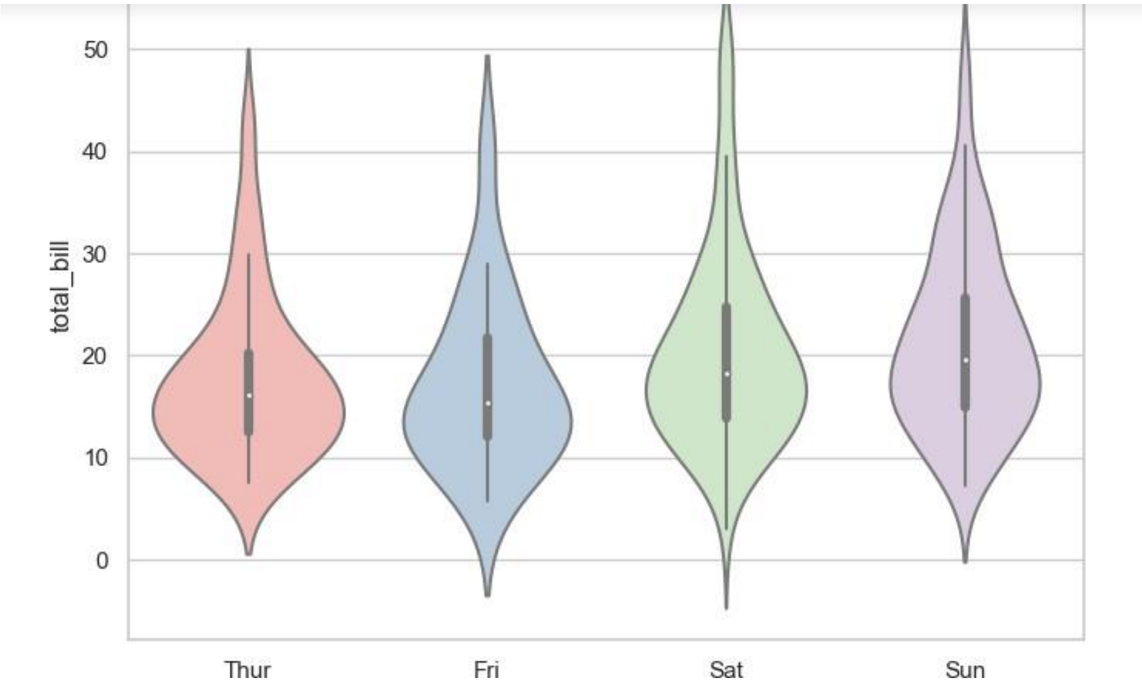
```
import seaborn as sns
import matplotlib.pyplot as plt
data = sns.load_dataset('tips')
sns.set(style='whitegrid')
plt.figure(figsize=(8, 6))
sns.boxplot(x='day', y='total_bill', data=data, palette='Set2')
plt.title('Box Plot of Total Bill by Day')
plt.show()
```

OUTPUT:



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

# Sample data
data = sns.load_dataset('tips')

# Set the style to 'whitegrid'
sns.set(style='whitegrid')

# Create a violin plot
plt.figure(figsize=(8, 6))
sns.violinplot(x='day', y='total_bill', data=data, palette='Pastel1')
plt.title('Violin Plot of Total Bill by Day')
plt.show()
```

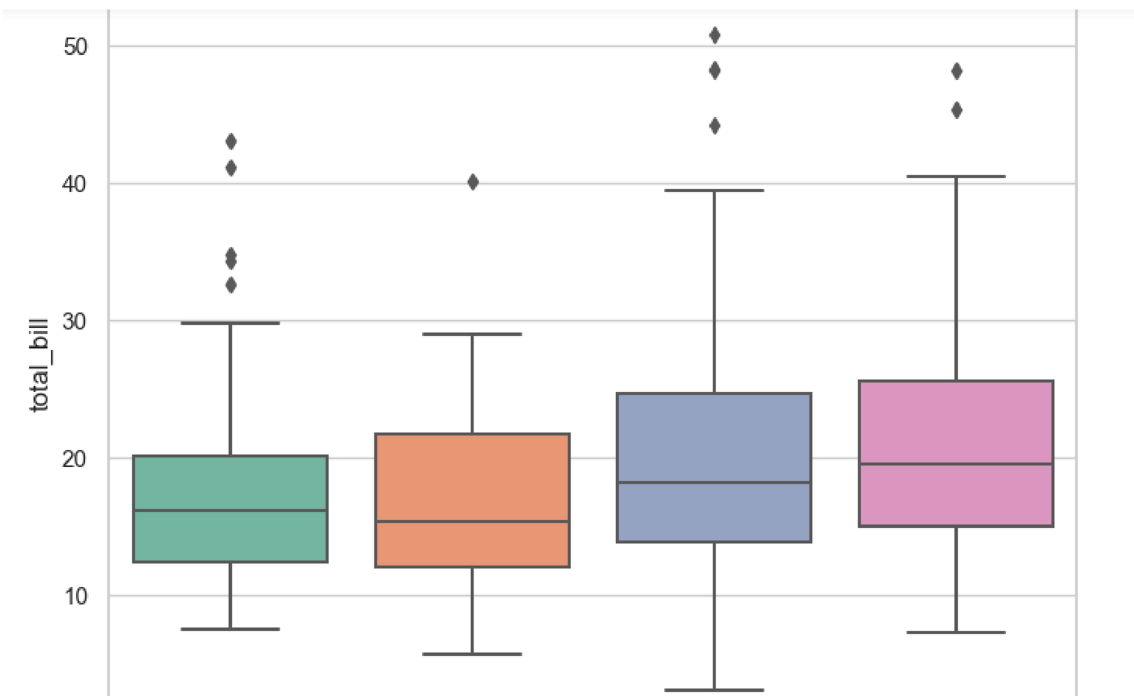
OUTPUT:



regression plots

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

# Sample data
data = sns.load_dataset('tips')
sns.set(style='whitegrid')
plt.figure(figsize=(8, 6))
sns.boxplot(x='day', y='total_bill', data=data, palette='Set2')
plt.title('Box Plot of Total Bill by Day')
plt.show()
```

OUTPUT:



import seaborn as sns

import matplotlib.pyplot as plt

Sample data

data = sns.load_dataset('tips')

Set the style to 'whitegrid'

sns.set(style='whitegrid')

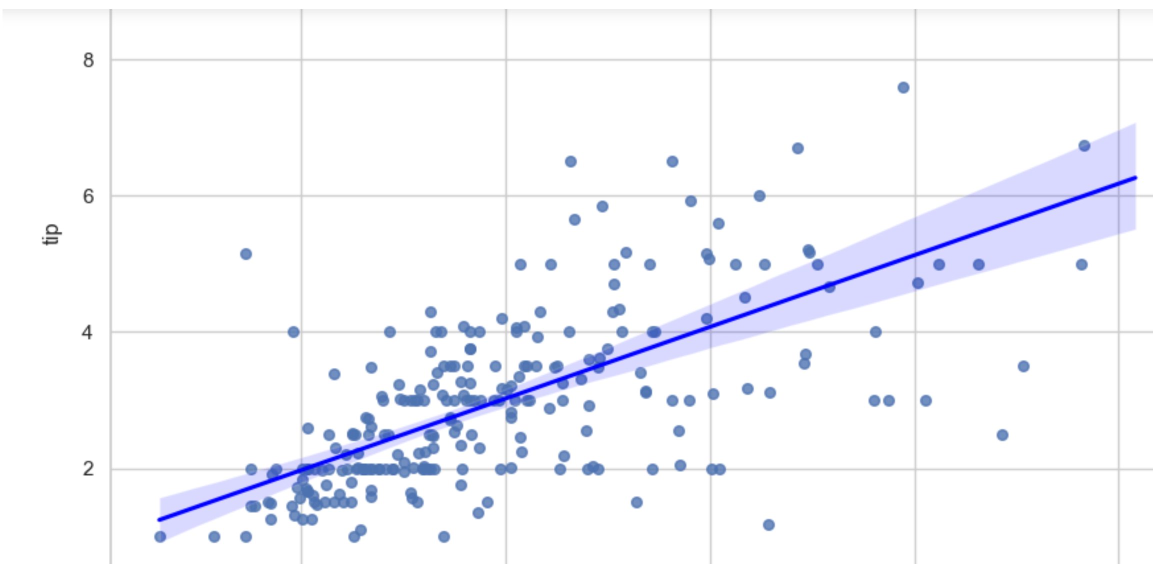
Create a scatter plot with a regression line using lmplot

sns.lmplot(x='total_bill', y='tip', data=data, height=6, aspect=1.5, scatter_kws={'s': 30}, line_kws={'color': 'blue'})

plt.title('Scatter Plot with Regression Line')

plt.show()

OUTPUT:



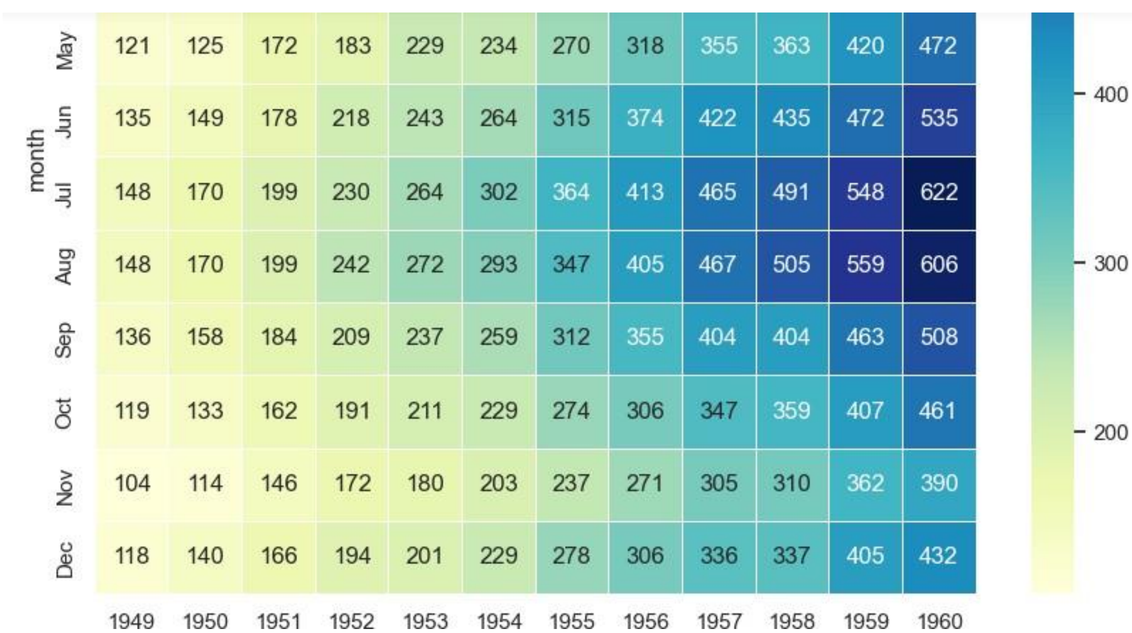
heatmaps and clustered matrice

import seaborn as sns

import matplotlib.pyplot as plt

```
# Sample data
data = sns.load_dataset('flights')

# Pivot the data to create a matrix
flights_matrix = data.pivot_table(index='month', columns='year', values='passengers')

# Set the style to 'whitegrid'
sns.set(style='whitegrid')
# Create a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(flights_matrix, cmap='YlGnBu', annot=True, fmt='d', linewidths=.5)
plt.title('Flights Data - Heatmap')
plt.show()
```

OUTPUT:



Stock market analysis ,customer segmentation , credit card fraud

detection

pip install yfinance pandas matplotlib

import yfinance as yf

import pandas as pd

import matplotlib.pyplot as plt

def stock_analysis(ticker, start_date, end_date):

    # Download historical stock data

    stock_data = yf.download(ticker, start=start_date, end=end_date)

    # Calculate daily returns

    stock_data['Daily_Return'] = stock_data['Adj Close'].pct_change()

    # Calculate cumulative returns

    stock_data['Cumulative_Return'] = (1 + stock_data['Daily_Return']).cumprod()

```
    # Plotting
    plt.figure(figsize=(10, 6))



# Plot stock prices
    plt.subplot(2, 1, 1)
    plt.plot(stock_data['Adj Close'])
    plt.title(f'{ticker} Stock Price')
    plt.xlabel('Date')
    plt.ylabel('Stock Price')

    # Plot cumulative returns
    plt.subplot(2, 1, 2)
    plt.plot(stock_data['Cumulative_Return'], color='r')
    plt.title(f'{ticker} Cumulative Returns')
    plt.xlabel('Date')
    plt.ylabel('Cumulative Returns')

    plt.tight_layout()
    plt.show()
```

```
if __name__ == "__main__":
    # Input stock symbol (ticker), start date, and end date
    stock_ticker = input("Enter stock symbol (e.g., AAPL): ")
    start_date = input("Enter start date (YYYY-MM-DD): ")
    end_date = input("Enter end date (YYYY-MM-DD): ")

    # Perform stock analysis
    stock_analysis(stock_ticker, start_date, end_date)
```

```
pip install pandas numpy matplotlib scikit-learn
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
```

```
# Generate some random customer data for demonstration purposes
np.random.seed(42)
data = {
    'Age': np.random.randint(18, 65, 100),
    'Income': np.random.randint(20000, 100000, 100),
}

df = pd.DataFrame(data)

# Standardize the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)

# Determine the optimal number of clusters using the Elbow Method
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)
    kmeans.fit(scaled_data)
```

```
    wcss.append(kmeans.inertia_)

# Plot the Elbow Method graph
plt.plot(range(1, 11), wcss)
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')  # Within-Cluster Sum of Squares
plt.show()

# Based on the Elbow Method, choose the optimal number of clusters (k)
k_optimal = int(input("Enter the optimal number of clusters (k): "))

# Apply k-means clustering
kmeans = KMeans(n_clusters=k_optimal, init='k-means++', max_iter=300, n_init=10, random_state=0)
df['Cluster'] = kmeans.fit_predict(scaled_data)

# Display the segmented customers
print("\nSegmented Customers:")
```

```
print(df.groupby('Cluster').mean())

# Visualize the clusters
plt.scatter(scaled_data[:, 0], scaled_data[:, 1],
c=df['Cluster'], cmap='viridis')
plt.scatter(kmeans.cluster_centers_[:, 0],
kmeans.cluster_centers_[:, 1], s=300, c='red')
plt.title('Customer Segmentation')
plt.xlabel('Scaled Age')
plt.ylabel('Scaled Income')
plt.show()
```

```
pip install pandas scikit-learn
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import LocalOutlierFactor
from sklearn.metrics import classification_report,
confusion_matrix
```

```
# Load the credit card fraud dataset (replace with your
dataset)
df = pd.read_csv('credit_card_fraud_dataset.csv')

# Separate features and labels
X = df.drop('Class', axis=1)
y = df['Class']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the Local Outlier Factor model
model = LocalOutlierFactor(contamination=0.01)
y_pred = model.fit_predict(X_test)
```

```
# Convert the predictions (-1 for anomalies, 1 for normal) to binary labels (0 for normal, 1 for fraud)
y_pred_binary = [1 if pred == -1 else 0 for pred in y_pred]

# Evaluate the model
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred_binary))
print("\nClassification Report:\n", classification_report(y_test, y_pred_binary))
```