Exercise 1: PRAACTICE PANDAS LIBRARY ON diabetes DATASET

Importing pandas

To begin working with pandas, import the pandas Python package as shown below. When importing pandas, the most common alias for pandas is pd.

```
import pandas as pd
```

Load Dataset

Mounting Google Drive

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

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
c:\Users\aadit\Desktop\BTech\S5\Foundations Of Data Science\LAB\lab 1\Untitled.ipynb
Cell 6 line 1
----> <a href='vscode-notebook-
cell:/c%3A/Users/aadit/Desktop/BTech/S5/Foundations%20Of%20Data%20Science/LAB/lab%201/Ur
line=0'>1</a> from google.colab import drive
      <a href='vscode-notebook-
cell:/c%3A/Users/aadit/Desktop/BTech/S5/Foundations%20Of%20Data%20Science/LAB/lab%201/Ur
line=1'>2</a> drive.mount('/content/drive')

ModuleNotFoundError: No module named 'google'
```

Importing csv files

Use read_csv() with the path to the CSV file to read a comma-separated values file.

```
df=pd.read_csv('diabetes.csv')
```

Importing Text Files

Reading text files is similar to CSV files. The only nuance is that you need to specify a separator with the sep argument, as shown below.

The separator argument refers to the symbol used to separate rows in a Df.

Comma (sep = ","), whitespace(sep = "\s"), tab (sep = "\t"), and colon(sep = ":") are the commonly used separators. Here \s represents a single white space character.

```
df=pd.read_csv('diabetes.txt',sep='\s')
```

Importing Excel files (single sheet)

Reading excel files (both XLS and XLSX) is as easy as the read_excel() function, using the file path as an input.

```
df=pd.read_excel('diabetes.xlsx')
```

Viewing and understanding Dfs using pandas

After reading tabular data as a Df, you would need to have a glimpse of the data. You can either view a small sample of the dataset or a summary of the data in the form of summary statistics.

How to view data using .head() and .tail()

You can view the first few or last few rows of a Df using the .head() or .tail() methods, respectively. You can specify the number of rows through the n argument (the default value is 5).

```
df.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu

```
df.tail(n=9)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
759	6	190	92	0	0	35.5	
760	2	88	58	26	16	28.4	
761	9	170	74	31	0	44.0	
762	9	89	62	0	0	22.5	
763	10	101	76	48	180	32.9	
764	2	122	70	27	0	36.8	
765	5	121	72	23	112	26.2	
766	1	126	60	0	0	30.1	
767	1	93	70	31	0	30.4	

Understanding data using .describe()

The .describe() method prints the summary statistics of all numeric columns, such as count, mean, standard deviation, range, and quartiles of numeric columns.

```
df.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Dia
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	

You can also modify the quartiles using the percentiles argument. Here, for example, we’re looking at the 30%, 50%, and 70% percentiles of the numeric columns in Df df

```
df.describe(percentiles=[0.3,0.5,0.7])
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Dia
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
30%	1.000000	102.000000	64.000000	8.200000	0.000000	28.200000	
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	
70%	5.000000	134.000000	78.000000	31.000000	106.000000	35.490000	
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	

You can isolate specific data types in your summary output by using the include argument. Here, for example, we’re only summarizing the columns with the integer data type.

```
df.describe(include=[int])
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	Age	(
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768
mean	3.845052	120.894531	69.105469	20.536458	79.799479	33.240885	0
std	3.369578	31.972618	19.355807	15.952218	115.244002	11.760232	0
min	0.000000	0.000000	0.000000	0.000000	0.000000	21.000000	0
25%	1.000000	99.000000	62.000000	0.000000	0.000000	24.000000	0
50%	3.000000	117.000000	72.000000	23.000000	30.500000	29.000000	0
75%	6.000000	140.250000	80.000000	32.000000	127.250000	41.000000	1
max	17.000000	199.000000	122.000000	99.000000	846.000000	81.000000	1

Try transposing them with the .T attribute.

```
df.describe().T
```

	count	mean	std	min	25%	50%	7
Pregnancies	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00
Glucose	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25
BloodPressure	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00
SkinThickness	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00
Insulin	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25
BMI	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60
DiabetesPedigreeFunction	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62
Age	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00
Outcome	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00

Understanding data using .info()

The .info() method is a quick way to look at the data types, missing values, and data size of a DataFrame.. When verbose is set to True, it prints the full summary from .info().

```
df.info(show_counts=True,memory_usage=True,verbose=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

Understanding your data using .shape

The number of rows and columns of a DataFrame can be identified using the .shape attribute of the DataFrame. It returns a tuple (row, column) and can be indexe

```
df.shape # Get the number of rows and columns
```

```
(768, 9)
```

```
df.shape[0] # Get the number of rows only
```

```
768
```

```
df.shape[1] # Get the number of columns onlyd to get only rows, and only columns count as ou
```

```
9
```

Get all columns and column names

Calling the .columns attribute of a DataFrame object returns the column names in the form of an Index object. As a reminder, a pandas index is the address/label of the row or column.

```
df.columns
```

```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

Checking for missing values in pandas with .isnull()

The sample DataFrame does not have any missing values. Let's introduce a few to make things interesting. The .copy() method makes a copy of the original DataFrame. This is done to ensure that any changes to the copy don’t reflect in the original DataFrame. Using .loc (to be discussed later), you can set rows two to five of the Pregnancies column to NaN values, which denote missing values.

```
df2 = df.copy()
```

```
df2.loc[2:5,'Pregnancies'] = None
df2.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
0	6.0	148	72	35	0	33.6	
1	1.0	85	66	29	0	26.6	

```
df2.head(7)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
0	6.0	148	72	35	0	33.6	
1	1.0	85	66	29	0	26.6	
2	NaN	183	64	0	0	23.3	
3	NaN	89	66	23	94	28.1	
4	NaN	137	40	35	168	43.1	
5	NaN	116	74	0	0	25.6	
6	3.0	78	50	32	88	31.0	

You can check whether each element in a DataFrame is missing using the .isnull() method.

```
df2.isnull().head(7)
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
0	False	False	False	False	False	False	
1	False	False	False	False	False	False	
2	True	False	False	False	False	False	
3	True	False	False	False	False	False	
4	True	False	False	False	False	False	
5	True	False	False	False	False	False	
6	False	False	False	False	False	False	

Given it's often more useful to know how much missing data you have, you can combine .isnull() with .sum() to count the number of nulls in each column.

```
df2.isnull().sum()
```

```
Pregnancies                 4
Glucose                     0
```

```
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

You can also do a double sum to get the total number of nulls in the DataFrame.

```
df2.isnull().sum().sum()
```

```
4
```

Slicing and Extracting Data in pandas

The pandas package offers several ways to subset, filter, and isolate data in your DataFrames. Here, we'll see the most common ways.

• Isolating one column using [ ]

You can isolate a single column using a square bracket [ ] with a column name in it. The output is a pandas Series object. A pandas Series is a one-dimensional array containing data of any type, including integer, float, string, boolean, python objects, etc. A DataFrame is comprised of many series that act as columns.

```
df['Outcome']
```

```
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

Isolating two or more columns using [[ ]]

You can also provide a list of column names inside the square brackets to fetch more than one column. Here, square brackets are used in two different ways. We use the outer square brackets to indicate a subset of a DataFrame, and the inner square brackets to create a list.

```
df[['Pregnancies', 'Outcome']]
```

	Pregnancies	Outcome
0	6	1
1	1	0
2	8	1
3	1	0
4	0	1
...	...	...
763	10	0
764	2	0
765	5	0
766	1	1
767	1	0

768 rows × 2 columns

Isolating one row using [ ]

A single row can be fetched by passing in a boolean series with one True value. In the example below, the second row with index = 1 is returned. Here, .index returns the row labels of the DataFrame, and the comparison turns that into a Boolean one-dimensional array.

```
df[df.index==1]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
1	1	85	66	29	0	26.6	

Isolating two or more rows using [ ]

Similarly, two or more rows can be returned using the .isin() method instead of a == operator.

```
df[df.index.isin(range(2,10))]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	
5	5	116	74	0	0	25.6	
6	3	78	50	32	88	31.0	
7	10	115	0	0	0	35.3	
8	2	197	70	45	543	30.5	
9	8	125	96	0	0	0.0	

Using .loc[] and .iloc[] to fetch rows

You can fetch specific rows by labels or conditions using .loc[] and .iloc[] ("location" and "integer location"). .loc[] uses a label to point to a row, column or cell, whereas .iloc[] uses the numeric position. To understand the difference between the two, let's modify the index of df2 created earlier.

```
df2.index = range(1,769)
```

The below example returns a pandas Series instead of a DataFrame. The 1 represents the row index (label), whereas the 1 in .iloc[] is the row position (first row).

```
df2.loc[1]
```

```
Pregnancies                   6.000
Glucose                     148.000
BloodPressure                72.000
SkinThickness                35.000
Insulin                       0.000
BMI                          33.600
DiabetesPedigreeFunction      0.627
Age                          50.000
Outcome                       1.000
Name: 1, dtype: float64
```

```
df2.iloc[1]
```

```
Pregnancies                  1.000
Glucose                     85.000
BloodPressure               66.000
SkinThickness               29.000
Insulin                      0.000
BMI                         26.600
DiabetesPedigreeFunction     0.351
Age                         31.000
Outcome                      0.000
Name: 2, dtype: float64
```

You can also fetch multiple rows by providing a range in square brackets.

```
df2.loc[100:110]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
100	1.0	122	90	51	220	49.7	
101	1.0	163	72	0	0	39.0	
102	1.0	151	60	0	0	26.1	
103	0.0	125	96	0	0	22.5	
104	1.0	81	72	18	40	26.6	
105	2.0	85	65	0	0	39.6	
106	1.0	126	56	29	152	28.7	
107	1.0	96	122	0	0	22.4	
108	4.0	144	58	28	140	29.5	
109	3.0	83	58	31	18	34.3	
110	0.0	95	85	25	36	37.4	

You can also subset with .loc[] and .iloc[] by using a list instead of a range.

```
df2.loc[[100, 200, 300]]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
100	1.0	122	90	51	220	49.7	
200	4.0	148	60	27	318	30.9	
300	8.0	112	72	0	0	23.6	

You can also select specific columns along with rows. This is where .iloc[] is different from .loc[] – it requires column location and not column labels.

```
df2.loc[100:110, ['Pregnancies', 'Glucose', 'BloodPressure']]
```

	Pregnancies	Glucose	BloodPressure
100	1.0	122	90
101	1.0	163	72
102	1.0	151	60
103	0.0	125	96
104	1.0	81	72
105	2.0	85	65
106	1.0	126	56
107	1.0	96	122
108	4.0	144	58
109	3.0	83	58
110	0.0	95	85

For faster workflows, you can pass in the starting index of a row as a range.

```
df2.loc[760:, ['Pregnancies', 'Glucose', 'BloodPressure']]
```

	Pregnancies	Glucose	BloodPressure
760	6.0	190	92
761	2.0	88	58
762	9.0	170	74
763	9.0	89	62
764	10.0	101	76
765	2.0	122	70
766	5.0	121	72
767	1.0	126	60
768	1.0	93	70

Conditional slicing (that fits certain conditions)

pandas lets you filter data by conditions over row/column values. For example, the below code selects the row where Blood Pressure is exactly 122. Here, we are isolating rows using the brackets [ ] as seen in previous sections. However, instead of inputting row indices or column names, we are inputting a condition where the column BloodPressure is equal to 122. We denote this condition using df.BloodPressure == 122.

```
df[df.BloodPressure == 122]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
106	1	96	122	0	0	22.4	

The below example fetched all rows where Outcome is 1. Here df.Outcome selects that column, df.Outcome == 1 returns a Series of Boolean values determining which Outcomes are equal to 1, then [] takes a subset of df where that Boolean Series is True.

```
df[df.Outcome == 1]
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigree
0	6	148	72	35	0	33.6	
2	8	183	64	0	0	23.3	
4	0	137	40	35	168	43.1	
6	3	78	50	32	88	31.0	
8	2	197	70	45	543	30.5	
...	...	...	...	...	...	...	
755	1	128	88	39	110	36.5	
757	0	123	72	0	0	36.3	
759	6	190	92	0	0	35.5	
761	9	170	74	31	0	44.0	
766	1	126	60	0	0	30.1	

268 rows × 9 columns

You can use a > operator to draw comparisons. The below code fetches Pregnancies, Glucose, and BloodPressure for all records with BloodPressure greater than 100.

```
df.loc[df['BloodPressure'] > 100, ['Pregnancies', 'Glucose', 'BloodPressure' ]]
```

	Pregnancies	Glucose	BloodPressure
43	9	171	110
84	5	137	108
106	1	96	122
177	0	129	110
207	5	162	104
362	5	103	108
369	1	133	102
440	0	189	104
549	4	189	110
658	11	127	106
662	8	167	106
672	10	68	106
691	13	158	114

Data analysis in pandas

The main value proposition of pandas lies in its quick data analysis functionality. In this section, we'll focus on a set of analysis techniques you can use in pandas.

Summary operators (mean, mode, median)

As you saw earlier, you can get the mean of each column value using the .mean() method.

```
df.mean()
```

```
Pregnancies                   3.845052
Glucose                     120.894531
BloodPressure                69.105469
SkinThickness                20.536458
```

```
Insulin                      79.799479
BMI                          31.992578
DiabetesPedigreeFunction      0.471876
Age                          33.240885
Outcome                       0.348958
dtype: float64
```

Create new columns based on existing columns

pandas provides fast and efficient computation by combining two or more columns like scalar variables. The below code divides each value in the column Glucose with the corresponding value in the Insulin column to compute a new column named Glucose_Insulin_Ratio.

```
df2['Glucose_Insulin_Ratio'] = df2['Glucose']/df2['Insulin']
```

```
df2.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
1	6.0	148	72	35	0	33.6	
2	1.0	85	66	29	0	26.6	
3	NaN	183	64	0	0	23.3	
4	NaN	89	66	23	94	28.1	
5	NaN	137	40	35	168	43.1	

Counting using .value_counts()

Often times you'll work with categorical values, and you'll want to count the number of observations each category has in a column. Category values can be counted using the .value_counts() methods. Here, for example, we are counting the number of observations where Outcome is diabetic (1) and the number of observations where the Outcome is nondiabetic (0).

```
df['Outcome'].value_counts()
```

```
Outcome
0    500
1    268
Name: count, dtype: int64
```

Aggregating data with .groupby() in pandas

pandas lets you aggregate values by grouping them by specific column values. You can do that by combining the .groupby() method with a summary method of your choice. The code below displays the mean of each numeric column grouped by Outcome.

```
df.groupby('Outcome').mean()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Di
Outcome							
0	3.298000	109.980000	68.184000	19.664000	68.792000	30.304200	
1	4.865672	141.257463	70.824627	22.164179	100.335821	35.142537	

.groupby() enables grouping by more than one column by passing a list of column names, as shown below

```
df.groupby(['Pregnancies', 'Outcome']).mean()
```

		Glucose	BloodPressure	SkinThickness	Insulin	BMI	Di
Pregnancies	**Outcome**						
0	**0**	111.945205	69.205479	21.054795	77.561644	31.727397	
	1	144.236842	63.210526	24.605263	89.578947	39.213158	
1	**0**	104.254717	66.830189	23.047170	84.320755	29.616038	
	1	143.793103	71.310345	29.517241	151.137931	37.793103	
2	**0**	105.214286	61.940476	20.107143	72.619048	29.679762	
	1	135.473684	69.052632	28.210526	144.315789	34.578947	
3	**0**	109.604167	65.708333	17.520833	62.020833	29.231250	
	1	148.444444	68.148148	24.629630	132.666667	32.548148	
4	**0**	117.555556	71.577778	18.422222	78.466667	31.255556	
	1	139.913043	67.000000	10.913043	51.782609	33.873913	
5	**0**	111.666667	74.666667	17.166667	46.861111	31.100000	
	1	131.190476	78.857143	17.761905	75.190476	36.780952	
6	**0**	115.352941	66.382353	18.705882	69.029412	29.591176	
	1	132.375000	72.750000	15.375000	52.000000	31.775000	
7	**0**	121.000000	70.350000	19.350000	72.500000	29.975000	
	1	148.800000	71.120000	21.040000	94.040000	34.756000	

Any summary method can be used alongside .groupby(), including .min(), .max(), .mean(), .median(), .sum(), .mode(), and more.

9 **0** 107.000000 70.400000 22.400000 71.200000 28.840000

Exercise 2: PRACTICE PANDAS EXERCISE FROM

https://www.w3resource.com/python-exercises/pandas/index.php

Exercise 3: Using Wine Quality dataset do the following:

1 116.750000 71.500000 30.250000 213.500000 34.575000

1. Copy the given winequality dataset to your local folder.

1 133.800000 73.200000 12.600000 5.800000 36.720000

2. Load the winequality dataset using pandas

```
import pandas as pd
df = pd.read_csv('WineQT.csv')
```

3. Find the size of the dataset

```
df.shape
```

```
(1143, 13)
```

4. Get the statistical summary of the data

```
df.describe()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	t su di
count	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.000000	1143.00
mean	8.311111	0.531339	0.268364	2.532152	0.086933	15.615486	45.9
std	1.747595	0.179633	0.196686	1.355917	0.047267	10.250486	32.7
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.00
25%	7.100000	0.392500	0.090000	1.900000	0.070000	7.000000	21.00
50%	7.900000	0.520000	0.250000	2.200000	0.079000	13.000000	37.00
75%	9.100000	0.640000	0.420000	2.600000	0.090000	21.000000	61.00

5. Print the first and last five rows

```
df.head()
```

free total

```
df.tail()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	su
1138	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	
1139	6.8	0.620	0.08	1.9	0.068	28.0	38.0	0.99651	3.42	
1140	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	
1141	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	

6. Print the first 7 rows of the dataset.

```
df.head(7)
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulph
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	
5	7.4	0.66	0.00	1.8	0.075	13.0	40.0	0.9978	3.51	

7. Display the first 7 rows of 4th column to 6th column.

```
df.head(7).T.head(6).tail(3).T
```

	residual sugar	chlorides	free sulfur dioxide
0	1.9	0.076	11.0

8. Select all the rows of the 3rd and 6th column

```
df[['citric acid', 'free sulfur dioxide' ]]
```

	citric acid	free sulfur dioxide
0	0.00	11.0
1	0.00	25.0
2	0.04	15.0
3	0.56	17.0
4	0.00	11.0
...	...	...
1138	0.13	29.0
1139	0.08	28.0
1140	0.08	32.0
1141	0.10	39.0
1142	0.12	32.0

1143 rows × 2 columns

9. Check the number of null and non-null values in the dataset columns wise.

```
df.isnull().sum()
```

```
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
pH                      0
sulphates               0
alcohol                 0
quality                 0
Id                      0
dtype: int64
```

```
df.notnull().sum()
```

```
fixed acidity           1143
volatile acidity        1143
citric acid             1143
residual sugar          1143
chlorides               1143
free sulfur dioxide     1143
total sulfur dioxide    1143
density                 1143
pH                      1143
sulphates               1143
alcohol                 1143
quality                 1143
Id                      1143
dtype: int64
```

10. Rename the ph column as potential of hydrogen

```
df.rename(columns={'pH':'potential of hydrogen'}, inplace=True)
```

11. Create a new column named total_free ratio which is the ratio of the total sulphur dioxide and free sulphurdioxide

```
df['total_free ratio']=df['free sulfur dioxide']/df['total sulfur dioxide']
```

```
df.head()
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	potential of hydrogen
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16

12. Aggregate the data based on quality

```
df.describe()['quality']
```

```
count    1143.000000
mean        5.657043
std         0.805824
min         3.000000
25%         5.000000
50%         6.000000
75%         6.000000
max         8.000000
Name: quality, dtype: float64
```

13. Slice the data frame and create a new one such that the rows containing total_free ratio values less than 2.7 and greater than 3.2

```
df2 = df.copy()
df2 = df2[(df2['total_free ratio'] < 2.7) | (df2['total_free ratio'] > 3.2)]
```

14. Drop second column values from the dataset

```
df2.drop(['volatile acidity'], inplace=True, axis=1)
```

```
df2
```

	fixed acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	potential of hydrogen	sulpha
0	7.4	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	C
1	7.8	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	C
2	7.8	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	C
3	11.2	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	C
4	7.4	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	C
...	...	...	...	...	...	...	...	...	
1138	6.3	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	C
1139	6.8	0.08	1.9	0.068	28.0	38.0	0.99651	3.42	C
1140	6.2	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	C
1141	5.9	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	C
1142	5.9	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	C

15. Sort the DataFrame first by '' in descending order, then by '' in ascending order

16. Find the sum and the cumulative sum of second attribute

```
sum_second_attribute = df['citric acid'].sum()
cumulative_sum_second_attribute = df['citric acid'].cumsum()

print(sum_second_attribute, cumulative_sum_second_attribute)
```

```
306.74 0         0.00
1         0.00
2         0.04
3         0.60
4         0.60
         ...
1138    306.36
1139    306.44
1140    306.52
1141    306.62
1142    306.74
Name: citric acid, Length: 1143, dtype: float64
```

17. Find the minimum and maximum of third attribute

```
minimum = df['residual sugar'].min()
maximum = df['residual sugar'].max()

print("Minimum:", minimum)
print("Maximum:", maximum)
```

```
Minimum: 0.9
Maximum: 15.5
```

18. Rename columns of a DataFrame

```
df.rename(columns={'potential of hydrogen': 'ph'}, inplace=True)
df.columns
```

```
Index(['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'ph', 'sulphates', 'alcohol', 'quality', 'Id', 'total_free ratio'],
      dtype='object')
```

19. Change the order of a DataFrame columns.

```
df = df[['Id', 'fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',
       'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',
       'ph', 'sulphates', 'alcohol', 'quality', 'total_free ratio']]
```

20. Write a DataFrame to CSV file using tab separator.

```
df.to_csv('output_file.csv', sep='\t', index=False)
```

21. Replace all the NaN values with Zero's in a column of a dataframe

```
df.isnull().sum()
```

```
Id                      0
fixed acidity           0
volatile acidity        0
citric acid             0
residual sugar          0
chlorides               0
free sulfur dioxide     0
total sulfur dioxide    0
density                 0
ph                      0
sulphates               0
alcohol                 0
quality                 0
total_free ratio        0
dtype: int64
```

```
df['fixed acidity'].fillna(0, inplace=True)
```

22. Divide a data frame in 60:40 ratio.

```
from sklearn.model_selection import train_test_split

train_df, test_df = train_test_split(df, test_size=0.4, random_state=42)

print("Train set shape:", train_df.shape)
print("Test set shape:", test_df.shape)
```

```
Train set shape: (685, 14)
Test set shape: (458, 14)
```

23. Combine two series into a DataFrame.

```
import pandas as pd

series1 = pd.Series([1, 2, 3])
series2 = pd.Series([4, 5, 6])

df = pd.concat([series1, series2], axis=1)

print(df)
```

```
   0  1
0  1  4
1  2  5
2  3  6
```

24. Find the row for where the value of a first attribute is maximum.

```
max_row = df[df['residual sugar'] == df['residual sugar'].max()]
print(max_row)
```

```
     fixed acidity  volatile acidity  citric acid  residual sugar  chlorides  \
339           10.6              0.28         0.39            15.5      0.069

     free sulfur dioxide  total sulfur dioxide  density    pH  sulphates  \
339                  6.0                  23.0   1.0026  3.12       0.66

     alcohol  quality   Id
339      9.2        5  480
```

25. Get the datatypes of columns of a DataFrame.

```
df.dtypes
```

```
fixed acidity           float64
volatile acidity        float64
citric acid             float64
residual sugar          float64
chlorides               float64
free sulfur dioxide     float64
total sulfur dioxide    float64
density                 float64
pH                      float64
sulphates               float64
alcohol                 float64
quality                   int64
Id                        int64
dtype: object
```