Statistics Assignment

Sireesha

In [205]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statistics as stat
import random
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:

```
# loading Diamond dataset
```

In [207]:

```
Data= pd.read_csv(r"C:\Users\kastu\OneDrive\Desktop\Diamond_dataset.csv")
Data
```

Out[207]:

	carat	cut	color	clarity	depth	table	weight	size	price
0	0.23	Ideal	E	SI2	61.5	55.0	3.95	3.98	326
1	0.21	Premium	E	SI1	59.8	61.0	3.89	3.84	326
2	0.23	Good	E	VS1	56.9	65.0	4.05	4.07	327
3	0.29	Premium	I	VS2	62.4	58.0	4.20	4.23	334
4	0.31	Good	J	SI2	63.3	58.0	4.34	4.35	335
...	...	...	...	...	...	...	...	...	...
53935	0.72	Ideal	D	SI1	60.8	57.0	5.75	5.76	2757
53936	0.72	Good	D	SI1	63.1	55.0	5.69	5.75	2757
53937	0.70	Very Good	D	SI1	62.8	60.0	5.66	5.68	2757
53938	0.86	Premium	H	SI2	61.0	58.0	6.15	6.12	2757
53939	0.75	Ideal	D	SI2	62.2	55.0	5.83	5.87	2757

53940 rows × 9 columns

In [5]:

```
# Cheking for data types
```

In [209]:

```
Data.dtypes
```

Out[209]:

```
carat      float64
cut         object
```

```
color       object
clarity     object
depth      float64
table      float64
weight     float64
size       float64
price        int64
dtype: object
```

A. Create 2 dataframes out of this dataframe – 1 with all numerical variables and other with all categorical variables.

In [ ]:

```
# column names
```

In [211]:

```
Data.columns
```

Out[211]:

```
Index(['carat', 'cut', 'color', 'clarity', 'depth', 'table', 'weight', 'size',
       'price'],
      dtype='object')
```

In [ ]:

```
# extracting numerical columns
```

In [217]:

```
num_col=[fea for fea in Data.columns if Data[fea].dtypes!='object']
num_col
```

Out[217]:

```
['carat', 'depth', 'table', 'weight', 'size', 'price']
```

In [219]:

```
Data_numerical=Data[num_col]
Data_numerical
```

Out[219]:

	carat	depth	table	weight	size	price
0	0.23	61.5	55.0	3.95	3.98	326
1	0.21	59.8	61.0	3.89	3.84	326
2	0.23	56.9	65.0	4.05	4.07	327
3	0.29	62.4	58.0	4.20	4.23	334
4	0.31	63.3	58.0	4.34	4.35	335
...	...	...	...	...	...	...
53935	0.72	60.8	57.0	5.75	5.76	2757
53936	0.72	63.1	55.0	5.69	5.75	2757
53937	0.70	62.8	60.0	5.66	5.68	2757
53938	0.86	61.0	58.0	6.15	6.12	2757
53939	0.75	62.2	55.0	5.83	5.87	2757

53940 rows × 6 columns

In [ ]:

```
# extracting categorical columns
```

In [ ]:

```

```

In [213]:

```
cat_col=[fea for fea in Data.columns if Data[fea].dtypes=='object']
cat_col
```

Out[213]:

```
['cut', 'color', 'clarity']
```

In [215]:

```
Data_cat=Data[cat_col]
Data_cat
```

Out[215]:

	cut	color	clarity
0	Ideal	E	SI2
1	Premium	E	SI1
2	Good	E	VS1
3	Premium	I	VS2
4	Good	J	SI2
...	...	...	...
53935	Ideal	D	SI1
53936	Good	D	SI1
53937	Very Good	D	SI1
53938	Premium	H	SI2
53939	Ideal	D	SI2

53940 rows × 3 columns

B. Calculate the measure of central tendency of numerical variables using Pandas and statistics libraries and check if the calculated values are different between these 2 libraries.

In [ ]:

```
# describing statistical summary
```

In [17]:

```
mean_pandas=Data_numerical.describe()
mean_pandas
```

Out[17]:

	carat	depth	table	weight	size	price
count	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000	53940.000000
mean	0.797940	61.749405	57.457184	5.731157	5.734526	3932.799722
std	0.474011	1.432621	2.234491	1.121761	1.142135	3989.439738
min	0.200000	43.000000	43.000000	0.000000	0.000000	326.000000

	carat	depth	table	weight	size	price
25%	0.400000	61.000000	56.000000	4.710000	4.720000	950.000000
50%	0.700000	61.800000	57.000000	5.700000	5.710000	2401.000000
75%	1.040000	62.500000	59.000000	6.540000	6.540000	5324.250000
max	5.010000	79.000000	95.000000	10.740000	58.900000	18823.000000

In [43]:

```
# mean using pandas library
```

In [21]:

```
Pandas_mean=Data_numerical.mean()
Pandas_mean
```

Out[21]:

```
carat        0.797940
depth       61.749405
table       57.457184
weight       5.731157
size         5.734526
price     3932.799722
dtype: float64
```

In [47]:

```
# mean using statistical library
```

In [23]:

```
stat_mean=np.mean(Data_numerical,axis=0)
stat_mean
```

Out[23]:

```
carat        0.797940
depth       61.749405
table       57.457184
weight       5.731157
size         5.734526
price     3932.799722
dtype: float64
```

In [51]:

```
# median using pandas library
```

In [53]:

```
Pd_median=Data_numerical.median()
Pd_median
```

Out[53]:

```
carat        0.70
depth       61.80
table       57.00
weight       5.70
size         5.71
price     2401.00
dtype: float64
```

In [55]:

```
# median using statistical library
```

In [57]:

```
stat_median=np.median(Data_numerical, axis=0)
stat_median
```

Out[57]:

```
array([7.000e-01, 6.180e+01, 5.700e+01, 5.700e+00, 5.710e+00, 2.401e+03])
```

In [59]:

```
# mode of pandas
```

In [61]:

```
Data_numerical.mode()
```

Out[61]:

	carat	depth	table	weight	size	price
0	0.3	62.0	56.0	4.37	4.34	605

In [63]:

```
# checking for mode of table column to compare with pandas data
```

In [65]:

```
stat.mode(Data_numerical['table'])
```

Out[65]:

```
56.0
```

The values of Central Tendency of pandas and statistics are same.

C. Check the skewness of all numeric variables. Mention against each variable if its highly skewed/light skewed/ Moderately skewed.

In [68]:

```
original_skewness = Data_numerical[['carat', 'depth', 'table', 'weight', 'size', 'price'
print("Original Skewness:")
print(original_skewness)
```

```
Original Skewness:
carat     1.116646
depth    -0.082294
table     0.796896
weight    0.378676
size      2.434167
price     1.618395
dtype: float64
```

Here carat,size,price are highly skewed

depth and weight are lightly skewed

table is moderatly skewed.

carat,size,price,table are positive right skewed

depth is negative left skewed

D. Use the different transformation techniques to convert skewed data found in previous question into normal distribution.

In [71]:

```
# log transformation for highly skewed data
```

In [73]:

```
Data_numerical['log_carat'] = np.log(Data_numerical['carat'])
Data_numerical['log_carat']
```

Out[73]:

```
0       -1.469676
1       -1.560648
2       -1.469676
3       -1.237874
4       -1.171183
           ...
53935   -0.328504
53936   -0.328504
53937   -0.356675
53938   -0.150823
53939   -0.287682
Name: log_carat, Length: 53940, dtype: float64
```

In [75]:

```
Data_numerical['log_size'] = np.log(Data_numerical['size'])
Data_numerical['log_size']
```

Out[75]:

```
0        1.381282
1        1.345472
2        1.403643
3        1.442202
4        1.470176
           ...
53935    1.750937
53936    1.749200
53937    1.736951
53938    1.811562
53939    1.769855
Name: log_size, Length: 53940, dtype: float64
```

In [25]:

```
Data_numerical['log_price'] = np.log(Data_numerical['price'])
Data_numerical['log_price']
```

Out[25]:

```
0        5.786897
1        5.786897
2        5.789960
3        5.811141
4        5.814131
           ...
53935    7.921898
53936    7.921898
53937    7.921898
53938    7.921898
53939    7.921898
Name: log_price, Length: 53940, dtype: float64
```

In [79]:

```
# square root transformation for Moderately right-skewed data
```

In [27]:

```
Data_numerical['sqrt_table'] = np.sqrt(Data_numerical['table'])
Data_numerical['sqrt_table']
```

Out[27]:

```
0        7.416198
1        7.810250
2        8.062258
3        7.615773
4        7.615773
           ...
53935    7.549834
53936    7.416198
53937    7.745967
53938    7.615773
53939    7.416198
Name: sqrt_table, Length: 53940, dtype: float64
```

In [83]:

```
# cube root transformation for lightly or negetively skewed data
```

In [29]:

```
Data_numerical['cbrt_weight'] = np.cbrt(Data_numerical['weight'])
Data_numerical['cbrt_weight']
```

Out[29]:

```
0        1.580759
1        1.572714
2        1.593988
3        1.613429
4        1.631160
           ...
53935    1.791524
53936    1.785271
53937    1.782128
53938    1.832139
53939    1.799794
Name: cbrt_weight, Length: 53940, dtype: float64
```

In [31]:

```
Data_numerical['cbrt_depth'] = np.cbrt(Data_numerical['depth'])
Data_numerical['cbrt_depth']
```

Out[31]:

```
0        3.947223
1        3.910513
2        3.846249
3        3.966385
4        3.985363
           ...
53935    3.932190
53936    3.981161
53937    3.974842
53938    3.936497
53939    3.962143
Name: cbrt_depth, Length: 53940, dtype: float64
```

In [89]:

```
# checking for skewness after transformation
```

In [ ]:

```
transformed_skewness = Data_numerical[['log_carat', 'cbrt_depth', 'sqrt_table', 'cbrt_we
print(transformed_skewness)
```

E.Create a user defined function in python to check the outliers using IQR method. Then pass all numeric variables in that function to check outliers.

In [93]:

```
Data_numerical
```

Out[93]:

	carat	depth	table	weight	size	price	log_carat	log_size	log_price	sqrt_table	cbrt_weight
0	0.23	61.5	55.0	3.95	3.98	326	-1.469676	1.381282	5.786897	7.416198	1.580759
1	0.21	59.8	61.0	3.89	3.84	326	-1.560648	1.345472	5.786897	7.810250	1.572714
2	0.23	56.9	65.0	4.05	4.07	327	-1.469676	1.403643	5.789960	8.062258	1.593988
3	0.29	62.4	58.0	4.20	4.23	334	-1.237874	1.442202	5.811141	7.615773	1.613429
4	0.31	63.3	58.0	4.34	4.35	335	-1.171183	1.470176	5.814131	7.615773	1.631160
...	...	...	...	...	...	...	...	...	...	...	...
53935	0.72	60.8	57.0	5.75	5.76	2757	-0.328504	1.750937	7.921898	7.549834	1.791524
53936	0.72	63.1	55.0	5.69	5.75	2757	-0.328504	1.749200	7.921898	7.416198	1.785271
53937	0.70	62.8	60.0	5.66	5.68	2757	-0.356675	1.736951	7.921898	7.745967	1.782128
53938	0.86	61.0	58.0	6.15	6.12	2757	-0.150823	1.811562	7.921898	7.615773	1.832139
53939	0.75	62.2	55.0	5.83	5.87	2757	-0.287682	1.769855	7.921898	7.416198	1.799794

53940 rows × 12 columns

In [95]:

```
Data_numerical.isnull().sum()
```

Out[95]:

```
carat          0
depth          0
table          0
weight         0
size           0
price          0
log_carat      0
log_size       0
log_price      0
sqrt_table     0
cbrt_weight    0
cbrt_depth     0
dtype: int64
```

In [35]:

```
num_col=[fea for fea in Data.columns if Data[fea].dtypes!='object']
num_col
```

Out[35]:

```
['carat', 'depth', 'table', 'weight', 'size', 'price']
```

In [262]:

```python
def outliers(Data, cols):
    outlier_dict = {}

    for col in cols:
        Q1 = Data[col].quantile(0.25)
        Q3 = Data[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        outlier_values = Data.loc[(Data[col] < lower_bound) | (Data[col] > upper_bound),

        outlier_dict[col] = [Q1, Q3, IQR, outlier_values.values]

    return outlier_dict

outlier = outliers(Data, num_col)
df = pd.DataFrame(outlier).set_index(pd.Index(['Q1', 'Q3', 'IQR', 'outlier']))
print(df)
```

```
                                                     carat  \
Q1                                                     0.4
Q3                                                    1.04
IQR                                                   0.64
outlier  [2.06, 2.14, 2.15, 2.22, 2.01, 2.01, 2.27, 2.0...

                                                     depth  \
Q1                                                    61.0
Q3                                                    62.5
IQR                                                    1.5
outlier  [56.9, 65.1, 58.1, 58.2, 65.2, 58.4, 57.9, 55....

                                                     table  \
Q1                                                    56.0
Q3                                                    59.0
IQR                                                    3.0
outlier  [65.0, 69.0, 64.0, 64.0, 67.0, 64.0, 66.0, 70....

                                                    weight  \
Q1                                                    4.71
Q3                                                    6.54
IQR                                                   1.83
outlier  [0.0, 0.0, 0.0, 9.54, 9.38, 9.53, 9.44, 9.49, ...

                                                      size  \
Q1                                                    4.72
Q3                                                    6.54
IQR                                                   1.82
outlier  [0.0, 0.0, 9.38, 9.31, 9.48, 58.9, 9.4, 9.42, ...

                                                     price
Q1                                                   950.0
```

```
Q3                                          5324.25
IQR                                         4374.25
outlier  [11886, 11886, 11888, 11888, 11888, 11897, 118...
```

In [268]:

```
df.size
```

Out[268]:

```
24
```

F. Convert categorical variables into numerical variables using LabelEncoder technique.

In [125]:

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
le.fit(Data_cat['cut'])
list(le.classes_)
```

Out[125]:

```
['Fair', 'Good', 'Ideal', 'Premium', 'Very Good']
```

In [127]:

```
le.transform(Data_cat['cut'])
```

Out[127]:

```
array([2, 3, 1, ..., 4, 3, 2])
```

In [129]:

```
le = LabelEncoder()
le.fit(Data_cat['color'])
list(le.classes_)
le.transform(Data_cat['color'])
```

Out[129]:

```
array([1, 1, 1, ..., 0, 4, 0])
```

In [131]:

```
le = LabelEncoder()
le.fit(Data_cat['clarity'])
list(le.classes_)
le.transform(Data_cat['clarity'])
```

Out[131]:

```
array([3, 2, 4, ..., 2, 3, 3])
```

G. Use both the feature scaling techniques (standardscaler/min max scaler) on all the variables.

In [47]:

```
Data_standard = (Data_numerical - Data_numerical.mean())/Data_numerical.std()
Data_standard.head()
```

Out[47]:

	carat	depth	table	weight	size	price
0	-1.198157	-0.174090	-1.099662	-1.587823	-1.536181	-0.904087
1	-1.240350	-1.360726	1.585514	-1.641310	-1.658759	-0.904087
2	-1.198157	-3.384987	3.375631	-1.498677	-1.457382	-0.903836
3	-1.071577	0.454129	0.242926	-1.364959	-1.317293	-0.902081
4	-1.029384	1.082348	0.242926	-1.240155	-1.212227	-0.901831

In [41]:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_Data = scaler.fit_transform(Data_numerical)
scaled_Data
```

Out[41]:

```
array([[-1.19816781, -0.17409151, -1.09967199, -1.58783745, -1.53619556,
        -0.90409516],
       [-1.24036129, -1.36073849,  1.58552871, -1.64132529, -1.65877419,
        -0.90409516],
       [-1.19816781, -3.38501862,  3.37566251, -1.49869105, -1.45739502,
        -0.9038445 ],
       ...,
       [-0.20662095,  0.73334442,  1.13799526, -0.06343409, -0.04774083,
        -0.29473076],
       [ 0.13092691, -0.52310533,  0.24292836,  0.37338325,  0.33750627,
        -0.29473076],
       [-0.10113725,  0.31452784, -1.09967199,  0.08811478,  0.11861587,
        -0.29473076]])
```

In [43]:

```
scaled_Data = pd.DataFrame(scaled_Data, columns=Data_numerical.columns)
scaled_Data
```

Out[43]:

	carat	depth	table	weight	size	price
0	-1.198168	-0.174092	-1.099672	-1.587837	-1.536196	-0.904095
1	-1.240361	-1.360738	1.585529	-1.641325	-1.658774	-0.904095
2	-1.198168	-3.385019	3.375663	-1.498691	-1.457395	-0.903844
3	-1.071587	0.454133	0.242928	-1.364971	-1.317305	-0.902090
4	-1.029394	1.082358	0.242928	-1.240167	-1.212238	-0.901839
...	...	...	...	...	...	...
53935	-0.164427	-0.662711	-0.204605	0.016798	0.022304	-0.294731
53936	-0.164427	0.942753	-1.099672	-0.036690	0.013548	-0.294731
53937	-0.206621	0.733344	1.137995	-0.063434	-0.047741	-0.294731
53938	0.130927	-0.523105	0.242928	0.373383	0.337506	-0.294731
53939	-0.101137	0.314528	-1.099672	0.088115	0.118616	-0.294731

53940 rows × 6 columns

In [49]:

```
round(Data_standard.mean())
```

Out[49]:

```
carat     0.0
depth    -0.0
table     0.0
weight    0.0
size     -0.0
price    -0.0
dtype: float64
```

In [51]:

```
Data_standard.std()
```

Out[51]:

```
carat     1.0
depth     1.0
table     1.0
weight    1.0
size      1.0
price     1.0
dtype: float64
```

In [53]:

```
Data_numerical.head()
```

Out[53]:

	carat	depth	table	weight	size	price
0	0.23	61.5	55.0	3.95	3.98	326
1	0.21	59.8	61.0	3.89	3.84	326
2	0.23	56.9	65.0	4.05	4.07	327
3	0.29	62.4	58.0	4.20	4.23	334
4	0.31	63.3	58.0	4.34	4.35	335

In [55]:

```
Data_normal = (Data_numerical - Data_numerical.min())/(Data_numerical.max() - Data_numer
Data_normal.head()
```

Out[55]:

	carat	depth	table	weight	size	price
0	0.006237	0.513889	0.230769	0.367784	0.067572	0.000000
1	0.002079	0.466667	0.346154	0.362197	0.065195	0.000000
2	0.006237	0.386111	0.423077	0.377095	0.069100	0.000054
3	0.018711	0.538889	0.288462	0.391061	0.071817	0.000433
4	0.022869	0.563889	0.288462	0.404097	0.073854	0.000487

In [57]:

```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
scaler=MinMaxScaler()
Data_minmax = scaler.fit_transform(Data_numerical)
Data_minmax
```

Out[57]:

```
array([[6.23700624e-03, 5.13888889e-01, 2.30769231e-01, 3.67783985e-01,
        6.75721562e-02, 0.00000000e+00],
       [2.07900208e-03, 4.66666667e-01, 3.46153846e-01, 3.62197393e-01,
        6.51952462e-02, 0.00000000e+00],
       [6.23700624e-03, 3.86111111e-01, 4.23076923e-01, 3.77094972e-01,
        6.91001698e-02, 5.40628210e-05],
       ...,
       [1.03950104e-01, 5.50000000e-01, 3.26923077e-01, 5.27001862e-01,
```

```
        9.64346350e-02, 1.31426718e-01],
       [1.37214137e-01, 5.00000000e-01, 2.88461538e-01, 5.72625698e-01,
        1.03904924e-01, 1.31426718e-01],
       [1.14345114e-01, 5.33333333e-01, 2.30769231e-01, 5.42830540e-01,
        9.96604414e-02, 1.31426718e-01]])
```

In [59]:

```
Data_normal_minmax = pd.DataFrame(Data_minmax, columns=Data_numerical.columns)
Data_normal_minmax
```

Out[59]:

	carat	depth	table	weight	size	price
0	0.006237	0.513889	0.230769	0.367784	0.067572	0.000000
1	0.002079	0.466667	0.346154	0.362197	0.065195	0.000000
2	0.006237	0.386111	0.423077	0.377095	0.069100	0.000054
3	0.018711	0.538889	0.288462	0.391061	0.071817	0.000433
4	0.022869	0.563889	0.288462	0.404097	0.073854	0.000487
...	...	...	...	...	...	...
53935	0.108108	0.494444	0.269231	0.535382	0.097793	0.131427
53936	0.108108	0.558333	0.230769	0.529795	0.097623	0.131427
53937	0.103950	0.550000	0.326923	0.527002	0.096435	0.131427
53938	0.137214	0.500000	0.288462	0.572626	0.103905	0.131427
53939	0.114345	0.533333	0.230769	0.542831	0.099660	0.131427

53940 rows × 6 columns

In [61]:

```
Data_normal.min()
```

Out[61]:

```
carat     0.0
depth     0.0
table     0.0
weight    0.0
size      0.0
price     0.0
dtype: float64
```

In [63]:

```
Data_normal.max()
```

Out[63]:

```
carat     1.0
depth     1.0
table     1.0
weight    1.0
size      1.0
price     1.0
dtype: float64
```

In [65]:

```
Data_normal.mean()
```

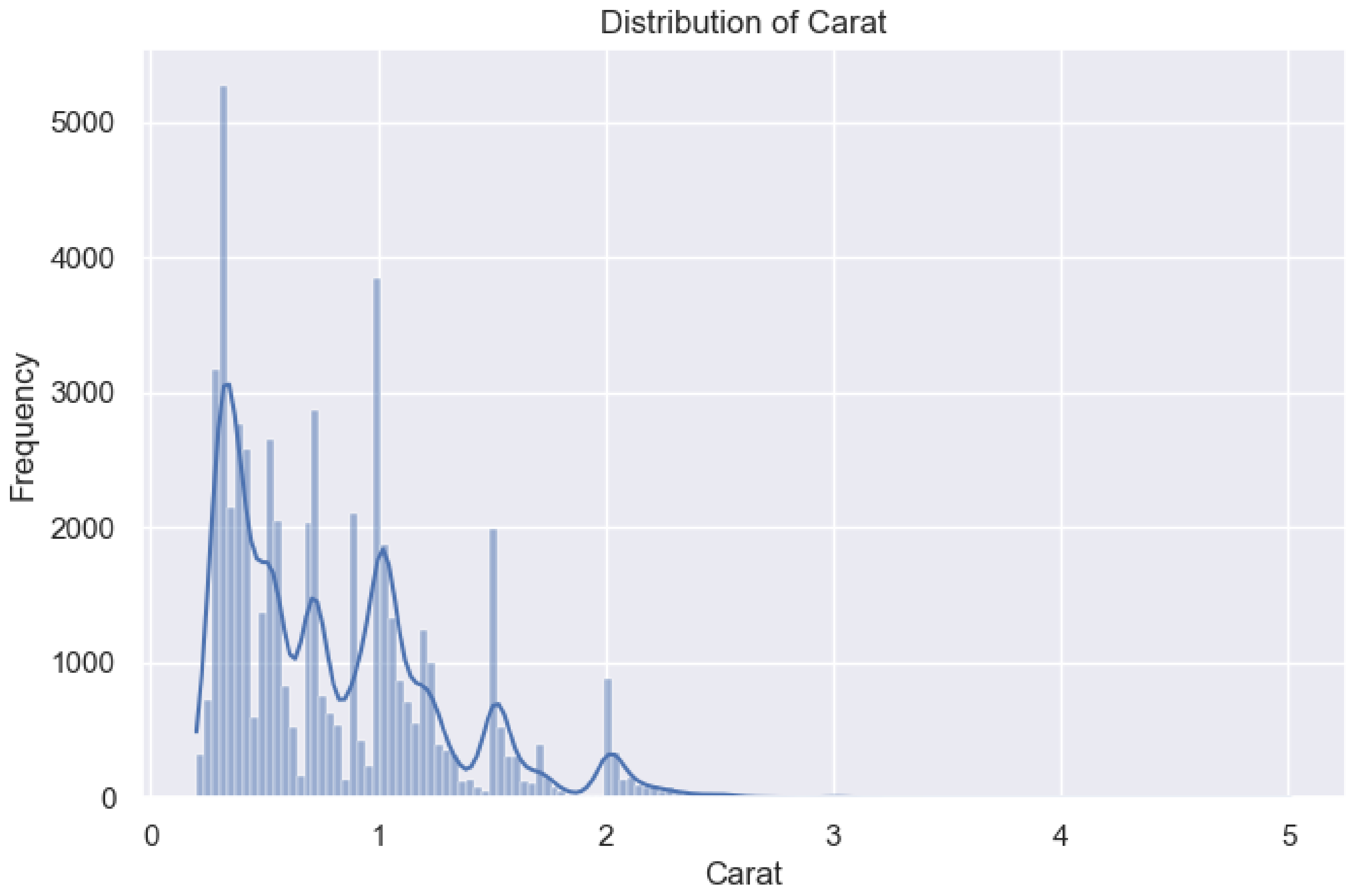
Out[65]:
```
carat     0.124312
depth     0.520817
table     0.278023
weight    0.533627
size      0.097360
price     0.194994
dtype: float64
```

In [67]:
```
Data_normal.std()
```

Out[67]:
```
carat     0.098547
depth     0.039795
table     0.042971
weight    0.104447
size      0.019391
price     0.215680
dtype: float64
```

H. Create the Histogram for all numeric variables and draw the KDE plot on that.

In [87]:
```
sns.histplot(Data_numerical['carat'], kde=True)
plt.title('Distribution of Carat')
plt.xlabel('Carat')
plt.ylabel('Frequency')
plt.show()
```

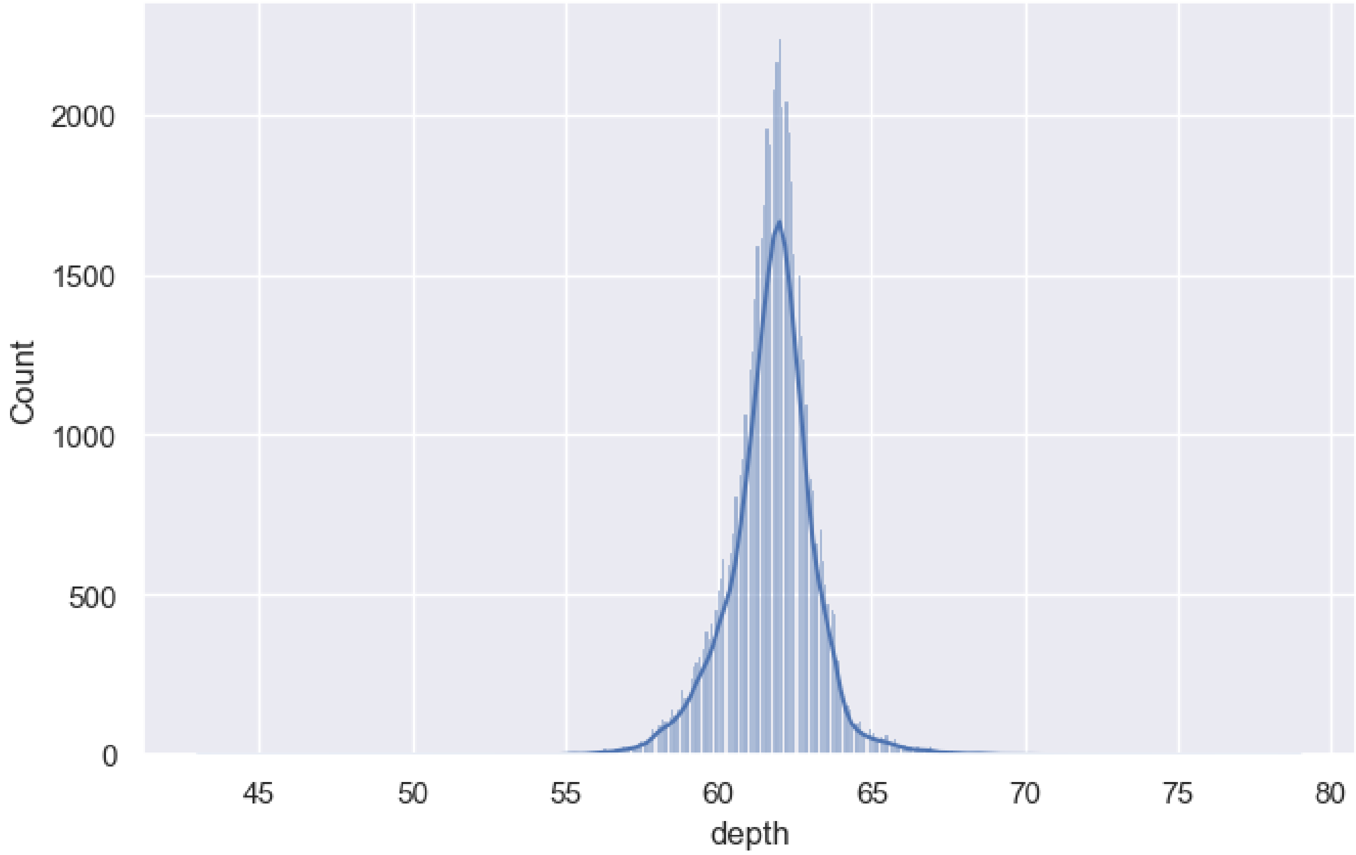


In [75]:
```
sns.set(rc={'figure.figsize':(8,5)})
```

```
sns.histplot(Data_numerical['depth'], kde=True)
```

Out[75]:

```
<Axes: xlabel='depth', ylabel='Count'>
```

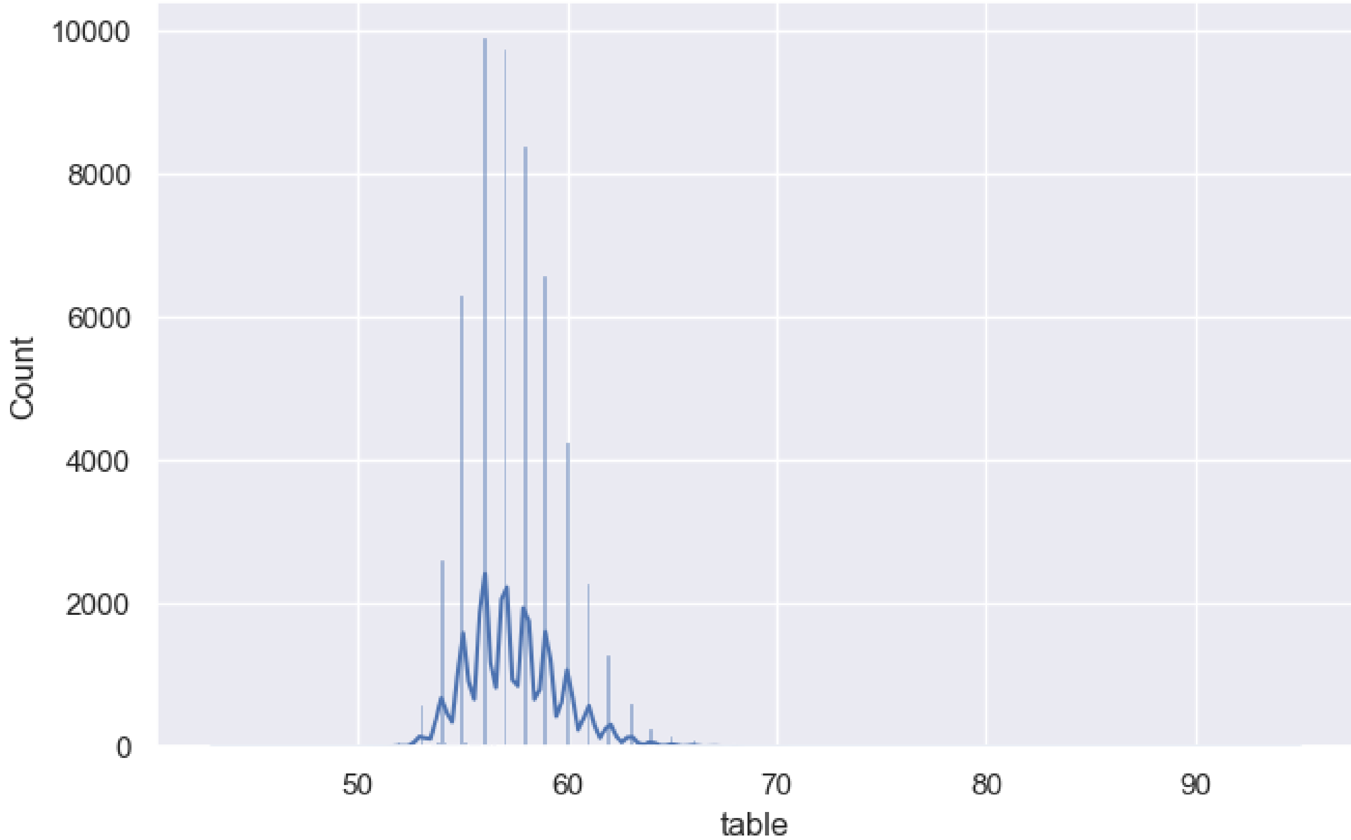


In [77]:

```
sns.set(rc={'figure.figsize':(8,5)})
sns.histplot(Data_numerical['table'], kde=True)
```

Out[77]:

```
<Axes: xlabel='table', ylabel='Count'>
```

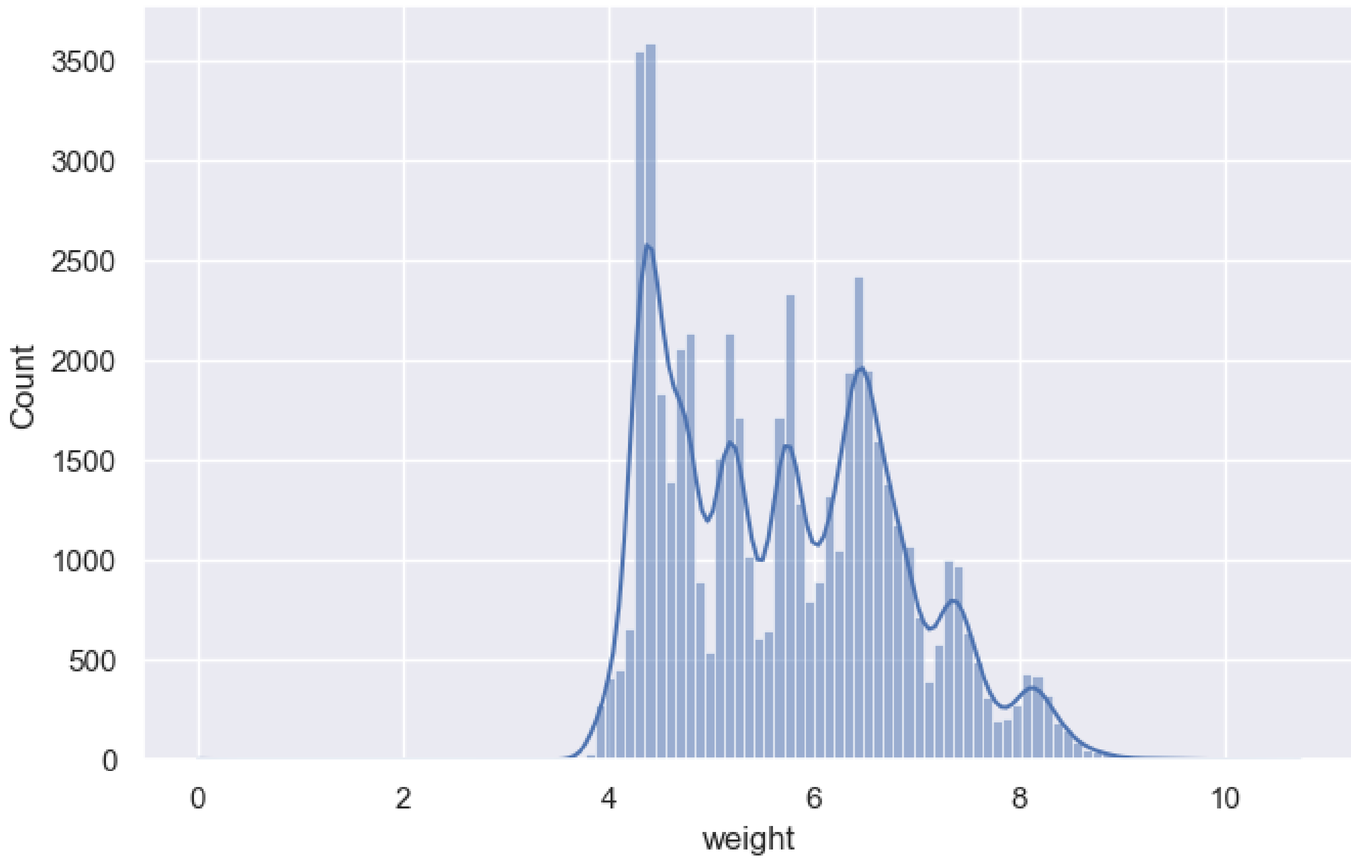


In [79]:

```
sns.set(rc={'figure.figsize':(8,5)})
sns.histplot(Data_numerical['weight'], kde=True)
```

Out[79]:

```
<Axes: xlabel='weight', ylabel='Count'>
```

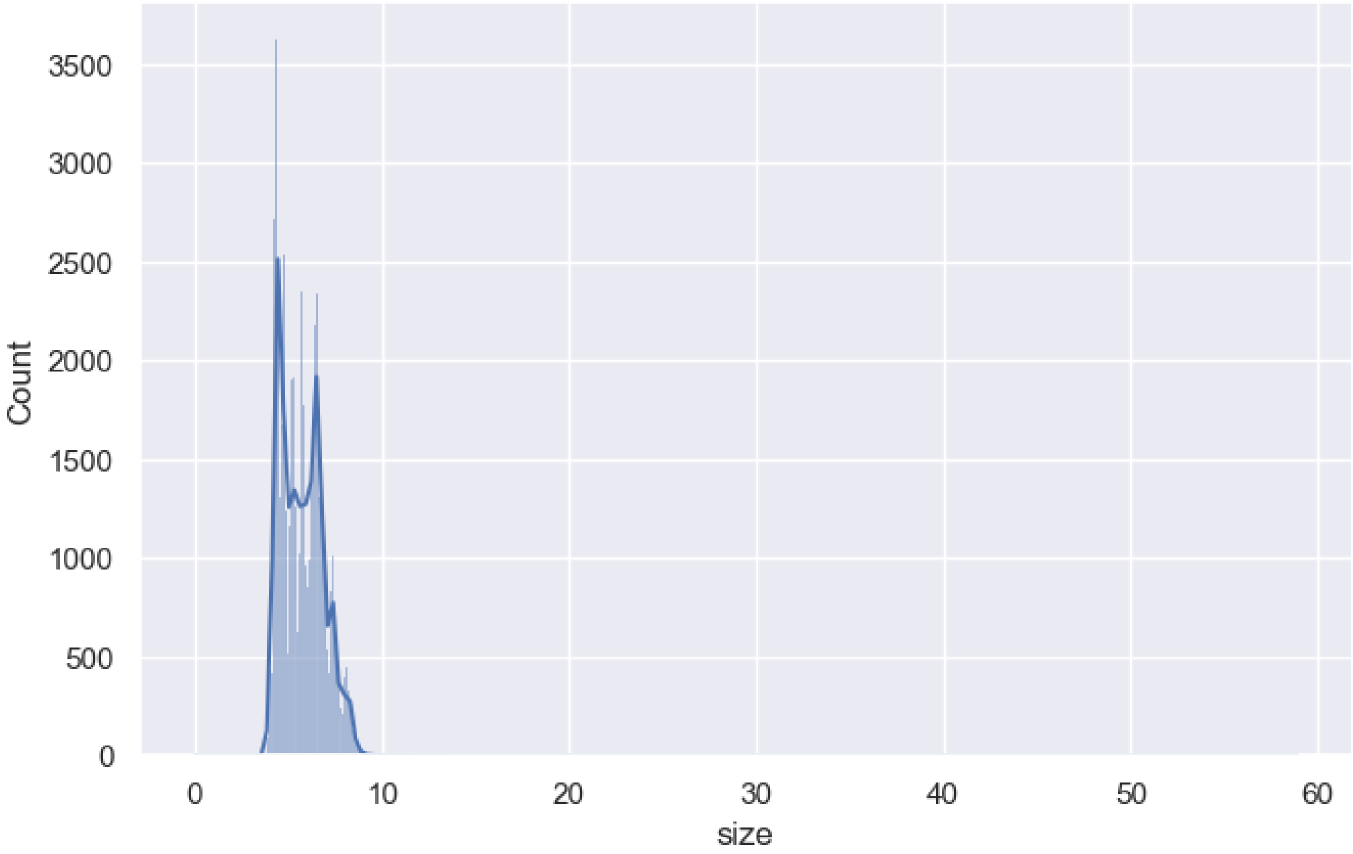


In [81]:

```
sns.set(rc={'figure.figsize':(8,5)})
sns.histplot(Data_numerical['size'], kde=True)
```

Out[81]:

<Axes: xlabel='size', ylabel='Count'>

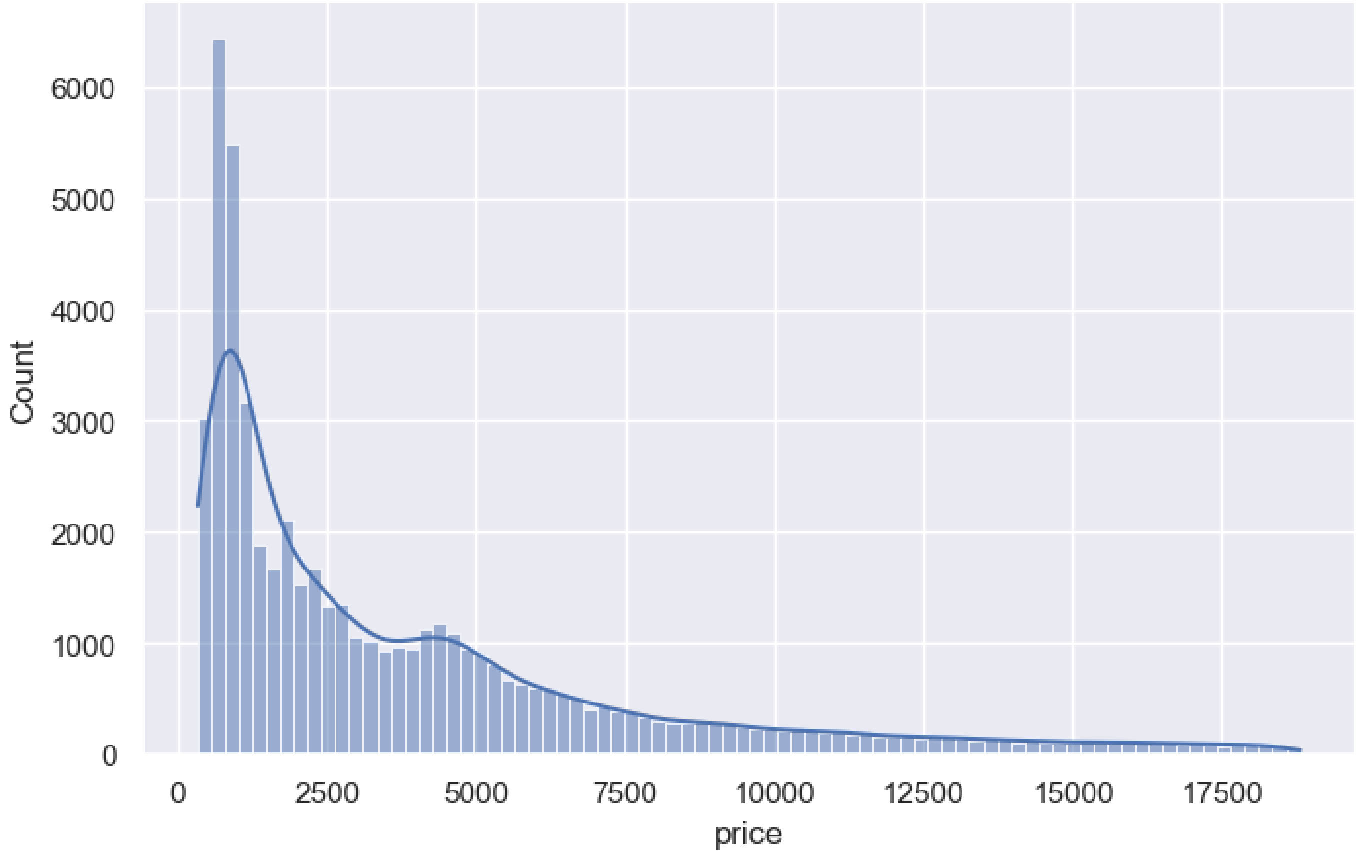


In [83]:

```
sns.set(rc={'figure.figsize':(8,5)})
sns.histplot(Data_numerical['price'], kde=True)
```

Out[83]:

<Axes: xlabel='price', ylabel='Count'>



In [ ]:

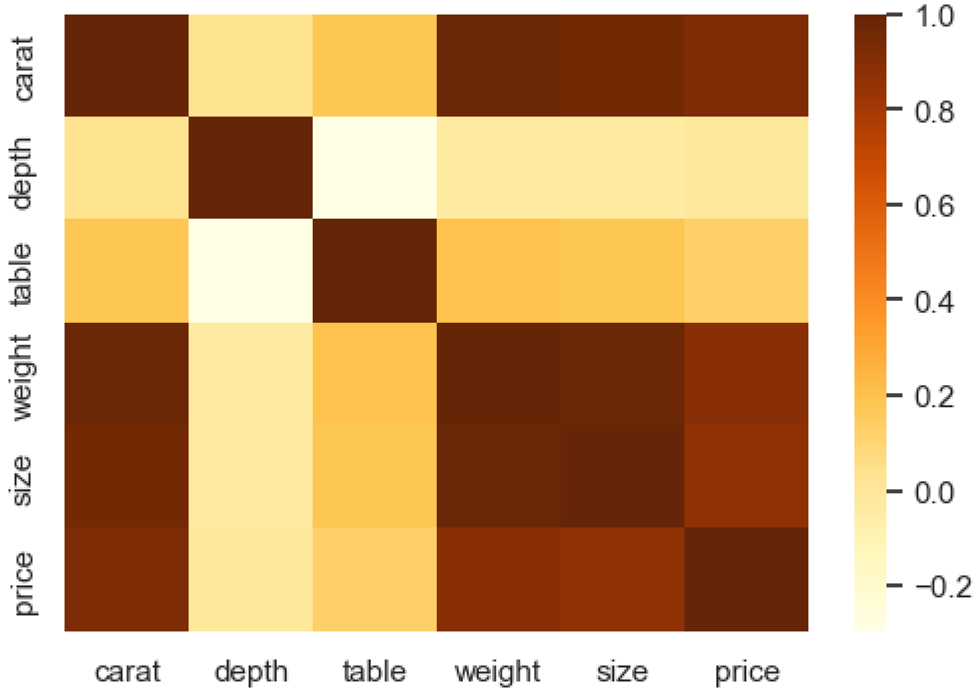
I. Check the correlation between all the numeric variables using HeatMap and try to draw some conclusion about the data.

In [329]:

```
plt.figure(figsize=(6,4))
sns.heatmap(Data_numerical.corr(),cmap="YlOrBr")
```

Out[329]:

```
<Axes: >
```



Conclusion : carat has a good correlation with weight,size,price

weight and size with table has moderate correlation

table and depth have no correlation.

Q2. Explain Gradient descent in detail. How changing the values of learning rate can impact the convergence in Gradient Descent.Gradient descent is an iterative optimization algorithm used to find the minimum of a function, commonly employed in machine learning to minimize the cost function. It works by taking steps in the opposite direction of the gradient (the direction of steepest ascent) of the function, gradually moving towards the minimum point.The learning rate, a crucial hyperparameter, controls the step size during each iteration, significantly impacting the algorithm's convergence speed and stability. The learning rate (α) is a hyperparameter that determines how big a step Gradient Descent takes toward minimizing the loss function at each iteration. How it works: 1. Initialization: Start with an initial set of parameter values (e.g., weights and biases in a neural network). 2. Gradient Calculation: Compute the gradient of the function with respect to each parameter. The gradient indicates the direction of the steepest increase of the function. 3. Parameter Update: Update the parameters by moving them in the opposite direction of the gradient, scaled by a learning rate. The learning rate controls the step size of each update. 4. Iteration: Repeat steps 2 and 3 until the parameters converge to a minimum or a specified number of iterations is reached.

In [ ]:

```
Learning Rate   Speed                Stability           Risk
Too small           Slow                         High    Underfitting
Optimal         Fast                      High             Low
Too large           Fast (initially)        Low Overshooting / Divergence
```