#FEATURE ENGINEERING

###Binning and Binarization

-> Encoding numerical data/features
-> Discretization
-> Types of Discretization
-> Equal Width Binning (Uniform Binning)
-> Equal Frequency Binning (Quantile Binning)
-> Kmeans Binning
-> Encoding the discretized variable
-> Some Examples
-> Custom/Domain based Binning -> Binarization

Ex:

```
+----------+ ----
|    AGE   |     \
+----------+      \
|    27    |       \
+----------+        ---> CATEGORICAL
|    28    |       /
+----------+      /
|    34    |     /
+----------+ ----
```

Sometimes numerical data doesn't represents better than categorical data.

Ex:
Consider we have dataset for Google Play Store where we have number of Downloads.

```
+----------+----------+
|    APP   | DOWNLOAD |
+----------+----------+
|     A    |    23    |
+----------+----------+
|     B    | 10102344 |
+----------+----------+
|     C    |   9921   |
```

So this kinf of data we can convert into categorical using "Bins"

Bins:

```
100+ Downloads  -|
                 |
1000+ Downloads -|
                 |=====> This type can us made easy to work with the data
1M+ Downloads   -|
                 |
1B+ Downloads   -|
```

```
                       Numerical to Categorical
                                  |
             -------------------------------------------
             |                                         |
         Discretization                           Binarization
           (Binning)
```

Discretization is the process of transforming continuous variables into discrete variables by creating a set of contiguous intervals that span the range of the variable's value.
Discretization is also called as "Binning", where Bin is an Alternative name for interval.

Why to use Discretization?
(i) To handle Outliers
(ii) To improve the value spread.

```
+----------+
|   Age    |
+----------+
|   22     |
+----------+
|   23     |
+----------+
|   42     |
+----------+
|   57     |
+----------+
|   81     |
+----------+
|   101    |
+----------+
```

So this kind of data is converted into range interval like:
0-10, 10-20, 20-30, 30-40, 40-50, 50-60, 60-70, 70-80, 80-90, 90-100, 100-110.

```
So like in range:
20-30 we have 2 data i.e., 22 and 23
40-50 we have 1 data i.e., 42
50-60 we have 1 data i.e., 57
80-90 we have 1 data i.e., 81
100-110 we have 1 data i.e., 101
```

Types:

```
                                   Binning
    ----------------------------------|----------------------------------
    |                                 |                                 |
    Unsupervised                  Supervised                       Custom
    Binning                         Binning                       Binning
       |                              |
       |-Equal Width Binning          |-DecisionTree
```

###**Equal Width Binning:**
Equal Width Binning, also known as Uniform Binning, is a method of binning continuous data into intervals of equal width. In this approach, the range of the data is divided into n equal-width intervals, and data points are assigned to the appropriate bin based on their values. This method is simple and easy to implement but may not always be suitable for capturing the underlying distribution of the data.

Steps:

1. Determine the Range: Find the range of the data (R), which is the difference between the maximum and minimum values.
2. Determine Bin Width: Calculate the bin width (W) by dividing the range by the desired number of bins (n): W = R/n
3. Assign Data to Bins: Assign each data point to the bin corresponding to its value.

Advantages:

1. *Simplicity:* Equal Width Binning is straightforward and easy to understand.
2. *Uniformity:* Bins have equal widths, providing a uniform representation of the data.

Formula: W= R/n

where:
W is the bin width.
R is the range of the data.
n is the desired number of bins.

Example:
Consider the dataset:
[5,8,10,12,15,18,20,25,30,40]

If we want to perform Equal Width Binning with n=3 bins:

Determine the Range:
R = Max(Data)−Min(Data)=40−5=35

Determine Bin Width:
W = R/n = 35/3 ≈11.67

Assign Data to Bins:

Bin 1:
[5,16.67]
Bin 2:
[16.67,28.33]
Bin 3:
[28.33,40]

When to Use:

-Equal Width Binning is suitable in the following scenarios:

-Simplicity Requirement: When simplicity is crucial, and a quick, easy-to-understand representation of data is needed.

-Visualizations: For visualizations where a uniform representation of data is more important than capturing underlying distribution details.

-Initial Exploration: As an initial step in data exploration when the primary goal is to gain a broad overview of the data.

Note: Equal Width Binning may not capture the characteristics of skewed or non-uniformly distributed data effectively. In such cases, other binning methods like Equal Frequency Binning or Custom Binning might be more appropriate.

###**Equal Frequency Binning:** Equal Frequency Binning, also known as Quantile Binning, is a data binning technique where data is divided into bins, each containing approximately the same number of data points. This method helps maintain an equal distribution of data across bins.

Advantages:

Robust to Outliers: Equal frequency binning is less sensitive to outliers because it focuses on distributing the data based on quantiles rather than the actual values.

Balanced Representation: Ensures that each bin represents a similar proportion of the dataset, preventing skewed representations in individual bins.

Handling Skewed Distributions: Suitable for datasets with skewed distributions, as it distributes data uniformly based on quantiles rather than raw values.

Formula:

Quantile Calculation:

Q = [k * (n + 1)] / N **bold text**

where:
Q is the quantile for the k th bin.
k is the desired number of bins.
n is the number of data points below the k th quantile.
N is the total number of data points.
Binning:

Once the quantiles are calculated, the data is divided into bins based on these quantiles. Each bin will have approximately the same number of data points.

Example:

Consider the dataset:
[5,8,12,15,18,22,25,30,35,40]

If we want to bin this data into three equal-frequency bins, we calculate the quantiles:
Q1 = [1 * (10 + 1)] / 3 ≈ 4
Q2 = [2 * (10 + 1)] / 3 ≈ 8

Now, we can create three bins:

Bin 1:
[5,8,12,15]
Bin 2:
[18,22,25,30]
Bin 3:
[35,40]
Each bin contains approximately one-third of the data points.

When to Use:

Skewed Distributions: When dealing with datasets that have a skewed distribution, equal frequency binning can be more effective than equal-width binning.

Outlier Sensitivity: In situations where sensitivity to outliers is a concern, equal frequency binning may be a more robust choice.

Balanced Representation: When you want each bin to represent an equal portion of the dataset, ensuring a balanced representation in each bin.

###**K-means Binning:**
K-Means Binning is a data preprocessing technique that involves grouping continuous numeric data into discrete bins or intervals. It is based on the principles of the K-Means clustering algorithm.

Advantages:

-> Simplicity: K-Means Binning is a straightforward method that simplifies continuous data into discrete categories.
-> Reduction of Noise: It can help reduce the impact of outliers or noisy data points.
-> Interpretability: Binned data is often more interpretable and user-friendly, especially in scenarios where the exact numeric values are not critical.
Formula:
The formula for K-Means Binning involves two main steps:

-> K-Means Clustering: The algorithm identifies **'K'** cluster centroids based on the distribution of the data.
-> Assigning Values to Bins: After clustering, each data point is assigned to the bin represented by the nearest cluster centroid.
Example:
Let's consider a dataset of people's ages:
[21,25,28,35,40,45,50,60,70,75]

If we decide to use K=3 for our K-Means Binning, the algorithm may identify centroids at 25,50, and 70. The bins would then be:

Bin 1: Ages close to 25
Bin 2: Ages close to 50
Bin 3: Ages close to 70
Each person's age is then assigned to the nearest bin.

Explanation:

-A person aged 28 would be assigned to Bin 1 because it's closer to the centroid 25.
-A person aged 45 would be assigned to Bin 2 because it's closer to the centroid 50.
This process creates discrete bins that represent different age groups.

When to Use:

- Data Compression: When dealing with large datasets, K-Means Binning can compress continuous data into a smaller set of discrete values.
- Simplification: If the analysis or visualization task benefits from simplified, categorized data.
- Handling Outliers: It can be useful when you want to handle outliers or extreme values by placing them in specific bins.

In [ ]:

```

```

###**ENCODING THE DISCRETIZED VARIABLE**
from sklearn library from preprocessing we can use **KBinsDiscretizer**
where we need to give three parameters: no. of bins, strategy and encoding.
no. of bins = n,
strategy = uniform, quantile or Kmeans
encoding = after converting in discrete variable and getting output we need to tell how encoding should be done (i) Ordinal (ii) OneHotEncoding

Documentation:
Parameters: ####n_binsint or array-like of shape (n_features,), default=5
The number of bins to produce. Raises ValueError if n_bins < 2.

####**encode{‘onehot’, ‘onehot-dense’, ‘ordinal’}, default=’onehot’**
Method used to encode the transformed result.

‘onehot’: Encode the transformed result with one-hot encoding and return a sparse matrix. Ignored features are always stacked to the right.

‘onehot-dense’: Encode the transformed result with one-hot encoding and return a dense array. Ignored features are always stacked to the right.

‘ordinal’: Return the bin identifier encoded as an integer value.

####**strategy{‘uniform’, ‘quantile’, ‘kmeans’}, default=’quantile’**
Strategy used to define the widths of the bins.

‘uniform’: All bins in each feature have identical widths.

‘quantile’: All bins in each feature have the same number of points.

‘kmeans’: Values in each bin have the same nearest center of a 1D k-means cluster.

Lets import some necessary libraries.

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score

from sklearn.preprocessing import KBinsDiscretizer
from sklearn.compose import ColumnTransformer
```

Import dataset.

In [3]:
```
titanic_dataset = pd.read_csv('titanic_dataset.csv',usecols=['Age','Fare','Survived'])
```

So I choose Titanic dataset

In [4]:
```
titanic_dataset
```

Out[4]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500
...	...	...	...
886	0	27.0	13.0000
887	1	19.0	30.0000
888	0	NaN	23.4500
889	1	26.0	30.0000
890	0	32.0	7.7500

891 rows × 3 columns

Here we can see the dataset.
Lets check the missing values.

In [5]:
```
titanic_dataset.isna().sum()
```

Out[5]:
```
Survived      0
Age         177
Fare          0
dtype: int64
```

Here, **Age** is having 177 missing values.

In [6]:
```
titanic_dataset.dropna(inplace=True)
```

So I just dropped the null value rows.

In [7]:
```
titanic_dataset
```

Out[7]:

	Survived	Age	Fare
0	0	22.0	7.2500
1	1	38.0	71.2833
2	1	26.0	7.9250
3	1	35.0	53.1000
4	0	35.0	8.0500
...	...	...	...
885	0	39.0	29.1250
886	0	27.0	13.0000
887	1	19.0	30.0000
889	1	26.0	30.0000
890	0	32.0	7.7500

714 rows × 3 columns

Here we can see, before there were 892 rows and now there are 715 rows.

In [8]:
```
titanic_dataset.isna().sum()
```

Out[8]:
```
Survived    0
Age         0
Fare        0
dtype: int64
```

And no missing values are present here.

In [9]:
```
X = titanic_dataset.iloc[:,1:]
y = titanic_dataset.iloc[:,0]
```

I choose Independent variables as Age and Fare so I store these independent variables in X.
Dependent variables as Survived so I stored this dependent variables in y.

In [10]:
```
X
```

Out[10]:

	Age	Fare
0	22.0	7.2500
1	38.0	71.2833
2	26.0	7.9250
3	35.0	53.1000
4	35.0	8.0500
...	...	...
885	39.0	29.1250
886	27.0	13.0000
887	19.0	30.0000
889	26.0	30.0000
890	32.0	7.7500

714 rows × 2 columns

In [11]:
```
y
```

Out[11]:
```
0      0
1      1
2      1
3      1
4      0
      ..
885    0
886    0
887    1
889    1
890    0
Name: Survived, Length: 714, dtype: int64
```

In [12]:
```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2,random_state=42)
```

I divided **X** and **y** data for training and testing using split. For that I just used **train_test_split**.

In [13]:
```
DTC = DecisionTreeClassifier()
```

Stored **DecisionTreeClassifier** algorithm in object DTC which we need for training and predicting.

In [14]:
```
DTC.fit(X_train,y_train)
y_pred = DTC.predict(X_test)
```

So I **fit** the Decision Tree Classifier algorithm onn training data i.e., **X_train** and **y_train**.
Then after that I just predicted the results for testing data i.e., **X_test** and stored those predicted values in variable **y_pred**

In [15]:
```
y_pred
```

Out[15]:
```
array([1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1,
       1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0,
       1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
       1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0,
       0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1,
       1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0,
       1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1])
```

So these are y_pred values

In [16]:
```
accuracy_score(y_test,y_pred)
```

Out[16]:
```
0.6293706293706294
```

Then using accuracy_score, I just calculated **accuracy** between actual values i.e., **y_test** and predicted values i.e., **y_pred**.
The accuracy comes out as **63.6%**.
So without applying any sort of transformation technique on our numerical columns we are getting accuracy of around 63.6%.

In [17]:
```
np.mean(cross_val_score(DecisionTreeClassifier(), X, y, cv=10,scoring = 'accuracy'))
```

Out[17]:
```
0.6288732394366197
```

Here I cross validate the reults after running Decision Tree Classifier on different random samples of data (doing this 10 times) and the accuracy comes out to **63.6%**

#####**Applying Discretizer**

In [18]:
```
kbin_age = KBinsDiscretizer(n_bins=15,encode='ordinal',strategy='quantile')
kbin_fare = KBinsDiscretizer(n_bins=15,encode='ordinal',strategy='quantile')
```

Here we choose Binning technique with parameters as:
-no. of bins = 15 (**n_bins**).
-encoding technique = Ordinal (**encode**).

-strategy applies = *Equal Frequency* also called as *Quantile Binning* (**strategy**).
One chosen for **Age** and one chosen for **Fare** and stored these techniques in object **kbin_age** and **kbin_fare**.

- strategies we have: **'uniform', 'quantile', 'kmeans'.**
- encoding we have: **'onehot', 'onehot-dense', 'ordinal'.**

In [19]:
```
CT = ColumnTransformer([
    ('first',kbin_age,[0]),
    ('second',kbin_fare,[1])
])
```

ColumnTransformer:
We used this for applying transformers to columns of a dataset.
('first', kbin_age, [0]):
The first transformation is applied to the column at index **0**. **'first'** is just a name or label for this specific transformation. **kbin_age** is a transformer that will be applied to the specified column.
[0] indicates that this transformation is applied to the column at index **0** in the dataset. **('second', kbin_fare, [1]):**
The second transformation is applied to the column at index **1**. **'second'** is the name or label for this transformation. **kbin_fare** is another transformer, designed for handling fare data.
[1] indicates that this transformation is applied to the column at index **1** in the dataset.
CT:
CT is the resulting ColumnTransformer object that combines both transformations.
It can now be used to apply these transformations to a dataset selectively.
In simple terms, the code is creating a ColumnTransformer named CT that applies two different transformations (**kbin_age and kbin_fare**) to specific columns in a dataset. The first transformation (**kbin_age**) is applied to the column at index **0**, and the second transformation (**kbin_fare**) is applied to the column at index **1**.

In [20]:
```
X_train_CT = CT.fit_transform(X_train)
X_test_CT = CT.transform(X_test)
```

CT.fit_transform(X_train):
We created object **CT** which is like our tool that knows how to transform certain parts of your data.
X_train is our training data.
CT.fit_transform(X_train) means, CT learn how to transform the training data.

After this line, **X_train_CT** contains the transformed training data.
CT.transform(X_test):
Now, we know how **CT** transform data.

X_test is testing data.
CT.transform(X_test) says, CT use what you've learned to transform this new data.
X_test_CT now holds the transformed new data.

In simple words, we're training your transformation tool (CT) with the training data, and then using the same tool to transform new data (testing

In [21]:
```
CT.named_transformers_
```

Out[21]:
```
{'first': KBinsDiscretizer(encode='ordinal', n_bins=15),
 'second': KBinsDiscretizer(encode='ordinal', n_bins=15)}
```

We get transformers we used

In [22]:
```
CT.named_transformers_['first'].n_bins_
```

Out[22]:
```
array([15])
```

no. of bins in step 'first'

In [23]:
```
CT.named_transformers_['first'].bin_edges_
```

Out[23]:
```
array([array([ 0.42,  6.  , 16.  , 19.  , 21.  , 23.  , 25.  , 28.  , 30.  ,
              32.  , 35.  , 38.  , 42.  , 47.  , 54.  , 80.  ])              ],
      dtype=object)
```

Range in which bin is created.
Here, for step 'first' the range is:
0.42 - 6.
6 - 16.
16 - 19.
19 - 21.
21 - 23.
23 - 25.

25 - 28.
28 - 30.
30 - 32.
32 - 35.
35 - 38.
38 - 42.
42 - 47.
47 - 54.
54 - 80.

CT.named_transformers_['first'].bin_edges_ is a way to access the information about the bins (edges) that were created during the transformation of the 'first' part in your ColumnTransformer (CT).

Here's a breakdown:

CT.named_transformers_['first']: This part is retrieving the transformer named **'first'** from our ColumnTransformer (**CT**).

.bin_edges_: This is accessing a property or information stored within the **'first'** transformer. Specifically, **bin_edges_** refers to the edges of the bins that were created.

So, when we run **CT.named_transformers_['first'].bin_edges_**, it gives us the bin edges that were used when transforming the data in the **'first'** part of our ColumnTransformer. These bin edges can be useful for understanding how the data was grouped or binned during the transformation.

In [24]:
```
CT.named_transformers_['second'].bin_edges_
```

Out[24]:
```
array([array([  0.    ,   7.25  ,   7.775 ,   7.8958,   8.1583,  10.5   ,
               13.    ,  14.4542,  18.75  ,  26.    ,  26.55  ,  31.275 ,
               51.4792,  76.2917, 108.9   , 512.3292])                   ],
      dtype=object)
```

CT.named_transformers_['second'].bin_edges_ is a similar concept to the explanation I provided earlier, but in this case, it's specifically referring to the **'second'** part of our ColumnTransformer (**CT**).

Breaking it down:

CT.named_transformers_['second']: This part is retrieving the transformer named **'second'** from our ColumnTransformer (**CT**).

.bin_edges_: Like before, this is accessing a property or information stored within the **'second'** transformer. **bin_edges_** specifically refers to the edges of the bins that were created.

So, when I run **CT.named_transformers_['second'].bin_edges_**, it gives us the bin edges that were used when transforming the data in the **'second'** part of our ColumnTransformer. These bin edges can provide insights into how the data was grouped or binned during the transformation process.

In [25]:
```
output = pd.DataFrame({
    'age':X_train['Age'],
    'age_CT':X_train_CT[:,0],
    'fare':X_train['Fare'],
    'fare_CT':X_train_CT[:,1]
})
```

Creating a DataFrame (**output**) to compare the original features (**'age'** and **'fare'**) with their transformed counterparts after applying the **ColumnTransformer**.

Breaking it down:

'age': X_train['Age']: This column in the DataFrame represents the original **'Age'** feature from our training data.

'age_CT': X_train_CT[:,0]: This column represents the transformed **'age'** feature obtained after applying the first transformer in our **ColumnTransformer** (**kbin_age**).

'fare': X_train['Fare']: Similar to **'age'**, this column represents the original **'Fare'** feature.

'fare_CT': X_train_CT[:,1]: This column represents the transformed **'fare'** feature obtained after applying the second transformer in our **ColumnTransformer** (**kbin_fare**).

output allows us to inspect the original and transformed values side by side for **'age'** and **'fare'**. It's a handy way to understand how our transformations have affected the data.

In [26]:
```
output['age_labels'] = pd.cut(x=X_train['Age'],
                                    bins=CT.named_transformers_['first'].bin_edges_[0].tolist())
```

Creating a new column **'age_labels'** in the DataFrame **output**. The purpose of this column is to categorize the original **'Age'** values into specific bins based on the edges defined by the first transformer (**'first'**) in our **ColumnTransformer**.

Breaking it down:

pd.cut(x=X_train['Age'], bins=CT.named_transformers_['first'].bin_edges_[0].tolist()):
uses the cut function from pandas to bin the **'Age'** values. The **x** parameter takes the original **'Age'** values, and the bins parameter specifies the edges of the bins. In this case, it uses the bin edges obtained from the first transformer in our **ColumnTransformer (CT.named_transformers_['first'].bin_edges_[0])**.

output['age_labels'] = ...: This assigns the resulting bin labels to the new column **'age_labels'** in the **output** DataFrame.

'age_labels' will contain categorical labels corresponding to the bins into which the original **'Age'** values have been grouped based on the bin edges defined by the first transformer.

In [27]:
```
output['fare_labels'] = pd.cut(x=X_train['Fare'],
                               bins=CT.named_transformers_['second'].bin_edges_[0].tolist())
```

Creating a new column **'fare_labels'** in the DataFrame **output**. Similar to the previous, the purpose of this column is to categorize the original **'Fare'** values into specific bins based on the edges defined by the second transformer (**'second'**) in our **ColumnTransformer**.

Breaking it down:

pd.cut(x=X_train['Fare'], bins=CT.named_transformers_['second'].bin_edges_[0].tolist()):
uses the cut function from pandas to bin the **'Fare'** values. The **x** parameter takes the original **'Fare'** values, and the bins parameter specifies the edges of the bins. In this case, it uses the bin edges obtained from the second transformer in our **ColumnTransformer (CT.named_transformers_['second'].bin_edges_[0])**.

output['fare_labels'] = ...: This assigns the resulting bin labels to the new column **'fare_labels'** in the output DataFrame.

'fare_labels' will contain categorical labels corresponding to the bins into which the original **'Fare'** values have been grouped based on the bin edges defined by the second transformer.

In [28]:
```
output.sample(5)
```

Out[28]:

	age	age_CT	fare	fare_CT	age_labels	fare_labels
636	32.0	9.0	7.9250	3.0	(30.0, 32.0]	(7.896, 8.158]
621	42.0	12.0	52.5542	12.0	(38.0, 42.0]	(51.479, 76.292]
348	3.0	0.0	15.9000	7.0	(0.42, 6.0]	(14.454, 18.75]
452	30.0	8.0	27.7500	10.0	(28.0, 30.0]	(26.55, 31.275]
27	19.0	3.0	263.0000	14.0	(16.0, 19.0]	(108.9, 512.329]

Obtained 5 random samples.

We can observe difference in age and age_CT, fare and fare_CT.
age_labels->
age_CT: **9.0** comes in range age_labels **30.0, 32.0**
age_CT: **12.0** comes in range age_labels **38.0 - 42.0**
age_CT: **0.0** comes in range age_labels **0.42, 6.0**
age_CT: **8.0** comes in range age_labels **28.0, 30.0**
age_CT: **3.0** comes in range age_labels **16.0, 19.0**
Same goes for fare,
fare_CT: **3.0** comes in range fare_labels **7.896, 8.158**
fare_CT: **12.0** comes in range fare_labels **51.479, 76.292**
fare_CT: **7.0** comes in range fare_labels **14.454, 18.75**
fare_CT: **10.0** comes in range fare_labels **26.55, 31.275**
fare_CT: **14.0** comes in range fare_labels **108.9, 512.329**

Since transformation is applied on train and test data, let's again use Decision Tree Classifier on transformed data and predict results on X_train_CT and obtain accuracy score.
(if no change in accuracy then try to increase or decrease the bin size, try to use different strategy and encoding method)

In [29]:
```
DTC_1 = DecisionTreeClassifier()
DTC_1.fit(X_train_CT,y_train)
y_pred2 = DTC_1.predict(X_test_CT)
```

Using a Decision Tree Classifier (**DecisionTreeClassifier**) to train a model (**DTC_1**) on the transformed training data (**X_train_CT**) and corresponding labels (**y_train**). After training, it predicts the labels for the transformed test data (**X_test_CT**) using the predict method and stores the predictions in the variable **y_pred2**.

DTC_1 = DecisionTreeClassifier(): This line creates an instance of the Decision Tree Classifier.

DTC_1.fit(X_train_CT, y_train): This line trains the model on the transformed training data (**X_train_CT**) and the corresponding labels (**y_train**).

y_pred2 = DTC_1.predict(X_test_CT): This line predicts the labels for the transformed test data (**X_test_CT**) using the trained model and stores the predictions in the variable **y_pred2**.

In [30]:
```
accuracy_score(y_test,y_pred2)
```

Out[30]:
```
0.6363636363636364
```

Checked accuracy for actual values and the predicted values wich is **63.6%**.

In [31]:
```
X_CT = CT.fit_transform(X)
np.mean(cross_val_score(DecisionTreeClassifier(),X,y,cv=10,scoring='accuracy'))
```

Out[31]:
```
0.6288928012519561
```

Using cross-validation to evaluate the accuracy of a Decision Tree Classifier model on the dataset.

X_CT = CT.fit_transform(X):
This line transforms the entire dataset **X** using the ColumnTransformer named **CT**. It applies the specified transformations to the columns.

cross_val_score(DecisionTreeClassifier(), X, y, cv=10, scoring='accuracy'):
This line performs 10-fold cross-validation using a Decision Tree Classifier. It evaluates the accuracy of the model on each fold and returns an array of accuracy scores.

np.mean(...): This line calculates the mean of the accuracy scores obtained from cross-validation. It provides a single metric representing the average accuracy of the model across all folds.

The code is transforming the dataset, applying 10-fold cross-validation to a Decision Tree Classifier, and then calculating the average accuracy of the model.

In [32]:
```
def discretize(bins,strategy):
    kbin_age = KBinsDiscretizer(n_bins=bins,encode='ordinal',strategy=strategy)
    kbin_fare = KBinsDiscretizer(n_bins=bins,encode='ordinal',strategy=strategy)

    CT = ColumnTransformer([
        ('first',kbin_age,[0]),
        ('second',kbin_fare,[1])
    ])

    X_CT = CT.fit_transform(X)
    print(np.mean(cross_val_score(DecisionTreeClassifier(),X,y,cv=10,scoring='accuracy')))

    plt.figure(figsize=(14,4))
    plt.subplot(121)
    plt.hist(X['Age'])
    plt.title("Before")

    plt.subplot(122)
    plt.hist(X_CT[:,0],color='red')
    plt.title("After")

    plt.show()

    plt.figure(figsize=(14,4))
    plt.subplot(121)
    plt.hist(X['Fare'])
    plt.title("Before")

    plt.subplot(122)
    plt.hist(X_CT[:,1],color='red')
    plt.title("Fare")

    plt.show()
```

Created function 'discretize' and pass two parameters bins and strategy.
Rest I did the same things as earlier and plotted.

This function, discretize, takes two parameters (**bins and strategy**) and performs tasks:

- Initializes two KBinsDiscretizer instances (**kbin_age and kbin_fare**) with the specified number of **bins** and **strategy**.
- Creates a ColumnTransformer (**CT**) with two transformers, one for each feature (**'Age' and 'Fare'**).
- Transforms the dataset (**X**) using the ColumnTransformer and applies discretization to the specified features.
- Prints the mean accuracy obtained from 10-fold cross-validation using a Decision Tree Classifier on the transformed dataset.
- Plots histograms to visualize the distribution of the features before and after discretization.

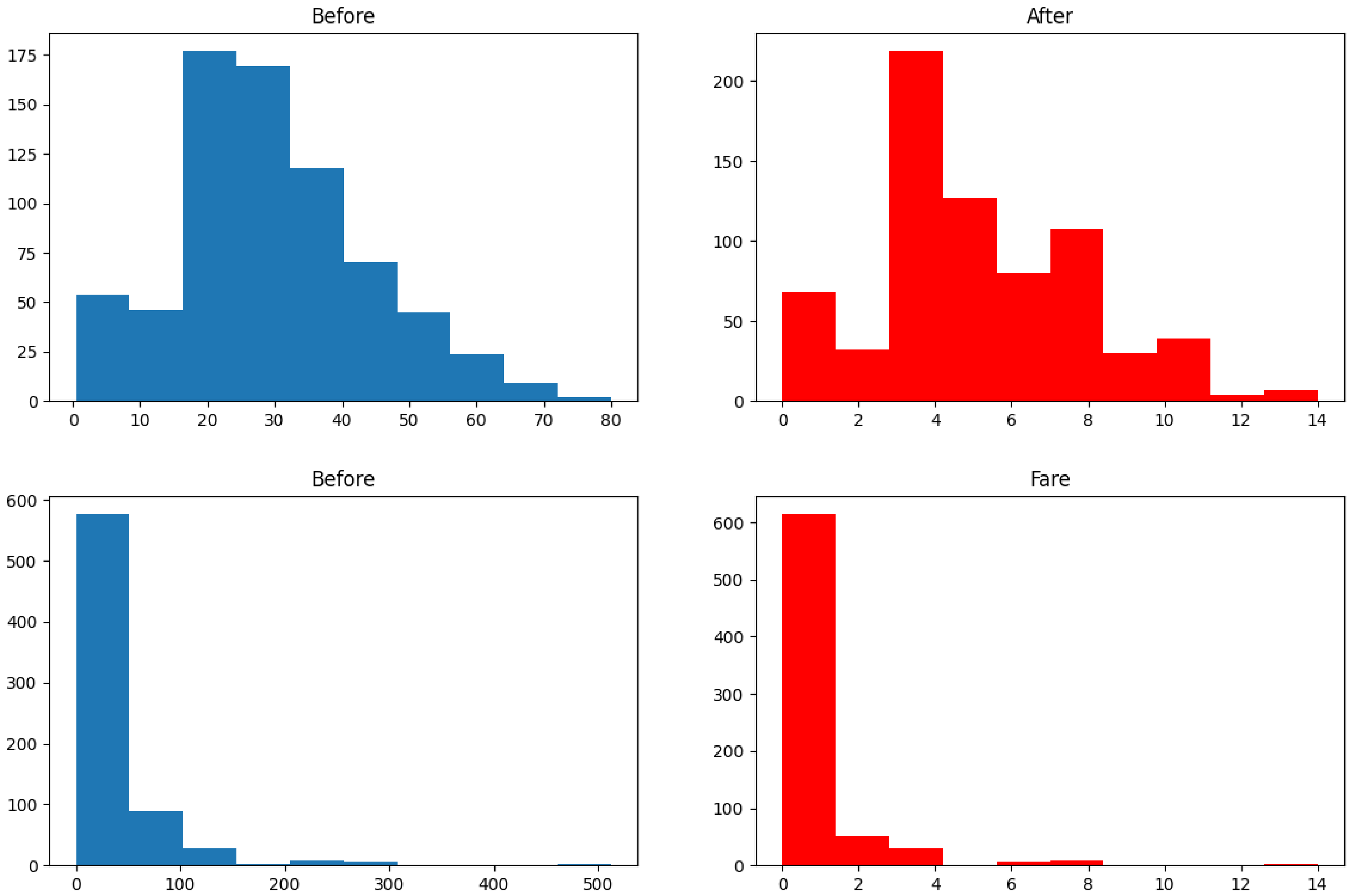
For **'Age':**

The first subplot shows the histogram of **'Age'** before discretization. The second subplot shows the histogram of the transformed **'Age'** after discretization.
For **'Fare':**

The first subplot shows the histogram of **'Fare'** before discretization. The second subplot shows the histogram of the transformed **'Fare'** after discretization. This function provides a visual representation of how discretization affects the distribution of features and evaluates the model's accuracy after the transformation.

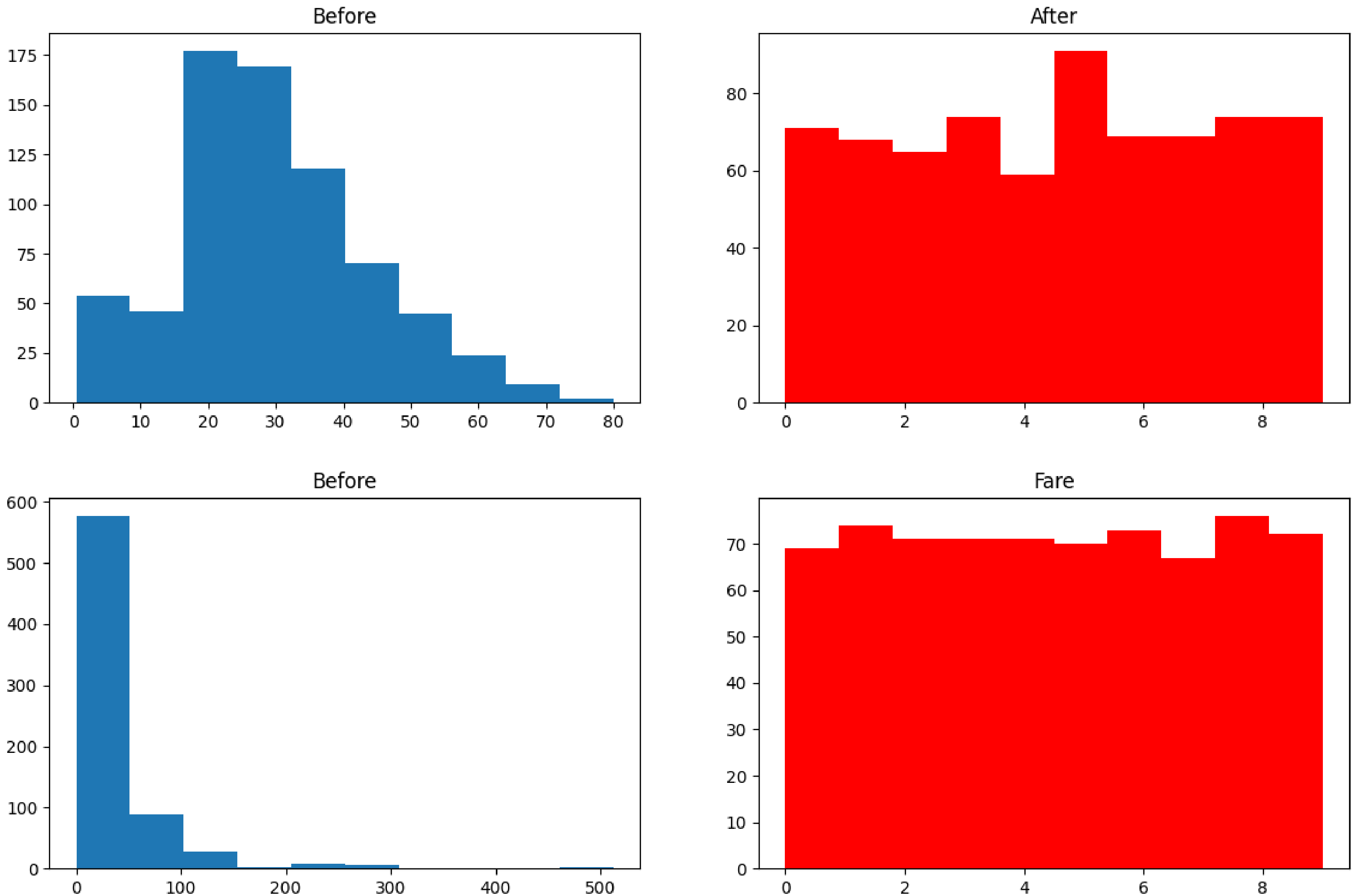
In [33]:
```
discretize(15,'uniform')
```

```
0.6331181533646322
```



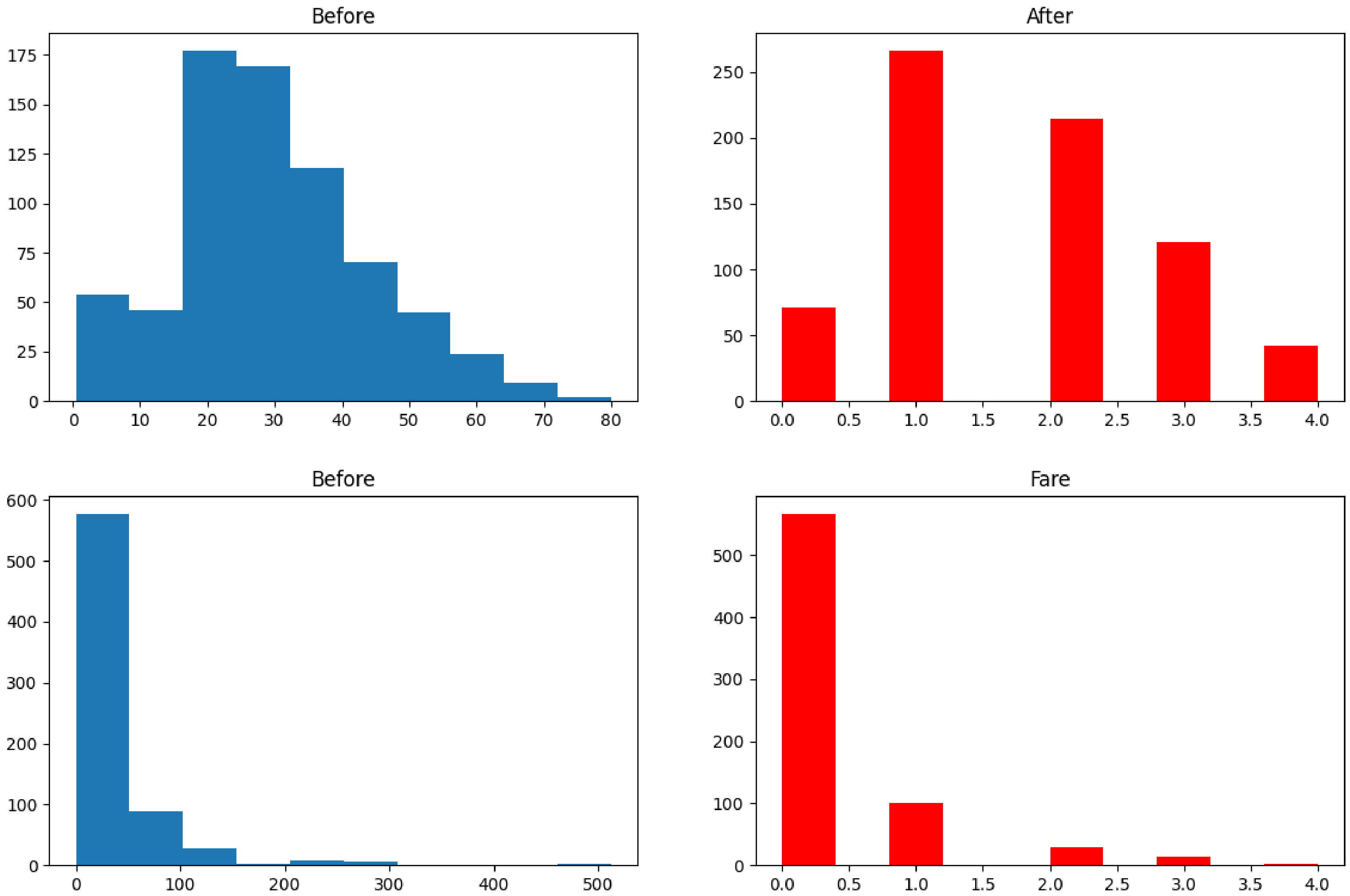
In [34]:
```
discretize(10,'quantile')
```

```
0.6303012519561815
```



In [35]:
```
discretize(5,'kmeans')
```

```
0.6317292644757433
```



discretize function is applied with **bins=5** and **strategy='kmeans'**. This means that both **'Age'** and **'Fare'** features are discretized into 5 bins using the k-means binning strategy.
output:

Accuracy Evaluation: The function prints the mean accuracy obtained from 10-fold cross-validation using a Decision Tree Classifier on the transformed dataset.

Histograms: The function displays histograms to visualize the distribution of features before and after discretization.

For 'Age':

The first subplot shows the histogram of **'Age'** before discretization. The second subplot shows the histogram of the transformed **'Age'** after k-means discretization. **For 'Fare':**

The first subplot shows the histogram of **'Fare'** before discretization. The second subplot shows the histogram of the transformed **'Fare'** after k-means discretization. This allows us to visually compare how the distribution of features changes after applying k-means discretization with 5 bins. The accuracy score provides an insight into how well a Decision Tree model performs on the discretized data.

In [35]:

```
```

In [35]:

```
```

In [35]:

```
```

###CUSTOMIZED BINNING

In [58]:
```
import pandas as pd
import numpy as np

np.random.seed(42)

# Generate sample data
data = {'CustomerID': range(1, 101),
        'TotalPurchaseAmount': np.random.uniform(50, 500, size=100)}

customer_data = pd.DataFrame(data)
```

In [59]:
```
customer_data
```

Out[59]:

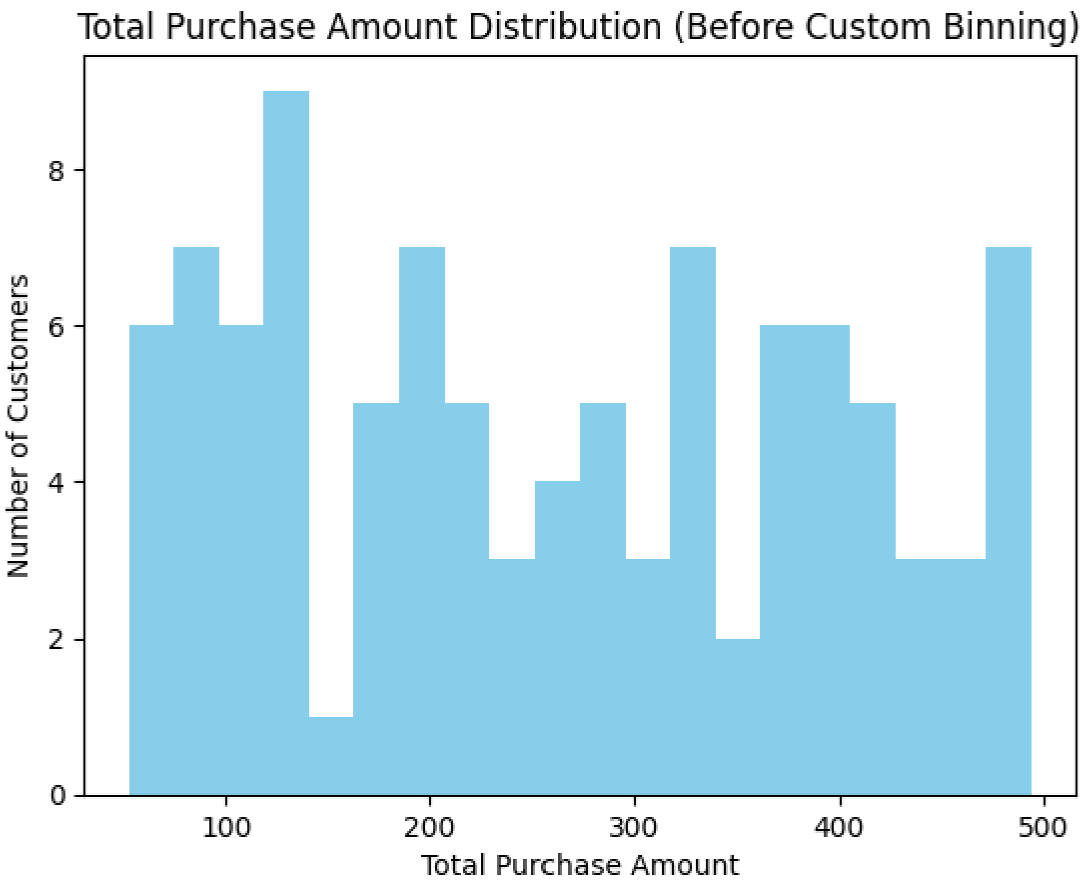
	CustomerID	TotalPurchaseAmount
0	1	218.543053
1	2	477.821438
2	3	379.397274
3	4	319.396318
4	5	120.208388
...	...	...
95	96	272.208018
96	97	285.229773
97	98	242.393458
98	99	61.438607
99	100	98.551142

100 rows × 2 columns

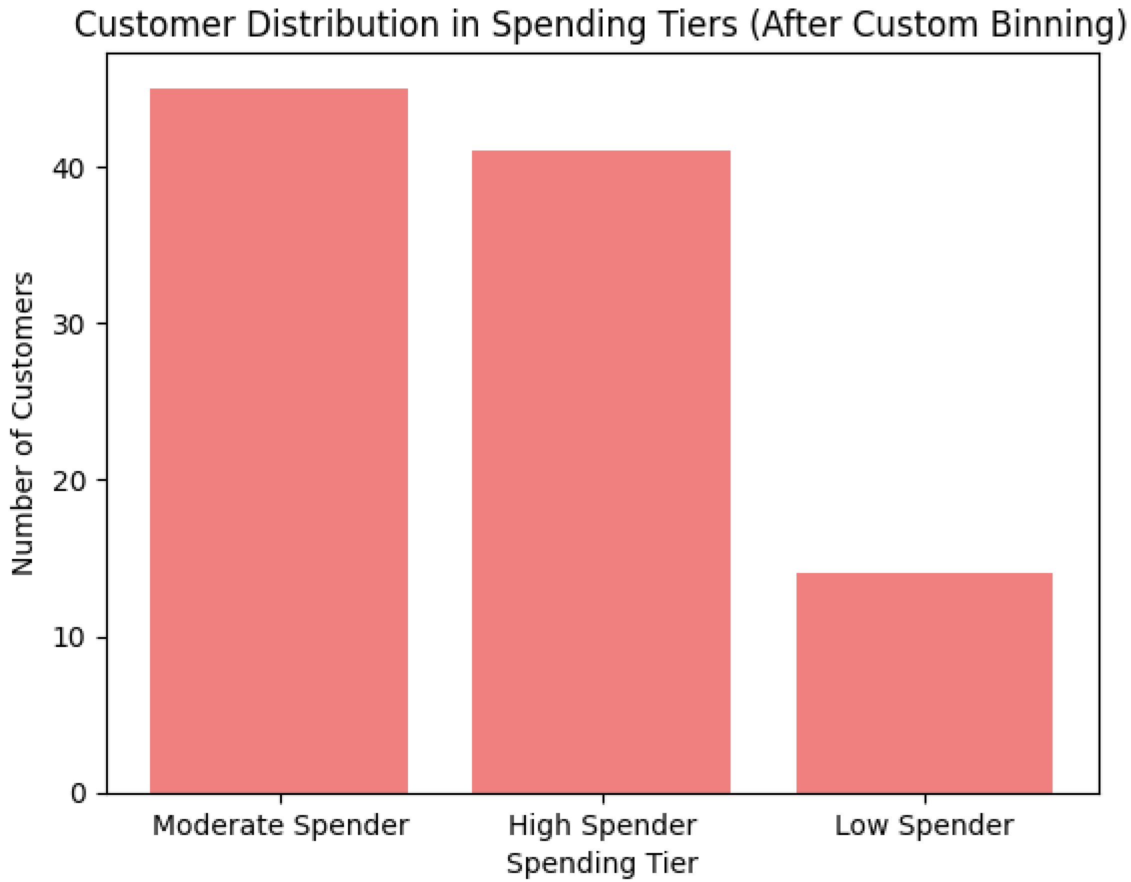
In [60]:
```
def custom_binning(amount):
    if amount < 100:
        return 'Low Spender'
    elif 100 <= amount < 300:
        return 'Moderate Spender'
    elif 300 <= amount < 500:
        return 'High Spender'
    else:
        return 'Very High Spender'
```

In [61]:
```
customer_data['SpendingTier'] = customer_data['TotalPurchaseAmount'].apply(custom_binning)
```

In [62]:
```
import matplotlib.pyplot as plt

plt.hist(customer_data['TotalPurchaseAmount'], bins=20, color='skyblue')
plt.title('Total Purchase Amount Distribution (Before Custom Binning)')
plt.xlabel('Total Purchase Amount')
plt.ylabel('Number of Customers')
plt.show()
```



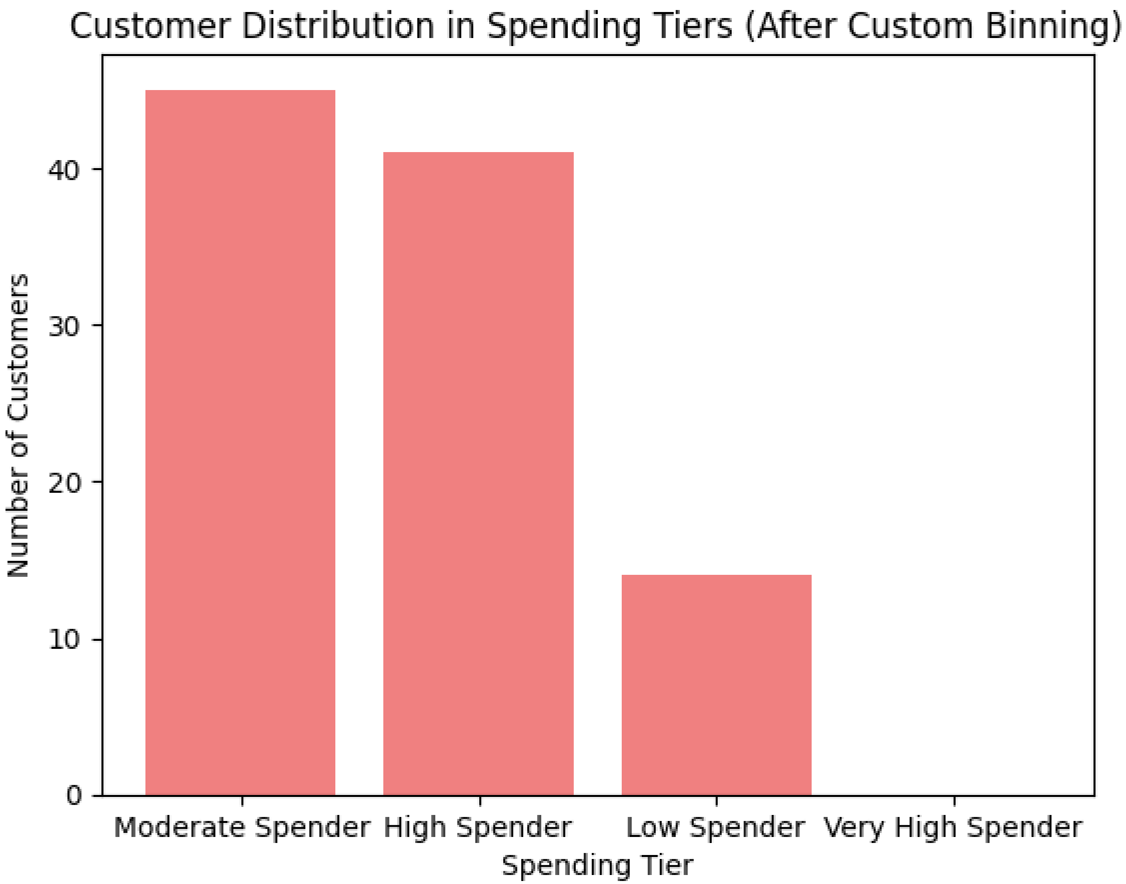
In [63]:
```
plt.bar(customer_data['SpendingTier'].value_counts().index, customer_data['SpendingTier'].value_counts(), color='light
plt.title('Customer Distribution in Spending Tiers (After Custom Binning)')
plt.xlabel('Spending Tier')
plt.ylabel('Number of Customers')
plt.show()
```



In [64]:
```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Prepare data for prediction
X_original = customer_data[['TotalPurchaseAmount']]
X_transformed = pd.get_dummies(customer_data[['SpendingTier']], drop_first=True)  # One-hot encode spending tiers
y = np.random.choice([0, 1], size=len(customer_data))  # Random binary classification labels

# Split data
X_original_train, X_original_test, X_transformed_train, X_transformed_test, y_train, y_test = train_test_split(
    X_original, X_transformed, y, test_size=0.2, random_state=42
)

# Train and predict on original data
dtc_original = DecisionTreeClassifier(random_state=42)
dtc_original.fit(X_original_train, y_train)
y_pred_original = dtc_original.predict(X_original_test)

# Train and predict on transformed data
dtc_transformed = DecisionTreeClassifier(random_state=42)
dtc_transformed.fit(X_transformed_train, y_train)
y_pred_transformed = dtc_transformed.predict(X_transformed_test)

# Compare accuracies
accuracy_original = accuracy_score(y_test, y_pred_original)
accuracy_transformed = accuracy_score(y_test, y_pred_transformed)

print(f'Accuracy - Original Data: {accuracy_original:.2%}')
print(f'Accuracy - Transformed Data: {accuracy_transformed:.2%}')
```

```
Accuracy - Original Data: 35.00%
Accuracy - Transformed Data: 35.00%
```

In [65]:
```
# Generate labels with more structure
customer_data['Label'] = np.where(customer_data['TotalPurchaseAmount'] >= 300, 1, 0)

# Adjust the order of spending tiers
spending_tiers = ['Low Spender', 'Moderate Spender', 'High Spender', 'Very High Spender']
customer_data['SpendingTier'] = pd.Categorical(customer_data['SpendingTier'], categories=spending_tiers, ordered=True)

# Visualize the distribution of spending tiers after adjusting the label generation
plt.bar(customer_data['SpendingTier'].value_counts().index, customer_data['SpendingTier'].value_counts(), color='light
plt.title('Customer Distribution in Spending Tiers (After Custom Binning)')
plt.xlabel('Spending Tier')
plt.ylabel('Number of Customers')
plt.show()
```



In [66]:
```
# Prepare data for prediction with adjusted labels
y_adjusted = customer_data['Label']

# Split data
X_original_train, X_original_test, X_transformed_train, X_transformed_test, y_train, y_test = train_test_split(
    X_original, X_transformed, y_adjusted, test_size=0.2, random_state=42
)

# Train and predict on original data
dtc_original = DecisionTreeClassifier(random_state=42)
dtc_original.fit(X_original_train, y_train)
y_pred_original = dtc_original.predict(X_original_test)

# Train and predict on transformed data
dtc_transformed = DecisionTreeClassifier(random_state=42)
dtc_transformed.fit(X_transformed_train, y_train)
y_pred_transformed = dtc_transformed.predict(X_transformed_test)

# Compare accuracies
accuracy_original = accuracy_score(y_test, y_pred_original)
accuracy_transformed = accuracy_score(y_test, y_pred_transformed)

print(f'Accuracy - Original Data: {accuracy_original:.2%}')
print(f'Accuracy - Transformed Data: {accuracy_transformed:.2%}')
```

```
Accuracy - Original Data: 100.00%
Accuracy - Transformed Data: 100.00%
```

I just created this as a dummy for just a rough idea.

In [ ]:

In [ ]:

In [ ]:

#Binarization

It is a special case of Discretization.
We convert a continuous value to discrete values but in Binarization we convert continuous values to binary values (0 or 1).
Ex: If annual income is < 6 Lakh then it don't come under taxable zone (0), but if it is > 6 Lakh then it comes under taxable zone (1).
It is very useful in some cases.
For ex: Image Processing where there are pixcels for an image which color ranges from (0-255).
While converting in Black and White image we create a threshold (e.g., 127.5):
if value < 127.5 then black (0), if > 127.5 then white(1).

We have class in scikit learn named 'Binarizer' in which we need to give two inputs (i) threshold (ii) copy

threshold -> for boundary (like below threshold value we can assign 0 and above threshold we can assign 1) and vice versa.
copy -> **True** or **False** (if **True**, it will create new columns with binarization implemented. if **False**, it will modify the same column on which we apply Binarizer)

Let's work practically to understand it.

Dataset -> Titanic

In [51]:
```
import pandas as pd
import numpy as np
```

In [52]:
```
from sklearn.model_selection import train_test_split,cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.compose import ColumnTransformer
```

In [53]:
```
data = pd.read_csv('titanic_dataset.csv')
```

In [54]:
```
data
```

Out[54]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
...	...	...	...	...	...	...	...	...	...	...	...	...
886	887	0	2	Montvila, Rev. Juozas	male	27.0	0	0	211536	13.0000	NaN	S
887	888	1	1	Graham, Miss. Margaret Edith	female	19.0	0	0	112053	30.0000	B42	S
888	889	0	3	Johnston, Miss. Catherine Helen "Carrie"	female	NaN	1	2	W./C. 6607	23.4500	NaN	S
889	890	1	1	Behr, Mr. Karl Howell	male	26.0	0	0	111369	30.0000	C148	C
890	891	0	3	Dooley, Mr. Patrick	male	32.0	0	0	370376	7.7500	NaN	Q

891 rows × 12 columns

In [55]:
```
data = data[['Age', 'Fare', 'SibSp', 'Parch', 'Survived']]
```

In [56]:
```
data
```

Out[56]:

	Age	Fare	SibSp	Parch	Survived
0	22.0	7.2500	1	0	0
1	38.0	71.2833	1	0	1
2	26.0	7.9250	0	0	1
3	35.0	53.1000	1	0	1
4	35.0	8.0500	0	0	0
...	...	...	...	...	...
886	27.0	13.0000	0	0	0
887	19.0	30.0000	0	0	1
888	NaN	23.4500	1	2	0
889	26.0	30.0000	0	0	1
890	32.0	7.7500	0	0	0

891 rows × 5 columns

Chose columns **Age, Fare, SibSp, Parch, Survived** because these parameters make sense for prediction.
Age -> Denotes age of the passenger.
Fare -> Ticket fare.
SibSp -> Sibling Spouse.
Parch -> Parent Child.
Survived -> Survived or not.

In [57]:
```
data.isna().sum()
```

Out[57]:
```
Age         177
Fare          0
SibSp         0
Parch         0
Survived      0
dtype: int64
```

As there are some missing values which we can see, let's drop these values.

In [58]:
```
data['Family'] = data['SibSp'] + data['Parch']
```

```
<ipython-input-58-1c7da3b7290a>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  data['Family'] = data['SibSp'] + data['Parch']
```

data['Family'] = data['SibSp'] + data['Parch']
Performed feature extraction where we know sibling spouse and parent child lives in same family, so I assemble them.

In [59]:
```
data
```

Out[59]:

	Age	Fare	SibSp	Parch	Survived	Family
0	22.0	7.2500	1	0	0	1
1	38.0	71.2833	1	0	1	1
2	26.0	7.9250	0	0	1	0
3	35.0	53.1000	1	0	1	1
4	35.0	8.0500	0	0	0	0
...	...	...	...	...	...	...
886	27.0	13.0000	0	0	0	0
887	19.0	30.0000	0	0	1	0
888	NaN	23.4500	1	2	0	3
889	26.0	30.0000	0	0	1	0
890	32.0	7.7500	0	0	0	0

891 rows × 6 columns

Now we can remove Sibling Spouse and Parent Child from the dataset as we assembled them in family.

In [60]:
```
data.drop(columns=['SibSp', 'Parch'], inplace=True)
```

```
<ipython-input-60-1350b0175899>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  data.drop(columns=['SibSp', 'Parch'], inplace=True)
```

In [61]:
```
data
```

Out[61]:

	Age	Fare	Survived	Family
0	22.0	7.2500	0	1
1	38.0	71.2833	1	1
2	26.0	7.9250	1	0
3	35.0	53.1000	1	1
4	35.0	8.0500	0	0
...	...	...	...	...
886	27.0	13.0000	0	0
887	19.0	30.0000	1	0
888	NaN	23.4500	0	3
889	26.0	30.0000	1	0
890	32.0	7.7500	0	0

891 rows × 4 columns

In [62]:
```
data.dropna(inplace=True)
```

```
<ipython-input-62-f1116dacf2bb>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy)
  data.dropna(inplace=True)
```

In [63]:
```
X = data[['Age', 'Fare', 'Family']]
y = data['Survived']
```

Here,
X -> independent variables.
y -> dependent variables.

In [64]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [65]:
```
X_train
```

Out[65]:

	Age	Fare	Family
328	31.0	20.5250	2
73	26.0	14.4542	1
253	30.0	16.1000	1
719	33.0	7.7750	0
666	25.0	13.0000	0
...	...	...	...
92	46.0	61.1750	1
134	25.0	13.0000	0
337	41.0	134.5000	0
548	33.0	20.5250	2
130	33.0	7.8958	0

571 rows × 3 columns

In [66]:
```
y_train
```

Out[66]:
```
328    1
73     0
253    0
719    0
666    0
      ..
92     0
134    0
337    1
548    0
130    0
Name: Survived, Length: 571, dtype: int64
```

In [67]:
```
X_test
```

Out[67]:

	Age	Fare	Family
149	42.0	13.0000	0
407	3.0	18.7500	2
53	29.0	26.0000	1
369	24.0	69.3000	0
818	43.0	6.4500	0
...	...	...	...
819	10.0	27.9000	5
164	1.0	39.6875	5
363	35.0	7.0500	0
56	21.0	10.5000	0
136	19.0	26.2833	2

143 rows × 3 columns

In [68]:
```
y_test
```

Out[68]:
```
149    0
407    1
53     1
369    1
818    0
      ..
819    0
164    0
363    0
56     1
136    1
Name: Survived, Length: 143, dtype: int64
```

In [68]:

####WITHOUT BINARIZATION

In [69]:
```
DT = DecisionTreeClassifier()
```

In [70]:
```
DT.fit(X_train, y_train)
```

Out[70]:
```
DecisionTreeClassifier()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [71]:
```
y_pred = DT.predict(X_test)
```

Used Decision Tree Classifier algorithm, fit the training data and predicted results for testing data

In [72]:
```
accuracy_score(y_pred, y_test)
```

Out[72]:
```
0.6363636363636364
```

Here, the accuracy score comes out around as 64%

Lets cross validate.

In [73]:
```
np.mean(cross_val_score(DecisionTreeClassifier(),X,y,cv=10,scoring='accuracy'))
```

Out[73]:
```
0.645696400625978
```

Cross validation score comes out as 65%.

In [73]:

####WITH BINARIZER

In [74]:
```
from sklearn.preprocessing import Binarizer
```

If the passenger is travelling alone or with family let's modify the value where,
if 'Family' = 0 (travelling alone)
if 'Family' > 0 (travelling with family)

In [75]:
```
CT = ColumnTransformer([('Bin', Binarizer(copy=False),
                         ['Family'])], remainder='passthrough')
```

Created transformer,
CT = ColumnTransformer([('Bin', Binarizer(copy=False), ['Family'])], remainder='passthrough') Created Column Transformer and applied Binarizer on 'Family' and set copy as False which denotes that modify within same column and rest of columns must be as it is.

Now it's time to apply this transformer.
So I fit and transform training data and testing data and stored changes in X_train_CT and X_test_CT.

In [76]:
```
X_train_CT = CT.fit_transform(X_train)
X_test_CT = CT.transform(X_test)
```

This X_train_CT and X_test_CT is in format of array, let's convert it into dataframe.

In [77]:
```
pd.DataFrame(X_train_CT,columns=['family','Age','Fare'])
```

Out[77]:

	family	Age	Fare
0	1.0	31.0	20.5250
1	1.0	26.0	14.4542
2	1.0	30.0	16.1000
3	0.0	33.0	7.7750
4	0.0	25.0	13.0000
...	...	...	...
566	1.0	46.0	61.1750
567	0.0	25.0	13.0000
568	0.0	41.0	134.5000
569	1.0	33.0	20.5250
570	0.0	33.0	7.8958

571 rows × 3 columns

In [78]:
```
DT_1 = DecisionTreeClassifier()
```

In [79]:
```
DT_1.fit(X_train_CT, y_train)
```

Out[79]:
```
DecisionTreeClassifier()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

In [80]:
```
y_pred_CT = DT_1.predict(X_test_CT)
```

In [81]:
```
accuracy_score(y_test, y_pred_CT)
```

Out[81]:
```
0.6153846153846154
```

Again applied Decision Tree Classifier and fit trained transformed data and predicted results for testing data.
There is not improvement in accuracy i.e., 62%.

In [82]:
```
X_CT = CT.fit_transform(X)
```

In [83]:
```
np.mean(cross_val_score(DecisionTreeClassifier(),X_CT,y,cv=10,scoring='accuracy'))
```

Out[83]:
```
0.6276212832550861
```

Then fit and transformed our independent data X and cross validate it.
i.e., 63%

In [ ]:

In [ ]:

In [ ]:

#Handling Mixed Values

What is mixed data?
In our column, there is both type of data, numerical and categorical, and this problem really make us difficult to proceed.

Problems:

(i)
Just take an example of Train Tickets
Me and two of my friends Abhi and Datta booked railways ticket.

Name	Seat no.
Mohit	B5
Abhi	A2
Datta	C7

Now here we can't categorize for this kind of data else many more categories will be created.
For ex: B5 [where B is Class and 5 is Seat no.]
So we can say,
5 is numerical data.
So what we can do is create seperate column for both categorical and numerical data.

Name	Seat no.	Category	Numeric
Mohit	B5	B	5
Abhi	A2	A	2
Datta	C7	C	7

(ii)

Data

```
|    7     |
+----------+
|    3     |
+----------+
|    1     |
+----------+
|    A     |
+----------+
|    C     |
+----------+
|    4     |
+----------+
|    D     |
+----------+
```

```
Here some values are numerical and some are categorical.
So what to do in such cases, where to proceed?
So we can create seperate column for categoric and numeric data like:

+----------+----------+
|  Numeric | Categoric|
+----------+----------+
|    7     |    NA    |
+----------+----------+
|    3     |    NA    |
+----------+----------+
|    1     |    NA    |
```

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
titanic_dataset = pd.read_csv('titanic_1_dataset.csv')
```

In [3]:
```
titanic_dataset
```

Out[3]:

	Cabin	Ticket	number	Survived
0	NaN	A/5 21171	5	0
1	C85	PC 17599	3	1
2	NaN	STON/O2. 3101282	6	1
3	C123	113803	3	1
4	NaN	373450	A	0
...	...	...	...	...
886	NaN	211536	3	0
887	B42	112053	3	1
888	NaN	W./C. 6607	1	0
889	C148	111369	2	1
890	NaN	370376	3	0

891 rows × 4 columns

Cabin -> Both numerical and categorical values in single cell.
Ticket -> Both numerical and Categorical values in single cell.
number ->
if A: it means the person is travelling alone,
if it numeric then person is travelling with those number of people.

Let's first call the unique values of numbers

In [4]:
```
titanic_dataset['number'].unique()
```

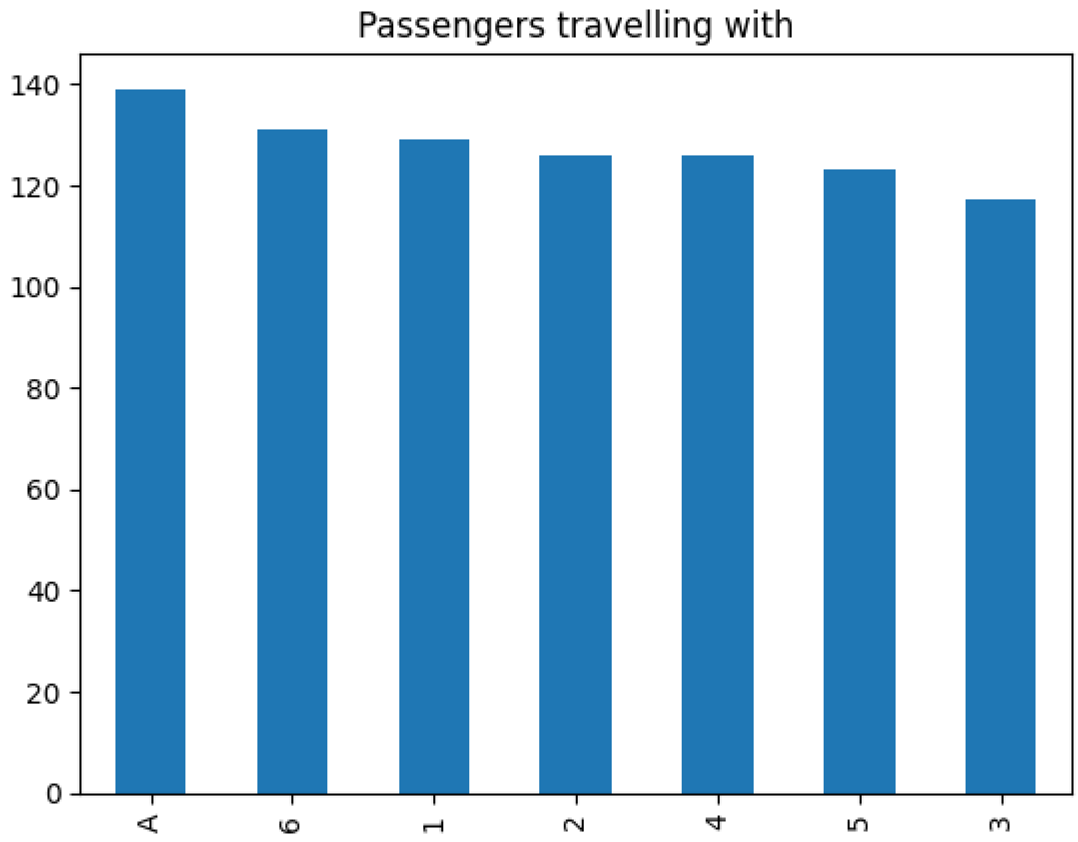
Out[4]:
```
array(['5', '3', '6', 'A', '2', '1', '4'], dtype=object)
```

So as we can see, there are few values which are unique values in column **'number'**

Lets plot,

In [5]:
```
fig = titanic_dataset['number'].value_counts().plot.bar()
fig.set_title('Passengers travelling with')
```

Out[5]:
```
Text(0.5, 1.0, 'Passengers travelling with')
```



The plot here is for counts for each unique values.

If in data where in single column there is numerical data and categorical data but not in single cell

In [6]:
```
# extract numerical part

titanic_dataset['number_numerical'] = pd.to_numeric(titanic_dataset["number"],errors='coerce',downcast='integer')
```

So I made new numerical column 'number_numerical' and call function pd.to_numeric and pass column number, coerce the errors and downcast to integers.
Here in column 'number_numeric' there will be numerical values.

In [7]:
```
# extract categorical part

titanic_dataset['number_categorical'] = np.where(titanic_dataset['number_numerical'].isnull(),titanic_dataset['number'
```

Created column 'number_categorical' based on the values in the column 'number_numerical'. If a value in 'number_numerical' is null, it fills the corresponding entry in 'number_categorical' with the value from the column 'number'. If the value in 'number_numerical' is not null, it fills the entry with NaN. This is a way to handle missing values in a dataset.

titanic_dataset['number_numerical'].isnull():
This part checks if the values in the column 'number_numerical' are null (i.e., missing or NaN). The result is a boolean mask where True indicates a missing value.

np.where():
This function is a vectorized way to perform an if-else operation. The syntax is np.where(condition, x, y), where it returns elements chosen from x or y depending on the condition.

titanic_dataset['number']:
This is the value to be assigned to the new column 'number_categorical' where the condition is True (i.e., where 'number_numerical' is null). It takes the values from the existing column 'number'.

np.nan:
This is the value to be assigned to the new column 'number_categorical' where the condition is False (i.e., where 'number_numerical' is not null). It assigns NaN (Not a Number) to represent missing values.

In [8]:
```
titanic_dataset.head()
```

Out[8]:

	Cabin	Ticket	number	Survived	number_numerical	number_categorical
0	NaN	A/5 21171	5	0	5.0	NaN
1	C85	PC 17599	3	1	3.0	NaN
2	NaN	STON/O2. 3101282	6	1	6.0	NaN
3	C123	113803	3	1	3.0	NaN
4	NaN	373450	A	0	NaN	A

Let's find out unique values for Cabin, so let's just implement.

In [9]:
```
titanic_dataset['Cabin'].unique()
```

Out[9]:
```
array([nan, 'C85', 'C123', 'E46', 'G6', 'C103', 'D56', 'A6',
       'C23 C25 C27', 'B78', 'D33', 'B30', 'C52', 'B28', 'C83', 'F33',
       'F G73', 'E31', 'A5', 'D10 D12', 'D26', 'C110', 'B58 B60', 'E101',
       'F E69', 'D47', 'B86', 'F2', 'C2', 'E33', 'B19', 'A7', 'C49', 'F4',
       'A32', 'B4', 'B80', 'A31', 'D36', 'D15', 'C93', 'C78', 'D35',
       'C87', 'B77', 'E67', 'B94', 'C125', 'C99', 'C118', 'D7', 'A19',
       'B49', 'D', 'C22 C26', 'C106', 'C65', 'E36', 'C54',
       'B57 B59 B63 B66', 'C7', 'E34', 'C32', 'B18', 'C124', 'C91', 'E40',
       'T', 'C128', 'D37', 'B35', 'E50', 'C82', 'B96 B98', 'E10', 'E44',
       'A34', 'C104', 'C111', 'C92', 'E38', 'D21', 'E12', 'E63', 'A14',
       'B37', 'C30', 'D20', 'B79', 'E25', 'D46', 'B73', 'C95', 'B38',
       'B39', 'B22', 'C86', 'C70', 'A16', 'C101', 'C68', 'A10', 'E68',
       'B41', 'A20', 'D19', 'D50', 'D9', 'A23', 'B50', 'A26', 'D48',
       'E58', 'C126', 'B71', 'B51 B53 B55', 'D49', 'B5', 'B20', 'F G63',
       'C62 C64', 'E24', 'C90', 'C45', 'E8', 'B101', 'D45', 'C46', 'D30',
       'E121', 'D11', 'E77', 'F38', 'B3', 'D6', 'B82 B84', 'D17', 'A36',
       'B102', 'B69', 'E49', 'C47', 'D28', 'E17', 'A24', 'C50', 'B42',
       'C148'], dtype=object)
```

Using .unique() function I find all the unique values from column 'Cabin'.
Seeing it's results one thing is very clear that we can't proceed with these much categories as count of unique values is very high.

In [10]:
```
titanic_dataset['Ticket'].unique()
```

Out[10]:
```
array(['A/5 21171', 'PC 17599', 'STON/O2. 3101282', '113803', '373450',
       '330877', '17463', '349909', '347742', '237736', 'PP 9549',
       '113783', 'A/5. 2151', '347082', '350406', '248706', '382652',
       '244373', '345763', '2649', '239865', '248698', '330923', '113788',
       '347077', '2631', '19950', '330959', '349216', 'PC 17601',
       'PC 17569', '335677', 'C.A. 24579', 'PC 17604', '113789', '2677',
       'A./5. 2152', '345764', '2651', '7546', '11668', '349253',
       'SC/Paris 2123', '330958', 'S.C./A.4. 23567', '370371', '14311',
       '2662', '349237', '3101295', 'A/4. 39886', 'PC 17572', '2926',
       '113509', '19947', 'C.A. 31026', '2697', 'C.A. 34651', 'CA 2144',
       '2669', '113572', '36973', '347088', 'PC 17605', '2661',
       'C.A. 29395', 'S.P. 3464', '3101281', '315151', 'C.A. 33111',
       'S.O.C. 14879', '2680', '1601', '348123', '349208', '374746',
       '248738', '364516', '345767', '345779', '330932', '113059',
       'SO/C 14885', '3101278', 'W./C. 6608', 'SOTON/OQ 392086', '343275',
       '343276', '347466', 'W.E.P. 5734', 'C.A. 2315', '364500', '374910',
       'PC 17754', 'PC 17759', '231919', '244367', '349245', '349215',
       '35281', '7540', '3101276', '349207', '343120', '312991', '349249',
       '371110', '110465', '2665', '324669', '4136', '2627',
       'STON/O 2. 3101294', '370369', 'PC 17558', 'A4. 54510', '27267',
       '370372', 'C 17369', '2668', '347061', '349241',
       'SOTON/O.Q. 3101307', 'A/5. 3337', '228414', 'C.A. 29178',
       'SC/PARIS 2133', '11752', '7534', 'PC 17593', '2678', '347081',
       'STON/O2. 3101279', '365222', '231945', 'C.A. 33112', '350043',
       '230080', '244310', 'S.O.P. 1166', '113776', 'A.5. 11206',
       'A/5. 851', 'Fa 265302', 'PC 17597', '35851', 'SOTON/OQ 392090',
       '315037', 'CA. 2343', '371362', 'C.A. 33595', '347068', '315093',
       '363291', '113505', 'PC 17318', '111240', 'STON/O 2. 3101280',
       '17764', '350404', '4133', 'PC 17595', '250653', 'LINE',
       'SC/PARIS 2131', '230136', '315153', '113767', '370365', '111428',
       '364849', '349247', '234604', '28424', '350046', 'PC 17610',
       '368703', '4579', '370370', '248747', '345770', '3101264', '2628',
       'A/5 3540', '347054', '2699', '367231', '112277',
       'SOTON/O.Q. 3101311', 'F.C.C. 13528', 'A/5 21174', '250646',
       '367229', '35273', 'STON/O2. 3101283', '243847', '11813',
       'W/C 14208', 'SOTON/OQ 392089', '220367', '21440', '349234',
       '19943', 'PP 4348', 'SW/PP 751', 'A/5 21173', '236171', '347067',
       '237442', 'C.A. 29566', 'W./C. 6609', '26707', 'C.A. 31921',
       '28665', 'SCO/W 1585', '367230', 'W./C. 14263',
       'STON/O 2. 3101275', '2694', '19928', '347071', '250649', '11751',
       '244252', '362316', '113514', 'A/5. 3336', '370129', '2650',
```

```
'PC 17585', '110152', 'PC 17755', '230433', '384461', '110413',
'112059', '382649', 'C.A. 17248', '347083', 'PC 17582', 'PC 17760',
'113798', '250644', 'PC 17596', '370375', '13502', '347073',
'239853', 'C.A. 2673', '336439', '347464', '345778', 'A/5. 10482',
'113056', '349239', '345774', '349206', '237798', '370373',
'19877', '11967', 'SC/Paris 2163', '349236', '349233', 'PC 17612',
'2693', '113781', '19988', '9234', '367226', '226593', 'A/5 2466',
'17421', 'PC 17758', 'P/PP 3381', 'PC 17485', '11767', 'PC 17608',
'250651', '349243', 'F.C.C. 13529', '347470', '29011', '36928',
'16966', 'A/5 21172', '349219', '234818', '345364', '28551',
'111361', '113043', 'PC 17611', '349225', '7598', '113784',
'248740', '244361', '229236', '248733', '31418', '386525',
'C.A. 37671', '315088', '7267', '113510', '2695', '2647', '345783',
'237671', '330931', '330980', 'SC/PARIS 2167', '2691',
'SOTON/O.Q. 3101310', 'C 7076', '110813', '2626', '14313',
'PC 17477', '11765', '3101267', '323951', 'C 7077', '113503',
'2648', '347069', 'PC 17757', '2653', 'STON/O 2. 3101293',
'349227', '27849', '367655', 'SC 1748', '113760', '350034',
'3101277', '350052', '350407', '28403', '244278', '240929',
'STON/O 2. 3101289', '341826', '4137', '315096', '28664', '347064',
'29106', '312992', '349222', '394140', 'STON/O 2. 3101269',
'343095', '28220', '250652', '28228', '345773', '349254',
'A/5. 13032', '315082', '347080', 'A/4. 34244', '2003', '250655',
'364851', 'SOTON/O.Q. 392078', '110564', '376564', 'SC/AH 3085',
'STON/O 2. 3101274', '13507', 'C.A. 18723', '345769', '347076',
'230434', '65306', '33638', '113794', '2666', '113786', '65303',
'113051', '17453', 'A/5 2817', '349240', '13509', '17464',
'F.C.C. 13531', '371060', '19952', '364506', '111320', '234360',
'A/S 2816', 'SOTON/O.Q. 3101306', '113792', '36209', '323592',
'315089', 'SC/AH Basle 541', '7553', '31027', '3460', '350060',
'3101298', '239854', 'A/5 3594', '4134', '11771', 'A.5. 18509',
'65304', 'SOTON/OQ 3101317', '113787', 'PC 17609', 'A/4 45380',
'36947', 'C.A. 6212', '350035', '315086', '364846', '330909',
'4135', '26360', '111427', 'C 4001', '382651', 'SOTON/OQ 3101316',
'PC 17473', 'PC 17603', '349209', '36967', 'C.A. 34260', '226875',
'349242', '12749', '349252', '2624', '2700', '367232',
'W./C. 14258', 'PC 17483', '3101296', '29104', '2641', '2690',
'315084', '113050', 'PC 17761', '364498', '13568', 'WE/P 5735',
'2908', '693', 'SC/PARIS 2146', '244358', '330979', '2620',
'347085', '113807', '11755', '345572', '372622', '349251',
'218629', 'SOTON/OQ 392082', 'SOTON/O.Q. 392087', 'A/4 48871',
'349205', '2686', '350417', 'S.W./PP 752', '11769', 'PC 17474',
```

```
'14312', 'A/4. 20589', '358585', '243880', '2689',
'STON/O 2. 3101286', '237789', '13049', '3411', '237565', '13567',
'14973', 'A./5. 3235', 'STON/O 2. 3101273', 'A/5 3902', '364848',
'SC/AH 29037', '248727', '2664', '349214', '113796', '364511',
'111426', '349910', '349246', '113804', 'SOTON/O.Q. 3101305',
'370377', '364512', '220845', '31028', '2659', '11753', '350029',
'54636', '36963', '219533', '349224', '334912', '27042', '347743',
'13214', '112052', '237668', 'STON/O 2. 3101292', '350050',
'349231', '13213', 'S.O./P.P. 751', 'CA. 2314', '349221', '8475',
'330919', '365226', '349223', '29751', '2623', '5727', '349210',
'STON/O 2. 3101285', '234686', '312993', 'A/5 3536', '19996',
'29750', 'F.C. 12750', 'C.A. 24580', '244270', '239856', '349912',
'342826', '4138', '330935', '6563', '349228', '350036', '24160',
'17474', '349256', '2672', '113800', '248731', '363592', '35852',
'348121', 'PC 17475', '36864', '350025', '223596', 'PC 17476',
'PC 17482', '113028', '7545', '250647', '348124', '34218', '36568',
'347062', '350048', '12233', '250643', '113806', '315094', '36866',
'236853', 'STON/O2. 3101271', '239855', '28425', '233639',
'349201', '349218', '16988', '376566', 'STON/O 2. 3101288',
'250648', '113773', '335097', '29103', '392096', '345780',
'349204', '350042', '29108', '363294', 'SOTON/O2 3101272', '2663',
'347074', '112379', '364850', '8471', '345781', '350047',
'S.O./P.P. 3', '2674', '29105', '347078', '383121', '36865',
'2687', '113501', 'W./C. 6607', 'SOTON/O.Q. 3101312', '374887',
'3101265', '12460', 'PC 17600', '349203', '28213', '17465',
'349244', '2685', '2625', '347089', '347063', '112050', '347087',
'248723', '3474', '28206', '364499', '112058', 'STON/O2. 3101290',
'S.C./PARIS 2079', 'C 7075', '315098', '19972', '368323', '367228',
'2671', '347468', '2223', 'PC 17756', '315097', '392092', '11774',
'SOTON/O2 3101287', '2683', '315090', 'C.A. 5547', '349213',
'347060', 'PC 17592', '392091', '113055', '2629', '350026',
'28134', '17466', '233866', '236852', 'SC/PARIS 2149', 'PC 17590',
'345777', '349248', '695', '345765', '2667', '349212', '349217',
'349257', '7552', 'C.A./SOTON 34068', 'SOTON/OQ 392076', '211536',
'112053', '111369', '370376'], dtype=object)
```

Similarly when applying .unique() function on 'Ticket' the result comes out much more large (as there are many unique values)

Now, let's try to handle these,
So what I can do is,

In [11]:
```
titanic_dataset['cabin_num'] = titanic_dataset['Cabin'].str.extract('(\d+)')    # captures numerical part
titanic_dataset['cabin_cat'] = titanic_dataset['Cabin'].str[0]                   # captures the first letter
```

I created column 'cabin_num' in which I stored numeric part from column 'Cabin'.
Here, str.extract('(\d+)'), where \d+ -> captures numeric part of data.
Similarly, I created column 'cabin_cat' in which I stored first value from string.
e.x., C10 (so str[0] is C)
So like this str[0] fetches first value from string from column 'Cabin'.

In [12]:
```
titanic_dataset.head()
```

Out[12]:

	Cabin	Ticket	number	Survived	number_numerical	number_categorical	cabin_num	cabin_cat
0	NaN	A/5 21171	5	0	5.0	NaN	NaN	NaN
1	C85	PC 17599	3	1	3.0	NaN	85	C
2	NaN	STON/O2. 3101282	6	1	6.0	NaN	NaN	NaN
3	C123	113803	3	1	3.0	NaN	123	C
4	NaN	373450	A	0	NaN	A	NaN	NaN

Now, we get proper categories,

In [13]:
```
titanic_dataset['cabin_cat'].value_counts()
```

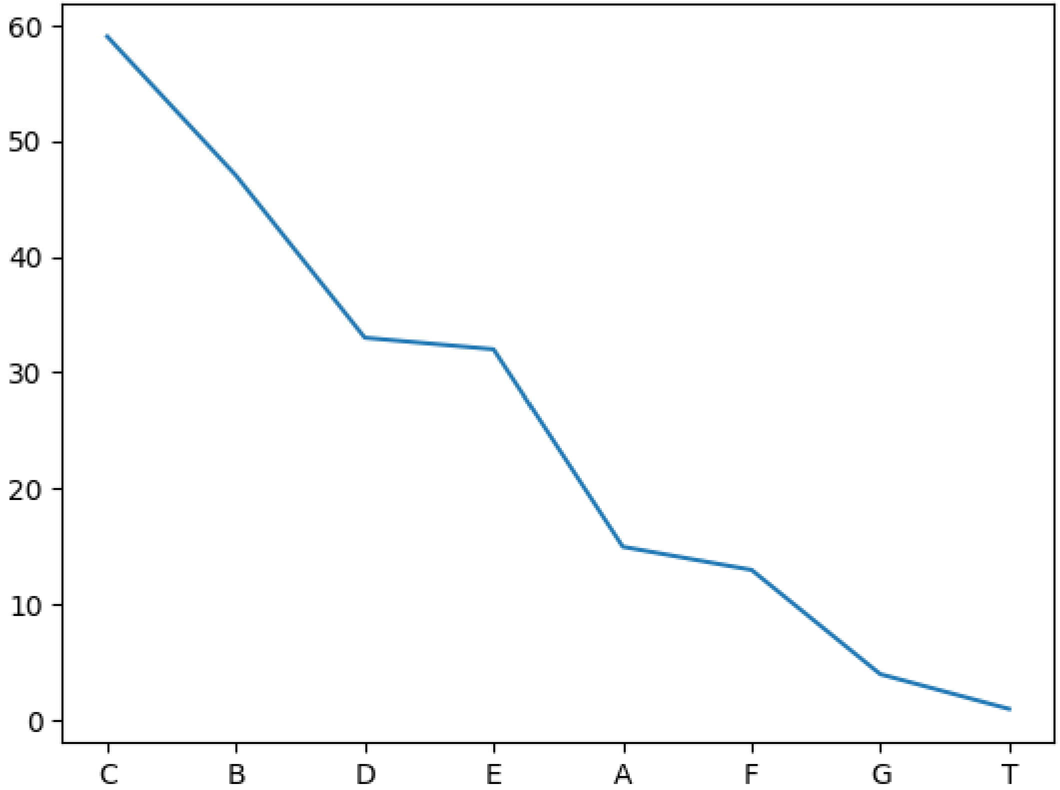
Out[13]:
```
C    59
B    47
D    33
E    32
A    15
F    13
G     4
T     1
Name: cabin_cat, dtype: int64
```

Here, titanic_data['cabin_cat'].value_counts() gives us the proper categories.

We can also plot it using plot() function

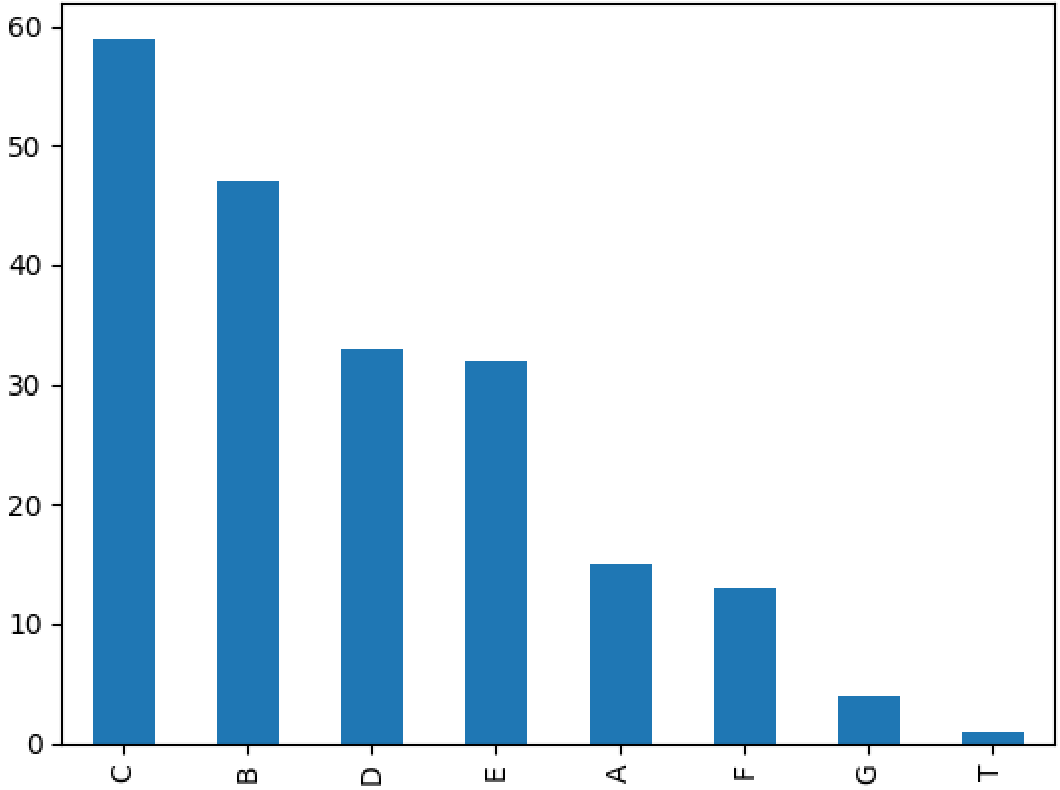
In [19]:
```
titanic_dataset['cabin_cat'].value_counts().plot()
```

Out[19]: <Axes: >



In [14]:
```
titanic_dataset['cabin_cat'].value_counts().plot(kind='bar')
```

Out[14]: <Axes: >



Here we can see the count for each category.

So this made us easy to process.

In [15]:
```
# extract the last bit of ticket as number

titanic_dataset['ticket_num'] = titanic_dataset['Ticket'].apply(lambda s: s.split()[-1])
titanic_dataset['ticket_num'] = pd.to_numeric(titanic_dataset['ticket_num'], errors='coerce', downcast='integer')
```

Created a new column 'ticket_num' in the 'titanic_dataset' DataFrame, containing the last part of each 'Ticket' value.

titanic_dataset['Ticket']:
selects the **'Ticket'** column from the **'titanic_dataset'** DataFrame. The **'Ticket'** column typically contains mixed values.

.apply(lambda s: s.split()[-1]):
This applies a lambda function to each element in the 'Ticket' column. The lambda function **lambda s: s.split()[-1]** splits the string into a list using whitespaces as separators (**split()**), and then extracts the last element (**[-1]**). This is done to extract the last part of the ticket, which contain a numerical value.

titanic_dataset['ticket_num']:
Update a column **'ticket_num'** in the **'titanic_dataset'** DataFrame and assigns the results of the lambda function to this new column.

pd.to_numeric(titanic_dataset['ticket_num'], errors='coerce', downcast='integer'):
Used the **pd.to_numeric** function from pandas to convert the values in the **'ticket_num'** column to numeric format. **errors='coerce'** parameter is used to replace any values that cannot be converted to numeric with **NaN** (Not a Number). **downcast='integer'** parameter is used to downcast the resulting numeric values to integers, if possible.

titanic_dataset['ticket_num']:
This updates the 'ticket_num' column in the 'titanic_dataset' DataFrame with the converted numeric values.

Let's extract first part of ticket as Category.
Then Converts the values in the 'ticket_num' column to numeric format, replacing any non-numeric values with NaN and downcasting the numeric values to integers.

In [16]:
```
# extract the first part of ticket as category

titanic_dataset['ticket_cat'] = titanic_dataset['Ticket'].apply(lambda s: s.split()[0])
titanic_dataset['ticket_cat'] = np.where(titanic_dataset['ticket_cat'].str.isdigit(), np.nan, titanic_dataset['ticket_
```

So I created column 'ticket_cat' and applied split() function for first bit i.e., [0]
Here I applied lambda where I splitted the string s (i.e., values of 'Ticket')

lambda s: s.split()[0]: This is an anonymous (lambda) function that takes a string s (each ticket value in this context), splits it using whitespace as the separator (.split()), and then retrieves the first part (index 0) of the resulting list.

titanic_dataset['ticket_cat'].str.isdigit(): This checks if each element in the **'ticket_cat'** column is composed entirely of digits i.e., numeric characters.

np.where(): This NumPy function that performs element-wise conditional operations. It takes three arguments:

- titanic_dataset['ticket_cat'].str.isdigit() ->
  first argument is the condition to check (in this case, whether the element is a digit).
- np.nan ->
  second argument is the value to assign if the condition is True (in this case, np.nan, which represents a missing or undefined value).
- titanic_dataset['ticket_cat'] ->
  third argument is the value to assign if the condition is False (in this case, the original value from **'ticket_cat'**).

titanic_dataset['ticket_cat'] = :
This assigns the results of the np.where operation back to the 'ticket_cat' column, effectively replacing numeric values with np.nan and leaving non-numeric values unchanged.

In [17]:
```
titanic_dataset.head(20)
```

Out[17]:

	Cabin	Ticket	number	Survived	number_numerical	number_categorical	cabin_num	cabin_cat	ticket_num	ticket_cat
0	NaN	A/5 21171	5	0	5.0	NaN	NaN	NaN	21171.0	A/5
1	C85	PC 17599	3	1	3.0	NaN	85	C	17599.0	PC
2	NaN	STON/O2. 3101282	6	1	6.0	NaN	NaN	NaN	3101282.0	STON/O2.
3	C123	113803	3	1	3.0	NaN	123	C	113803.0	NaN
4	NaN	373450	A	0	NaN	A	NaN	NaN	373450.0	NaN
5	NaN	330877	2	0	2.0	NaN	NaN	NaN	330877.0	NaN
6	E46	17463	2	0	2.0	NaN	46	E	17463.0	NaN
7	NaN	349909	5	0	5.0	NaN	NaN	NaN	349909.0	NaN
8	NaN	347742	1	1	1.0	NaN	NaN	NaN	347742.0	NaN
9	NaN	237736	A	1	NaN	A	NaN	NaN	237736.0	NaN
10	G6	PP 9549	1	1	1.0	NaN	6	G	9549.0	PP
11	C103	113783	1	1	1.0	NaN	103	C	113783.0	NaN
12	NaN	A/5. 2151	3	0	3.0	NaN	NaN	NaN	2151.0	A/5.
13	NaN	347082	3	0	3.0	NaN	NaN	NaN	347082.0	NaN
14	NaN	350406	5	0	5.0	NaN	NaN	NaN	350406.0	NaN
15	NaN	248706	3	1	3.0	NaN	NaN	NaN	248706.0	NaN
16	NaN	382652	3	0	3.0	NaN	NaN	NaN	382652.0	NaN
17	NaN	244373	2	1	2.0	NaN	NaN	NaN	244373.0	NaN
18	NaN	345763	5	0	5.0	NaN	NaN	NaN	345763.0	NaN
19	NaN	2649	4	1	4.0	NaN	NaN	NaN	2649.0	NaN

In [18]:
```
titanic_dataset['ticket_cat'].unique()
```

Out[18]:
```
array(['A/5', 'PC', 'STON/O2.', nan, 'PP', 'A/5.', 'C.A.', 'A./5.',
       'SC/Paris', 'S.C./A.4.', 'A/4.', 'CA', 'S.P.', 'S.O.C.', 'SO/C',
       'W./C.', 'SOTON/OQ', 'W.E.P.', 'STON/O', 'A4.', 'C', 'SOTON/O.Q.',
       'SC/PARIS', 'S.O.P.', 'A.5.', 'Fa', 'CA.', 'LINE', 'F.C.C.', 'W/C',
       'SW/PP', 'SCO/W', 'P/PP', 'SC', 'SC/AH', 'A/S', 'A/4', 'WE/P',
       'S.W./PP', 'S.O./P.P.', 'F.C.', 'SOTON/O2', 'S.C./PARIS',
       'C.A./SOTON'], dtype=object)
```

In [18]:
```
```

#Handling Date and Time variables

Ex:

```
                                Date : 23 March 2023
    -------------------------------------------|---------------------------------------------
    |            |            |           |            |           |            |
    Day          Month        Year        Day of week  Quarter     Semester     Decade Year  Weekend
```

```
                              Time : 10 : 30
    _____________________________________|_____________________________________
    |                                    |                                    |
    Hour                                 Minutes                              Seconds
```

I will be working on two datasets
(i) orders_dataset: E-commerce dataset
(ii) messages.csv: Chat dataset

Date related work I will be doing in orders_dataset.csv
Time related work I will be doing in messages_dataset.csv

In [20]:
```
import pandas as pd
import numpy as np
import datetime
```

In [21]:
```
Order = pd.read_csv('orders_dataset.csv')
```

In [22]:
```
Order
```

Out[22]:

	date	product_id	city_id	orders
0	2019-12-10	5628	25	3
1	2018-08-15	3646	14	157
2	2018-10-23	1859	25	1
3	2019-08-17	7292	25	1
4	2019-01-06	4344	25	3
...	...	...	...	...
995	2018-10-08	255	13	1
996	2018-12-06	5521	7	1
997	2019-05-07	487	26	14
998	2019-03-03	1503	21	2
999	2019-10-15	6371	7	22

1000 rows × 4 columns

orders_dataset.csv :
date -> Denotes date.
product_id -> denotes id of product.
city_id -> denotes city code from where product is ordered.
orders -> how much sold.

In [23]:
```
Chat = pd.read_csv('messages_dataset.csv')
```

In [24]:
```
Chat
```

Out[24]:

	date	msg
0	2013-12-15 00:50:00	ищу на сегодня мужика 37
1	2014-04-29 23:40:00	ПАРЕНЬ БИ ИЩЕТ ДРУГА СЕЙЧАС!! СМС ММС 0955532826
2	2012-12-30 00:21:00	Днепр.м 43 позн.с д/ж *.о 067.16.34.576
3	2014-11-28 00:31:00	КИЕВ ИЩУ Д/Ж ДО 45 МНЕ СЕЙЧАС СКУЧНО 093 629 9...
4	2013-10-26 23:11:00	Зая я тебя никогда не обижу люблю тебя!) Даше
...	...	...
995	2012-03-16 00:50:00	ПАРЕНЬ СДЕЛАЕТ МАССАЖ ЖЕНЩИНАМ -066-877-32-44
996	2014-01-23 23:14:00	сельский п 23 ищу девушку для отношений
997	2012-10-15 23:37:00	Д+Д ДЛЯ серьезных отношений. Мой номер 093-156...
998	2012-06-21 23:34:00	7 ДНЕПР М.34 ПОЗ.С Д/Ж ДЛЯ ВСТРЕЧ.Т.098 809 15 14
999	2014-06-19 23:25:00	Парень поласкает девушке... т.0662035584

1000 rows × 2 columns

messages_dataset.csv :
date -> denotes date and time.
msg -> messages of chat.

In [24]:
```

```

####Let's work with Dates :

In [25]:
```
Order.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype 
---  ------      --------------  ----- 
 0   date        1000 non-null   object
 1   product_id  1000 non-null   int64 
 2   city_id     1000 non-null   int64 
 3   orders      1000 non-null   int64 
dtypes: int64(3), object(1)
memory usage: 31.4+ KB
```

Here we can see that the column **date** from Order is having an datatype as object.

In [26]:
```
Order['date'] = pd.to_datetime(Order['date'])
```

Updated values of column **date** with proper dates.

In [27]:
```
Order.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 4 columns):
 #   Column      Non-Null Count  Dtype         
---  ------      --------------  -----         
 0   date        1000 non-null   datetime64[ns]
 1   product_id  1000 non-null   int64         
 2   city_id     1000 non-null   int64         
 3   orders      1000 non-null   int64         
dtypes: datetime64[ns](1), int64(3)
memory usage: 31.4 KB
```

pd.to_datetime function helps us to fetch datetime.
Order.info() -> The datatype for column **date** we can see here is updated as datetime.

How can we extract year from date?

In [28]:
```
Order['date_year'] = Order['date'].dt.year
```

In [29]:
```
Order
```

Out[29]:

	date	product_id	city_id	orders	date_year
0	2019-12-10	5628	25	3	2019
1	2018-08-15	3646	14	157	2018
2	2018-10-23	1859	25	1	2018
3	2019-08-17	7292	25	1	2019
4	2019-01-06	4344	25	3	2019
...	...	...	...	...	...
995	2018-10-08	255	13	1	2018
996	2018-12-06	5521	7	1	2018
997	2019-05-07	487	26	14	2019
998	2019-03-03	1503	21	2	2019
999	2019-10-15	6371	7	22	2019

1000 rows × 5 columns

Created variable **'date_year'** where I stored year of each particular date.
Here, the **.dt.year** gives us year from particular date

How can we extract month from date?

In [30]:
```
Order['date_month'] = Order['date'].dt.month
```

In [31]:
```
Order
```

Out[31]:

	date	product_id	city_id	orders	date_year	date_month
0	2019-12-10	5628	25	3	2019	12
1	2018-08-15	3646	14	157	2018	8
2	2018-10-23	1859	25	1	2018	10
3	2019-08-17	7292	25	1	2019	8
4	2019-01-06	4344	25	3	2019	1
...	...	...	...	...	...	...
995	2018-10-08	255	13	1	2018	10
996	2018-12-06	5521	7	1	2018	12
997	2019-05-07	487	26	14	2019	5
998	2019-03-03	1503	21	2	2019	3
999	2019-10-15	6371	7	22	2019	10

1000 rows × 6 columns

Created variable **'date_month'** where I stored month of each particular date.
Here, the **.dt.month** gives us month from particular date.

For month names we can do as,

In [32]:
```
Order['date_month_name'] = Order['date'].dt.month_name()
```

In [33]:
```
Order
```

Out[33]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name
0	2019-12-10	5628	25	3	2019	12	December
1	2018-08-15	3646	14	157	2018	8	August
2	2018-10-23	1859	25	1	2018	10	October
3	2019-08-17	7292	25	1	2019	8	August
4	2019-01-06	4344	25	3	2019	1	January
...	...	...	...	...	...	...	...
995	2018-10-08	255	13	1	2018	10	October
996	2018-12-06	5521	7	1	2018	12	December
997	2019-05-07	487	26	14	2019	5	May
998	2019-03-03	1503	21	2	2019	3	March
999	2019-10-15	6371	7	22	2019	10	October

1000 rows × 7 columns

Here we can see month names we extracted from date.

How can we extract day from date?

In [34]:
```
Order['date_day'] = Order['date'].dt.day
```

In [35]:
```
Order
```

Out[35]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name	date_day
0	2019-12-10	5628	25	3	2019	12	December	10
1	2018-08-15	3646	14	157	2018	8	August	15
2	2018-10-23	1859	25	1	2018	10	October	23
3	2019-08-17	7292	25	1	2019	8	August	17
4	2019-01-06	4344	25	3	2019	1	January	6
...	...	...	...	...	...	...	...	...
995	2018-10-08	255	13	1	2018	10	October	8
996	2018-12-06	5521	7	1	2018	12	December	6
997	2019-05-07	487	26	14	2019	5	May	7
998	2019-03-03	1503	21	2	2019	3	March	3
999	2019-10-15	6371	7	22	2019	10	October	15

1000 rows × 8 columns

Created variable **'date_day'** where I stored day of each particular date.
Here, the **.dt.day** gives us day from particular date.

How can we extract day of the week from date?

In [36]:
```
Order['date_day_of_week'] = Order['date'].dt.dayofweek
```

In [37]:
```
Order
```

Out[37]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name	date_day	date_day_of_week
0	2019-12-10	5628	25	3	2019	12	December	10	1
1	2018-08-15	3646	14	157	2018	8	August	15	2
2	2018-10-23	1859	25	1	2018	10	October	23	1
3	2019-08-17	7292	25	1	2019	8	August	17	5
4	2019-01-06	4344	25	3	2019	1	January	6	6
...	...	...	...	...	...	...	...	...	...
995	2018-10-08	255	13	1	2018	10	October	8	0
996	2018-12-06	5521	7	1	2018	12	December	6	3
997	2019-05-07	487	26	14	2019	5	May	7	1
998	2019-03-03	1503	21	2	2019	3	March	3	6
999	2019-10-15	6371	7	22	2019	10	October	15	1

1000 rows × 9 columns

How can we extract day name of week from date?

In [38]:
```
Order['date_day_name_of_week'] = Order['date'].dt.day_name()
```

In [39]:
```
Order
```

Out[39]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name	date_day	date_day_of_week	date_day_name_of_week
0	2019-12-10	5628	25	3	2019	12	December	10	1	Tuesday
1	2018-08-15	3646	14	157	2018	8	August	15	2	Wednesday
2	2018-10-23	1859	25	1	2018	10	October	23	1	Tuesday
3	2019-08-17	7292	25	1	2019	8	August	17	5	Saturday
4	2019-01-06	4344	25	3	2019	1	January	6	6	Sunday
...	...	...	...	...	...	...	...	...	...	...
995	2018-10-08	255	13	1	2018	10	October	8	0	Monday
996	2018-12-06	5521	7	1	2018	12	December	6	3	Thursday
997	2019-05-07	487	26	14	2019	5	May	7	1	Tuesday
998	2019-03-03	1503	21	2	2019	3	March	3	6	Sunday
999	2019-10-15	6371	7	22	2019	10	October	15	1	Tuesday

1000 rows × 10 columns

0 - Monday
1 - Tuesday
2 - Wednesday
3 - Thursday
4 - Friday
5 - Saturday
6 - Sunday

Created variable **'date_day_name_of_week'** where I stored day name in week of each particular date.
Here, the **.dt.dayofweek** gives us day of week from particular date

How to check whether the day from date is weekend or not?

In [41]:
```
Order['date_is_weekend'] = np.where(Order['date_day_of_week'].isin(['Sunday', 'Saturday']), 1,0)
```

In [42]:
```
Order
```

Out[42]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name	date_day	date_day_of_week	date_day_name_of_week	date_is_
0	2019-12-10	5628	25	3	2019	12	December	10	1	Tuesday	
1	2018-08-15	3646	14	157	2018	8	August	15	2	Wednesday	
2	2018-10-23	1859	25	1	2018	10	October	23	1	Tuesday	
3	2019-08-17	7292	25	1	2019	8	August	17	5	Saturday	
4	2019-01-06	4344	25	3	2019	1	January	6	6	Sunday	
...	...	...	...	...	...	...	...	...	...	...	
995	2018-10-08	255	13	1	2018	10	October	8	0	Monday	
996	2018-12-06	5521	7	1	2018	12	December	6	3	Thursday	
997	2019-05-07	487	26	14	2019	5	May	7	1	Tuesday	
998	2019-03-03	1503	21	2	2019	3	March	3	6	Sunday	
999	2019-10-15	6371	7	22	2019	10	October	15	1	Tuesday	

1000 rows × 11 columns

Here, I applied logic as if the **date_day_name_of_week** from dataset is **Saturday** or **Sunday** then value for **date_is_weekend** will be **1** that is True else it will be **False**.

How can we extract week of the year?

In [43]:
```
Order['week_of_year'] = Order['date'].dt.week
```

```
<ipython-input-43-bf2d5d71f23a>:1: FutureWarning: Series.dt.weekofyear and Series.dt.week have been deprecated. Pleas
e use Series.dt.isocalendar().week instead.
  Order['week_of_year'] = Order['date'].dt.week
```

In [44]:
```
Order
```

Out[44]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name	date_day	date_day_of_week	date_day_name_of_week	date_is_
0	2019-12-10	5628	25	3	2019	12	December	10	1	Tuesday	
1	2018-08-15	3646	14	157	2018	8	August	15	2	Wednesday	
2	2018-10-23	1859	25	1	2018	10	October	23	1	Tuesday	
3	2019-08-17	7292	25	1	2019	8	August	17	5	Saturday	
4	2019-01-06	4344	25	3	2019	1	January	6	6	Sunday	
...	...	...	...	...	...	...	...	...	...	...	
995	2018-10-08	255	13	1	2018	10	October	8	0	Monday	
996	2018-12-06	5521	7	1	2018	12	December	6	3	Thursday	
997	2019-05-07	487	26	14	2019	5	May	7	1	Tuesday	
998	2019-03-03	1503	21	2	2019	3	March	3	6	Sunday	
999	2019-10-15	6371	7	22	2019	10	October	15	1	Tuesday	

1000 rows × 12 columns

Created variable 'week_of_year' where I stored week in year of each particular date.
Here, the **.dt.week** gives us week of the year from particular date.

How can we extract Quarter?

In [45]:
```
Order['Quarter'] = Order['date'].dt.quarter
```

In [46]:
```
Order
```

Out[46]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name	date_day	date_day_of_week	date_day_name_of_week	date_is_
0	2019-12-10	5628	25	3	2019	12	December	10	1	Tuesday	
1	2018-08-15	3646	14	157	2018	8	August	15	2	Wednesday	
2	2018-10-23	1859	25	1	2018	10	October	23	1	Tuesday	
3	2019-08-17	7292	25	1	2019	8	August	17	5	Saturday	
4	2019-01-06	4344	25	3	2019	1	January	6	6	Sunday	
...	...	...	...	...	...	...	...	...	...	...	
995	2018-10-08	255	13	1	2018	10	October	8	0	Monday	
996	2018-12-06	5521	7	1	2018	12	December	6	3	Thursday	
997	2019-05-07	487	26	14	2019	5	May	7	1	Tuesday	
998	2019-03-03	1503	21	2	2019	3	March	3	6	Sunday	
999	2019-10-15	6371	7	22	2019	10	October	15	1	Tuesday	

1000 rows × 13 columns

January, February, March - Quarter 1.
April, May, June - Quarter 2.
July, August, September - Quarter 3.
October, November, December - Quarter - 4.

How can we extract Semester?

In [47]:
```
Order['Semester'] = np.where(Order['Quarter'].isin([1,2]), 1, 2)
```

In [48]:
```
Order
```

Out[48]:

	date	product_id	city_id	orders	date_year	date_month	date_month_name	date_day	date_day_of_week	date_day_name_of_week	date_is_
0	2019-12-10	5628	25	3	2019	12	December	10	1	Tuesday	
1	2018-08-15	3646	14	157	2018	8	August	15	2	Wednesday	
2	2018-10-23	1859	25	1	2018	10	October	23	1	Tuesday	
3	2019-08-17	7292	25	1	2019	8	August	17	5	Saturday	
4	2019-01-06	4344	25	3	2019	1	January	6	6	Sunday	
...	...	...	...	...	...	...	...	...	...	...	
995	2018-10-08	255	13	1	2018	10	October	8	0	Monday	
996	2018-12-06	5521	7	1	2018	12	December	6	3	Thursday	
997	2019-05-07	487	26	14	2019	5	May	7	1	Tuesday	
998	2019-03-03	1503	21	2	2019	3	March	3	6	Sunday	
999	2019-10-15	6371	7	22	2019	10	October	15	1	Tuesday	

1000 rows × 14 columns

Here I applied logic that if Quarter from our dataset is 1 or 2 i.e, January, February or March, April then the Semester is 1 else Semester is 2

How to find gap between two dates (Extract time elapsed between dates)

I already imported necessary library 'datetime'

In [49]:
```
today = datetime.datetime.today()
```

In [50]:
```
today
```

Out[50]:
```
datetime.datetime(2023, 11, 21, 7, 12, 19, 796919)
```

Basically this is current date which we stored in variable today

In [52]:
```
today - Order['date']
```

Out[52]:
```
0     1442 days 07:12:19.796919
1     1924 days 07:12:19.796919
2     1855 days 07:12:19.796919
3     1557 days 07:12:19.796919
4     1780 days 07:12:19.796919
                 ...
995   1870 days 07:12:19.796919
996   1811 days 07:12:19.796919
997   1659 days 07:12:19.796919
998   1724 days 07:12:19.796919
999   1498 days 07:12:19.796919
Name: date, Length: 1000, dtype: timedelta64[ns]
```

Subtracted column date from today which will give the total days gap between today and the date in dataset.

if only want to see days gap

In [55]:
```
(today - Order['date']).dt.days
```

Out[55]:
```
0      1442
1      1924
2      1855
3      1557
4      1780
       ...
995    1870
996    1811
997    1659
998    1724
999    1498
Name: date, Length: 1000, dtype: int64
```

if we want months passed I will use as: ((today - Order['date']) / np.timedelta64(1, 'M'), 0)

In [56]:
```
((today - Order['date']) / np.timedelta64(1, 'M'), 0)
```

Out[56]:
```
(0      47.386607
 1      63.222661
 2      60.955674
 3      51.164919
 4      58.491558
          ...
 995    61.448497
 996    59.510059
 997    54.516117
 998    56.651684
 999    49.226480
 Name: date, Length: 1000, dtype: float64,
 0)
```

In [57]:
```
np.round((today - Order['date']) / np.timedelta64(1, 'M'), 0)
```

Out[57]:
```
0      47.0
1      63.0
2      61.0
3      51.0
4      58.0
       ...
995    61.0
996    60.0
997    55.0
998    57.0
999    49.0
Name: date, Length: 1000, dtype: float64
```

So using the **np.timedelta64(1, 'M')** we get the gap of total number of months passed from the date in dataset to today's date.

In [ ]:

#**Let's work with Time**

In [58]:
```
Chat.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   date    1000 non-null   object
 1   msg     1000 non-null   object
dtypes: object(2)
memory usage: 15.8+ KB
```

Here we can see the datatype of date is object, So firstly I will be converting it in datatype datatime.

In [59]:
```
Chat
```

Out[59]:

	date	msg
0	2013-12-15 00:50:00	ищу на сегодня мужика 37
1	2014-04-29 23:40:00	ПАРЕНЬ БИ ИЩЕТ ДРУГА СЕЙЧАС!! СМС ММС 0955532826
2	2012-12-30 00:21:00	Днепр.м 43 позн.с д/ж *.о 067.16.34.576
3	2014-11-28 00:31:00	КИЕВ ИЩУ Д/Ж ДО 45 МНЕ СЕЙЧАС СКУЧНО 093 629 9...
4	2013-10-26 23:11:00	Зая я тебя никогда не обижу люблю тебя!) Даше
...	...	...
995	2012-03-16 00:50:00	ПАРЕНЬ СДЕЛАЕТ МАССАЖ ЖЕНЩИНАМ -066-877-32-44
996	2014-01-23 23:14:00	сельский п 23 ищу девушку для отношений
997	2012-10-15 23:37:00	Д+Д ДЛЯ серьезных отношений. Мой номер 093-156...
998	2012-06-21 23:34:00	7 ДНЕПР М.34 ПОЗ.С Д/Ж ДЛЯ ВСТРЕЧ.Т.098 809 15 14
999	2014-06-19 23:25:00	Парень поласкает девушке... т.0662035584

1000 rows × 2 columns

In [60]:
```
Chat['date'] = pd.to_datetime(Chat['date'])
```

In [61]:
```
Chat
```

Out[61]:

	date	msg
0	2013-12-15 00:50:00	ищу на сегодня мужика 37
1	2014-04-29 23:40:00	ПАРЕНЬ БИ ИЩЕТ ДРУГА СЕЙЧАС!! СМС ММС 0955532826
2	2012-12-30 00:21:00	Днепр.м 43 позн.с д/ж *.о 067.16.34.576
3	2014-11-28 00:31:00	КИЕВ ИЩУ Д/Ж ДО 45 МНЕ СЕЙЧАС СКУЧНО 093 629 9...
4	2013-10-26 23:11:00	Зая я тебя никогда не обижу люблю тебя!) Даше
...	...	...
995	2012-03-16 00:50:00	ПАРЕНЬ СДЕЛАЕТ МАССАЖ ЖЕНЩИНАМ -066-877-32-44
996	2014-01-23 23:14:00	сельский п 23 ищу девушку для отношений
997	2012-10-15 23:37:00	Д+Д ДЛЯ серьезных отношений. Мой номер 093-156...
998	2012-06-21 23:34:00	7 ДНЕПР М.34 ПОЗ.С Д/Ж ДЛЯ ВСТРЕЧ.Т.098 809 15 14
999	2014-06-19 23:25:00	Парень поласкает девушке... т.0662035584

1000 rows × 2 columns

In time there are mainly 3 parameters hour, minute, second.
So let's just extract those...

For extracting Hour from datetime

In [62]:
```
Chat['Hour'] = Chat['date'].dt.hour
```

In [63]:
```
Chat
```

Out[63]:

	date	msg	Hour
0	2013-12-15 00:50:00	ищу на сегодня мужика 37	0
1	2014-04-29 23:40:00	ПАРЕНЬ БИ ИЩЕТ ДРУГА СЕЙЧАС!! СМС ММС 0955532826	23
2	2012-12-30 00:21:00	Днепр.м 43 позн.с д/ж *.о 067.16.34.576	0
3	2014-11-28 00:31:00	КИЕВ ИЩУ Д/Ж ДО 45 МНЕ СЕЙЧАС СКУЧНО 093 629 9...	0
4	2013-10-26 23:11:00	Зая я тебя никогда не обижу люблю тебя!) Даше	23
...	...	...	...
995	2012-03-16 00:50:00	ПАРЕНЬ СДЕЛАЕТ МАССАЖ ЖЕНЩИНАМ -066-877-32-44	0
996	2014-01-23 23:14:00	сельский п 23 ищу девушку для отношений	23
997	2012-10-15 23:37:00	Д+Д ДЛЯ серьезных отношений. Мой номер 093-156...	23
998	2012-06-21 23:34:00	7 ДНЕПР М.34 ПОЗ.С Д/Ж ДЛЯ ВСТРЕЧ.Т.098 809 15 14	23
999	2014-06-19 23:25:00	Парень поласкает девушке... т.0662035584	23

1000 rows × 3 columns

So, **.dt.hour** helps us to extract hour from the time.

For extracting minutes from date

In [64]:
```
Chat['Minute'] = Chat['date'].dt.minute
```

In [65]:
```
Chat
```

Out[65]:

	date	msg	Hour	Minute
0	2013-12-15 00:50:00	ищу на сегодня мужика 37	0	50
1	2014-04-29 23:40:00	ПАРЕНЬ БИ ИЩЕТ ДРУГА СЕЙЧАС!! СМС ММС 0955532826	23	40
2	2012-12-30 00:21:00	Днепр.м 43 позн.с д/ж *.о 067.16.34.576	0	21
3	2014-11-28 00:31:00	КИЕВ ИЩУ Д/Ж ДО 45 МНЕ СЕЙЧАС СКУЧНО 093 629 9...	0	31
4	2013-10-26 23:11:00	Зая я тебя никогда не обижу люблю тебя!) Даше	23	11
...	...	...	...	...
995	2012-03-16 00:50:00	ПАРЕНЬ СДЕЛАЕТ МАССАЖ ЖЕНЩИНАМ -066-877-32-44	0	50
996	2014-01-23 23:14:00	сельский п 23 ищу девушку для отношений	23	14
997	2012-10-15 23:37:00	Д+Д ДЛЯ серьезных отношений. Мой номер 093-156...	23	37
998	2012-06-21 23:34:00	7 ДНЕПР М.34 ПОЗ.С Д/Ж ДЛЯ ВСТРЕЧ.Т.098 809 15 14	23	34
999	2014-06-19 23:25:00	Парень поласкает девушке... т.0662035584	23	25

1000 rows × 4 columns

So, **.dt.minute** helps us to extract minute from the time.

For extracting second from date

In [66]:
```
Chat['second'] = Chat['date'].dt.second
```

In [67]:
```
Chat
```

Out[67]:

	date	msg	Hour	Minute	second
0	2013-12-15 00:50:00	ищу на сегодня мужика 37	0	50	0
1	2014-04-29 23:40:00	ПАРЕНЬ БИ ИЩЕТ ДРУГА СЕЙЧАС!! СМС ММС 0955532826	23	40	0
2	2012-12-30 00:21:00	Днепр.м 43 позн.с д/ж *.о 067.16.34.576	0	21	0
3	2014-11-28 00:31:00	КИЕВ ИЩУ Д/Ж ДО 45 МНЕ СЕЙЧАС СКУЧНО 093 629 9...	0	31	0
4	2013-10-26 23:11:00	Зая я тебя никогда не обижу люблю тебя!) Даше	23	11	0
...	...	...	...	...	...
995	2012-03-16 00:50:00	ПАРЕНЬ СДЕЛАЕТ МАССАЖ ЖЕНЩИНАМ -066-877-32-44	0	50	0
996	2014-01-23 23:14:00	сельский п 23 ищу девушку для отношений	23	14	0
997	2012-10-15 23:37:00	Д+Д ДЛЯ серьезных отношений. Мой номер 093-156...	23	37	0
998	2012-06-21 23:34:00	7 ДНЕПР М.34 ПОЗ.С Д/Ж ДЛЯ ВСТРЕЧ.Т.098 809 15 14	23	34	0
999	2014-06-19 23:25:00	Парень поласкает девушке... т.0662035584	23	25	0

1000 rows × 5 columns

So, **.dt.second** helps us to extract second from the time.

If only wants to extract time

In [68]:
```
Chat['Time'] = Chat['date'].dt.time
```

In [69]:
```
Chat
```

Out[69]:

	date	msg	Hour	Minute	second	Time
0	2013-12-15 00:50:00	ищу на сегодня мужика 37	0	50	0	00:50:00
1	2014-04-29 23:40:00	ПАРЕНЬ БИ ИЩЕТ ДРУГА СЕЙЧАС!! СМС ММС 0955532826	23	40	0	23:40:00
2	2012-12-30 00:21:00	Днепр.м 43 позн.с д/ж *.о 067.16.34.576	0	21	0	00:21:00
3	2014-11-28 00:31:00	КИЕВ ИЩУ Д/Ж ДО 45 МНЕ СЕЙЧАС СКУЧНО 093 629 9...	0	31	0	00:31:00
4	2013-10-26 23:11:00	Зая я тебя никогда не обижу люблю тебя!) Даше	23	11	0	23:11:00
...	...	...	...	...	...	...
995	2012-03-16 00:50:00	ПАРЕНЬ СДЕЛАЕТ МАССАЖ ЖЕНЩИНАМ -066-877-32-44	0	50	0	00:50:00
996	2014-01-23 23:14:00	сельский п 23 ищу девушку для отношений	23	14	0	23:14:00
997	2012-10-15 23:37:00	Д+Д ДЛЯ серьезных отношений. Мой номер 093-156...	23	37	0	23:37:00
998	2012-06-21 23:34:00	7 ДНЕПР М.34 ПОЗ.С Д/Ж ДЛЯ ВСТРЕЧ.Т.098 809 15 14	23	34	0	23:34:00
999	2014-06-19 23:25:00	Парень поласкает девушке... т.0662035584	23	25	0	23:25:00

1000 rows × 6 columns

Using **.dt.time** we can extract time from the given datetime.

In [ ]:
```

```

If we want to obtain time difference/gap.

In [70]:
```
today - Chat['date']
```

Out[70]:
```
0      3628 days 06:22:19.796919
1      3492 days 07:32:19.796919
2      3978 days 06:51:19.796919
3      3280 days 06:41:19.796919
4      3677 days 08:01:19.796919
                  ...
995    4267 days 06:22:19.796919
996    3588 days 07:58:19.796919
997    4053 days 07:35:19.796919
998    4169 days 07:38:19.796919
999    3441 days 07:47:19.796919
Name: date, Length: 1000, dtype: timedelta64[ns]
```

here, the today parameter is what we used earlier for current datetime.
So from today to the datetime in dataset the gap is shown above.

gap in seconds

In [71]:
```
(today - Chat['date'])/np.timedelta64(1, 's')
```

Out[71]:
```
0       3.134821e+08
1       3.017359e+08
2       3.437239e+08
3       2.834161e+08
4       3.177217e+08
            ...
995     3.686917e+08
996     3.100319e+08
997     3.502065e+08
998     3.602291e+08
999     2.973304e+08
Name: date, Length: 1000, dtype: float64
```

In [73]:
```
np.round((today - Chat['date'])/np.timedelta64(1, 's'))
```

Out[73]:
```
0      313482140.0
1      301735940.0
2      343723880.0
3      283416080.0
4      317721680.0
          ...
995    368691740.0
996    310031900.0
997    350206520.0
998    360229100.0
999    297330440.0
Name: date, Length: 1000, dtype: float64
```

gap in minutes

In [74]:
```
(today - Chat['date'])/np.timedelta64(1, 'm')
```

Out[74]:
```
0      5.224702e+06
1      5.028932e+06
2      5.728731e+06
3      4.723601e+06
4      5.295361e+06
           ...
995    6.144862e+06
996    5.167198e+06
997    5.836775e+06
998    6.003818e+06
999    4.955507e+06
Name: date, Length: 1000, dtype: float64
```

In [75]:
```
np.round((today - Chat['date'])/np.timedelta64(1, 'm'))
```

Out[75]:
```
0      5224702.0
1      5028932.0
2      5728731.0
3      4723601.0
4      5295361.0
         ...
995    6144862.0
996    5167198.0
997    5836775.0
998    6003818.0
999    4955507.0
Name: date, Length: 1000, dtype: float64
```

gap in hours

In [76]:
```
(today - Chat['date'])/np.timedelta64(1, 'h')
```

Out[76]:
```
0       87078.372166
1       83815.538832
2       95478.855499
3       78726.688832
4       88256.022166
           ...
995    102414.372166
996     86119.972166
997     97279.588832
998    100063.638832
999     82591.788832
Name: date, Length: 1000, dtype: float64
```

In [77]:
```
np.round((today - Chat['date'])/np.timedelta64(1, 'h'))
```

Out[77]:
```
0       87078.0
1       83816.0
2       95479.0
3       78727.0
4       88256.0
          ...
995    102414.0
996     86120.0
997     97280.0
998    100064.0
999     82592.0
Name: date, Length: 1000, dtype: float64
```

In [ ]:
```

```

#Timezone data

In [116]:
```
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import pytz
```

We use pytz library to handle time zones.

In [117]:
```
np.random.seed(42)

n_events = 100
start_date = datetime(2023, 1, 1, tzinfo=pytz.UTC)
end_date = datetime(2023, 12, 31, tzinfo=pytz.UTC)

event_dates = [start_date + timedelta(days=np.random.randint(365)) for _ in range(n_events)]

time_zones = ['US/Eastern', 'Europe/London', 'Asia/Tokyo', 'Australia/Sydney']
event_time_zones = np.random.choice(time_zones, n_events)

events_df = pd.DataFrame({
    'EventDate': event_dates,
    'TimeZone': event_time_zones
})
```

Just created a timezone dataset where I generate random dates and times for events.
Then assigned time zones to events.

In [118]:
```
events_df
```

Out[118]:

	EventDate	TimeZone
0	2023-04-13 00:00:00+00:00	Australia/Sydney
1	2023-12-15 00:00:00+00:00	Australia/Sydney
2	2023-09-28 00:00:00+00:00	Australia/Sydney
3	2023-04-17 00:00:00+00:00	Asia/Tokyo
4	2023-03-13 00:00:00+00:00	Asia/Tokyo
...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London
96	2023-11-23 00:00:00+00:00	Asia/Tokyo
97	2023-01-09 00:00:00+00:00	Australia/Sydney
98	2023-12-10 00:00:00+00:00	Europe/London
99	2023-05-09 00:00:00+00:00	Asia/Tokyo

100 rows × 2 columns

Local Time: Let's convert the event times to local time for better interpretation.

This is the main task.

In [119]:
```
def convert_to_local_time(row):
    return row['EventDate'].astimezone(pytz.timezone(row['TimeZone']))
```

In [120]:
```
events_df['LocalTime'] = events_df.apply(convert_to_local_time, axis=1)
```

In [121]:
```
events_df
```

Out[121]:

	EventDate	TimeZone	LocalTime
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 10:00:00+10:00
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 11:00:00+11:00
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 10:00:00+10:00
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 09:00:00+09:00
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 09:00:00+09:00
...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 09:00:00+09:00
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 11:00:00+11:00
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 09:00:00+09:00

100 rows × 3 columns

Created a function to convert UTC time to local time using pytz.
Then I applied function to create new feature 'LocalTime'.

Hour of Day: Now let's extract the hour of the day when each event occurred.

In [122]:
```
events_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   EventDate  100 non-null    datetime64[ns, UTC]
 1   TimeZone   100 non-null    object
 2   LocalTime  100 non-null    object
dtypes: datetime64[ns, UTC](1), object(2)
memory usage: 2.5+ KB
```

As LocalTime here is an object let's convert it into datatype as datetime.

In [123]:
```
events_df
```

Out[123]:

	EventDate	TimeZone	LocalTime
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 10:00:00+10:00
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 11:00:00+11:00
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 10:00:00+10:00
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 09:00:00+09:00
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 09:00:00+09:00
...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 09:00:00+09:00
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 11:00:00+11:00
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 09:00:00+09:00

100 rows × 3 columns

In [124]:
```
events_df['LocalTime'] = pd.to_datetime(events_df['LocalTime'], utc=True)
```

Here I converted datatype of date from object to datetime.
We use utc=True because the timing is in format of UTC.
NOTE: If working with timezone we must apply utc=True

In [125]:
```
events_df
```

Out[125]:

	EventDate	TimeZone	LocalTime
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00
...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00

100 rows × 3 columns

In [126]:
```
events_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 3 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   EventDate  100 non-null    datetime64[ns, UTC]
 1   TimeZone   100 non-null    object
 2   LocalTime  100 non-null    datetime64[ns, UTC]
dtypes: datetime64[ns, UTC](2), object(1)
memory usage: 2.5+ KB
```

In [127]:
```
events_df['HourOfDay'] = events_df['LocalTime'].dt.hour
```

In [128]:
```
events_df
```

Out[128]:

	EventDate	TimeZone	LocalTime	HourOfDay
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0
...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0

100 rows × 4 columns

In [129]:
```
events_df['Minutes'] = events_df['LocalTime'].dt.minute
```

In [130]:
```
events_df
```

Out[130]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0
...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0

100 rows × 5 columns

In [131]:
```
events_df['Second'] = events_df['LocalTime'].dt.second
```

In [132]:
```
events_df
```

Out[132]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes	Second
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0	0
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0	0
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0	0
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0	0
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0	0
...	...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0	0
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0	0
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0	0
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0	0
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0	0

100 rows × 6 columns

In [133]:
```
events_df['Day'] = events_df['LocalTime'].dt.day
```

In [134]:
```
events_df
```

Out[134]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes	Second	Day
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0	0	13
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0	0	15
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0	0	28
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0	0	17
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0	0	13
...	...	...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0	0	11
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0	0	23
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0	0	9
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0	0	10
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0	0	9

100 rows × 7 columns

In [135]:
```
events_df['Month'] = events_df['LocalTime'].dt.month
```

In [136]:
```
events_df
```

Out[136]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes	Second	Day	Month
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0	0	13	4
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0	0	15	12
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0	0	28	9
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0	0	17	4
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0	0	13	3
...	...	...	...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0	0	11	12
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0	0	23	11
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0	0	9	1
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0	0	10	12
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0	0	9	5

100 rows × 8 columns

In [137]:
```
events_df['Month Name'] = events_df['LocalTime'].dt.month_name()
```

In [138]:
```
events_df
```

Out[138]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes	Second	Day	Month	Month Name
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0	0	13	4	April
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0	0	15	12	December
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0	0	28	9	September
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0	0	17	4	April
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0	0	13	3	March
...	...	...	...	...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0	0	11	12	December
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0	0	23	11	November
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0	0	9	1	January
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0	0	10	12	December
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0	0	9	5	May

100 rows × 9 columns

In [139]:
```
events_df['Year'] = events_df['LocalTime'].dt.year
```

In [140]:
```
events_df
```

Out[140]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes	Second	Day	Month	Month Name	Year
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0	0	13	4	April	2023
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0	0	15	12	December	2023
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0	0	28	9	September	2023
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0	0	17	4	April	2023
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0	0	13	3	March	2023
...	...	...	...	...	...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0	0	11	12	December	2023
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0	0	23	11	November	2023
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0	0	9	1	January	2023
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0	0	10	12	December	2023
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0	0	9	5	May	2023

100 rows × 10 columns

In [141]:
```
events_df['Week'] = events_df['LocalTime'].dt.dayofweek
```

In [142]:
```
events_df
```

Out[142]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes	Second	Day	Month	Month Name	Year	Week
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0	0	13	4	April	2023	3
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0	0	15	12	December	2023	4
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0	0	28	9	September	2023	3
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0	0	17	4	April	2023	0
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0	0	13	3	March	2023	0
...	...	...	...	...	...	...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0	0	11	12	December	2023	0
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0	0	23	11	November	2023	3
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0	0	9	1	January	2023	0
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0	0	10	12	December	2023	6
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0	0	9	5	May	2023	1

100 rows × 11 columns

In [143]:
```
events_df['Weekday'] = events_df['LocalTime'].dt.day_name()
```

In [144]:
```
events_df
```

Out[144]:

	EventDate	TimeZone	LocalTime	HourOfDay	Minutes	Second	Day	Month	Month Name	Year	Week	Weekday
0	2023-04-13 00:00:00+00:00	Australia/Sydney	2023-04-13 00:00:00+00:00	0	0	0	13	4	April	2023	3	Thursday
1	2023-12-15 00:00:00+00:00	Australia/Sydney	2023-12-15 00:00:00+00:00	0	0	0	15	12	December	2023	4	Friday
2	2023-09-28 00:00:00+00:00	Australia/Sydney	2023-09-28 00:00:00+00:00	0	0	0	28	9	September	2023	3	Thursday
3	2023-04-17 00:00:00+00:00	Asia/Tokyo	2023-04-17 00:00:00+00:00	0	0	0	17	4	April	2023	0	Monday
4	2023-03-13 00:00:00+00:00	Asia/Tokyo	2023-03-13 00:00:00+00:00	0	0	0	13	3	March	2023	0	Monday
...	...	...	...	...	...	...	...	...	...	...	...	...
95	2023-12-11 00:00:00+00:00	Europe/London	2023-12-11 00:00:00+00:00	0	0	0	11	12	December	2023	0	Monday
96	2023-11-23 00:00:00+00:00	Asia/Tokyo	2023-11-23 00:00:00+00:00	0	0	0	23	11	November	2023	3	Thursday
97	2023-01-09 00:00:00+00:00	Australia/Sydney	2023-01-09 00:00:00+00:00	0	0	0	9	1	January	2023	0	Monday
98	2023-12-10 00:00:00+00:00	Europe/London	2023-12-10 00:00:00+00:00	0	0	0	10	12	December	2023	6	Sunday
99	2023-05-09 00:00:00+00:00	Asia/Tokyo	2023-05-09 00:00:00+00:00	0	0	0	9	5	May	2023	1	Tuesday

100 rows × 12 columns

These day, month, year, month name, week, weekday, hour, minute, second...
These all I did same as I did earlier.

In [ ]: