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

Table of Contents

    * Setting Index

3. Data Cleaning
   3.1 Handling Missing Data

    * Check Missing Data in a Column
    * To Find the count of missing values in each column
    * Filling the missing Data using fillna
    * Drop Rows or columns with missing data using dropna

   3.2 Handing Duplicates

    * Finding Duplicates using .duplicated
    * Drop the duplicated rows using .drop_duplicates

   3.3 String Operations
   3.4 Data Type Conversion

4. Data Manipulation
   4.1 Applying Functions to DataFrames

    * .apply for series and DataFrames
    * .map for series
    * .applymap for entire dataframe

   4.2 Adding and Removing Columns
   4.3 Combining DataFrames

    * Concatenation of DataFrames
    * Merging using inner join
    * Merging using left join
    * Merging one df column with other df index

5. Data Aggregation
   5.1 Grouping Data

    * Group with a single column using groupby method.
    * Group with single column and apply to entire Dataframe.
    * Group with multiple columns

   5.2 Aggregate Functions

    * Apply multiple aggregate functions to the grouped data.
    * Apply multiple aggregate functions for selected columns.

   5.3 Pivot Tables and Cross-Tabulations

6. Data Visualizations
   6.1 Find the Correlations
   6.2 Sorting Data and Creating Plots

1 - Data Loading and Inspection

In [2]:
```
# Loading Data from a csv file
ufo_data = pd.read_csv('http://bit.ly/uforeports')
```

In [3]:
```
# reading tsv files
chips_data = pd.read_table('http://bit.ly/chiporders')
```

In [4]:
```
# reading tsv files with read_csv, but by using the sep parameter
chips_data = pd.read_csv('http://bit.ly/chiporders',sep='\t')
```

1.1 Displaying Data Frames

In [5]:
```
# # This is to observe the first n rows of the data
chips_data.head(5)
```

Out[5]:

	order_id	quantity	item_name	choice_description	item_price
0	1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	1	Izze	[Clementine]	$3.39
2	1	1	Nantucket Nectar	[Apple]	$3.39
3	1	1	Chips and Tomatillo-Green Chili Salsa	NaN	$2.39
4	2	2	Chicken Bowl	[Tomatillo-Red Chili Salsa (Hot), [Black Beans...	$16.98

In [6]:
```
# This is to observe the last n rows of the data
chips_data.tail(5)
```

Out[6]:

	order_id	quantity	item_name	choice_description	item_price
4617	1833	1	Steak Burrito	[Fresh Tomato Salsa, [Rice, Black Beans, Sour ...	$11.75
4618	1833	1	Steak Burrito	[Fresh Tomato Salsa, [Rice, Sour Cream, Cheese...	$11.75
4619	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Pinto...	$11.25
4620	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Lettu...	$8.75
4621	1834	1	Chicken Salad Bowl	[Fresh Tomato Salsa, [Fajita Vegetables, Pinto...	$8.75

In [7]:
```
# This is useful for exploring diverse parts of the dataset.
chips_data.sample(5)
```

Out[7]:

	order_id	quantity	item_name	choice_description	item_price
60	28	1	Chips and Guacamole	NaN	$4.45
1560	634	1	Chicken Soft Tacos	[Fresh Tomato Salsa]	$8.75
2578	1021	1	Bottled Water	NaN	$1.50
4301	1715	1	Canned Soft Drink	[Coke]	$1.25
4299	1715	1	Steak Burrito	[Fresh Tomato Salsa, [Rice, Pinto Beans, Chees...	$11.75

1.2 Data Inspection and Exploration

shape of the data

In [8]:
```
chips_data.shape
```

Out[8]:
```
(4622, 5)
```

Info of the Data

In [9]:
```
chips_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4622 entries, 0 to 4621
Data columns (total 5 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   order_id            4622 non-null   int64 
 1   quantity            4622 non-null   int64 
 2   item_name           4622 non-null   object
 3   choice_description  3376 non-null   object
 4   item_price          4622 non-null   object
dtypes: int64(2), object(3)
memory usage: 180.7+ KB
```

This method provides a concise summary of the DataFrame, including the data types, non-null counts, and memory usage.

Basic Statistics of the data

In [10]:
```
chips_data.describe()
```

Out[10]:

	order_id	quantity
count	4622.000000	4622.000000
mean	927.254868	1.075725
std	528.890796	0.410186
min	1.000000	1.000000
25%	477.250000	1.000000
50%	926.000000	1.000000
75%	1393.000000	1.000000
max	1834.000000	15.000000

The method generates basic statistics for each numeric column in the DataFrame, such as count,

In [11]:
```
chips_data.describe(include='object')
```

Out[11]:

	item_name	choice_description	item_price
count	4622	3376	4622
unique	50	1043	78
top	Chicken Bowl	[Diet Coke]	$8.75
freq	726	134	730

(include='object') will describe about the object data. Similarly, you can use (include='all') if you want to see all at once.

Accessing Columns

In [12]:
```
chips_data.item_name
```

Out[12]:
```
0                Chips and Fresh Tomato Salsa
1                                        Izze
2                            Nantucket Nectar
3       Chips and Tomatillo-Green Chili Salsa
4                                Chicken Bowl
                        ...
4617                           Steak Burrito
4618                           Steak Burrito
4619                      Chicken Salad Bowl
4620                      Chicken Salad Bowl
4621                      Chicken Salad Bowl
Name: item_name, Length: 4622, dtype: object
```

In [13]:
```
# Accessing a single column
chips_data['item_name']
```

Out[13]:
```
0                Chips and Fresh Tomato Salsa
1                                        Izze
2                            Nantucket Nectar
3       Chips and Tomatillo-Green Chili Salsa
4                                Chicken Bowl
                        ...
4617                           Steak Burrito
4618                           Steak Burrito
4619                      Chicken Salad Bowl
4620                      Chicken Salad Bowl
4621                      Chicken Salad Bowl
Name: item_name, Length: 4622, dtype: object
```

If there is a column name with spaces, you should use this approach to acces a column.

Finding no.of unique values from each column

In [14]:
```
chips_data.nunique()
```

Out[14]:
```
order_id              1834
quantity                 9
item_name               50
choice_description    1043
item_price              78
dtype: int64
```

This method calculates the number of unique values in each column. It's handy for understanding the diversity of data in categorical columns.

Finding the count of unique values from each column

In [15]:
```
# As i only want to see top 5, gave head(5)
chips_data['item_name'].value_counts().head(5)
```

Out[15]:
```
Chicken Bowl           726
Chicken Burrito        553
Chips and Guacamole    479
Steak Burrito          368
Canned Soft Drink      301
Name: item_name, dtype: int64
```

Use this method on a specific column to count the occurrences of each unique value. It's particularly useful for categorical columns.

Finding stats for a specific column

In [16]:
```
chips_data.item_name.mode()
```

Out[16]:
```
0    Chicken Bowl
dtype: object
```

In [17]:
```
chips_data.quantity.mean()
```

Out[17]:
```
1.0757247944612722
```

2 - Data Selection and Indexing

2.1 Selecting Columns and Rows

Using squared brackets [ ]:

To select one or more columns by their names, you can use square brackets with the column names as a list.

In [18]:
```
# selecting multiple columns
numeric_data = chips_data[['order_id','quantity']]
```

In [19]:
```
numeric_data.head(3)
```

Out[19]:

	order_id	quantity
0	1	1
1	1	1
2	1	1

Using .loc[] for Label-Based Selection

- The .loc[rows, columns] indexer allows you to select rows and columns by label.
- You can specify both row and column labels.
- In the below example the row labels are also numbers, hence we use numbers for the rows.
- If you specify multiple rows or columns using index slicing, the inner and outer indices both are inclusive. Hence, 3,4,5,6 all the rows are included.

In [20]:
```
# For selecting specific rows and columns
chips_data.loc[3:6,['order_id','quantity']]
```

Out[20]:

	order_id	quantity
3	1	1
4	2	2
5	3	1
6	3	1

In [21]:
```
# For selecting a single cell from a specific row and column
chips_data.loc[100,'item_name']
```

Out[21]:
```
'Chips and Guacamole'
```

In [22]:
```
# For selecting a row values of a specific row
chips_data.loc[100,:]
```

Out[22]:
```
order_id                               44
quantity                                1
item_name             Chips and Guacamole
choice_description                    NaN
item_price                         $4.45
Name: 100, dtype: object
```

In [23]:
```
# For seleting all column values of a specific column
chips_data.loc[:,'item_name']
```

Out[23]:
```
0               Chips and Fresh Tomato Salsa
1                                       Izze
2                           Nantucket Nectar
3       Chips and Tomatillo-Green Chili Salsa
4                               Chicken Bowl
                        ...
4617                          Steak Burrito
4618                          Steak Burrito
4619                     Chicken Salad Bowl
4620                     Chicken Salad Bowl
4621                     Chicken Salad Bowl
Name: item_name, Length: 4622, dtype: object
```

By this time we know this can be easily done by chips_data['item_name'], but it's just good to know the capability of a feature to it's extent.

Using .iloc[] for Integer-Based Selection

- The .iloc[] indexer lets you select rows and columns by integer location, which is useful for numeric indexing.
- If you specify multiple rows or columns using index slicing, only the inner is inclusive, and the outer is exclusive. Hence only 1,2,3 rows will be shown and 0,1 columns will be shown.

In [24]:
```
chips_data.iloc[1:4,0:2]
```

Out[24]:

	order_id	quantity
1	1	1
2	1	1
3	1	1

2.2 Indexing Methods

Setting Index

In [25]:
```
chips_data.set_index('order_id').head(3)
```

Out[25]:

order_id	quantity	item_name	choice_description	item_price
1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	Izze	[Clementine]	$3.39
1	1	Nantucket Nectar	[Apple]	$3.39

- As you can observe now order_id has become the index. But this doesn't get saved until you modify this with your data frame. EG: chips_data = chips_data.set_index('order_id')
- If you directly wanna save it, then you can use inplace=True

In [26]:
```
chips_data.set_index('order_id',inplace=True)
```

In [27]:
```
chips_data.head(3)
```

Out[27]:

order_id	quantity	item_name	choice_description	item_price
1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	Izze	[Clementine]	$3.39
1	1	Nantucket Nectar	[Apple]	$3.39

Reset Index

In [28]:
```
chips_data.reset_index(inplace=True)
```

In [29]:
```
chips_data.head(3)
```

Out[29]:

	order_id	quantity	item_name	choice_description	item_price
0	1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	1	Izze	[Clementine]	$3.39
2	1	1	Nantucket Nectar	[Apple]	$3.39

3 - Data Cleaning

3.1 Handling Missing Data

Check Missing Data in a Column

In [30]:
```
chips_data.choice_description.isna()
```

Out[30]:
```
0        True
1       False
2       False
3        True
4       False
        ...
4617    False
4618    False
4619    False
4620    False
4621    False
Name: choice_description, Length: 4622, dtype: bool
```

- Applying the .isna() method for a column will return the boolean list with True for the indices where there is a missing value.
- And passing this list to a dataframe will return the rows where that column values are null.

In [31]:
```
# Rows where the choice_description column value is missing.
chips_data[chips_data.choice_description.isna()].head(5)
```

Out[31]:

	order_id	quantity	item_name	choice_description	item_price
0	1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
3	1	1	Chips and Tomatillo-Green Chili Salsa	NaN	$2.39
6	3	1	Side of Chips	NaN	$1.69
10	5	1	Chips and Guacamole	NaN	$4.45
14	7	1	Chips and Guacamole	NaN	$4.45

To Find the count of missing values in each column

In [32]:
```
chips_data.isna().sum()
```

Out[32]:
```
order_id                0
quantity                0
item_name               0
choice_description    1246
item_price              0
dtype: int64
```

In [33]:
```
# Similarly if you want to check the missing data count for one column
chips_data.choice_description.isna().sum()
```

Out[33]:
```
1246
```

Filling the missing Data using fillna

In [34]:
```
ufo_data.isna().sum()
```

Out[34]:
```
City                  25
Colors Reported    15359
Shape Reported      2644
State                  0
Time                   0
dtype: int64
```

- As for Categorical data we can fill the data with a value if we have any, if not we can prefer the mode.
- Mean imputation is often used when the missing values are numerical and the distribution of the variable is approximately normal.
- Median imputation is preferred when the distribution is skewed, as the median is less sensitive to outliers than the mean

In [35]:
```
most_repeated = ufo_data['Shape Reported'].mode()[0]
most_repeated
```

Out[35]:
```
'LIGHT'
```

In [36]:
```
# using inplace=True will automatically save the data, Here we are filling the m
ufo_data['Shape Reported'].fillna(most_repeated,inplace=True)
```

In [37]:
```
ufo_data.isna().sum()
```

Out[37]:
```
City                  25
Colors Reported    15359
Shape Reported         0
State                  0
Time                   0
dtype: int64
```

- Now you can observe that there are no null values in the Shapes Reported column as they are filled with the mode.
- If you want cross check further, you can check the value counts.

Drop Rows or columns with missing data using dropna

In [38]:
```
ufo_data.shape
```

Out[38]:
```
(18241, 5)
```

In [39]:
```
# This drops the rows where any of the column value is missing
ufo_data.dropna().head(3)
```

Out[39]:

	City	Colors Reported	Shape Reported	State	Time
12	Belton	RED	SPHERE	SC	6/30/1939 20:00
19	Bering Sea	RED	OTHER	AK	4/30/1943 23:00
36	Portsmouth	RED	FORMATION	VA	7/10/1945 1:30

- But, by doing so, we are losing a lot of data (15359) as that many rows doesn't have data for Colors Reported, in such cases, we can use subset to check only for certain columns while dropping.
- Now, The below case only checks for the city column, and if observed any missing data in the city column, that particular rows will be dropped.

In [40]:
```
# City only has 25 missing rows, we can drop the missing rows.
ufo_data.dropna(subset=['City']).head(3)
```

Out[40]:

	City	Colors Reported	Shape Reported	State	Time
0	Ithaca	NaN	TRIANGLE	NY	6/1/1930 22:00
1	Willingboro	NaN	OTHER	NJ	6/30/1930 20:00
2	Holyoke	NaN	OVAL	CO	2/15/1931 14:00

In [41]:
```
# Using inplace=True to save the data
ufo_data.dropna(subset=['City'],inplace=True)
```

In [42]:
```
ufo_data.shape
```

Out[42]:
```
(18216, 5)
```

In [43]:
```
ufo_data.dropna(how='all').head(3)
```

Out[43]:

	City	Colors Reported	Shape Reported	State	Time
0	Ithaca	NaN	TRIANGLE	NY	6/1/1930 22:00
1	Willingboro	NaN	OTHER	NJ	6/30/1930 20:00
2	Holyoke	NaN	OVAL	CO	2/15/1931 14:00

- By Using how = ‘any’, it will drop the rows where any of the column values are missing.

- By using how = ‘all’, it will drop the rows where all of the specified column values are missing.
- In this case, it hasn't dropped any row (observe the shape), as there isn't any row with all missing values.

3.2 Handing Duplicates

- Duplicate rows can skew your analysis results. Pandas offers a simple way to remove duplicates:

Finding Duplicates using .duplicated

1. By Default duplicated, uses keep='first’ , which keeps the first observed row in the dataframe and marks the later observed similar rows as True, which specifies they are duplicated ones.
2. If you want to keep the last observed duplicated row in the dataframe then you can give keep='last' .
3. If you want to see all the duplicates, then you can give keep='False'
4. If you want to check duplicates based on specific columns, then you need to give

In [44]:
```
# To check the count of duplicated rows
chips_data.duplicated().sum()
```

Out[44]: 59

In [45]:
```
# Results the duplicated rows when there is an any other row with exact match of
duplicates = chips_data[chips_data.duplicated()]
duplicates.head(3)
```

Out[45]:

	order_id	quantity	item_name	choice_description	item_price
238	103	1	Steak Burrito	[Tomatillo Red Chili Salsa, [Rice, Black Beans...	$11.75
248	108	1	Canned Soda	[Mountain Dew]	$1.09
297	129	1	Steak Burrito	[Tomatillo Green Chili Salsa, [Rice, Cheese, G...	$11.75

In [46]:
```
# Results the duplicated columns when there is a match of the specified columns
id_price_duplicates = chips_data[chips_data.duplicated(subset=['order_id','item_
id_price_duplicates.head(3)
```

Out[46]:

	order_id	quantity	item_name	choice_description	item_price
2	1	1	Nantucket Nectar	[Apple]	$3.39
3	1	1	Chips and Tomatillo-Green Chili Salsa	NaN	$2.39
12	6	1	Chicken Soft Tacos	[Roasted Chili Corn Salsa, [Rice, Black Beans,...	$8.75

Drop the duplicated rows using .drop_duplicates

In [47]:
```
chips_data.duplicated().sum()
```

Out[47]:
```
59
```

In [48]:
```
# As we know there are only 59 duplicated rows and we choose to drop them, then w
chips_data.drop_duplicates().head(3)
```

Out[48]:

	order_id	quantity	item_name	choice_description	item_price
0	1	1	Chips and Fresh Tomato Salsa	NaN	$2.39
1	1	1	Izze	[Clementine]	$3.39
2	1	1	Nantucket Nectar	[Apple]	$3.39

In [49]:
```
# It is advisable to check the shape after dropping before saving it
chips_data.drop_duplicates().shape
```

Out[49]:
```
(4563, 5)
```

In [50]:
```
# If you are satisfied with the resulting shape, you can save it.
chips_data.drop_duplicates(inplace=True)
```

3.3 String Operations

When working with text data, Pandas offers string operations through .str an accessor to apply for the entire column which is of object data type.

- .str.lower() and .str.upper(): These methods convert strings to lowercase or uppercase for the entire column values.
- .str.replace(): Use this method to replace substrings within strings.
- .str.Contains() : This method allows you to check if a specific substring or pattern exists within a string. It returns a boolean Series indicating whether each element contains the specified pattern.
- .str.slice(): You can extract a substring from each string in a Series using the .str.slice() method. Specify the start and end positions to define the slice.

In [51]:
```
chips_data.item_name
```

Out[51]:
```
0                Chips and Fresh Tomato Salsa
1                                        Izze
2                            Nantucket Nectar
3       Chips and Tomatillo-Green Chili Salsa
4                                Chicken Bowl
                        ...
4617                            Steak Burrito
4618                            Steak Burrito
4619                       Chicken Salad Bowl
4620                       Chicken Salad Bowl
4621                       Chicken Salad Bowl
Name: item_name, Length: 4563, dtype: object
```

In [52]:
```
# say we want to convert all the values in item_name to upper case
chips_data.item_name.str.upper()
```

Out[52]:
```
0                CHIPS AND FRESH TOMATO SALSA
1                                        IZZE
2                            NANTUCKET NECTAR
3       CHIPS AND TOMATILLO-GREEN CHILI SALSA
4                                CHICKEN BOWL
                        ...
4617                            STEAK BURRITO
4618                            STEAK BURRITO
4619                       CHICKEN SALAD BOWL
4620                       CHICKEN SALAD BOWL
4621                       CHICKEN SALAD BOWL
Name: item_name, Length: 4563, dtype: object
```

In [53]:
```
# saving the original data with the modified one, as it doesn't have inplace opt
chips_data.item_name = chips_data.item_name.str.upper()
```

In [54]:
```
# Using contains to find how many rows have item names with chicken
chips_data.item_name.str.contains('CHICKEN').sum()
```

Out[54]:
```
1540
```

3.4 Data Type Conversion

In [55]:
```
chips_data.item_price
```

Out[55]:
```
0        $2.39
1        $3.39
2        $3.39
3        $2.39
4       $16.98
         ...
4617    $11.75
4618    $11.75
4619    $11.25
4620     $8.75
4621     $8.75
Name: item_price, Length: 4563, dtype: object
```

We can see that item price is in object type as it has the $ dollar symbol, but it is supposed to be in float. To convert the data types we luckily have a function in pandas - .astype

In [56]:
```
# As now we know about string operations, we can also utilize that here to remove
chips_data.item_price.str.replace('$','')
```

Out[56]:
```
0        2.39
1        3.39
2        3.39
3        2.39
4       16.98
         ...
4617    11.75
4618    11.75
4619    11.25
4620     8.75
4621     8.75
Name: item_price, Length: 4563, dtype: object
```

- But the type is still in object!
- So to change that we need to use .astype

In [57]:
```
chips_data.item_price.str.replace('$','').astype('float')
```

Out[57]:
```
0        2.39
1        3.39
2        3.39
3        2.39
4       16.98
        ...
4617    11.75
4618    11.75
4619    11.25
4620     8.75
4621     8.75
Name: item_price, Length: 4563, dtype: float64
```

- Here you can see that the type has changed to float and now we can perform mathematical operations on this series.

In [58]:
```
# Saving the series
chips_data.item_price = chips_data.item_price.str.replace('$','').astype('float'
```

4 - Data Manipulation

Data manipulation is a core task in data analysis and involves transforming and modifying your data to derive insights or prepare it for further analysis.

4.1 Applying Functions to DataFrames

.apply for series and DataFrames

- Use this method to apply a custom function to a series or to the entire dataframe.
- when you use this on series, Each element of the original column will be passed to the function.
- when you use this for the entire dataframe, based on the axis (1 - row, 0 -column), the entire row or the entire column will be passed to the function.

In [59]:
```
# say we want to normalize the price.
min_price = chips_data.item_price.min()
max_price = chips_data.item_price.max()
def normalize(x):
    return (x-min_price)/(max_price-min_price)

normalized_price = chips_data.item_price.apply(normalize)
normalized_price
```

Out[59]:
```
0       0.030120
1       0.053290
2       0.053290
3       0.030120
4       0.368165
          ...
4617    0.246988
4618    0.246988
4619    0.235403
4620    0.177479
4621    0.177479
Name: item_price, Length: 4563, dtype: float64
```

In the above case, we used apply function on a particular series, and so each item of the series is sent to the function.

In [60]:
```
# Sample DataFrame
data = {'A': [1, 2, 3, 1, 2], 'B': [4, 5, 6, 4, 5]}
df = pd.DataFrame(data)

# Define a function to calculate the average of a row
def average_row(row):
    return row.mean()

# Apply the function row-wise.
df['Row_Average'] = df.apply(average_row, axis=1)
df
```

Out[60]:

	A	B	Row_Average
0	1	4	2.5
1	2	5	3.5
2	3	6	4.5
3	1	4	2.5
4	2	5	3.5

- In this case, we used apply function on the entire dataframe and used the axis=1, so each row will be sent to the function.
- If you had to perform opeartion on row wise, you should opt this method.

.map for series

It's particularly useful for transforming one column based on values from another.

In [61]:
```
# Mapping values in a column based on a dictionary
mapping_dict = {1: 'one', 2: 'two', 3: 'three'}
df['Encoded_A'] = df['A'].map(mapping_dict)
df
```

Out[61]:

	A	B	Row_Average	Encoded_A
0	1	4	2.5	one
1	2	5	3.5	two
2	3	6	4.5	three
3	1	4	2.5	one
4	2	5	3.5	two

.applymap for entire dataframe

When you want to apply a function to each element in the entire DataFrame, you can use .applymap().

In [62]:
```
# Sample DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data)

# Define a function to add 10 to a value
def add_10(x):
    return x + 10

# Apply the function to the entire DataFrame
df = df.applymap(add_10)
df
```

Out[62]:

	A	B
0	11	14
1	12	15
2	13	16

4.2 Adding and Removing Columns

In [63]:
```
# To add a new column, we just give the name of the column in [] and equate it to
chips_data['normalized_price'] = normalized_price
```

In [64]:
```
chips_data.head(3)
```

Out[64]:

	order_id	quantity	item_name	choice_description	item_price	normalized_price
0	1	1	CHIPS AND FRESH TOMATO SALSA	NaN	2.39	0.03012
1	1	1	IZZE	[Clementine]	3.39	0.05329
2	1	1	NANTUCKET NECTAR	[Apple]	3.39	0.05329

In [65]:
```
# Using .drop() to remove columns
ufo_data.head(3)
```

Out[65]:

	City	Colors Reported	Shape Reported	State	Time
0	Ithaca	NaN	TRIANGLE	NY	6/1/1930 22:00
1	Willingboro	NaN	OTHER	NJ	6/30/1930 20:00
2	Holyoke	NaN	OVAL	CO	2/15/1931 14:00

In [66]:
```
# if we don't want colors reported and shape reported columns
ufo_data.drop(['Colors Reported','Shape Reported'], axis=1, inplace=True)
```

In [67]:
```
ufo_data.head(3)
```

Out[67]:

	City	State	Time
0	Ithaca	NY	6/1/1930 22:00
1	Willingboro	NJ	6/30/1930 20:00
2	Holyoke	CO	2/15/1931 14:00

4.3 Combining DataFrames

Concatenation of DataFrames

- You can concatenate DataFrames vertically or horizontally using pd.concat().
- axis=0 will concatenate them in the rows, axis=1 will concatenate them in the columns.
- It will check for the common columns between both the dataframes and for the matched columns, it will concatenate in the rows.

In [68]:

```
# Sample DataFrames with the same column names
data1 = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C':[1,2,3]}
data2 = {'A': [7, 8, 9], 'B': [10, 11, 12], 'D':[1,2,3]}
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Concatenate df1 and df2 horizontally (along columns) with same column names
horizontal_concat = pd.concat([df1, df2], axis=0)

# Display the concatenated DataFrame
print(horizontal_concat)
```

```
   A   B    C    D
0  1   4  1.0  NaN
1  2   5  2.0  NaN
2  3   6  3.0  NaN
0  7  10  NaN  1.0
1  8  11  NaN  2.0
2  9  12  NaN  3.0
```

- Use axis = 1, if you want vertical concatenation.

In [69]:

```
# To get the proper index
horizontal_concat.reset_index(drop=True)
```

Out[69]:

	A	B	C	D
0	1	4	1.0	NaN
1	2	5	2.0	NaN
2	3	6	3.0	NaN
3	7	10	NaN	1.0
4	8	11	NaN	2.0
5	9	12	NaN	3.0

Merging using inner join

In [70]:
```
# Sample DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Age': [25, 30, 22]})

# Inner join on 'ID'
result = pd.merge(df1, df2, on='ID', how='inner')

# Display the merged DataFrame
print("Inner Join")
print(result)
```

```
Inner Join
   ID     Name  Age
0   2      Bob   25
1   3  Charlie   30
```

- We have to specify on which common columns, we want to check for the matching cells.
- Here, we have ID for both the dataframes and we want it to merge the rows where there are common ids in both the dataframes, in such cases we need to use inner.
- As we have 2,3 common in both the IDs, we only got those as result.

Merging using left join

In [71]:
```
# Left join on 'ID'
result = pd.merge(df1, df2, on='ID', how='left')

# Display the merged DataFrame
print("Left Join")
print(result)
```

```
Left Join
   ID     Name   Age
0   1    Alice   NaN
1   2      Bob  25.0
2   3  Charlie  30.0
```

- When you use left join, it will keep all the rows from the first given table. And when a row is not there in the second dataframe, that value will just be null.

Merging for columns with mismatched names

- When you want to merge based on a column with two different names on both the datasets, then you need to specify the parameters left_on and right_on.
- And in this case, it will keep both the columns after merge.

In [72]:
```
# Sample DataFrames
df1 = pd.DataFrame({'ID1': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID2': [2, 3, 4], 'Age': [25, 30, 22]})

# Inner join on 'ID'
result = pd.merge(df1, df2, left_on='ID1', right_on='ID2', how='inner')

# Display the merged DataFrame
print("Inner Join")
print(result)
```

```
Inner Join
   ID1     Name  ID2  Age
0    2      Bob    2   25
1    3  Charlie    3   30
```

Merging one df column with other df index

- When you want to merge one data frame column with other dataframe index, you need to use, left_index=True or right_index=True, based on which index you want to compare on.

In [73]:
```
# Sample DataFrames
df1 = pd.DataFrame({'ID': [1, 2, 3], 'Name': ['Alice', 'Bob', 'Charlie']})
df2 = pd.DataFrame({'ID': [2, 3, 4], 'Age': [25, 30, 22]})

# Setting ID Column as the index for the df1 table
df1.set_index('ID',inplace=True)

# Inner join on index for the left table and 'ID' column for the right table
result = pd.merge(df1, df2, left_index=True, right_on='ID', how='inner')

# Display the merged DataFrame
print("Inner Join")
print(result)
```

```
Inner Join
      Name  ID  Age
0      Bob   2   25
1  Charlie   3   30
```

- similarly, if you want both of them to be merged on index, then you need to use left_index=True and right_index=True

5 - Data Aggregation

5.1 Grouping Data

Group with a single column using groupby method.

- This method allows you to group data based on one or more columns. You can think of it as a powerful version of the SQL GROUP BY statement.
- First, you need to pass the column name on which you want to group the data.
- After that, you can use the grouped data and choose the column between which you want to compare this grouped data, and then select the aggregate function(mean, sum, max, min, etc..).
- When you apply an aggregation function to grouped data without specifying a column, it will be applied to all the numeric columns in the DataFrame.

In [74]:
```
# Sample DataFrame
data = {'Class': ['A', 'B', 'A', 'B', 'A', 'B'],
        'Gender': ['Male', 'Male', 'Female', 'Female', 'Male', 'Female'],
        'Math_Score': [85, 92, 78, 89, 90, 86],
        'English_Score': [88, 94, 80, 92, 92, 88],
        'Physics_Score': [78, 90, 85, 92, 88, 84]}
df = pd.DataFrame(data)


# Grouping by 'Category'
grouped_data = df.groupby('Gender')

# Choosing sales column to compare with grouped data and using sum function
# This gives the total sales for each category.
total_sales = grouped_data['Math_Score'].sum()

print(total_sales)
```

```
Gender
Female    253
Male      267
Name: Math_Score, dtype: int64
```

Group with single column and apply to entire Dataframe.

In [75]:
```
# Grouping by 'Class' and 'Gender'
grouped_data = df.groupby('Gender')

# Applying the mean aggregation function to all numeric columns
aggregated_data = grouped_data.mean()

print(aggregated_data)
```

```
        Math_Score  English_Score  Physics_Score
Gender
Female   84.333333      86.666667      87.000000
Male     89.000000      91.333333      85.333333
```

Group with multiple columns

In [76]:
```
# Grouping by 'Class' and 'Gender' and calculating statistics
grouped_data = df.groupby(['Class', 'Gender'])

# Calculate the mean for Math_score
agg_results = grouped_data['Math_Score'].mean()

print(agg_results)
```

```
Class  Gender
A      Female    78.0
       Male      87.5
B      Female    87.5
       Male      92.0
Name: Math_Score, dtype: float64
```

In [77]:
```
# Grouping by 'Class' and 'Gender' and calculating statistics
grouped_data = df.groupby(['Class', 'Gender'])

# Calculate the mean for all numeric columns
agg_results = grouped_data.mean()

print(agg_results)
```

```
              Math_Score  English_Score  Physics_Score
Class Gender
A     Female        78.0           80.0           85.0
      Male          87.5           90.0           83.0
B     Female        87.5           90.0           88.0
      Male          92.0           94.0           90.0
```

5.2 Aggregate Functions

- Aggregation functions are essential for summarizing data within groups.

- Common Aggregation Functions are sum(), max(), min(), mean(), median(), count(), agg()-this

Apply multiple aggregate functions to the grouped data.

In [78]:
```
# Grouping by 'Class' and 'Gender' and calculating statistics
grouped_data = df.groupby(['Class', 'Gender'])

# Calculate the mean, min, and max scores for Math_score
agg_results = grouped_data.Math_Score.agg(['mean', 'min', 'max'])

print(agg_results)
```

```
               mean  min  max
Class Gender
A     Female  78.0   78   78
      Male    87.5   85   90
B     Female  87.5   86   89
      Male    92.0   92   92
```

Apply multiple aggregate functions for selected columns.

In [79]:
```
# Applying aggregation functions to 'Math_Score' and 'Physics_Score'
aggregated_data = grouped_data.agg({
    'Math_Score': ['mean', 'min', 'max'],
    'Physics_Score': ['mean', 'min', 'max']
})

print(aggregated_data)
```

```
              Math_Score          Physics_Score
                    mean min max          mean min max
Class Gender
A     Female       78.0  78  78            85  85  85
      Male         87.5  85  90            83  78  88
B     Female       87.5  86  89            88  84  92
      Male         92.0  92  92            90  90  90
```

5.3 Pivot Tables and Cross-Tabulations

- we can use pd.pivot_table to create pivot tables.
- Cross-tabulations (crosstabs) are another method to aggregate data, especially when dealing with categorical variables using pd.crosstab

In [80]:
```
import pandas as pd

# Sample DataFrame with sales data
data = {'Category': ['Electronics', 'Clothing', 'Electronics', 'Clothing'],
        'Region': ['North', 'South', 'North', 'South'],
        'Sales': [1000, 500, 800, 750],
        'Profit': [150, 50, 120, 100]}
df = pd.DataFrame(data)

# Pivot Table: Sum of Sales by Category and Region
pivot_table = pd.pivot_table(df, index='Category', columns='Region', values='Sale

# Cross-Tabulation: Count of Category by Region
cross_tab = pd.crosstab(df['Category'], df['Region'])

print("Pivot Table:")
print(pivot_table)

print("\nCross-Tabulation:")
print(cross_tab)
```

```
Pivot Table:
Region        North   South
Category
Clothing        NaN  1250.0
Electronics  1800.0     NaN

Cross-Tabulation:
Region       North  South
Category
Clothing         0      2
Electronics      2      0
```

6 - Data Visualizations

6.1 Find the Correlations

- To get the correlation of each column with every other column you can use dataframe.corr() .
- The numbers closer to +1 are highly positively correlated and numbers closer to -1 are highly negatively correlated.

In [81]:
```
chips_data.corr()
```

Out[81]:

	order_id	quantity	item_price	normalized_price
order_id	1.000000	0.032684	-0.000574	-0.000574
quantity	0.032684	1.000000	0.264701	0.264701
item_price	-0.000574	0.264701	1.000000	1.000000
normalized_price	-0.000574	0.264701	1.000000	1.000000

6.2 Sorting Data and Creating Plots

Sorting Data

.sort_values() : Use this method to sort a series or dataframe. You can use the by parameter to specify based on which column you want to sort, and use ascending the parameter to set ascending or descending.

In [82]:
```
chips_data.sort_values(by='item_price')
```

Out[82]:

	order_id	quantity	item_name	choice_description	item_price	normalized_price
3477	1396	1	CANNED SODA	[Dr. Pepper]	1.09	0.000000
2754	1093	1	BOTTLED WATER	NaN	1.09	0.000000
2768	1098	1	CANNED SODA	[Sprite]	1.09	0.000000
179	81	1	CANNED SODA	[Coca Cola]	1.09	0.000000
180	81	1	CANNED SODA	[Dr. Pepper]	1.09	0.000000
...	...	...	...	...	...	...
3601	1443	3	VEGGIE BURRITO	[Fresh Tomato Salsa, [Fajita Vegetables, Rice,...	33.75	0.756719
1254	511	4	CHICKEN BURRITO	[Fresh Tomato Salsa, [Fajita Vegetables, Rice,...	35.00	0.785681
3602	1443	4	CHICKEN BURRITO	[Fresh Tomato Salsa, [Rice, Black Beans, Chees...	35.00	0.785681
3480	1398	3	CARNITAS BOWL	[Roasted Chili Corn Salsa, [Fajita Vegetables,...	35.25	0.791474
3598	1443	15	CHIPS AND FRESH TOMATO SALSA	NaN	44.25	1.000000

4563 rows × 6 columns

Creating Plots

Here are some plots you can plot with pandas. In the X and y, you can specify the series which you want to plot with.

1. Line Plot: df.plot(x='X', y='Y', kind='line')
2. Bar Plot: df.plot(x='Category', y='Count', kind='bar')
3. Barh Plot (Horizontal Bar Plot): df.plot(x='Count', y='Category', kind='barh')
4. Histogram: df['Value'].plot(kind='hist', bins=20)
5. Box Plot: df.plot(y='Value', kind='box')
6. Area Plot: df.plot(x='X', y='Y', kind='area')
7. Scatter Plot: df.plot(x='X', y='Y', kind='scatter')

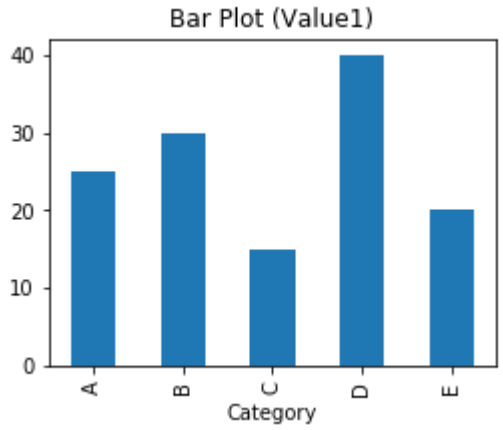
8. Pie Chart: df['Category'].value_counts().plot(kind='pie')
9. Hexbin Plot: df.plot(x='X', y='Y', kind='hexbin', gridsize=20)
10. Stacked Bar Plot: df.pivot_table(index='Category', columns='Subcategory', values='Value', aggfunc='sum').plot(kind='bar', stacked=True)
11. Line plot with multiple Lines: df.plot(x='Date',y=['Series1','Series2'],kind='line')

Advanced Plots:

1. KDE Plot (Kernel Density Estimate): df['Value'].plot(kind='kde')
2. Density Plot: df['Value'].plot(kind='density')
3. Boxen Plot: df.plot(y='Value', kind='boxen')

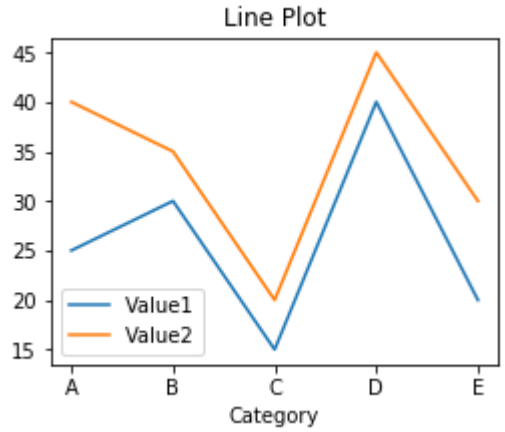
In [83]:
```
import pandas as pd

# Create a sample DataFrame
data = {'Category': ['A', 'B', 'C', 'D', 'E'],
        'Value1': [25, 30, 15, 40, 20],
        'Value2': [40, 35, 20, 45, 30]}
df = pd.DataFrame(data)

# Set 'Category' column as the index
df.set_index('Category', inplace=True)

# Create various plots using Pandas
df['Value1'].plot(kind='bar', title='Bar Plot (Value1)',figsize=(4, 3))
```

Out[83]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x268627398c8>
```



In [84]:
```
df.plot(kind='line', title='Line Plot',figsize=(4, 3))
```

Out[84]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x268632f2088>
```



7 - Time Series Data Handling

7.1 Working with DateTime Data

- pd.to_datetime : This method allows you to convert your series with datetime to a pandas datetime series through which you can do much more analysis seamlessly. You can get the year,month,day, and hours.

In [85]:
```
import pandas as pd

# Sample DataFrame with a DateTime column
data = {'DateTime': ['2023-01-01 08:30:00', '2023-02-01 14:45:00', '2023-03-01 2(
df = pd.DataFrame(data)

# Convert the 'DateTime' column to DateTime
df['DateTime'] = pd.to_datetime(df['DateTime'])

# Extract year, month, day, and hour
df['Year'] = df['DateTime'].dt.year
df['Month'] = df['DateTime'].dt.month
df['Day'] = df['DateTime'].dt.day
df['Hour'] = df['DateTime'].dt.hour

print(df)
```

```
             DateTime  Year  Month  Day  Hour
0 2023-01-01 08:30:00  2023      1    1     8
1 2023-02-01 14:45:00  2023      2    1    14
2 2023-03-01 20:15:00  2023      3    1    20
```

7.2 Resampling and Shifting

Resampling

Resampling is the process of changing the frequency of your time series data. It allows you to aggregate or transform data from one time frequency to another. Common reasons for resampling include:

1. Aggregation: You may have high-frequency data (e.g., daily) and want to aggregate it to a lower frequency (e.g., monthly) to get a broader overview of the data.
2. Interpolation: You may have data at irregular intervals and want to resample it to a regular frequency for analysis or visualization.

In [86]:
```
import pandas as pd

# Sample DataFrame with daily sales data
data = {'Date': pd.date_range(start='2023-01-01', periods=40, freq='D'),
        'Sales': [i for i in range(40)]}
df = pd.DataFrame(data)

# Resample data to monthly frequency, calculating the sum of sales
monthly_sales = df.resample('M', on='Date').sum()
print("Data Before resampling:")
print(df.head(5))
print("\nData After resampling:")
print(monthly_sales)
```

```
Data Before resampling:
        Date  Sales
0 2023-01-01      0
1 2023-01-02      1
2 2023-01-03      2
3 2023-01-04      3
4 2023-01-05      4

Data After resampling:
            Sales
Date
2023-01-31    465
2023-02-28    315
```

In this example, we resample daily sales data to monthly frequency, aggregating it by summing the sales for each month.

Shifting

It is also known as time lag or time shifting, involves moving data points forward or backward in time. It is often used to calculate differences or time-based features in time series data. Common use cases for shifting include:

1. Calculating Differences: You can calculate the difference between the current data point and a previous or future data point. This is useful for understanding trends or changes in the data.
2. Time Lags: You may want to create time lags of a variable to analyze how past values of that variable affect future outcomes.

In [87]:
```
import pandas as pd

# Sample DataFrame with daily stock prices
data = {'Date': pd.date_range(start='2023-01-01', periods=5, freq='D'),
        'Price': [100, 105, 110, 108, 112]}
df = pd.DataFrame(data)

# Calculate one-day price changes (time lag of 1 day)
df['Price_Change'] = df['Price'] - df['Price'].shift(1)
df
```

Out[87]:

	Date	Price	Price_Change
0	2023-01-01	100	NaN
1	2023-01-02	105	5.0
2	2023-01-03	110	5.0
3	2023-01-04	108	-2.0
4	2023-01-05	112	4.0

In this example, we calculate the one-day price changes by subtracting the previous day's price from the current day's price.

7.3 Rolling Statistics

- Rolling statistics, also known as rolling calculations or rolling windows, are a common technique in time series analysis. They involve applying a statistical function to a fixed-size window of data points that "rolls" or moves through the dataset one step at a time.
- Rolling statistics can smooth out noise in time series data, making underlying patterns more visible. They help detect trends or patterns over time, such as moving averages.

- Window Size: The rolling window has a fixed size specified by you. This size determines how many data points are considered in each calculation. For example, a window size of 3 means that you consider the current data point and the two previous data points in each calculation.
- Rolling Function: You apply a specific function to the data only within the rolling window. Common functions include mean, sum, standard deviation, and more.

In [88]:
```
import pandas as pd

# Sample DataFrame with daily stock prices
data = {'Date': pd.date_range(start='2023-01-01', periods=10, freq='D'),
        'Price': [100, 105, 110, 108, 112, 115, 118, 120, 122, 125]}
df = pd.DataFrame(data)

# Calculate the 3-day rolling mean (moving average) of prices
df['Rolling_Mean'] = df['Price'].rolling(window=3).mean()
df
```

Out[88]:

	Date	Price	Rolling_Mean
0	2023-01-01	100	NaN
1	2023-01-02	105	NaN
2	2023-01-03	110	105.000000
3	2023-01-04	108	107.666667
4	2023-01-05	112	110.000000
5	2023-01-06	115	111.666667
6	2023-01-07	118	115.000000
7	2023-01-08	120	117.666667
8	2023-01-09	122	120.000000
9	2023-01-10	125	122.333333

8 - Handling Categorical Data

- Categorical data represents information that has distinct categories or labels.
- It's common in data analysis, but it needs special treatment to be used effectively, such a way that it can be used for the machine learning models.

8.1 Encoding Categorical Variables

Encoding categorical variables involves converting them into a numerical format that machine learning algorithms can understand.

One-Hot Encoding using pd.get_dummies

One-hot encoding creates binary columns for each category, indicating the presence or absence of a category for each data point. In a column if there are k unique values, it will create k new binary columns one for each. It removes the original column after encode.

In [89]:
```
import pandas as pd

# Sample DataFrame with a categorical column
data = {'Category': ['A', 'B', 'A', 'C', 'B'],
         'Count':[1,2,3,4,5]}
df = pd.DataFrame(data)

# Perform one-hot encoding
encoded_df = pd.get_dummies(df, columns=['Category'])
print(encoded_df)
```

```
   Count  Category_A  Category_B  Category_C
0      1           1           0           0
1      2           0           1           0
2      3           1           0           0
3      4           0           0           1
4      5           0           1           0
```

Label Encoding using Category type

It is another approach in which it assigns a unique numerical value to each category. .astype('category').cat.codes method is used to convert category columns to label encoding.

In [90]:
```
import pandas as pd

# Sample DataFrame with a categorical column
data = {'Category': ['A', 'B', 'A', 'C', 'B']}
df = pd.DataFrame(data)

# Perform label encoding
df['Category_Encoded'] = df['Category'].astype('category').cat.codes
print(df)
```

```
  Category  Category_Encoded
0        A                 0
1        B                 1
2        A                 0
3        C                 2
4        B                 1
```

8.2 Sorting Ordinal Data

- Ordinal data represents categories with a natural order or ranking, such as low, medium, high, or small, medium, or large.
- We can achieve this by converting the Ordinal column to a Pandas categorical column using pd.Categorical() , specifying the categories parameter as ordinal_order and setting ordered=True.

In [91]:
```
import pandas as pd

# Sample DataFrame with an ordinal column
data = {'Product': ['Product A', 'Product B', 'Product C', 'Product D'],
        'Size': ['Medium', 'Small', 'Large', 'Medium']}
df = pd.DataFrame(data)

# Define the custom ordinal order
ordinal_order = ['Small', 'Medium', 'Large']

print("Before Sorting Ordinal Data:\n")
print(df.sort_values(by='Size'))

# Sort the DataFrame based on the 'Size' column
df['Size'] = pd.Categorical(df['Size'], categories=ordinal_order, ordered=True)

print("\nAfter Sorting Ordinal Data:\n")
print(df.sort_values(by='Size'))
```

```
Before Sorting Ordinal Data:

     Product    Size
2  Product C   Large
0  Product A  Medium
3  Product D  Medium
1  Product B   Small

After Sorting Ordinal Data:

     Product    Size
1  Product B   Small
0  Product A  Medium
3  Product D  Medium
2  Product C   Large
```

9 - Advanced Topics

9.1 Multi-Indexing

Multi-indexing, also known as hierarchical indexing, allows you to create DataFrame structures with multiple levels of index hierarchy. It's especially useful for handling data with complex, multi-dimensional relationships.

In [92]:
```
import pandas as pd

# Sample hierarchical DataFrame
data = {'Department': ['HR', 'HR', 'Engineering', 'Engineering'],
        'Employee': ['Alice', 'Bob', 'Charlie', 'David'],
        'Salary': [60_000, 65_000, 80_000, 75_000]}
df = pd.DataFrame(data)

# Create a hierarchical index
hierarchical_df = df.set_index(['Department', 'Employee'])

# Accessing data
print(hierarchical_df.loc[('HR','Bob')])  # Access HR Bob department data
```

```
Salary    65000
Name: (HR, Bob), dtype: int64
```

In [93]:
```
# Now it has multi-index wtih Department and Employee
hierarchical_df
```

Out[93]:

		Salary
Department	**Employee**	
HR	**Alice**	60000
	Bob	65000
Engineering	**Charlie**	80000
	David	75000

9.2 Handling Outliers

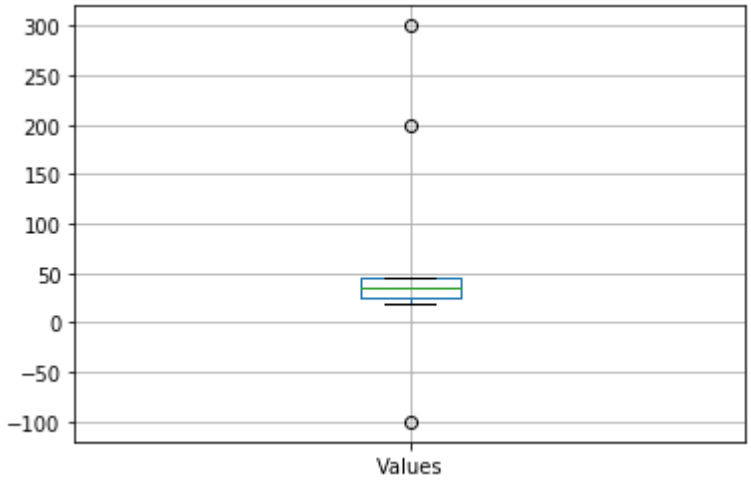
Handling outliers is an essential step in data preprocessing, as outliers can significantly impact the results of your analysis and statistical models. There are other ways to handle these using Scipy as well, but here's how we do it with pandas.

Identifying Outliers

Handling outliers is an essential step in data preprocessing, as outliers can significantly impact the results of your analysis and statistical models. There are other ways to handle these using Scipy as well, but here's how we do it with pandas.

In [94]:
```
# Create a sample DataFrame
data = {'Values': [25, 30, 200, 40, 20, 300, 35, 45, -100]}
df = pd.DataFrame(data)

# Box plot to visualize outliers
df.boxplot(column='Values')
```

Out[94]: <matplotlib.axes._subplots.AxesSubplot at 0x26864410bc8>



Outlier Handling by NaN

Once you've identified outliers, you can handle them manually. You have a few options. You can replace outlier values with NaN to effectively remove them from calculations. If you decide, here's how to do it.

In [95]:
```
upper_threshold = 100  # Define your threshold for outliers
lower_threshold = 0
df['Values'][(df['Values'] > upper_threshold) | (df['Values'] < lower_threshold)
```

In [96]:
```
df
```

Out[96]:

	Values
0	25.0
1	30.0
2	NaN
3	40.0
4	20.0
5	NaN
6	35.0
7	45.0
8	NaN

In this example, values 200,300, and -100 are outliers. So as they are out of threshold values, they will replaced with NaN

Outlier Handling by Clip Values

- Clipping is the process of setting upper and lower bounds on a variable to limit extreme values to a specified range.
- It replaces values in the DataFrame with the specified bounds if they fall outside the specified range.

In [97]:
```
# Create a sample DataFrame
data = {'Values': [25, 30, 200, 40, 20, 300, 35, 45, -100]}
df = pd.DataFrame(data)

# Define upper and lower bounds
lower_bound = 0
upper_bound = 100

# Clip values to the specified bounds
df['Values'] = df['Values'].clip(lower=lower_bound, upper=upper_bound)
```

In [98]:
```
df
```

Out[98]:

	Values
0	25
1	30
2	100
3	40
4	20
5	100
6	35
7	45
8	0

In this example, values 200,300, and -100 are outliers. So once we set a range of lower bound and upper bound, we can pass it to the clip function. Any values higher than the upper bound will be replaced with the upper bound values, and similarly lower bound value for the values lesser than the lower bound. So, 200 and 300 will be replaced with 100 and -100 will be replaced with 0.

10 - Memory Optimization

Optimizing memory usage is crucial when working with large datasets.

Optimizing memory usage is crucial when working with large datasets. Pandas provide techniques to reduce memory consumption while maintaining data integrity.

1. Choose the Right Data Types:
2. Use appropriate data types for your columns. For example, use int8 or int16 for integer columns with small value ranges, and float32 for floating-point columns.
3. Consider using categorical data types for columns with a limited number of unique values. This reduces memory usage and can improve performance when working with categorical data.
4. Use Sparse Data Structures if your data has too many missing values:
5. Pandas support sparse data structures, such as SparseDataFrame and SparseSeries, which are suitable for datasets with a lot of missing values.
6. Sparse data structures store only non-missing values, reducing memory usage.
7. Read Data in Chunks:
8. When reading large datasets from external sources, use the chunksize parameter of functions like read_csv() to read the data in smaller chunks rather than loading the entire dataset into memory at once.
9. Release Unneeded DataFrames:
10. Explicitly release memory by deleting DataFrames or Series that are no longer needed, using the del keyword. This frees up memory for other operations.
11. Optimize GroupBy Operations:
12. Use the as_index=False parameter when performing GroupBy operations to avoid creating a new index, which can consume additional memory

In [ ]: