A Quick Reference Handbook of the Essential Data Science Libraries

The One Guide to Kick Start your Data Science Journey.

- By RaviTeja G

Table of Contents

Useful Links

1. Check out my Github Profile for Detailed Code Files.
2. Check out my Medium Profile for Detailed Articles.

As part of my 100 Days Machine Learning and Deep Learning Challenge I Created these notes for my Revision and Quick Reference, and I felt it will be useful for others as well. Hence, Sharing it here, Happy Learning :)

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

Table of Contents

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]:
```python
# # 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]:
```python
# 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]:
```python
# 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]:

	quantity	item_name	choice_description	item_price
order_id				
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]:

	quantity	item_name	choice_description	item_price
order_id				
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 mi
         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]:
```python
# 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]:
```python
# 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 t
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]:
```python

# 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]:
```python
# 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')

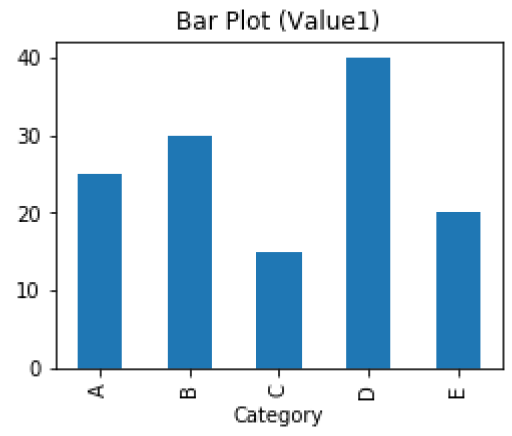
8. Pie Chart: df['Category'].value_counts().plot(kind='pie')
9. Hexbin Plot: df.plot(x='X', y='Y', kind='hexbin', gridsize=20)
10. Stacked Bar Plot: df.pivot_table(index='Category', columns='Subcategory', values='Value', aggfunc='sum').plot(kind='bar', stacked=True)
11. Line plot with multiple Lines: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

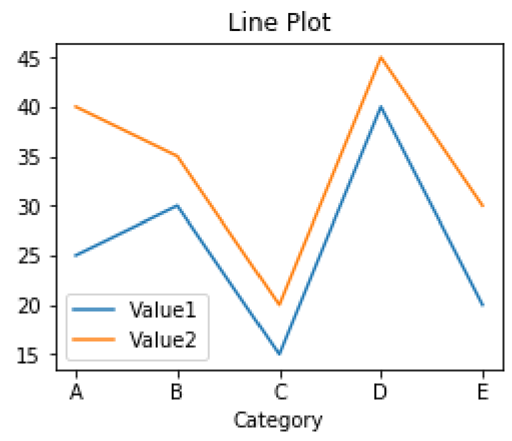
In [83]:
```
import pandas as pd

# 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 20
         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]:
```python
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]:
```python
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]:
```python
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

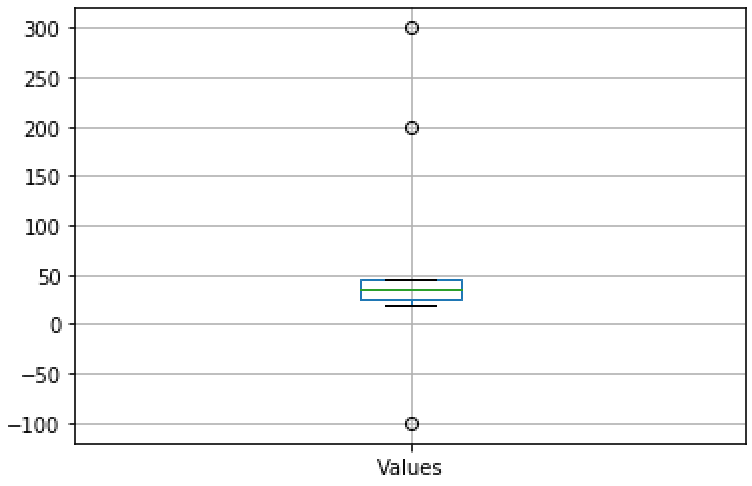
Handling outliers is an essential step in data preprocessing, as outliers can significantly impact the results of your analysis and statistical models. There are other ways to handle these using Scipy as well, but here’s how we do it with pandas.

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 [ ]:
```

NumPy: Welcome to the World of Matrices

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

Check out the Detailed Article in Medium - https://medium.com/@tejag311/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84 (https://medium.com/@tejag311/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84)

Table of Contents

1-Numpy Array Basics

Creating Numpy Arrays

In [2]:
```
# Creating an integer array with explicit dtype, which is not necessary.
int_array = np.array([1, 2, 3], dtype=np.int32)
int_array
```

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

In [3]:
```
# Create an 2D array
original_array = np.array([[1, 2, 3],
                           [4, 5, 6]])
original_array
```

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

```
In [4]: # Creating a 3D array (Tensor) # Dimension: 3 , Shape: (2, 2, 2)
        arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
        arr_3d
```

```
Out[4]: array([[[1, 2],
                [3, 4]],

               [[5, 6],
                [7, 8]]])
```

```
In [5]: # Create an array of 10 equally spaced values from 0 to 1
        linspace_arr = np.linspace(0, 1, 10)
        linspace_arr
```

```
Out[5]: array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
               0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

```
In [6]: # Create an array of 5 values spaced logarithmically from 1 to 100
        logspace_arr = np.logspace(0, 2, 5)
        logspace_arr
```

```
Out[6]: array([  1.        ,   3.16227766,  10.        ,  31.6227766 ,
               100.        ])
```

```
In [7]: # Create an array of values from 0 to 9 with a step size of 2
        arange_arr = np.arange(0, 10, 2)
        arange_arr
```

```
Out[7]: array([0, 2, 4, 6, 8])
```

```
In [8]: # Create a 3x3 array filled with zeros
        zeros_arr = np.zeros((3, 3))
        zeros_arr
```

```
Out[8]: array([[0., 0., 0.],
               [0., 0., 0.],
               [0., 0., 0.]])
```

```
In [9]: # Create a 2x4 array filled with ones
        ones_arr = np.ones((2, 4))
        ones_arr
```

```
Out[9]: array([[1., 1., 1., 1.],
               [1., 1., 1., 1.]])
```

```
In [10]: # Create a new array filled with zeros,
         # matching the shape and data type of the original array
         zeros_array = np.zeros_like(original_array)
         zeros_array
```

```
Out[10]: array([[0, 0, 0],
                [0, 0, 0]])
```

In [11]:
```
# Create a new array filled with ones,
# matching the shape and data type of the original array
ones_array = np.ones_like(original_array)
ones_array
```

Out[11]:
```
array([[1, 1, 1],
       [1, 1, 1]])
```

2-Array Inspection

2.1 Array Dimension and Shapes

In [12]:
```
# Creating a 1D array (Vector)
arr_1d = np.array([1, 2, 3])
# Dimesion: 1 , Shape: (3,), Size: 3
print(f"Dimension: {arr_1d.ndim}, Shape: {arr_1d.shape}, size: {arr_1d.size}")
```

```
Dimension: 1, Shape: (3,), size: 3
```

2.2 Array Indexing and Slicing

In [13]:
```
# Creating a NumPy array
arr = np.array([10, 20, 30, 40, 50])

# Accessing individual elements
first_element = arr[0]  # Access the first element (10)
print(f'First element - {first_element}')

third_element = arr[2]  # Access the third element (30)
print(f'Third element - {third_element}')

# Accessing elements using negative indices
last_element = arr[-1]  # Access the last element (50)
print(f'Last element - {last_element}')
```

```
First element - 10
Third element - 30
Last element - 50
```

In [14]:
```
# Creating a 2D NumPy array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Slicing along rows and columns
sliced_array = arr_2d[0,1]  # Element at row 0, column 1 (value: 2)
sliced_array
```

Out[14]:
```
2
```

```
In [15]: # Slicing the array to create a new array
         sliced_array = arr[1:4]  # Slice from index 1 to 3 (exclusive) [20,30,40]
         sliced_array

Out[15]: array([20, 30, 40])
```

```
In [16]: # Slicing with a step of 2
         sliced_array = arr[0::2]  # Start at index 0, step by 2 [10,30,50]
         sliced_array

Out[16]: array([10, 30, 50])
```

```
In [17]: # Slicing with negative index
         second_to_last = arr[-2::]  # Access the last two elements [40,50]
         second_to_last

Out[17]: array([40, 50])
```

```
In [18]: # Conditional slicing: Select elements greater than 30
         sliced_array = arr[arr > 30]  # Result: [40, 50]
         sliced_array

Out[18]: array([40, 50])
```

```
In [19]: # Creating a 2D NumPy array
         arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

         # Slicing along rows and columns
         sliced_array = arr_2d[1:3, 0:2] # Slice a 2x2 subarray: [[4, 5], [7, 8]]
         sliced_array

Out[19]: array([[4, 5],
                [7, 8]])
```

3-Array Operations

3.1 Element-wise Operations

```
In [20]: # Creating 1D NumPy arrays
         arr1 = np.array([1, 2, 3])
         arr2 = np.array([4, 5, 6])
         scalar = 2

         # Addition
         result_add = arr1 + arr2  # [5, 7, 9]
         result_add

Out[20]: array([5, 7, 9])
```

In [21]:
```
# Multiplication, Similarly subraction and division as well.
result_mul = arr1 * arr2  # [4, 10, 18]
result_mul
```

Out[21]:
```
array([ 4, 10, 18])
```

In [22]:
```
# Creating 2D NumPy arrays
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])

# Multiplication (element-wise, not matrix multiplication)
result_mul = matrix1 * matrix2  # [[5, 12], [21, 32]]
result_mul
```

Out[22]:
```
array([[ 5, 12],
       [21, 32]])
```

In [23]:
```
# Actual Matrix Multiplication using np.dot
matrix_multiplication = np.dot(matrix1,matrix2)
matrix_multiplication
```

Out[23]:
```
array([[19, 22],
       [43, 50]])
```

In [24]:
```
# Broadcasting: Multiply array by a scalar
result = arr1 * scalar  # [2, 4, 6]
result
```

Out[24]:
```
array([2, 4, 6])
```

3.2 Append and Delete

In [25]:
```
# Create an array
original_array = np.array([1, 2, 3])

# Append elements in-place
original_array = np.append(original_array, [4, 5, 6])
original_array
```

Out[25]:
```
array([1, 2, 3, 4, 5, 6])
```

In [26]:
```
# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Remove the item at index 2 (value 3)
new_arr = np.delete(arr, 2)
new_arr
```

Out[26]:
```
array([1, 2, 4, 5])
```

In [27]:
```python
# Create a 2D NumPy array
arr = np.array([[1, 2, 3], [4,5, 6], [7, 8, 9]])

# Remove the second row (index 1)
new_arr = np.delete(arr, 1, axis=0)
new_arr
```

Out[27]:
```
array([[1, 2, 3],
       [7, 8, 9]])
```

3.3 Aggregation Functions and ufuncs

In [28]:
```python
# Creating a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Aggregation functions
mean_value = np.mean(arr)  # Mean: 3.0
print(mean_value)
```

```
3.0
```

In [29]:
```python
median_value = np.median(arr) # Median: 3.0
variance = np.var(arr) # Variance 2.0
standard_deviation = np.std(arr) # std: 1.414

sum_value = np.sum(arr)    # Sum: 15
min_value = np.min(arr)    # Minimum: 1
max_value = np.max(arr)    # Maximum: 5
```

In [30]:
```python
# Universal functions
sqrt_arr = np.sqrt(arr) # Square Root
print(f'square root - {sqrt_arr}')
exp_arr = np.exp(arr) # Exponential
print(f'exponential array - {exp_arr}')
```

```
square root - [1.         1.41421356 1.73205081 2.         2.23606798]
exponential array - [  2.71828183   7.3890561   20.08553692  54.59815003 148.413
1591 ]
```

3.4 Reshaping Arrays

In [31]:
```python
# Creating 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Here as we are passing only 6, it will conver the 2d array to a 1d array
# with 6 elements, you cannot pass anything other than 6,
# as it doesn't match the original array!
arr_1d = arr_2d.reshape(6)
arr_1d
```

Out[31]:
```
array([1, 2, 3, 4, 5, 6])
```

```
In [32]: # Creating 1D array
         arr_1d = np.array([1, 2, 3, 4, 5, 6])

         # converting 1D array to 2D
         arr_2d = arr_1d.reshape(2, 3)
         arr_2d
```

```
Out[32]: array([[1, 2, 3],
                [4, 5, 6]])
```

understanding how to use -1: You can use -1 as a placeholder in any one dimension of the new shape, and NumPy will automatically calculate the size for that dimension.

```
In [33]: from skimage import data
         # Load a sample grayscale image
         image = data.coins()

         # Original shape of the image
         print("Original Image Shape:", image.shape) # Original Image Shape: (303, 384)

         # So, if you want to convert it to 1D, you have to pass 116352 (303*384)
         # Instead, if you don't want to calculate that and let numpy deal with it,
         # IN such cases, you can just pass -1, and it will calculate 116352
         reshaped_image = image.reshape(-1)

         print("Reshapped array:", reshaped_image.shape)
```

```
Original Image Shape: (303, 384)
Reshapped array: (116352,)
```

```
In [34]: # Creating a 1D array with 12 elements
         arr = np.arange(12)
         print("original array shape:",arr.shape)
         # Reshaping into a 2D array with an unknown number of columns (-1)
         reshaped_arr = arr.reshape(4, -1)
         print("Reshaped array shape:",reshaped_arr.shape)
```

```
original array shape: (12,)
Reshaped array shape: (4, 3)
```

- so you can see that the 3 is calculate automatically by just giving -1

4-Working with Numpy Arrays

4.1 Combining Arrays

In [35]:
```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Concatenate along the 0-axis (rows)
combined = np.concatenate((arr1, arr2)) # Result: [1, 2, 3, 4, 5, 6]
combined
```

Out[35]:
```
array([1, 2, 3, 4, 5, 6])
```

In [36]:
```
# Vertical stacking
vertical_stack = np.vstack((arr1, arr2)) # Result: [[1, 2, 3], [4, 5, 6]]
vertical_stack
```

Out[36]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [37]:
```
# Horizontal stacking
horizontal_stack = np.hstack((arr1, arr2)) # Result: [1, 2, 3, 4, 5, 6]
horizontal_stack
```

Out[37]:
```
array([1, 2, 3, 4, 5, 6])
```

4.2 Splitting Arrays

In [38]:
```
arr = np.array([1, 2, 3, 4, 5, 6])

# Split into three equal parts
split_arr = np.split(arr, 3)
split_arr
# Result: [array([1, 2]), array([3, 4]), array([5, 6])].
```

Out[38]:
```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

4.3 Alias vs. View vs. Copy of Arrays

- Alias: An alias refers to multiple variables that all point to the same underlying NumPy array object. They share the same data in memory. Changes in alias array will affect the original array.
- View: The .view() method creates a new array object that looks at the same data as the original array but does not share the same identity. It provides a way to view the data differently or with different data types, but it still operates on the same underlying data.
- Copy: A copy is a completely independent duplicate of a NumPy array. It has its own data in memory, and changes made to the copy will not affect the original array, and vice versa.

In [39]:
```python
original_arr = np.array([1, 2, 3])

# alias of original array
alias_arr = original_arr

# chaing a value in alias array
alias_arr[0]=10

# you can observe that it will also change the original array
original_arr
```

Out[39]:
```
array([10,  2,  3])
```

In [40]:
```python
original_arr = np.array([1, 2, 3])
# Changes to view_arr will affect the original array
view_arr = original_arr.view()

# Modify an element in the view
view_arr[0] = 99

# Check the original array
print(original_arr)
```

```
[99  2  3]
```

In [41]:
```python
original_arr = np.array([1, 2, 3])
# Changes to copy_arr won't affect the original array
copy_arr = original_arr.copy()

copy_arr[0] = 100

# Copy doesn't change the original array
print(original_arr)
```

```
[1 2 3]
```

4.4 Sorting Numpy Arrays

In [42]:
```python
data = np.array([3, 1, 5, 2, 4])
sorted_data = np.sort(data)  # Ascending order
print("Ascending sort", sorted_data)

reverse_sorted_data = np.sort(data)[::-1]  # Descending order
print("Descending sort", reverse_sorted_data)

# Returns Indices that would sort the array.
sorted_indices = np.argsort(data)
print("Sorted Indices", sorted_indices)
```

```
Ascending sort [1 2 3 4 5]
Descending sort [5 4 3 2 1]
Sorted Indices [1 3 0 4 2]
```

5-Numpy for Data Cleaning

5.1 Identify Missing Values

NumPy provides functions to check for missing values in a numeric array, represented as NaN (Not a Number).

In [43]:
```
# Create a NumPy array with missing values
data = np.array([1, 2, np.nan, 4, np.nan, 6])

# Check for missing values
has_missing = np.isnan(data)
print(has_missing)
```

```
[False False  True False  True False]
```

5.2 Removing rows or columns with Missing Values

We can use np.isnan to get a boolean matrix with True for the indices where there is a missing value. And when we pass it to np.any, it will return a 1D array with True for the index where any row item is True. And finally we ~ (not), and pass the boolean to the original Matrix, which will remove the rows with missing values.

In [44]:
```
# Create a 2D array with missing values
data = np.array([[1, 2, 3], [4, np.nan, 6], [7, 8, 9]])

# Remove rows with any missing values
cleaned_data = data[~np.any(np.isnan(data), axis=1)]
print(cleaned_data) # Result: [[1,2,3],[7,8,9]]
```

```
[[1. 2. 3.]
 [7. 8. 9.]]
```

6-Numpy for Statistical Analysis

6.1 Data Transformation

Data transformation involves modifying data to meet specific requirements or assumptions. Numpy doesn’t have these features directly, but we can utilize the existing features to perform these.

In [45]:
```python
# Data Centering
data = np.array([10, 20, 30, 40, 50])
mean = np.mean(data)
centered_data = data - mean
print('Centered data = ',centered_data)

# Standardization
std_dev = np.std(data)
standardized_data = (data - mean) / std_dev
print("standardized data = ",standardized_data)

# Log Transformation
log_transformed_data = np.log(data)
print("log_transformed_data = ",log_transformed_data)
```

```
Centered data =  [-20. -10.   0.  10.  20.]
standardized data =  [-1.41421356 -0.70710678  0.          0.70710678  1.4142135
6]
log_transformed_data =  [2.30258509 2.99573227 3.40119738 3.68887945 3.91202301]
```

6.2 Random Sampling and Generation

Sampling

- Simple Random Sampling: Select a random sample of a specified size from a dataset. When sampling without replacement, each item selected is not returned to the population.
- Bootstrap Sampling: Bootstrap sampling involves sampling with replacement to create multiple datasets. This is often used for estimating statistics' variability. # Simple Random Sampling W

In [46]:
```python
# Simple Random Sampling Without replacement
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
random_samples = np.random.choice(data, size=5, replace=False)
random_samples
```

Out[46]:
```
array([2, 8, 1, 7, 6])
```

In [47]:
```python
# Bootstrap Sampling
num_samples = 1000
bootstrap_samples = np.random.choice(data, size=(num_samples, len(data)), replac
e=True)
bootstrap_samples
```

Out[47]:
```
array([[ 9,  7,  4, ...,  4,  6,  8],
       [ 9,  8,  4, ...,  6,  5,  7],
       [ 7,  3, 10, ...,  8,  7,  6],
       ...,
       [ 6,  3,  8, ...,  5,  3,  6],
       [ 3, 10,  8, ...,  3,  8,  7],
       [ 8,  3, 10, ...,  1,  8,  3]])
```

Generation

In [48]:
```
np.random.randint(0,100)
```

Out[48]:
```
99
```

In [49]:
```
# Generates 5 random values from a standard normal distribution
mean = 0
std_dev = 1
normal_values = np.random.normal(mean, std_dev, 5)
print(normal_values)
```

```
[-0.20126629 -1.24514517 -0.37233003  1.64812603  1.94393548]
```

In [50]:
```
# Simulates 5 sets of 10 trials with a success probability of 0.5
n_trials = 10
probability = 0.5
binomial_values = np.random.binomial(n_trials, probability, 5)
print(binomial_values)
```

```
[4 2 5 9 5]
```

In [51]:
```
# Generates 5 random values following a Poisson distribution with a rate of 2.5
rate = 2.5
poisson_values = np.random.poisson(rate, 5)
print(poisson_values)
```

```
[4 1 3 8 2]
```

In [52]:
```
 # Generates 5 random values following an exponential distribution with a scale
parameter of 0.5
scale_parameter = 0.5
exponential_values = np.random.exponential(scale_parameter, 5)
print(exponential_values)
```

```
[0.20435827 0.43703618 0.631526   0.27823771 0.6482197 ]
```

In [53]:
```
# Generates 5 random values following a log-normal distribution
mean_of_log = 0
std_dev_of_log = 0.5
lognormal_values = np.random.lognormal(mean_of_log, std_dev_of_log, 5)
print(lognormal_values)
```

```
[1.08121668 1.01669187 1.30597122 0.81159366 2.37190169]
```

In [54]:
```
# Simulates 5 sets of 10 multinomial trials with the given probabilities
n_trials = 10
probabilities = [0.2, 0.3, 0.5]  # Probabilities of each outcome
multinomial_values = np.random.multinomial(n_trials, probabilities, 5)
print(multinomial_values)
```

```
[[4 3 3]
 [1 0 9]
 [4 4 2]
 [2 3 5]
 [3 3 4]]
```

7-Numpy for Linear Algebra

7.1 Complex Matrix Operations

We have already seen Creating vectors, matrices, and the amazing matrix operations we can do with numpy. Now, Let’s see even complex matrix operations.

In [55]:
```
A = np.array([[1, 2], [3, 4]])

# Calculate the inverse of A
A_inv = np.linalg.inv(A)
print(A_inv)
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

In [56]:
```
A = np.array([[2, -1], [1, 1]])

# Compute eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(A)
print("eigenvalues:",eigenvalues)
print("eigenvectors:",eigenvectors)
```

```
eigenvalues: [1.5+0.8660254j 1.5-0.8660254j]
eigenvectors: [[0.35355339+0.61237244j 0.35355339-0.61237244j]
 [0.70710678+0.j         0.70710678-0.j        ]]
```

In [57]:
```
A = np.array([[1, 2], [3, 4], [5, 6]])

# Compute the  Singular Value Decomposition (SVD)
U, S, VT = np.linalg.svd(A)
```

7.2 Solve Linear Equations

Yes, You can even solve linear equations with numpy features. Solve systems of linear equations using np.linalg.solve()

In [58]:
```
A = np.array([[2, 3], [4, 5]])
b = np.array([6, 7])

# Solve Ax = b for x
x = np.linalg.solve(A, b)
print(x)
```

```
[-4.5  5. ]
```

8-Advanced Numpy Techniques

8.1 Masked Arrays

Masked arrays in NumPy allow you to work with data where certain elements are invalid or missing. A mask is a Boolean array that indicates which elements should be considered valid and which should be masked (invalid or missing).

Masked arrays enable you to perform operations on valid data while ignoring the masked elements.

```
In [59]: import numpy.ma as ma

         # Temperature dataset with missing values (-999 represents missing values)
         temperatures = np.array([22.5, 23.0, -999, 24.5, -999, 26.0, 27.2, -999, 28.5])

         # Calculate the mean temperature without handling missing values
         mean_temperature = np.mean(temperatures)

         # Print the result = -316.14
         print("Mean Temperature (without handling missing values):", mean_temperature)


         # Create a mask for missing values (-999)
         mask = (temperatures == -999)

         # Create a masked array
         masked_temperatures = ma.masked_array(temperatures, mask=mask)

         # Calculate the mean temperature (excluding missing values)
         mean_temperature = ma.mean(masked_temperatures)

         # Print the result = 25.28
         print("Mean Temperature (excluding missing values):", mean_temperature)
```

```
Mean Temperature (without handling missing values): -316.14444444444445
Mean Temperature (excluding missing values): 25.28333333333333
```

8.2 Structured Arrays

Structured arrays allow you to work with heterogeneous data, similar to a table with named columns. Each element of a structured array can have different data types. Create your datatypes by using np.dtype and add the column name and datatype as a tuple. Then you can pass it to your array.

```
In [60]: # Define data types for fields
         dt = np.dtype([('name', 'S20'), ('age', int), ('salary', float)])

         # Create a structured array
         employees = np.array([('Alice', 30, 50000.0), ('Bob', 25, 60000.0)], dtype=dt)

         # Access the 'name' field of the first employee
         print(employees['name'][0])

         # Access the 'age' field of all employees
         print(employees['age'])
```

```
b'Alice'
[30 25]
```

Conclusion

In this NumPy guide, we've covered essential aspects and advanced techniques for data science and numerical computing. Remember, NumPy is a vast library with endless possibilities. What we have seen is still basic and we can do even a lot more, explore further to unlock its full potential and elevate your data-driven solutions.

In [1]:
```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Table of Contents

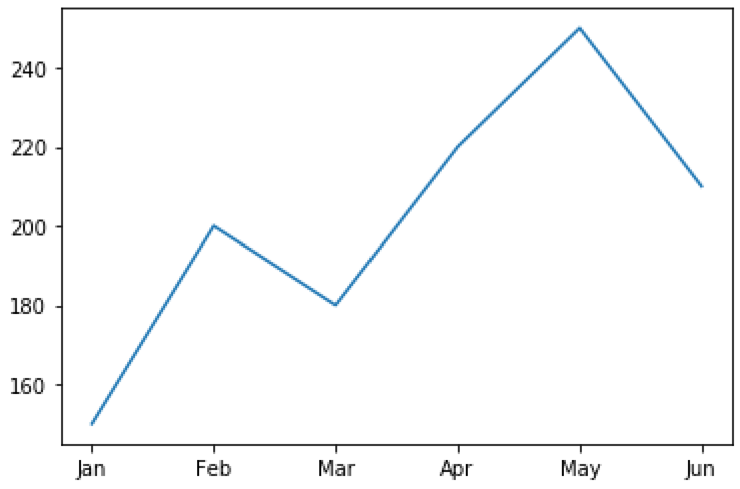
1-Basic Plotting

First things first, we need to import matplotlib.pyplot to access the plotting functions.

1.1 Creating Simple Line Plots

As the name suggests, data points are connected by straight lines which are useful for displaying data that varies continuously over a range, making it easy to identify patterns and trends. Use `plt.plot(x,y)` for a simple line plot, and `plt.show()` to show the plot.

```
In [2]: # Sample data representing monthly website traffic (in thousands)
        months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
        traffic = [150, 200, 180, 220, 250, 210]

        # Create a line plot
        plt.plot(months, traffic)

        plt.show()
```

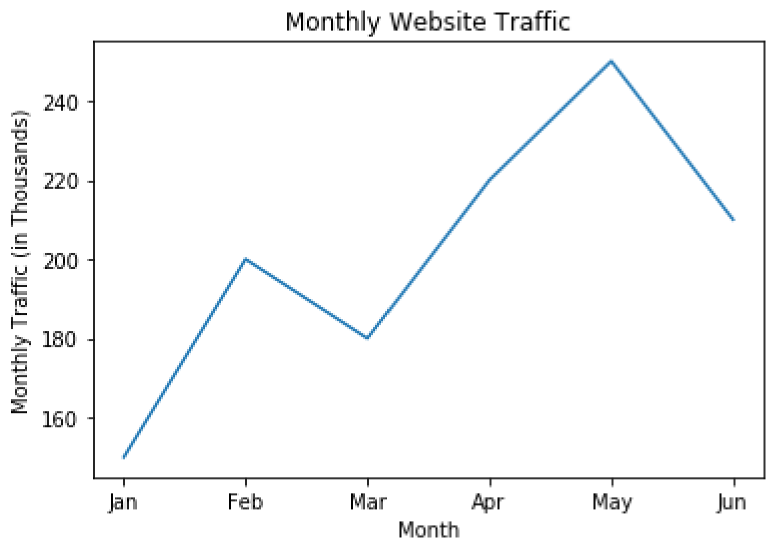


- But as we can see, it doesn’t have any labels or titles as such.

1.2 Adding Labels and Titles

We can convey the information with much clarity by customizing the plots. Matplotlib offers numerous customization options, allowing you to control color, line style, markers, and more.

In [3]:
```python
import matplotlib.pyplot as plt

# Sample data representing monthly website traffic (in thousands)
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
traffic = [150, 200, 180, 220, 250, 210]

# Create a line plot
plt.plot(months, traffic)

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Traffic (in Thousands)')
plt.title('Monthly Website Traffic')

# Display the plot
plt.show()
```



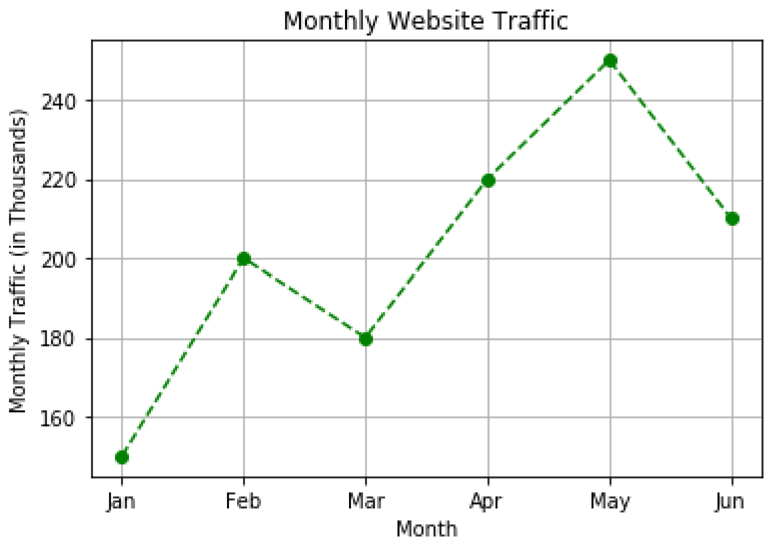
Note: Remember that plt.show() should always be at the end of your plot settings. If you give label commands after the plt.show()then they won't be displayed!

1.3 Customizing the plot

Use the parameter `marker` to mark the points, linetsyle to change the styling of line, and add grid to the plots by using `plt.grid(True)`

Adding styles to the graph

In [4]:
```
# Create a line plot with custom appearance
plt.plot(months, traffic, marker='o', linestyle='--', color='g')

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Traffic (in Thousands)')
plt.title('Monthly Website Traffic')

plt.grid(True)

# Display the plot
plt.show()
```



Changing the Entire Plot Style

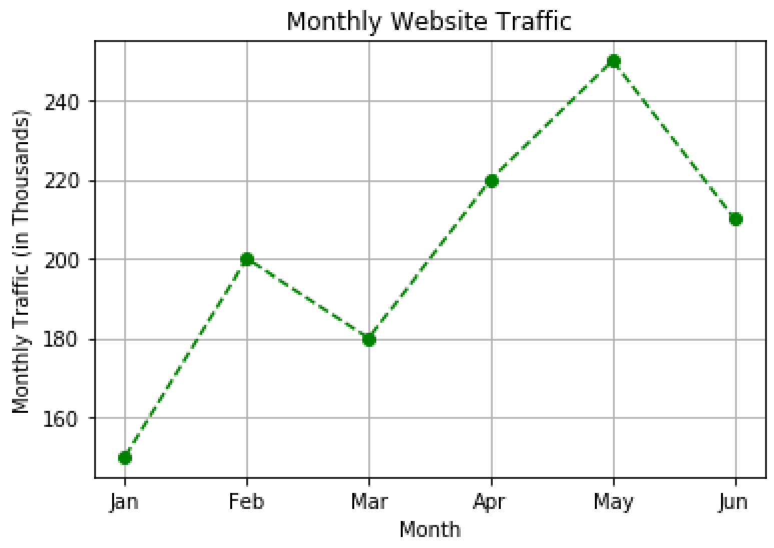
There are various styles available in Matplotlib, to check the available styles, use the command `plt.style.available` . Use plt.style.use(desired_style) to change the style of the entire plot. To use a comic-style plot, you can use plt.xkcd() , this will give a cool plot like below.

```
In [5]: plt.style.available
```

```
Out[5]: ['bmh',
         'classic',
         'dark_background',
         'fast',
         'fivethirtyeight',
         'ggplot',
         'grayscale',
         'seaborn-bright',
         'seaborn-colorblind',
         'seaborn-dark-palette',
         'seaborn-dark',
         'seaborn-darkgrid',
         'seaborn-deep',
         'seaborn-muted',
         'seaborn-notebook',
         'seaborn-paper',
         'seaborn-pastel',
         'seaborn-poster',
         'seaborn-talk',
         'seaborn-ticks',
         'seaborn-white',
         'seaborn-whitegrid',
         'seaborn',
         'Solarize_Light2',
         'tableau-colorblind10',
         '_classic_test']
```

These are all the available styles of the plots.

In [6]:
```python
# Create a line plot with custom appearance
plt.plot(months, traffic, marker='o', linestyle='--', color='g')

# Add labels and a title
plt.xlabel('Month')
plt.ylabel('Monthly Traffic (in Thousands)')
plt.title('Monthly Website Traffic')

plt.style.use('fivethirtyeight')
plt.grid(True)
# Display the plot
plt.show()
```



To bring the plot changes to default, use plt.style.use("default"). And to use a comic style plot, use `plt.xkcd()`

Note: Make sure to use these style commands before the plt.show()

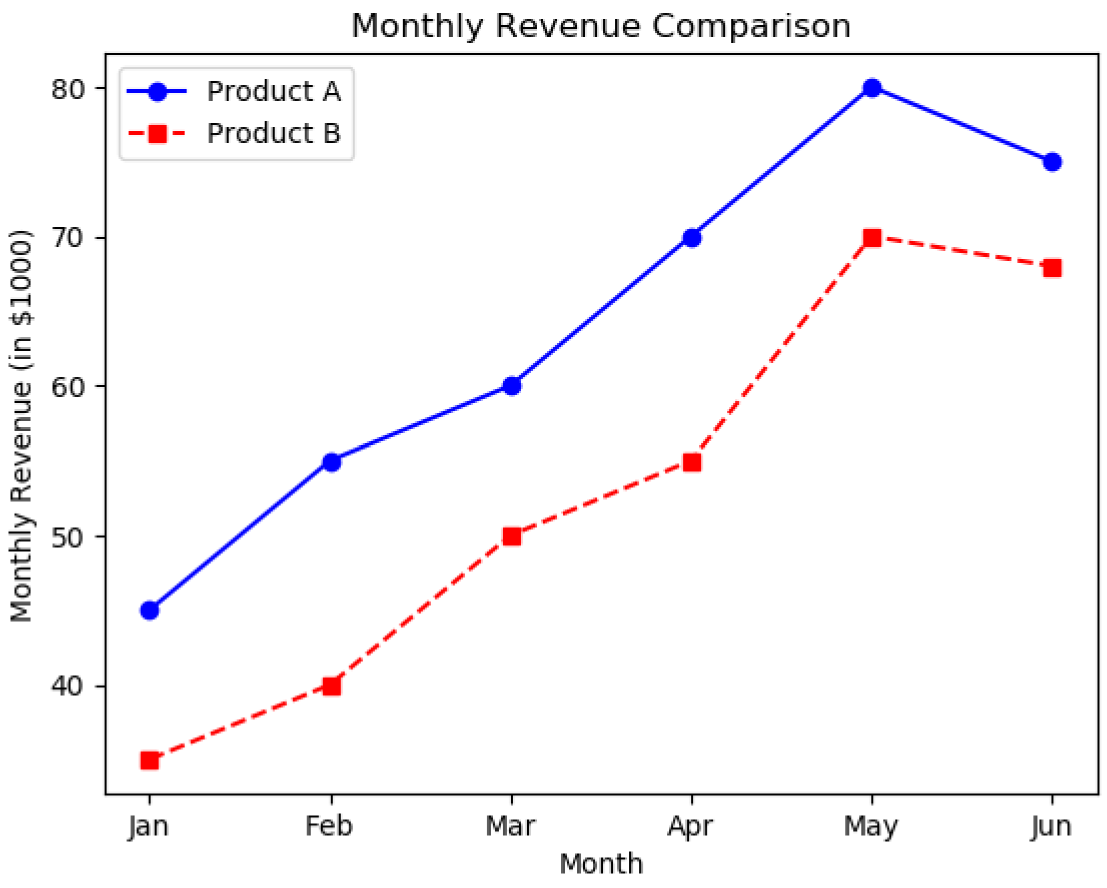
In [7]:
```python
plt.style.use("default")
```

We often have to adjust the plot size, right? And to do that, we need to use plt.figure(figsize=(x_size,y_size)) , Make sure to use this before the .plot command.

1.4 Multiple Line Plot

- In the case of plotting multiple lines in the same graph, You can do so by using the plot command two times for the variables you want. But the issue is to differentiate them properly, for this, we have a parameter called a label , along with that you also need to use plt.legend()

```
In [8]: # Sample data for two products' monthly revenue (in thousands of dollars)
        months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun']
        product_a_revenue = [45, 55, 60, 70, 80, 75]
        product_b_revenue = [35, 40, 50, 55, 70, 68]

        # Create a line plot for Product A with a blue line and circular markers
        plt.plot(months, product_a_revenue, marker='o', linestyle='-', color='blue', lab
        el='Product A')

        # Create a line plot for Product B with a red dashed line and square markers
        plt.plot(months, product_b_revenue, marker='s', linestyle='--', color='red', lab
        el='Product B')

        # Add labels and a title
        plt.xlabel('Month')
        plt.ylabel('Monthly Revenue (in $1000)')
        plt.title('Monthly Revenue Comparison')

        # Display a legend to differentiate between Product A and Product B
        plt.legend()

        # Display the plot
        plt.show()
```



1.5 Saving the plot as an image

In Jupyter Notebook: When working in Jupyter Notebook if you wish to save the plot as an image file, you have to use `plt.savefig(path/to/directory/plot_name.png)` . You can specify the complete file path, and you can specify the desired file name and format(Eg: .jpg, .png, .pdf)

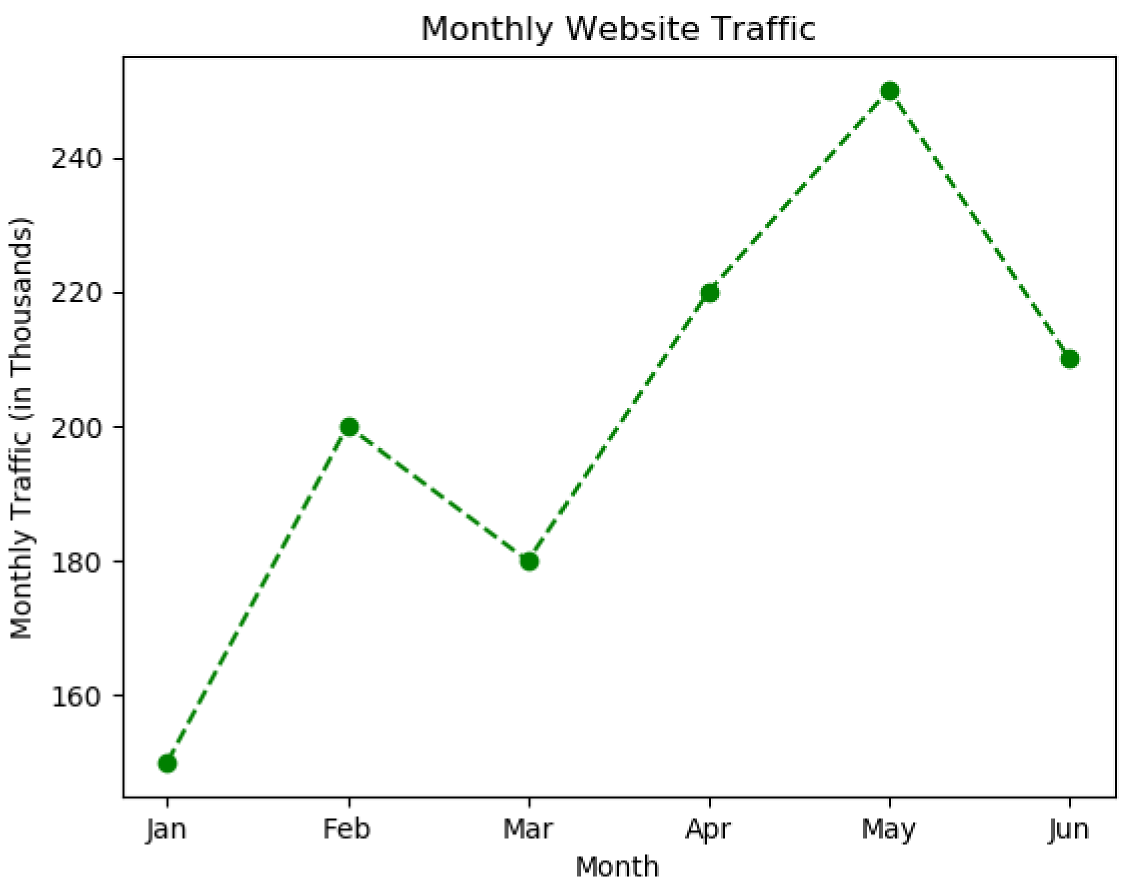
In Google Colab: When working in Google Colab, if you wish to save the plot as an image file, you have to first mount the drive and use plt.savefig().

from google.colab import drive

Mount Google Drive using `drive.mount('/content/drive')` `path = '/content/drive/My Drive/'`

Save the plot as an image file in Colab `plt.savefig(path+'saving_line_plot.png')`

```
In [9]: # Create a line plot with custom appearance
        plt.plot(months, traffic, marker='o', linestyle='--', color='g')

        # Add labels and a title
        plt.xlabel('Month')
        plt.ylabel('Monthly Traffic (in Thousands)')
        plt.title('Monthly Website Traffic')

        # Save the plot as an image file in Colab
        plt.savefig('saving_line_plot.png')  # Specify the complete file path

        # Display the plot
        plt.show()
```



It will be saved now :)

2-Plot Types

We have seen the basic line plot in the previous section, but Matplotlib has a lot more kinds of plots to offer such as Bar Charts, Histograms, Scatter Plots, Pie Charts, Box Plot (Box and whisker Plot), Heatmaps, Displaying images, etc. Now, Let’s understand when to use them along with a few use cases.

2.1 Bar Plots

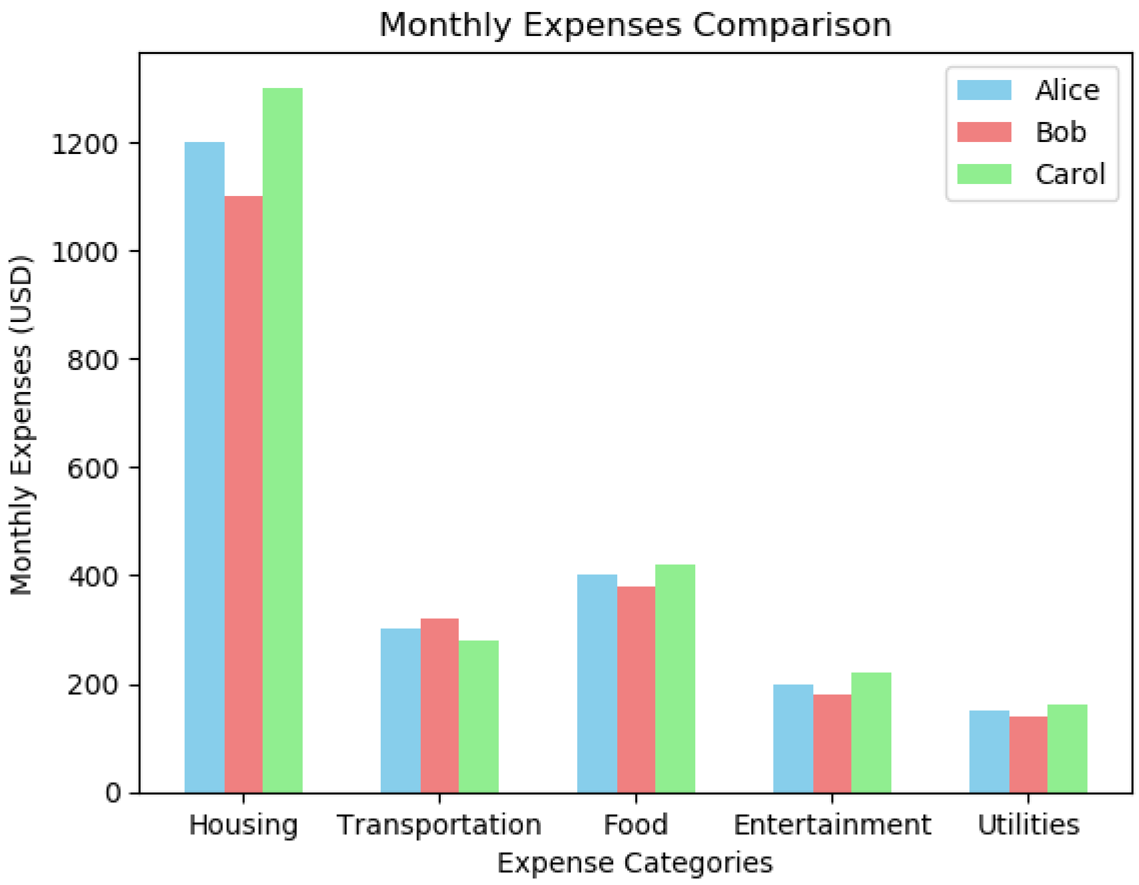
Bar charts represent categorical data with rectangular bars, where the length or height of each bar represents a value. You can use the command plt.bar(x,y) to generate vertical bar charts and plt.barh(x,y) for horizontal bar charts.

Few Use Cases:

1. Comparing sales performance of different products.
2. Showing population distribution by country.

Eg: Multi Bar plot in a single Graph

```
In [10]: import matplotlib.pyplot as plt
import numpy as np

# Expense categories
categories = ['Housing', 'Transportation', 'Food', 'Entertainment', 'Utilities']

# Monthly expenses for Alice, Bob, and Carol
alice_expenses = [1200, 300, 400, 200, 150]
bob_expenses = [1100, 320, 380, 180, 140]
carol_expenses = [1300, 280, 420, 220, 160]

# Create an array for the x-axis positions
x = np.arange(len(categories))

# Width of the bars
bar_width = 0.2

# Create bars for Alice's expenses
plt.bar(x - bar_width, alice_expenses, width=bar_width, label='Alice', color='sk
yblue')

# Create bars for Bob's expenses
plt.bar(x, bob_expenses, width=bar_width, label='Bob', color='lightcoral')

# Create bars for Carol's expenses
plt.bar(x + bar_width, carol_expenses, width=bar_width, label='Carol', color='li
ghtgreen')

# Add labels, a title, and a legend
plt.xlabel('Expense Categories')
plt.ylabel('Monthly Expenses (USD)')
plt.title('Monthly Expenses Comparison')
plt.xticks(x, categories)
plt.legend()

# Display the plot
plt.show()
```



- so, to get these bars, for the first bar we subtracted the x-labels with the bar width, and for the last bar, we added the label with the bar width. We set the width parameter to be equal to the bar width for all.

2.2 Histograms

Histograms are used to visualize the distribution of continuous or numerical data and they help us identify patterns in data. In a histogram plot the data is grouped into "bins," and the height of each bar represents the frequency or count of data points in that bin. It takes the lower and upper limits of the given data and divides it into the no of bins given.

You can use the command plt.hist(x) to generate histograms. Unlike the bar plot, here you don't need the 'y', as it only represents the frequency of one continuous data. The default bins are 10, and they can be changed. You can override the bins range as well with your desired bins range. You can also add the edgecolor for bars.

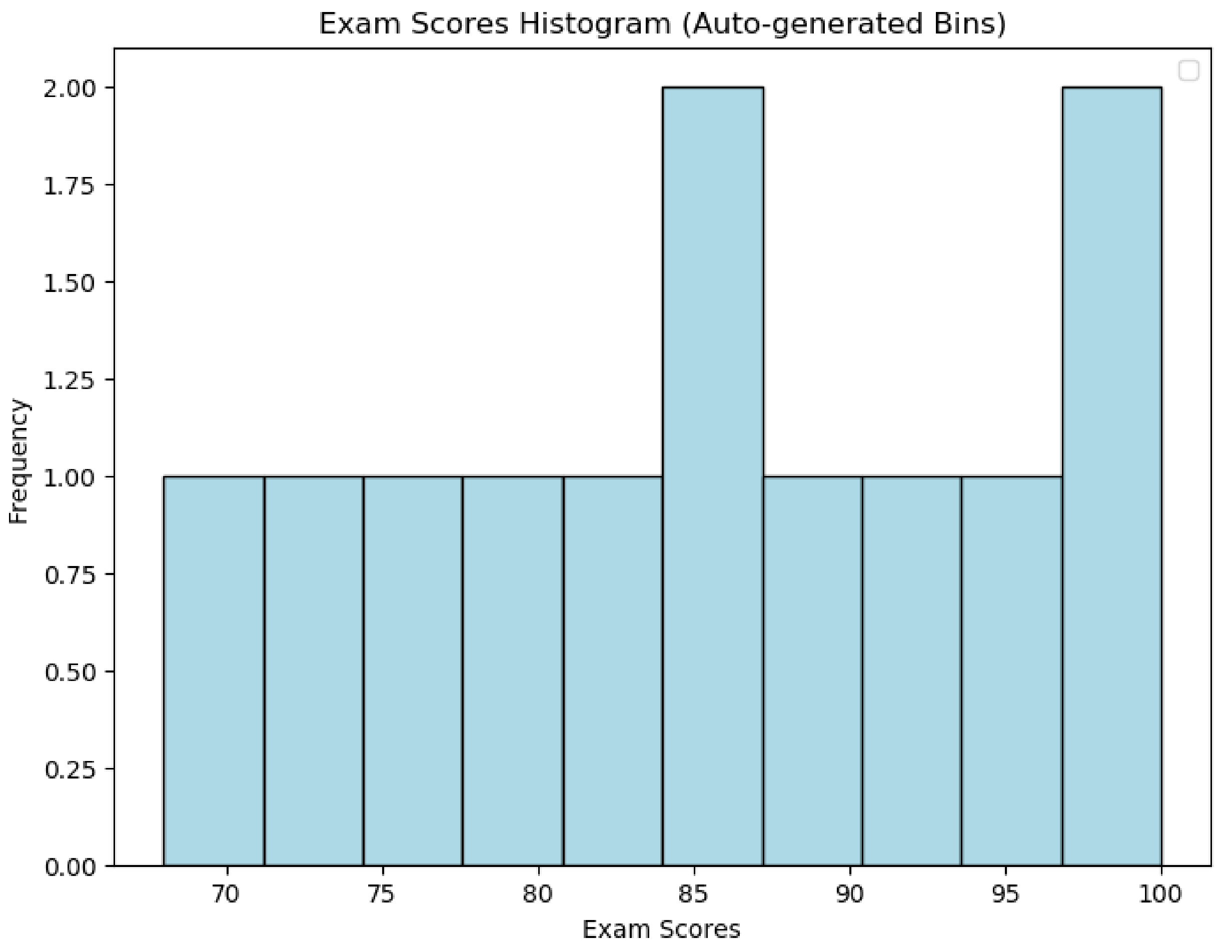
Along with the histogram plot, in the same graph if you want to add any line, say the mean or median, you can do so by calculating the value and passing to to plt.axvline(calculated_mean,label=desired_label) . This can be used with any other plot.

Few Use Cases:

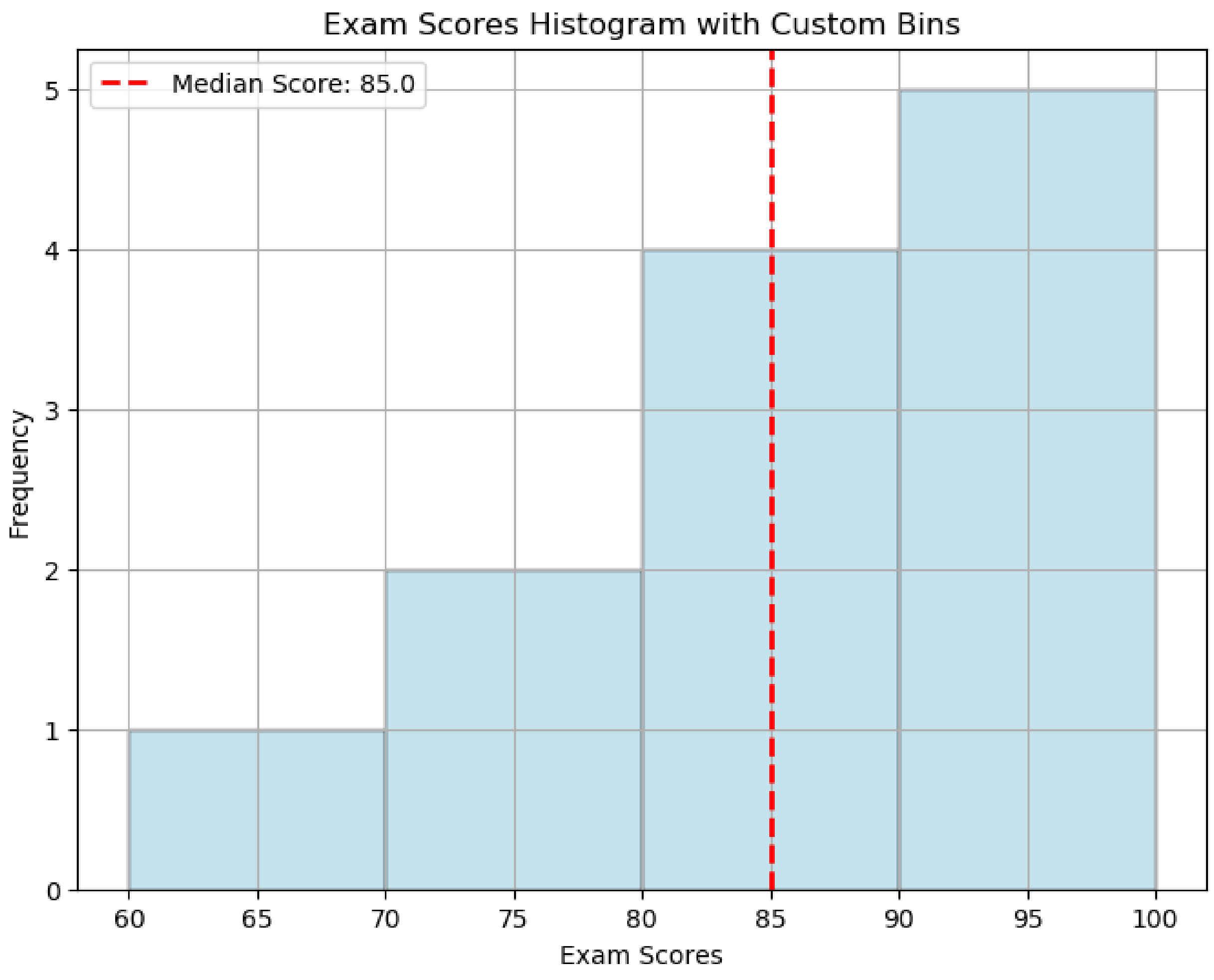
1. Analyzing age distribution in a population.
2. Examining exam score distribution in a classroom.

```
In [11]: # Sample exam scores data
         exam_scores = [68, 72, 75, 80, 82, 84, 86, 90, 92, 95, 98, 100]

         # Create a histogram without specifying bin ranges
         plt.figure(figsize=(8, 6))  # Set the figure size
         plt.hist(exam_scores, color='lightblue', edgecolor='black')

         # Add labels and a title
         plt.xlabel('Exam Scores')
         plt.ylabel('Frequency')
         plt.title('Exam Scores Histogram (Auto-generated Bins)')

         # Add a legend
         plt.legend()
         plt.show()
```

```
No handles with labels found to put in legend.
```



```
In [12]: # Sample exam scores data
         exam_scores = [68, 72, 75, 80, 82, 84, 86, 90, 92, 95, 98, 100]

         # Custom bin ranges
         bin_ranges = [60, 70, 80, 90, 100]

         # Create a histogram with custom bin ranges
         plt.figure(figsize=(8, 6))  # Set the figure size
         plt.hist(exam_scores, bins=bin_ranges, color='lightblue', edgecolor='black', alp
         ha=0.7)

         # Add labels and a title
         plt.xlabel('Exam Scores')
         plt.ylabel('Frequency')
         plt.title('Exam Scores Histogram with Custom Bins')

         # Calculate and add a median line
         median_score = np.median(exam_scores)
         plt.axvline(median_score, color='red', linestyle='dashed', linewidth=2, label=
         f'Median Score: {median_score}')

         # Add a legend
         plt.legend()

         # Display the plot
         plt.grid(True)  # Add a grid for better readability
         plt.show()
```



In this histogram example:

- We have a dataset of exam scores in the exam_scores list.
- We specify custom bin ranges using the bin_ranges list, which defines the edges of the bins.
- The plt.hist() function is used to create the histogram with custom bins. We set the color, edge color, and transparency (alpha) for the bars.
- We calculate the median score using np.median() and add it as a dashed red line using plt.axvline().
- Labels, a title, and a legend are added for better understanding.

2.3 Scatter Plot

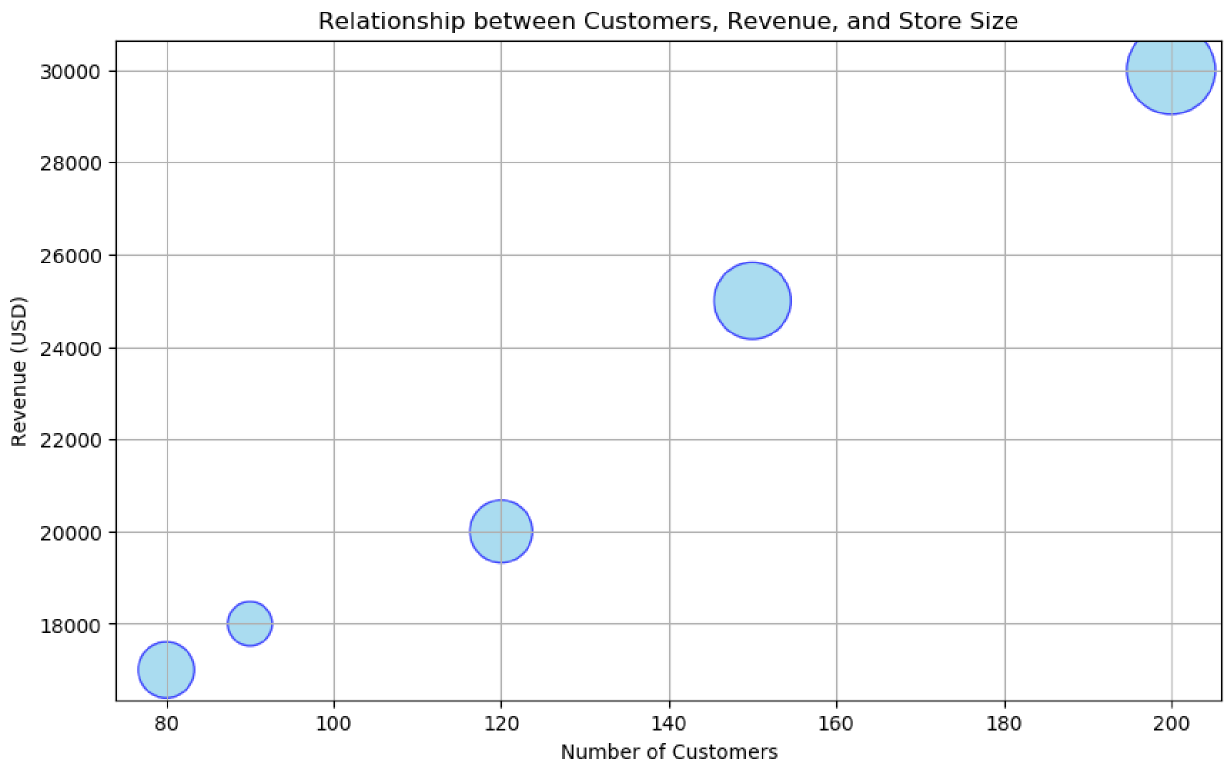
Scatter plots display individual data points as dots on a two-dimensional plane. And they are used to explore relationships or correlations between two numerical variables. In this, each axis represents one variable, and the dots represent data points.

You can use plt.scatter(x,y) to generate scatter plots. To change the size of the points use the parameter s , c for the color, and marker to change the marker instead of a dot. And alpha parameter controls the intensity of the color. For the size, you can even send a different list of sizes for each point.

Few Use Cases:

1. Investigating the relationship between study hours and exam scores.
2. Analyzing the correlation between temperature and ice cream sales.

In [13]:
```
# Sample data for stores
stores = ['Store A', 'Store B', 'Store C', 'Store D', 'Store E']
customers = [120, 90, 150, 80, 200]
revenue = [20000, 18000, 25000, 17000, 30000]
store_size = [10, 5, 15, 8, 20]  # Represents the size of each store in 100sq.ft

# Scale the store sizes for point sizes in the scatter plot
point_sizes = [size * 100 for size in store_size]

# Create a scatter plot with different point sizes
plt.figure(figsize=(10, 6))  # Set the figure size
plt.scatter(customers, revenue, s=point_sizes, c='skyblue', alpha=0.7, edgecolor
s='b')

# Add labels, a title, and a legend
plt.xlabel('Number of Customers')
plt.ylabel('Revenue (USD)')
plt.title('Relationship between Customers, Revenue, and Store Size')

# Display the plot
plt.grid(True)  # Add a grid for better readability
plt.show()
```



2.4 Pie Charts

Pie charts represent parts of a whole as slices of a circular pie. They are suitable for showing the composition of a single categorical variable in terms of percentages. But this won’t look good when there are more than six categories as they get clumsy, in such cases horizontal bar might be preferred.

Use the command plt.pie(x,labels=your_category_names, colors=desired_colors_list) , if you have a desired colors list, you can provide that and you can also change the edge color of the pie chart with the parameter wedgeprops={'edgecolor':your_color} .

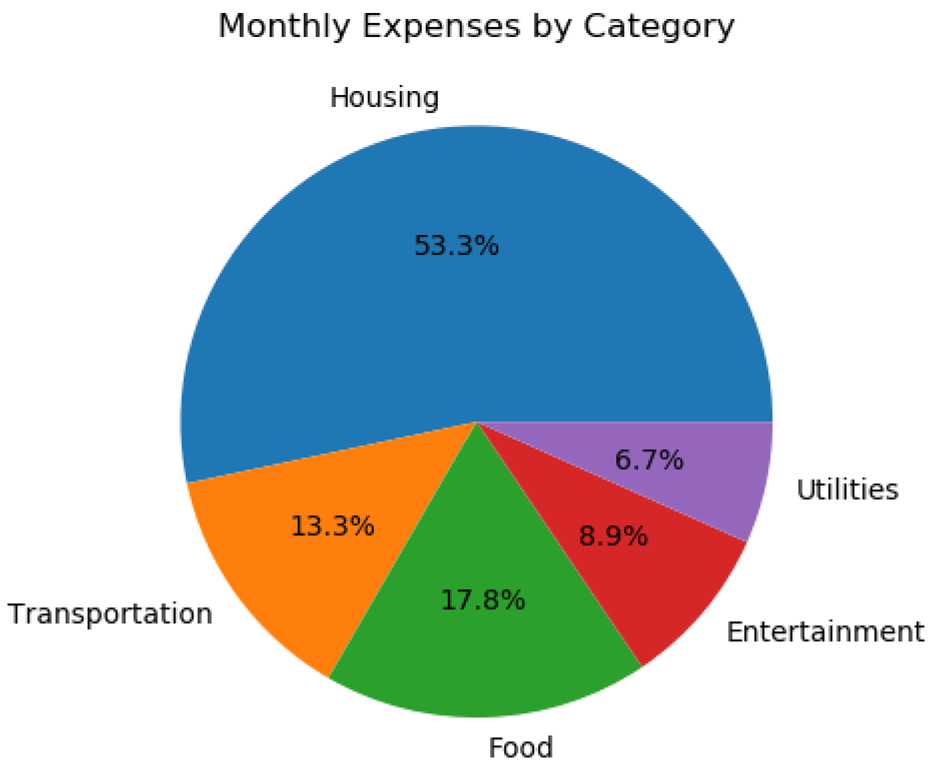
We can also highlight particular segments using explode parameter by passing a tuple where each element is the amount by which it has to explode. And autopct parameter enables you to choose how many values after the decimal are to be shown in the plot.

Few Use Cases:

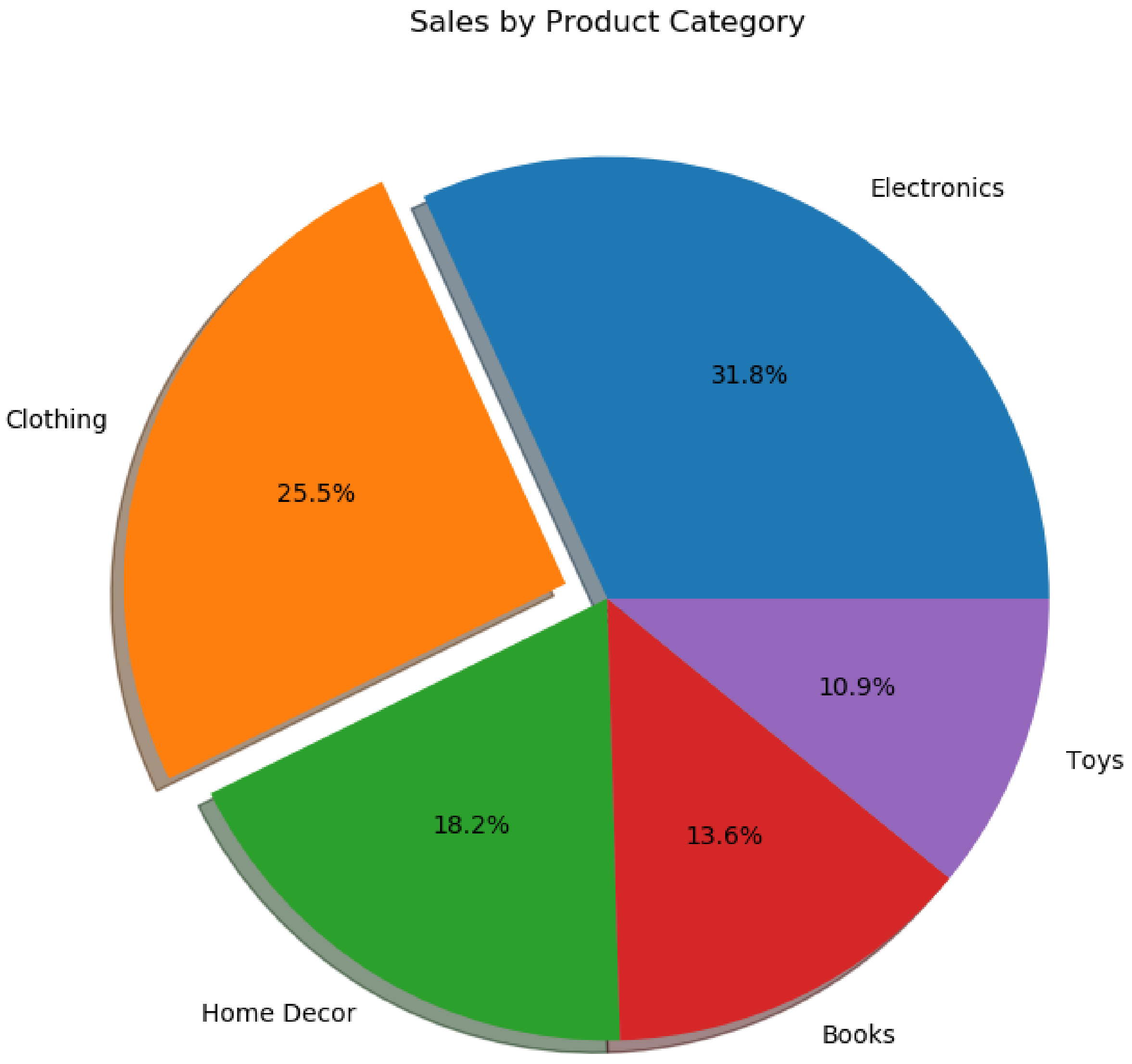
1. Displaying the distribution of a budget by expense categories.
2. Showing the market share of various smartphone brands.

Eg: Exploding a particular segment for better storytelling during presentations.

In [14]:
```python
# Sample data (expenses)
categories = ['Housing', 'Transportation', 'Food', 'Entertainment', 'Utilities']
expenses = [1200, 300, 400, 200, 150]

# Create a pie chart
plt.pie(expenses, labels=categories, autopct='%1.1f%%')

# Add a title
plt.title('Monthly Expenses by Category')

# Display the plot
plt.show()
```



In [15]:
```
# Product categories
categories = ['Electronics', 'Clothing', 'Home Decor', 'Books', 'Toys']

# Sales data for each category
sales = [3500, 2800, 2000, 1500, 1200]

# Explode a specific segment (e.g., 'Clothing')
explode = (0, 0.1, 0, 0, 0)  # The second value (0.1) is the amount by which the
segment 'Clothing' will be exploded

# Create a pie chart with explode and shadow
plt.figure(figsize=(8, 8))  # Set the figure size
plt.pie(sales, labels=categories, explode=explode, shadow=True, autopct='%1.1f%
%')
plt.title('Sales by Product Category')

# Display the plot
plt.show()
```



2.5 Box plot

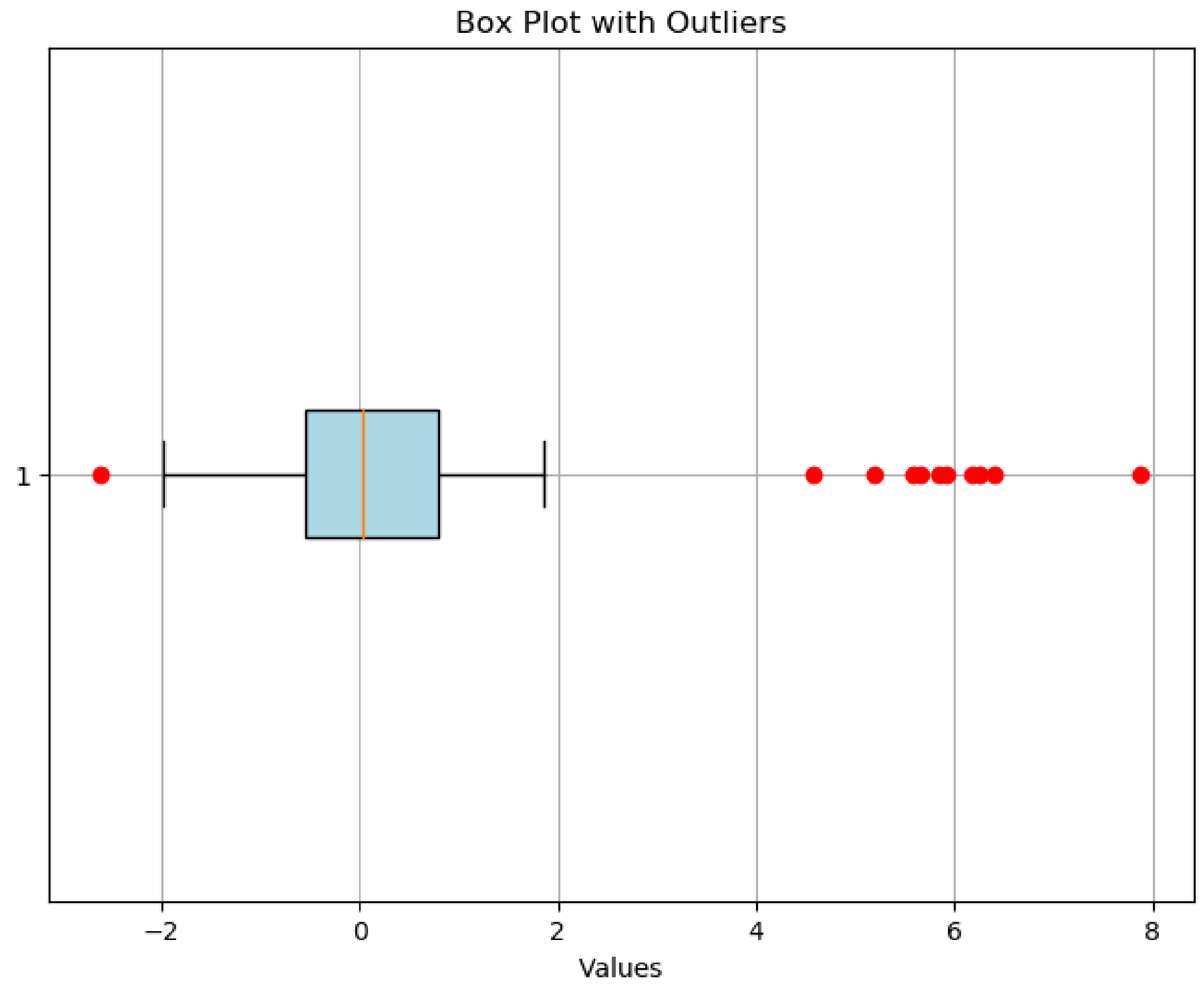
Box plots are the ones that look complicated, right? Simply put they summarize the distribution of numerical data by displaying quartiles, outliers, and potential skewness. They provide insights into data spread, central tendency, and variability. Box plots are especially useful for identifying outliers and comparing distributions.

You can use plt.boxplot(data) to plot the box plot. You can customize the appearance of the box and outliers using boxprops and flierprops , use vert=False to make the box plot horizontal and patch_artist=True to fill the box with color.

Few Use Cases:

1. Analyzing the distribution of salaries in a company.
2. Assessing the variability of housing prices in different neighborhoods.

In [16]:
```
# Generate random data with outliers
np.random.seed(42)
data = np.concatenate([np.random.normal(0, 1, 100), np.random.normal(6, 1, 10)])

# Create a box plot with outliers
plt.figure(figsize=(8, 6))  # Set the figure size
plt.boxplot(data, vert=False, patch_artist=True, boxprops={'facecolor': 'lightbl
ue'}, flierprops={'marker': 'o', 'markerfacecolor': 'red', 'markeredgecolor': 'r
ed'})

# Add labels and a title
plt.xlabel('Values')
plt.title('Box Plot with Outliers')

# Display the plot
plt.grid(True)  # Add a grid for better readability
plt.show()
```

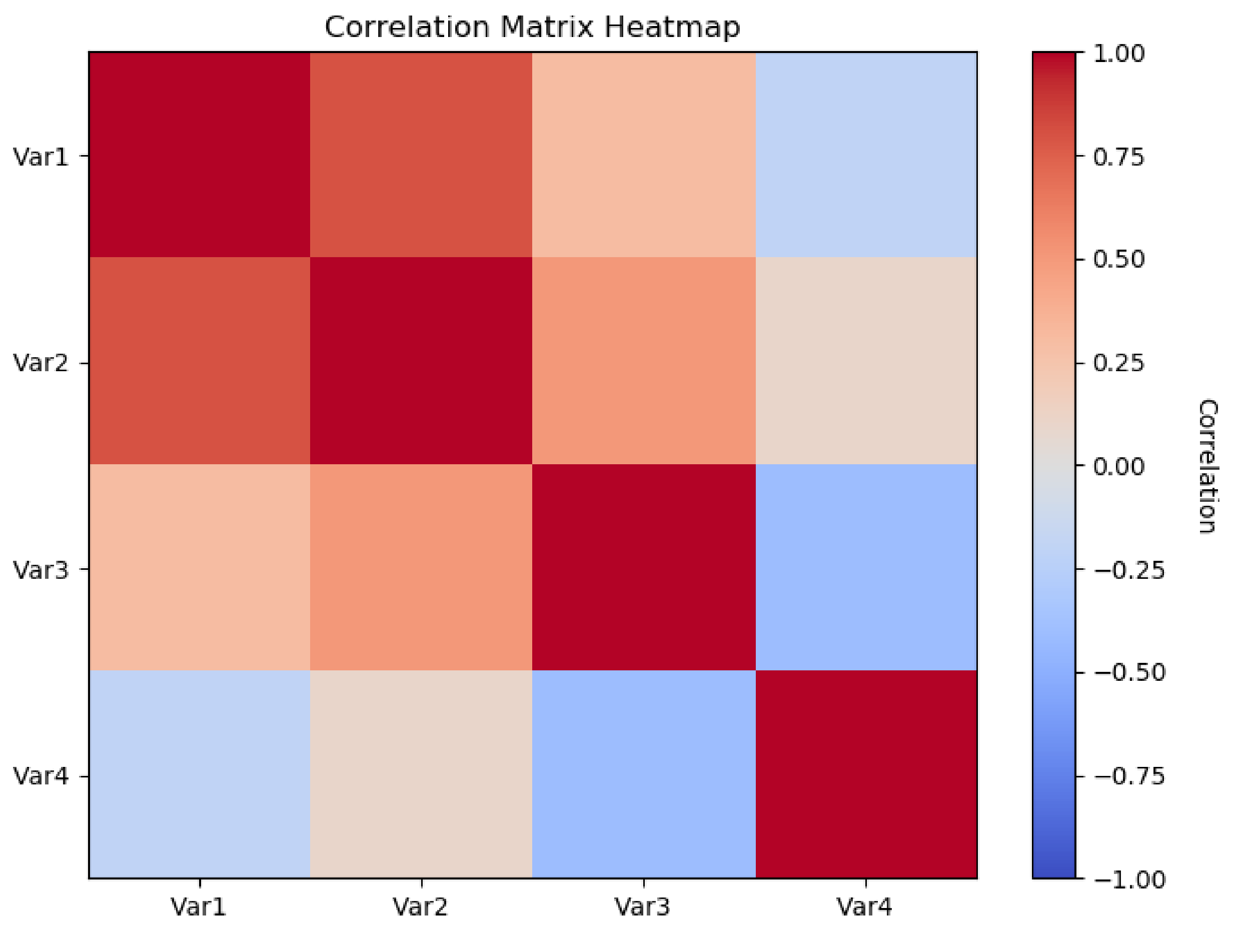


2.6 Displaying Images, Array Visualization

plt.imshow() is a Matplotlib function that is used for displaying 2D image data, visualizing 2D arrays, or showing images in various formats.

Using imshow for heatmap: Heatmap is a visualization for correlation matrix, which will give us a sense of how each variable is correlated with the other variable. Here, we’ll create a heatmap to visualize a correlation matrix, and we’ll use a color map to show this relationship visually. Pass the correlation matrix to imshow to visualize the heatmap.

In [17]:
```
# Create a sample correlation matrix
correlation_matrix = np.array([[1.0, 0.8, 0.3, -0.2],
                               [0.8, 1.0, 0.5, 0.1],
                               [0.3, 0.5, 1.0, -0.4],
                               [-0.2, 0.1, -0.4, 1.0]])

# Create a heatmap for the correlation matrix
plt.figure(figsize=(8, 6))  # Set the figure size
plt.imshow(correlation_matrix, cmap='coolwarm', vmin=-1, vmax=1, aspect='auto',
origin='upper')

# Add a colorbar
cbar = plt.colorbar()
cbar.set_label('Correlation', rotation=270, labelpad=20)

# Add labels and a title
plt.title('Correlation Matrix Heatmap')
plt.xticks(range(len(correlation_matrix)), ['Var1', 'Var2', 'Var3', 'Var4'])
plt.yticks(range(len(correlation_matrix)), ['Var1', 'Var2', 'Var3', 'Var4'])

plt.show()
```

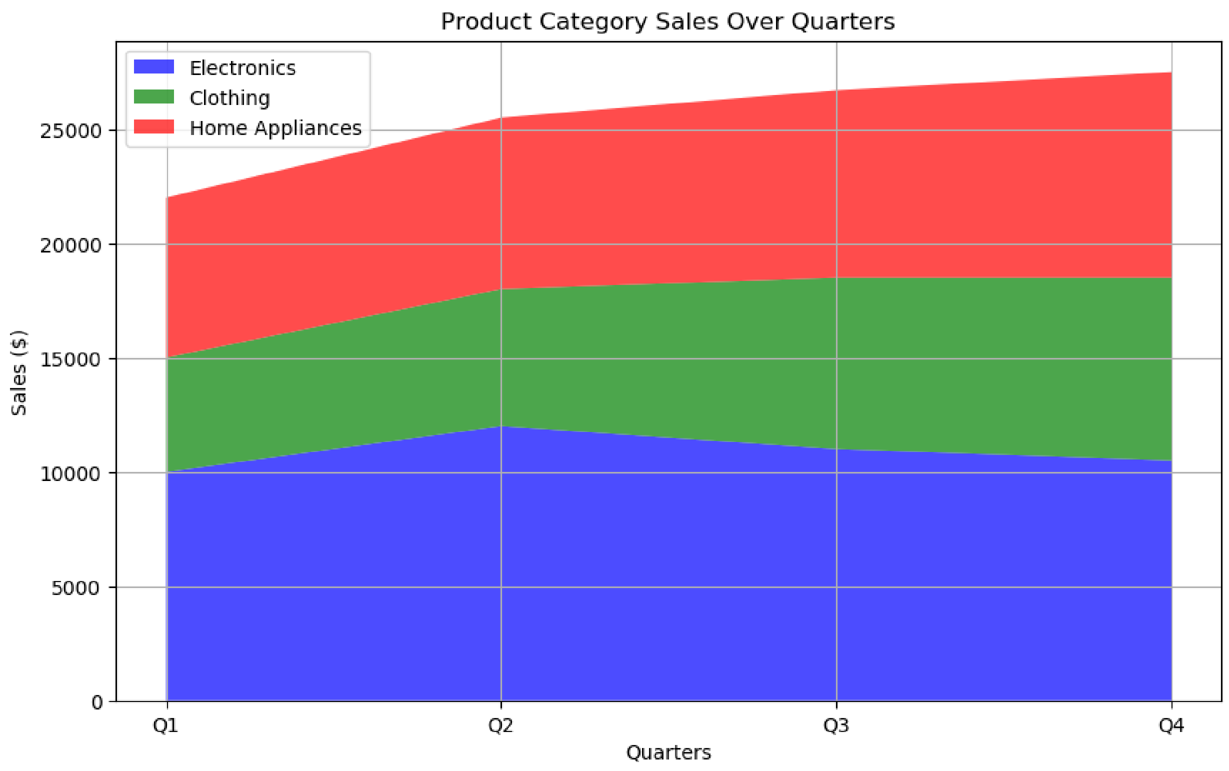


2.7 Stack Plot

Imagine you want to visualize how three product categories (electronics, clothing, and home appliances) contribute to total sales over four quarters (Q1 to Q4). Then you can represent each category’s sales as layers in the plot, and the plot helps us understand their contributions and trends over time. That’s exactly what the stack plot does.

A stack plot, which is also known as a stacked area plot, is a type of data visualization that displays multiple datasets as layers stacked on top of one another, with each layer representing a different category or component of the data. Stack plots are particularly useful for visualizing how individual components contribute to a whole over a continuous time period or categorical domain. Use it as plt.stackplot(x,y1,y2) , as many stacks as you want!

In [18]:
```python
import matplotlib.pyplot as plt

# Sample data for stack plot
quarters = ['Q1', 'Q2', 'Q3', 'Q4']
electronics = [10000, 12000, 11000, 10500]
clothing = [5000, 6000, 7500, 8000]
home_appliances = [7000, 7500, 8200, 9000]

# Create a stack plot
plt.figure(figsize=(10, 6))  # Set the figure size
plt.stackplot(quarters, electronics, clothing, home_appliances, labels=['Electro
nics', 'Clothing', 'Home Appliances'],
              colors=['blue', 'green', 'red'], alpha=0.7)

# Add labels, legend, and title
plt.xlabel('Quarters')
plt.ylabel('Sales ($)')
plt.title('Product Category Sales Over Quarters')
plt.legend(loc='upper left')

# Display the plot
plt.grid(True)

plt.show()
```



In this code:

We have sample data for three product categories: electronics, clothing, and home appliances, with sales data for each quarter (Q1, Q2, Q3, Q4).

We use plt.stackplot() to create the stack plot. Each category's sales are represented as a separate argument, and we provide labels and colors for better visualization.

Labels, legends, and a title are added to enhance the plot's readability and context.

The resulting stack plot visually shows how each product category's sales contribute to the total sales over the four quarters of the year. It's a helpful tool for business analysts to track and understand category performance.

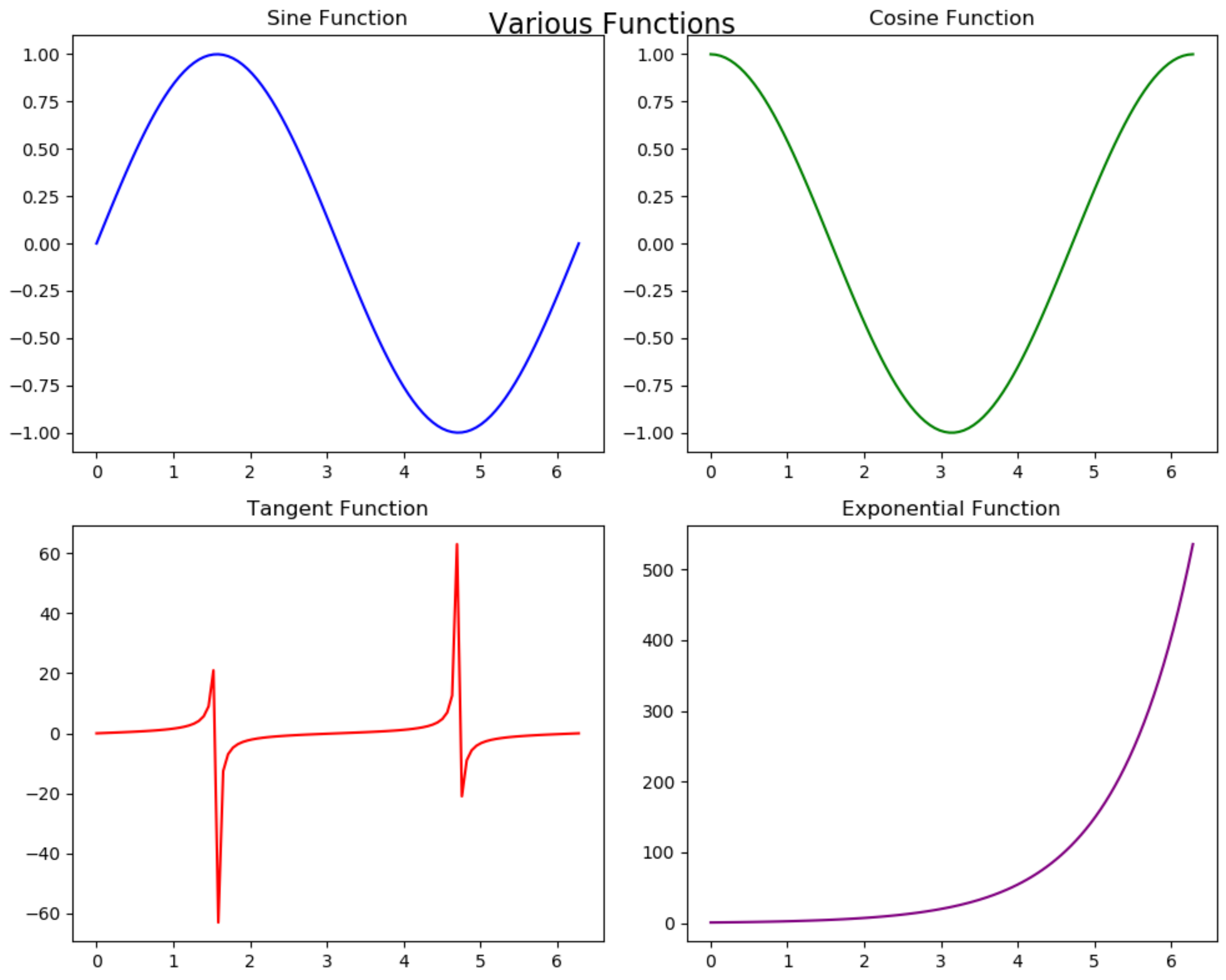
3-Multiple Subplots

Say, You are working with a dataset that has the age of a person, the software they are working on, and their salary. You want to visualize the Python developers' ages and salaries and then compare them with Java developers. By Now, you know you can do that by making two plots one in each cell of the notebook. But then, you have to move back and forth to compare, we better not talk about what if there are 4 things to compare!!

To Ease this issue, we have a feature called subplots, in the same plot there will be different subplots of each. You can create the subplots using plt.subplots(nrows=x,ncols=y) . By default nrows=1, and ncols=1. plt.subplots() returns two things one(fig) is to style the entire plot, and the other(axes) is to make subplots. Plot an each subplot using axes[row, column], where row and column specify the location of the subplot in the grid. You can use the sharex or sharey parameters when you have common axes for the subplots. Let's see a few examples to make it clear.

3.1 Creating Multiple Plots in a single figure

In [19]:
```
# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)
y4 = np.exp(x)

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

# Plot the first subplot (top-left)
axes[0, 0].plot(x, y1, color='blue')
axes[0, 0].set_title('Sine Function')

# Plot the second subplot (top-right)
axes[0, 1].plot(x, y2, color='green')
axes[0, 1].set_title('Cosine Function')

# Plot the third subplot (bottom-left)
axes[1, 0].plot(x, y3, color='red')
axes[1, 0].set_title('Tangent Function')

# Plot the fourth subplot (bottom-right)
axes[1, 1].plot(x, y4, color='purple')
axes[1, 1].set_title('Exponential Function')

# Adjust spacing between subplots
plt.tight_layout()

# Add a common title for all subplots
fig.suptitle('Various Functions', fontsize=16)

# Display the subplots
plt.show()
```

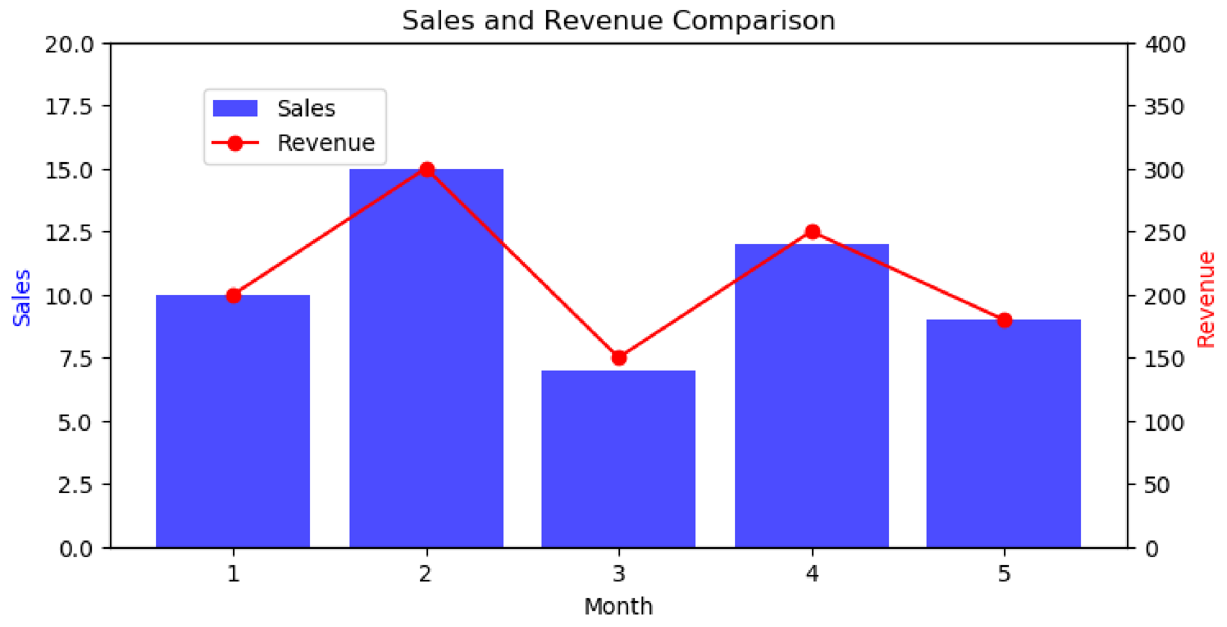


3.2 Combining Different Types of plots

When talking about comparing plots, we will not always wish to have the same axes for both plots, right? There will be cases where we have one common axis and other different!

In such cases, You can combine these different plots within a single figure using the twinx() or twiny() functions to share one axis while having independent y or x-axes. For example, you can combine a line plot of Month vs. Revenue, with a bar plot of Month vs. Sales, to visualize two related datasets with different scales. Here we have a common x-axis but a different y-axis.

In [20]:
```python
# Sample data
x = np.arange(1, 6)
y1 = np.array([10, 15, 7, 12, 9])
y2 = np.array([200, 300, 150, 250, 180])

# Create a bar plot
fig, ax1 = plt.subplots(figsize=(8, 4))
ax1.bar(x, y1, color='b', alpha=0.7, label='Sales')
ax1.set_xlabel('Month')
ax1.set_ylabel('Sales', color='b')
ax1.set_ylim(0, 20)  # Set y-axis limits for the left y-axis

# Create a line plot sharing the x-axis
ax2 = ax1.twinx()
ax2.plot(x, y2, color='r', marker='o', label='Revenue')
ax2.set_ylabel('Revenue', color='r')
ax2.set_ylim(0, 400)  # Set y-axis limits for the right y-axis

# Add a legend
fig.legend(loc='upper left', bbox_to_anchor=(0.15, 0.85))

# Add a title
plt.title('Sales and Revenue Comparison')

# Show the plot
plt.show()
```



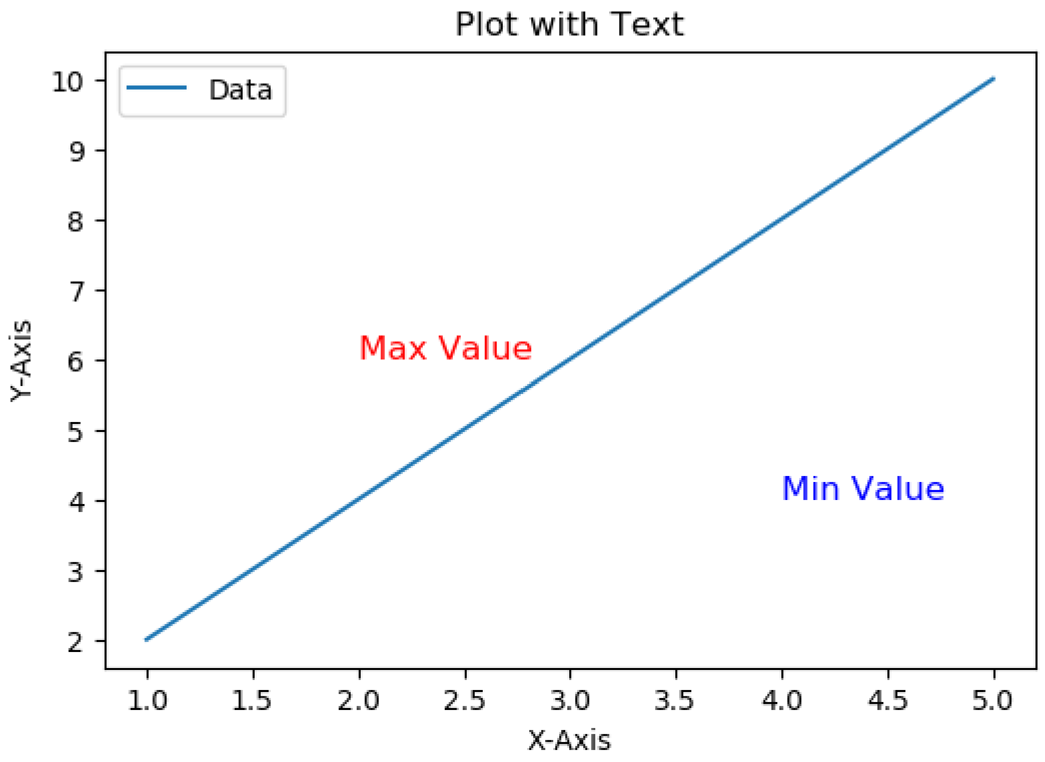
4-Advanced Features

4.1 Annoate and Text for the plots

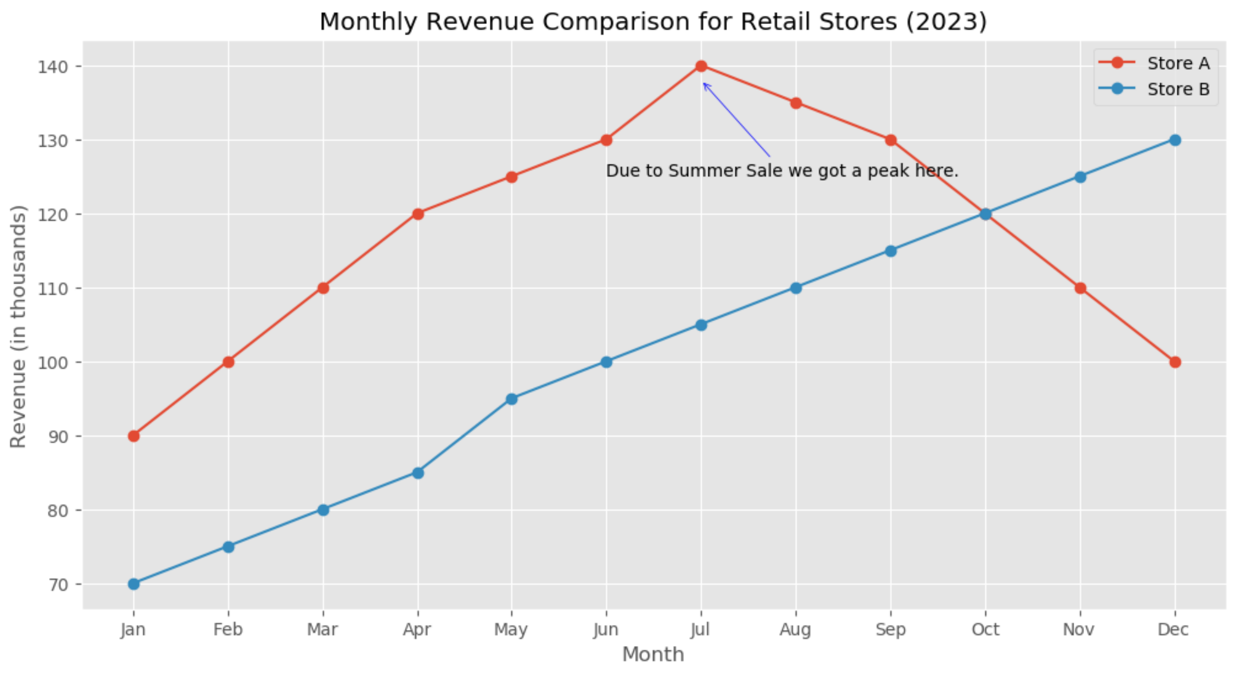
In Matplotlib, you can incorporate annotations and text using various methods. This is very useful during presentations, it is a powerful technique to enhance the communication of insights and highlight key points in your plots.

- **Adding text with text functions:** The plt.text(x_pos,y_pos,desired_text,fontsize=desired_size) function allows you to add custom text at specified coordinates on the plot.
- **Annotating with annotate() Function:** The plt.annotate(desired_text,xy=arrow_pos,xytext=text_post) function allows you to add text with an associated arrow or marker pointing to a specific location on the plot.
- **Labeling Data Points:** You can label individual data points in a scatter plot using text() or annotate()

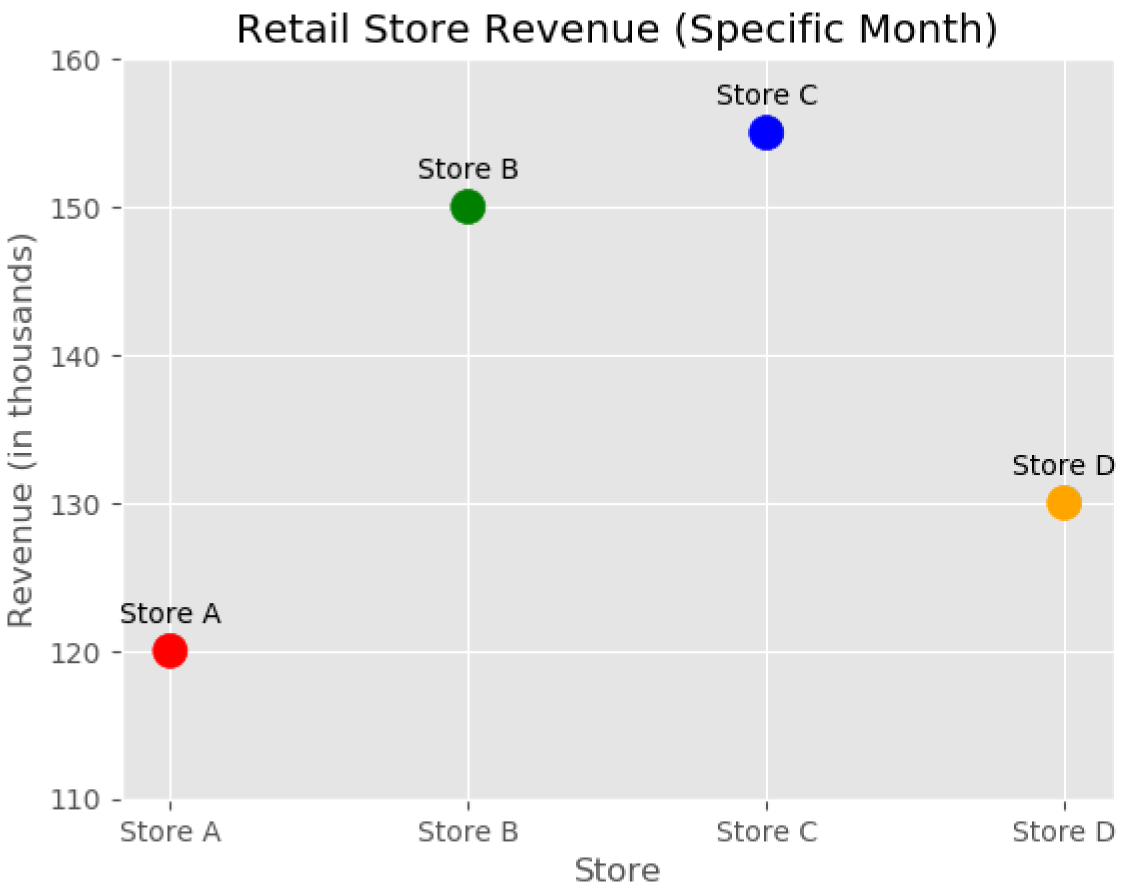
```
In [21]: # Create a simple plot
         plt.figure(figsize=(6, 4))
         plt.plot([1, 2, 3, 4, 5], [2, 4, 6, 8, 10], label='Data')

         # Add text to the plot
         plt.text(2, 6, 'Max Value', fontsize=12, color='red')
         plt.text(4, 4, 'Min Value', fontsize=12, color='blue')

         # Customize the text
         plt.xlabel('X-Axis')
         plt.ylabel('Y-Axis')
         plt.title('Plot with Text')
         plt.legend()

         # Show the plot
         plt.show()
```



```
In [22]: # Sample data for retail shop revenue
         months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
         'Nov', 'Dec']
         store_locations = ['Store A', 'Store B']
         revenue_data = np.array([[90, 100, 110, 120, 125, 130, 140, 135, 130, 120, 110,
         100],
                                  [70, 75, 80, 85, 95, 100, 105, 110, 115, 120, 125, 13
         0]])

         # Create the plot
         plt.figure(figsize=(12, 6))

         plt.style.use('ggplot')

         # Plot monthly revenue for each store location
         for i in range(len(store_locations)):
             plt.plot(months, revenue_data[i], marker='o', label=store_locations[i])

         # Highlight special promotions
         plt.annotate('Due to Summer Sale we got a peak here.',  xy=('Jul', 138), xytext=
         ('Jun', 125),
                      arrowprops=dict(arrowstyle='->', color='blue'))

         # Add title and labels
         plt.title('Monthly Revenue Comparison for Retail Stores (2023)')
         plt.xlabel('Month')
         plt.ylabel('Revenue (in thousands)')
         plt.grid(True)
         plt.legend()

         plt.show()
```



```
In [23]: # Sample data for retail shop revenue in a specific month
         store_names = ['Store A', 'Store B', 'Store C', 'Store D']
         revenue_data = np.array([120, 150, 155, 130])  # Revenue in thousands
         colors = ['red', 'green', 'blue', 'orange']

         # Plot store revenue as points
         plt.scatter(store_names, revenue_data, s=150, c=colors, marker='o', label='Reven
         ue')

         # Label each store
         for i, store in enumerate(store_names):
             plt.annotate(store, (store, revenue_data[i]), textcoords="offset points", xy
         text=(0, 10), ha='center')
             # plt.text(store, revenue_data[i]+1,store)

         # Add title and labels
         plt.title('Retail Store Revenue (Specific Month)')
         plt.xlabel('Store')
         plt.ylabel('Revenue (in thousands)')
         plt.ylim(110,160)
         plt.grid(True)
         plt.show()
```



In the above example of the plot with labels, To add labels, we need to iterate through the names and use the annotate method for each point of the name. In this case, you don't need to use xy and arrowprops parameters. But you do need to use textcoords='offset points' , this ensures that the positions specified for the label (in this case, xytext) are interpreted in a coordinate system where the origin (0, 0).

In [24]:
```
plt.style.use('default')
```

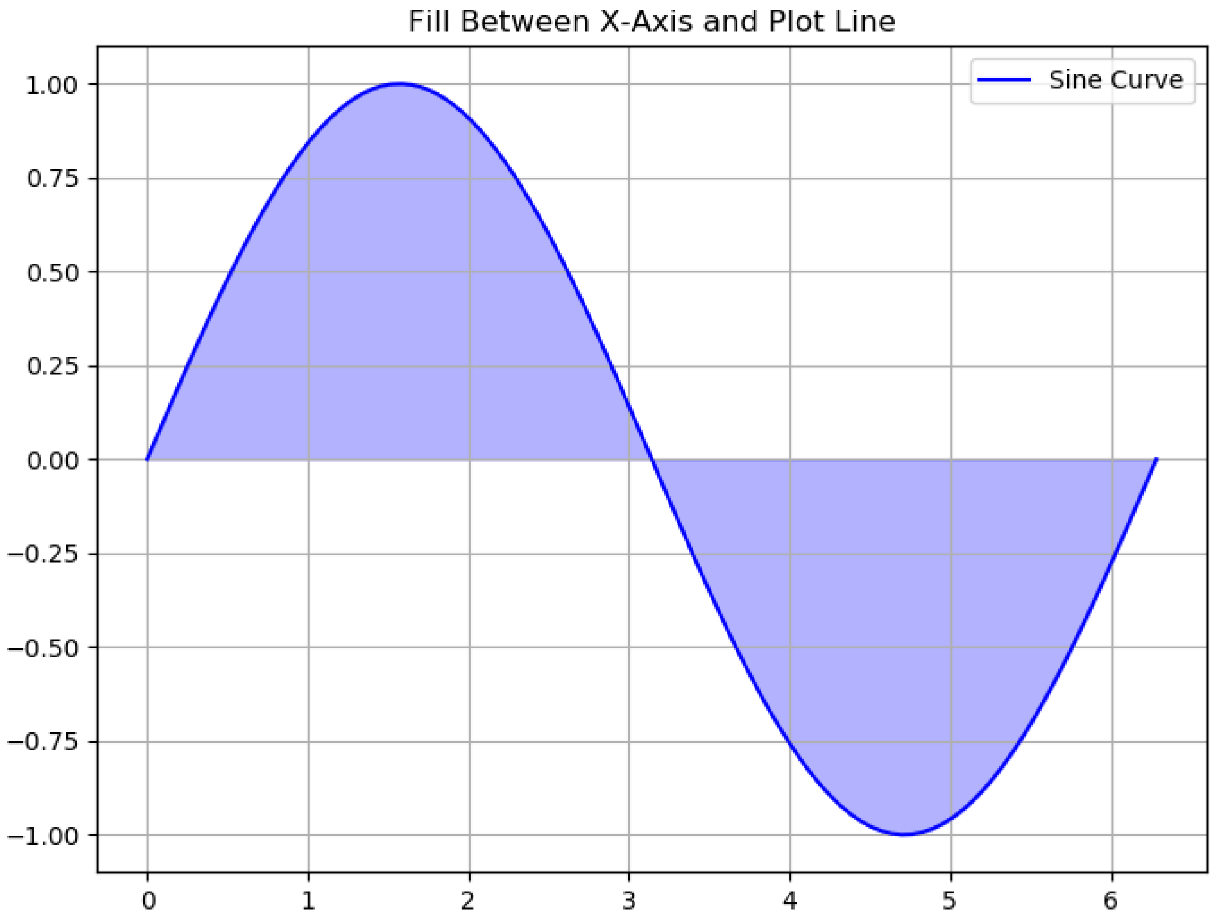
4.2 Fill the Area Between the plots

Sometimes we need to highlight the regions between two line plots, which can help viewers understand where one curve surpasses another. And this can be achieved through fill_between method in Matplotlib. The intensity of the fill color can be controlled through alpha parameter.

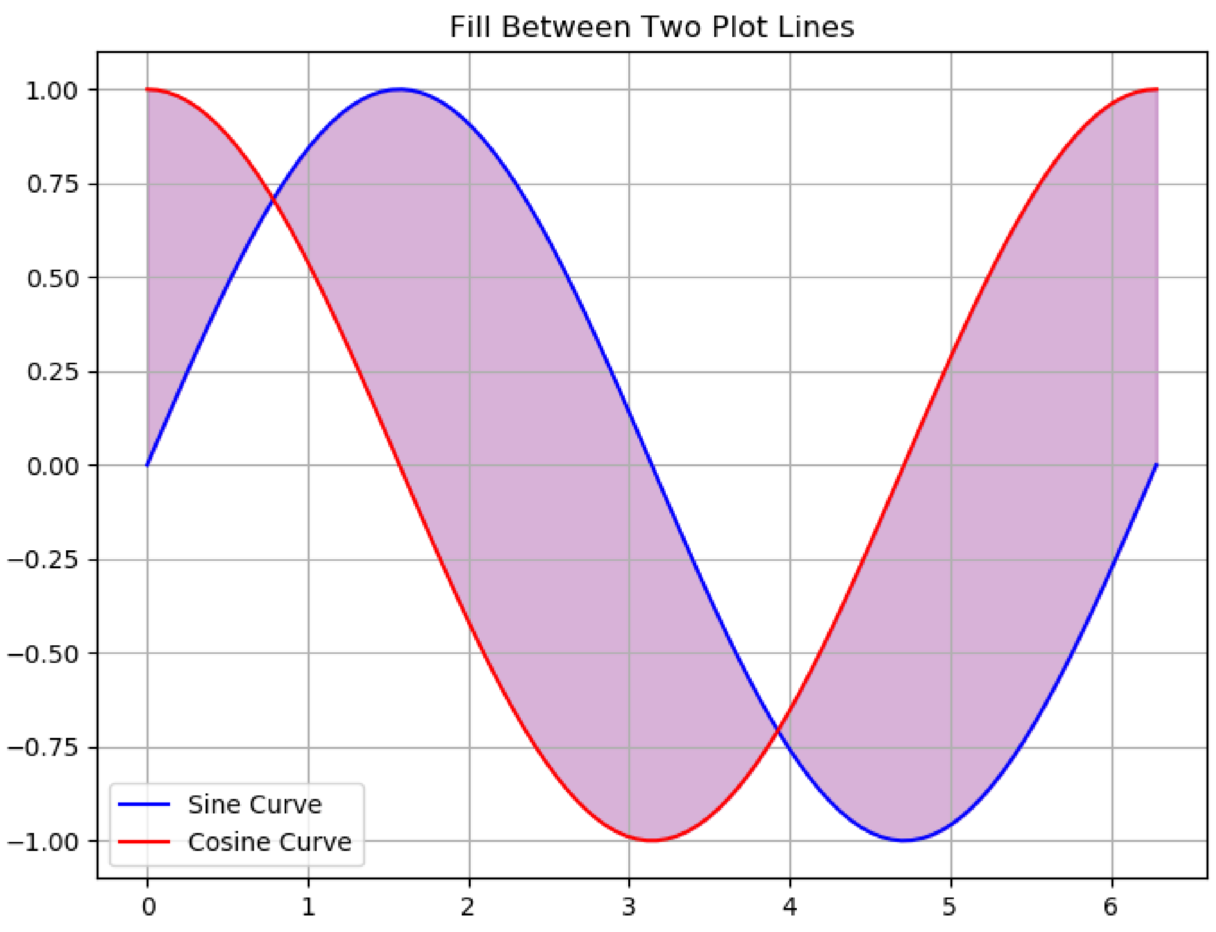
- To Fill all the Region between the x-axis and the plot line, you can use the command plt.fill_between(x,y)
- To Fill the intersection between two plot lines, you can use the command plt.fill_between(x,y1,y2)
- To Fill the intersection between two plot lines only if they satisfy a specified condition, you can use the command plt.fill_between(x,y1,y2,where=condition)
- To Fill more than one region of the plot with different conditions and different colors.

Here are a Few Examples of the above cases.

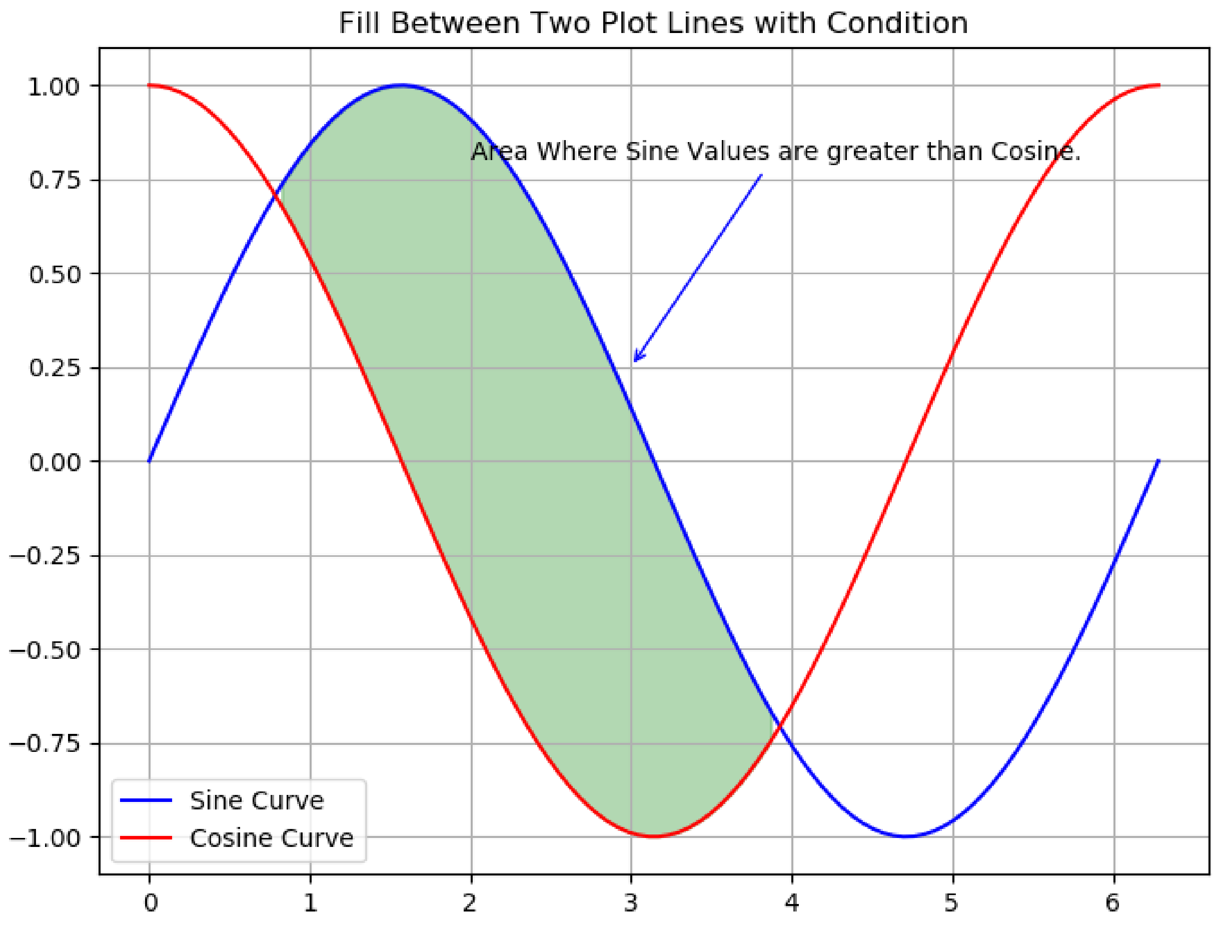
```
In [25]: # Sample data
         x = np.linspace(0, 2 * np.pi, 100)
         y = np.sin(x)

         # Create the plot
         plt.figure(figsize=(8, 6))

         # Plot the curve
         plt.plot(x, y, label='Sine Curve', color='blue')

         # Fill the region between the curve and the x-axis
         plt.fill_between(x, 0, y, alpha=0.3, color='blue')

         # Add title and labels
         plt.title('Fill Between X-Axis and Plot Line')
         plt.grid(True)
         plt.legend()
         plt.show()
```



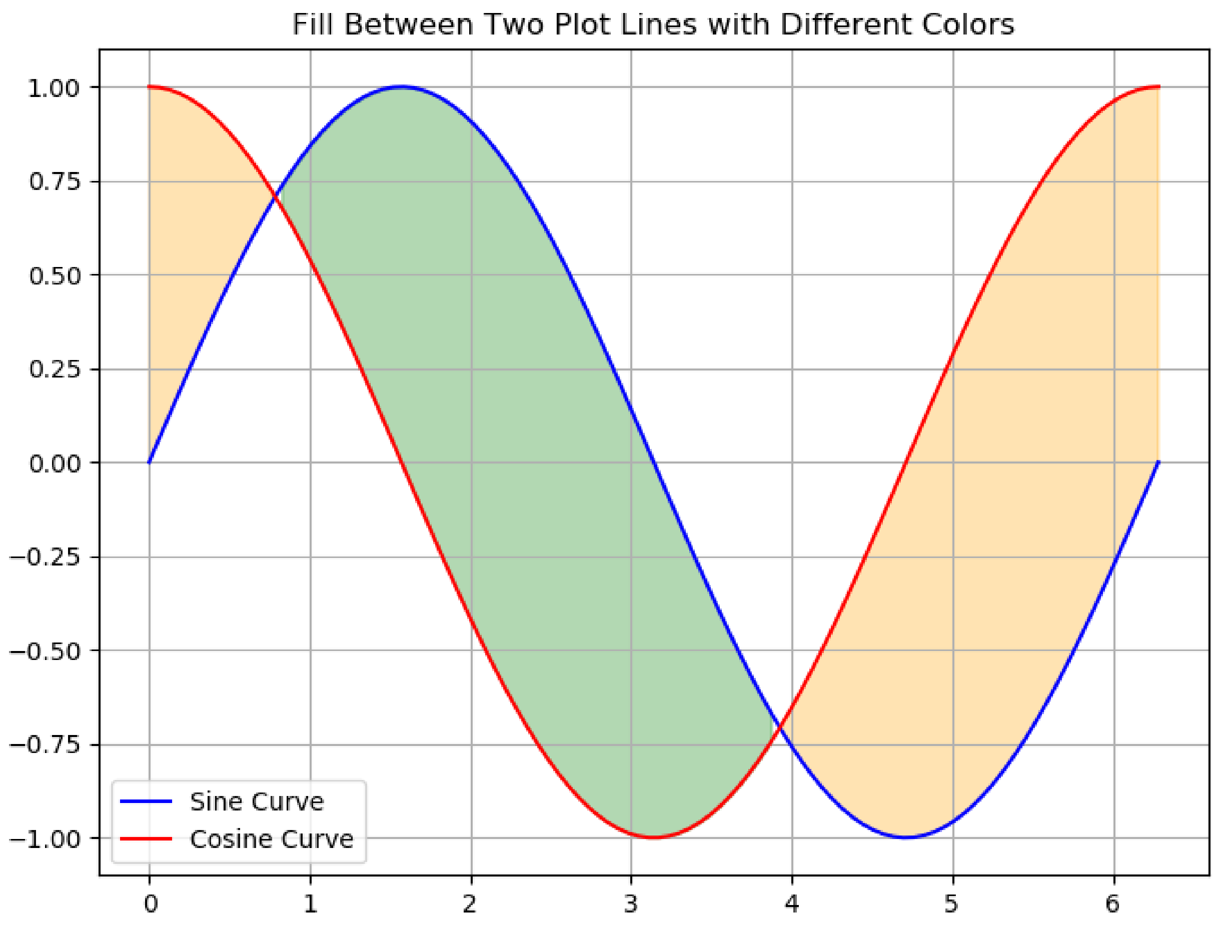
In [26]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the two curves
plt.plot(x, y1, label='Sine Curve', color='blue')
plt.plot(x, y2, label='Cosine Curve', color='red')

# Fill the region between the two curves
plt.fill_between(x, y1, y2, alpha=0.3, color='purple')

# Add title and labels
plt.title('Fill Between Two Plot Lines')
plt.grid(True)
plt.legend()
plt.show()
```



In [27]:
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the two curves
plt.plot(x, y1, label='Sine Curve', color='blue')
plt.plot(x, y2, label='Cosine Curve', color='red')

# Fill the region between the two curves where y1 > y2
plt.fill_between(x, y1, y2, where=(y1 > y2), alpha=0.3, color='green')

# Highlight special promotions
plt.annotate('Area Where Sine Values are greater than Cosine.', xy=(3, 0.25), xy
text=(2, 0.80),
             arrowprops=dict(arrowstyle='->', color='blue'))

# Add title and labels
plt.title('Fill Between Two Plot Lines with Condition')
plt.grid(True)
plt.legend()
plt.show()
```



In [28]:
```
import matplotlib.pyplot as plt
import numpy as np

# Sample data
x = np.linspace(0, 2 * np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create the plot
plt.figure(figsize=(8, 6))

# Plot the two curves
plt.plot(x, y1, label='Sine Curve', color='blue')
plt.plot(x, y2, label='Cosine Curve', color='red')

# Fill multiple regions with different colors
plt.fill_between(x, y1, y2, where=(y1 > y2), alpha=0.3, color='green')
plt.fill_between(x, y1, y2, where=(y1 <= y2), alpha=0.3, color='orange')

# Add title and labels
plt.title('Fill Between Two Plot Lines with Different Colors')
plt.grid(True)
plt.legend()
plt.show()
```

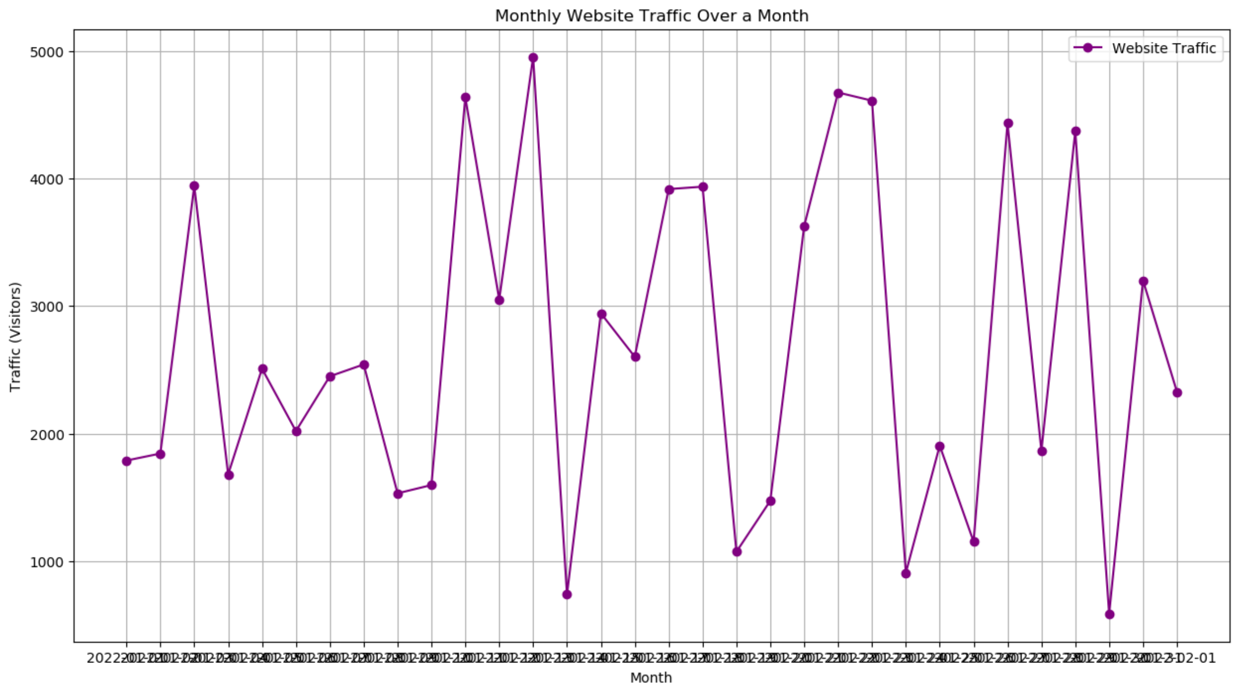


4.3 Plotting Time Series Data

We all know that Time series data is very common in many fields such as finance, climate science, business analytics, etc. And also the data will be very huge in these cases, we can't make the most out of data by just doing some aggregations! Matplotlib offers us ways to easily interpret the time-series data.

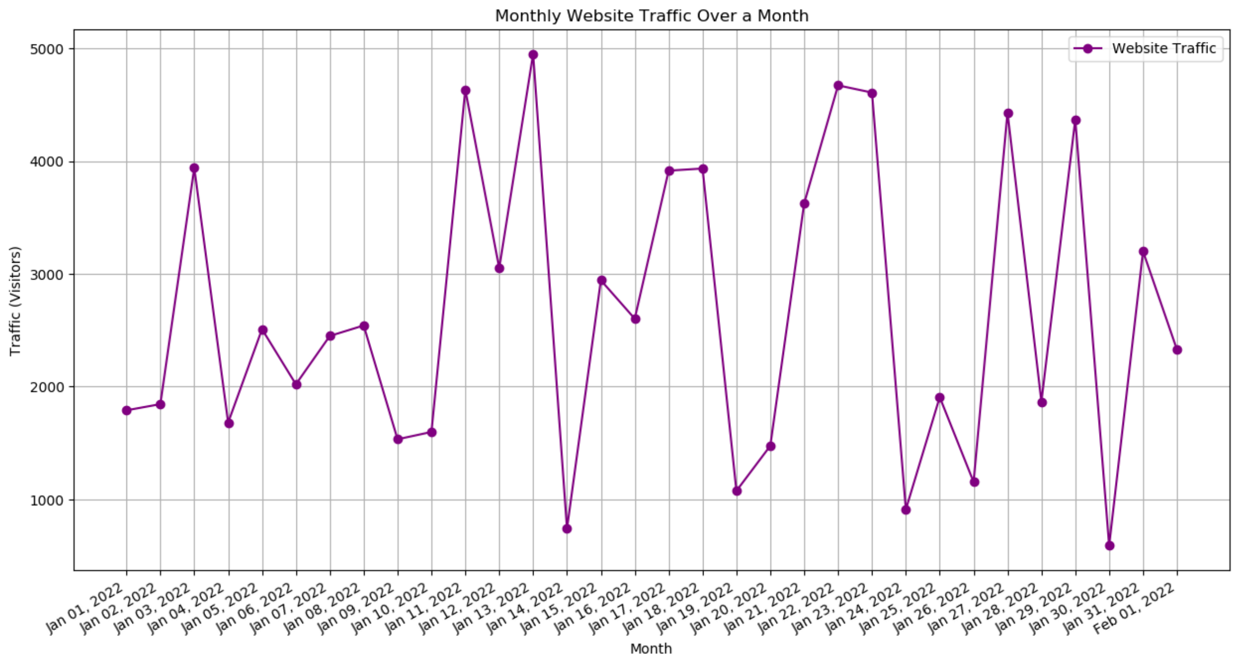
Imagine, you want to plot website traffic over one month. If you make a line plot, the x-axis will be very clumsy with all the dates and you can't see any dates properly! Something like below.

```
In [29]: import pandas as pd
         import matplotlib.dates as mdates
         import numpy as np
         np.random.seed(10)

         # Let's generate sample time series data for one month
         date_rng = pd.date_range(start='2022-01-01', end='2022-02-01')

         # Generate random website traffic values for one month
         traffic_data = np.random.randint(500, 5000,len(date_rng))

         # Create a DataFrame
         traffic_df = pd.DataFrame({'Month': date_rng, 'Traffic (Visitors)': traffic_dat
         a})

         # Set the figure size
         plt.figure(figsize=(15, 8))

         # Now, let's create a time series plot to visualize the monthly website traffic.
         plt.plot_date(traffic_df['Month'], traffic_df['Traffic (Visitors)'], label='Webs
         ite Traffic', color='purple', linestyle='-')

         # To show all the dates of one month
         plt.xticks(traffic_df['Month'])

         # Adding Labels and Title:
         plt.xlabel('Month')
         plt.ylabel('Traffic (Visitors)')
         plt.title('Monthly Website Traffic Over a Month')

         # Adding Grid Lines and Legends:
         plt.grid(True)
         plt.legend(['Website Traffic'], loc='upper right')

         # Display the plot
         plt.show()
```



Let's see the same example but with just three additional lines of customization that make the time series plot easily interpretable!

In [30]:
```python
# Set the figure size
plt.figure(figsize=(15, 8))

# Now, let's create a time series plot to visualize the monthly website traffic.
plt.plot_date(traffic_df['Month'], traffic_df['Traffic (Visitors)'], label='Webs
ite Traffic', color='purple', linestyle='-')

# To show all the dates of one month
plt.xticks(traffic_df['Month'])

# Adding Labels and Title:
plt.xlabel('Month')
plt.ylabel('Traffic (Visitors)')
plt.title('Monthly Website Traffic Over a Month')

# Set x-axis Date Format: Month Day, Year
date_format = mdates.DateFormatter('%b %d, %Y')
# Customize date formatting by using DateFormatter
plt.gca().xaxis.set_major_formatter(date_format)
# Autoformatting the x-axis
plt.gcf().autofmt_xdate()

# Adding Grid Lines and Legends:
plt.grid(True)
plt.legend(['Website Traffic'], loc='upper right')

# Display the plot
plt.show()
```



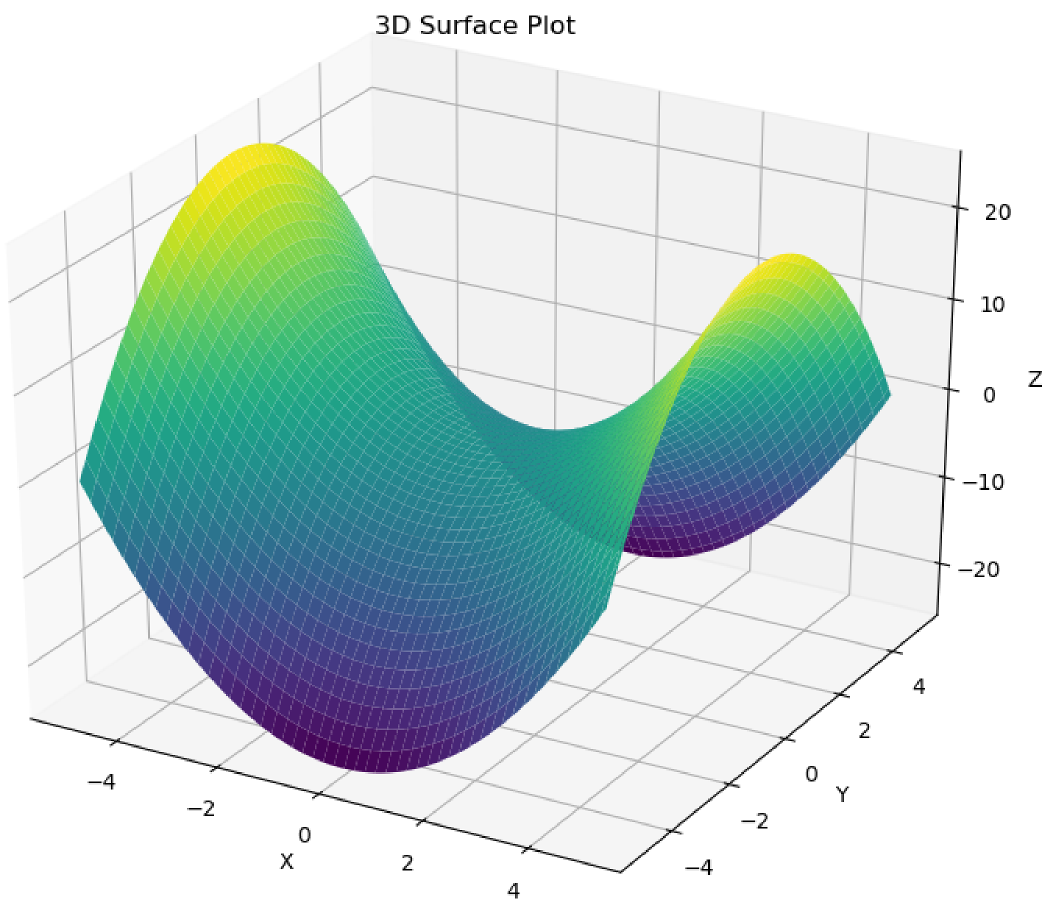
4.4 3D Plots

Creating 3D plots using Matplotlib involves using the mpl_toolkits.mplot3d toolkit, which provides functions for creating various types of 3D visualizations. you need to import the Axes3D to visualize the plots in 3D with the following command from mpl_toolkits.mplot3d import Axes3D .

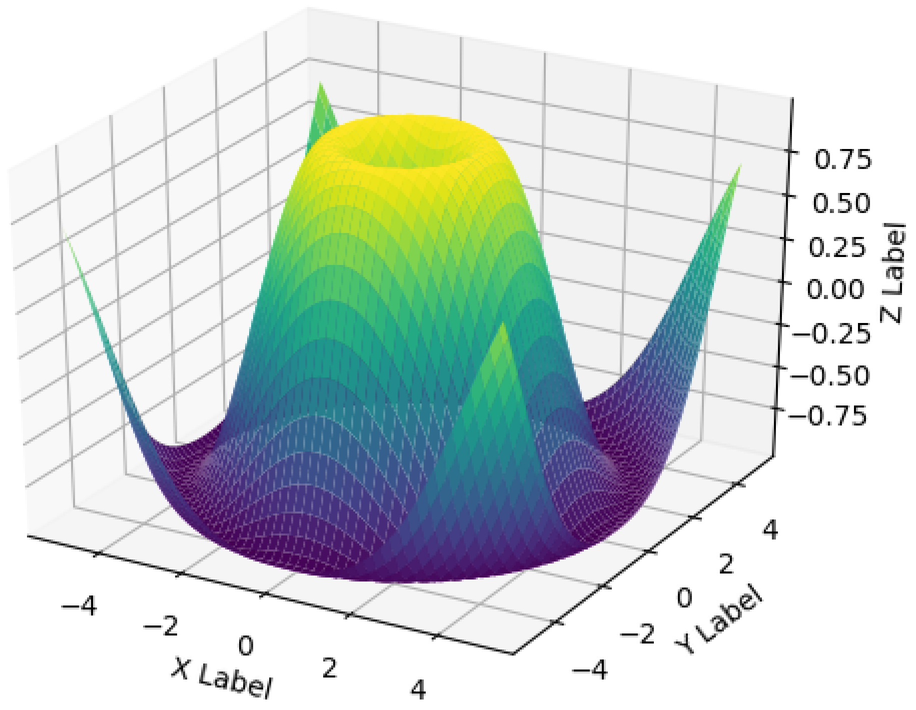
First, we need to create a Matplotlib figure object using fig=plt.figure() . To add a 3D subplot to the figure we need to use the add_subplot method, axes=fig.add_subplt(111,projection='3d') . In this case, (1, 1, 1) means there is only one row, and one column, and the current subplot is in the first (and only) position.

Let's create a 3D surface plot, you can also create a 3D line plot, 3D scatter plot, etc.

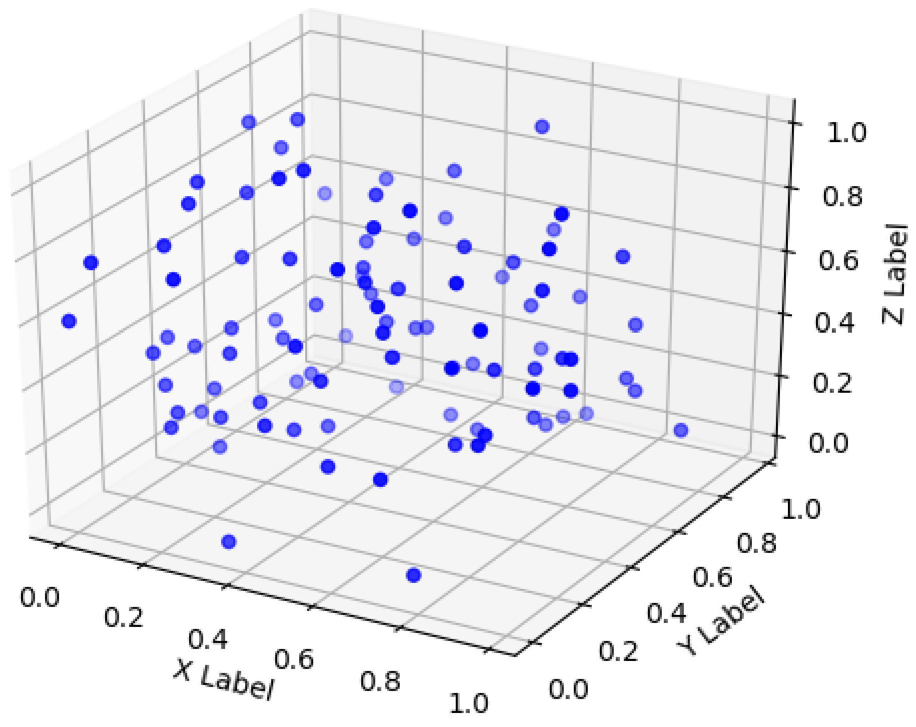
In [31]:
```python
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

# Create a meshgrid of X and Y values
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)

# Define the function to plot (example: a saddle shape)
Z = X**2 - Y**2

# Create a 3D surface plot
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

ax.plot_surface(X, Y, Z, cmap='viridis')

# Add title and labels
ax.set_title('3D Surface Plot')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')

plt.show()
```



In [32]:
```python
# Create a 3D surface plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Generate 3D data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create the surface plot
ax.plot_surface(X, Y, Z, cmap='viridis')

# Add labels
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```



In [33]:
```python
# Create a 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Generate random 3D data
x = np.random.rand(100)
y = np.random.rand(100)
z = np.random.rand(100)

# Create the scatter plot
ax.scatter(x, y, z, c='b', marker='o')

# Add labels
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```



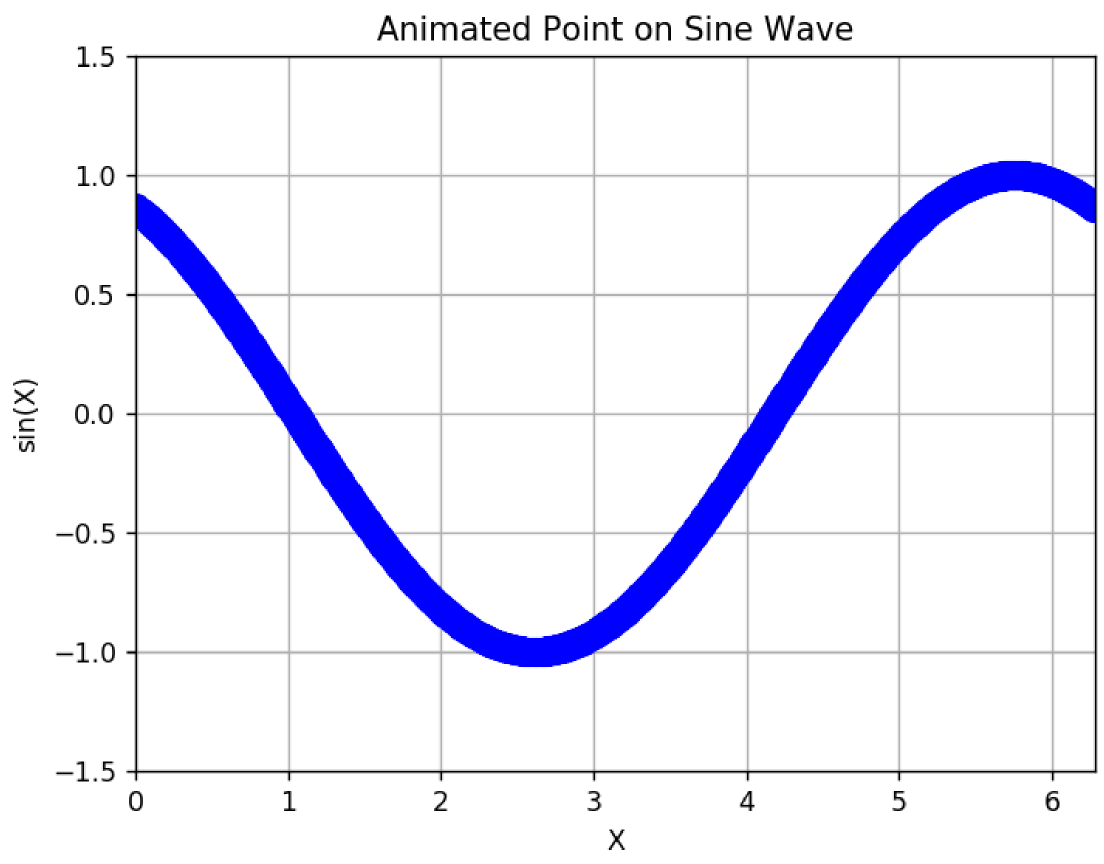
4.5 Animation

It would be nice to rotate, zoom, and hover to see the location of the above 3D plot, right? Guess what, we can actually do that in one line! Use the command %matplotlib notebook in your Jupyter Notebook to make the plot interactive. If you want to change it back to static plots use %matplotlib inline .

When the interactive plots are enabled, you can also create nice animation plots, like a moving sine wave, etc. That can be achieved by using FuncAnimation methods from matplotlib.animation module.

The FncAnimation method takes in the figure object, the function to call repeatedly, interval time to call the function. The below code will create an animated sine wave. So, Here animate function will be called every 1 second, and the resulting plot will be plotted in the figure object, so as it happens continuously we get an animation plot.

In [34]:
```python
from matplotlib.animation import FuncAnimation

%matplotlib notebook
# Create a figure and axis
fig, ax = plt.subplots()
ax.set_xlim(0, 2 * np.pi)
ax.set_ylim(-1.5, 1.5)

# Initialize the point to be animated
point, = ax.plot([], [], 'bo', markersize=10)

# Function to initialize the plot
def init():
    point.set_data([], [])
    return point,

# Function to update the animation
def animate(frame):
    x = np.linspace(0, 2 * np.pi, 1000)
    y = np.sin(x + 0.1 * frame)  # Vary the phase to create animation
    point.set_data(x, y)
    return point,

# Create the animation
ani = FuncAnimation(fig, animate, frames=100, init_func=init, blit=True, interva
l=50)

plt.title('Animated Point on Sine Wave')
plt.xlabel('X')
plt.ylabel('sin(X)')
plt.grid(True)

plt.show()
```



As you continue your journey in the world of data science and analysis, remember that Matplotlib provides the foundation for creating compelling visuals that tell your data's story, so just play with it. I hope you find this guide useful.

Throughout this two-part series, we have used Numpy often to create test data, if you are curious about learning more about Numpy, check out the below article.

Mastering Numpy - https://medium.com/python-in-plain-english/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84 (https://medium.com/python-in-plain-english/mastering-numpy-a-data-enthusiasts-essential-companion-392cdbe39e84)

Mastering Pandas - https://medium.com/python-in-plain-english/pandas-demystified-a-comprehensive-handbook-for-data-enthusiasts-part-1-136127e407f (https://medium.com/python-in-plain-english/pandas-demystified-a-comprehensive-handbook-for-data-enthusiasts-part-1-136127e407f)

Happy Learning!

Understanding the Complex Plots of Seaborn for Better Visualization!

In [2]:
```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

Table of Contents

3.Univariate Plots

- KDE Plot
- Rug Plot
- Dist Plot
- Box Plot & Violin Plot
- Strip Plot

4.Bivariate Plots

- Regression Plot
- Joint Plot
- Hexbin Plot

5.Multivariate Plots

- Using Parameters
- Relational Plot
- Facet Grid
- Pair Plot
- Pair Grid

6.Matrix Plots

- Heatmap
- Cluster Map

1. Seaborn Introduction

First things first, import seaborn as sns , ‘sns’ is the commonly used alias. We know NumPy is short for Numerical Python, and Matplotlib is short for Mathematical Plotting Library. But Guess what Seaborn is short for? A Movie Character!! Yes, Apparently it’s named after a character named Samuel Norman Seaborn from the television show “The West Wing” — thus, the standard alias is the character’s initials (“sns”).

There are many plots in Seaborn and it gets confusing to remember them! When I was thinking of an easy way to remember them, I realized that we can highly categorize them into the following sections based on variables, such as:

- Categorical Plots: Plots to visualize the Categorical data. E.g. Bar Plot, Count Plot, Swarm Plot, Point Plot, Cat Plot, Categorical Box Plot, Categorical Violin Plot, Categorical Swarm Plot, etc.
- Univariate Plots: Plots that visualize a single variable. E.g. Histogram, KDE Plot, Rug Plot, Box Plot, Dist Plot, Violin Plot, Strip Plot, etc.
- Bivariate Plots: Plots that visualize the relationship between 2 variables. E.g. Scatter Plot, Line Plot, Regression Plot, Join Plot, Hexbin Plot, etc.
- Multivariate Plots: plots that involve more than two variables. E.g. Pair Plot, Facet Grid, Relational Plot.
- Matrix Plots: plots visualize relationships within matrices of data. E.g. Heatmap, Cluster Map.

In [3]:
```
# Use sns.get_dataset_names() see the available datasets in seaborn
# Let's use the restaurant tips data available in seaborn
tips = sns.load_dataset('tips')
```

In [4]:
```
tips.head()
```

Out[4]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

2. Categorical Plots

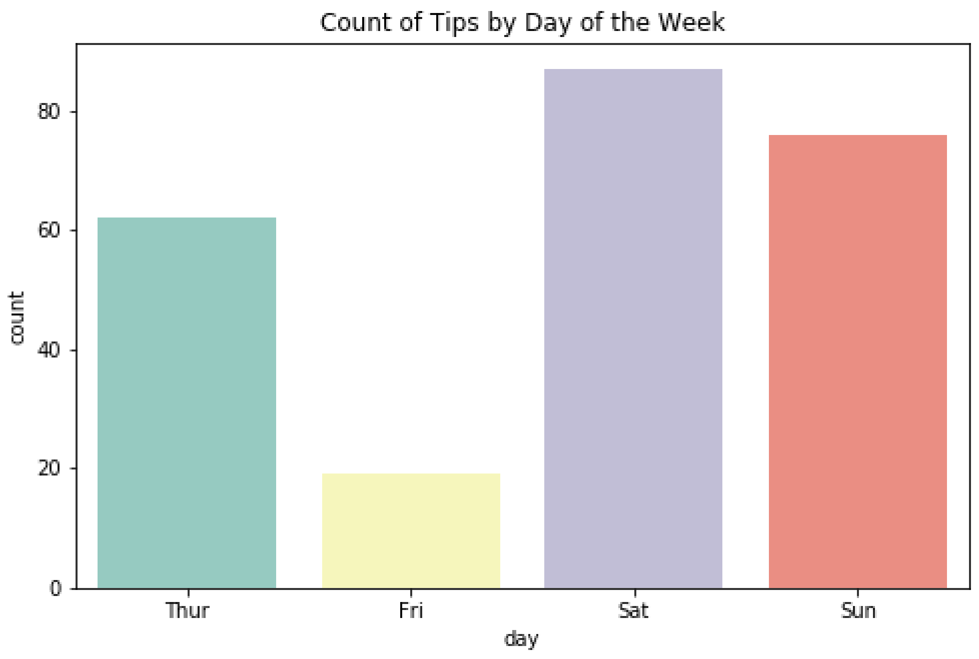
Categorical plots in data visualization are used to visualize categorical data, which consists of discrete and distinct categories or groups.

Count Plot

The Count Plot displays the counts of observations in each category. Suitable for visualizing the frequency or distribution of categorical data.

Use sns.countplot(x,data=dataframe) to create a count plot. If you see the command you can observe why I said Seaborn loves pandas! That is, unlike in Matplotlib, here you don’t need to pass the series to plot command (dataframe[column]), there is a parameter called data for which you need to assign your dataframe, then for the x and y parameters you can just assign your column name.

In [5]:
```
# Count Plot
plt.figure(figsize=(8, 5))
sns.countplot(x="day", data=tips, palette="Set3")
plt.title("Count of Tips by Day of the Week")
plt.show()
```



Few Other Use Cases: Analyzing the distribution of customer ratings (5-star, 4-star, 3-star) for a product or Visualizing the distribution of car types(sedan, SUV, truck) in a dealership dataset.

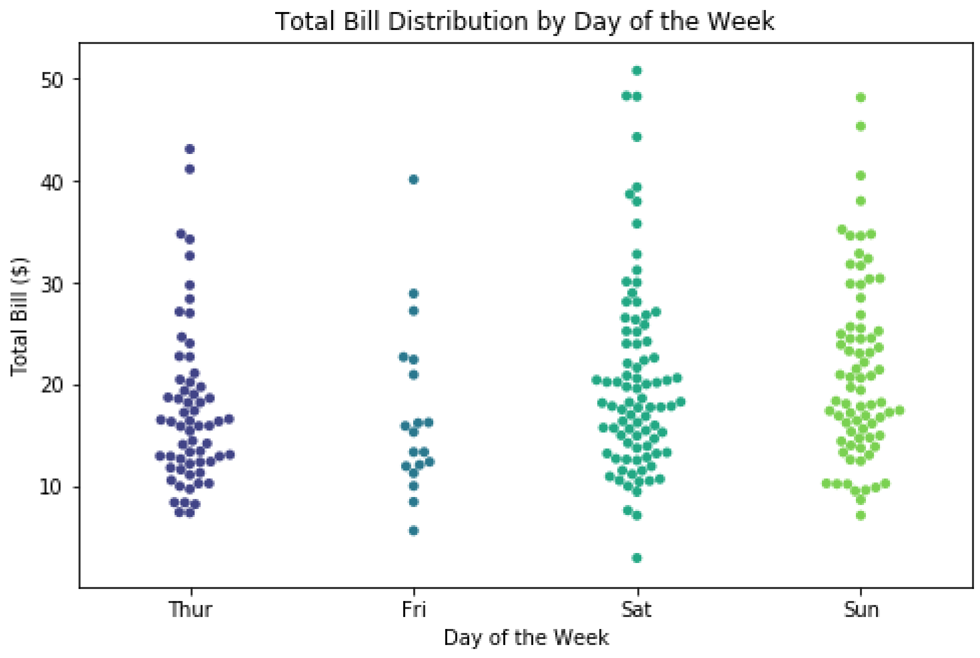
Swarm Plot

Swarm Plot typically has a categorical variable on the x-axis and a numerical variable on the y-axis, and it displays individual data points along each category. Now, As you can guess this gives us a visualization of the distribution and density of data points within categories.

One of the main features of a Swarm Plot is that it minimizes overlap between data points. This means that each data point is plotted in such a way that it does not overlap with other points in the same category. This makes it easier to see the density of data. But it’s difficult to visualize with larger data, so it’s effective with relatively small datasets.

Use sns.swarmplot(x,y,data) to make the swarm plot and use palette to change the colors.

```
In [6]: # Swarm Plot
        plt.figure(figsize=(8, 5))
        sns.swarmplot(x="day", y="total_bill", data=tips, palette="viridis")
        plt.title("Total Bill Distribution by Day of the Week")
        plt.xlabel("Day of the Week")
        plt.ylabel("Total Bill ($)")
        plt.show()
```

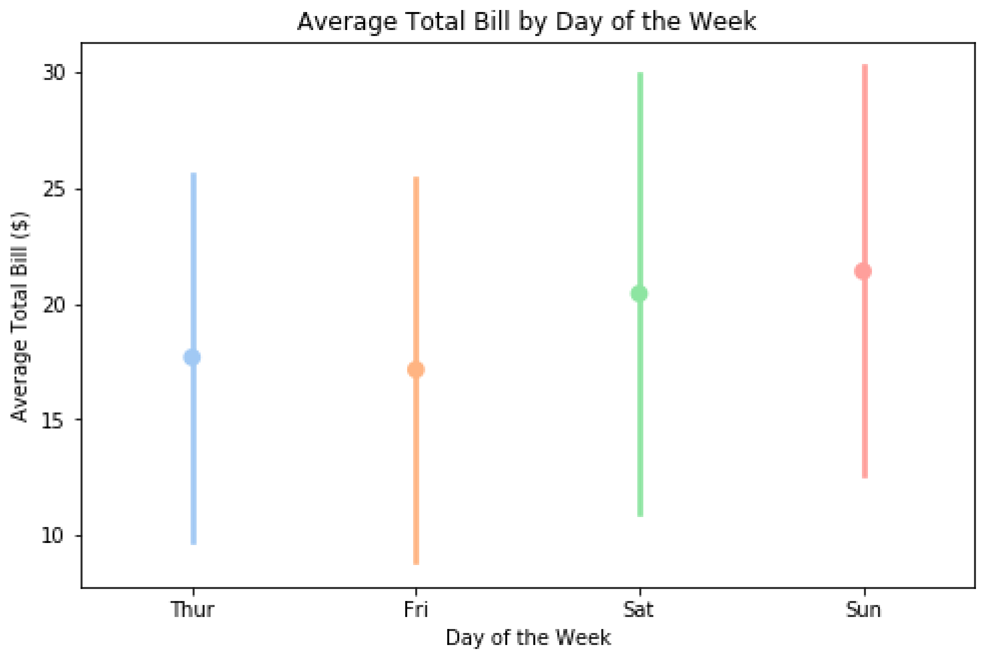


As you can see, They can help identify outliers or unusual data points within a category. so we can positively say that a swarm plot is useful in understanding the density of data points as they provide a clear representation of where data points are concentrated and where they are sparse.

Point Plot

The Point Plot typically has a categorical variable on the x-axis and a numerical variable on the y-axis and can provide insights into the distribution and central tendency of data within each category, such as a line or a point.

```
In [7]: # Point Plot
        plt.figure(figsize=(8, 5))
        sns.pointplot(x="day", y="total_bill", data=tips, ci="sd", palette="pastel")
        plt.title("Average Total Bill by Day of the Week")
        plt.xlabel("Day of the Week")
        plt.ylabel("Average Total Bill ($)")
        plt.show()
```



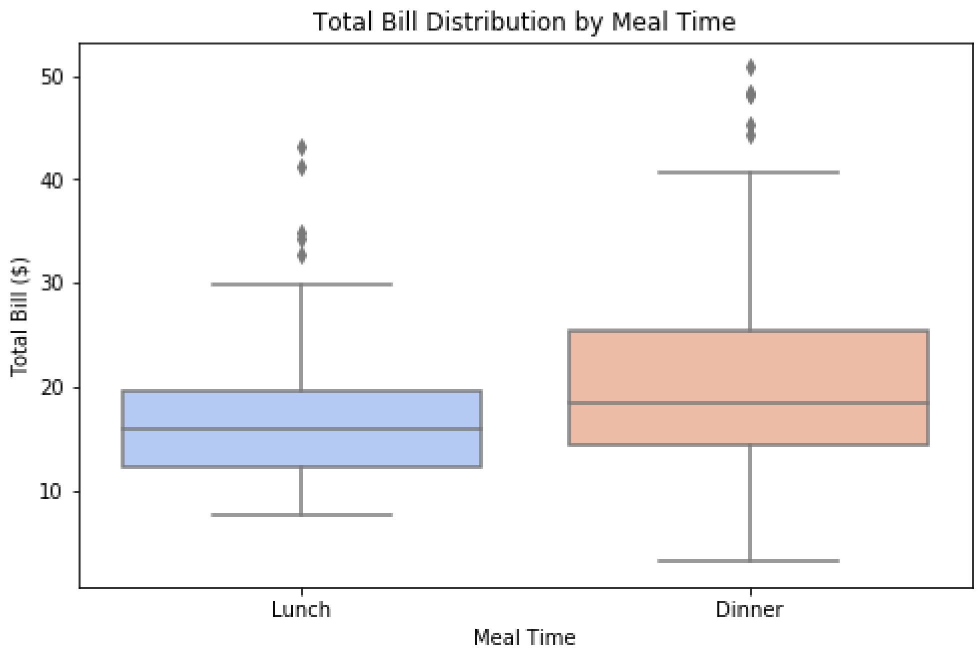
- This line or point represents the average or median value of the numerical variable for each category. So by using this, you can quickly see whether there are differences in the average or median values across categories.
- The same above use cases would also apply here, but just that we will observe the central tendency through this plot. This gives us idea of what is average, low, and high scores in each class.

Categorical Box Plot

A Categorical Box Plot, often referred to simply as a Box Plot or Box-and-Whisker Plot, is a type of categorical plot used to visualize the distribution of a numerical variable within different categories or groups, It displays quartiles (median, upper, and lower quartiles), potential outliers, and overall data spread.

Use sns.boxplot(x,y,data) to create a categorical box plot, but this plot is not necessarily for categories, you can even pass a numerical column to the x-axis.

```
In [8]: # Categorical Box Plot
        plt.figure(figsize=(8, 5))
        sns.boxplot(x="time", y="total_bill", data=tips, palette="coolwarm")
        plt.title("Total Bill Distribution by Meal Time")
        plt.xlabel("Meal Time")
        plt.ylabel("Total Bill ($)")
        plt.show()
```



- The plot consists of a rectangular “box” that spans the interquartile range (IQR) of the data, with a line inside the box representing the median (50th percentile) of the data. “Whiskers” extend(often 1.5 times the IQR) from the box to the minimum and maximum values within a defined range. And points outside the whiskers are potentially considered to be outliers.
- You can use this plot when you want Basic Statistics for a categorical Variable over a Numerical Variable. In fact, you can also use this for two numeric, e.g., size and total_bills in this tips datasets, try that out.

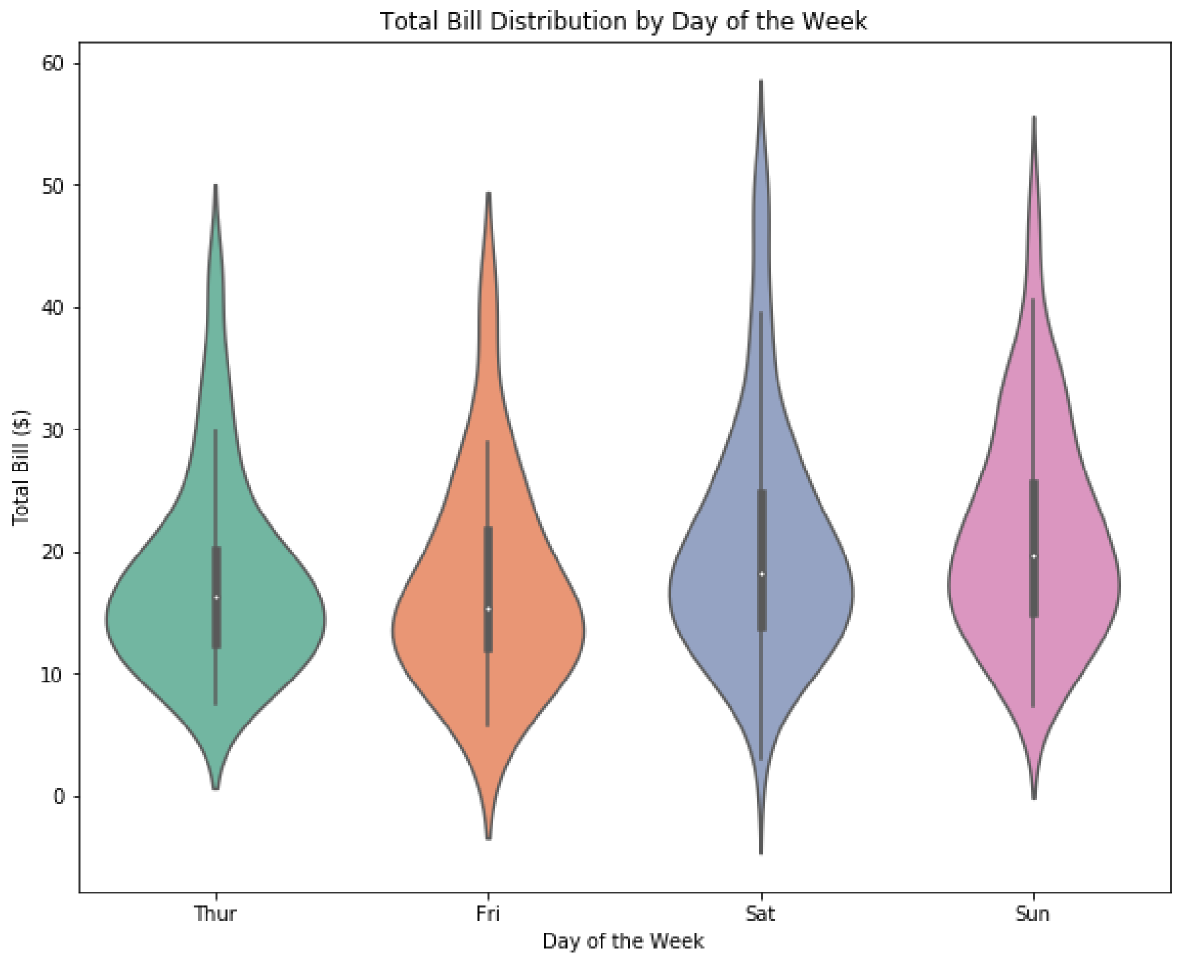
Categorical Violin Plot

The categorical variable determines the groups or categories, and a numerical variable is used to create the Violin Plots within each category. The width of the plot at a specific point indicates the density of data points at that value, and the central part of a Violin Plot resembles a violin, which is why it’s called a Violin Plot.

Inside the Violin, there is often a box plot representation that includes the median, quartiles, and potential outliers. Hence, This provides a summary of the central tendency and spread of the data to provide a comprehensive view of the distribution of a numerical variable within each category.

Use sns.violinplot(x,y,data) to create a violin plot.

```
In [9]: # Categorical Violin Plot
        plt.figure(figsize=(10, 8))
        sns.violinplot(x="day", y="total_bill", data=tips, palette="Set2")
        plt.title("Total Bill Distribution by Day of the Week")
        plt.xlabel("Day of the Week")
        plt.ylabel("Total Bill ($)")
        plt.show()
```



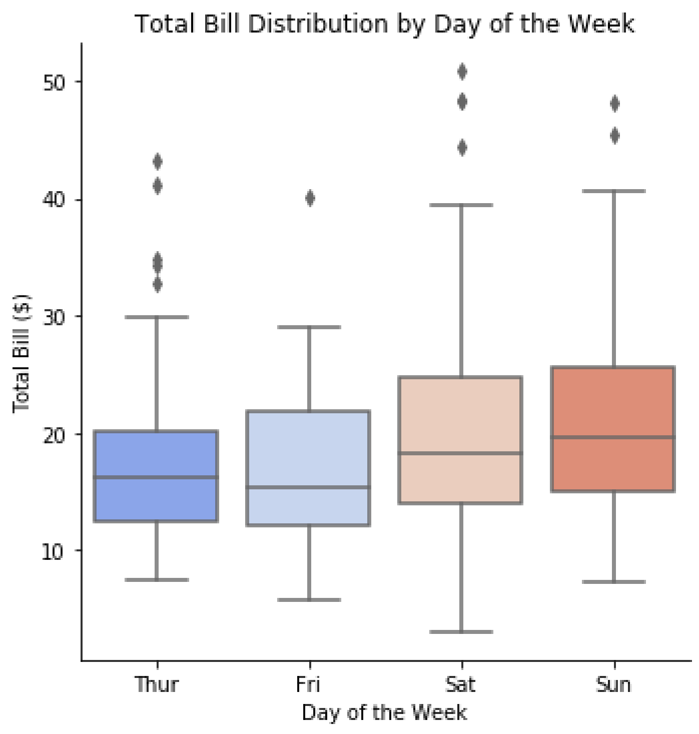
- As we already observed the density from the swarm plot, we can observe that it combines features of a box plot and a density plot.
- Hence, we can say that Categorical Violin Plots are effective for comparing the distribution of numerical data across different categories. They allow you to see not only the central tendency and spread but also the shape and skewness of the distribution.

Cat Plot

A Cat Plot, short for “Categorical Plot,” is a very powerful plotting function in Seaborn. It allows you to all the categorical plots we have seen above with just one parameter called kind in the command sns.catplot(x,y).You can usekind=bar, swarm , box, violin , count , point , etc.

```
In [10]: # Box Plot using cat plot
         plt.figure(figsize=(8, 5))
         sns.catplot(x="day", y="total_bill", data=tips, kind="box", palette="coolwar
         m")
         plt.title("Total Bill Distribution by Day of the Week")
         plt.xlabel("Day of the Week")
         plt.ylabel("Total Bill ($)")
         plt.show()
```

```
<Figure size 576x360 with 0 Axes>
```



3. Univariate Plots

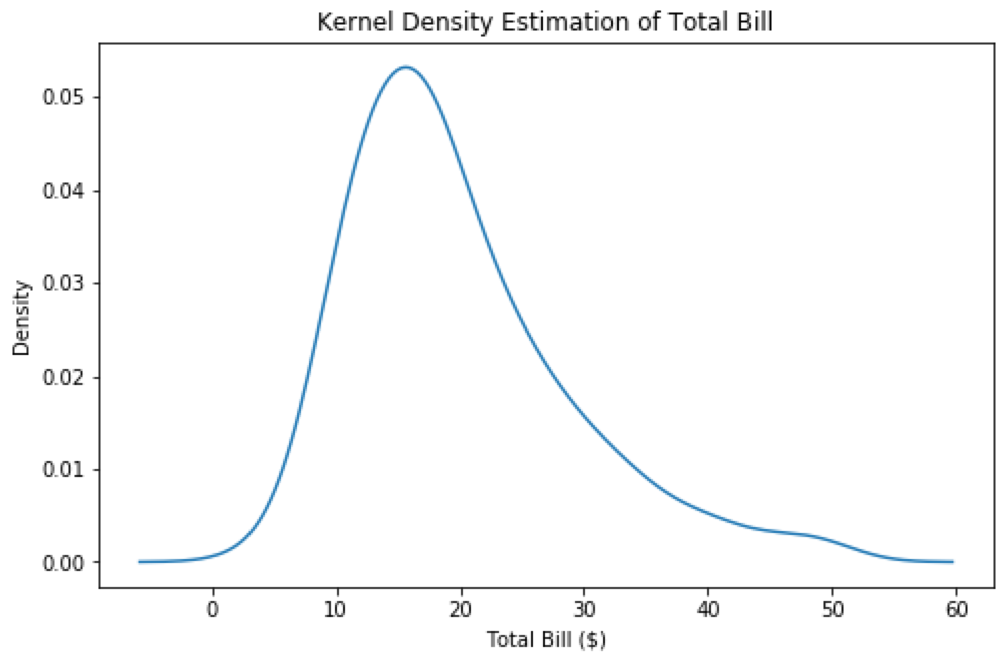
Univariate plots are used to visualize and analyze a single variable at a time. They are useful for understanding the distribution, and central tendency, identifying outliers, and checking for patterns or trends within a single variable.

KDE Plot

A KDE (Kernel Density Estimation) Plot is a type of data visualization that is used to estimate the probability density function of a continuous variable. So, we will have our continuous variable on the x-axis and the y-axis represents the estimated probability density area under the curve sums to 1, indicating that the data distribution is normalized.

Use sns.kdeplot(data=dataframe[column]) to create a KDE plot for a continuous variable. You can customize the appearance of the KDE Plot by adjusting parameters such as bandwidth, kernel type, color, and more.

```
In [11]: # KDE Plot
         plt.figure(figsize=(8, 5))
         sns.kdeplot(data=tips['total_bill'])
         plt.title("Kernel Density Estimation of Total Bill")
         plt.xlabel("Total Bill ($)")
         plt.ylabel("Density")
         plt.savefig('kde_plot.png')
         plt.show()
```



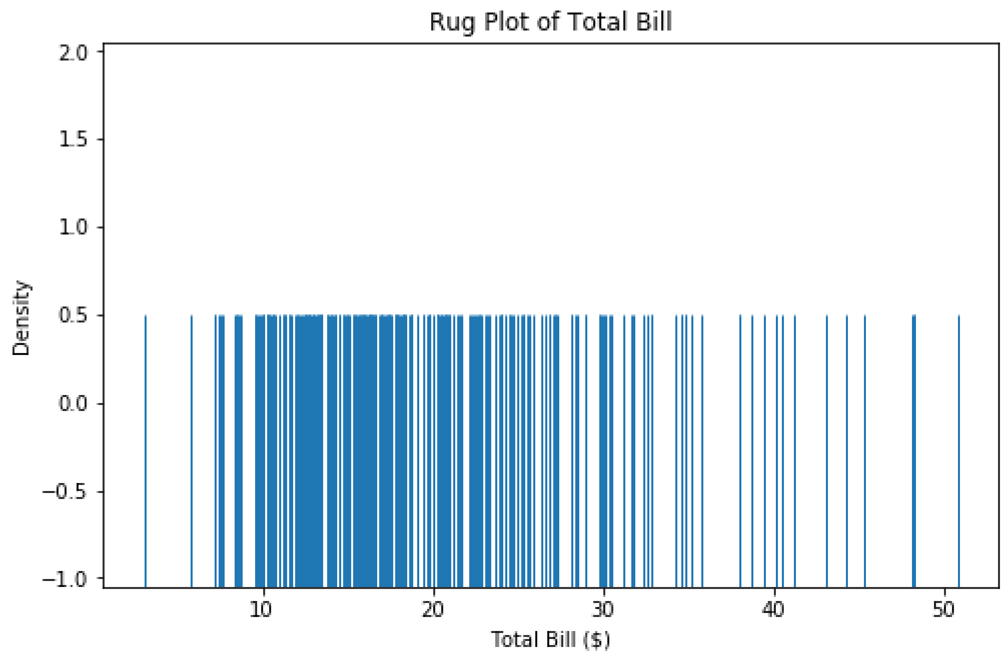
Hence, KDE Plots provide insights into the shape, peaks, modes, and spread of the data, and these are valuable for estimating the probability density of data points, which can be helpful in statistical analysis and hypothesis testing.

Rug Plot

A Rug Plot consists of vertical lines (or “ticks”) positioned along a single axis (usually the x-axis). Each tick represents the location of an individual data point, and the closeness of the ticks represents the density of the data points; typically denser areas have more ticks closely packed together.

Use sns.rugplot(data=dataframe[column]) to create a rug plot. You can customize the appearance of Rug Plots by adjusting parameters such as the height, color, and orientation of the ticks.

```
In [12]: # Rug Plot
         plt.figure(figsize=(8, 5))
         sns.rugplot(data=tips['total_bill'], height=0.5)
         plt.title("Rug Plot of Total Bill")
         plt.xlabel("Total Bill ($)")
         plt.ylabel("Density")
         plt.savefig('rug_plot.png')
         plt.show()
```

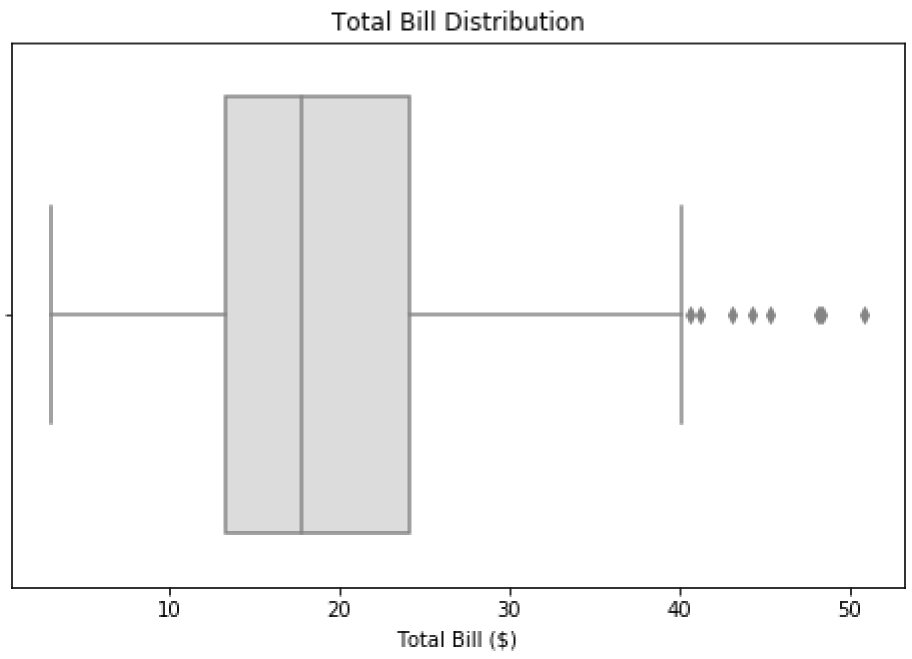


- From KDE, we got the insight that the Bill Amount peak is somewhere between 10–20, but through the Rug plot, we get an idea that the density is more at 15–20.
- Hence, This plot is particularly useful when you want to see the exact position of individual data points. Rug Plots are often combined with histograms, KDE plots, or box plots to provide additional context and detail. They can help highlight outliers or areas of high data density.

Box Plot

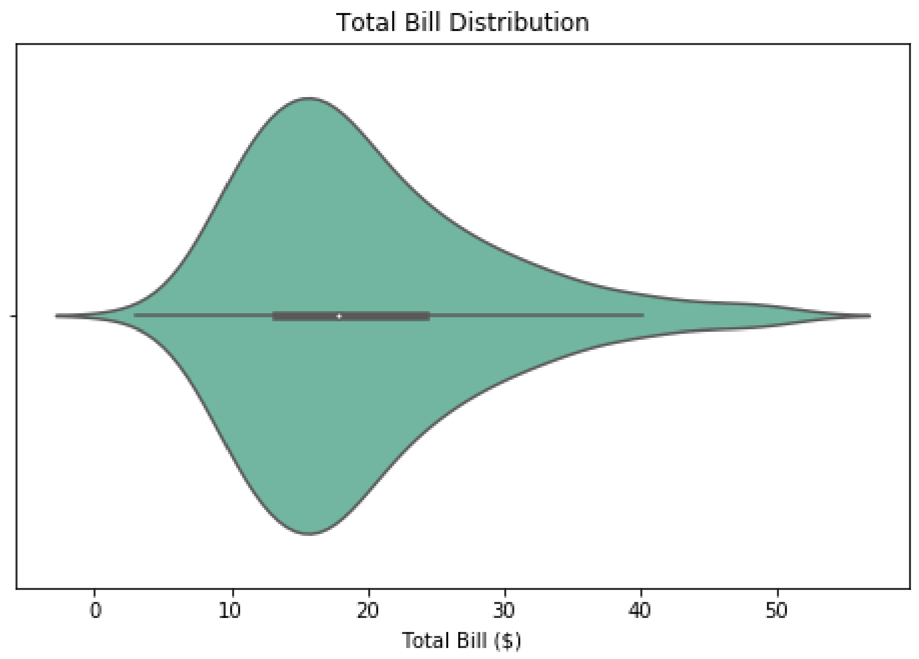
We have discussed these in the categorical plots section, everything is the same, just that you pass only one variable when you want to do univariate analysis.

In [13]:
```python
# Box Plot for Univariate Visualization
plt.figure(figsize=(8, 5))
sns.boxplot(x=tips['total_bill'], palette="coolwarm")
plt.title("Total Bill Distribution")
plt.xlabel("Total Bill ($)")
plt.savefig('univariate_box_plot.png')
plt.show()
```



Violin Plot

```
In [14]: # Violin Plot for Univariate Visualization
         plt.figure(figsize=(8, 5))
         sns.violinplot(x=tips['total_bill'], palette="Set2")
         plt.title("Total Bill Distribution")
         plt.xlabel("Total Bill ($)")
         plt.show()
```

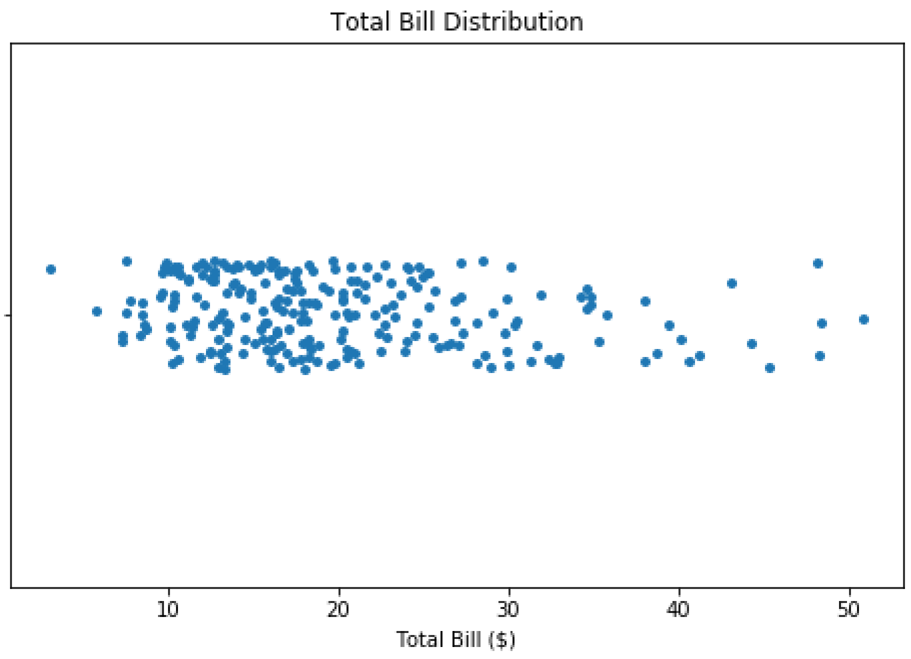


Strip Plot

A Strip Plot is a bit similar to a Swarm Plot and displays individual data points along a single axis. However, the data points will overlap in the strip plot, so we can say that Swarm Plots are particularly useful with smaller datasets and when you want to prevent overlap.

Use sns.stripplot(data=dataframe[column]) , and you can also use this for bivariate. Jittering (adding a small amount of random noise) can be applied to the data points to reduce the overlap and improve visibility.

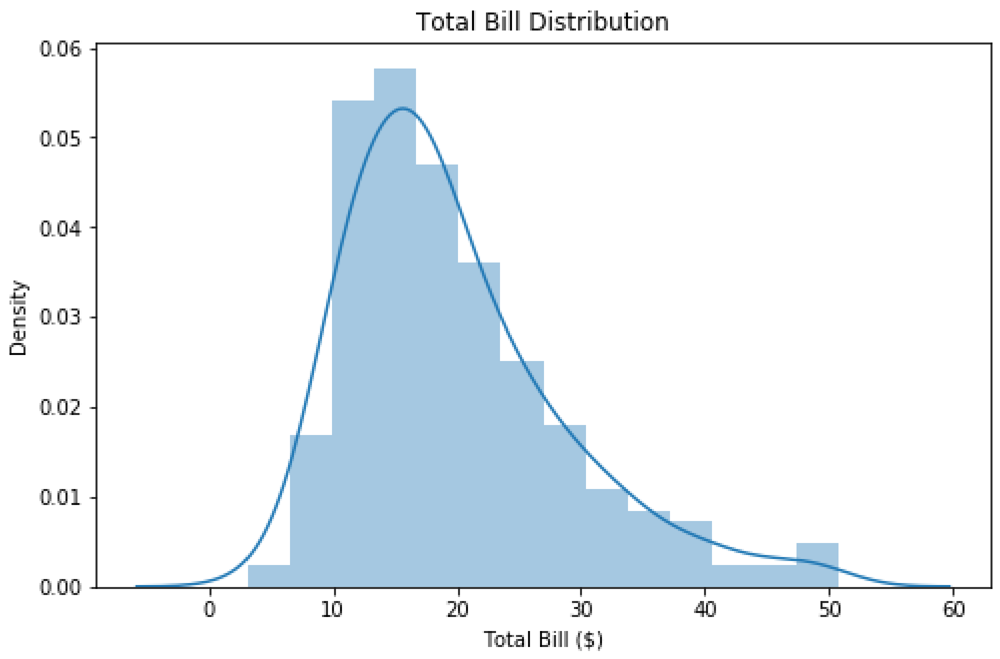
```
In [15]: # Strip Plot for Univariate Visualization
         plt.figure(figsize=(8, 5))
         sns.stripplot(x=tips['total_bill'], jitter=True)
         plt.title("Total Bill Distribution")
         plt.xlabel("Total Bill ($)")
         plt.savefig('strip_plot.png')
         plt.show()
```



Dist Plot

A Dist Plot short for Distribution plot typically resembles a histogram, but additionally, it includes a smoothed curve, which is a KDE plot. It divides the range of the numerical variable into bins or intervals and displays the frequency or density of data points within each bin.

In [16]:
```python
# Strip Plot for Univariate Visualization
plt.figure(figsize=(8, 5))
sns.distplot(a=tips['total_bill'])
plt.title("Total Bill Distribution")
plt.xlabel("Total Bill ($)")
plt.savefig('distribution_plot.png')
plt.show()
```



Optionally, a rug plot can be added along the x-axis, showing individual data points as small vertical lines. This provides insight into the density and distribution of individual data points.

4. Bi-Variate Plots

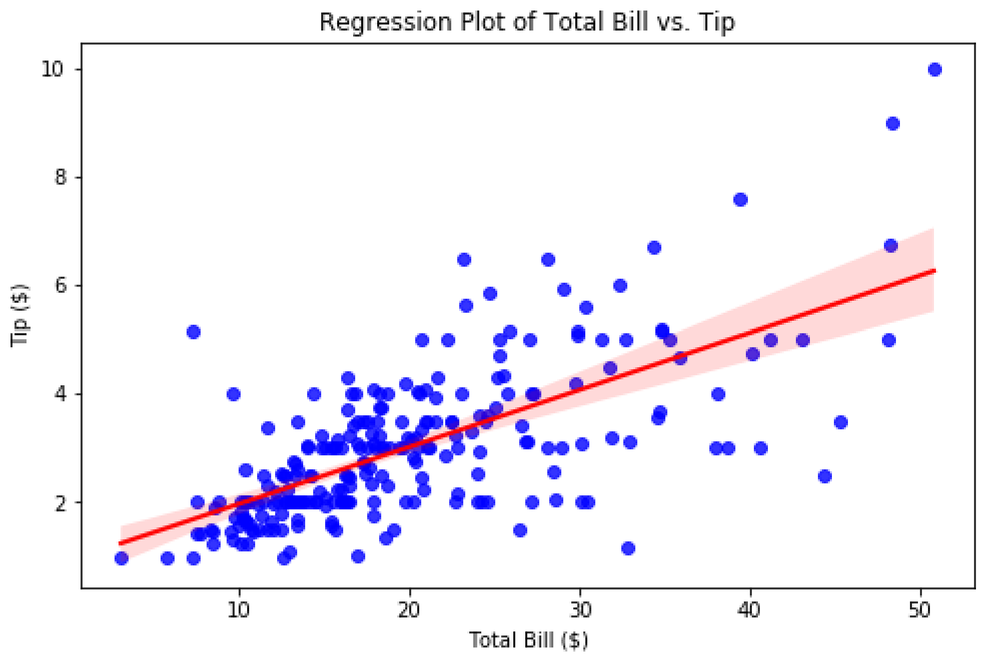
Bivariate plots involve the visualization and analysis of the relationship between two variables simultaneously. They are used to explore how two variables are related or correlated.

Regression Plot

A Regression Plot focuses on the relationship between two numerical variables: the independent variable (often on the x-axis) and the dependent variable (on the y-axis). There are individual data points are displayed as dots and the central element of a Regression Plot is the regression line or curve, which represents the best-fitting mathematical model that describes the relationship between the variables.

Use sns.regplot(x,y,data) to create a regression plot.

```
In [17]: # Regression Plot
         plt.figure(figsize=(8, 5))
         sns.regplot(x="total_bill", y="tip", data=tips, scatter_kws={"color": "blu
         e"}, line_kws={"color": "red"})
         plt.title("Regression Plot of Total Bill vs. Tip")
         plt.xlabel("Total Bill ($)")
         plt.ylabel("Tip ($)")
         plt.savefig('regression_plot.png')
         plt.show()
```



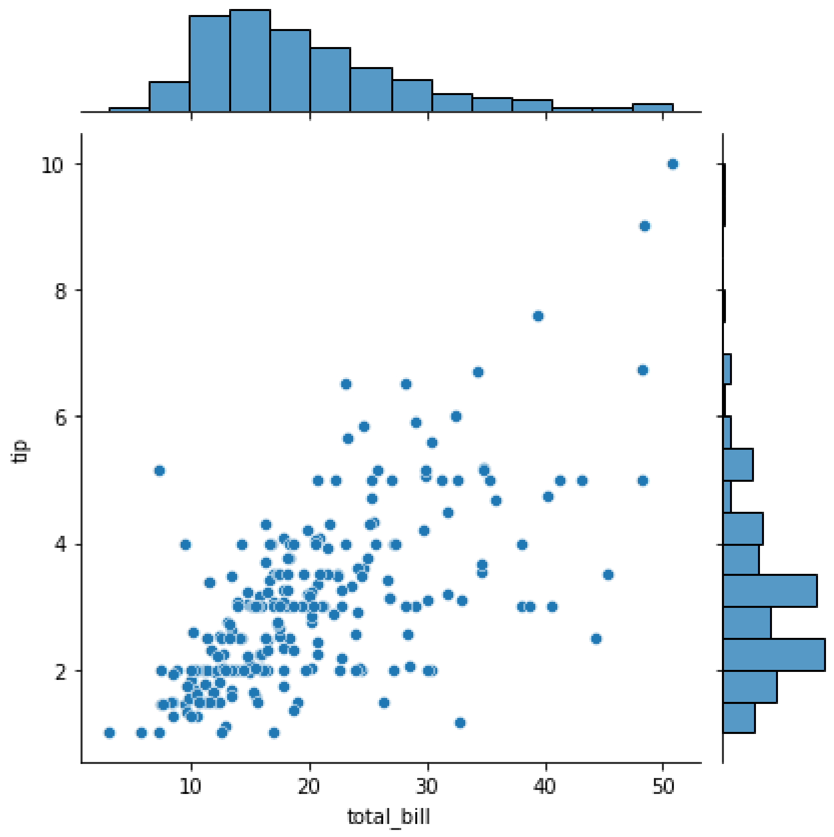
The regression line represents the best-fitting linear model for predicting tips based on total bill amounts. The scatter points show individual data points, and you can observe how they cluster around the regression line. This plot is useful for understanding the linear relationship between these two variables.

Joint Plot

A joint plot combines scatter plots, histograms, and density plots to visualize the relationship between two numerical variables. The central element of a Joint Plot is a Scatter Plot that displays the data points of the two variables against each other, along the x-axis and y-axis of the Scatter Plot, there are histograms or Kernel Density Estimation (KDE) plots for each individual variable. These marginal plots show the distribution of each variable separately.

Use sns.jointplot(x,y,data=dataframe,kind) , kind can be one of [‘scatter’, ‘hist’, ‘hex’, ‘kde’, ‘reg’, ‘resid’] these.

```
In [18]: # Joint Plot
         sns.jointplot(x="total_bill", y="tip", data=tips, kind="scatter")
         plt.savefig('joint_plot.png')
         plt.show()
```



As we can see, this shows the relation between the two variables through a scatter plot, while the marginal histograms show the distribution of each variable separately.

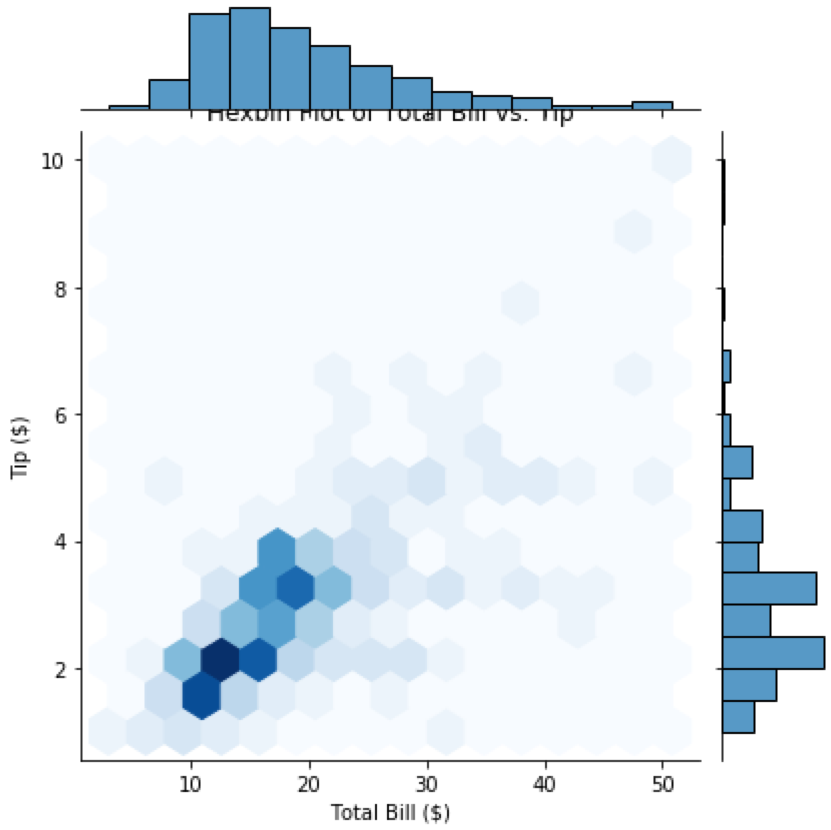
Hexbin Plot

A Hexbin plot, short for Hexagonal Binning plot, groups data points into hexagonal bins, allowing you to visualize data density and patterns more effectively. These are especially valuable with large datasets when scatter plots with individual points become too crowded and hard to interpret!

You can create a Hexbin plot, by using the kind parameter of joint plot. You can customize the color map, grid size, and other plot parameters to fine-tune the appearance of the Hexbin Plot.

```
In [19]: # Hexbin Plot
         plt.figure(figsize=(8, 5))
         sns.jointplot(x="total_bill", y="tip",kind='hex', data=tips, gridsize=15, cma
         p="Blues")
         plt.title("Hexbin Plot of Total Bill vs. Tip")
         plt.xlabel("Total Bill ($)")
         plt.ylabel("Tip ($)")
         plt.savefig('hexbin_plot.png')
         plt.show()
```

```
<Figure size 576x360 with 0 Axes>
```



Now, this confirms that the Hexbin plot gives much more clarity than the scatter plot for a large dataset. Each hexagon in the plot is color-coded to indicate the density of data points within that bin.

5. MultiVariate Plots

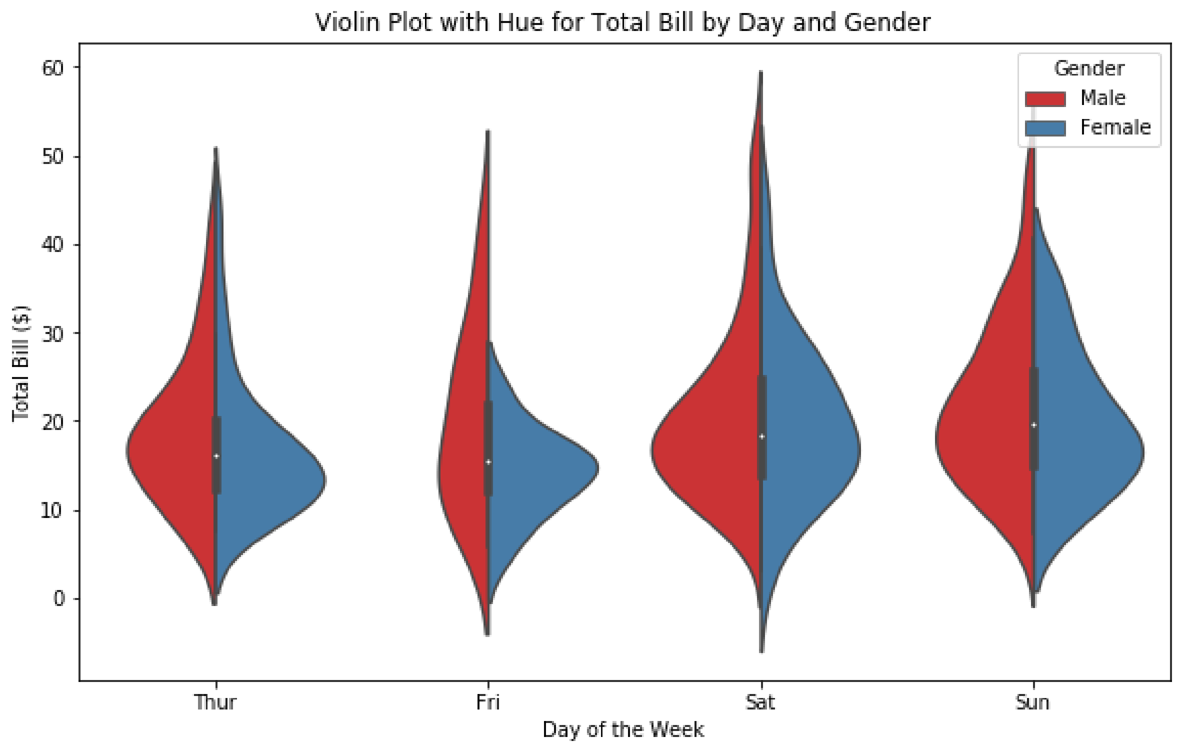
These are my favorite, these plots give us a lot of flexibility to explore the relationships and patterns among three or more variables simultaneously. That is, Multivariate plots extend the analysis to more than two variables, which will be often needed in the Data Analysis.

Using Parameters

Using Hue Parameter:

Using the hue parameter will add color to the plot based on the provided categorical variable, specifying a unique color for each of the categories. This parameter can be used almost all of the plots like .scatterplot() , .boxplot() , .violinplot() , .lineplot() , etc.

In [20]:
```
# Violin Plot with Hue
plt.figure(figsize=(10, 6))
sns.violinplot(
    x="day",          # x-axis: Days of the week (categorical)
    y="total_bill",   # y-axis: Total bill amount (numerical)
    data=tips,
    hue="sex",        # Color by gender (categorical)
    palette="Set1",   # Color palette
    split=True        # Split violins by hue categories
)

plt.title("Violin Plot with Hue for Total Bill by Day and Gender")
plt.xlabel("Day of the Week")
plt.ylabel("Total Bill ($)")
plt.legend(title="Gender")
plt.show()
```

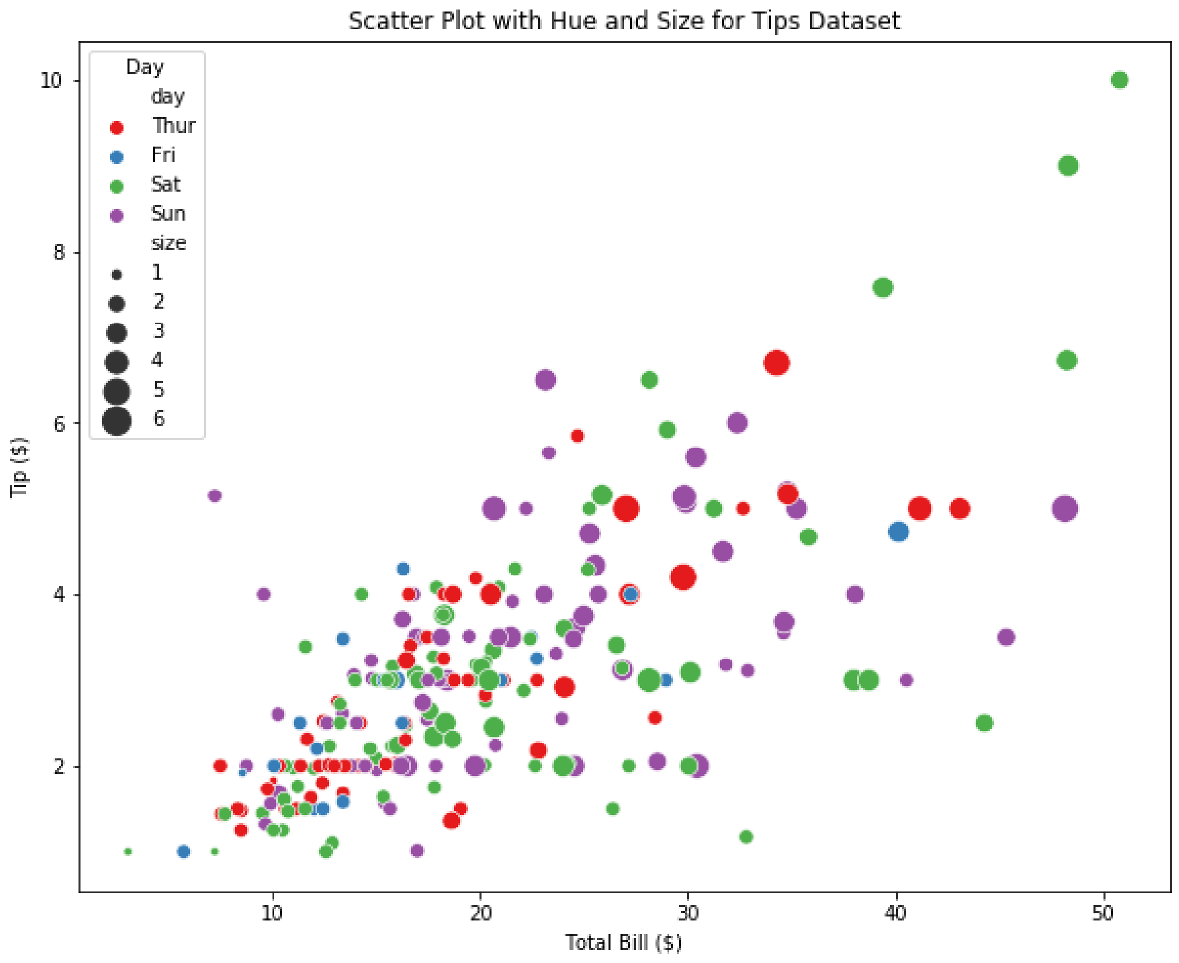


- We use the sex column to color the violins based on gender, adding one dimension to the plot. Each gender is represented by a different color.

Using hue and size parameters:

size is another parameter that can be used to add another numeric variable dimension.

```
In [21]: import seaborn as sns
         import matplotlib.pyplot as plt

         # Load the "tips" dataset
         tips = sns.load_dataset("tips")

         # Scatter Plot with Hue and Size
         plt.figure(figsize=(10, 8))
         sns.scatterplot(
             x="total_bill",
             y="tip",
             data=tips,
             hue="day",       # Color by day (categorical)
             size="size",     # Vary marker size by size column (numerical)
             sizes=(20, 200),  # Define the size range for markers
             palette="Set1"   # Color palette
         )

         plt.title("Scatter Plot with Hue and Size for Tips Dataset")
         plt.xlabel("Total Bill ($)")
         plt.ylabel("Tip ($)")
         plt.legend(title="Day")
         plt.savefig('scatterplot_with_hue_and_size.png')
         plt.show()
```



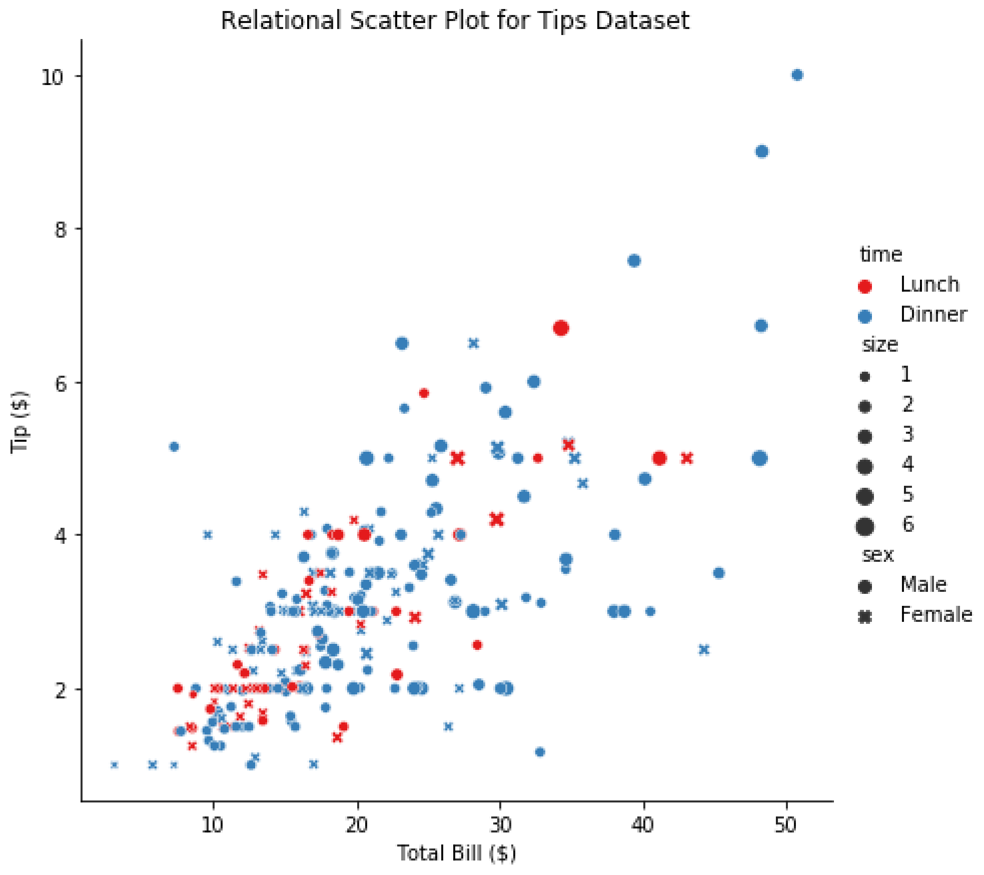
Each day is represented by a different color with the hue parameter adding a third dimension, and marker size based on the size column in the dataset to add a fourth dimension.

Relational Plot

A Relational Plot allows you to visualize the relationship between two numerical variables, along with additional categorical or numerical dimensions.

Use sns.relplot(x,y,data,hue,size,style). you can use the hue parameter to color data points by a categorical variable, the size parameter to vary marker size based on a numerical variable, and the style parameter to differentiate markers or lines by a categorical variable. The kind parameter allows you to specify the type of relational plot you want to create.

In [22]:
```
# Create a scatterplot using a Relational Plot (relplot)
sns.relplot(x="total_bill", y="tip", data=tips, hue="time", style="sex", size
="size", palette="Set1", height=6)
plt.title("Relational Scatter Plot for Tips Dataset")
plt.xlabel("Total Bill ($)")
plt.ylabel("Tip ($)")
plt.savefig('relplot_with_mv.png')
plt.show()
```

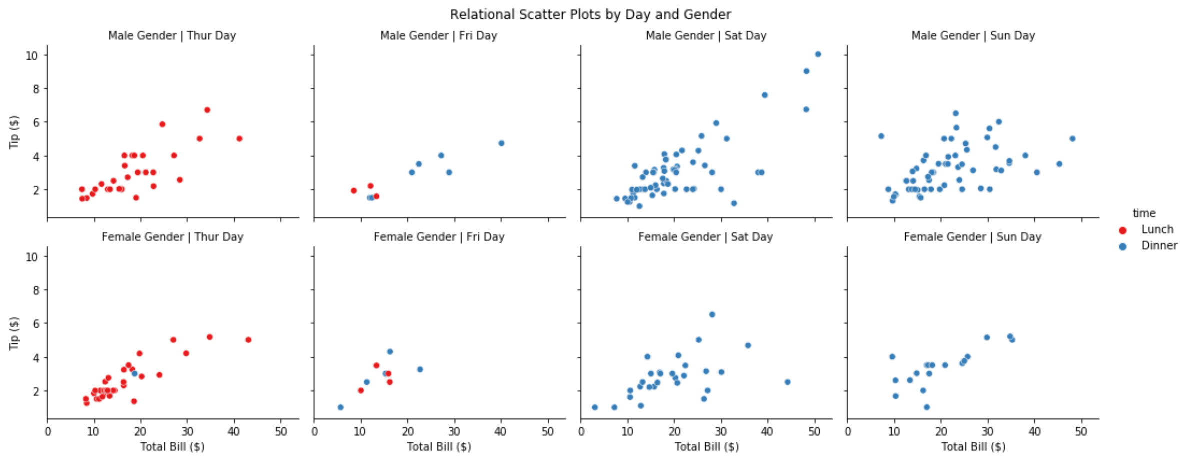


The time column is used for coloring data points with the hue parameter, size column for varying size with the size parameter, and sex column for the marker parameter.

FacetGrid by using col and row parameters of relplot

You can also create a Relational Plot (relplot) using the col and row parameters to create a facet grid, allowing you to visualize relationships within different subsets of the data. Assign the category columns for col and row.

```
In [23]: import seaborn as sns
         import matplotlib.pyplot as plt

         # Load the "tips" dataset
         tips = sns.load_dataset("tips")

         # Create a facet grid using a Relational Plot (relplot)
         g = sns.relplot(
             x="total_bill",
             y="tip",
             data=tips,
             hue="time",
             col="day",  # Separate plots by day (columns)
             row="sex",  # Separate plots by gender (rows)
             palette="Set1",
             height=3,  # Height of each subplot
             aspect=1.2  # Aspect ratio of each subplot
         )

         # Set titles and labels for the facets
         g.set_titles(col_template="{col_name} Day", row_template="{row_name} Gender")
         g.set_axis_labels("Total Bill ($)", "Tip ($)")

         plt.suptitle("Relational Scatter Plots by Day and Gender")
         plt.subplots_adjust(top=0.9)  # Adjust the title position
         plt.savefig('relplot_col_row.png')
         plt.show()
```

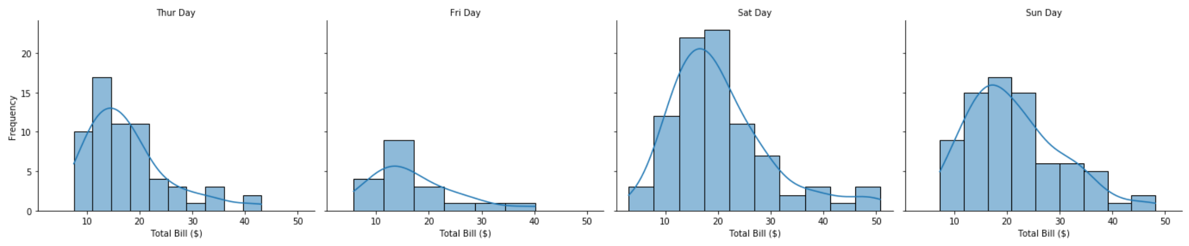


FacetGrid

A Facet Grid is a feature in Seaborn that allows you to create a grid of subplots, each representing a different subset of your data. In this way, Facet Grids are used to compare patterns or relationships with multiple variables within different categories.

Use sns.FacetGrid(data,col,row) to create a facet grid, which returns the grid object. After creating the grid object you need to map it to any plot of your choice.

In [24]:
```
# Create a Facet Grid of histograms for different days
g = sns.FacetGrid(tips, col="day", height=4, aspect=1.2)
g.map(sns.histplot, "total_bill", kde=True)
g.set_axis_labels("Total Bill ($)", "Frequency")
g.set_titles(col_template="{col_name} Day")
plt.savefig('facet_grid_hist_plot.png')
plt.show()
```

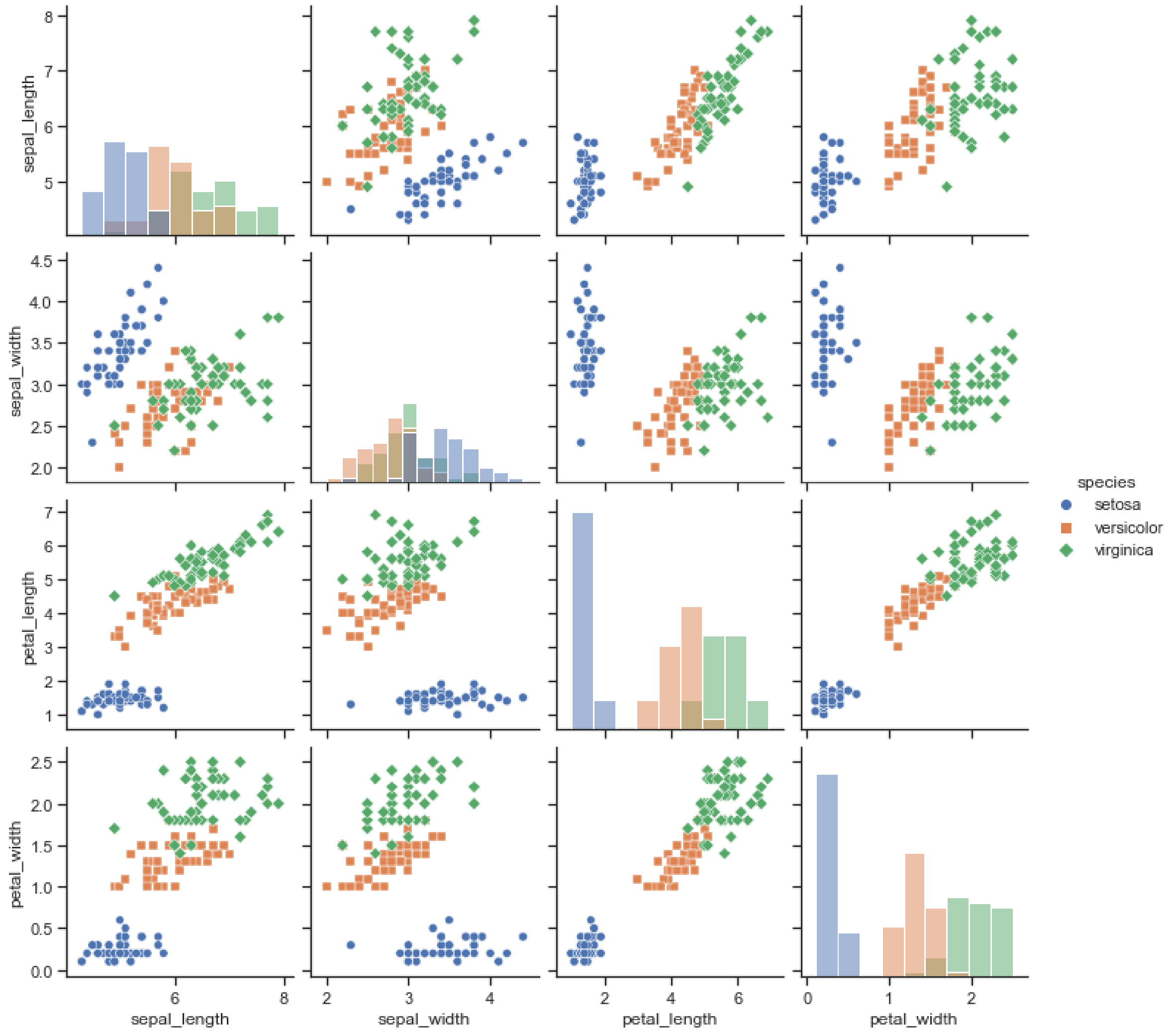


Pair Plot

A Pair plot provides a grid of scatterplots, and histograms, where each plot shows the relationship between two variables, which is why it is also called a Pairwise Plot or Scatterplot Matrix.

The diagonal cells typically display histograms or kernel density plots for individual variables, showing their distributions. The off-diagonal cells in the grid often display scatterplots, showing how two variables are related. Pair Plots are particularly useful for understanding patterns, correlations, and distributions across multiple dimensions in your data.

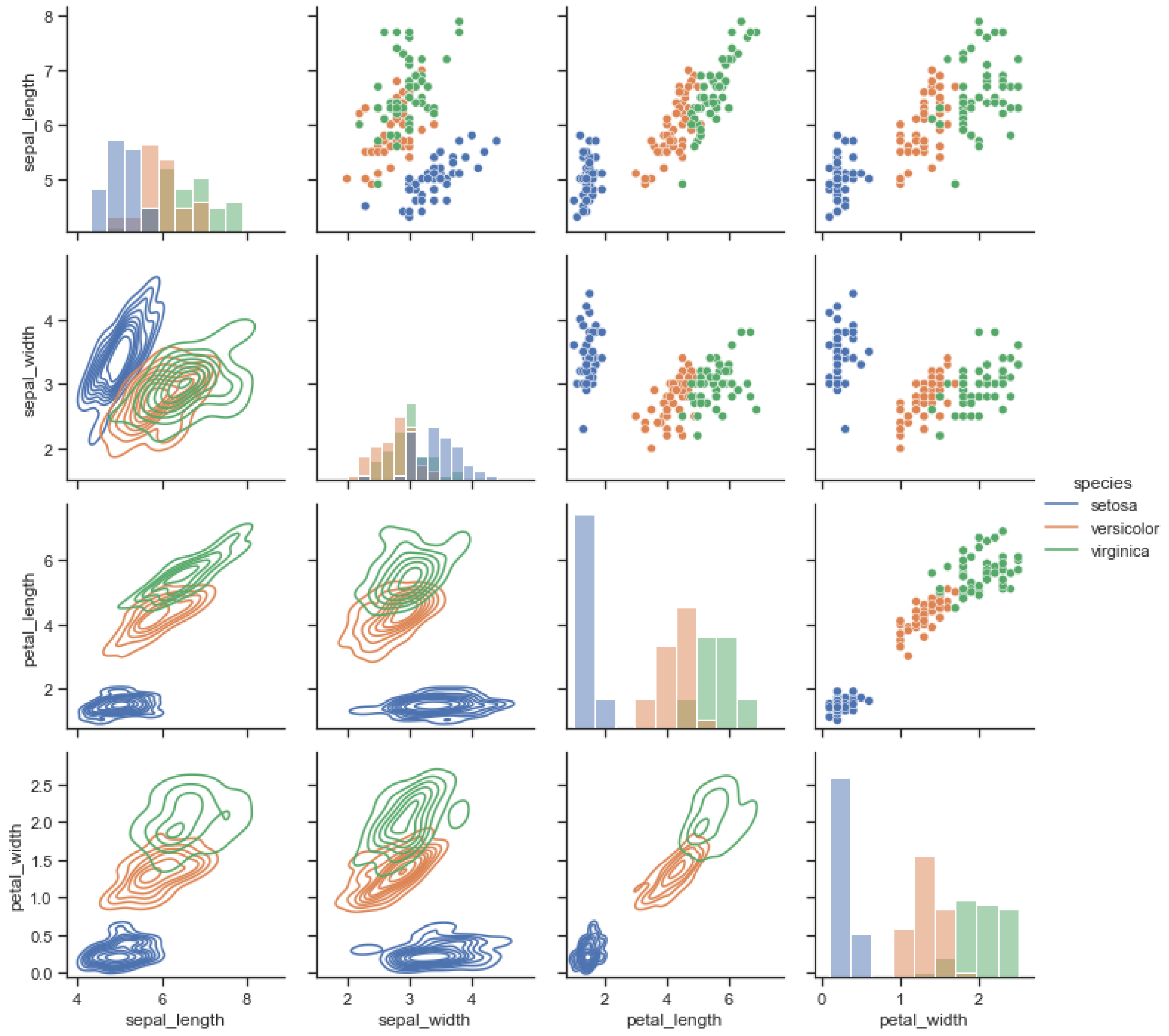
In [25]:
```
# Load the "iris" dataset
iris = sns.load_dataset("iris")

# Pair Plot
sns.set(style="ticks")
sns.pairplot(iris,diag_kind='hist', hue="species", markers=["o", "s", "D"])
plt.savefig('pair_plot.png')
plt.show()
```



Pair Grid

By using a pair grid you can customize the lower, upper, and diagonal plots individually.

In [26]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the "iris" dataset
iris = sns.load_dataset("iris")

# Create a Facet Grid of pairwise scatterplots
g = sns.PairGrid(iris, hue="species")
g.map_upper(sns.scatterplot)
g.map_diag(sns.histplot, kde_kws={"color": "k"})
g.map_lower(sns.kdeplot)
g.add_legend()
plt.savefig('pair_grid.png')
plt.show()
```



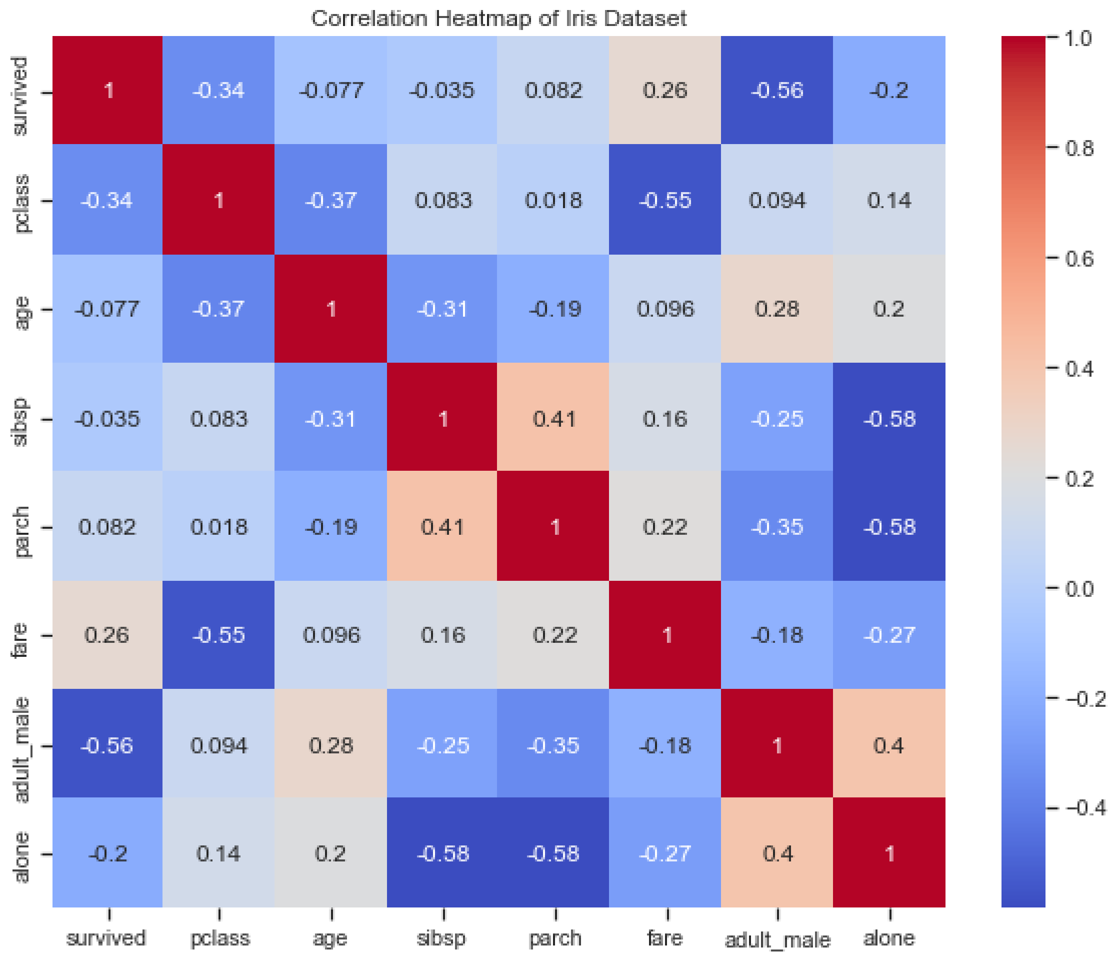
6. Matrix Plots

These plots visualize relationships within matrices or grids of data.

Heat Map

Use dataframe.corr() pandas data frame method to get the correlations, and then pass it to the sns.heatmap(corr_matrix) method to plot the heatmap. Let's see an example.

```
In [27]: # Sample data as a correlation matrix
         correlation_matrix = sns.load_dataset("titanic").corr()
         plt.figure(figsize=(10, 8))
         # Create a heatmap of the correlation matrix
         sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm")
         plt.title("Correlation Heatmap of Iris Dataset")
         plt.savefig('corr_matrix.png')
         plt.show()
```



	survived	pclass	age	sibsp	parch	fare	adult_male	alone
survived	1	-0.34	-0.077	-0.035	0.082	0.26	-0.56	-0.2
pclass	-0.34	1	-0.37	0.083	0.018	-0.55	0.094	0.14
age	-0.077	-0.37	1	-0.31	-0.19	0.096	0.28	0.2
sibsp	-0.035	0.083	-0.31	1	0.41	0.16	-0.25	-0.58
parch	0.082	0.018	-0.19	0.41	1	0.22	-0.35	-0.58
fare	0.26	-0.55	0.096	0.16	0.22	1	-0.18	-0.27
adult_male	-0.56	0.094	0.28	-0.25	-0.35	-0.18	1	0.4
alone	-0.2	0.14	0.2	-0.58	-0.58	-0.27	0.4	1

The main point from the heatmap is that if you find variables that are highly correlated, it's better to combine those columns as they both contribute the same.

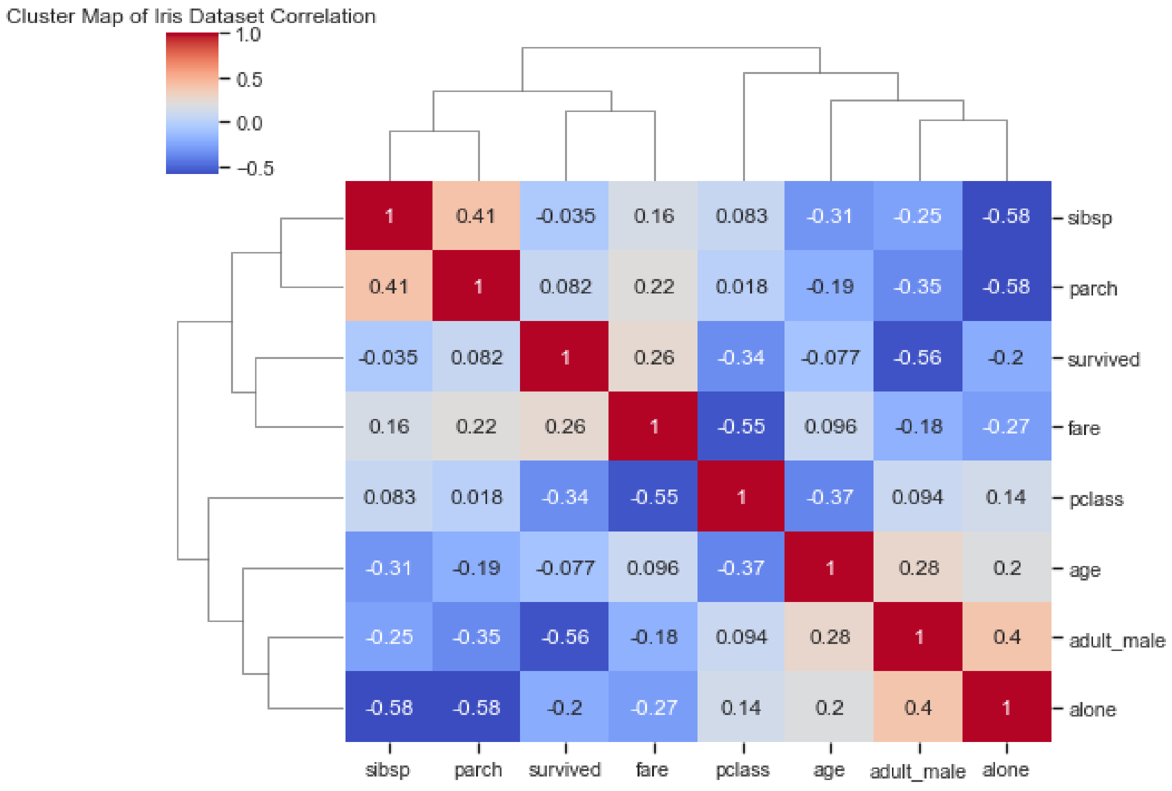
Cluster Map

Last but not least, The Cluster Map! Don't you think it will be easy to interpret if the like color squares are close to each other? And that's what exactly cluster map does. The primary purpose of a Cluster Map is to reveal clusters or groups of similar rows and columns.

A Cluster Map is a type of heatmap in Seaborn that not only displays data as a grid of colored cells but also arranges rows and columns in a way that groups similar data together, it uses hierarchical clustering algorithms to reorder rows and columns in the heatmap. The side of the grouped heatmap, it includes dendrograms to make it a cluster map. Dendrograms are tree-like diagrams that show the hierarchical relationships between rows and columns.

Use sns.clustermap(data.corr()) to create a cluster map.

In [28]:
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Load a sample dataset
data = sns.load_dataset("titanic")

# Create a Cluster Map of the correlation matrix
sns.clustermap(data.corr(), annot=True, cmap="coolwarm", figsize=(8, 6))
plt.title("Cluster Map of Iris Dataset Correlation")
plt.savefig('cluster_map.png')
plt.show()
```



	sibsp	parch	survived	fare	pclass	age	adult_male	alone
sibsp	1	0.41	-0.035	0.16	0.083	-0.31	-0.25	-0.58
parch	0.41	1	0.082	0.22	0.018	-0.19	-0.35	-0.58
survived	-0.035	0.082	1	0.26	-0.34	-0.077	-0.56	-0.2
fare	0.16	0.22	0.26	1	-0.55	0.096	-0.18	-0.27
pclass	0.083	0.018	-0.34	-0.55	1	-0.37	0.094	0.14
age	-0.31	-0.19	-0.077	0.096	-0.37	1	0.28	0.2
adult_male	-0.25	-0.35	-0.56	-0.18	0.094	0.28	1	0.4
alone	-0.58	-0.58	-0.2	-0.27	0.14	0.2	0.4	1

If you observe now, The rows are reordered based on how strongly they correlate with each other. Variables that are more strongly negatively correlated with each other are placed close together, creating clusters of related variables. Similarly, the Positively correlated ones. That’s what a cluster map does pointing clusters in a heatmap.

Conclusion

Now you can leverage this versatile library to create any kind of Categorical, Univariate, Bivariate, Multivariate, and Matrix plots, for beautiful data storytelling. I hope this journey through visualization possibilities has unveiled Seaborn’s remarkable versatility. Keep exploring and just play with it, seaborn is a versatile tool and you can do a lot more with it.

Visualization Wizardry: Let your Charts Talk Through Plotly!

In [1]:

```
# !pip install plotly
```

In [2]:

```
import plotly.express as px
import plotly.graph_objects as go

import pandas as pd
import numpy as np
%matplotlib inline
```

Table of Contents

2.Basic Plots

- Line Plots
- Bar Plots
- Scatter Plots
- Pie Charts
- Histograms

3.Advanced Plots

- Box Plot
- Violin Plot
- Density Heatmap
- 3D Plots
- Scatter Matrix
- Facet Grid
- Animated Plots

Important Plotly Modules

If you haven't used Plotly before, you are really missing out on visualization wizardry, Plotly shines as a powerful Python library that enables you to create interactive and visually appealing charts and graphs for data visualization with very low code.

Mastering Plotly can make you understand your data much more clearly and most importantly it can take your data storytelling to the next level through its interactiveness! In this article, we'll delve deep into the world of Plotly, exploring its most important spells (I mean methods), features, and capabilities, Assuming that you know the theory of these plots. Here's a glimpse of what lies ahead.

Plotly has various modules for different purposes, let's explore them.

1. **Plotly.express:** It is a high-level API for creating quick and easy visualizations. It is used for creating simple charts with minimal code, import plotly.express as px to use Plotly express. When using this, first you can pass the dataframe, and for x and y parameters you can just pass the column name, as it is compatable with pandas dataframes.
2. **plotly.graph_objects:** This module is part of Plotly's core functionality and is used for creating a wide range of charts and graphs, import plotly.graph_objects as go to use Plotly graph objects. When using this, we cannot pass the data, we can only pass the series for x and y parameters.
3. **plotly.offline:** In some cases, you might want to work with Plotly offline, especially if you're generating visualizations for offline use or within Jupyter notebooks.

Basic Plots

we can create all the basic plots with low code and amazing interactivity such as:

1. Hover information displays details when you mouse over the data points.
2. while zooming and panning enable closer examination of the plot.
3. You can select specific data series or data points by clicking on them in the legend or on the plot itself.
4. You can add custom buttons or controls to the plot that trigger specific actions or updates when clicked, and much more.

Line Plots

- **Using Plotly Express for Simple Plots**: Let's take a dataset available in Plotly to explore a few line plots.

In [3]:

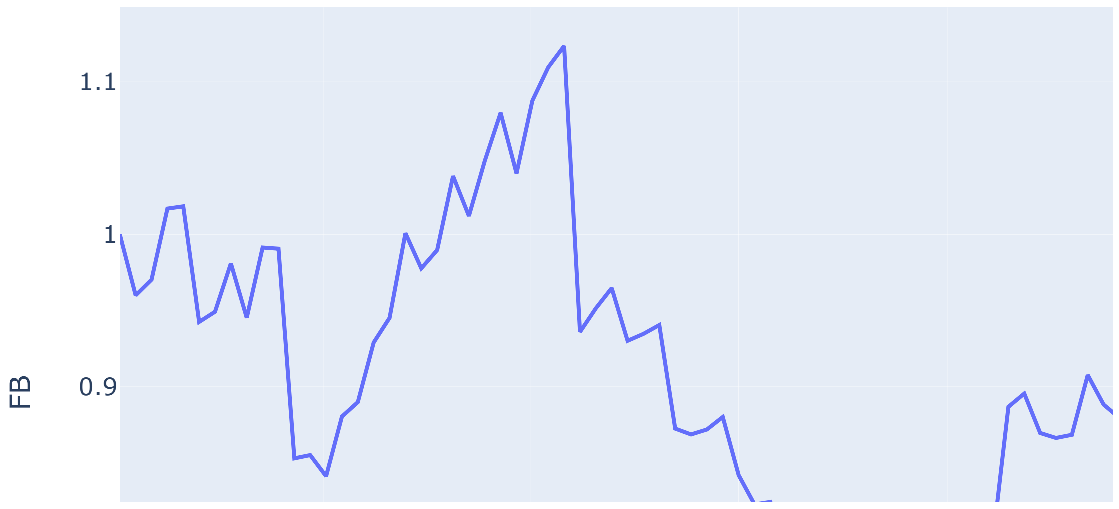
```
# Stock data available in the plotly express module
df_stocks = px.data.stocks()
df_stocks.head()
```

Out[3]:

```
         date      GOOG      AAPL      AMZN        FB      NFLX      MSFT
0  2018-01-01  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000
1  2018-01-08  1.018172  1.011943  1.061881  0.959968  1.053526  1.015988
2  2018-01-15  1.032008  1.019771  1.053240  0.970243  1.049860  1.020524
3  2018-01-22  1.066783  0.980057  1.140676  1.016858  1.307681  1.066561
4  2018-01-29  1.008773  0.917143  1.163374  1.018357  1.273537  1.040708
```

In [4]:

```
# Exploring Facebook stocks through an interactive line plot
px.line(df_stocks, x='date', y='FB')
```

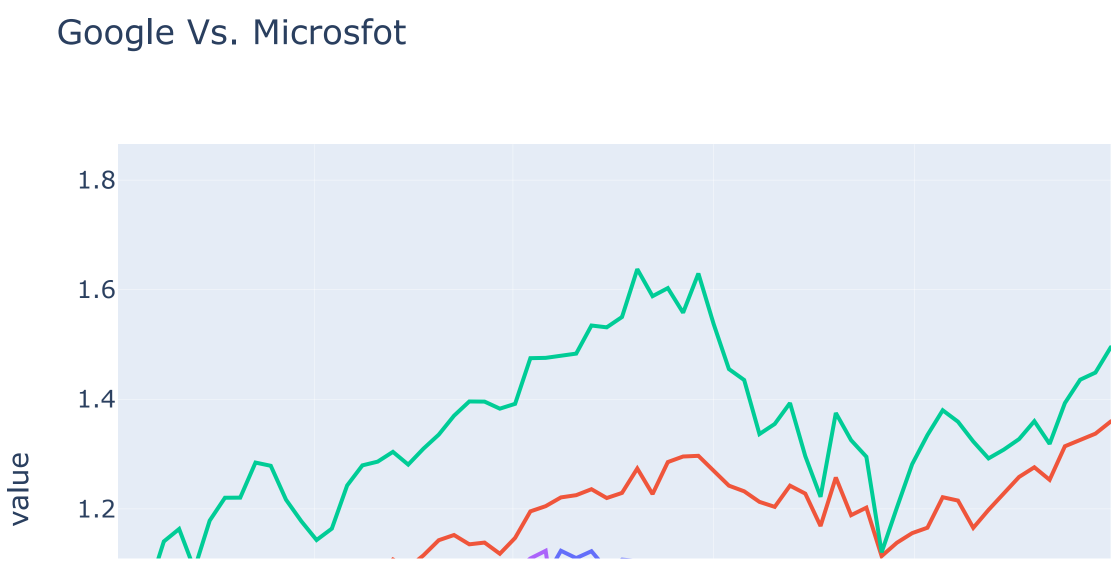


By running the single above code line, you will be able to generate such a beautifully interactive plot like this.
By running the single above code line, you will be able to generate such a beautifully interactive plot like this.

- **Generating Multiple Line plots by just providing a list**

In [5]:

```
# Generate multiple line plots by just providing a list of series names
px.line(df_stocks, x='date', y=['GOOG','MSFT','AMZN','FB'], title='Google Vs. Microsfo
t')
```



Now, if you only want to see two or three, you can disable the rest by just clicking on the label name on the right side, this is so cool, right?

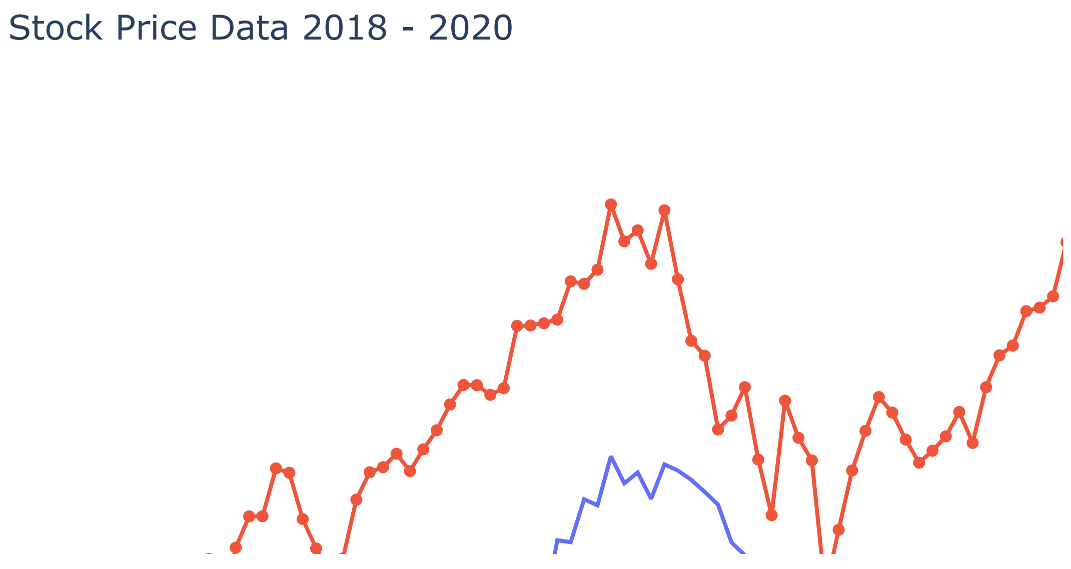
- **Using Graph Objects for further customizations**

To achieve these kinds of things, we need to use the graph_objects, which is why it was said that we need the graph_objects module for more complex charts and plotly_express for simpler charts.

1. Create a figure object using fig=go.Figure().
2. To add a plot to the figure use fig.add_trace()and pass the graph object to the figure object.
3. Use go.scatter(x,y,mode)to create graph objects for scatter plots and line plots.
4. And here we need to use the mode parameter to specify the plot style, by default it will be assigned to lines which will give a line plot. And for the Facebook, as we gave mode='lines+markers' it will have the lines along with markers for the points, easy styling!
5. For line styling, use theline parameter and assign a dictionary with various stylings you need.

In [6]:

```
# Create a figure object for which we can later add the plots
fig = go.Figure()

# pass the graph objects to the add trace method, assign a series to x and y parameters
of graph objects.
fig.add_trace(go.Scatter(x=df_stocks.date, y=df_stocks.AAPL, mode='lines', name='Appl
e'))
fig.add_trace(go.Scatter(x=df_stocks.date, y=df_stocks.AMZN, mode='lines+markers', name
='Amazon'))

# Customizing a particular line
fig.add_trace(go.Scatter(x=df_stocks.date, y=df_stocks.GOOG,
                        mode='lines+markers', name='Google',
                        line=dict(color='darkgreen', dash='dot')))

# Further style the figure
fig.update_layout(title='Stock Price Data 2018 - 2020',
                    # customize axis by using xaxis/yaxis '_' and style
                    xaxis_title='Price',
                    # customize various styles axis by passing a dictionary
                    yaxis=dict(
                        showgrid=False,
                        zeroline=False,
                        showline=False,
                        showticklabels=False,
                    ),
                    # customize entire plot style
                    plot_bgcolor='white')
```



Bar Charts

Using Plotly Express for simple Bar Plots:

Use px.bar(data,x,y) to create your bar chart. You can add an extra dimension to the bar plot by simply using the parameter, in that case you will get the stacked bar plot, if you want it side by side, you can change it with the parameter barmode='group' . The below snippets are just images.

In [8]:

```
# Tips data avaialbe in plotly module
tips = px.data.tips()
tips.head()
```

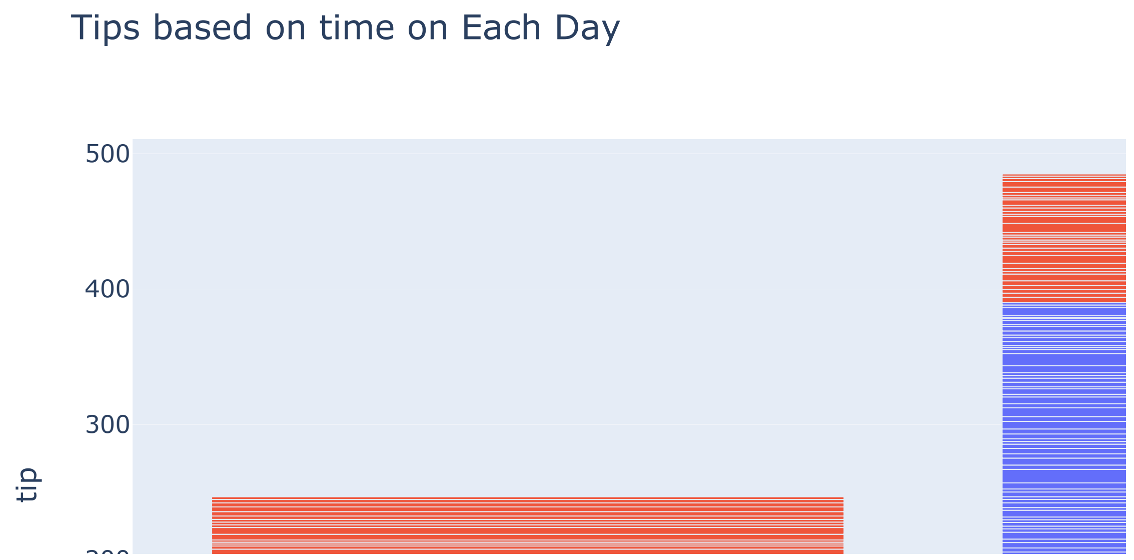
Out[8]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Using Plotly Express for Simple Bar plots

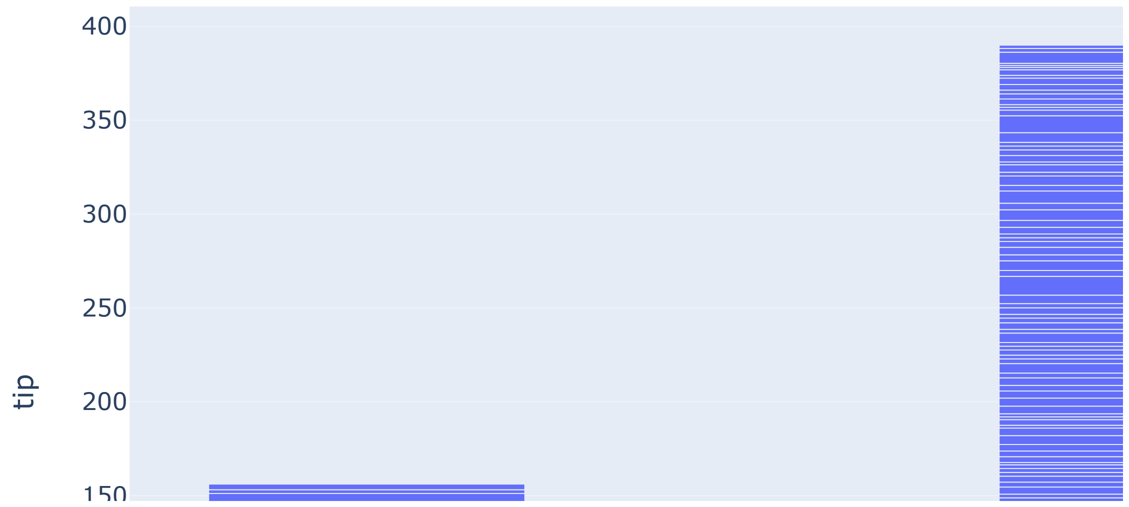
In [11]:

```
# Stacked bar plot
px.bar(tips, x='sex', y='tip', color='time', title='Tips based on time on Each Day')
```



In [13]:

```
# Place bars next to each other
px.bar(tips, x="sex", y="tip", color='time', barmode='group')
```



customized Bar plots

In [14]:

```
# Filtering data for countries in 2007 greater than 50000000
population = px.data.gapminder()
filtered_data = population[(population.year==2007) & (population['pop']>50000000)]
filtered_data.head()
```

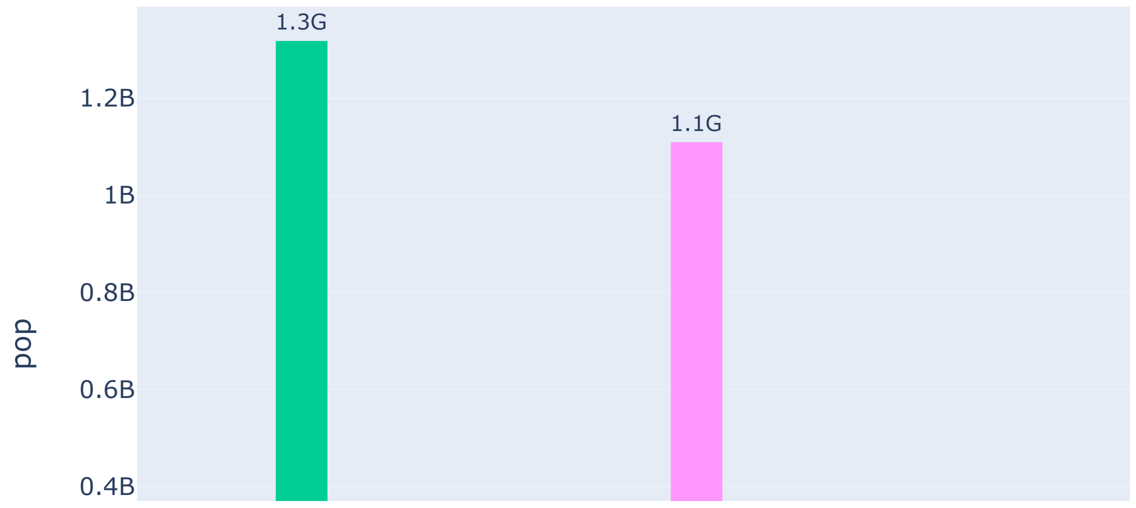
Out[14]:

	country	continent	year	lifeExp	pop	gdpPercap	iso_alpha	iso_num
107	Bangladesh	Asia	2007	64.062	150448339	1391.253792	BGD	50
179	Brazil	Americas	2007	72.390	190010647	9065.800825	BRA	76
299	China	Asia	2007	72.961	1318683096	4959.114854	CHN	156
335	Congo, Dem. Rep.	Africa	2007	46.462	64606759	277.551859	COD	180
467	Egypt	Africa	2007	71.338	80264543	5581.180998	EGY	818

Instead of directly printing the plotly express object, if we save to figure, then we can make more customizations using the update_trace and update_layout methods of the figure.

In [15]:

```
# Using colors to distinguish a dimension
fig = px.bar(filtered_data, y='pop', x='country', text='pop', color='country')

# Put bar total value above bars with 2 values of precision
fig.update_traces(texttemplate='%{text:.2s}', textposition='outside')

# Rotate labels 45 degrees
fig.update_layout(xaxis_tickangle=-45)
```



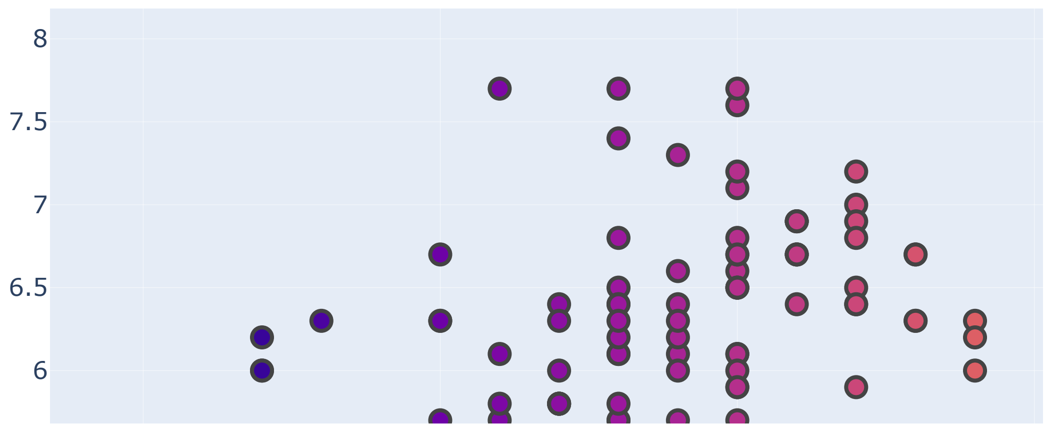
If we hadn't given the color dimension to be country, all of the bars would have the same color. And by using text parameter we were able to set the total population at the top of the bars.

Scatter Plot

Use px.scatter(data,x,y) to create simple scatter plots. You can add more dimension using color and size parameters. By using the graph object, use go.scatter(data,x,y,mode='markers') to create a scatter plot, and we can even add a numeric variable as a dimension with color a parameter.

In [16]:

```
# Use included Iris data set
df_iris = px.data.iris()

# Create a customized scatter
fig = go.Figure()
fig.add_trace(go.Scatter(
    x=df_iris.sepal_width, y=df_iris.sepal_length,
    mode='markers',
    marker_color=df_iris.sepal_width,
    text=df_iris.species,
    marker_showscale=True
))
fig.update_traces(marker_line_width=2, marker_size=10)
```



Pie Charts

Use px.Pie(data,values=series_name) to create a pie chart. For further customizations, you can use graph object pie chart with go.Pie() and we use the update_traces of the figure object to customize the hover info, text, and pull amount (which is the same as explode, if you are familiar with seaborn).

In [17]:

```
population.head()
```

Out[17]:

	country	continent	year	lifeExp	pop	gdpPercap	iso_alpha	iso_num
0	Afghanistan	Asia	1952	28.801	8425333	779.445314	AFG	4
1	Afghanistan	Asia	1957	30.332	9240934	820.853030	AFG	4
2	Afghanistan	Asia	1962	31.997	10267083	853.100710	AFG	4
3	Afghanistan	Asia	1967	34.020	11537966	836.197138	AFG	4
4	Afghanistan	Asia	1972	36.088	13079460	739.981106	AFG	4

In [18]:

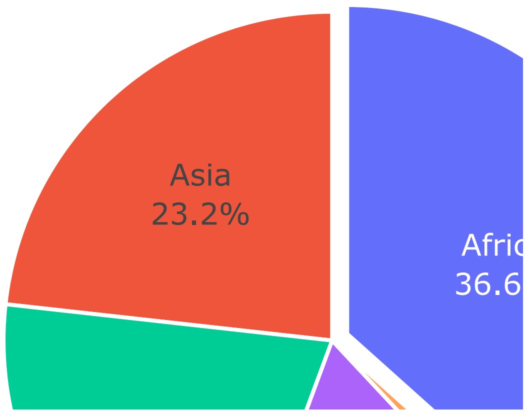
```
population.continent.value_counts()
```

Out[18]:

```
Africa      624
Asia        396
Europe      360
Americas    300
Oceania      24
Name: continent, dtype: int64
```

In [19]:

```
# Customize pie chart
fig = go.Figure(go.Pie(labels=population.continent.value_counts().index,
                       values=population.continent.value_counts()))

# use update traces to customise the hover info, text, pull amount for each pie slice,
and stroke
fig.update_traces(hoverinfo='label+percent', textfont_size=15,
                  textinfo='label+percent', pull=[0.05, 0, 0, 0, 0],
                  marker_line=dict(color='#FFFFFF', width=2))
```

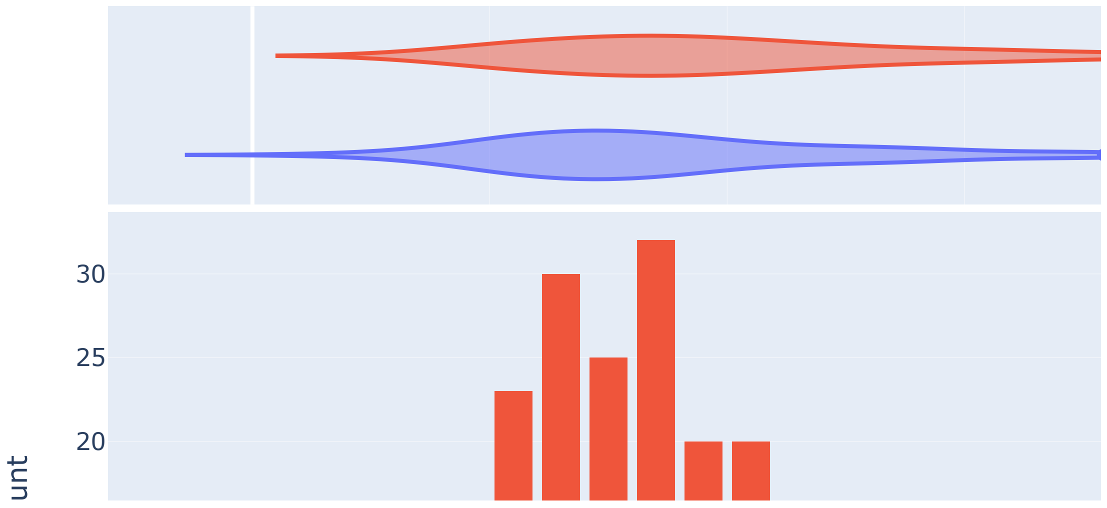


Histograms

Use px.histogram(values) to create a histogram, and you can stack another histogram on top of that using thecolor parameter. And here is an amazing with histogram plots, you can use marginal parameter to add a layer of another plot on the top, such as box, violin, or rug. Here's an example of adding a violin plot on top of the stack bar chart.

In [20]:

```
# Stack histograms based on different column data
df_tips = px.data.tips()
fig = px.histogram(df_tips, x="total_bill", color="sex", marginal='violin')
fig.update_layout(bargap=0.2)
```



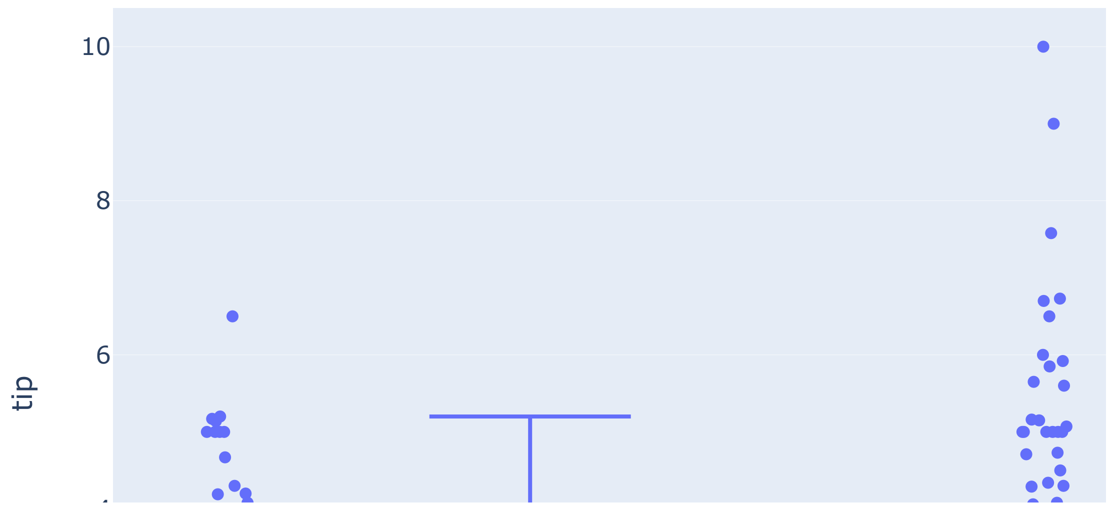
- You can hover over the bar charts to see more details and interact with them.
- Hover on the violin plot to see the quartile details.
- And you can click on the side labels to enable or disable them, so cool, huh!

Box Plots

Use px.box(data,x,y) to create the box plot, you can use the color parameter to add another dimension for the box plot. And using the points='all'parameter will show all the points scattered to the left. On hover, you can see the quartile values.

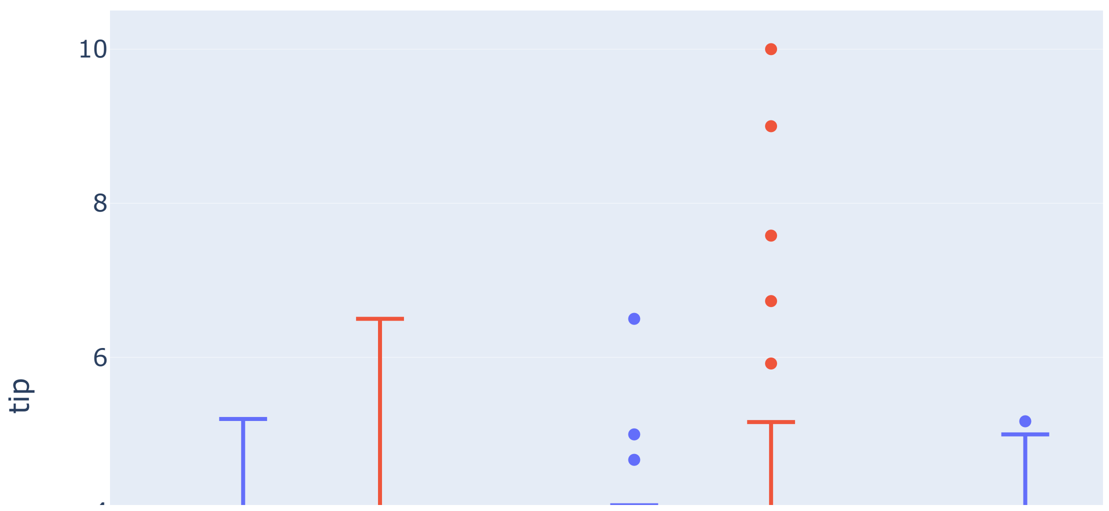
In [21]:

```
df_tips = px.data.tips()
# We can see which sex tips the most, points displays all the data points
px.box(df_tips, x='sex', y='tip', points='all')
```



In [22]:

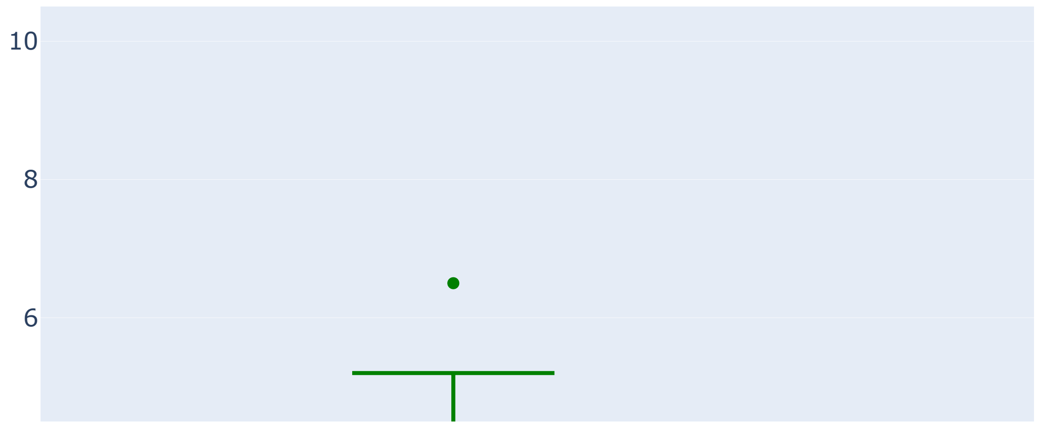
```
# Display tip sex data by day
px.box(df_tips, x='day', y='tip', color='sex')
```



You can do further customizations using go.Box(x,y), Using that you can also add mean and standard deviation with the parameter boxmean='sd' . Remember when using graph object, we won't be passing the data, but just the series. Points parameters is not used in the graph object box method.

In [23]:

```
# Adding standard deviation and mean
fig = go.Figure()
fig.add_trace(go.Box(x=df_tips.sex, y=df_tips.tip, marker_color='green',boxmean='sd'))
```

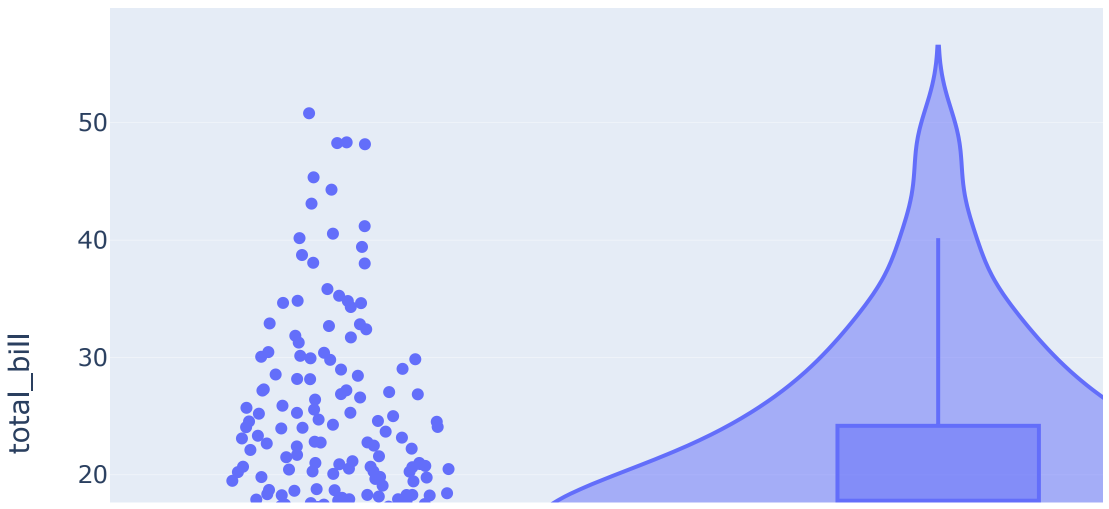


Violin Plots

Use px.violin(data,x,y) to create violin plots. By just giving y parameters you can create a violin plot for one numeric variable. But if you want to create violin plots for a numeric variable based on a category column, then you can specify the category column in the x parameter. And additionaly you can add one more category column dimension using the color parameter. Use box=True parameter to show the box in the plot.

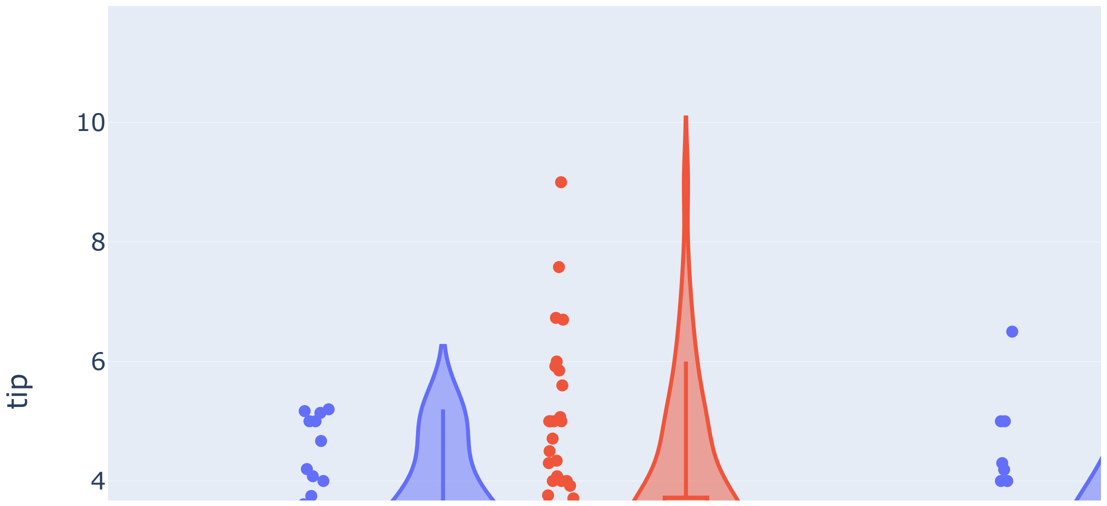
In [24]:

```
# Violin plot for the total bill
df_tips = px.data.tips()
px.violin(df_tips, y="total_bill", box=True, points='all')
```



In [25]:

```
# Multiple plots
px.violin(df_tips, y="tip", x="smoker", color="sex", box=True, points="all",
          hover_data=df_tips.columns)
```

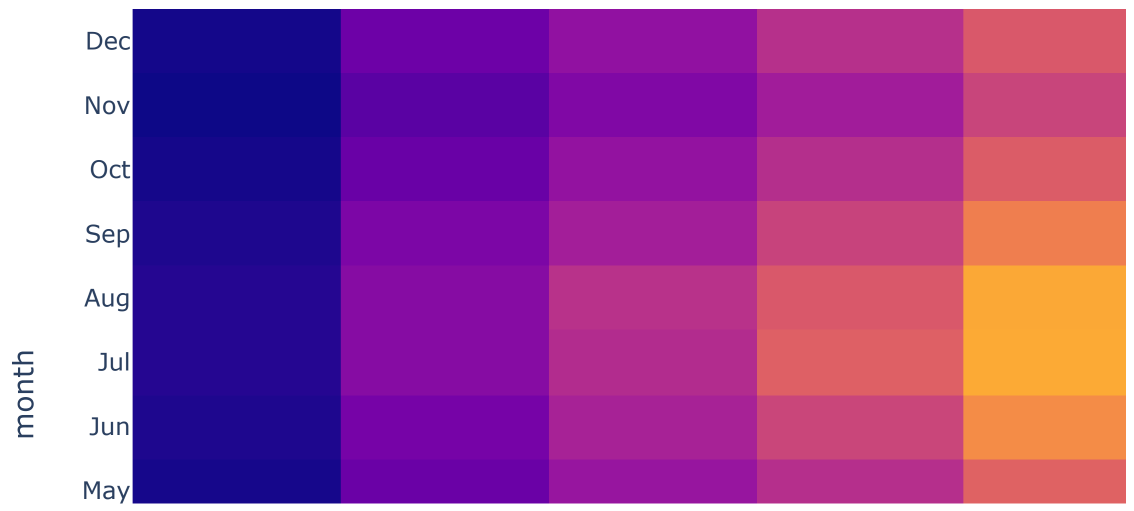


Density Heatmaps

If you need to do an analysis for three variables, you can easily opt for the density heatmaps. Use px.density_heatmap(data,x,y,z) to create a density heat map.

In [28]:

```
# Create a heatmap using Seaborn data
import seaborn as sns
flights = sns.load_dataset("flights")
px.density_heatmap(flights, x='year', y='month', z='passengers')
```



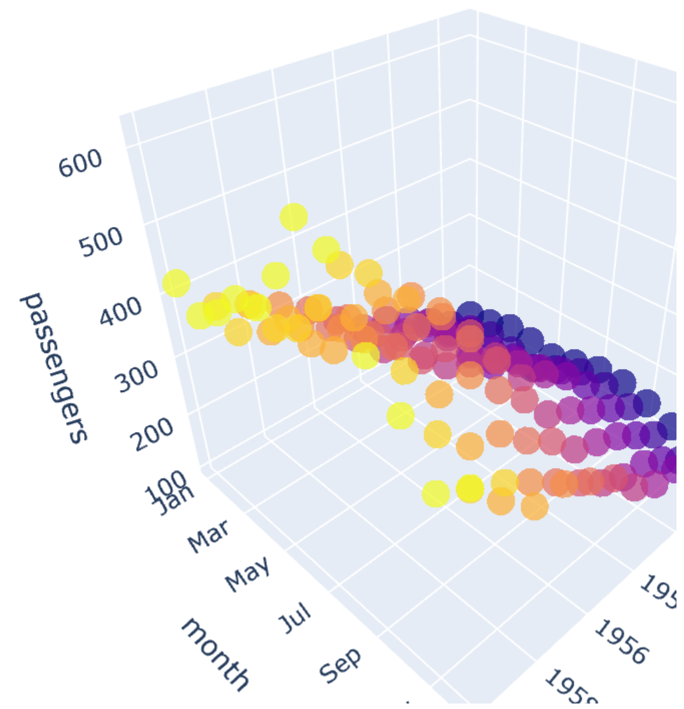
3D Plots

3D Scatter plot

To create a 3D Scatter plot, you can use px.scatter_3d(data,x,y,z,color)

In [29]:

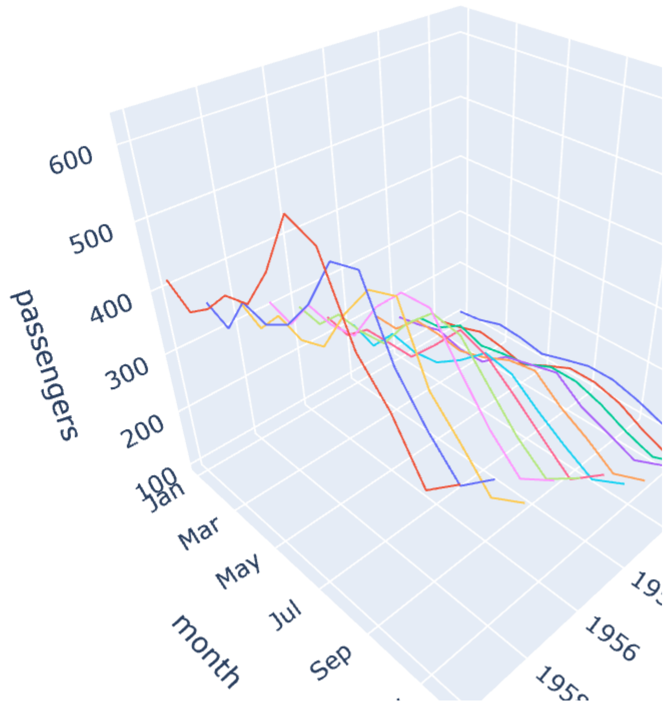
```
# Create a 3D scatter plot using flight data
px.scatter_3d(flights, x='year', y='month', z='passengers', color='year', opacity=0.7)
```



Similarly, you can also create a 3D Line plot using px.line_3d(data,x,y,z,color)

In [30]:

```
px.line_3d(flights, x='year', y='month', z='passengers', color='year')
```

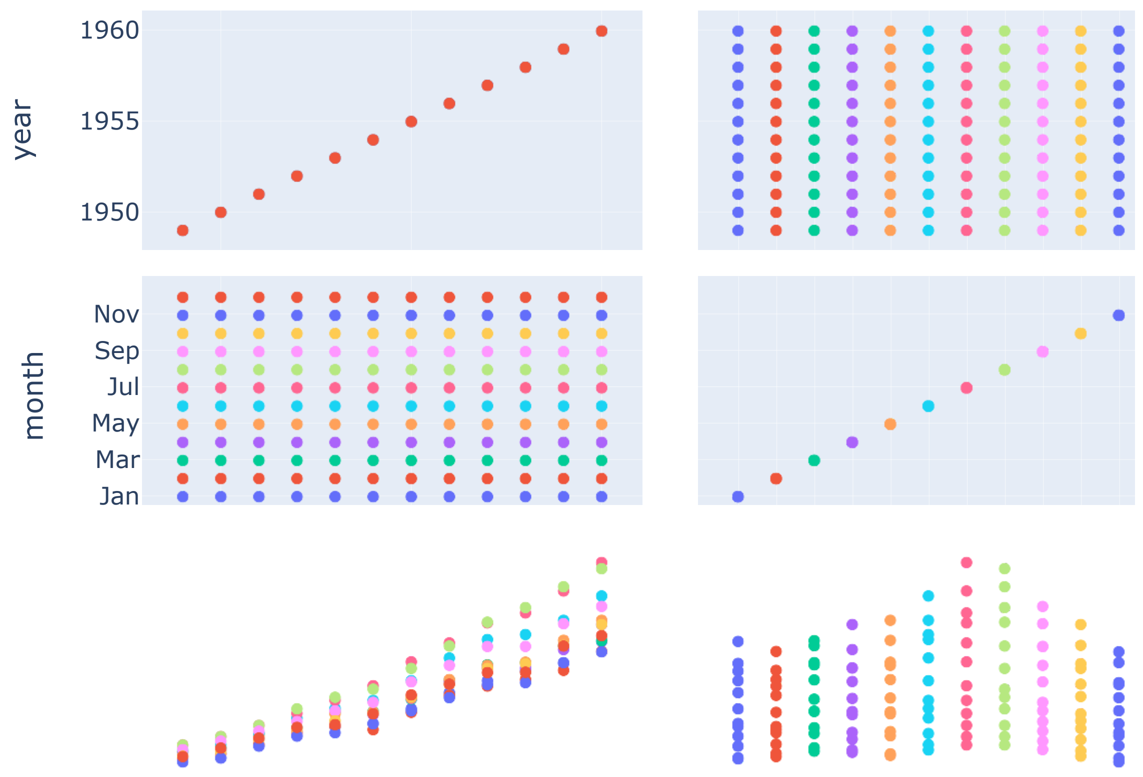


Scatter Matrix

Scatter matrix is similar to the paiplot from sns, which gives an exhaustive idea of the numeric variables. Use px.scatter_matrix(data) to display scatter matrix. To differentiate further based on any category, as always you an use the color parameter.

In [31]:

```
px.scatter_matrix(flights, color='month')
```

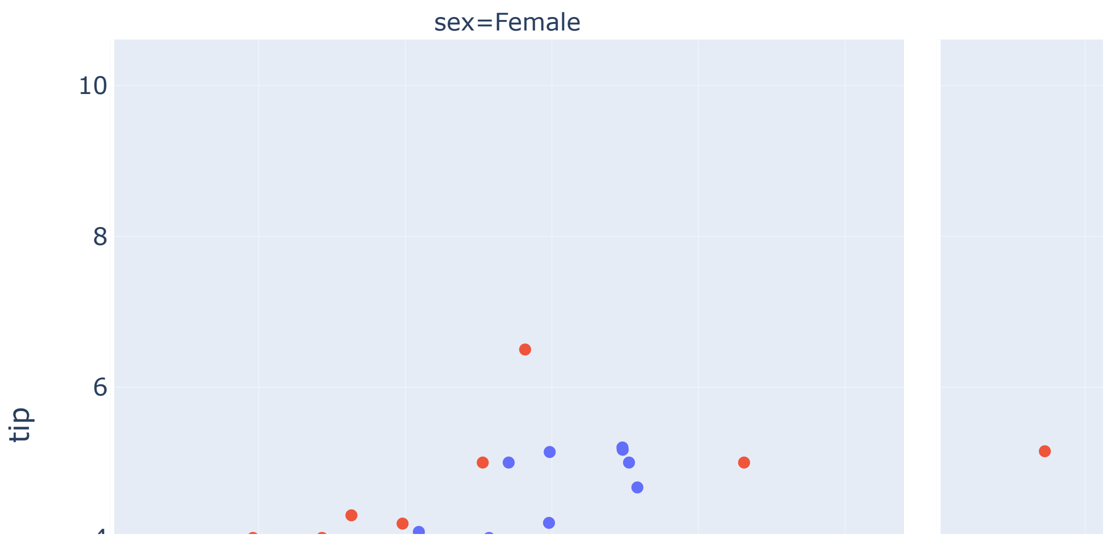


Facet Grids

For all the plotly express plots we have seen so far, we can add the facet grids, by using the parameters face_col and face_row parameters.

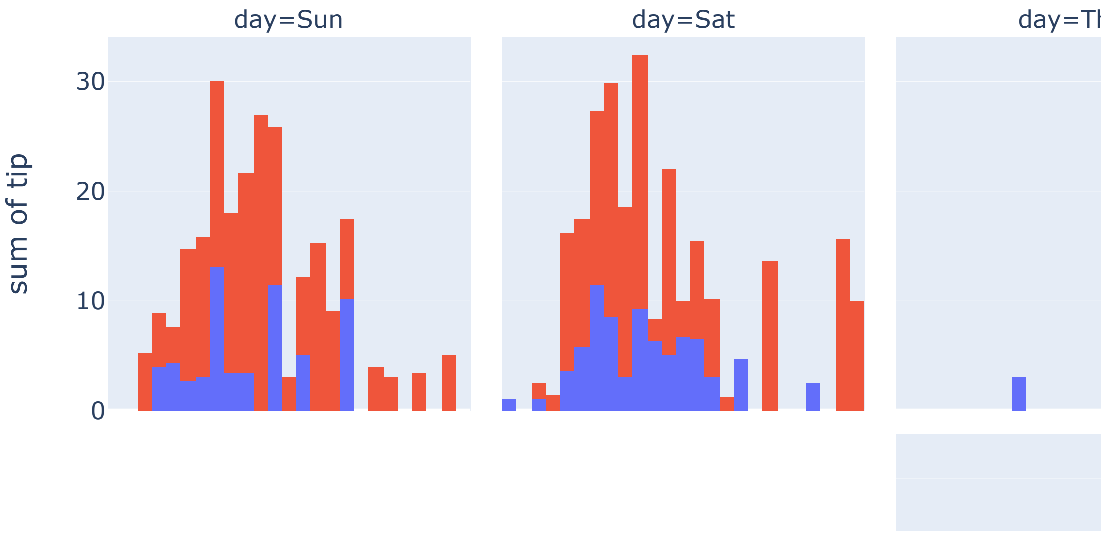
In [32]:

```
df_tips = px.data.tips()
px.scatter(df_tips, x="total_bill", y="tip", color="smoker", facet_col="sex")
```



In [33]:

```python
# We can line up data in rows and columns
px.histogram(df_tips, x="total_bill", y="tip", color="sex", facet_row="time", facet_col
="day")
```

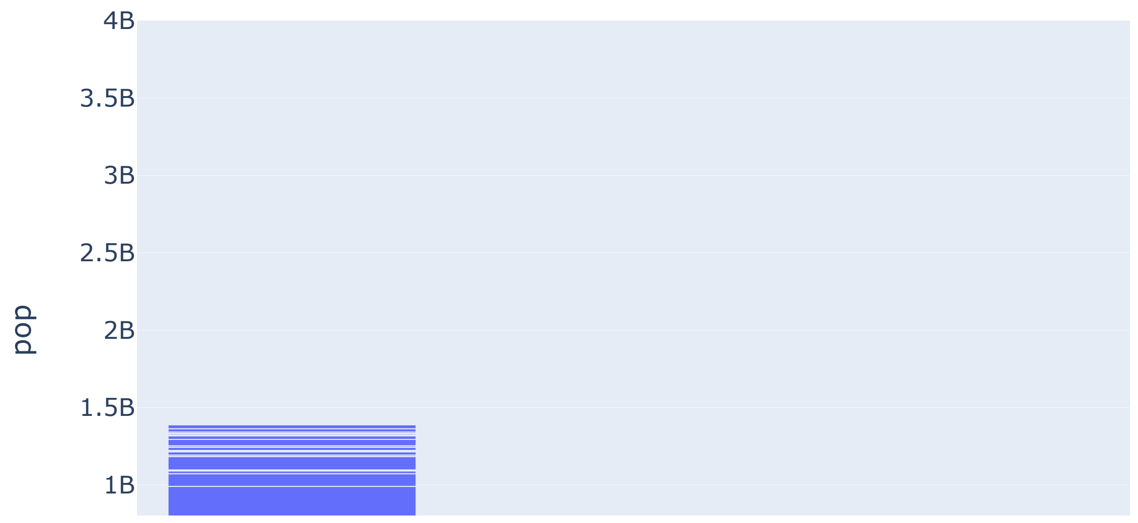


Animated Plots

You can add animation to your plotly express plots. The animation_frame parameter specifies which variable in your dataset should be used to create animation frames. The animation_group parameter is used to group data points within each animation frame. For example, if you're visualizing the movement of objects over time, you might use animation_frame to specify the time steps and animation_group to group objects by their IDs or labels.

In [34]:

```
# Watch as bars chart population changes
population = px.data.gapminder()
px.bar(population, x="continent", y="pop", color="continent",
  animation_frame="year", range_y=[0,4000000000])
```



Conclusion

I guess now you would agree how powerful yet simple plotly is, With Plotly, you're not limited to static graphs but invited to weave stories, paint insights, and orchestrate data's symphony. From the elegance of scatter plots to the drama of animated charts, Plotly lets you be the virtuoso of your data narrative. So, embrace Plotly, and let your data dance, and captivate your audience. It's not just a tool; it's your brush to paint the future of data visualization. Unleash your creativity with plotly, Happy Learning

Loan Defaulter Dataset

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Load Dataset

In [2]:
```
# application data
data = pd.read_csv('\loan_data\application_data.csv')

# previous applicaiton data
pdata = pd.read_csv('\loan_data\previous_application.csv')
```

Inspecting application data

In [3]:
```
data.shape
```

Out[3]:
```
(307511, 122)
```

122 columns is a lot!! Better start checking the null values.

In [4]:
```
data.isna().sum().sort_values(ascending=False)
```

Out[4]:
```
COMMONAREA_MEDI                 214865
COMMONAREA_AVG                  214865
COMMONAREA_MODE                 214865
NONLIVINGAPARTMENTS_MODE        213514
NONLIVINGAPARTMENTS_MEDI        213514
                                 ...  
REG_CITY_NOT_LIVE_CITY               0
LIVE_REGION_NOT_WORK_REGION          0
REG_REGION_NOT_WORK_REGION           0
HOUR_APPR_PROCESS_START              0
SK_ID_CURR                           0
Length: 122, dtype: int64
```

With this many columns, we can clearly see the null values if we convert it to dataframe. Will be useful further.

Handling Missing Values

In [5]:
```
missing = pd.DataFrame(data.isna().sum().sort_values(ascending=False))
missing
```

Out[5]:

	0
COMMONAREA_MEDI	214865
COMMONAREA_AVG	214865
COMMONAREA_MODE	214865
NONLIVINGAPARTMENTS_MODE	213514
NONLIVINGAPARTMENTS_MEDI	213514
...	...
REG_CITY_NOT_LIVE_CITY	0
LIVE_REGION_NOT_WORK_REGION	0
REG_REGION_NOT_WORK_REGION	0
HOUR_APPR_PROCESS_START	0
SK_ID_CURR	0

122 rows × 1 columns

We can see that the index is of col name, so let's reset the index

In [6]:
```
missing.reset_index(inplace=True)
```

In [7]:
```
missing.head()
```

Out[7]:

	index	0
0	COMMONAREA_MEDI	214865
1	COMMONAREA_AVG	214865
2	COMMONAREA_MODE	214865
3	NONLIVINGAPARTMENTS_MODE	213514
4	NONLIVINGAPARTMENTS_MEDI	213514

Now that we have our missing values dataframe, we can start dealing with them.

But before that we need a percentage column to analyze the missing values more accurately

In [8]:
```
missing.rename(columns={'index':'column',0:'null_count'},inplace=True)
missing['percent'] = missing['null_count']/data.shape[0]
```

In [9]:
```
missing.head()
```

Out[9]:

	column	null_count	percent
0	COMMONAREA_MEDI	214865	0.698723
1	COMMONAREA_AVG	214865	0.698723
2	COMMONAREA_MODE	214865	0.698723
3	NONLIVINGAPARTMENTS_MODE	213514	0.694330
4	NONLIVINGAPARTMENTS_MEDI	213514	0.694330

In [10]:
```
missing[missing.percent>0.4].shape[0]
```

Out[10]:
```
49
```

So we see that there are 49 columns with atleast 40 percent of data is missing!! I believe this kind of data will not make much sense even by imputation, so decided to remove those columns.

In [11]:
```
data.drop(missing[missing.percent>0.4]['column'].values,axis=1,inplace=True)
```

In [12]:
```
data.shape
```

Out[12]:
```
(307511, 73)
```

So previously we had 122 columns, and now we removed 40 columns as they atleast 50 percent of data missing, so we are left out with 81 columns.

```
In [13]: data.columns
```

```
Out[13]: Index(['SK_ID_CURR', 'TARGET', 'NAME_CONTRACT_TYPE', 'CODE_GENDER',
                'FLAG_OWN_CAR', 'FLAG_OWN_REALTY', 'CNT_CHILDREN', 'AMT_INCOME_TOTAL',
                'AMT_CREDIT', 'AMT_ANNUITY', 'AMT_GOODS_PRICE', 'NAME_TYPE_SUITE',
                'NAME_INCOME_TYPE', 'NAME_EDUCATION_TYPE', 'NAME_FAMILY_STATUS',
                'NAME_HOUSING_TYPE', 'REGION_POPULATION_RELATIVE', 'DAYS_BIRTH',
                'DAYS_EMPLOYED', 'DAYS_REGISTRATION', 'DAYS_ID_PUBLISH', 'FLAG_MOBIL',
                'FLAG_EMP_PHONE', 'FLAG_WORK_PHONE', 'FLAG_CONT_MOBILE', 'FLAG_PHONE',
                'FLAG_EMAIL', 'OCCUPATION_TYPE', 'CNT_FAM_MEMBERS',
                'REGION_RATING_CLIENT', 'REGION_RATING_CLIENT_W_CITY',
                'WEEKDAY_APPR_PROCESS_START', 'HOUR_APPR_PROCESS_START',
                'REG_REGION_NOT_LIVE_REGION', 'REG_REGION_NOT_WORK_REGION',
                'LIVE_REGION_NOT_WORK_REGION', 'REG_CITY_NOT_LIVE_CITY',
                'REG_CITY_NOT_WORK_CITY', 'LIVE_CITY_NOT_WORK_CITY',
                'ORGANIZATION_TYPE', 'EXT_SOURCE_2', 'EXT_SOURCE_3',
                'OBS_30_CNT_SOCIAL_CIRCLE', 'DEF_30_CNT_SOCIAL_CIRCLE',
                'OBS_60_CNT_SOCIAL_CIRCLE', 'DEF_60_CNT_SOCIAL_CIRCLE',
                'DAYS_LAST_PHONE_CHANGE', 'FLAG_DOCUMENT_2', 'FLAG_DOCUMENT_3',
                'FLAG_DOCUMENT_4', 'FLAG_DOCUMENT_5', 'FLAG_DOCUMENT_6',
                'FLAG_DOCUMENT_7', 'FLAG_DOCUMENT_8', 'FLAG_DOCUMENT_9',
                'FLAG_DOCUMENT_10', 'FLAG_DOCUMENT_11', 'FLAG_DOCUMENT_12',
                'FLAG_DOCUMENT_13', 'FLAG_DOCUMENT_14', 'FLAG_DOCUMENT_15',
                'FLAG_DOCUMENT_16', 'FLAG_DOCUMENT_17', 'FLAG_DOCUMENT_18',
                'FLAG_DOCUMENT_19', 'FLAG_DOCUMENT_20', 'FLAG_DOCUMENT_21',
                'AMT_REQ_CREDIT_BUREAU_HOUR', 'AMT_REQ_CREDIT_BUREAU_DAY',
                'AMT_REQ_CREDIT_BUREAU_WEEK', 'AMT_REQ_CREDIT_BUREAU_MON',
                'AMT_REQ_CREDIT_BUREAU_QRT', 'AMT_REQ_CREDIT_BUREAU_YEAR'],
               dtype='object')
```

Investigating the columns

At first glance we can see that there are some columns with that starts with FLAG, so we can investigate them first

```
In [14]: cols_with_flag = data.columns[data.columns.str.startswith('FLAG')]
```

```
In [15]: flag_cols_data = data[np.concatenate([cols_with_flag,np.array(['TARGET'])])]
```

```
In [16]: cols_with_flag
```

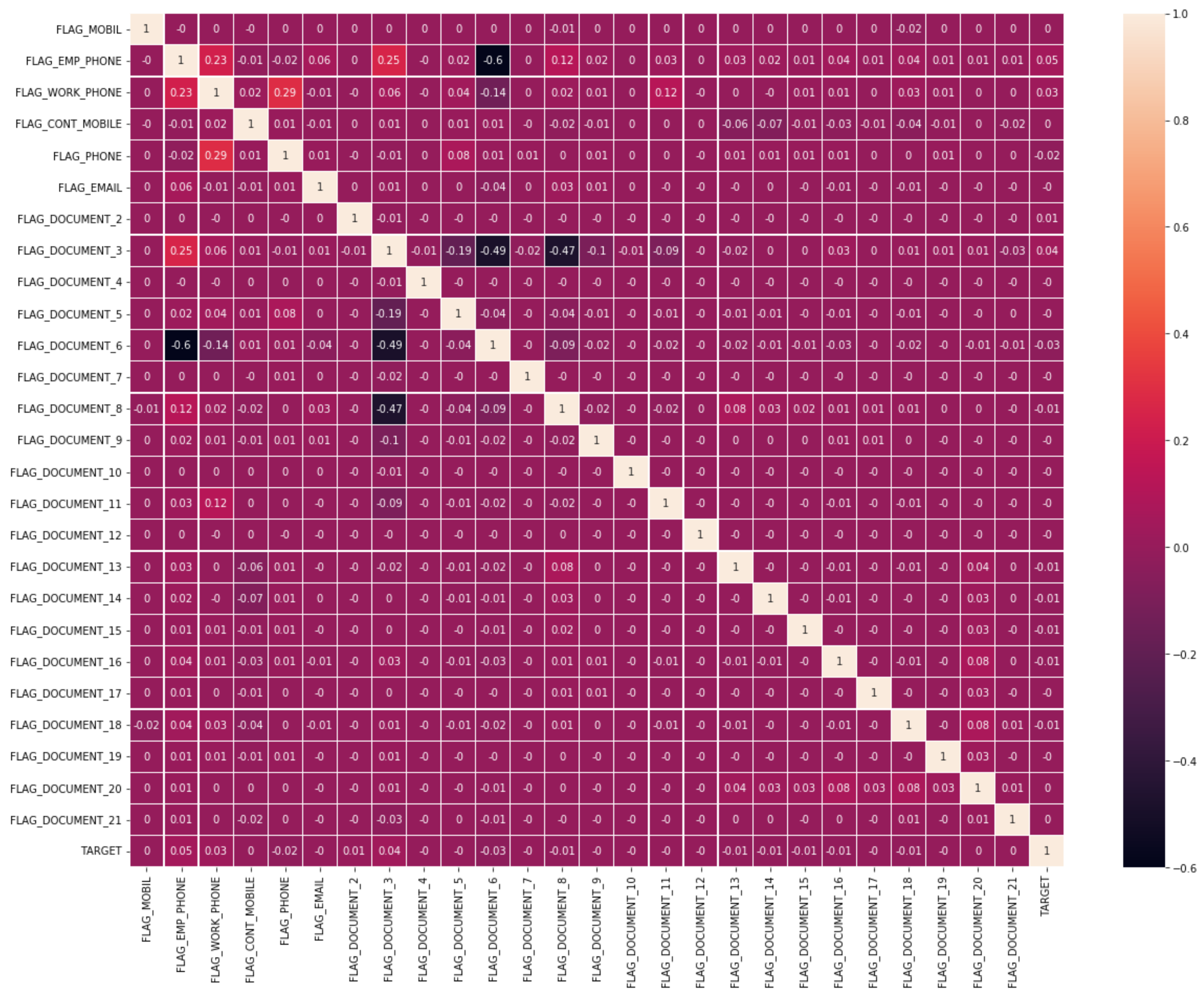
```
Out[16]: Index(['FLAG_OWN_CAR', 'FLAG_OWN_REALTY', 'FLAG_MOBIL', 'FLAG_EMP_PHONE',
                'FLAG_WORK_PHONE', 'FLAG_CONT_MOBILE', 'FLAG_PHONE', 'FLAG_EMAIL',
                'FLAG_DOCUMENT_2', 'FLAG_DOCUMENT_3', 'FLAG_DOCUMENT_4',
                'FLAG_DOCUMENT_5', 'FLAG_DOCUMENT_6', 'FLAG_DOCUMENT_7',
                'FLAG_DOCUMENT_8', 'FLAG_DOCUMENT_9', 'FLAG_DOCUMENT_10',
                'FLAG_DOCUMENT_11', 'FLAG_DOCUMENT_12', 'FLAG_DOCUMENT_13',
                'FLAG_DOCUMENT_14', 'FLAG_DOCUMENT_15', 'FLAG_DOCUMENT_16',
                'FLAG_DOCUMENT_17', 'FLAG_DOCUMENT_18', 'FLAG_DOCUMENT_19',
                'FLAG_DOCUMENT_20', 'FLAG_DOCUMENT_21'],
               dtype='object')
```

Visualizing how each FLAG Column impact the target

Observing the correlation for these columns

```
In [17]: plt.figure(figsize=(20,15))
         corr_matrix = round(flag_cols_data.corr(),2)
         sns.heatmap(corr_matrix,linewidth=0.2,annot=True)
```

```
Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x23889e48ac8>
```



As we can see that they have very less correlation with the target, which is totally insignificant, hence we can remove them.

```
In [18]: data.drop(cols_with_flag,axis=1,inplace=True)
```

```
In [19]: data.shape
```

```
Out[19]: (307511, 45)
```

Checking if any other columns can be removed

```
In [20]: data.head()
```

Out[20]:

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	CODE_GENDER	CNT_CHILDREN	AMT_INCOME_TOTAL
0	100002	1	Cash loans	M	0	202500.0
1	100003	0	Cash loans	F	0	270000.0
2	100004	0	Revolving loans	M	0	67500.0
3	100006	0	Cash loans	F	0	135000.0
4	100007	0	Cash loans	M	0	121500.0

5 rows × 45 columns

```
In [21]: data.columns
```

```
Out[21]: Index(['SK_ID_CURR', 'TARGET', 'NAME_CONTRACT_TYPE', 'CODE_GENDER',
       'CNT_CHILDREN', 'AMT_INCOME_TOTAL', 'AMT_CREDIT', 'AMT_ANNUITY',
       'AMT_GOODS_PRICE', 'NAME_TYPE_SUITE', 'NAME_INCOME_TYPE',
       'NAME_EDUCATION_TYPE', 'NAME_FAMILY_STATUS', 'NAME_HOUSING_TYPE',
       'REGION_POPULATION_RELATIVE', 'DAYS_BIRTH', 'DAYS_EMPLOYED',
       'DAYS_REGISTRATION', 'DAYS_ID_PUBLISH', 'OCCUPATION_TYPE',
       'CNT_FAM_MEMBERS', 'REGION_RATING_CLIENT',
       'REGION_RATING_CLIENT_W_CITY', 'WEEKDAY_APPR_PROCESS_START',
       'HOUR_APPR_PROCESS_START', 'REG_REGION_NOT_LIVE_REGION',
       'REG_REGION_NOT_WORK_REGION', 'LIVE_REGION_NOT_WORK_REGION',
       'REG_CITY_NOT_LIVE_CITY', 'REG_CITY_NOT_WORK_CITY',
       'LIVE_CITY_NOT_WORK_CITY', 'ORGANIZATION_TYPE', 'EXT_SOURCE_2',
       'EXT_SOURCE_3', 'OBS_30_CNT_SOCIAL_CIRCLE', 'DEF_30_CNT_SOCIAL_CIRCLE',
       'OBS_60_CNT_SOCIAL_CIRCLE', 'DEF_60_CNT_SOCIAL_CIRCLE',
       'DAYS_LAST_PHONE_CHANGE', 'AMT_REQ_CREDIT_BUREAU_HOUR',
       'AMT_REQ_CREDIT_BUREAU_DAY', 'AMT_REQ_CREDIT_BUREAU_WEEK',
       'AMT_REQ_CREDIT_BUREAU_MON', 'AMT_REQ_CREDIT_BUREAU_QRT',
       'AMT_REQ_CREDIT_BUREAU_YEAR'],
      dtype='object')
```

In [22]:
```
missing = pd.DataFrame(data.isna().sum().sort_values(ascending=False))
missing.reset_index(inplace=True)
missing.rename(columns={'index':'column',0:'null_count'},inplace=True)
missing['percent'] = missing['null_count']/data.shape[0]
missing
```

Out[22]:

	column	null_count	percent
0	OCCUPATION_TYPE	96391	0.313455
1	EXT_SOURCE_3	60965	0.198253
2	AMT_REQ_CREDIT_BUREAU_YEAR	41519	0.135016
3	AMT_REQ_CREDIT_BUREAU_MON	41519	0.135016
4	AMT_REQ_CREDIT_BUREAU_WEEK	41519	0.135016
5	AMT_REQ_CREDIT_BUREAU_DAY	41519	0.135016
6	AMT_REQ_CREDIT_BUREAU_HOUR	41519	0.135016
7	AMT_REQ_CREDIT_BUREAU_QRT	41519	0.135016
8	NAME_TYPE_SUITE	1292	0.004201
9	OBS_30_CNT_SOCIAL_CIRCLE	1021	0.003320
10	DEF_30_CNT_SOCIAL_CIRCLE	1021	0.003320
11	OBS_60_CNT_SOCIAL_CIRCLE	1021	0.003320
12	DEF_60_CNT_SOCIAL_CIRCLE	1021	0.003320
13	EXT_SOURCE_2	660	0.002146
14	AMT_GOODS_PRICE	278	0.000904
15	AMT_ANNUITY	12	0.000039
16	CNT_FAM_MEMBERS	2	0.000007
17	DAYS_LAST_PHONE_CHANGE	1	0.000003
18	NAME_FAMILY_STATUS	0	0.000000
19	NAME_EDUCATION_TYPE	0	0.000000
20	NAME_INCOME_TYPE	0	0.000000
21	NAME_CONTRACT_TYPE	0	0.000000
22	AMT_CREDIT	0	0.000000
23	AMT_INCOME_TOTAL	0	0.000000
24	CNT_CHILDREN	0	0.000000
25	CODE_GENDER	0	0.000000
26	REGION_POPULATION_RELATIVE	0	0.000000
27	TARGET	0	0.000000
28	NAME_HOUSING_TYPE	0	0.000000
29	REGION_RATING_CLIENT_W_CITY	0	0.000000
30	DAYS_BIRTH	0	0.000000
31	DAYS_EMPLOYED	0	0.000000
32	DAYS_REGISTRATION	0	0.000000
33	DAYS_ID_PUBLISH	0	0.000000
34	REGION_RATING_CLIENT	0	0.000000
35	WEEKDAY_APPR_PROCESS_START	0	0.000000
36	HOUR_APPR_PROCESS_START	0	0.000000
37	REG_REGION_NOT_LIVE_REGION	0	0.000000

	column	null_count	percent
38	REG_REGION_NOT_WORK_REGION	0	0.000000
39	LIVE_REGION_NOT_WORK_REGION	0	0.000000
40	REG_CITY_NOT_LIVE_CITY	0	0.000000
41	REG_CITY_NOT_WORK_CITY	0	0.000000
42	LIVE_CITY_NOT_WORK_CITY	0	0.000000
43	ORGANIZATION_TYPE	0	0.000000
44	SK_ID_CURR	0	0.000000

In [23]:
```
missing[missing.percent>0.4]
```

Out[23]:

column	null_count	percent

Still we can see few columns that are almost 50 percent null values, we can remove them as well.

In [24]:
```
data.drop(missing[missing.percent>0.4].column.values,axis=1,inplace=True)
```

In [25]:
```
data.shape
```

Out[25]:
```
(307511, 45)
```

Feature Engineering

In [26]:
```
data.isnull().sum().sort_values(ascending=False)
```

Out[26]:
```
OCCUPATION_TYPE                96391
EXT_SOURCE_3                   60965
AMT_REQ_CREDIT_BUREAU_YEAR     41519
AMT_REQ_CREDIT_BUREAU_MON      41519
AMT_REQ_CREDIT_BUREAU_WEEK     41519
AMT_REQ_CREDIT_BUREAU_DAY      41519
AMT_REQ_CREDIT_BUREAU_HOUR     41519
AMT_REQ_CREDIT_BUREAU_QRT      41519
NAME_TYPE_SUITE                 1292
OBS_30_CNT_SOCIAL_CIRCLE        1021
DEF_30_CNT_SOCIAL_CIRCLE        1021
OBS_60_CNT_SOCIAL_CIRCLE        1021
DEF_60_CNT_SOCIAL_CIRCLE        1021
EXT_SOURCE_2                     660
AMT_GOODS_PRICE                  278
AMT_ANNUITY                       12
CNT_FAM_MEMBERS                    2
DAYS_LAST_PHONE_CHANGE             1
NAME_FAMILY_STATUS                 0
NAME_EDUCATION_TYPE                0
NAME_INCOME_TYPE                   0
NAME_CONTRACT_TYPE                 0
AMT_CREDIT                         0
AMT_INCOME_TOTAL                   0
CNT_CHILDREN                       0
CODE_GENDER                        0
REGION_POPULATION_RELATIVE         0
TARGET                             0
NAME_HOUSING_TYPE                  0
REGION_RATING_CLIENT_W_CITY        0
DAYS_BIRTH                         0
DAYS_EMPLOYED                      0
DAYS_REGISTRATION                  0
DAYS_ID_PUBLISH                    0
REGION_RATING_CLIENT               0
WEEKDAY_APPR_PROCESS_START         0
HOUR_APPR_PROCESS_START            0
REG_REGION_NOT_LIVE_REGION         0
REG_REGION_NOT_WORK_REGION         0
LIVE_REGION_NOT_WORK_REGION        0
REG_CITY_NOT_LIVE_CITY             0
REG_CITY_NOT_WORK_CITY             0
LIVE_CITY_NOT_WORK_CITY            0
ORGANIZATION_TYPE                  0
SK_ID_CURR                         0
dtype: int64
```

Dealing Missing values of Numeric Variables

The mean is used for normal number distributions, which have a low amount of outliers.

If there are more outliers in the data, then median is generally used as it returns the central tendency for skewed number distributions.

we can deal column wise for the rest of missing values, and if we see from the last we have

DAYS_LAST_PHONE_CHANGE

```
In [27]: data['DAYS_LAST_PHONE_CHANGE'].isna().sum()
Out[27]: 1
```

```
In [28]: data.dropna(subset=['DAYS_LAST_PHONE_CHANGE'],inplace=True)
```

CNT_FAM_MEMBERS Column

As there is only one row with null value, decided to remove it.

```
In [29]: data['CNT_FAM_MEMBERS'] = data['CNT_FAM_MEMBERS'].fillna((data['CNT_FAM_MEMBERS'].mode()
         [0]))
```

```
In [30]: data['CNT_FAM_MEMBERS'].isnull().sum()
Out[30]: 0
```

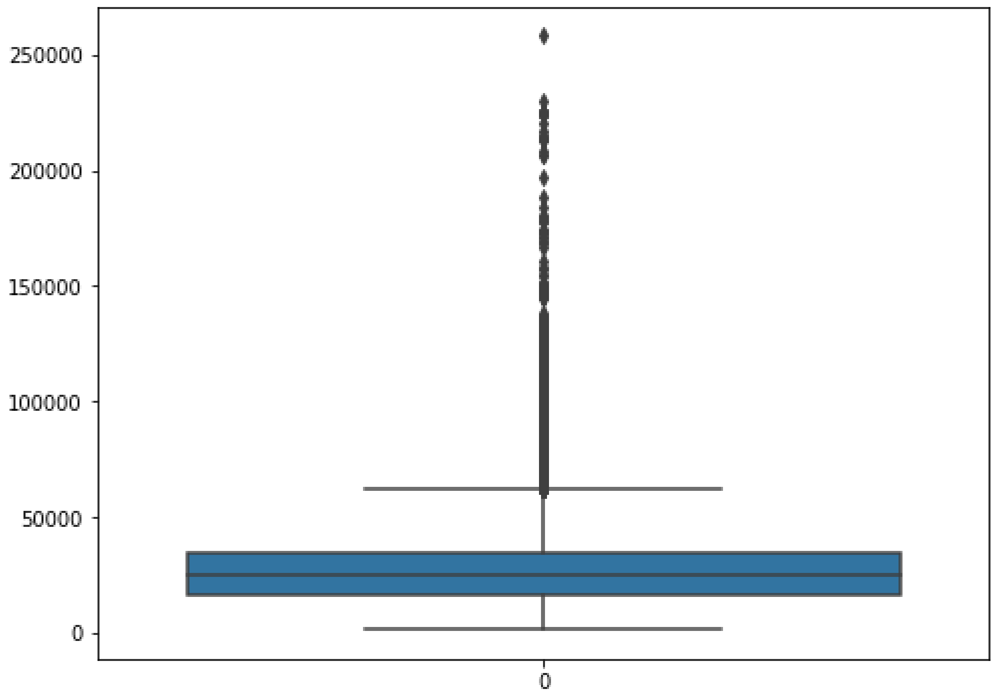
AMT_Annuity Column

```
In [31]: data['AMT_ANNUITY']
Out[31]: 0         24700.5
         1         35698.5
         2          6750.0
         3         29686.5
         4         21865.5
                    ...
         307506    27558.0
         307507    12001.5
         307508    29979.0
         307509    20205.0
         307510    49117.5
         Name: AMT_ANNUITY, Length: 307510, dtype: float64
```

```
In [32]: data['AMT_ANNUITY'].isna().sum()
Out[32]: 12
```

```
In [33]: plt.figure(figsize=(8,6))
         sns.boxplot(data['AMT_ANNUITY'])
```

```
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x2388d438f08>
```



Observing that it has significant amount of outliers, decided to impute with median

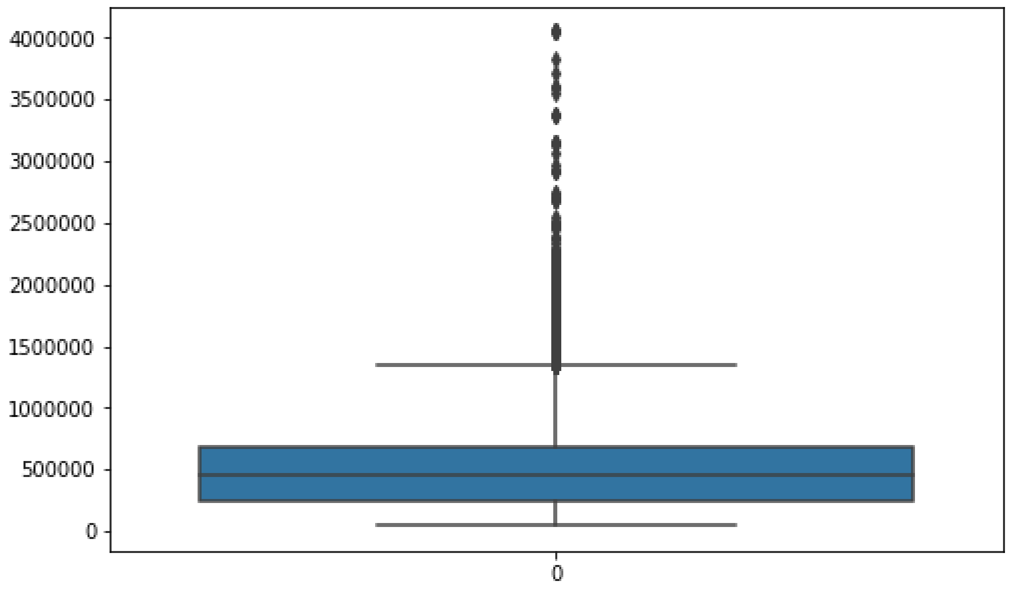
```
In [34]: data['AMT_ANNUITY'] = data['AMT_ANNUITY'].fillna((data['AMT_ANNUITY'].median()))
```

```
In [35]: data['AMT_ANNUITY'].isna().sum()
```

```
Out[35]: 0
```

AMT_GOODS_PRICE Column

```
In [36]: plt.figure(figsize=(8,5))
         sns.boxplot(data['AMT_GOODS_PRICE'])

Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x2388d418a48>
```



Observing that it has significant amount of outliers, decided to impute with median

```
In [37]: data['AMT_GOODS_PRICE'] = data['AMT_GOODS_PRICE'].fillna((data['AMT_GOODS_PRICE'].median
         ()))

In [38]: data['AMT_GOODS_PRICE'].isna().sum()

Out[38]: 0

In [39]: # Fill with median value
         data['AMT_GOODS_PRICE'] = data['AMT_GOODS_PRICE'].fillna((data['AMT_GOODS_PRICE'].median
         ()))
```

Dealing Missing Values of Categorical Variables

```
In [40]: # Fill missing values with a new class 'Unknown'
         data['OCCUPATION_TYPE'] = data['OCCUPATION_TYPE'].fillna('Unknown')

In [41]: # Fill the missing values with mode
         data['DEF_60_CNT_SOCIAL_CIRCLE'] = data['DEF_60_CNT_SOCIAL_CIRCLE'].fillna((data['DEF_60
         _CNT_SOCIAL_CIRCLE'].mode()[0]))
         data['OBS_30_CNT_SOCIAL_CIRCLE'] = data['OBS_30_CNT_SOCIAL_CIRCLE'].fillna((data['OBS_30
         _CNT_SOCIAL_CIRCLE'].mode()[0]))
         data['DEF_30_CNT_SOCIAL_CIRCLE'] = data['DEF_30_CNT_SOCIAL_CIRCLE'].fillna((data['DEF_30
         _CNT_SOCIAL_CIRCLE'].mode()[0]))
         data['OBS_60_CNT_SOCIAL_CIRCLE'] = data['OBS_60_CNT_SOCIAL_CIRCLE'].fillna((data['OBS_60
         _CNT_SOCIAL_CIRCLE'].mode()[0]))
```

In [42]:
```
# NAME_TYPE_SUITE
data['NAME_TYPE_SUITE'].value_counts()

# 'Unaccompanied' class is purely dominating the distribution. So, we use it to fill the
missing values
data['NAME_TYPE_SUITE'] = data['NAME_TYPE_SUITE'].fillna((data['NAME_TYPE_SUITE'].mode()
[0]))
```

In [43]:
```
data.isna().sum().sort_values(ascending=False).head(20)
```

Out[43]:
```
EXT_SOURCE_3                  60964
AMT_REQ_CREDIT_BUREAU_YEAR    41518
AMT_REQ_CREDIT_BUREAU_MON     41518
AMT_REQ_CREDIT_BUREAU_WEEK    41518
AMT_REQ_CREDIT_BUREAU_DAY     41518
AMT_REQ_CREDIT_BUREAU_HOUR    41518
AMT_REQ_CREDIT_BUREAU_QRT     41518
EXT_SOURCE_2                    659
NAME_TYPE_SUITE                   0
DAYS_EMPLOYED                     0
DAYS_BIRTH                        0
REGION_POPULATION_RELATIVE        0
NAME_HOUSING_TYPE                 0
NAME_FAMILY_STATUS                0
NAME_EDUCATION_TYPE               0
NAME_INCOME_TYPE                  0
AMT_CREDIT                        0
AMT_GOODS_PRICE                   0
AMT_ANNUITY                       0
DAYS_ID_PUBLISH                   0
dtype: int64
```

Dealing with columns related to date

In [44]:
```
data[data['AMT_REQ_CREDIT_BUREAU_DAY'].isna()].head()
```

Out[44]:

	SK_ID_CURR	TARGET	NAME_CONTRACT_TYPE	CODE_GENDER	CNT_CHILDREN	AMT_INCOME_TOTAL
3	100006	0	Cash loans	F	0	135000.0
9	100012	0	Revolving loans	M	0	135000.0
14	100018	0	Cash loans	F	0	189000.0
17	100021	0	Revolving loans	F	1	81000.0
20	100024	0	Revolving loans	M	0	135000.0

5 rows × 45 columns

In [45]:
```
# Fetching the columns
amt_req = []

for k in data.columns:
    if k.startswith('AMT_REQ_CREDIT_BUREAU_'):
        amt_req.append(k)  # Add features to list

amt_req
```

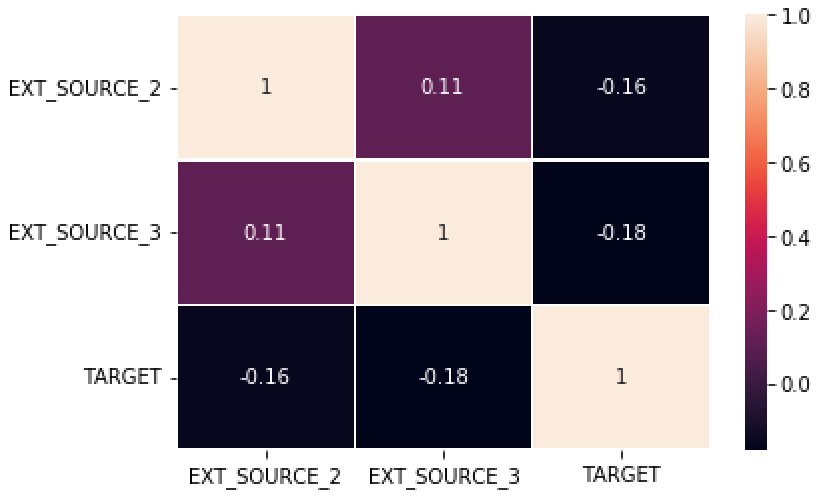
Out[45]:
```
['AMT_REQ_CREDIT_BUREAU_HOUR',
 'AMT_REQ_CREDIT_BUREAU_DAY',
 'AMT_REQ_CREDIT_BUREAU_WEEK',
 'AMT_REQ_CREDIT_BUREAU_MON',
 'AMT_REQ_CREDIT_BUREAU_QRT',
 'AMT_REQ_CREDIT_BUREAU_YEAR']
```

In [46]:
```
# Impute missing values with median
for col in amt_req:
    data[col] = data[col].fillna((data[col].median()))
```

In [47]:
```
data.isna().sum().sort_values(ascending=False).head(20)
```

Out[47]:
```
EXT_SOURCE_3                  60964
EXT_SOURCE_2                    659
DAYS_ID_PUBLISH                   0
DAYS_REGISTRATION                 0
DAYS_EMPLOYED                     0
DAYS_BIRTH                        0
REGION_POPULATION_RELATIVE        0
NAME_HOUSING_TYPE                 0
NAME_FAMILY_STATUS                0
NAME_EDUCATION_TYPE               0
NAME_INCOME_TYPE                  0
AMT_REQ_CREDIT_BUREAU_YEAR        0
CNT_FAM_MEMBERS                   0
NAME_TYPE_SUITE                   0
AMT_GOODS_PRICE                   0
AMT_ANNUITY                       0
AMT_CREDIT                        0
AMT_INCOME_TOTAL                  0
CNT_CHILDREN                      0
CODE_GENDER                       0
dtype: int64
```

```
In [48]: # Correlation matrix
         plt.figure(figsize=(6,4))
         sns.heatmap(round(data[['EXT_SOURCE_2', 'EXT_SOURCE_3', 'TARGET']].corr(),2),
                     linewidths=0.5, annot=True)
         plt.show()
```



```
In [49]: # Drop features
         data = data.drop(columns=['EXT_SOURCE_2','EXT_SOURCE_3'])
```

```
In [50]: data.isna().sum().sort_values(ascending=False).head(20)
```

```
Out[50]: AMT_REQ_CREDIT_BUREAU_YEAR    0
         NAME_INCOME_TYPE              0
         DAYS_ID_PUBLISH               0
         DAYS_REGISTRATION             0
         DAYS_EMPLOYED                 0
         DAYS_BIRTH                    0
         REGION_POPULATION_RELATIVE    0
         NAME_HOUSING_TYPE             0
         NAME_FAMILY_STATUS            0
         NAME_EDUCATION_TYPE           0
         NAME_TYPE_SUITE               0
         CNT_FAM_MEMBERS               0
         AMT_GOODS_PRICE               0
         AMT_ANNUITY                   0
         AMT_CREDIT                    0
         AMT_INCOME_TOTAL              0
         CNT_CHILDREN                  0
         CODE_GENDER                   0
         NAME_CONTRACT_TYPE            0
         TARGET                        0
         dtype: int64
```

Numerical Variables Binning for Data Visualization

In [51]:
```
data.select_dtypes(include='float')
```

Out[51]:

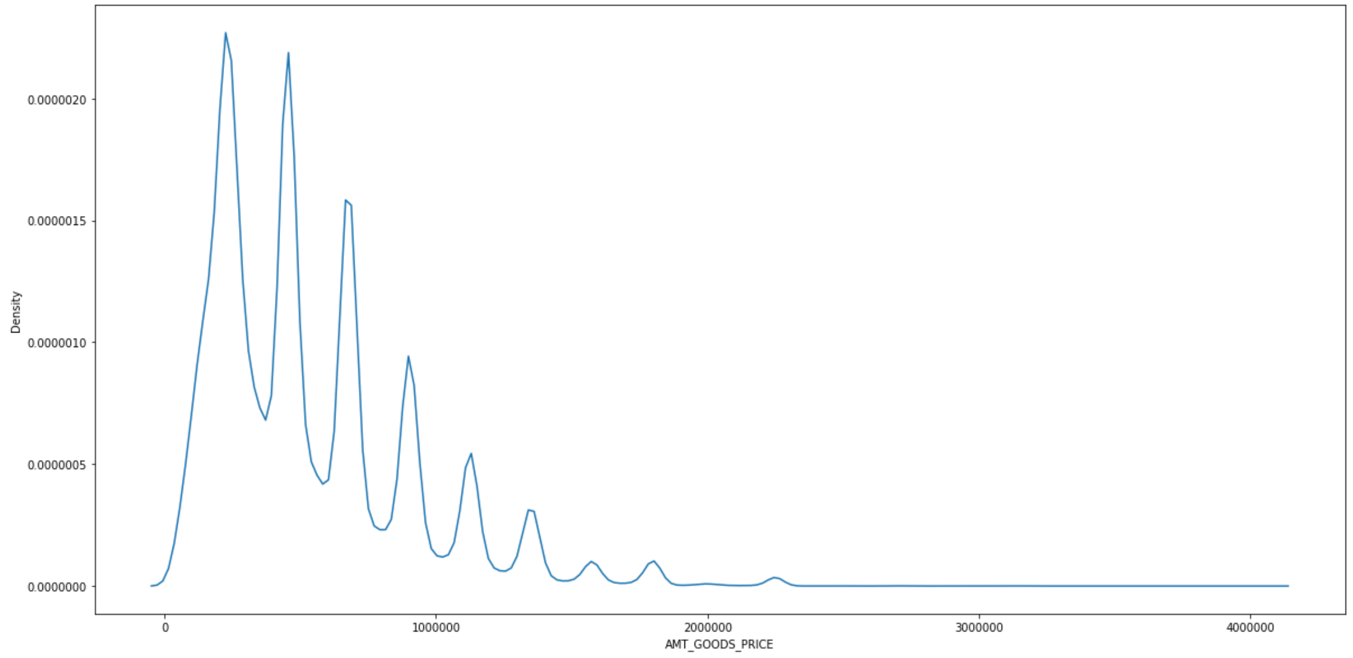
	AMT_INCOME_TOTAL	AMT_CREDIT	AMT_ANNUITY	AMT_GOODS_PRICE	REGION_POPULATION_REL
0	202500.0	406597.5	24700.5	351000.0	0.0
1	270000.0	1293502.5	35698.5	1129500.0	0.0
2	67500.0	135000.0	6750.0	135000.0	0.0
3	135000.0	312682.5	29686.5	297000.0	0.0
4	121500.0	513000.0	21865.5	513000.0	0.0
...	...	...	...	...	
307506	157500.0	254700.0	27558.0	225000.0	0.0
307507	72000.0	269550.0	12001.5	225000.0	0.0
307508	153000.0	677664.0	29979.0	585000.0	0.0
307509	171000.0	370107.0	20205.0	319500.0	0.0
307510	157500.0	675000.0	49117.5	675000.0	0.0

307510 rows × 18 columns

```
In [52]: # Number of unique values
         data.nunique().sort_values(ascending=False)
```

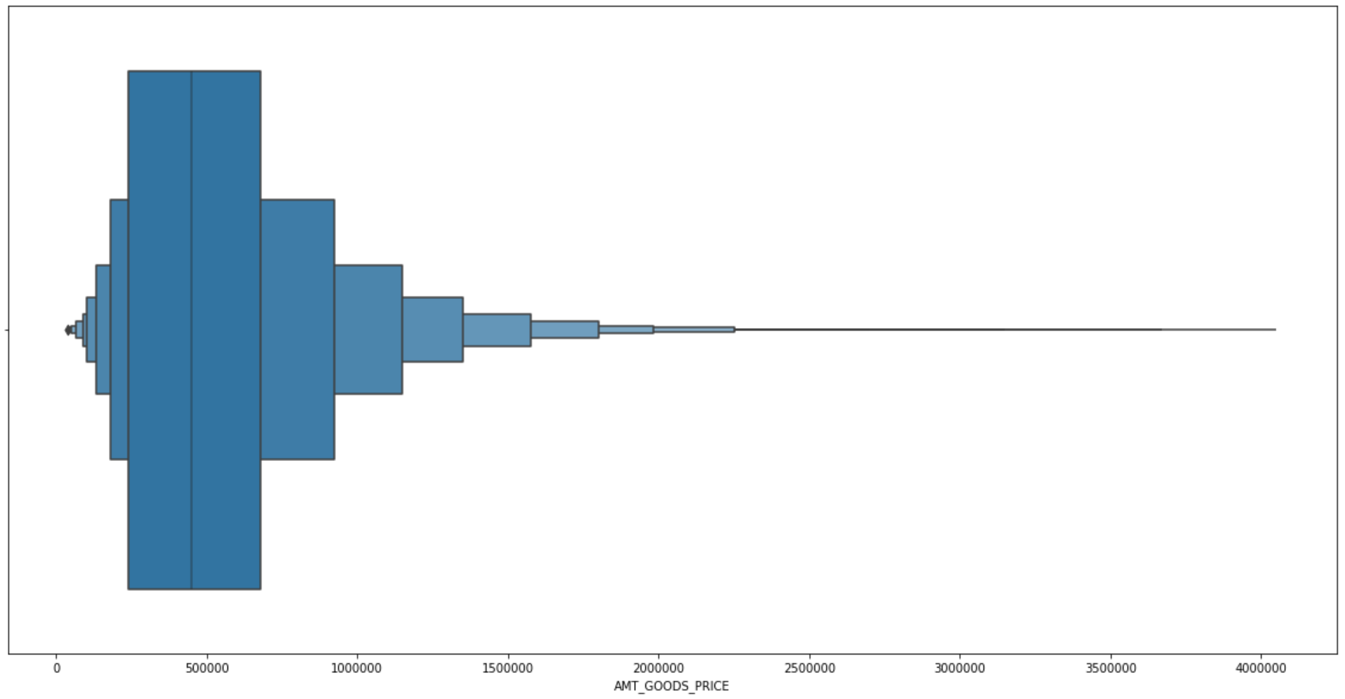
```
Out[52]: SK_ID_CURR                     307510
         DAYS_BIRTH                      17460
         DAYS_REGISTRATION               15688
         AMT_ANNUITY                     13672
         DAYS_EMPLOYED                   12574
         DAYS_ID_PUBLISH                  6168
         AMT_CREDIT                       5603
         DAYS_LAST_PHONE_CHANGE           3773
         AMT_INCOME_TOTAL                 2548
         AMT_GOODS_PRICE                  1002
         REGION_POPULATION_RELATIVE         81
         ORGANIZATION_TYPE                  58
         OBS_60_CNT_SOCIAL_CIRCLE           33
         OBS_30_CNT_SOCIAL_CIRCLE           33
         AMT_REQ_CREDIT_BUREAU_YEAR         25
         AMT_REQ_CREDIT_BUREAU_MON          24
         HOUR_APPR_PROCESS_START            24
         OCCUPATION_TYPE                    19
         CNT_FAM_MEMBERS                    17
         CNT_CHILDREN                       15
         AMT_REQ_CREDIT_BUREAU_QRT          11
         DEF_30_CNT_SOCIAL_CIRCLE           10
         AMT_REQ_CREDIT_BUREAU_WEEK          9
         DEF_60_CNT_SOCIAL_CIRCLE            9
         AMT_REQ_CREDIT_BUREAU_DAY           9
         NAME_INCOME_TYPE                    8
         NAME_TYPE_SUITE                     7
         WEEKDAY_APPR_PROCESS_START          7
         NAME_FAMILY_STATUS                  6
         NAME_HOUSING_TYPE                   6
         AMT_REQ_CREDIT_BUREAU_HOUR          5
         NAME_EDUCATION_TYPE                 5
         CODE_GENDER                         3
         REGION_RATING_CLIENT                3
         REGION_RATING_CLIENT_W_CITY         3
         REG_REGION_NOT_WORK_REGION          2
         LIVE_CITY_NOT_WORK_CITY             2
         REG_CITY_NOT_WORK_CITY              2
         REG_CITY_NOT_LIVE_CITY              2
         REG_REGION_NOT_LIVE_REGION          2
         NAME_CONTRACT_TYPE                  2
         TARGET                              2
         LIVE_REGION_NOT_WORK_REGION         2
         dtype: int64
```

```
In [53]: # KDE-plot
         plt.figure(figsize=(20,10))
         sns.kdeplot(data=data, x='AMT_GOODS_PRICE')
         plt.show()
```



```
In [54]: plt.figure(figsize=(20,10))
         sns.boxenplot(data=data, x='AMT_GOODS_PRICE')
```

```
Out[54]: <matplotlib.axes._subplots.AxesSubplot at 0x2388b6de608>
```

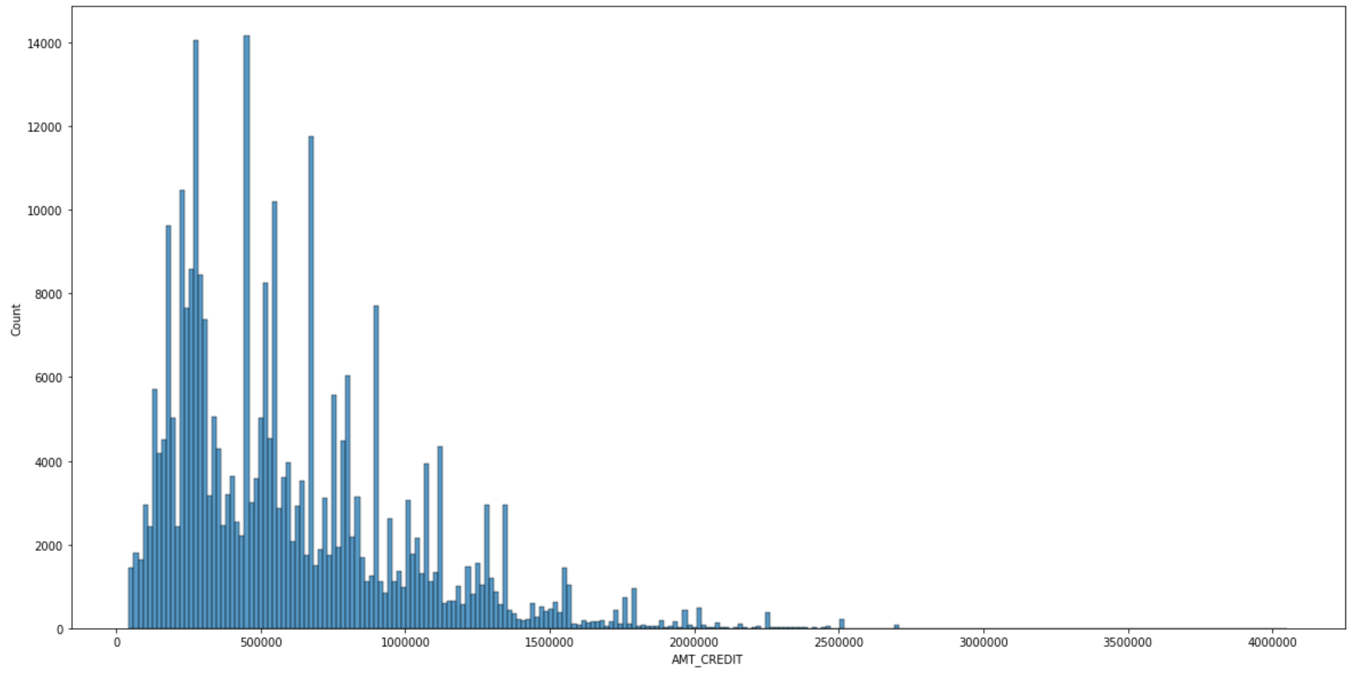


```
In [55]: # AMT_GOODS_PRICE
         data['AMT_GOODS_PRICE'].quantile([0.1,0.25,0.50,0.75,0.90])
```

```
Out[55]: 0.10     180000.0
         0.25     238500.0
         0.50     450000.0
         0.75     679500.0
         0.90    1093500.0
         Name: AMT_GOODS_PRICE, dtype: float64
```

```
In [56]: plt.figure(figsize=(20,10))
         sns.histplot(data['AMT_CREDIT'])
```

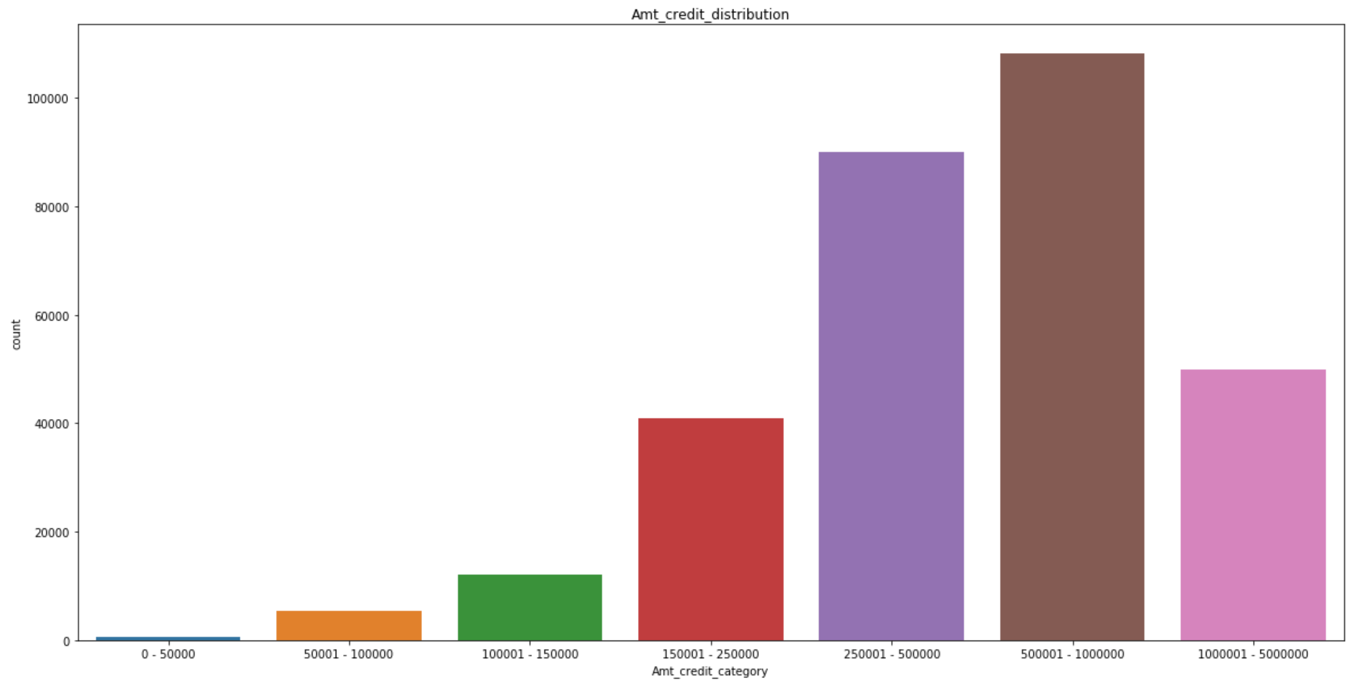
```
Out[56]: <matplotlib.axes._subplots.AxesSubplot at 0x2388b777e88>
```



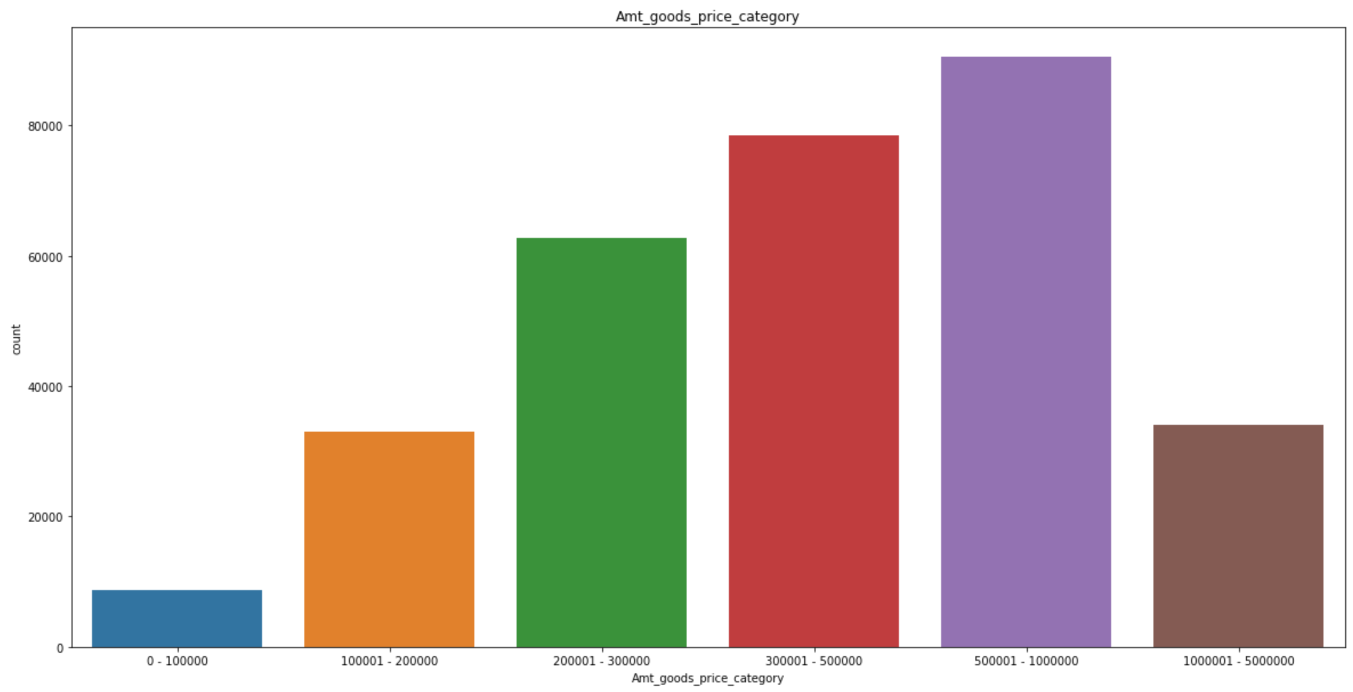
```
In [57]: data['AMT_CREDIT'].describe().loc[['min','max']]
```

```
Out[57]: min      45000.0
         max    4050000.0
         Name: AMT_CREDIT, dtype: float64
```

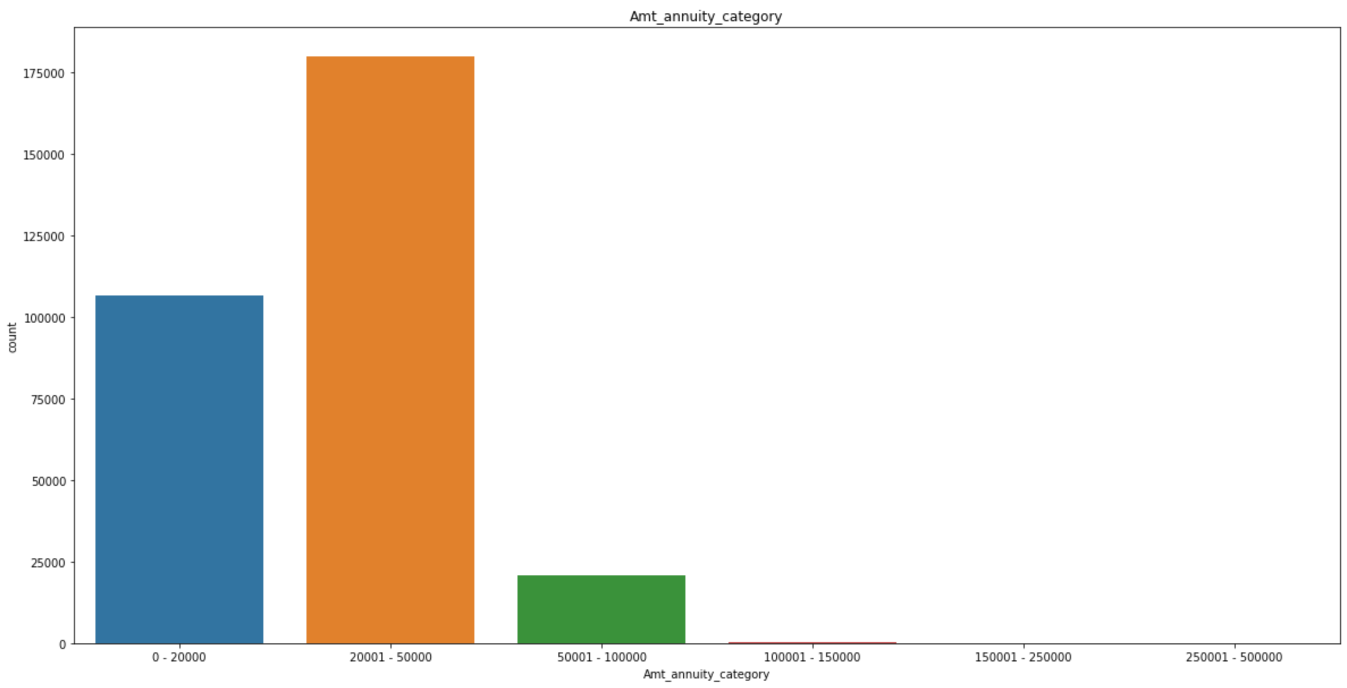
In [58]:
```
# Amt_Credit
labels= ['0 - 50000','50001 - 100000','100001 - 150000','150001 - 250000','250001 - 500000','500001 - 1000000', '1000001 - 5000000']
data['Amt_credit_category'] = pd.cut(data['AMT_CREDIT'], bins=[0,50000,100000,150000,250000,500000,1000000,5000000], labels=labels)

plt.figure(figsize=(20,10))
sns.countplot(x=data['Amt_credit_category'])
plt.title('Amt_credit_distribution')
plt.show()
```



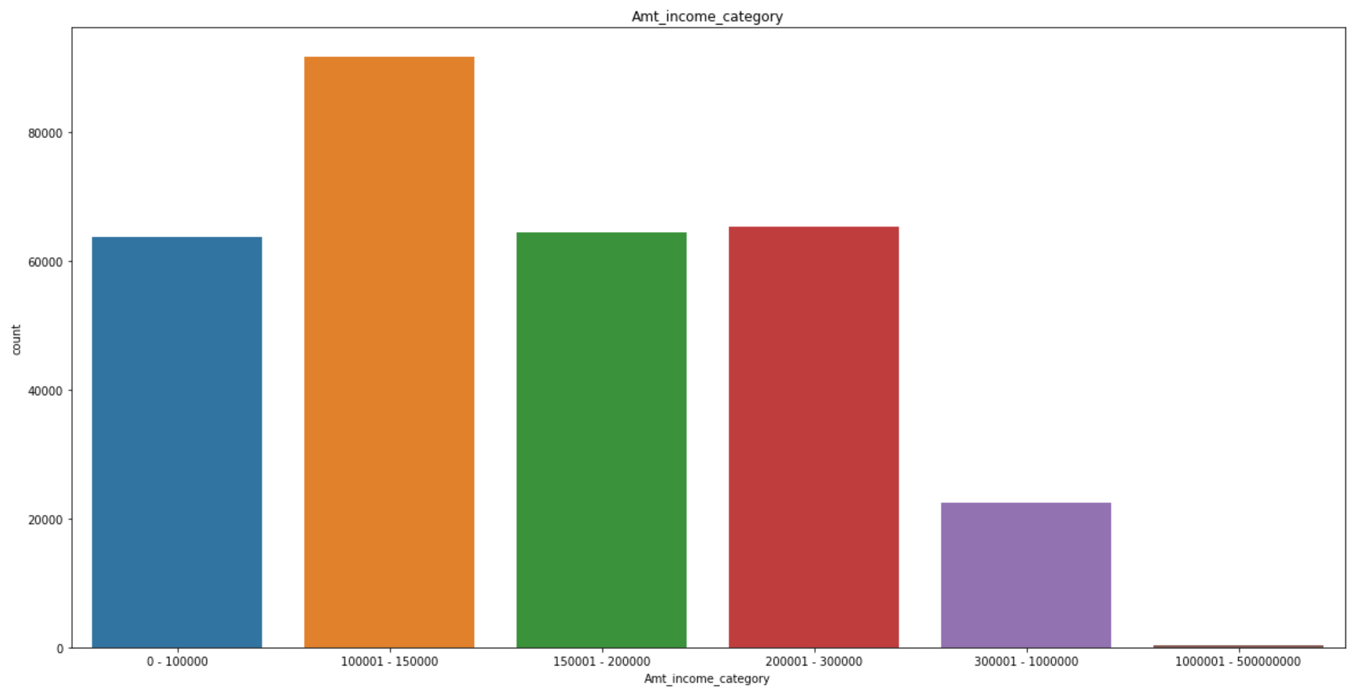
```
In [59]: # AMT_GOODS_PRICE
         labels= ['0 - 100000','100001 - 200000','200001 - 300000','300001 - 500000','500001 - 10
         00000','1000001 - 5000000']
         data['Amt_goods_price_category'] = pd.cut(data['AMT_GOODS_PRICE'], bins=[0,100000,20000
         0,300000,500000,1000000,5000000], labels=labels)

         plt.figure(figsize=(20,10))
         sns.countplot(x=data['Amt_goods_price_category'])
         plt.title('Amt_goods_price_category')
         plt.show()
```



In [60]:
```python
# AMT_ANNUITY
labels= ['0 - 20000','20001 - 50000','50001 - 100000','100001 - 150000','150001 - 250000','250001 - 500000']
data['Amt_annuity_category'] = pd.cut(data['AMT_ANNUITY'], bins=[0,20000,50000,100000,150000,250000,300000], labels=labels)

plt.figure(figsize=(20,10))
sns.countplot(x=data['Amt_annuity_category'])
plt.title('Amt_annuity_category')
plt.show()
```



In [61]:
```python
# AMT_INCOME_TOTAL
labels= ['0 - 100000','100001 - 150000','150001 - 200000','200001 - 300000','300001 - 10
00000','1000001 - 500000000']
data['Amt_income_category'] = pd.cut(data['AMT_INCOME_TOTAL'], bins=[0,100000,150000,200
000,300000,1000000,500000000], labels=labels)

plt.figure(figsize=(20,10))
sns.countplot(x=data['Amt_income_category'])
plt.title('Amt_income_category')
plt.show()
```

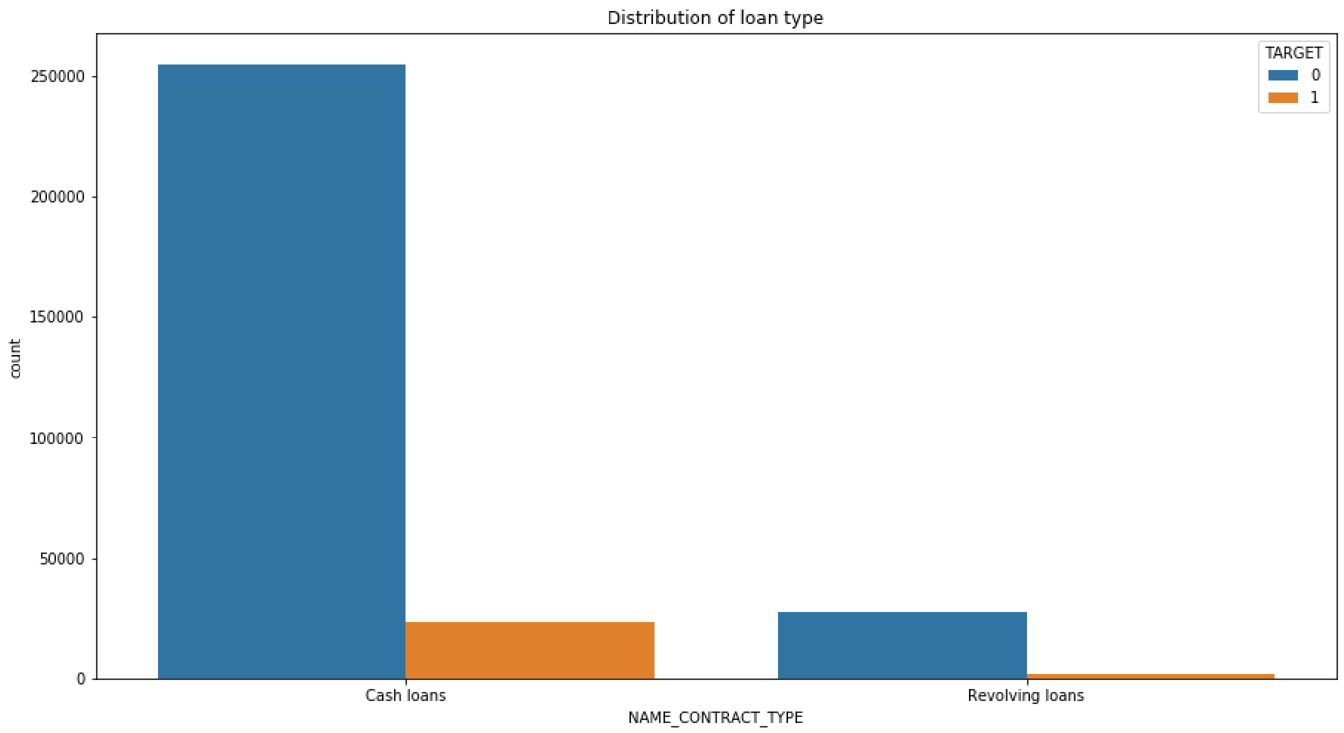


Categorical variables data visualization

In [62]:
```python
# NAME_CONTRACT_TYPE
data['NAME_CONTRACT_TYPE'].value_counts()
```

Out[62]:
```
Cash loans         278231
Revolving loans     29279
Name: NAME_CONTRACT_TYPE, dtype: int64
```

In [63]:
```python
# Countplot
plt.figure(figsize=(15,8))
sns.countplot(x='NAME_CONTRACT_TYPE', data=data, hue='TARGET')
plt.title("Distribution of loan type")
plt.show()
```



By observation we can say that those who have taken cash loan defaulted the loan most.

In [64]:
```python
# NAME_CONTRACT_TYPE
data['NAME_CONTRACT_TYPE'].value_counts()
```

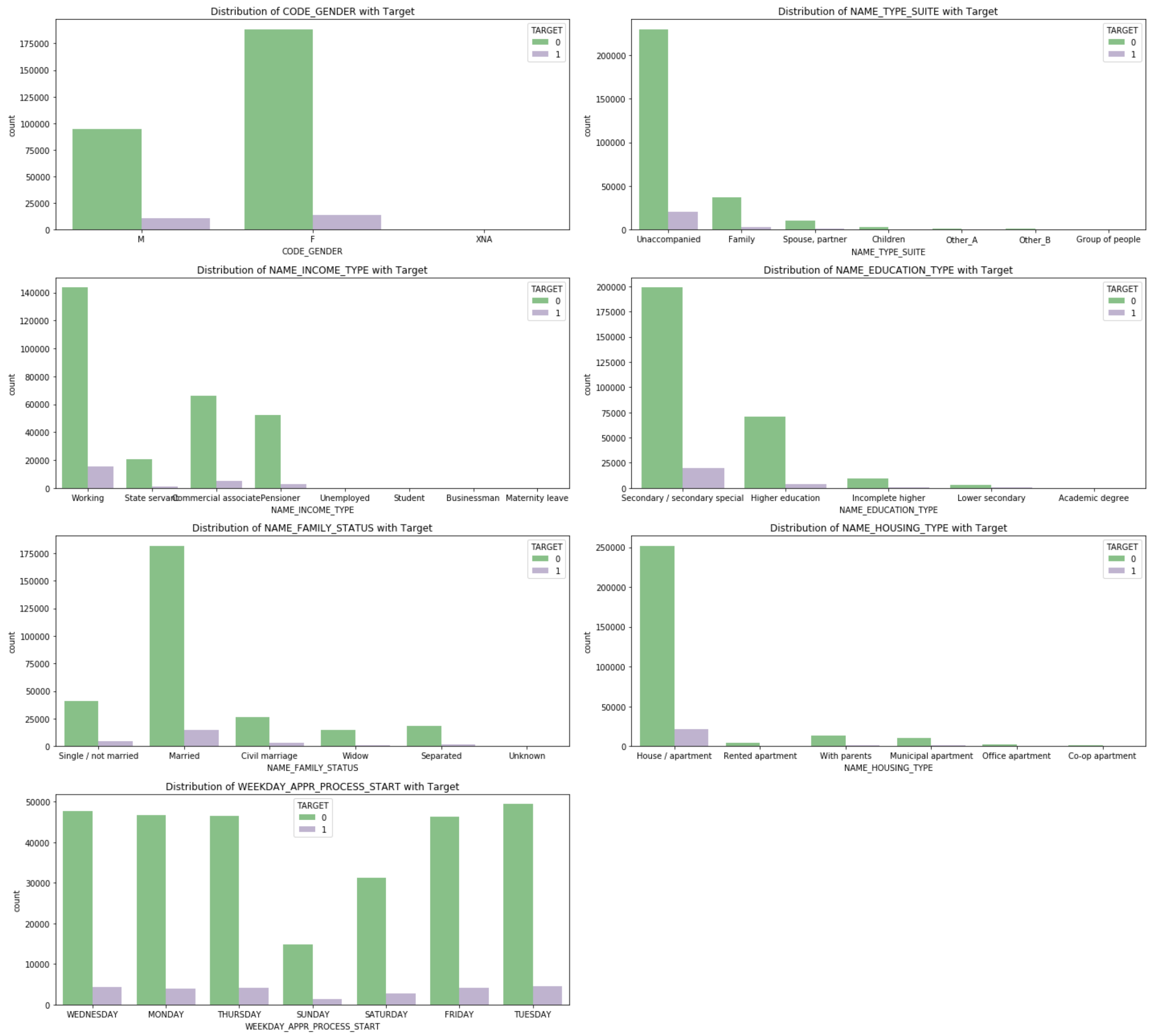
Out[64]:
```
Cash loans         278231
Revolving loans     29279
Name: NAME_CONTRACT_TYPE, dtype: int64
```

In [65]:
```python
# Dataframe for loan type with target
loan_with_target = data.groupby(['NAME_CONTRACT_TYPE', 'TARGET']).size().reset_index(nam
e='count')

loan_with_target['Percentage'] = round((loan_with_target['count']/len(data['NAME_CONTRAC
T_TYPE']))*100,2)
loan_with_target
```

Out[65]:

	NAME_CONTRACT_TYPE	TARGET	count	Percentage
0	Cash loans	0	255010	82.93
1	Cash loans	1	23221	7.55
2	Revolving loans	0	27675	9.00
3	Revolving loans	1	1604	0.52

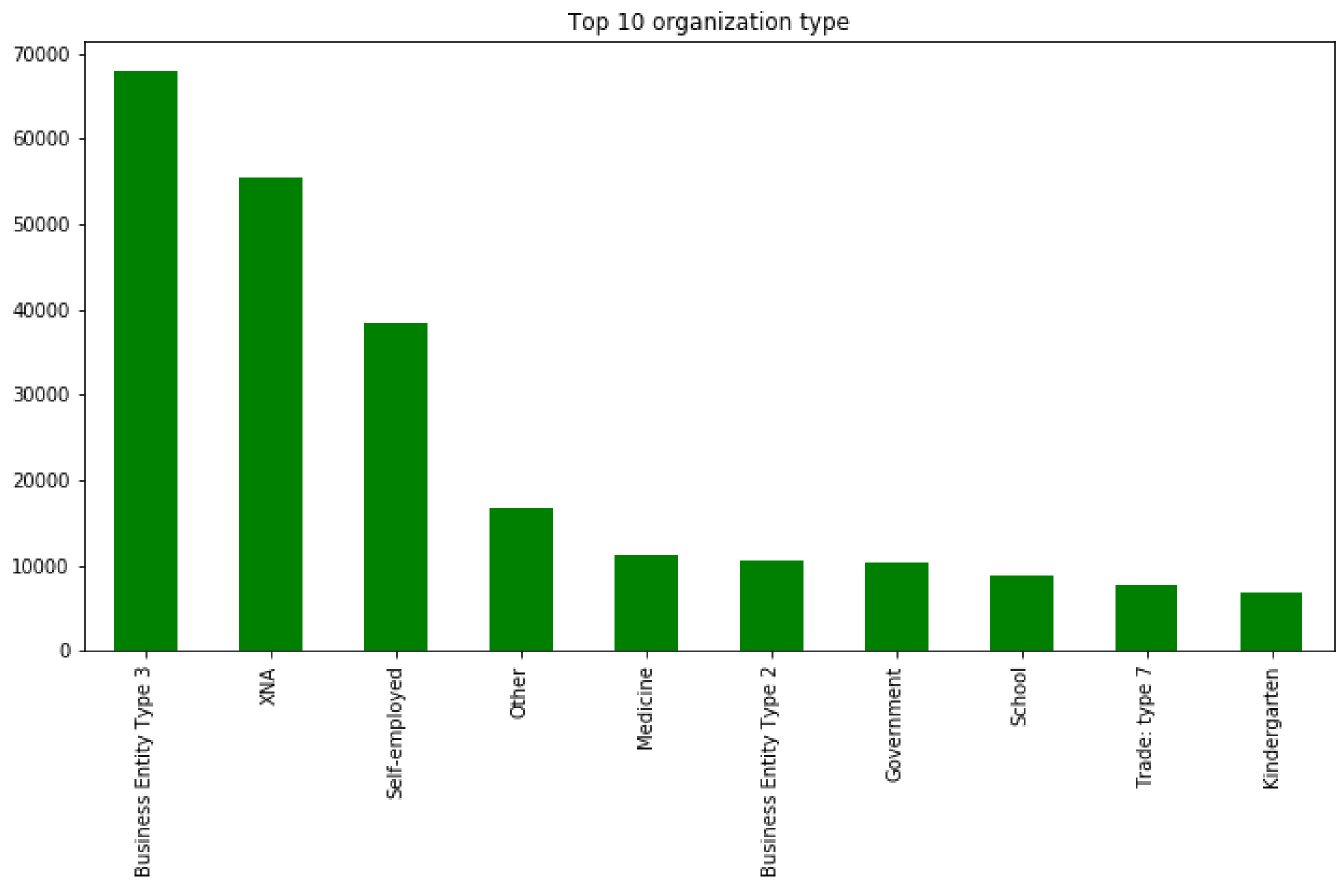
In [66]:
```python
cols = ['CODE_GENDER', 'NAME_TYPE_SUITE','NAME_INCOME_TYPE','NAME_EDUCATION_TYPE','NAME_
FAMILY_STATUS',
       'NAME_HOUSING_TYPE', 'WEEKDAY_APPR_PROCESS_START']

# Countplot
plt.figure(figsize=(20,18))
for index, c in enumerate(cols):
    plt.subplot(4,2, index+1)
    sns.countplot(x=c, data=data, hue='TARGET', palette='Accent')
    plt.title(f"Distribution of {c} with Target")

plt.tight_layout()
```



By Close observation of each bar chart, we can come to following conclusions:

1. Females are less likely to default the loan than male.
2. Working client, Commercial associate and Pensioner have taken more loans.
3. Unaccompanied has taken most number of loans.
4. Married client has received more number of credits.
5. Most of the clients have their house apartment.
6. All days have equal number of application received, except sunday.

In [67]:
```
# Organization type
plt.figure(figsize=(12,6))
data['ORGANIZATION_TYPE'].value_counts().sort_values(ascending=False)[:10].plot(kind='ba
r', color='green')
plt.title("Top 10 organization type")
plt.show()
```



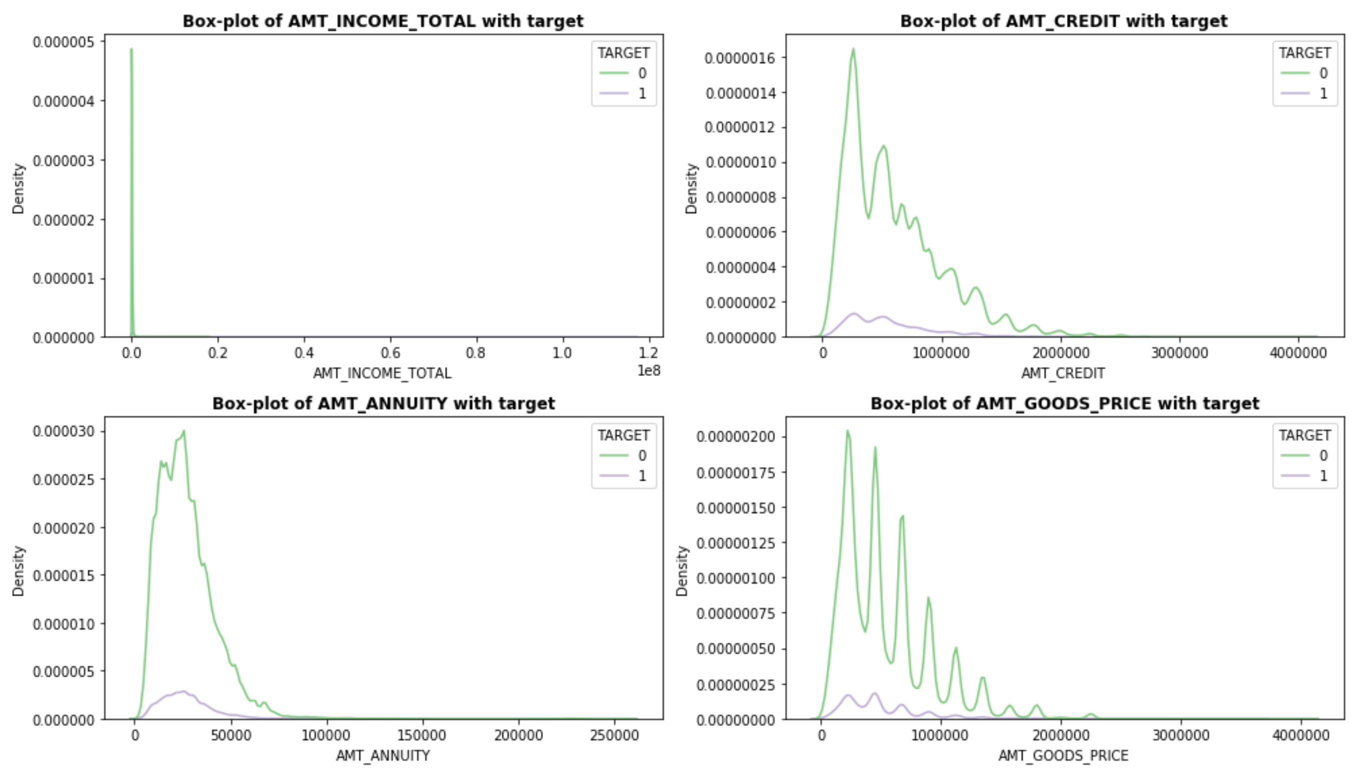
In [68]:
```
# Numeric features and categorical features
num_features = data.select_dtypes(include=['int', 'float']).columns
num_cat_features = data.select_dtypes(include=['int', 'float', 'category']).columns
```

```
In [69]: data['TARGET']
```

```
Out[69]: 0         1
         1         0
         2         0
         3         0
         4         0
                  ..
         307506    0
         307507    0
         307508    0
         307509    1
         307510    0
         Name: TARGET, Length: 307510, dtype: int64
```

```
In [70]: # Numeric dataframe
         num_data = data[np.concatenate([num_features,np.array(['TARGET'])])]

         defaulters = num_data[num_data['TARGET']==1]  # Dataframe for defaulters
         repayers = num_data[num_data['TARGET']==0]   # Dataframe for non-defaulters
```

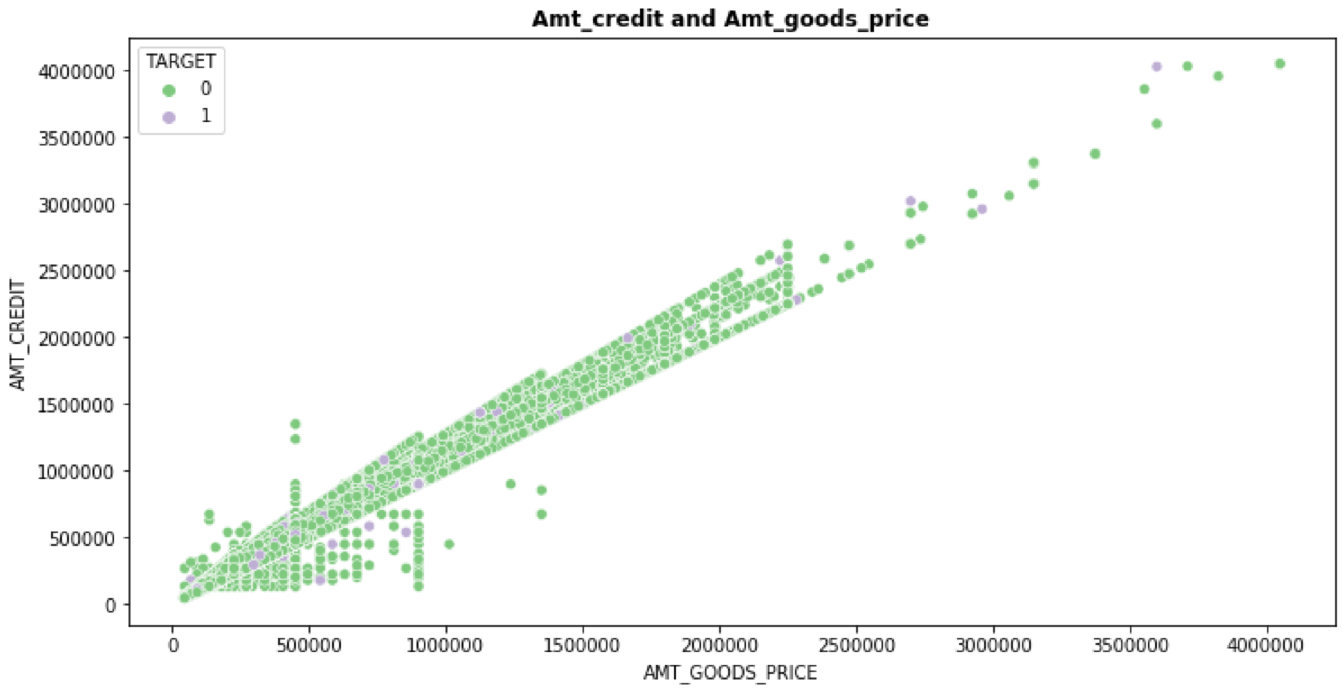
```
In [72]: # Amt_features
         amt_var = ['AMT_INCOME_TOTAL','AMT_CREDIT','AMT_ANNUITY','AMT_GOODS_PRICE']

         plt.figure(figsize=(14,8))
         for index, k in enumerate(amt_var):
             plt.subplot(2,2, index+1)
             sns.kdeplot(x=k, data=num_data, hue='TARGET', palette='Accent')
             plt.title(f"Box-plot of {k} with target", fontweight='bold')

         plt.tight_layout()
```



Observations:

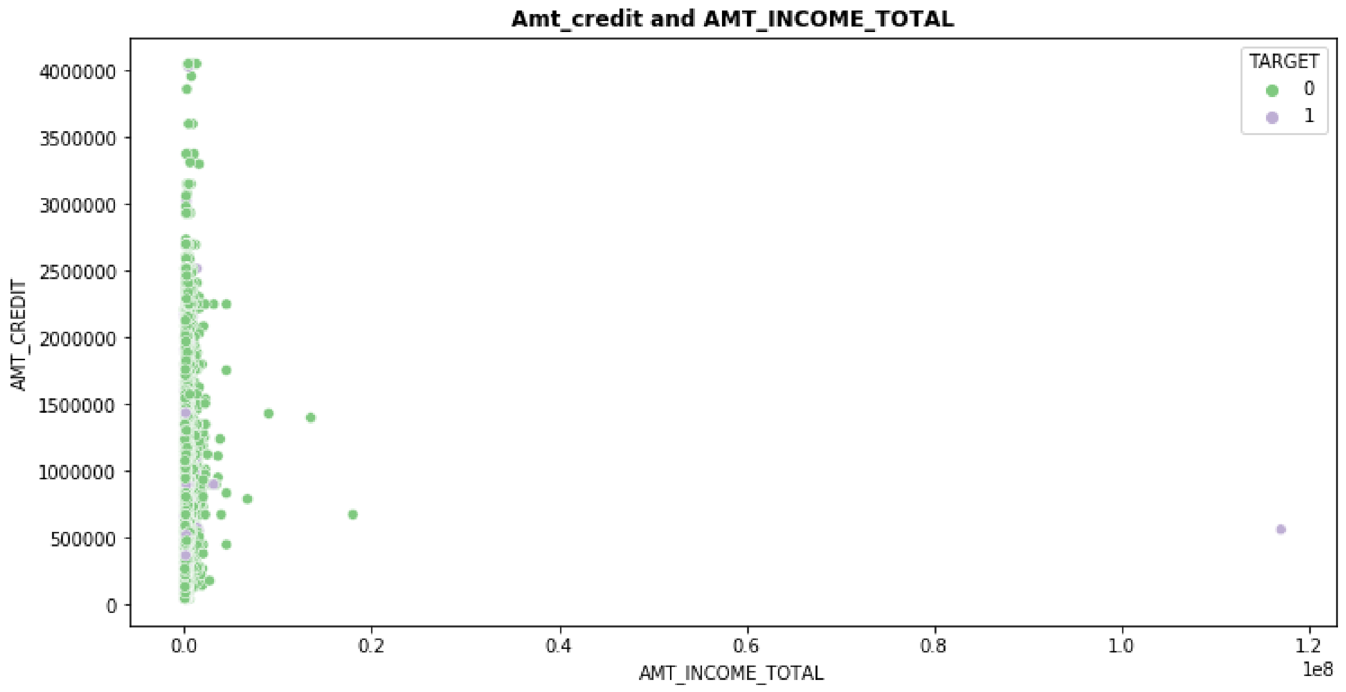
1. Most of the defaulters are from high-income groups.
2. Most defaulters fall under the category of amt_credit between 0 to 1 million.
3. Annuity payment of 0 to 50000 have more number of defaults.
4. Amount goods price between o to 1 million have more number of defaults.

```
In [73]: # Scatter plot
         plt.figure(figsize=(12,6))
         sns.scatterplot(data=num_data, x='AMT_GOODS_PRICE', y='AMT_CREDIT', palette='Accent', hu
         e='TARGET')
         plt.title("Amt_credit and Amt_goods_price", fontweight='bold')
         plt.show()
```



Here we can observe that Amt_goods_price and Amt_credit have linear relation. And, most of the defaulters are under 1 million level.

In [74]:
```
plt.figure(figsize=(12,6))
sns.scatterplot(data=num_data, x='AMT_INCOME_TOTAL', y='AMT_CREDIT', palette='Accent', h
ue='TARGET')
plt.title("Amt_credit and AMT_INCOME_TOTAL", fontweight='bold')
plt.show()
```



People with income less than 1 million is taking more number of loans. And, people who got credit/loans less than 150,000 are more likely to default.

Final Observations:

1. Female loan has less default rate. So, the bank should give a little bit priority to females.
2. Those clients who do not have any accompany should be focused.
3. Safest segementation of employment are workers, commercial associates and pensioners.
4. Client who have the higher education should be given more loans.
5. Married clients are safer than unmarried.
6. People having house/apartment are safer to provide loans.
7. Low-skill laborers and drivers should be given less priority as they have high probability of making defaults.
8. People having income less than 1 million and taking loans near to 1 million have higher chance of defaults. So, should not be given focus.
9. Married couples having children less than five are safe for providng loans.
10. Client having annuity less than 100K are safer side for the bank.

In [ ]:
```

```