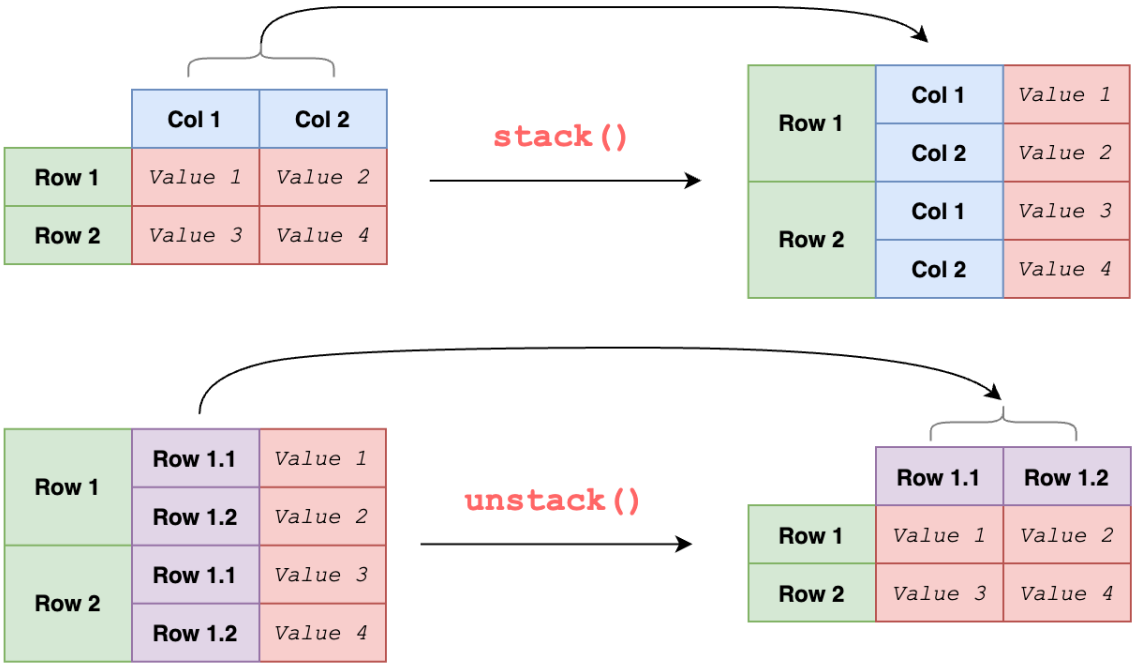
Stacking and Unstacking in MultiIndex Object in Pandas

In pandas, a multi-index object is a way to represent hierarchical data structures within a DataFrame or Series. Multi-indexing allows you to have multiple levels of row or column indices, providing a way to organize and work with complex, structured data.

"Stacking" and "unstacking" are operations that you can perform on multi-indexed DataFrames to change the arrangement of the data, essentially reshaping the data between a wide and a long format (or vice versa).



1. Stacking:

- Stacking is the process of "melting" or pivoting the innermost level of column labels to become the innermost level of row labels.
- This operation is typically used when you want to convert a wide DataFrame with multi-level columns into a long format.
- You can use the `.stack()` method to perform stacking. By default, it will stack the innermost level of columns.

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

In [ ]:
```
# Create a DataFrame with multi-level columns
df = pd.DataFrame(np.random.rand(4, 4), columns=[['A', 'A', 'B', 'B'], ['X', 'Y', '
print(df)
```

```
          A                   B
          X         Y         X         Y
0  0.960684  0.118538  0.900984  0.485585
1  0.946716  0.049913  0.444658  0.991469
2  0.656110  0.158270  0.759727  0.203801
3  0.360581  0.965035  0.797212  0.102426
```

In [ ]:
```
# Stack the innermost level of columns
stacked_df = df.stack()
print(stacked_df)
```

```
            A         B
0 X  0.960684  0.900984
  Y  0.118538  0.485585
1 X  0.946716  0.444658
  Y  0.049913  0.991469
2 X  0.656110  0.759727
  Y  0.158270  0.203801
3 X  0.360581  0.797212
  Y  0.965035  0.102426
```

2. **Unstacking**:

- Unstacking is the reverse operation of stacking. It involves pivoting the innermost level of row labels to become the innermost level of column labels.
- You can use the `.unstack()` method to perform unstacking. By default, it will unstack the innermost level of row labels.

Example:

In [ ]:
```
# Unstack the innermost level of row labels
unstacked_df = stacked_df.unstack()
print(unstacked_df)
```

```
          A                   B
          X         Y         X         Y
0  0.960684  0.118538  0.900984  0.485585
1  0.946716  0.049913  0.444658  0.991469
2  0.656110  0.158270  0.759727  0.203801
3  0.360581  0.965035  0.797212  0.102426
```

You can specify the level you want to stack or unstack by passing the `level` parameter to the `stack()` or `unstack()` methods. For example:

In [ ]:
```
# Stack the second level of columns
stacked_df = df.stack(level=1)
stacked_df
```

Follow for more AI content: https://lnkd.in/gaJtbwcu

Out[ ]:

		A	B
0	**X**	0.960684	0.900984
	Y	0.118538	0.485585
1	**X**	0.946716	0.444658
	Y	0.049913	0.991469
2	**X**	0.656110	0.759727
	Y	0.158270	0.203801
3	**X**	0.360581	0.797212
	Y	0.965035	0.102426

In [ ]:
```
# Unstack the first level of row labels
unstacked_df = stacked_df.unstack(level=0)
unstacked_df
```

Out[ ]:

	A				B			
	0	**1**	**2**	**3**	**0**	**1**	**2**	**3**
X	0.960684	0.946716	0.65611	0.360581	0.900984	0.444658	0.759727	0.797212
Y	0.118538	0.049913	0.15827	0.965035	0.485585	0.991469	0.203801	0.102426

In [ ]:
```
index_val = [('cse',2019),('cse',2020),('cse',2021),('cse',2022),('ece',2019),('ece
multiindex = pd.MultiIndex.from_tuples(index_val)
multiindex.levels[1]
```

Out[ ]:
```
Index([2019, 2020, 2021, 2022], dtype='int64')
```

In [ ]:
```
branch_df1 = pd.DataFrame(
    [
        [1,2],
        [3,4],
        [5,6],
        [7,8],
        [9,10],
        [11,12],
        [13,14],
        [15,16],
    ],
    index = multiindex,
    columns = ['avg_package','students']
)

branch_df1
```

Out[ ]:

		avg_package	students
cse	2019	1	2
	2020	3	4
	2021	5	6
	2022	7	8
ece	2019	9	10
	2020	11	12
	2021	13	14
	2022	15	16

In [ ]:
```
# multiindex df from columns perspective
branch_df2 = pd.DataFrame(
    [
        [1,2,0,0],
        [3,4,0,0],
        [5,6,0,0],
        [7,8,0,0],
    ],
    index = [2019,2020,2021,2022],
    columns = pd.MultiIndex.from_product([['delhi','mumbai'],['avg_package','studer
)

branch_df2
```

Out[ ]:

	delhi		mumbai	
	avg_package	students	avg_package	students
2019	1	2	0	0
2020	3	4	0	0
2021	5	6	0	0
2022	7	8	0	0

In [ ]:
```
branch_df1
```

Out[ ]:

		avg_package	students
cse	2019	1	2
	2020	3	4
	2021	5	6
	2022	7	8
ece	2019	9	10
	2020	11	12
	2021	13	14
	2022	15	16

In [ ]:
```
branch_df1.unstack().unstack()
```

Out[ ]:
```
avg_package  2019  cse     1
                    ece     9
             2020  cse     3
                    ece    11
             2021  cse     5
                    ece    13
             2022  cse     7
                    ece    15
students     2019  cse     2
                    ece    10
             2020  cse     4
                    ece    12
             2021  cse     6
                    ece    14
             2022  cse     8
                    ece    16
dtype: int64
```

In [ ]:
```
branch_df1.unstack().stack()
```

Out[ ]:

		avg_package	students
cse	**2019**	1	2
	2020	3	4
	2021	5	6
	2022	7	8
ece	**2019**	9	10
	2020	11	12
	2021	13	14
	2022	15	16

In [ ]:
```
branch_df2
```

Out[ ]:

	delhi		mumbai	
	avg_package	**students**	**avg_package**	**students**
2019	1	2	0	0
2020	3	4	0	0
2021	5	6	0	0
2022	7	8	0	0

In [ ]:
```
branch_df2.stack()
```

Out[ ]:

		delhi	mumbai
2019	**avg_package**	1	0
	students	2	0
2020	**avg_package**	3	0
	students	4	0
2021	**avg_package**	5	0
	students	6	0
2022	**avg_package**	7	0
	students	8	0

In [ ]:
```
branch_df2.stack().stack()
```

Out[ ]:
```
2019  avg_package  delhi     1
                   mumbai    0
      students     delhi     2
                   mumbai    0
2020  avg_package  delhi     3
                   mumbai    0
      students     delhi     4
                   mumbai    0
2021  avg_package  delhi     5
                   mumbai    0
      students     delhi     6
                   mumbai    0
2022  avg_package  delhi     7
                   mumbai    0
      students     delhi     8
                   mumbai    0
dtype: int64
```

Stacking and unstacking can be very useful when you need to reshape your data to make it more suitable for different types of analysis or visualization. They are common operations in data manipulation when working with multi-indexed DataFrames in pandas.

Working with MultiIndex DataFrames

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

Creating MutiIndex Dataframe

In [ ]:
```
index_val = [('cse',2019),('cse',2020),('cse',2021),('cse',2022),('ece',2019),('ece
multiindex = pd.MultiIndex.from_tuples(index_val)
multiindex.levels
```

Out[ ]:
```
FrozenList([['cse', 'ece'], [2019, 2020, 2021, 2022]])
```

In [ ]:
```
branch_df = pd.DataFrame(
    [
        [1,2,0,0],
        [3,4,0,0],
        [5,6,0,0],
        [7,8,0,0],
        [9,10,0,0],
        [11,12,0,0],
        [13,14,0,0],
        [15,16,0,0],
    ],
    index = multiindex,
    columns = pd.MultiIndex.from_product([['delhi','mumbai'],['avg_package','studer
)

branch_df
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
	2022	7	8	0	0
ece	2019	9	10	0	0
	2020	11	12	0	0
	2021	13	14	0	0
	2022	15	16	0	0

Basic Checks

In [ ]:
```
# HEAD
branch_df.head()
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
	2022	7	8	0	0
ece	2019	9	10	0	0

In [ ]:
```
# Tail
branch_df.tail()
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2022	7	8	0	0
ece	2019	9	10	0	0
	2020	11	12	0	0
	2021	13	14	0	0
	2022	15	16	0	0

In [ ]:
```
#shape
branch_df.shape
```

Out[ ]:
```
(8, 4)
```

In [ ]:
```
# info
branch_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 8 entries, ('cse', 2019) to ('ece', 2022)
Data columns (total 4 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   (delhi, avg_package)   8 non-null      int64
 1   (delhi, students)      8 non-null      int64
 2   (mumbai, avg_package)  8 non-null      int64
 3   (mumbai, students)     8 non-null      int64
dtypes: int64(4)
memory usage: 632.0+ bytes
```

In [ ]:
```
# duplicated
branch_df.duplicated().sum()
```

Out[ ]:
```
0
```

In [ ]:
```
# isnull
branch_df.isnull().sum()
```

Out[ ]:
```
delhi   avg_package    0
        students       0
mumbai  avg_package    0
        students       0
dtype: int64
```

How to Extract

In [ ]:
```
# extracting single row
branch_df.loc[('cse',2022)]
```

Out[ ]:
```
delhi   avg_package    7
        students       8
mumbai  avg_package    0
        students       0
Name: (cse, 2022), dtype: int64
```

In [ ]:
```
branch_df
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
	2022	7	8	0	0
ece	2019	9	10	0	0
	2020	11	12	0	0
	2021	13	14	0	0
	2022	15	16	0	0

In [ ]:
```
# extract multiple rows
branch_df.loc[('cse',2021):('ece',2021)]
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2021	5	6	0	0
	2022	7	8	0	0
ece	2019	9	10	0	0
	2020	11	12	0	0
	2021	13	14	0	0

In [ ]:
```
# using iloc
branch_df.iloc[2:5]
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2021	5	6	0	0
	2022	7	8	0	0
ece	2019	9	10	0	0

In [ ]:
```
branch_df.iloc[2:8:2]
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2021	5	6	0	0
ece	2019	9	10	0	0
	2021	13	14	0	0

In [ ]:
```
# extacting cols
branch_df['delhi']['students']
```

Out[ ]:
```
cse  2019     2
     2020     4
     2021     6
     2022     8
ece  2019    10
     2020    12
     2021    14
     2022    16
Name: students, dtype: int64
```

In [ ]:
```
branch_df.iloc[:,1:3]
```

Out[ ]:

		delhi	mumbai
		students	avg_package
cse	2019	2	0
	2020	4	0
	2021	6	0
	2022	8	0
ece	2019	10	0
	2020	12	0
	2021	14	0
	2022	16	0

In [ ]:
```
# Extracting both
branch_df.iloc[[0,4],[1,2]]
```

Out[ ]:

		delhi	mumbai
		students	avg_package
cse	2019	2	0
ece	2019	10	0

Sorting

In [ ]:
```
branch_df.sort_index(ascending=False)
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
ece	2022	15	16	0	0
	2021	13	14	0	0
	2020	11	12	0	0
	2019	9	10	0	0
cse	2022	7	8	0	0
	2021	5	6	0	0
	2020	3	4	0	0
	2019	1	2	0	0

In [ ]:
```
branch_df.sort_index(ascending=[False,True])
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
ece	2019	9	10	0	0
	2020	11	12	0	0
	2021	13	14	0	0
	2022	15	16	0	0
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
	2022	7	8	0	0

In [ ]:
```
branch_df.sort_index(level=0,ascending=[False])
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
ece	2019	9	10	0	0
	2020	11	12	0	0
	2021	13	14	0	0
	2022	15	16	0	0
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
	2022	7	8	0	0

In [ ]:
```
# multiindex dataframe(col) -> transpose
branch_df.transpose()
```

Out[ ]:

		cse				ece			
		2019	2020	2021	2022	2019	2020	2021	2022
delhi	avg_package	1	3	5	7	9	11	13	15
	students	2	4	6	8	10	12	14	16
mumbai	avg_package	0	0	0	0	0	0	0	0
	students	0	0	0	0	0	0	0	0

In [ ]:
```
# swaplevel
branch_df.swaplevel(axis=1)
```

Out[ ]:

		avg_package	students	avg_package	students
		delhi	delhi	mumbai	mumbai
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
	2022	7	8	0	0
ece	2019	9	10	0	0
	2020	11	12	0	0
	2021	13	14	0	0
	2022	15	16	0	0

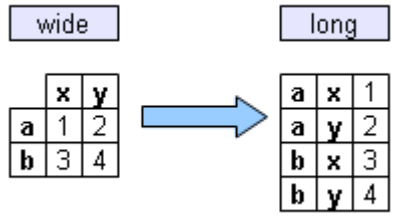
In [ ]:
```
branch_df.swaplevel()
```

Out[ ]:

		delhi		mumbai	
		avg_package	students	avg_package	students
2019	cse	1	2	0	0
2020	cse	3	4	0	0
2021	cse	5	6	0	0
2022	cse	7	8	0	0
2019	ece	9	10	0	0
2020	ece	11	12	0	0
2021	ece	13	14	0	0
2022	ece	15	16	0	0

Long vs. Wide Data Formats in Data Analysis

Long Vs Wide Data



"Long" and "wide" are terms often used in data analysis and data reshaping in the context of data frames or tables, typically in software like R or Python. They describe two different ways of organizing and structuring data.

1. Long Format (also called "Tidy" or "Melted"):

- In the long format, each row typically represents an individual observation or data point, and each column represents a variable or attribute.
- This format is useful when you have a dataset where you want to store multiple measurements or values for the same individuals or entities.
- Long data is often more convenient for various types of statistical analysis, and it can be easier to filter, subset, and manipulate the data.

2. Wide Format (also called "Spread" or "Pivoted"):

- In the wide format, each row represents an individual, and multiple variables are stored in separate columns.
- This format is useful when you have a dataset with a few variables and you want to see the values for each individual at a glance, which can be more readable for humans.
- Wide data can be useful for simple summaries and initial data exploration.

Here's a simple example to illustrate the difference:

Long Format:

ID	Variable	Value
1	Age	25
1	Height	175
1	Weight	70
2	Age	30
2	Height	160
2	Weight	60

Wide Format:

ID	Age	Height	Weight
1	25	175	70
2	30	160	60

Converting data between long and wide formats is often necessary depending on the specific analysis or visualization tasks you want to perform. In software like R and Python, there are functions and libraries available for reshaping data between these formats, such as `tidyr` in R and `pivot` functions in Python's pandas library for moving from wide to long format, and `gather` in R and `melt` in pandas for moving from long to wide format.

Data Conversion : melt

- wide is converted into long

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

In [ ]:
```
pd.DataFrame({'cse':[120]})
```

Out[ ]:

	cse
0	120

In [ ]:
```
pd.DataFrame({'cse':[120]}).melt()
```

Out[ ]:

	variable	value
0	cse	120

In [ ]:
```
# melt -> branch with year
pd.DataFrame({'cse':[120],'ece':[100],'mech':[50]})
```

Out[ ]:

	cse	ece	mech
0	120	100	50

In [ ]:
```
# melt -> branch with year
pd.DataFrame({'cse':[120],'ece':[100],'mech':[50]}).melt()
```

Out[ ]:

	variable	value
0	cse	120
1	ece	100
2	mech	50

In [ ]:
```
# var_name and value_name
pd.DataFrame({'cse':[120],'ece':[100],'mech':[50]}).melt(var_name='branch',value_na
```

Out[ ]:

	branch	num_students
0	cse	120
1	ece	100
2	mech	50

In [ ]:
```
pd.DataFrame(
    {
        'branch':['cse','ece','mech'],
        '2020':[100,150,60],
        '2021':[120,130,80],
        '2022':[150,140,70]
    }
)
```

Out[ ]:

	branch	2020	2021	2022
0	cse	100	120	150
1	ece	150	130	140
2	mech	60	80	70

In [ ]:
```
pd.DataFrame(
    {
        'branch':['cse','ece','mech'],
        '2020':[100,150,60],
        '2021':[120,130,80],
        '2022':[150,140,70]
    }
).melt()
```

Out[ ]:

	variable	value
0	branch	cse
1	branch	ece
2	branch	mech
3	2020	100
4	2020	150
5	2020	60
6	2021	120
7	2021	130
8	2021	80
9	2022	150
10	2022	140
11	2022	70

In [ ]:
```
# id_vars ->  we don't want to convert
pd.DataFrame(
    {
        'branch':['cse','ece','mech'],
        '2020':[100,150,60],
```

```
        '2021':[120,130,80],
        '2022':[150,140,70]
    }
).melt(id_vars=['branch'],var_name='year',value_name='students')
```

Out[ ]:

	branch	year	students
0	cse	2020	100
1	ece	2020	150
2	mech	2020	60
3	cse	2021	120
4	ece	2021	130
5	mech	2021	80
6	cse	2022	150
7	ece	2022	140
8	mech	2022	70

Real-World Example:

- In the context of COVID-19 data, data for deaths and confirmed cases are initially stored in wide formats.
- The data is converted to long format, making it easier to conduct analyses.
- In the long format, each row represents a specific location, date, and the corresponding number of deaths or confirmed cases. This format allows for efficient merging and analysis, as it keeps related data in one place and facilitates further data exploration.

In [ ]:
```
# melt -> real world example
death = pd.read_csv('Data\Day42\death_covid.csv')
confirm = pd.read_csv('Data\Day42\Confirmed_covid.csv')
```

In [ ]:
```
death.head()
```

Out[ ]:

	Province/State	Country/Region	Lat	Long	1/22/20	1/23/20	1/24/20	1/25/20	1/2
0	NaN	Afghanistan	33.93911	67.709953	0	0	0	0	
1	NaN	Albania	41.15330	20.168300	0	0	0	0	
2	NaN	Algeria	28.03390	1.659600	0	0	0	0	
3	NaN	Andorra	42.50630	1.521800	0	0	0	0	
4	NaN	Angola	-11.20270	17.873900	0	0	0	0	

5 rows × 1081 columns

In [ ]:
```
confirm.head()
```

Out[ ]:

	Province/State	Country/Region	Lat	Long	1/22/20	1/23/20	1/24/20	1/25/20	1/2
0	NaN	Afghanistan	33.93911	67.709953	0	0	0	0	
1	NaN	Albania	41.15330	20.168300	0	0	0	0	
2	NaN	Algeria	28.03390	1.659600	0	0	0	0	
3	NaN	Andorra	42.50630	1.521800	0	0	0	0	
4	NaN	Angola	-11.20270	17.873900	0	0	0	0	

5 rows × 1081 columns

In [ ]:
```
death = death.melt(id_vars=['Province/State','Country/Region','Lat','Long'],var_nam
confirm = confirm.melt(id_vars=['Province/State','Country/Region','Lat','Long'],var
```

In [ ]:
```
death.head()
```

Out[ ]:

	Province/State	Country/Region	Lat	Long	date	num_deaths
0	NaN	Afghanistan	33.93911	67.709953	1/22/20	0
1	NaN	Albania	41.15330	20.168300	1/22/20	0
2	NaN	Algeria	28.03390	1.659600	1/22/20	0
3	NaN	Andorra	42.50630	1.521800	1/22/20	0
4	NaN	Angola	-11.20270	17.873900	1/22/20	0

In [ ]:
```
confirm.head()
```

Out[ ]:

	Province/State	Country/Region	Lat	Long	date	num_cases
0	NaN	Afghanistan	33.93911	67.709953	1/22/20	0
1	NaN	Albania	41.15330	20.168300	1/22/20	0
2	NaN	Algeria	28.03390	1.659600	1/22/20	0
3	NaN	Andorra	42.50630	1.521800	1/22/20	0
4	NaN	Angola	-11.20270	17.873900	1/22/20	0

In [ ]:
```
confirm.merge(death,on=['Province/State','Country/Region','Lat','Long','date'])
```

Out[ ]:

	Province/State	Country/Region	Lat	Long	date	num_cases	num_deaths
0	NaN	Afghanistan	33.939110	67.709953	1/22/20	0	0
1	NaN	Albania	41.153300	20.168300	1/22/20	0	0
2	NaN	Algeria	28.033900	1.659600	1/22/20	0	0
3	NaN	Andorra	42.506300	1.521800	1/22/20	0	0
4	NaN	Angola	-11.202700	17.873900	1/22/20	0	0
...	...	...	...	...	...	...	...
311248	NaN	West Bank and Gaza	31.952200	35.233200	1/2/23	703228	5708
311249	NaN	Winter Olympics 2022	39.904200	116.407400	1/2/23	535	0
311250	NaN	Yemen	15.552727	48.516388	1/2/23	11945	2159
311251	NaN	Zambia	-13.133897	27.849332	1/2/23	334661	4024
311252	NaN	Zimbabwe	-19.015438	29.154857	1/2/23	259981	5637

311253 rows × 7 columns

In [ ]:
```
confirm.merge(death,on=['Province/State','Country/Region','Lat','Long','date'])[['C
```

Out[ ]:

	Country/Region	date	num_cases	num_deaths
0	Afghanistan	1/22/20	0	0
1	Albania	1/22/20	0	0
2	Algeria	1/22/20	0	0
3	Andorra	1/22/20	0	0
4	Angola	1/22/20	0	0
...	...	...	...	...
311248	West Bank and Gaza	1/2/23	703228	5708
311249	Winter Olympics 2022	1/2/23	535	0
311250	Yemen	1/2/23	11945	2159
311251	Zambia	1/2/23	334661	4024
311252	Zimbabwe	1/2/23	259981	5637

311253 rows × 4 columns

The choice between long and wide data formats depends on the nature of the dataset and the specific analysis or visualization tasks you want to perform. Converting data between these formats can help optimize data organization for different analytical needs.

Pivot Table in Python

In Python, you can create pivot tables using libraries like pandas or NumPy, which are commonly used for data manipulation and analysis.

Now, let's create a simple pivot table:

In [ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
```

In [ ]:
```
# Sample data
data = {
    'Date': ['2023-01-01', '2023-01-01', '2023-01-02', '2023-01-02'],
    'Product': ['A', 'B', 'A', 'B'],
    'Sales': [100, 200, 150, 250],
}
```

In [ ]:
```
# Create a pandas DataFrame
df = pd.DataFrame(data)
```

In [ ]:
```
# Create a pivot table
pivot_table = pd.pivot_table(df, values='Sales', index='Date', columns='Product', a
```

In [ ]:
```
print(pivot_table)
```

```
Product       A    B
Date
2023-01-01  100  200
2023-01-02  150  250
```

In this example, we first create a DataFrame using sample data. Then, we use the `pd.pivot_table` function to create a pivot table. Here's what each argument does:

- `df`: The DataFrame containing the data.
- `values`: The column for which you want to aggregate values (in this case, 'Sales').
- `index`: The column that you want to use as the rows in the pivot table (in this case, 'Date').
- `columns`: The column that you want to use as the columns in the pivot table (in this case, 'Product').
- `aggfunc`: The aggregation function to apply when there are multiple values that need to be combined. You can use functions like 'sum', 'mean', 'max', etc., depending on your requirements.

Real-world Examples

In [ ]:
```
df = sns.load_dataset('tips')
```

In [ ]:
```
df.head()
```

Out[ ]:

	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

In [ ]:
```
df.pivot_table(index='sex',columns='smoker',values='total_bill')
```

Out[ ]:

smoker	Yes	No
sex		
Male	22.284500	19.791237
Female	17.977879	18.105185

In [ ]:
```
## aggfunc# aggfunc
df.pivot_table(index='sex',columns='smoker',values='total_bill',aggfunc='std')
```

Out[ ]:

smoker	Yes	No
sex		
Male	9.911845	8.726566
Female	9.189751	7.286455

In [ ]:
```
# all cols together
df.pivot_table(index='sex',columns='smoker')['size']
```

```
<ipython-input-7-dd5735ad60ca>:2: FutureWarning: pivot_table dropped a column beca
use it failed to aggregate. This behavior is deprecated and will raise in a future
version of pandas. Select only the columns that can be aggregated.
  df.pivot_table(index='sex',columns='smoker')['size']
```

Out[ ]:

smoker	Yes	No
sex		
Male	2.500000	2.711340
Female	2.242424	2.592593

In [ ]:
```
# multidimensional
df.pivot_table(index=['sex','smoker'],columns=['day','time'],aggfunc={'size':'mean'
```

Out[ ]:

		size									
	day	Thur		Fri		Sat	Sun	Thur			
	time	Lunch	Dinner	Lunch	Dinner	Dinner	Dinner	Lunch	Dinner	Lunch	D
sex	smoker										
Male	Yes	2.300000	NaN	1.666667	2.4	2.629630	2.600000	5.00	NaN	2.20	
	No	2.500000	NaN	NaN	2.0	2.656250	2.883721	6.70	NaN	NaN	
Female	Yes	2.428571	NaN	2.000000	2.0	2.200000	2.500000	5.00	NaN	3.48	
	No	2.500000	2.0	3.000000	2.0	2.307692	3.071429	5.17	3.0	3.00	

In [ ]:
```
# margins
df.pivot_table(index='sex',columns='smoker',values='total_bill',aggfunc='sum',margi
```

Out[ ]:

smoker	Yes	No	All
sex			
Male	1337.07	1919.75	3256.82
Female	593.27	977.68	1570.95
All	1930.34	2897.43	4827.77

Plotting graph

In [ ]:
```
df = pd.read_csv('Data\Day43\expense_data.csv')
```

In [ ]:
```
df.head()
```

Out[ ]:

	Date	Account	Category	Subcategory	Note	INR	Income/Expense	Note.1	Amou
0	3/2/2022 10:11	CUB - online payment	Food	NaN	Brownie	50.0	Expense	NaN	5
1	3/2/2022 10:11	CUB - online payment	Other	NaN	To lended people	300.0	Expense	NaN	30
2	3/1/2022 19:50	CUB - online payment	Food	NaN	Dinner	78.0	Expense	NaN	7
3	3/1/2022 18:56	CUB - online payment	Transportation	NaN	Metro	30.0	Expense	NaN	3
4	3/1/2022 18:22	CUB - online payment	Food	NaN	Snacks	67.0	Expense	NaN	6

In [ ]:
```
df['Category'].value_counts()
```

Out[ ]:
```
Category
Food                156
Other                60
Transportation       31
Apparel               7
Household             6
Allowance             6
Social Life           5
Education             1
Salary                1
Self-development      1
Beauty                1
Gift                  1
Petty cash            1
Name: count, dtype: int64
```

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 277 entries, 0 to 276
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Date            277 non-null    object 
 1   Account         277 non-null    object 
 2   Category        277 non-null    object 
 3   Subcategory     0 non-null      float64
 4   Note            273 non-null    object 
 5   INR             277 non-null    float64
 6   Income/Expense  277 non-null    object 
 7   Note.1          0 non-null      float64
 8   Amount          277 non-null    float64
 9   Currency        277 non-null    object 
 10  Account.1       277 non-null    float64
dtypes: float64(5), object(6)
memory usage: 23.9+ KB
```

In [ ]:
```
df['Date'] = pd.to_datetime(df['Date'])
```

In [ ]:
```
df['month'] = df['Date'].dt.month_name()
```

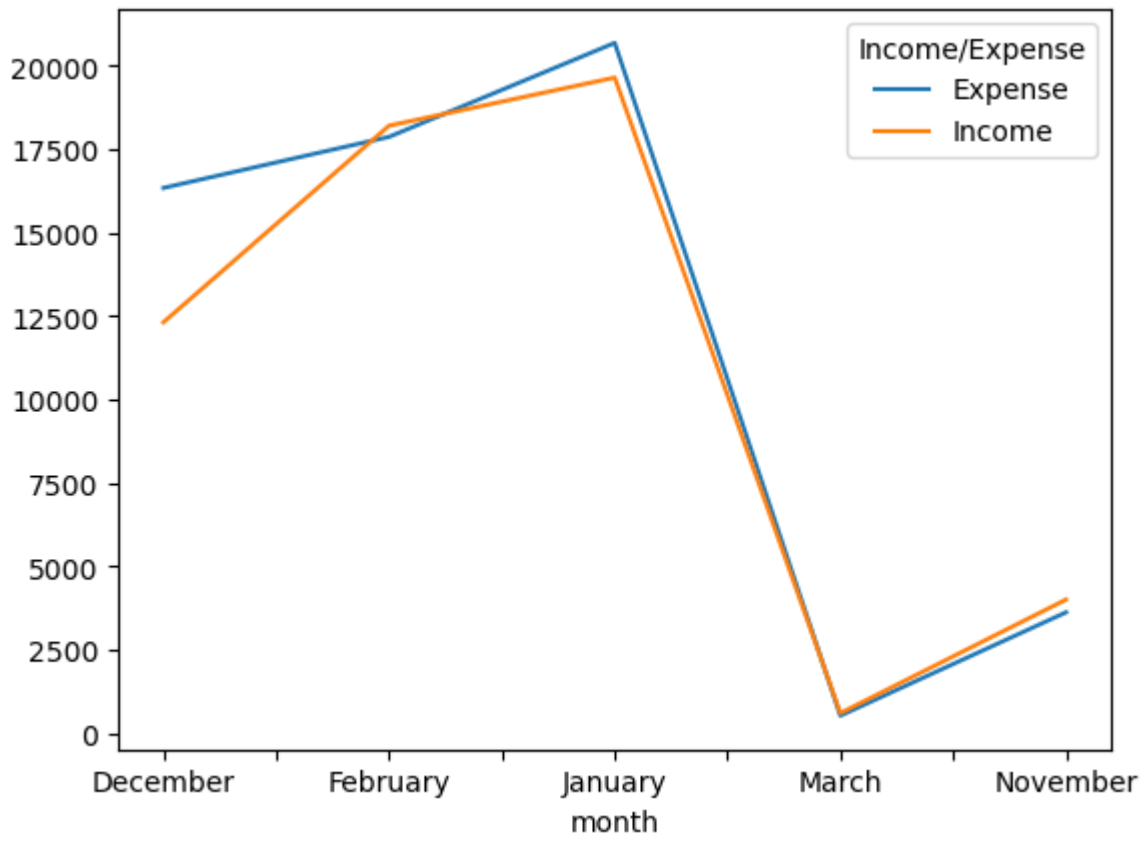
In [ ]:
```
df.head()
```

Out[ ]:

	Date	Account	Category	Subcategory	Note	INR	Income/Expense	Note.1	Amou
0	2022-03-02 10:11:00	CUB - online payment	Food	NaN	Brownie	50.0	Expense	NaN	50
1	2022-03-02 10:11:00	CUB - online payment	Other	NaN	To lended people	300.0	Expense	NaN	300
2	2022-03-01 19:50:00	CUB - online payment	Food	NaN	Dinner	78.0	Expense	NaN	78
3	2022-03-01 18:56:00	CUB - online payment	Transportation	NaN	Metro	30.0	Expense	NaN	30
4	2022-03-01 18:22:00	CUB - online payment	Food	NaN	Snacks	67.0	Expense	NaN	67

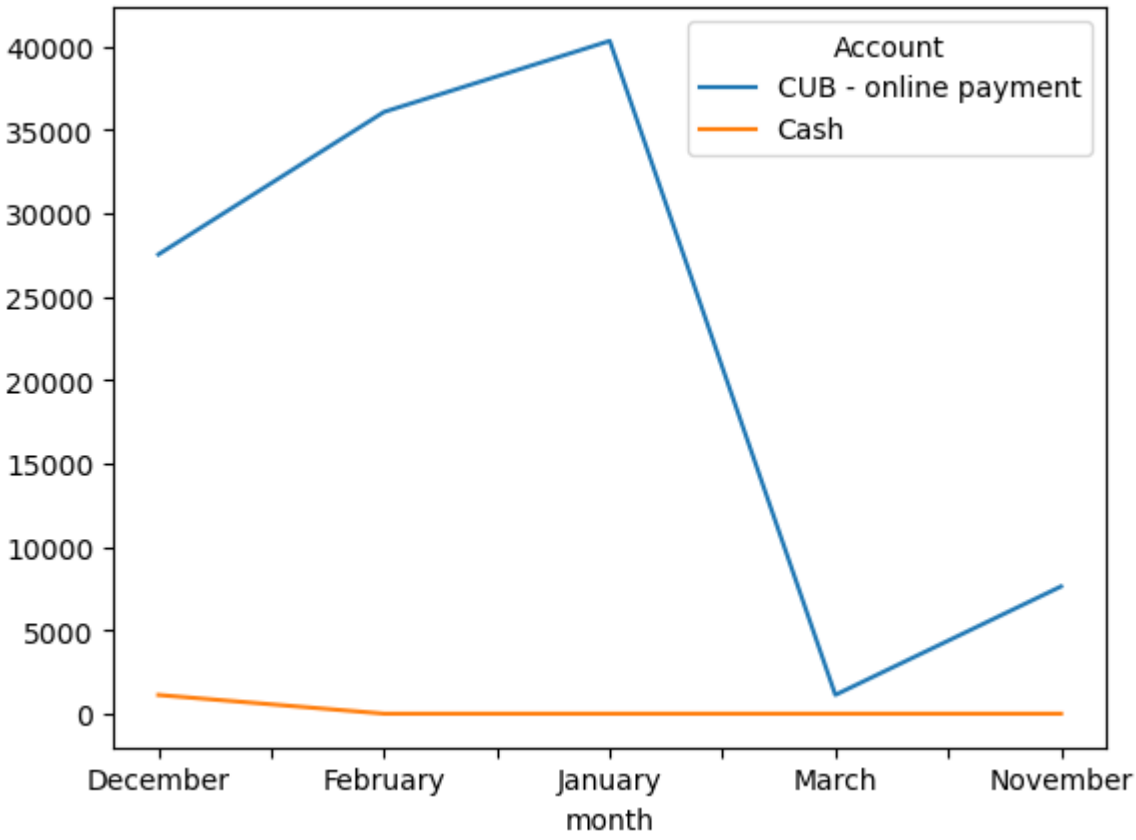
In [ ]:
```
df.pivot_table(index='month',columns='Income/Expense',values='INR',aggfunc='sum',fi
```

Out[ ]:
```
<Axes: xlabel='month'>
```



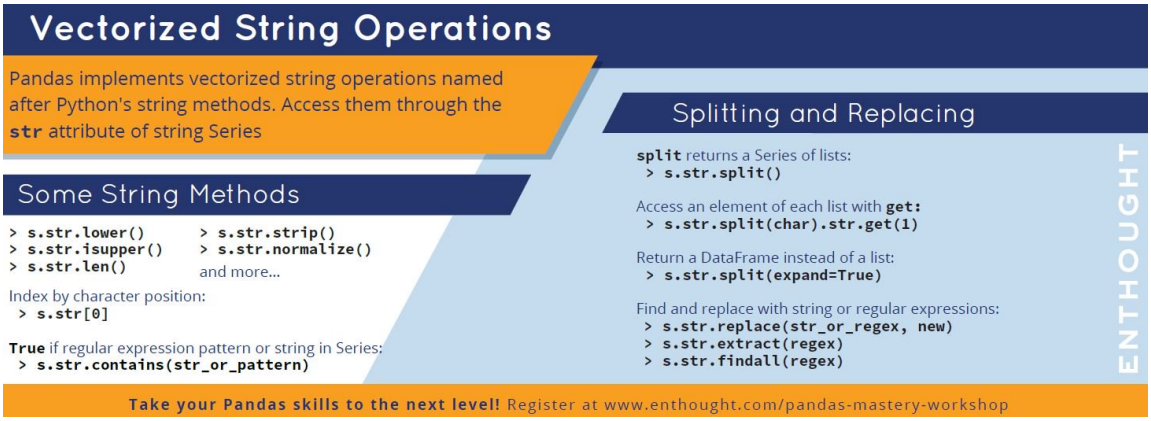
In [ ]:
```
df.pivot_table(index='month',columns='Account',values='INR',aggfunc='sum',fill_valu
```

Out[ ]:
```
<Axes: xlabel='month'>
```


Account
CUB - online payment
Cash
40000
35000
30000
25000
20000
15000
10000
5000
0
December
February
January
March
November
month

Vectorized String Opearations in Pandas

- Vectorized string operations in Pandas refer to the ability to apply string operations to a series or dataframe of strings in a single operation, rather than looping over each string individually. This can be achieved using the str attribute of a Series or DataFrame object, which provides a number of vectorized string methods.
- Vectorized string operations in Pandas refer to the ability to apply string functions and operations to entire arrays of strings (columns or Series containing strings) without the need for explicit loops or iteration. This is made possible by Pandas' integration with the NumPy library, which allows for efficient element-wise operations.
- When you have a Pandas DataFrame or Series containing string data, you can use various string methods that are applied to every element in the column simultaneously. This can significantly improve the efficiency and readability of your code. Some of the commonly used vectorized string operations in Pandas include methods like `.str.lower()`, `.str.upper()`, `.str.strip()`, `.str.replace()`, and many more.
- Vectorized string operations not only make your code more concise and readable but also often lead to improved performance compared to explicit for-loops, especially when dealing with large datasets.



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

In [ ]:
```
s = pd.Series(['cat','mat',None,'rat'])
```

In [ ]:
```
# str ->  string accessor
s.str.startswith('c')
```

Out[ ]:
```
0     True
1    False
2     None
3    False
dtype: object
```

Real-world Dataset - Titanic Dataset

In [ ]:
```
df = pd.read_csv('Data\Day44\Titanic.csv')
```

In [ ]:
```
df.head()
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN

In [ ]:
```
df['Name']
```

Out[ ]:
```
0                              Braund, Mr. Owen Harris
1    Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss. Laina
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
                           ...
886                              Montvila, Rev. Juozas
887                       Graham, Miss. Margaret Edith
888           Johnston, Miss. Catherine Helen "Carrie"
889                              Behr, Mr. Karl Howell
890                                Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

Common Functions

In [ ]:
```
# lower/upper/capitalize/title
df['Name'].str.upper()
df['Name'].str.capitalize()
df['Name'].str.title()
```

Out[ ]:
```
0                                Braund, Mr. Owen Harris
1      Cumings, Mrs. John Bradley (Florence Briggs Th...
2                                 Heikkinen, Miss. Laina
3           Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                               Allen, Mr. William Henry
                             ...
886                                Montvila, Rev. Juozas
887                         Graham, Miss. Margaret Edith
888             Johnston, Miss. Catherine Helen "Carrie"
889                                Behr, Mr. Karl Howell
890                                  Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

In [ ]:
```
# len

df['Name'][df['Name'].str.len()]
```

Out[ ]:
```
23                       Sloper, Mr. William Thompson
51                       Nosworthy, Mr. Richard Cater
22                        McGowan, Miss. Anna "Annie"
44                      Devaney, Miss. Margaret Delia
24                      Palsson, Miss. Torborg Danira
                            ...
21                             Beesley, Mr. Lawrence
28                      O'Dwyer, Miss. Ellen "Nellie"
40    Ahlin, Mrs. Johan (Johanna Persdotter Larsson)
21                             Beesley, Mr. Lawrence
19                           Masselmani, Mrs. Fatima
Name: Name, Length: 891, dtype: object
```

In [ ]:
```
df['Name'][df['Name'].str.len() == 82].values[0]
```

Out[ ]:
```
'Penasco y Castellana, Mrs. Victor de Satode (Maria Josefa Perez de Soto y Vallej
o)'
```

In [ ]:
```
# strip
df['Name'].str.strip()
```

Out[ ]:
```
0                                Braund, Mr. Owen Harris
1      Cumings, Mrs. John Bradley (Florence Briggs Th...
2                                 Heikkinen, Miss. Laina
3           Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                               Allen, Mr. William Henry
                             ...
886                                Montvila, Rev. Juozas
887                         Graham, Miss. Margaret Edith
888             Johnston, Miss. Catherine Helen "Carrie"
889                                Behr, Mr. Karl Howell
890                                  Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

In [ ]:
```
# split -> get
df['lastname'] = df['Name'].str.split(',').str.get(0)
df.head()
```

Follow for more AI content: https://lnkd.in/gaJtbwcu

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN

In [ ]:
```
df[['title','firstname']] = df['Name'].str.split(',').str.get(1).str.strip().str.sp
df.head()
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN

In [ ]:
```
df['title'].value_counts()
```

Out[ ]:
```
title
Mr.          517
Miss.        182
Mrs.         125
Master.       40
Dr.            7
Rev.           6
Mlle.          2
Major.         2
Col.           2
the            1
Capt.          1
Ms.            1
Sir.           1
Lady.          1
Mme.           1
Don.           1
Jonkheer.      1
Name: count, dtype: int64
```

In [ ]:
```
# replace
df['title'] = df['title'].str.replace('Ms.','Miss.')
df['title'] = df['title'].str.replace('Mlle.','Miss.')
```

In [ ]:
```
df['title'].value_counts()
```

Out[ ]:
```
title
Mr.          517
Miss.        185
Mrs.         125
Master.       40
Dr.            7
Rev.           6
Major.         2
Col.           2
Don.           1
Mme.           1
Lady.          1
Sir.           1
Capt.          1
the            1
Jonkheer.      1
Name: count, dtype: int64
```

filtering

In [ ]:
```
# startswith/endswith
df[df['firstname'].str.endswith('A')]
```

Out[ ]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
64	65	0	1	Stewart, Mr. Albert A	male	NaN	0	0	PC 17605	27.7208	NaN
303	304	1	2	Keane, Miss. Nora A	female	NaN	0	0	226593	12.3500	E101

In [ ]:
```
# isdigit/isalpha...
df[df['firstname'].str.isdigit()]
```

Out[ ]:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked

slicing

In [ ]:
```
df['Name'].str[::-1]
```

Out[ ]:
```
0                                sirraH newO .rM ,dnuarB
1      )reyahT sggirB ecnerolF( yeldarB nhoJ .srM ,sg...
2                                 aniaL .ssiM ,nenikkieH
3           )leeP yaM yliL( htaeH seuqcaJ .srM ,ellertuF
4                               yrneH mailliW .rM ,nellA
                              ...
886                                sazouJ .veR ,alivtnoM
887                           htidE teragraM .ssiM ,maharG
888              "eirraC" neleH enirehtaC .ssiM ,notsnhoJ
889                                llewoH lraK .rM ,rheB
890                                  kcirtaP .rM ,yelooD
Name: Name, Length: 891, dtype: object
```

Date and Time in Pandas

In Pandas, you can work with dates and times using the `datetime` data type. Pandas provides several data structures and functions for handling date and time data, making it convenient for time series data analysis.

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

1. `Timestamp`:

This represents a single timestamp and is the fundamental data type for time series data in Pandas.

Time stamps reference particular moments in time (e.g., Oct 24th, 2022 at 7:00pm)

In [ ]:
```
# Creating a Timestamp object
pd.Timestamp('2023/1/5')
```

Out[ ]:
```
Timestamp('2023-01-05 00:00:00')
```

In [ ]:
```
# variations
pd.Timestamp('2023-1-5')
pd.Timestamp('2023, 1, 5')
```

Out[ ]:
```
Timestamp('2023-01-05 00:00:00')
```

In [ ]:
```
# only year
pd.Timestamp('2023')
```

Out[ ]:
```
Timestamp('2023-01-01 00:00:00')
```

In [ ]:
```
# using text
pd.Timestamp('5th January 2023')
```

Out[ ]:
```
Timestamp('2023-01-05 00:00:00')
```

In [ ]:
```
# providing time also
pd.Timestamp('5th January 2023 9:21AM')
```

Out[ ]:
```
Timestamp('2023-01-05 09:21:00')
```

Using Python Datetime Object

In [ ]:
```
# using datetime.datetime object
import datetime as dt

x = pd.Timestamp(dt.datetime(2023,1,5,9,21,56))
x
```

Out[ ]:
```
Timestamp('2023-01-05 09:21:56')
```

In [ ]:
```
# fetching attributes
x.year
```

Out[ ]:
```
2023
```

In [ ]:
```
x.month
```

Out[ ]:
```
1
```

In [ ]:
```
x.day
x.hour
x.minute
x.second
```

Out[ ]:
```
56
```

why separate objects to handle data and time when python already has datetime functionality?

Python's `datetime` module provides a comprehensive way to work with dates and times, and it is a fundamental part of Python's standard library. However, Pandas introduces separate objects to handle dates and times for several reasons:

1. **Efficiency**: The `datetime` module in Python is flexible and comprehensive, but it may not be as efficient when dealing with large datasets. Pandas' datetime objects are optimized for performance and are designed for working with data, making them more suitable for operations on large time series datasets.

2. **Data Alignment**: Pandas focuses on data manipulation and analysis, so it provides tools for aligning data with time-based indices and working with irregular time series. This is particularly useful in financial and scientific data analysis.

3. **Convenience**: Pandas provides a high-level API for working with time series data, which can make your code more concise and readable. It simplifies common operations such as resampling, aggregation, and filtering.

4. **Integration with DataFrames**: Pandas seamlessly integrates its date and time objects with DataFrames. This integration allows you to easily create, manipulate, and analyze time series data within the context of your data analysis tasks.

5. **Time Zones**: Pandas has built-in support for handling time zones and daylight saving time, making it more suitable for working with global datasets and international time series data.

6. **Frequency-Based Data**: Pandas introduces concepts like `Period` and `PeriodIndex` to work with fixed-frequency time data, which is common in various applications, such as financial time series analysis.

While Python's `datetime` module is powerful and flexible, it is a general-purpose module and is not specifically optimized for data analysis and manipulation. Pandas complements Python's `datetime` module by providing a more data-centric and efficient approach to working with dates and times, especially within the context of data analysis and time series data. This separation of functionality allows Pandas to offer a more streamlined and efficient experience when dealing with time-related data in the realm of data science and data analysis.

2. `DatetimeIndex`:

This is an index that consists of `Timestamp` objects. It is used to create time series data in Pandas DataFrames.

In [ ]:
```
# from strings
pd.DatetimeIndex(['2023/1/1','2022/1/1','2021/1/1'])
```

Out[ ]:
```
DatetimeIndex(['2023-01-01', '2022-01-01', '2021-01-01'], dtype='datetime64[ns]',
freq=None)
```

In [ ]:
```
# from strings
type(pd.DatetimeIndex(['2023/1/1','2022/1/1','2021/1/1']))
```

Out[ ]:
```
pandas.core.indexes.datetimes.DatetimeIndex
```

In [ ]:
```
# using python datetime object
pd.DatetimeIndex([dt.datetime(2023,1,1),dt.datetime(2022,1,1),dt.datetime(2021,1,1)
```

Out[ ]:
```
DatetimeIndex(['2023-01-01', '2022-01-01', '2021-01-01'], dtype='datetime64[ns]',
freq=None)
```

In [ ]:
```
# using pd.timestamps
dt_index = pd.DatetimeIndex([pd.Timestamp(2023,1,1),pd.Timestamp(2022,1,1),pd.Times
```

In [ ]:
```
dt_index
```

Out[ ]:
```
DatetimeIndex(['2023-01-01', '2022-01-01', '2021-01-01'], dtype='datetime64[ns]',
freq=None)
```

In [ ]:
```
# using datatimeindex as series index

pd.Series([1,2,3],index=dt_index)
```

Out[ ]:
```
2023-01-01    1
2022-01-01    2
2021-01-01    3
dtype: int64
```

3. date_range function

In [ ]:
```
# generate daily dates in a given range
pd.date_range(start='2023/1/5',end='2023/2/28',freq='D')
```

Out[ ]:
```
DatetimeIndex(['2023-01-05', '2023-01-06', '2023-01-07', '2023-01-08',
               '2023-01-09', '2023-01-10', '2023-01-11', '2023-01-12',
               '2023-01-13', '2023-01-14', '2023-01-15', '2023-01-16',
               '2023-01-17', '2023-01-18', '2023-01-19', '2023-01-20',
               '2023-01-21', '2023-01-22', '2023-01-23', '2023-01-24',
               '2023-01-25', '2023-01-26', '2023-01-27', '2023-01-28',
               '2023-01-29', '2023-01-30', '2023-01-31', '2023-02-01',
               '2023-02-02', '2023-02-03', '2023-02-04', '2023-02-05',
               '2023-02-06', '2023-02-07', '2023-02-08', '2023-02-09',
               '2023-02-10', '2023-02-11', '2023-02-12', '2023-02-13',
               '2023-02-14', '2023-02-15', '2023-02-16', '2023-02-17',
               '2023-02-18', '2023-02-19', '2023-02-20', '2023-02-21',
               '2023-02-22', '2023-02-23', '2023-02-24', '2023-02-25',
               '2023-02-26', '2023-02-27', '2023-02-28'],
              dtype='datetime64[ns]', freq='D')
```

In [ ]:
```
# alternate days in a given range
pd.date_range(start='2023/1/5',end='2023/2/28',freq='2D')
```

Out[ ]:
```
DatetimeIndex(['2023-01-05', '2023-01-07', '2023-01-09', '2023-01-11',
               '2023-01-13', '2023-01-15', '2023-01-17', '2023-01-19',
               '2023-01-21', '2023-01-23', '2023-01-25', '2023-01-27',
               '2023-01-29', '2023-01-31', '2023-02-02', '2023-02-04',
               '2023-02-06', '2023-02-08', '2023-02-10', '2023-02-12',
               '2023-02-14', '2023-02-16', '2023-02-18', '2023-02-20',
               '2023-02-22', '2023-02-24', '2023-02-26', '2023-02-28'],
              dtype='datetime64[ns]', freq='2D')
```

In [ ]:
```
# B -> business days
pd.date_range(start='2023/1/5',end='2023/2/28',freq='B')
```

Out[ ]:
```
DatetimeIndex(['2023-01-05', '2023-01-06', '2023-01-09', '2023-01-10',
               '2023-01-11', '2023-01-12', '2023-01-13', '2023-01-16',
               '2023-01-17', '2023-01-18', '2023-01-19', '2023-01-20',
               '2023-01-23', '2023-01-24', '2023-01-25', '2023-01-26',
               '2023-01-27', '2023-01-30', '2023-01-31', '2023-02-01',
               '2023-02-02', '2023-02-03', '2023-02-06', '2023-02-07',
               '2023-02-08', '2023-02-09', '2023-02-10', '2023-02-13',
               '2023-02-14', '2023-02-15', '2023-02-16', '2023-02-17',
               '2023-02-20', '2023-02-21', '2023-02-22', '2023-02-23',
               '2023-02-24', '2023-02-27', '2023-02-28'],
              dtype='datetime64[ns]', freq='B')
```

In [ ]:
```
# W -> one week per day
pd.date_range(start='2023/1/5',end='2023/2/28',freq='W-THU')
```

Out[ ]:
```
DatetimeIndex(['2023-01-05', '2023-01-12', '2023-01-19', '2023-01-26',
               '2023-02-02', '2023-02-09', '2023-02-16', '2023-02-23'],
              dtype='datetime64[ns]', freq='W-THU')
```

In [ ]:
```
# H -> Hourly data(factor)
pd.date_range(start='2023/1/5',end='2023/2/28',freq='6H')
```

Out[ ]:
```
DatetimeIndex(['2023-01-05 00:00:00', '2023-01-05 06:00:00',
               '2023-01-05 12:00:00', '2023-01-05 18:00:00',
               '2023-01-06 00:00:00', '2023-01-06 06:00:00',
               '2023-01-06 12:00:00', '2023-01-06 18:00:00',
               '2023-01-07 00:00:00', '2023-01-07 06:00:00',
               ...
               '2023-02-25 18:00:00', '2023-02-26 00:00:00',
               '2023-02-26 06:00:00', '2023-02-26 12:00:00',
               '2023-02-26 18:00:00', '2023-02-27 00:00:00',
               '2023-02-27 06:00:00', '2023-02-27 12:00:00',
               '2023-02-27 18:00:00', '2023-02-28 00:00:00'],
              dtype='datetime64[ns]', length=217, freq='6H')
```

In [ ]:
```
# M -> Month end
pd.date_range(start='2023/1/5',end='2023/2/28',freq='M')
```

Out[ ]:
```
DatetimeIndex(['2023-01-31', '2023-02-28'], dtype='datetime64[ns]', freq='M')
```

In [ ]:
```
# A -> Year end
pd.date_range(start='2023/1/5',end='2030/2/28',freq='A')
```

Out[ ]:
```
DatetimeIndex(['2023-12-31', '2024-12-31', '2025-12-31', '2026-12-31',
               '2027-12-31', '2028-12-31', '2029-12-31'],
              dtype='datetime64[ns]', freq='A-DEC')
```

In [ ]:
```
# MS -> Month start
pd.date_range(start='2023/1/5',end='2023/2/28',freq='MS')
```

Out[ ]:
```
DatetimeIndex(['2023-02-01'], dtype='datetime64[ns]', freq='MS')
```

4. to_datetime function

converts an existing objects to pandas timestamp/datetimeindex object

In [ ]:
```
# simple series example

s = pd.Series(['2023/1/1','2022/1/1','2021/1/1'])
pd.to_datetime(s).dt.day_name()
```

Out[ ]:
```
0      Sunday
1    Saturday
2      Friday
dtype: object
```

In [ ]:
```
# with errors
s = pd.Series(['2023/1/1','2022/1/1','2021/130/1'])
pd.to_datetime(s,errors='coerce').dt.month_name()
```

Out[ ]:
```
0    January
1    January
2        NaN
dtype: object
```

In [ ]:
```
df = pd.read_csv('Data\Day43\expense_data.csv')
df.shape
```

Out[ ]:
```
(277, 11)
```

In [ ]:
```
df.head()
```

Out[ ]:

	Date	Account	Category	Subcategory	Note	INR	Income/Expense	Note.1	Amou
0	3/2/2022 10:11	CUB - online payment	Food	NaN	Brownie	50.0	Expense	NaN	5(
1	3/2/2022 10:11	CUB - online payment	Other	NaN	To lended people	300.0	Expense	NaN	30(
2	3/1/2022 19:50	CUB - online payment	Food	NaN	Dinner	78.0	Expense	NaN	7(
3	3/1/2022 18:56	CUB - online payment	Transportation	NaN	Metro	30.0	Expense	NaN	3(
4	3/1/2022 18:22	CUB - online payment	Food	NaN	Snacks	67.0	Expense	NaN	6(

In [ ]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 277 entries, 0 to 276
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   Date            277 non-null    object 
 1   Account         277 non-null    object 
 2   Category        277 non-null    object 
 3   Subcategory     0 non-null      float64
 4   Note            273 non-null    object 
 5   INR             277 non-null    float64
 6   Income/Expense  277 non-null    object 
 7   Note.1          0 non-null      float64
 8   Amount          277 non-null    float64
 9   Currency        277 non-null    object 
 10  Account.1       277 non-null    float64
dtypes: float64(5), object(6)
memory usage: 23.9+ KB
```

In [ ]:
```
df['Date'] = pd.to_datetime(df['Date'])
```

In [ ]:
```
df.info()
```

Follow for more AI content: https://lnkd.in/gaJtbwcu

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 277 entries, 0 to 276
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   Date            277 non-null    datetime64[ns]
 1   Account         277 non-null    object
 2   Category        277 non-null    object
 3   Subcategory     0 non-null      float64
 4   Note            273 non-null    object
 5   INR             277 non-null    float64
 6   Income/Expense  277 non-null    object
 7   Note.1          0 non-null      float64
 8   Amount          277 non-null    float64
 9   Currency        277 non-null    object
 10  Account.1       277 non-null    float64
dtypes: datetime64[ns](1), float64(5), object(5)
memory usage: 23.9+ KB
```

5. `dt accessor`

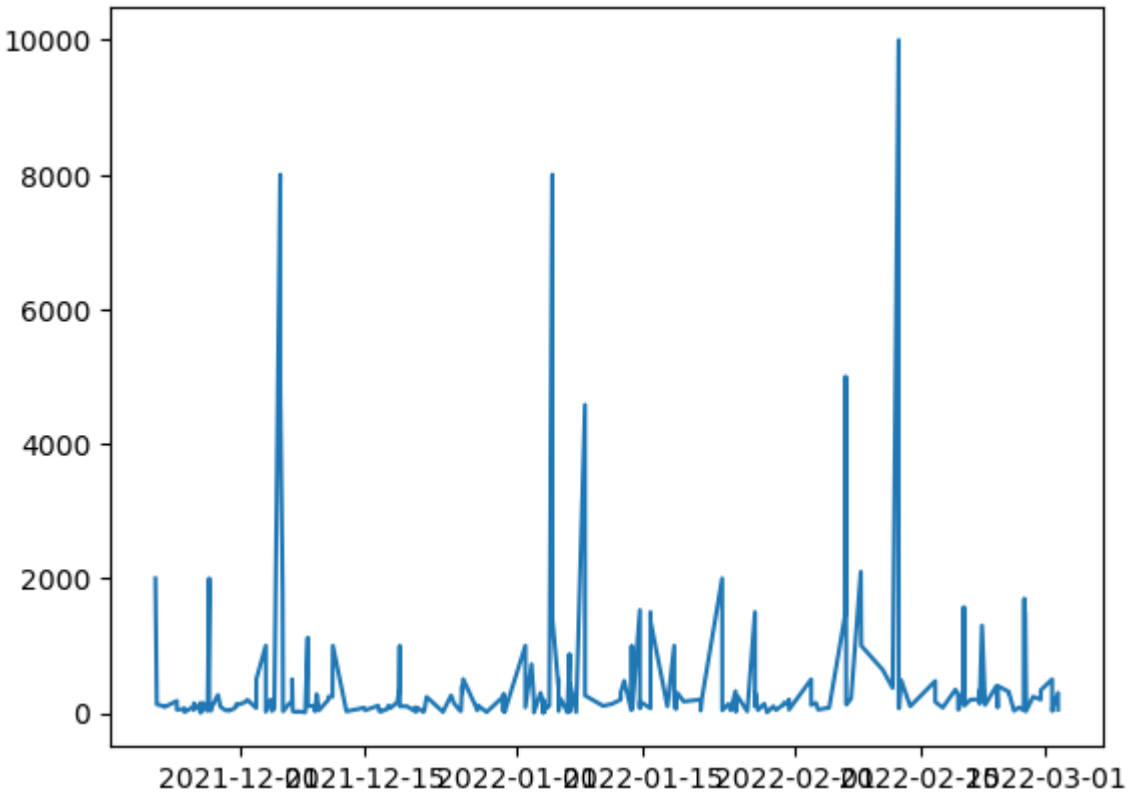
Accessor object for datetimelike properties of the Series values.

In [ ]:
```
df['Date'].dt.is_quarter_start
```

Out[ ]:
```
0      False
1      False
2      False
3      False
4      False
       ...
272    False
273    False
274    False
275    False
276    False
Name: Date, Length: 277, dtype: bool
```

In [ ]:
```
# plot graph
import matplotlib.pyplot as plt
plt.plot(df['Date'],df['INR'])
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x181faeba430>]
```



In [ ]:
```
# day name wise bar chart/month wise bar chart

df['day_name'] = df['Date'].dt.day_name()
```

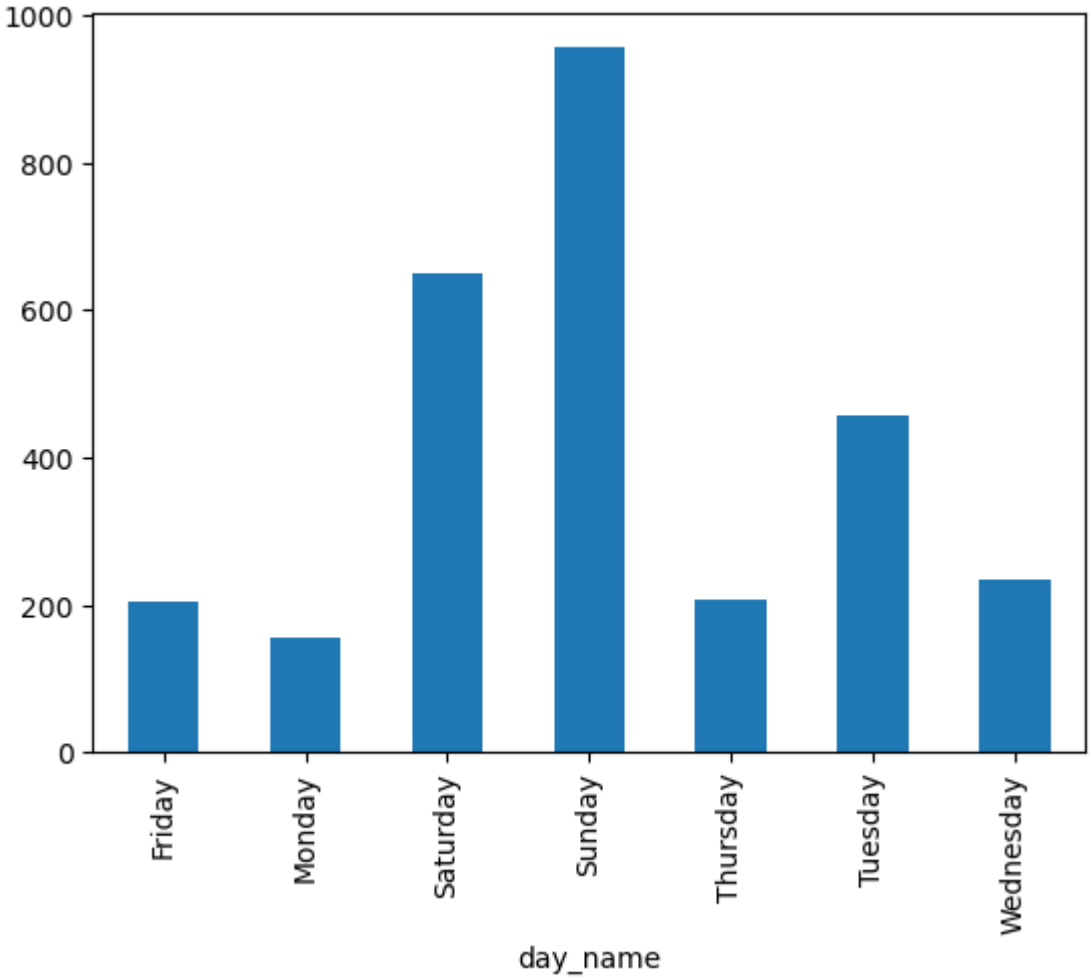
In [ ]:
```
df.head()
```

Out[ ]:

	Date	Account	Category	Subcategory	Note	INR	Income/Expense	Note.1	Amou
0	2022-03-02 10:11:00	CUB - online payment	Food	NaN	Brownie	50.0	Expense	NaN	50
1	2022-03-02 10:11:00	CUB - online payment	Other	NaN	To lended people	300.0	Expense	NaN	300
2	2022-03-01 19:50:00	CUB - online payment	Food	NaN	Dinner	78.0	Expense	NaN	78
3	2022-03-01 18:56:00	CUB - online payment	Transportation	NaN	Metro	30.0	Expense	NaN	30
4	2022-03-01 18:22:00	CUB - online payment	Food	NaN	Snacks	67.0	Expense	NaN	67

In [ ]:
```
df.groupby('day_name')['INR'].mean().plot(kind='bar')
```

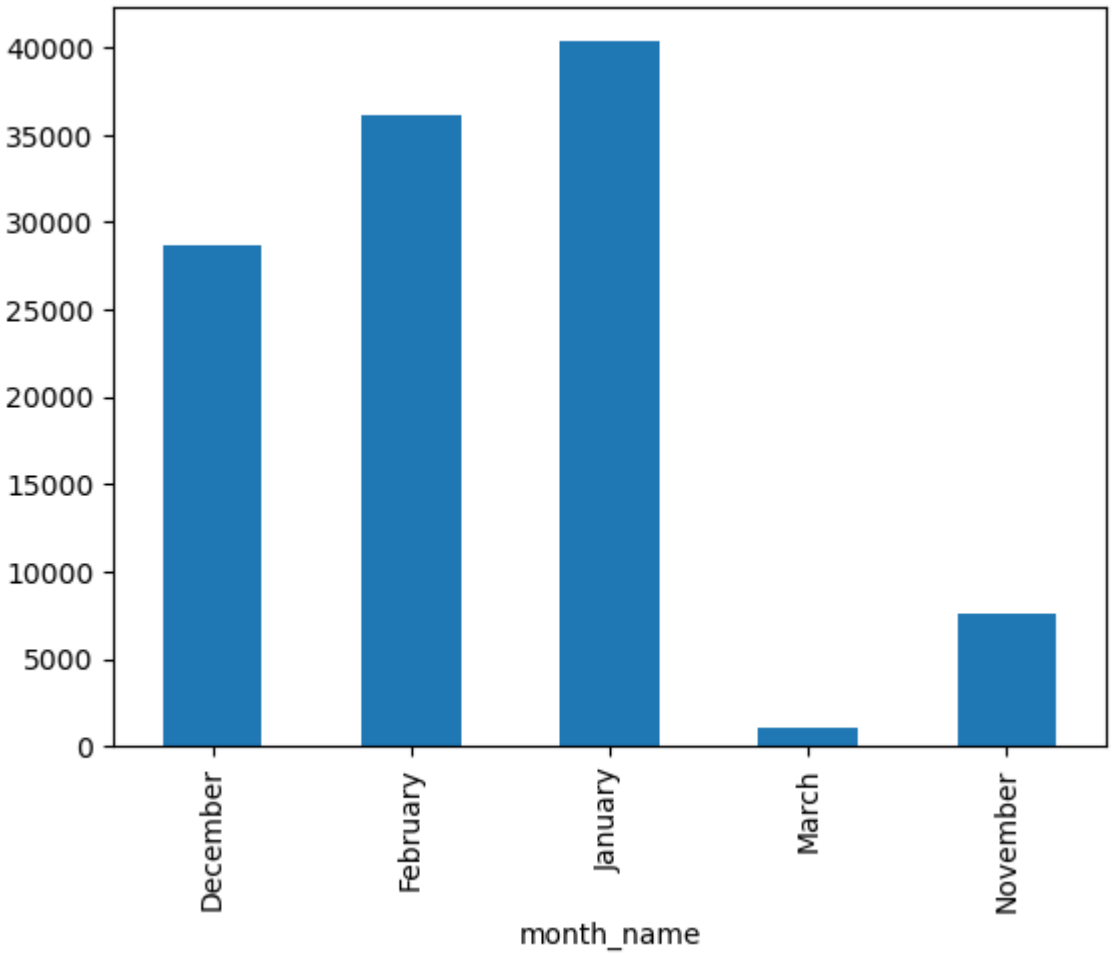
Out[ ]:
```
<Axes: xlabel='day_name'>
```



In [ ]:
```
df['month_name'] = df['Date'].dt.month_name()
```

In [ ]:
```
df.groupby('month_name')['INR'].sum().plot(kind='bar')
```

Out[ ]:
```
<Axes: xlabel='month_name'>
```



Pandas also provides powerful time series functionality, including the ability to resample, group, and perform various time-based operations on data. You can work with date and time data in Pandas to analyze and manipulate time series data effectively.

2D Line Plot in Matplotlib

Firstly,

What is Matplotlib?

Matplotlib is a popular data visualization library in Python. It provides a wide range of tools for creating various types of plots and charts, making it a valuable tool for data analysis, scientific research, and data presentation. Matplotlib allows you to create high-quality, customizable plots and figures for a variety of purposes, including line plots, bar charts, scatter plots, histograms, and more.

Matplotlib is highly customizable and can be used to control almost every aspect of your plots, from the colors and styles to labels and legends. It provides both a functional and an object-oriented interface for creating plots, making it suitable for a wide range of users, from beginners to advanced data scientists and researchers.

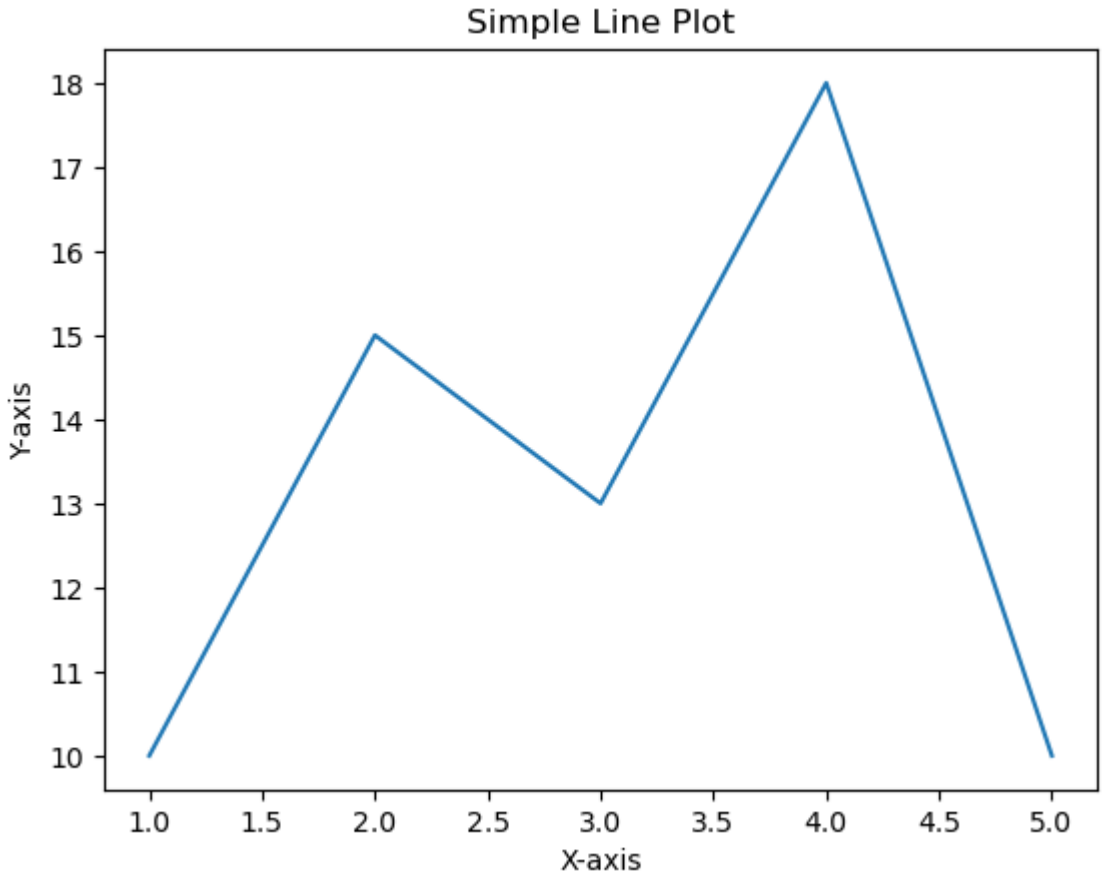
Matplotlib can be used in various contexts, including Jupyter notebooks, standalone Python scripts, and integration with web applications and GUI frameworks. It also works well with other Python libraries commonly used in data analysis and scientific computing, such as NumPy and Pandas.

To use Matplotlib, you typically need to import the library in your Python code, create the desired plot or chart, and then display or save it as needed. Here's a simple example of creating a basic line plot using Matplotlib:

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

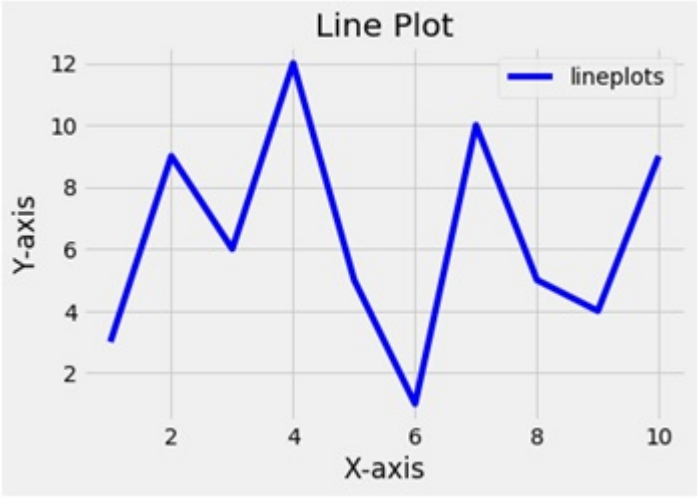
In [ ]:
```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 10]
```

```
# Create a line plot
plt.plot(x, y)

# Add labels and a title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')

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



This is just a basic introduction to Matplotlib. The library is quite versatile, and you can explore its documentation and tutorials to learn more about its capabilities and how to create various types of visualizations for your data.

Line Plot

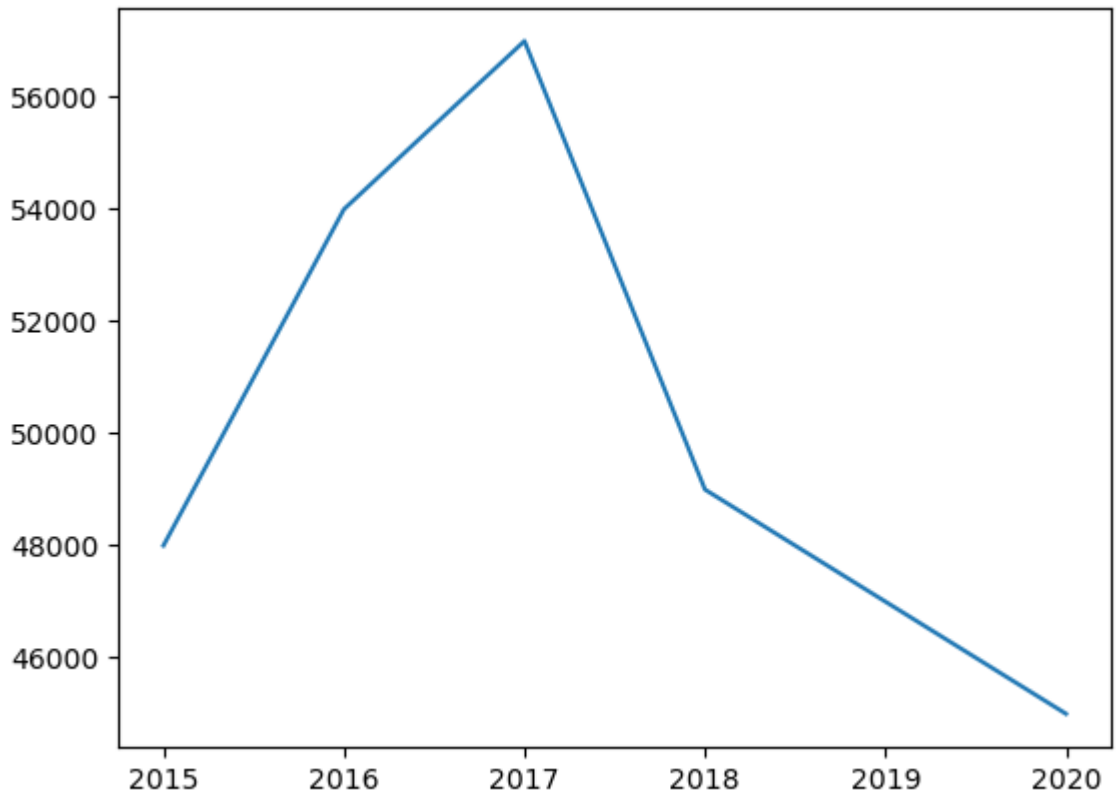


A 2D line plot is one of the most common types of plots in Matplotlib. It's used to visualize data with two continuous variables, typically representing one variable on the x-axis and another on the y-axis, and connecting the data points with lines. This type of plot is useful for showing trends, relationships, or patterns in data over a continuous range.

- Bivariate Analysis
- categorical -> numerical and numerical -> numerical
- Use case - Time series data

In [ ]:
```
# plotting simple graphs

price = [48000,54000,57000,49000,47000,45000]
year = [2015,2016,2017,2018,2019,2020]

plt.plot(year,price)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x28b7c3742e0>]
```



Real-world Dataset

In [ ]:
```
batsman = pd.read_csv('Data\Day45\sharma-kohli.csv')
```

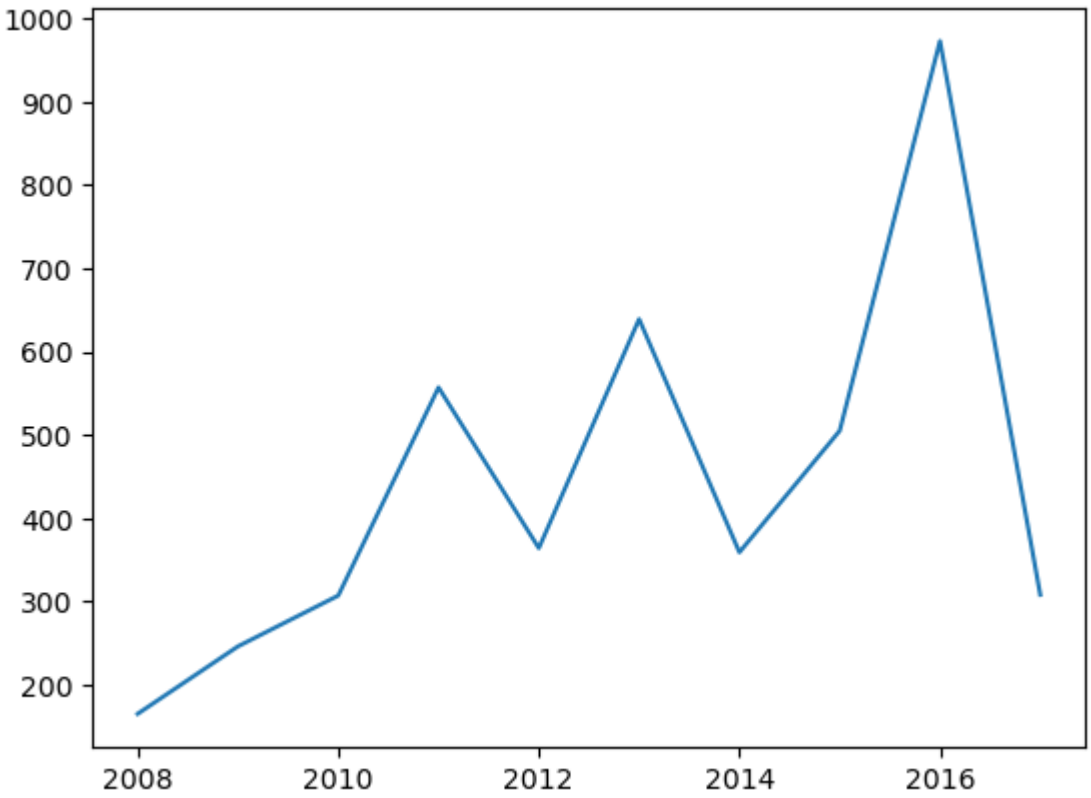
In [ ]:
```
batsman.head()
```

Out[ ]:

	index	RG Sharma	V Kohli
0	2008	404	165
1	2009	362	246
2	2010	404	307
3	2011	372	557
4	2012	433	364

In [ ]:
```
# plot the graph
plt.plot(batsman['index'],batsman['V Kohli'])
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x28b7c40ca60>]
```

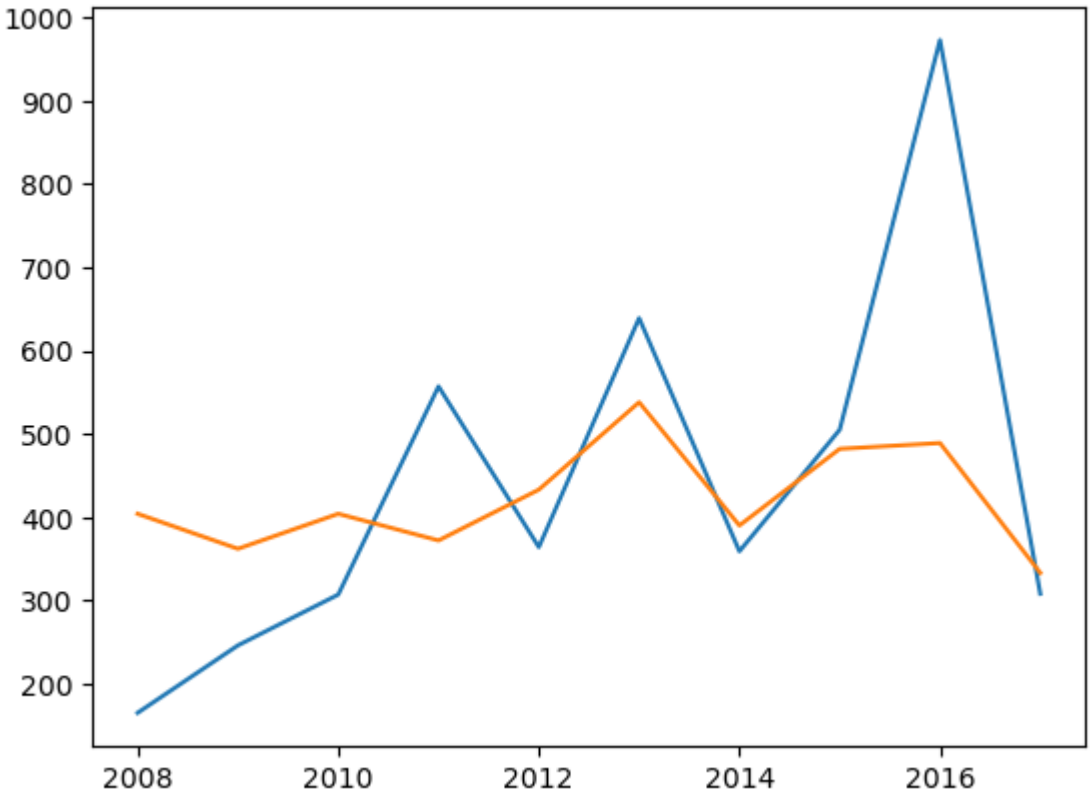


Multiple Plots:

It's possible to create multiple lines on a single plot, making it easy to compare multiple datasets or variables. In the example, both Rohit Sharma's and Virat Kohli's career runs are plotted on the same graph.

In [ ]:
```
# plotting multiple plots
plt.plot(batsman['index'],batsman['V Kohli'])
plt.plot(batsman['index'],batsman['RG Sharma'])
```

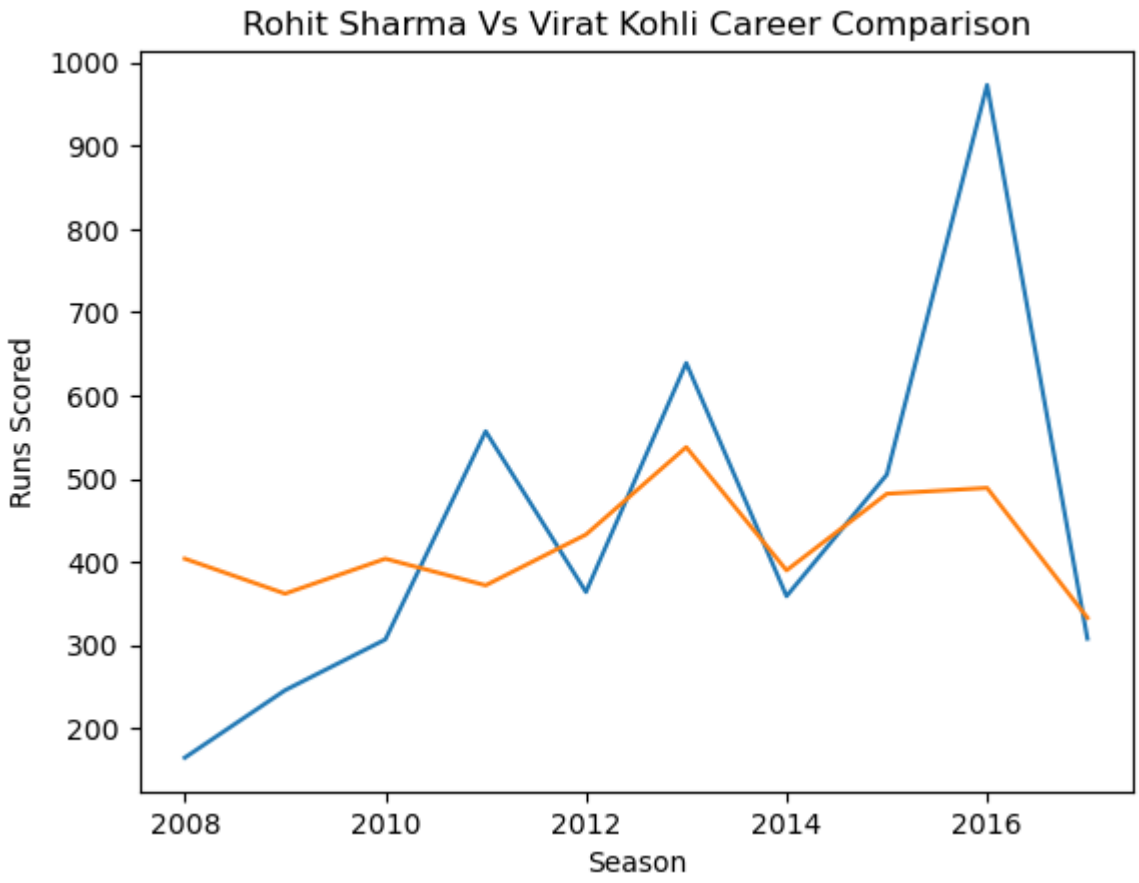
Out[ ]:
```
[<matplotlib.lines.Line2D at 0x28b7d4e2610>]
```



In [ ]:
```
# Labels title
plt.plot(batsman['index'],batsman['V Kohli'])
plt.plot(batsman['index'],batsman['RG Sharma'])

plt.title('Rohit Sharma Vs Virat Kohli Career Comparison')
plt.xlabel('Season')
plt.ylabel('Runs Scored')
```

Out[ ]:
```
Text(0, 0.5, 'Runs Scored')
```

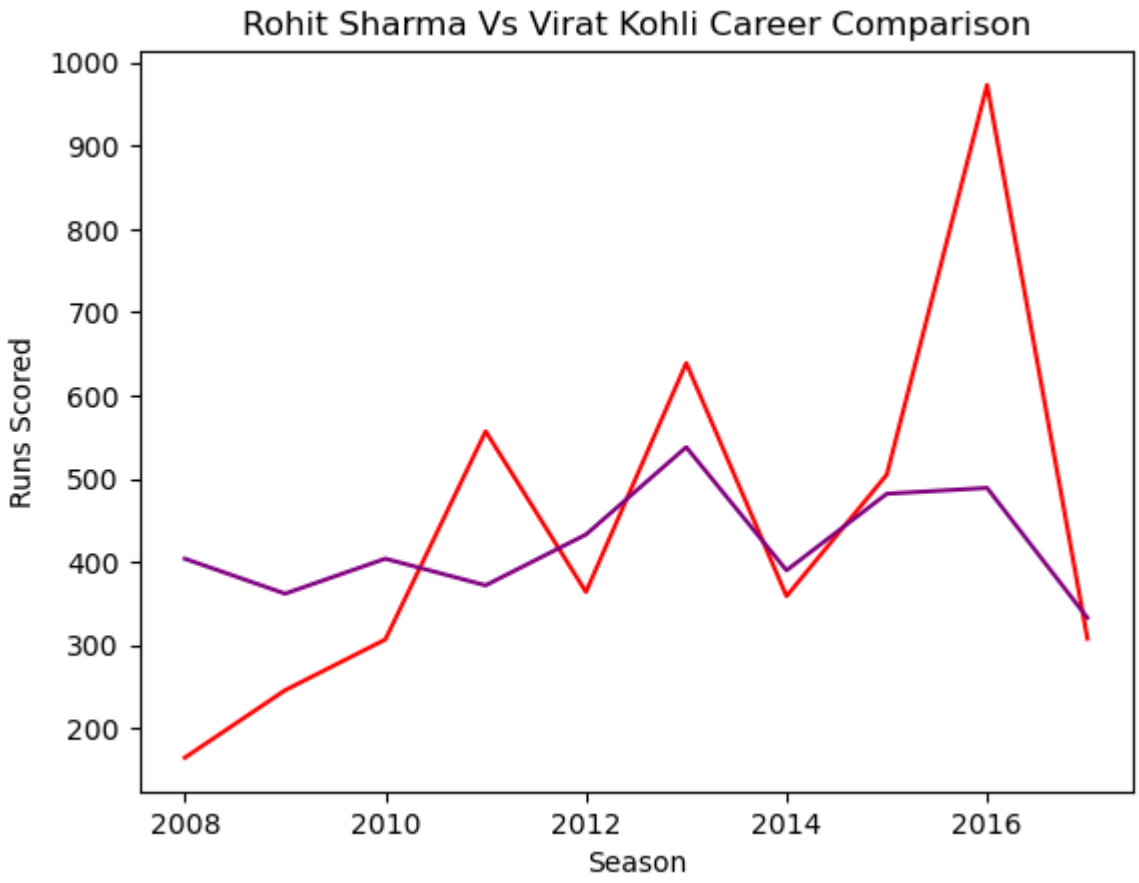
Follow for more AI content: https://lnkd.in/gaJtbwcu



colors(hex) and line(width and style) and marker(size)

In [ ]:
```
#colors
plt.plot(batsman['index'],batsman['V Kohli'],color='Red')
plt.plot(batsman['index'],batsman['RG Sharma'],color='Purple')

plt.title('Rohit Sharma Vs Virat Kohli Career Comparison')
plt.xlabel('Season')
plt.ylabel('Runs Scored')
```

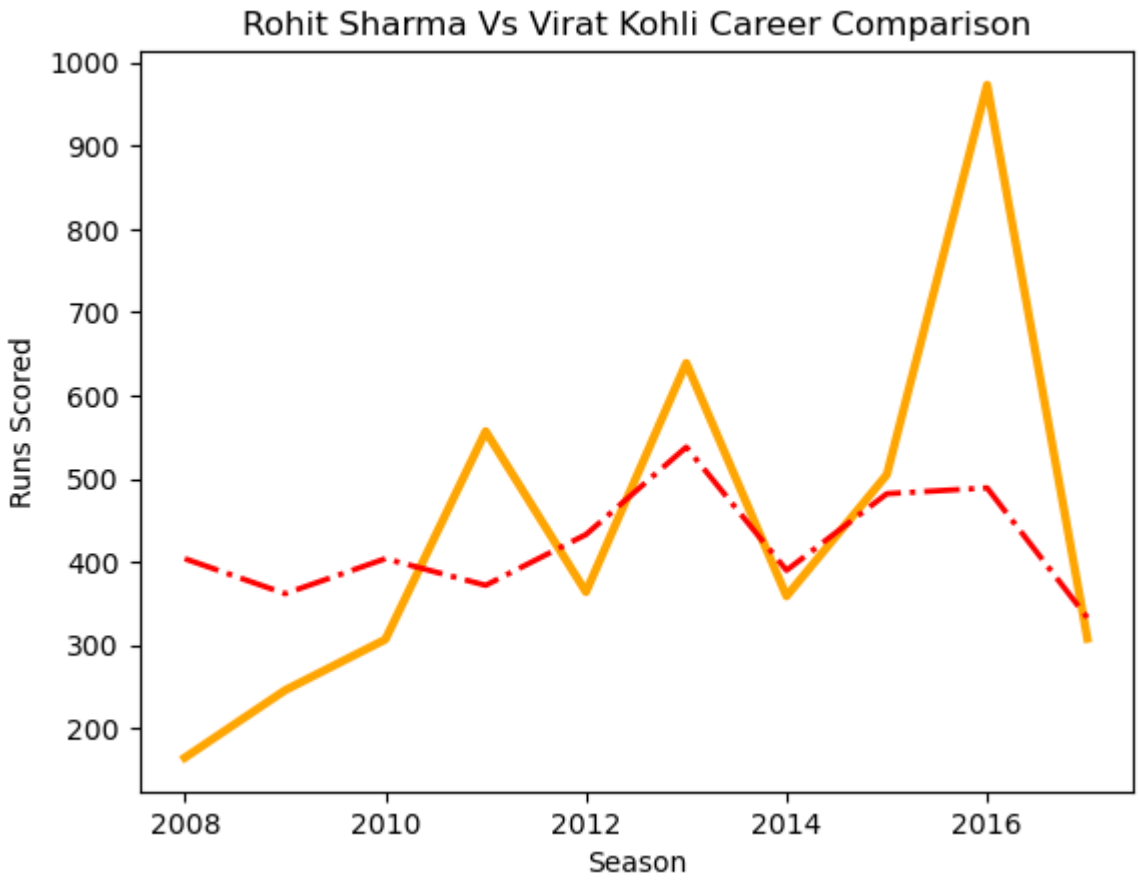
Out[ ]:
```
Text(0, 0.5, 'Runs Scored')
```



You can specify different colors for each line in the plot. In the example, colors like 'Red' and 'Purple' are used to differentiate the lines.

In [ ]:
```
# linestyle and linewidth
plt.plot(batsman['index'],batsman['V Kohli'],color='Orange',linestyle='solid',linew
plt.plot(batsman['index'],batsman['RG Sharma'],color='Red',linestyle='dashdot',line

plt.title('Rohit Sharma Vs Virat Kohli Career Comparison')
plt.xlabel('Season')
plt.ylabel('Runs Scored')
```

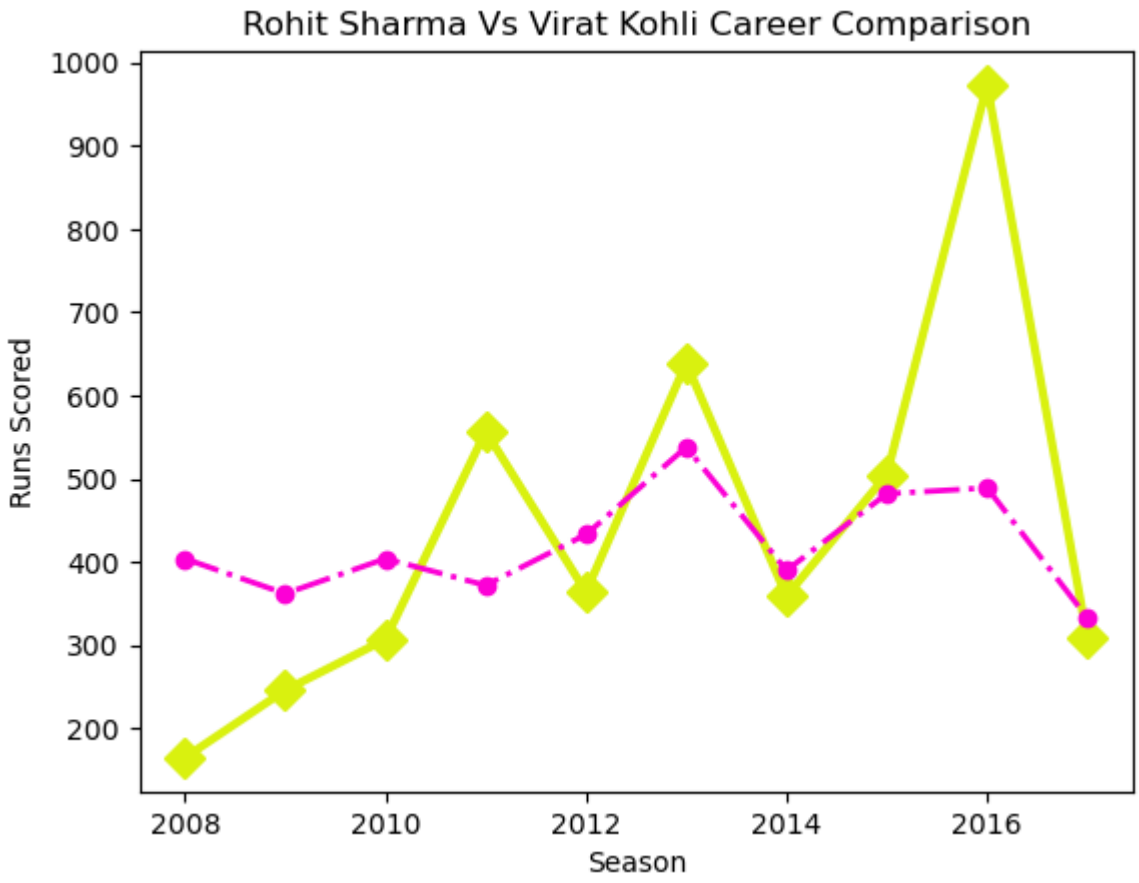
Out[ ]:
```
Text(0, 0.5, 'Runs Scored')
```



You can change the style and width of the lines. Common line styles include 'solid,' 'dotted,' 'dashed,' etc. In the example, 'solid' and 'dashdot' line styles are used.

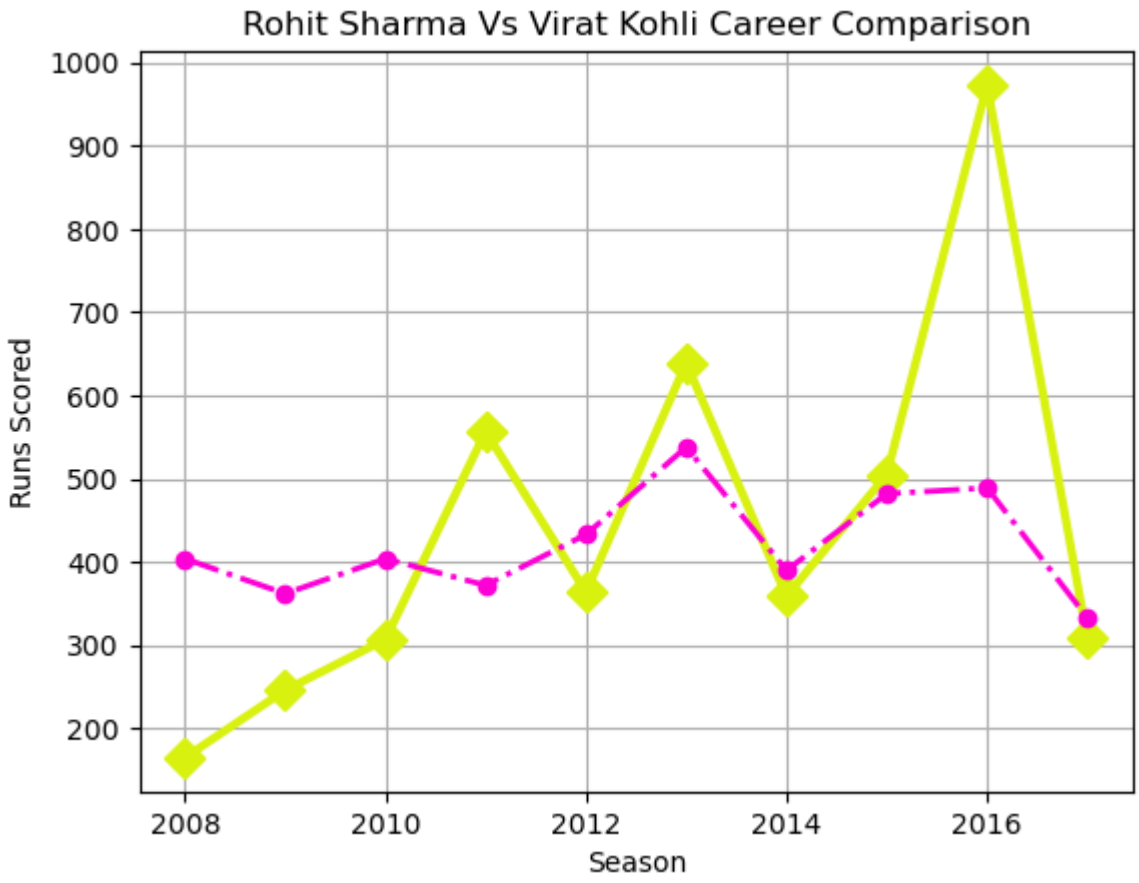
In [ ]:
```
# Marker
plt.plot(batsman['index'],batsman['V Kohli'],color='#D9F10F',linestyle='solid',line
plt.plot(batsman['index'],batsman['RG Sharma'],color='#FC00D6',linestyle='dashdot',

plt.title('Rohit Sharma Vs Virat Kohli Career Comparison')
plt.xlabel('Season')
plt.ylabel('Runs Scored')
```

Out[ ]:
```
Text(0, 0.5, 'Runs Scored')
```



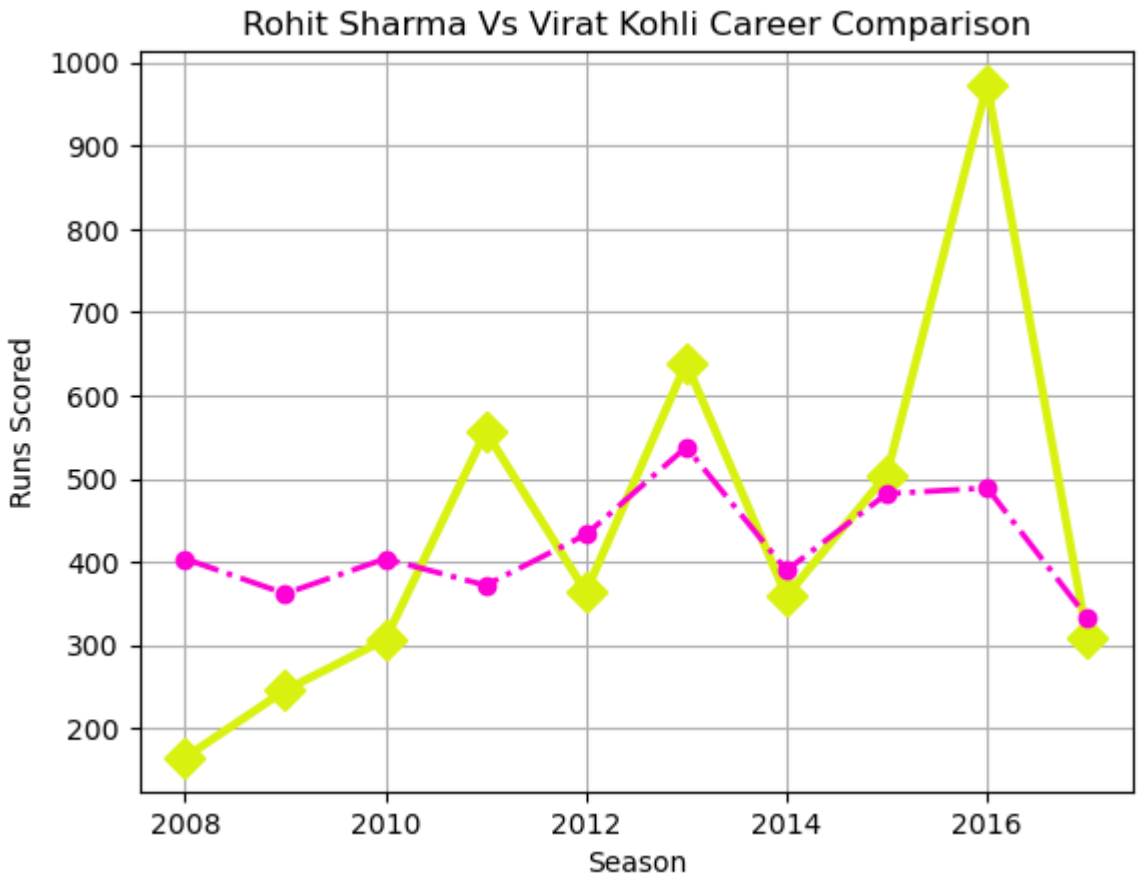
Markers are used to highlight data points on the line plot. You can customize markers' style and size. In the example, markers like 'D' and 'o' are used with different colors.

In [ ]:
```
# grid
plt.plot(batsman['index'],batsman['V Kohli'],color='#D9F10F',linestyle='solid',line
plt.plot(batsman['index'],batsman['RG Sharma'],color='#FC00D6',linestyle='dashdot',

plt.title('Rohit Sharma Vs Virat Kohli Career Comparison')
plt.xlabel('Season')
plt.ylabel('Runs Scored')

plt.grid()
```



Adding a grid to the plot can make it easier to read and interpret the data. The grid helps in aligning the data points with the tick marks on the axes.

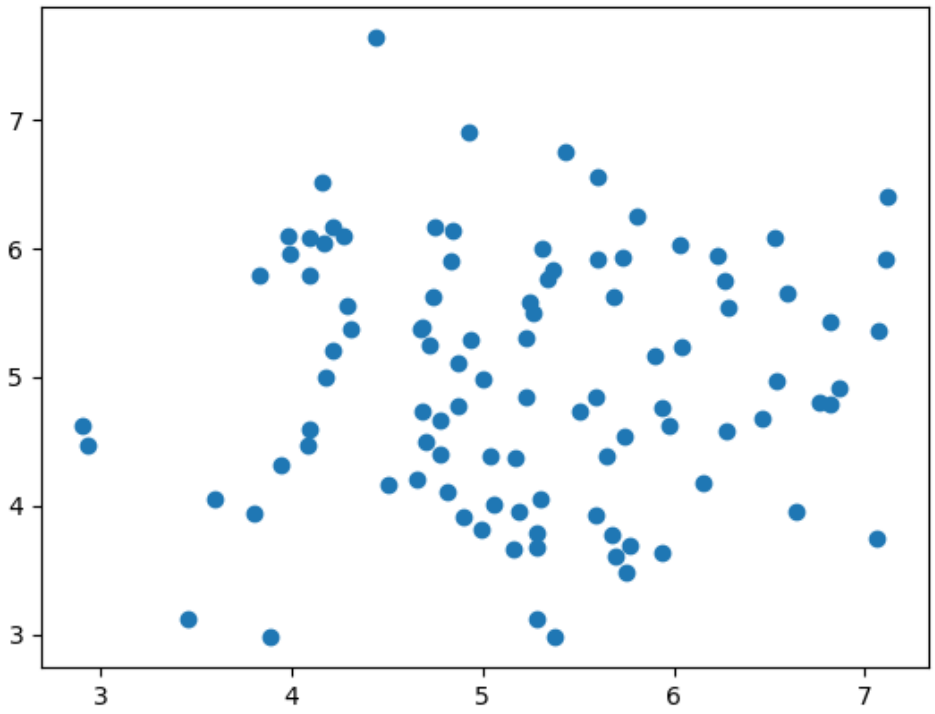
In [ ]:
```
# show
plt.plot(batsman['index'],batsman['V Kohli'],color='#D9F10F',linestyle='solid',line
plt.plot(batsman['index'],batsman['RG Sharma'],color='#FC00D6',linestyle='dashdot',

plt.title('Rohit Sharma Vs Virat Kohli Career Comparison')
plt.xlabel('Season')
plt.ylabel('Runs Scored')

plt.grid()

plt.show()
```



After customizing your plot, you can use plt.show() to display it. This command is often used in Jupyter notebooks or standalone Python scripts.

2D line plots are valuable for visualizing time series data, comparing trends in multiple datasets, and exploring the relationship between two continuous variables. Customization options in Matplotlib allow you to create visually appealing and informative plots for data analysis and presentation.

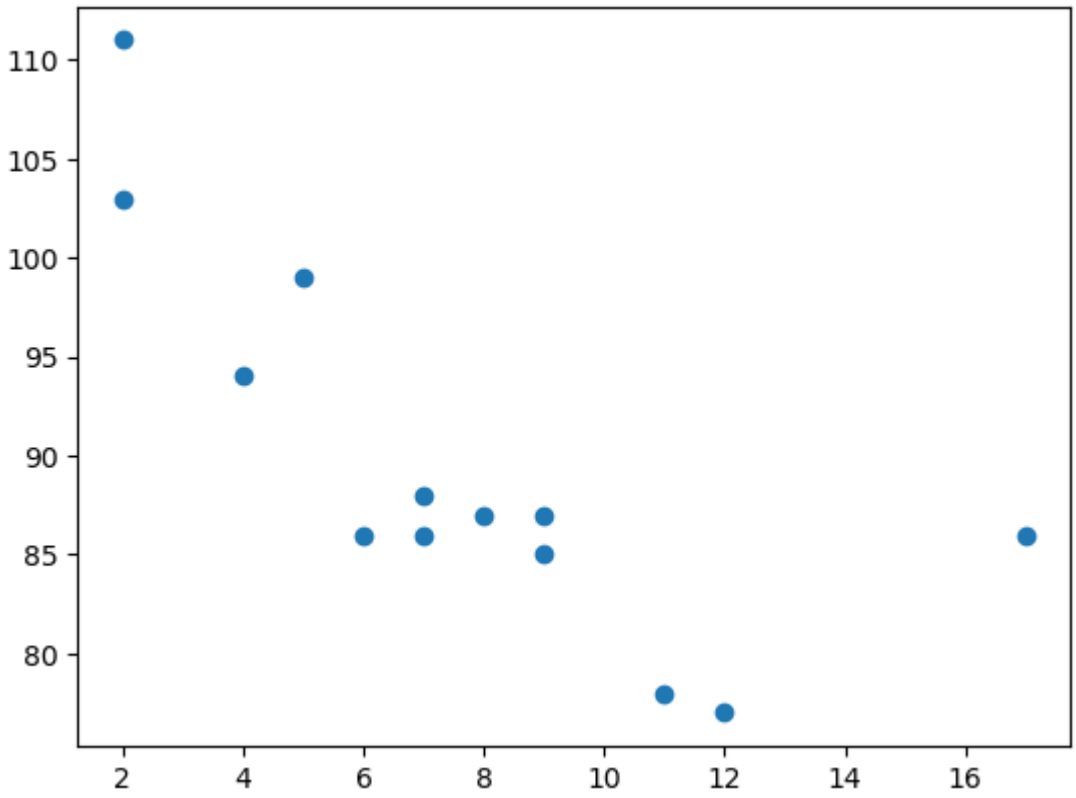
ScatterPlot and Bar Chart in Matplotlib



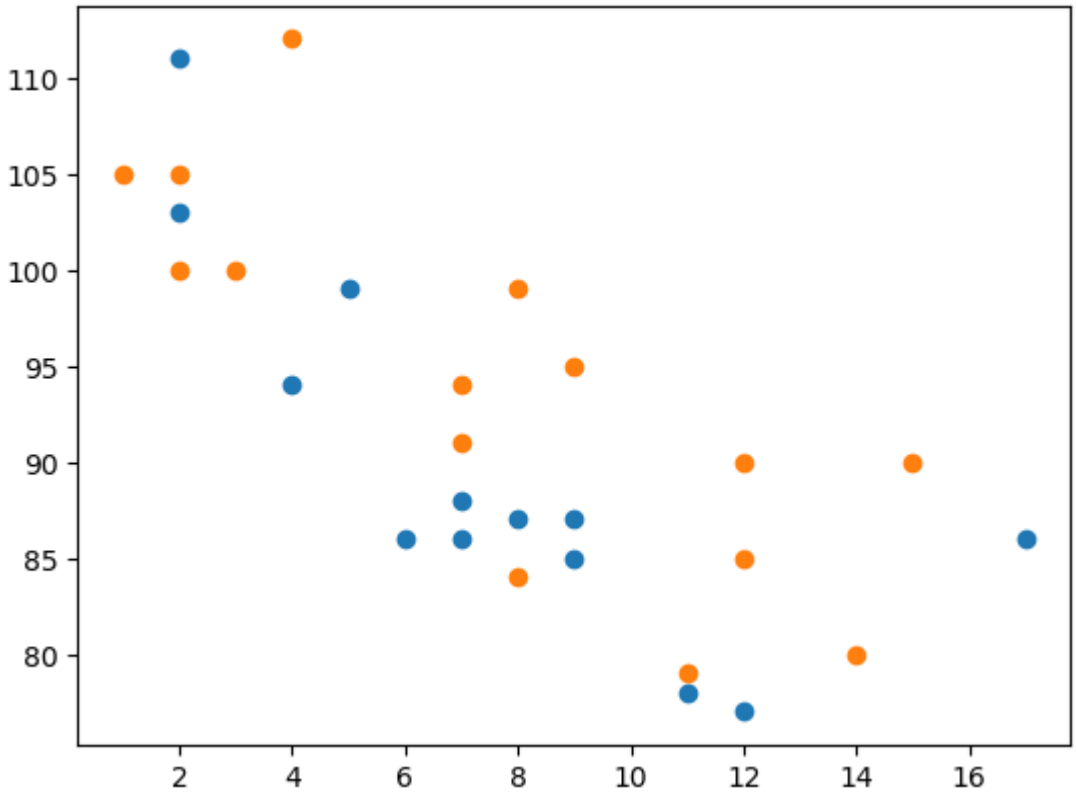
A scatter plot, also known as a scatterplot or scatter chart, is a type of data visualization used in statistics and data analysis. It's used to display the relationship between two variables by representing individual data points as points on a two-dimensional graph. Each point on the plot corresponds to a single data entry with values for both variables, making it a useful tool for identifying patterns, trends, clusters, or outliers in data.

- Bivariate Analysis
- numerical vs numerical
- Use case - Finding correlation

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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])

plt.scatter(x, y)
plt.show()
```



In [ ]:
```
#day one, the age and speed of 13 cars:
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)

#day two, the age and speed of 15 cars:
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)

plt.show()
```

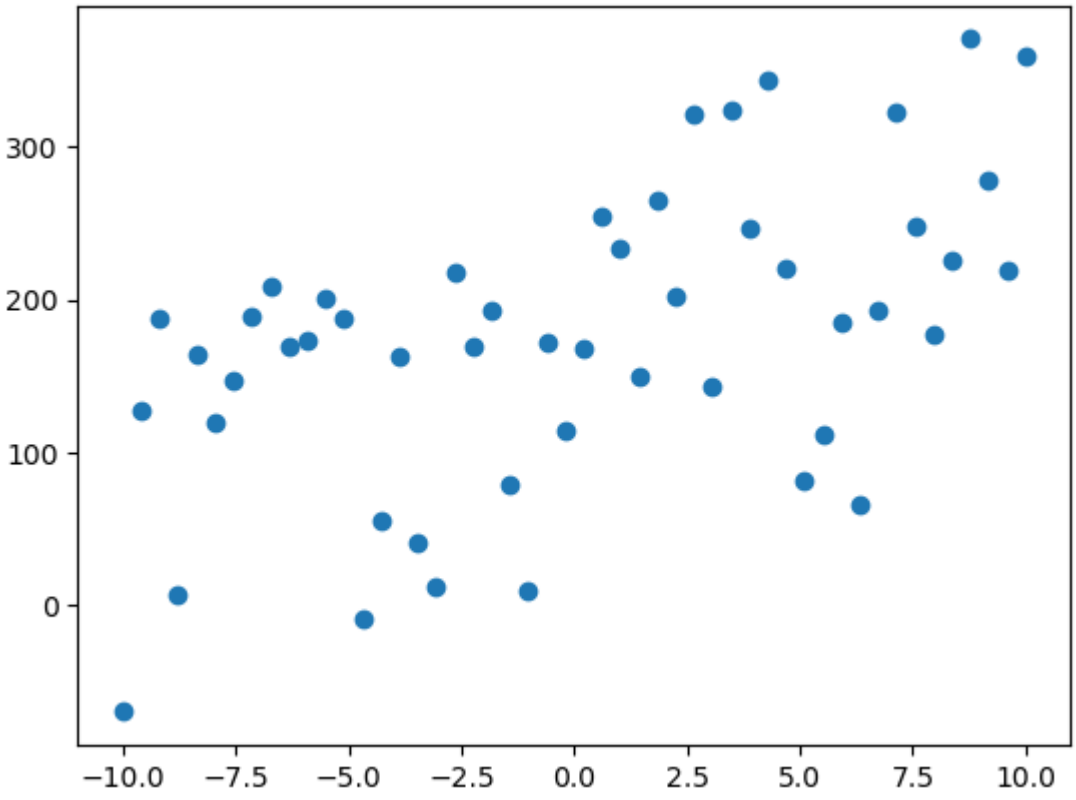


In [ ]:
```
# plt.scatter simple function
x = np.linspace(-10,10,50)

y = 10*x + 3 + np.random.randint(0,300,50)
y
```

Out[ ]:
```
array([-70.        , 127.08163265, 187.16326531,   6.24489796,
       164.32653061, 119.40816327, 146.48979592, 189.57142857,
       208.65306122, 169.73469388, 173.81632653, 200.89795918,
       187.97959184,  -8.93877551,  55.14285714, 162.2244898 ,
        40.30612245,  12.3877551 , 218.46938776, 169.55102041,
       193.63265306,  78.71428571,   9.79591837, 171.87755102,
       113.95918367, 168.04081633, 255.12244898, 233.20408163,
       150.28571429, 265.36734694, 202.44897959, 321.53061224,
       142.6122449 , 324.69387755, 246.7755102 , 343.85714286,
       220.93877551,  81.02040816, 111.10204082, 185.18367347,
        66.26530612, 193.34693878, 323.42857143, 248.51020408,
       177.59183673, 225.67346939, 370.75510204, 277.83673469,
       218.91836735, 360.        ])
```

In [ ]:
```
plt.scatter(x,y)
```

Out[ ]:
```
<matplotlib.collections.PathCollection at 0x264627ccc70>
```



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

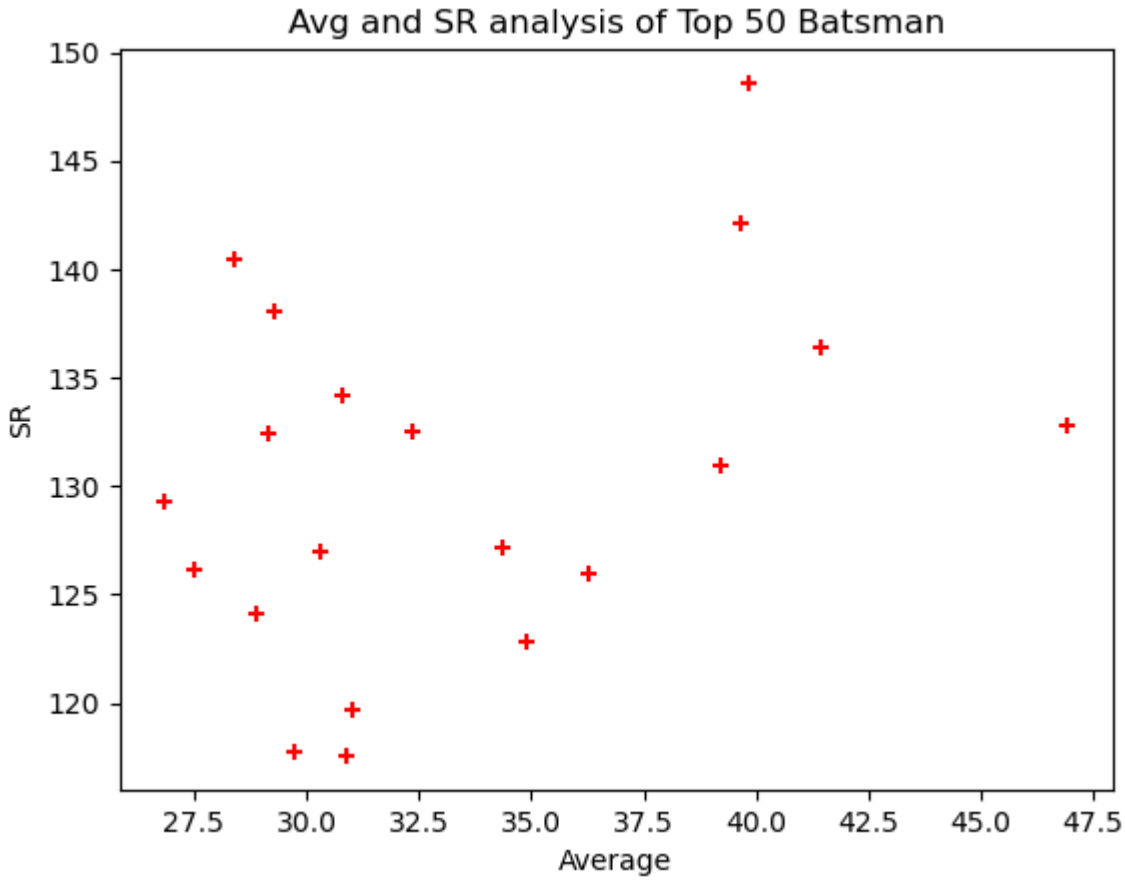
In [ ]:
```
# plt.scatter on pandas data
df = pd.read_csv('Data\Day47\Batter.csv')
df = df.head(20)
df
```

Out[ ]:

	batter	runs	avg	strike_rate
0	V Kohli	6634	36.251366	125.977972
1	S Dhawan	6244	34.882682	122.840842
2	DA Warner	5883	41.429577	136.401577
3	RG Sharma	5881	30.314433	126.964594
4	SK Raina	5536	32.374269	132.535312
5	AB de Villiers	5181	39.853846	148.580442
6	CH Gayle	4997	39.658730	142.121729
7	MS Dhoni	4978	39.196850	130.931089
8	RV Uthappa	4954	27.522222	126.152279
9	KD Karthik	4377	26.852761	129.267572
10	G Gambhir	4217	31.007353	119.665153
11	AT Rayudu	4190	28.896552	124.148148
12	AM Rahane	4074	30.863636	117.575758
13	KL Rahul	3895	46.927711	132.799182
14	SR Watson	3880	30.793651	134.163209
15	MK Pandey	3657	29.731707	117.739858
16	SV Samson	3526	29.140496	132.407060
17	KA Pollard	3437	28.404959	140.457703
18	F du Plessis	3403	34.373737	127.167414
19	YK Pathan	3222	29.290909	138.046272

In [ ]:
```
# marker
plt.scatter(df['avg'],df['strike_rate'],color='red',marker='+')
plt.title('Avg and SR analysis of Top 50 Batsman')
plt.xlabel('Average')
plt.ylabel('SR')
```

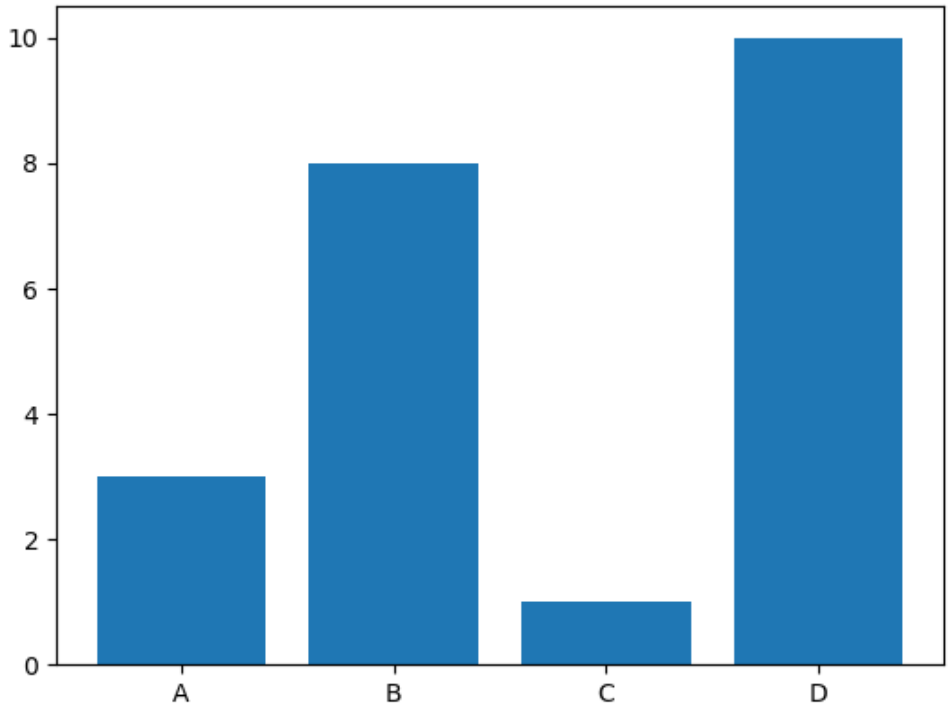
Out[ ]:
```
Text(0, 0.5, 'SR')
```



Scatter plots are particularly useful for visualizing the distribution of data, identifying correlations or relationships between variables, and spotting outliers. You can adjust the appearance and characteristics of the scatter plot to suit your needs, including marker size, color, and transparency. This makes scatter plots a versatile tool for data exploration and analysis.

Bar plot

Follow for more AI content: https://lnkd.in/gaJtbwcu

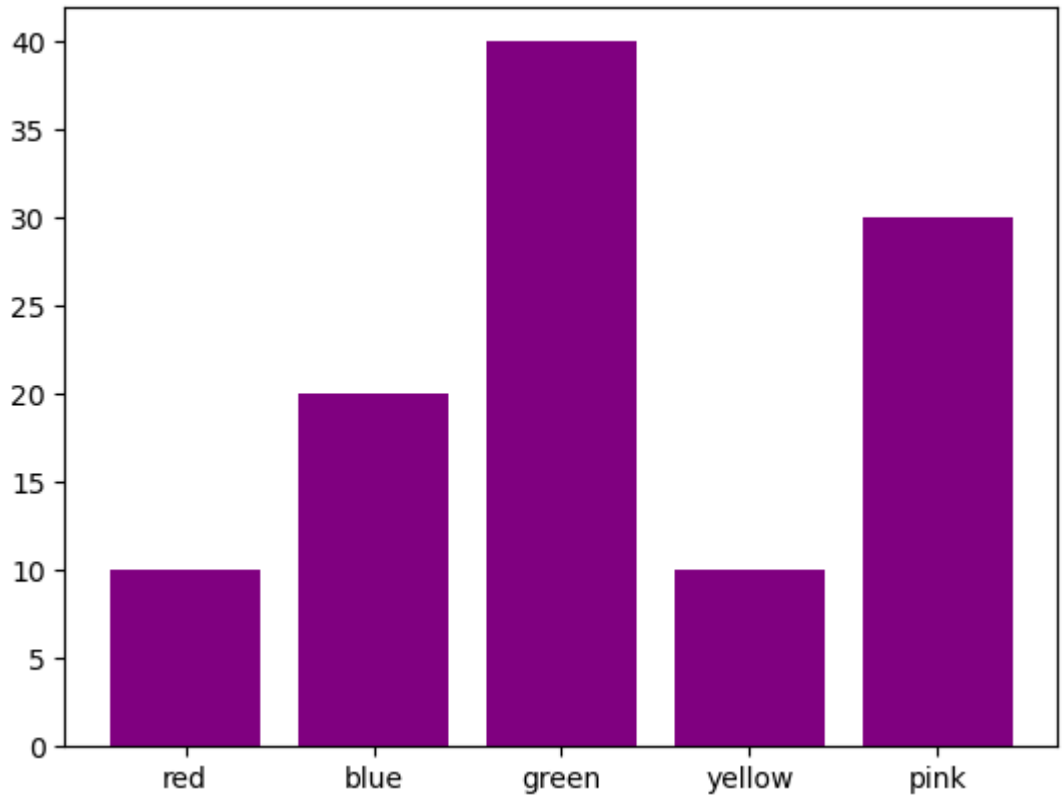


A bar plot, also known as a bar chart or bar graph, is a type of data visualization that is used to represent categorical data with rectangular bars. Each bar's length or height is proportional to the value it represents. Bar plots are typically used to compare and display the relative sizes or quantities of different categories or groups.

- Bivariate Analysis
- Numerical vs Categorical
- Use case - Aggregate analysis of groups

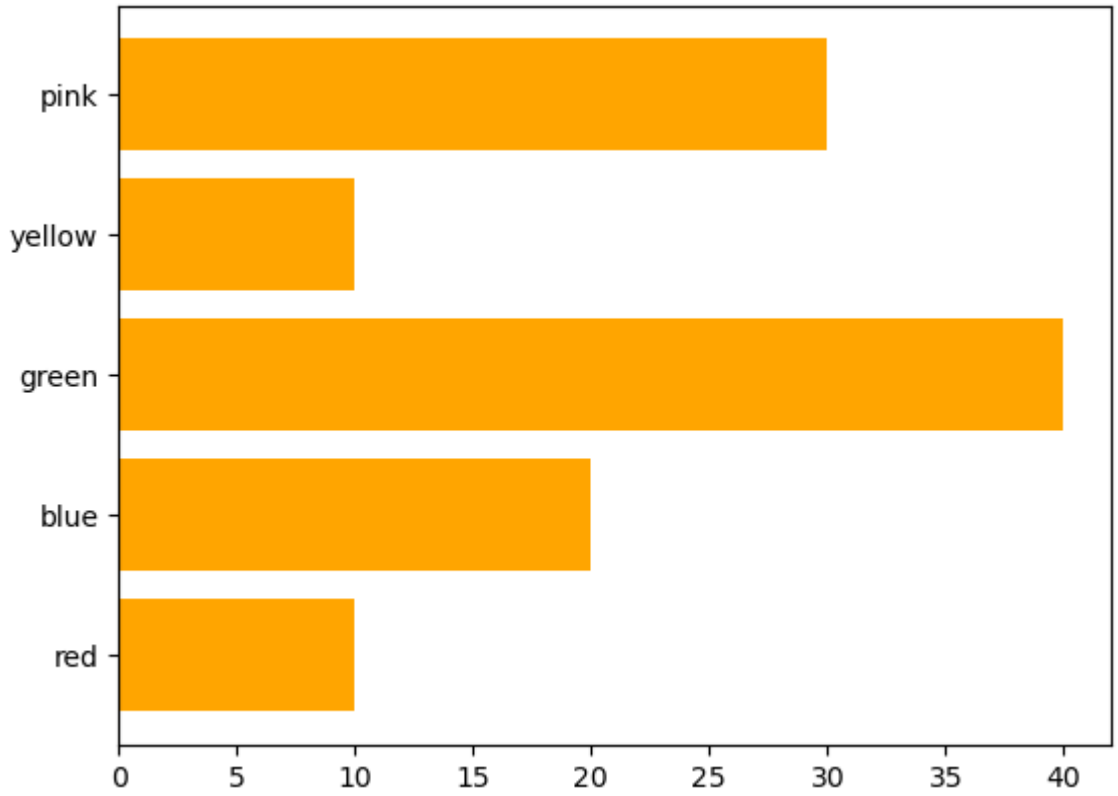
In [ ]:
```
# simple bar chart
children = [10,20,40,10,30]
colors = ['red','blue','green','yellow','pink']

plt.bar(colors,children,color='Purple')
```

Out[ ]:
```
<BarContainer object of 5 artists>
```



In [ ]:
```
# horizontal bar chart
plt.barh(colors,children,color='Orange')
```

Out[ ]:
```
<BarContainer object of 5 artists>
```

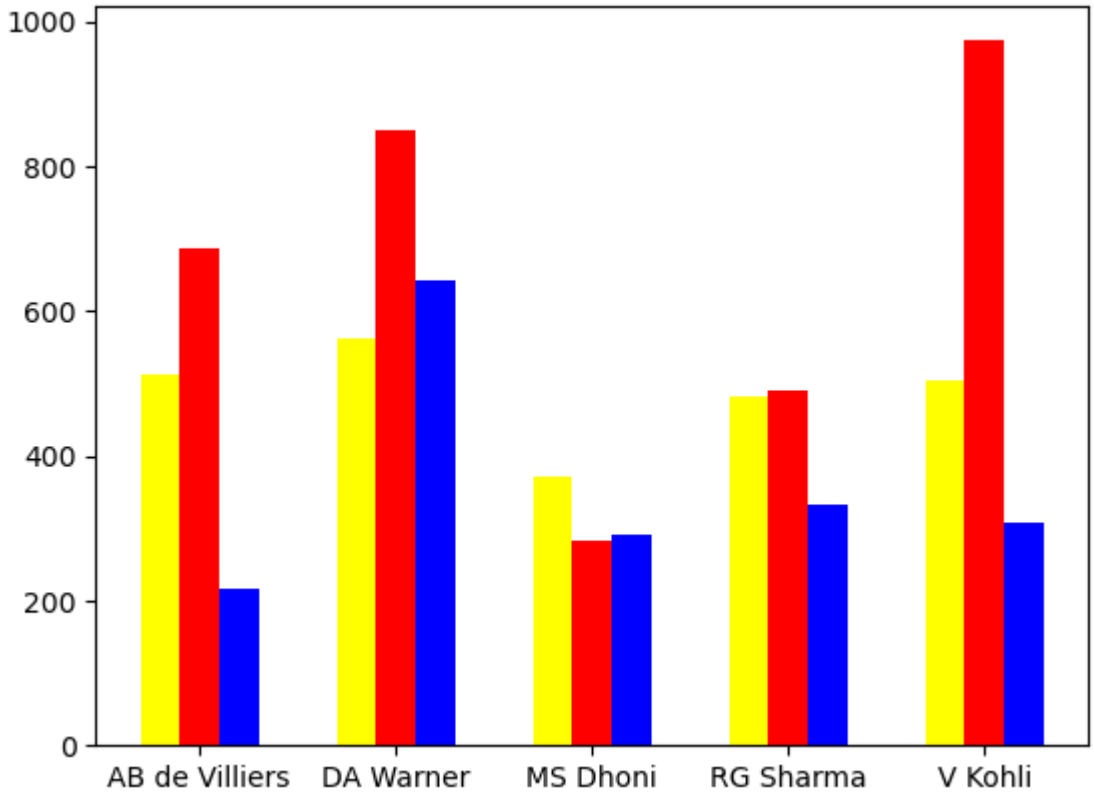


In [ ]:
```
# color and label
df = pd.read_csv('Data\Day47\Batsman_season.csv')
df
```

Out[ ]:

	batsman	2015	2016	2017
0	AB de Villiers	513	687	216
1	DA Warner	562	848	641
2	MS Dhoni	372	284	290
3	RG Sharma	482	489	333
4	V Kohli	505	973	308

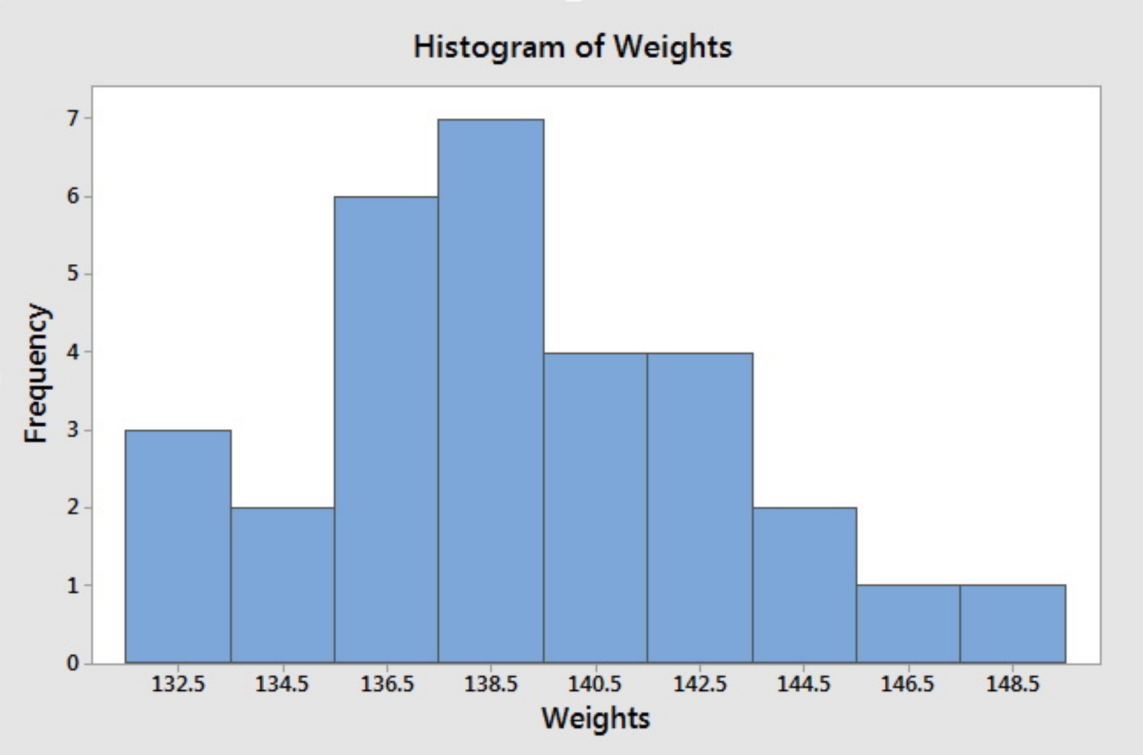
In [ ]:
```
plt.bar(np.arange(df.shape[0]) - 0.2,df['2015'],width=0.2,color='yellow')
plt.bar(np.arange(df.shape[0]),df['2016'],width=0.2,color='red')
plt.bar(np.arange(df.shape[0]) + 0.2,df['2017'],width=0.2,color='blue')

plt.xticks(np.arange(df.shape[0]), df['batsman'])

plt.show()
```



Bar plots are useful for comparing the values of different categories and for showing the distribution of data within each category. They are commonly used in various fields, including business, economics, and data analysis, to make comparisons and convey information about categorical data. You can customize bar plots to make them more visually appealing and informative.

Histogram and Pie chart in Matplotlib

Histogram



A histogram is a type of chart that shows the distribution of numerical data. It's a graphical representation of data where data is grouped into continuous number ranges and each range corresponds to a vertical bar. The horizontal axis displays the number range, and the vertical axis (frequency) represents the amount of data that is present in each range.

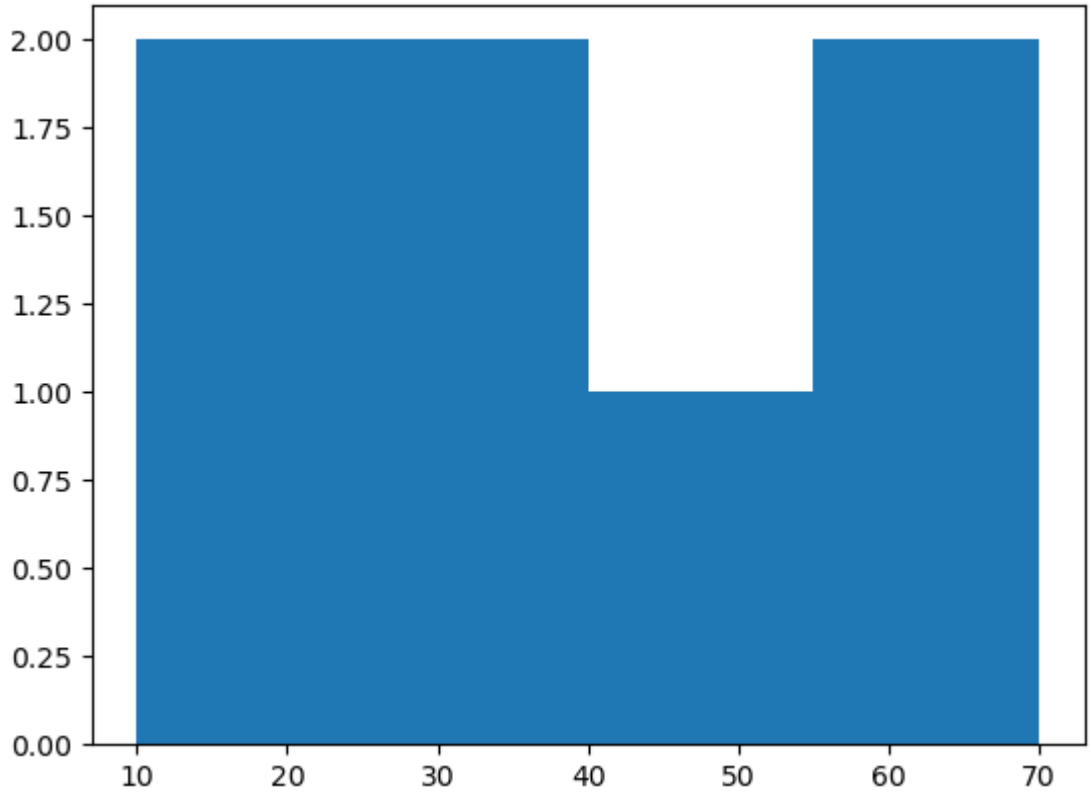
A histogram is a set of rectangles with bases along with the intervals between class boundaries and with areas proportional to frequencies in the corresponding classes. The x-axis of the graph represents the class interval, and the y-axis shows the various frequencies corresponding to different class intervals. A histogram is a type of data visualization used to represent the distribution of a dataset, especially when dealing with continuous or numeric data. It displays the frequency or count of data points falling into specific intervals or "bins" along a continuous range. Histograms provide insights into the shape, central tendency, and spread of a dataset.

- Univariate Analysis
- Numerical col
- Use case - Frequency Count

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

In [ ]:
```
# simple data

data = [32,45,56,10,15,27,61]

plt.hist(data,bins=[10,25,40,55,70])
```

Out[ ]:
```
(array([2., 2., 1., 2.]),
 array([10., 25., 40., 55., 70.]),
 <BarContainer object of 4 artists>)
```



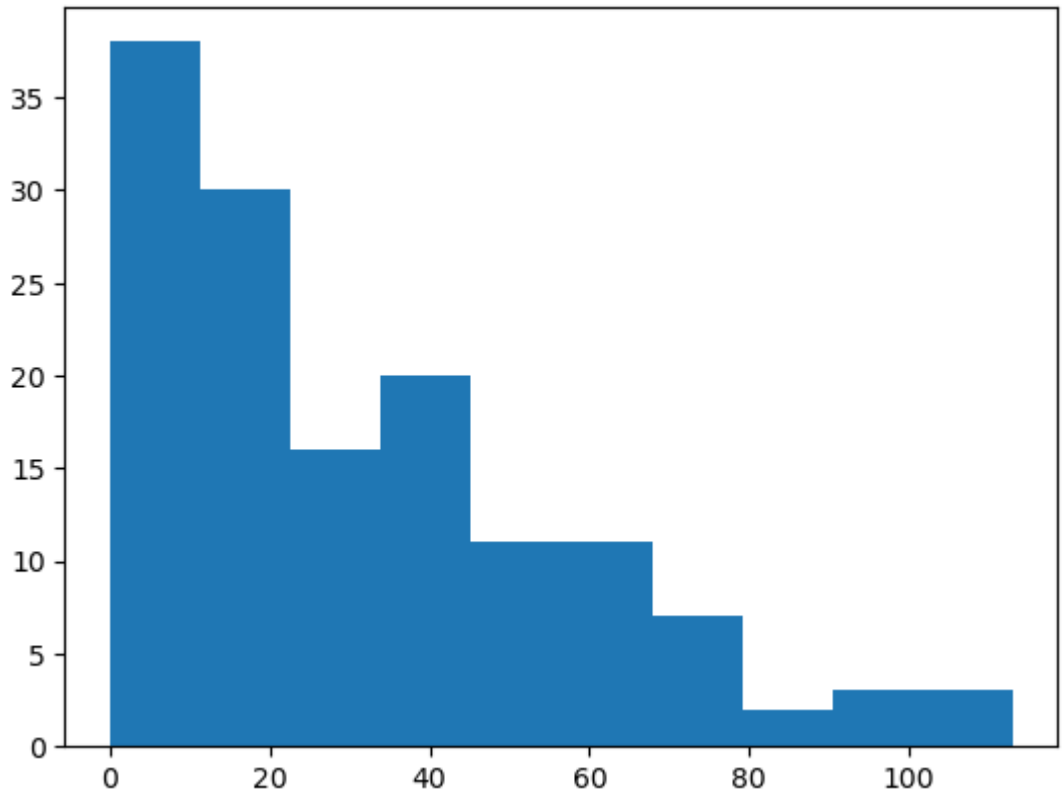
In [ ]:
```
# on some data
df = pd.read_csv('Data\Day48\Vk.csv')
df
```

Out[ ]:

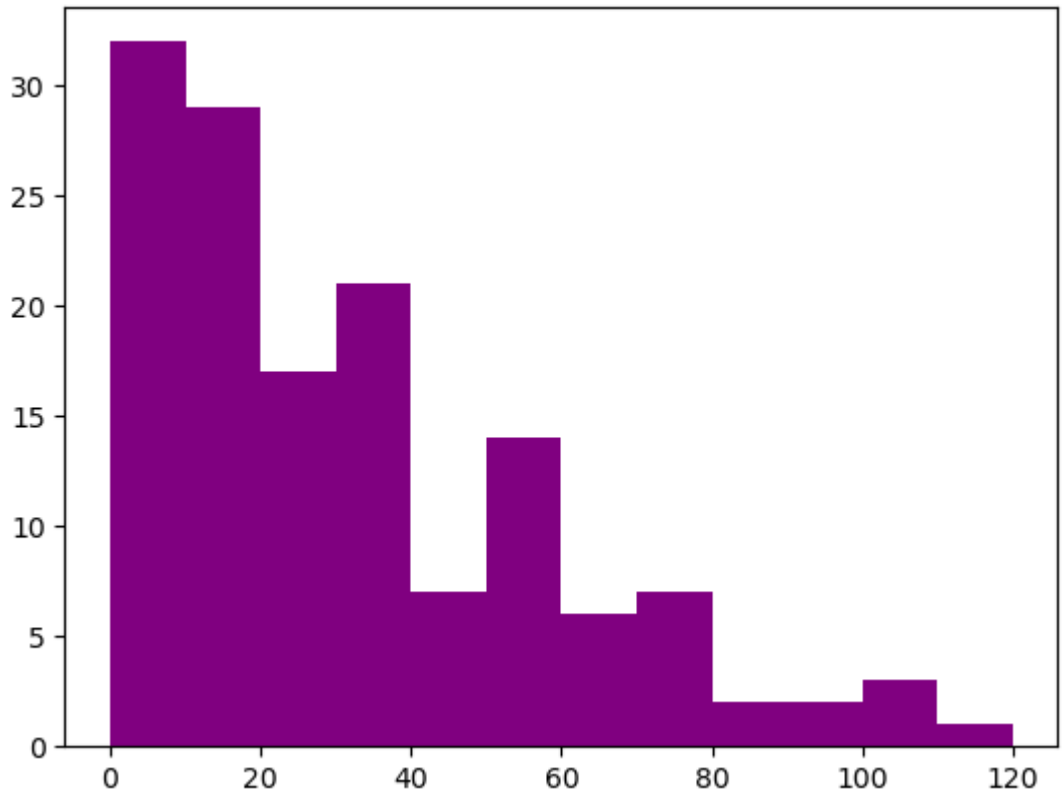
	match_id	batsman_runs
0	12	62
1	17	28
2	20	64
3	27	0
4	30	10
...	...	...
136	624	75
137	626	113
138	632	54
139	633	0
140	636	54

141 rows × 2 columns

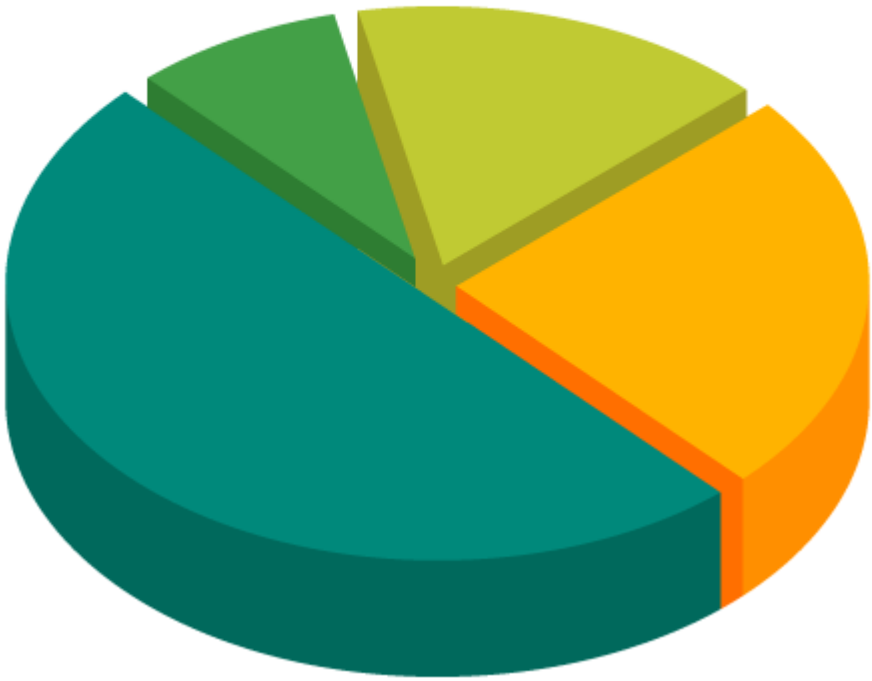
In [ ]:
```
plt.hist(df['batsman_runs'])
plt.show()
```



In [ ]:
```
# handling bins
plt.hist(df['batsman_runs'],bins=[0,10,20,30,40,50,60,70,80,90,100,110,120],color='
plt.show()
```



Pie Chart

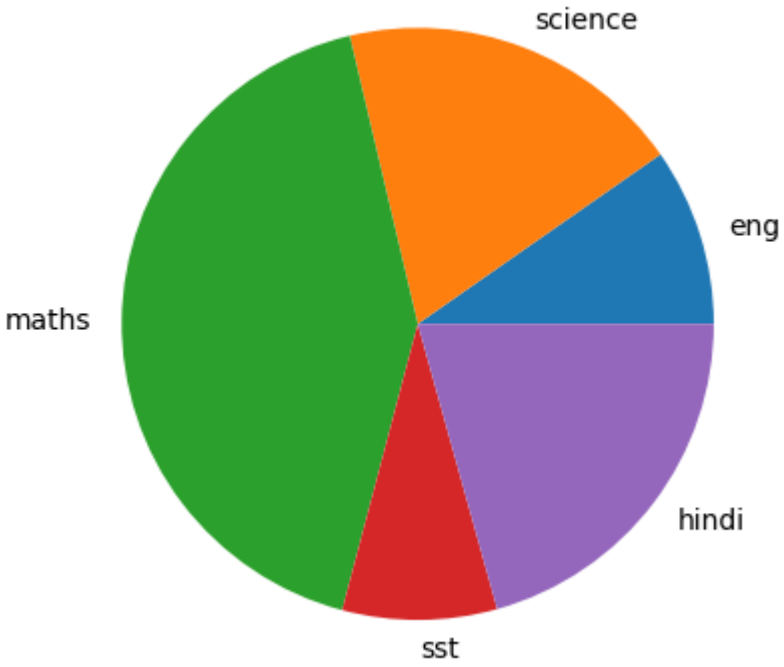


A pie chart is a circular graph that's divided into slices to illustrate numerical proportion. The slices of the pie show the relative size of the data. The arc length of each slice, and consequently its central angle and area, is proportional to the quantity it represents.

All slices of the pie add up to make the whole equaling 100 percent and 360 degrees. Pie charts are often used to represent sample data. Each of these categories is represented as a “slice of the pie”. The size of each slice is directly proportional to the number of data points that belong to a particular category.

- Univariate/Bivariate Analysis
- Categorical vs numerical
- Use case - To find contibution on a standard scale

In [ ]:
```
# simple data
data = [23,45,100,20,49]
subjects = ['eng','science','maths','sst','hindi']
plt.pie(data,labels=subjects)

plt.show()
```

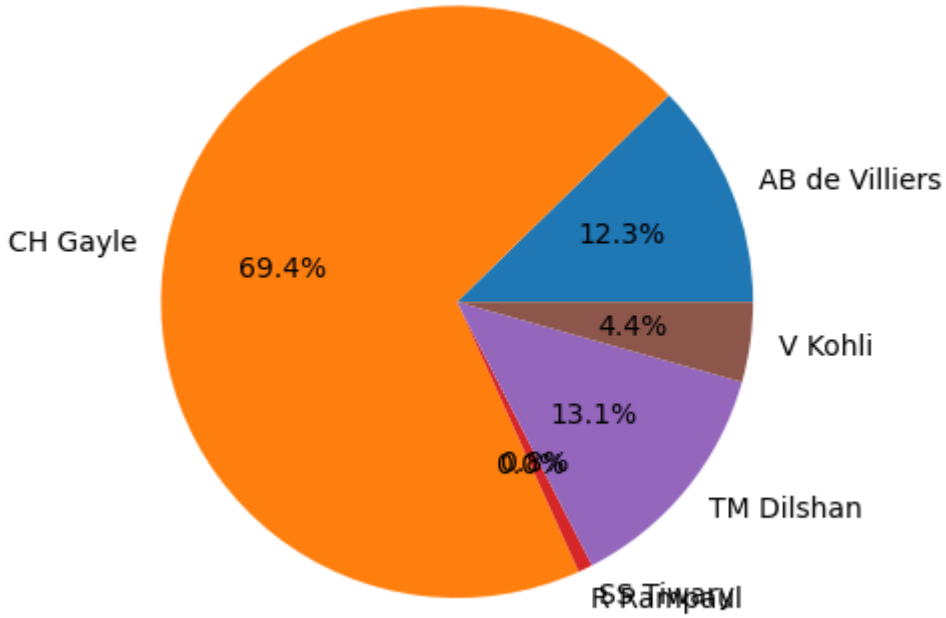


In [ ]:
```
# dataset
df = pd.read_csv('Data\Day48\Gayle-175.csv')
df
```

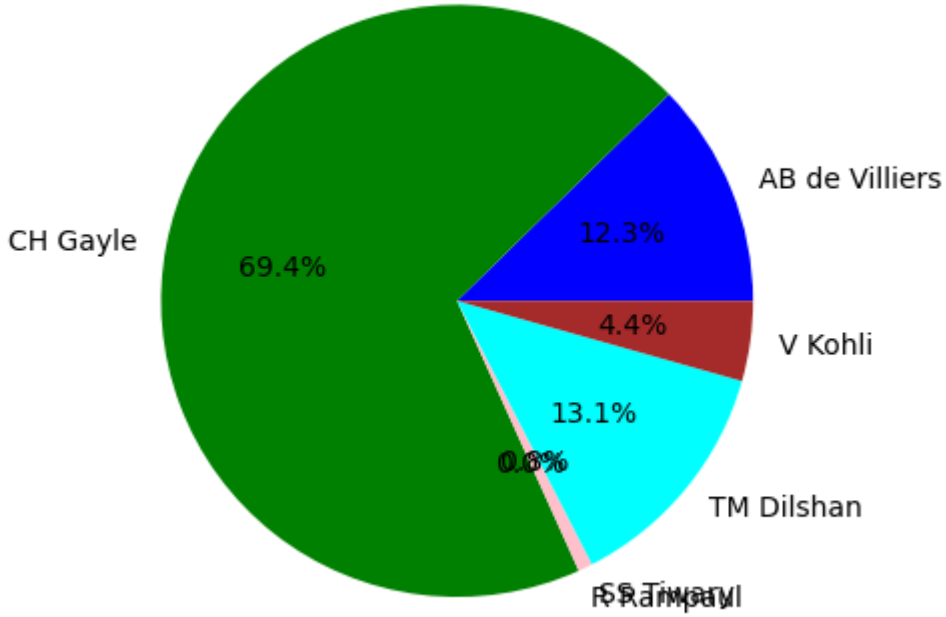
Out[ ]:

	batsman	batsman_runs
0	AB de Villiers	31
1	CH Gayle	175
2	R Rampaul	0
3	SS Tiwary	2
4	TM Dilshan	33
5	V Kohli	11

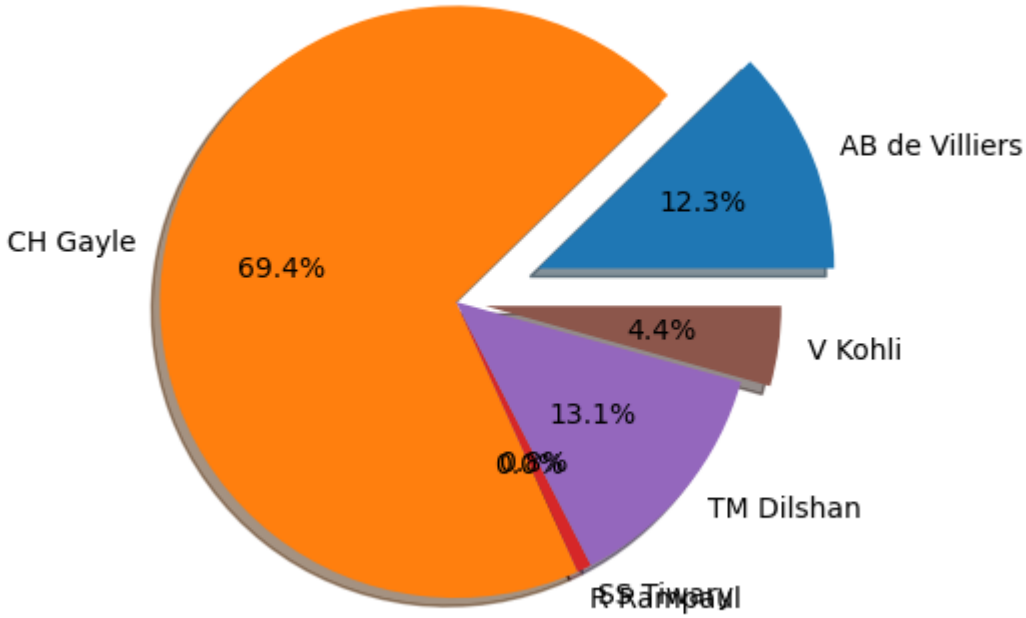
In [ ]:
```
plt.pie(df['batsman_runs'],labels=df['batsman'],autopct='%0.1f%%')
plt.show()
```



In [ ]:
```
# percentage and colors
plt.pie(df['batsman_runs'],labels=df['batsman'],autopct='%0.1f%%',colors=['blue','g
plt.show()
```



In [ ]:
```
# explode shadow
plt.pie(df['batsman_runs'],labels=df['batsman'],autopct='%0.1f%%',explode=[0.3,0,0,
plt.show()
```



Matplotlib Doc Website :

https://matplotlib.org/stable/

Advanced Matplotlib(part-1)

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

Colored Scatterplots

In [ ]:
```
iris = pd.read_csv('Data\Day49\iris.csv')
iris.sample(5)
```

Out[ ]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
146	147	6.3	2.5	5.0	1.9	Iris-virginica
123	124	6.3	2.7	4.9	1.8	Iris-virginica
78	79	6.0	2.9	4.5	1.5	Iris-versicolor
127	128	6.1	3.0	4.9	1.8	Iris-virginica
143	144	6.8	3.2	5.9	2.3	Iris-virginica

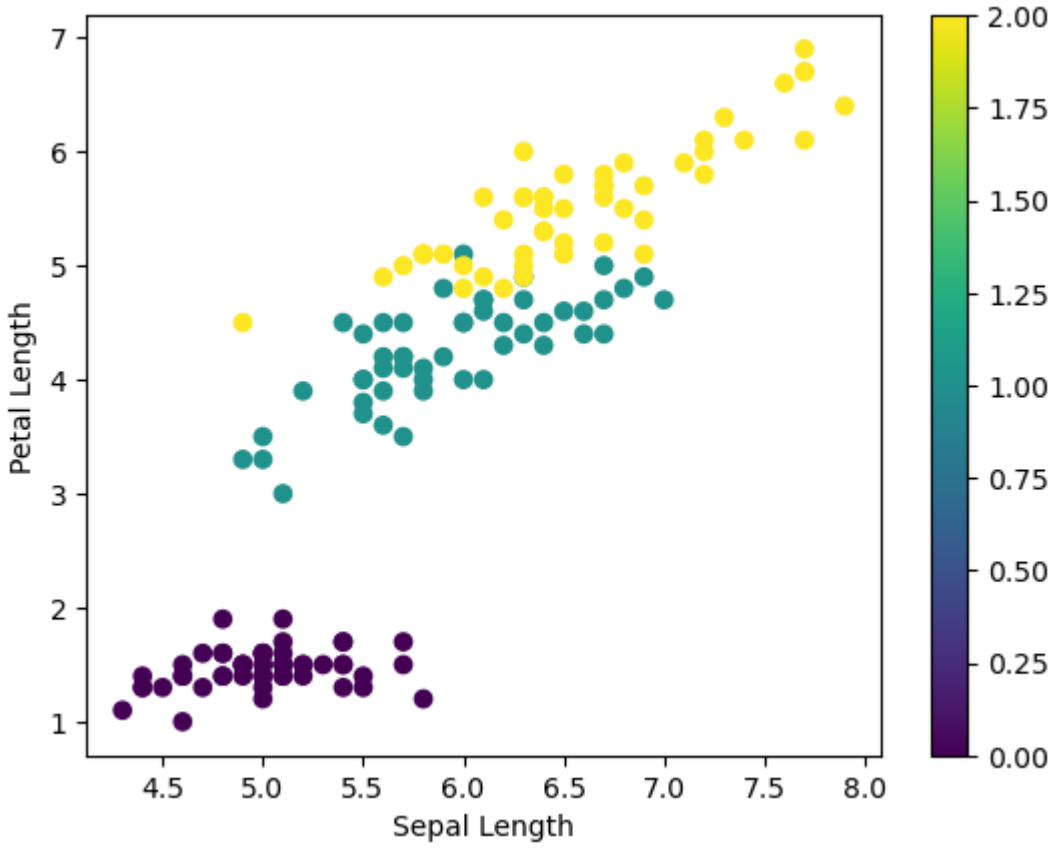
In [ ]:
```
iris['Species'] = iris['Species'].replace({'Iris-setosa':0,'Iris-versicolor':1,'Iri
iris.sample(5)
```

Out[ ]:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
90	91	5.5	2.6	4.4	1.2	1
25	26	5.0	3.0	1.6	0.2	0
89	90	5.5	2.5	4.0	1.3	1
95	96	5.7	3.0	4.2	1.2	1
148	149	6.2	3.4	5.4	2.3	2

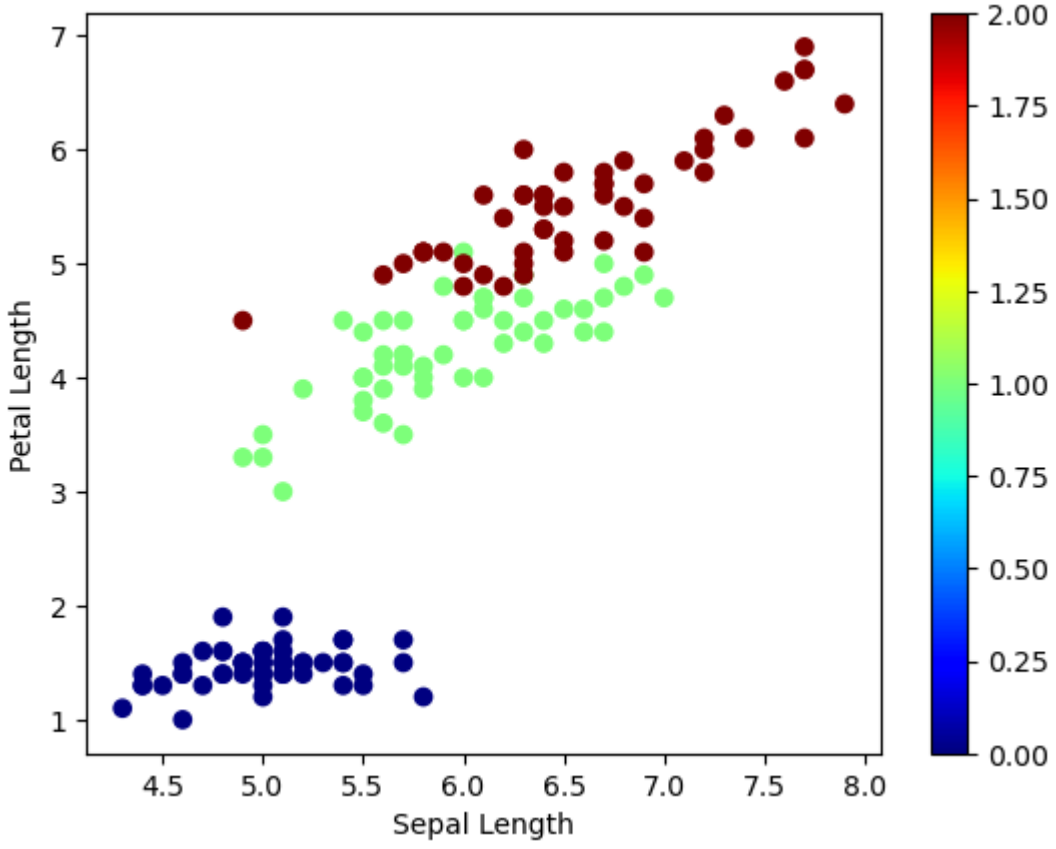
In [ ]:
```
plt.scatter(iris['SepalLengthCm'],iris['PetalLengthCm'],c=iris['Species'])
plt.xlabel('Sepal Length')
plt.ylabel('Petal Length')
plt.colorbar()
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x1ec76de6880>
```



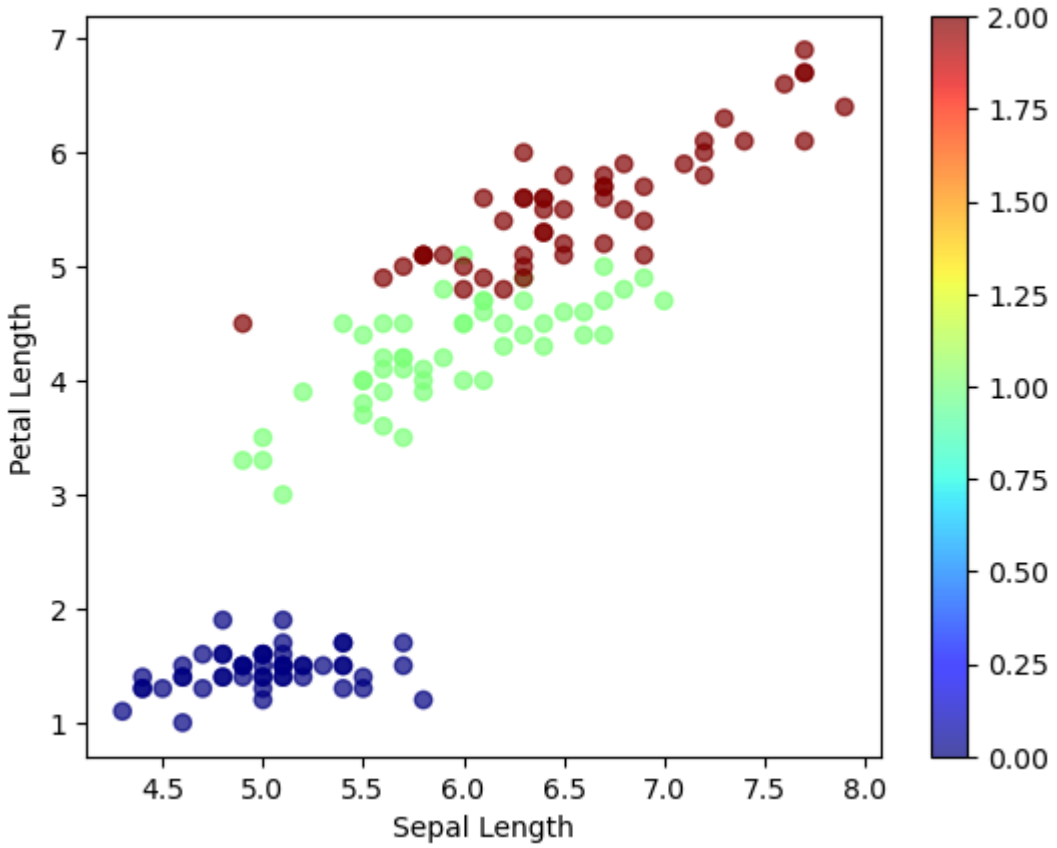
In [ ]:
```
# cmap
plt.scatter(iris['SepalLengthCm'],iris['PetalLengthCm'],c=iris['Species'],cmap='jet
plt.xlabel('Sepal Length')
plt.ylabel('Petal Length')
plt.colorbar()
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x1ec75d12f10>
```



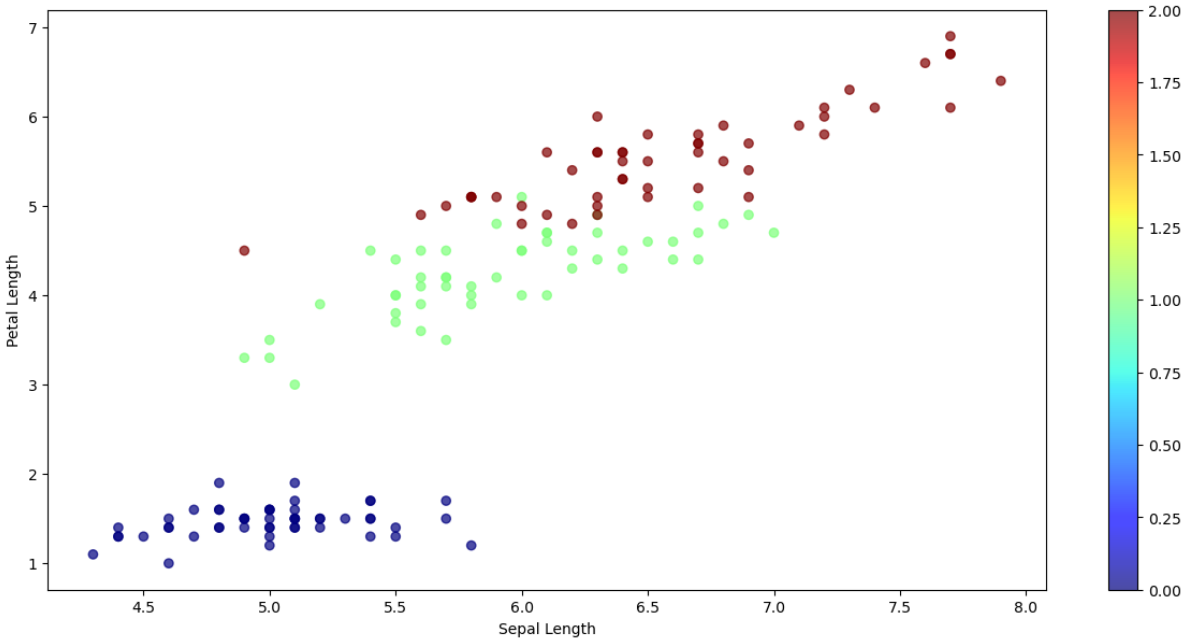
In [ ]:
```
# alpha
plt.scatter(iris['SepalLengthCm'],iris['PetalLengthCm'],c=iris['Species'],cmap='jet
plt.xlabel('Sepal Length')
plt.ylabel('Petal Length')
plt.colorbar()
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x1ec76e54790>
```



In [ ]:
```
# plot size
plt.figure(figsize=(15,7))

plt.scatter(iris['SepalLengthCm'],iris['PetalLengthCm'],c=iris['Species'],cmap='jet
plt.xlabel('Sepal Length')
plt.ylabel('Petal Length')
plt.colorbar()
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x1ec76f3bf40>
```



Annotations

In [ ]:
```
batters = pd.read_csv('Data\Day49\Batter.csv')
```

In [ ]:
```
sample_df = batters.head(100).sample(25,random_state=5)
```

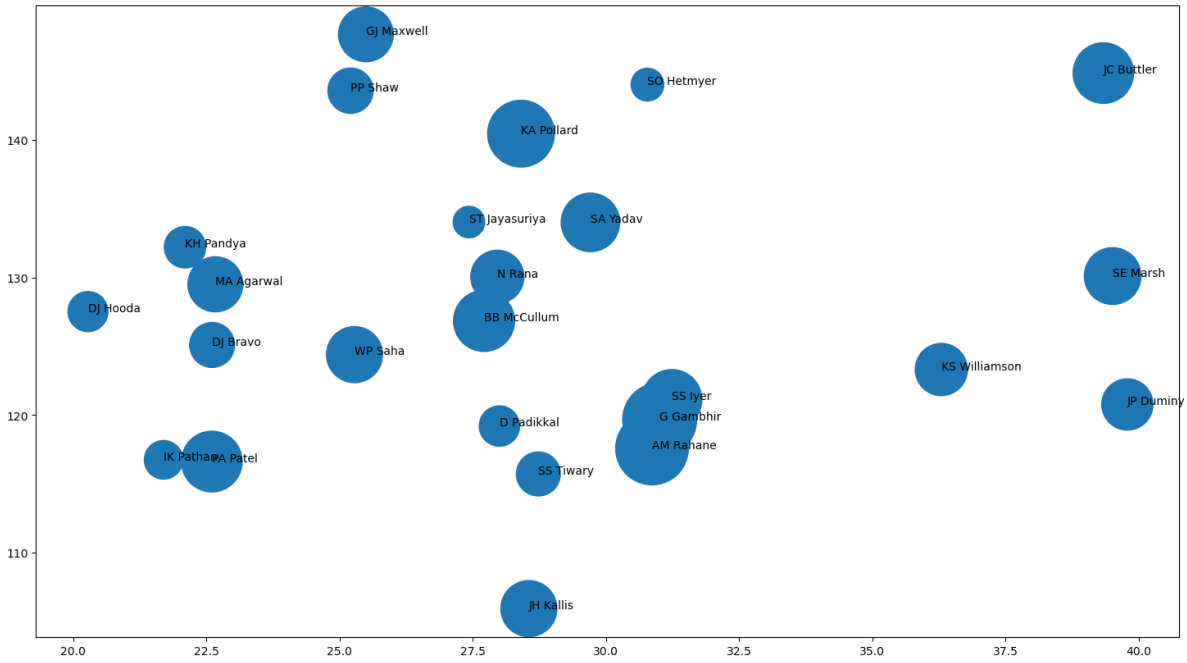
In [ ]:
```
sample_df
```

Follow for more AI content: https://lnkd.in/gaJtbwcu

Out[ ]:

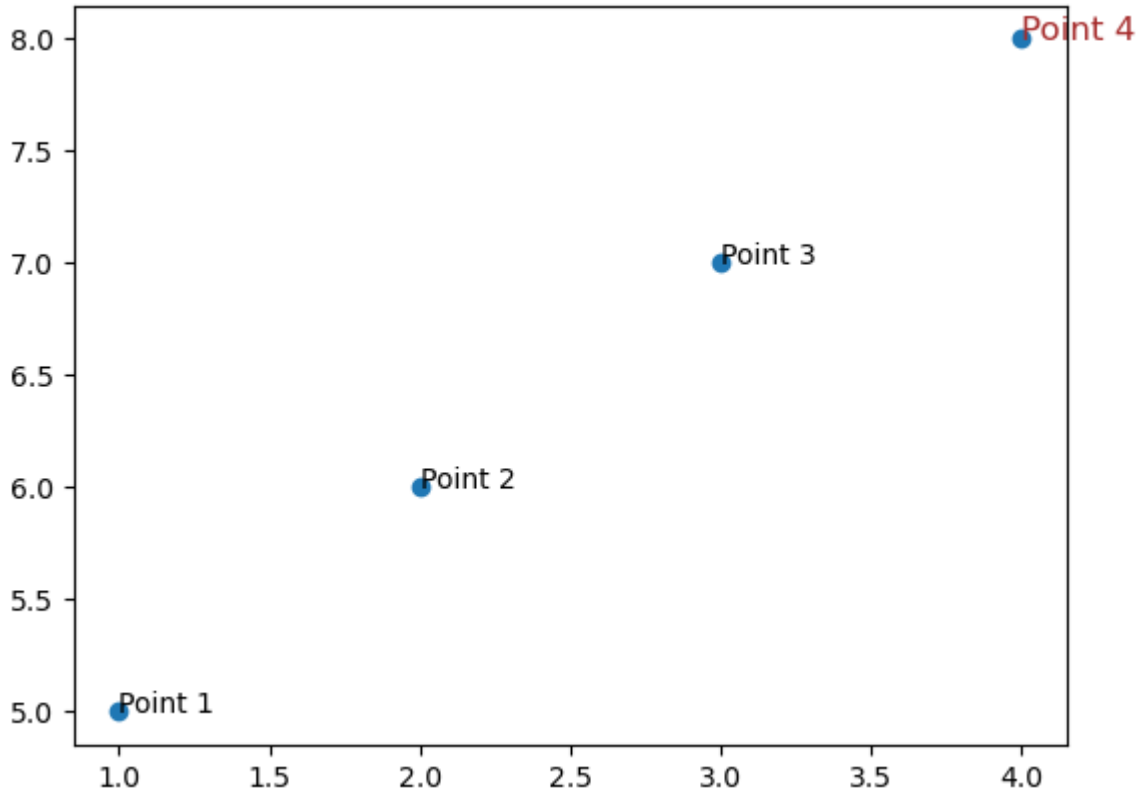
	batter	runs	avg	strike_rate
66	KH Pandya	1326	22.100000	132.203390
32	SE Marsh	2489	39.507937	130.109775
46	JP Duminy	2029	39.784314	120.773810
28	SA Yadav	2644	29.707865	134.009123
74	IK Pathan	1150	21.698113	116.751269
23	JC Buttler	2832	39.333333	144.859335
10	G Gambhir	4217	31.007353	119.665153
20	BB McCullum	2882	27.711538	126.848592
17	KA Pollard	3437	28.404959	140.457703
35	WP Saha	2427	25.281250	124.397745
97	ST Jayasuriya	768	27.428571	134.031414
37	MA Agarwal	2335	22.669903	129.506378
70	DJ Hooda	1237	20.278689	127.525773
40	N Rana	2181	27.961538	130.053667
60	SS Tiwary	1494	28.730769	115.724245
34	JH Kallis	2427	28.552941	105.936272
42	KS Williamson	2105	36.293103	123.315759
57	DJ Bravo	1560	22.608696	125.100241
12	AM Rahane	4074	30.863636	117.575758
69	D Padikkal	1260	28.000000	119.205298
94	SO Hetmyer	831	30.777778	144.020797
56	PP Shaw	1588	25.206349	143.580470
22	PA Patel	2848	22.603175	116.625717
39	GJ Maxwell	2320	25.494505	147.676639
24	SS Iyer	2780	31.235955	121.132898

In [ ]:
```
plt.figure(figsize=(18,10))
plt.scatter(sample_df['avg'],sample_df['strike_rate'],s=sample_df['runs'])

for i in range(sample_df.shape[0]):
  plt.text(sample_df['avg'].values[i],sample_df['strike_rate'].values[i],sample_df[
```



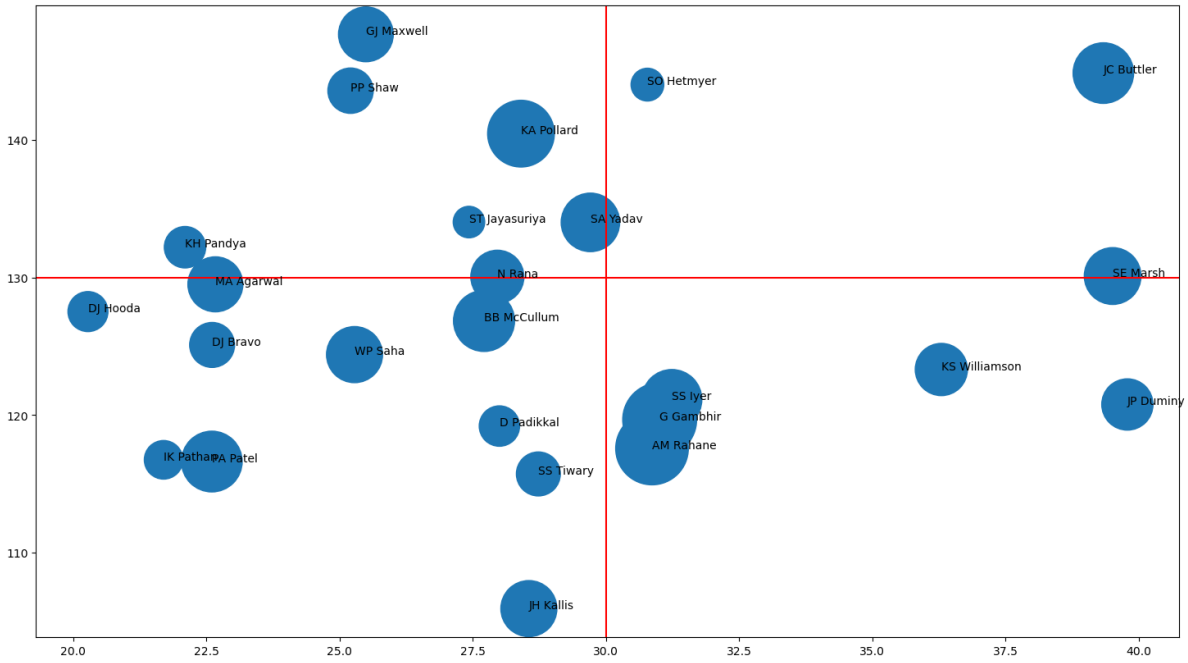
In [ ]:
```
x = [1,2,3,4]
y = [5,6,7,8]

plt.scatter(x,y)
plt.text(1,5,'Point 1')
plt.text(2,6,'Point 2')
plt.text(3,7,'Point 3')
plt.text(4,8,'Point 4',fontdict={'size':12,'color':'brown'})
```

Out[ ]:
```
Text(4, 8, 'Point 4')
```



In [ ]:
```
### Horizontal and Vertical lines

plt.figure(figsize=(18,10))
plt.scatter(sample_df['avg'],sample_df['strike_rate'],s=sample_df['runs'])
```

```
plt.axhline(130,color='red')

plt.axvline(30,color='red')

for i in range(sample_df.shape[0]):
  plt.text(sample_df['avg'].values[i],sample_df['strike_rate'].values[i],sample_df[
```



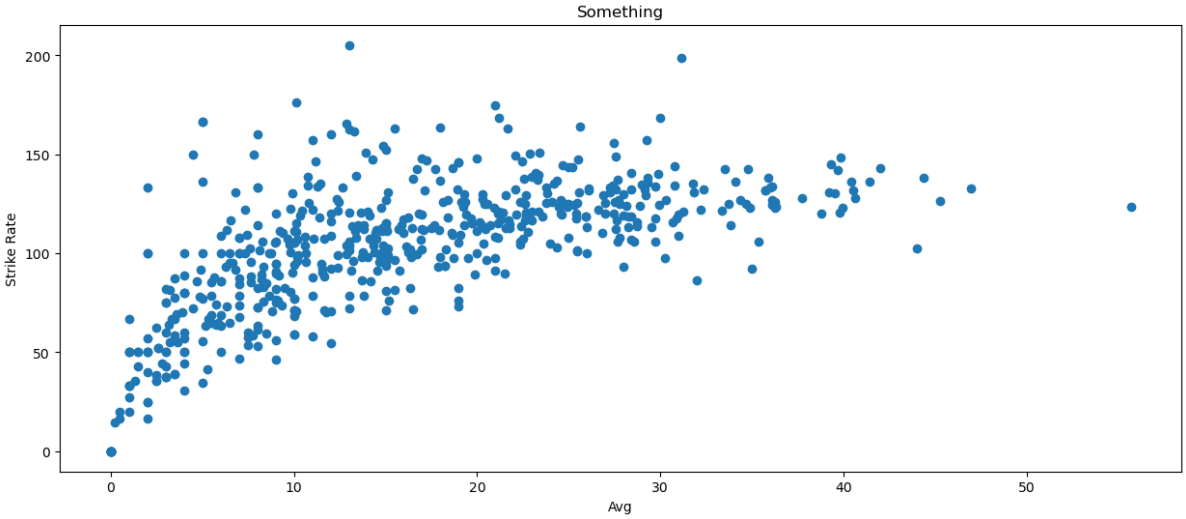
Subplots

In [ ]:
```
# A diff way to plot graphs
batters.head()
```

Out[ ]:

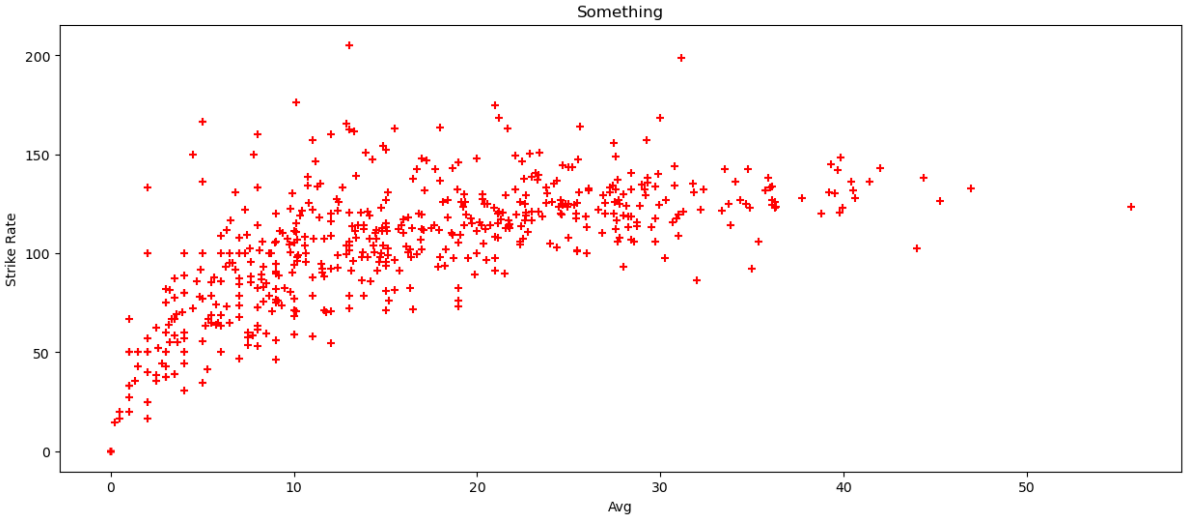
	batter	runs	avg	strike_rate
0	V Kohli	6634	36.251366	125.977972
1	S Dhawan	6244	34.882682	122.840842
2	DA Warner	5883	41.429577	136.401577
3	RG Sharma	5881	30.314433	126.964594
4	SK Raina	5536	32.374269	132.535312

In [ ]:
```
plt.figure(figsize=(15,6))
plt.scatter(batters['avg'],batters['strike_rate'])
plt.title('Something')
plt.xlabel('Avg')
plt.ylabel('Strike Rate')

plt.show()
```



In [ ]:
```
fig,ax = plt.subplots(figsize=(15,6))

ax.scatter(batters['avg'],batters['strike_rate'],color='red',marker='+')
ax.set_title('Something')
ax.set_xlabel('Avg')
ax.set_ylabel('Strike Rate')

fig.show()
```

```
C:\Users\disha\AppData\Local\Temp\ipykernel_4684\3179312453.py:8: UserWarning: Mat
plotlib is currently using module://matplotlib_inline.backend_inline, which is a n
on-GUI backend, so cannot show the figure.
  fig.show()
```

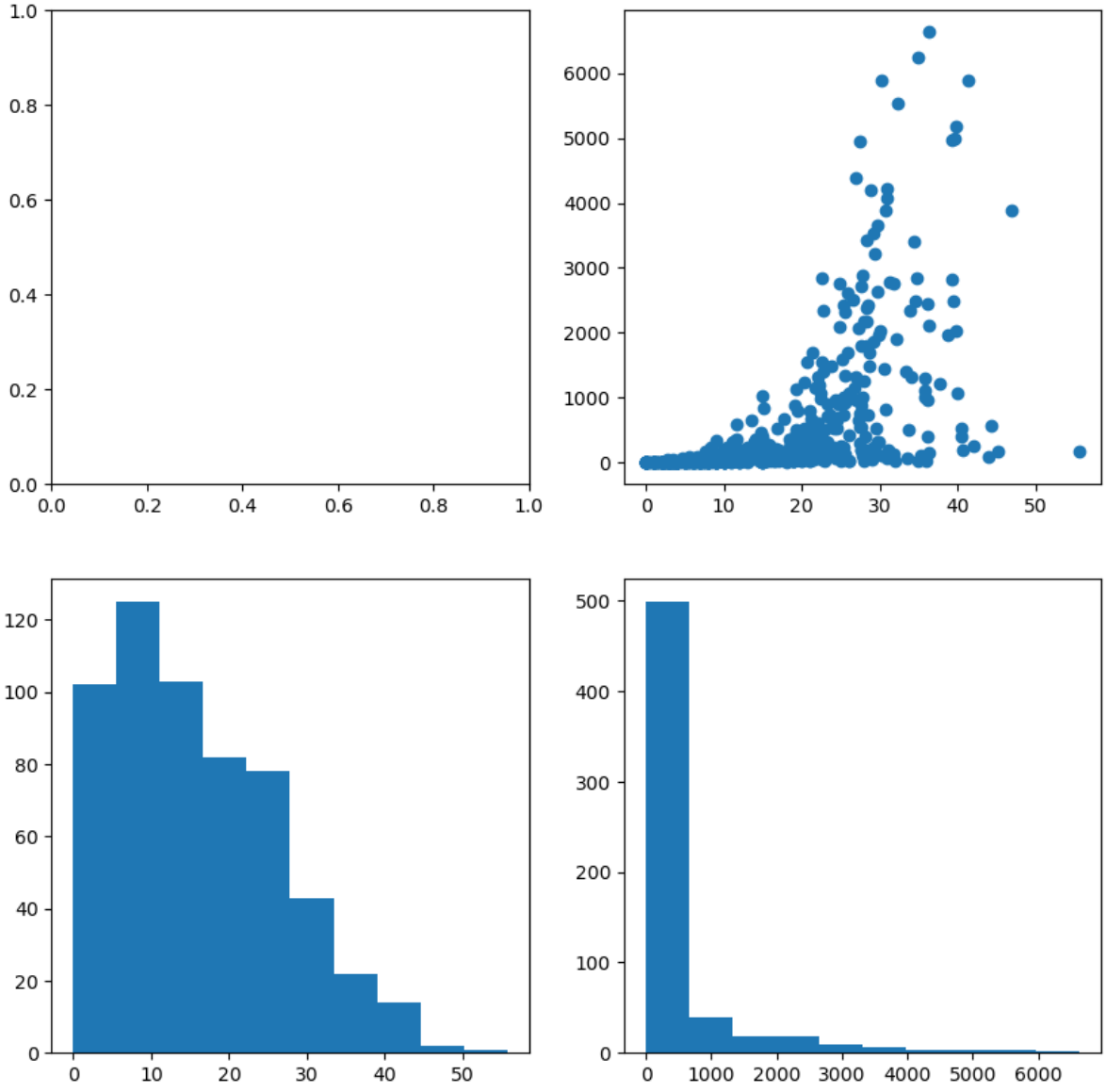


In [ ]:
```
ig, ax = plt.subplots(nrows=2,ncols=1,sharex=True,figsize=(10,6))

ax[0].scatter(batters['avg'],batters['strike_rate'],color='red')
ax[1].scatter(batters['avg'],batters['runs'])

ax[0].set_title('Avg Vs Strike Rate')
ax[0].set_ylabel('Strike Rate')


ax[1].set_title('Avg Vs Runs')
ax[1].set_ylabel('Runs')
ax[1].set_xlabel('Avg')
```

Out[ ]:
```
Text(0.5, 0, 'Avg')
```

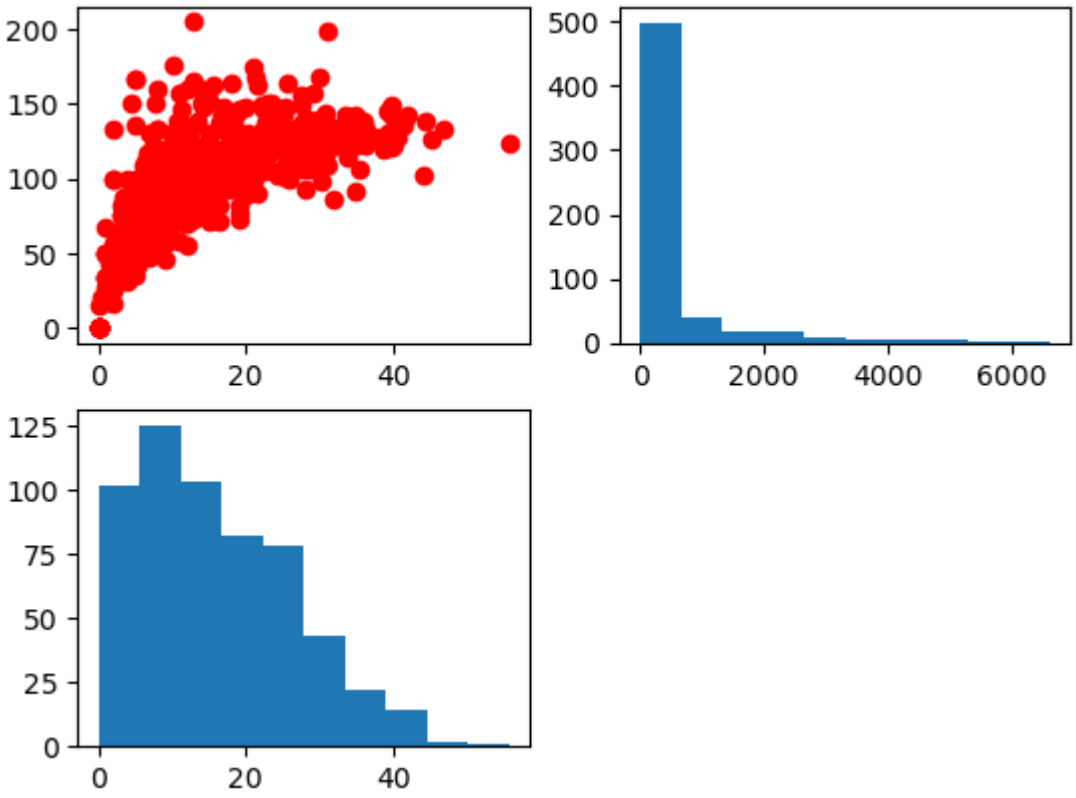
In [ ]:
```
fig, ax = plt.subplots(nrows=2,ncols=2,figsize=(10,10))

ax[0,0]
ax[0,1].scatter(batters['avg'],batters['runs'])
ax[1,0].hist(batters['avg'])
ax[1,1].hist(batters['runs'])
```

Out[ ]:
```
(array([499.,  40.,  19.,  19.,   9.,   6.,   4.,   4.,   3.,   2.]),
 array([   0. ,  663.4, 1326.8, 1990.2, 2653.6, 3317. , 3980.4, 4643.8,
        5307.2, 5970.6, 6634. ]),
 <BarContainer object of 10 artists>)
```



In [ ]:
```
fig = plt.figure()

ax1 = fig.add_subplot(2,2,1)
ax1.scatter(batters['avg'],batters['strike_rate'],color='red')

ax2 = fig.add_subplot(2,2,2)
ax2.hist(batters['runs'])

ax3 = fig.add_subplot(2,2,3)
ax3.hist(batters['avg'])
```

Out[ ]:
```
(array([102., 125., 103.,  82.,  78.,  43.,  22.,  14.,   2.,   1.]),
 array([ 0.        ,  5.56666667, 11.13333333, 16.7       , 22.26666667,
        27.83333333, 33.4       , 38.96666667, 44.53333333, 50.1       ,
        55.66666667]),
 <BarContainer object of 10 artists>)
```


200
150
100
50
0
0
20
40
500
400
300
200
100
0
0
2000
4000
6000
125
100
75
50
25
0
0
20
40

Advanced Matplotlib(part-2)

3D scatter Plot

- A 3D scatter plot is used to represent data points in a three-dimensional space.

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

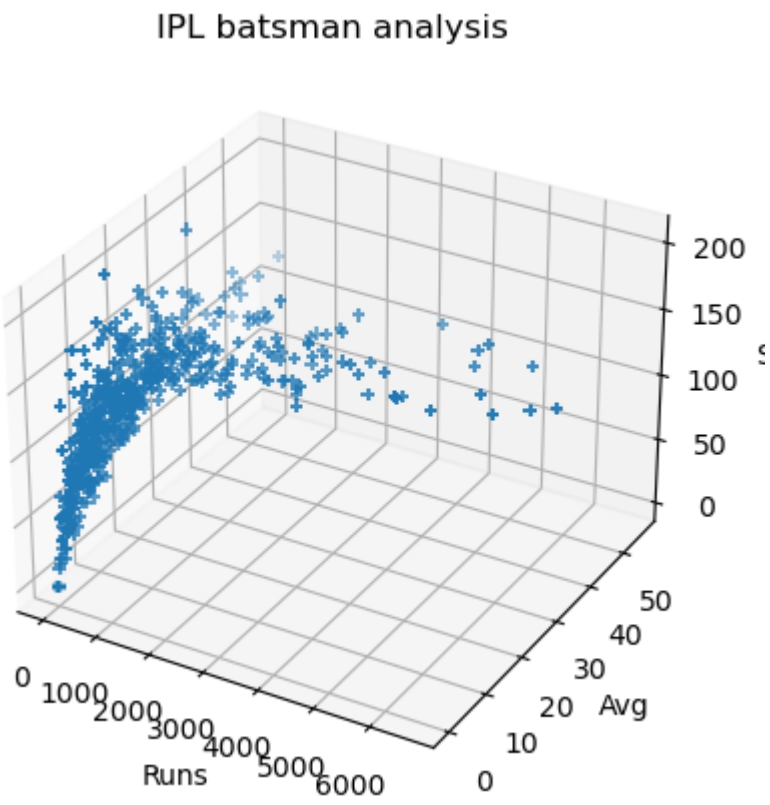
In [ ]:
```
batters = pd.read_csv('Data\Day49\Batter.csv')
batters.head()
```

Out[ ]:

	batter	runs	avg	strike_rate
0	V Kohli	6634	36.251366	125.977972
1	S Dhawan	6244	34.882682	122.840842
2	DA Warner	5883	41.429577	136.401577
3	RG Sharma	5881	30.314433	126.964594
4	SK Raina	5536	32.374269	132.535312

In [ ]:
```
fig = plt.figure()

ax = plt.subplot(projection='3d')

ax.scatter3D(batters['runs'],batters['avg'],batters['strike_rate'],marker='+')
ax.set_title('IPL batsman analysis')

ax.set_xlabel('Runs')
ax.set_ylabel('Avg')
ax.set_zlabel('SR')
```

Out[ ]:
```
Text(0.5, 0, 'SR')
```



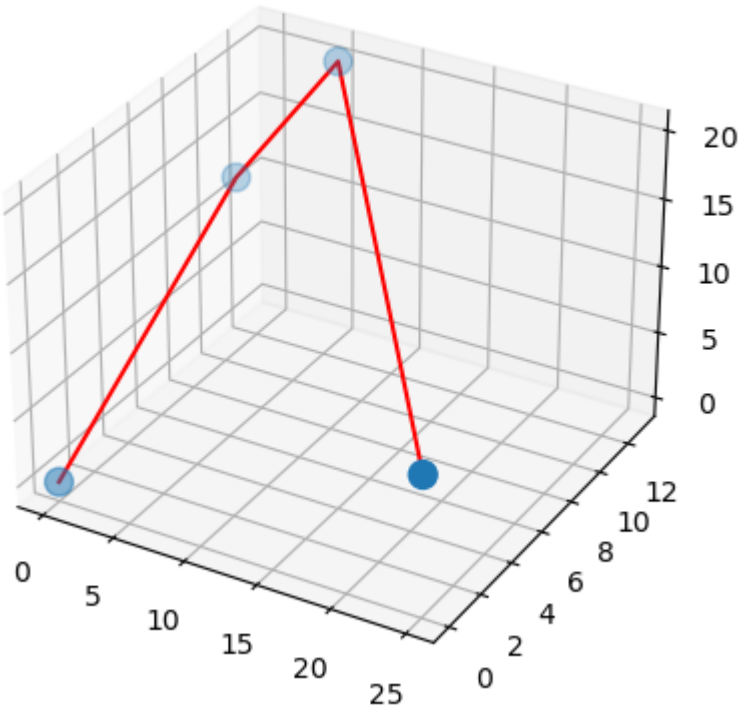
- In the example, you created a 3D scatter plot to analyze IPL batsmen based on runs, average (avg), and strike rate (SR).
- The ax.scatter3D function was used to create the plot, where the three variables were mapped to the x, y, and z axes.

3D Line Plot

- A 3D line plot represents data as a line in three-dimensional space.

In [ ]:
```
x = [0,1,5,25]
y = [0,10,13,0]
z = [0,13,20,9]

fig = plt.figure()

ax = plt.subplot(projection='3d')

ax.scatter3D(x,y,z,s=[100,100,100,100])
ax.plot3D(x,y,z,color='red')
```

Out[ ]:
```
[<mpl_toolkits.mplot3d.art3d.Line3D at 0x23ec4988340>]
```



- In the given example, you created a 3D line plot with three sets of data points represented by lists x, y, and z.
- The ax.plot3D function was used to create the line plot.

3D Surface Plots

- 3D surface plots are used to visualize functions of two variables as surfaces in three-dimensional space.

In [ ]:
```
x = np.linspace(-10,10,100)
y = np.linspace(-10,10,100)
```

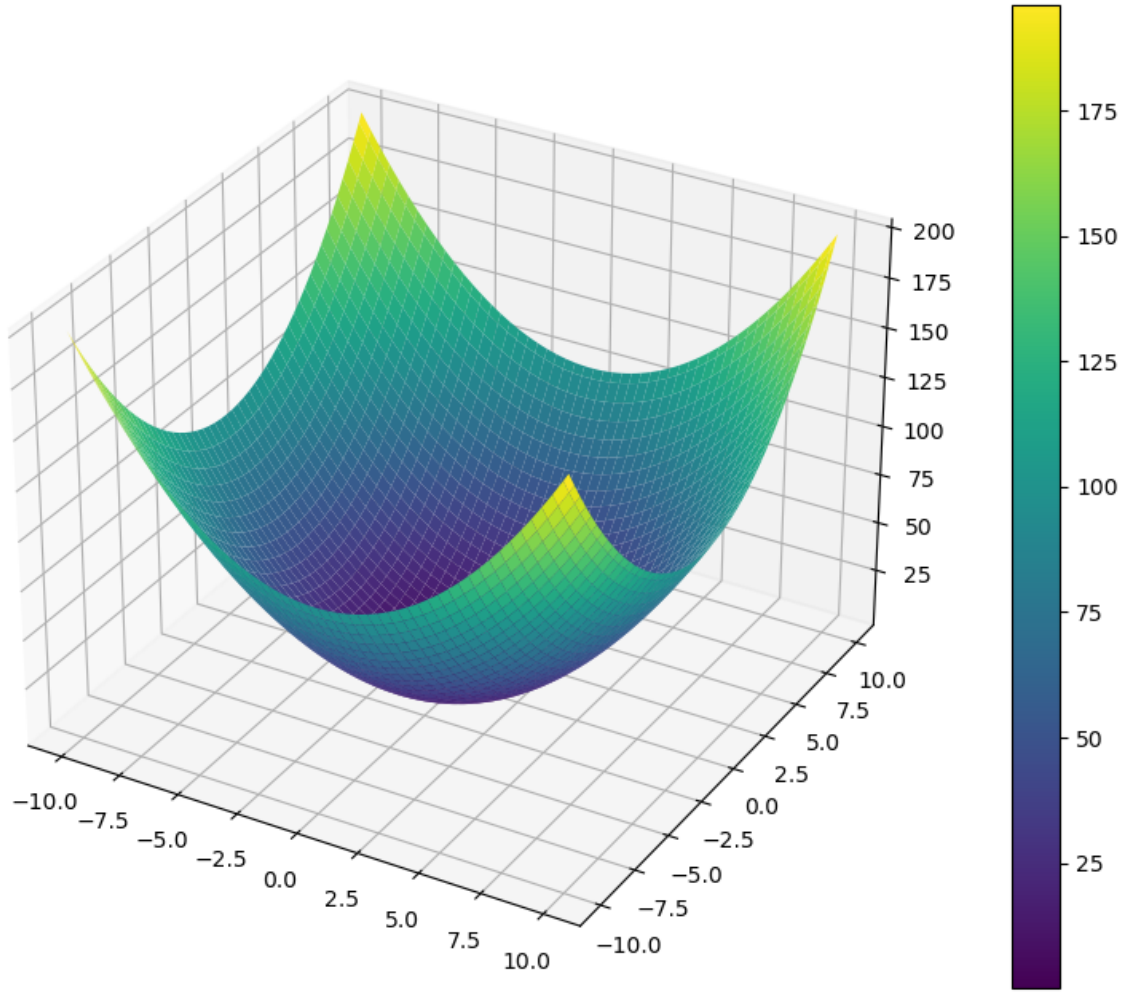
In [ ]:
```
xx, yy = np.meshgrid(x,y)
```

In [ ]:
```
z = xx**2 + yy**2
z.shape
```

Out[ ]:
```
(100, 100)
```

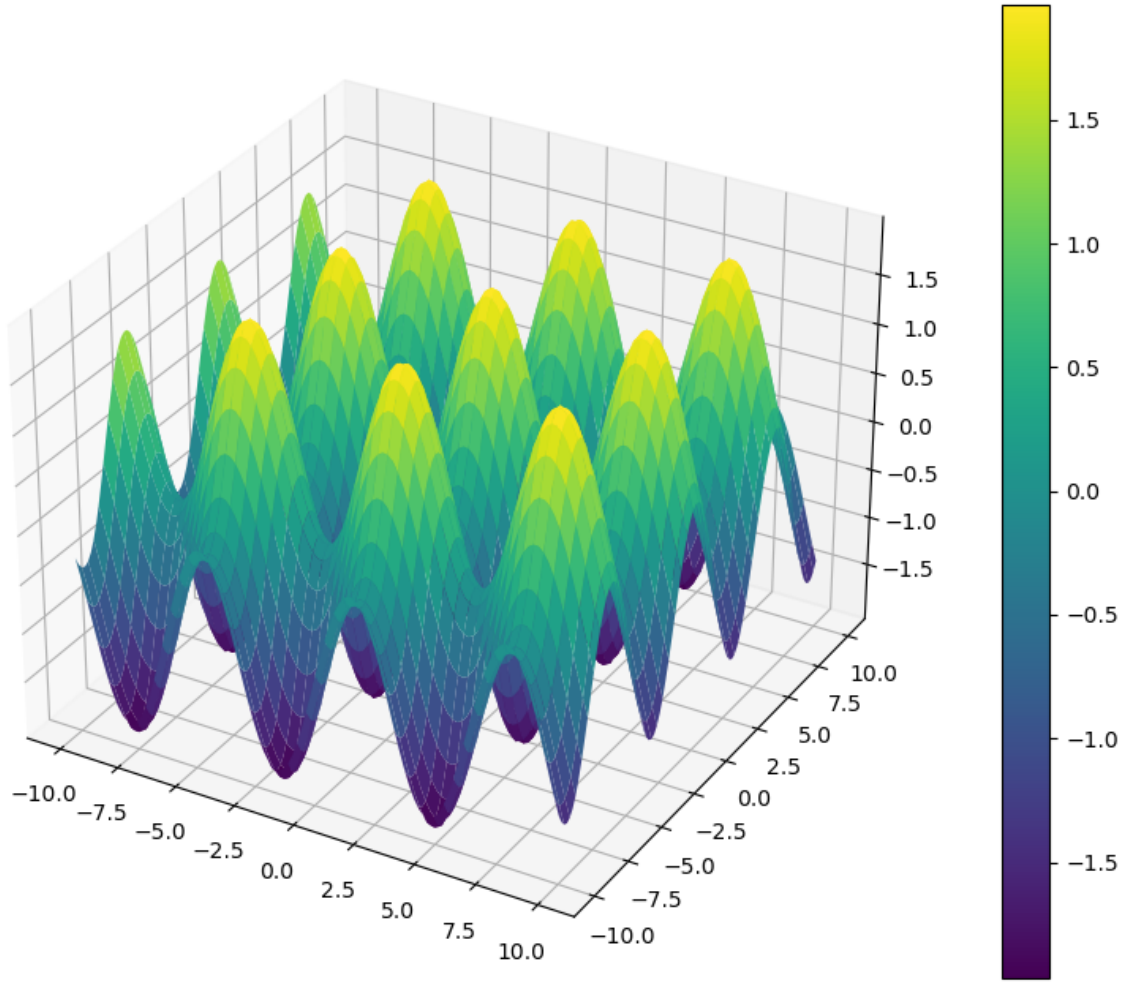
In [ ]:
```
fig = plt.figure(figsize=(12,8))

ax = plt.subplot(projection='3d')

p = ax.plot_surface(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec66ca9a0>
```



In [ ]:
```
z = np.sin(xx) + np.cos(yy)

fig = plt.figure(figsize=(12,8))

ax = plt.subplot(projection='3d')

p = ax.plot_surface(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec33aa520>
```



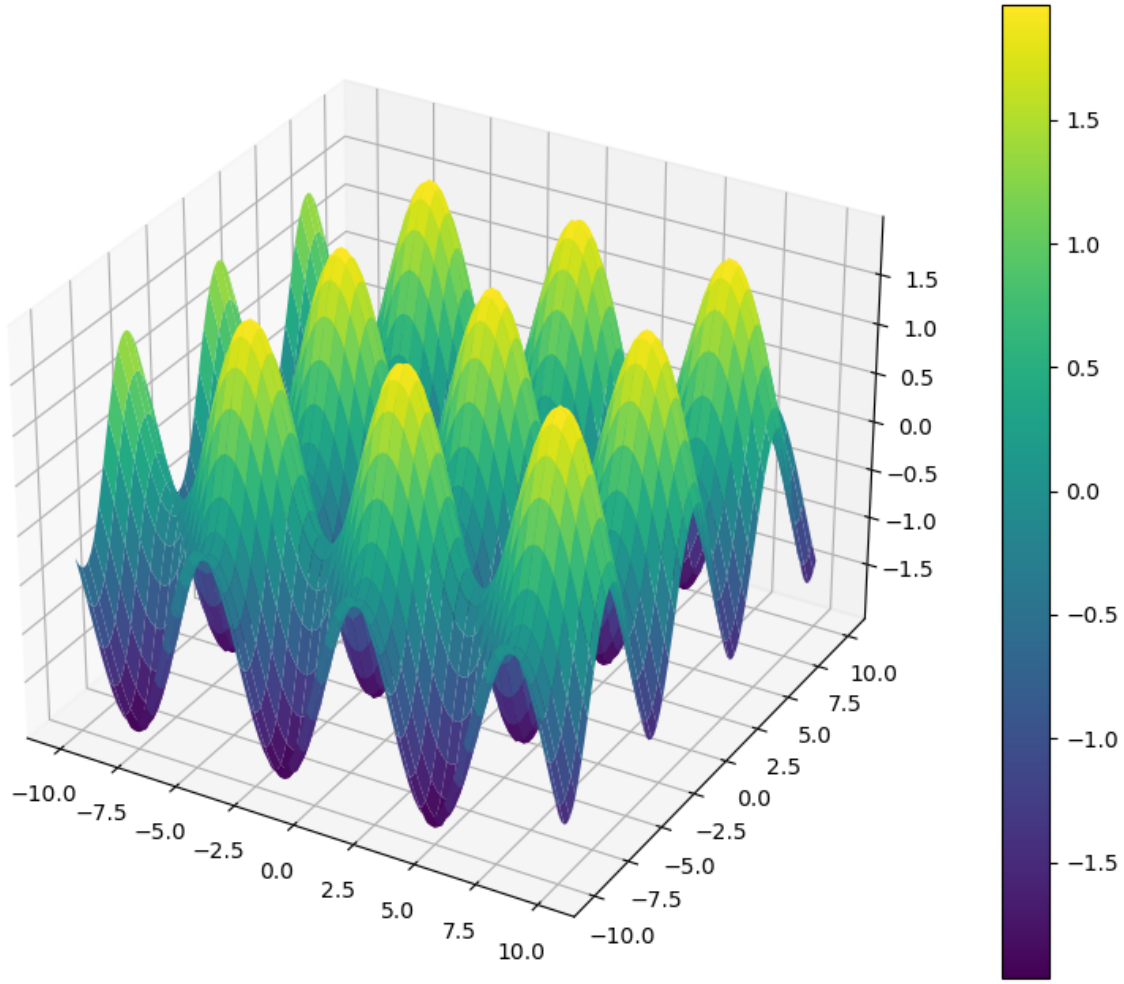
surface plot using the ax.plot_surface function. In First example, you plotted a parabolic surface, and in Seound, you plotted a surface with sine and cosine functions.

Contour Plots

- Contour plots are used to visualize 3D data in 2D, representing data as contours on a 2D plane.

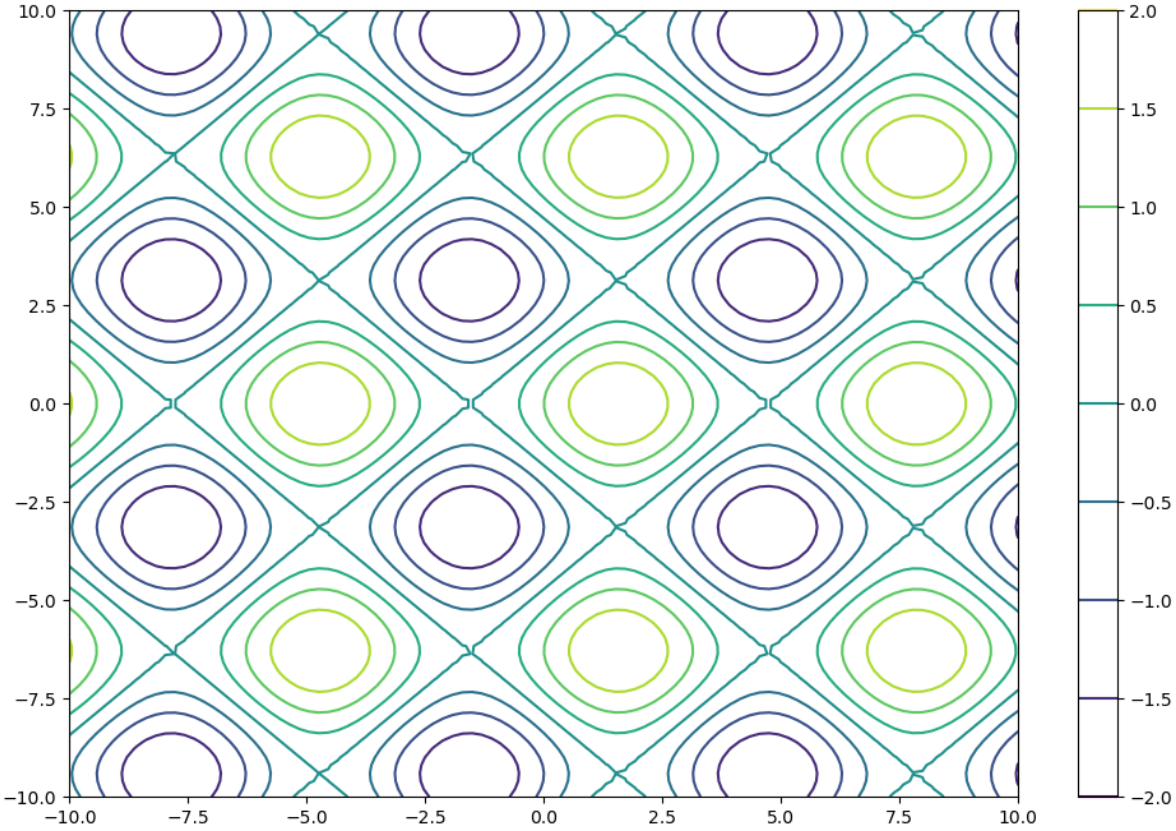
In [ ]:
```
fig = plt.figure(figsize=(12,8))

ax = plt.subplot(projection='3d')

p = ax.plot_surface(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec616e0d0>
```



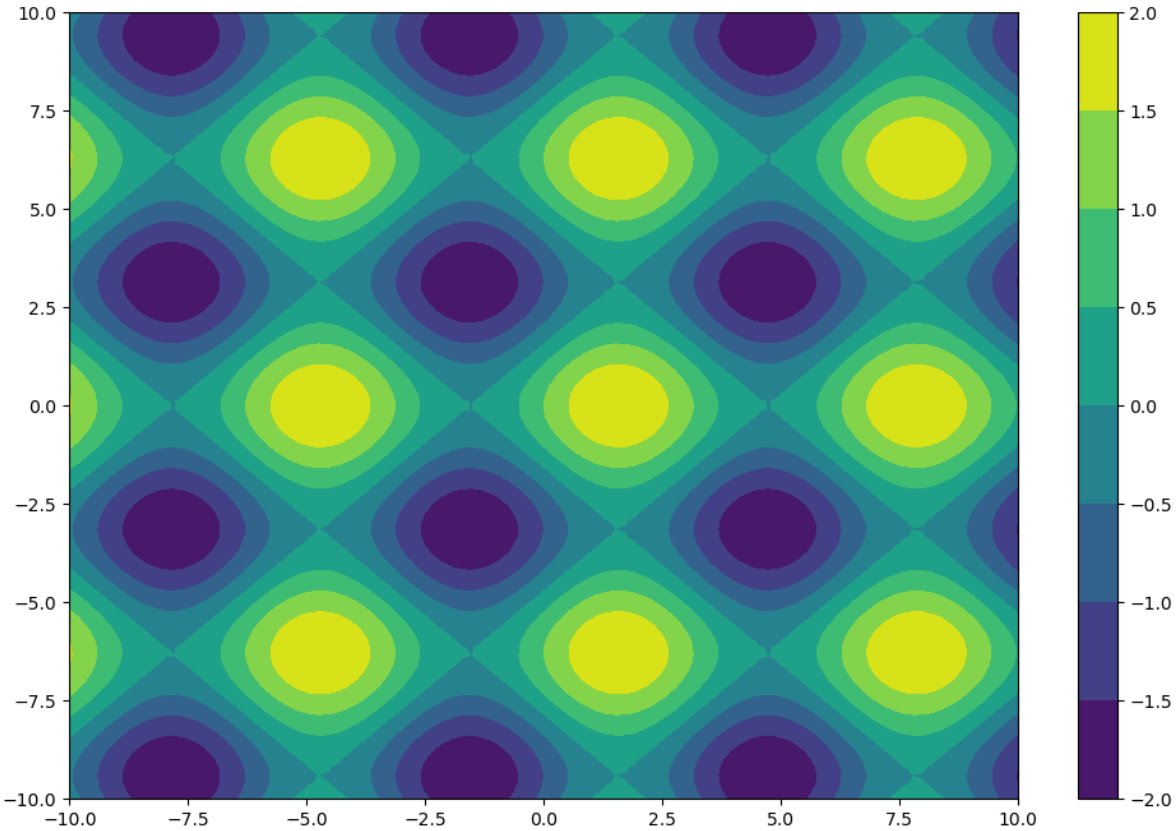
In [ ]:
```
fig = plt.figure(figsize=(12,8))

ax = plt.subplot()

p = ax.contour(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec56e4be0>
```



In [ ]:
```
z = np.sin(xx) + np.cos(yy)

fig = plt.figure(figsize=(12,8))

ax = plt.subplot()

p = ax.contourf(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

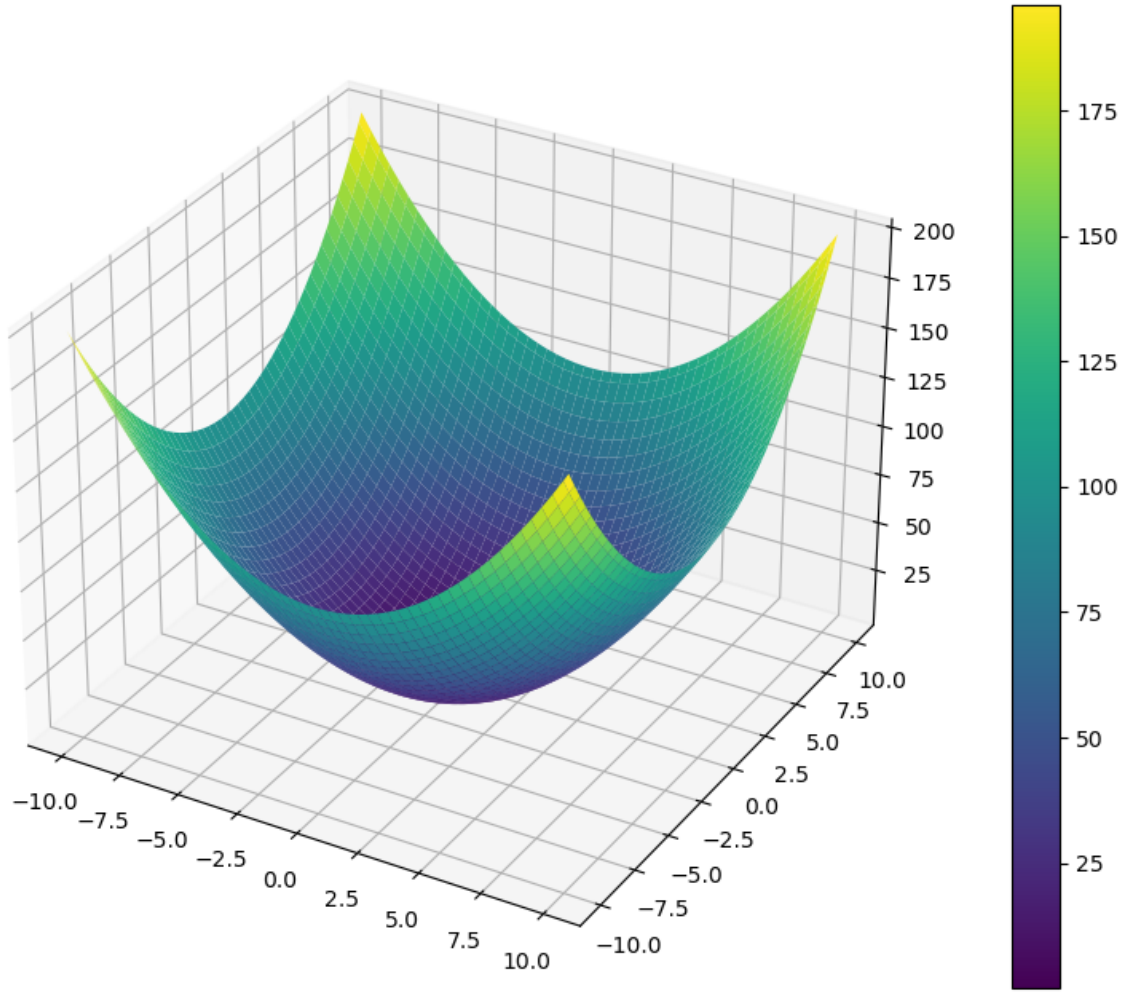
Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec8865f40>
```



You created both filled contour plots (ax.contourf) and contour line plots (ax.contour) in 2D space. These plots are useful for representing functions over a grid.

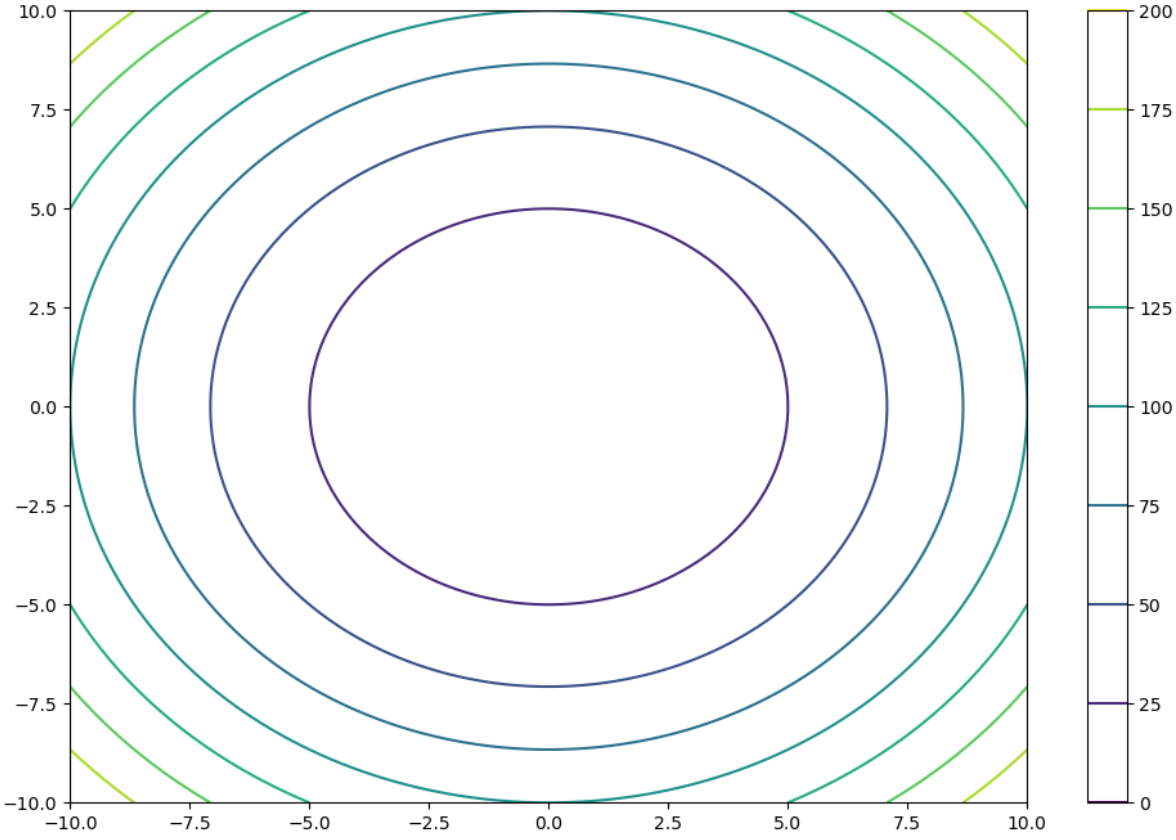
In [ ]:
```
fig = plt.figure(figsize=(12,8))

ax = plt.subplot(projection='3d')

p = ax.plot_surface(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec7b7ca00>
```



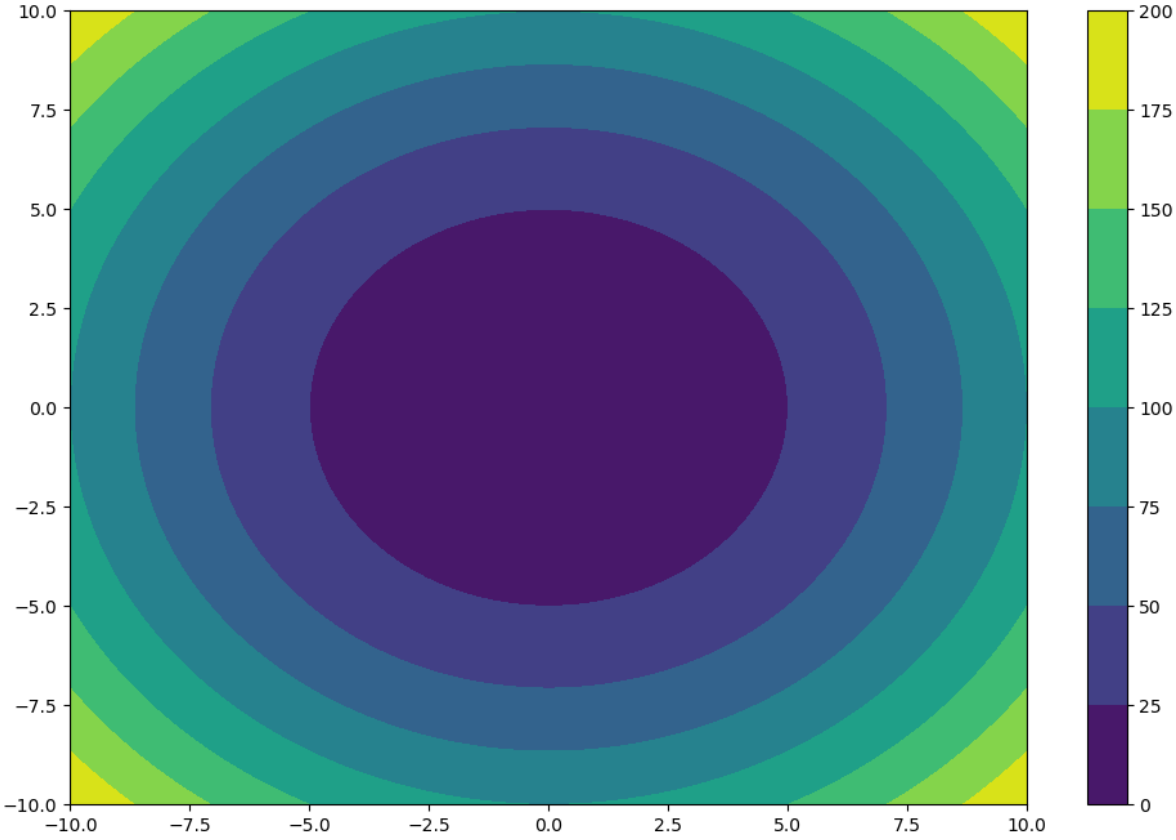
In [ ]:
```
fig = plt.figure(figsize=(12,8))

ax = plt.subplot()

p = ax.contour(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec7c698e0>
```



In [ ]:
```
fig = plt.figure(figsize=(12,8))

ax = plt.subplot()

p = ax.contourf(xx,yy,z,cmap='viridis')
fig.colorbar(p)
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec7ed9700>
```



Heatmap

A heatmap is a graphical representation of data in a 2D grid, where individual values are represented as colors.

In [ ]:
```
delivery = pd.read_csv('Data\Day50\IPL_Ball_by_Ball_2008_2022.csv')
delivery.head()
```

Out[ ]:

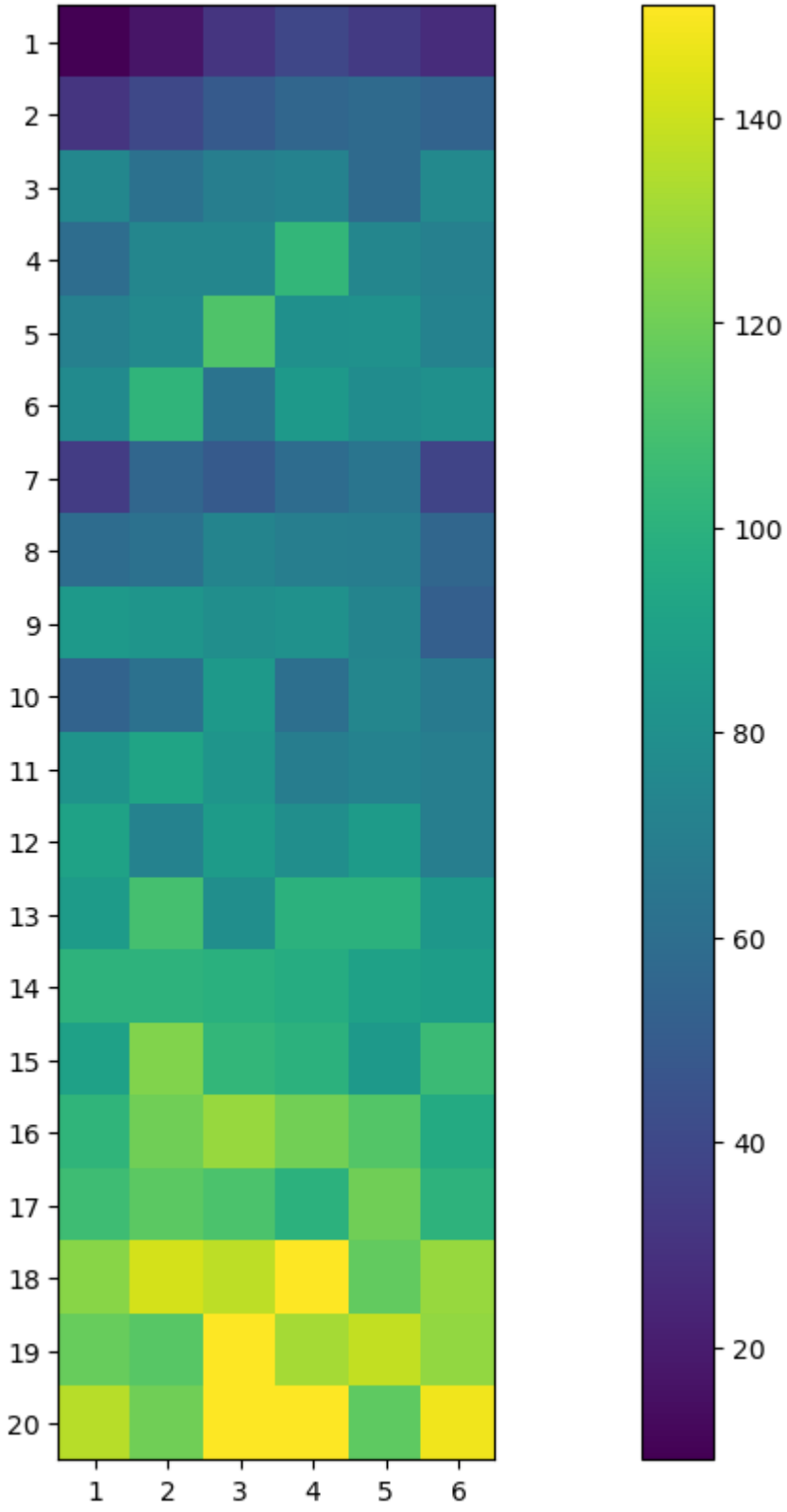
	ID	innings	overs	ballnumber	batter	bowler	non-striker	extra_type	batsman_run	ex
0	1312200	1	0	1	YBK Jaiswal	Mohammed Shami	JC Buttler	NaN	0	
1	1312200	1	0	2	YBK Jaiswal	Mohammed Shami	JC Buttler	legbyes	0	
2	1312200	1	0	3	JC Buttler	Mohammed Shami	YBK Jaiswal	NaN	1	
3	1312200	1	0	4	YBK Jaiswal	Mohammed Shami	JC Buttler	NaN	0	
4	1312200	1	0	5	YBK Jaiswal	Mohammed Shami	JC Buttler	NaN	0	

In [ ]:
```
temp_df = delivery[(delivery['ballnumber'].isin([1,2,3,4,5,6])) & (delivery['batsma
```

In [ ]:
```
grid = temp_df.pivot_table(index='overs',columns='ballnumber',values='batsman_run',
```

In [ ]:
```
plt.figure(figsize=(20,10))
plt.imshow(grid)
plt.yticks(delivery['overs'].unique(), list(range(1,21)))
plt.xticks(np.arange(0,6), list(range(1,7)))
plt.colorbar()
```

Out[ ]:
```
<matplotlib.colorbar.Colorbar at 0x23ec9384820>
```



- In the given example, we used the `imshow` function to create a heatmap of IPL deliveries.
- The grid represented ball-by-ball data with the number of sixes (batsman_run=6) in each over and ball number.
- Heatmaps are effective for visualizing patterns and trends in large datasets.

These techniques provide powerful tools for visualizing complex data in three dimensions and for representing large datasets effectively. Each type of plot is suitable for different types of data and can help in gaining insights from the data.