01 Introduction to Matplotlib & Plots

Matplotlib : Standard Python Visualization Library

- Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms

Layers in Matplotlib

1. **Scripting Layer**
    - This is the highest-level interface where users write scripts to create visualizations It includes functions from the `pyplot` module (e.g., `plt.plot()`, `plt.scatter()`, `plt.show()`)
    - This layer is user-friendly and designed for quick and easy plotting
2. **Object-Oriented Layer**
    - This layer provides a more complex and powerful way to create plots using object-oriented programming
    - Users can create figure and axes objects explicitly, giving more control over the plot elements and layout
    - For example, you can create a figure and add axes with `fig = plt.figure()` and `ax = fig.add_subplot()`
3. **Artist Layer**
    - The artist layer consists of all the drawable elements (artists) in the plot
    - Each artist can be modified individually (e.g., changing colors, adding annotations), allowing for detailed customization of the plot
4. **Backend Layer**
    - The backend layer is responsible for rendering the figure to a specific format or display
    - It translates the high-level plotting commands into lower-level rendering commands
    - Users can switch backends as needed based on the environment (e.g., saving to a file or displaying in a window)

Matplotlib.Pyplot

- One of the core aspects of Matplotlib is `matplotlib.pyplot`
- It is Matplotlib's scripting layer which we studied in details in the videos about Matplotlib

- Recall that it is a collection of command style functions that make Matplotlib work like MATLAB
- Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc

```
# we are using the inline backend
%matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt
```

```
print('Matplotlib version : ', mpl.__version__)  # >= 2.0.0
```

```
print(plt.style.available)
mpl.style.use(['ggplot']) # optional: for ggplot-like style
```

Two types of plotting

- There are two styles/options of plotting with `matplotlib` , plotting using the Artist layer and plotting using the scripting layer.

01 Scripting layer (procedural method)

- using `matplotlib.pyplot` as `plt`
- use `plt` i.e. `matplotlib.pyplot` and add more elements by calling different methods procedurally
- for example, `plt.title(...)` to add title or `plt.xlabel(...)` to add label to the x-axis

```
# Option 1: This is what we have been using so far
df.plot(kind='line', figsize=(20, 10))
plt.title('Option 1')
plt.xlabel('Labels')
plt.ylabel('Values')
```

`annotate` method of the **scripting layer** or the **pyplot interface**.

- `s` : str, the text of annotation
- `xy` : Tuple specifying the (x,y) point to annotate (in this case, end point of arrow)

- `xytext`: Tuple specifying the (x,y) point to place the text (in this case, start point of arrow)
- `xycoords`: The coordinate system that xy is given in - 'data' uses the coordinate system of the object being annotated (default)
- `arrowprops`: Takes a dictionary of properties to draw the arrow
    - `arrowstyle`: Specifies the arrow style, `'->'` is standard arrow
    - `connectionstyle`: Specifies the connection type. `arc3` is a straight line
    - `color`: Specifies color of arrow
    - `lw`: Specifies the line width

02 Artist layer (Object oriented method)

- using an `Axes` instance from `Matplotlib`
- use an `Axes` instance of your current plot and store it in a variable (eg. `ax`)
- add more elements by calling methods with a little change in syntax (by adding "`set_`" to the previous methods)
- for example, use `ax.set_title()` instead of `plt.title()` to add title, or `ax.set_xlabel()` instead of `plt.xlabel()` to add label to the x-axis.

```
# option 2: preferred option with more flexibility
ax = df.plot(kind='line', figsize=(20, 10))
ax.set_title('Option 2')
ax.set_xlabel('Labels')
ax.set_ylabel('Values')
```

Some Pandas

```
import numpy as np  # useful for many scientific computing in Python
import pandas as pd # primary data structure library
```

```
df.head()
# tip: You can specify the number of rows you'd like to see as follows:
df_can.head(10)
```

```
# view the bottom 5 rows of the dataset
df.tail()
```

```
# short Summary of the dataframe
df.info(verbose=False)
```

```python
# to get the list of column headers
df.columns
```

```python
# to get the list of indices
df.index
```

```python
# The default type of intance variables index & columns are NOT list
df.columns.tolist()
df.index.tolist()
```

```python
# size of dataframe (rows, columns)
df.shape
```

```python
# in pandas axis=0 represents rows (default) and axis=1 represents columns
df.drop(['AREA','REG','DEV','Type','Coverage'], axis=1, inplace=True)
```

```python
df.rename(columns={'OdName':'Country', 'AreaName':'Continent',
'RegName':'Region'}, inplace=True)
```

```python
df['Total'] = df.sum(axis=1)
```

```python
# to check how many null objects in the dataset
df.isnull().sum()
```

```python
# view a quick summary of each column in dataframe
df.describe()
```

```python
df.column_name # returns series
df['column']
df[['column1', 'column2']] # returns dataframe
```

```python
df.loc[label]    # filters by the labels of the index/column
df.iloc[index]   # filters by the positions of the index/column
```

```python
df.set_index('Country', inplace=True)
# tip: The opposite of set is reset. So to reset the index, we can use
df.reset_index()
```

```
# optional: to remove the name of the index
df.index.name = None
```

```
# 1. the full row data (all columns)
df.loc['Japan']
```

```
# 2. for year 2013
df.loc['Japan', 2013]
```

```
# 3. for years 1980 to 1985
df.loc['Japan', [1980, 1981, 1982, 1983, 1984, 1984]]
```

```
# to avoid this ambuigity, let's convert the column names into strings
df.columns = list(map(str, df.columns))
```

```
# we can pass multiple criteria in the same line.
# let's filter for AreaNAme = Asia and RegName = Southern Asia

df[(df['Continent']=='Asia') & (df['Region']=='Southern Asia')]

# note: When using 'and' and 'or' operators, pandas requires we use '&'
and '|' instead of 'and' and 'or'
# don't forget to enclose the two conditions in parentheses
```

```
df.sort_values(['Total'], ascending=False, axis=0, inplace=True)
```

Types of Plots

- `line` for line plots
- `bar` for vertical bar plots
- `barh` for horizontal bar plots
- `hist` for histogram
- `box` for boxplot
- `kde` or `density` for density plots
- `area` for area plots
- `pie` for pie plots
- `scatter` for scatter plots
- `hexbin` for hexbin plot

01 Line Plots

- A line chart or line plot is a type of plot which displays information as a series of data points called 'markers' connected by straight line segments
- It is a basic type of chart common in many field
- Use line plot when you have a continuous data set
- These are best suited for trend-based visualizations of data over a period of time

```
# Data
years = [2000, 2005, 2010, 2015, 2020]
values = [100, 200, 300, 400, 500]

# Create a line plot
plt.plot(years, values)
plt.xlabel('Years')
plt.ylabel('Values')
plt.title('Growth Over Years')
plt.show()
```

Customizing Line Plots

- To use `kind="line"` in your code, you would leverage the Pandas DataFrame `plot` method, which simplifies the process of creating various types of plots
- This method internally uses Matplotlib for plotting but provides a more convenient and consistent interface for plotting data directly from Pandas DataFrame

```
# Data
data = { 'Years': [2000, 2005, 2010, 2015, 2020],
                'Values': [100, 200, 300, 400, 500]
}
df = pd.DataFrame(data)

# Create a line plot using Pandas
df.plot(x='Years', y='Values', kind='line', marker='o')
plt.xlabel('Years')
plt.ylabel('Values')
plt.title('Growth Over Years')
plt.show()
```

02 Area Plots

- Visualize plot as a cumulative plot, also knows as a **Stacked Line Plot** or **Area plot**
- Area plots are stacked by default

- To produce a stacked area plot, each column must be either all positive or all negative values (any `NaN`, i.e. not a number, values will default to 0)
- To produce an unstacked plot, set parameter `stacked` to value `False`
- The unstacked plot has a default transparency (alpha value) at 0.5

```
# Data
years = [2000, 2005, 2010, 2015, 2020]
values = [100, 200, 300, 400, 500]

# Create an area plot using fill_between
plt.fill_between(years, values, color='skyblue', alpha=0.5)
plt.xlabel('Years')
plt.ylabel('Values')
plt.title('Growth Over Years')
plt.show()
```

Customizing Area Plots

```
# Data
data = {
    'Years': [2000, 2005, 2010, 2015, 2020],
    'Values': [100, 200, 300, 400, 500]
}
df = pd.DataFrame(data)

# Create an area plot
df.plot(x='Years', y='Values', kind='area', alpha=0.5)
plt.xlabel('Years')
plt.ylabel('Values')
plt.title('Growth Over Years')
plt.show()
```

03 Bar Charts (Dataframe)

- A bar plot is a way of representing data where the *length* of the bars represents the magnitude/size of the feature/variable
- bar graphs usually represent numerical and categorical variables grouped in intervals
- To create a bar plot, we can pass one of two arguments via `kind` parameter in `plot()`
  - `kind=bar` creates a *vertical* bar plot
  - `kind=barh` creates a *horizontal* bar plot

Bar Plot (`bar`)

- A bar plot displays categorical data with rectangular bars with lengths proportional to the values they represent
- The bars can be oriented vertically

```
# Sample data
categories = ['A', 'B', 'C', 'D']
values = [4, 7, 1, 8]

# Create a bar plot
plt.bar(categories, values, color='lightblue')
plt.title('Vertical Bar Plot')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()
```

Horizontal Bar Plot (`barh`)

- A horizontal bar plot is similar to a bar plot but displays the bars horizontally
- This is particularly useful when you have long category names or when you want to emphasize the comparison of values

```
# Data
data = { 'Years': [2000, 2005, 2010, 2015, 2020],
                'Values': [100, 200, 300, 400, 500]
}
df = pd.DataFrame(data)

# Create a line plot using Pandas
df.plot(x='Years', y='Values', kind='barh')
plt.xlabel('Years')
plt.ylabel('Values')
plt.title('Growth Over Years')
plt.show()
```

04 Histogram

- A histogram is a way of representing the *frequency* distribution of numeric dataset
- The way it works is it partitions the x-axis into *bins*, assigns each data point in our dataset to a bin, and then counts the number of data points that have been assigned to each bin
- **Bins** : The range of values is divided into intervals known as bins. Each bin has a certain width, and the bins are contiguous

- **Frequency** : The height of each bar represents the number of data points that fall within that bin
- **Density** : Sometimes, histograms are normalized to show densities instead of frequencies, where the area of each bin represents the proportion of the data

```
# Generate random data
data = np.random.randn(1000)
# Create a histogram
plt.hist(data, bins=30, edgecolor='black')

plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram of Random Data')
plt.show()
```

05 Pie Plots

- A pie chart is a circular statistical graphic that is divided into slices to illustrate numerical proportions
- Each slice of the pie represents a category's contribution to the total, making it easy to visualize the relative sizes of different parts in comparison to the whole
- **Slices**
    - Each slice represents a category's proportion relative to the whole dataset
    - The angle of each slice is proportional to the quantity it represents
- **Total** : The total of all the slices equals 100%, which corresponds to the whole pie
- **Labels** : Each slice is often labeled with the category name and its corresponding percentage or value to provide clarity
- **Colors** : Different colors are usually used for each slice to enhance visual distinction and make the chart easier to read

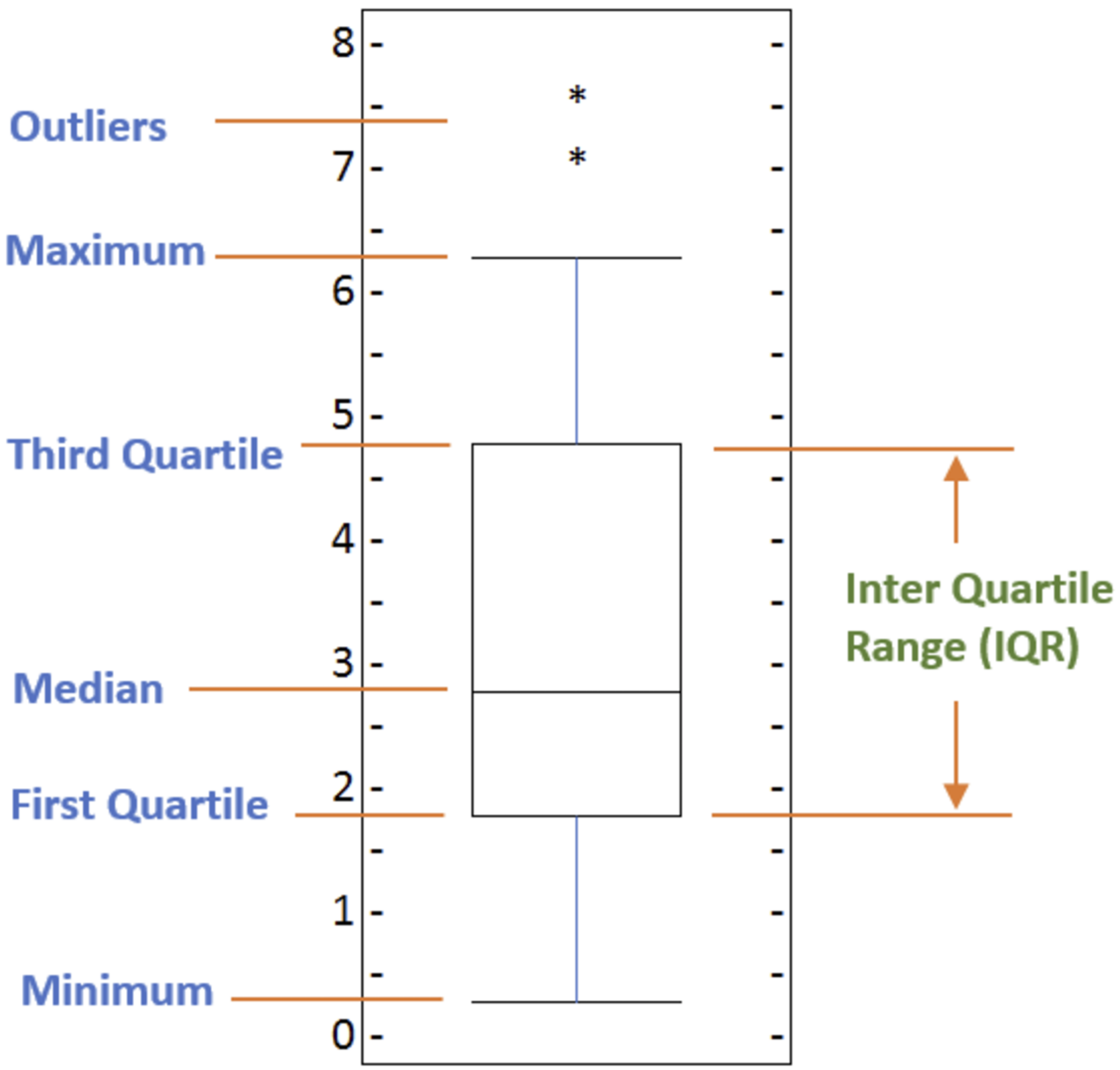
```
# Sample data
labels = ['Category A', 'Category B', 'Category C', 'Category D']
sizes = [15, 30, 45, 10]  # Values representing each category
colors = ['gold', 'lightcoral', 'lightskyblue', 'lightgreen']
explode = (0.1, 0, 0, 0)  # explode the 1st slice (Category A)

# Create the pie chart
# plt.figure(figsize=(8, 8))
plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True, startangle=140)

plt.title('Pie Chart Example')
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a
```

```
circle.
plt.show()
```

- `kind = 'pie'` keyword, along with the following additional parameters
- `autopct` - is a string or function used to label the wedges with their numeric value. The label will be placed inside the wedge. If it is a format string, the label will be `fmt%pct`
- `startangle` - rotates the start of the pie chart by angle degrees counterclockwise from the x-axis
- `shadow` - Draws a shadow beneath the pie (to give a 3D feel)
- `pctdistance` - Push out the percentages to sit just outside the pie chart

06 Box Plots

- A box plot (or whisker plot) is a standardized way to display the distribution of a dataset based on a five-number summary: minimum, first quartile (Q1), median (Q2), third quartile (Q3), and maximum



- It provides a visual representation of the central tendency, variability, and potential outliers in the data
    - **Minimum :** The smallest number in the dataset excluding the outliers
    - **First quartile :** Middle number between the `minimum` and the `median`
    - **Second quartile (Median) :** Middle number of the (sorted) dataset
    - **Third quartile :** Middle number between `median` and `maximum`
    - **Maximum :** The largest number in the dataset excluding the outliers

```
# Sample data
data = {
    'Group A': [12, 15, 14, 10, 14, 13, 15, 16, 20, 19],
    'Group B': [22, 21, 20, 24, 25, 21, 20, 22, 30, 29],
    'Group C': [32, 31, 30, 29, 35, 33, 28, 30, 40, 38]
}
df = pd.DataFrame(data)

# Create a box plot
# plt.figure(figsize=(10, 6))

# df.boxplot()
df.plot(kind='box', grid=True)

plt.title('Box Plot Example')
plt.ylabel('Values')
plt.xlabel('Groups')
plt.grid(axis='y')
plt.show()
```

- `vert` parameter in the **plot** function and assign it to `False` for horizontal box plots
- `color` to specify a different color

07 Scatter Plots

- A `scatter plot` (2D) is a useful method of comparing variables against each other
- `Scatter` plots look similar to `line plots` in that they both map independent and dependent variables on a 2D graph
- With further analysis using tools like regression, we can mathematically calculate this relationship b/w points and use it to predict trends outside the dataset
- To investigate the relationship or correlation between two continuous variables
- To identify patterns, trends, clusters, and outliers in the data
- To visualize the distribution of data points in two dimension

```
# Sample data
data = {
'Height': [150, 160, 170, 180, 190, 200, 210, 220],
'Weight': [50, 55, 60, 70, 75, 85, 90, 95]
}
df = pd.DataFrame(data)

# Create a scatter plot
# plt.figure(figsize=(10, 6))
# df.plot(kind='scatter', x='Height', y='Weight', color='blue')
plt.scatter(df['Height'], df['Weight'], color='blue', alpha=0.5)

plt.title('Scatter Plot of Height vs. Weight')
plt.xlabel('Height (cm)')
plt.ylabel('Weight (kg)')
plt.show()
```

08 Bubble Plots

- A `bubble plot` is a variation of the `scatter plot` that displays three dimensions of data (x, y, z)
- The data points are replaced with bubbles, and the size of the bubble is determined by the third variable `z`, also known as the weight
- In `maplotlib`, we can pass in an array or scalar to the parameter `s` to `plot()`, that contains the weight of each point

```
# Sample data
data = {
    'Years': [2000, 2005, 2010, 2015, 2020],
    'Values': [100, 200, 300, 400, 500],
    'Growth': [1.2, 2.5, 3.0, 4.2, 5.5]  # This will be represented by
bubble size
}
df = pd.DataFrame(data)

# Create a bubble plot
# ax = df.plot(kind='scatter', x='Years', y='Values', s=df['Growth']*100,
alpha=0.5, c='blue', edgecolor='w', linewidth=1.2)

plt.scatter(df['Years'], df['Values'], s=df['Growth']*100, alpha=0.5,
c='blue', edgecolors='w', linewidth=1.5)

# plt.figure(figsize=(10, 6))
plt.xlabel('Years')
plt.ylabel('Values')
```

```
plt.title('Bubble Plot of Growth Over Years')
plt.show()
```