2024S2 - IT2313

Programming for Data Science

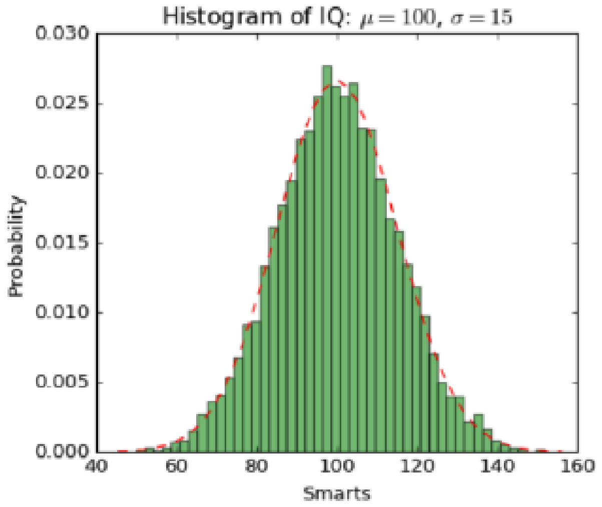
Data Visualisation with Matplotlib

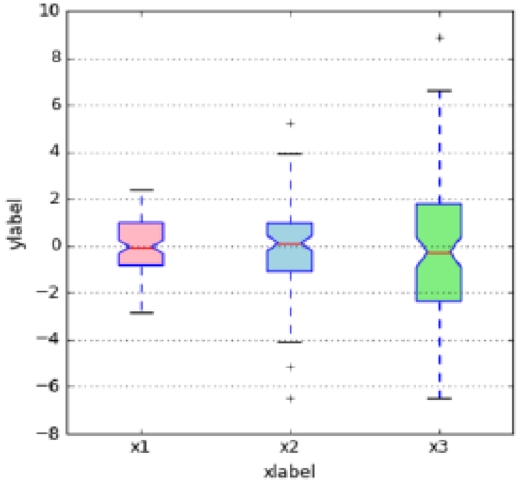
Matplotlib and Seaborn

Data Visualisation

Very important in Data Analysis

- Explore data
- Report insights
- Trend analysis





Introduction on Matplotlib and Seaborn

Matplotlib and Seaborn

Python's two most widely used data visualization libraries are Matplotlib and Seaborn. While both libraries are designed to create high-quality graphics and visualizations, they have several key differences that make them better suited for different use cases.

Matplotlib is a low-level plotting library that provides a wide range of tools for creating highly customizable visualizations. It is a highly flexible library, allowing users to create almost any type of plot they can imagine. This flexibility comes at the cost of a steeper learning curve and more verbose code.

Seaborn, on the other hand, is a high-level interface for creating statistical graphics. It is built on top of Matplotlib and provides a simpler, more intuitive interface for creating common statistical plots. Seaborn is designed to work with Pandas dataframes, making it easy to create visualizations with minimal code. It also offers a range of built-in statistical functions, allowing users to easily perform complex statistical analyses with their visualizations.

Basic Plots with Matplotlib

Basic Plots with Matplotlib

Commonly Used Graphical Plots

Each type of plot serves a different purpose, and the choice depends on the nature of your data and the story you want to tell. Here's a brief description of the perspectives each plot provides:

S/N	Name of Plot	Purpose	When to use
1	Line Plot	Shows the trend or change in a variable over a continuous range, such as time.	Useful for visualizing trends, patterns, or relationships in data over time or any ordered sequence.
2	Bar Plot	Compares values between different categories or groups.	Effective for comparing discrete categories and showing the magnitude of differences between them.
3	Scatter Plot	Displays individual data points in two dimensions and shows how they relate to each other.	Ideal for visualizing relationships and identifying patterns or outliers in bivariate data.
4	Area Plot	Shows the cumulative contribution of individual data points to a whole over a continuous range.	Useful for visualizing the overall trend or composition of a dataset over time or any ordered sequence.

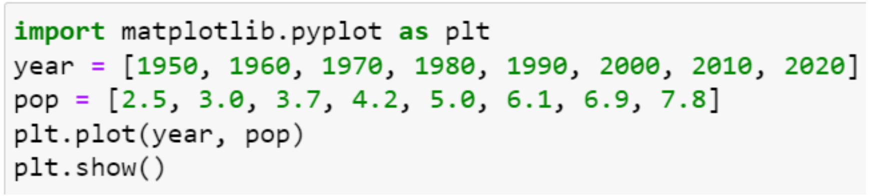
Basic Plots with Matplotlib

Commonly Used Graphical Plots

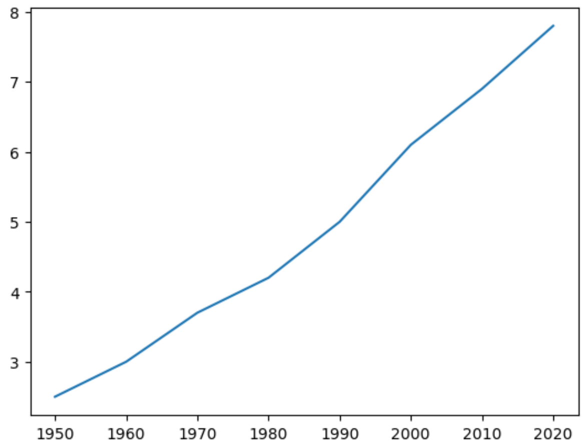
S/N	Name of Plot	Purpose	When to use
5	Pie Chart	Represents the distribution of parts within a whole and highlights the proportion of each category relative to the total.	Want to show the composition or percentage contribution of different categories to a whole. Suitable for displaying a small number of categories (3-7) to avoid visual clutter.
6	Donut Chart	Similar to a Pie Chart but with a hole in the center (donut shape).	When you want to present the same information as a Pie Chart but with a more visually appealing and distinctive appearance. Useful when you want to draw attention to the entire dataset while still showing the distribution.
7	Histogram	Represents the distribution of a continuous dataset by dividing it into bins and displaying the frequency of each bin.	Helpful for understanding the underlying distribution of a variable and identifying patterns or outliers.
8	Box Plot	Summarizes the distribution of a dataset, highlighting the median, quartiles, and potential outliers.	Useful for comparing the distribution of multiple datasets and identifying variability and skewness.

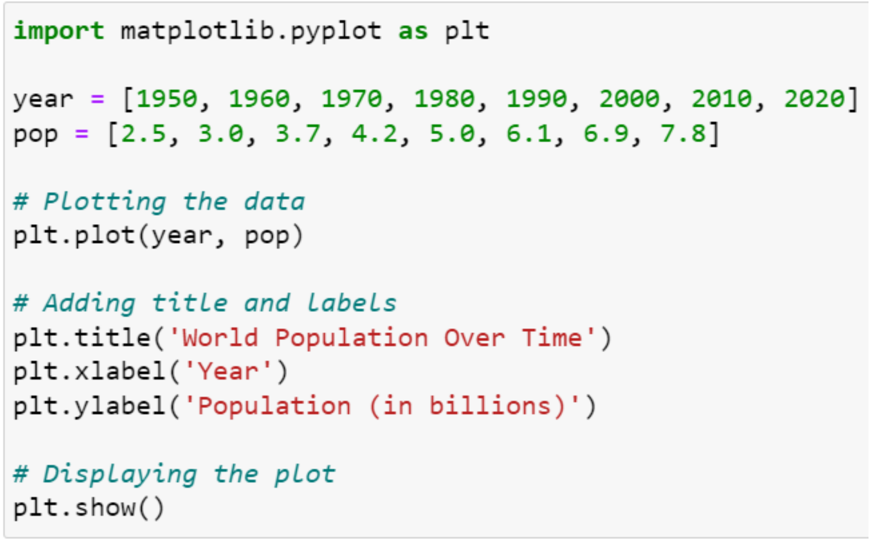
Basic Plots with Matplotlib

Line Plot

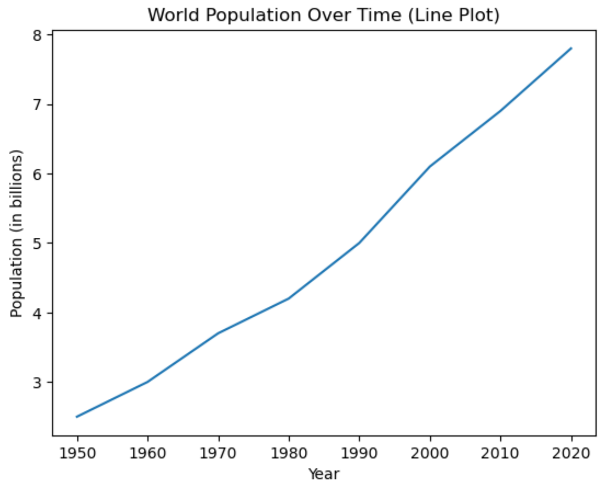


```python
import matplotlib.pyplot as plt
year = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]
pop = [2.5, 3.0, 3.7, 4.2, 5.0, 6.1, 6.9, 7.8]
plt.plot(year, pop)
plt.show()
```



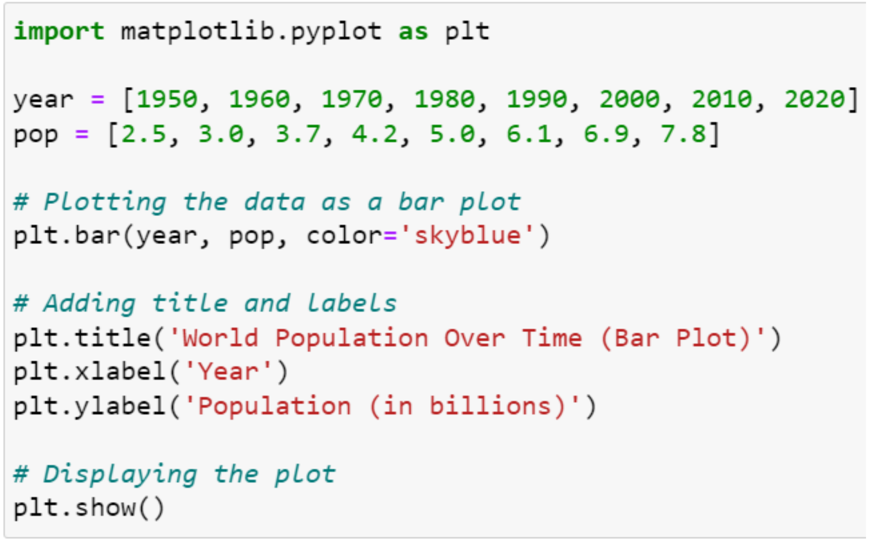


```python
import matplotlib.pyplot as plt

year = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]
pop = [2.5, 3.0, 3.7, 4.2, 5.0, 6.1, 6.9, 7.8]

# Plotting the data
plt.plot(year, pop)

# Adding title and labels
plt.title('World Population Over Time')
plt.xlabel('Year')
plt.ylabel('Population (in billions)')

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

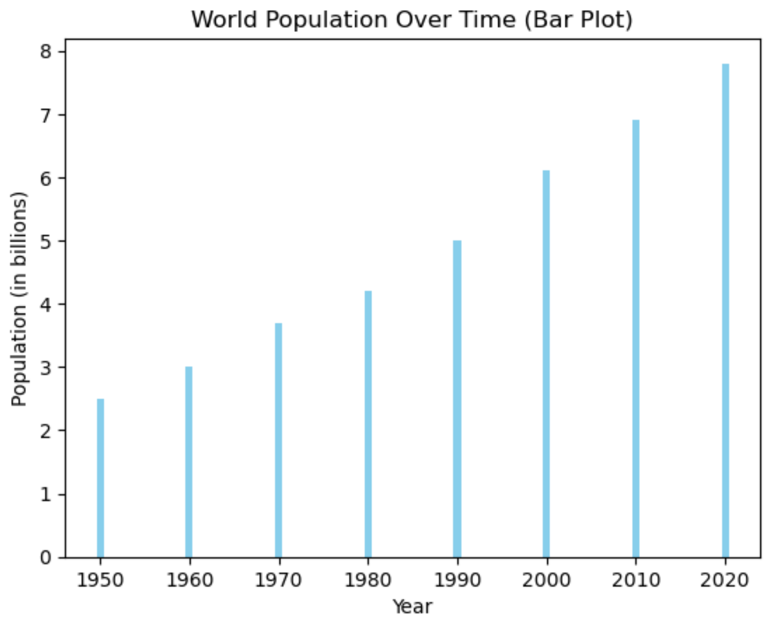


Basic Plots with Matplotlib

Bar Plot

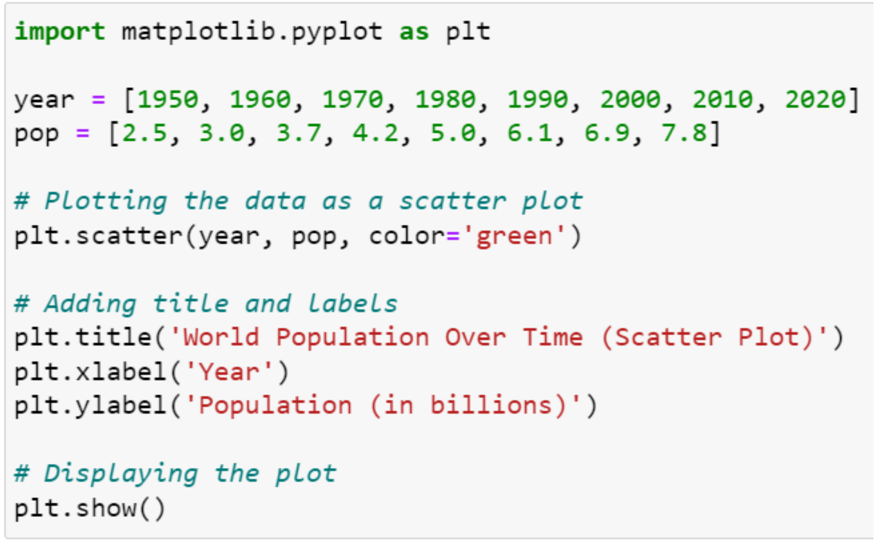


```python
import matplotlib.pyplot as plt

year = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]
pop = [2.5, 3.0, 3.7, 4.2, 5.0, 6.1, 6.9, 7.8]

# Plotting the data as a bar plot
plt.bar(year, pop, color='skyblue')

# Adding title and labels
plt.title('World Population Over Time (Bar Plot)')
plt.xlabel('Year')
plt.ylabel('Population (in billions)')

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

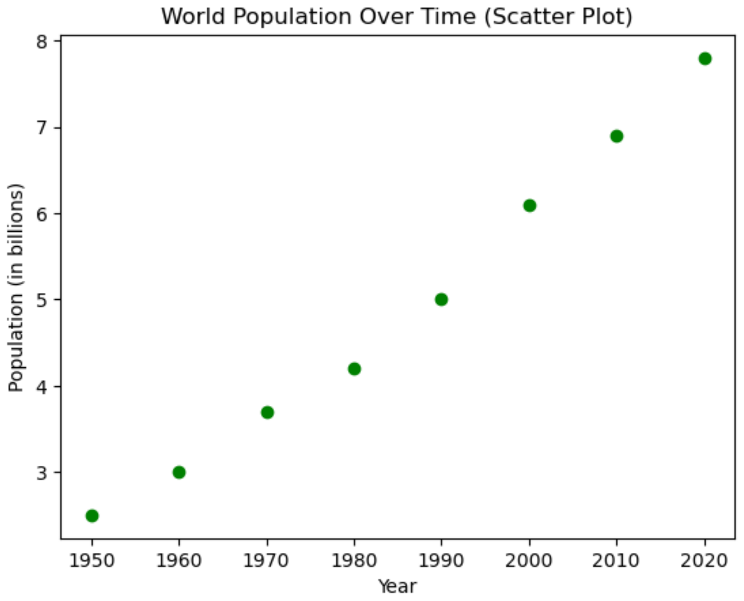


Basic Plots with Matplotlib

Scatter Plot



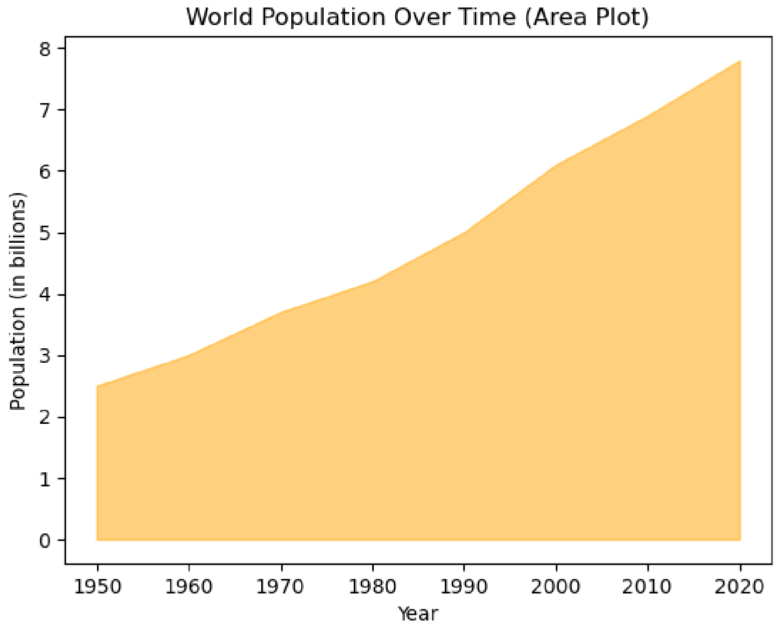
```
import matplotlib.pyplot as plt

year = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]
pop = [2.5, 3.0, 3.7, 4.2, 5.0, 6.1, 6.9, 7.8]

# Plotting the data as a scatter plot
plt.scatter(year, pop, color='green')

# Adding title and labels
plt.title('World Population Over Time (Scatter Plot)')
plt.xlabel('Year')
plt.ylabel('Population (in billions)')

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



Basic Plots with Matplotlib

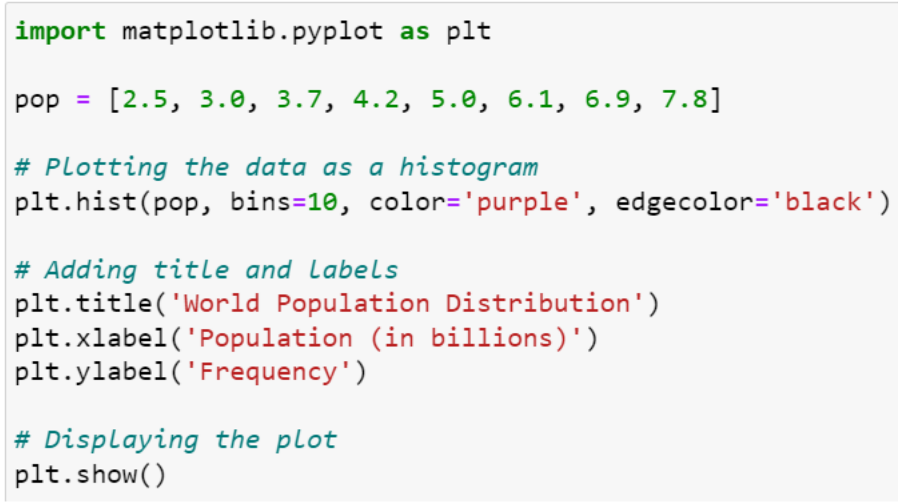
Area Plot

```
import matplotlib.pyplot as plt

year = [1950, 1960, 1970, 1980, 1990, 2000, 2010, 2020]
pop = [2.5, 3.0, 3.7, 4.2, 5.0, 6.1, 6.9, 7.8]

# Plotting the data as an area plot
plt.fill_between(year, pop, color='orange', alpha=0.5)

# Adding title and labels
plt.title('World Population Over Time (Area Plot)')
plt.xlabel('Year')
plt.ylabel('Population (in billions)')

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

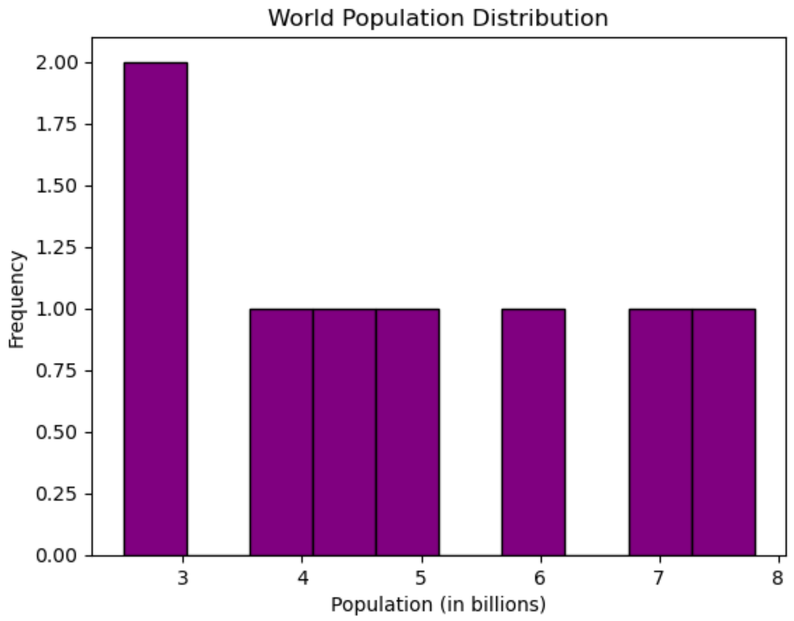


Basic Plots with Matplotlib

Histogram

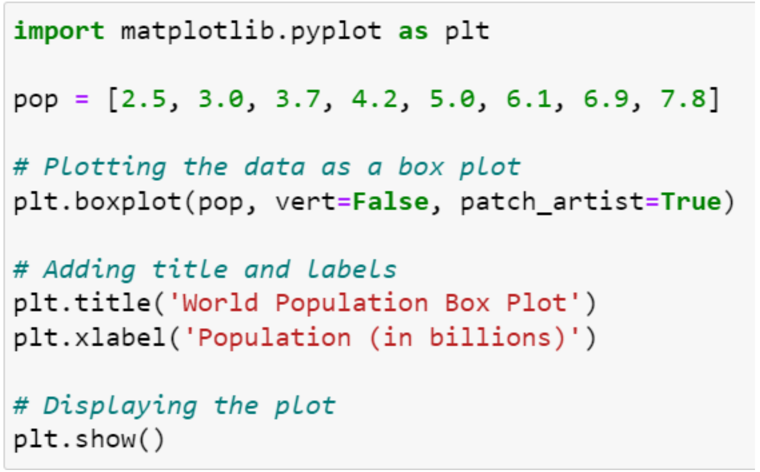


```
import matplotlib.pyplot as plt

pop = [2.5, 3.0, 3.7, 4.2, 5.0, 6.1, 6.9, 7.8]

# Plotting the data as a histogram
plt.hist(pop, bins=10, color='purple', edgecolor='black')

# Adding title and labels
plt.title('World Population Distribution')
plt.xlabel('Population (in billions)')
plt.ylabel('Frequency')

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

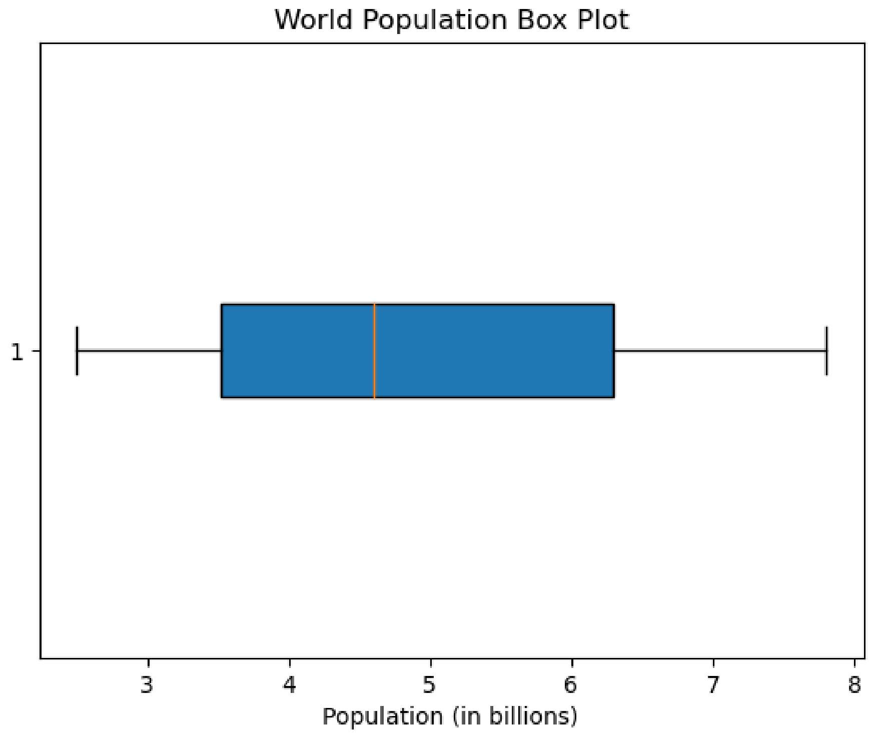


Basic Plots with Matplotlib

Box Plot

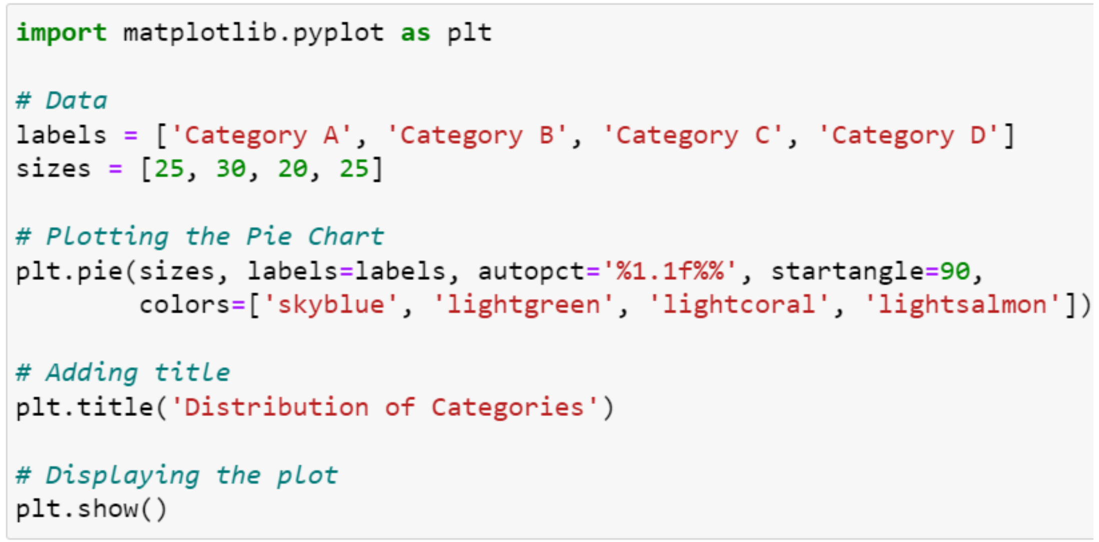


```
import matplotlib.pyplot as plt

pop = [2.5, 3.0, 3.7, 4.2, 5.0, 6.1, 6.9, 7.8]

# Plotting the data as a box plot
plt.boxplot(pop, vert=False, patch_artist=True)

# Adding title and labels
plt.title('World Population Box Plot')
plt.xlabel('Population (in billions)')

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

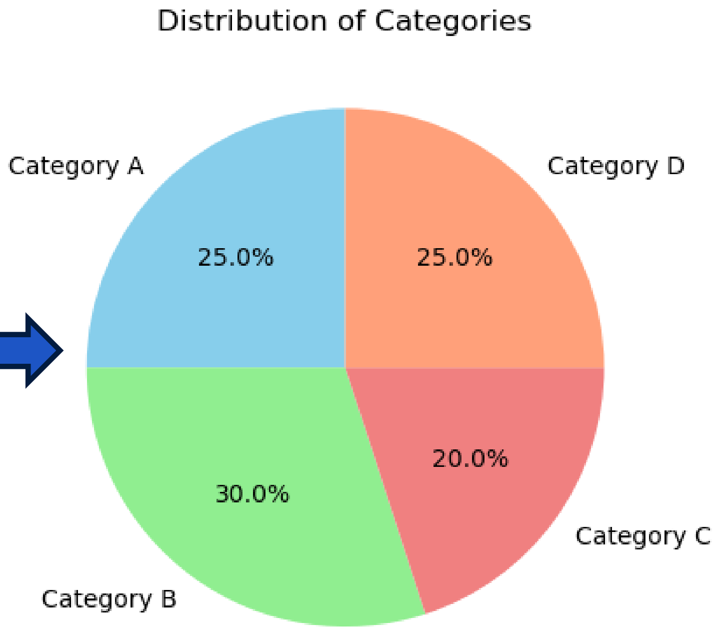


Basic Plots with Matplotlib

Pie Chart

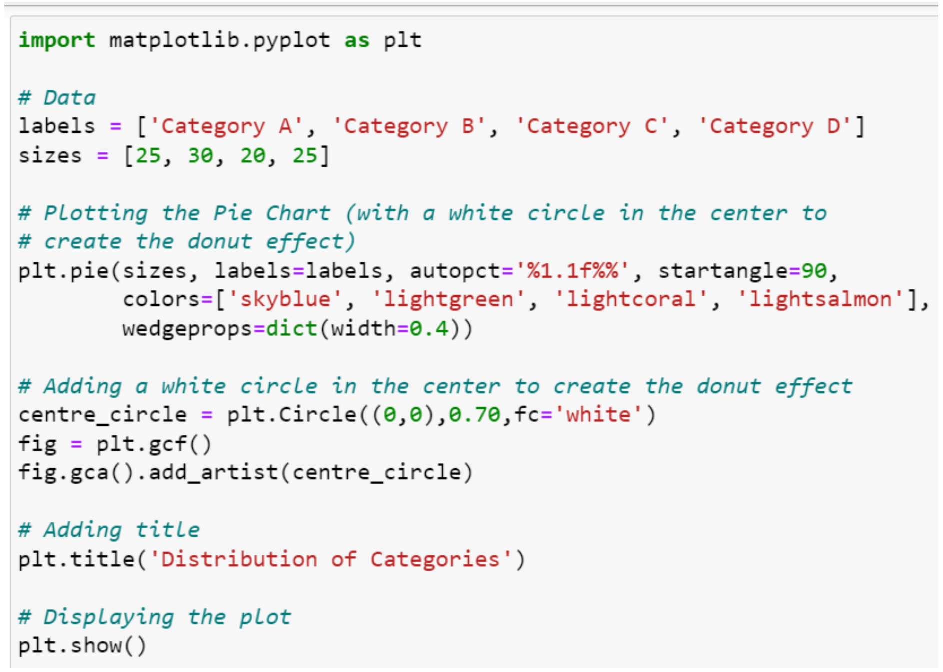


```
import matplotlib.pyplot as plt

# Data
labels = ['Category A', 'Category B', 'Category C', 'Category D']
sizes = [25, 30, 20, 25]

# Plotting the Pie Chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90,
        colors=['skyblue', 'lightgreen', 'lightcoral', 'lightsalmon'])

# Adding title
plt.title('Distribution of Categories')

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

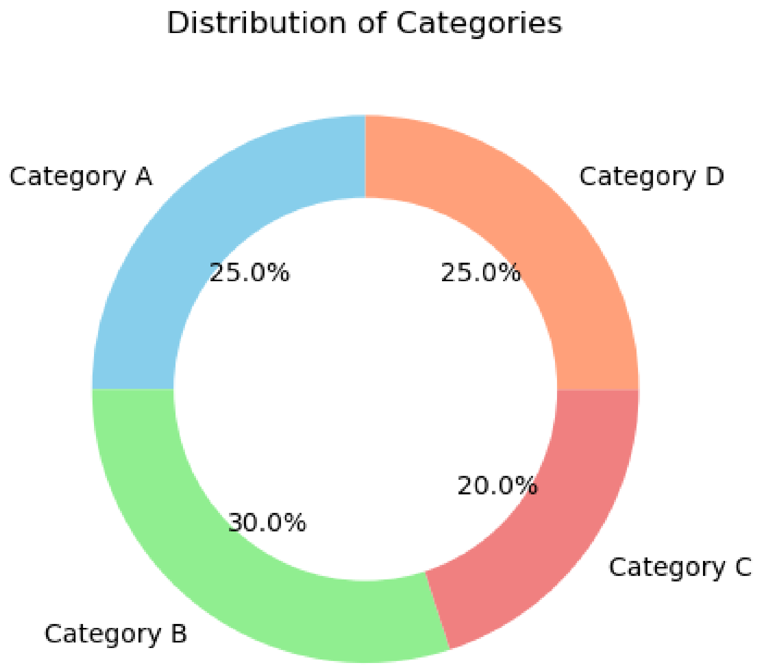


Basic Plots with Matplotlib

Donut Chart



```python
import matplotlib.pyplot as plt

# Data
labels = ['Category A', 'Category B', 'Category C', 'Category D']
sizes = [25, 30, 20, 25]

# Plotting the Pie Chart (with a white circle in the center to
# create the donut effect)
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90,
        colors=['skyblue', 'lightgreen', 'lightcoral', 'lightsalmon'],
        wedgeprops=dict(width=0.4))

# Adding a white circle in the center to create the donut effect
centre_circle = plt.Circle((0,0),0.70,fc='white')
fig = plt.gcf()
fig.gca().add_artist(centre_circle)

# Adding title
plt.title('Distribution of Categories')

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



“Object-Oriented" Approach in Matplotlib

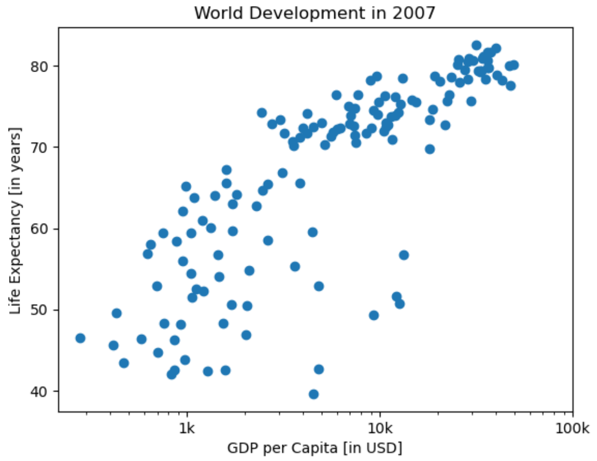
“Object-Oriented" Approach in Matplotlib

In the "object-oriented" approach of Matplotlib, you explicitly create a figure and axis, utilizing methods of the axis object for plotting and customization. This method offers direct interaction with Figure and Axes objects, providing finer control over plot elements. It is favored for its flexibility and power, particularly in handling multiple subplots or intricate visualizations.

fig, ax = plt.subplots()

- plt.subplots() returns a figure object (fig) and an axis object (ax).
- The figure is the whole window or page that everything is drawn on.
- The axis is the part of the figure where the data is plotted.

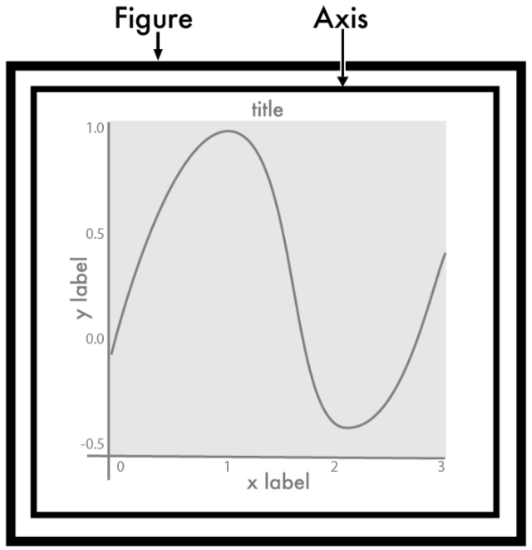
By creating the figure and axis explicitly, you gain more control over the layout and properties of the plot. The ax object is then used to perform various plotting and customization tasks.



“Object-Oriented" Approach in Matplotlib

There are two primary objects associated with a matplotlib plot:

- figure object: the overall figure space that can contain one or more plots.
- Axis objects: the individual plots that are rendered within the figure.



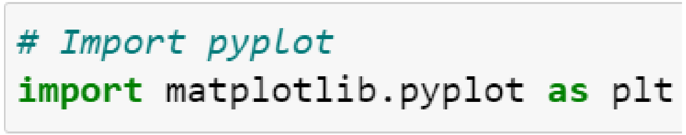
You can think of the figure object as your plot canvas. You can think about the axis object as an individual plot. A figure can hold one or more axis objects. This structure allows you to create figures with one or more plots on them.

“Object-Oriented" Approach in Matplotlib

While Matplotlib contains many modules that provide different plotting functionality, the most commonly used module is pyplot.

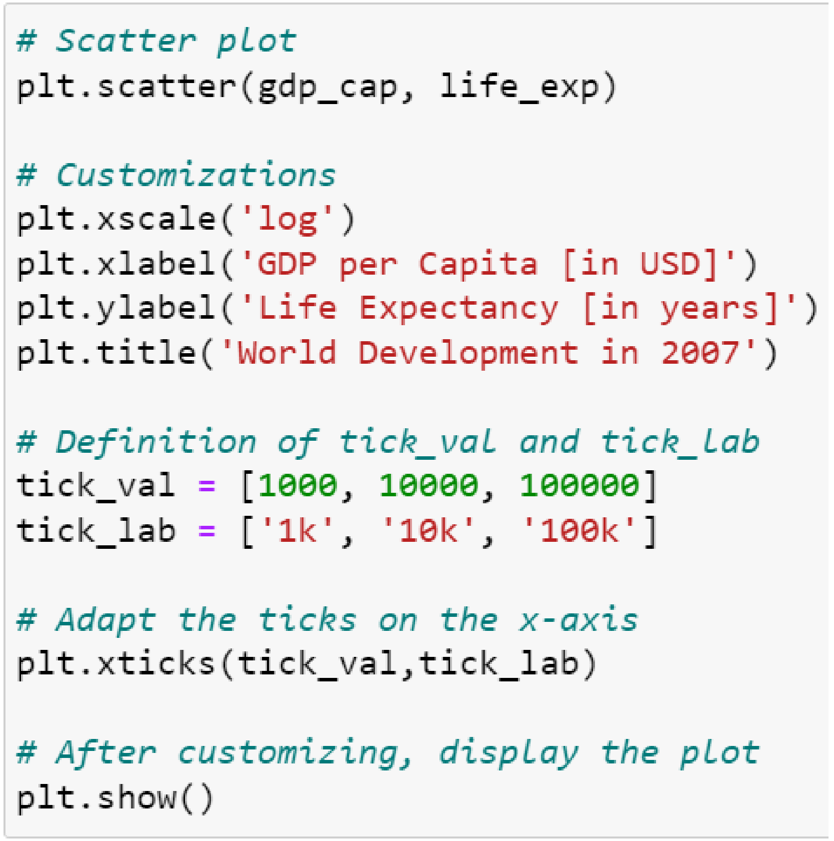
Pyplot provides methods that can be used to add different components to figure objects, including creating the individual plots as axis objects, also known as subplots.

The pyplot module is typically imported using the alias plt as demonstrated below.

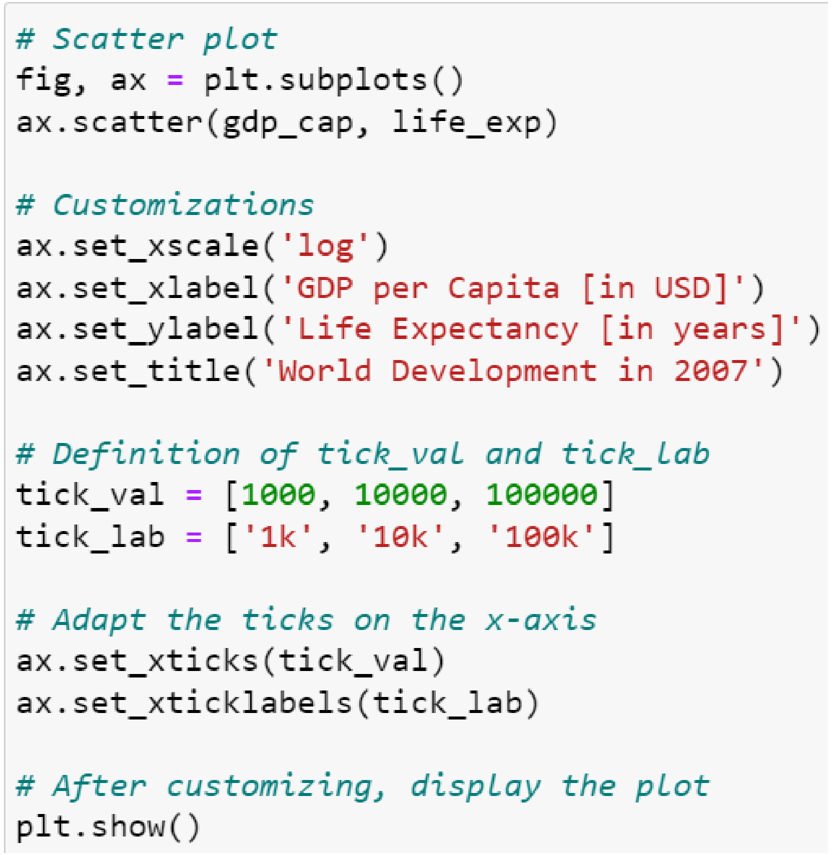


```python
# Import pyplot
import matplotlib.pyplot as plt
```

“Object-Oriented" Approach in Matplotlib



```python
# Scatter plot
plt.scatter(gdp_cap, life_exp)

# Customizations
plt.xscale('log')
plt.xlabel('GDP per Capita [in USD]')
plt.ylabel('Life Expectancy [in years]')
plt.title('World Development in 2007')

# Definition of tick_val and tick_lab
tick_val = [1000, 10000, 100000]
tick_lab = ['1k', '10k', '100k']

# Adapt the ticks on the x-axis
plt.xticks(tick_val,tick_lab)

# After customizing, display the plot
plt.show()
```



```python
# Scatter plot
fig, ax = plt.subplots()
ax.scatter(gdp_cap, life_exp)

# Customizations
ax.set_xscale('log')
ax.set_xlabel('GDP per Capita [in USD]')
ax.set_ylabel('Life Expectancy [in years]')
ax.set_title('World Development in 2007')

# Definition of tick_val and tick_lab
tick_val = [1000, 10000, 100000]
tick_lab = ['1k', '10k', '100k']

# Adapt the ticks on the x-axis
ax.set_xticks(tick_val)
ax.set_xticklabels(tick_lab)

# After customizing, display the plot
plt.show()
```

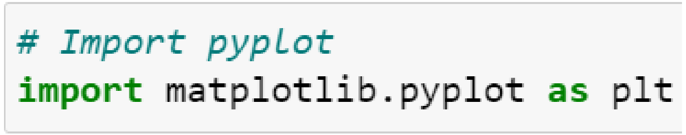
Create Plots Using Matplotlib

Create Plots Using Matplotlib

While Matplotlib contains many modules that provide different plotting functionality, the most commonly used module is pyplot.

Pyplot provides methods that can be used to add different components to figure objects, including creating the individual plots as axis objects, also known as subplots.

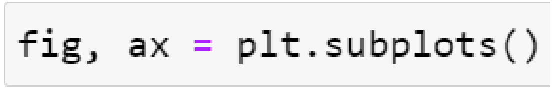
The pyplot module is typically imported using the alias plt as demonstrated below.



```python
# Import pyplot
import matplotlib.pyplot as plt
```

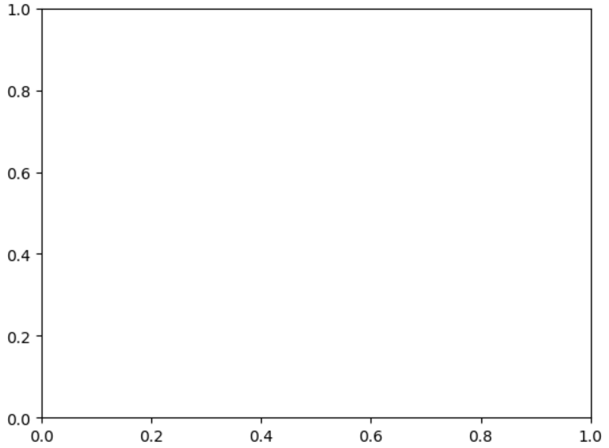
Create Plots Using Matplotlib

To create a plot using matplotlib’s object oriented approach, you first create the figure (which you can call fig) and at least one axis (which you can call ax) using the subplots() function from the pyplot module.



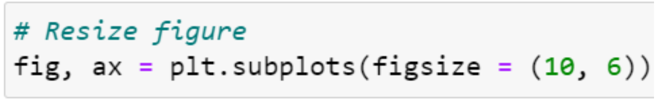
```
fig, ax = plt.subplots()
```

Notice that the fig and ax are created at the same time by setting them equal to the output of the pyplot.subplots() function. As no other arguments have been provided, the result is a figure with one plot that is empty but ready for data.

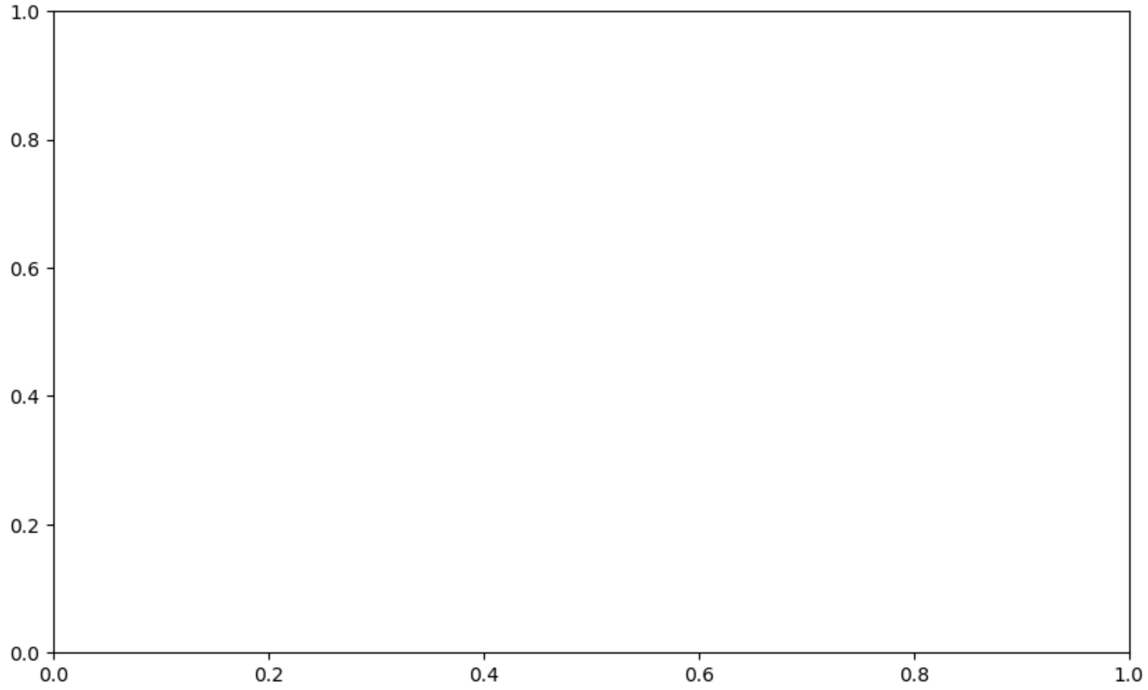


Change Figure Size

You can change the size of your figure using the argument figsize to specify a width and height for your figure: `figsize = (width, height)`

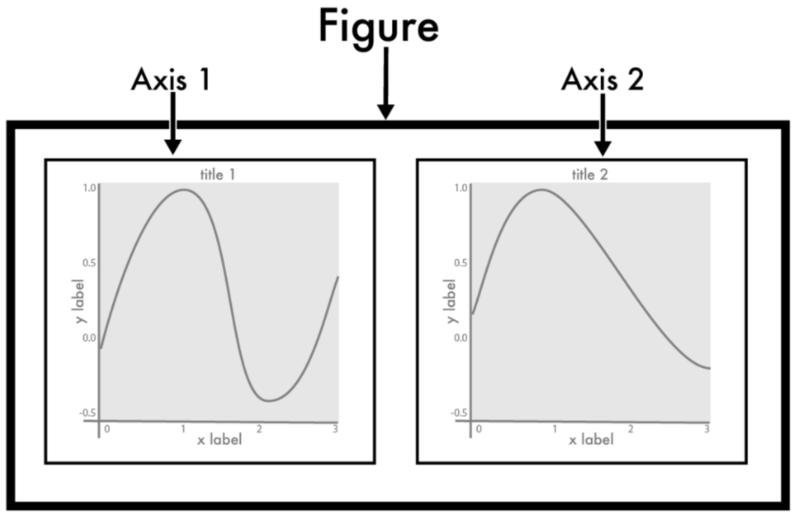


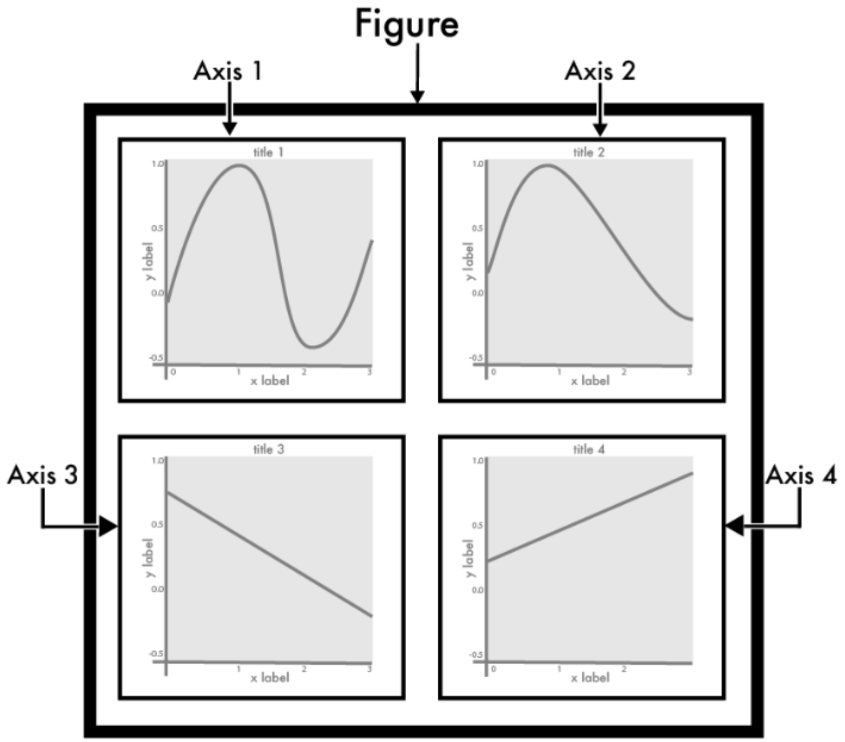
```
# Resize figure
fig, ax = plt.subplots(figsize = (10, 6))
```



Multi-plot Figures

Using matplotlib's object-oriented approach makes it easier to include more than one plot in a figure by creating additional axis objects.

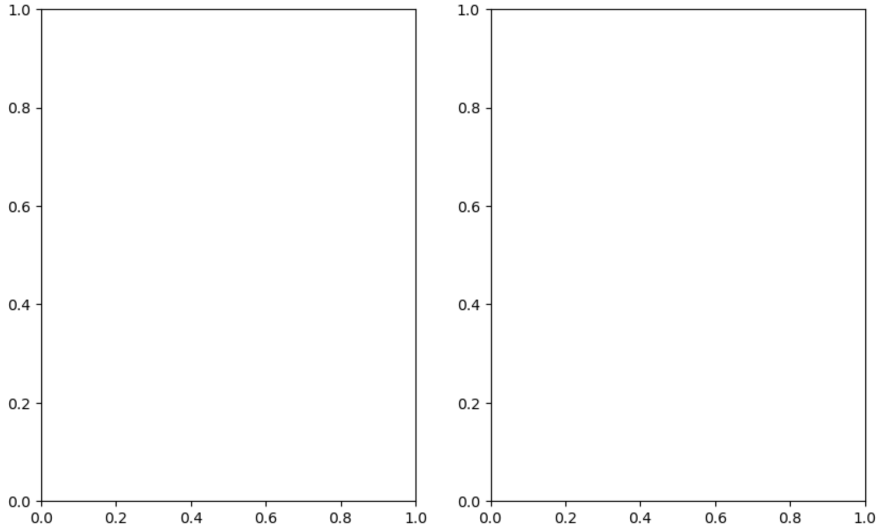




Multi-plot Figures

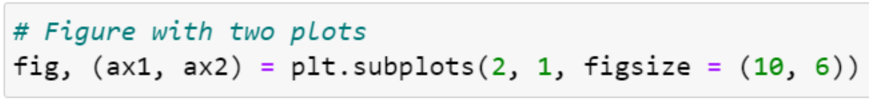
When adding more than one axis object, it is good practice to give them distinct names (such as ax1 and ax2), so you can easily work with each axis individually. You will need to provide new arguments to plt.subplots for the layout of the figure: number of rows and columns: `plt.subplots(1, 2)`

In this example, 1, 2 indicates that you want the plot layout to be 1 row across 2 columns



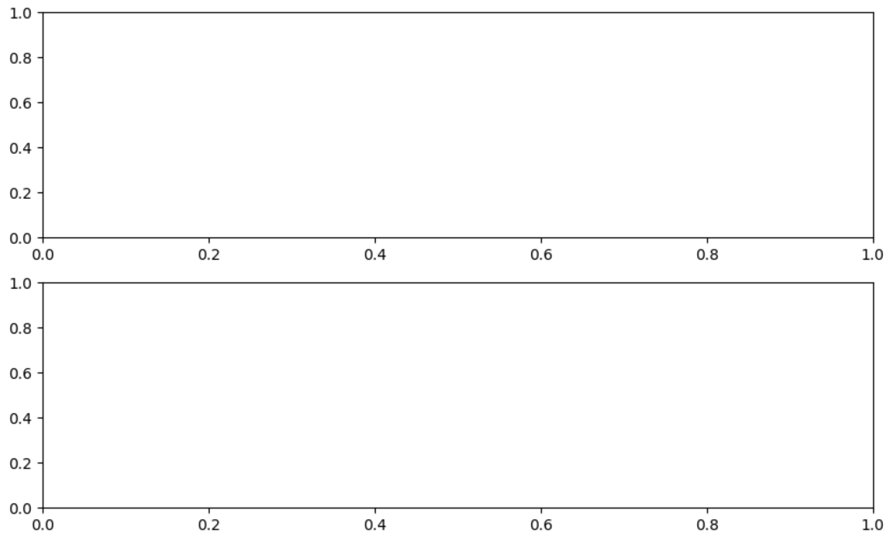
Multi-plot Figures

Conversely, 2, 1 indicates that you want the plot layout to be 2 rows across one column. Because you have defined figsize=(10, 6), the figure space remains the same size regardless of how many rows or columns you request.



```
# Figure with two plots
fig, (ax1, ax2) = plt.subplots(2, 1, figsize = (10, 6))
```

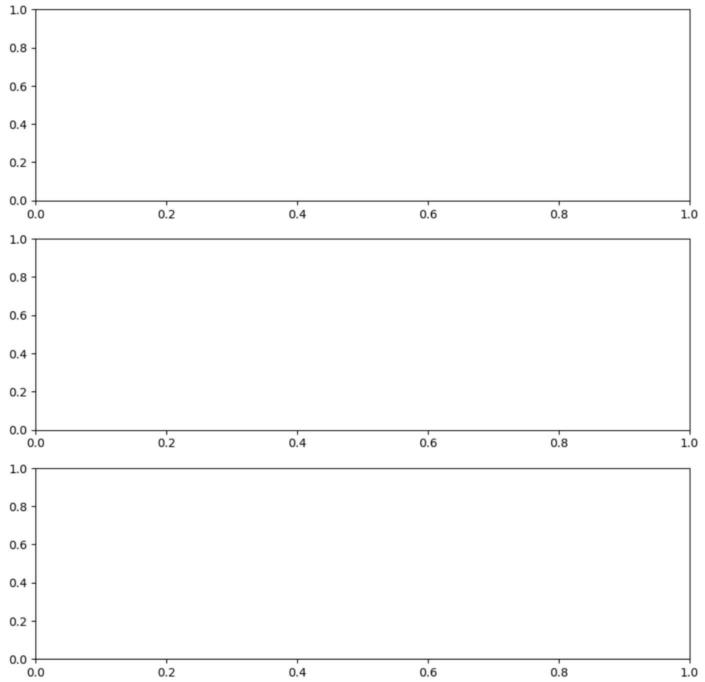
You can play around with both the number of rows and columns as well as figsize to arrive at the look that you want.



Multi-plot Figures

You can continue to add as many as axis objects as you need to create the overall layout of the desired figure and continue adjusting the figsize as needed.

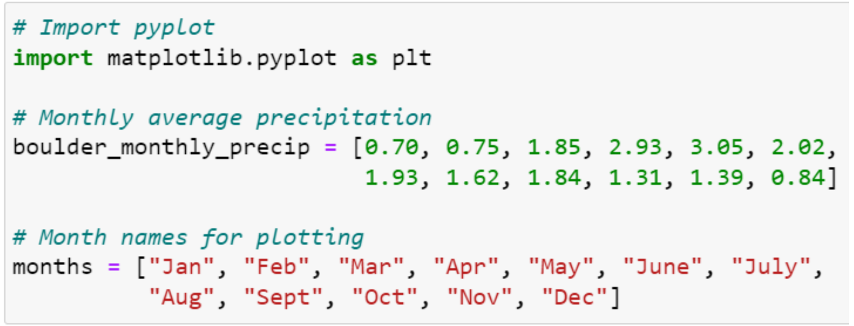
```python
# Figure with three plots
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize = (10, 10))
```



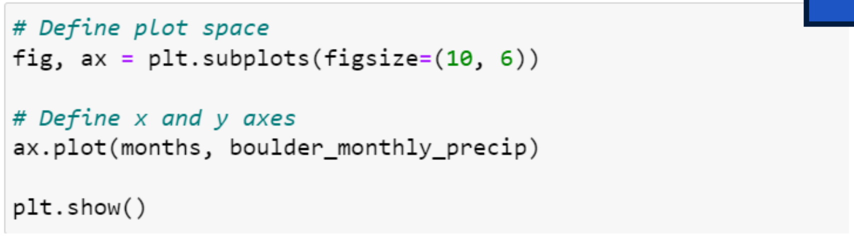
Customize Your Plots Using Matplotlib

Plot Your Data Using Matplotlib

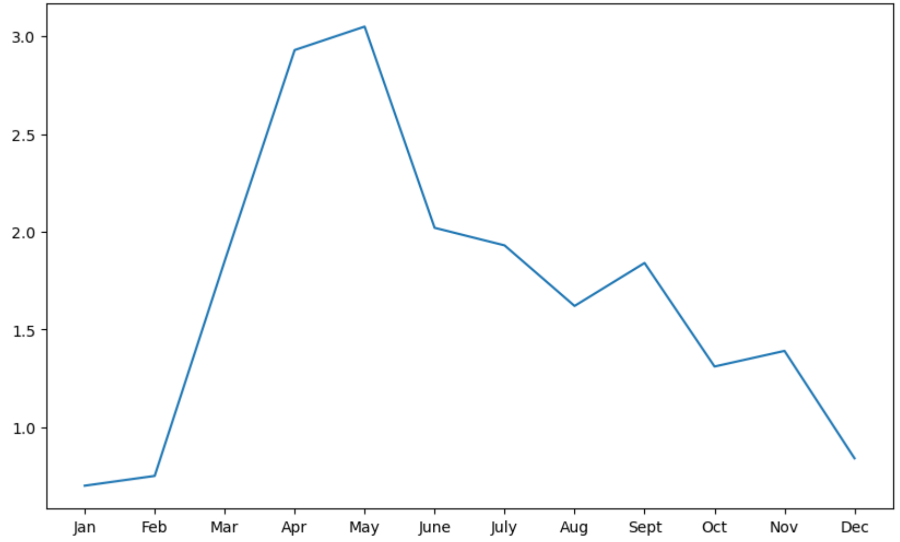
You can continue to add as many as axis objects as you need to create the overall layout of the desired figure and continue adjusting the figsize as needed.



```python
# Import pyplot
import matplotlib.pyplot as plt

# Monthly average precipitation
boulder_monthly_precip = [0.70, 0.75, 1.85, 2.93, 3.05, 2.02,
                          1.93, 1.62, 1.84, 1.31, 1.39, 0.84]

# Month names for plotting
months = ["Jan", "Feb", "Mar", "Apr", "May", "June", "July",
          "Aug", "Sept", "Oct", "Nov", "Dec"]
```

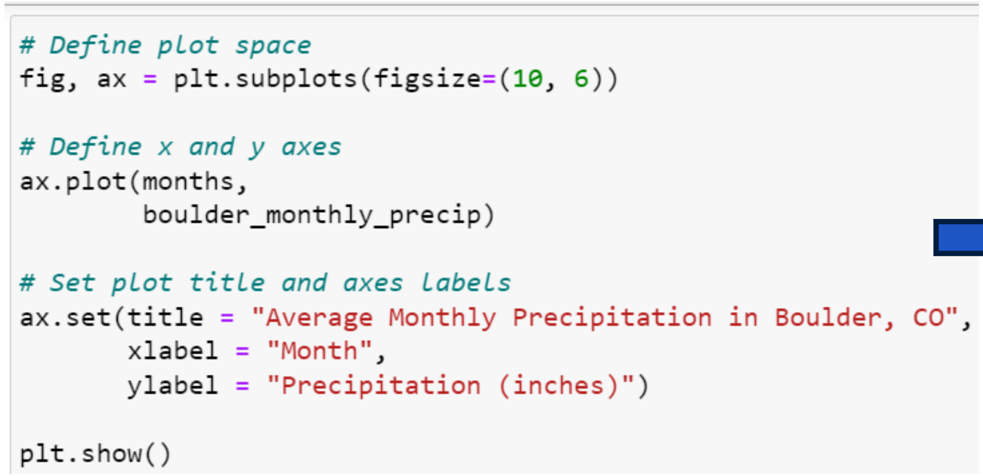


```python
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Define x and y axes
ax.plot(months, boulder_monthly_precip)

plt.show()
```

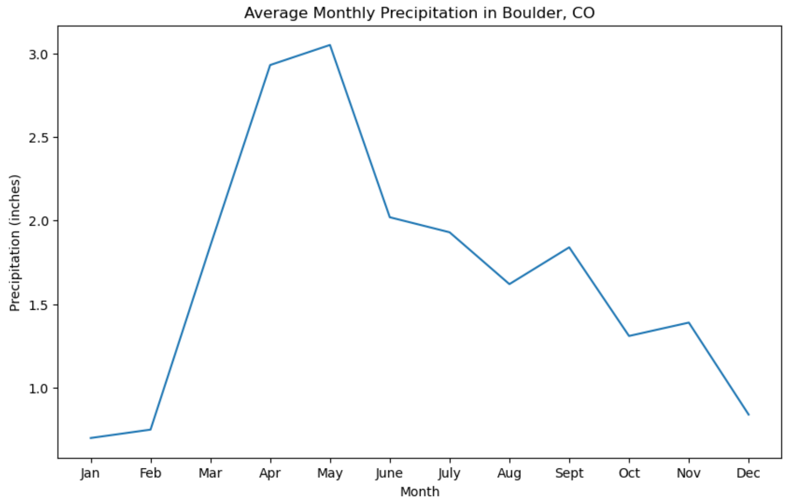


Customize Plot Title and Axes Labels

You can customize and add more information to your plot by adding a plot title and labels for the axes using the title, xlabel, ylabel arguments within the ax.set() method:

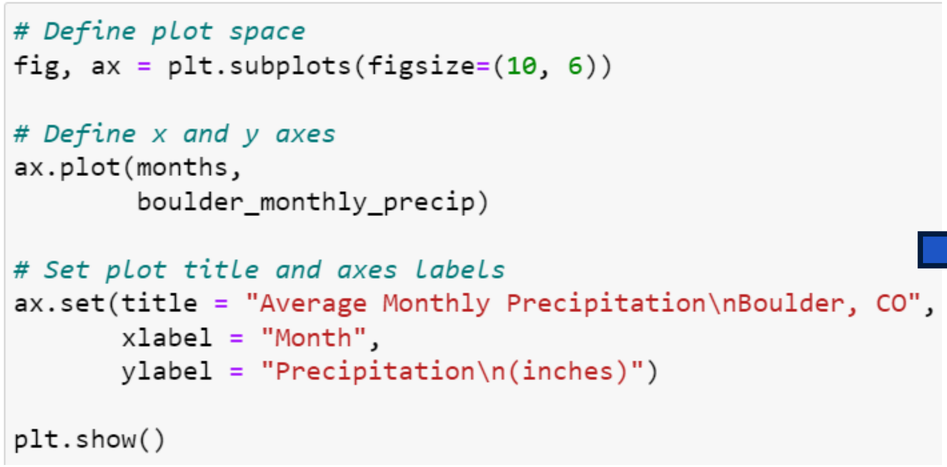


```
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Define x and y axes
ax.plot(months,
        boulder_monthly_precip)

# Set plot title and axes labels
ax.set(title = "Average Monthly Precipitation in Boulder, CO",
       xlabel = "Month",
       ylabel = "Precipitation (inches)")

plt.show()
```

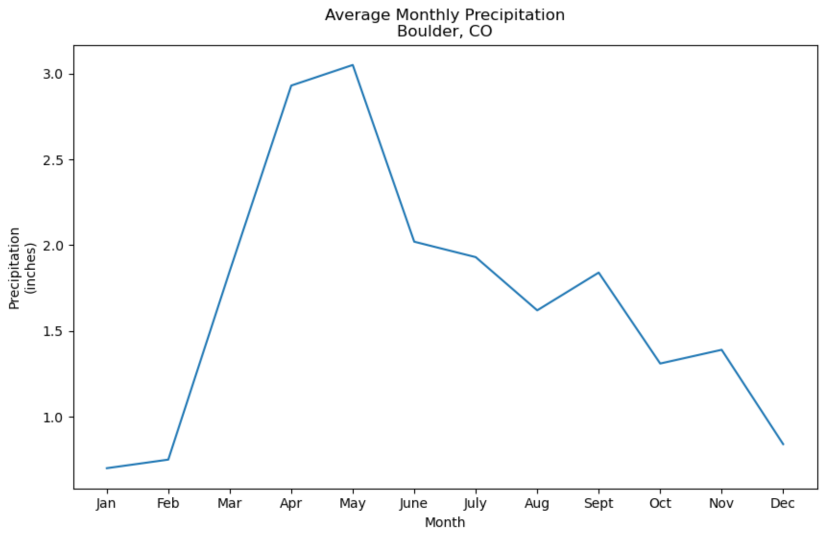


Multi-line Titles and Labels

You can also create titles and axes labels with have multiple lines of text using the new line character \n between two words to identity the start of the new line.



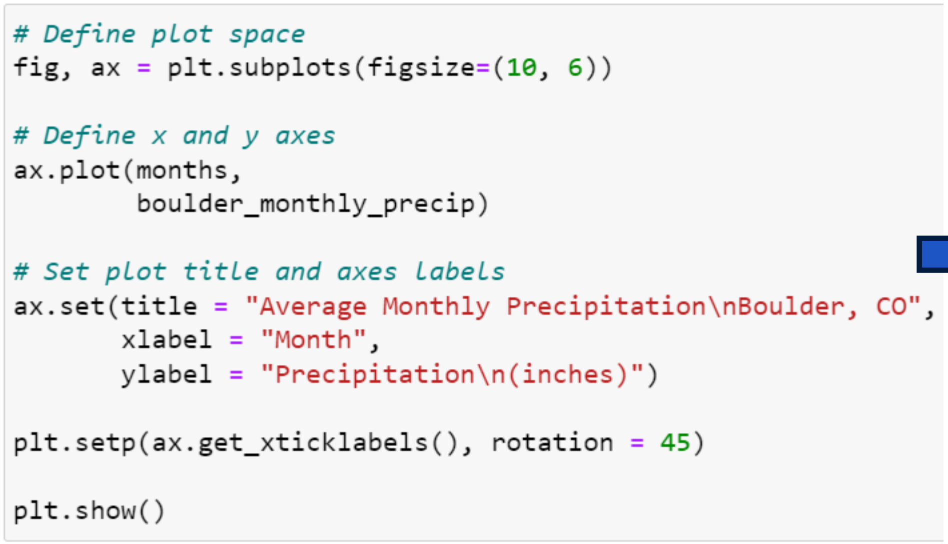
```
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Define x and y axes
ax.plot(months,
        boulder_monthly_precip)

# Set plot title and axes labels
ax.set(title = "Average Monthly Precipitation\nBoulder, CO",
       xlabel = "Month",
       ylabel = "Precipitation\n(inches)")

plt.show()
```



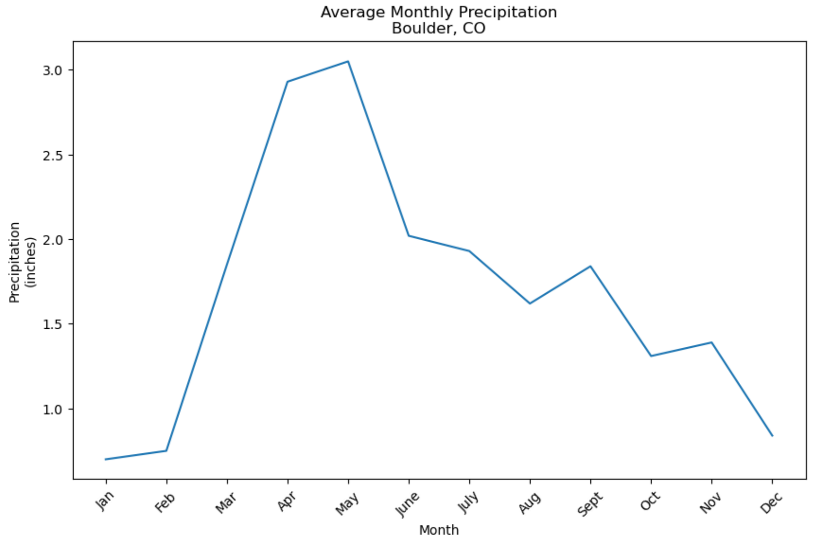
Rotate Labels

You can use plt.setp to set properties in your plot, such as customizing labels including the tick labels.

In the example below, ax.get_xticklabels() grabs the tick labels from the x axis, and then the rotation argument specifies an angle of rotation (e.g. 45), so that the tick labels along the x axis are rotated 45 degrees.



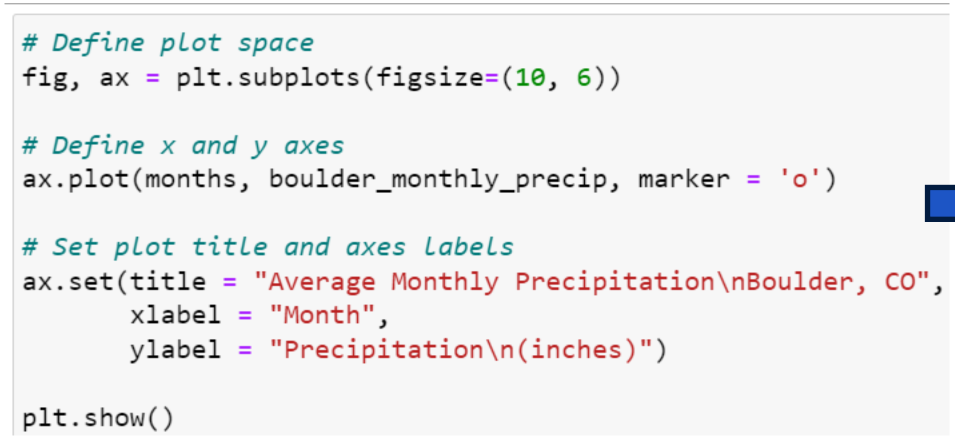
```
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Define x and y axes
ax.plot(months,
        boulder_monthly_precip)

# Set plot title and axes labels
ax.set(title = "Average Monthly Precipitation\nBoulder, CO",
       xlabel = "Month",
       ylabel = "Precipitation\n(inches)")

plt.setp(ax.get_xticklabels(), rotation = 45)

plt.show()
```



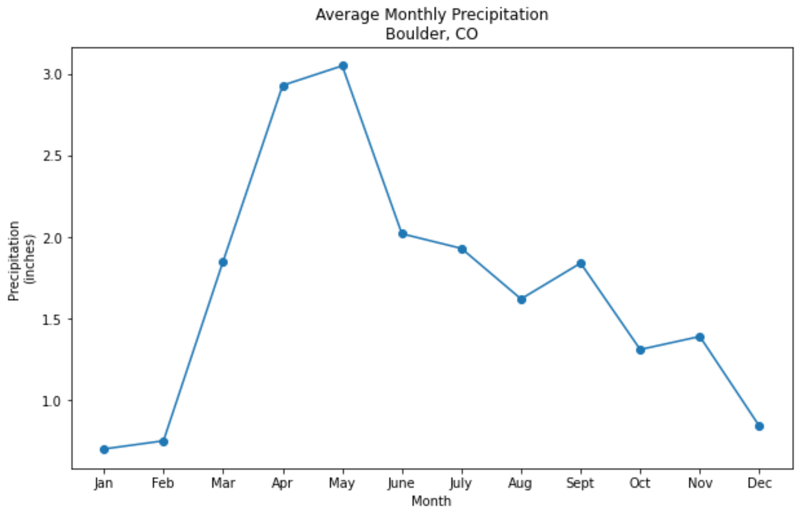
Custom Markers in Line Plots

You can change the point marker type in your line or scatter plot using the argument marker = and setting it equal to the symbol that you want to use to identify the points in the plot. For example, "," will display the point markers as a pixel or box, and “o” will display point markers as a circle.

To see a comprehensive list of markers, head to: https://matplotlib.org/stable/api/markers_api.html



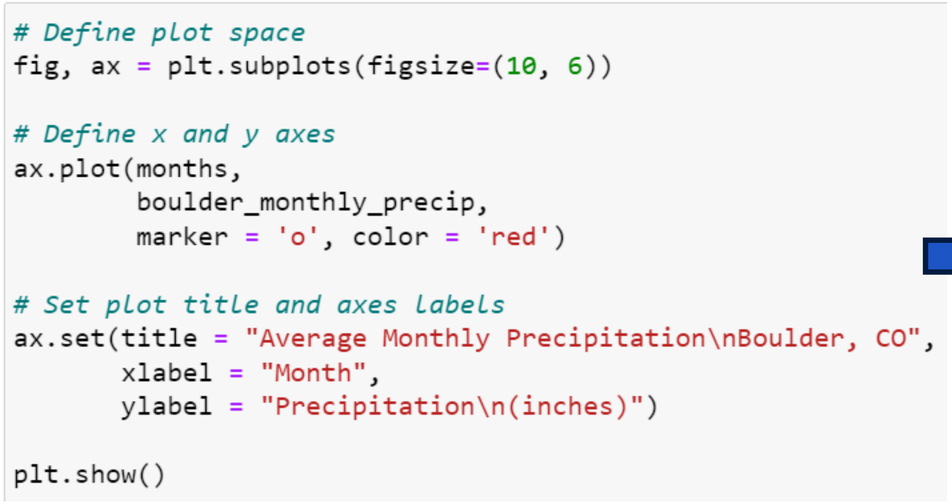
```
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Define x and y axes
ax.plot(months, boulder_monthly_precip, marker = 'o')

# Set plot title and axes labels
ax.set(title = "Average Monthly Precipitation\nBoulder, CO",
       xlabel = "Month",
       ylabel = "Precipitation\n(inches)")

plt.show()
```



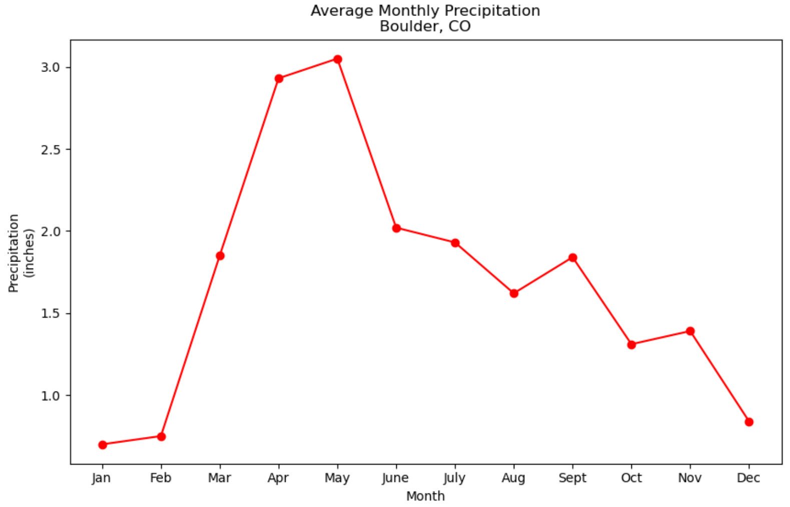
Customize Plot Colors

You can customize the color of your plot using the color argument and setting it equal to the color that you want to use for the plot. A list of some of the base color options available in matplotlib is below:

b: blue, g: green, r: red, c: cyan, m: magenta, y: yellow, k: black, w: white



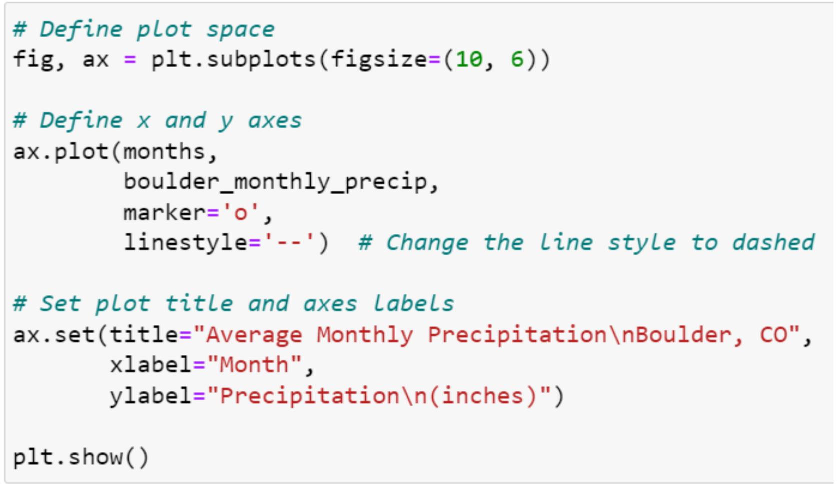
```
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Define x and y axes
ax.plot(months,
        boulder_monthly_precip,
        marker = 'o', color = 'red')

# Set plot title and axes labels
ax.set(title = "Average Monthly Precipitation\nBoulder, CO",
       xlabel = "Month",
       ylabel = "Precipitation\n(inches)")

plt.show()
```



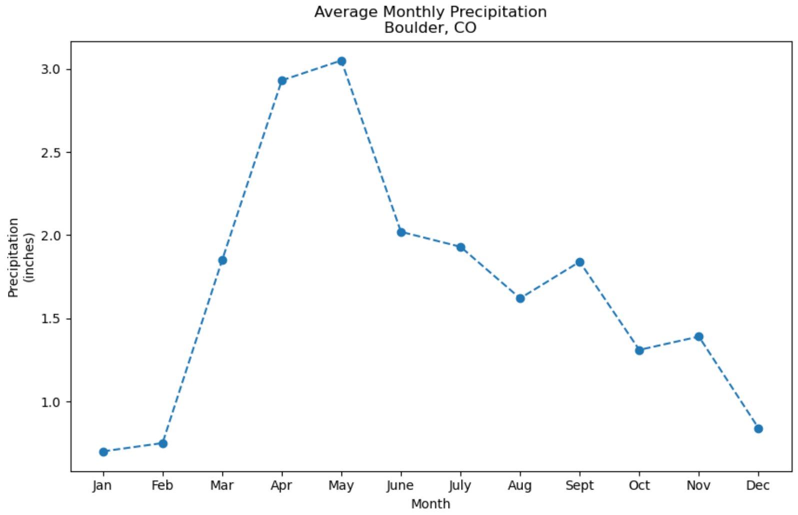
Customize Plot Line Style

To enhance the readability, you can plot your graph with different line styles. it helps to differentiate from other graphs plotted in the same figure.

To see a comprehensive list of markers, head to:https://matplotlib.org/stable/gallery/lines_bars_and_markers/linestyles.html

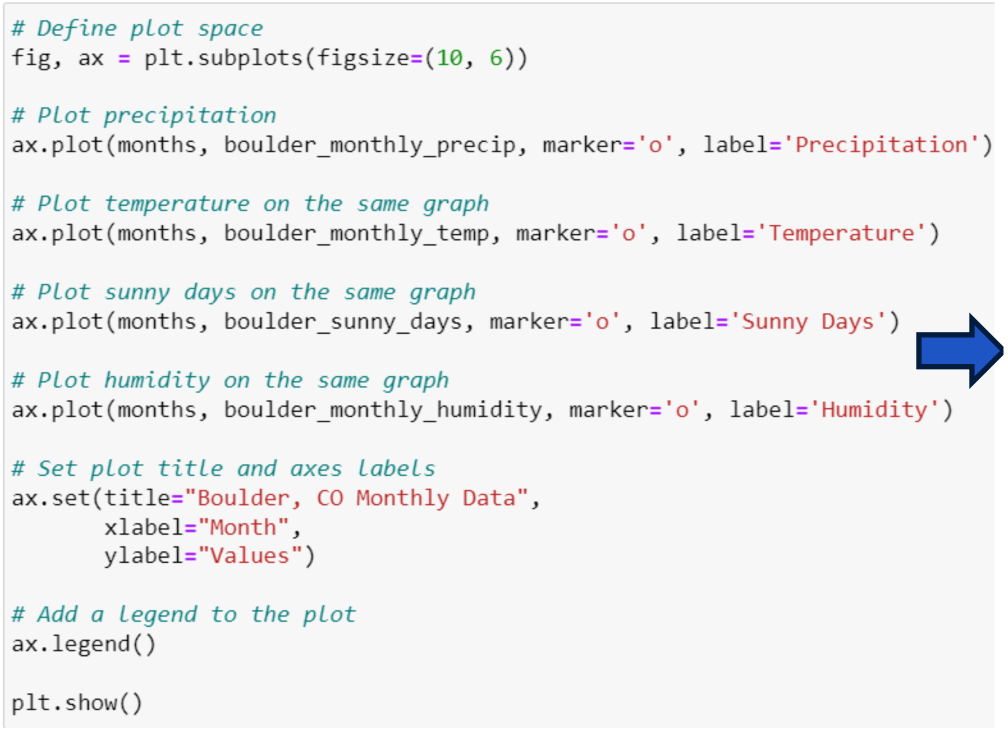


```
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Define x and y axes
ax.plot(months,
        boulder_monthly_precip,
        marker='o',
        linestyle='--')  # Change the line style to dashed

# Set plot title and axes labels
ax.set(title="Average Monthly Precipitation\nBoulder, CO",
       xlabel="Month",
       ylabel="Precipitation\n(inches)")

plt.show()
```

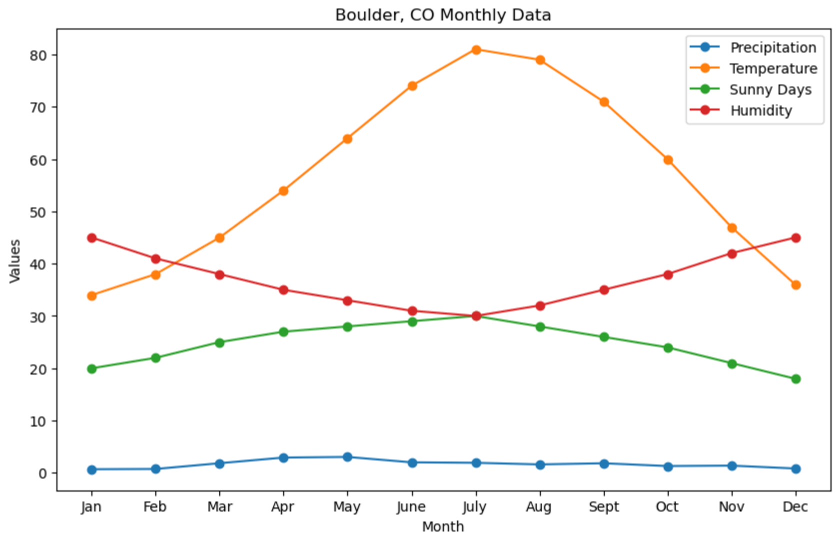


Customise Multiple Plots

As shown below, having too many plots in one figure can confuse the reader.

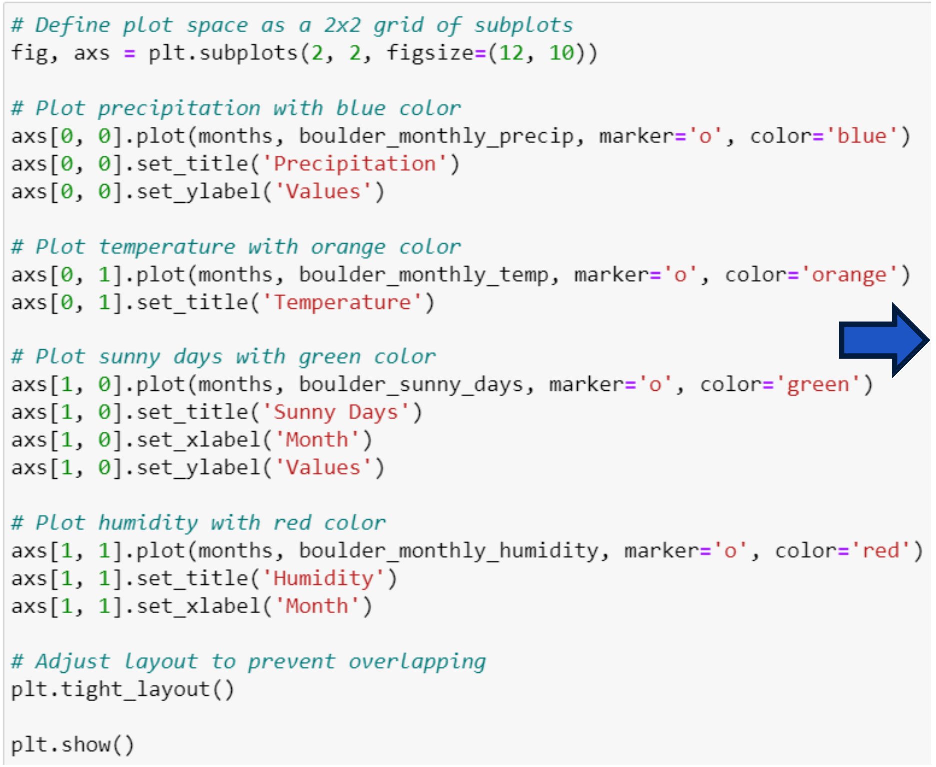


```python
# Define plot space
fig, ax = plt.subplots(figsize=(10, 6))

# Plot precipitation
ax.plot(months, boulder_monthly_precip, marker='o', label='Precipitation')

# Plot temperature on the same graph
ax.plot(months, boulder_monthly_temp, marker='o', label='Temperature')

# Plot sunny days on the same graph
ax.plot(months, boulder_sunny_days, marker='o', label='Sunny Days')

# Plot humidity on the same graph
ax.plot(months, boulder_monthly_humidity, marker='o', label='Humidity')

# Set plot title and axes labels
ax.set(title="Boulder, CO Monthly Data",
       xlabel="Month",
       ylabel="Values")

# Add a legend to the plot
ax.legend()

plt.show()
```

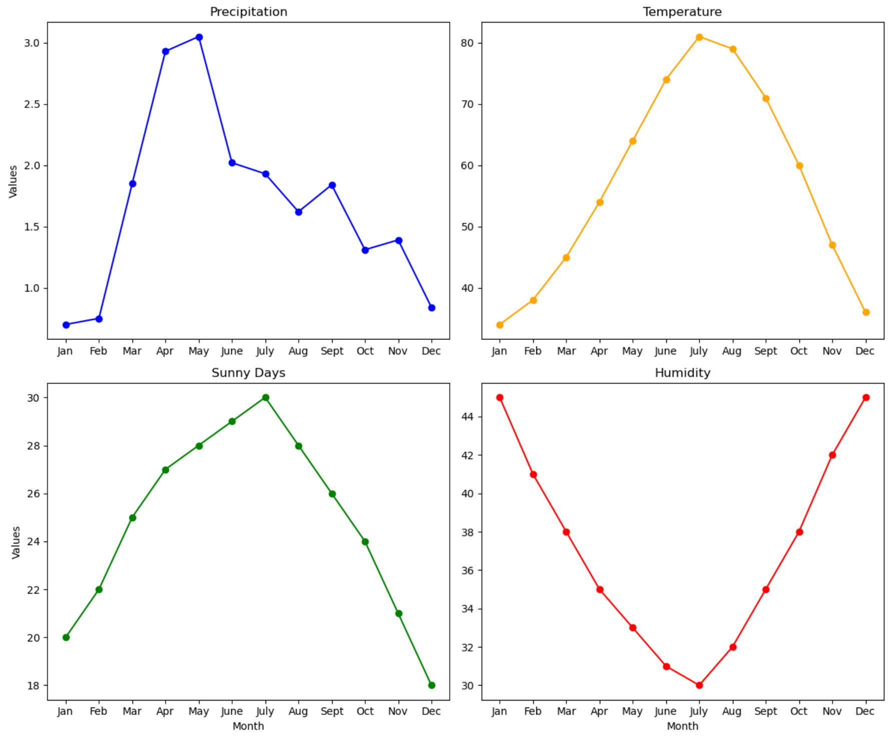


Customise Multiple Plots

Let do it in different plots..



```
# Define plot space as a 2x2 grid of subplots
fig, axs = plt.subplots(2, 2, figsize=(12, 10))

# Plot precipitation with blue color
axs[0, 0].plot(months, boulder_monthly_precip, marker='o', color='blue')
axs[0, 0].set_title('Precipitation')
axs[0, 0].set_ylabel('Values')

# Plot temperature with orange color
axs[0, 1].plot(months, boulder_monthly_temp, marker='o', color='orange')
axs[0, 1].set_title('Temperature')

# Plot sunny days with green color
axs[1, 0].plot(months, boulder_sunny_days, marker='o', color='green')
axs[1, 0].set_title('Sunny Days')
axs[1, 0].set_xlabel('Month')
axs[1, 0].set_ylabel('Values')

# Plot humidity with red color
axs[1, 1].plot(months, boulder_monthly_humidity, marker='o', color='red')
axs[1, 1].set_title('Humidity')
axs[1, 1].set_xlabel('Month')

# Adjust layout to prevent overlapping
plt.tight_layout()

plt.show()
```



Plotting Time Series Data

Plotting Time Series Data

The dataset contains monthly records of atmospheric CO_2 concentrations (in ppm) and relative temperature changes (in °C) from 1958 to recent years, providing insights into climate change trends over time.

Let setup the necessary libraries and import the Dataset

```python
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
```

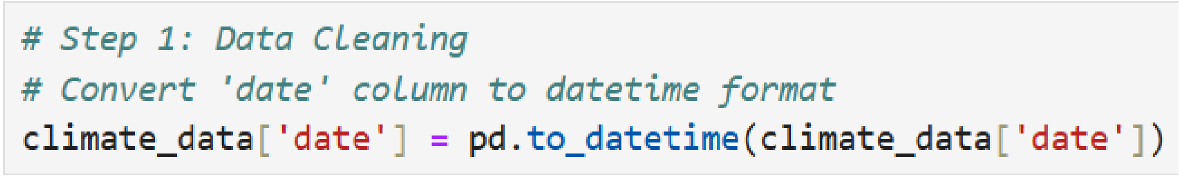
```python
# Load the dataset
file_path = 'climate_change.csv'
climate_data = pd.read_csv(file_path)
climate_data
```

	date	co2	relative_temp
0	1958-03-06	315.71	0.10
1	1958-04-06	317.45	0.01
2	1958-05-06	317.50	0.08
3	1958-06-06	NaN	-0.05
4	1958-07-06	315.86	0.06
...	...	...	...
701	2016-08-06	402.27	0.98
702	2016-09-06	401.05	0.87
703	2016-10-06	401.59	0.89
704	2016-11-06	403.55	0.93
705	2016-12-06	404.45	0.81

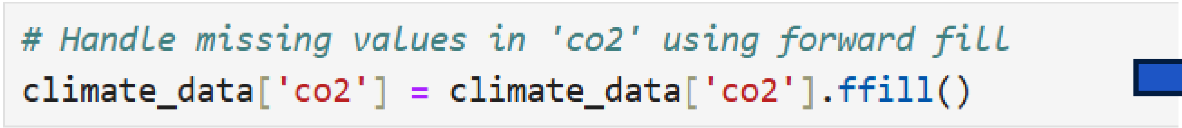
706 rows × 3 columns

Plotting Time Series Data

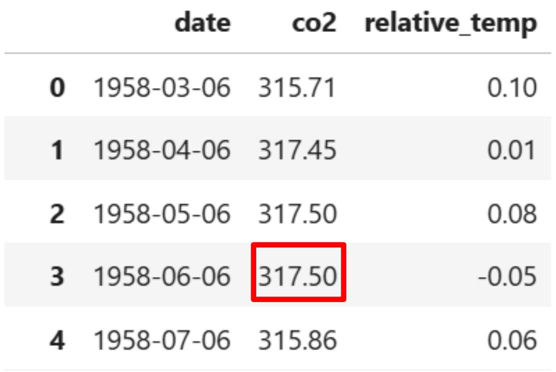
Let perform Data Cleaning using Forward Fill



```python
# Step 1: Data Cleaning
# Convert 'date' column to datetime format
climate_data['date'] = pd.to_datetime(climate_data['date'])
```



```python
# Handle missing values in 'co2' using forward fill
climate_data['co2'] = climate_data['co2'].ffill()
```



	date	co2	relative_temp
0	1958-03-06	315.71	0.10
1	1958-04-06	317.45	0.01
2	1958-05-06	317.50	0.08
3	1958-06-06	317.50	-0.05
4	1958-07-06	315.86	0.06

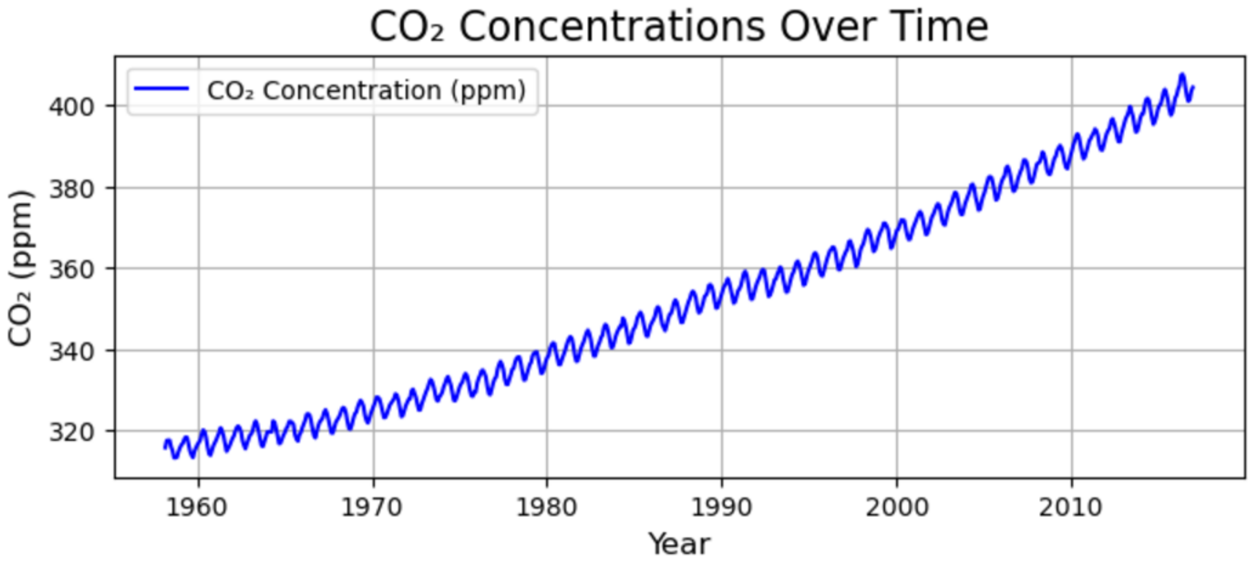
In the provided code, the missing data in the co2 column was handled using a **forward fill** method. Here's the explanation and alternatives:

What it does: *Replaces a missing value with the most recent non-missing value above it in the dataset.*

Why I chose it: *Forward fill is commonly used for time series data where consecutive values are usually correlated (e.g., monthly CO_2 concentrations tend to follow a trend).*

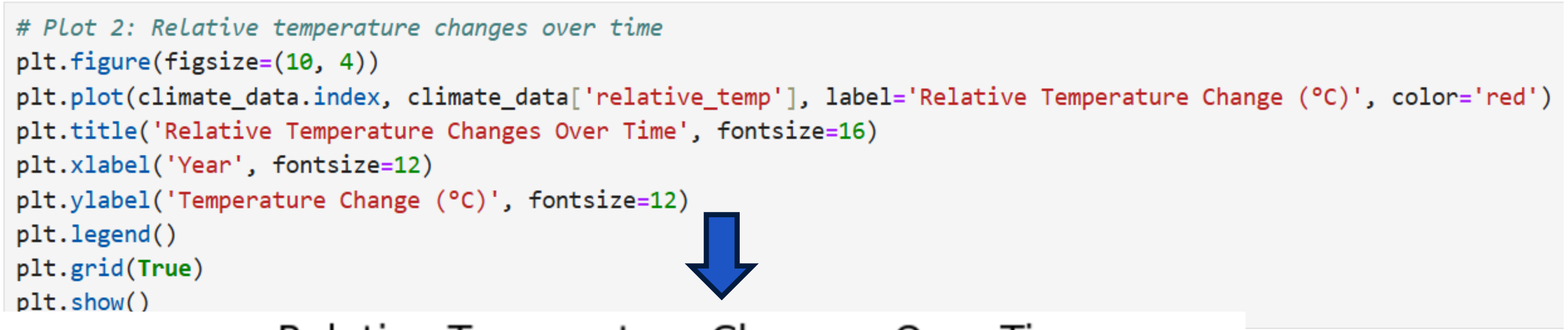
Plotting Time Series Data

```
# Plot 1: CO₂ concentrations over time
plt.figure(figsize=(10, 4))
plt.plot(climate_data.index, climate_data['co2'], label='CO₂ Concentration (ppm)', color='blue')
plt.title('CO₂ Concentrations Over Time', fontsize=16)
plt.xlabel('Year', fontsize=12)
plt.ylabel('CO₂ (ppm)', fontsize=12)
plt.legend()
plt.grid(True)
plt.show()
```

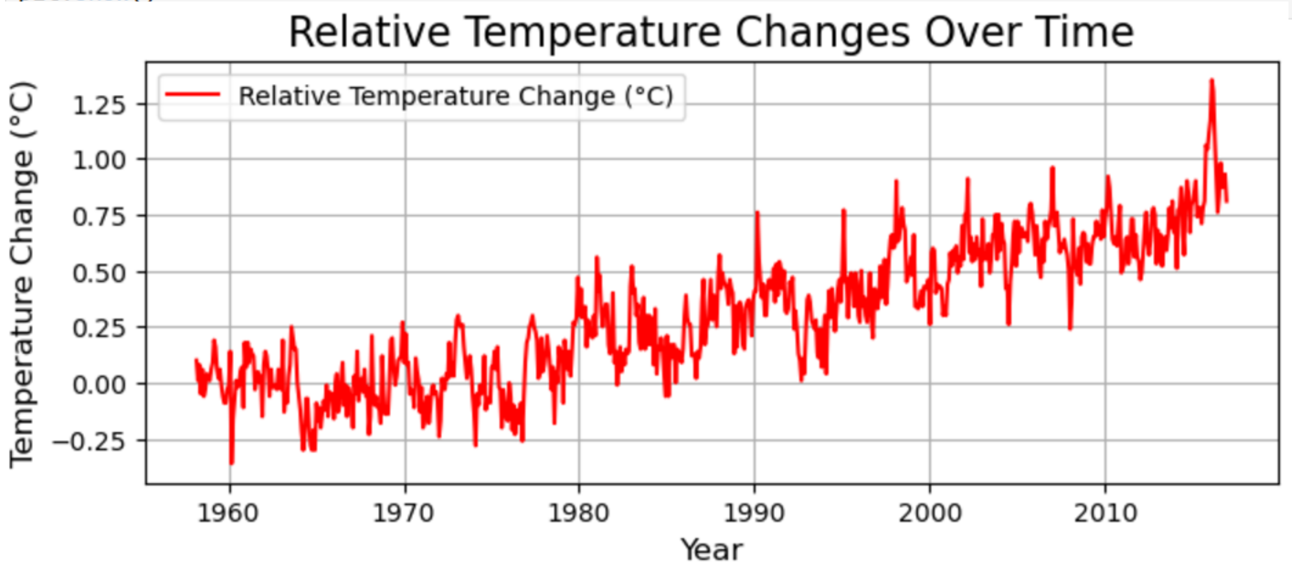


Shows a steady increase in atmospheric CO_2 levels over time, with seasonal fluctuations.

Plotting Time Series Data

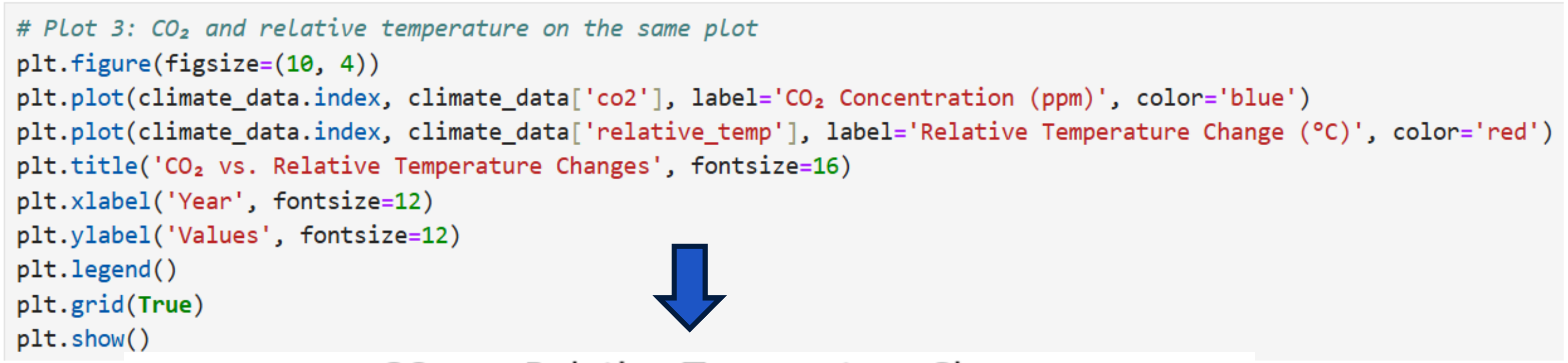


```python
# Plot 2: Relative temperature changes over time
plt.figure(figsize=(10, 4))
plt.plot(climate_data.index, climate_data['relative_temp'], label='Relative Temperature Change (°C)', color='red')
plt.title('Relative Temperature Changes Over Time', fontsize=16)
plt.xlabel('Year', fontsize=12)
plt.ylabel('Temperature Change (°C)', fontsize=12)
plt.legend()
plt.grid(True)
plt.show()
```

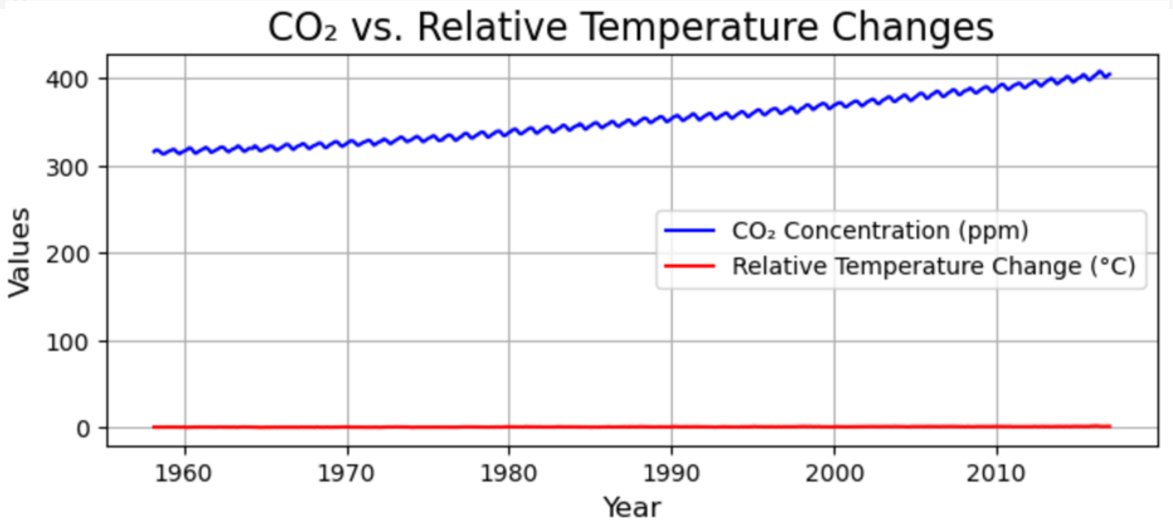


Displays global temperature changes, with an upward trend indicating warming over the years.

Plotting Time Series Data



```
# Plot 3: CO₂ and relative temperature on the same plot
plt.figure(figsize=(10, 4))
plt.plot(climate_data.index, climate_data['co2'], label='CO₂ Concentration (ppm)', color='blue')
plt.plot(climate_data.index, climate_data['relative_temp'], label='Relative Temperature Change (°C)', color='red')
plt.title('CO₂ vs. Relative Temperature Changes', fontsize=16)
plt.xlabel('Year', fontsize=12)
plt.ylabel('Values', fontsize=12)
plt.legend()
plt.grid(True)
plt.show()
```



Highlights the correlation between rising CO_2 levels and increasing temperatures, suggesting a link to climate change.

Plotting Time Series Data

Let revisit the various Data Cleaning methods other than **backward fill**

```python
# Handle missing values in 'co2' using backward fill
climate_data['co2'] = climate_data['co2'].bfill()
climate_data
```

	date	co2	relative_temp
0	1958-03-06	315.71	0.10
1	1958-04-06	317.45	0.01
2	1958-05-06	317.50	0.08
3	1958-06-06	315.86	-0.05
4	1958-07-06	315.86	0.06

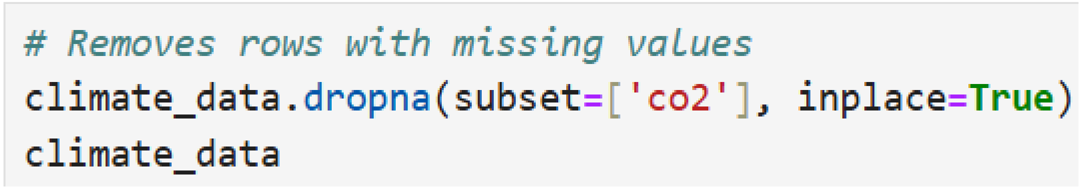
Using Interpolation

```python
# Fills missing values by interpolating between known data points
climate_data['co2'] = climate_data['co2'].interpolate()
climate_data
```

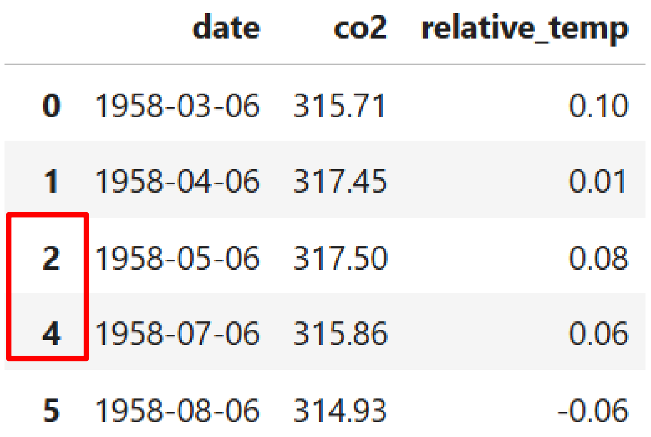
	date	co2	relative_temp
0	1958-03-06	315.71	0.10
1	1958-04-06	317.45	0.01
2	1958-05-06	317.50	0.08
3	1958-06-06	316.68	-0.05
4	1958-07-06	315.86	0.06

Plotting Time Series Data

Removing Rows with missing data

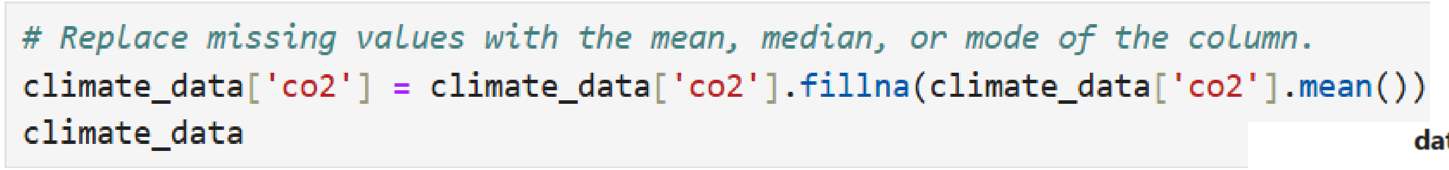


```python
# Removes rows with missing values
climate_data.dropna(subset=['co2'], inplace=True)
climate_data
```

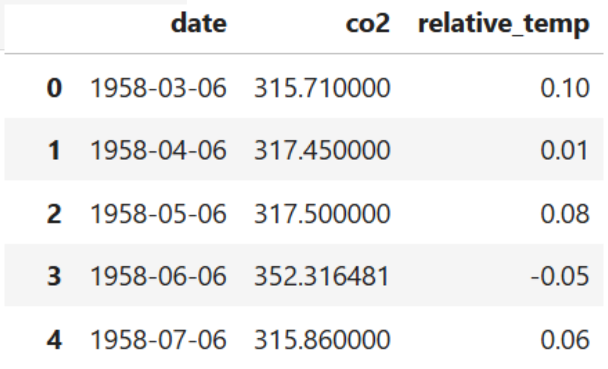


	date	co2	relative_temp
0	1958-03-06	315.71	0.10
1	1958-04-06	317.45	0.01
2	1958-05-06	317.50	0.08
4	1958-07-06	315.86	0.06
5	1958-08-06	314.93	-0.06

Replace missing values with the **mean, median, or mode** of the column.



```python
# Replace missing values with the mean, median, or mode of the column.
climate_data['co2'] = climate_data['co2'].fillna(climate_data['co2'].mean())
climate_data
```



	date	co2	relative_temp
0	1958-03-06	315.710000	0.10
1	1958-04-06	317.450000	0.01
2	1958-05-06	317.500000	0.08
3	1958-06-06	352.316481	-0.05
4	1958-07-06	315.860000	0.06

Plotting Time Series Data

Summary of the Data Cleaning Methods

Method	Description	Example Code
Forward Fill (ffill)	Fills missing values with the last available non-missing value.	climate_data['co2'] = climate_data['co2'].ffill()
Backward Fill (bfill)	Fills missing values with the next available non-missing value.	climate_data['co2'] = climate_data['co2'].bfill()
Interpolation	Estimates missing values by interpolating between known values (default: linear).	climate_data['co2'] = climate_data['co2'].interpolate()
Drop Missing Values	Removes rows containing missing values entirely.	climate_data.dropna(subset=['co2'], inplace=True)
Impute with Mean	Replaces missing values with the mean of the column.	climate_data['co2'] = climate_data['co2'].fillna(climate_data['co2'].mean())
Impute with Median	Replaces missing values with the median of the column.	climate_data['co2'] = climate_data['co2'].fillna(climate_data['co2'].median())
Impute with Mode	Replaces missing values with the mode (most frequent value) of the column.	climate_data['co2'] = climate_data['co2'].fillna(climate_data['co2'].mode()[0])

Thank You!

NANYANG

THE INNOVATIVE POLYTECHNIC

www.nyp.edu.sg