Python level 2

Intermediate Course
Class 3

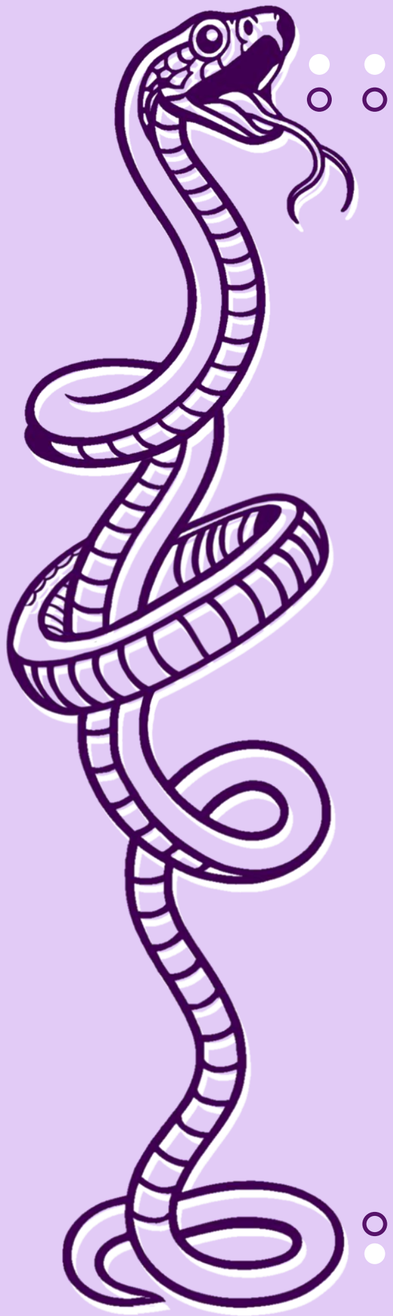
Alexandra Oliveira

September 30

AGENDA

Part III – Data visualization

1. **Introduction**
   a. The importance of visualization
   b. Main visualization libraries
2. **Matplotlib**
3. **Seaborn**
4. **Matplotlib vs Seaborn**



Introduction

The importance of visualization

Why Visualize Data?

- Turns complex information into intuitive graphics
- Makes it easier to spot patterns, trends, and key changes
- Helps uncover insights hidden in raw numbers
- Bridges the gap between data and decision-making
- Communicates information clearly and efficiently

The main python visualization libraries

```
pip install matplotlib seaborn
```

Matplotlib

A fundamental library for creating 2D visualizations in Python, offering great flexibility and control over every aspect of a chart. It allows you to generate anything from simple plots to complex visualizations, adjusting details such as colors, labels, and layouts.

Seaborn

A library built on top of Matplotlib, focused on more appealing and user-friendly visualizations. It comes with preconfigured styles and functions that simplify the creation of exploratory plots, making data analysis more intuitive and visually consistent.

Matplotlib

Importing matplotlib

Whenever you need to plot your script, just import the library. Most of the Matplotlib utilities lies under the **pyplot submodule** and are usually imported under the **plt** alias.

Motivation.txt

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

The plot( ) function

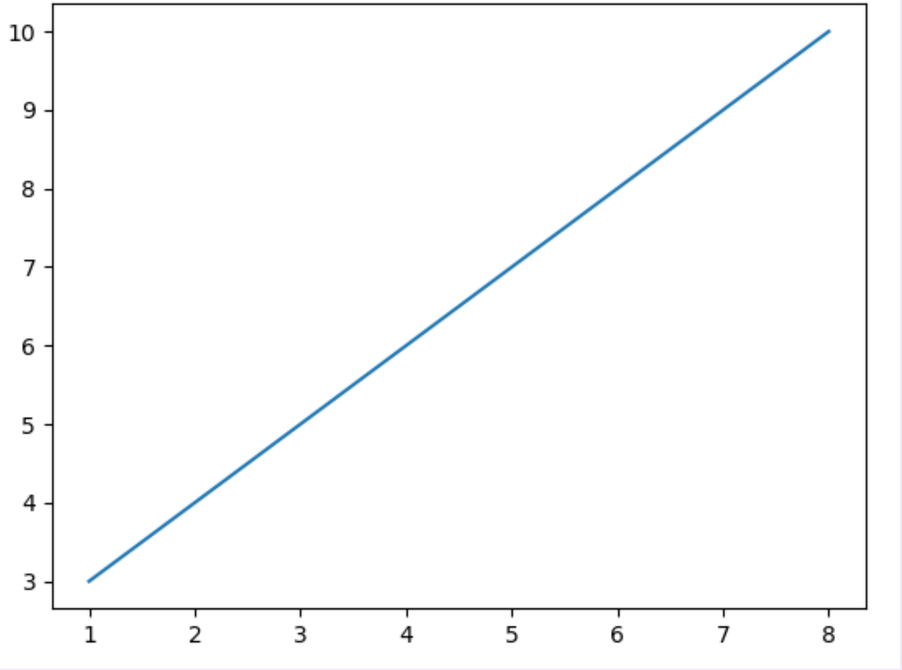
- The **plot()** function draws points (markers) in a diagram.
- By default, the plot() function draws a line from point to point.
- The function takes parameters for specifying points in the diagram.
- **Parameter 1** is an array containing the points on the x-axis.
- **Parameter 2** is an array containing the points on the y-axis.

The plot( ) function

Example: If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

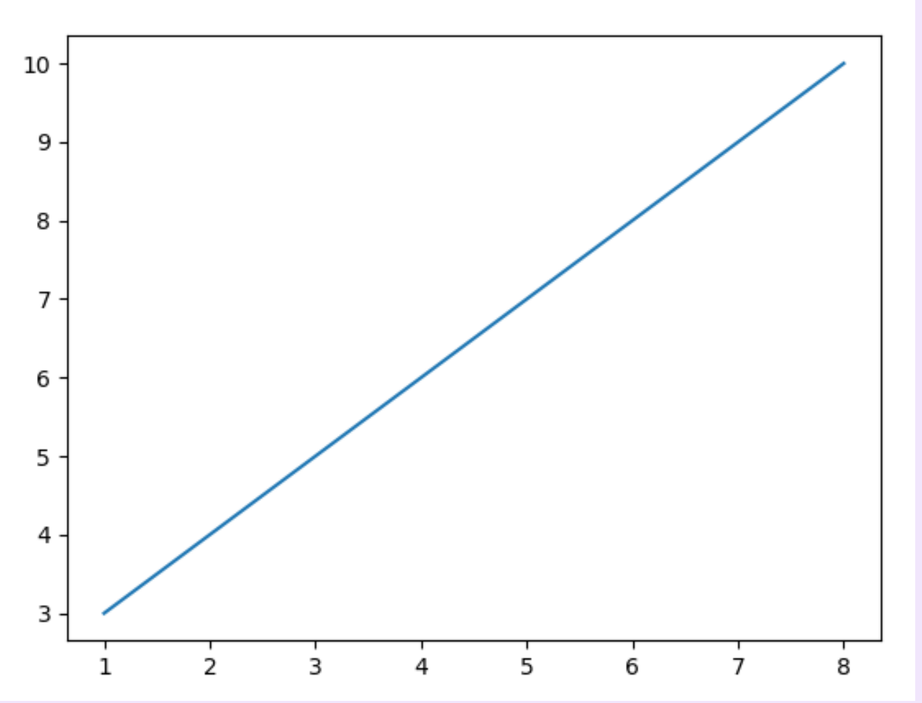


The plot( ) function

If we do not specify the points on the x-axis, they will get the default values - 0, 1, 2, 3... - depending on the length of the y-points. If we take the same example as above, and leave out the x-points, the diagram will be the same.

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 10])

plt.plot(ypoints)
plt.show()
```

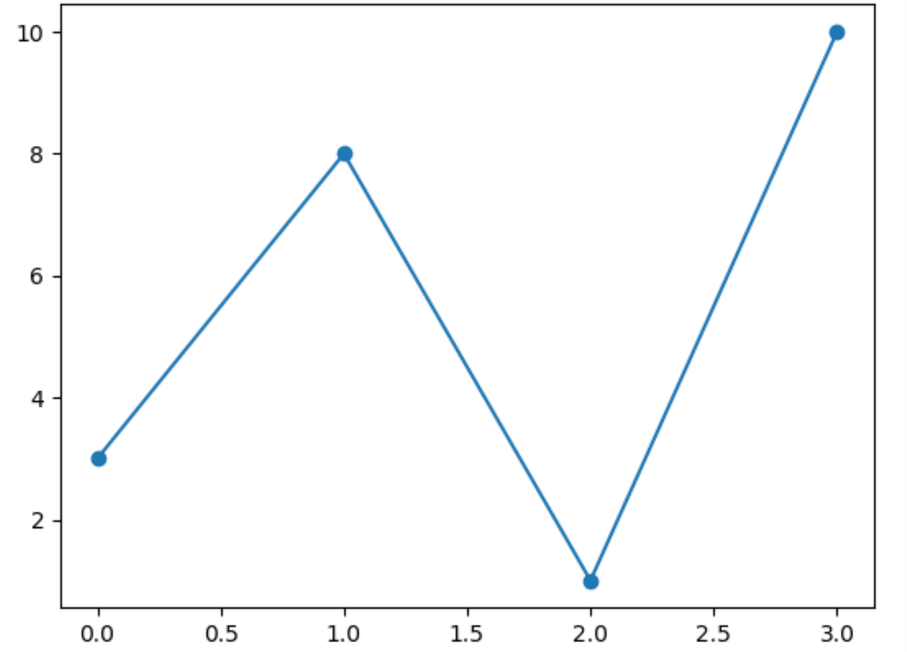


Markers

You can use the keyword argument **marker** to emphasise each point with a specified marker

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker='o')
plt.show()
```

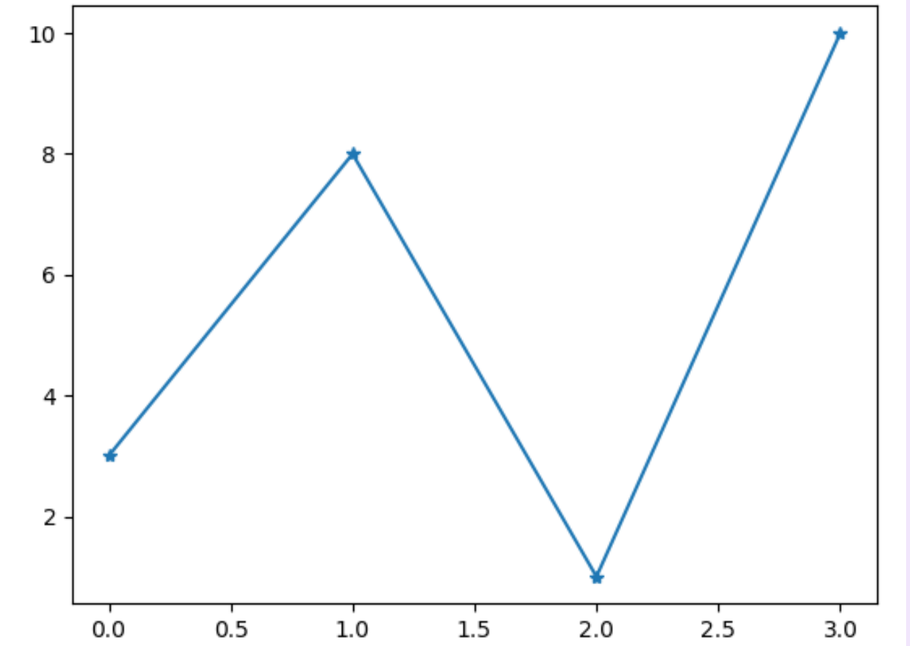


Markers

You can use the keyword argument **marker** to emphasise each point with a specified marker

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker='*')
plt.show()
```



Markers

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'o'	Circle
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon

Markers

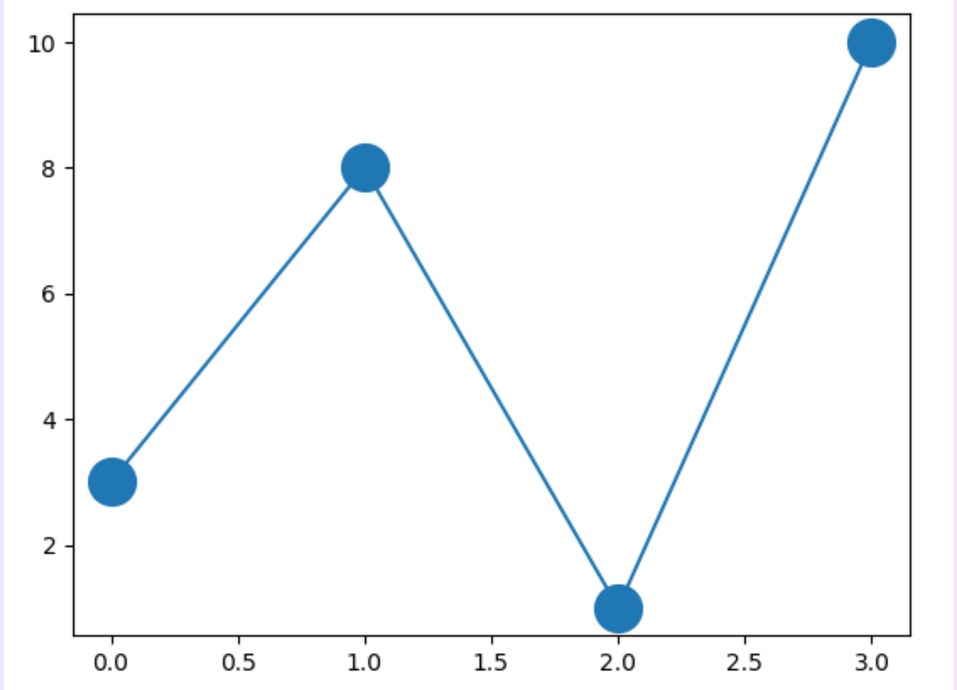
Marker	Description
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

Markers

You can use the keyword argument **markersize** or the shorter version, **ms** to set the size of the markers

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker='o', ms=20)
plt.show()
```

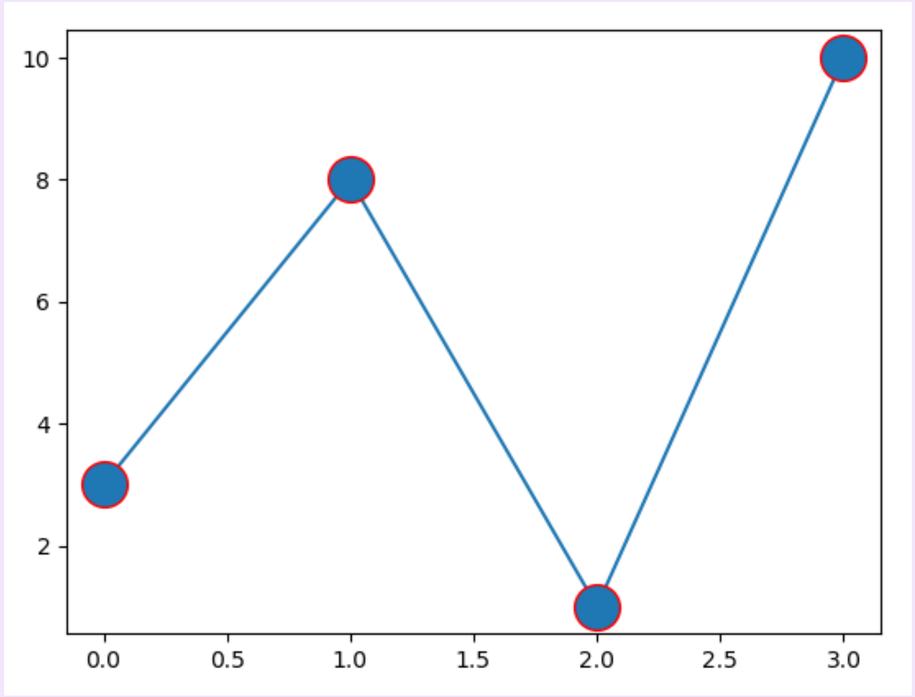


Markers

You can use the keyword argument **markeredgecolor** or the shorter version, **mec,** to set the color of the edge of the markers

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker='o', ms=20,
mec='red')
plt.show()
```

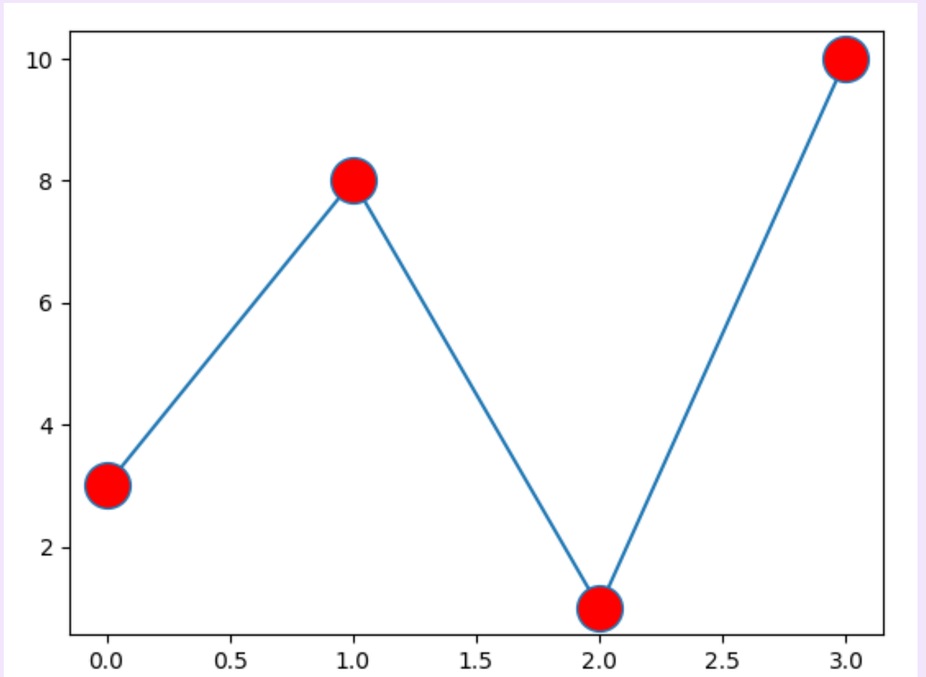


Markers

You can use the keyword argument **markerfacecolor** or the shorter version, **mfc,** to set the color inside the markers

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker='o', ms=20,
mfc='r')
plt.show()
```

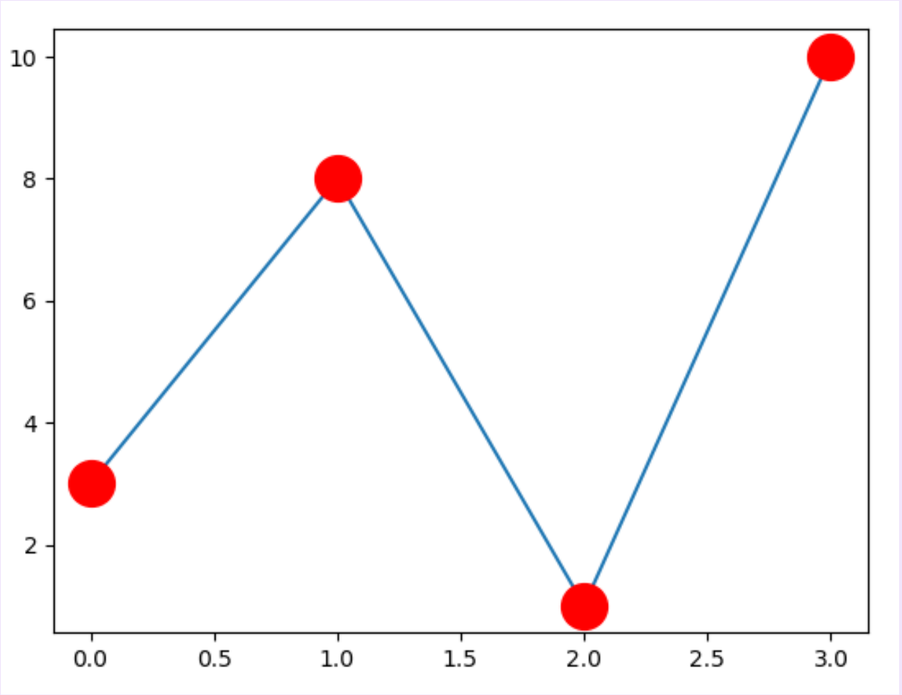


Markers

You can use the both **mfc** and **mec** arguments to colour the entire marker

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker='o', ms=20,
mfc='r', mec='r')
plt.show()
```



Markers Colors

Color syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

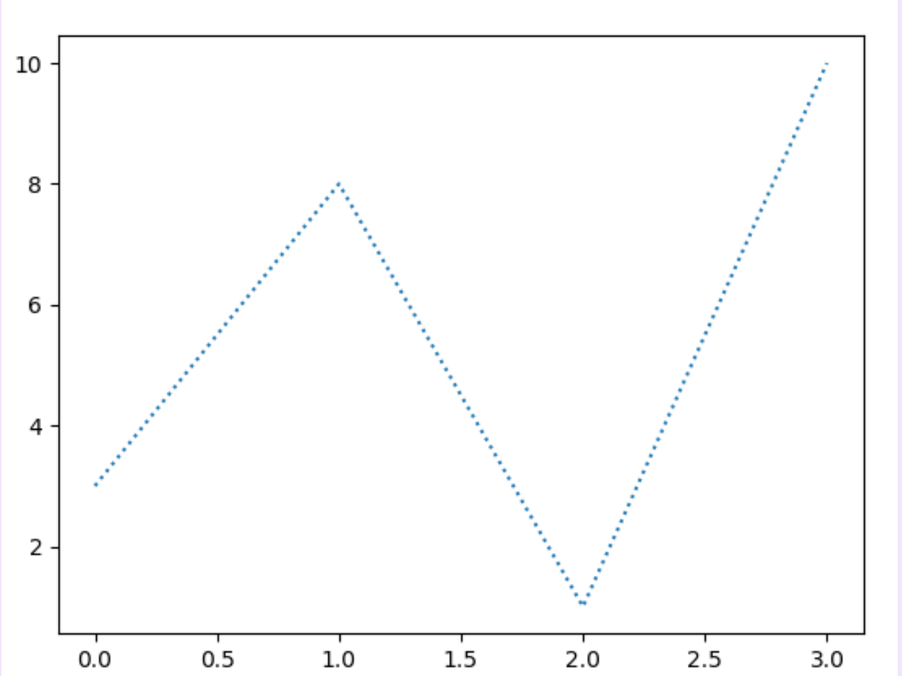
More colours: https://matplotlib.org/stable/gallery/color/named_colors.html

Line

You can use the keyword argument **linestyle**, or shorter **ls**, to change the style of the plotted line

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle='dotted')
plt.show()
```



Markers Colors

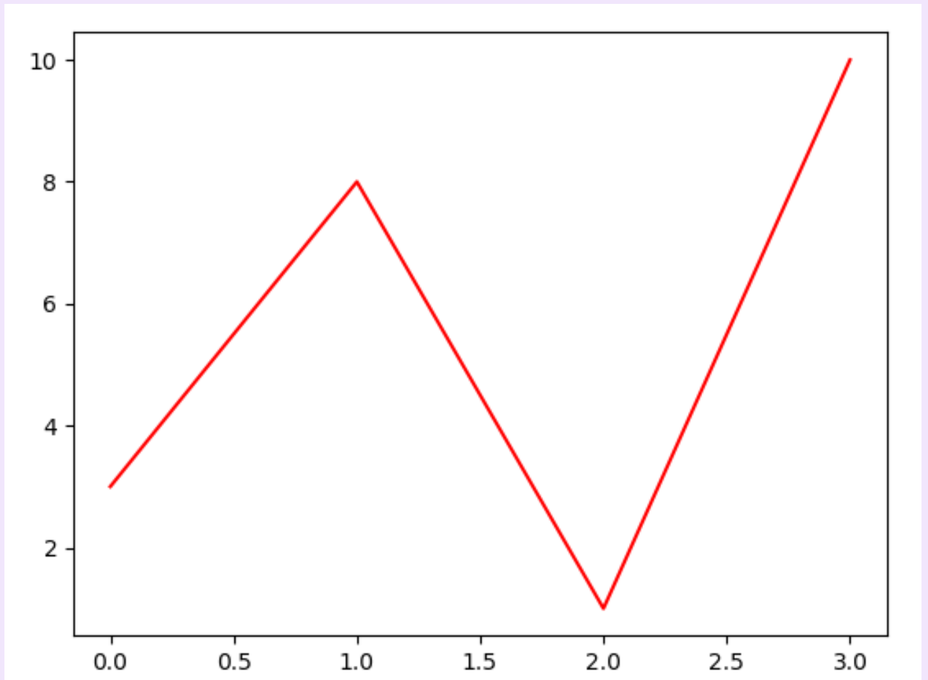
Line syntax	Description
'solid' (default)	'-'
'dotted'	':'
'dashed'	'--'
'dashdot'	'-.'

Line

You can use the keyword argument **color** or the shorter **c** to set the colour of the line

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, color='#FF0000')
plt.show()
```

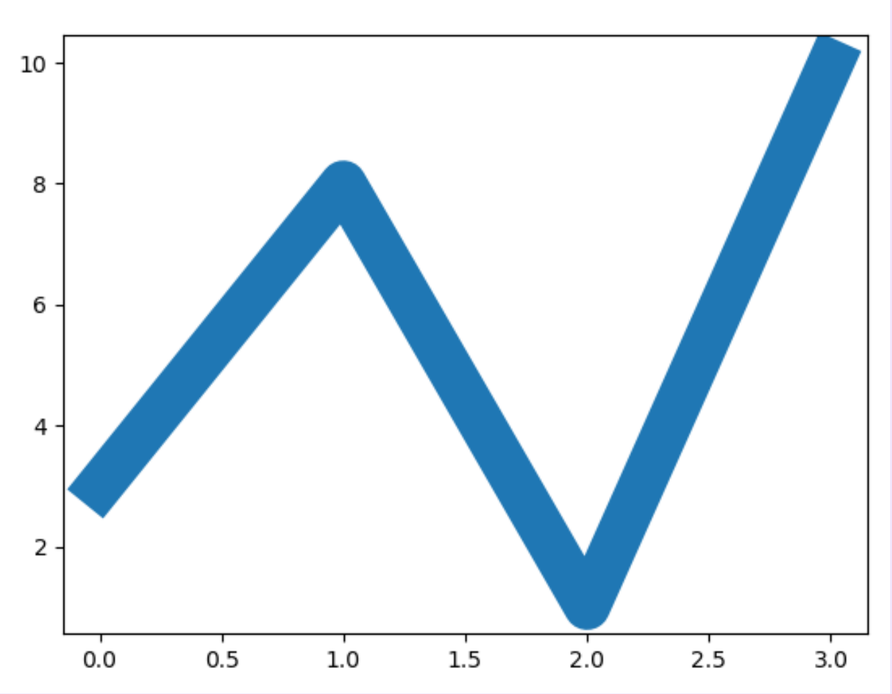


Line

You can use the keyword argument **linewidth** or the shorter **lw** to change the width of the line. The value is a floating number, in points

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth=20.5)
plt.show()
```

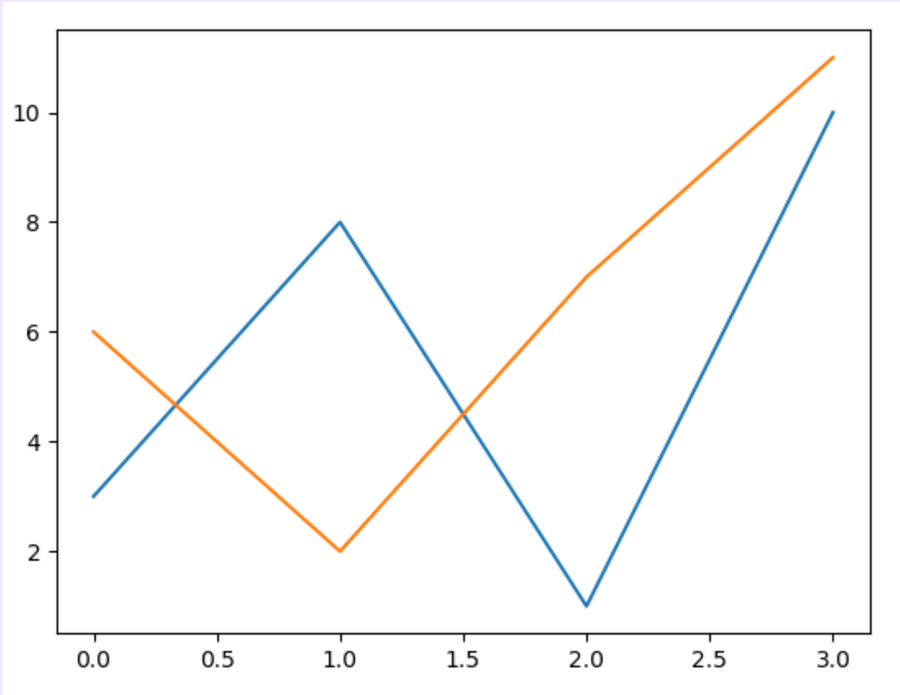


Line

You can plot as many lines as you like by simply **adding more plt.plot() functions**. It will automatically attribute different colours

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)
plt.show()
```

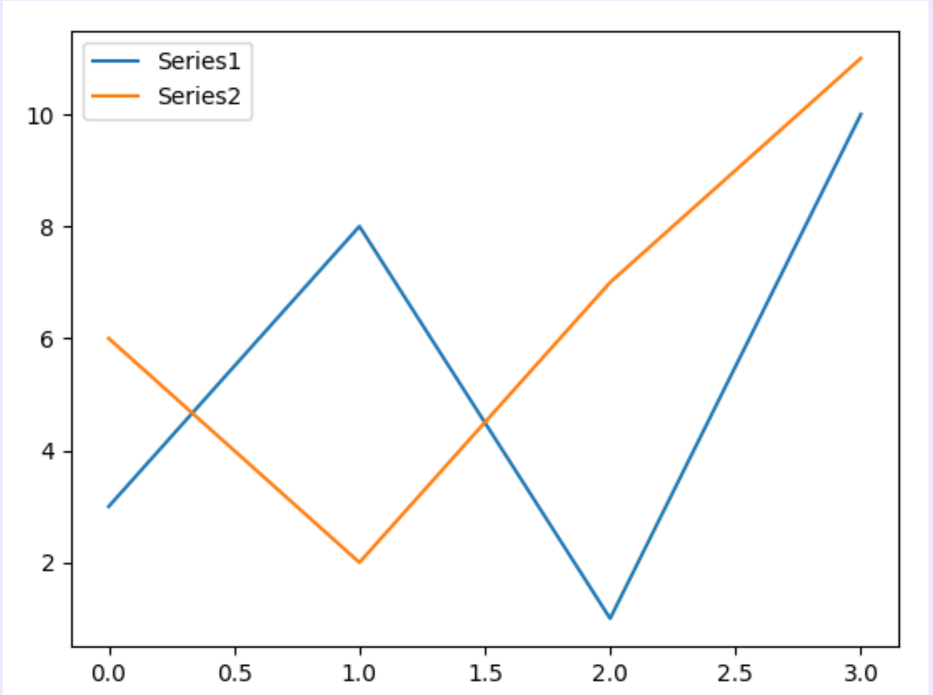


Legend

Having multiple lines on a plot, demands a **legend** to distinguish the different series

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1, label='Series 1')
plt.plot(y2, label ='Series 2')
Plt.legend()
plt.show()
```

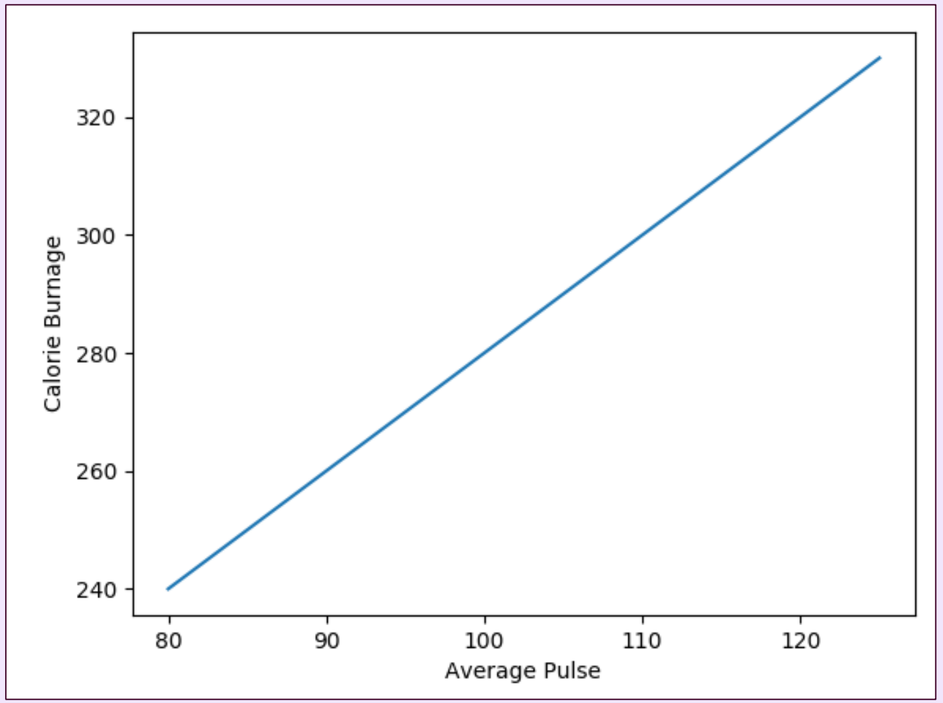


Labels

With Pyplot, you can use the **xlabel()** and **ylabel()** functions to set a label for the x- and y-axis

Motivation.txt

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100,
105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280,
290, 300, 310, 320, 330])

plt.plot(x, y)
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.show()
```

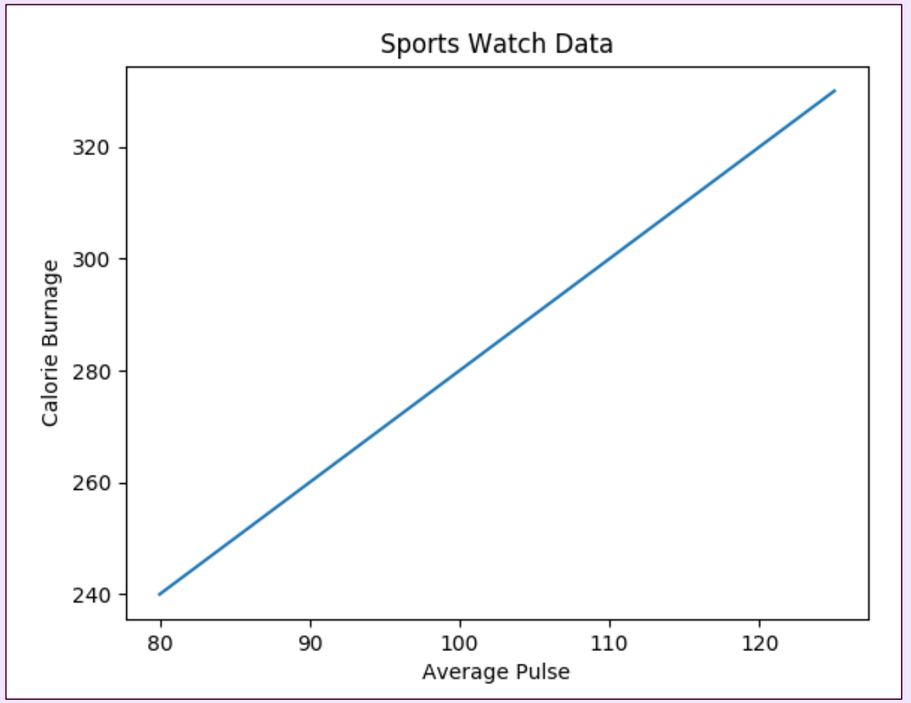


Title

With Pyplot, you can use the **title()** function to set a title for the plot

Motivation.txt

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105,
110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280,
290, 300, 310, 320, 330])

plt.plot(x, y)
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.title (“Sports Watch Data")
plt.show()
```

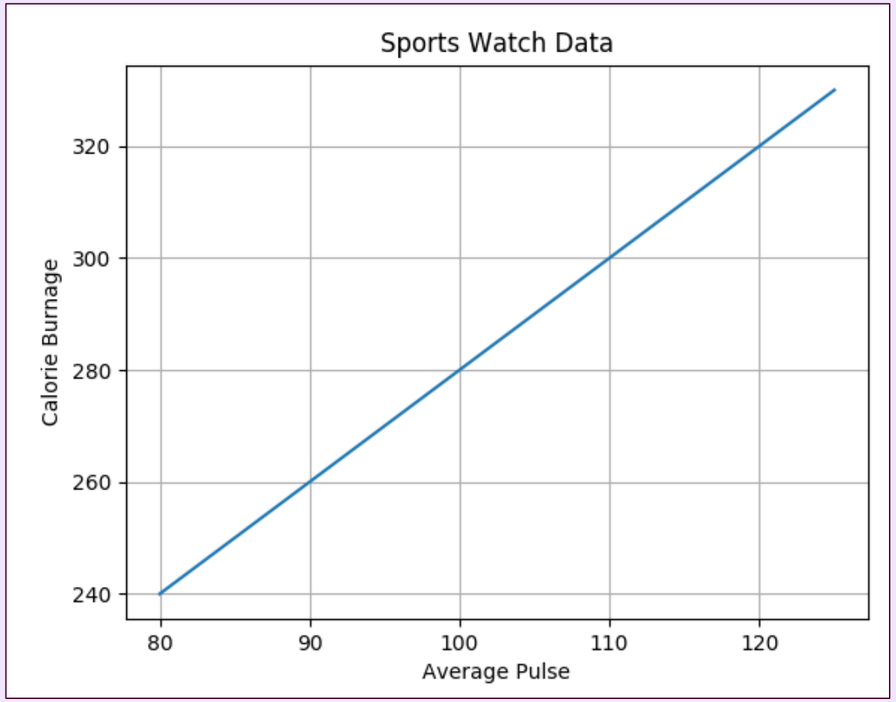


Grid

With Pyplot, you can use the **grid()** function to add grid lines to the plot

Motivation.txt

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105,
110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290,
300, 310, 320, 330])

plt.plot(x, y)
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.title (“Sports Watch Data")
plt.grid()
plt.show()
```

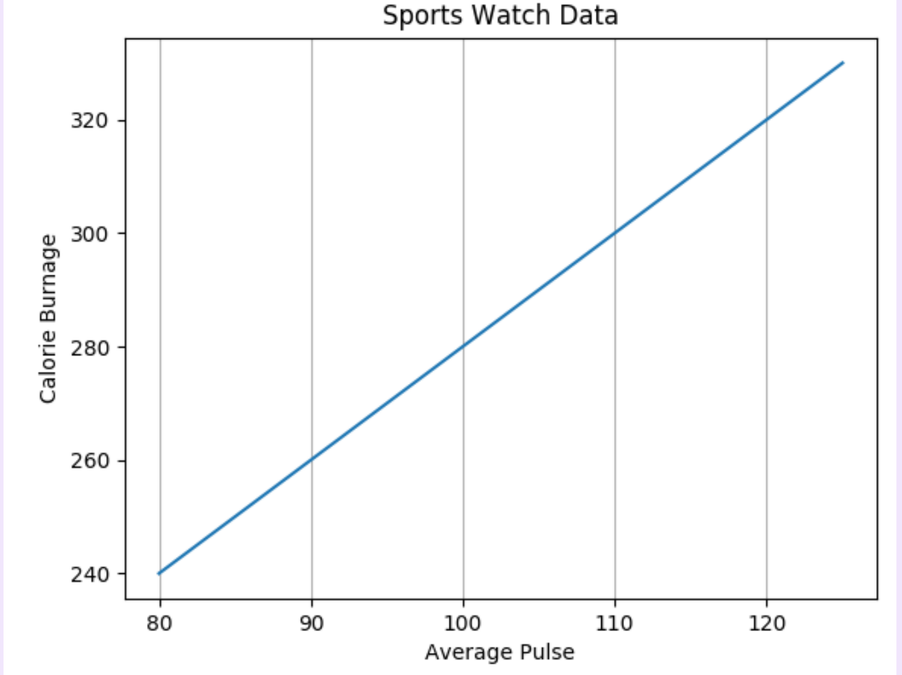


Grid

You can use the **axis** parameter in the grid() function to specify which grid lines to display

Motivation.txt

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105,
110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290,
300, 310, 320, 330])

plt.plot(x, y)
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.title ("Sports Watch Data")
plt.grid(axis='x')
plt.show()
```

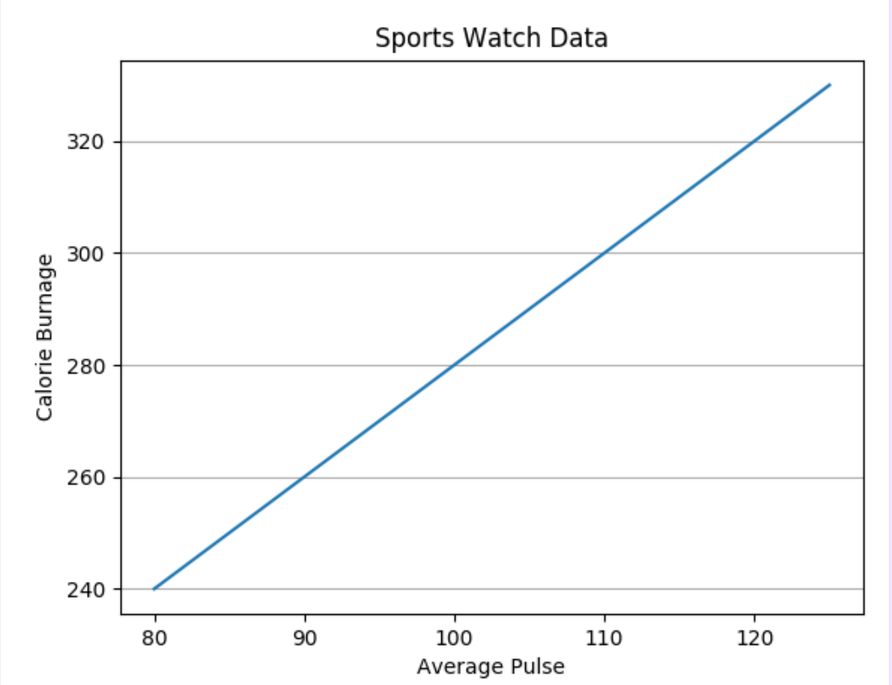


Grid

You can use the **axis** parameter in the grid() function to specify which grid lines to display

Motivation.txt

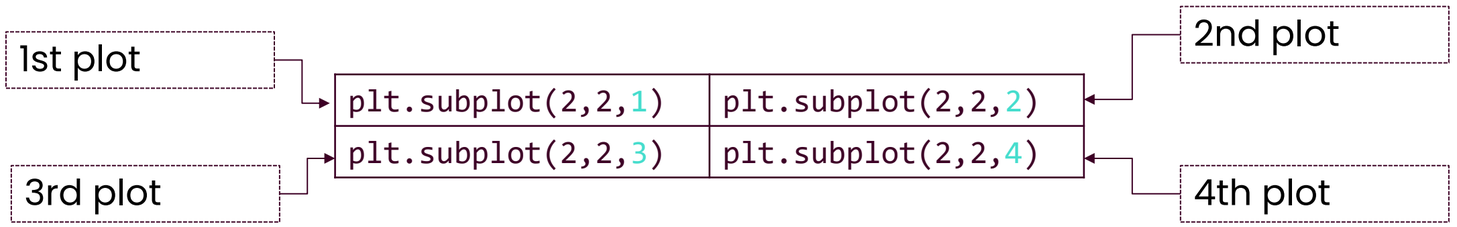
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105,
110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290,
300, 310, 320, 330])

plt.plot(x, y)
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.title ("Sports Watch Data")
plt.grid(axis='y')
plt.show()
```



The subplot( ) function

- With the **subplot()** function you can draw multiple plots in one figure.
- The **subplot()** function takes **three arguments** that describes the layout of the figure.
- The layout is organized in **rows and columns**, which are represented by the **first and second arguments**.
- The **third argument** represents the **index** of the current plot.

This figure consists of 4 charts arranged in 2 rows and 2 columns.

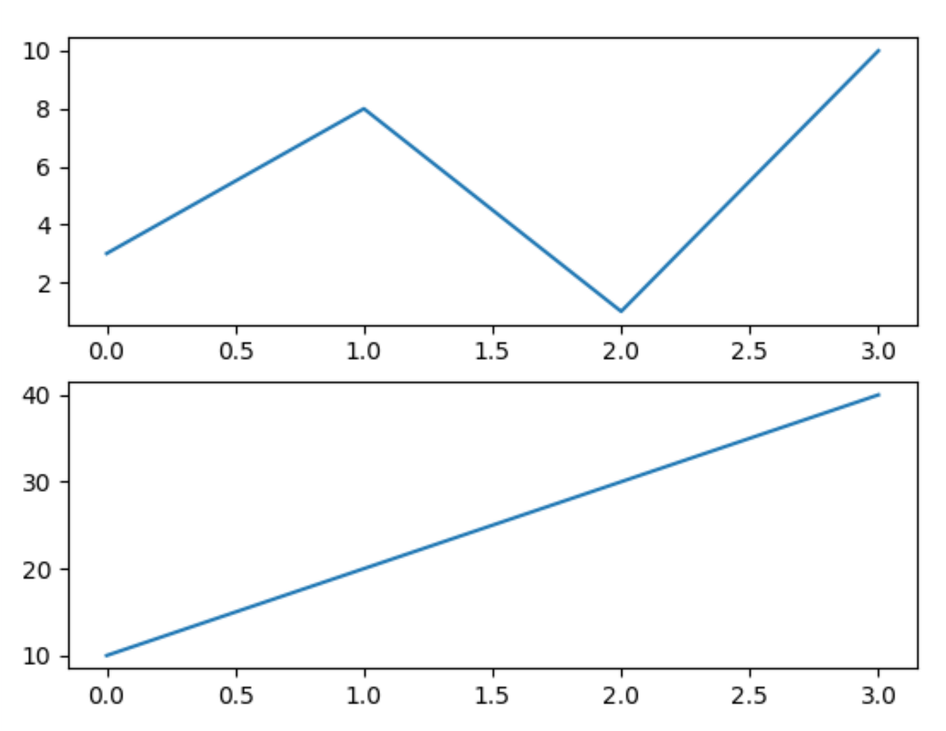


The arrangement of the figures should be chosen by the user, as well as their order. The figures can be arranged in a single column, in a single row, or in a grid.

Subplots

Motivation.txt

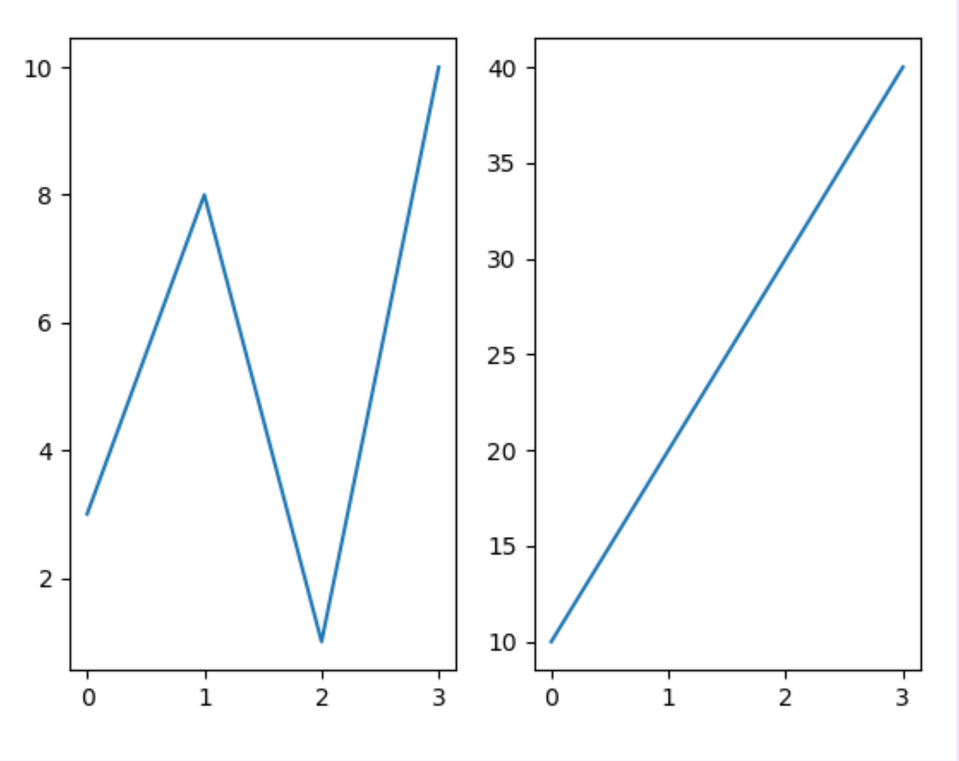
```
import matplotlib.pyplot as plt
import numpy as np

# Plot 1
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

# The figure has 2 rows and 1 column
# This is the first plot
plt.subplot(2, 1, 1)
plt.plot(x, y)

# Plot 2
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

# The figure has 2 rows and 1 column
# This is the second plot
plt.subplot(2, 1, 2)
plt.plot(x, y)
plt.show()
```



Subplots

Motivation.txt

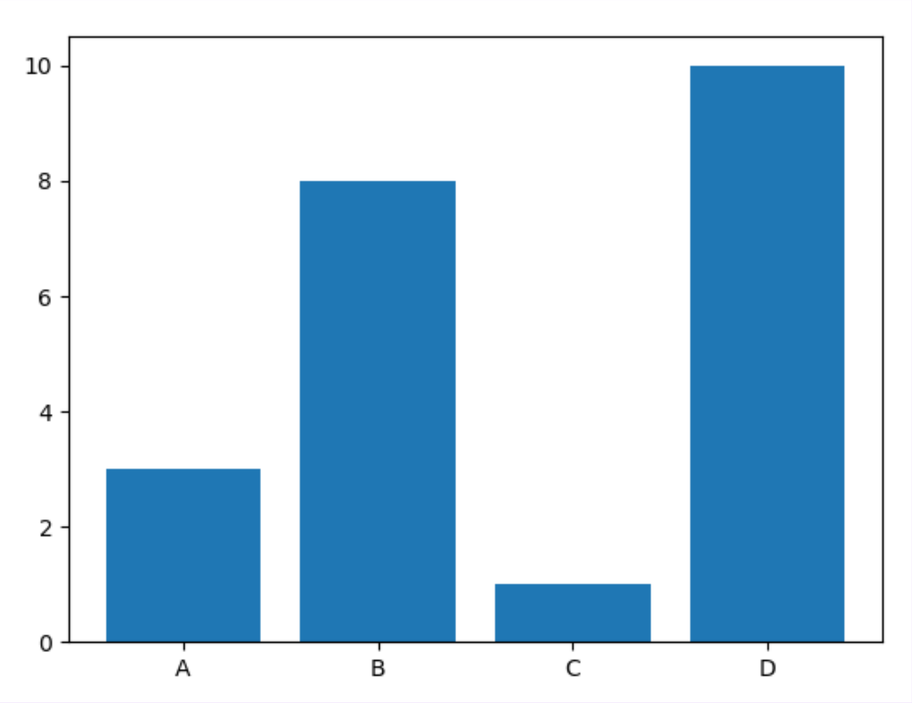
```
import matplotlib.pyplot as plt
import numpy as np

# Plot 1
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

# The figure has 1 row and 2 columns
# This is the first plot
plt.subplot(1, 2, 1)
plt.plot(x, y)

# Plot 2
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

# The figure has 1 row and 2 columna
# This is the second plot
plt.subplot(1, 2, 2)
plt.plot(x, y)
plt.show()
```



Bar plot

Motivation.txt

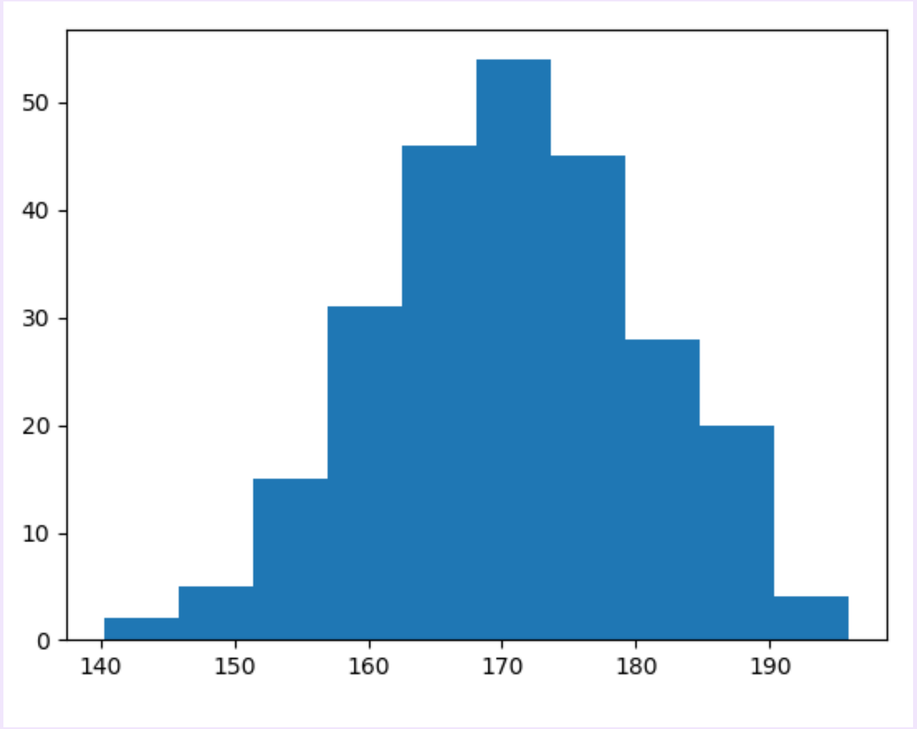
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

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



Histogram

Motivation.txt

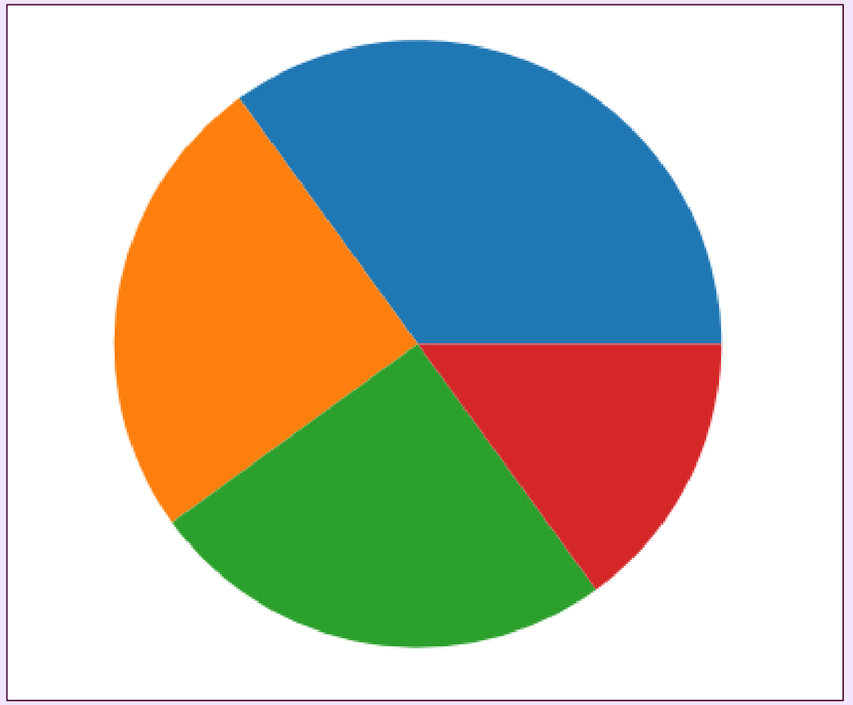
```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.normal(170, 10, 250)

plt.hist(x)
plt.show()
```



Pie chart

Motivation.txt

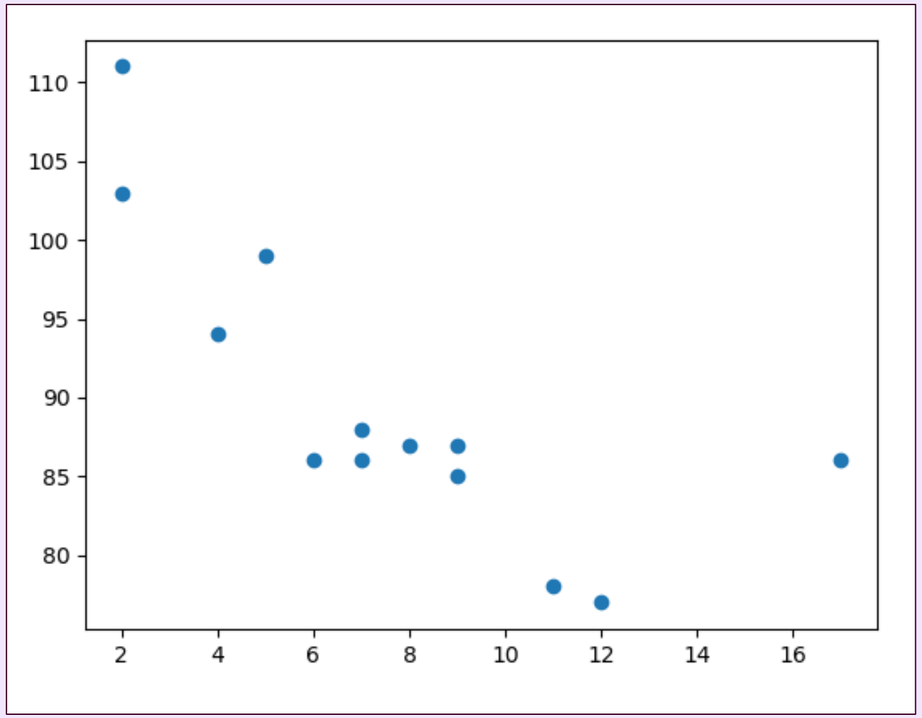
```
import matplotlib.pyplot as plt
import numpy as np

x = np.random.normal(35, 25, 25, 15)

plt.pie(x)
plt.show()
```



Scatter plot

Motivation.txt

```
import matplotlib.pyplot as plt
import numpy as np


# Sample data
x = np.array([5, 7, 8, 7, 2, 17, 2, 9, 4,
11, 12, 9, 6])
y = np.array([99, 86, 87, 88, 100, 86,
103, 87, 94, 78, 77, 85, 86])


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

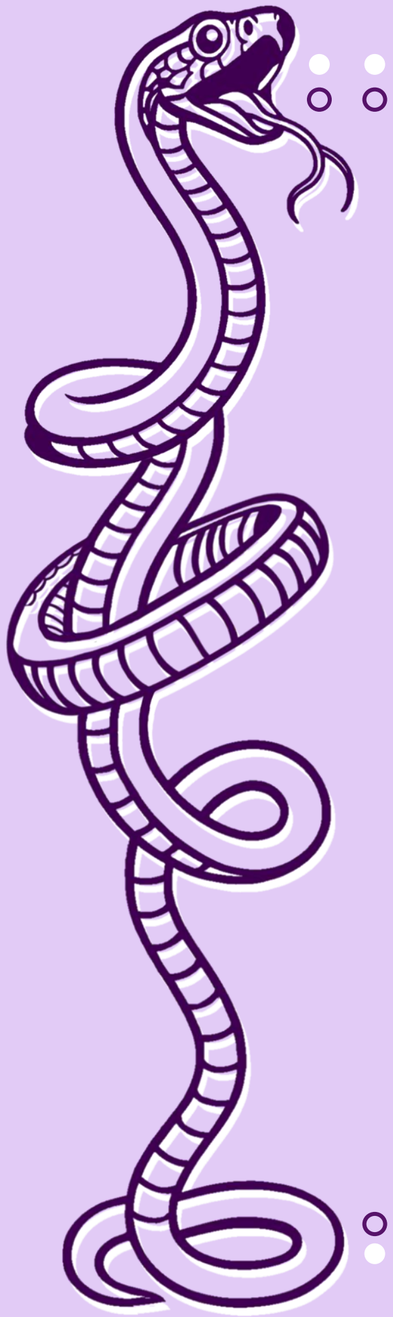


Save plots to figures

Using the function **savefig()**, you can save any plot in any format. The function takes as an argument the path of the directory where you want to keep the figure

Motivation.txt

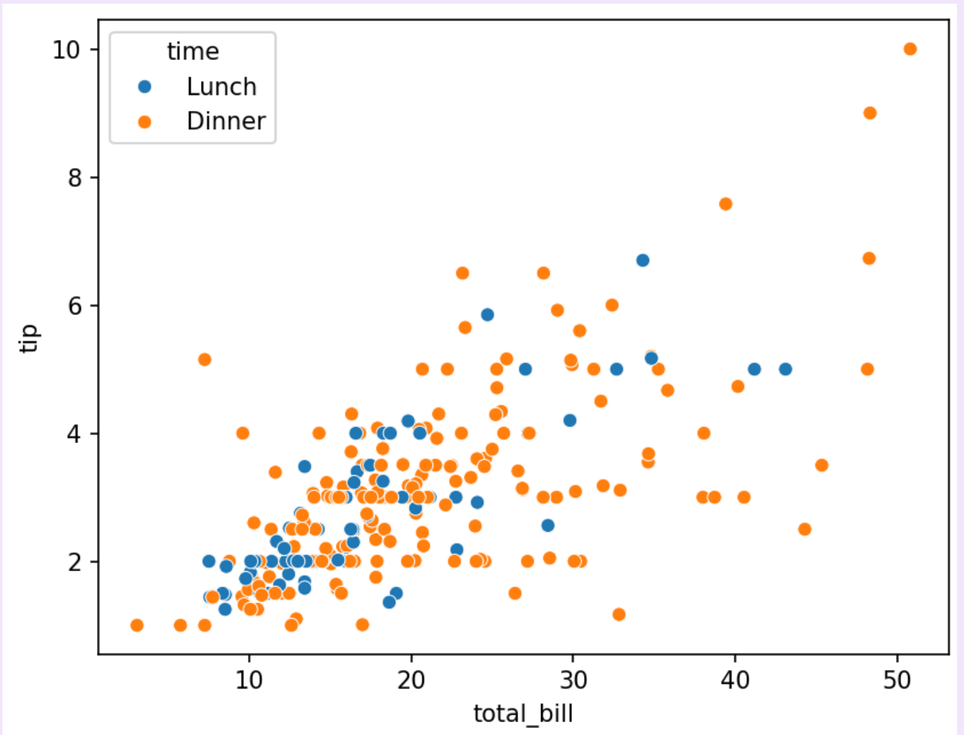
```
import matplotlib.pyplot as plt
import numpy as np

# Data
y = np.array([35, 25, 25, 15])

# Create pie chart
plt.pie(y)

# Save in different formats
plt.savefig('foo.png')
plt.savefig('foo.pdf')
plt.savefig('foo.jpeg')
```



Seaborn

Seaborn

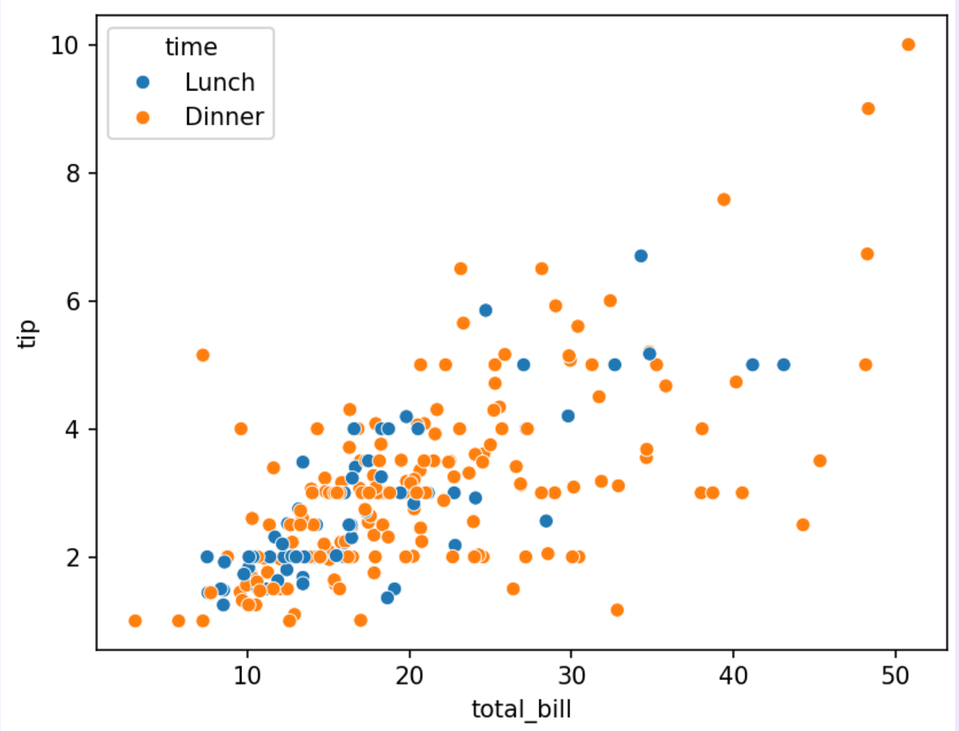
- Built on top of Matplotlib
- Provides **beautiful default styles and simpler syntax**
- Great for exploratory data analysis
- Works seamlessly with Pandas DataFrames



Seaborn

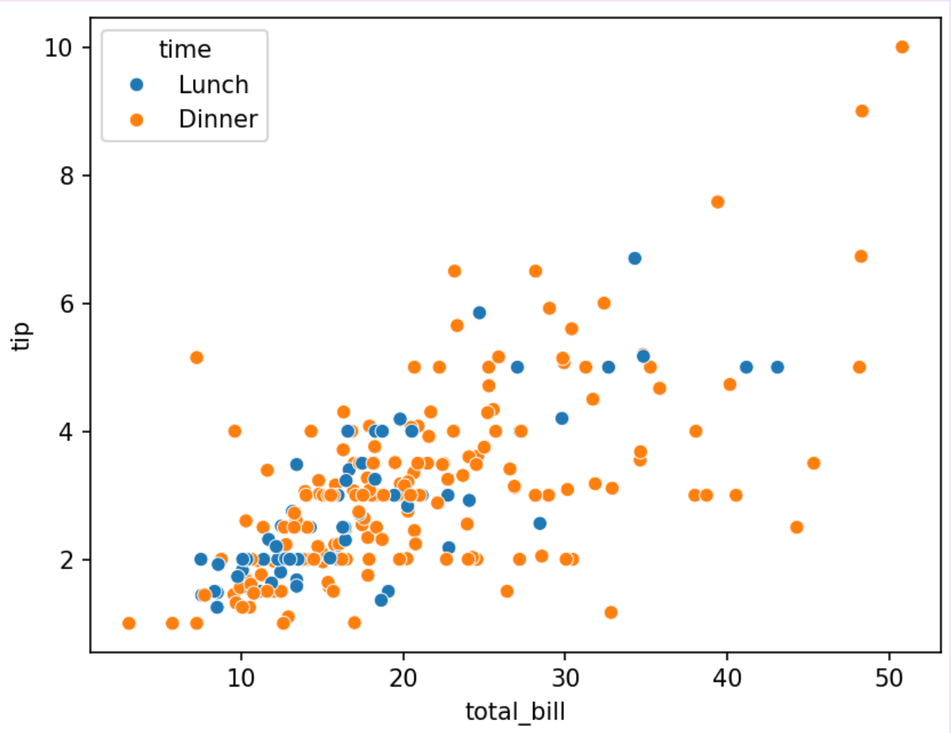
Motivation.txt

```
import pandas as pd

# Sample dataset
tips = sns.load_dataset("tips")

# Quick visualization
sns.scatterplot(data=tips, x="total_bill",
y="tip", hue="time")
plt.show()
```



The hue parameter

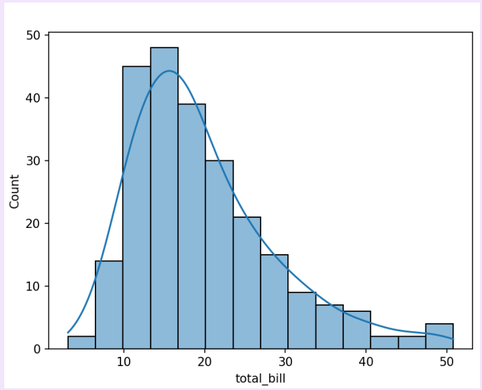
- **hue** adds a **categorical dimension** to your plot.
- It **automatically assigns** different colours to data points/lines based on the values of that category.
- Makes it easy to compare groups in the same visualization.

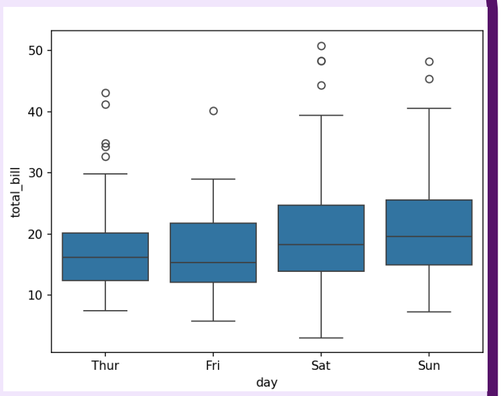


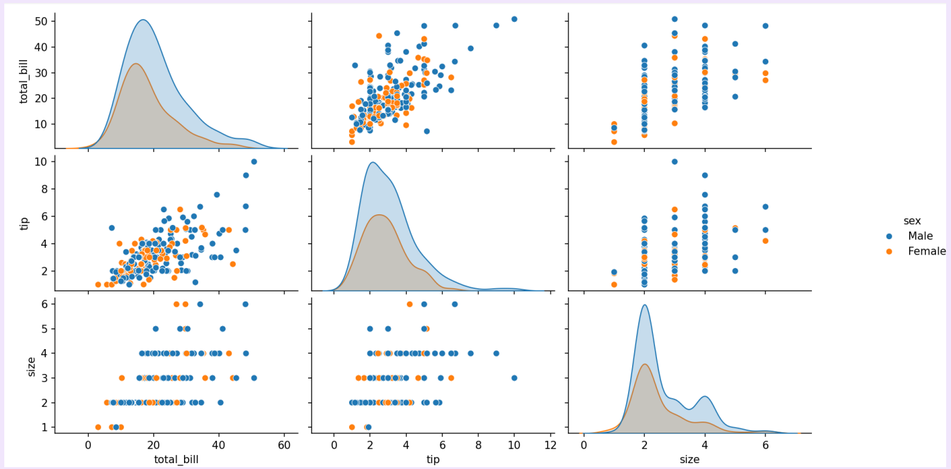
Common Plots Made Easy

Motivation.txt

```
# Distribution
sns.histplot(data=tips, x="total_bill",
kde=True)


# Boxplot
sns.boxplot(data=tips, x="day",
y="total_bill")


# Pairplot
sns.pairplot(tips, hue="sex")
plt.show()
```







When to use matplotlib and seaborn

Matplotlib

- (+) Low-level library → full control over plots
- (+) Great for fine-tuned customization
- (+) Best when you need to build complex or highly specific figures
- (-) More verbose code
- **Use case**: final polishing, layout adjustments, saving in specific formats

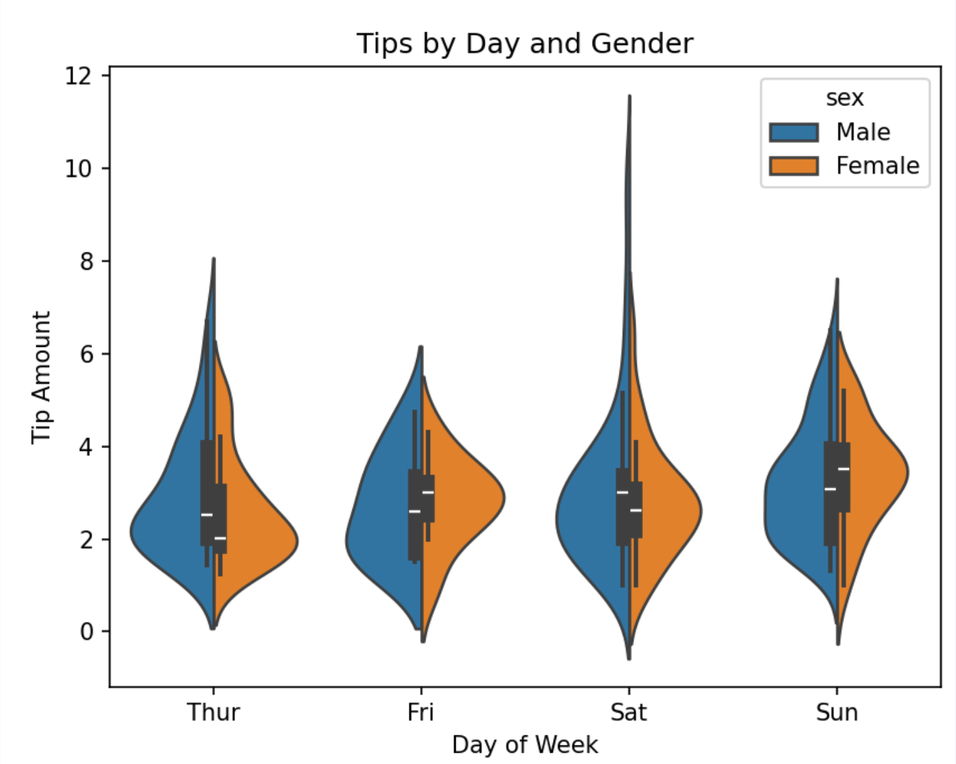
Seaborn

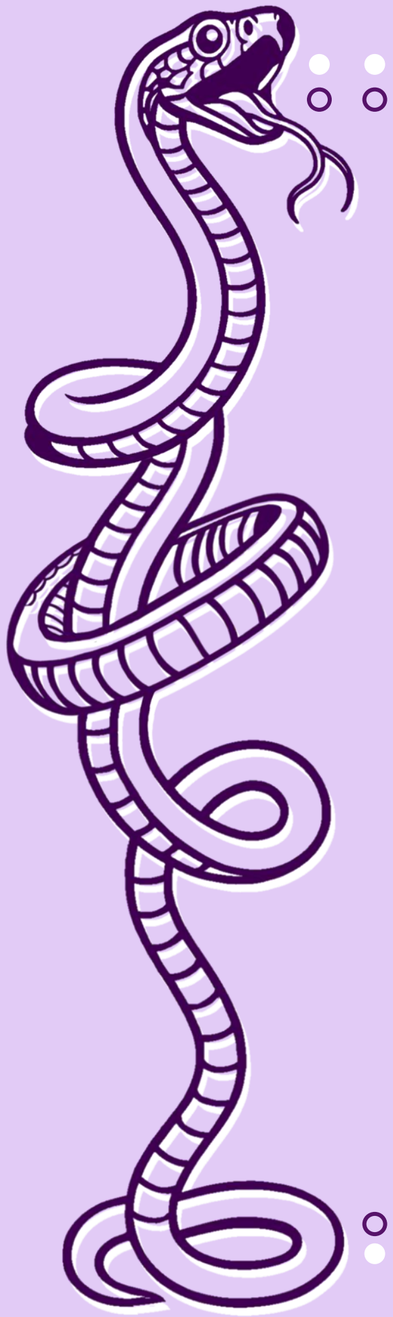
- (+) Built on top of Matplotlib → high-level interface
- (+) Great for quick, beautiful plots with minimal code
- (+) Works directly with Pandas DataFrames
- (-) Less flexible
- **Use case**: quick insights, exploratory plots, statistical visualizations

When They Complement Each Other

Motivation.txt

```
import seaborn as sns
import matplotlib.pyplot as plt

tips = sns.load_dataset("tips")
sns.violinplot(data=tips, x="day",
y="tip", hue="sex", split=True)

# Matplotlib customization
plt.title("Tips by Day and Gender")
plt.xlabel("Day of Week")
plt.ylabel("Tip Amount")
plt.show()
```

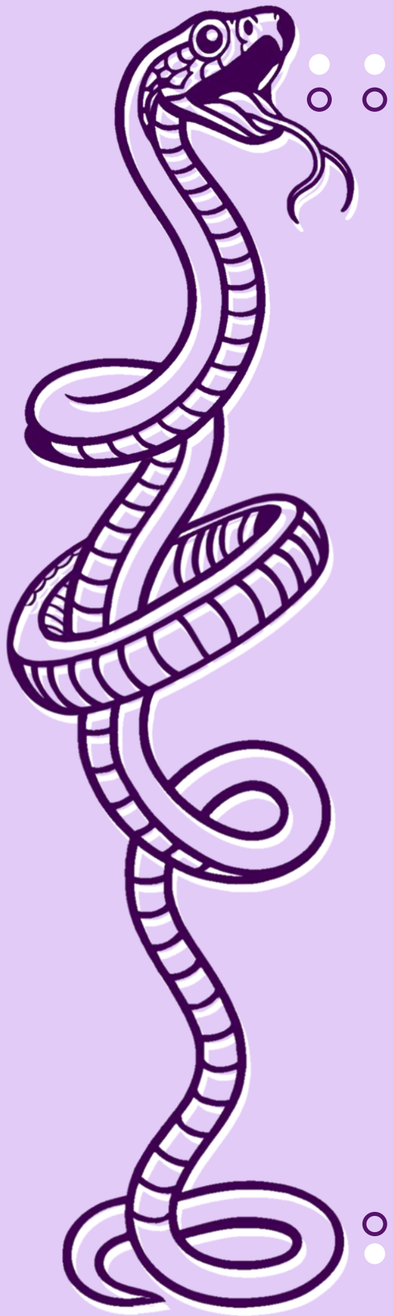




Class wrap-up

What have we learned today?

- **Why we visualize data**
  - To simplify complex information
  - To identify patterns, trends, and insights
  - To bridge the gap between raw numbers and decisions 🔹
- **Matplotlib basics**
  - Creating different types of plots (line, bar, scatter, histogram, pie, subplots)
  - Customizing plots (markers, colors, labels, legends, titles, grids)
  - Saving plots to file
- **Seaborn essentials**
  - High-level, easy-to-use interface built on Matplotlib
  - Common plots with minimal code (histograms, boxplots, pairplots, violinplots)
  - Using hue to compare categories
- **When to use Seaborn vs. Matplotlib and how they complement each other**



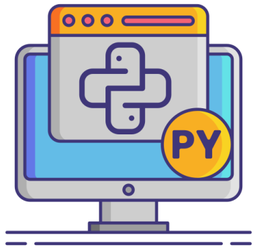
PRACTICE
PRACTICE
PRACTICE

You won't master a skill if you don't practice!

Exercises - Learn by doing!

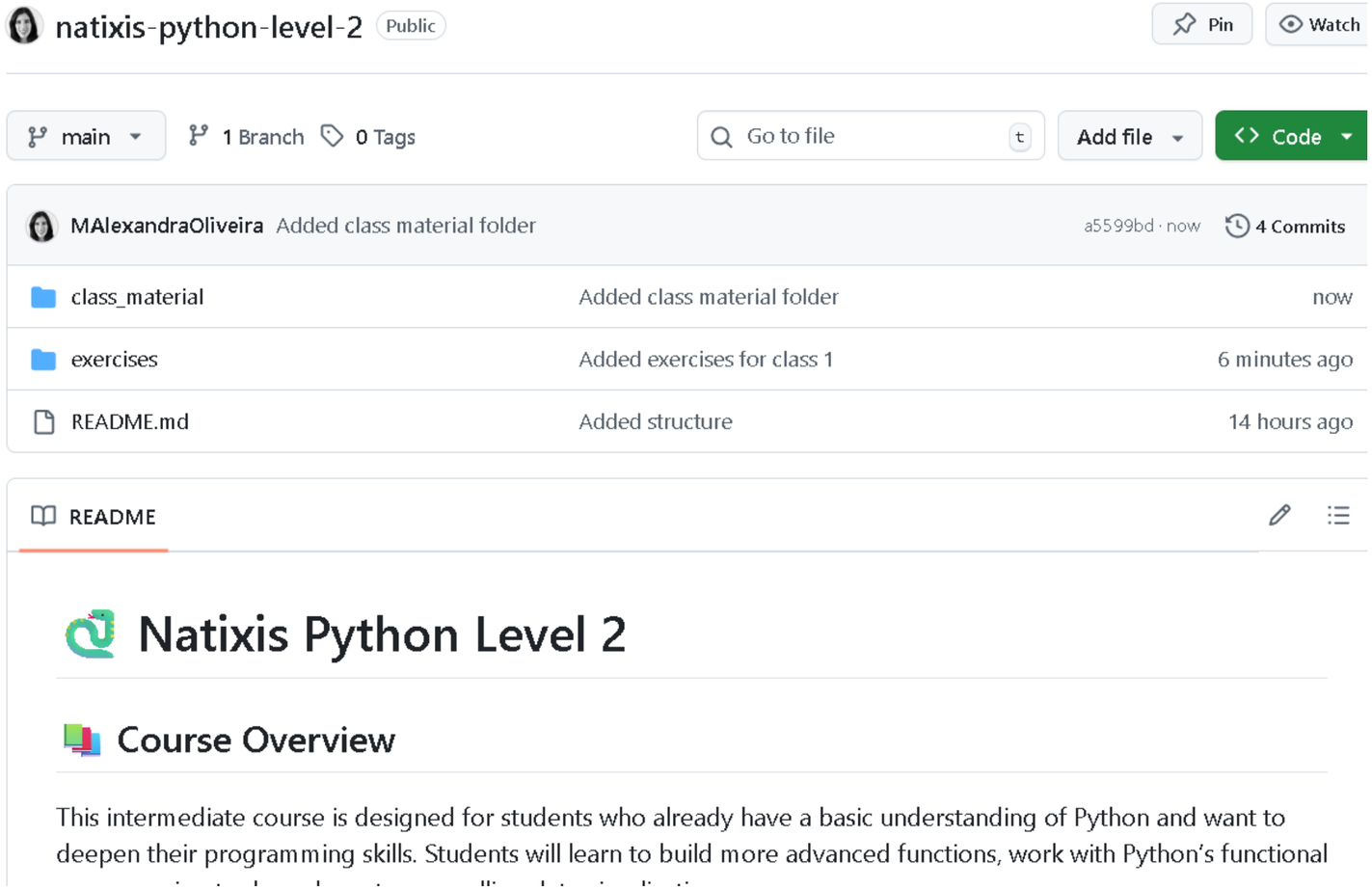
In order to facilitate the learning process of Python **we have prepared for each session a python file** where you can find **exercises** that will help you to grasp the introduced Python concepts.

Visual Studio Code



We will use **VS CODE** as our Python program IDE

Exercises for today



Link to exercises: https://github.com/MAlexandraOliveira/natixis-python-level-2/blob/main/exercises/Class3_exercises.py

Why should you deactivate Copilot? (for now)

As **beginners in Python programming**, it's crucial to focus on truly understanding how code works, rather than just seeing it appear. Tools like GitHub Copilot can be tempting, but they **often offer solutions without explanation**, making it easy to skip the learning process. While these tools are designed to assist, **not replace your thinking**, they can encourage you to rely on solutions you don't fully grasp—and they're not always correct. To truly learn, you need to write, debug, and explore code on your own. **By turning off Copilot** during the early stages of your learning, you give yourself the opportunity to develop real problem-solving skills, build confidence, and create a strong foundation. Later, when you have a solid grasp of the basics, Copilot can serve as a useful support tool, but always approach its suggestions with a critical mindset, not blind trust.

Steps to turn-off GitHub Copilot:

1. Go to Settings (File > Preferences > Settings or press Ctrl+,).
2. In the search bar, type: Copilot.
3. Find the setting GitHub Copilot: Enable.
4. Uncheck it to disable Copilot globally.

THANK YOU

Questions?

Alexandra Oliveira m.alexandra.ro@gmail.com