Module 05 : Data analysis with Numpy and Matplotlib

Working with data

To work with data we need:

- *Data*! Sometimes *a lot* of data (large datasets) from some *source* we *trust* (a sensor, an app, any system, a web service, spreadsheets, etc.)
- *Efficient* and *convenient data structures* to manage, explore and transform data
- *Visualizations* to explore, understand, find interesting things to exploit in large datasets, to support *decision-making*
- Some knowledge in mathematics (statistics, linear algebra) to build *new*, *useful* and *correct* data representations.

> Be careful not to make the data say just anything!

Numerical computing with NumPy

NumPy, short for *Numerical Python*, is the foundational package for scientific computing in Python. It provides, among other things:

- A *fast* and *efficient* multidimensional array object `ndarray`
- Functions for performing *element-wise* computations with arrays or mathematical operations between arrays
- Linear algebra operations, Fourier transforms, and random number generation, etc.

> NumPy is used everyday by hundred of thousands of people. It was key (along with other tools) to produce the first image of a black hole in 2019!

Using NumPy

In the previous chapter we already have installed the *NumPy* package.

To use it, just import the package `numpy`:

```
import numpy as np
```

> We are going to *alias* `numpy` as `np`, because it's shorter and we are lazy!

Ndarray: the *Heart* of the library

The whole NumPy library is based on *one* main object: `ndarray` (which stands for *N-dimensional array*).

NumPy propose a new *iterable* data structure to build and work with multi-dimensional arrays:

```
import numpy as np
# Creating an array from a list with the array() function
array = np.array([0, 1, 2])
print(array)

# Check the type of this new datastructure
print(type(array))

# Type of values stored in the array
print(array.dtype)
```

We need to learn basics to handle this new datatype correctly. It shares some properties with Python `list` your already know and add a lot of useful features!

> To learn NumPy basics, follow the guide NumPy: the absolute basics for beginners

Why not just use lists?

What are the benefits of using NumPy arrays over "standard" Python lists in this case?

While standard Python lists are *flexible* and *easy* to use, NumPy's `ndarray` is *essential* for numerical computing. The difference comes down to *performance* (when working with large amount of data) and *syntax convenience*.

With standard Python lists, *term-to-term* operations (called *element-wise* operations) require *explicit loops* (or list comprehensions), whereas NumPy handles them *natively*. Nice!

Ndarray vs List, an example

Let suppose we have a list of numbers and we want to create a new list where each value is doubled: `[1, 2, 3, 4] => [2, 4, 6, 8]`

```
import numpy as np

lst = [1, 2, 3, 4]
print(lst + lst)# [1, 2, 3, 4, 1, 2, 3, 4] -> list concatenation, NOT what we want!

array = np.array([1, 2, 3, 4])
print(array + array) # [2, 4, 6, 8] -> element-wise addition, what we want !

# To do the same thing with lists,
# we need explicit loop:
lst_double = []
for x in lst:
    lst_double.append(x+x)
print(lst_double)

# Or, with list comprehension:
print([x+x for x in lst])
```

> arrays allows to write element-wise operation easily! Arrays are "optimized lists" for doing computation

What is an array?

- In computer programming, an *array* is a structure for storing and retrieving data. Data are *ordered* and *contiguous* in memory.
- Array is like a *grid*, with each *cell* storing *one element*.

> For convenience, in this course, we will refer to *ndarray* simply as *array*.

Restrictions on arrays

Most NumPy arrays have some *restrictions*:

- All elements of the array must be of the **same type of data** (contrary to lists)
- Once created, the total *size* of the array *can't change*! You can't add or remove elements (contrary to lists)
- The shape must be *rectangular*: each *row* of a two-dimensional array must have the same number of columns.

Creating an array

```
a = np.array([10, 20, 40])
print(type(a))
print(a)
```

Multidimensional arrays 1/2

We might visualize a *one-dimensional* array like a *list*:

```
One-dimensional array (similar to a list)
[1, 2, 3, 4]
```

A two-dimensional array would be like a *table*:

```
Two-dimensional array (table)
[
 [ 1, 2, 3, 4 ],
 [ 5, 6, 7, 8 ],
]
```

What is a *multidimensional* array ? It's just an array *of* arrays!

> We can create 2d, 3d, 4d,..., Nd arrays! (hence the name *ndarray*)

Multidimensional arrays 2/2

```
import numpy as np

one_dim = np.array([1, 2, 3, 4])

# Access element by index, just like with lists!
print(one_dim[0])

# 2d array is just an array of arrays!
two_dim = np.array([ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ])

print(two_dim[0], type(two_dim)) # [1, 2, 3, 4] <class 'numpy.ndarray'>

# Access individual elements:
print(two_dim[0][0], type(two_dim[0][0])) # 1 <class 'numpy.int64'>
print(two_dim[0][1])                      # 2
print(two_dim[2][3])                      # 12
# How to access the value 5?
```

Array attributes: dimension(s), shape and size

```
a = np.array([
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12]
  ])

# number of axes, or dimensions, of the array.
a.ndim # 2 (table)

# Number of elements along each axis or dimension
a.shape # (3, 4)

# Total number of elements
a.size # 12
```

Slicing 2d arrays

Slicing, just like for lists, lets you grab a portion of an array rather than just a single element. The syntax for a 2D slice is:

```
array[row_start:row_stop, column_start:column_stop]
```

Slicing a 2D array returns another 2D array.

> Slicing arrays work the same way as slicing lists! Remember that `*_start` is *included*, `*_end` is *excluded*!

Note: **Leaving a side of the colon empty** (`:`) means *"take everything from that dimension"*

Slicing 2d arrays, some examples

```
# A simple 3x3 grid
data = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# Copy of the original data (all rows and cols)
data[:,:]

# All columns of second row
data[1:2, :] # array([[4, 5, 6]])

# Sub grid: first two rows and two last columns
data[0:2, 1:3] #array([[2, 3], [5, 6]])
```

Integer indexing

Instead of passing a *range*, you are passing a *specific integer index* for the row axis. Selecting a single element using an integer *removes/collapses that axis*. Since one dimension is removed, the result will be a 1D array:

```
# All rows in column 0
data[:, 0] # array([1, 4])

# All cols for row 0
data[0, :] # array([1, 2, 3])

# First two cols for row 0
data[0, 0:2] # array([1, 2])
```

More about creating arrays: `zeros`, `ones` and `eye`

The `zeros()` method, for example, creates a full array of zeros with dimensions defined by the shape argument. For example, to create a two-dimensional array 3x3:

```
zeros = np.zeros((3,3))
print(zeros)
```

The `ones()` function creates an array full of ones in a very similar way:

```
ones = np.ones((2,2))
print(ones)
```

The function `eye()` creates the *identity* matrix:

```
id_matrix = np.eye(4) # 1 on the diagonal, 0 elsewhere
print(id_matrix)
matrix = np.random.random((4,4))
print(matrix)
# Matrix product (A @ I = A by definition):
print(matrix @ id_matrix)
```

Creating empty arrays

The function `empty` creates an array whose initial content is *random* and depends on the state of the memory. The reason to use `empty` over `zeros` (or something similar) is *speed*.

```
empty_arr = np.empty((3,3))
print(empty_arr)
```

Just **make sure to fill every element afterwards**!

Random array values

Numpy provides ways to generate arrays filled with random values:

```
# Try this
print(np.random.random((3,3)))
print(np.random.random((5,5)))
print(np.random.randint(0, 100, size=(2, 2)))
```

Output:

```
[[0.0260948  0.55257481 0.91550205]
 [0.77040681 0.40863254 0.19638786]
 [0.06447059 0.96170963 0.30521651]]

[[0.45660994 0.02564421 0.09505229 0.75434    0.43673664]
 [0.64206406 0.85134895 0.94516467 0.15298589 0.33782115]
 [0.38609371 0.80283616 0.5094258  0.12349448 0.30232964]
 [0.50544114 0.88387591 0.94692192 0.27555088 0.37534425]
 [0.82641219 0.31479287 0.9597241  0.23700839 0.49438009]]

[[ 0 54]
 [72 57]]
```

More about creating arrays: `arange`

A feature that will be particularly useful is `arange()`. This function **generates NumPy arrays with numerical sequences** that respond to particular rules depending on the passed arguments:

```
print(np.arange(10)) # [0 1 2 3 4 5 6 7 8 9]
print(np.arange(1, 10, 3)) # [1 4 7]
```

> `arange`, for *array range*, is similar to the `range()` function we have already seen and works the same way! Good! We can reuse what we have learned before!

More about creating array: `linspace`

Another function very *similar* to `arange()` is `linspace()`.

This function still takes as its first two arguments the initial and end values of the sequence, *but the third argument*, instead of specifying the distance between one element and the next, *defines the number of elements* into which we want the interval to be split.

An example worth thousand words:

```
# We want 5 evenly distributed elements between 0 and 10:
print(np.linspace(0, 10, 5)) # [ 0.   2.5  5.   7.5 10. ]

# We want 6 evenly distributed elements between 1 and 2:
print(np.linspace(1, 2, 6)) #  [1.  1.2 1.4 1.6 1.8 2. ]
```

Flatten: turn a nd array into a 1D array

To *flatten* an array (convert it to an 1d array), use the flatten method:

```
data = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

data_in_one_dim = data.flatten()

print(data_in_one_dim.shape)
```

Reshape

To generate two-dimensional arrays you can use the `arange()` function but combined with the `reshape()` function. This function *divides* a linear array in different parts in the manner specified by the *shape* argument:

```
# Reshape the 1d array in a 2d array with 3 rows and 4 columns
print(np.arange(0, 12).reshape(3, 4))
```

Output:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Element-wise operations and broadcasting

Many operations are defined and possible on arrays.

Array operations are performed *element-wise* (term-to-term). As a general rule, to do arrays operation, arrays must have the *same shape*:

I you multiply an array by a scalar, every element is multiplied by this scalar. This is called *broadcasting*:

```
import numpy as np

array1 = np.array([1, 2, 3, 4])
array2 = np.array([5, 6, 7, 8])
array3 = np.array([[1, 0], [0, 1]])

print(array1 * 2) # broadcasting
print(array1 / 100 - array1)
print(array1 + 2 * array1 - 50)
print(array1 + array2)
# print(array1 + array3) Error: array1 and array3 do not have the same shape !
```

Practice: Creating arrays of different shapes

1. **Create** 1D-array including multipliers of 3 between 0 and 100.
2. Given `a = np.arange(4)`, **what** is the output of `a * 2`?
3. **Reshape** the array to create a 2x17 ndarray
4. **Create** a 3D array with shape (2, 3, 4) containing all zeros.
5. **Create** a 2D array with shape (2, 4) containing random *integers* between 0 and 9.
6. **Create** a 2D array `id` with shape (4, 4) containing the *eye* matrix.
7. **Create** two 1D array `a` and `b` with shape `(4,)` containing ones. **Add** them together and multiply the result by 5. What is the result?
8. What is the result of `a * id`? What is the result of `a @ id`?

Correction

```
import numpy as np

multiples_of_3 = np.arange(0, 101, 3)
print(multiples_of_3)
reshaped = multiples_of_3.reshape(2, 17)
print(reshaped)
print(np.zeros((2,3,4)))
print(np.random.randint(0, 10, size=(2, 4)))
print(np.eye(4))

a = np.ones((4,))
b = np.ones((4,))
print((a+b) * 5)

# * operator performs *element-wise* multiplication
print(a * id)

# @ operator performs *vector-matrix dot product*
# Multiplying any vector by the identity matrix yields the original vector itself.
print(a @ id)
```

Practice

Write a NumPy program to convert Centigrade degrees (C) into Fahrenheit degrees (F). Values are stored in a NumPy array. Round the value to two decimal places with the `np.round` function.

Here is the relation between Centigrade and Fahrenheit degrees:

$$\mathrm{C} = \frac{5}{9}(\mathrm{F} - 32)$$

```
import numpy as np
# List of Fahrenheit values
fvalues = np.array([0, 12, 45.21, 34, 99.91, 32])
print("Values in  Centigrade degrees:")
#...
```

Solution

```
import numpy as np

# List of Fahrenheit values
fvalues = np.array([0, 12, 45.21, 34, 99.91, 32])

print("Values in Centigrade degrees:")

# We can directly use broadcasting:
print(np.round((5 * fvalues / 9 - 5 * 32 / 9), 2))
```

Practice on DataCamp

- Course: Introduction to Numpy, complete *Chapter 1 : Understanding NumPy Arrays* only for now

Operations over arrays

`sum`, `mean`, `min`, and `max` are the most common operations you will use when analyzing data. By default, **they operate on every single element** in the array, **returning a single number** (a scalar).

```
import numpy as np
# A quick dataset of test scores (/100) for 3 students across 4 subjects
scores = np.array([
    [85, 90, 78, 92], # Student 1
    [70, 85, 88, 79], # Student 2
    [95, 92, 96, 98]  # Student 3
])

# Overall array operations
print("Total sum of all scores:",np.sum(scores))    # Grand total
print("Average score overall:",  np.mean(scores))   # Mean (average)
print("Median score overall:",   np.median(scores)) # Median
print("Lowest score in class:",  np.min(scores))    # Minimum value
print("Highest score in class:", np.max(scores))    # Maximum value
```

> See documentation on sums, products, differences and extrema finding

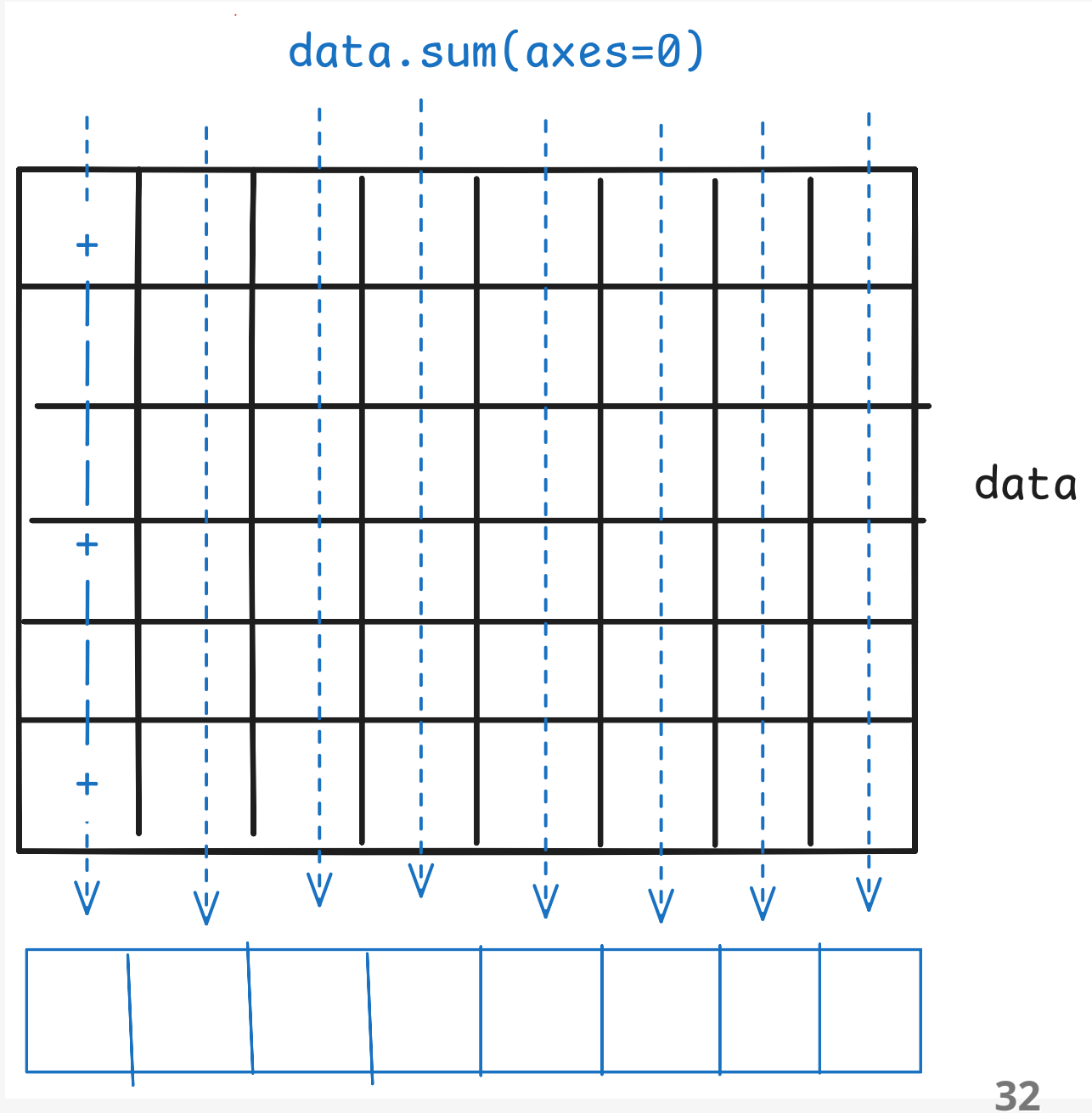
Working with array axes

When performing math operations like `sum()` or `mean()` on 2D arrays, NumPy asks you to specify an **axis**:

- `axis=0` (Vertical Axis): Operations move *down the rows*. It collapses the rows to give you a result for *each column*.
- `axis=1` (Horizontal Axis): Operations move *across the columns*. It collapses the columns to give you a result for *each row*.

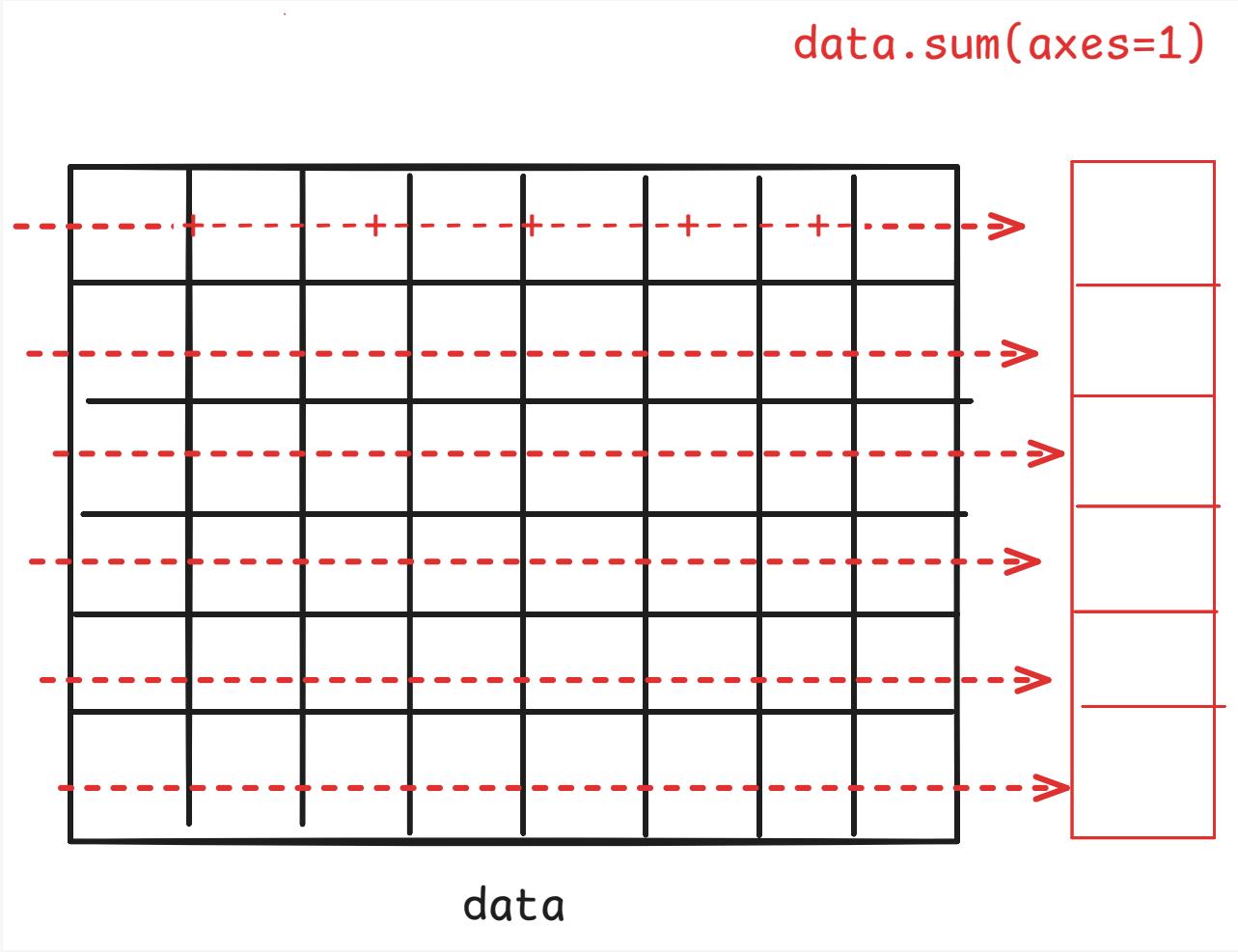
Operations with `axis=0` (down the rows)

```
data.sum(axis=0)
#or
np.sum(data, axis=0)
```



Operations with `axis=1` (across columns)

```
data.sum(axis=1)
#or
np.sum(data, axis=1)
```



Working with array axes, examples

```
import numpy as np

# A quick dataset of test scores (/100) for 3 students across 4 subjects
scores = np.array([
    [85, 90, 78, 92], # Student 1
    [70, 85, 88, 79], # Student 2
    [95, 92, 96, 98]  # Student 3
])

# Average score for *each subject* (collapsing the rows, axis=0)
subject_averages = np.mean(scores, axis=0)
print("Averages per subject:", subject_averages)

# Average score for *each student* (collapsing the columns, axis=1)
student_averages = np.mean(scores, axis=1)
print("Averages per student:", student_averages)
```

Fancy indexing

In NumPy, *fancy indexing* allows us to use an *array of indices* to access multiple array elements *at once*:

```
array = np.array([10, 20, 30, 40, 50])
print(array[ [0, 1, 4] ]) # array([10, 20, 50])
```

Filtering arrays

Filtering arrays in NumPy is commonly used to select and operate on subsets of data that fulfill specified conditions, such as values greater than a threshold.

This is achieved using *boolean indexing*, where a boolean array, derived from operations on the original array, is used to select elements.

```
import numpy as np

array = np.array([1, 2, 3, 4, 5])
condition = array > 3
greater_than_3 = array[condition]
print(greater_than_3) # Output: [4, 5]
```

Filtering with multiple conditions

```
import numpy as np

array = np.array([10, 15, 20, 25, 30])

between_15_and_30 = array[(array > 15) & (array < 30)]

print(between_15_and_30) # Output: [20, 25]
```

Using `np.where` for conditional filtering

The use of `np.where` provides more flexibility by returning *the indices of elements* that meet the condition, which can then be used for further operations:

```
import numpy as np

array = np.array([5, 10, 15, 20, 25, 31])

# indexes of even numbers or numbers greater than 20
indices = np.where((array % 2 == 0) | (array > 20))

filtered_array = array[indices]

print(filtered_array) # Output: [10, 20, 25, 31]
```

Practice Numpy on DataCamp

- Course: Introduction to Numpy, complete *Chapters 2* and *Chapter 3*
- Problem 1: Arrays and sales

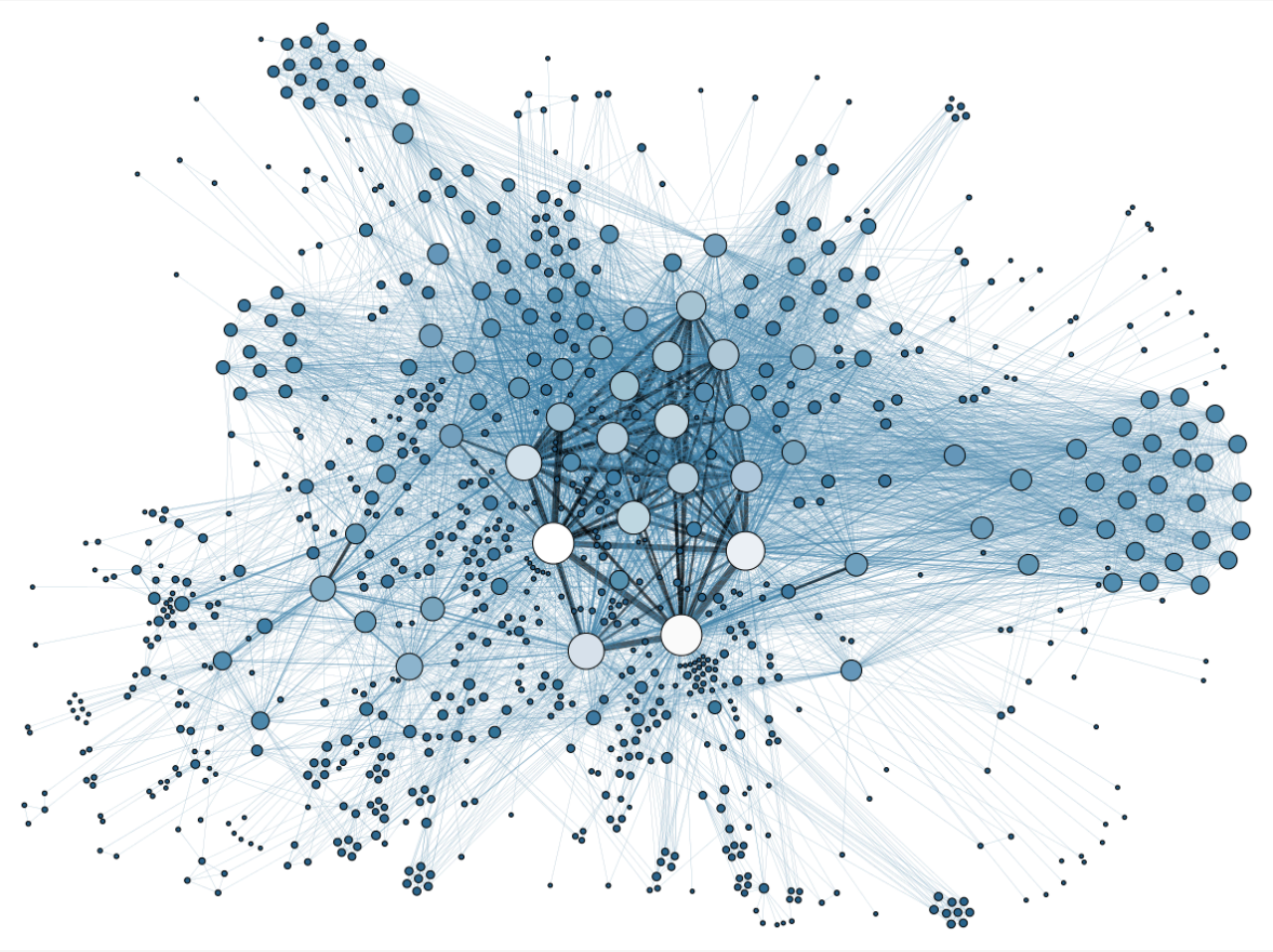
Visualizing data with Matplotlib

Data visualization is the *art* of graphical representation of data (information).

This can be achieved using visual elements like charts, graphs, maps and more.

> Source: Social Network Analysis Visualization (Wikimedia Commons)

> Based on Prof.Karim ZKIK's Course Material



Why data visualization *matter*?

Data visualization is really important to *explore* and *understand* data. You can convert *complex* and *large* datasets into an easy to understand representation.

It is also really important to *deliver your message*, *support your arguments* and for good *decision-making*.

> A *good* chart is worth a thousand words!

Matplotlib

Matplotlib is a Python library specializing in the development of two-dimensional charts (including 3D charts). In recent years, it has been widespread in scientific and engineering circles.

Features:

- Simplicity in its use
- *Gradual development* and *interactive* data visualization
- *Full control* over graphic elements
- Export in *many formats* both *raster* (e.g. PNG, JPG) and *vector* (e.g. PDF, EPS, SVG).
- Support for expressions and texts in LaTeX

Installing and importing `matplotlib.pyplot`

In the previous chapter, we already have installed Matplotlib.

We are going to use the Matplotlib module called `pyplot` for making plots:

```
import matplotlib.pyplot as plt
# Our code...
```

> `plt` is the standard alias used by the community

Matplotlib tutorial

To grasp the basics, and have a first experience with `pyplot`, you can follow the official tutorial

> We will only *scratch the surface* of this very complete and powerful library!

Making figures

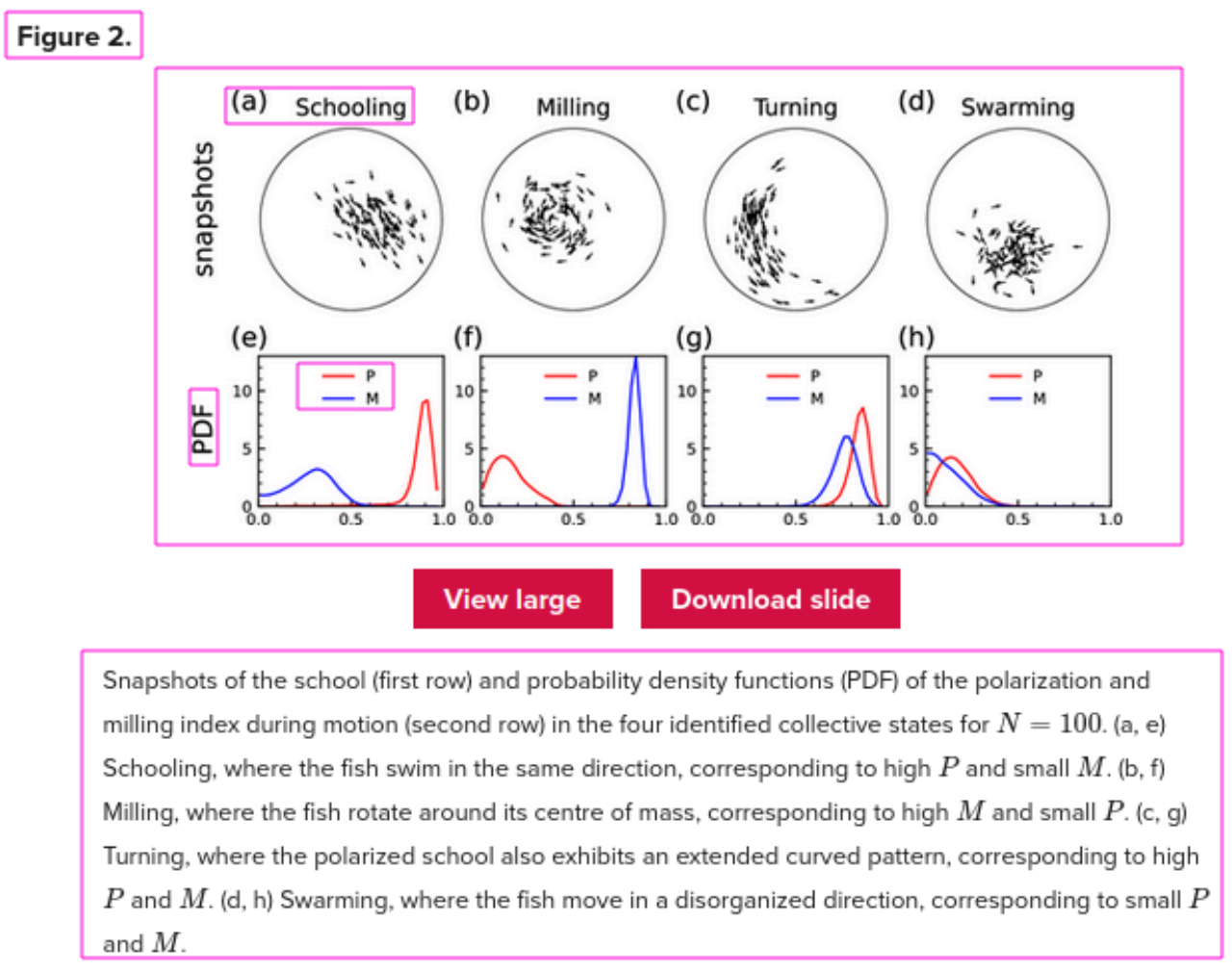
With pyplot, we are going to produce a *figure*.

A *figure* is any non-text visual element included in a document or presentation.

A figure is made of *different parts*:

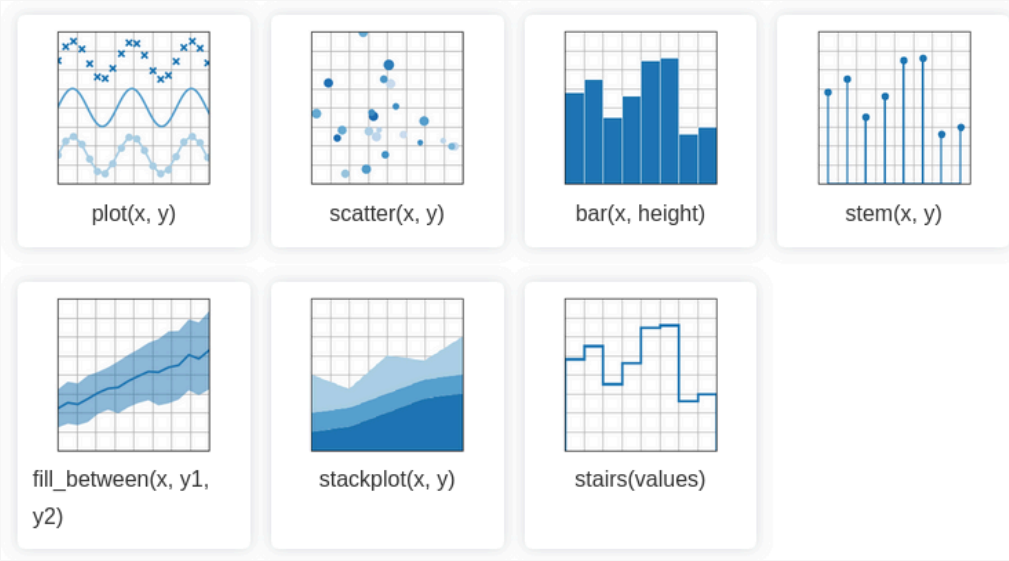
- The *visual* graphic (plot)
- *Labels* (axis titles, legends)
- A *caption* with a *concise title* and *explanation* so the graphic can be understood independently from the context.

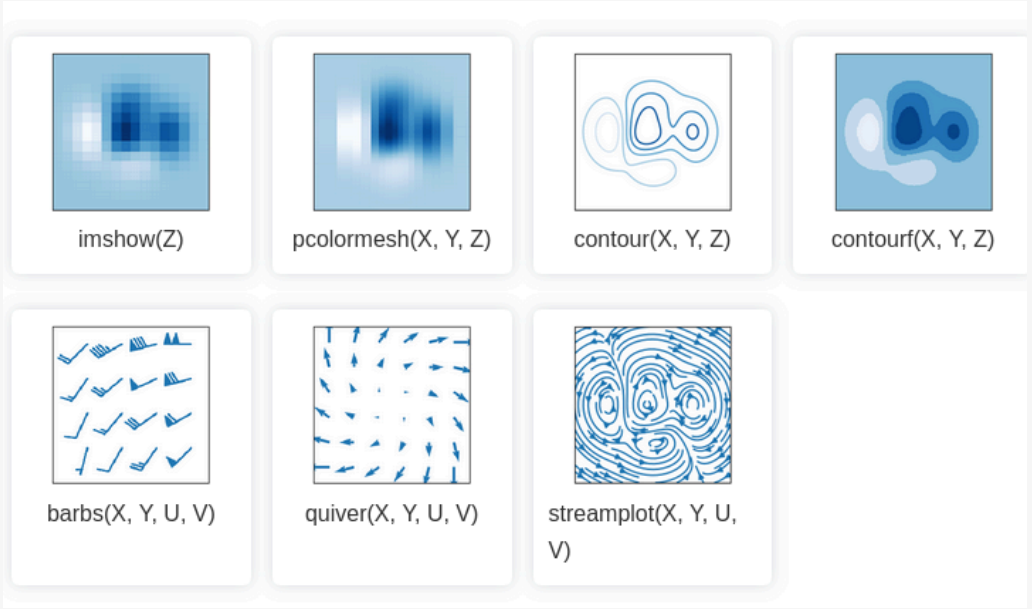
> Source



Plot types

Matplotlib allows you to create all kinds of visualizations: plots, histograms, power spectra, bar charts, error-charts, scatter-plots, etc., *with just a few lines of code*.





See all different plot types you can make

Basics

Each `pyplot` function makes *some change* to the *figure*:

- creates a *figure*,
- creates a *plotting area* in the figure,
- *plots* some lines in the plotting area,
- *decorates* the plot with labels, etc.

Whenever you plot with matplotlib, the *two main code lines* should be considered:

- **Type of graph**: this is where you define a bar chart, line chart, etc.
- **Show the graph** : this is to display the graph

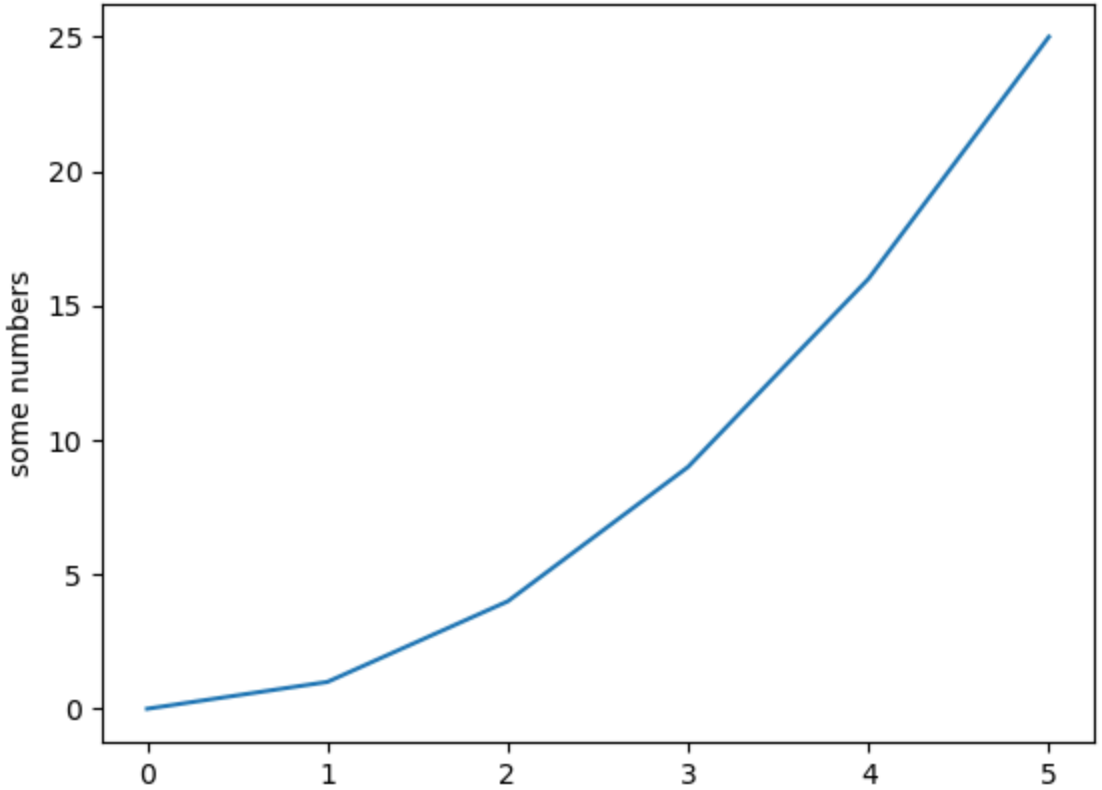
Your first plot

1. **Create** a file `my_first_plot.py` with the following content:

```
import matplotlib.pyplot as plt

plt.plot([0, 1, 4, 9, 16, 25])
plt.ylabel('some numbers')
plt.show()
```

2. **Run** this program.
3. **Add** `plot([0, 1, 4, 9, 16, 25], 'bo')` *after* `plt.plot([0, 1, 4, 9, 16, 25])`. Run the program again.



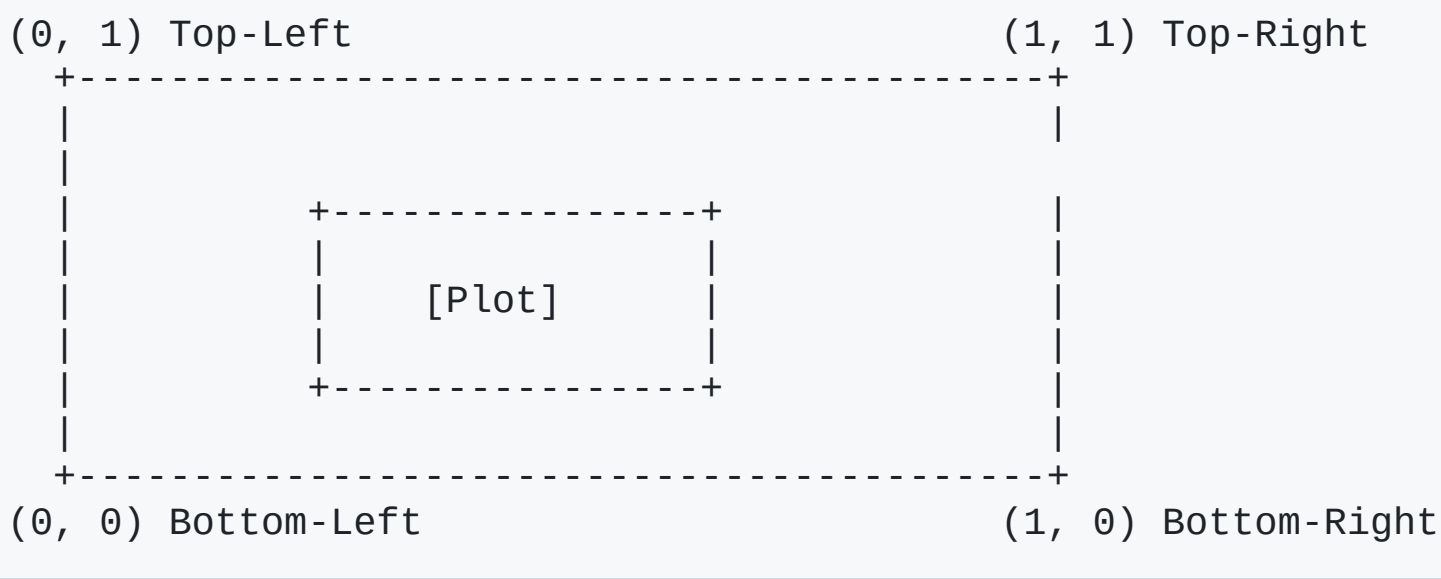
Your first *figure*

Let's turn this plot into a proper *figure*, with nice labels, title and caption:

4. **Replace** *ylabel* with the following value: `"$y$"`
5. **Add** an *xlabel* `"$x$"` with the `xlabel()` method
6. **Add** a *title* `"$y=x^2$"` with the `title()` method
7. **Add** a *caption* at the bottom with the `figtext()` method:

```
plt.figtext(
    0.5, 0.025,  # (x, y): coordinates of the anchor point
    r"Figure 1: This plot shows the function $y = x^2$",
    fontsize=10,
    ha="center", # text horizontal alignement relative to the anchor point
)
```

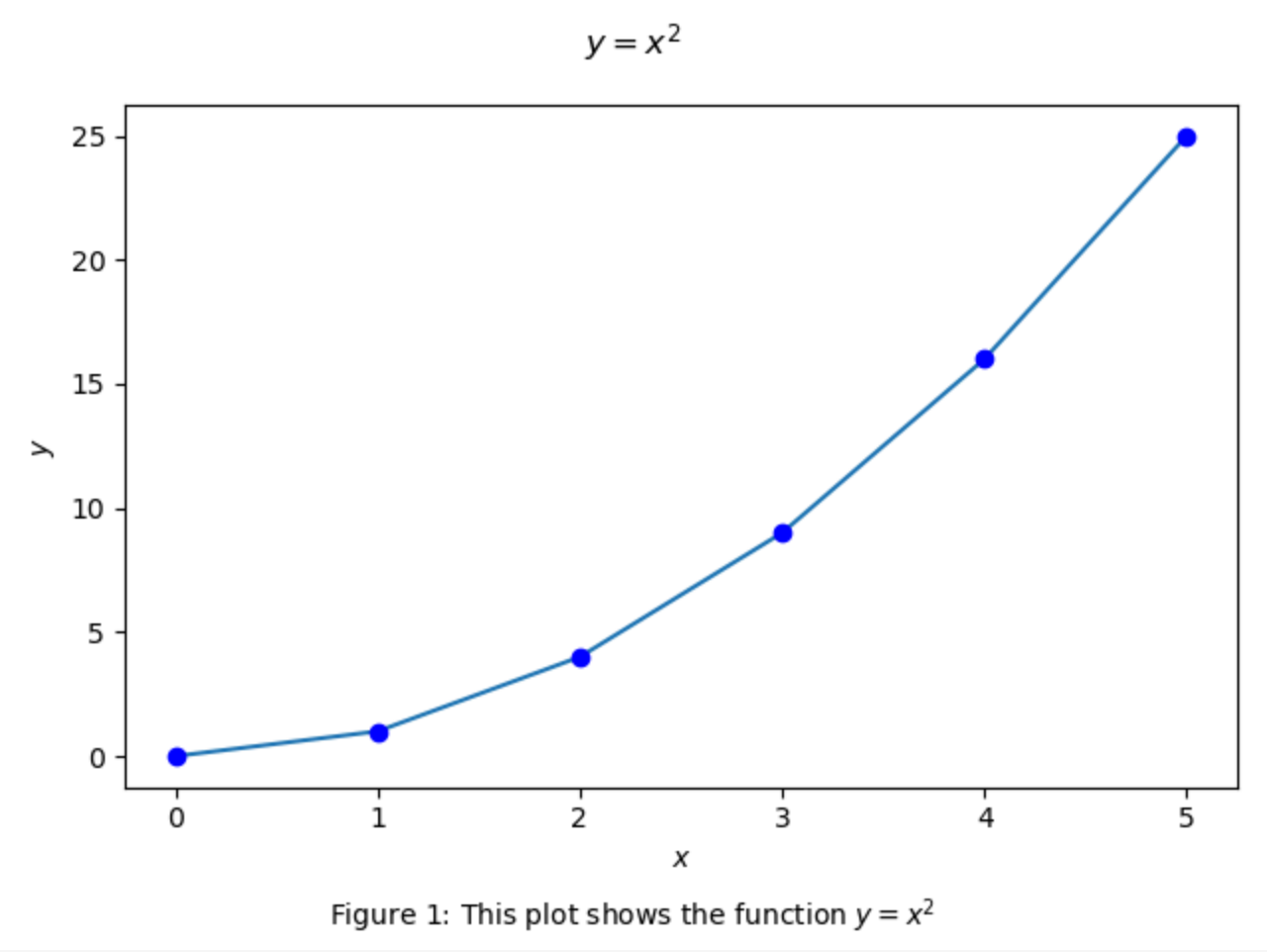
How figure coordinates work



Your first complete figure

```
import matplotlib.pyplot as plt

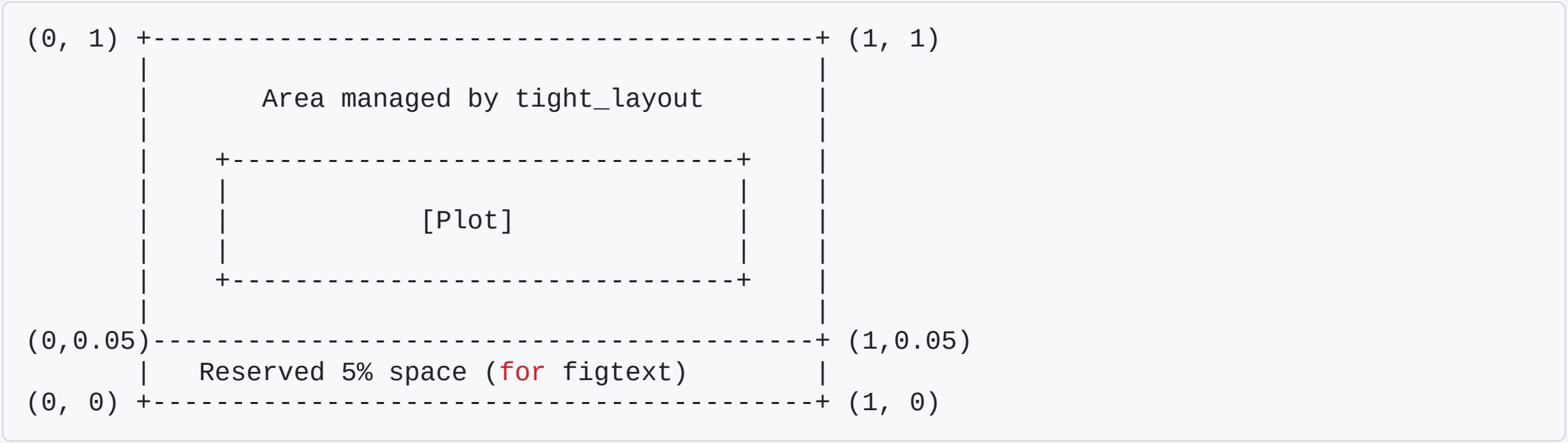
plt.plot([0, 1, 4, 9, 16, 25])
plt.plot([0, 1, 4, 9, 16, 25], 'bo')
plt.ylabel(r"$y$")
plt.xlabel(r'$x$')
plt.suptitle("$y=x^2$")
plt.figtext(
    0.5, 0.025,  # (x,y), coordinates of the anchor point
    r"Figure 1: This plot shows the function $y = x^2$",
    fontsize=10,
    ha="center", # text horizontal alignement relative to the anchor point (left by default)
)
# Resize graphic area on this rect, to leave room for caption
plt.tight_layout(rect=[0, 0.05, 1, 1])
plt.show()
```



Avoiding caption overlapping label axes

The rect parameter defines a bounding box (in figure coordinates from 0 to 1) inside which `tight_layout` is allowed to place the plot and its axes.

```
plt.tight_layout(rect=[0, 0.05, 1, 1])
```



About LaTeX expressions

To use *LaTeX notation* in Matplotlib (for axis labels, titles, and text), **we wrap our mathematical expressions in dollar signs** (`$...$`) inside *a raw string* (prefixed with `r` , like r `"$...$"`).

- Superscripts/Subscripts: `r"$x^2$"` , `r"$x_{min}$"` , `r"$x^n_{0}$` will be printed as x^2, x_{min} and x_0^n
- Fractions: `r"$\frac{a}{b}$"` will be printed as $\frac{a}{b}$
- Greek Letters: `r"$\alpha, \beta, \psi, \pi$"` will be printed as α, β, ψ, π
- Functions: `r"$y = \sin(kx)$"` will be printed as $y = \sin(kx)$

> The `r` prefix mean "raw string", just like f"..." means *formatted string*. Raw strings prevents Python from treating backslashes (\) as escape characters.

> Learn more about LaTeX formula syntax to improve the quality of your figures

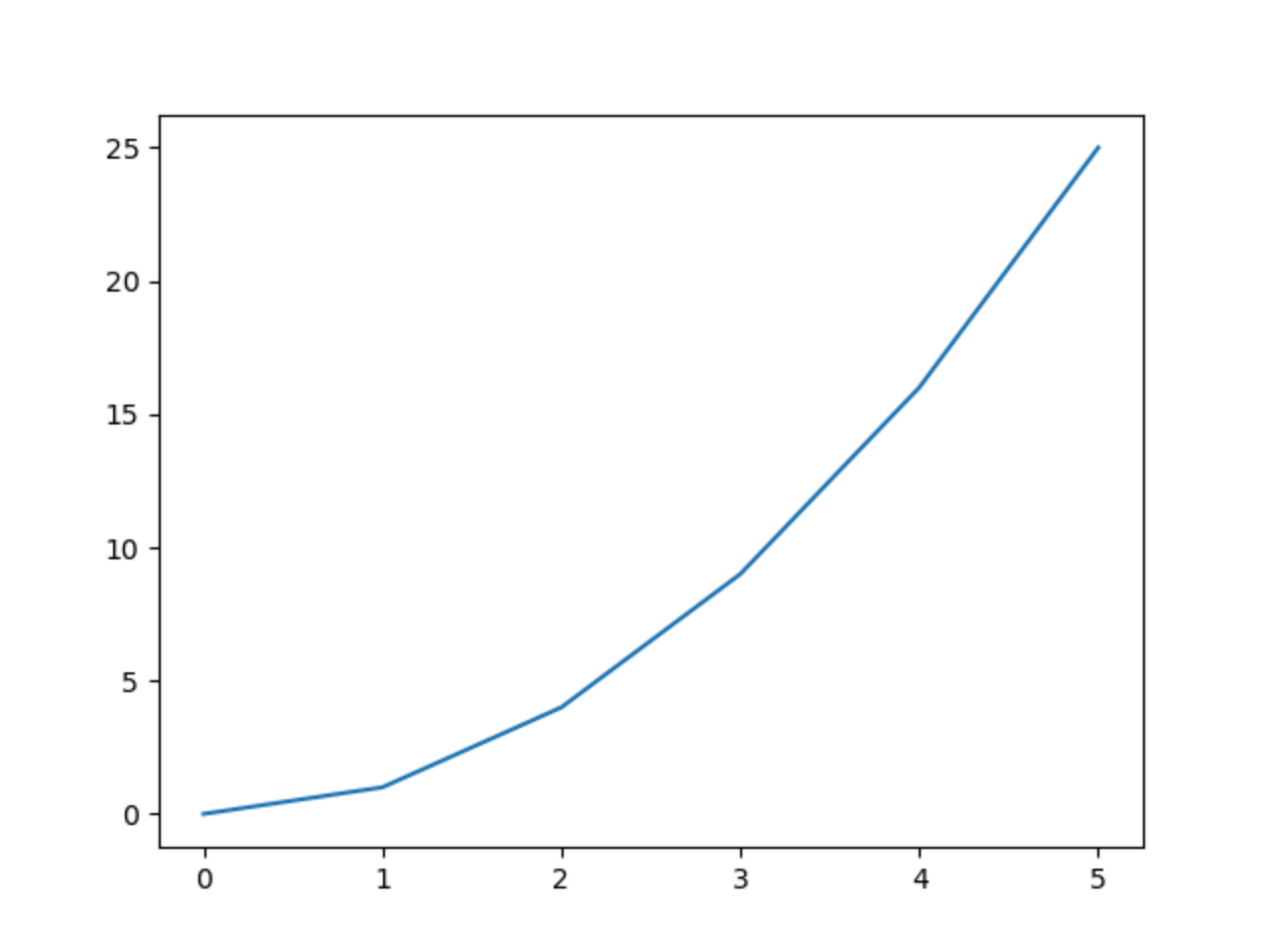
Line graph 1/3

When to use:

- Used for numeric data
- Used to show trends
- Compare two or more different variables over time
- Could be used to make *predictions* (*fit*)

Line graph 2/3

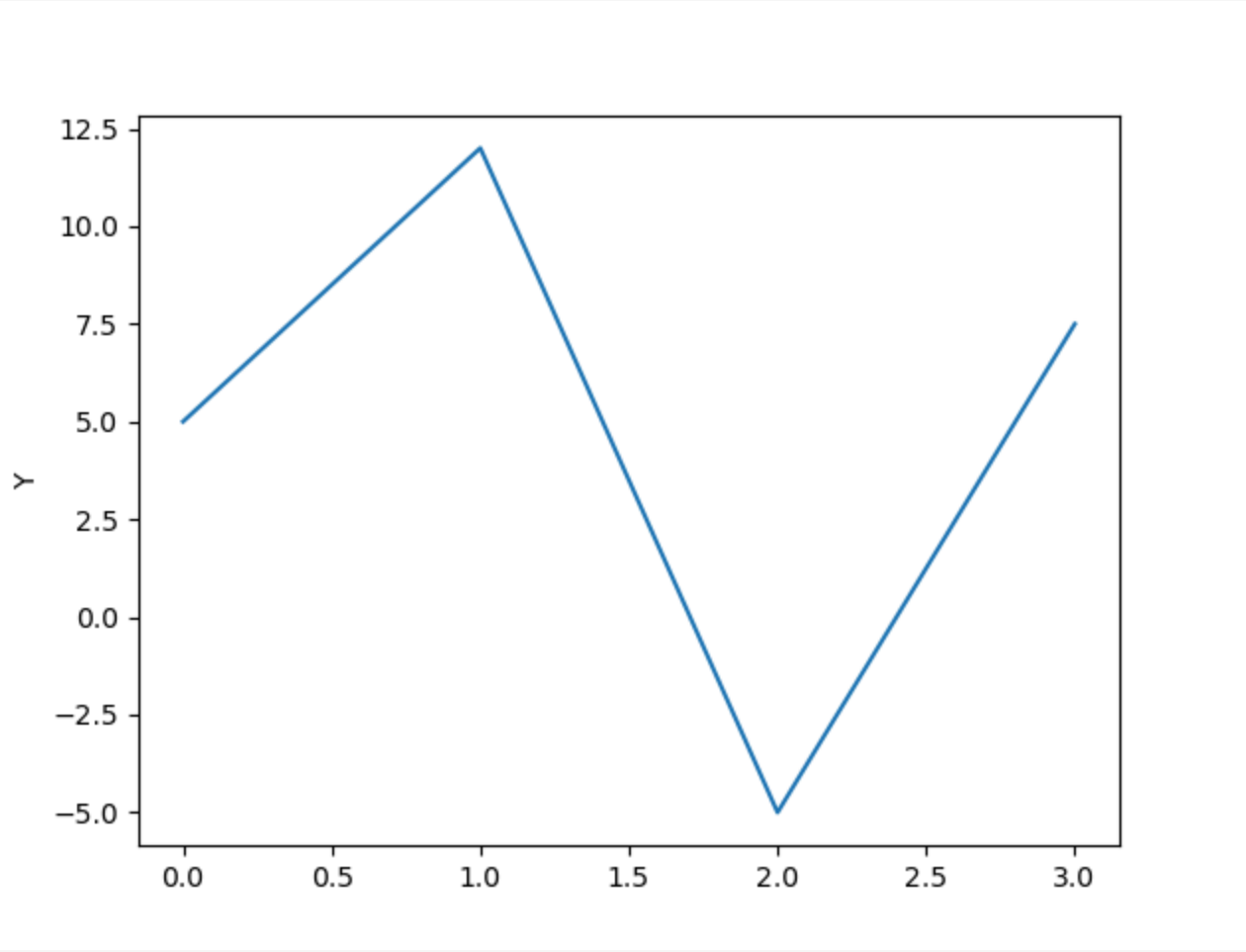
```
import matplotlib.pyplot as plt
import numpy as np
#create data
x_values = np.array([0, 1, 2, 3, 4, 5 ])
y_values = np.array([0, 1, 4, 9, 16,25])
#the default graph style for plot is a line
plt.plot(x_values, y_values)
#display the graph
plt.show()
```



Line graph 3/3

If you provide a single list or array to the `plot()` method, then `matplotlib` assumes it is a sequence of y values, and automatically generates the x values.

```
import matplotlib.pyplot as plt
import numpy as np
#Plot the array
plt.plot(np.array([1, 2, 3, 4]))
plt.ylabel('Y')
plt.show()
```



Adding further elements to the visualization 1/2

- As discussed earlier, every data visualization requires a *clear caption* to be a *complete*, *shareable*, and *publication-ready* figure.
- A complete caption clearly identifies:
  - *What* is shown
  - *Units* of measurement (axis labels)
  - *Why* it matters (key insights)

Without caption and labels, a figure is *meaningless*!

> While exploring data, captions are not "strictly" necessary (yet). However, they become *essential* once you are ready to publish or share your work (in a presentation, website, journal, etc.).

Adding further elements to the visualization 2/2

`pyplot` offers many methods to add information to your plot:

- `text()`: adds text in an *arbitrary* location
- `xlabel()`: adds text to the x-axis
- `ylabel()`: adds text to the y-axis
- `title()`: adds title to the plot
- `legend()`: shows a legend on the plot
- `savefig()`: **saves** your figure to a file on disk

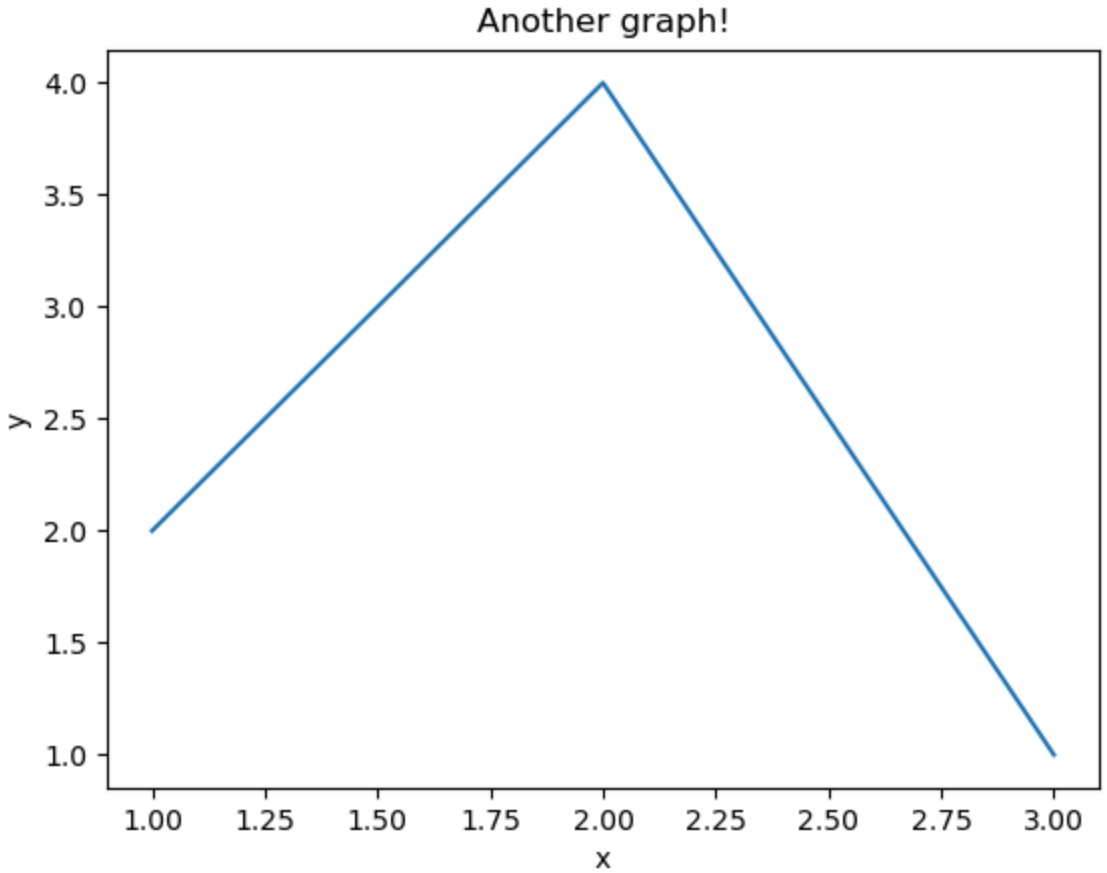
> Check the reference documentation

Practice

1. **Define** the x-axis (1, 2, 3) and corresponding y-axis values (2, 4, 1) as Ndarrays (provided by `numpy`)
2. **Plot** them on canvas using the `.plot()` method.
3. **Give** a **name** 'x' to x-axis **and** 'y' to y-axis using `.xlabel()` and `.ylabel()` methods
4. **Give** a **title** "Another graph!" to your plot using `.title()` method.
5. **View** your plot using `.show()` method.

Practice : Suggestion

```
# importing the required module
import matplotlib.pyplot as plt
import numpy as np
# x axis values
x = np.array([1,2,3])
# corresponding y axis values
y = np.array([2,4,1])
# plotting the points
plt.plot(x, y)
# naming the x axis
plt.xlabel('x')
# naming the y axis
plt.ylabel('y')
# giving a title to my graph
plt.title('Another graph!')
# showing the plot in an interactive window
plt.show()
```

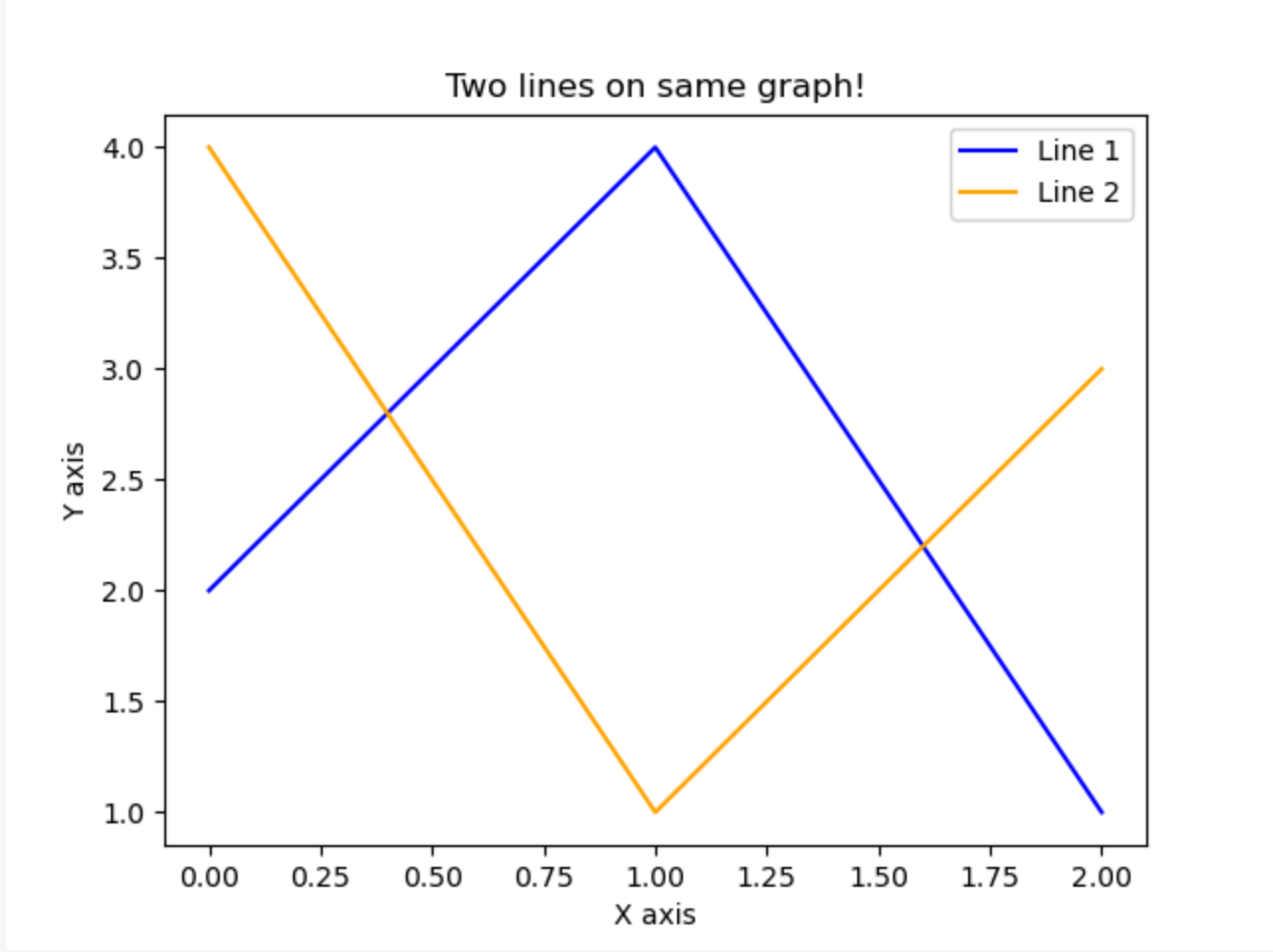


Two lines plot

- Here, we plot **two lines** on same graph. We differentiate between them by giving them a *name*(label) and a *color*
- The small rectangular box giving information about type of line and its color is called *legend*. We can add a *legend* to our plot using the `legend()` method.

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

y1 = np.array([2,4,1]) # first dataset
plt.plot(y1, label="Line 1", color= "blue")
y2 = np.array([4,1,3]) # second dataset
plt.plot(y2, label = "Line 2", color ="orange")
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.title('Two lines on same graph!')
plt.legend() # add a legend
plt.show()
```



Customization of plots

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

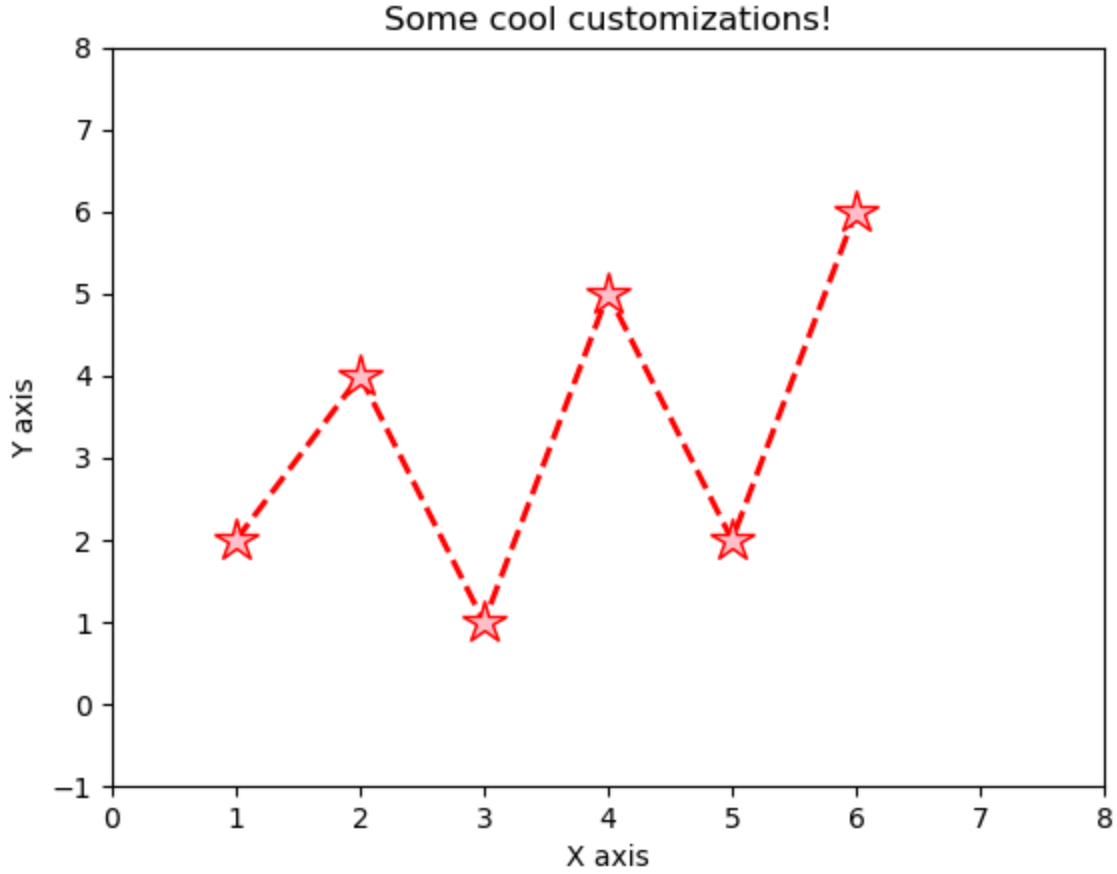
x = np.array([1,2,3,4,5,6])
y = np.array([2,4,1,5,2,6])

# plotting the points with custom styles
plt.plot(x, y, color='red', linestyle='dashed', linewidth = 2, marker='*',  markerfacecolor='pink', markersize=16)
# setting x and y axis range
plt.ylim(-1,8)
plt.xlim(0,8)

plt.xlabel('X axis')
plt.ylabel('Y axis')

plt.title('Some cool customizations!')
plt.show()
```

> Check the documentation for more customization options

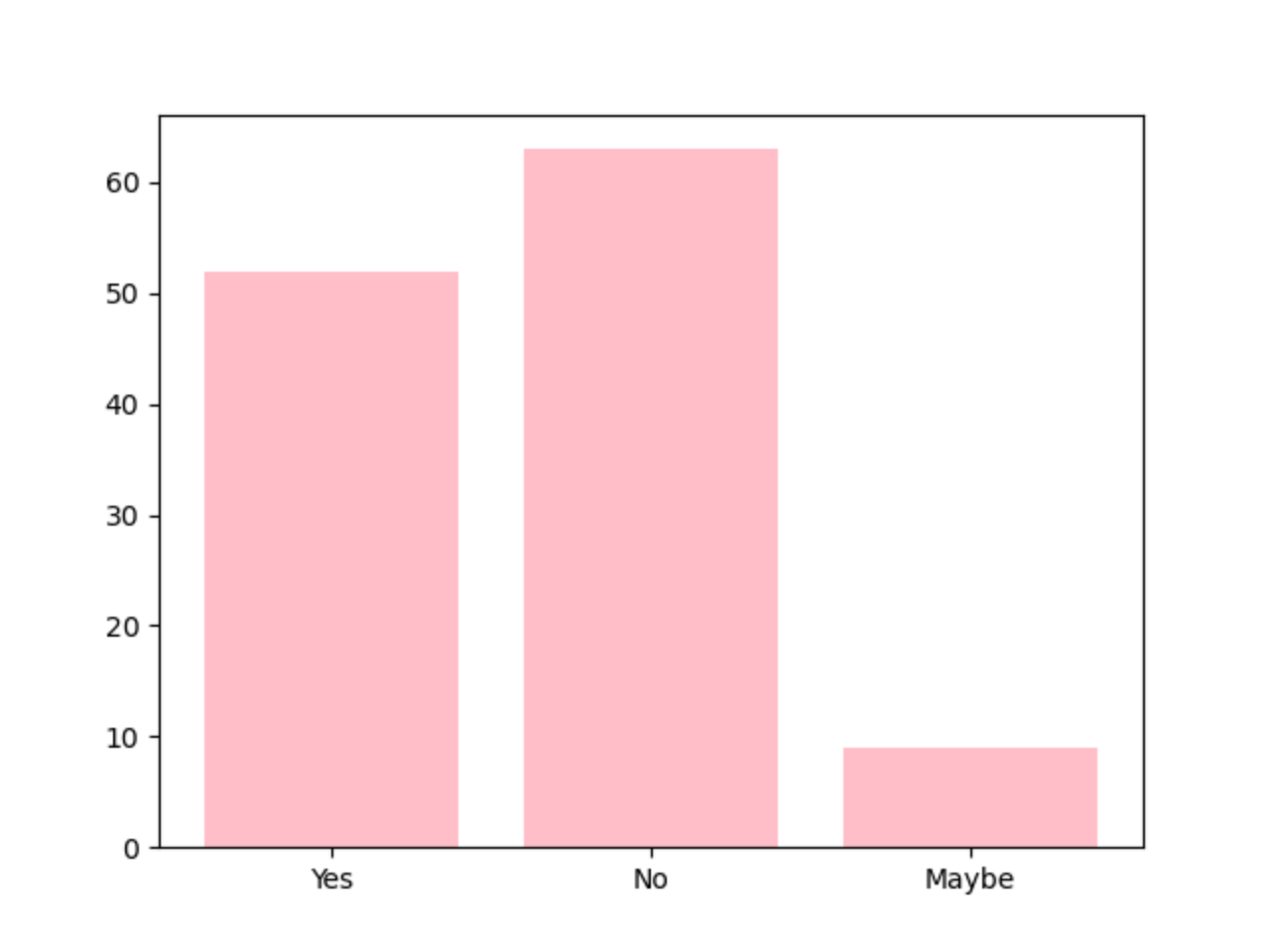


Bar graphs 1/3

When to use:

- Used for nominal or ordinal categories
- Compare data amongst different categories
- Ideal for more than three categories
- Can show large data changes over time

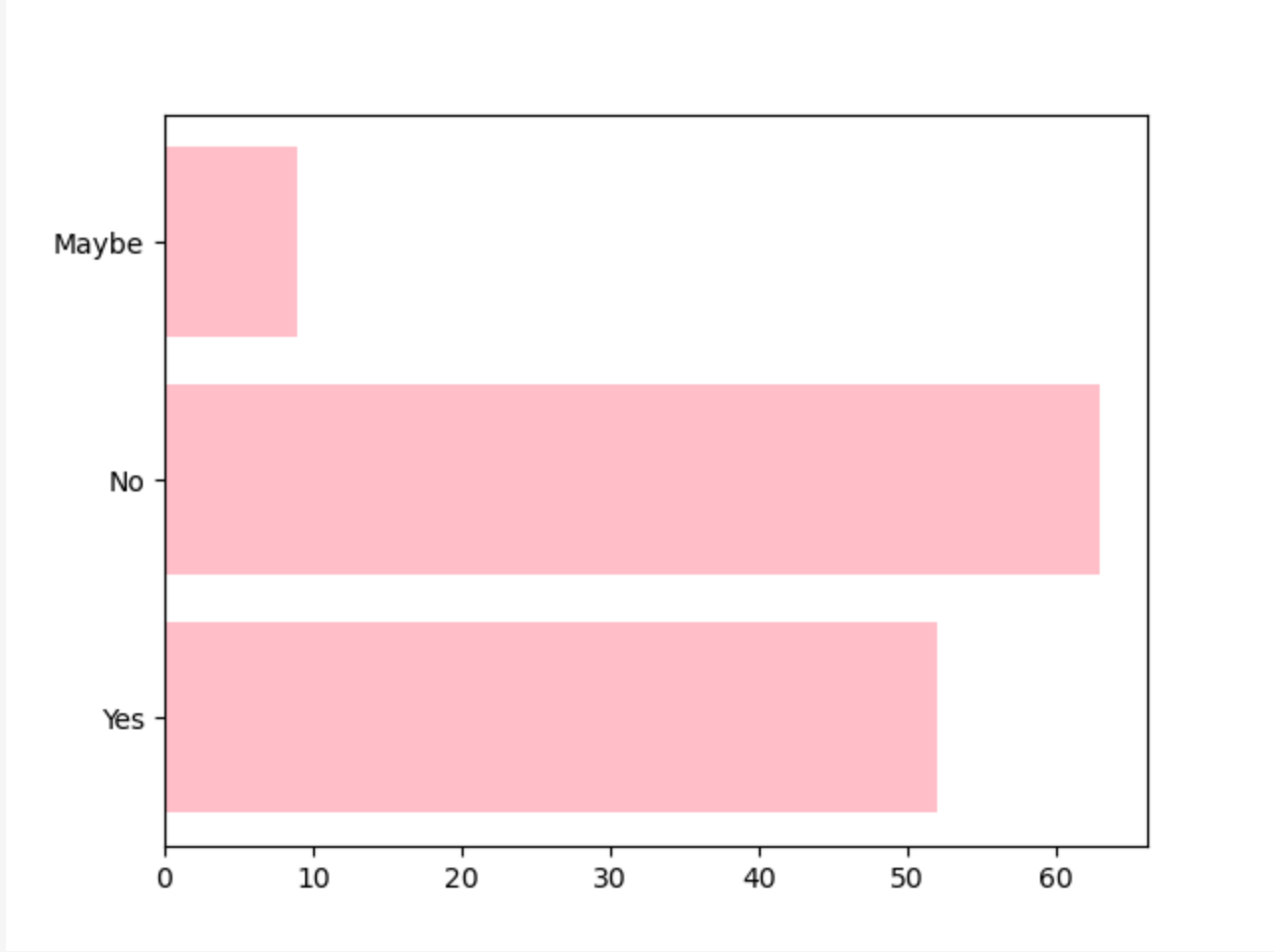
```
import matplotlib.pyplot as plt
import numpy as np
values = np.array([52, 63, 9])
names = np.array(["Yes", "No", "Maybe"])
plt.bar(names, values, color="pink") # .barh() method used now!
plt.show()
```



Bar graph 2/3

You can also flip the bar graph **horizontally** using the `barh()` method:

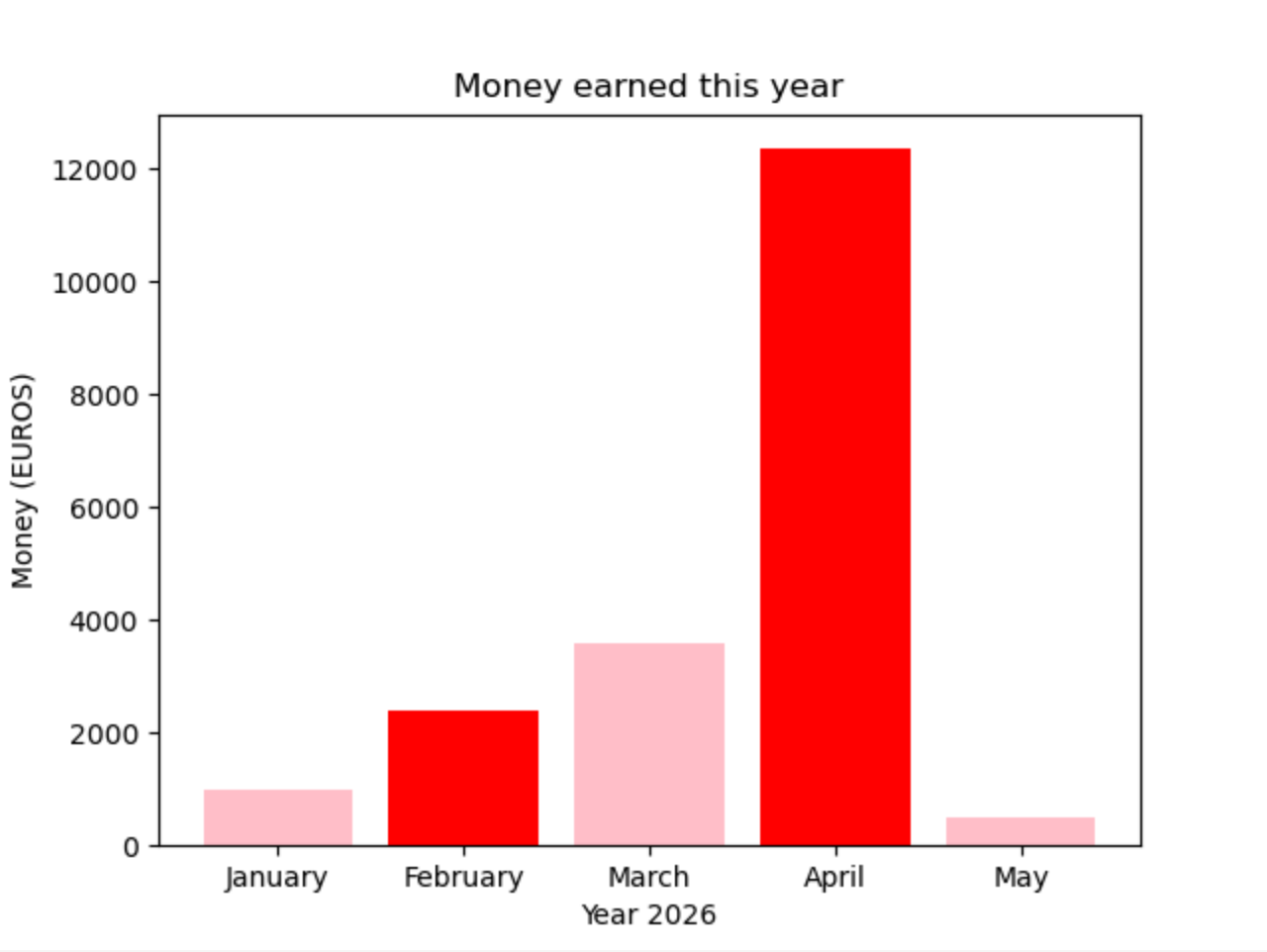
```
import matplotlib.pyplot as plt
import numpy as np
values = np.array([52, 63, 9])
names = np.array(["Yes", "No", "Maybe"])
plt.barh(names, values, color="pink") # .bar() method used now!
plt.show()
```



Bar graph 3/3

When you pass a list of colors to `plt.bar()`, matplotlib automatically cycles through that list, element by element:

```
import matplotlib.pyplot as plt
import numpy as np
money = np.array([1000, 2400, 3600, 12352, 500])
months=np.array(['January','February','March','April','May'])
colors =['pink', 'red']
plt.bar(months, money, width=0.8, color=colors)
plt.xlabel('Year 2026')
plt.ylabel('Money (EUROS)')
plt.title('Money earned this year')
plt.show()
```



Histogram 1/3

When to use:

- Plot **probability/frequency distributions**
- Compare multiple distributions
- Find an analytical expression to infere statistical properties of a dataset

Histogram 2/3

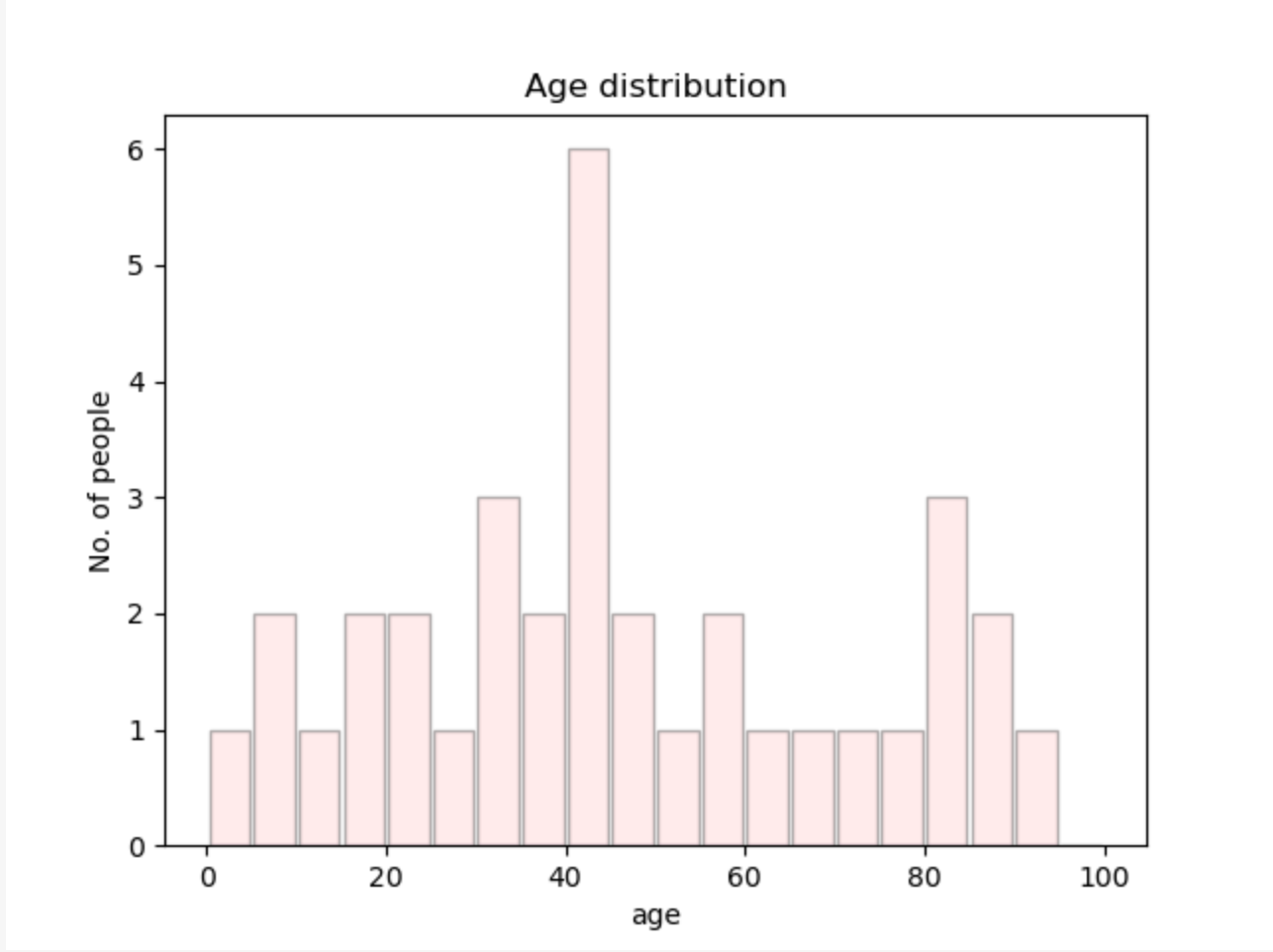
```
import matplotlib.pyplot as plt
import numpy as np

# Dataset
ages=np.array([2,5,70,40,30,45,50,45,43,40,44,60,
7,13,57,18,90,77,32,21,  27, 20,40, 37, 38,
67, 42, 33, 17, 55, 83, 81, 82, 87, 88])

# setting the ranges and no. of intervals
range = (0, 100)
bins = 20 # each bin (box) is 5 units wide

# plotting a histogram with plt.hist() method
plt.hist(ages, bins, range, color='pink', alpha= 0.3, histtype='bar',rwidth=0.9, edgecolor='black')

plt.xlabel('age')
plt.ylabel('No. of people')
plt.title('Age distribution')
plt.show()
```



Histogram 3/3

Looking at the code snippet, we added two new arguments:

```
plt.hist(ages, bins, range, color='pink', alpha= 0.3, histtype='bar',rwidth=0.9, edgecolor='black')
```

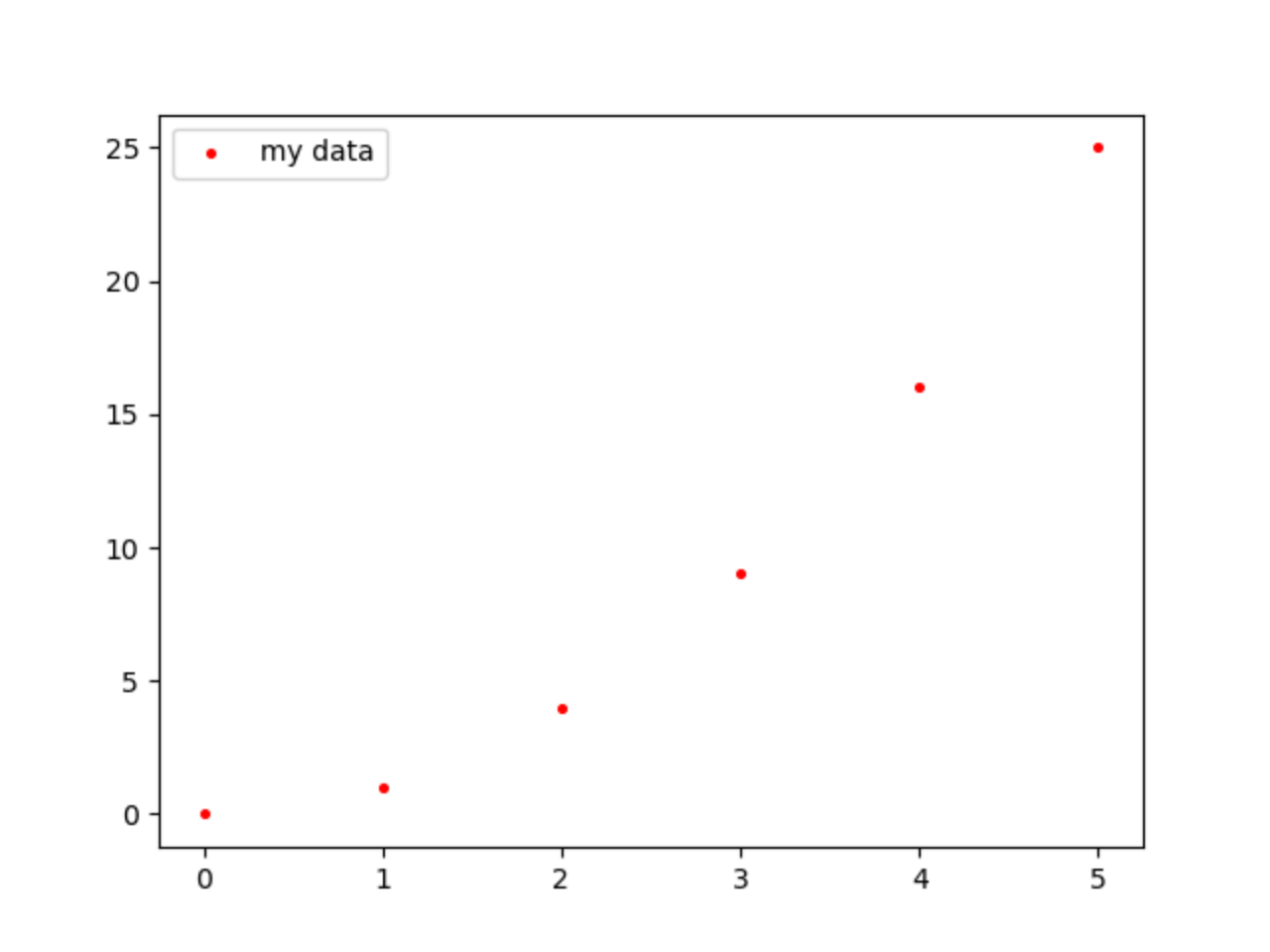
- `bins` is an argument specific to a histogram and allows the user to customize how many bins (boxes) he wants to distribute data
- `alpha`: is an argument that displays the level of *transparency* of the data points (0 is completely transparent, 1 completely opaque)

Scatter plot

When to use:

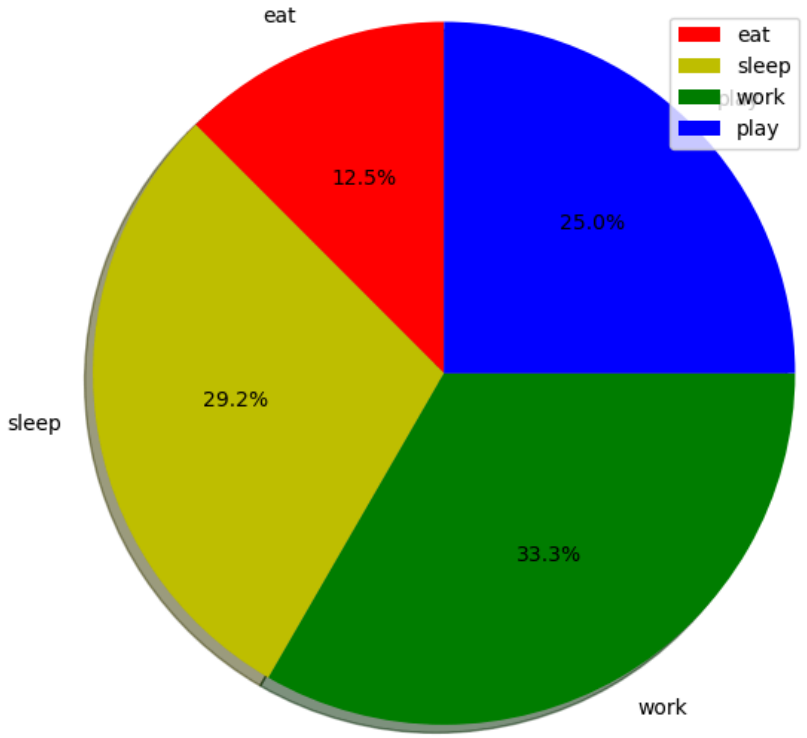
- Visualize relation between two numeric variables
- Visualize *correlation* in a large data set
- *Predict* behavior

```
import matplotlib.pyplot as plt
#create data for plotting
x_values = [0,1,2,3,4,5]
y_values = [0,1,4,9,16,25]
# use .scatter() method!
plt.scatter(x_values, y_values, s=7, label="my data", color="red")
plt.legend()
plt.show()
```



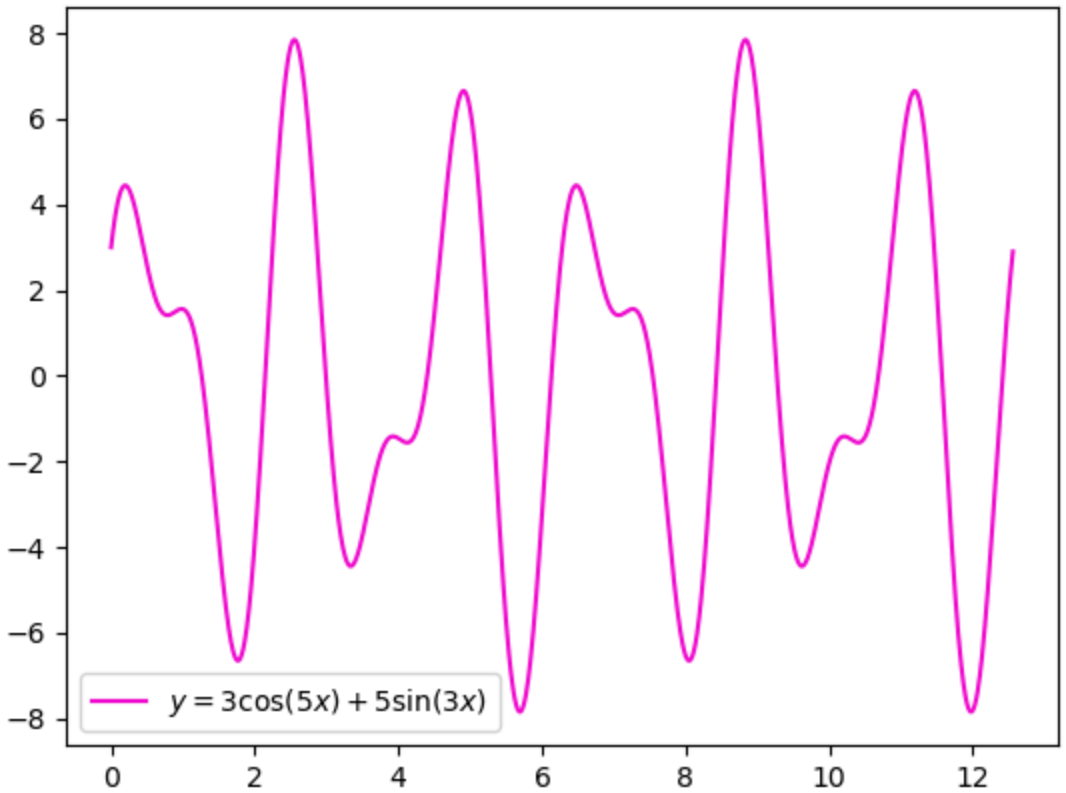
Pie-Chart

```
import matplotlib.pyplot as plt
# defining labels
activities = ['eat', 'sleep', 'work', 'play']
# portion covered by each label
slices = [3, 7, 8, 6]
# color for each label
colors = ['r', 'y', 'g', 'b']
# plotting the pie chart
plt.pie(slices, labels = activities, colors=colors,
startangle=90, shadow = True,
radius = 1.2, autopct = '%1.1f%%')
# plotting legend
plt.legend()
# showing the plot
plt.show()
```



Plotting curves of given equation

```
# importing the required modules
import matplotlib.pyplot as plt
import numpy as np
# setting the x - coordinates
x = np.arange(0, 4*(np.pi), 0.01)
# setting the corresponding y - coordinates
y = 3* np.cos(5 * x) + 5* np.sin(3 * x)
# potting the points
plt.plot(x, y, label=r'$y = 3\cos(5x) + 5\sin(3x)$', color="#f00ad1")
plt.legend()
# function to show the plot
plt.show()
```



Sub-plots 1/3

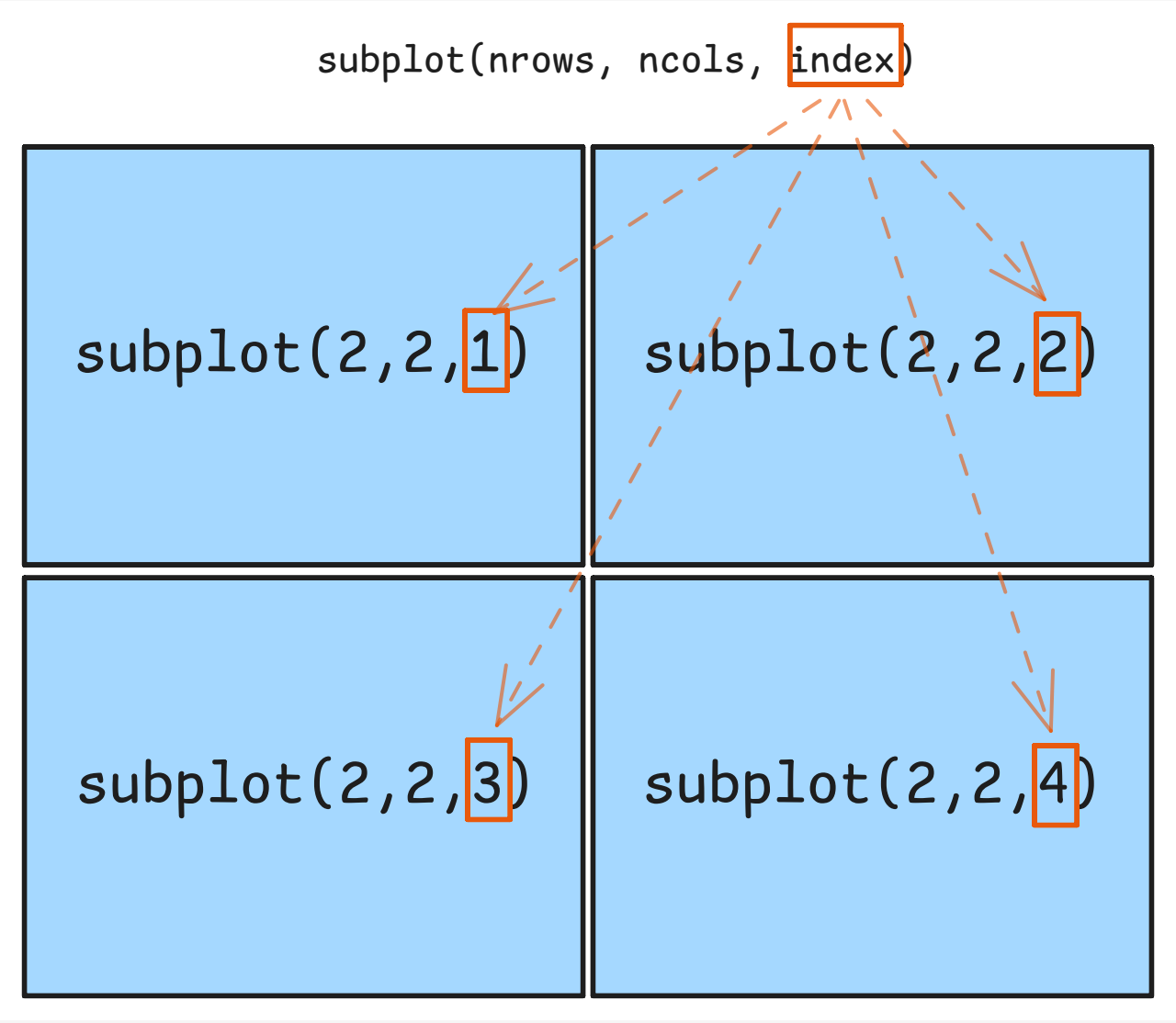
- Subplots are required when we want to **show two or more plots in the same figure**.
- The subplot function returns the axes object at a given grid position. The call signature of this function is:

```
plt.subplot(nrows, ncols, index)
```

This makes a subplot and a grid (`nrows * ncols`) where we can place a plot at a given position (index)

Sub-plots 2/3

- Indexes run from 1 to `nrows * ncols` in **row-major order**.
- If all three values are single digits (< 10), they can be combined into a **single three-digit number** (e.g., `plt.subplot(2, 2, 1)` is identical to `plt.subplot(221)`).



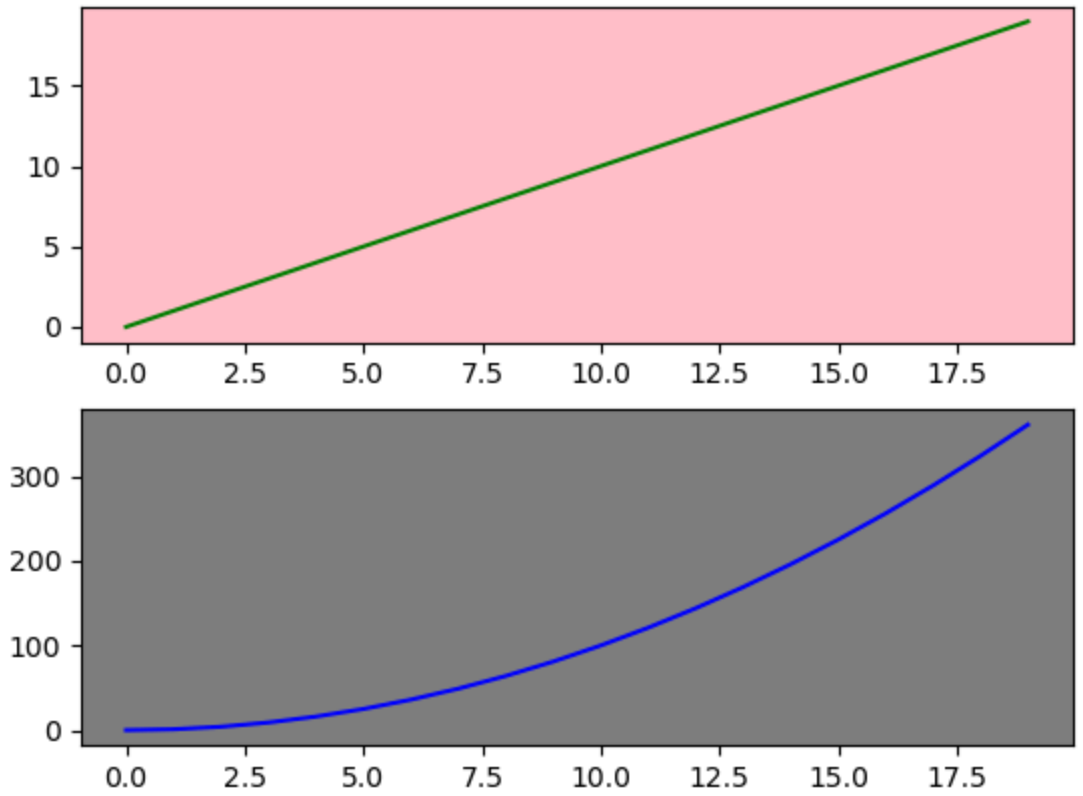
Sub-plots 3/3

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

# Create data
x=np.arange (20)
y1=x
y2=x**2

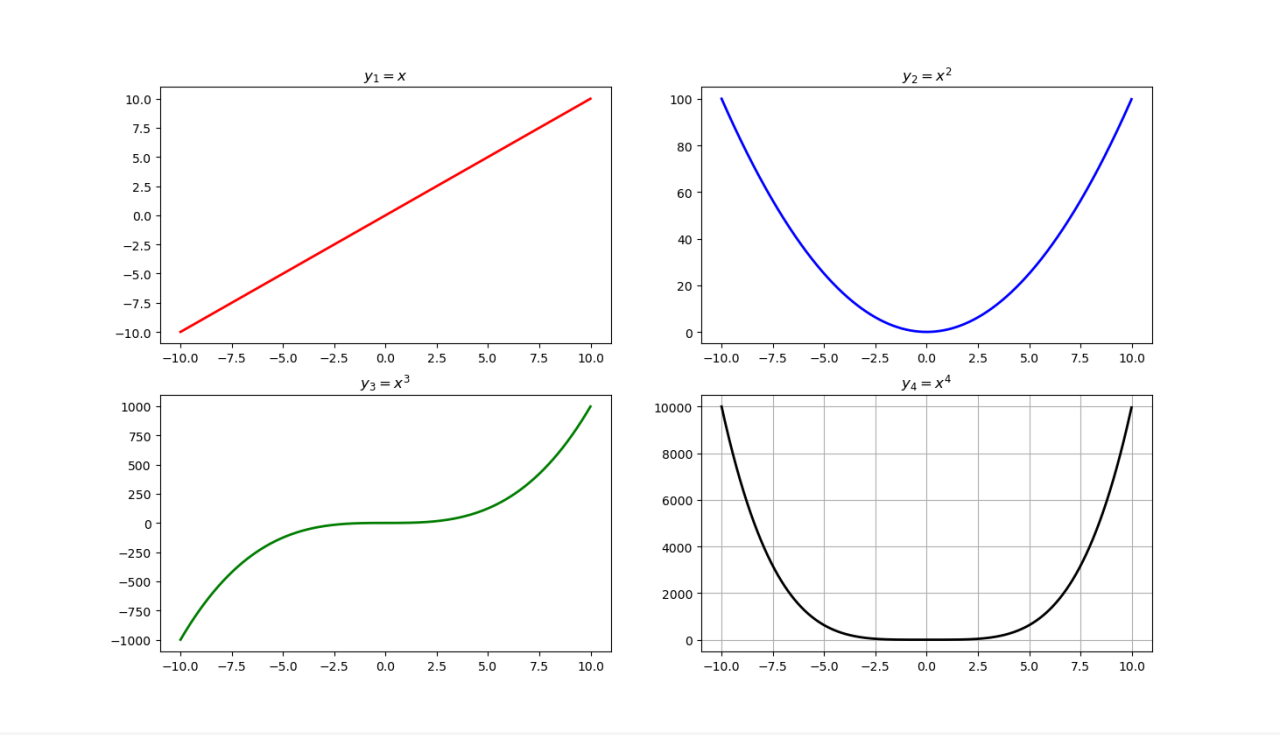
# create a subplot of a grid with 2 rows and 1 column.
plt.subplot(211, facecolor='pink')
plt.plot(x, y1, color='green')

# creates 2nd subplot with yellow background
plt.subplot(212, facecolor='grey')
plt.plot(x, y2, color='blue')
plt.show ()
```



Practice: Subplots

1. **Create** x between -10 and 10 with step of 0.01
2. **Create**:
   i. $y_1 = x$
   ii. $y_2 = x^2$
   iii. $y_3 = x^3$
   iv. $y_4 = x^4$
3. **Create** four subplots
4. **Plot** your graphics with the *same colors* shown in the output



Data visualization, useful references

To learn more about creating *good* visualizations, check the useful references on the repository

Practice on DataCamp

Continue to work on the interactive courses on DataCamp:

- Course: Introduction to Python:
    - Chapter 4: *Numpy*
- Course: Introduction to Data Visualization with Matplotlib
    - Chapter 1: Introduction to Matplotlib
    - Chapter 2: Plotting time-series

> See all asignments here (reminder)

Next (and last) module

Go to next module