



AI4D LAB TRAINING

[Zephania Reuben \(https://nsoma.me\)](https://nsoma.me)

July 17, 2023

DATA SCIENCE | MATPLOTLIB

OUTLINES:

- Introduction to matplotlib
- Getting started with matplotlib
- Decorate Graphs with Plot Styles and Types

1.0 Introduction to Matplotlib

What is Matplotlib?

- **Matplotlib** is a Python package for 2D plotting that generates production-quality graphs. It supports interactive and non-interactive plotting, and can save images in several output formats (PNG, PS, and others).
- It provides a wide variety of plot types (lines, bars, pie charts, histograms, and many more). In addition to this, it is highly customizable, flexible, and easy to use.

- The dual nature of Matplotlib allows it to be used in both interactive and non-interactive scripts. It can be used in scripts without a graphical display, embedded in graphical applications, or on web pages. It can also be used interactively with the Python interpreter or IPython.
- **Matplotlib** was born in the scientific area of computing, where **gnuplot** and **MATLAB** were (and still are) used a lot.
- **Matplotlib** was modeled on **MATLAB**, because graphing was something that MATLAB did very well. The high degree of compatibility between them made many people move from **MATLAB** to **Matplotlib**.

But what are the points that built the success of Matplotlib?

Let's look at some of them:

- **It uses Python:**
 - Python is a very interesting language for scientific purposes (it's interpreted, high-level, easy to learn, easily extensible, and has a powerful standard library) and is now used by major institutions such as NASA, JPL, Google, DreamWorks, Disney, and many more.
- **It's open source, so no license to pay:**
 - This makes it very appealing for professors and students, who often have a low budget.
- **It's very customizable and extensible:**
 - Matplotlib can fit every use case because it has a lot of graph types, features, and configuration options.
- **It's cross-platform and portable:**
 - Matplotlib can run on Linux, Windows, Mac OS X, and Sun Solaris (and Python can run on almost every architecture available).

About dependencies

- Matplotlib has its origin in scientific fields, so it is commonly used to plot huge datasets. Python's native support for long lists becomes impractical for such sizes, so Matplotlib needs better support for arrays, so NumPy has to be available to use Matplotlib.
- **Build dependencies**
 - **Python:** Version (≥ 3.6)
 - **NumPy:** Version (≥ 1.11)
 - **libpng:** Version 1.2 or higher is needed to load or save PNG images (Windows users can skip this requirement)
 - **FreeType:** Version 2.3 or higher is needed for reading TrueType font files (Windows users can skip this requirement)

Installing Matplotlib

- **Installing Matplotlib on Linux**
 - In the following table, we will present some of the common Linux distributions package names for Matplotlib and the tools we can use to install the package:

Distribution	Package Name
Debian or Ubuntu (And other Debian derivatives)	<code>sudo apt-get install python3-matplotlib</code>
Fedora	<code>sudo dnf install python3-matplotlib</code>
Red hat	<code>sudo yum install python3-matplotlib</code>
Arch	<code>sudo pacman -S python-matplotlib</code>

- **Installing Matplotlib on Windows**

- Open command line interface (CMD) use the next command to intall **Matplotlib**
- `python -m pip install matplotlib`

Testing our installation

- To ensure we have correctly installed Matplotlib and its dependencies, a very simple test can be carried out in the following manner:

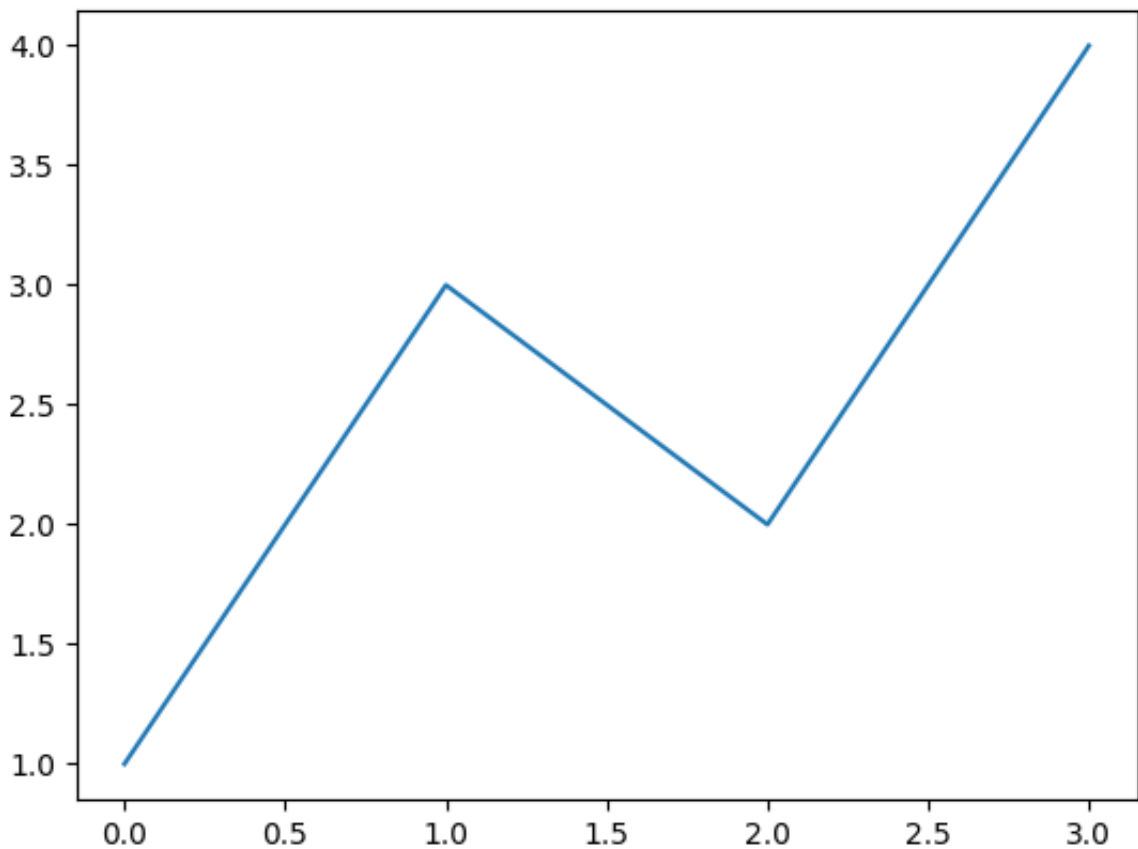
```
In [ ]: import matplotlib
print("Matplotlib version : ", matplotlib.__version__)
import numpy
print("Numpy version : ", numpy.__version__)
```

2.0 Getting started with Matplotlib

First plots with Matplotlib

```
In [ ]: import seaborn
seaborn.set_style('whitegrid')
```

```
In [3]: import matplotlib.pyplot as plt
plt.plot([1, 3, 2, 4])
plt.show()
```



Let's look at each line of the previous code in detail:

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

- This is the preferred format to import the main Matplotlib submodule for plotting, pyplot. It's the best practice and in order to avoid pollution of the global namespace, it's strongly encouraged to never import like:

```
from <module> import *
```

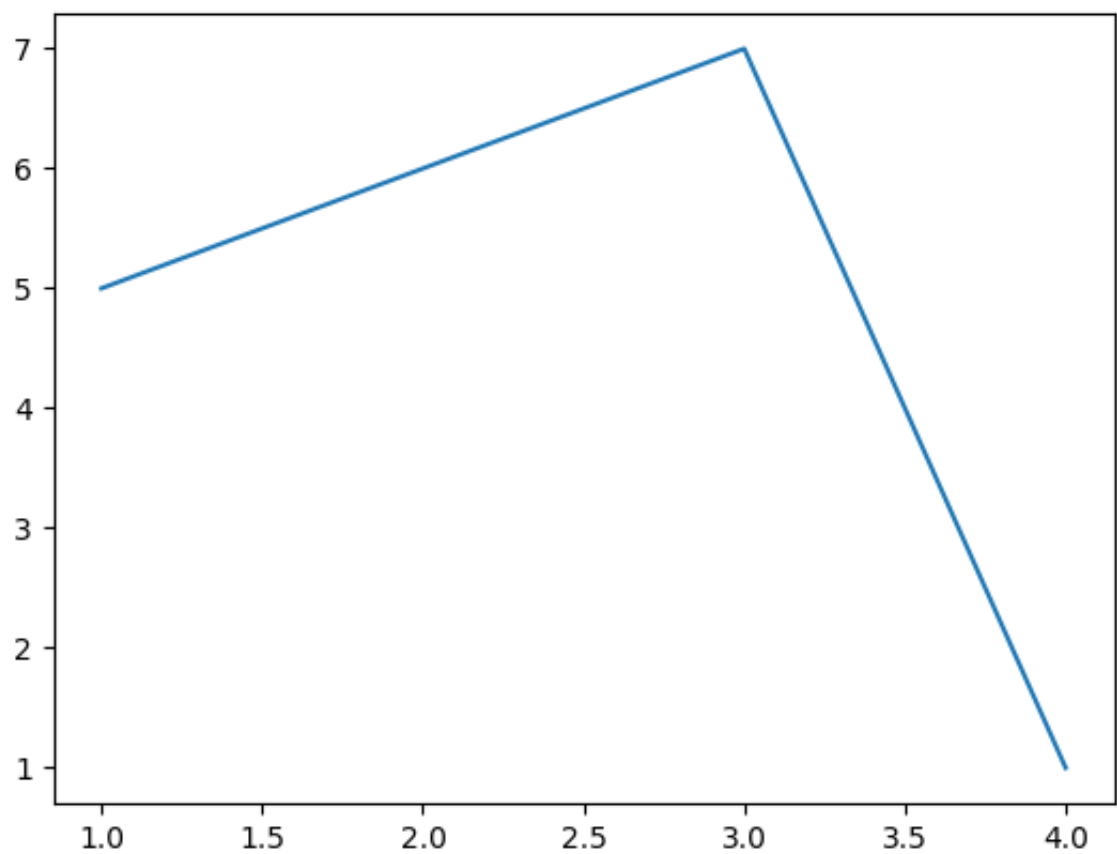
```
[2]: plt.plot([1, 3, 2, 4])
```

- This code line is the actual plotting command. We have specified only a list of values that represent the vertical coordinates of the points to be plotted. Matplotlib will use an implicit horizontal values list, from 0 (the first value) to N-1 (where N is the number of items in the list).

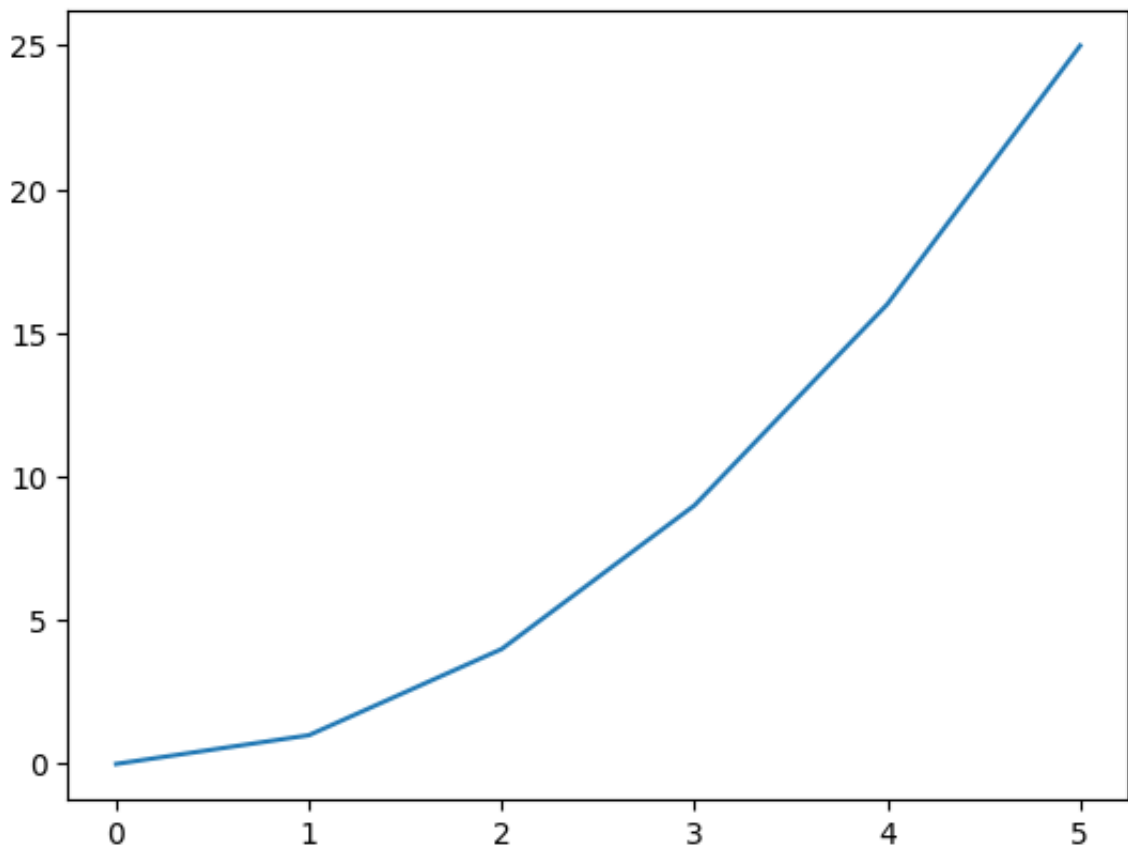
```
[3]: plt.show()
```

- This command actually opens the window containing the plot image.
- Of course, we can also explicitly specify both the lists:

```
In [4]: plt.plot([1,2,3,4],[5,6,7,1])  
plt.show()
```

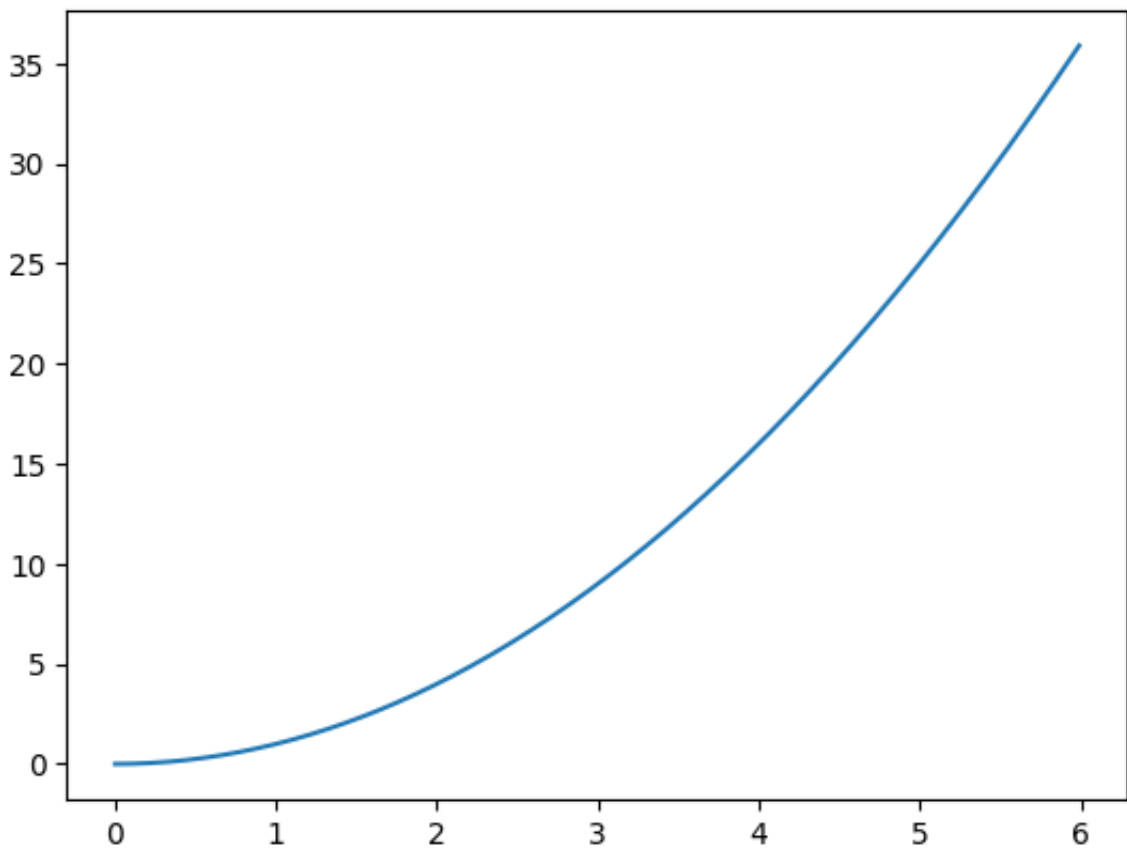


```
In [5]: import matplotlib.pyplot as plt
x = range(6)
# y = [0, 1, 4, 9, 16, 25]
plt.plot(x, [xi**2 for xi in x])
plt.show()
```



- we can start introducing the interaction with NumPy with one of its most used functions, `arange()`, and highlighting the difference with `range()`:
 - `range(i, j, k)` is a Python built-in function that generates a sequence of integers from `i` to `j` with an increment of `k` (both, the initial value and the step are optional).
 - `arange(x, y, z)` is a part of NumPy, and it generates a sequence of elements (with data type determined by parameter types) from `x` to `y` with a spacing `z` (with the same optional parameters as that of the previous function).
- Let's use `arange()`

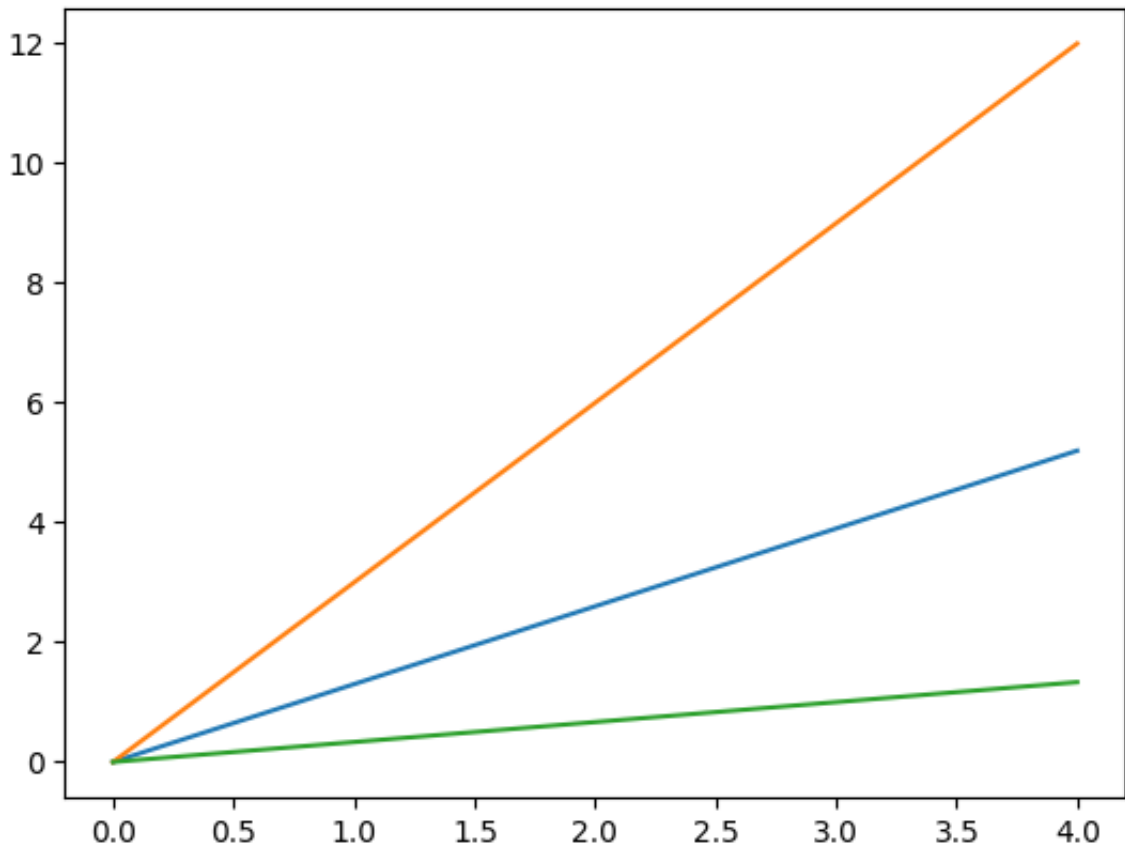
```
In [6]: import matplotlib.pyplot as plt
import numpy as np
x = np.arange(0.0, 6.0, 0.01)
plt.plot(x, [i**2 for i in x])
plt.show()
```



Multiline plots

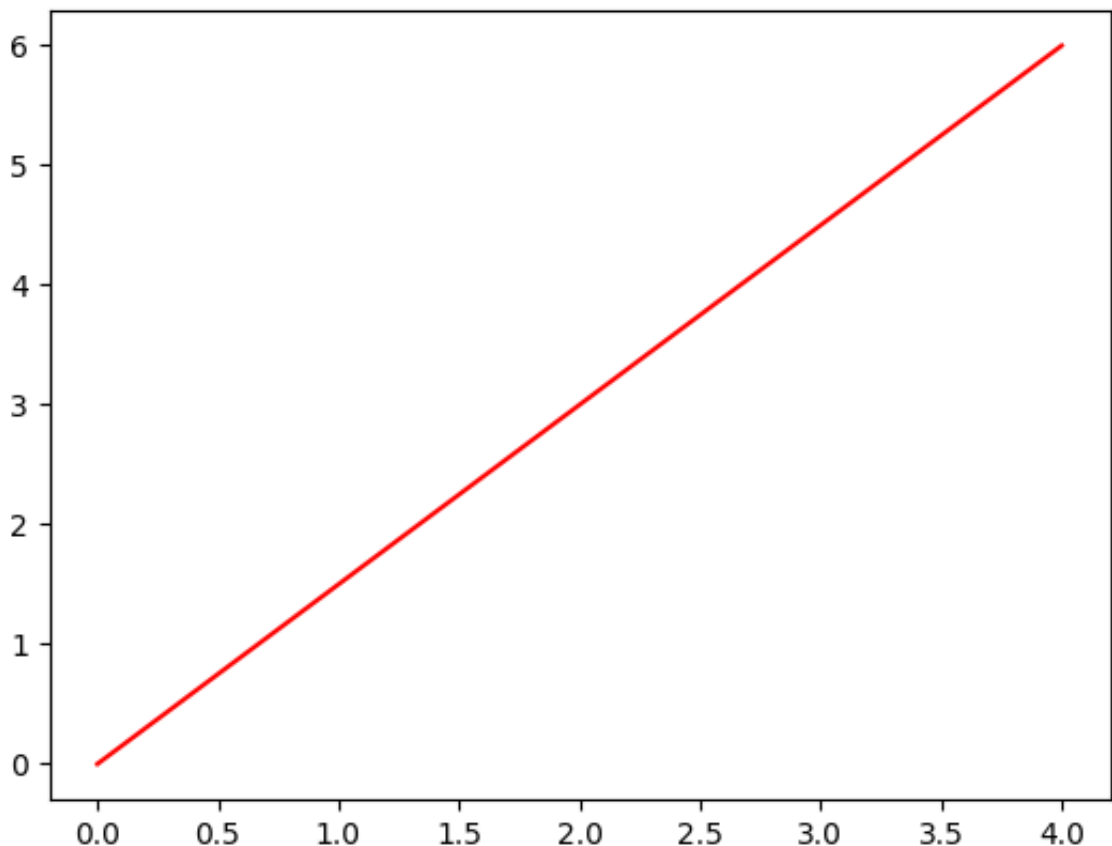
- It's fun to plot a line, but it's even more fun when we can plot more than one line on the same figure. This is really easy with Matplotlib as we can simply plot all the lines that we want before calling `show()`.

```
In [7]: import matplotlib.pyplot as plt
import numpy as np
x = np.arange(5)
plt.plot(x,[y*1.3 for y in x])
plt.plot(x,[y*3 for y in x])
plt.plot(x,[y/3 for y in x])
plt.show()
```



- **Note** how Matplotlib automatically chooses different colors for each line—green for the first line, blue for the second line, and red for the third one (from bottom to top).
- `plot()` supports another syntax useful in this situation. We can plot multiline figures by passing the X and Y values list in a single `plot()` call:


```
In [16]: import matplotlib.pyplot as plt
x = range(5)
plt.plot(x, [xi*1.5 for xi in x], 'r')
plt.grid(False)
plt.show()
```

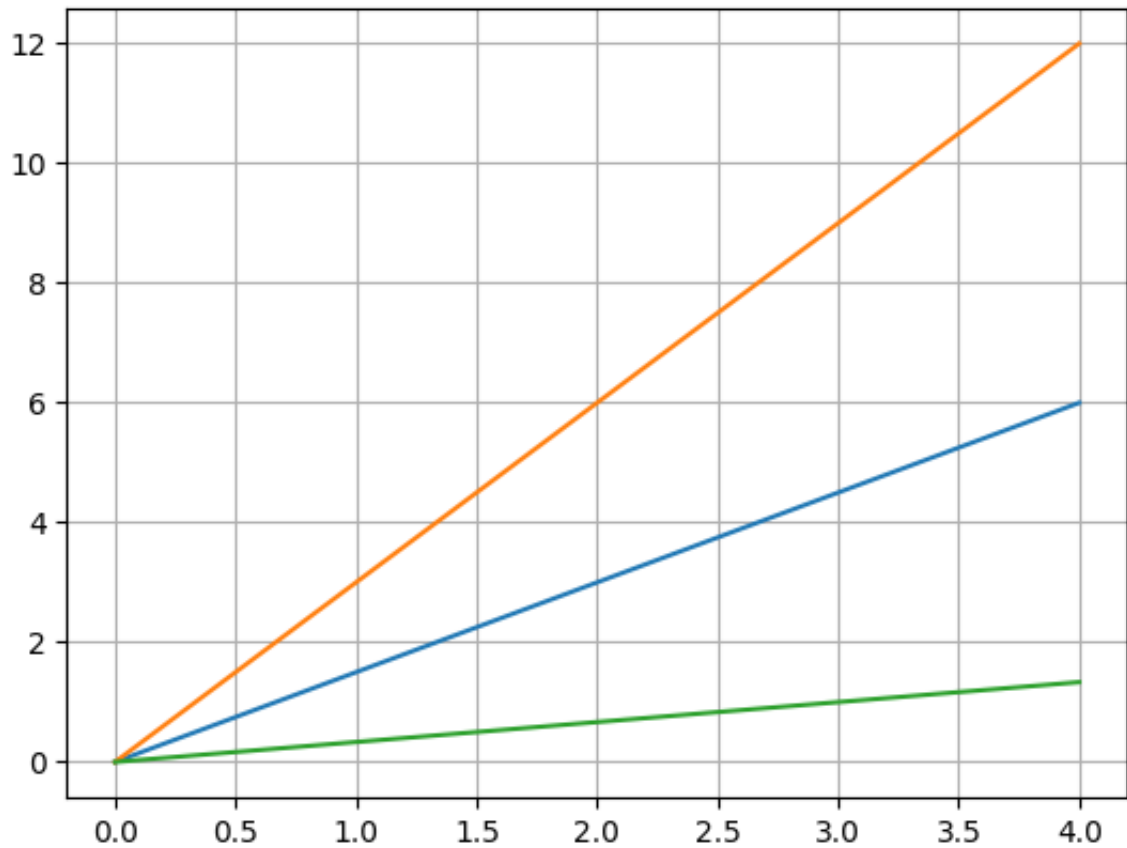


Grid, axes, and labels

- **Adding grid**

- In the previous images, we saw that the background of the figure was completely blank.
- While it might be nice for some plots, there are situations where having a reference system would improve the comprehension of the plot—for example with multiline plots. We can add a grid to the plot by calling the `grid()` function; -it takes one parameter, a Boolean value, to enable (if True) or disable (if False) the grid:

```
In [17]: import matplotlib.pyplot as plt
import numpy as np
x = np.arange(5)
plt.plot(x, x*1.5, x, x*3.0, x, x/3.0)
In [5]: plt.grid(True)
In [6]: plt.show()
```

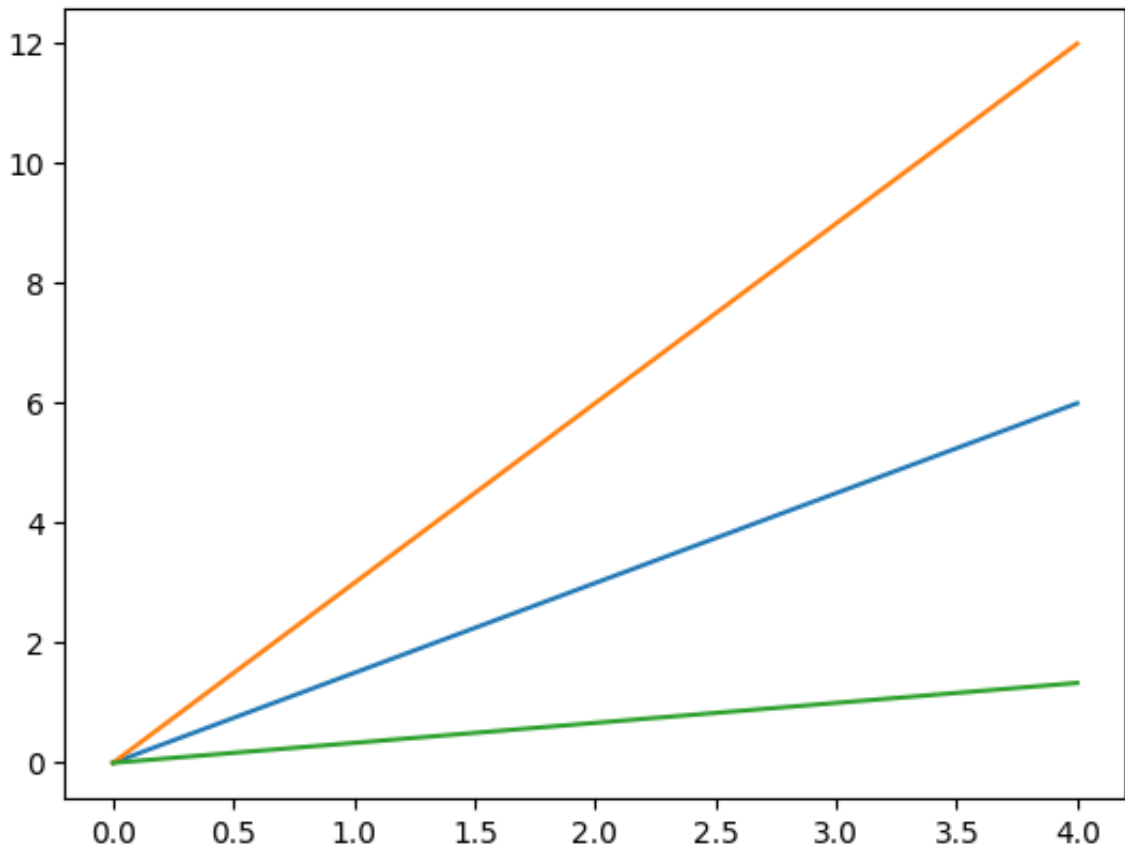


• Handling axes

- You might have noticed that Matplotlib automatically sets the limits of the figure to precisely contain the plotted datasets.
- However, sometimes we want to set the axes limits ourself (defining the scale of the chart). Let's take the first multiline plot.
- Wouldn't it be better to have more spaces between lines and borders? We can achieve this with the following code:

```
In [18]: import matplotlib.pyplot as plt
import numpy as np
x = np.arange(5)
plt.plot(x, x*1.5, x, x*3.0, x, x/3.0)
plt.axis() # shows the current axis limits values
```

Out[18]: (-0.2, 4.2, -0.6000000000000001, 12.6)



```
In [ ]: import matplotlib.pyplot as plt
import numpy as np
x = np.arange(5)
plt.plot(x, x*1.5, x, x*3.0, x, x/3.0)
plt.axis([0,5, -1, 13]) # set new axes limits
plt.show()
```

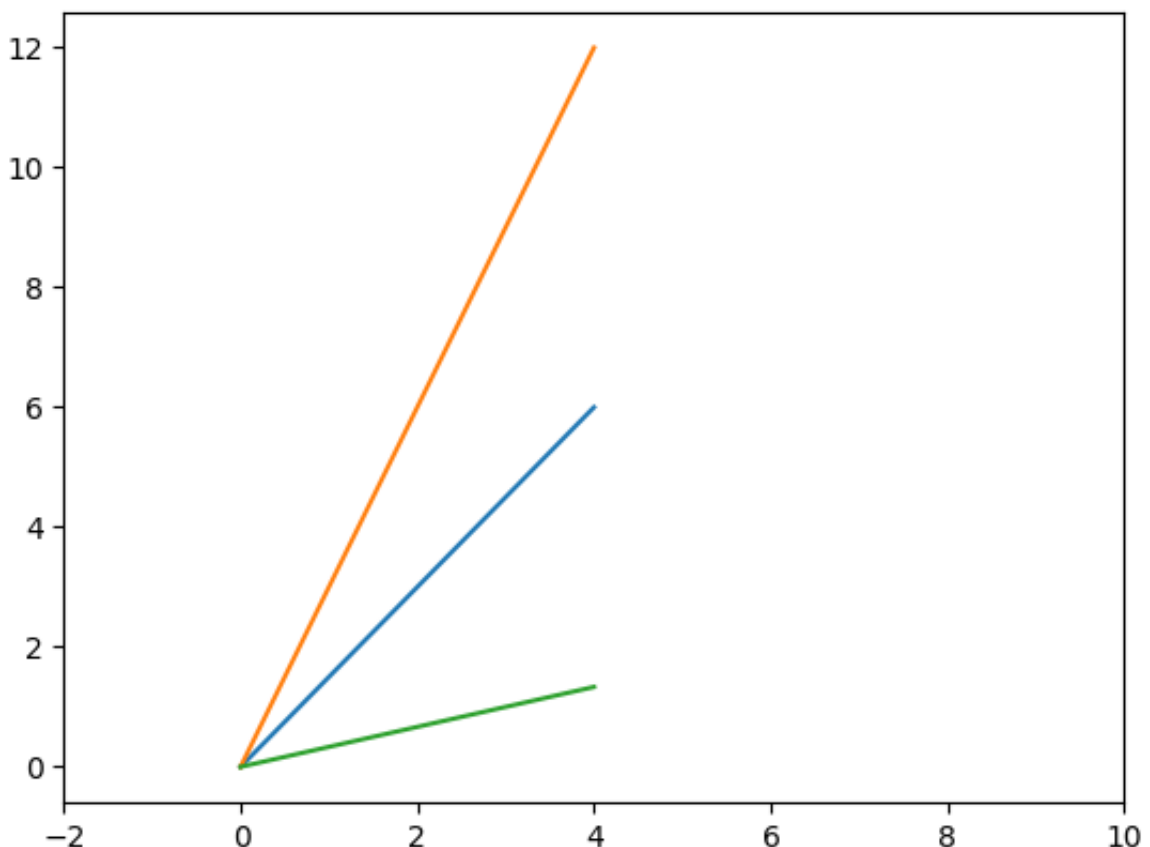
- If we execute `axis()` without parameters, it returns the actual axis limits. There are two ways to pass parameters to `axis()`: by a list of four values or by keyword arguments.
- The list of values, that's the whole set of four values of keyword arguments [`xmin`, `xmax`, `ymin`, `ymax`], allows us to specify at the same time, the minimum and maximum limits respectively for the X-axis and the Y-axis. We can use the specific keyword arguments, for example:

```
python
plt.axis(xmin=NNN, ymax=NNN)
```

- If we wish to set only some of these limits (in the previous line, we set only the minimum value for X-axis and the maximum value for Y-axis).
- We can also control the limits for each axis separately using `xlim()` and `ylim()` functions. Let's take the previous code before calling the `axis()` function, and change it in the following way:

```
In [20]: import matplotlib.pyplot as plt
import numpy as np
x = np.arange(5)
plt.plot(x, x*1.5, x, x*3.0, x, x/3.0)
plt.xlim([-2,10])
print(plt.xlim())
print(plt.ylim())
```

```
(-2.0, 10.0)
(-0.6000000000000001, 12.6)
```

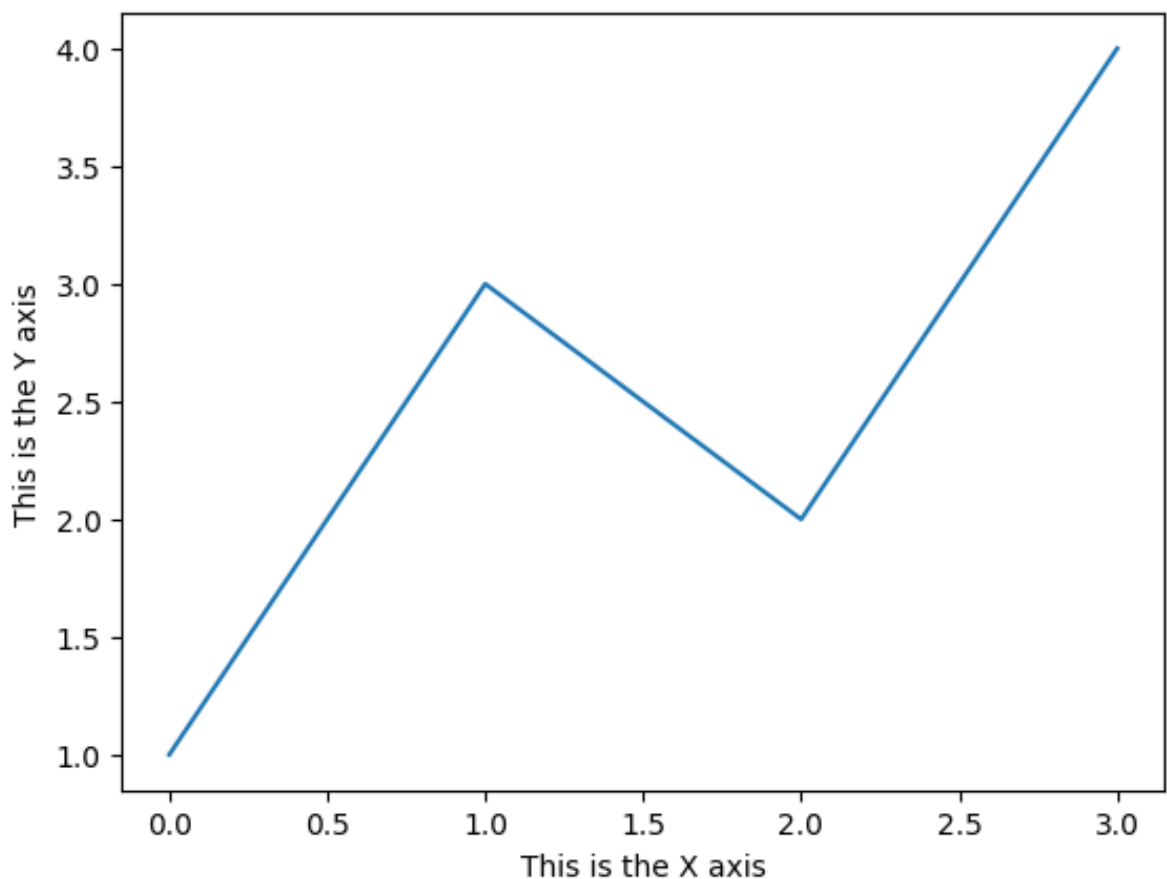


- Also for `xlim()` and `ylim()`, we can pass a list of two values (for example, `xlim([xmin, xmax])`), or use the keyword arguments.

Adding labels

- Now that we know how to manage the axes dimensions, another important piece of information to add to a plot is the axes labels, since they usually specify what kind of data we are plotting.
- Then add the `xlabel()` and `ylabel()` functions

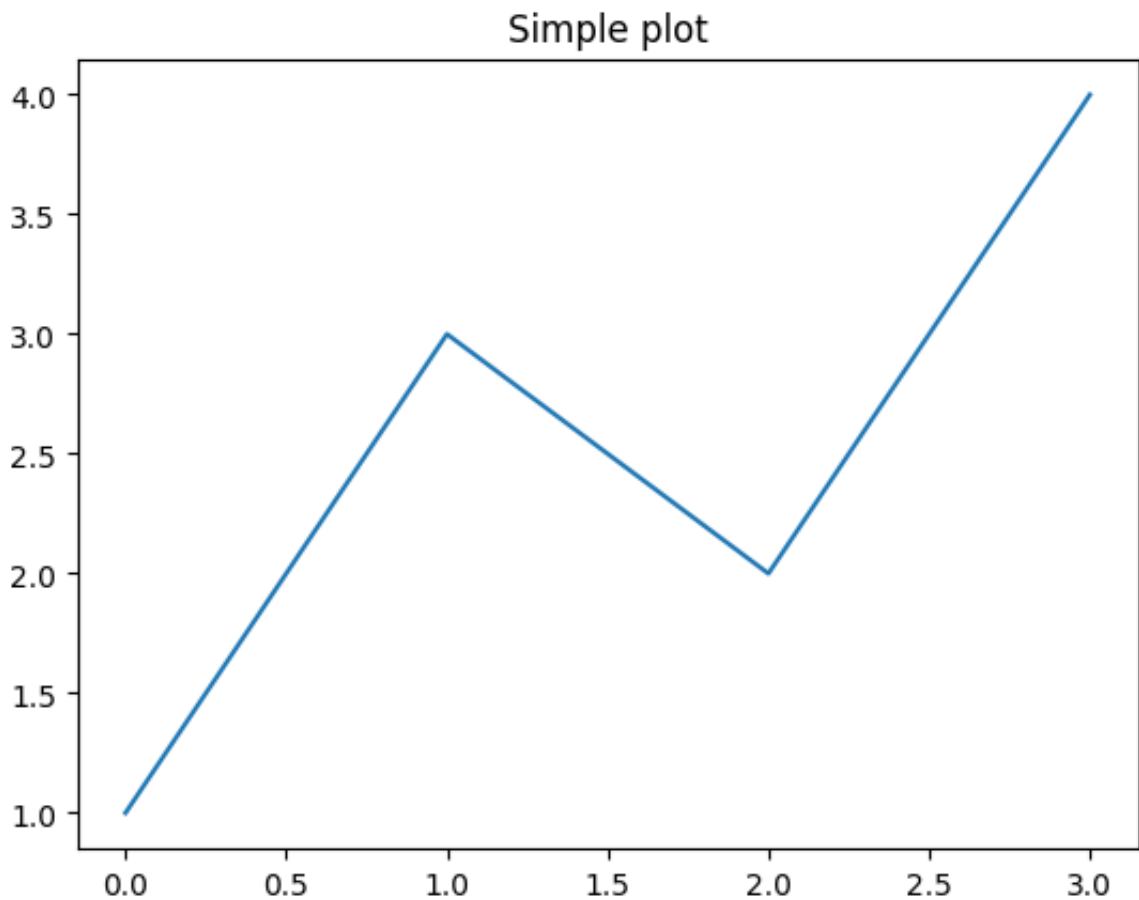
```
In [21]: import matplotlib.pyplot as plt
plt.plot([1, 3, 2, 4])
plt.xlabel('This is the X axis')
plt.ylabel('This is the Y axis')
plt.show()
```



Titles and legends

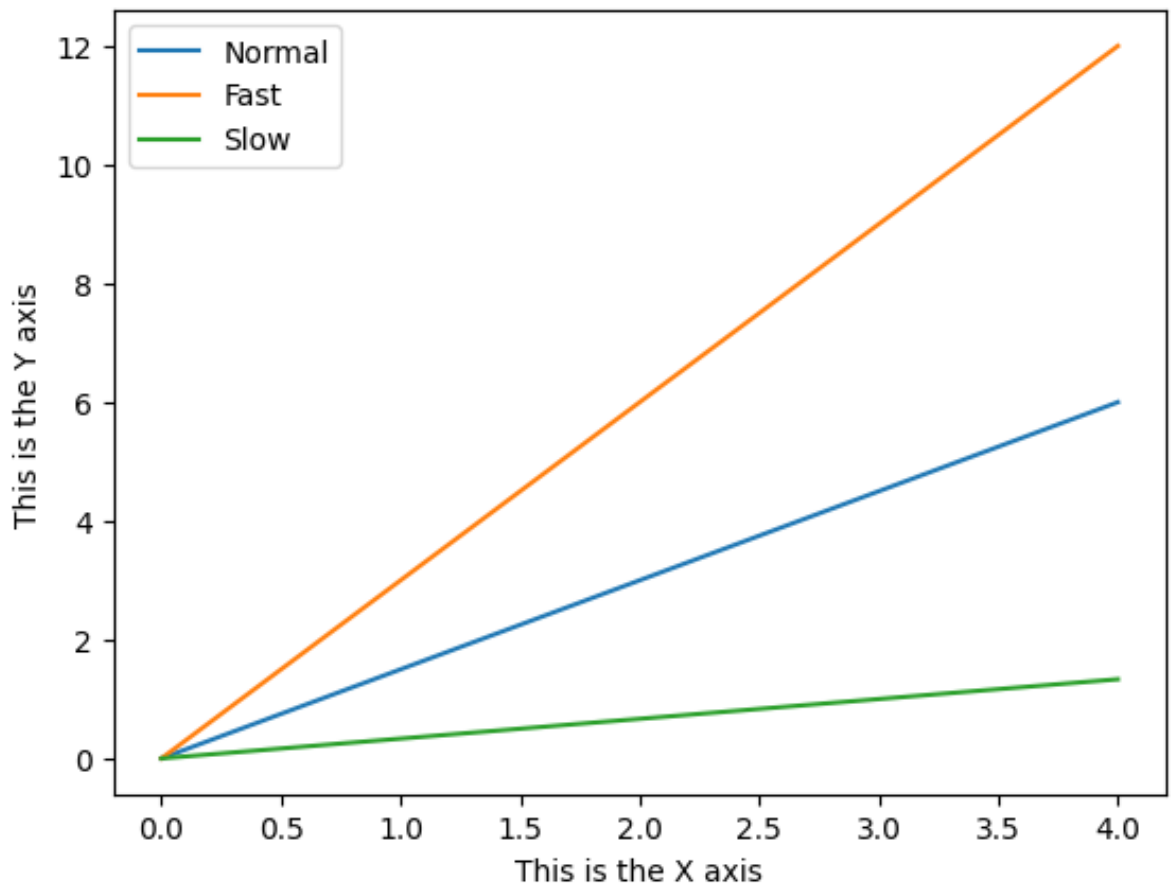
- We are about to introduce two other important plot features titles and legends
- **Adding a title**
 - Just like in a book or a paper, the title of a graph describes what it is:

```
In [22]: import matplotlib.pyplot as plt
plt.plot([1, 3, 2, 4])
plt.title('Simple plot')
plt.show()
```



- Matplotlib provides a simple function, `plt.title()`, to add a title to an image, as shown in the previous code.
- **Adding a legend**
 - The last thing we need to see to complete a basic plot is a legend
 - **Legends** are used to explain what each line means in the current figure. Let's take the multiline plot example again, and extend it with a legend:

```
In [29]: import matplotlib.pyplot as plt
import numpy as np
x = np.arange(5)
plt.plot(x, x*1.5, label='Normal')
plt.plot(x, x*3.0, label='Fast')
plt.plot(x, x/3.0, label='Slow')
plt.xlabel('This is the X axis')
plt.ylabel('This is the Y axis')
plt.legend(loc=2)
plt.show()
```



- The `legend()` function allows us to select several locations, which can be specified as an optional argument (or with the keyword argument, `loc`).
- The following table gives us the various positions at which the legend could be placed along with the equivalent codes for these positions.

String	Code
best	0
upper right	1
upper left	2
lower left	3
lower right	4
right	5
center left	6
center right	7
lower center	8
upper center	9
center	10

- An interesting functionality is the auto-legend positioning setting `loc='best'`, Matplotlib automatically tries to find the the optimal legend position.
- Another nice argument we'd like to mention is `ncol`; this argument specifies how many columns to use to layout the legend items.

Saving plots to a file

- Saving a plot to a file is an easy task. The following example shows how:

```
In [31]: import matplotlib.pyplot as plt
plt.plot([1, 2, 3])
plt.savefig('plot76.csv')
```

```
-----
-----
ValueError                                Traceback (most recent c
all last)
Cell In[31], line 3
      1 import matplotlib.pyplot as plt
      2 plt.plot([1, 2, 3])
----> 3 plt.savefig('plot76.csv')
```

File /opt/homebrew/lib/python3.11/site-packages/matplotlib/pyplot.
py:1023, in savefig(*args, **kwargs)
1020 @_copy_docstring_and_deprecators(Figure.savefig)


```

1021 def savefig(*args, **kwargs):
1022     fig = gcf()
-> 1023     res = fig.savefig(*args, **kwargs)
1024     fig.canvas.draw_idle() # Need this if 'transparent=True', to reset colors.
1025     return res

```

File /opt/homebrew/lib/python3.11/site-packages/matplotlib/figure.py:3343, in Figure.savefig(self, fname, transparent, **kwargs)

```

3339     for ax in self.axes:
3340         stack.enter_context(
3341             ax.patch._cm_set(facecolor='none', edgecolor='
none'))
-> 3343 self.canvas.print_figure(fname, **kwargs)

```

File /opt/homebrew/lib/python3.11/site-packages/matplotlib/backend_bases.py:2313, in FigureCanvasBase.print_figure(self, filename, dpi, facecolor, edgecolor, orientation, format, bbox_inches, pad_inches, bbox_extra_artists, backend, **kwargs)

```

2310     dpi = getattr(self.figure, '_original_dpi', self.figure.dpi)
2312 # Remove the figure manager, if any, to avoid resizing the
GUI widget.
-> 2313 with cbook._setattr_cm(self, manager=None), \
2314     self._switch_canvas_and_return_print_method(format, backend) \
2315         as print_method, \
2316     cbook._setattr_cm(self.figure, dpi=dpi), \
2317     cbook._setattr_cm(self.figure.canvas, _device_pixel_ratio=1), \
2318     cbook._setattr_cm(self.figure.canvas, _is_saving=True), \
2319     ExitStack() as stack:
2321     for prop in ["facecolor", "edgecolor"]:
2322         color = locals()[prop]

```

File /opt/homebrew/Cellar/python@3.11/3.11.3/Frameworks/Python.framework/Versions/3.11/lib/python3.11/contextlib.py:137, in _GeneratorContextManager.__enter__(self)

```

135 del self.args, self.kwds, self.func
136 try:
--> 137     return next(self.gen)
138 except StopIteration:
139     raise RuntimeError("generator didn't yield") from None

```

File /opt/homebrew/lib/python3.11/site-packages/matplotlib/backend_bases.py:2220, in FigureCanvasBase._switch_canvas_and_return_print_method(self, fmt, backend)

```

2218     canvas = self.switch_backends(canvas_class)
2219     if canvas is None:
-> 2220         raise ValueError(
2221             "Format {!r} is not supported (supported formats:
{})."

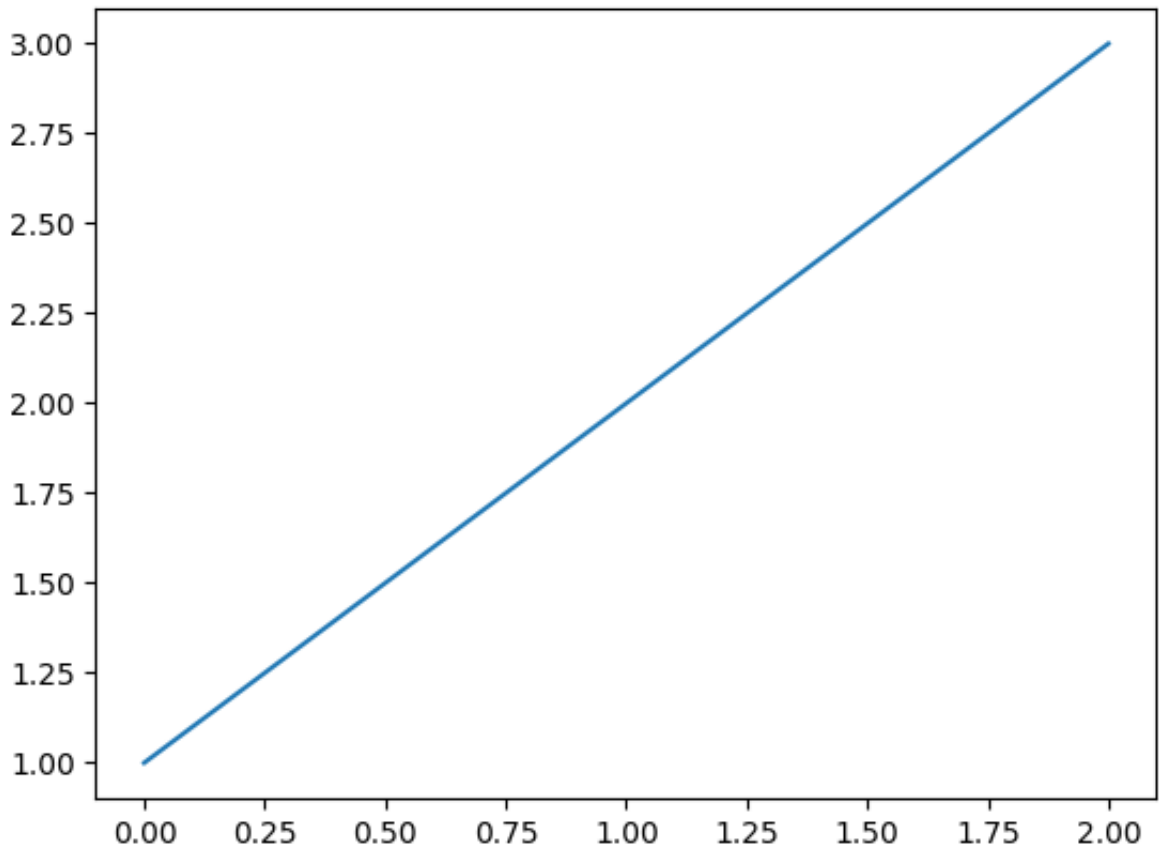
```

```

2222             fmt, ", ".join(sorted(self.get_supported_filet
ypes()))))
2223 meth = getattr(canvas, f"print_{fmt}")
2224 mod = (meth.func.__module__
2225         if hasattr(meth, "func") # partialmethod, e.g. bac
kend_wx.
2226         else meth.__module__)

```

ValueError: Format 'csv' is not supported (supported formats: eps, jpeg, jpg, pdf, pgf, png, ps, raw, rgba, svg, svgz, tif, tiff, webp)

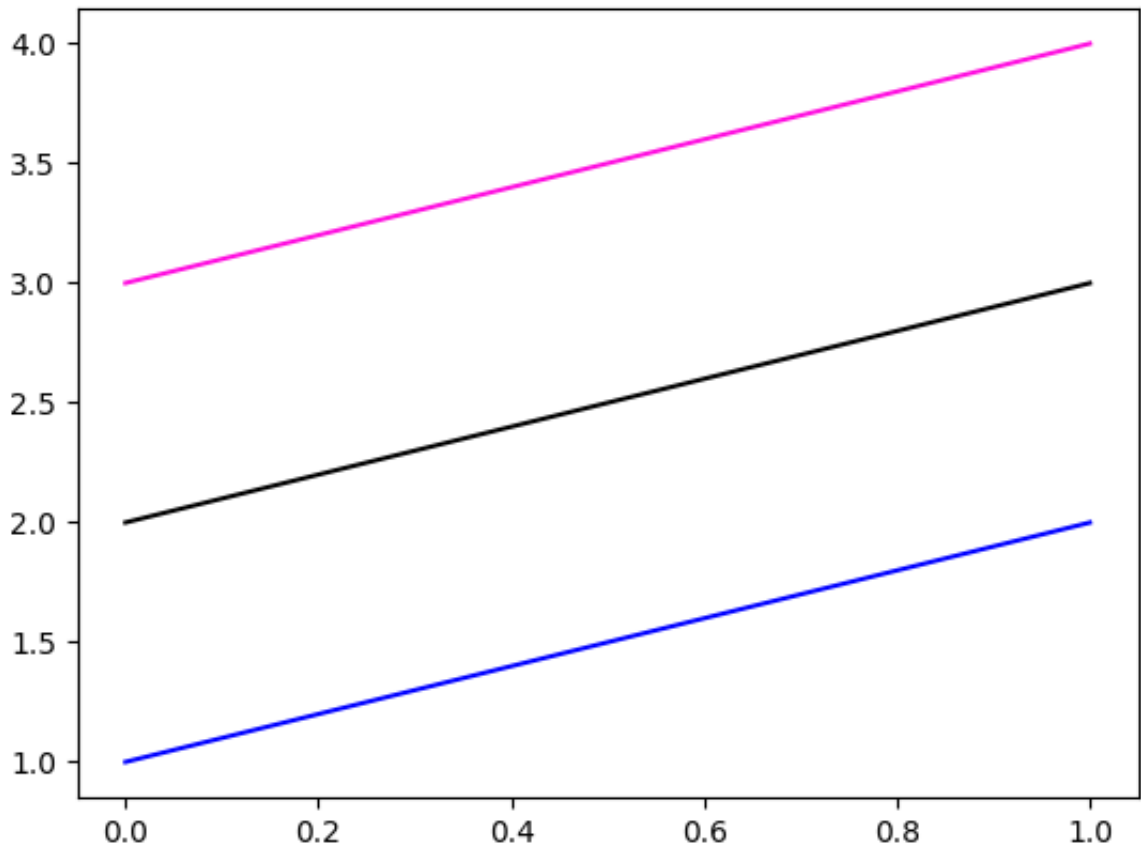


3.0 Decorate Graphs with Plot Styles and Types

Control colors

- We've already seen that in a multiline plot, Matplotlib automatically chooses different colors for different lines. We are also free to choose them by ourselves:

```
In [35]: import matplotlib.pyplot as plt
import numpy as np
y = np.arange(1, 3)
plt.plot(y, c='b');
plt.plot(y+1, c='k');
plt.plot(y+2, c='#ff0de0');
plt.show()
```



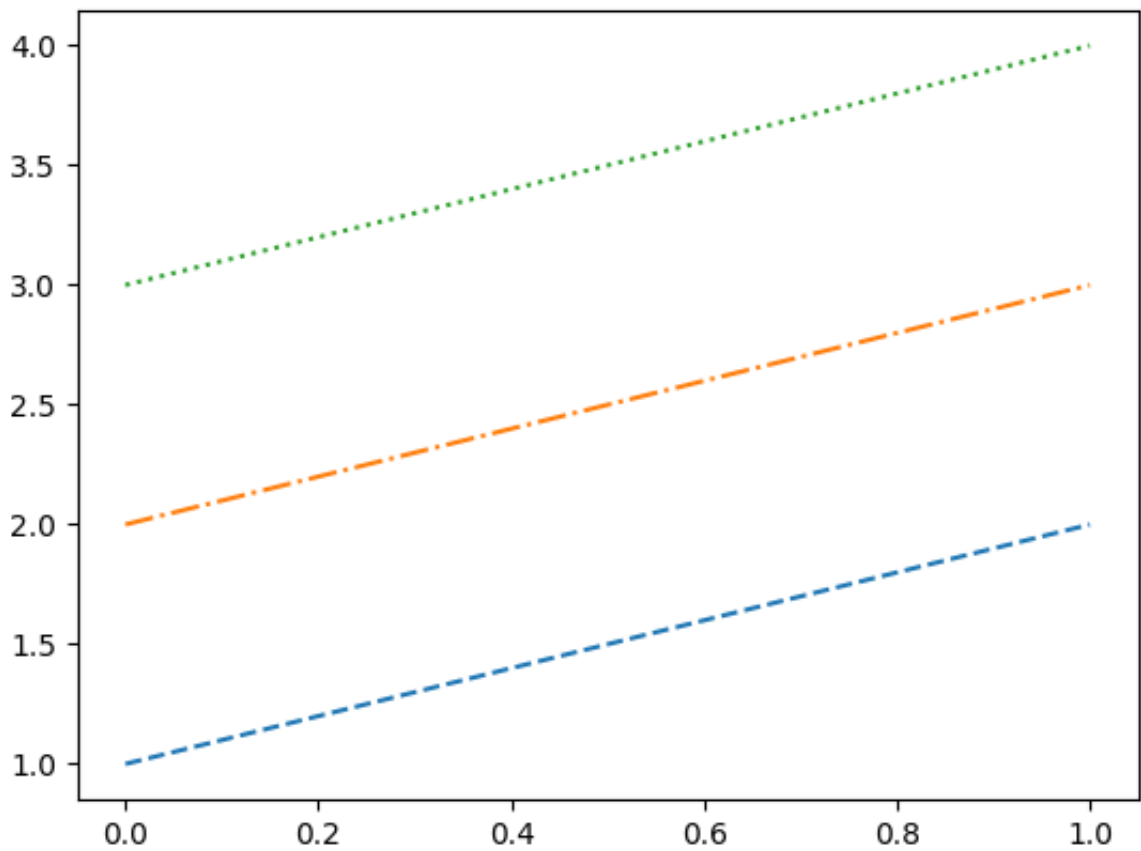
- Here is a table of the abbreviations used to select colors:

Color abbreviation	Color Name
b	blue
c	cyan
g	green
k	black
m	magenta
r	red
w	white
y	yellow

Control line styles

- All the lines seen until now are proper ones without any dots or dashes. Matplotlib allows us to use different line styles, for example:

```
In [36]: import matplotlib.pyplot as plt
import numpy as np
y = np.arange(1, 3)
plt.plot(y, '--', y+1, '-.', y+2, ':');
plt.show()
```



Style abbreviation	Style
-	solid line
--	dashed line
-.	dash-dot line
:	dotted line

Control marker styles

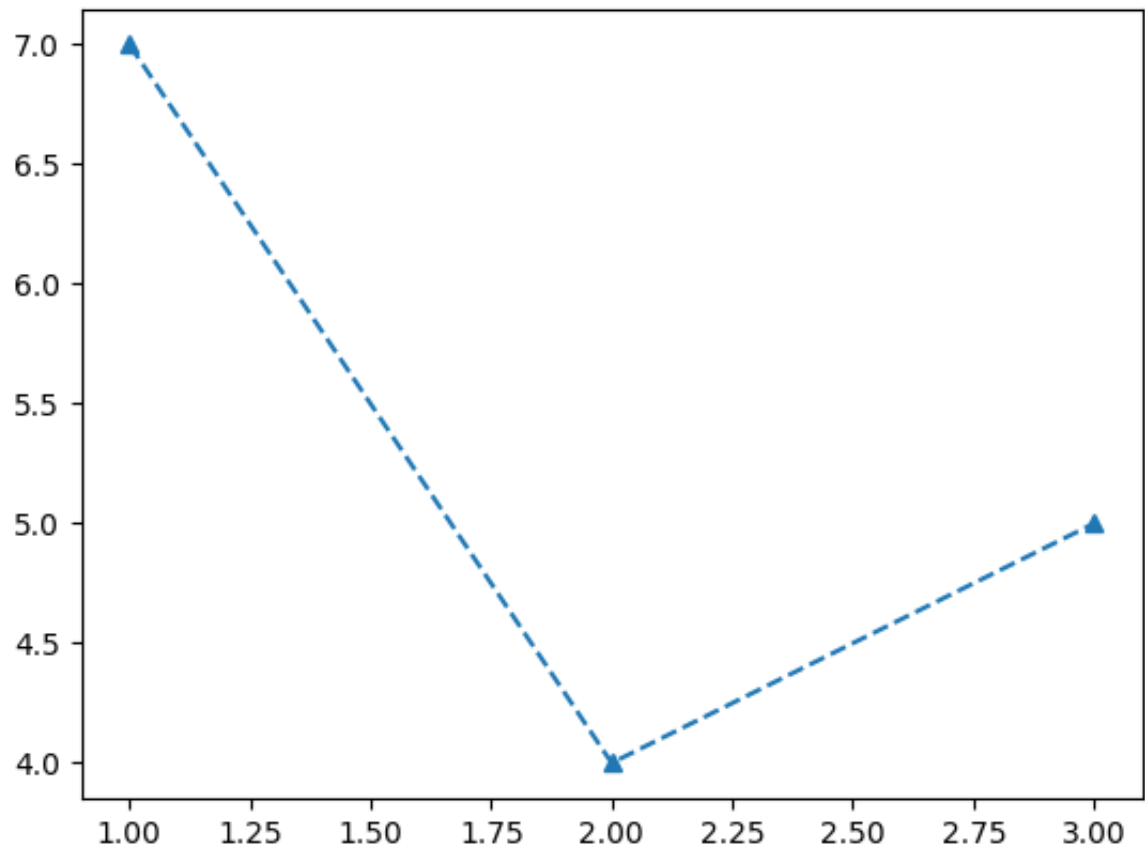
- Markers are, by default, drawn as point markers. They are just a location on the figure where segments join.
- Matplotlib provides a lot of customization options for markers. The following table contains a list of the some styles:

Marker abbreviation	Marker style
.	Point marker
,	Pixel marker
o	Circle marker
v	Triangle down marker
^	Triangle up marker
<	Triangle left marker
>	Triangle right marker

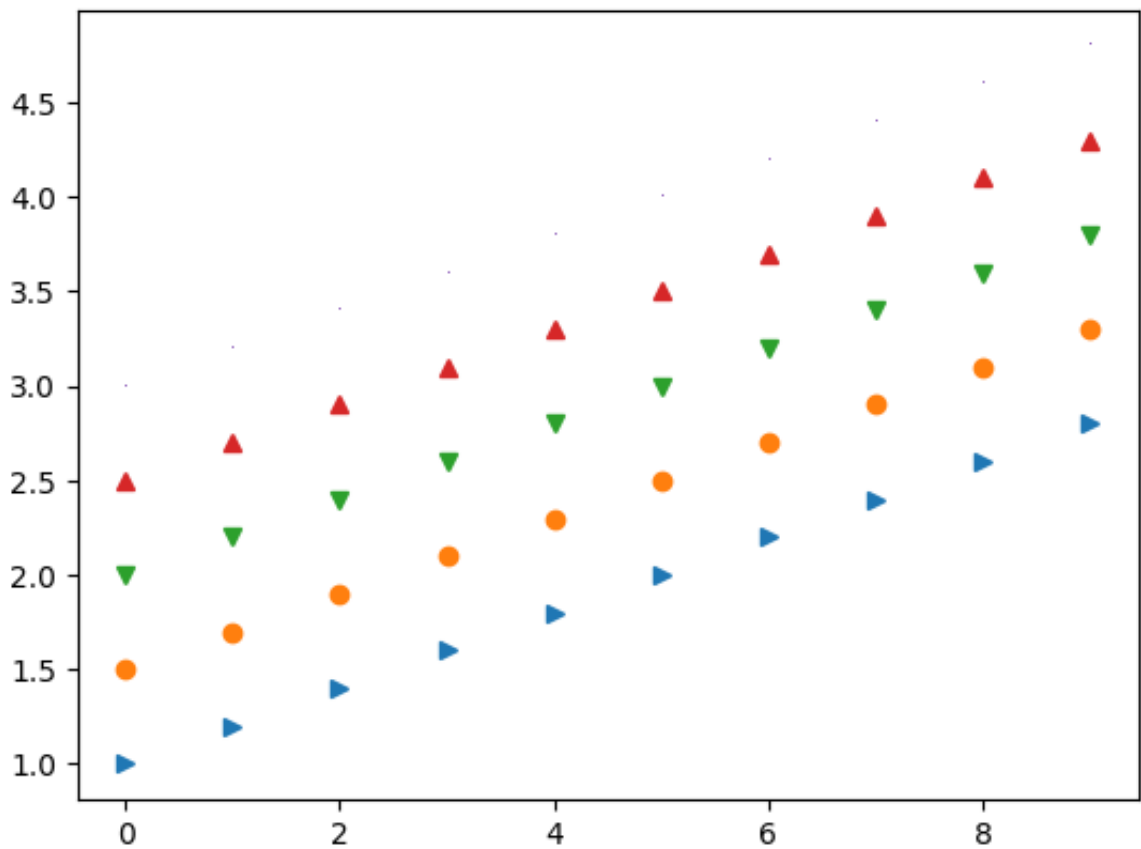
- Let's look at some of them:

```
In [41]: plt.plot([1,2,3],[7,4,5], '^', linestyle='--')
```

Out[41]: [`<matplotlib.lines.Line2D at 0x12edb7f90>`]



```
In [42]: import matplotlib.pyplot as plt
import numpy as np
y = np.arange(1, 3, 0.2)
plt.plot(y, '>', y+0.5, 'o', y+1, 'v', y+1.5, '^', y+2, ',');
plt.show()
```

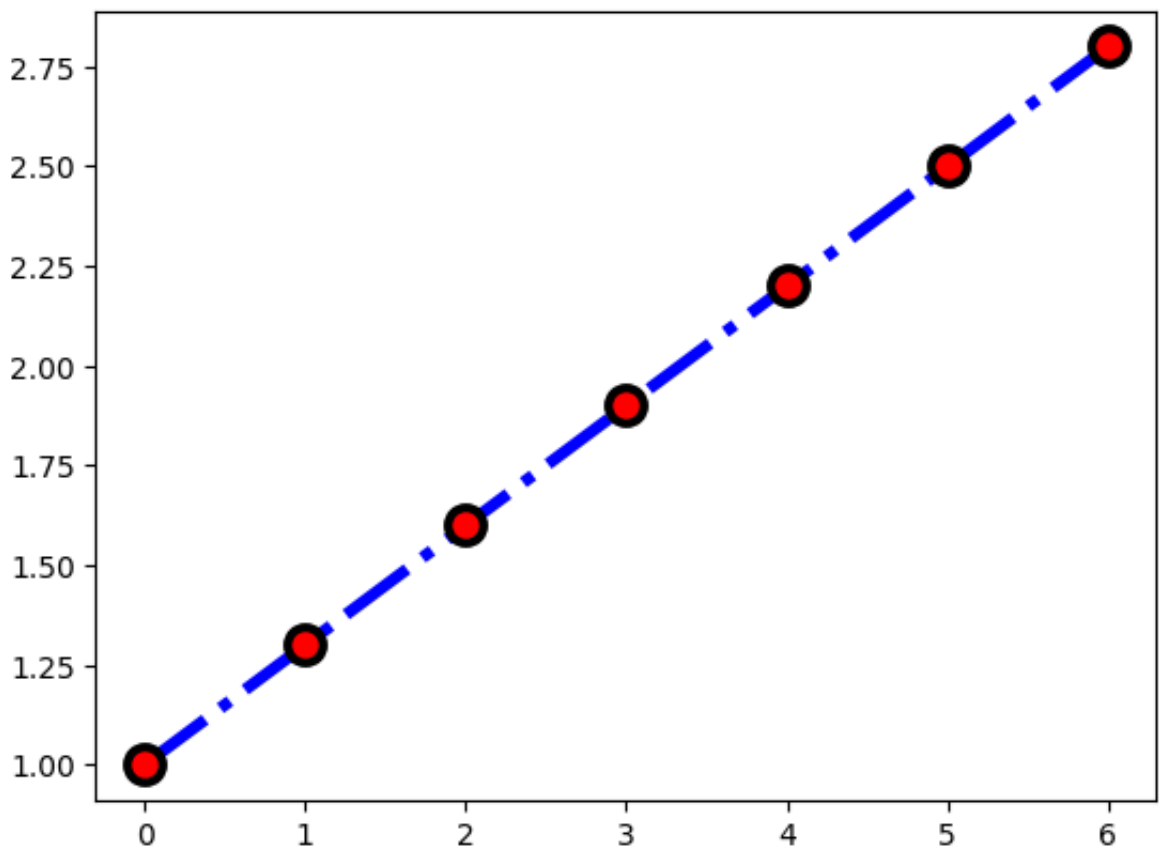


Control marker styles

- Format strings are really useful, but they have some drawbacks. For example, they don't allow us to specify different colors for lines and markers, as we saw in the previous example.
- `plot()` is a really rich function, and there are some keyword arguments to configure colors, markers, and line styles:

Keyword argument	Description
color or c	Sets the color of the line; accepts any Matplotlib color format.
linestyle	Sets the line style; accepts the line styles seen previously.
linewidth	Sets the line width; accepts a float value in points.
marker	Sets the line marker style.
markeredgecolor	Sets the marker edge color; accepts any Matplotlib colorformat.
markeredgewidth	Sets the marker edge width; accepts float value in points.
markerfacecolor	Sets the marker face color; accepts any Matplotlib color format.
markersize	Sets the marker size in points; accepts float values.

```
In [43]: import matplotlib.pyplot as plt
import numpy as np
y = np.arange(1, 3, 0.3)
plt.plot(y, color='blue', linestyle='dashdot', linewidth=4,
marker='o', markerfacecolor='red', markeredgecolor='black',
markeredgewidth=3, markersize=12);
plt.show()
```



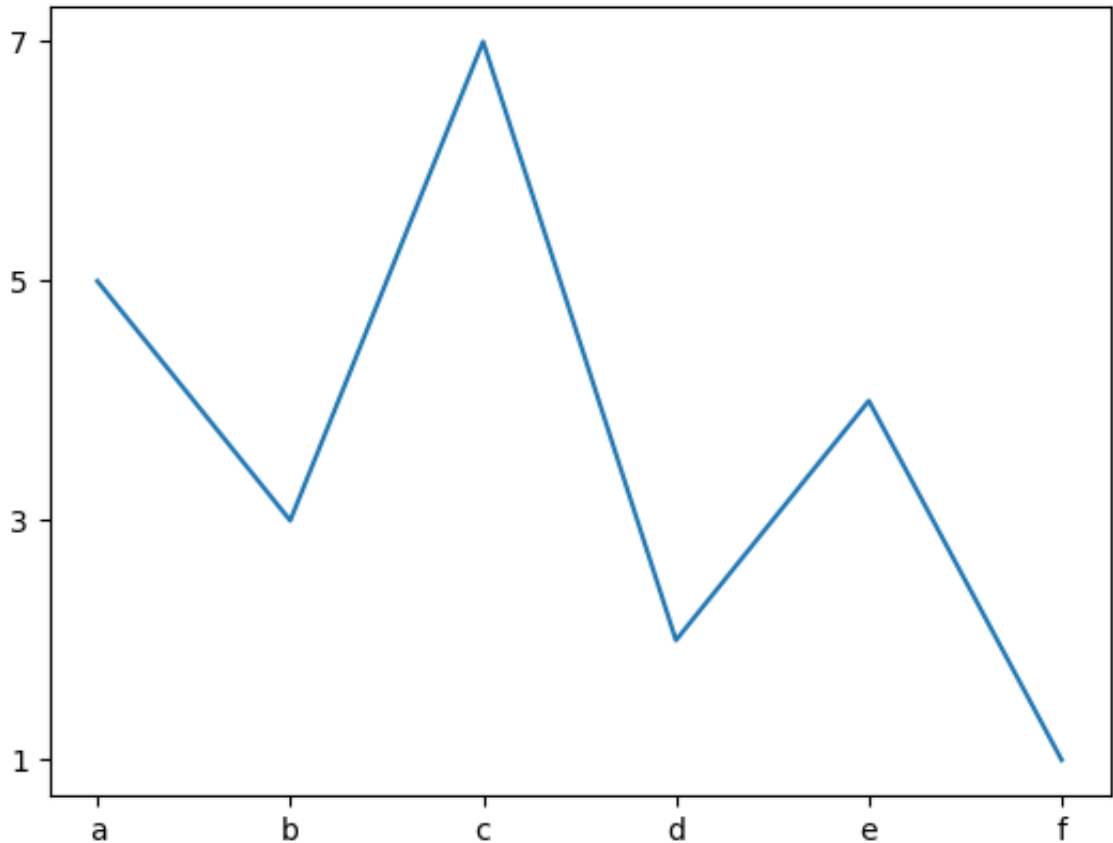
Handling X and Y ticks

- Matplotlib provides two basic functions to manage them—`xticks()` and `yticks()`. They behave in the same way, so the description for one function will apply to the other too.
- Executing with no arguments, the tick function returns the current ticks' locations and the labels corresponding to each of them:

```
locs, labels = plt.xticks()
```

- The arguments (in the form of lists) that we can pass to the function are:
 - **Locations of the ticks**
 - **Labels to draw at these locations (if necessary)**
- Let's try to explain it with an example:

```
In [44]: import matplotlib.pyplot as plt
x = [5, 3, 7, 2, 4, 1]
plt.plot(x)
plt.xticks(range(len(x)), ['a', 'b', 'c', 'd', 'e', 'f'])
plt.yticks(range(1, 8, 2))
plt.show()
```

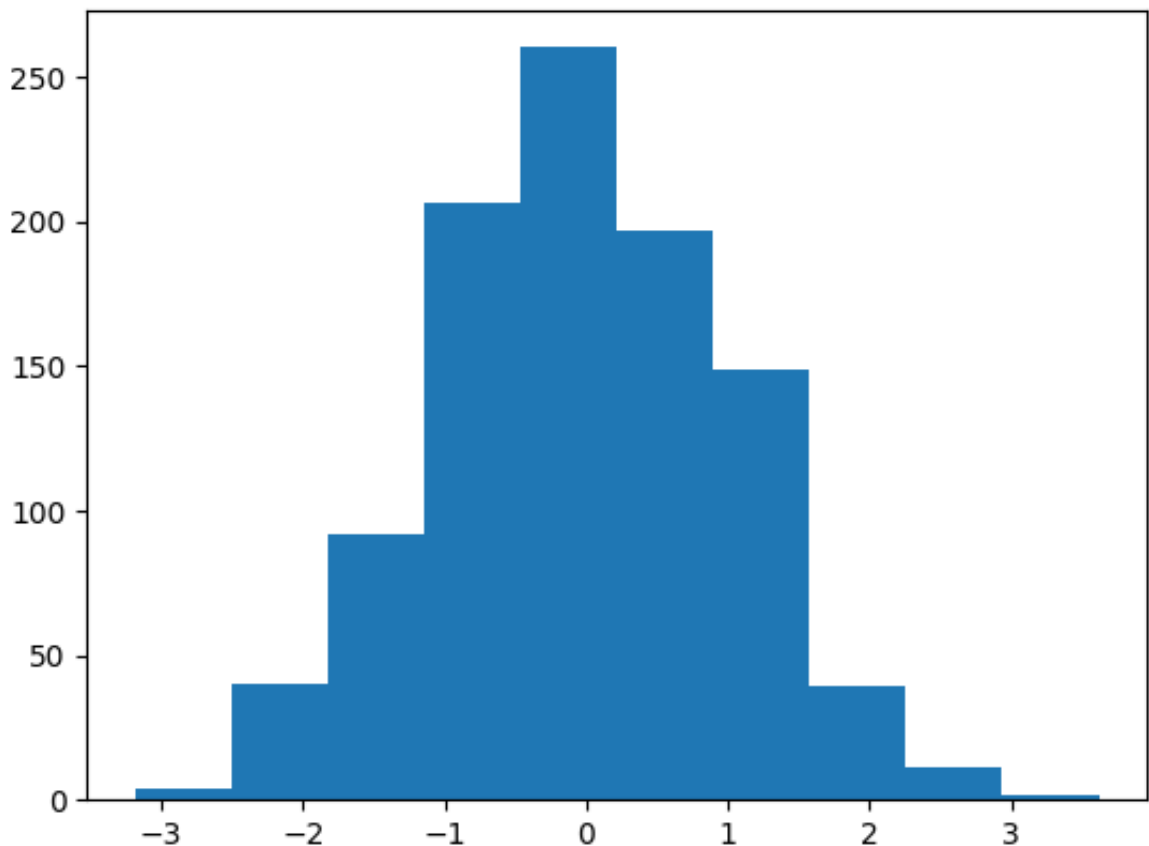


Plot types

Histogram charts

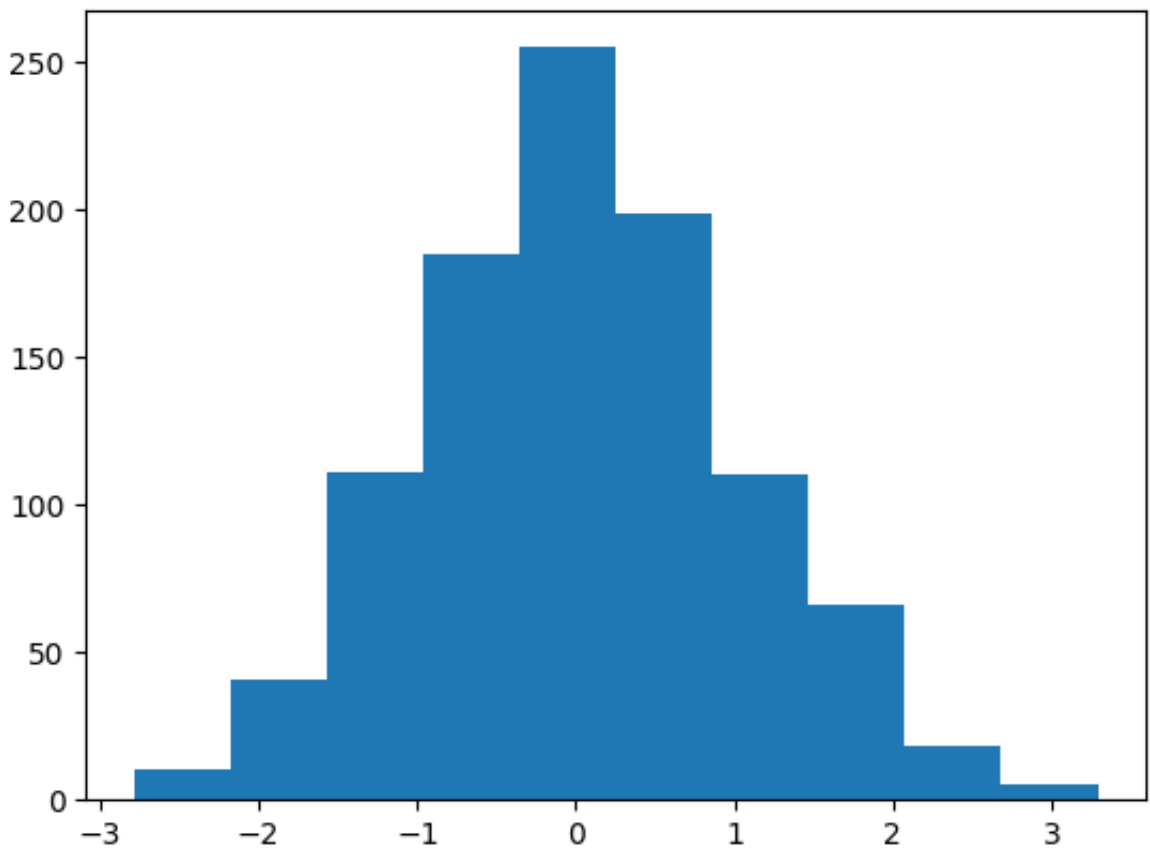
- Histogram charts are a graphical display of frequencies, represented as bars.
- They show what portion of the dataset falls into each category, usually specified as non-overlapping intervals. Matplotlib calls those categories `bins`.

```
In [45]: import matplotlib.pyplot as plt
import numpy as np
y = np.random.randn(1000)
plt.hist(y)
plt.show()
```



- Histogram plots group up values into bins of values. By default, `hist()` uses a `bin` value of 10 (so only ten categories, or bars, are computed), but we can customize it, either by passing an additional parameter, for example, in `hist(y,)`, or using the `bin` keyword argument as `hist(y, bin=)`.
- Replotting the previous dataset, but with `bin=25` :

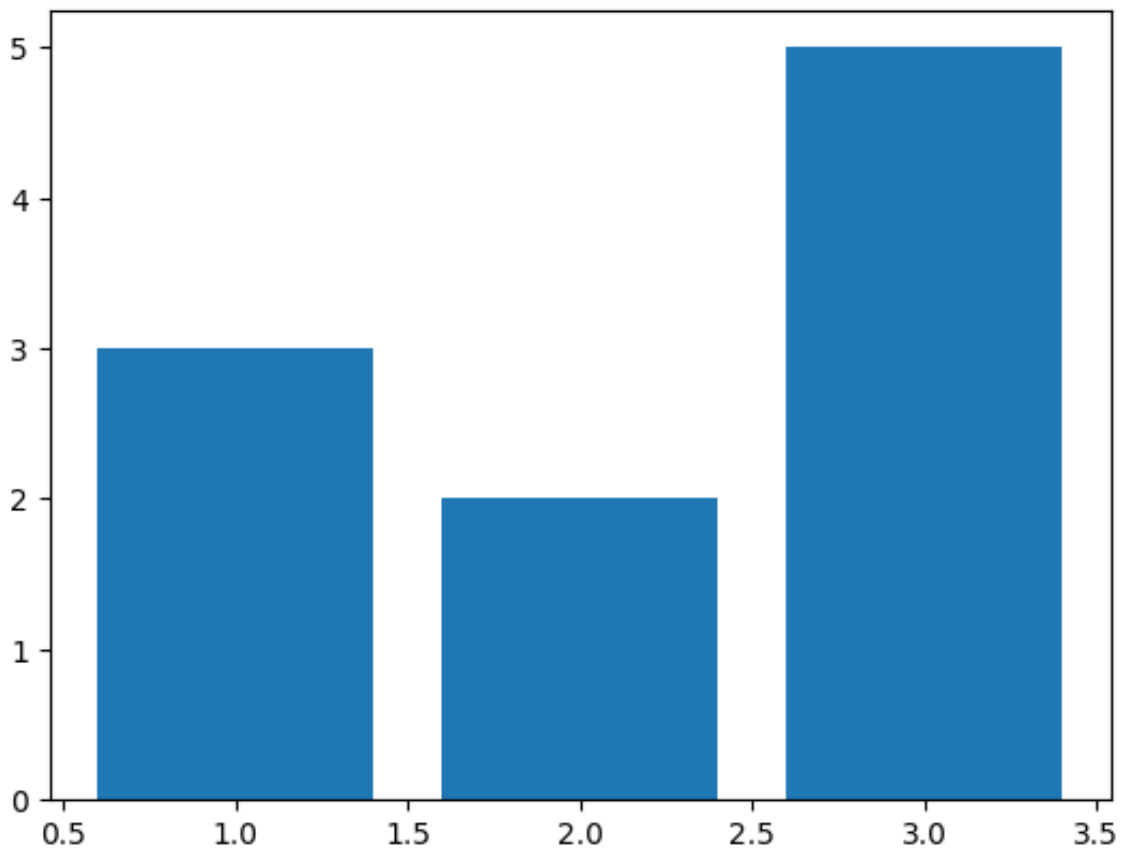
```
In [46]: import matplotlib.pyplot as plt
import numpy as np
y = np.random.randn(1000)
plt.hist(y, bins=10)
plt.show()
```



Bar charts

- Bar charts display rectangular bars (either vertical or horizontal) with their length proportional to the values they represent. They are commonly used to visually compare two or more values.
- The `bar()` function is used to generate bar charts in Matplotlib. The function expects two lists of values: the X coordinates that are the positions of the bar's left margin and the heights of the bars:

```
In [47]: import matplotlib.pyplot as plt
plt.bar([1, 2, 3], [3, 2, 5]);
plt.show()
```



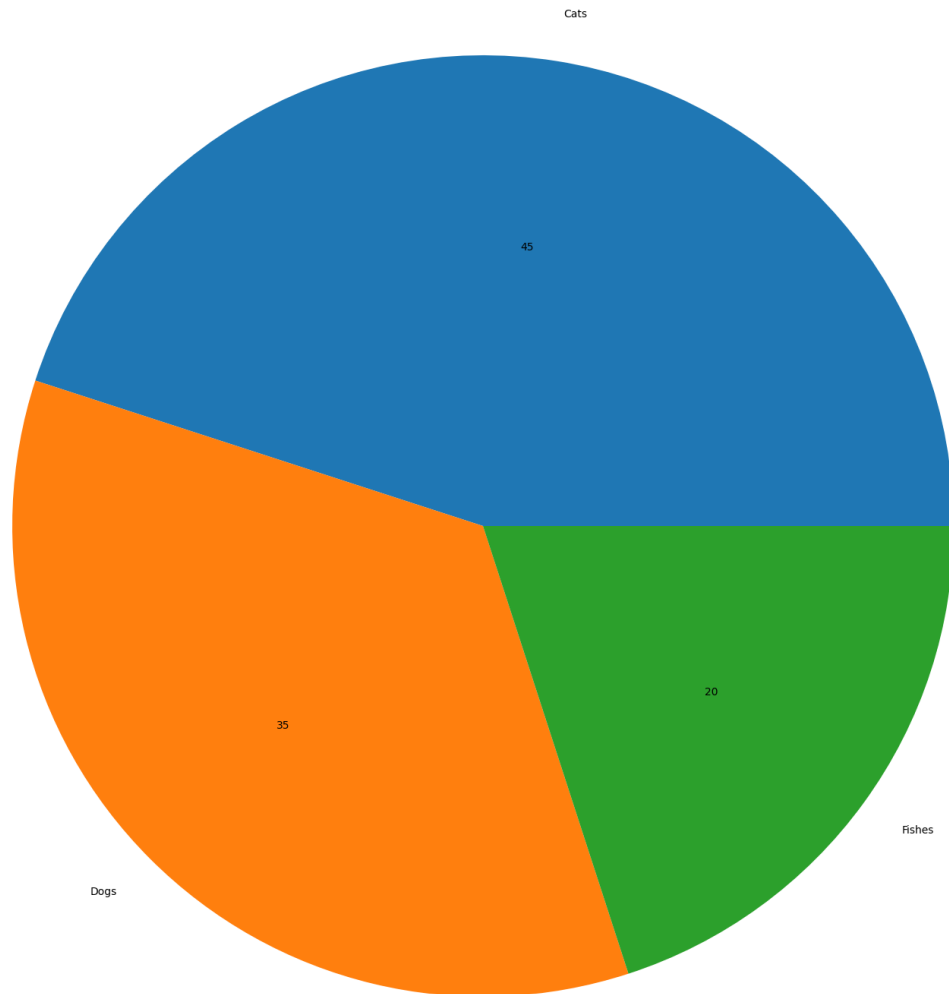
Pie charts

- Pie charts are circular representations, divided into sectors (also called wedges). The arc length of each sector is proportional to the quantity we're describing.
- It's an effective way to represent information when we are interested mainly in comparing the wedge against the whole pie, instead of wedges against each other.
- Matplotlib provides the `pie()` function to plot pie charts from an array `X`. Wedges are created proportionally, so that each value `x` of array `X` generates a wedge proportional to $x/\text{sum}(X)$.
- Please note that if $\text{sum}(X)$ is less than 1, then the pie is drawn using `X` values directly and no normalization is done, resulting in a pie with discontinuity.
- In the next example, we are going to plot a simple pie, using the `legend` keyword argument to give names to the wedges:

```
In [ ]:
```

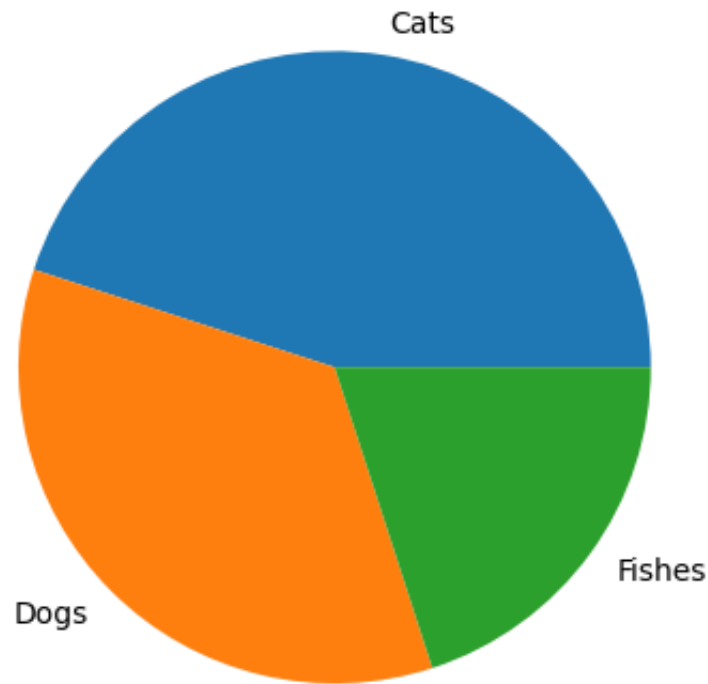
```
In [63]: import matplotlib.pyplot as plt
plt.figure(figsize=(3,3));
x = [45, 35, 20]
labels = ['Cats', 'Dogs', 'Fishes']
plt.figure(figsize=(20,30))
plt.pie(x, labels=labels, autopct='%.0f')
plt.show()
```

<Figure size 300x300 with 0 Axes>



```
In [56]: dir(plt.pie(x, labels=labels))
```

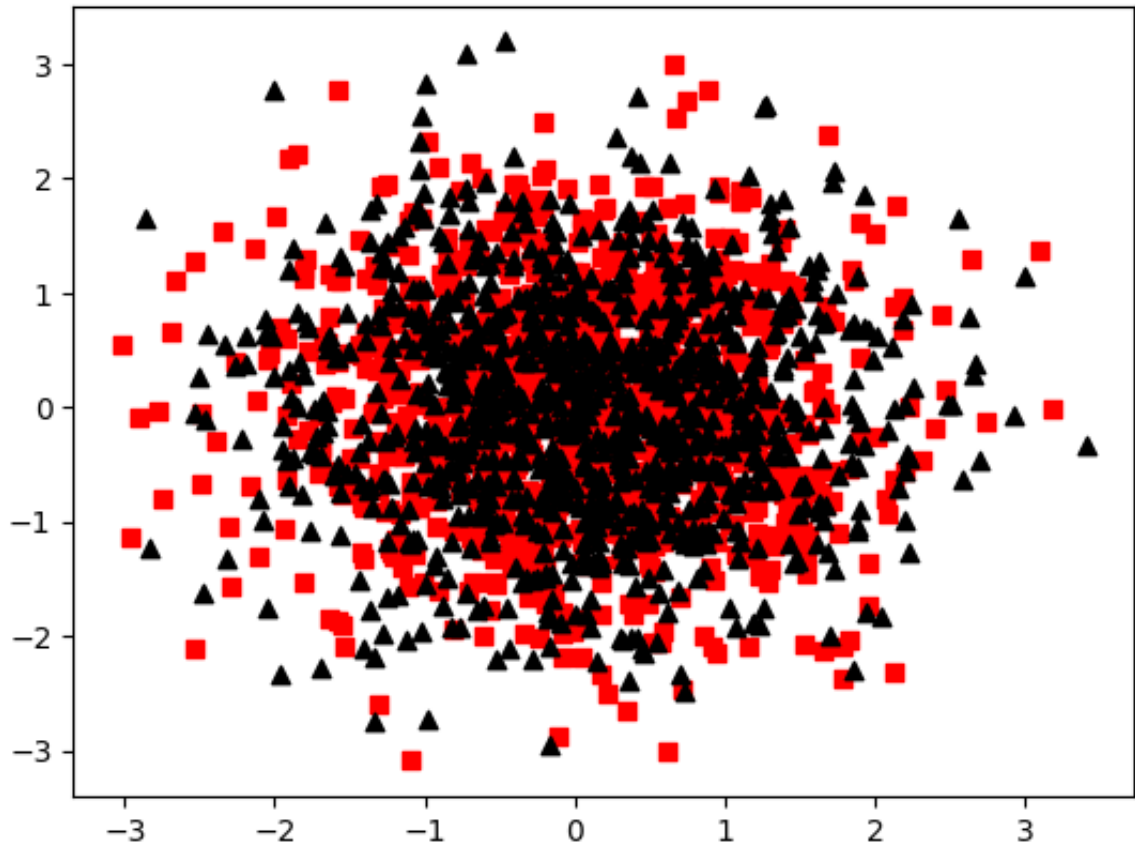
```
Out[56]: ['__add__',
          '__class__',
          '__class_getitem__',
          '__contains__',
          '__delattr__',
          '__dir__',
          '__doc__',
          '__eq__',
          '__format__',
          '__ge__',
          '__getattr__',
          '__getitem__',
          '__getnewargs__',
          '__getstate__',
          '__gt__',
          '__hash__',
          '__init__',
          '__init_subclass__',
          '__iter__',
          '__le__',
          '__len__',
          '__lt__',
          '__mul__',
          '__ne__',
          '__new__',
          '__reduce__',
          '__reduce_ex__',
          '__repr__',
          '__rmul__',
          '__setattr__',
          '__sizeof__',
          '__str__',
          '__subclasshook__',
          'count',
          'index']
```

Scatter plots

- Scatter plots display values for two sets of data. The data visualization is done as a collection of points not connected by lines. Each of them has its coordinates determined by the value of the variables (one variable determines the X position, the other the Y position).
- A scatter plot is often used to identify potential association between two variables, and it's often drawn before working on a fitting regression function. It gives a good visual picture of the correlation, in particular for nonlinear relationships.
- Matplotlib provides the `scatter()` function to plot X versus Y unidimensional array of the same length as scatter plot.

```
In [64]: import matplotlib.pyplot as plt
import numpy as np
x = np.random.randn(1000)
y = np.random.randn(1000)
plt.scatter(x, y, marker='s', c='r');
x = np.random.randn(1000)
y = np.random.randn(1000)
plt.scatter(x, y, marker='^', c='k');
plt.show()
```



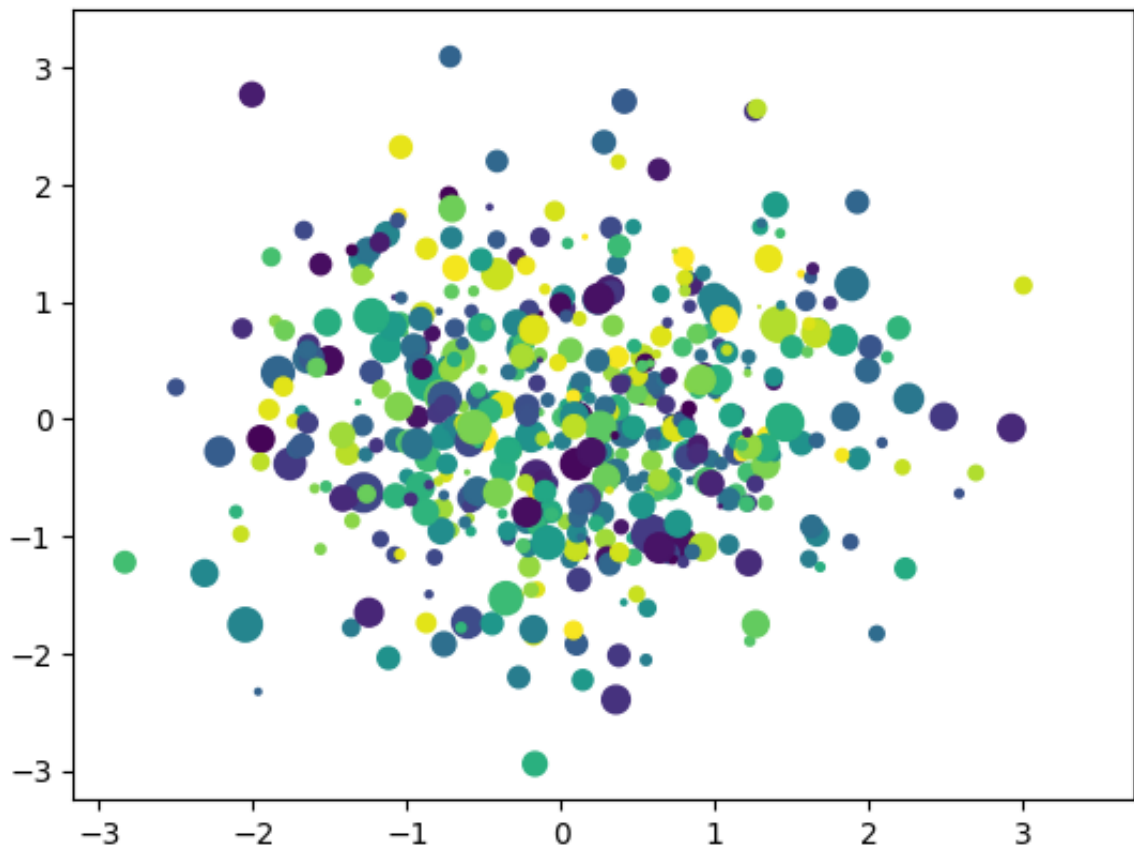
- We can decorate the chart by using some of the following keyword arguments:
- **s**: This stands for the size of the markers in pixel*pixel. It can be a single value (to be used for all the points) or an array of the same size of X and Y (so that each point will have its own size).
- **c**: This is the points color. It can be a single value or a list of colors (that will be cycled on the points plotted) eventually of the same size of X and Y. The values can be the Matplotlib color codes or even numbers mapped to colors using color maps.
- **marker**: This specifies the marker to use to plot the points; the available values are:

Marker value	Description
s	Square
o	Circle
^	Triangle up
v	Triangle down
>	Triangle right
<	Triangle left
d	Diamond
p	Pentagon
h	Hexagon
8	Octagon
+	Plus
x	Cross

- We can now apply some of them to the next example, where we specify a different size and color for each point of the previous dataset.

```
In [65]: import matplotlib.pyplot as plt
import numpy as np
size = 50*np.random.randn(1000)
colors = np.random.rand(1000)
plt.scatter(x, y, s=size, c=colors);
plt.show()
```

```
/opt/homebrew/lib/python3.11/site-packages/matplotlib/collections.
py:963: RuntimeWarning: invalid value encountered in sqrt
  scale = np.sqrt(self._sizes) * dpi / 72.0 * self._factor
```



Referencies

- Matplotlib Release 2.2.2, John Hunter, Darren Dale, Eric Firing, Michael Droettboom, March, 2018.
- Matplotlib tutorial, Nicolas P. Rougier - Euroscipy 2012 – Prace, 2013