A Comprehensive Guide to Data Visualization Using Matplotlib For Complete Beginners

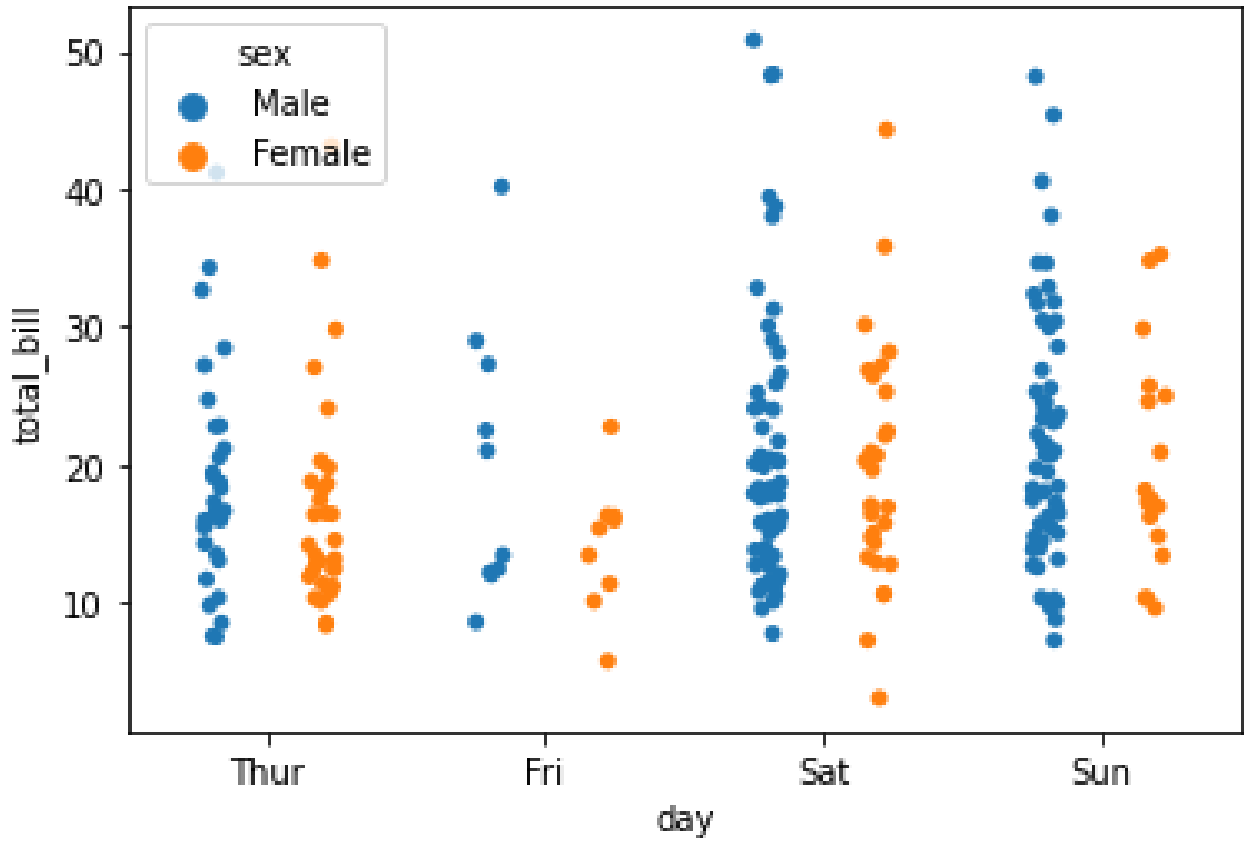
helloml.org/a-comprehensive-guide-to-data-visualization-using-matplotlib-for-complete-beginners/

July 6, 2021

Today we will start from complete scratch and learn how to plot basic graphs and charts using Matplotlib, a data visualization library for Python and its associated linear algebra and mathematical extensions providing library, NumPy.

We'll start right from the basics and build up from there. Additionally, we'll see some of the basic chart types and how to plot them in Matplotlib. So, let us dive straight in!

Brief Introduction to Data Visualization



Data visualization is at the heart of data science – An example of providing information through a **strip plot**.

Visualizing data in the form of graphs, charts, etc. (basically figures) adds more than just an aesthetic element to data analysis. Not only can one better identify trends and associated patterns from the given data, but it also becomes easier to know more information about the given data, such as predicting what algorithms/models the data is more suited for.

Visualizing data in the form of figures is way better than having to inspect hundreds or thousands (or maybe even millions) of records stored in a tabular format (database).

Brief Introduction to Matplotlib

If someone has even a slight familiarity with Matlab (another programming language) programming, they will be able to figure out the similarities between plotting figures using MatLab and Matplotlib. Matplotlib feels natural and comes across as being quite instinctive for MatLab users. That is because Matplotlib was created keeping MatLab functionalities in mind.

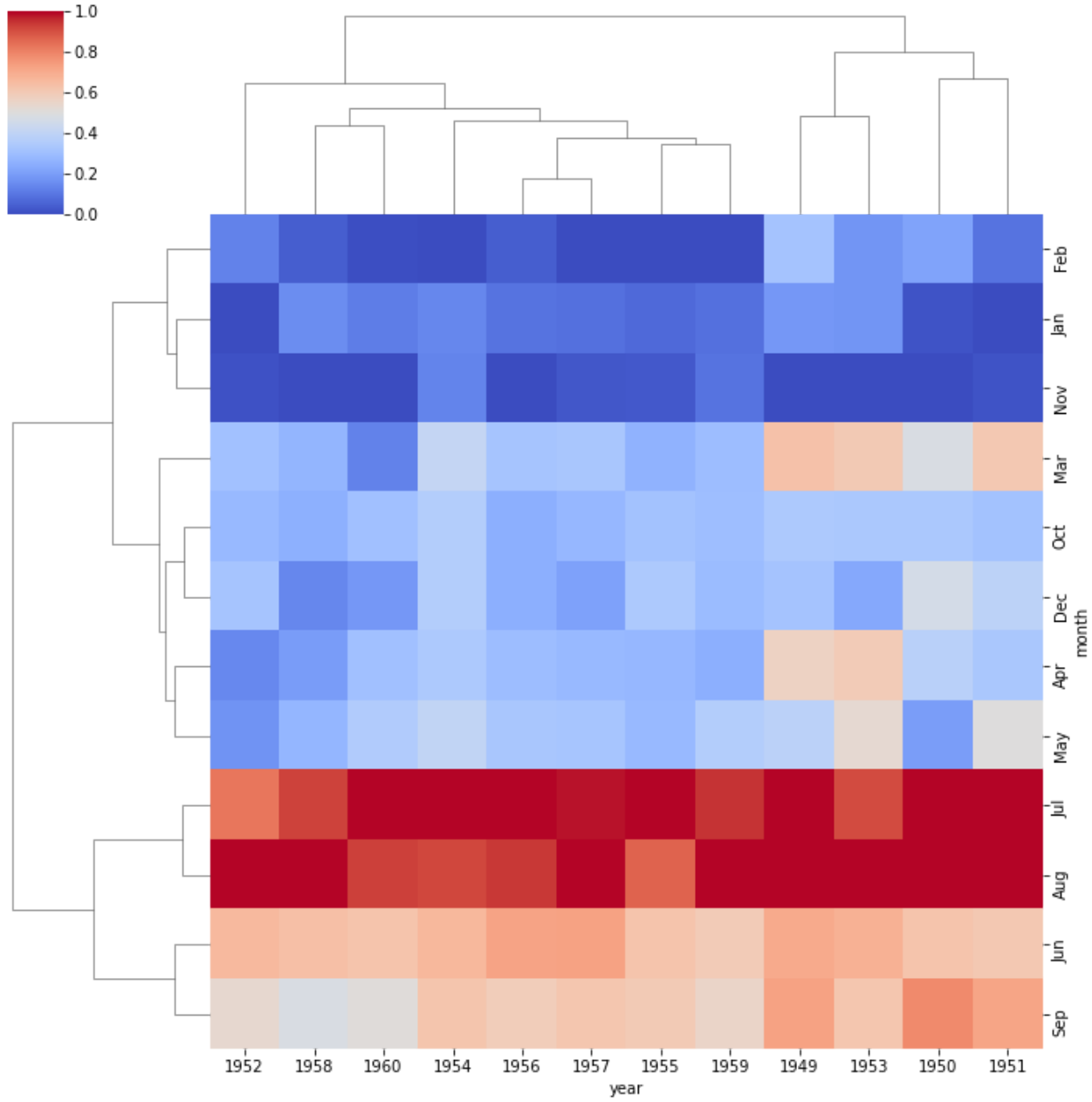
Matplotlib is at the root of data visualization with Python. It was created by **John Hunter**. He created it to try to replicate MatLab's plotting capabilities in Python. It is an excellent 2-D and 3-D graphics library for generating scientific figures.

Some positive points of Matplotlib are:

- Generally easy to get started for simple plots.
- It provides support for custom labels and texts.
- The user has great control of every element in a figure.
- Matplotlib provides high-quality output in many formats.
- It is very customizable in general, i.e. it is just like a blank canvas for painting. The user is free to do anything (or at least, a lot of things!) they like on top of Matplotlib's general functionalities.

In short, Matplotlib allows us to create reproducible figures programmatically. The official Matplotlib web page is a great starting point to know and learn more about the library: http://matplotlib.org/.

Why Learn Matplotlib? What About Other Data Visualization Libraries for Python?



An example of plotting advanced plots using Seaborn – A **heatmap** to display data. This begs the question that if one can get such articulate and (relatively) beautiful plots with just a few lines of code using Seaborn, why bother to learn Matplotlib?

There are many libraries that can be used for data visualization using Python, such as **Matplotlib**, **Seaborn**, **Plotly**, etc. However, Matplotlib is the most fundamental data visualization library. Libraries such as Seaborn and Plotly may be more powerful and equipped to handle complicated charts (like violin plots, heatmaps, choropleth maps, etc.) at times, but they all implement Matplotlib under the hood. Thus, Matplotlib is at the very core of data visualization using Python and is a must for someone who is an aspiring Data Analyst/Scientist or even simply an amateur enthusiast to be well equipped with Matplotlib.

Next, we will be making some basic plots using Matplotlib (To be more specific, We'll be using the **matplotlib.pyplot** module).

Installing Matplotlib

Assuming you're working on a Jupyter notebook (Anaconda installation of Python), installing Matplotlib on your PC is very simple. All you need to do is type in the following command in the Anaconda command prompt:

```
conda install matplotlib
```

Or you you are using it without anaconda, you can also install it using pip.

```
pip install matplotlib
```

Import the `matplotlib.pyplot` module under the name `plt` (most commonly used).

```
import matplotlib.pyplot as plt
%matplotlib inline
```

The second line is used to see the plots directly in the notebook. However, keep in mind that it is only for jupyter notebooks, if you are using another editor, you'll have to use: **plt.show()** at the end of all your plotting commands to have the figure pop up in another window.

Basic Example

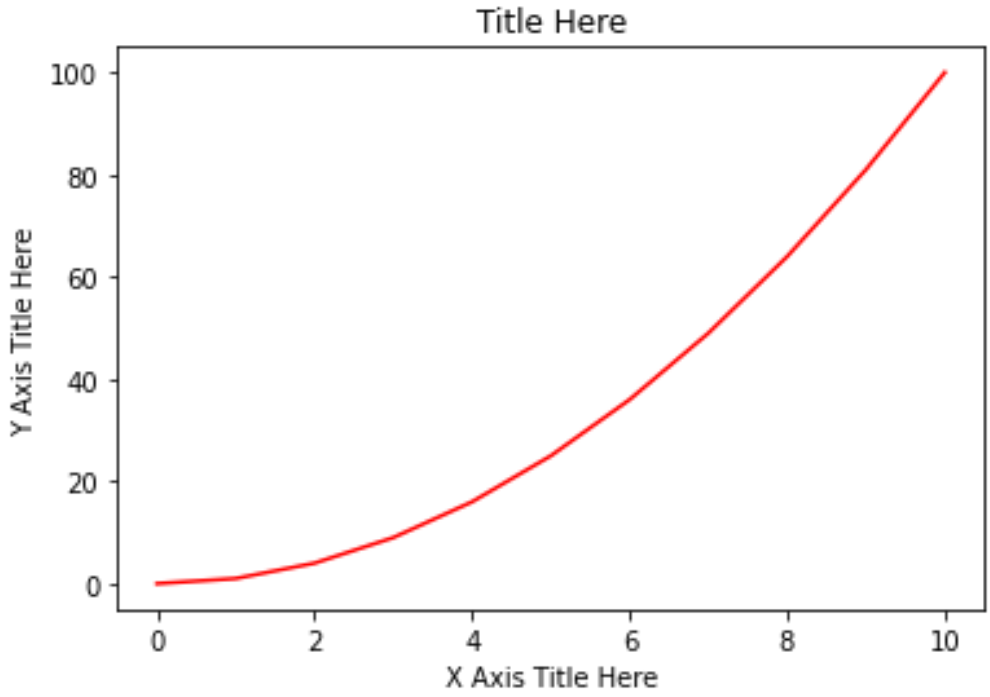
Let's walk through a very simple example using two NumPy arrays. You can also use lists, but most likely you'll be passing NumPy arrays or pandas columns (which essentially also behave like arrays). **The data we want to plot is simple: x is a set of integers from 0 to 10 and y is x squared**.

```
import numpy as np
x = np.linspace(0, 10, 11)
y = x ** 2

x

array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])

y

array([  0.,   1.,   4.,   9.,  16.,  25.,  36.,  49.,  64.,  81., 100.])
```

Basic Matplotlib Commands

We can create a very simple line plot using the following (In Jupyter Notebook, press Shift+Tab to check out the documentation strings for the functions we are using).

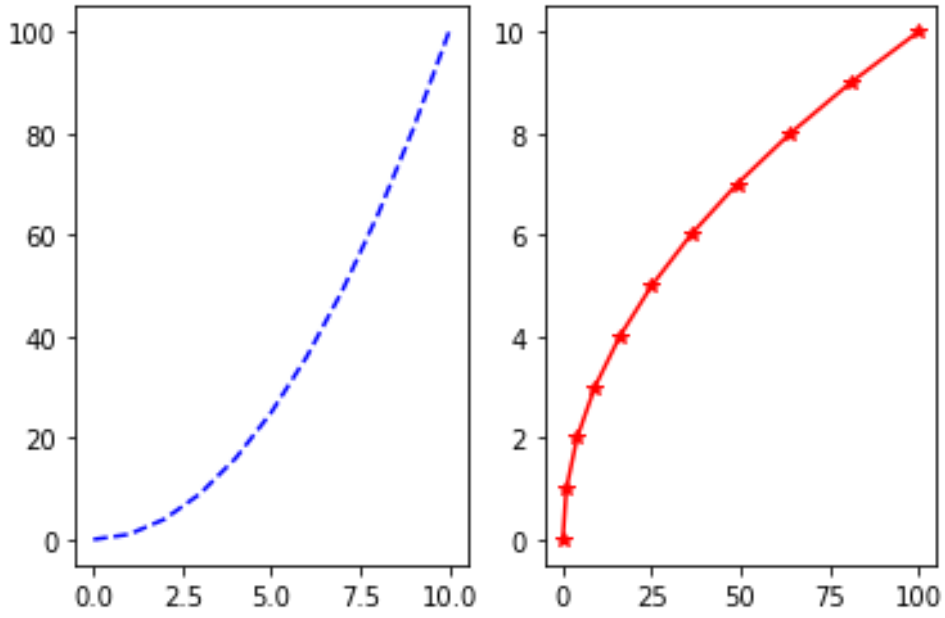
```
plt.plot(x, y, 'r')
# 'r' refers to the color red (More on colors below)
plt.xlabel('Insert X Axis title')
plt.ylabel('Insert Y Axis title')
plt.title('Title Here')
plt.show()
```



Output

Creating Multiplots on the Same Canvas

```
# use like this: plt.subplot(number_of_rows, number_of_columns,
current_plot_number)
plt.subplot(1,2,1)
plt.plot(x, y, 'b--') # More on color options later
plt.subplot(1,2,2)
plt.plot(y, x, 'r*-');
```



Output

Matplotlib Object Oriented Method

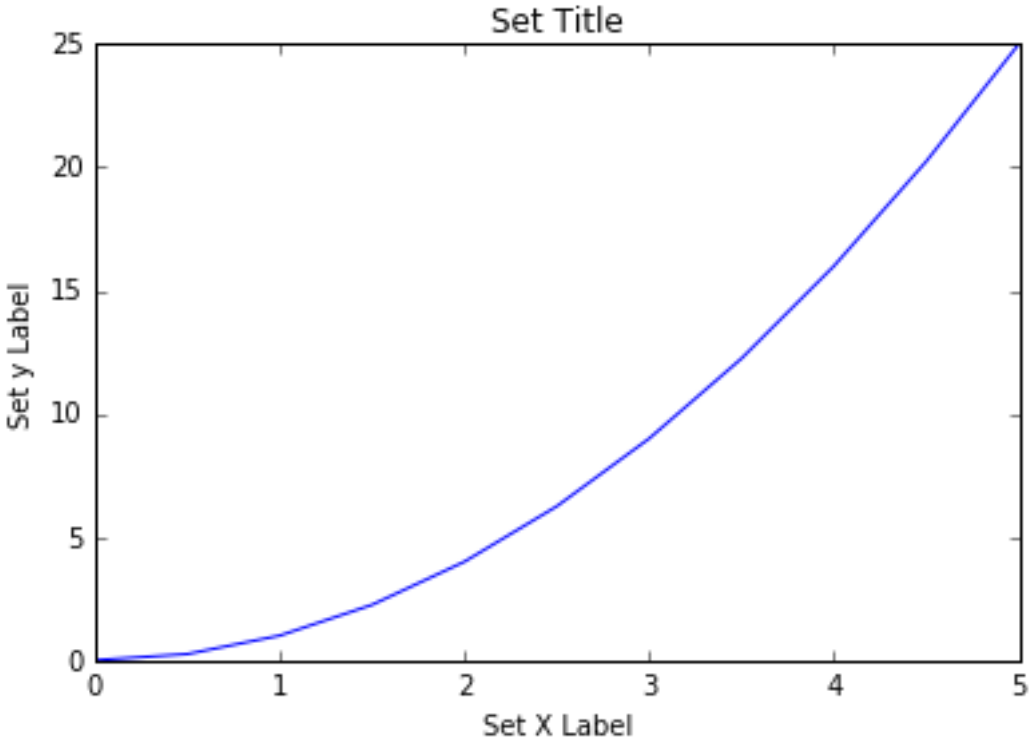
Now that we've seen the basics, let's break it all down with a more formal introduction of Matplotlib's Object-Oriented API. This means we will instantiate figure objects and then call methods or attributes from that object.

Introduction to the Object Oriented Method

Till now, we used Matplotlib in a functional way, i.e. calling functions off of the module. However, the main idea in using the more formal Object Oriented method is to create figure objects and then just call methods or attributes off of that object. This approach is nicer when dealing with a canvas that has multiple plots on it.

To begin we create a figure instance. Then we can add axes to that figure:

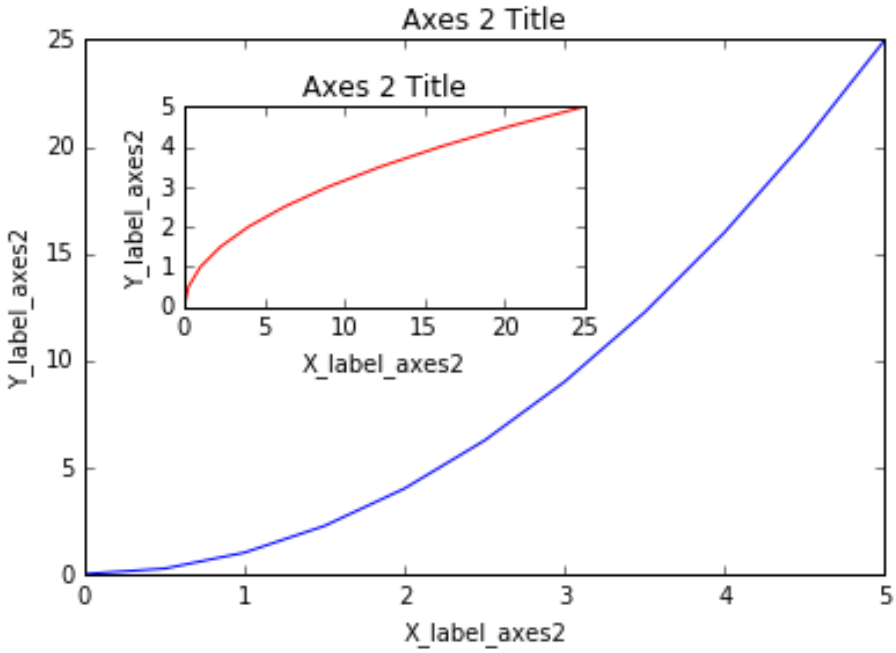
```
# Create Figure (creates an empty canvas)
fig = plt.figure()

# Add set of axes to figure
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range 0
to 1)

# Plot on that set of axes
axes.plot(x, y, 'b')
axes.set_xlabel('Set X Label') # Notice the use of set_ to begin methods
axes.set_ylabel('Set y Label')
axes.set_title('Set Title')
```



Output

As you can see, the code for using object-oriented way is a little more complicated, but the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure:

```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3]) # inset axes

# Larger Figure Axes 1
axes1.plot(x, y, 'b')
axes1.set_xlabel('X_label_axes2')
axes1.set_ylabel('Y_label_axes2')
axes1.set_title('Axes 2 Title')

# Insert Figure Axes 2
axes2.plot(y, x, 'r')
axes2.set_xlabel('X_label_axes2')
axes2.set_ylabel('Y_label_axes2')
axes2.set_title('Axes 2 Title');
```

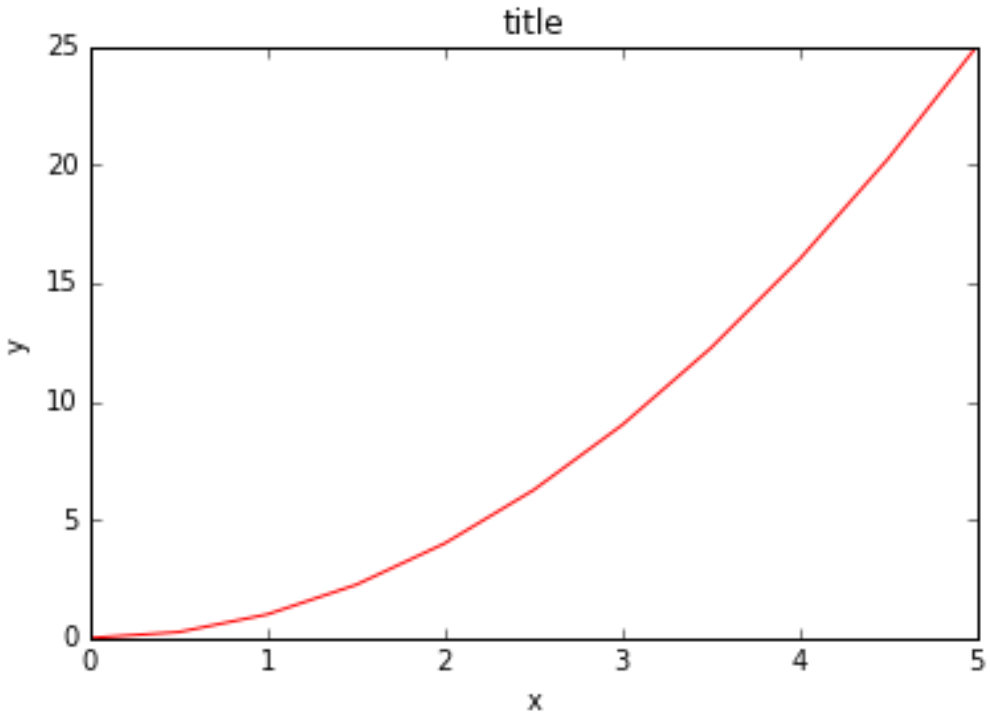


Output

subplots()

The plt.subplots() object will act as a more automatic axis manager. Its basic use cases are shown below.

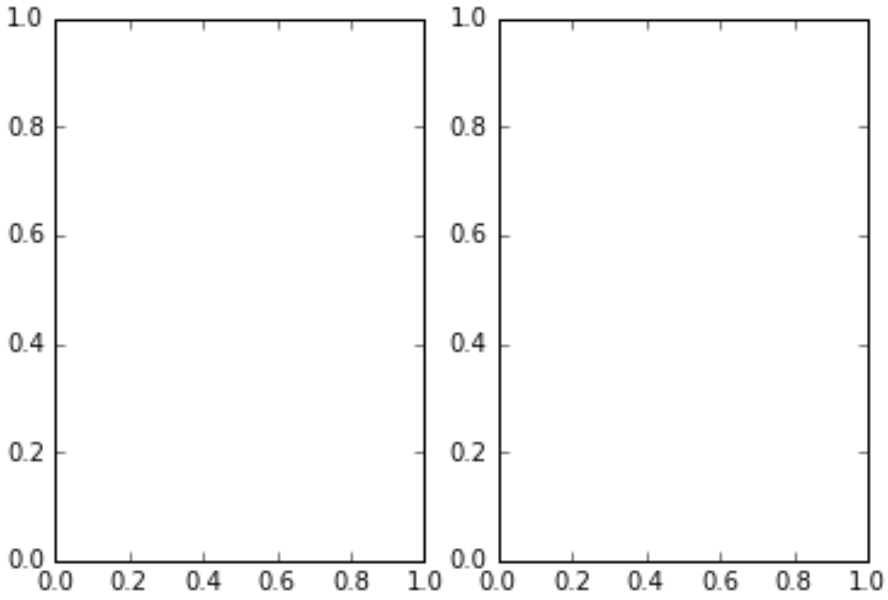
```
# Use similar to plt.figure() except use tuple unpacking to grab fig and axes
fig, axes = plt.subplots()

# Now use the axes object to add stuff to plot
axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



Output

Then you can specify the number of rows and columns when creating the subplots() object.

```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```

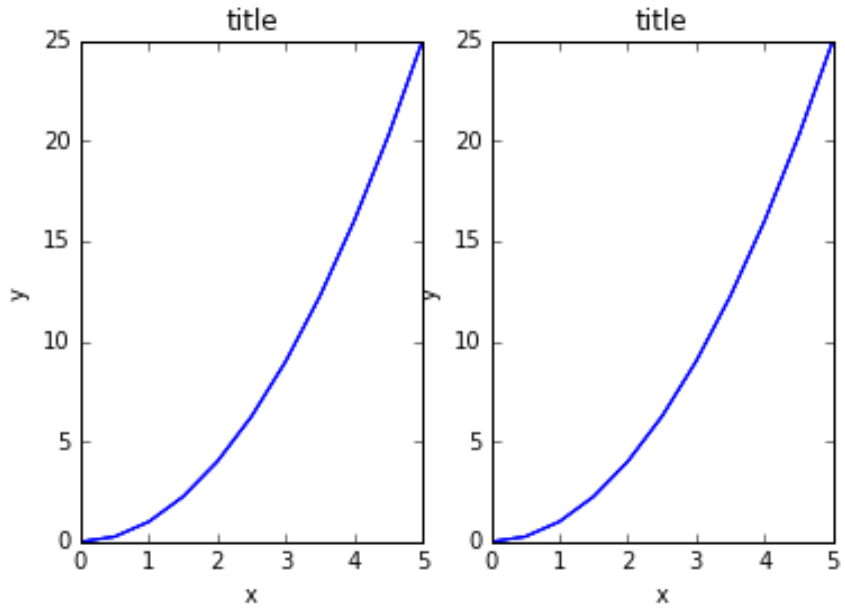


Output

```
# Axes is an array of axes to plot on
axes

array([<matplotlib.axes._subplots.AxesSubplot object at 0x111f0f8d0>,
<matplotlib.axes._subplots.AxesSubplot object at 0x1121f5588>], dtype=object)
```

We can iterate through this array (Axes):

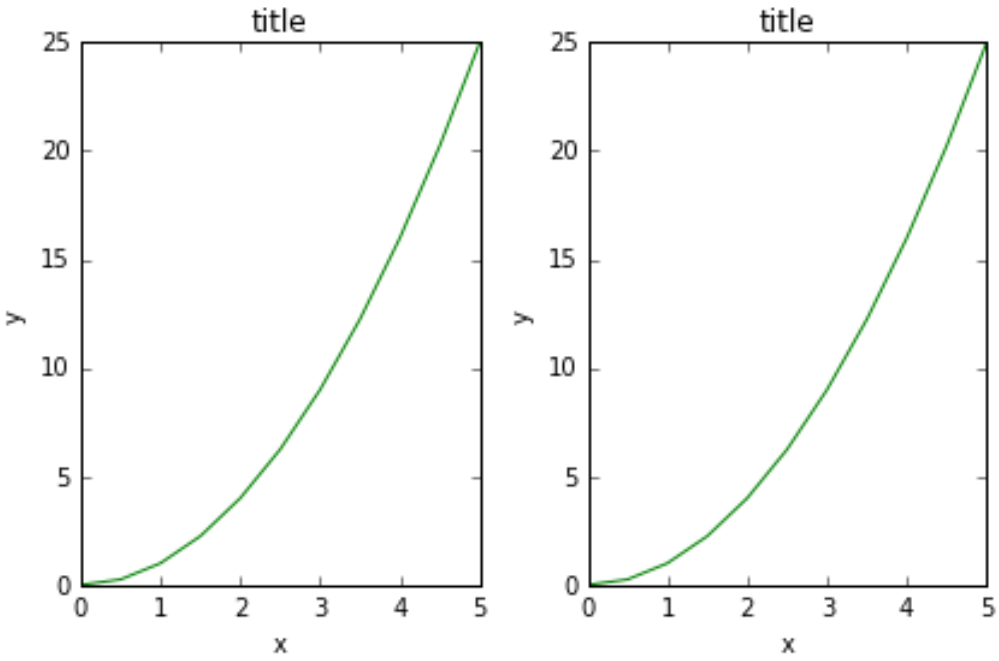
```
for ax in axes:
    ax.plot(x, y, 'b')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

# Display the figure object
fig
```



Output

A common issue with matplolib is **overlapping** subplots or figures. We can use **fig.tight_layout()** or **plt.tight_layout()** method, which automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

```
fig, axes = plt.subplots(nrows=1, ncols=2)

for ax in axes:
    ax.plot(x, y, 'g')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig
plt.tight_layout()
```



Output

Figure Size and Aspect Ratio

Matplotlib allows the aspect ratio and figure size to be specified when the Figure object is created. You can use the figsize keyword argument.

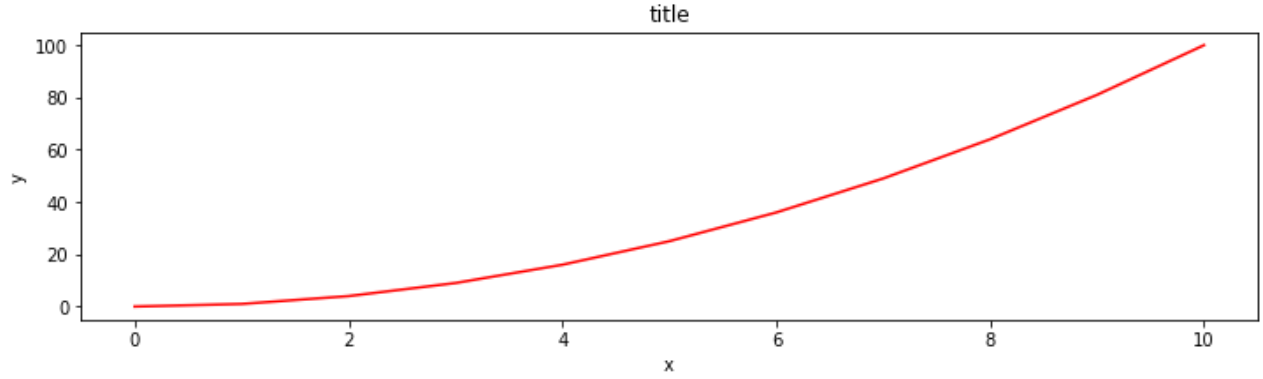
The figsize keyword is a tuple of the width and height of the figure in inches. For example:

```
fig = plt.figure(figsize=(8,4))
```

```
<Figure size 800x400 with 0 Axes>
```

The same arguments can also be passed to layout managers, such as the `subplots` function.

```
fig, axes = plt.subplots(figsize=(12,3))

axes.plot(x, y, 'r')
axes.set_xlabel('x')
axes.set_ylabel('y')
axes.set_title('title');
```



Output

Saving figures

Matplotlib can generate high-quality output in a number of formats, including PNG, JPG, EPS, SVG, PGF and PDF.

To save a figure to a file we can use the `savefig` method in the `Figure` class:

```
fig.savefig("filename.png")
```

Here we can also optionally specify the DPI (Dots per inch) and choose between different output formats.

```
fig.savefig("filename.png", dpi=200)
```

Legends, Labels and Titles

Now that we have covered the basics of how to create a figure canvas and add axes instances to the canvas, let's look at how to decorate a figure with titles, axis labels, and legends.

Figure titles

A title can be added to each axis instance in a figure. To set the title, use the `set_title` method in the axes instance:

```
ax.set_title("title");
```

Axis labels

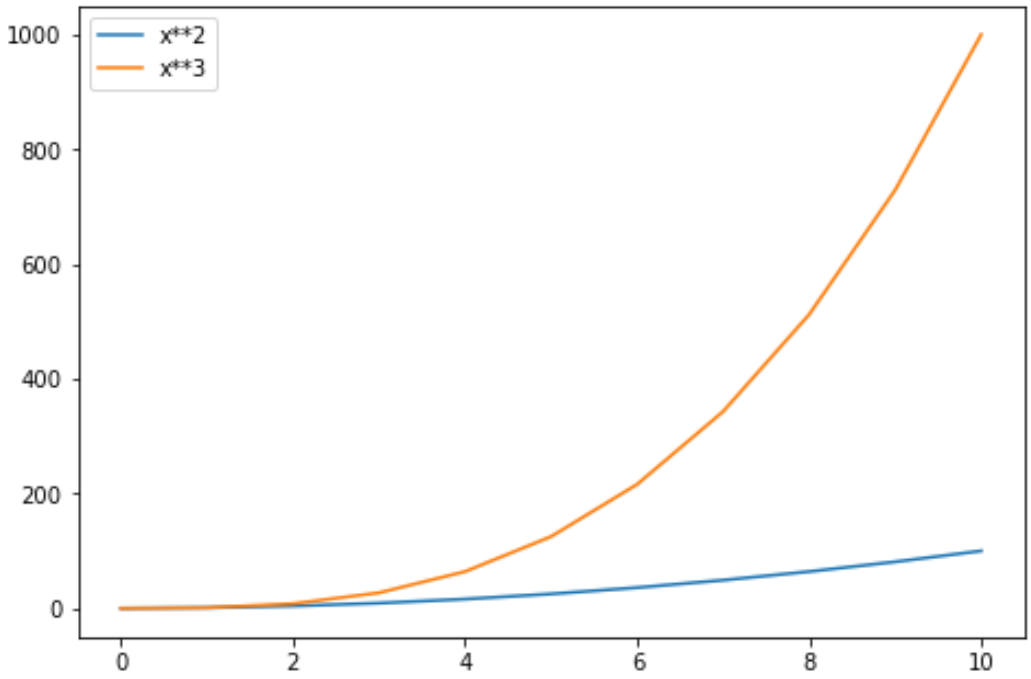
Similarly, with the methods `set_xlabel` and `set_ylabel`, we can set the labels of the X and Y axes:

```
ax.set_xlabel("x")
ax.set_ylabel("y");
```

Legends

You can use the **label="label text"** keyword argument when plots or other objects are added to the figure, and then using the **legend** method without arguments to add the legend to the figure:

```
fig = plt.figure()

ax = fig.add_axes([0,0,1,1])

ax.plot(x, x**2, label="x**2")
ax.plot(x, x**3, label="x**3")
ax.legend()
```

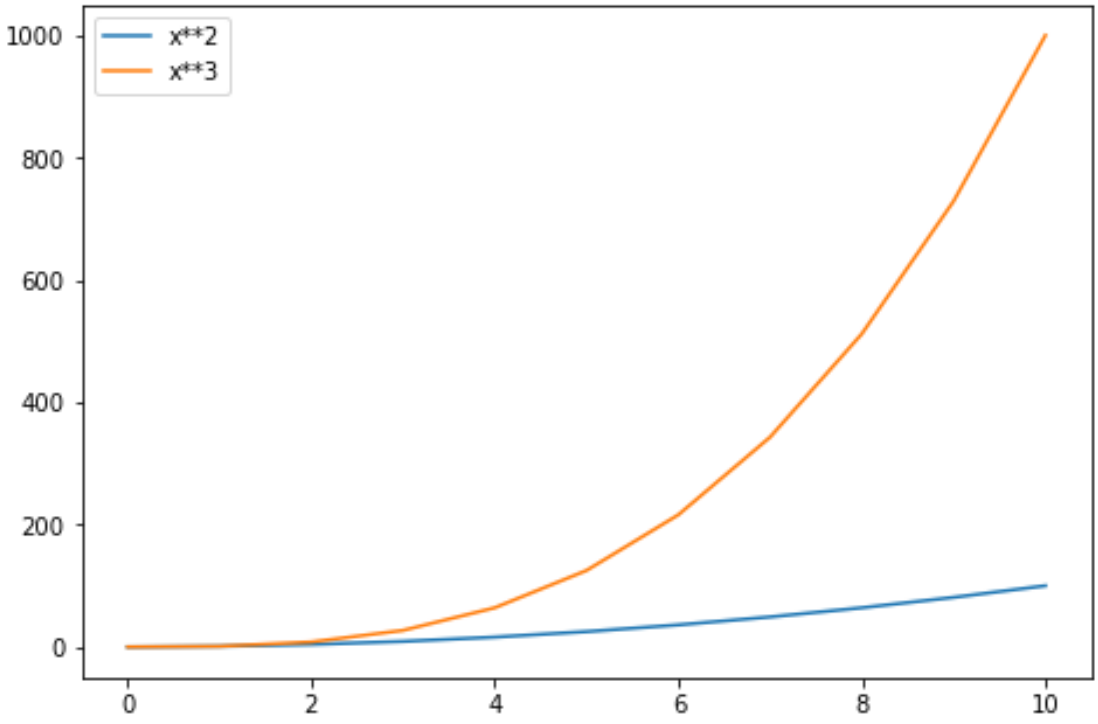


Output

Notice how the legend overlaps some of the actual plot!

The **legend** function takes an optional keyword argument **loc** that can be used to specify where in the figure the legend is to be drawn. The allowed values of **loc** are numerical codes for the various places the legend can be drawn. Some of the most common **loc** values are:

```
# Lots of options....

ax.legend(loc=1) # upper right corner
ax.legend(loc=2) # upper left corner
ax.legend(loc=3) # lower left corner
ax.legend(loc=4) # lower right corner

# .. many more options are available

# Most common to choose
ax.legend(loc=0) # let matplotlib decide the optimal location
fig
```



Output

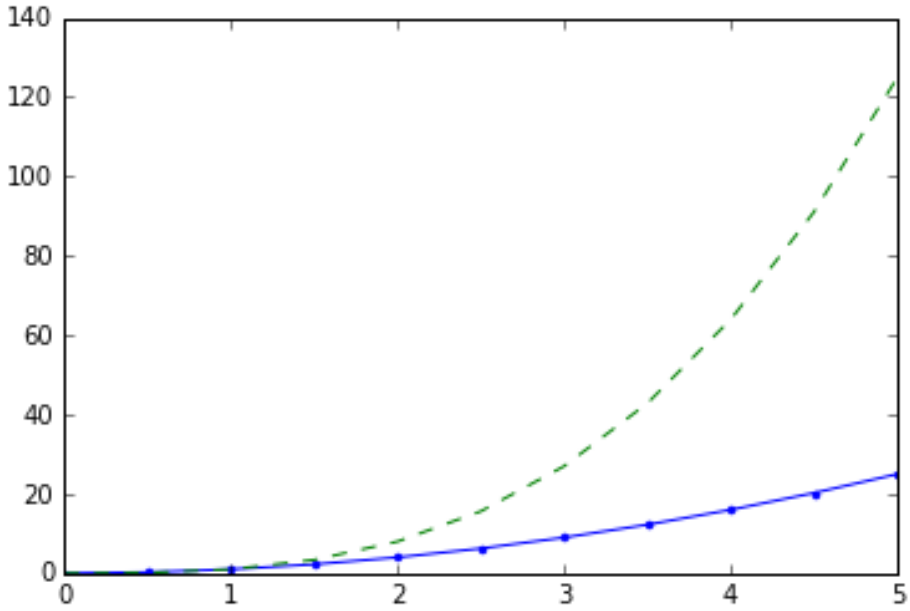
Setting colors, linewidths, linetypes

Matplotlib gives you *a lot* of options for customizing colours, linewidths, and line types.

Colors with MatLab like syntax

With matplotlib, we can define the colours of lines and other graphical elements in a number of ways. First of all, we can use the MATLAB-like syntax where `'b'` means blue, `'g'` means green, etc. The MATLAB API for selecting line styles are also supported: where, for example, 'b.-' means a blue line with dots:

```
# MATLAB style line color and style
fig, ax = plt.subplots()
ax.plot(x, x**2, 'b.-') # blue line with dots
ax.plot(x, x**3, 'g--') # green dashed line
```

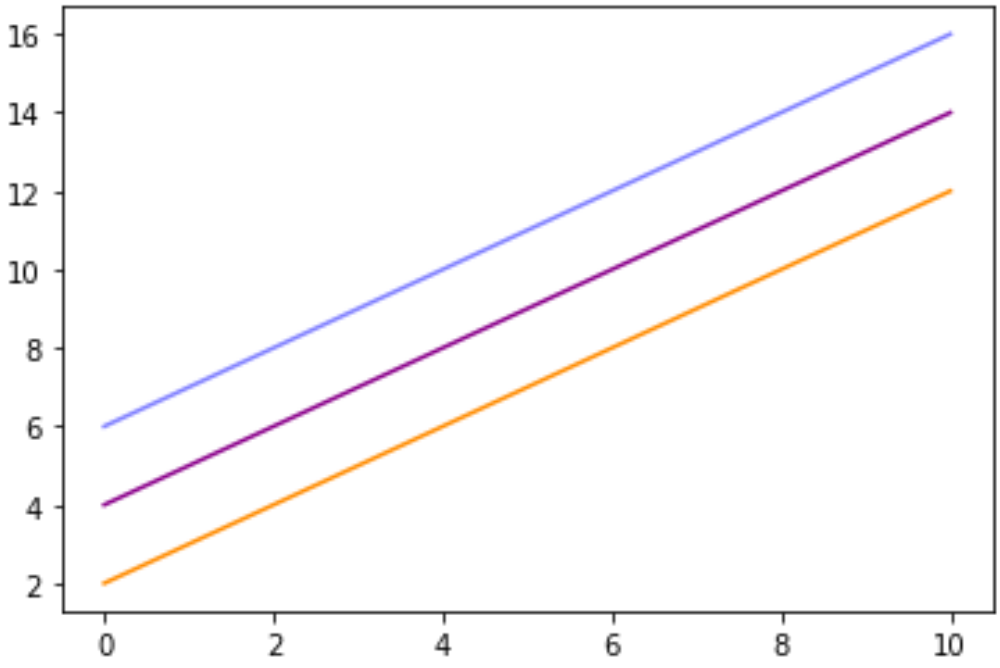


Output

Colors with the color parameter

We can also define colours by their names or RGB hex codes and optionally provide an alpha value using the `color` and `alpha` keyword arguments. Alpha indicates opacity.

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

ax.plot(x, x+6, color="blue", alpha=0.5) # half-transparant
ax.plot(x, x+4, color="#8B008B")        # RGB hex code
ax.plot(x, x+2, color="#FF8C00")        # RGB hex code
```

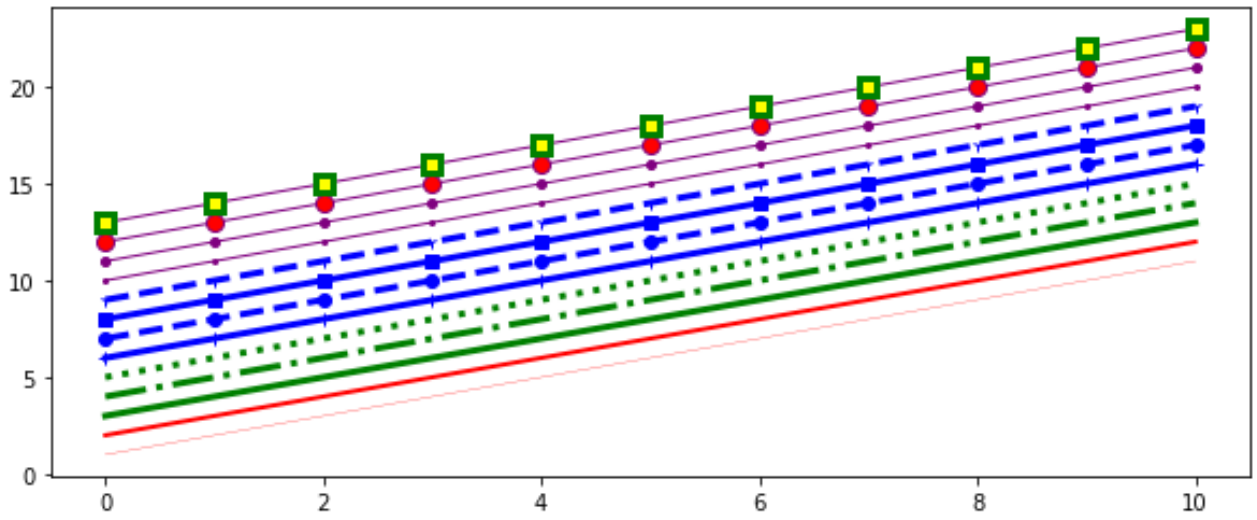


Output

Line and marker styles

To change the line width, we can use the `linewidth` or `lw` keyword argument. The line style can be selected using the `linestyle` or `ls` keyword arguments:

```
fig, ax = plt.subplots(figsize=(10,4))

ax.plot(x, x+1, color="red", linewidth=0.25)
ax.plot(x, x+2, color="red", linewidth=2.00)

# possible linestype options ‘-‘, ‘–’, ‘-.’, ‘:’, ‘steps’
ax.plot(x, x+3, color="green", lw=3, linestyle='-')
ax.plot(x, x+4, color="green", lw=3, ls='-.')
ax.plot(x, x+5, color="green", lw=3, ls=':')


# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3',
'4', ...
ax.plot(x, x+6, color="blue", lw=3, ls='-', marker='+')
ax.plot(x, x+7, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+8, color="blue", lw=3, ls='-', marker='s')
ax.plot(x, x+9, color="blue", lw=3, ls='--', marker='1')

# marker size and color
ax.plot(x, x+10, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+11, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+12, color="purple", lw=1, ls='-', marker='o', markersize=8,
markerfacecolor="red")
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="green");
```



Output

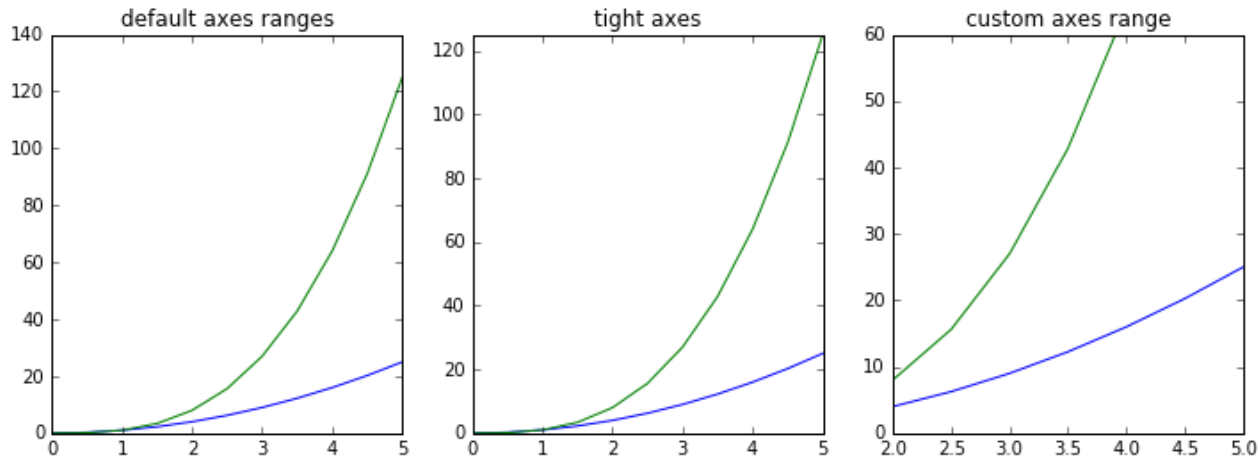
Control Over Axis Appearance

In this section we will look at controlling axis sizing properties in a Matplotlib figure.

Plot range

We can configure the ranges of the axes using the `set_ylim` and `set_xlim` methods in the axis object, or a `xis('tight')` for automatically getting “tightly fitted” axes ranges.

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

axes[0].plot(x, x**2, x, x**3)
axes[0].set_title("default axes ranges")

axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range");
```



Output

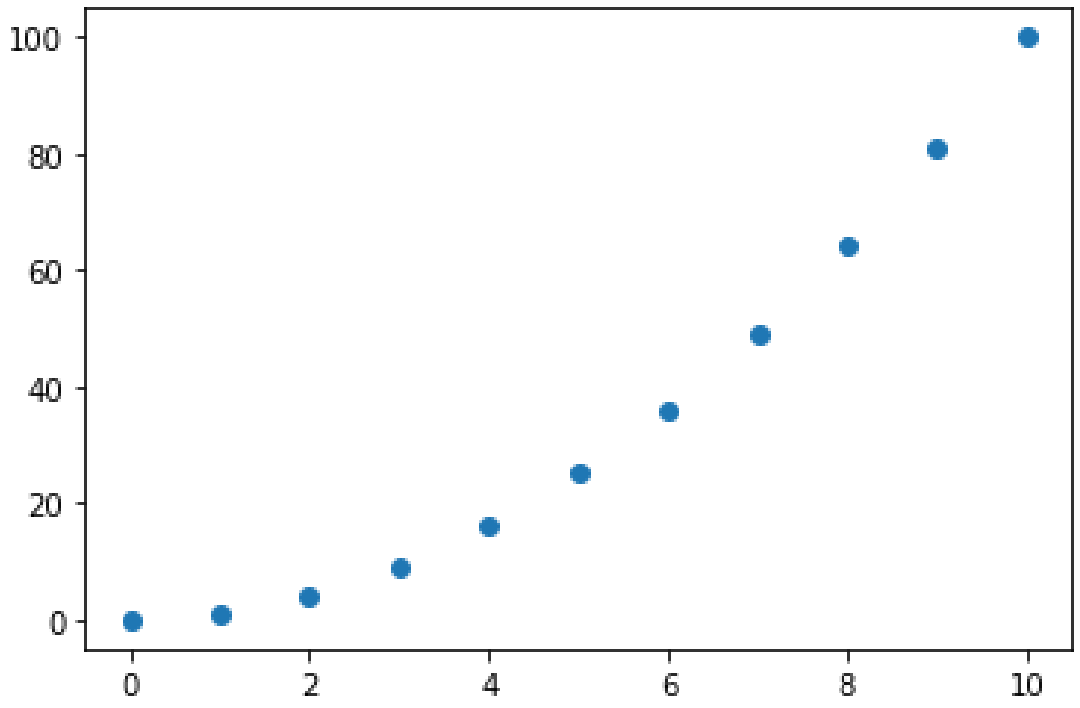
Special Plot Types

There are many specialized plots we can create, such as bar plots, histograms, scatter plots, and much more. Most of these types of plots can be easily created using **seaborn**, a statistical plotting library for Python. But here are a few examples of these types of plots:

Scatter Plots

```
plt.scatter(x,y)
```

```
<matplotlib.collections.PathCollection at 0x268c6a21880>
```

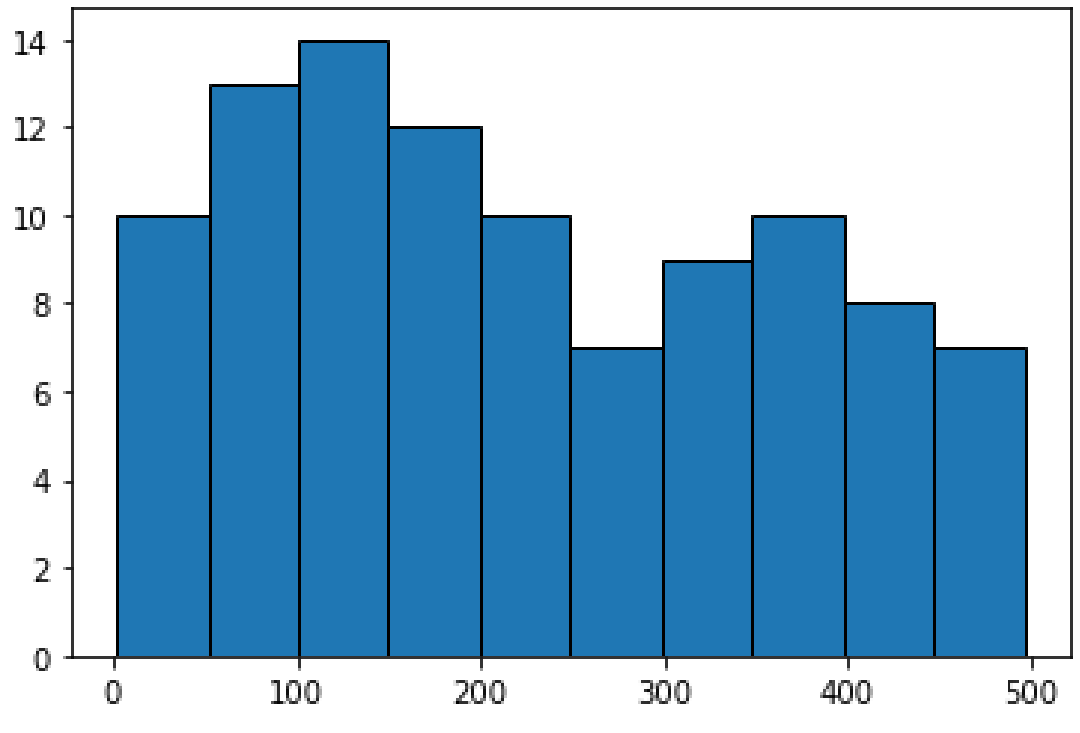


Output

Histograms

```
from random import sample
data = sample(range(1, 500), 100)
plt.hist(data,edgecolor='black')
```

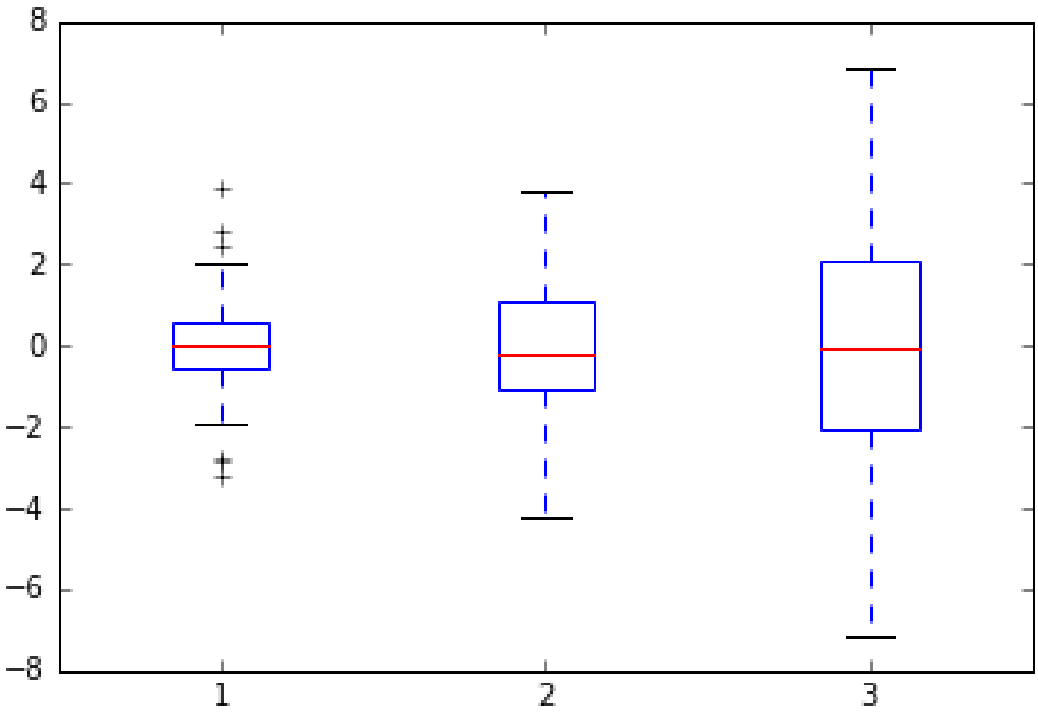
```
(array([10., 13., 14., 12., 10.,  7.,  9., 10.,  8.,  7.]),  array([  2. ,  51.4,
100.8, 150.2, 199.6, 249. , 298.4, 347.8, 397.2,         446.6, 496. ]),
<BarContainer object of 10 artists>
```



Output

Box Plots

```
data = [np.random.normal(0, std, 100) for std in range(1, 4)]

# rectangular box plot
plt.boxplot(data,vert=True,patch_artist=True);
```



Output

Pie Charts

For plotting of other common charts/plots, Seaborn is the better option to go for as it is relatively simpler to use and visualization can be done with fewer lines of code. However, pie charts aren't available to use directly in Seaborn. Therefore, we'll have a look at making pie charts in Matplotlib.

Shown below is a series of pie charts I personally used to visualize the demographics of the final year students at my college. At my college, there are 193 final year B.Tech students (Out of which 123 are in the CSE branch and 70 are in the ECE branch) and 39 M.Tech students (Out of which 32 belong to the CSE branch and 7 belong to the ECE branch).

Hence, first we'll create the data to be visualized accordingly. To do so, we'll import Pandas and move up from there:

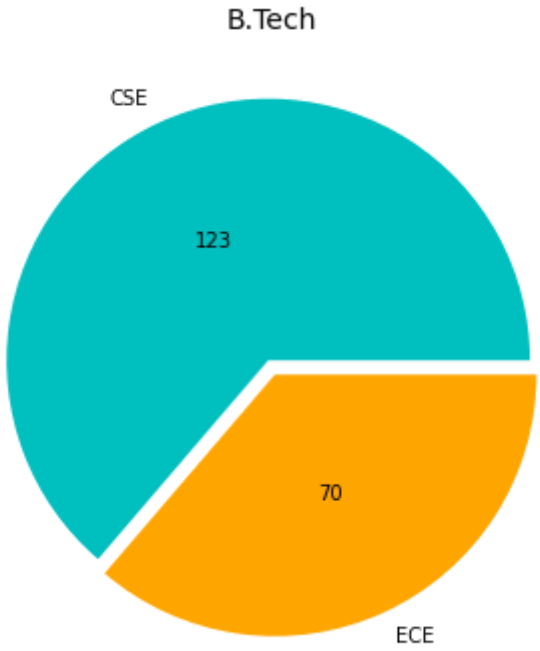
```
import pandas as pd

dataB = pd.DataFrame(data=[123, 70], index=['CSE', 'ECE'],columns=['Number of
Students'])
dataB
```

```
Number of Students
CSE   123
ECE    70
```

Creating the Pie Chart and Saving the figure

```
pie, ax = plt.subplots(figsize=[10,6])
total = sum(dataB['Number of Students'])
labels = ['CSE','ECE']
color_set=["c","orange"]
plt.pie(x=dataB['Number of Students'], autopct=lambda s: '{:.0f}'.format(s * total
/ 100), explode=[0.03]*2, labels=labels, colors = color_set, pctdistance=0.5)
plt.title("B.Tech", fontsize=14);
pie.savefig("B.Tech.png")
```

As you can see, the pie chart is exploding since we have passed in the attribute 'explode=value'. We can do the same process for M.Tech demographics.



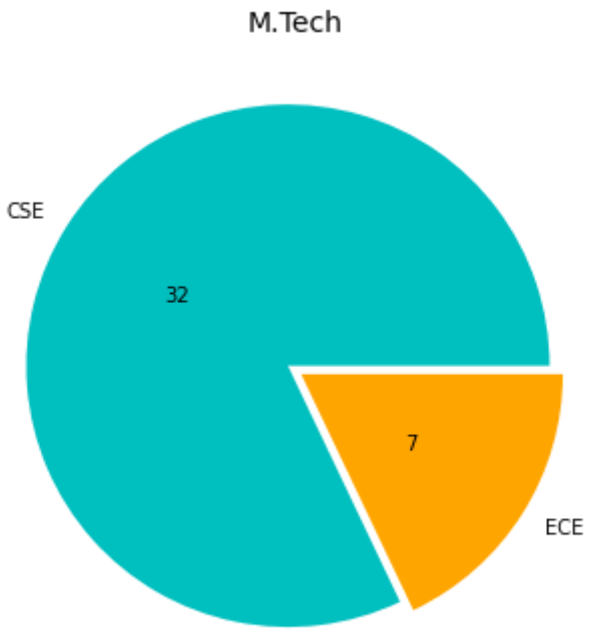
Output

```
dataM = pd.DataFrame(data=
[32,7],index=['CSE','ECE'],columns=
['NoS'])
pie, ax = plt.subplots(figsize=
(10,6))
total = sum(dataM['NoS'])
labels = ['CSE','ECE']
color_set=["c","orange"]
plt.pie(x=dataM['NoS'],
autopct=lambda s: '{:.0f}'.format(s
* total / 100), explode=[0.03]*2,
labels=labels, colors = color_set,
pctdistance=0.5)
plt.title("M.Tech", fontsize=14);
pie.savefig("M.Tech.png")
```

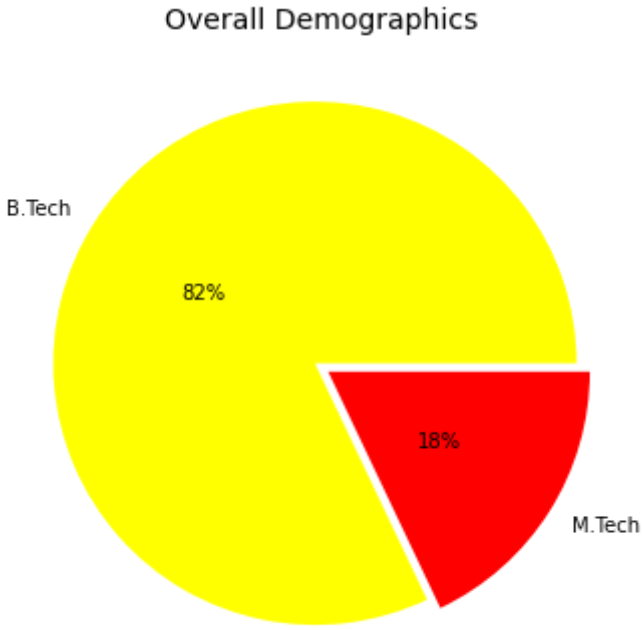
Another Pie Chart to sum it up – let's plot the overall demographics! Also notice that this time, instead of passing in the raw numbers as labels to the pie chart we are simply passing in **percentage** values.

```
pie, ax = plt.subplots(figsize=(10,6))
total = sum(data['NoS'])
labels = ['B.Tech','M.Tech']
color_set=["yellow","red"]
plt.pie(x=dataM['NoS'], autopct=lambda s: '{:.0f}%'.format(s), explode=
[0.03,0.03], labels=labels, colors = color_set, pctdistance=0.5)
plt.title("Overall Demographics", fontsize=14);
pie.savefig("Total.png")
```

Well, thats it folks! We hope that this comprehensive coverage of Matplotlib is just the beginning of your journey to visualizing data in Python.



Output



Output

Further reading

- http://www.matplotlib.org – The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib – The source code for matplotlib.
- http://matplotlib.org/gallery.html – To view all the possibilities with Matplotlib, visit this gallery – Highly recommended!

iamajit.pythonanywhere.com