Data Visualization with MATPLOTLIB

Well, as a backend lover, I have to say this is not article that I'm happy to make. Visualization is definitely not my strong point. But as we've learned about data analysis tools like `numpy` and `pandas` and how to use them, I should make a artcile on `data visualization` tools too. So, The most popular data visualization and also the the most customizable tool is `matplotlib`. It is a very powerful tool that can do a lot of things and I'll try my best to cover all the things I know about it.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Installation

If you have `conda` installed, Just run the following command in your terminal:

```
conda install matplotlib
```

Or if you dont't have `conda` installed, You can also run `pip install matplotlib` in your terminal.

Introduction

`Matplotlib` is like the python version of `MatLab`. `MATLAB` is a tool for `data analysis` and visualization. `Matplotlib` is exactly that but in code. It is a very powerful tool that can do a lot of things. It can do so many things that I don't know even 30% of them. You can check out the gallery for more information and code examples of different plots and how to customise them.

But first just try it out.

> Okay quick explanation before we run this cell. Try to read the code once, then run it, and then change one thing (like `bins` or `marker`) to see how the plot reacts.

Let's import the `matplotlib` library:

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

Let's walk through a very simple example using two numpy arrays:

In [2]:
```
import numpy as np
x = np.arange(5, 20)
y = x*3 + 1
```

In [3]:
```
x
```

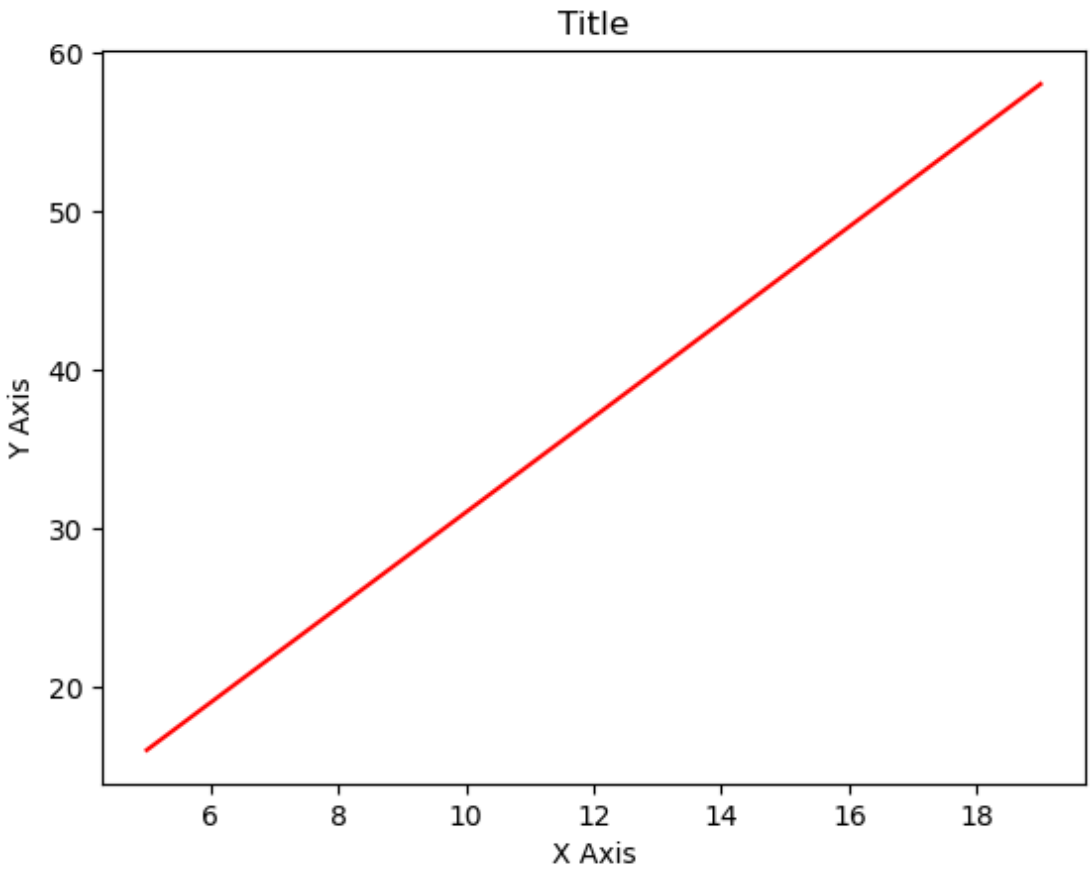
Out[3]:
```
array([ 5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
```

In [4]:
```
y
```

Out[4]:
```
array([16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58])
```

Now we can create a basic line plot using `matplotlib.pyplot.plot`.

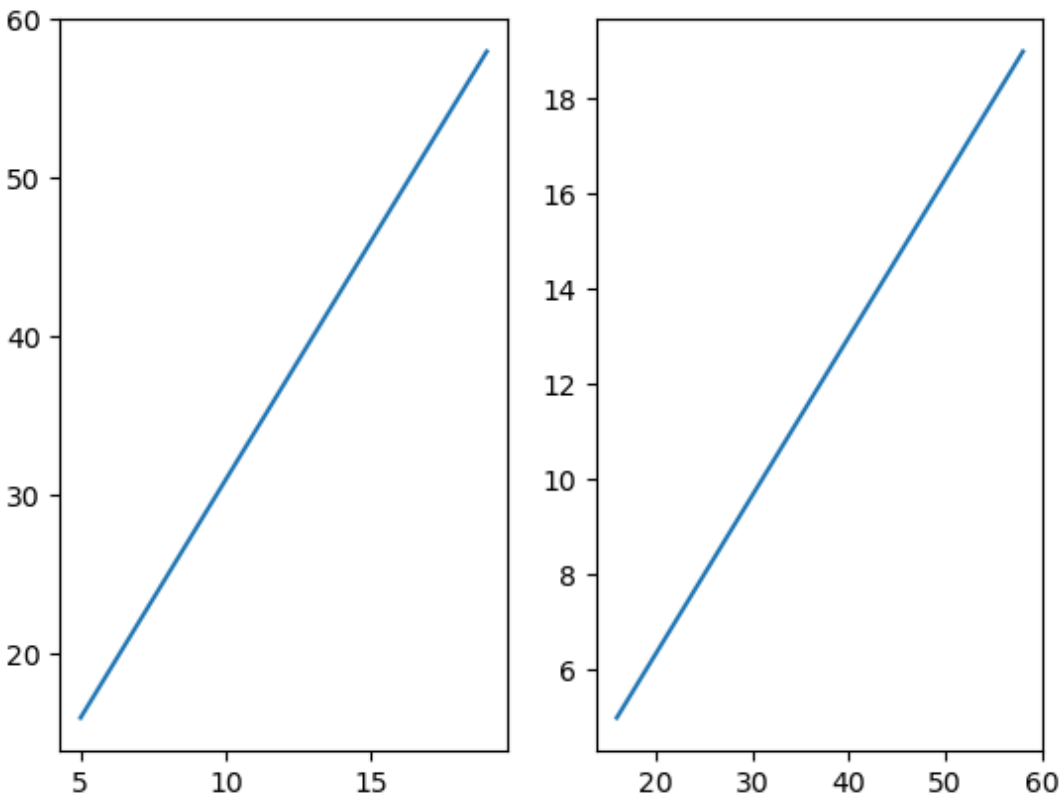
In [5]:
```
plt.plot(x, y, 'r') # 'r' is for red
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Title')
plt.show()
```



> the `plot()` method is used to create a line plot. The first argument is the x-axis values, the second argument is the y-axis values, and the third argument is the color of the line.

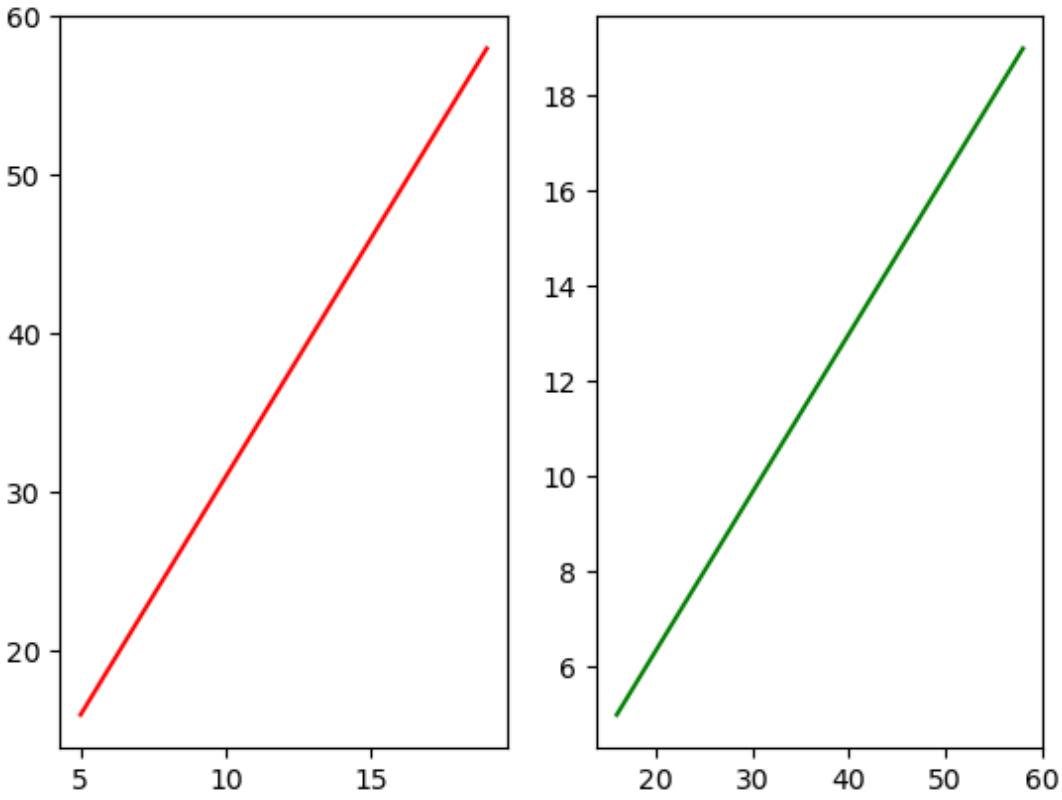
One good thing about `plt` is that it gives us a the ability to create multiple plots on the same figure. It's called a `subplot`.

In [6]:
```
plt.subplot(1,2,1)
plt.plot(x, y)
plt.subplot(1,2,2)
plt.plot(y, x);
plt.show()
```



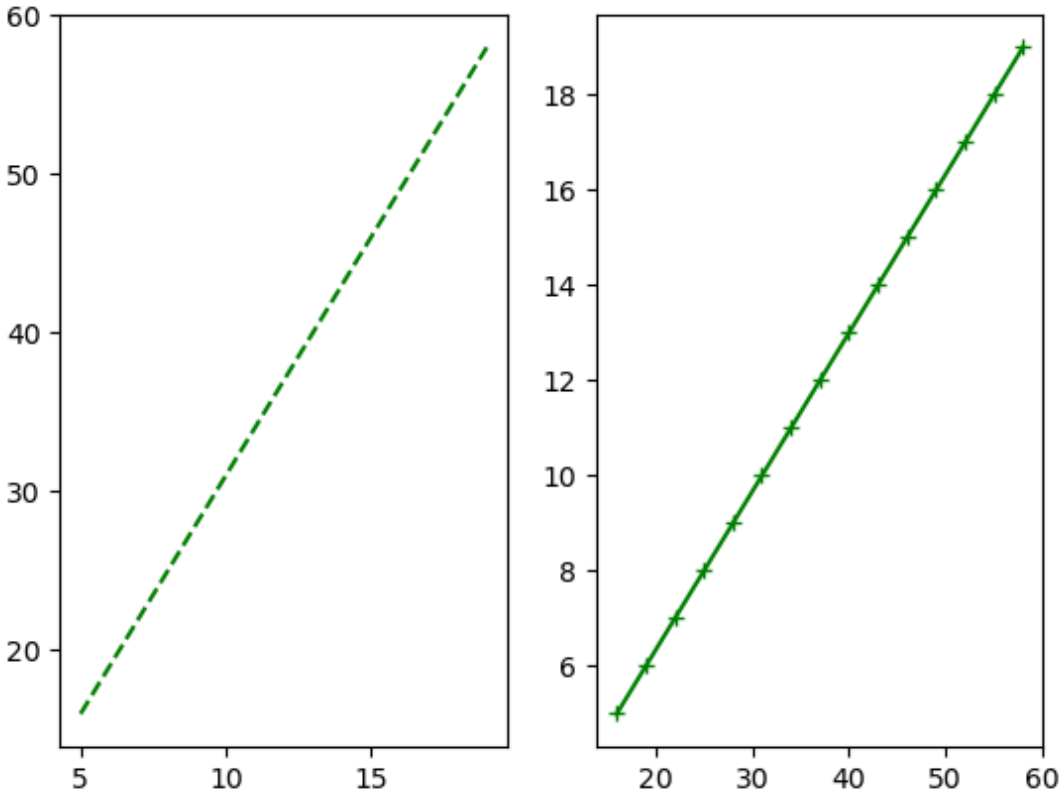
By default the color is set to `blue`. You can change the color by adding a third argument to the `plot()` method.

In [7]:
```
plt.subplot(1,2,1)
plt.plot(x, y, 'r')
plt.subplot(1,2,2)
plt.plot(y, x, 'g');
plt.show()
```



Another fun thing is you can specify hwo the line should look like in the same color argument as a string like this.

In [8]:
```
plt.subplot(1,2,1)
plt.plot(x, y, 'g--')
plt.subplot(1,2,2)
plt.plot(y, x, 'g+-');
plt.show()
```



Now let's break down the code. First we use the `subplot()` to specify how many rows and columns we want. I passed `1,2` meaning `1` row and `2` columns. Then we pass `1` to specify the plot number. So for each row it goes left to right.

So, I first specified the first plot and then the second plot. The first plot is on the left and the second plot is on the right.

And call the `show()` method to display the plot.

Object Oriented Aproach

Matplotlib has a lot of functions built in that can help generate a lot of plots for good visualization but the main `functionalities` and `customizability` is in the `object oriented approach`.

Matplotlib gives us a function called `figure` that returns a `figure` object. This figure represents an empty canvas where we can add axes to it.

What is an axes? It is a `set of positions` on the figure where we can plot things.

Let's see what it is.

In [9]:
```
fig = plt.figure()
fig
```

Out[9]:
```
<Figure size 640x480 with 0 Axes>
```
<Figure size 640x480 with 0 Axes>

Well, here you see the figure object. Now let's define a axis.

In [10]:
```
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])
axes
```

Out[10]:
```
<Axes: >
```

Inside the `add_axis` method we can pass the dimentions of the axes.

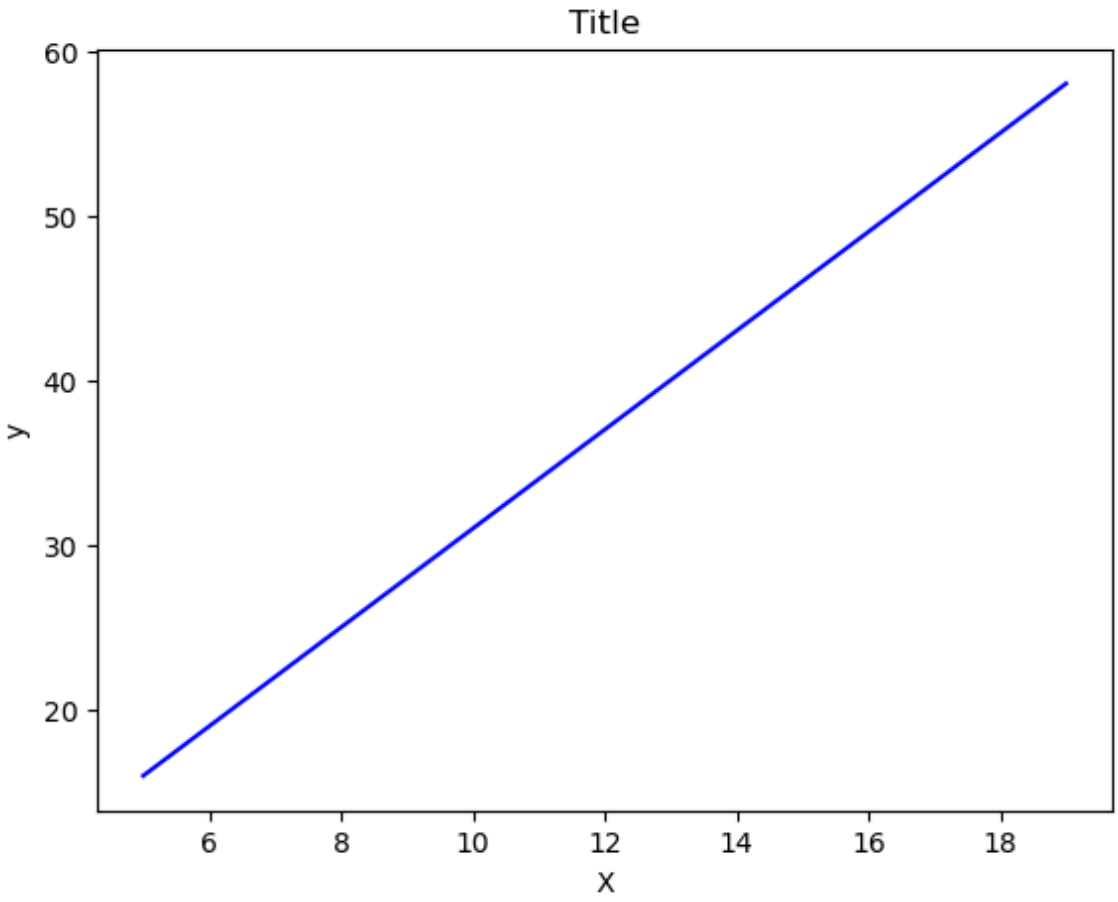
> The dimentions are the left, bottom, width, height. These values are the range of the axes.

These values define the size of the canvas.

Now we can create plots just like we did before.

In [11]:
```
fig = plt.figure()
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])

axes.plot(x, y, 'b')
axes.set_xlabel('X')
axes.set_ylabel('y')
axes.set_title('Title')
```

Out[11]: Text(0.5, 1.0, 'Title')



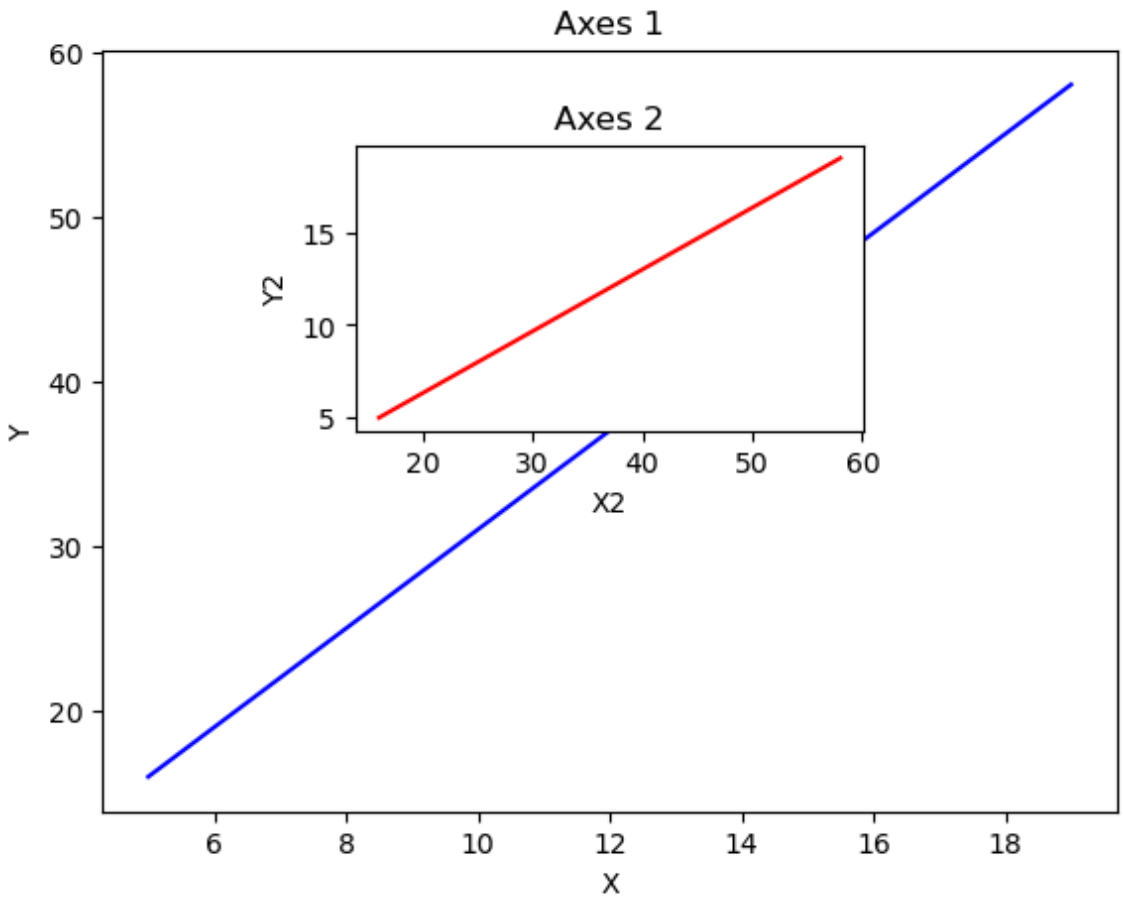
> To set the title, label, or color of the `axes` we use the `set_title`, `set_xlabel`, and `set_ylabel` methods.

> Remember figure is the empty canvas and when you want to create a plot you cannot use an instance of a figure from another cell. You have to create another figure and add axes to it then plot it in the same cell.

This may look complex but in the long run it gives you full control of the canvas.

That's why we can do plots like this.

In [12]:
```
fig = plt.figure()

axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])
axes2 = fig.add_axes([0.3, 0.5, 0.4, 0.3])

axes1.plot(x, y, 'b')
axes1.set_xlabel('X')
axes1.set_ylabel('Y')
axes1.set_title('Axes 1')

axes2.plot(y, x, 'r')
axes2.set_xlabel('X2')
axes2.set_ylabel('Y2')
axes2.set_title('Axes 2');
```



Multiple axes, one inside another. We just define the dimensions and plot the values.

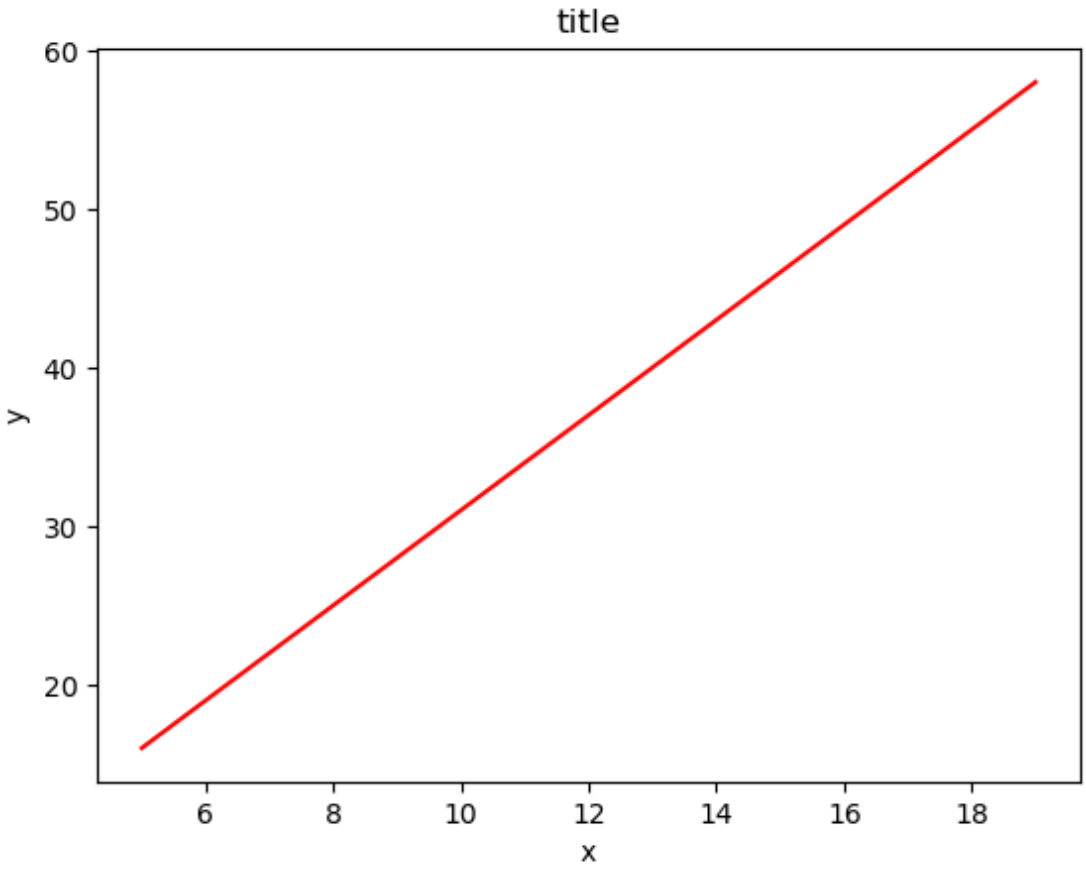
subplots()

It's not the `subplot()` method, there is an `s` at the end.

The `subplots()` method returns a tuple of figure and axes.

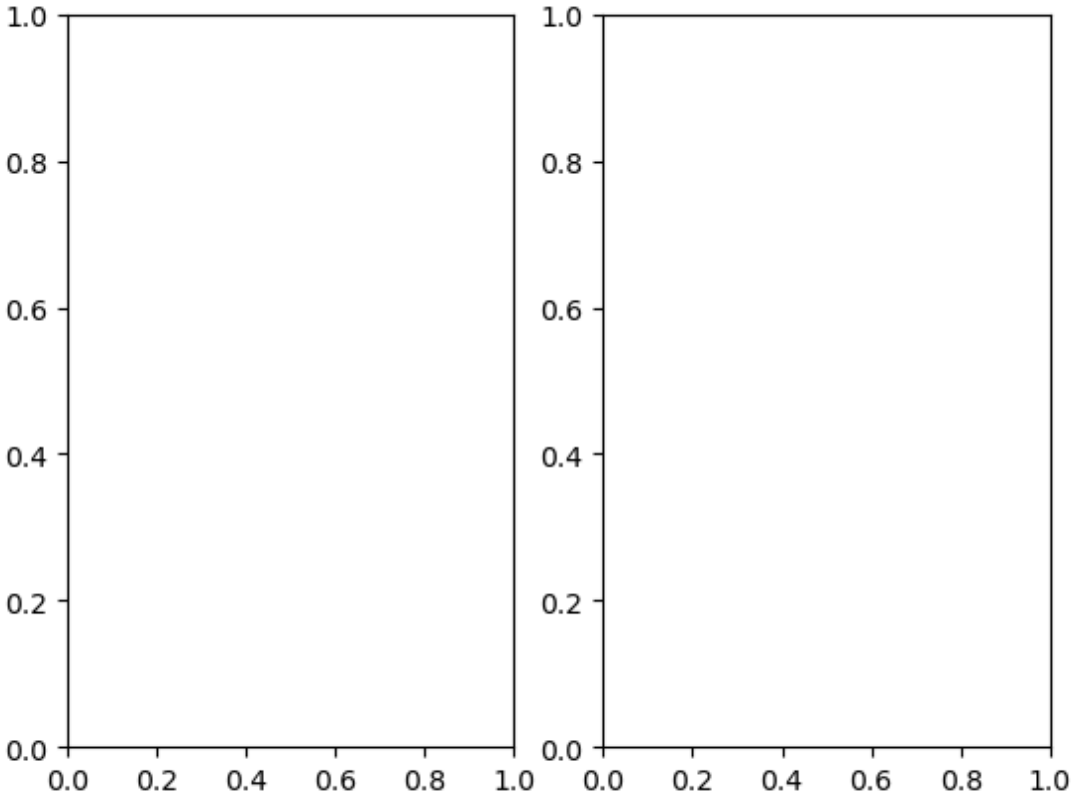
We can use the figure to set size and dpi and other things and the `axes` to add plots.

In [13]:
```
fig, axes = plt.subplots()

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



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

In [14]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)
```



When we use `nrows` and `ncols` to specify the number of rows and columns, it returns an array of axes.

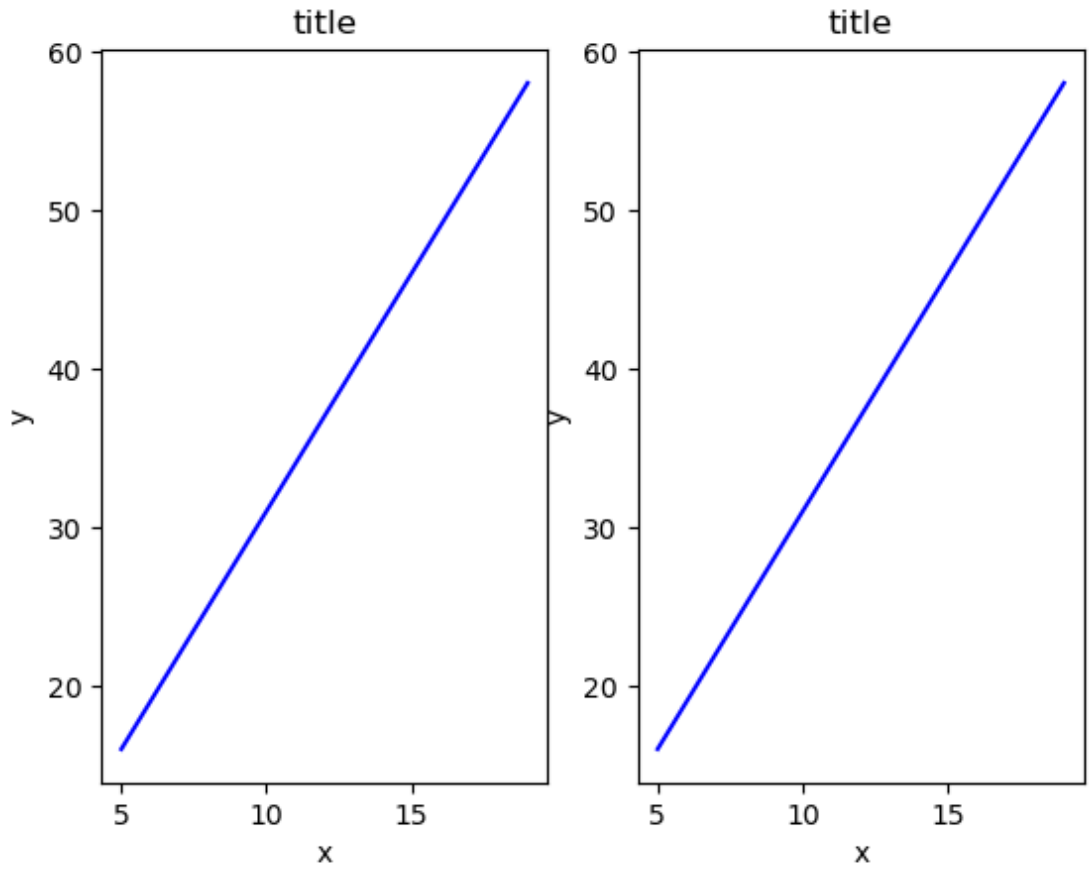
In [15]:
```
axes
```

Out[15]:
```
array([<Axes: >, <Axes: >], dtype=object)
```

WE can iterate over the axes and add plots to them.

In [16]:
```
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
```

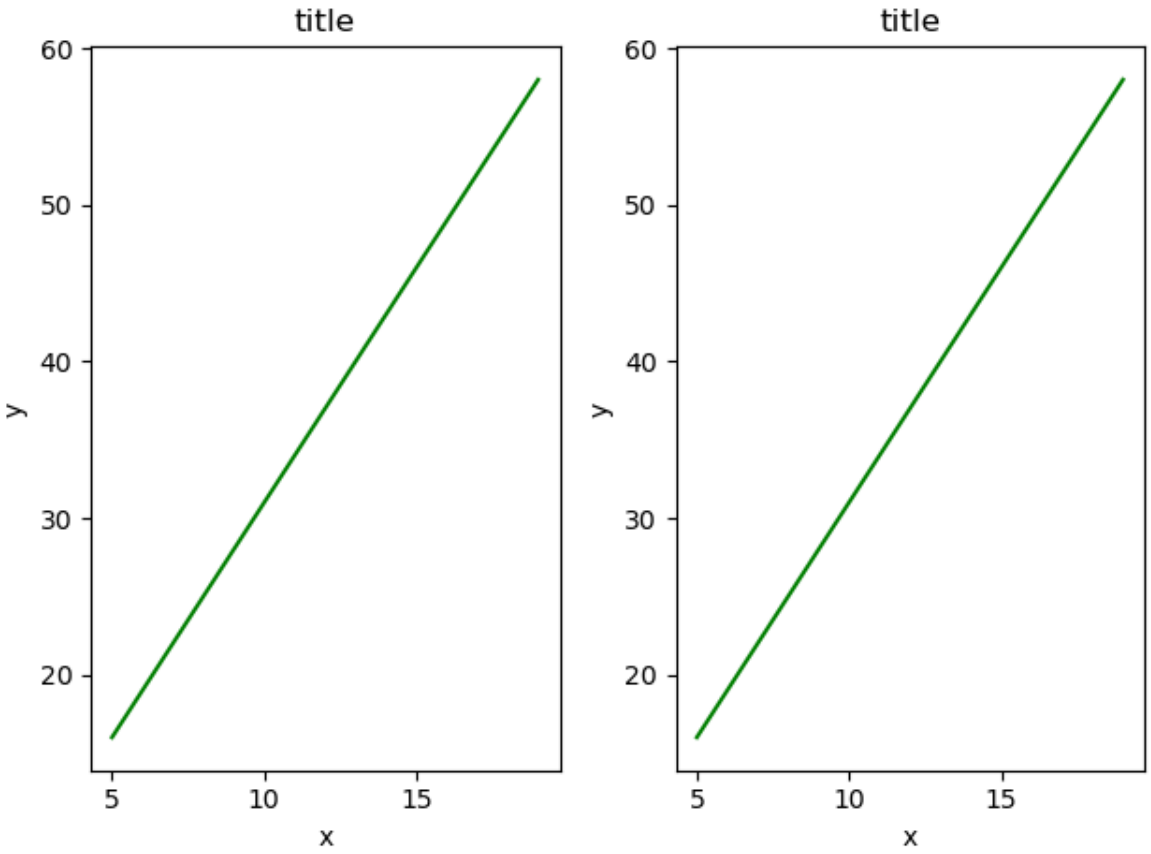
Out[16]:



Sometimes, when you have multiple plots, you might see some overlapping. You can use the `tight_layout()` method to fix that.

In [17]:
```
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()
```



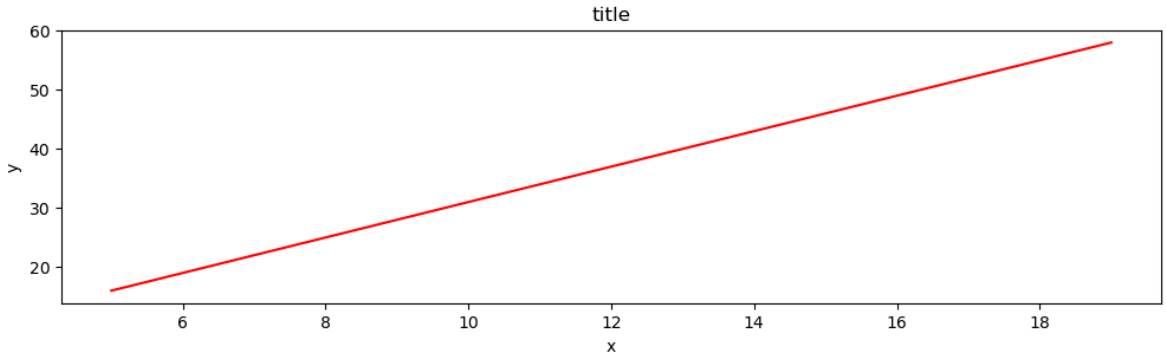
There's some things I haven't talked about Like the `figsize` and `dpi` arguments.

- `figsize` is the size of the figure. We can pass a tuple of width and height to this argument.
- `dpi` is the resolution of the figure. We can pass a number to this argument. DPI stands for `dots per inch`.

In [18]:
```
fig = plt.figure(figsize=(8,4), dpi=100)
```

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

In [19]:
```
fig, axes = plt.subplots(figsize=(12,3))

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



Try changing these two argument values and see how the plot changes.

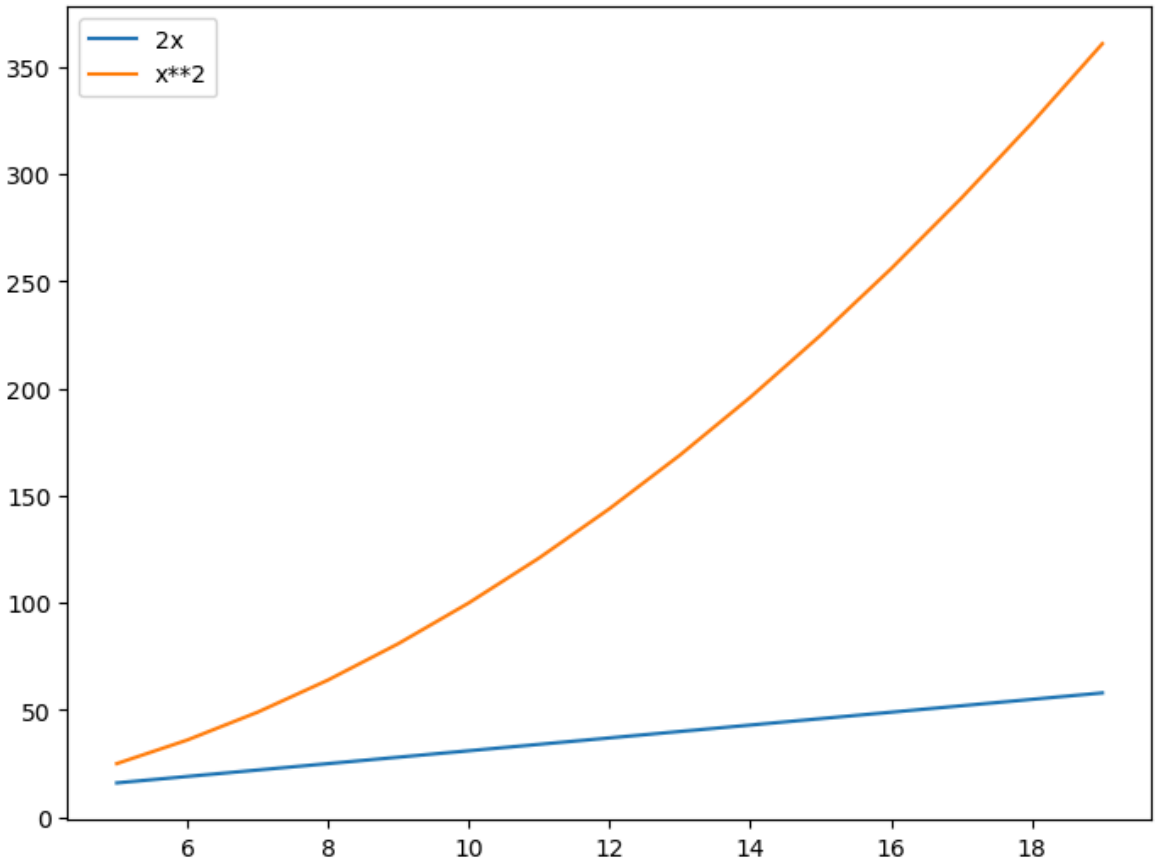
Legends

Legend is a feature that helps us distinguish between different plots. If you have multiple plots or a comparison between multiple plots, you can add a `legend` to the plot.

Legend takes the `labels` as an argument which is passed to the `plot` method and after that we can call the `legend` method from the axes object Which will automatically generate a top box with the labels and colors of the plots.

In [20]:
```
fig = plt.figure()

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

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

Out[20]:
```
<matplotlib.legend.Legend at 0x7d0f248f8c10>
```

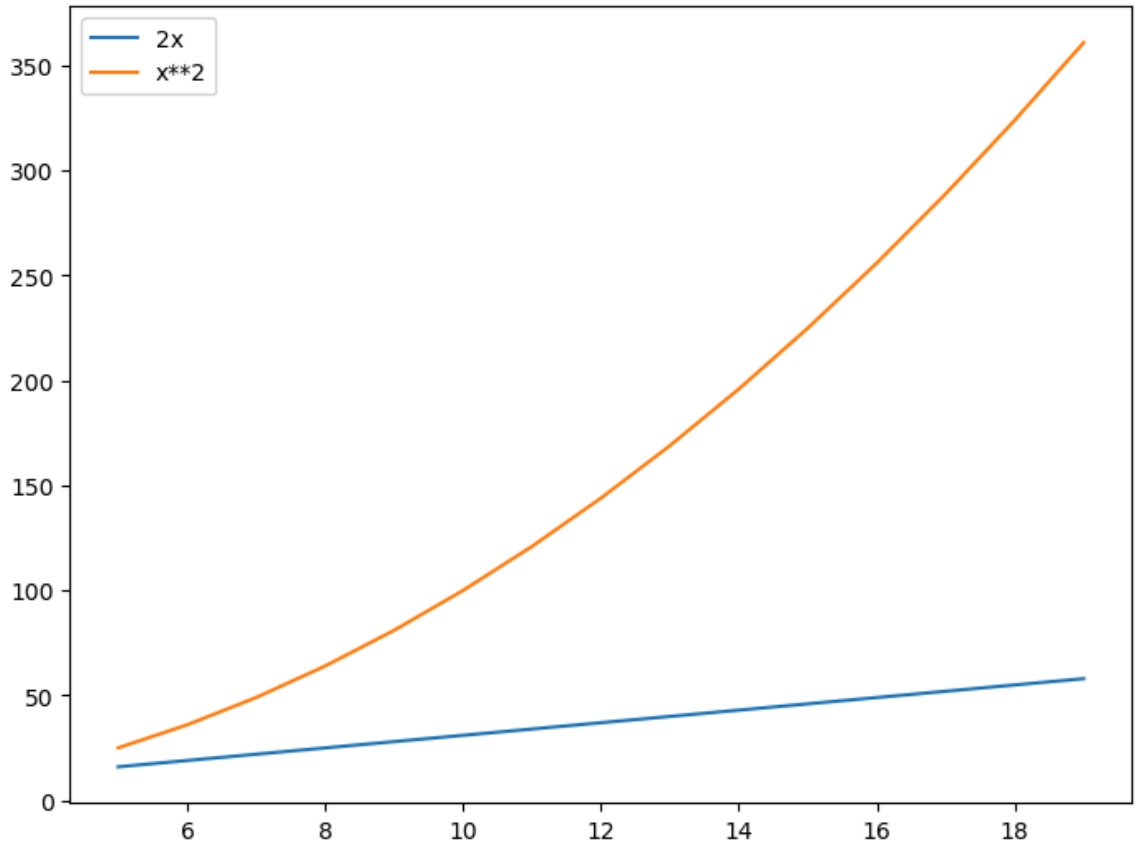


See, the colors are different and there is a box indicating the which plot it is.

There might be scenarios where the `lagend` box overlaps with the plot. We can use the `loc` argument to specify the location of the legend.

In [21]:
```
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

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

Out[21]:



> Note: The default value of `loc` is 0.

> `loc=1` means the legend will be placed in the upper right corner of the figure.

> `loc=2` means the legend will be placed in the upper left corner of the figure.

> `loc=3` means the legend will be placed in the lower left corner of the figure.

> `loc=4` means the legend will be placed in the lower right corner of the figure.

> `loc=0` means the legend will be placed in a good location based on the figure size and plot size.lll

Colors

Colors can be specified using the `color` keyword argument.

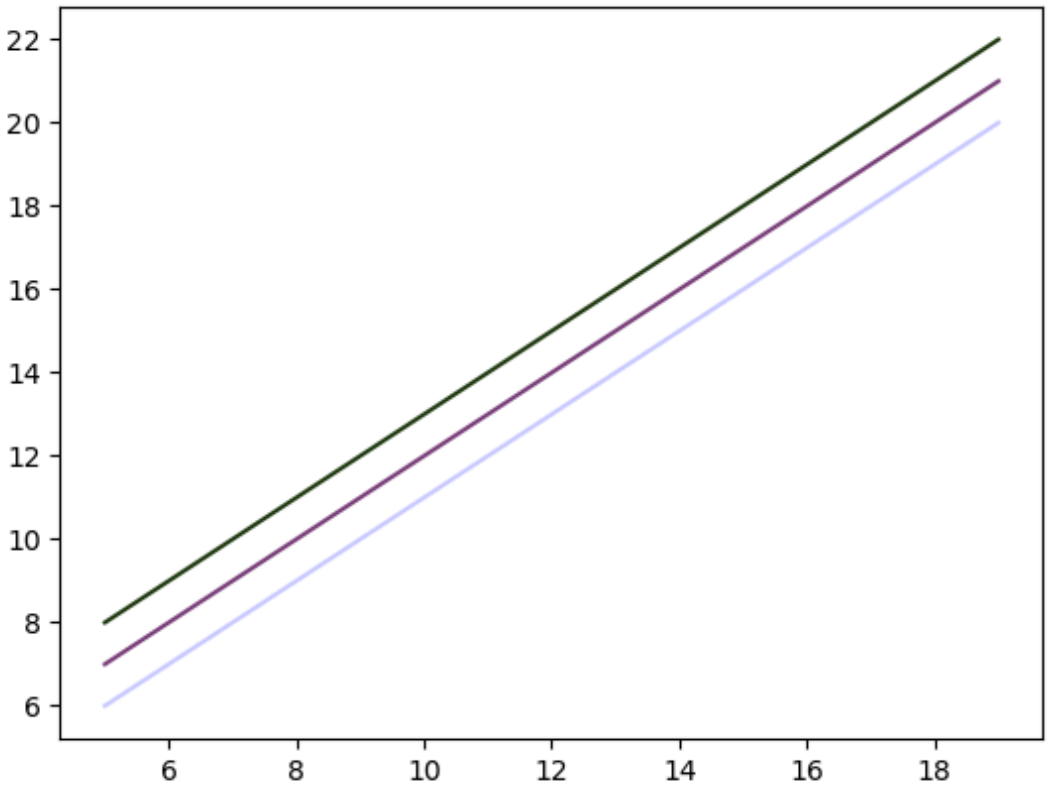
And we can pass `hex` color codes to the `color` argument too.

> Alpha is used to specify transparency of the color.

In [22]:
```
fig, ax = plt.subplots()
```

```
ax.plot(x, x+1, color="blue", alpha=0.2) # half-transparant
ax.plot(x, x+2, color="#7A407A")        # RGB hex code
ax.plot(x, x+3, color="#213B11")        # RGB hex code
```

Out[22]: [<matplotlib.lines.Line2D at 0x7d0f23cd3810>]



Line and marker styles

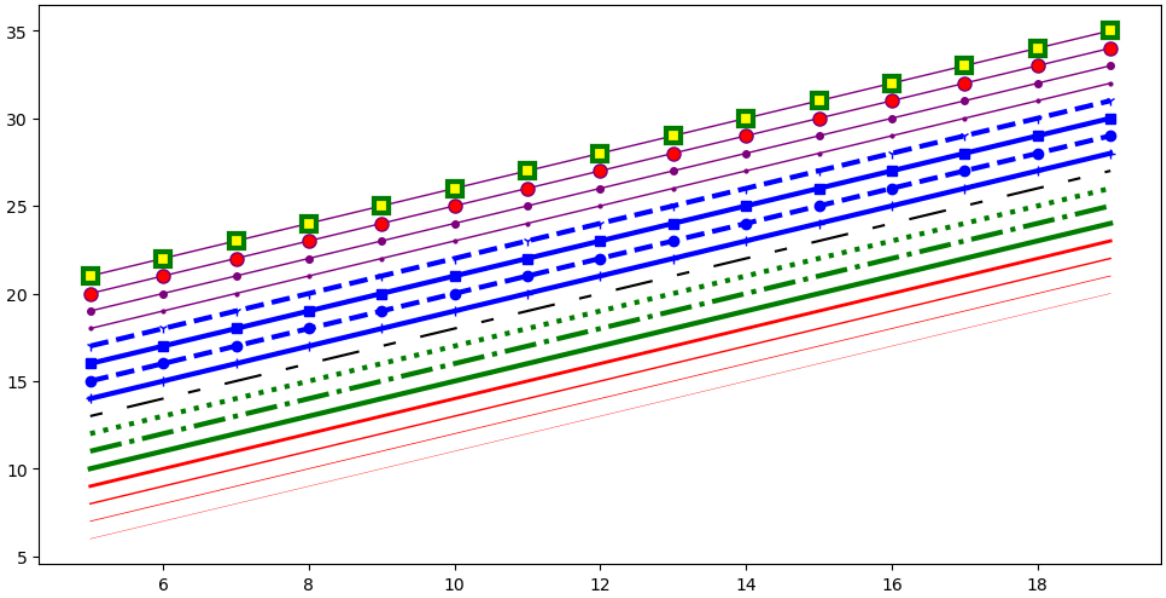
There are so many ways we can customize the line and marker styles of a plot.

I've found a script that demonstrates all these options.

> credit goes to udemi: python for data science and machine learning bootcamp

In [23]:
```
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x+1, color="red", linewidth=0.25)
ax.plot(x, x+2, color="red", linewidth=0.50)
ax.plot(x, x+3, color="red", linewidth=1.00)
ax.plot(x, x+4, color="red", linewidth=2.00)

# possible linestype options ‘-‘, ‘–’, ‘-.’, ‘:’, ‘steps’
ax.plot(x, x+5, color="green", lw=3, linestyle='-')
ax.plot(x, x+6, color="green", lw=3, ls='-.')
ax.plot(x, x+7, color="green", lw=3, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

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

```
ax.plot(x, x+10, color="blue", lw=3, ls='--', marker='o')
ax.plot(x, x+11, color="blue", lw=3, ls='-', marker='s')
ax.plot(x, x+12, color="blue", lw=3, ls='--', marker='1')

# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8, m
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
        markerfacecolor="yellow", markeredgewidth=3, markeredgecolor="gree
```



Control over axis appearance

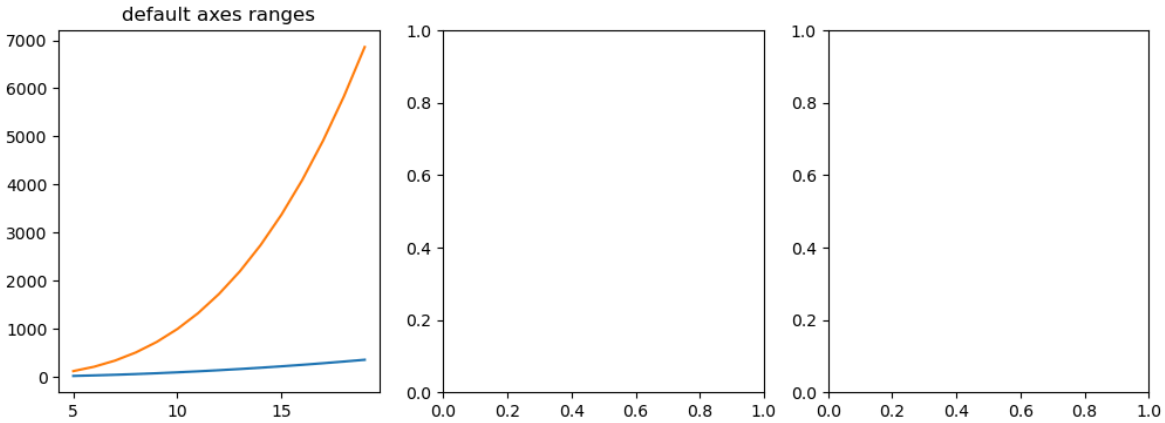
In this section we will look at controlling axis sizing properties in a matplotlib figure.

Plot range

The customizations doesn't stop there. You can set custom ranges for the axes.

In [ ]:
```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

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

Out[ ]:
```
Text(0.5, 1.0, 'default axes ranges')
```



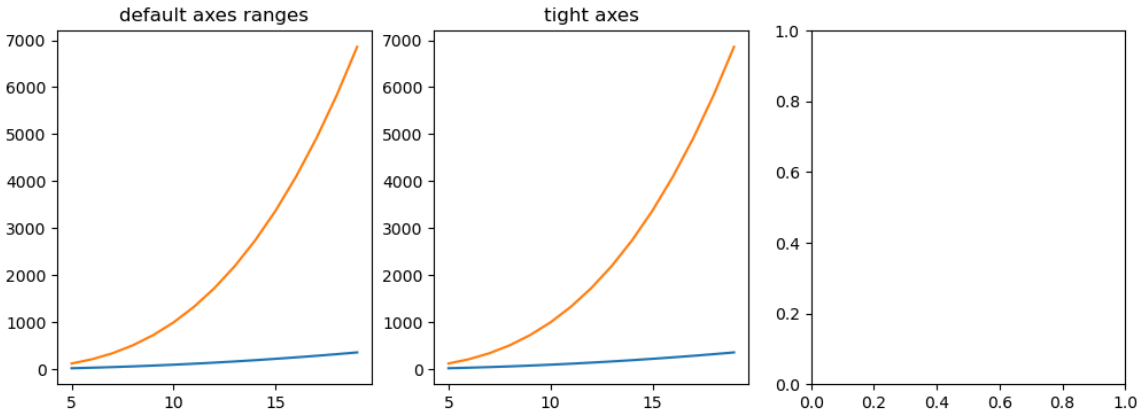
Default range is set to the maximum and minimum values of the data.

We set the axis to `tight` mode so that the axis range is set to the minimum and maximum values of the data.

This is the default behavior.

In [25]:
```
axes[1].plot(x, x**2, x, x**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")
fig
```

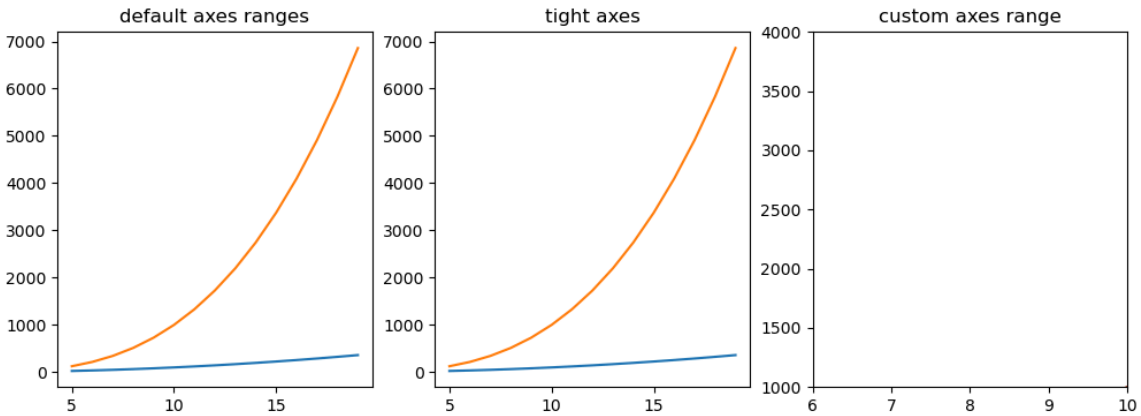
Out[25]:



We can use `set_xlim` and `set_ylim` to set the range of the axes.

In [ ]:
```
axes[2].plot(x, x**2, x, x**3)
axes[2].set_ylim([1000, 4000])
axes[2].set_xlim([6, 10])
axes[2].set_title("custom range");

fig
```

Out[ ]:



Built-in plots

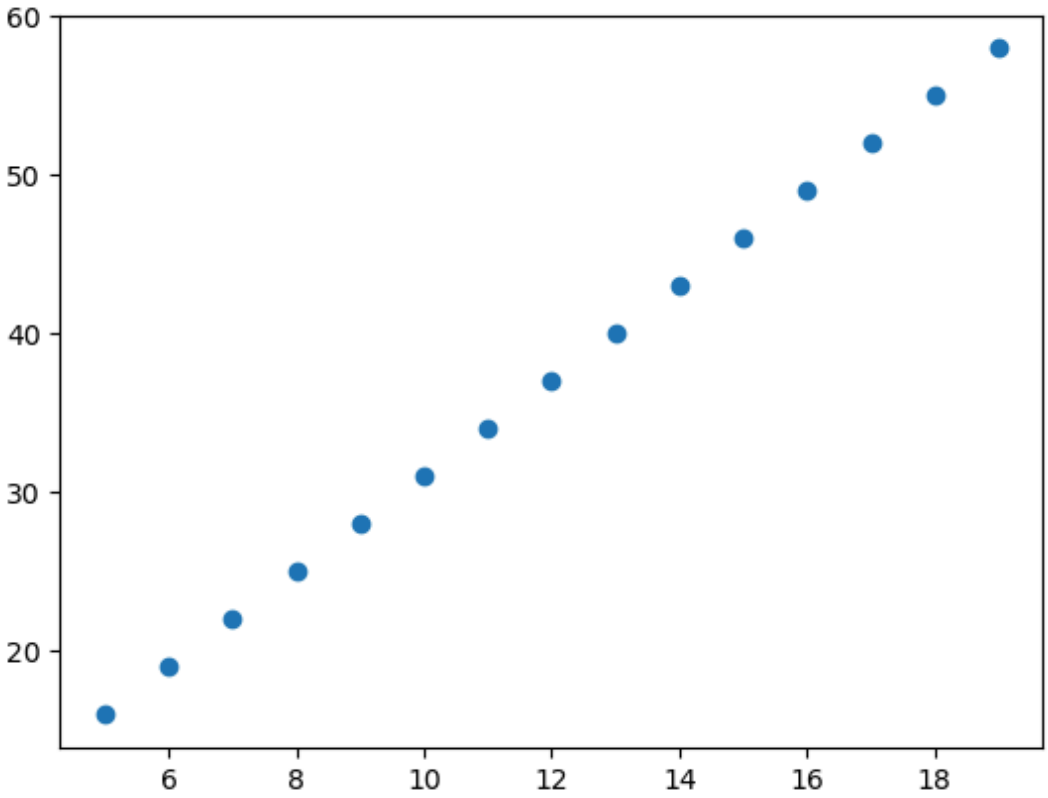
Now let's talk about some built in plots we can create using `matplotlib`.

One of them is scatter plots.

`Scatter plots` are used to show the `relationship` between `two variables`.

In [27]:
```
plt.scatter(x,y)
```

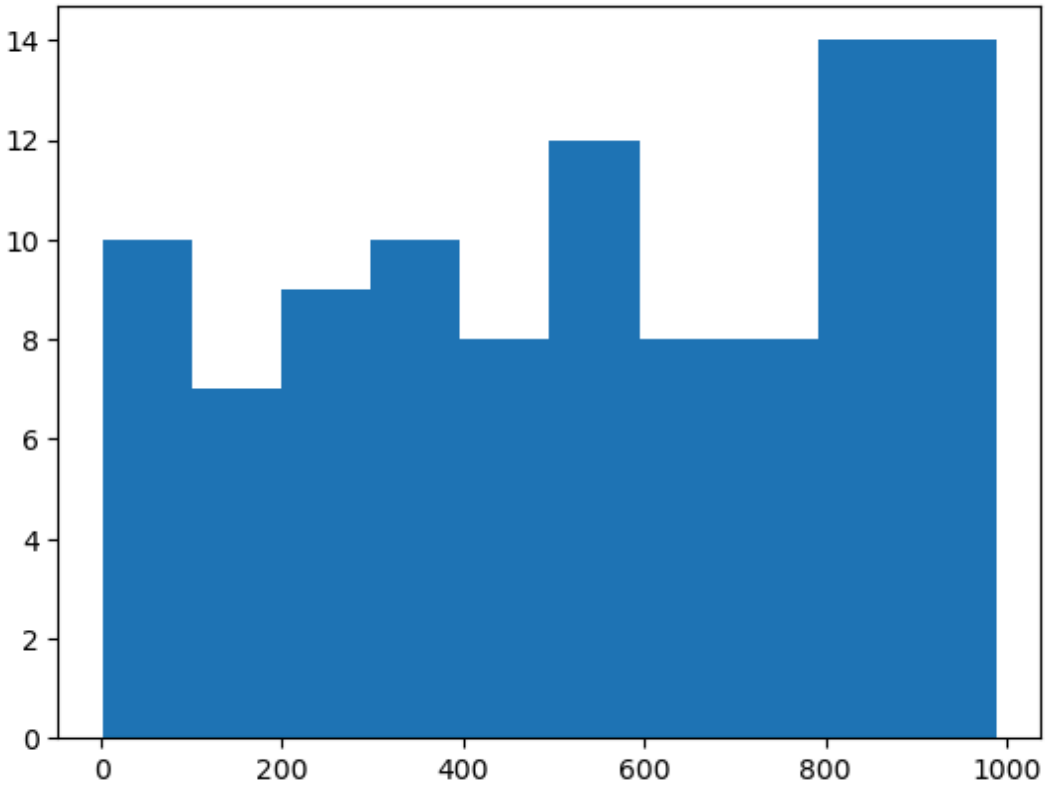
Out[27]:
```
<matplotlib.collections.PathCollection at 0x7d0f23ba7750>
```



We can also do histograms or bar plots.

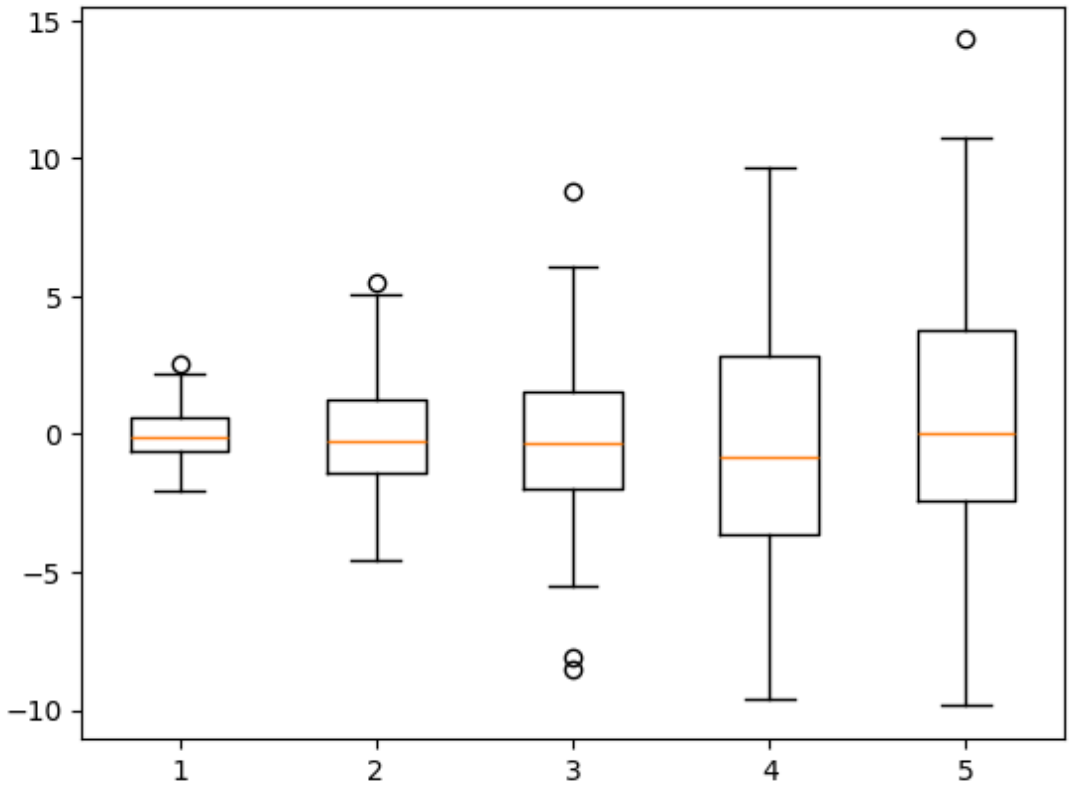
Histograms are used to show the `distribution` of a variable in a cirtain range of values.

In [ ]:
```
from random import sample,seed
seed(0)
data = sample(range(1, 1000), 100)
plt.hist(data)
```



Boxplots are like histograms but with more information of where the data is concentrated.

In [52]:
```
data = [np.random.normal(0, std, 100) for std in range(1, 6)]

# rectangular box plot
plt.boxplot(data);
```



> For any python users the `;` at the end of the cell might freak you out. Don't freak out, it's a jupyter thing. It just tells the `kernel` not to show the general outputs of the cell.

> Try removing the `;` and run the cell to see what happens.

When there are too many outputs or warnings you can use a `;` to hide the outputs.

And that's it.

One last thing before we end this article.

Saving figures

We can save a figure to a file with one simple command in the `matplotlib` library.

In [30]:
```
fig.savefig("filename.png")
```

In [31]:
```
fig.savefig("filename.png", dpi=200)
```

And that's it.

Last words

I know, I know, I didn't go into much details of when to use which type of plot or into much details of the plotting because `matplotlib` is a powerful tools but it's only use case is to visualize data and without real data you can't really see how to use it properly.

When go deep into data analysis, we will learn a lot more on how to use matplotlib properly.

And also another library that will make data visualization even easier would be `seabord` which is built with `matplotlib` but is more user friendly.

I hope you liked this article.

Happy Coding!!