Introduction to Matplotlib — Data Visualization in Python

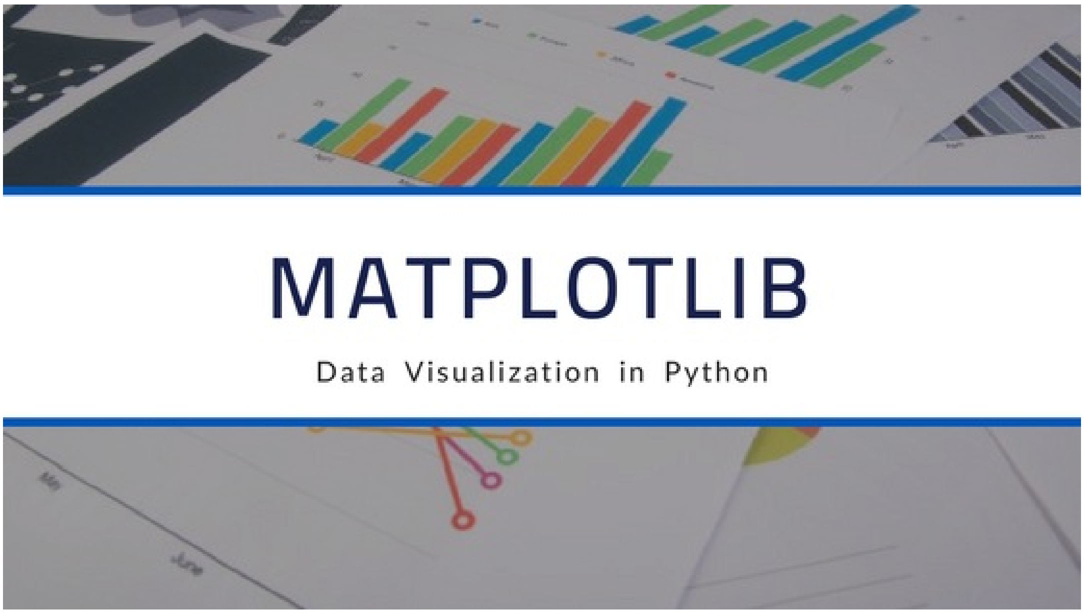
heartbeat.fritz.ai/introduction-to-matplotlib-data-visualization-in-python-d9143287ae39

Ehi Aigiomawu

Matplotlib is the most popular data visualization library in Python. It allows us to create figures and plots, and makes it very easy to produce static raster or vector files without the need for any GUIs.

This tutorial is intended to help you get up-and-running with Matplotlib quickly. We'll go over how to create the most commonly used plots, and discuss when to use each one.

Installing Matplotlib



If you have Anaconda, you can simply install Matplotlib from your terminal or command prompt using:

```
conda install matplotlib
```

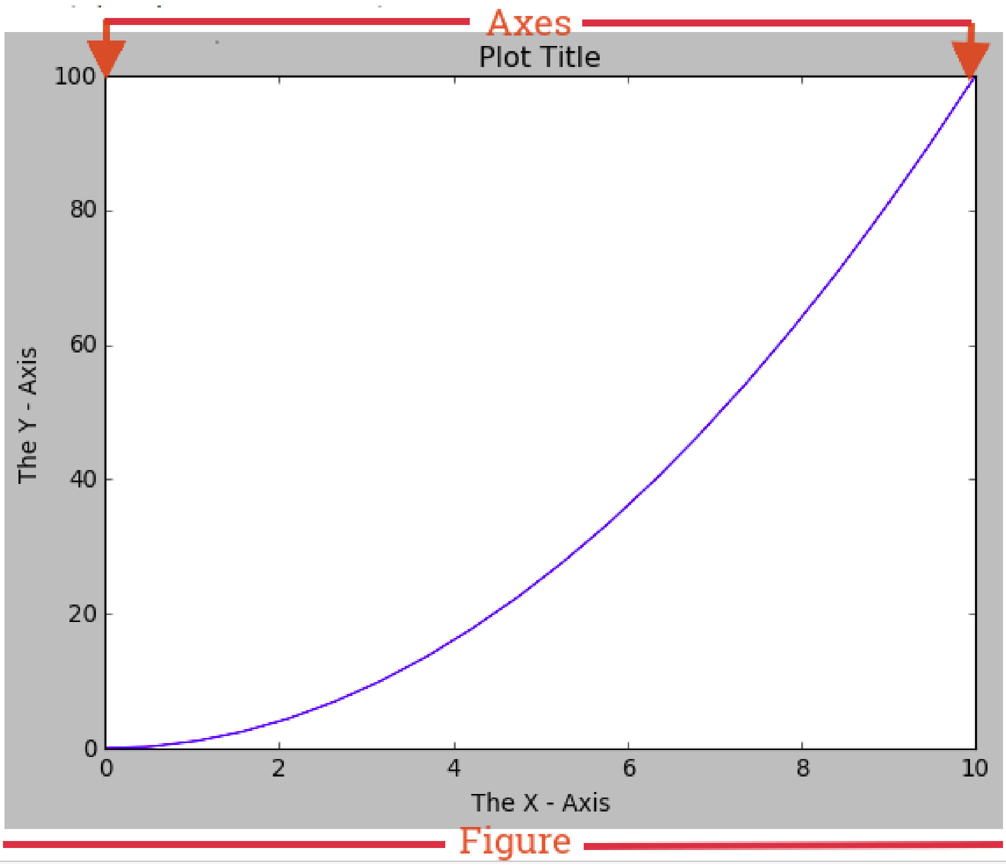
If you do not have Anaconda on your computer, install Matplotlib from your terminal using:

```
pip install matplotlib
```

Now that you have Matplotlib installed, let's begin by understanding the anatomy of a plot.

Anatomy of a Plot

There are two key components in a Plot; namely, Figure and Axes.



The `Figure` is the top-level container that acts as the window or page on which everything is drawn. It can contain multiple independent figures, multiple Axes, a subtitle (which is a centered title for the figure), a legend, a color bar, etc.

The `Axes` is the area on which we plot our data and any labels/ticks associated with it. Each `Axes` has an X-Axis and a Y-Axis (like in the image above). Let's go ahead to making plots.

Getting Started

We will begin by importing Matplotlib using:

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

Now that we have Matplotlib imported in our workspace, we need to be able to display the plots as it's being created. If you're using the Jupyter notebook we can easily display plots using: `%matplotlib inline`. However, if you're using Matplotlib from within a Python script, you have to add `plt.show()` method inside the file to be able display your plot.

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

In [2]: %matplotlib inline

        #OR
        plt.show() #to display your plots when not using the Jupyter notebook
```

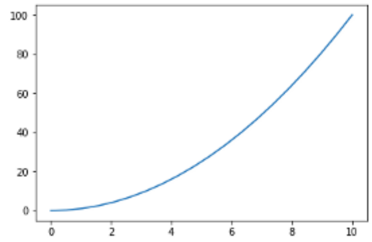
We are now ready to begin creating our plots. We can do this using two different approaches.

Two Approaches for creating Plots

1. **Functional Approach:** Using the basic Matplotlib command, we can easily create a plot. Let's plot an example using two Numpy arrays `x` and `y` :

```
In [3]: import numpy as np
        x = np.linspace(0, 10, 20) #Generate 20 datapoints between 0 and 10
        y = x**2                   #Generate array 'y' from square of 'x'
```

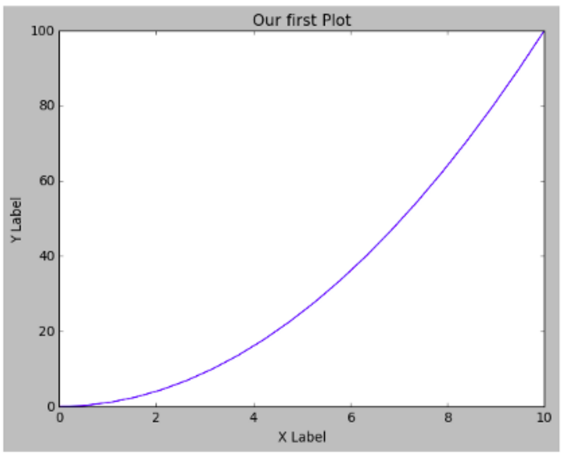
```
In [4]: plt.plot(x,y)
Out[4]: [<matplotlib.lines.Line2D at 0x7f5a339fcd30>]
```



If no plot was displayed or if you're using Matplotlib from within a Python script, don't forget to add `plt.show()` at the last line to display your plot.

Now that we have a plot, let's go on to name the x-axis, y-axis, and add a title using `.xlabel()` , `.ylabel()` and `.title()` using:

```
In [12]: plt.plot(x,y)
         plt.title('Our first Plot')
         plt.xlabel('X Label')
         plt.ylabel('Y Label')
Out[12]: Text(0,0.5,'Y Label')
```

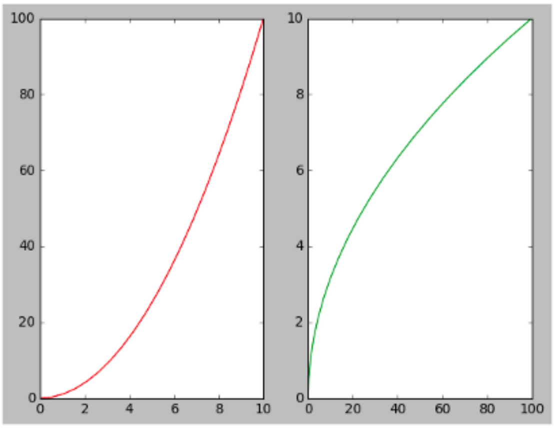


Imagine we needed more than one plot on that canvas. Matplotlib allows us easily create multi-plots on the same figure using the `.subplot()` method. This `.subplot()` method takes in three parameters, namely:

- `nrows` : the number of rows the `Figure` should have.
- `ncols` : the number of columns the `Figure` should have.
- `plot_number` : which refers to a specific plot in the `Figure` .

Using `.subplot()` we will create a two plots on the same canvas:

In [13]:
```python
# plt.subplot(nrows, ncols, plot_number)
plt.subplot(1, 2, 1)
plt.plot(x, y, 'red') # More on color options later

plt.subplot(1,2,2)
plt.plot(y, x, 'green');
```



Notice how the two plots have different colors. This is because we need to be able to differentiate the plots. This is possible by simply setting the color attribute to `‘red’` and `‘green’` as you can see above.

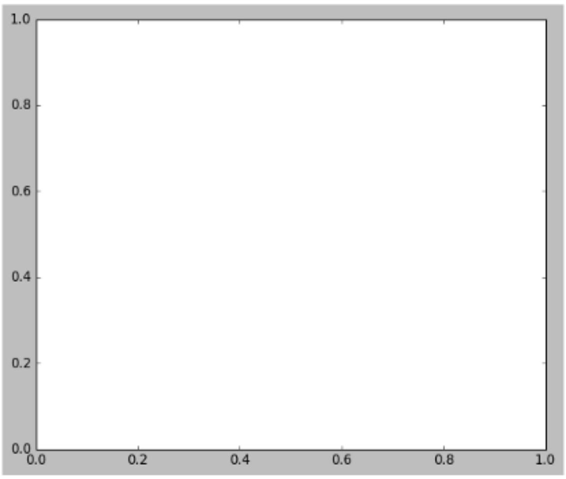
2. **Object oriented Interface:** This is the best way to create plots. The idea here is to create `Figure` objects and call methods off it. Let’s create a blank `Figure` using the `.figure()` method.

In [14]:
```python
fig = plt.figure()
```

```
<Figure size 640x480 with 0 Axes>
```

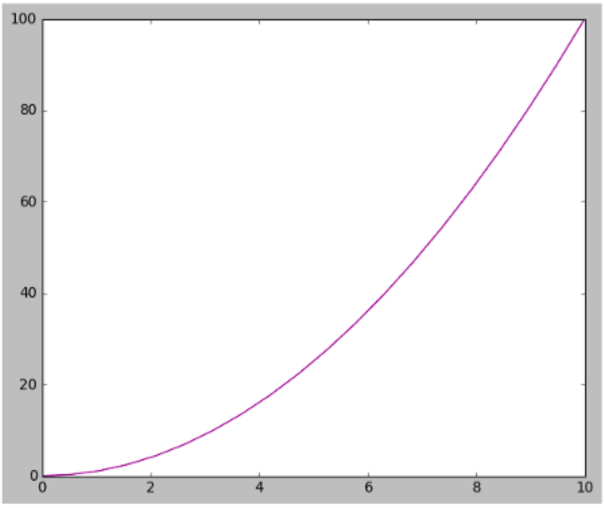
Now we need to add a set of axes to it using the `.add_axes()` method. The `add_axes()` method takes in a list of four arguments (left, bottom, width, and height—which are the positions where the axes should be placed) ranging from 0 to 1. Here’s an example:

In [17]:
```python
fig = plt.figure()

ax = fig.add_axes([0.1, 0.2, 0.8, 0.9])
```



As you can see, we have a blank set of axes. Now let’s plot our x and y arrays on it:

```
In [18]: fig = plt.figure()
         ax = fig.add_axes([0.1, 0.2, 0.8, 0.9])

         ax.plot(x,y, 'purple')
Out[18]: [<matplotlib.lines.Line2D at 0x7f00630b6208>]
```

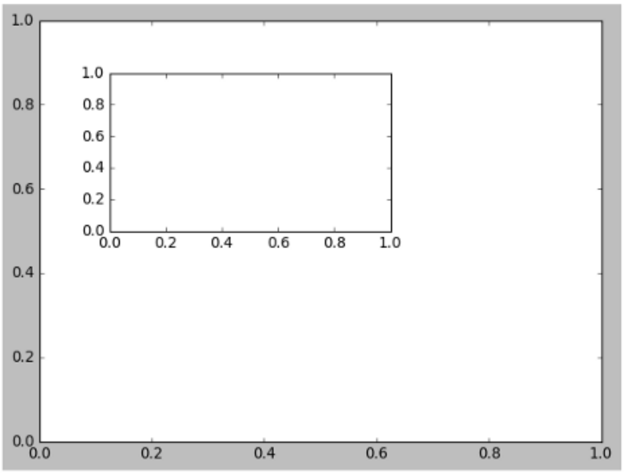


We can further add x and y labels and a title to our plot same way we did in the Function approach, but there's a slight difference here. Using `.set_xlabel()`, `.set_ylabel()` and `.set_title()` let us go ahead and add labels and a title to our plot:

```
In [22]: fig = plt.figure()
         ax = fig.add_axes([0.1, 0.2, 0.8, 0.9])

         ax.plot(x,y, 'purple')
         ax.set_xlabel('X Label')
         ax.set_ylabel('Y Label')
         ax.set_title('Our First Plot using Object Oriented Approach')
```

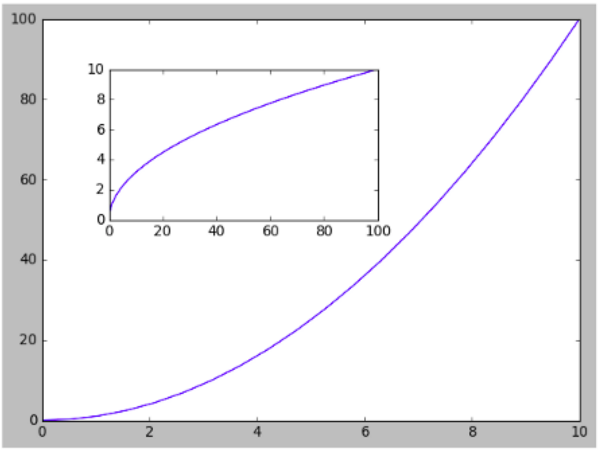
Remember, we noted that a `Figure` can contain multiple figures. Let's try to put in two sets of figures on one canvas:

```
In [24]: fig = plt.figure()

         axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])
         axes2 = fig.add_axes([0.2, 0.5, 0.4, 0.3])
```



Now let's plot our x and y arrays on the axes we have created:

```
In [26]: fig = plt.figure()

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

         axes1.plot(x,y)
         axes2.plot(y,x)
Out[26]: [<matplotlib.lines.Line2D at 0x7f00630991d0>]
```

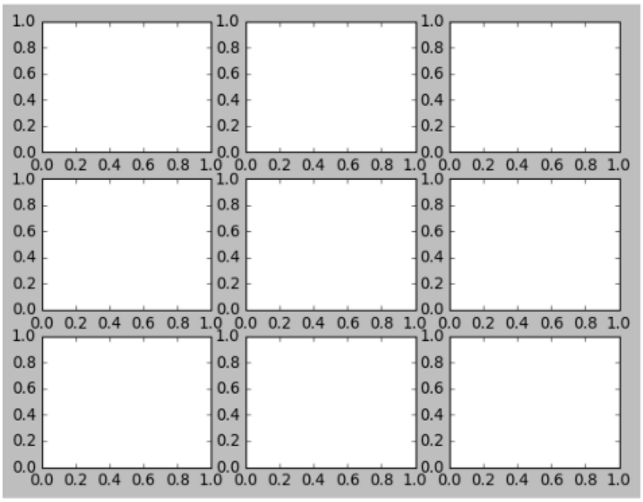


> Quick Exercise: Now that we have our plot ready, see if you can set the title, the x and y labels for both axes.

Like we did in the functional approach, we can also create multiple plots in the object-oriented approach using the `.subplots()` method, and NOT `.subplot()`. The `.subplots()` method takes in `nrows`, which is the number of rows the `Figure` should have, and `ncols`, the number of columns the `Figure` should have.

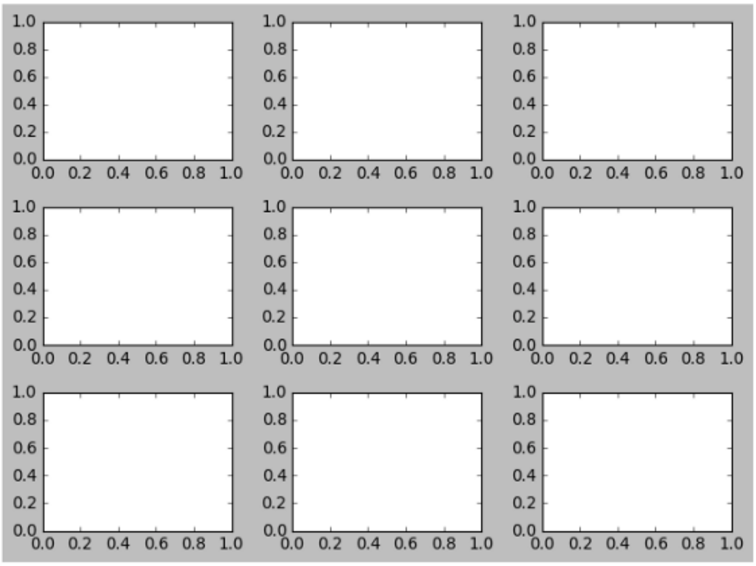
For example, we can create a 3 by 3 subplots like this:

```
In [30]: # Empty canvas of 3 by 3 subplots
         fig, axes = plt.subplots(nrows=3, ncols=3)
```



What we have just done is that we used tuple unpacking to grab the axes from the `Figure` object which gave us a 3 by 3 subplots. As we see, there is an issue of overlapping in the subplots we created. We can deal with that by using `.tight_layout()` method to space it out:

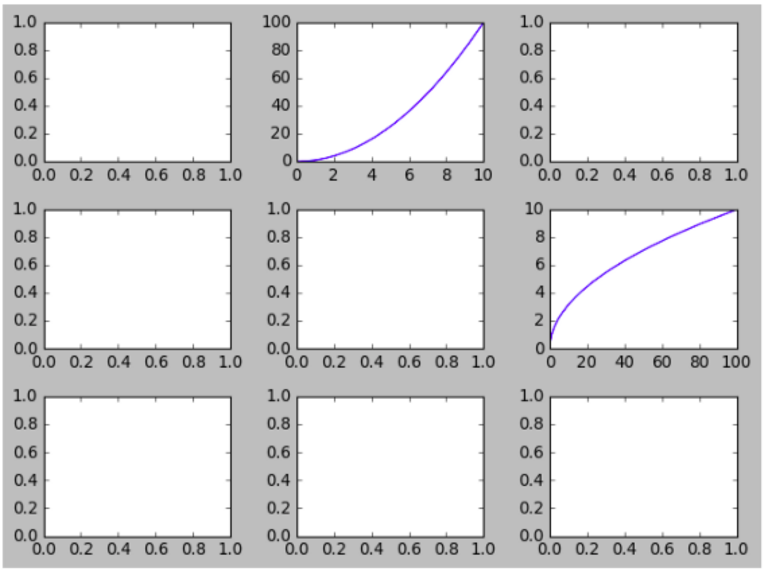
```
In [32]: # Empty canvas of 3 by 3 subplots
fig, ax = plt.subplots(nrows=3, ncols=3)

plt.tight_layout()
```



The only difference between `plt.figure()` and `plt.subplots()` is that `plt.subplots()` automatically does what the `.add_axes()` method of `.figure()` will do for you based off the number of rows and columns you specify.

Now that we know how to create subplots, let's see how we can plot our x and y arrays on them. We want to plot x, y on the axes at index position (0,1) and y, x on the axes at position (1,2) respectively:

```
In [34]: # Empty canvas of 3 by 3 subplots
fig, ax = plt.subplots(nrows=3, ncols=3)

ax[0,1].plot(x,y)
ax[1,2].plot(y,x)

plt.tight_layout()
```



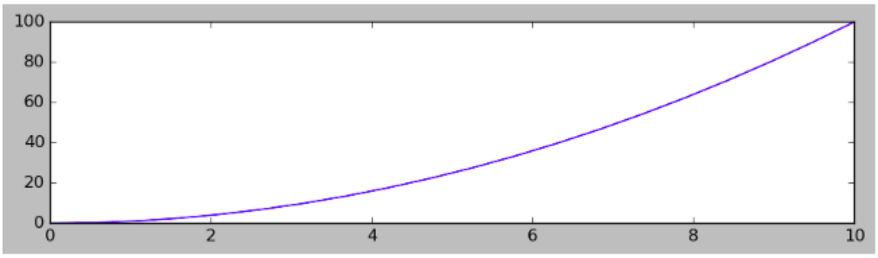
Quick Exercise: Go ahead and see if you can set the title and the x and y labels for both axes.

Figure size, aspect ratio, and DPI

Matplotlib allows us create customized plots by specifying the figure size, aspect ratio, and DPI by simply specifying the `figsize` and `dpi` arguments. The `figsize` is a tuple of the width and height of the figure (in inches), and `dpi` is the dots-per-inch (pixel-per-inch).

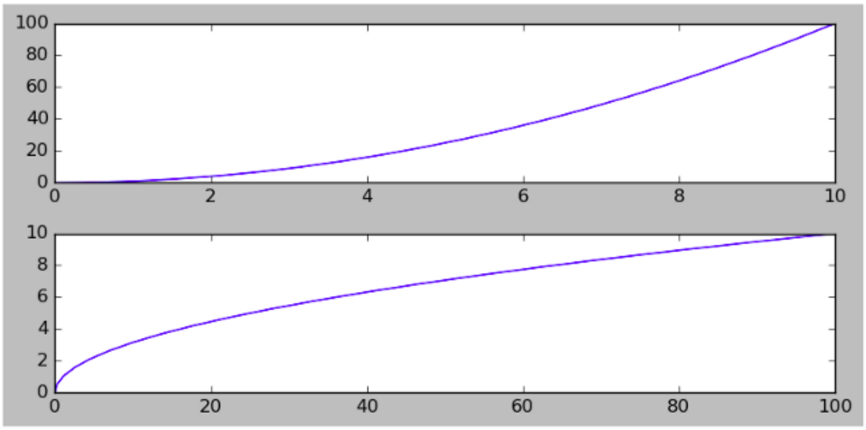
In the previous examples, we didn't specify the `figsize` and `dpi`, so Matplotlib assumed their default values. Now, let's go ahead and specify that we want a figure having `width=8`, `height=2`, and `dpi=100`.

```
In [39]: fig = plt.figure(figsize=(8,2), dpi = 100)

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

Out[39]: [<matplotlib.lines.Line2D at 0x7f00624b9a20>]
```



We can do the same thing with `subplots()` like this:

```
In [44]: fig, ax = plt.subplots(nrows=2, ncols=1,figsize=(8,4), dpi = 100)

         ax[0].plot(x,y)
         ax[1].plot(y,x)

         plt.tight_layout()
```



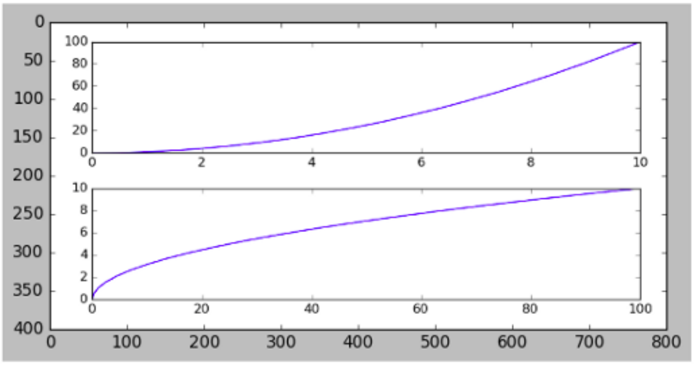
Now that we have learned how to create plots, let's learn how to save them for future use.

We can use Matplotlib to generate high quality figures and save them in a number of formats, such as png, jpg, svg, pdf, etc. Using the `.savefig()` method, we'll save the above figure in a file named `my_figure.png`:

```
fig.savefig('my_figure.png')
```

Go ahead and confirm the image by displaying it using:

```
In [52]: import matplotlib.image as mpimg
         plt.imshow(mpimg.imread('my_figure.png'))
Out[52]: <matplotlib.image.AxesImage at 0x7f00629afc88>
```



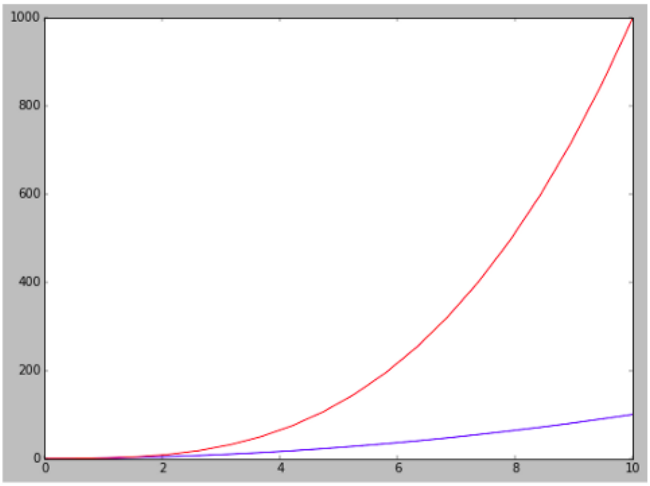
How to Decorate Figures

Now that we have covered the basics of how to create a figure and add axes, let's look at how to decorate a figure with legends and how we can customize our plot appearance.

Legends

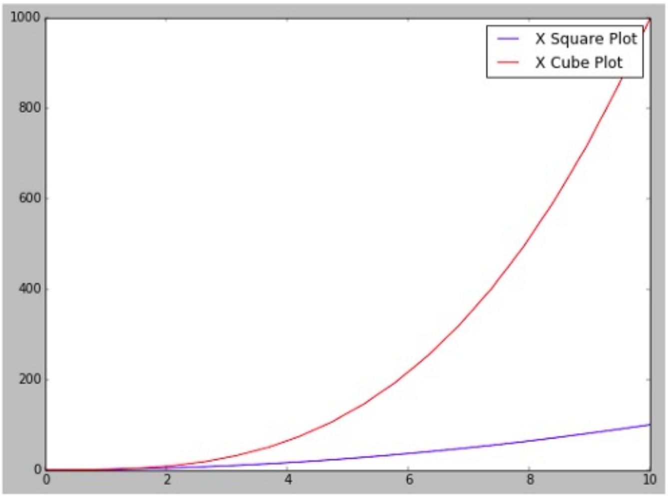
Legends allows us to distinguish between plots. With Legends, you can use label texts to identify or differentiate one plot from another. For example, say we have a figure having two plots like below:

```
In [74]: fig = plt.figure(figsize=(8,6), dpi = 60)
         ax = fig.add_axes([0,0,1,1])
         ax.plot(x,x**2)
         ax.plot(x,x**3, 'red')
Out[74]: [<matplotlib.lines.Line2D at 0x7f0062338f60>]
```



It could be really confusing to know what each plot represents. Hence, to identify the plots, we need to add a legend using `.legend()` and then specify the `label=” ”` attribute for each plot:

```
In [75]: fig = plt.figure(figsize=(8,6), dpi = 60)
         ax = fig.add_axes([0,0,1,1])
         ax.plot(x,x**2, label="X Square Plot")
         ax.plot(x,x**3, 'red', label='X Cube Plot')

         ax.legend()

Out[75]: <matplotlib.legend.Legend at 0x7f0061e0ee80>
```

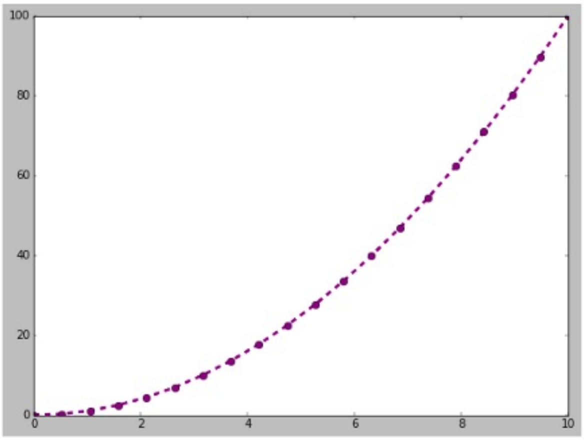


Plot Appearance

Matplotlib gives us a lot of options for customizing the appearance of our plots. By now, you should be familiar with changing line color using `color='red'` or `'red'` like we did in previous examples. Now we want to change `linewidth` or `lw`, `linestyle` or `ls`, and mark out data points using `marker`. You can find a whole list of what is possible here and here.

For the sake of this example, we want our plot to have a `linewidth` of `3`, our `linestyle` to be `double dashes`, we want to map out our datapoints using `'o'` as our `marker` having a the `markersize` of `8`:

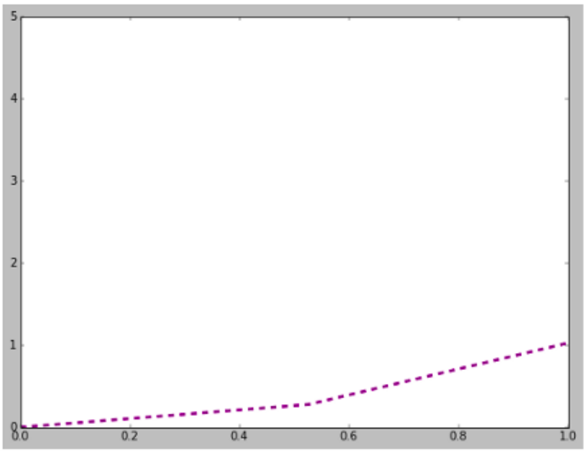
```
In [82]: fig = plt.figure(figsize=(8,6), dpi = 60)
         ax = fig.add_axes([0,0,1,1])

         #Change the line thickness to 3, line style to dashes, and mark out the datapoints
         ax.plot(x,y, color='purple', linewidth=3, linestyle='--', marker='o', markersize=8)

Out[82]: [<matplotlib.lines.Line2D at 0x7f0061bc2b00>]
```



Plot range

Matplotlib allows us to set limits for our plots. We can easily configure the range of our plots using the `set_ylim` and `set_xlim` methods of the axis object, or `axis('tight')` to automatically get "tightly fitted" axes ranges. For example, we can choose to show only plots between 0 to 1 of the `x axis`, and 0 to 5 of the `y axis`:

```
In [85]: fig = plt.figure(figsize=(8,6), dpi = 60)
         ax = fig.add_axes([0,0,1,1])

         ax.plot(x,y, color='purple', lw=3, ls='--')

         ax.set_xlim([0,1]) #[0,1] signifies the lower bound and upper bound of x axis
         ax.set_ylim([0,5]) #[0,5] signifies the lower bound and upper bound of y axis

Out[85]: (0, 5)
```



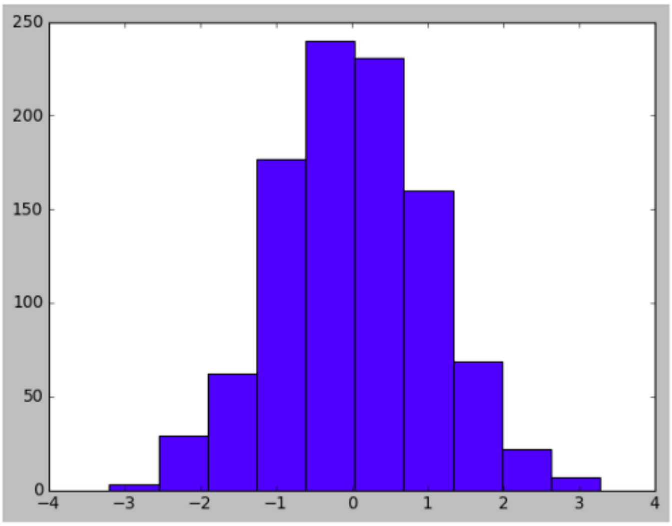
Now that we know how to create and customize basic line plots, it is important to mention that those are not the only kinds of plots possible in Matplotlib. Specialized plots such as barplots, histograms, scatter plots, etc can also be created in Matplotlib.

Special Plot Types

Matplotlib allows us create different kinds of plots ranging from histograms and scatter plots to bar graphs and bar charts. The key to knowing which plot to use depends on the purpose of the visualization. You may be trying to compare two quantitative variables to each other, or you might want to check for differences between groups, or you may be interested in knowing the distribution of a variable. Each of these goals is best served by different plots, and using the wrong one could distort the interpretation of the data. Let's see some of these plots and what they're best suited for.

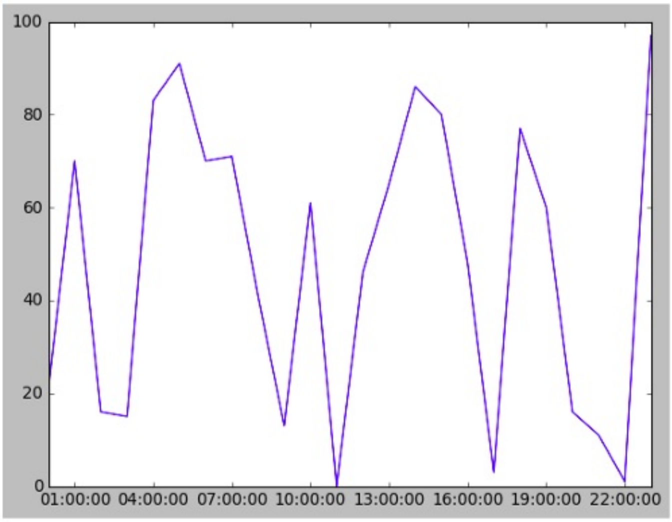
Histograms: help us understand the distribution of a numeric value in a way that you cannot with mean or median alone. Using `.hist()` method, we can create a simple histogram:

In [90]:
```
x = np.random.randn(1000)
plt.hist(x);
```



Time series (Line Plot) is a chart that shows a trend over a period of time. It allows you to test various hypotheses under certain conditions, like what happens different days of the week or between different times of the day.

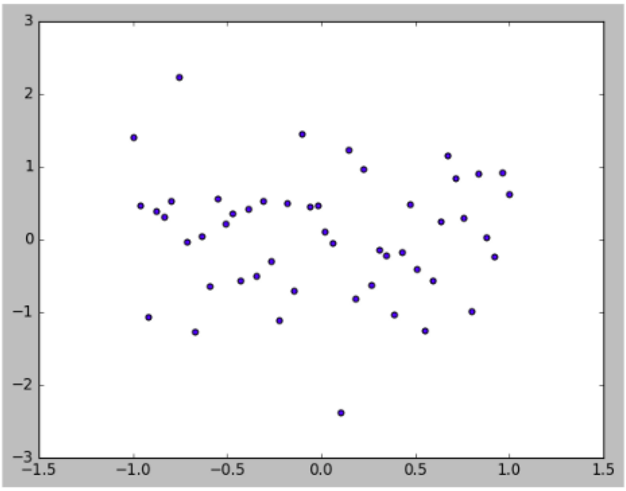
In [227]:
```
import matplotlib.pyplot as plt
import datetime
import numpy as np

x = np.array([datetime.datetime(2018, 9, 28, i, 0) for i in range(24)])
y = np.random.randint(100, size=x.shape)

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



Scatter plots offer a convenient way to visualize how two numeric values are related in your data. It helps in understanding relationships between multiple variables. Using `.scatter()` method, we can create a scatter plot:

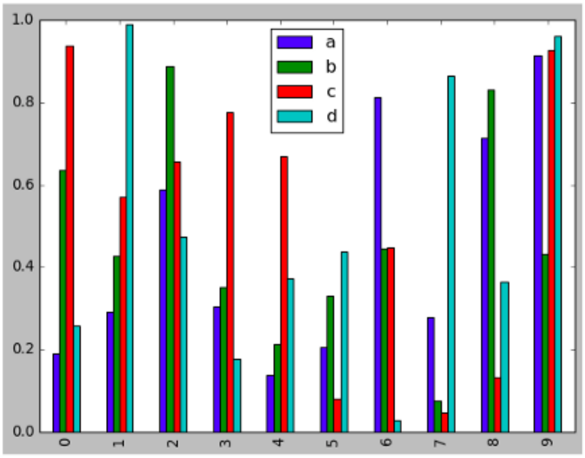
```
In [214]: fig, ax = plt.subplots()
          x = np.linspace(-1, 1, 50)
          y = np.random.randn(50)
          ax.scatter(x, y)
```

```
Out[214]: <matplotlib.collections.PathCollection at 0x7f004eaa30f0>
```



Bar graphs are convenient for comparing numeric values of several groups. Using `.bar()` method, we can create a bar graph:

```
In [216]: my_df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

          my_df.plot.bar();
```



Now that we have basic understanding of how to visualize data by creating plots, the different kinds of plot possible and situations they can be applied to, let's try our hands on a real world example.

Sample Application of Visualization

Imagine we were asked to find the richest country in the world on a per-person basis in the sample dataset *(download week 3).*

For simplicity, what we will do is compare different country's gdp per capita to try to answer this question following the steps below :

1. First, we will import all necessary packages.
2. Load our dataset.
3. Clean the dataset by filling in missing values.
4. Aggregate values using `.groupby()` .
5. Sort the values.

6. Represent our data in either line or bar plot.

```
import matplotlib.pyplot as plt
import pandas as pd

my_data = pd.read_csv('nations.csv')
```

Out[235]:

	iso2c	iso3c	country	year	gdp_percap	life_expect	population	birth_rate	neonat_mortal_rate	region	income
0	AD	AND	Andorra	1996	NaN	NaN	6.429100e+04	10.900	2.8	Europe & Central Asia	High income
1	AD	AND	Andorra	1994	NaN	NaN	6.270700e+04	10.900	3.2	Europe & Central Asia	High income
2	AD	AND	Andorra	2003	NaN	NaN	7.478300e+04	10.300	2.0	Europe & Central Asia	High income
3	AD	AND	Andorra	1990	NaN	NaN	5.451100e+04	11.900	4.3	Europe & Central Asia	High income
4	AD	AND	Andorra	2009	NaN	NaN	8.547400e+04	9.900	1.7	Europe & Central Asia	High income
5	AD	AND	Andorra	2011	NaN	NaN	8.232600e+04	NaN	1.6	Europe & Central Asia	High income
6	AD	AND	Andorra	2004	NaN	NaN	7.833700e+04	10.900	2.0	Europe & Central Asia	High income

Notice that the dataset contained missing values in the `'gdp_percap'` column. Let's replace those values with the `median` value of that column:

```
my_data['gdp_percap'].fillna(my_data['gdp_percap'].median(),
inplace=True)

my_data.head(5)
```

Out[237]:

	iso2c	iso3c	country	year	gdp_percap	life_expect	population	birth_rate	neonat_mortal_rate	region	income
0	AD	AND	Andorra	1996	6765.480489	NaN	64291.0	10.9	2.8	Europe & Central Asia	High income
1	AD	AND	Andorra	1994	6765.480489	NaN	62707.0	10.9	3.2	Europe & Central Asia	High income
2	AD	AND	Andorra	2003	6765.480489	NaN	74783.0	10.3	2.0	Europe & Central Asia	High income
3	AD	AND	Andorra	1990	6765.480489	NaN	54511.0	11.9	4.3	Europe & Central Asia	High income
4	AD	AND	Andorra	2009	6765.480489	NaN	85474.0	9.9	1.7	Europe & Central Asia	High income

Now let's find the mean `gdp_percap` for each country. We are going to group `my_data` by the `'country'` column then find the mean values of the other columns for each `'country'` for all the available years

```
my_data.groupby(['country']).mean()
```

Out[238]:

country	year	gdp_percap	life_expect	population	birth_rate	neonat_mortal_rate
Afghanistan	2001.000000	4141.239064	55.293460	2.102448e+07	44.649875	44.612500
Albania	2003.000000	6501.595694	75.102358	3.052655e+06	16.708444	9.766667
Algeria	2002.666667	9741.203655	71.220236	3.232797e+07	22.947500	19.600000
American Samoa	2003.333333	6765.480489	NaN	5.543989e+04	18.700000	NaN
Andorra	2001.000000	6765.480489	NaN	7.177557e+04	10.800000	2.514286
Angola	2002.857143	4834.637488	46.847136	1.751027e+07	49.498571	55.371429
Antigua and Barbuda	2002.750000	17267.710125	73.829588	7.809900e+04	18.146000	9.000000
Argentina	2004.000000	6765.480489	74.454676	3.873062e+07	18.950429	9.685714
Armenia	2004.000000	4709.846531	72.065331	3.085080e+06	14.451571	13.685714
Aruba	1998.444444	10015.434930	73.793748	8.538878e+04	15.246444	NaN
Australia	2001.700000	29016.526252	79.721220	1.973095e+07	13.410000	3.390000
Austria	2002.777778	33223.532353	78.750136	8.156150e+06	10.055556	3.011111

Now we can narrow down to find the average `gdp_percap` of all the available years for each country and save it in a new variable called `'avg_gdp_percap'`

```
avg_gdp_percap = my_data.groupby(['country']).mean()['gdp_percap']

avg_gdp_percap
```

```
Out[239]: country
Afghanistan              4141.239064
Albania                  6501.595694
Algeria                  9741.203655
American Samoa           6765.480489
Andorra                  6765.480489
Angola                   4834.637488
Antigua and Barbuda     17267.710125
Argentina                6765.480489
Armenia                  4709.846531
Aruba                   10015.434930
Australia               29016.526252
Austria                 33223.532353
Azerbaijan              10795.672694
Bahamas, The            18983.550120
Bahrain                 33523.697160
Bangladesh               1684.434114
Barbados                12075.093853
Belarus                  7183.708559
Belgium                 34207.233355
Belize                   7059.133411
Benin                    1443.655655
```

Now let's sort the countries according to their `gdp_percap` and display 5 countries with the highest `gdp_percap`. We will save this data in a new variable called `'top_five_countries'`

```
top_five_countries = avg_gdp_percap.sort_values(ascending=False).head()

top_five_countries
```

```
Out[240]: country
Macao SAR, China        74714.598108
United Arab Emirates    68811.427014
Luxembourg              67990.823173
Qatar                   59926.550092
Brunei Darussalam       59355.226782
Name: gdp_percap, dtype: float64
```

Notice that `'Macao SAR, China'` has the highest average `gdp_percap`. Having this information, let's look at `'Macao SAR, China'` in more details to find out if it's actually the most richest country in the world on a per-person basis.

```
china = my_data[my_data['country'] == 'Macao SAR, China']

china
```

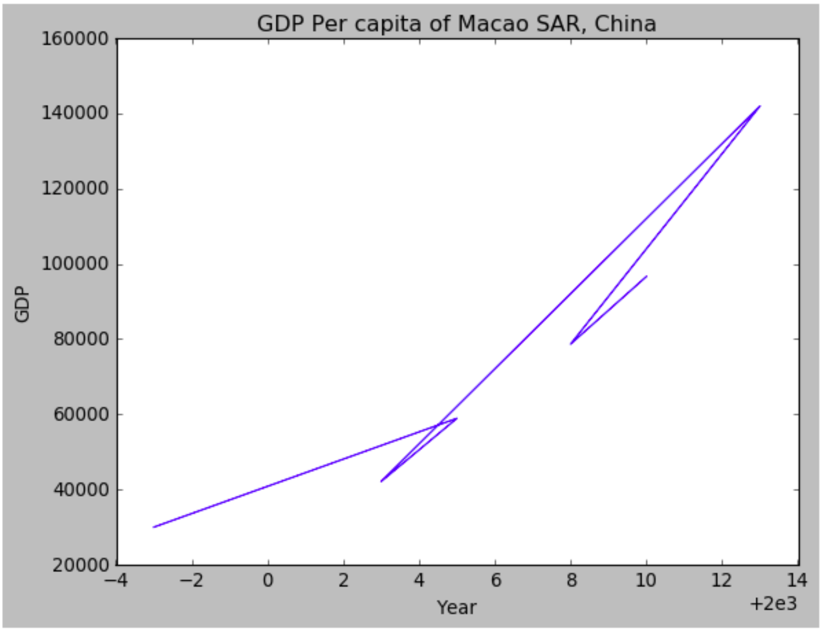
Out[241]:

	iso2c	iso3c	country	year	gdp_percap	life_expect	population	birth_rate	neonat_mortal_rate	region	income
931	MO	MAC	Macao SAR, China	1997	29984.138229	76.962756	412031.0	11.470	NaN	East Asia & Pacific	High income
932	MO	MAC	Macao SAR, China	2005	58922.097492	78.649024	468149.0	7.806	NaN	East Asia & Pacific	High income
933	MO	MAC	Macao SAR, China	2003	42126.842348	78.246293	450754.0	7.757	NaN	East Asia & Pacific	High income
934	MO	MAC	Macao SAR, China	2013	141968.100275	80.339146	568056.0	11.256	NaN	East Asia & Pacific	High income
935	MO	MAC	Macao SAR, China	2008	78666.565905	79.264610	507274.0	8.999	NaN	East Asia & Pacific	High income
936	MO	MAC	Macao SAR, China	2010	96619.844399	79.690390	534626.0	10.032	NaN	East Asia & Pacific	High income

Now let's plot how the `gdp_percap` in `'Macao SAR, China'` has changed over time:

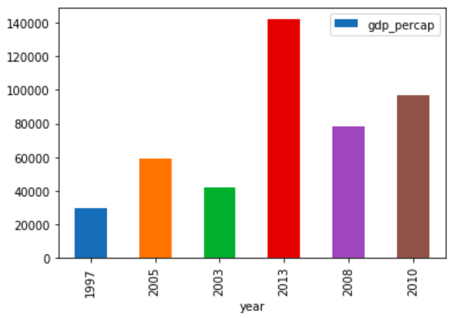
```
plt.plot(china['year'], china['gdp_percap'])

plt.xlabel('Year')

plt.ylabel('GDP')

plt.title('GDP Per capita of Macao SAR, China')
```

Out[242]: Text(0.5,1,'GDP Per capita of Macao SAR, China')



As you can see, the line plot doesn't give us a good representation that we can make meaning from, so let's try and visualize it in a barplot to get a better understanding of the data:

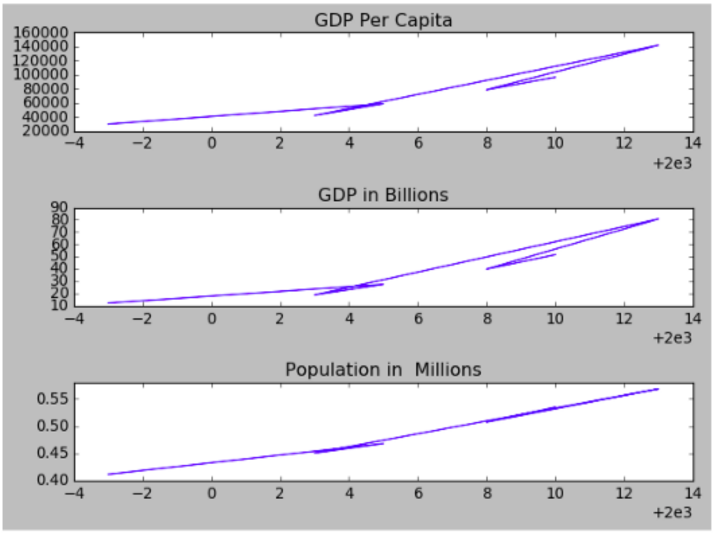
```
In [21]: china.plot.bar(x='year', y='gdp_percap')
Out[21]: <matplotlib.axes._subplots.AxesSubplot at 0x7f6c0d359fd0>
```



From the plot above, we see that china's `gdp_percap` was very high in 2013. Since gdp per capita is gdp per person, we will plot China's `gdp_percap` , `gdp` and `population` on the same graph using the `.subplot()` function.

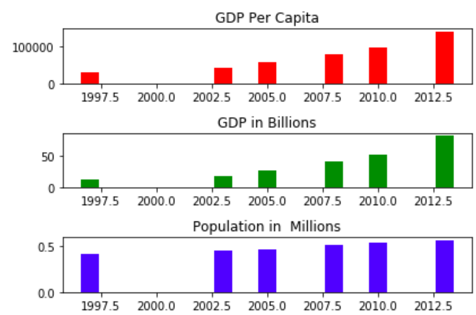
```
plt.subplot(311)

plt.title('GDP Per Capita')

plt.plot(china['year'], china['gdp_percap'])

plt.subplot(312)

plt.title('GDP in Billions')

plt.plot(china['year'], (china['population']*china['gdp_percap']/10**9))

plt.subplot(313)

plt.title('Population in Millions')
```

```
plt.plot(china['year'], china['population']/10**6)

plt.tight_layout()
```



Similarly for a clearer result, let's also plot it's bar graph:

```
In [47]: #Using Bar plot
plt.subplot(3,1,1)
plt.title('GDP Per Capita')
plt.bar(china['year'], china['gdp_percap'], color = 'r')


plt.subplot(3,1,2)
plt.title('GDP in Billions')
plt.bar(china['year'], (china['population']*china['gdp_percap']/10**9), color = 'g')


plt.subplot(3,1,3)
plt.title('Population in  Millions')
plt.bar(china['year'], china['population']/10**6, color = 'b')
plt.tight_layout()
```



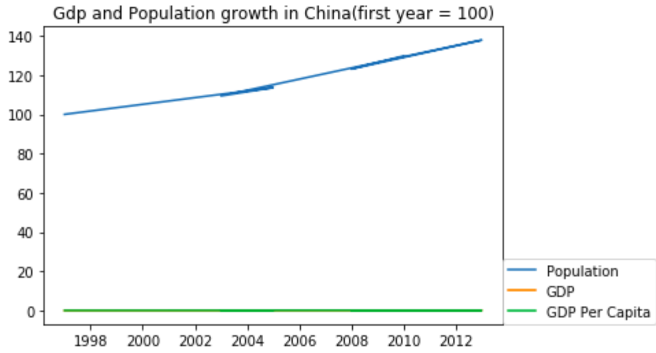
From the above plots, we see that China's gdp dropped significantly in the year 2000. In 2007, it picked up significantly but their population didn't rise.

However, how do we tell how much faster their population grew relative to their gdp? Let's try and compare their relative growth in a single plot by showing the population growth in the first year. We will set the first year's population to 100 as the basis of comparison, then repeat the same for `gdp` and `gdp_percap`

```
plt.plot(china['year'],
(china['population']/china['population'].iloc[0]*100))

china_gdp = china['population'] * china['gdp_percap']

plt.plot(china['year'], china_gdp/china_gdp.iloc[0]/100)
```

```
plt.plot(china[‘year’],
china[‘gdp_percap’]/china[‘gdp_percap’].iloc[0]/100)

plt.title(‘Gdp and Population growth in China(first year = 100)’)

plt.legend([‘Population’, ‘GDP’, ‘GDP Per Capita’], loc=4)
```

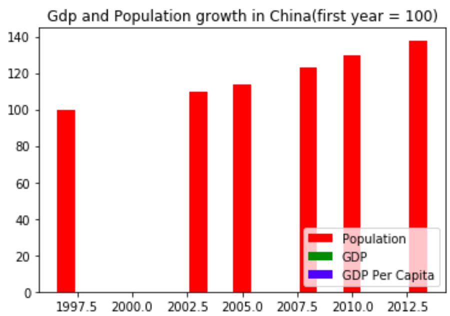
```
Out[58]: <matplotlib.legend.Legend at 0x7f6c01bc08d0>
```



Similarly, we could represent it in a bar plot for clearer view.

```
In [56]: plt.bar(china['year'], (china['population']/china['population'].iloc[0]*100), color='r')

         china_gdp = china['population'] * china['gdp_percap']
         plt.bar(china['year'], china_gdp/china_gdp.iloc[0]/100, color='g')

         plt.bar(china['year'], china['gdp_percap']/china['gdp_percap'].iloc[0]/100, color='b')

         plt.title('Gdp and Population growth in China(first year = 100)')
         plt.legend(['Population', 'GDP', 'GDP Per Capita'], loc=4)
```

```
Out[56]: <matplotlib.legend.Legend at 0x7f6c0221a470>
```

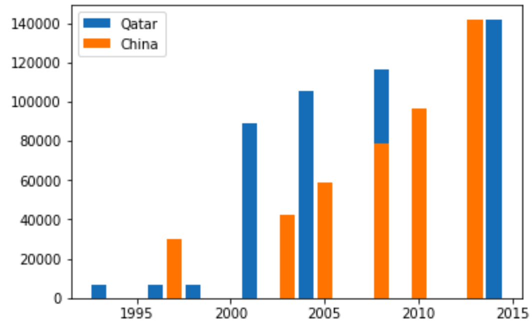


As we can see, at no point did China’s `gdp` ever catch up with the `population` growth.

To really answer this question, let’s go ahead and compare China’s `gdp_percap` with that of another country in the `top_five_countries` . Here, we will plot the `gdp` per capita growth in Qatar and China on the same chart.

In [63]:
```
#comparing Qatar with China
qt = my_data[my_data['country'] == 'Qatar']

plt.bar(qt['year'], qt['gdp_percap'])
plt.bar(china['year'], china['gdp_percap'])
plt.legend(['Qatar', 'China'])
```

Out[63]: <matplotlib.legend.Legend at 0x7f6c0208bf28>



We can see that in the year 2000, the `gdp_percap` in Qatar was much higher than in China, but became equal in 2015 . Hence, it's not really clear as to whether or not China has the highest gdp per capita on a per person basis.

Conclusion

If you made it this far, I am sure you now understand the basics of making visualizations using Matplotlib and how you can approach basic visualization problems. For more learning resources, realpython and the Matplotlib documentation are a great places to look.

Got questions, got stuck, or just want to say hi? Kindly use the comment box. If this tutorial was helpful to you in some way, show me some 👏.

Discuss this post on Hacker News.

Ready to dive into some code? Check out Fritz on GitHub. You'll find open source, mobile-friendly implementations of popular machine and deep learning models along with training scripts, project templates, and tools for building your own ML-powered iOS and Android apps.

And follow us on and for the all the latest content, news, and more from the mobile machine learning world.