Week 2 Notes

You can show matplotlib figures directly in the notebook by using the `%matplotlib notebook` and `%matplotlib inline` magic commands.

It is heavily inspired by MATLAB programming environment. In this module, we'll be focusing on some of the basics:

- scatter plots,
- line graphs
- bar charts

We will also touch on how to label pieces of your charts to improve the user's attention to specific details, and how to manipulate data elements in a plotting environment. From here on out there will be fewer lectures and more emphasis on the module will be put on the plotting and charting assignments.

At the end of this course, you should be able to produce publication quality charts and figures using a breadth of data. `%matplotlib notebook` provides an interactive environment.

In [1]:

```
%matplotlib notebook
```

Remember that in the Jupyter Notebook, the iPython magics are just helper functions which set up the environment so that **web based rendering** can be enabled.

When you use the magic function `%matplotlib inline`, its that matplotlib is configured to render into the browser.

This configuration is called a **backend** and matplotlib has a multiple of different backends available.

02-01 Matplotlib Architecture

The architecture to accomplish this is logically separated into three layers, which can be viewed as a stack. Each layer that sits above another layer knows how to talk to the layer below it, but the lower layer is not aware of the layers above it. The three layers from **bottom to top** are: backend, artist, and scripting.

Bottom: Backend Layer

At the bottom of the stack is the backend. The backend is an **abstraction layer** which knows how to interact with the operating environment, which can be an **operating system**, or an environment like the **browser**, and knows how to **render matplotlib commands.**

It provides **concrete implementations** of the abstract interface classes:

- `FigureCanvas` which encapsulates the concept of a surface to draw onto (eg. the paper)
- `Renderer` which does the drawing (e.g. 'the paintbrush')
- `Event` handles user inputs such as keyboard and mouse **events.**

Job of the `FigureCanvas`

For a user interface toolkit such as Qt, the `FigureCanvas` has a concrete implementation which knows...

1. How to insert itself into a native Qt window `QMainWindow`
2. Transfer the matplotlib Renderer commands onto the canvas `QPainter`
3. Translate **native Qt events** into the matplotlib `Event` framework, which *signals the callback dispatcher* to generate the events **so upstream listeners can handle them**. The abstract base classes reside in `matplotlib.backend_bases` and all of the derived classes live in modules like `matplotlib.backends.backend_qt4agg`

Job of the `Renderer`

The job of the Renderer is to provide a **low level drawing interface** for putting **ink onto the canvas**. The original `Renderer` API was motivated by the GDK `Drawable` interface, which implements **primitive methods** such as `draw_point`, `draw_line`, `draw_rectangle`, `draw_image`, `draw_polygon`, `draw_glyphs`.

Job of the `Event`

The matplotlib `Event` framework **maps underlying UI events** like `key-press-event` or `mouse-motion-event` to the matplotlib classes like `KeyEvent` or `MouseEvent` . Users **can connect to these events** to **callback functions** and **interact with their figure and data**. For example, to `pick` a data point or group of points, or manipiulate some aspect of the figure or its constituents. The following code samle illustrates how to **toggle all of the lines** in an `Axes` window when the user types `'t'`.

- Deals with the rendering of plots to screen or files
- In Jupyter notebooks we use the inline backend.
- There are also backends called **hard copy backends**, which support rendering to graphics formats, like scalable vector graphics, SVGs or PNGs.

Not all backends supports all features, especially **interactive features**, are often left out by many backends.

You'll notice that matplotlib has a lot of nonPython naming conventions.In particular, you'll be getting and accessing variables using `get()` and `set()` .

In [2]:

```
import matplotlib as mpl
mpl.get_backend()
```

Out[2]:

```
'nbAgg'
```

Middle: Artist Layer

The `Artist` hierarchy is the **middle layer** of the matplotlib stack, and is the place where much of the heavy lifting happens. Continuing with the analogy that the `FigureCanvas` from the backend is the paper, the `Artist` is the object that knows *how* to take the Renderer (the paintbrush) and put ink on the canvas. **Everything you see in a matplotlib Figure is an Artist instance**; *the title, the lines, the tick labels, the images*, and so on all correspond to individual Artist instances (see Figure 11.3). The base class is `matplotlib.artist.Artist`, which **contains attributes that every Artist shares**:

- the transformation which translates the artist coordinate system to the canvas coordinate system (discussed in more detail below),
- the visibility,
- the clip box which defines the region the artist can paint into,
- the label,
- the interface to handle user interaction such as "picking": that is, detecting when a mouse click happens over the artist.

The artist layer is an **abstraction around drawing and layout primitives.** The **root** of visuals is a set of containers which includes a `Figure` object **with one or more** `Subplots` **each with a series of one or more** `axes`.

Primitives & Collections

The artist layer also contains **primitives** and **collections**. **Primitives** are **base drawing items**, things like a *rectangle, ellipse* or a *line*. **Collections** of items such as a *path*, which might capture **many lines together** into a *polygon* shape.

Collections are easy to recognise as their name tends to end in the word `collection`

Primitive Artists & Composite Artists

There are two types of Artists in the hierarchy.

- **Primitive artists** represent the kinds of objects you see in a plot: *Line2D, Rectangle, Circle,* and *Text*.
- **Composite artists** are collections of Artists such as the *Axis, Tick, Axes,* and *Figure*. **Each composite artistmay contain other composite artists as well as primitive artists.**

For example, the Figure contains one or more composite Axes and the background of the Figure is a primitive Rectangle.

'Axes' Object

The most important composite artist is the `Axes`, as this is where most matplotlib API plotting methods are defined. Not only does the `Axes` contain most of the graphical elements that make up the background of the plot: the *ticks*, the *axis lines*, the *grid*, the *plot background*, and contains numerous **helper methods** that create primitive artists and add them to the `Axes` instance.
This is the most common one we'll interact with, where we can **change the range of a given axis**, **or plotting shapes to it**.

'Axes' object methods

Method	Creates	Stores in

Method	Creates	Stores in
`Axes.imshow`	one or more `matplotlib.image.AxesImage` objects.	`Axes.images`
`Axes.hist`	one or more `matplotlib.patch.Rectangle` objects.	`Axes.patches`
`Axes.plot`	one or more `matplotlib.lines.Line2D` objects.	`Axes.lines`

Interaction between Artist & Backend Layer

The coupling between the `Artist` hierarchy and the `backend` happens in the *draw* method. For example, in the mockup class below where we create `SomeArtist` which subclasses Artist, the essential method that **SomeArtist must implement is draw, which is passed a renderer from the backend.** The Artist doesn't know what kind of backend the renderer is going to draw onto (PDF, SVG, GTK+ DrawingArea, etc.) but it does know the **Renderer API and will call the appropriate method** (`draw_text` or `draw_path`). Since the Renderer has a *pointer* to its canvas and knows how to paint onto it, the *draw* method transforms the abstract representation of the Artist to colors in a pixel buffer, paths in an SVG file, or any other concrete representation.

Top: Scripting Layer (pyplot):

Pyplot and other scripting layers is a stateful interface that handles much of the boilerplate for creating figures and axes and connecting them to the backend of your choice, and maintains **module-level internal data structures** representing the current figure and axes to which it directs plotting commands.

Analysis of Important Code Commands on the Scripting Layer:

- `import matplotlib.pyplot as plt`: When the *pyplot* module is loaded, it **parses a local configuration file in which the user states** , among many other things, **their preference for a default backend**. For instance,this might be a user interface backend like `QtAgg`.
- `plt.hist(x, 100)` : This is the first plotting command in the script. *Pyplot* will **check** its internal data structures **to see if there is a current `Figure` instance**.
  - If so, it will **extract the current Axes** and direct plotting to the `Axes.hist` API call.
  - In the case whre there is none, **it will create a `Figure` and `Axes`**, set these as current, and direct the plotting ato the `Axes.hist` API call.
- `plt.title(r'Normal distribution with $\mu = 0$, $\sigma = 1$')` : As above, pyplot will check to see if there is an *Axes* and a *figure* and will use the current one if present, else create a new one.
- `plt.show()` : This will force the `Figure` to render, and if the user has indicated a default

Documentation of `matplotlib.pyplot.plot()`

@autogen_docstring(Axes.plot) def plot(*args, **kwargs): ax = gca()

```
ret = ax.plot(*args, **kwargs)
draw_if_interactive()

return ret
```

The `*arg` and `**kwargs` in the documentation signature are special conventions in Python to mean all the arguments and keyword arguments that are passed to the method. This allows us to forward them onto the corresponding API method. The call `ax = gca()` invokes the stateful machinery "get current Axes" (each Python interpreter can have only 1 current axes), and will create the `Figure` and `Axes` if necessary. The call to `ret= ax.plot(*args, **kwargs)` forwards the function call and its arguments to the appropriate `Axes` method, and storesthe return value to be returned later.

Thus, the pyplot interface is a **fairly thin wrapper** around the core `Artist` API, which tries to avoid as much code duplication as possible by exposing the API function, call signature and docstring in the scripting interface with a minimal amount of boilerplate code.

Procedural & Declarative Methods for Visualising Data

- The scripting layer we use in this course is called `pyplot`
    - The pyplot scripting layer is a **procedural method for building a visualisation**, in that we tell the underlying software which drawing actions we want it to take in order to render our data.
    - There are also **declarative methods for visualising our data** - for instance (HTML. HTML Documents are formatted as model of relationships in a document, often called the DOM or Document Object Model.

      These are 2 different ways of creating and representing graphical interfaces. The popular JS library, for instance, is an example of a declarative information visualisation method, while matplotlib's `pyplot` is a procedural information visualisation method.

02-02: matplotlib's Plot function

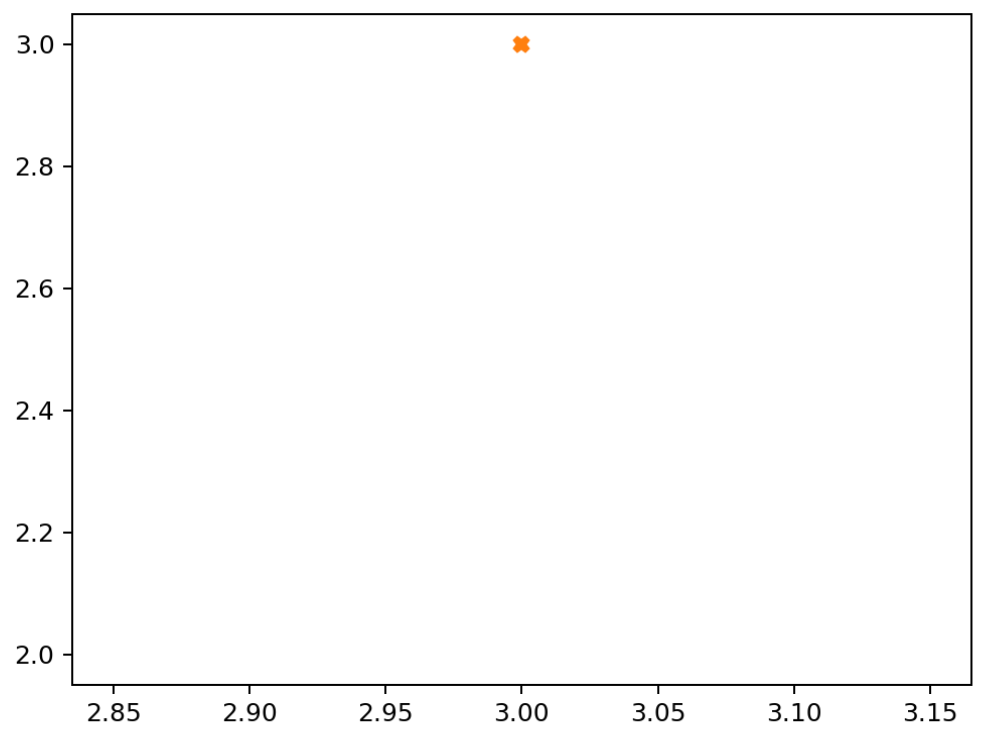
`plot(x,y)` takes in 2 parameters one for the x axis and one for y axis.

In [46]:

```
import matplotlib.pyplot as plt
plt.plot?
# The function supports any number of unnamed arguments.
# The function supports any number of named arguments
# This makes the function very flexible! Just that we don't know what is a prope
r argument.
```

In [50]:

```
# because the default is the line style '-',
# nothing will be shown if we only pass in one point (3,2)
plt.figure()
plt.plot(3, 2)
```



Out[50]:

```
[<matplotlib.lines.Line2D at 0x7f61cf6f7208>]
```

In [51]:

```
# we can pass in '.' to plt.plot to indicate that we want
# the point (3,2) to be indicated with a marker '.'
# you will realise only now you can see your datapoint, but it is on the origina
l plot.
# Note that the %matplotlib inline creates multiple plots each time you call tha
t function, which can be handy as well.
plt.plot(3, 3, 'X')
```

Out[51]:

```
[<matplotlib.lines.Line2D at 0x7f61cf72a6a0>]
```

What The scripting Layer is Doing:

The scripting layer does **behind the scenes work** just as we previously learnt. For instance, when we make a call to `pyplots plt.plot`, the scripting layer **actually looks to see if there's a figure that currently exists, and if not, it creates a new one.** It then **returns the axes** for this figure.

Let's see how to make a plot without using the scripting layer.

Directly interfacing with the Artist Layer

In [52]:

```
# First let's set the backend without using mpl.use() from the scripting layer
from matplotlib.backends.backend_agg import FigureCanvasAgg
from matplotlib.figure import Figure

# create a new figure
fig = Figure()

# associate fig with the backend
canvas = FigureCanvasAgg(fig)

# add a subplot to the fig
ax = fig.add_subplot(111) # this number "111" means we just want 1 plot.
# What we get back is an axis object.
# plot the point (3,2)
ax.plot(3, 2, 'X')

# save the figure to test.png
# you can see this figure in your Jupyter workspace afterwards by going to
# https://hub.coursera-notebooks.org/
canvas.print_png('test.png')
```

Jupyter notebooks cannot run this object immediately, because it expects pyplot to be creating all the objects not this FigureCanvasAgg

So first, we save the figure to a HTML file instead.

We can use html cell magic to display the image.

In [53]:

```
%%html
<img src='test.png' />
```

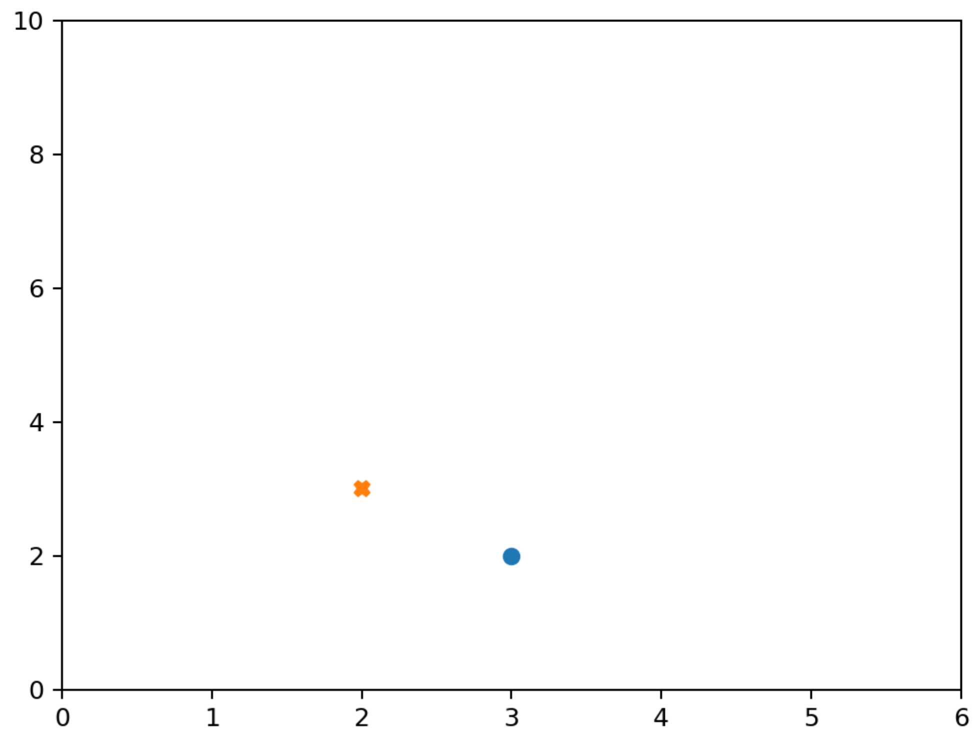
Scripting Layer: Using `gca()` , `gcf()` and `axis([xmin, xmax, ymin, ymax])`

To get access to the figure, you use the `gcf()` function, which **gets** the **current figure** of pyplot.

To get access to the axes, use the `gca()` function. Refer to the examples below for a reference.

In [54]:

```
# create a new figure
plt.figure()

# plot the point (3,2) using the circle marker
plt.plot(3, 2, 'o')
plt.plot(2, 3, 'X')

# get the current axes
ax = plt.gca()

# Set axis properties [xmin, xmax, ymin, ymax]
ax.axis([0,6,0,10])
```

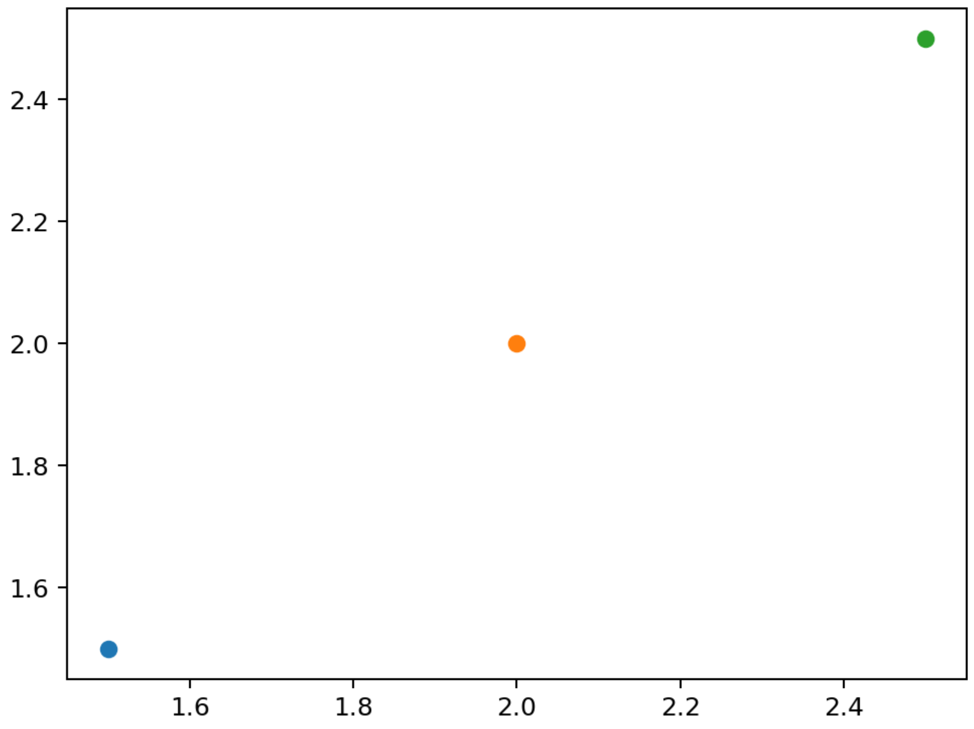


Out[54]:

```
[0, 6, 0, 10]
```

In [55]:

```
# create a new figure
plt.figure()

# plot the point (1.5, 1.5) using the circle marker
plt.plot(1.5, 1.5, 'o')
# plot the point (2, 2) using the circle marker
plt.plot(2, 2, 'o')
# plot the point (2.5, 2.5) using the circle marker
plt.plot(2.5, 2.5, 'o')
```



Out[55]:

```
[<matplotlib.lines.Line2D at 0x7f61cf56ff60>]
```

Using `get_children()`

In [10]:

```
# get current axes
ax = plt.gca()
# get all the child objects the axes contains
ax.get_children()
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x7f61ea8f5198>,
 <matplotlib.lines.Line2D at 0x7f61ed0914a8>,
 <matplotlib.lines.Line2D at 0x7f61ea8f5b00>,
 <matplotlib.spines.Spine at 0x7f61ed091e10>,
 <matplotlib.spines.Spine at 0x7f61ed09a048>,
 <matplotlib.spines.Spine at 0x7f61ed09a240>,
 <matplotlib.spines.Spine at 0x7f61ed09a438>,
 <matplotlib.axis.XAxis at 0x7f61ed09a5f8>,
 <matplotlib.axis.YAxis at 0x7f61ed0a7c18>,
 <matplotlib.text.Text at 0x7f61ed04e7b8>,
 <matplotlib.text.Text at 0x7f61ed04e828>,
 <matplotlib.text.Text at 0x7f61ed04e898>,
 <matplotlib.patches.Rectangle at 0x7f61ed04e8d0>]
```

02-03: Scatterplots

MatplotLib actually has a number of useful plotting methods in the **scripting layer**, which corresponds to different kinds of plots. We'll focus on some major ones.

Keep in Mind...

- Pyplot is going to retrieve the current figure **with the function `gcf()`** and then get the current axes **with the function `gca()`** .
- Pyplot is keeping track of the axes objects for you. But don't forget that they're there and we can get them when we want to get them using **the function `get_children()`**.
- Pyplot just **mirrors the API** of the axis objects, so you can call the plot function against the pyplot module. **But this is calling the axis functions underneath**.
- Remember that the function declaration of the functions in matplotlib **end with an open set of keyword arguments (**kwargs).** There are a lot of different proprties that you can control through these keyword arguments.

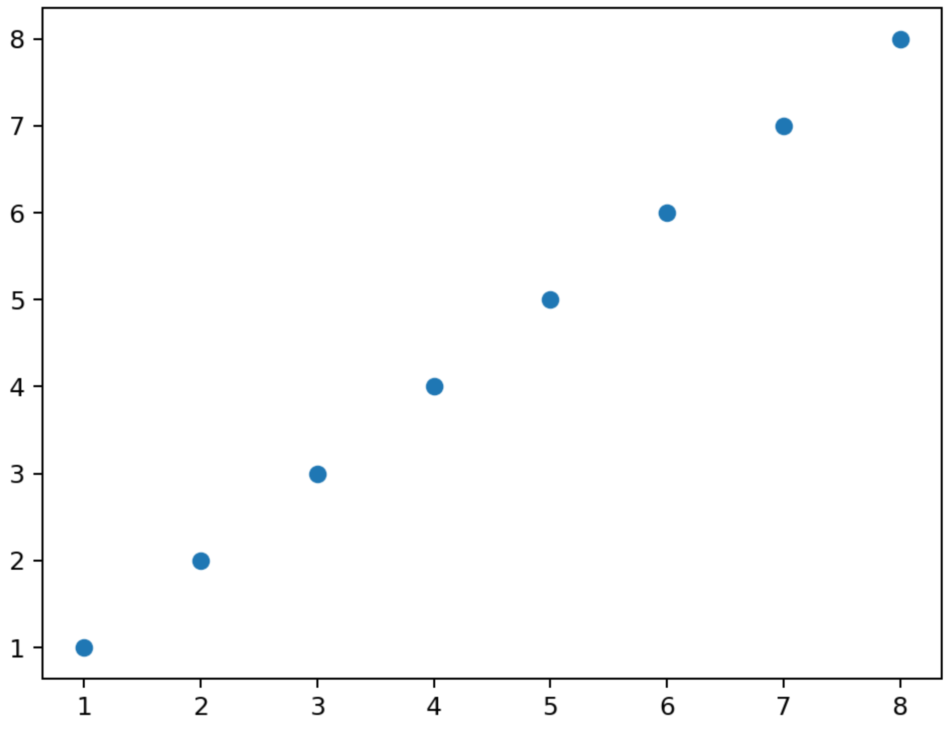
Understanding the matplotlib documentation is really key to understanding what options you have available to you.

Scripting Layer: Scatterplot Syntax

The `scatter()` function takes an x-axis value as a first argument and a y-axis value as the 2nd argument. If the two arguments are the same, we get a nice diagonal alignment of points.

In [56]:

```
import numpy as np

x = np.array([1,2,3,4,5,6,7,8])
y = x

plt.figure()
plt.scatter(x, y) # similar to plt.plot(x, y, '.'), but the underlying child obj
ects in the axes are not Line2D
```



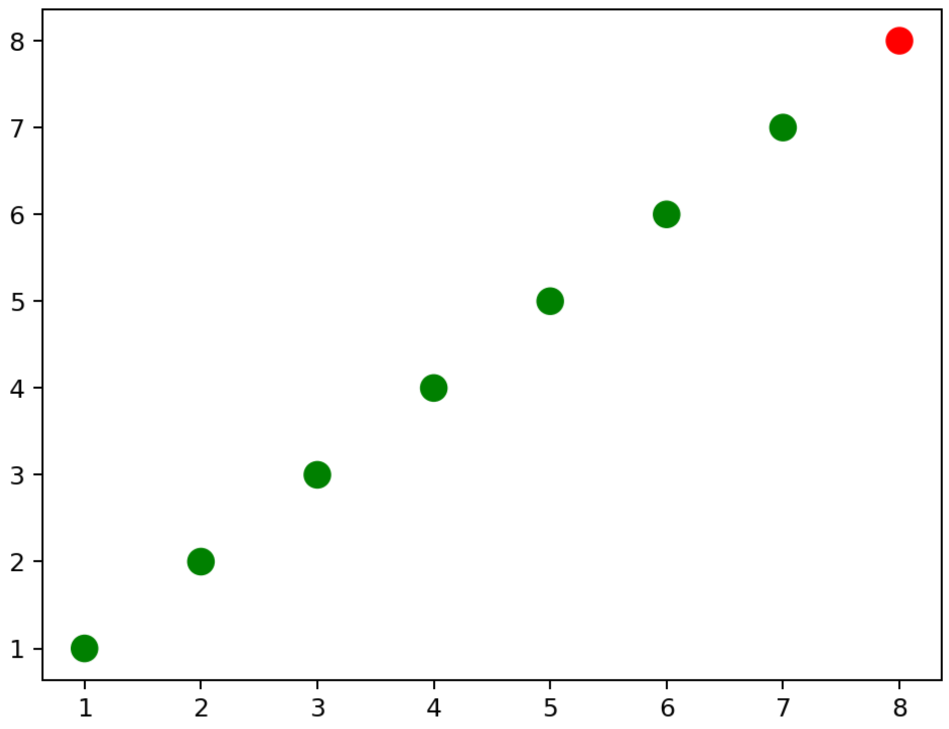
Out[56]:

```
<matplotlib.collections.PathCollection at 0x7f61cf48c898>
```

Changing Colors of Datapoints

In [57]:

```
import numpy as np

x = np.array([1,2,3,4,5,6,7,8])
y = x

# create a list of colors for each point to have
# ['green', 'green', 'green', 'green', 'green', 'green', 'green', 'red']
colors = ['green']*(len(x)-1)
colors.append('red')

plt.figure()

# plot the point with size 100 and chosen colors
plt.scatter(x, y, s=100, c=colors)
```



Out[57]:

```
<matplotlib.collections.PathCollection at 0x7f61cf3cb9e8>
```

In other OOP languages, they may conceptualise as a data point to be **its own instance**, which encapsulates all of its properties. For instance it has an x-value attribute and a y-value attribute, a color attribute and a size attribute. In Matplotlib however, datapoints are represented as some **slice of a larger dataset involved**. This is where it's useful to have some knowledge of **list comprehensions** and **lambdas** and **advanced accumulation** functions.

`zip()` Function & List Unpacking Techniques

Recall the `zip()` function makes a tuple out of two lists of numbers, matching elements based on index. Also, the `zip()` function has **lazy evaluation** because it is actually a generator in Python 3, which means we need to use **a list function** to see the results of iterating over zip().

In [58]:

```
# convert the two lists into a list of pairwise tuples
zip_generator = zip([1,2,3,4,5], [6,7,8,9,10])

print(list(zip_generator))
# the above prints:
# [(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]

zip_generator = zip([1,2,3,4,5], [6,7,8,9,10]) # python function
# The single star * unpacks a collection into positional arguments
print(*zip_generator)
# the above prints:
# (1, 6) (2, 7) (3, 8) (4, 9) (5, 10)
```

```
[(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
(1, 6) (2, 7) (3, 8) (4, 9) (5, 10)
```

In [59]:

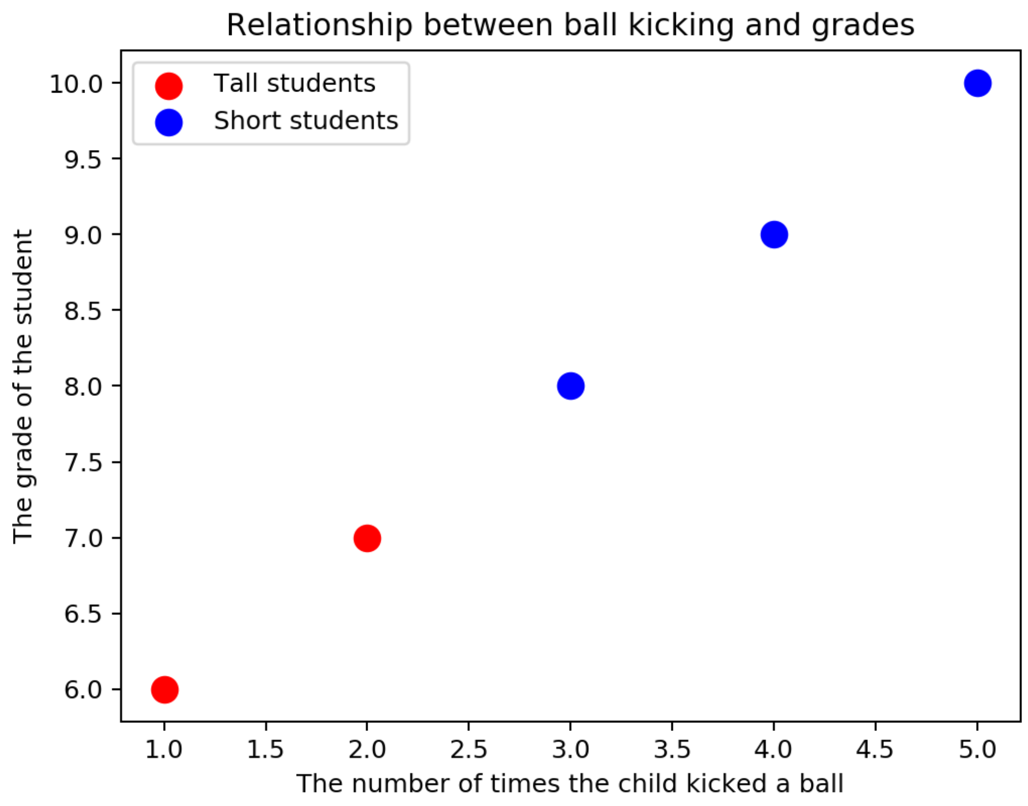
```
# use zip to convert 5 tuples with 2 elements each to 2 tuples with 5 elements e
ach
print(list(zip((1, 6), (2, 7), (3, 8), (4, 9), (5, 10)))) # you see that the zip
function works because
#they match elements based on syntax!
# the above prints:
# [(1, 2, 3, 4, 5), (6, 7, 8, 9, 10)]


zip_generator = zip([1,2,3,4,5], [6,7,8,9,10])
# let's turn the data back into 2 lists
x, y = zip(*zip_generator) # This is like calling zip((1, 6), (2, 7), (3, 8),
 (4, 9), (5, 10))
print(x)
print(y)
# the above prints:
# (1, 2, 3, 4, 5)
# (6, 7, 8, 9, 10)
```

```
[(1, 2, 3, 4, 5), (6, 7, 8, 9, 10)]
(1, 2, 3, 4, 5)
(6, 7, 8, 9, 10)
```

Passing Parameters into Scatter()

```
In [60]:
```

```
plt.figure()
# plot a data series 'Tall students' in red using the first two elements of x an
d y
plt.scatter(x[:2], y[:2], s=100, c='red', label='Tall students')
# plot a second data series 'Short students' in blue using the last three elemen
ts of x and y
plt.scatter(x[2:], y[2:], s=100, c='blue', label='Short students')
```



```
Out[60]:

<matplotlib.collections.PathCollection at 0x7f61cf307588>
```

```
In [62]:
```

```
# add a label to the x axis
plt.xlabel('The number of times the child kicked a ball')
# add a label to the y axis
plt.ylabel('The grade of the student')
# add a title
plt.title('Relationship between ball kicking and grades')
```

```
Out[62]:

<matplotlib.text.Text at 0x7f61cf37f0b8>
```

In [61]:

```
# add a legend (uses the labels from plt.scatter)
plt.legend()
```

Out[61]:

```
<matplotlib.legend.Legend at 0x7f61cf2ee7b8>
```

In [18]:

```
# add the legend to loc=4 (the lower right hand corner), also gets rid of the fr
ame and adds a title
plt.legend(loc=4, frameon=False, title='Legend')
```

Out[18]:

```
<matplotlib.legend.Legend at 0x7f61a04e4cc0>
```

In [19]:

```
# get children from current axes (the legend is the second last to last item in
 this list)
plt.gca().get_children()
```

Out[19]:

```
[<matplotlib.collections.PathCollection at 0x7f61a04d5eb8>,
 <matplotlib.collections.PathCollection at 0x7f61a04dda58>,
 <matplotlib.spines.Spine at 0x7f61a04f6320>,
 <matplotlib.spines.Spine at 0x7f61a04f6518>,
 <matplotlib.spines.Spine at 0x7f61a04f6710>,
 <matplotlib.spines.Spine at 0x7f61a04f6908>,
 <matplotlib.axis.XAxis at 0x7f61a04f6ac8>,
 <matplotlib.axis.YAxis at 0x7f61a0504b00>,
 <matplotlib.text.Text at 0x7f61a052a908>,
 <matplotlib.text.Text at 0x7f61a052a978>,
 <matplotlib.text.Text at 0x7f61a052a9e8>,
 <matplotlib.legend.Legend at 0x7f61a04e4cc0>,
 <matplotlib.patches.Rectangle at 0x7f61a052aa20>]
```

In [64]:

```
# get the legend from the current axes
legend = plt.gca().get_children()[-2]
```

In [65]:

```
# you can use get_children to navigate through the child artists
legend.get_children()[0].get_children()[1].get_children()[0].get_children()
```

Out[65]:

```
[<matplotlib.offsetbox.HPacker at 0x7f61cf3554e0>,
 <matplotlib.offsetbox.HPacker at 0x7f61cf355470>]
```

In [66]:

```
# import the artist class from matplotlib
from matplotlib.artist import Artist

def rec_gc(art, depth=0):
    if isinstance(art, Artist):
        # increase the depth for pretty printing
        print("  " * depth + str(art))
        for child in art.get_children():
            rec_gc(child, depth+2) #recursion

# Call this function on the legend artist to see what the legend is made up of
rec_gc(plt.legend())
```

```
Legend
    <matplotlib.offsetbox.VPacker object at 0x7f61cf3c1240>
        <matplotlib.offsetbox.TextArea object at 0x7f61cf2ee8d0>
            Text(0,0,'None')
        <matplotlib.offsetbox.HPacker object at 0x7f61cf2e7e10>
            <matplotlib.offsetbox.VPacker object at 0x7f61cf2e7da0>
                <matplotlib.offsetbox.HPacker object at 0x7f61cf2eeb
a8>
                    <matplotlib.offsetbox.DrawingArea object at 0x7f
61cf2e70b8>
                        <matplotlib.collections.PathCollection objec
t at 0x7f61cf2e77b8>
                    <matplotlib.offsetbox.TextArea object at 0x7f61c
f2e7a20>
                        Text(0,0,'Tall students')
                <matplotlib.offsetbox.HPacker object at 0x7f61cf2ee2
78>
                    <matplotlib.offsetbox.DrawingArea object at 0x7f
61cf2ee358>
                        <matplotlib.collections.PathCollection objec
t at 0x7f61cf2ee048>
                    <matplotlib.offsetbox.TextArea object at 0x7f61c
f2e7400>
                        Text(0,0,'Short students')
    FancyBboxPatch(0,0;1x1)
```

Summary

There is nothing magical about what matplotlib is doing. Calls to the scripting interface, just create figures, subplots and axes, etc. Then load these axes up with various artists, which the back-end renders to the screen or some other medium like a file. While you'll spend 95% of your time at the scripting layer, happily creating graphs and charts, it's important to understand how the library works underneath for the other 5% of the time.

The time that you'll tap into this understanding is when you really want to have control over, and to create your own charting functions.

02-04: Line Plots

There are a few differences between this and scatterplots:

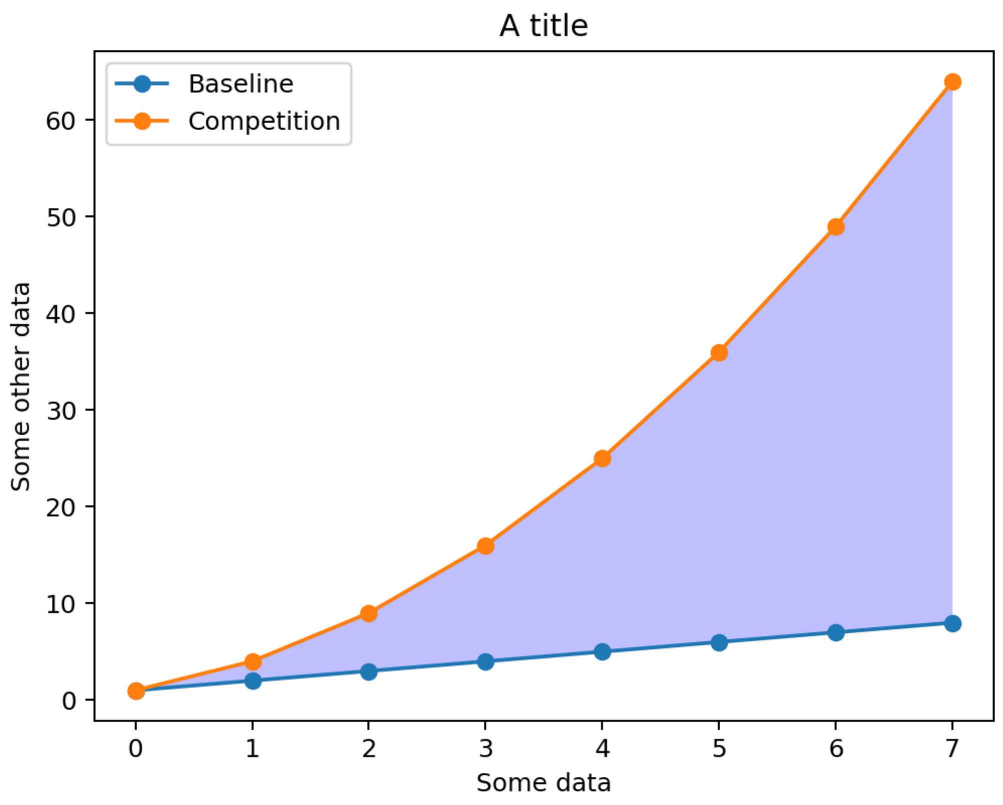
- First, we **only gave y-axis values** to our plot call, **no x-axis values**.
- Instead the plot function was smart enough to figure out that we wanted to use the **index** of the series as our x value, which is pretty handy when you want to make quick plots.

This is different from the scatterplot where we would have to label the points specifically.

```
In [67]:
```

```python
import numpy as np

linear_data = np.array([1,2,3,4,5,6,7,8])
exponential_data = linear_data**2 ## using broadcasting in numpy

plt.figure()
# plot the linear data and the exponential data
plt.plot(linear_data, '-o', exponential_data, '-o')
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



```
Out[67]:

[<matplotlib.lines.Line2D at 0x7f61f4768438>,
 <matplotlib.lines.Line2D at 0x7f61f47687f0>]
```

Like all python libraries, matplotlib allows for *string based mini language* for commonly used formatting. For instance, we could use an s inside of the formatting string which would plot another point using a square marker. Or we can use a series of dashes and dots to identify that a line should be dashed instead of solid.

In [24]:

```
# plot another series with a dashed red line
plt.plot([22,44,55], '--r')
```

Out[24]:

```
[<matplotlib.lines.Line2D at 0x7f61a04842b0>]
```

In [68]:

```
plt.xlabel('Some data')
plt.ylabel('Some other data')
plt.title('A title')
# add a legend with legend entries (because we didn't have labels when we plotte
d the data series)
plt.legend(['Baseline', 'Competition', 'Us'])
```

Out[68]:

```
<matplotlib.legend.Legend at 0x7f61cf63b668>
```

Highlight space between 2 graphs using `fill_between()` functions

In [69]:

```
# fill the area between the linear data and exponential data
plt.gca().fill_between(range(len(linear_data)),
                       linear_data, exponential_data,  # lower bounds, upper bou
nd
                       facecolor='blue', # color
                       alpha=0.25) #transparency (max= 1)
```

Out[69]:

```
<matplotlib.collections.PolyCollection at 0x7f61cf3199b0>
```

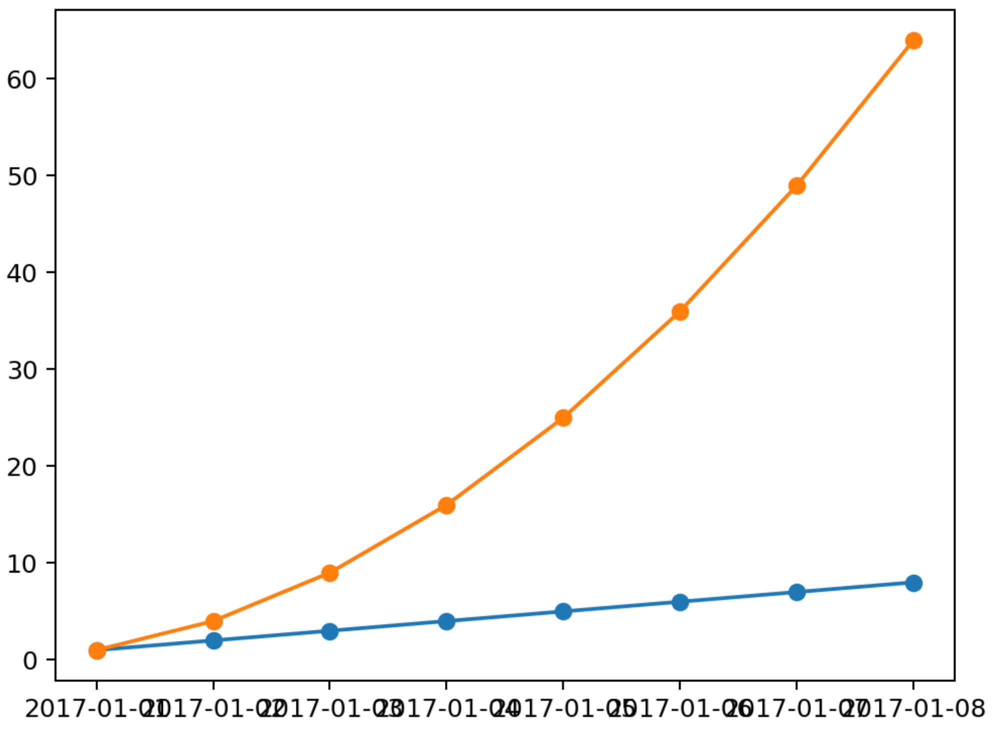
Using Datetime

Let's try working with dates!

In [70]:

```
plt.figure()

observation_dates = np.arange('2017-01-01', '2017-01-09', dtype='datetime64[D]')

plt.plot(observation_dates, linear_data, '-o',  observation_dates, exponential_d
ata, '-o')
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



Out[70]:

```
[<matplotlib.lines.Line2D at 0x7f61ef907940>,
 <matplotlib.lines.Line2D at 0x7f61ef936e10>]
```

DateTime Handling in Python

Unfortunately, as we can see above, that didn't give us the result we wanted.

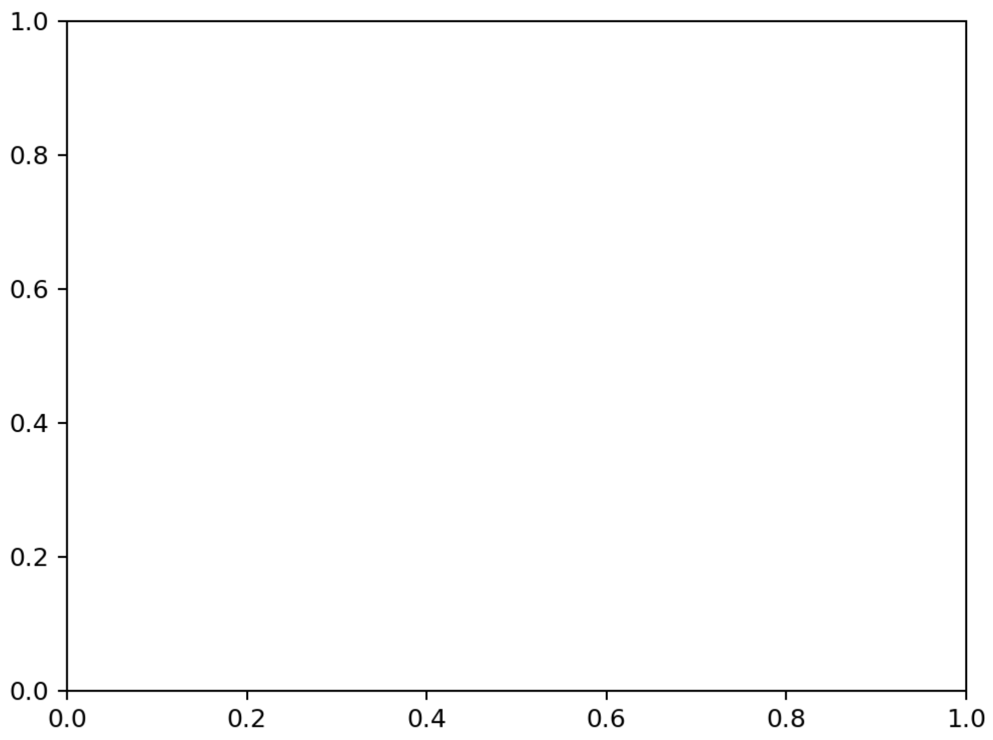
This is one of the painpoints of the language - the standard library does it in two different ways, while NumPy, which is used for scientific programming, does it a third way. In fact, there's probably a **dozen replacement libraries** for date times in Python.

DateTime Handling Using Pandas

One possible solution comes from a **helper library in Pandas** called `to_datetime()`. This specifically converts NumPy dates into standard library dates, which is what matplotlib is expecting.

In [28]:

```
import pandas as pd

plt.figure()
observation_dates = np.arange('2017-01-01', '2017-01-09', dtype='datetime64[D]')
observation_dates = map(pd.to_datetime, observation_dates) # trying to plot a ma
p will result in an error
# This will plot a function that will apply to_datetime for each date in observa
tion_dates, however, the results are in an iterator, not a list.
print(type(observation_dates))
plt.plot(observation_dates, linear_data, '-o',  observation_dates, exponential_d
ata, '-o')
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

```
<class 'map'>
```

```
---------------------------------------------------------------------
-------
AttributeError                            Traceback (most recent cal
l last)
/opt/conda/lib/python3.6/site-packages/matplotlib/units.py in get_co
nverter(self, x)
    144                 # get_converter
--> 145                 if not np.all(xravel.mask):
    146                     # some elements are not masked

AttributeError: 'numpy.ndarray' object has no attribute 'mask'

During handling of the above exception, another exception occurred:

TypeError                                 Traceback (most recent cal
l last)
<ipython-input-28-7d2fc3e609e4> in <module>()
      6 # This will plot a function that will apply to_datetime for
 each date in observation_dates, however, the results are in an iter
ator, not a list.
      7 print(type(observation_dates))
----> 8 plt.plot(observation_dates, linear_data, '-o',  observation_
dates, exponential_data, '-o')

/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py in plot
(*args, **kwargs)
   3316                       mplDeprecation)
   3317     try:
-> 3318         ret = ax.plot(*args, **kwargs)
   3319     finally:
   3320         ax._hold = washold

/opt/conda/lib/python3.6/site-packages/matplotlib/__init__.py in inn
er(ax, *args, **kwargs)
   1890                     warnings.warn(msg % (label_namer, func._
_name__),
   1891                                   RuntimeWarning, stacklevel
=2)
-> 1892             return func(ax, *args, **kwargs)
   1893         pre_doc = inner.__doc__
   1894         if pre_doc is None:

/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_axes.py in p
lot(self, *args, **kwargs)
   1404         kwargs = cbook.normalize_kwargs(kwargs, _alias_map)
   1405 
-> 1406         for line in self._get_lines(*args, **kwargs):
   1407             self.add_line(line)
   1408             lines.append(line)

/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_base.py in _
grab_next_args(self, *args, **kwargs)
    414                 isplit = 2
    415 
--> 416             for seg in self._plot_args(remaining[:isplit], k
wargs):
    417                 yield seg
    418             remaining = remaining[isplit:]

/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_base.py in _
plot_args(self, tup, kwargs)
```

```
    383             x, y = index_of(tup[-1])
    384
--> 385         x, y = self._xy_from_xy(x, y)
    386
    387         if self.command == 'plot':

/opt/conda/lib/python3.6/site-packages/matplotlib/axes/_base.py in _
xy_from_xy(self, x, y)
    215     def _xy_from_xy(self, x, y):
    216         if self.axes.xaxis is not None and self.axes.yaxis i
s not None:
--> 217             bx = self.axes.xaxis.update_units(x)
    218             by = self.axes.yaxis.update_units(y)
    219

/opt/conda/lib/python3.6/site-packages/matplotlib/axis.py in update_
units(self, data)
   1411         """
   1412
-> 1413         converter = munits.registry.get_converter(data)
   1414         if converter is None:
   1415             return False

/opt/conda/lib/python3.6/site-packages/matplotlib/units.py in get_co
nverter(self, x)
    156                 if (not isinstance(next_item, np.ndarray) or
    157                     next_item.shape != x.shape):
--> 158                     converter = self.get_converter(next_item
)
    159                 return converter
    160

/opt/conda/lib/python3.6/site-packages/matplotlib/units.py in get_co
nverter(self, x)
    159                 return converter
    160
--> 161         if converter is None and iterable(x) and (len(x) > 0
):
    162             thisx = safe_first_element(x)
    163             if classx and classx != getattr(thisx, '__class_
_', None):

TypeError: object of type 'map' has no len()
```

Hmm it results in an error. The `map()` function **returns an iterator object**, as you can see above from `class 'map'` .

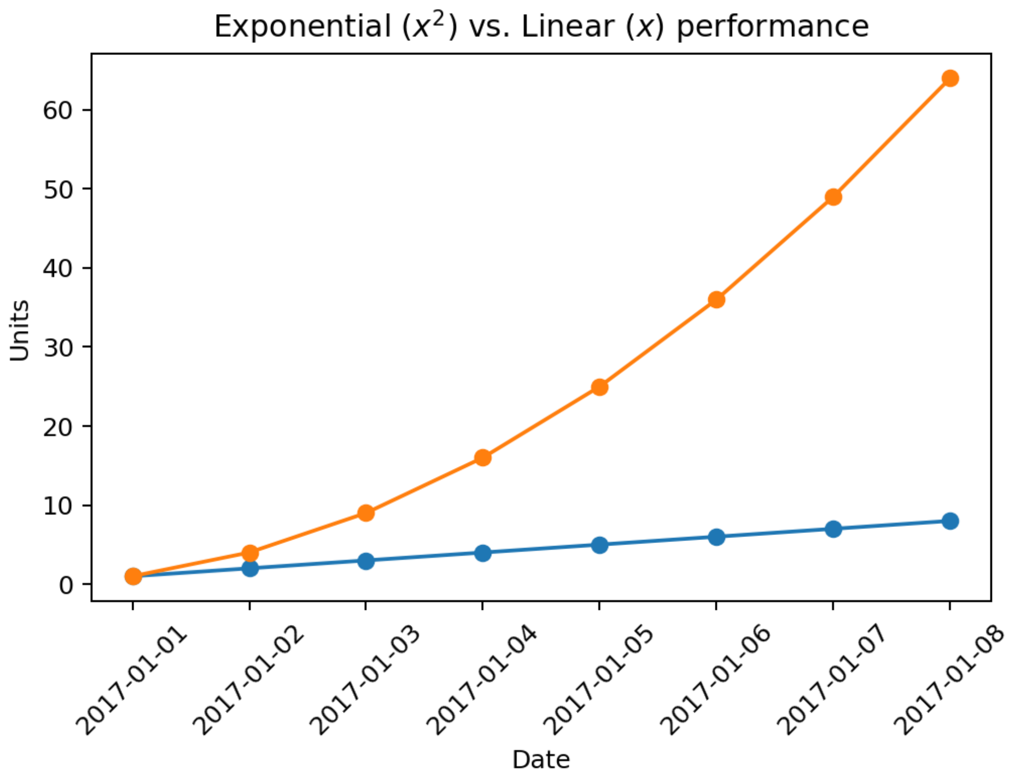
Matplotlib can't handle the iterator, so we need to convert it to a list first. These are some of the real problems you can run into when you're building plots in Python.

Here's a quick fix

```
In [72]:
```

```
import pandas as pd

plt.figure()
observation_dates = np.arange('2017-01-01', '2017-01-09', dtype='datetime64[D]')
observation_dates = list(map(pd.to_datetime, observation_dates)) # turned the ma
p iterator into a list
# This will plot a function that will apply to_datetime for each date in observa
tion_dates, however, the results are in an iterator, not a list.
print(type(observation_dates))
plt.plot(observation_dates, linear_data, '-o',  observation_dates, exponential_d
ata, '-o')
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



```
<class 'list'>
```

```
Out[72]:
```

```
[<matplotlib.lines.Line2D at 0x7f61ef8c6160>,
 <matplotlib.lines.Line2D at 0x7f61ef87bb70>]
```

Axes Objects

There are lots of interesting properties of the *axes* object and you use some of them in the assignment.

For instance, you can get the grid lines, tick locations for both the major and minor ticks and so on.

Just like all artists, an axes **has also a bunch of children** which are themselves artists.

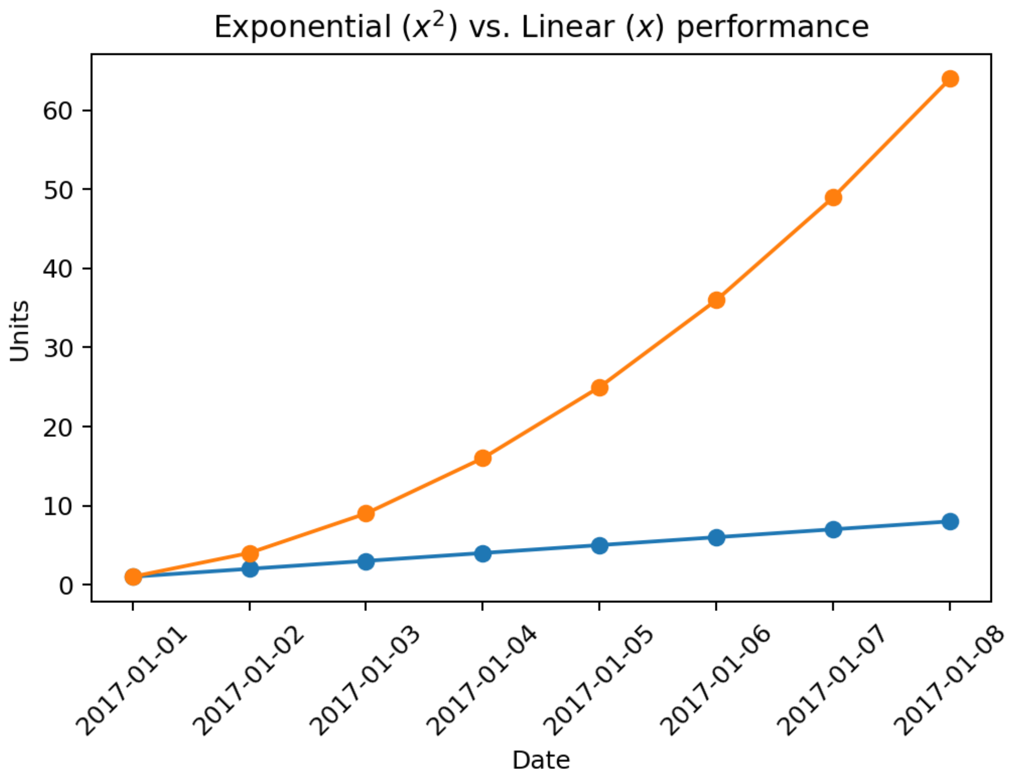
In [71]:

```
## Running the rec_gc() function to see what is a child of this axes object.

# Call this function on the legend artist to see what the legend is made up of
rec_gc(plt.axes())
```

```
Axes(0.125,0.11;0.775x0.77)
    Line2D(_line0)
    Line2D(_line1)
    Spine
    Spine
    Spine
    Spine
    XAxis(143.999996,95.039997)
        Text(0.5,43.54,'')
        Text(1,46.04,'')
        <matplotlib.axis.XTick object at 0x7f61ef90c588>
            Line2D((736330,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736330,0,'2017-01-01')
            Text(0,1,'2017-01-01')
        <matplotlib.axis.XTick object at 0x7f61cf39c080>
            Line2D((736331,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736331,0,'2017-01-02')
            Text(0,1,'2017-01-02')
        <matplotlib.axis.XTick object at 0x7f61ef936c18>
            Line2D((736332,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736332,0,'2017-01-03')
            Text(0,1,'2017-01-03')
        <matplotlib.axis.XTick object at 0x7f61ef8cbbe0>
            Line2D((736333,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736333,0,'2017-01-04')
            Text(0,1,'2017-01-04')
        <matplotlib.axis.XTick object at 0x7f61ef8d15f8>
            Line2D((736334,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736334,0,'2017-01-05')
            Text(0,1,'2017-01-05')
        <matplotlib.axis.XTick object at 0x7f61ef8d1fd0>
            Line2D((736335,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736335,0,'2017-01-06')
            Text(0,1,'2017-01-06')
        <matplotlib.axis.XTick object at 0x7f61ef8d79e8>
            Line2D((736336,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736336,0,'2017-01-07')
            Text(0,1,'2017-01-07')
        <matplotlib.axis.XTick object at 0x7f61ef8dc400>
            Line2D((736337,0))
            Line2D()
            Line2D((0,0),(0,1))
            Text(736337,0,'2017-01-08')
            Text(0,1,'2017-01-08')
    YAxis(143.999996,95.039997)
        Text(84.625,0.5,'')
        Text(0,767.82,'')
        <matplotlib.axis.YTick object at 0x7f61cf2aac88>
            Line2D()
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,0,'')
            Text(1,0,'')
        <matplotlib.axis.YTick object at 0x7f61f4810438>
            Line2D((0,0))
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,0,'0')
            Text(1,0,'0')
        <matplotlib.axis.YTick object at 0x7f61cf37f048>
            Line2D((0,10))
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,10,'10')
            Text(1,0,'10')
        <matplotlib.axis.YTick object at 0x7f61ef8dc0b8>
            Line2D((0,20))
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,20,'20')
            Text(1,0,'20')
        <matplotlib.axis.YTick object at 0x7f61ef8e95c0>
            Line2D((0,30))
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,30,'30')
            Text(1,0,'30')
        <matplotlib.axis.YTick object at 0x7f61ef8e9f60>
            Line2D((0,40))
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,40,'40')
            Text(1,0,'40')
        <matplotlib.axis.YTick object at 0x7f61ef8ef978>
            Line2D((0,50))
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,50,'50')
            Text(1,0,'50')
        <matplotlib.axis.YTick object at 0x7f61ef8f5390>
            Line2D((0,60))
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,60,'60')
            Text(1,0,'60')
        <matplotlib.axis.YTick object at 0x7f61cf2d99b0>
            Line2D()
            Line2D()
            Line2D((0,0),(1,0))
            Text(0,0,'')
            Text(1,0,'')
    Text(0.5,1,'')
    Text(0,1,'')
    Text(1,1,'')
    Rectangle(0,0;1x1)
```

In [30]:

```
plt.figure()
observation_dates = np.arange('2017-01-01', '2017-01-09', dtype='datetime64[D]')
observation_dates = list(map(pd.to_datetime, observation_dates)) # convert the m
ap to a list to get rid of the error
plt.plot(observation_dates, linear_data, '-o',  observation_dates, exponential_d
ata, '-o')
```



Out[30]:

```
[<matplotlib.lines.Line2D at 0x7f61d1c01c18>,
 <matplotlib.lines.Line2D at 0x7f61d1bd2cf8>]
```

Axes Objects: Rotations using `get_ticklabels()`

You can access the ticks using the `get_ticklabels()` function. Each of the tick labels are a text object which itself is an artist. This means you can use a number of different artist functions.

In [73]:

```
x = plt.gca().xaxis

# rotate the tick labels for the x axis
for item in x.get_ticklabels():
    item.set_rotation(45)
```

In [ ]:

In [74]:

```
# adjust the subplot so the text doesn't run off the image
plt.subplots_adjust(bottom=0.25)
```

In [75]:

```
ax = plt.gca()
ax.set_xlabel('Date')
ax.set_ylabel('Units')
ax.set_title('Exponential vs. Linear performance')
```

Out[75]:

```
<matplotlib.text.Text at 0x7f61ef8c6f98>
```

Matplotlib Connections with LaTeX

Matplotlib has a fairly strong connection with latex. This means that you can use a subset of Latex directly in your labels then matplotlib will render them as equations here. For instance, we can set the title of the x-axis to have an $x^2$ , and we **escape to the latex language using its dollar signs($)**, and this works irregardless of whether latex is installed or not.

In [76]:

```
# you can add mathematical expressions in any text element
ax.set_title("Exponential ($x^2$) vs. Linear ($x$) performance")
```

Out[76]:

```
<matplotlib.text.Text at 0x7f61ef8c6f98>
```

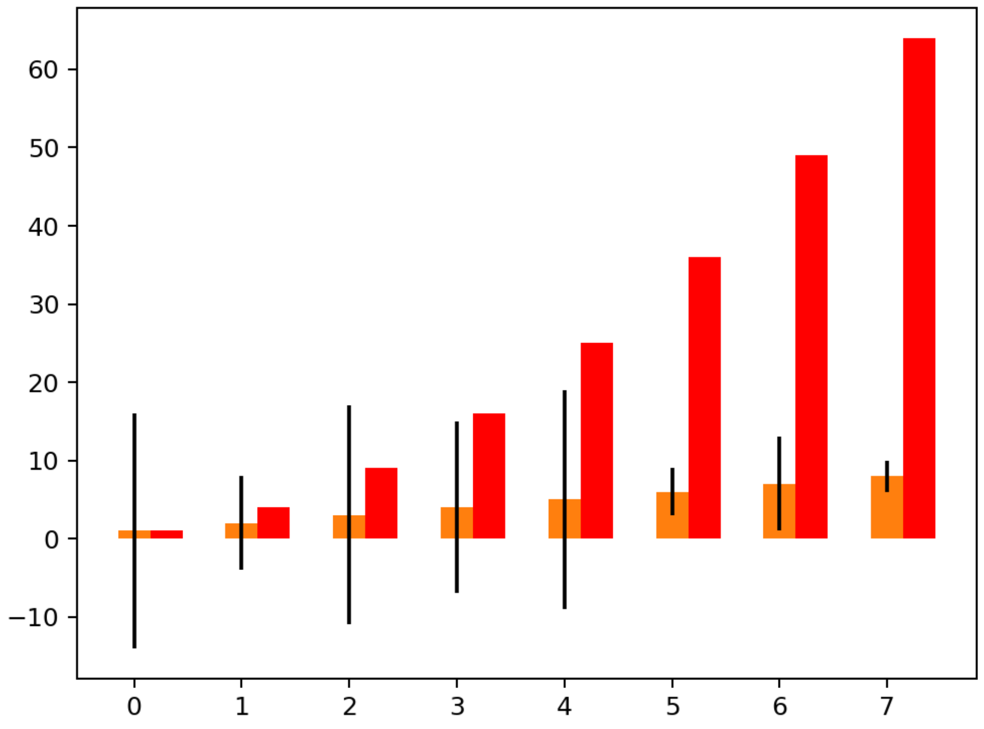
02-05: Bar Charts

Matplotlib has support for several kinds of bar charts.

In [77]:

```
plt.figure()
xvals = range(len(linear_data))
plt.bar(xvals, linear_data, width = 0.3)
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



Out[77]:

```
<Container object of 8 artists>
```

In [78]:

```
new_xvals = []

# plot another set of bars, adjusting the new xvals to make up for the first set
of bars plotted
for item in xvals:
    new_xvals.append(item+0.3)

plt.bar(new_xvals, exponential_data, width = 0.3 ,color='red')
```

Out[78]:

```
<Container object of 8 artists>
```

Creating Error Bars

In [79]:

```
from random import randint
linear_err = [randint(0,15) for x in range(len(linear_data))]

# This will plot a new set of bars with errorbars using the list of random error
values
plt.bar(xvals, linear_data, width = 0.3, yerr=linear_err)
```

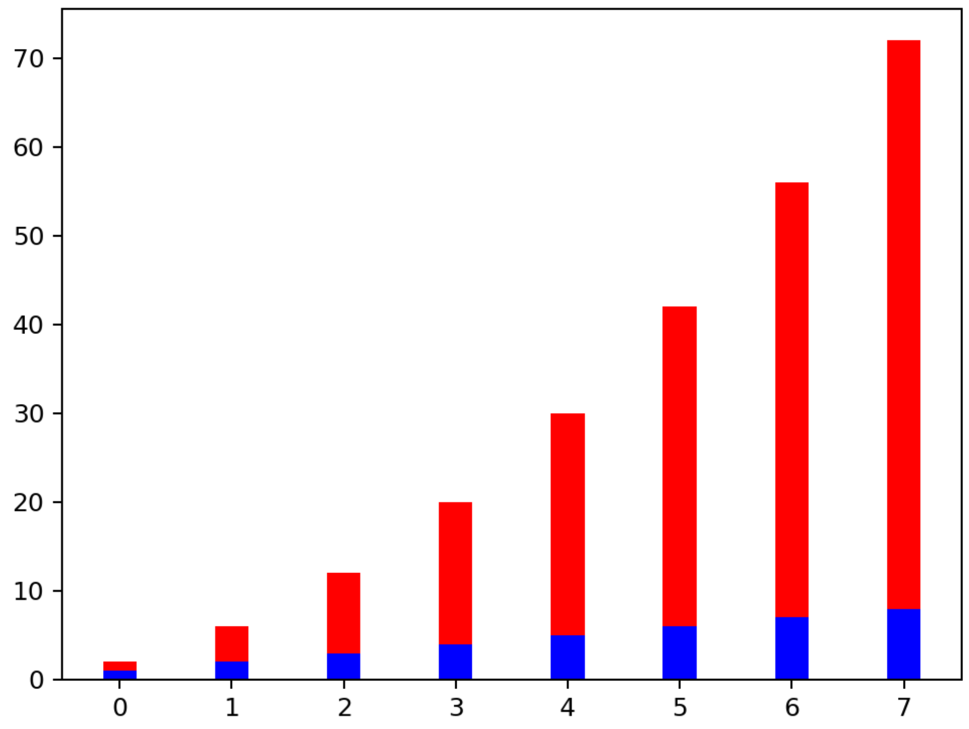
Out[79]:

```
<Container object of 8 artists>
```

Creating Stacked Bar Charts

In [38]:

```python
# stacked bar charts are also possible
plt.figure()
xvals = range(len(linear_data))
plt.bar(xvals, linear_data, width = 0.3, color='b')
plt.bar(xvals, exponential_data, width = 0.3, bottom=linear_data, color='r')
```

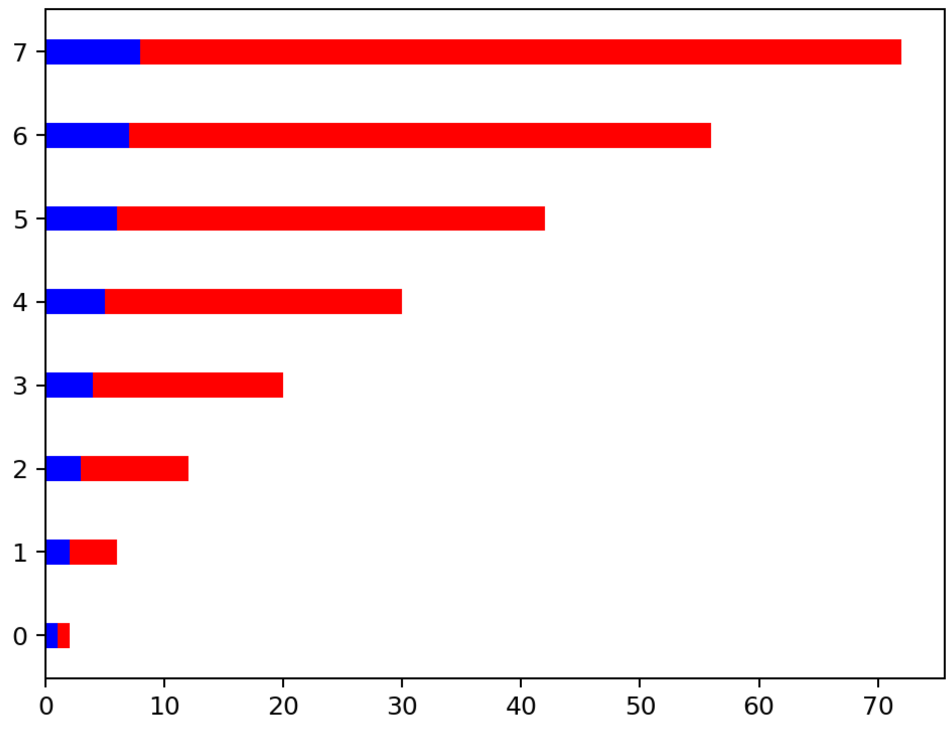


Out[38]:

```
<Container object of 8 artists>
```

```
In [39]:
```

```
# or use barh for horizontal bar charts
plt.figure()
xvals = range(len(linear_data))
plt.barh(xvals, linear_data, height = 0.3, color='b')
plt.barh(xvals, exponential_data, height = 0.3, left=linear_data, color='r')
```



```
Out[39]:
<Container object of 8 artists>
```

Summary

The scripting layer is a set of convenience functions on top of the object layer. Some people share a preference for one or the other. These functions **work together**, and being able to move back and forth between them is very important.

02-06: Dejunkifying A Plot

Here are 5 languages: Python, SQL, Java, C++ and JavaScript. Language: Python => 56%, SQL => 39%, Java => 34%, C++=>34%, JavaScript => 29%

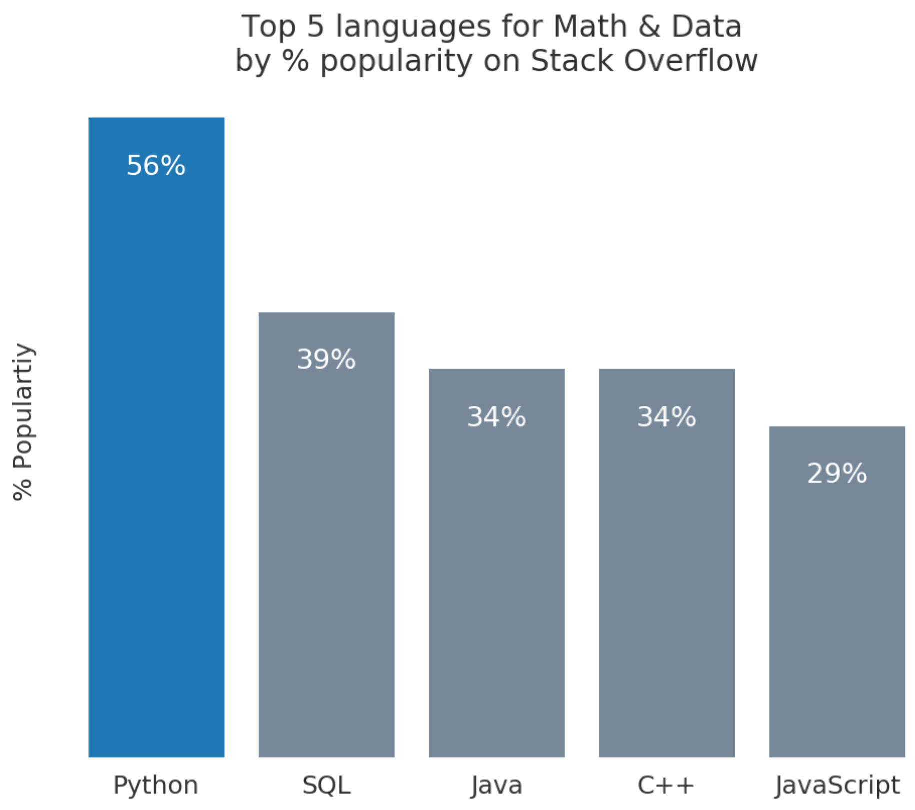
We'll create a bar chart based on rank and popularity, then add x and y ticks and set a title. When we call `plot.show()` the figure will render.

Here's a basic graph with defaults

```
In [81]:
```

```
import matplotlib.pyplot as plt
import numpy as np
plt.figure()

languages = ['Python','SQL', 'Java','C++','JavaScript']
pos = np.arange(len(languages))
popularity = [56, 39,34,34,29]
plt.bar(pos, popularity, width = 0.3, align = 'center')
plt.xticks(pos, languages)
#plt.ylabel('% Popularity') (From question #4)
plt.title('Top 5 Languages for Math & Data\n by % Popularity on Stack Overflow',
alpha=0.8)
plt.show()
```

```
/opt/conda/lib/python3.6/site-packages/matplotlib/pyplot.py:524: Run
timeWarning: More than 20 figures have been opened. Figures created
through the pyplot interface (`matplotlib.pyplot.figure`) are retain
ed until explicitly closed and may consume too much memory. (To cont
rol this warning, see the rcParam `figure.max_open_warning`).
  max_open_warning, RuntimeWarning)
```



Challenge #1 : Remove all the ticks (both axes) and tick labels on the Y axis

In [82]:

```
# Remove all the ticks (both axes) and tick labels on the Y axis
plt.tick_params(top = 'off', bottom = 'off', left = 'off',
                right = 'off', labelleft = 'off', labelbottom = 'on')
plt.show()
```

Challenge #2: Remove The Frame of the Chart using `spine` From Artist Layer

This is a bit more involved, but we can get the current axis, then iterate through all the spine, setting their visibility to false.

Already this chart now looks more lightweight at this point.

In [83]:

```
for spine in plt.gca().spines.values():
    spine.set_visible(False)
plt.show()
```

Challenge #3: Reducing Colours

The blue of each bar is okay, but it doesn't really help us differentiate between the bars at all. How about we soften all of the hard blacks to gray, then we change the bar colours to gray as well.

Also let's keep the Python bar the same color of blue that it was originally to make it stand out.

In [84]:

```
# Change the bar colors to be less bright blue
bars = plt.bar(pos, popularity, align='center', linewidth=0, color = 'lightslate
grey')
#Make 1 bar, the python bar, a contrasting color
bars[0].set_color('#1F77B4') # hexadecimal for the rgb value

#Soften all labels by turning grey
plt.xticks(pos, languages, alpha=0.8)
plt.ylabel('% Populartiy', alpha=0.8)
plt.title('Top 5 languages for Math & Data \nby % popularity on Stack Overflow',
alpha = 0.8)
plt.show()
```

Challenge #4: Remove y-axis label, Directly Label Bars

Removing the label is easy, but changing the bars is a little bit of a pain. We want to iterate over each of the bars and grab its height, then we want to create a new text object with the data information. Unfortunately, this means **you have to play with a padding**.

See how **the x location** is set up: x_location + width/2, and y-location to be the bar's height -5.

In [85]:

```
for bar in bars:
    plt.gca().text(bar.get_x() + bar.get_width()/2,
                   bar.get_height()-5,
                   str(int(bar.get_height()))+'%',
                  ha = 'center',
                  color = 'w',
                  fontsize=11)
plt.show()
```

End notes: 10 Simple Rules For Better Figures

Rule #1: Know Your Audience

Problems arise when how a visual is perceived differs significantly from the intent of the conveyer. Consequently, it is important to identify, as early as possible in the design process, the audience and the message the visual is to convey. **However, you should not be compromising the truth of the dataset.**

Rule #2: Identify Your Message

A figure is meant to express an idea or introduce some facts or a result that would be **too long (or nearly impossible) to explain only with words**, be it an article or during a time-limited oral presentation. Only after identifying the message will it be worth the time to develop your figure.

Rule #3: Adapt the Figure to the Support Medium

A figure can be displayed on a variety of media, such as a poster, a computer monitor, a projection screen or a simple sheet of paper. Each of thesr media represents different physical sizes for the figure, but more importantly, it implies **different ways of viewing and interacting with the figure**.

Rule #4: Captions Are Not Optional

A figure should be accompanied by a caption. The caption explains how to read the figure and provides additional precision for what cannot be graphically represented. This can be thought of as the explanation you would give during an oral presentation, or in front of a poster, but with the difference that you must think in advance about the questions people would ask.

Rule #5: Do Not Trust the Defaults

Any plotting library or software comes with a set of default settings. When the end-user does not specify anything, these default settings are used to specify size, font, colors, styles, ticks and marker, etc. Virtualy any setting can be specified, and you can usually recognise the specific style of each software package or library, thanks to the choice of these default settings.

Rule #6 : Use Color Effectively

Color is an important dimension in human vision and is consequently equally important in the design of a scientific figure. To highlight some element of a figure, you can use colour for this element while keeping other elements gray or black. This produces an enhancing effect, and lowers the **data-ink ratio**.

Rule #7: Do Not Mislead the Reader

What distinguishes a scientifc figure from other graphical artwork is the presence of data that needs to be shown as objectively as possible. A scientific figure is, by definition, tied to the data (be it an experimental setup, a model, or some results), and if you loosen this tie, you may unintentionally project a different message than intended.

Rule #8: Avoid 'Chart Junk'

Chartjunk refers to all unnecessary or confousing visual elements found in a figure that do not improve the message (in the best case) or add confusion (worst case). Chartjunk may include the use of too many colors, too many labels, gratuitously colored backgrounds, useless grid lines, borders of figures as well.

Rule #9: Message Trumps Beauty

Each scientific domain has developed its own set of best practices. It is important to know these standards, because they facilitate a more direct comparison between models, studies or experiments. More importantly, they can help you to spot obvious errors in your results.

Rule #10: Get The Right Tool

There exist many tools that can make your life easier when creating figures, and knowing a few of them can save you a lot of time. Depending on the type of visual you're trying to create, there is generally a dedicated tool that will do what you're trying to achieve. It is important to understand at this point that the software or library you're using to make a visualisation can be different from the software or library you're using to conduct your research and/or analyse your data.

Matplotlib is a python plotting library, primarily for 2-D plotting, but with some 3-D support, which produces publication-quality figures in a variety of hardcopy formats and interactive environments across platforms. It comes with a huge gallery of examples that cover virtually all scientific domains (http://matplotlib.org/gallery.html (http://matplotlib.org/gallery.html)).

R is a language and environment for statistical computing and graphics. R provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, etc.) and graphical techniques, and is highly extensible.

Inkscape is a professional vector graphics editor. It allows you to design complex figures and can be used, for example, to improve a script-generated figure or to read a PDF file in order to extract figures and transform them any way you like.

TikZ and **PGF** are TeX packages for creating graphics programmatically. TikZ is built on top of PGF and allows you to create sophisticated graphics in a rather intuitive and easy manner, as shown by the Tikz gallery (http://www.texample.net/tikz/examples/all/ (http://www.texample.net/tikz/examples/all/)).

GIMP is the GNU Image Manipulation Program. It is an application for such tasks as photo retouching, image composition, and image authoring. If you need to quickly retouch an image or add some legends or labels, GIMP is the perfect tool.

ImageMagick is a software suite to create, edit, compose, or convert bitmap images from the command line. It can be used to quickly convert an image into another format, and the huge script gallery (http://www.fmwconcepts.com/imagemagick/index.php (http://www.fmwconcepts.com/imagemagick/index.php)) by Fred Weinhaus will provide virtually any effect you might want to achieve.

D3.js (or just D3 for Data-Driven Documents) is a JavaScript library that offers an easy way to create and control interactive data-based graphical forms which run in web browsers, as shown in the gallery at http://github.com/mbostock/d3/wiki/Gallery (http://github.com/mbostock/d3/wiki/Gallery).

Cytoscape is a software platform for visualizing complex networks and integrating these with any type of attribute data. If your data or results are very complex, cytoscape may help you alleviate this complexity.

Circos was originally designed for visualizing genomic data but can create figures from data in any field. Circos is useful if you have data that describes relationships or multilayered annotations of one or more scales.

In [ ]:

```

```