tutorial012-Plotting-Matplotlib-Introduction

October 21, 2022

1 Basic plotting with matplotlib.pyplot

1.1 Learning goals

- to understand the anatomy of a matplotlib plot
- to learn how to make plot using
  - the pyplot interface
  - the "object oriented" method
- to use plotting themes
- to learn how to construct multi-panel figures

1.2 matplotlib

Matplotlib is the fundamental library for plotting in Python. It provides a comprehensive set of tools to create both static and dynamic (e.g., interactive) figures.

Matplotlib works really well with `numpy arrays`, the libary is very large and we will focus specifically on a submodule `matplotlib.pyplot` which provides an excellent starting point to learn about the platform.

This tutorial will focus less on data, programming and mathy stuff and more on understanding how to make, interpret and customize a plot using `pyplot`.

1.3 The anatomy of a plot

The basic anatomy of a plot reproduced from the `matplotlib` startup guide is shown below.

A figure in `matplotlib` consists of two key pieces:

- the ***figure***
- and one or more ***axes***

`Matplotlib` uses the object ***figure*** to handle plotting graphs. A ***figure*** is like a piece of paper or your canvas. Plots are made inside a figure and the objects used to refer to the plots is ***axes***. An ***axes*** is the area insde a ***figure*** where plots can be made. A Figure can contain one or more.

Confusingly, once we have made a plot in our ***axes***, our ***axes*** will now have at least one ***axis*** thingie on it – so, in the land of `matplotlib`, "axes" is **not** the plural of "axis"! Rather…

An **axes** is a part of a **figure** that contains one or more **axis** lines, as well as other things such as a title, x- and y-axis labels, and one or more plots of data.

Happily, we do all of our plotting and formatting by working with our ***axes***, we just have to make sure that when we say "axes", we mean "a plotting area on our figure" rather than "a bunch of 'axises' or 'axi' or whatever."

To avoid confusion, we'll refer to an **axes** as an **axes panel** from here on out.

1.4 Ways to make plots using matplotlib

There are two main ways to make plots using `matplotlib` (not counting the library `seaborn` - we'll revisit seaborn later on). The two main ways are:

- setting directly the properties of each object in a figure. This direct method is also called the "object oriented" interface
- using the `pyplot` interface

Below two examples of the same plot implemented using either style:

1.4.1 OO plotting style example

[1]:
```
import matplotlib as mpl        # the standard acronym to import matplotlib is
 ↪`mpl`
import matplotlib.pyplot as plt # the standard acronym to import pyplot is `plt`
import numpy as np
```

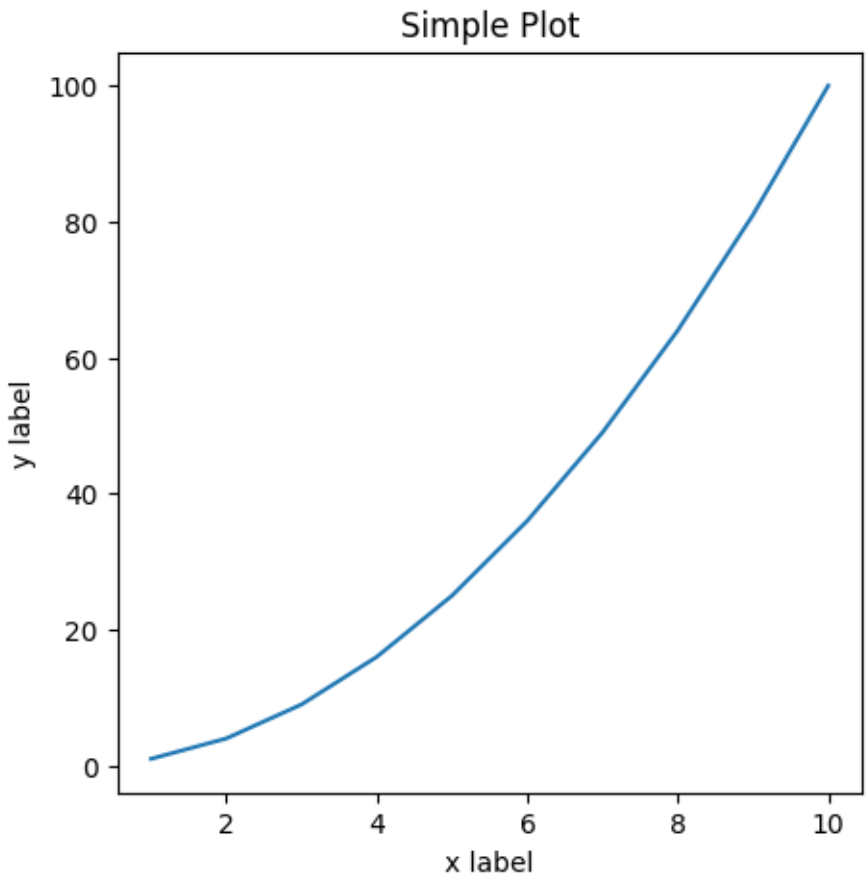
We plot a quadratic function as an example:

[4]:
```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9 ,10])
y = x**2
```

Next, w e create the figure (`fig`) and the axes (`ax`):

[5]:
```
fig, ax = plt.subplots(figsize=(5, 5)) ## Note here that for simplicity we
 ↪actually use pyplot to create the plot

# Plot the data on the axes
ax.plot(x, y)

# format the axes and add information to them
ax.set_xlabel('x label')  # Add an x-label to the axes.
ax.set_ylabel('y label')  # Add a y-label to the axes.
ax.set_title("Simple Plot")  # Add a title to the axes.
```

[5]:
```
Text(0.5, 1.0, 'Simple Plot')
```

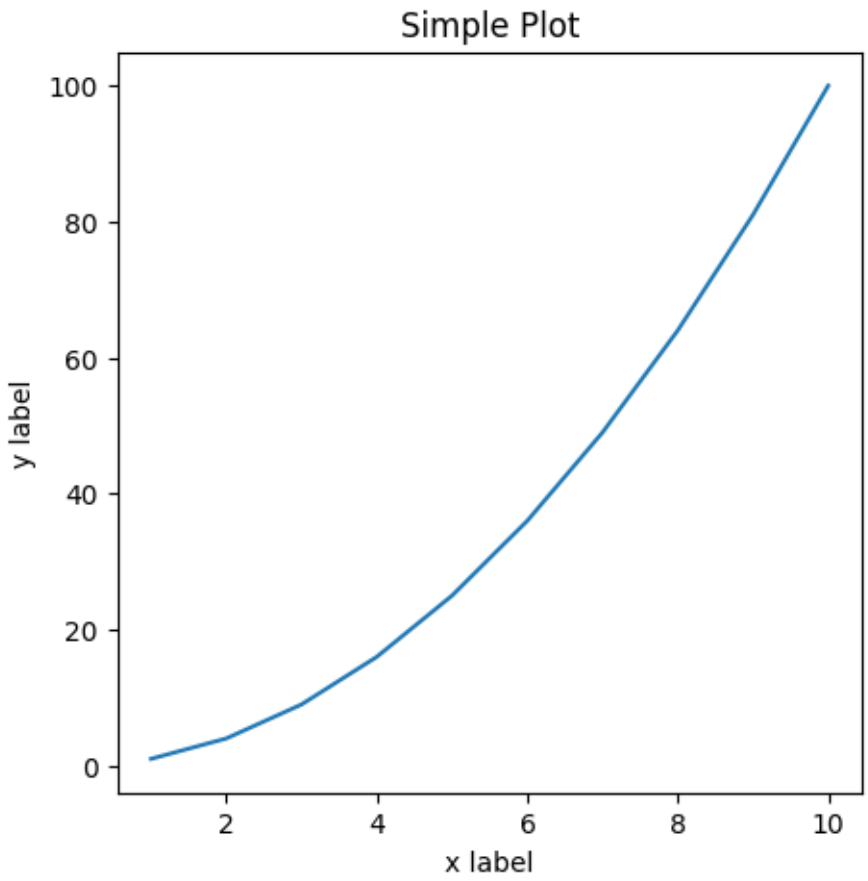


1.4.2 `pyplot` plotting style

Open the figure using pyplot:

[20]:
```
plt.figure(figsize=(5, 5))
plt.plot(x, y)  # Plot some data on the (implicit) axes.
# format the axis and add information to them
plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
```

[20]:
```
Text(0.5, 1.0, 'Simple Plot')
```



Note the difference with the previous call to `pyplot` where we returned both `fig` and `ax`, here instead we use the implicit method and change values to the parameters contained in the `pyplot` object `plt`, everything is set inside `plt`.

Complete the following exercise.

- What happens if you execute the command `plt.figure(figsize=(5, 5))` in the first `code` cell below and the rest of the commands above (from `plt.plot(x,y)` onwards) in the second `code` cell below?

[Use the `code` cell below to show the code]

[9]:
```
plt.figure(figsize=(5,5))
```

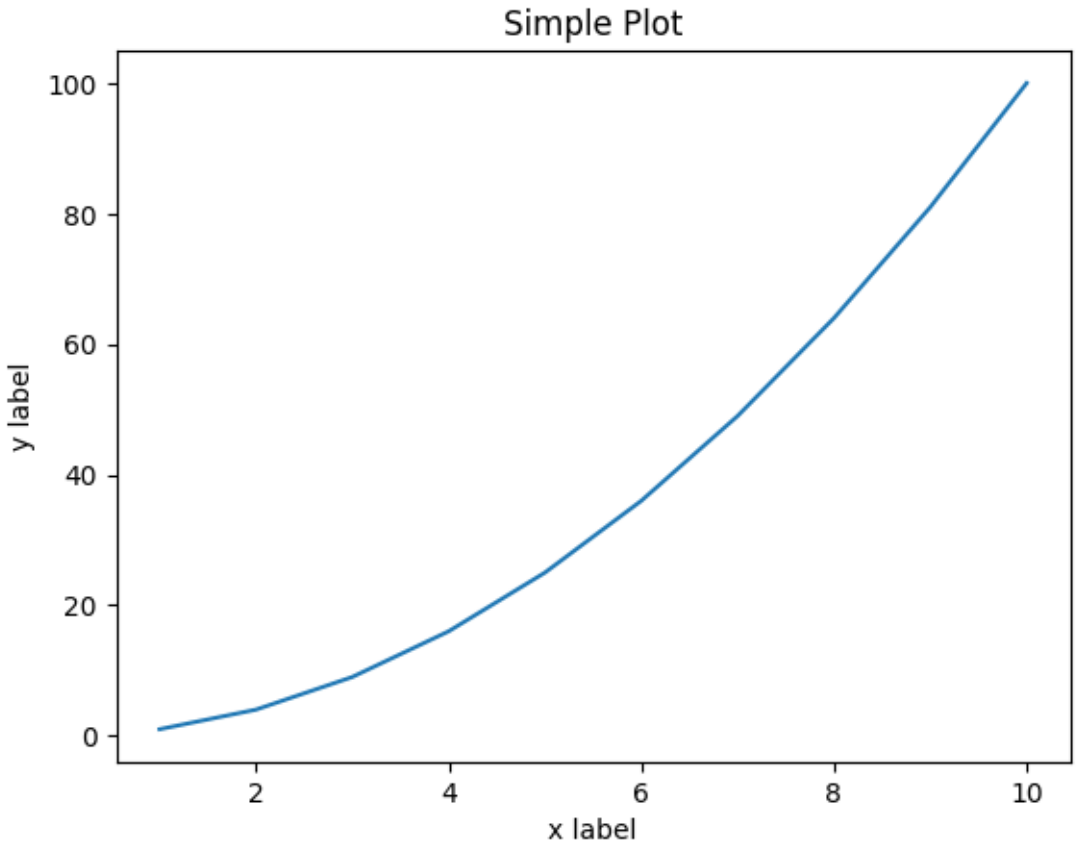
[9]:
```
<Figure size 500x500 with 0 Axes>
```

<Figure size 500x500 with 0 Axes>

[Use the `markdown` cell below to provide your interpretation of what might be happening here]

[10]:
```
plt.plot(x,y)
plt.xlabel('x label')
plt.ylabel('y label')
plt.title('Simple Plot')
```

[10]:
```
Text(0.5, 1.0, 'Simple Plot')
```



the dimensions are different.

We will focus first on the `pyplot` interface and go back to the OO interface later. This means that we will learn first how to

```
import matlibplot.pyplot as plt
```

and the use the `plt.plot()` command (which I like to pronounce "plit dot plot") to do our plotting. The `plt.plot()` automatically creates a **figure** and **axes panel** for us when we start plotting. For simple plots, we don't even have to know that there is an official thing called a "figure" and a another official thing called an "axes", we just start plotting!

In the object-oriented way, we create a **figure** and one or more **axes** (axes panels) explicitly, and then we use the methods (functions) associated with those objects to do our plotting and formatting. (This will make more sense when we do some examples.)

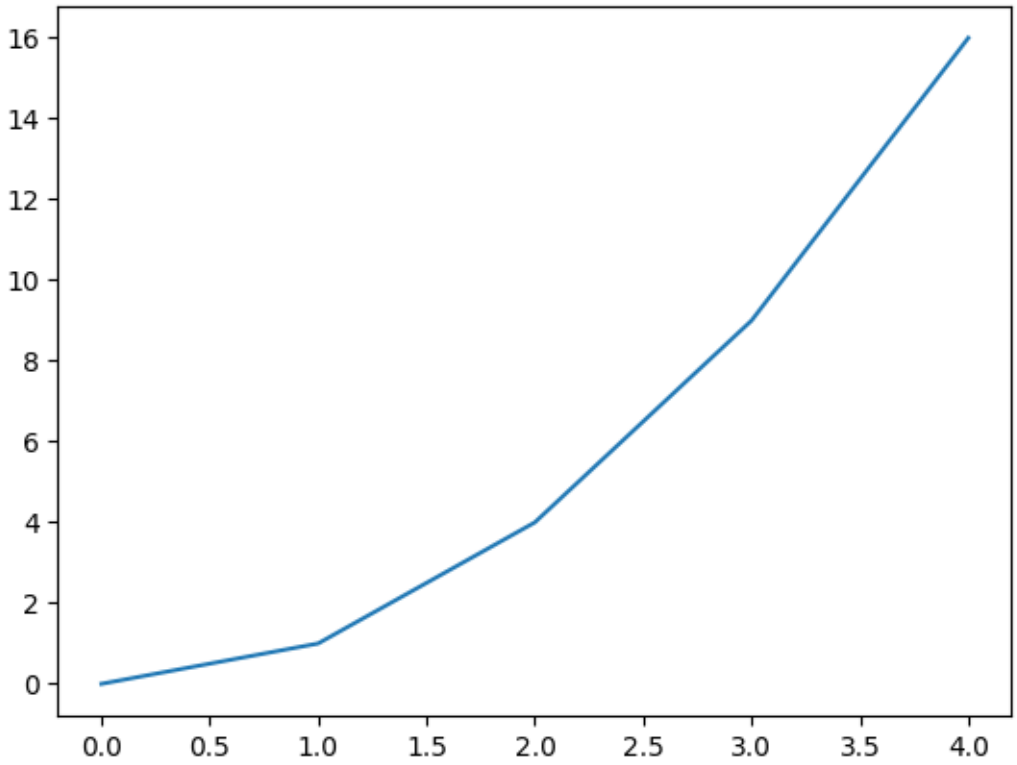
The fact that there are these two different ways to plot can make search results or StackExchange conversations confusing! So we'll go through these in turn, doing the same thing using two different methods.

1.5 Plotting Python `lists` using `matplotlib`'s `pyplot`

Let's breakdown the operations that `pyplot` performs for us, by making another plot! The steps are simple. We'll:

- import pyplot (we have done that above)
- make (simulate) some data to plot
- plot the data using plt.plot()
- breakdown the conceptual operations `plt` performs for us

[11]:
```
my_x_vals = [0, 1, 2, 3, 4]
my_y_vals = [0, 1, 4, 9, 16]

plt.plot(my_x_vals, my_y_vals);
```



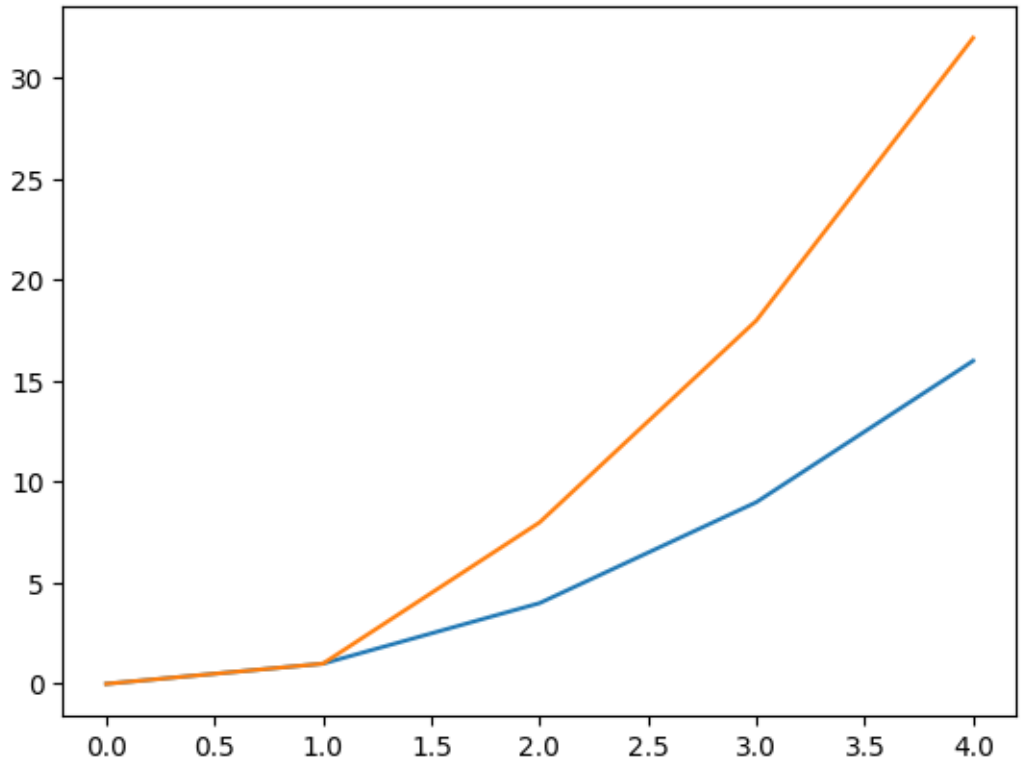
Yay! What `plt.plot()` has done for us here in a single command is

1. created a ***figure***
2. created a single ***axes panel*** on that figure
3. made both an x- and a y-axis (with tick marks) on the ***axes panel***

4. added x and y tick labels adjacent to each axis (on the ***axes panel***)
5. finally, plotted the data using the default line style and color (on the ***axes panel***)

Let's next plot a couple of functions on top of eachother:

[12]:
```
my_x_vals = [0, 1, 2, 3, 4]
my_y_vals = [0, 1, 4, 9, 16]
my_y_vals2 = [0, 1, 8, 18, 32]

plt.plot(my_x_vals, my_y_vals);
plt.plot(my_x_vals, my_y_vals2);
```



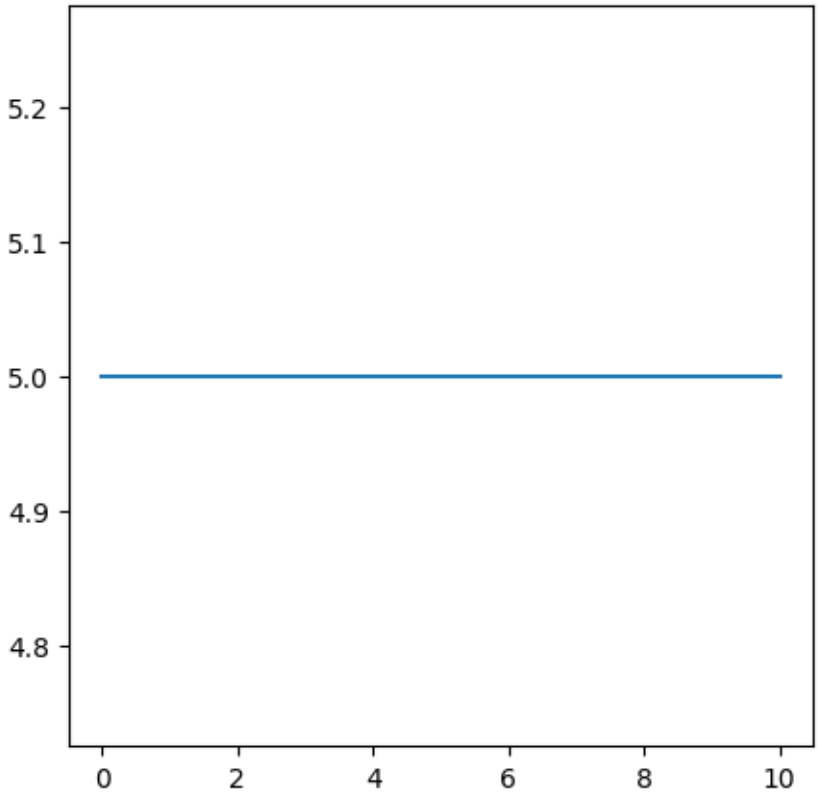
Complete the following exercise.

- given the data x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- create a new figure with y and x axis of equal dimensions (hint, we did this above)
- use `plt` to plot a horizontal line fixed at `5`

[Use the `code` cell below to implement the plot]

[27]:
```
plt.figure(figsize=(5, 5))
x_val = [0,1,2,3,4,5,6,7,8,9,10]
y_val = [5,5,5,5,5,5,5,5,5,5,5]
plt.plot(x_val,y_val)
```

[27]:
```
[<matplotlib.lines.Line2D at 0x1449536d0>]
```

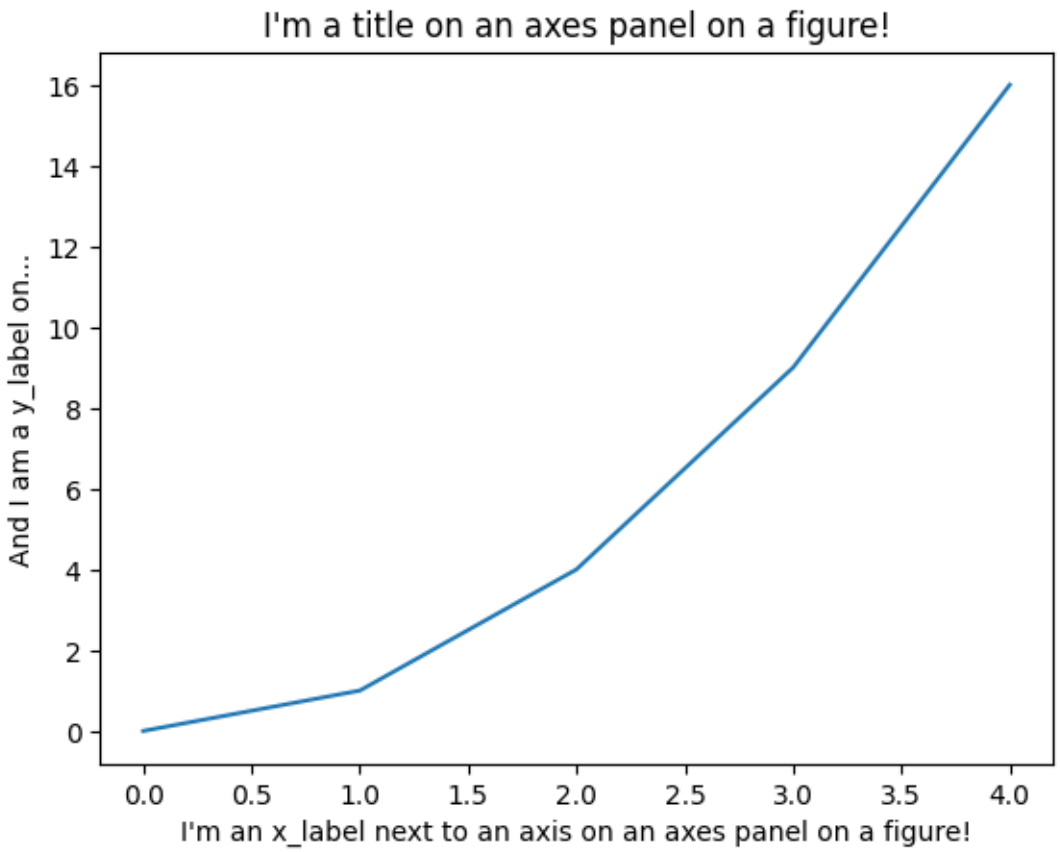


1.5.1 Adding axis labels and titles

Some common things we want to add are labels for the x- and y-axis, and a title for our plot. Handely, `pyplot` has functions for all of these. They are:

- `xlabel()`
- `ylabel()`
- `title()`

[28]:
```
plt.plot(my_x_vals, my_y_vals);
plt.xlabel('I\'m an x_label next to an axis on an axes panel on a figure!');
plt.ylabel('And I am a y_label on...');
plt.title('I\'m a title on an axes panel on a figure!');
```



Side note: Hey, what's up with the backslash inside the xlabel and title? In python, `strings` are defined by either single or double quotes. Like this:

[29]:
```
print('I am a string!')
```

```
I am a string!
```

[30]:
```
print("I am too!")
```

```
I am too!
```

But what happens when you want an apostophe (half a single quote) to be part of your string? This won't work:

[31]:
```
print('I'm a broken string')
```

```
  Cell In [31], line 1
    print('I'm a broken string')
                 ^
```

```
SyntaxError: unterminated string literal (detected at line 1)
```

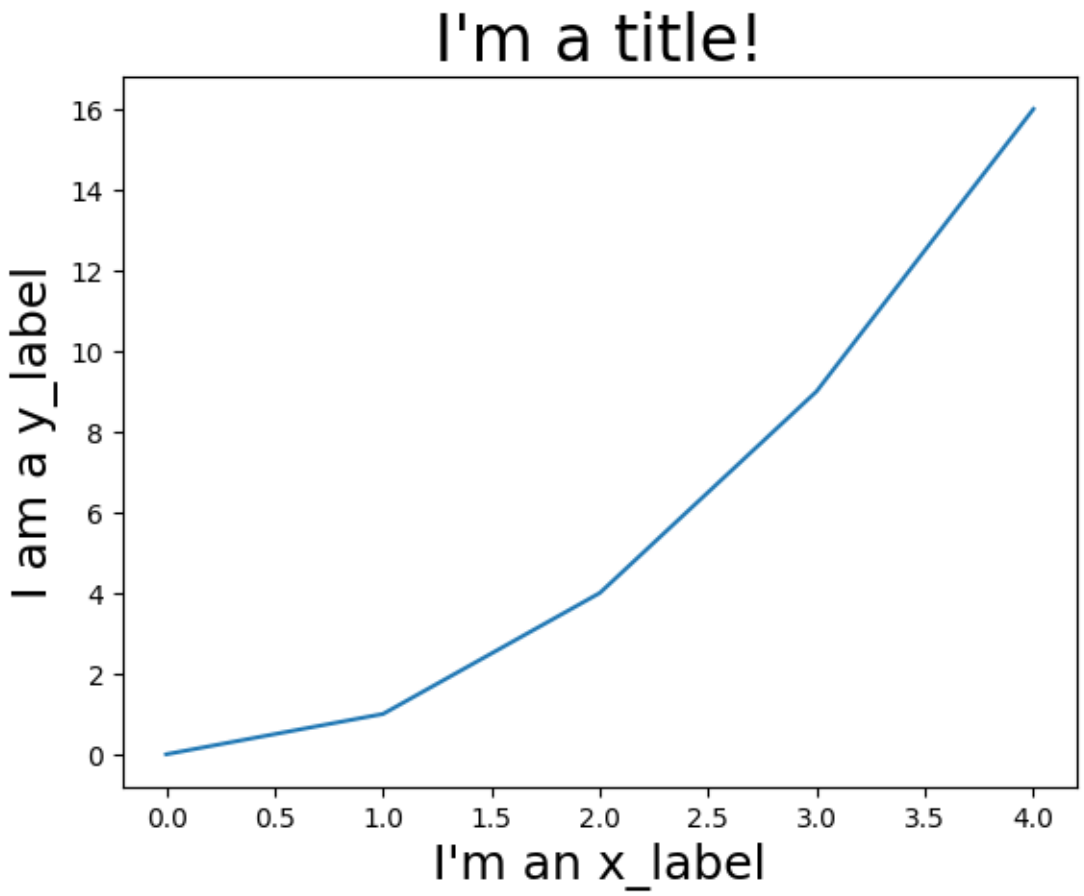
So Python (and all programming languages) provide an "escape character" that says "the thing right after me is part of the string!" So if we want to have our apostrophe be part of the sting, we "escape" it with the escape character "" (the backslash). So this works:

[32]:
```
print('I\'m a fixed string!')
```

```
I'm a fixed string!
```

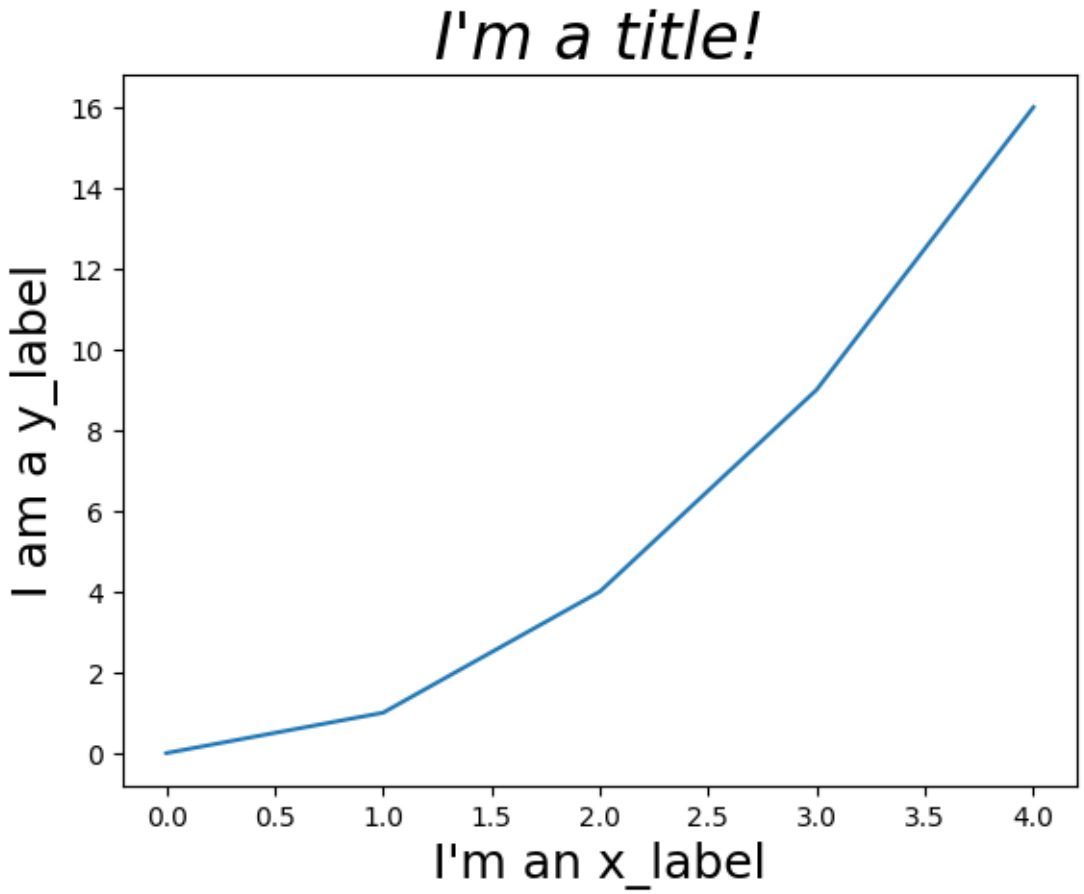
The label and title commands allow us to specify the font size as well (**Pro tip**: Unless you are plotting just for yourself, you almost always have to make the font sizes larger than the default!)

[33]:
```
plt.plot(my_x_vals, my_y_vals);
plt.xlabel('I\'m an x_label', fontsize = 18);
plt.ylabel('I am a y_label', fontsize = 18);
plt.title('I\'m a title!', fontsize = 24);
```



Each of these also allows you to do *italics* if you wish.

[34]:
```
plt.plot(my_x_vals, my_y_vals);
plt.xlabel('I\'m an x_label', fontsize = 18);
plt.ylabel('I am a y_label', fontsize = 18);
plt.title('I\'m a title!', fontsize = 24, fontstyle = 'italic');
```

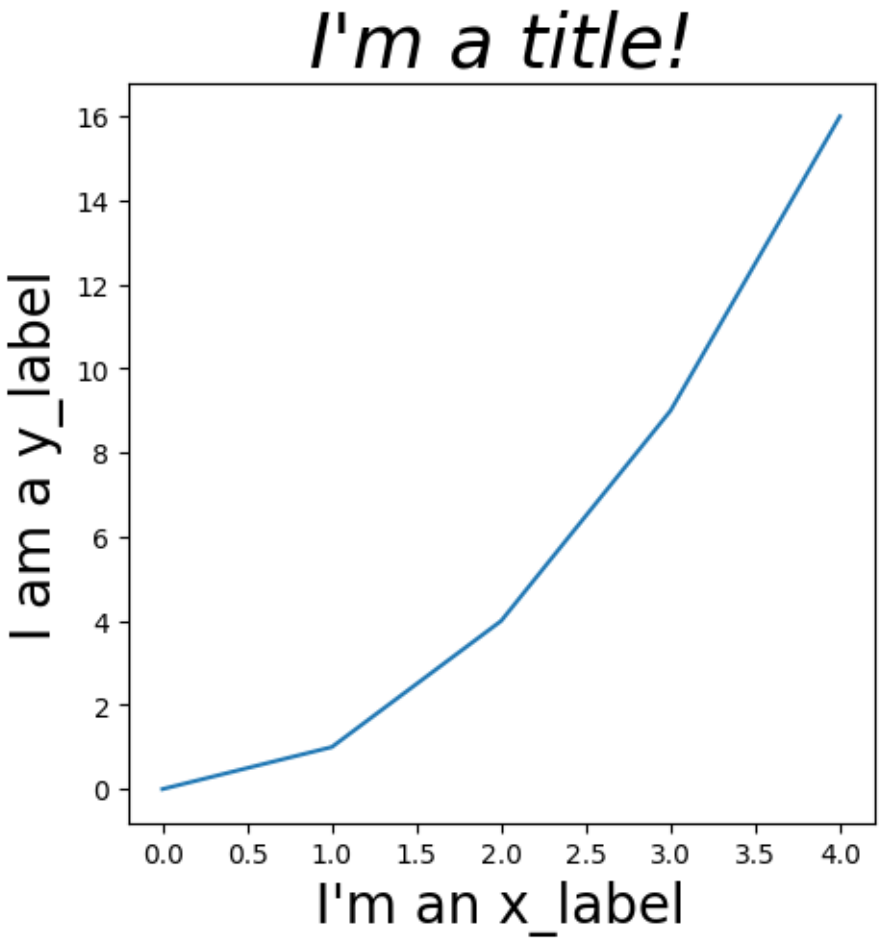


Complete the following exercise.

- repeate the plot in the previous cell but open a new figure of size x=5 and y=5 (hint we have done this above)
- change the font size of the title to 28 and of the axis to 20

[Use the `code` cell below to implement the plot]

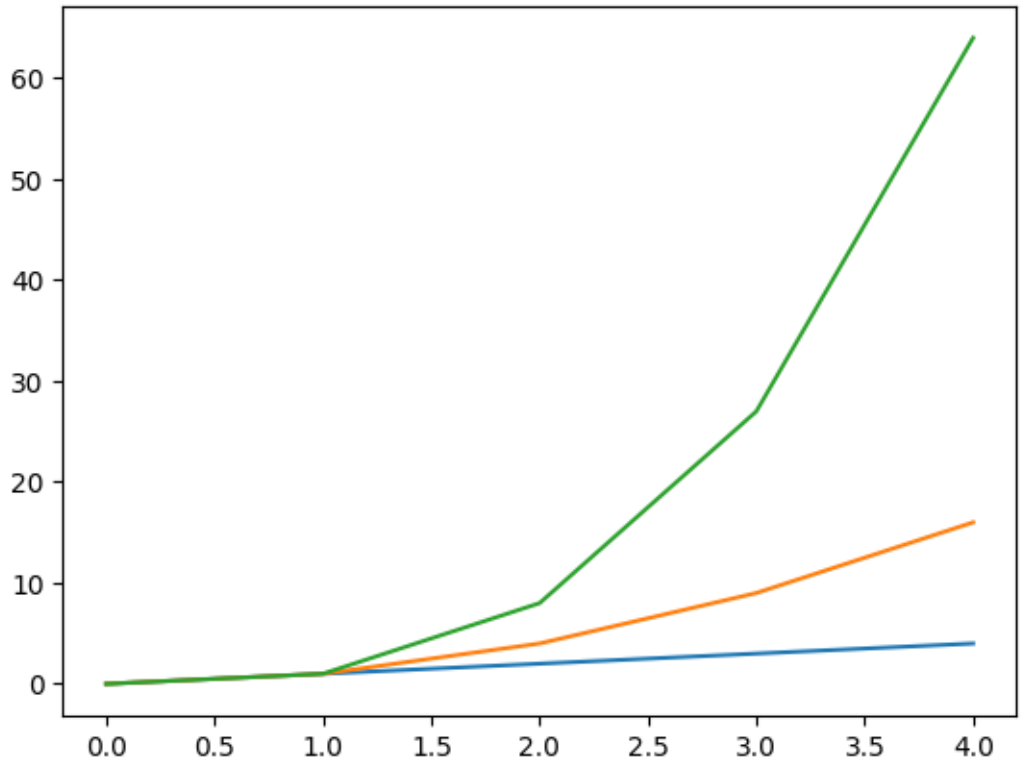
[35]:
```
plt.figure(figsize = (5,5))
plt.plot(my_x_vals, my_y_vals);
plt.xlabel('I\'m an x_label', fontsize = 20);
plt.ylabel('I am a y_label', fontsize = 20);
plt.title('I\'m a title!', fontsize = 28, fontstyle = 'italic');
```



1.5.2 Multiple plots on an axes panel

If we wish to plot multiple things, `plt.plot()` will change the colors for us automatically.

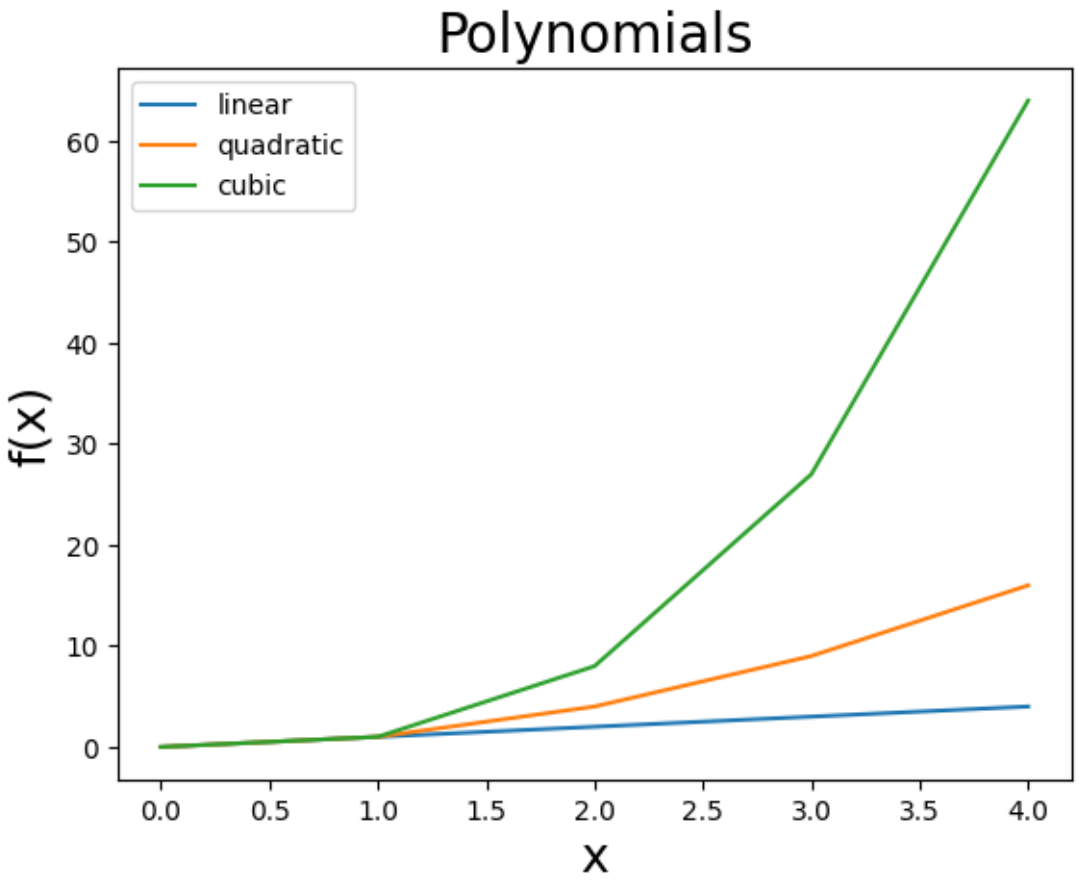
[36]:
```
my_x = np.array([0, 1, 2, 3, 4])

y_lin = my_x
y_quad = my_x**2
y_cube = my_x**3

plt.plot(my_x, y_lin);
plt.plot(my_x, y_quad);
plt.plot(my_x, y_cube);
```



1.5.3 Legends

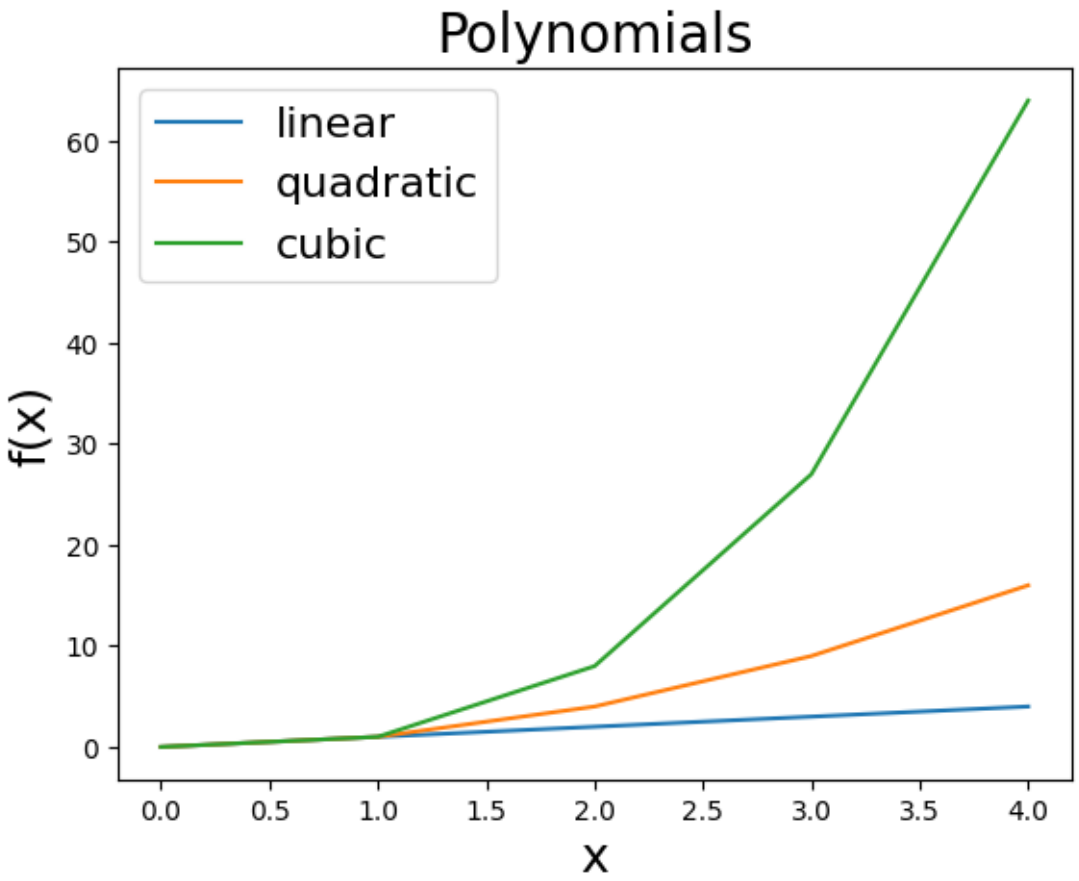
Crucially, we can give each plot a **label** so that we can add a **legend**.

[37]:
```
plt.plot(my_x, y_lin, label = 'linear');
plt.plot(my_x, y_quad, label = 'quadratic');
plt.plot(my_x, y_cube, label = 'cubic');

plt.xlabel('x', fontsize = 18);
plt.ylabel('f(x)', fontsize = 18);
plt.title('Polynomials', fontsize = 20,);

plt.legend();
```



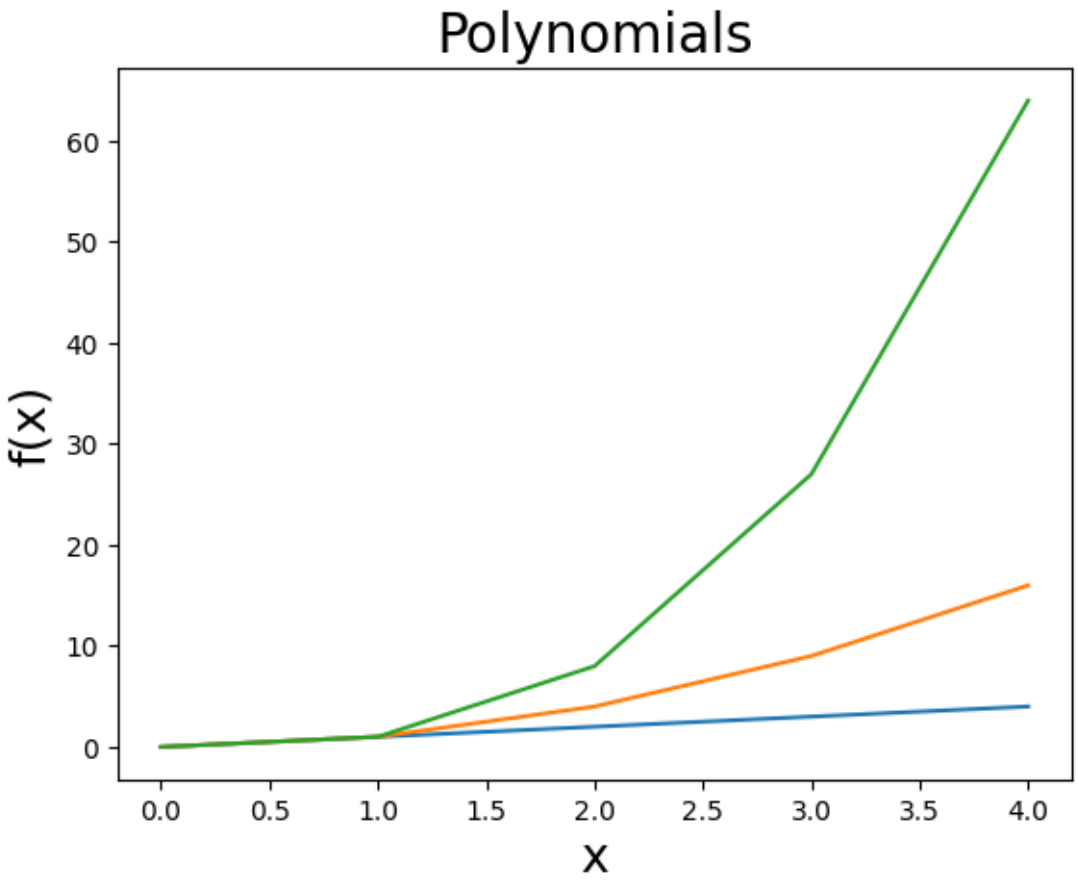
You can change the font size for the legend just like you do the title, etc. You can also specify the location of the legend with the `loc=` parameter. The default is 'best'.

[38]:
```
plt.plot(my_x, y_lin, label = 'linear');
plt.plot(my_x, y_quad, label = 'quadratic');
plt.plot(my_x, y_cube, label = 'cubic');

plt.xlabel('x', fontsize = 18);
plt.ylabel('f(x)', fontsize = 18);
plt.title('Polynomials', fontsize = 20,);

plt.legend(loc = 'upper left', fontsize = 16);
```



Let's take minute to try and guess what some different allowed locations are!

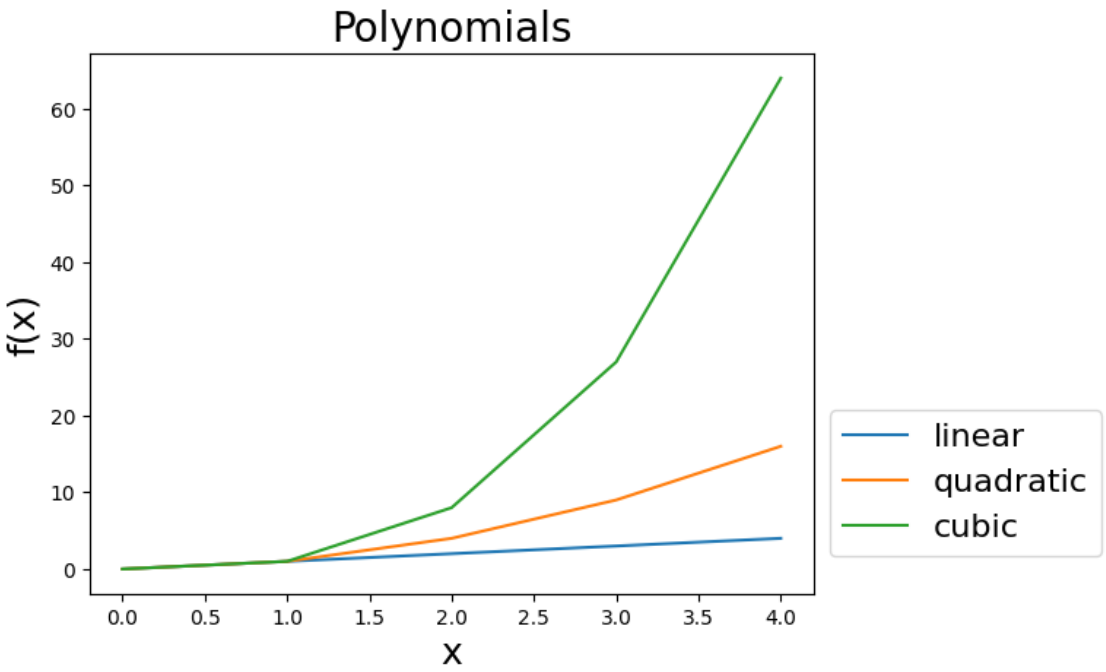
[39]:
```
plt.plot(my_x, y_lin, label = 'linear');
plt.plot(my_x, y_quad, label = 'quadratic');
plt.plot(my_x, y_cube, label = 'cubic');

plt.xlabel('x', fontsize = 18);
plt.ylabel('f(x)', fontsize = 18);
plt.title('Polynomials', fontsize = 20,);
```



If we want even finer control over legend position (if, for example, we wanted to move the legend outside the axis limits), we can specify the position "manually". To do this, we will need to set the property `bbox_to_anchor` instead of `loc`.

Let's try it:

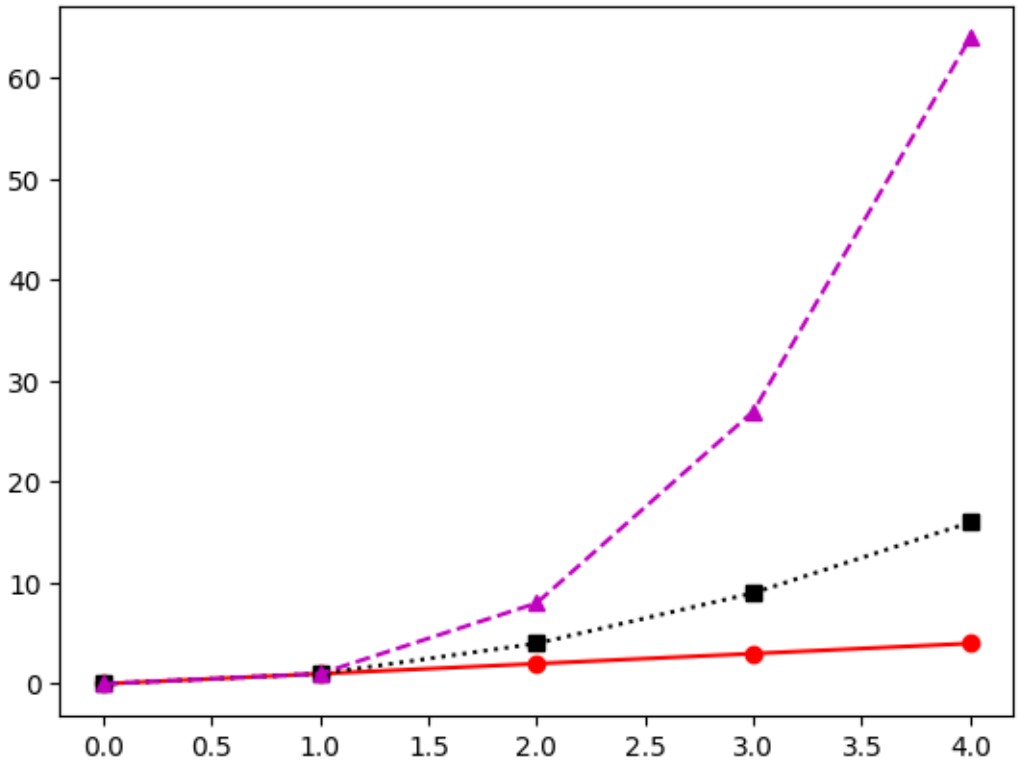
[40]:
```
plt.plot(my_x, y_lin, label = 'linear');
plt.plot(my_x, y_quad, label = 'quadratic');
plt.plot(my_x, y_cube, label = 'cubic');

plt.xlabel('x', fontsize = 18);
plt.ylabel('f(x)', fontsize = 18);
plt.title('Polynomials', fontsize = 20,);

plt.legend(bbox_to_anchor =(1.0, 0.37), fontsize = 16);
```



1.5.4 Adjusting colors, markers, and line styles

We can also specify "by hand" what the *colors*, *lines*, and data point *markers* look like by adding a `string` containing a code for each. The `string` has to come right after the data. Like this:

[41]:
```
plt.plot(my_x, y_lin, '-ro', label = 'linear'); # note the cryptic string in␣
↪the middle!
plt.plot(my_x, y_quad, ':ks', label = 'quadratic');
plt.plot(my_x, y_cube, '--m^', label = 'cubic');
```



There are just 4 linestyles, the three above and '-.' For reference, they are:

code	line style
-	solid
–	dashed
:	dotted
-.	dot-dash

The common colors that are available are:

code	line style
r	red
g	green
b	blue
k	black
y	yellow
m	majenta
c	cyan

There may be others…

Some common markers are:

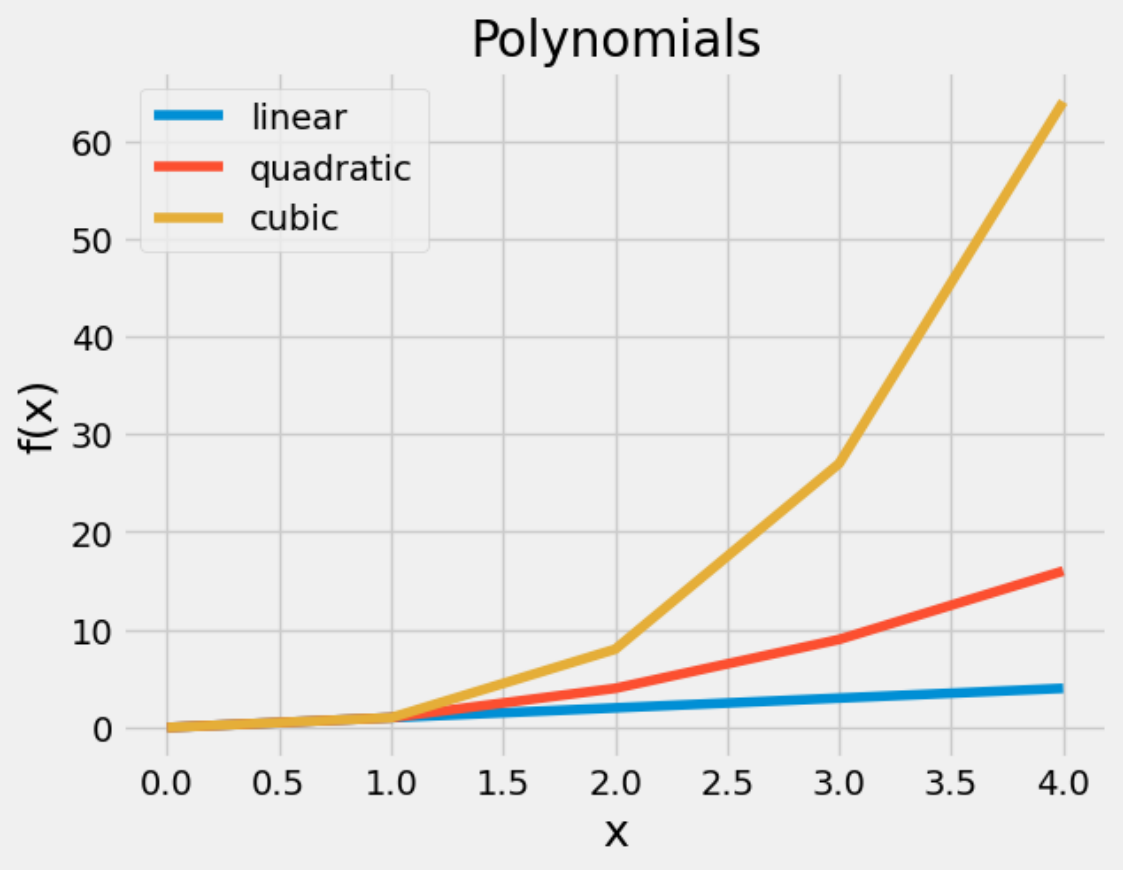
code	line style
o	circle
.	point
s	square
d	diamond
^, >, <, v	triangles

There are a few others, which you can find here.

1.5.5 Plot styles

`pyplot` offers several built-in styles that allow you to change the overall appearance – the look and feel – of your plots. We set these with the `plt.style.use()` function:

[42]:
```
plt.style.use('fivethirtyeight') # set style to Nate Silver's

plt.plot(my_x, y_lin, label = 'linear');
plt.plot(my_x, y_quad, label = 'quadratic');
plt.plot(my_x, y_cube, label = 'cubic');

plt.xlabel('x', fontsize = 18);
plt.ylabel('f(x)', fontsize = 18);
plt.title('Polynomials', fontsize = 20,);

plt.legend();
```



Nice!! You can see what styles are available like this:

[43]:
```
plt.style.available
```

[43]:
```
['Solarize_Light2',
 '_classic_test_patch',
 '_mpl-gallery',
 '_mpl-gallery-nogrid',
 'bmh',
 'classic',
 'dark_background',
 'fast',
 'fivethirtyeight',
 'ggplot',
 'grayscale',
 'seaborn-v0_8',
 'seaborn-v0_8-bright',
 'seaborn-v0_8-colorblind',
 'seaborn-v0_8-dark',
 'seaborn-v0_8-dark-palette',
```

```
 'seaborn-v0_8-darkgrid',
 'seaborn-v0_8-deep',
 'seaborn-v0_8-muted',
 'seaborn-v0_8-notebook',
 'seaborn-v0_8-paper',
 'seaborn-v0_8-pastel',
 'seaborn-v0_8-poster',
 'seaborn-v0_8-talk',
 'seaborn-v0_8-ticks',
 'seaborn-v0_8-white',
 'seaborn-v0_8-whitegrid',
 'tableau-colorblind10']
```

And you can see what each one looks like here.

You can always go back to the default style by specifying `default`, like this:

[44]:
```
plt.style.use('default') # back to default, please
```

1.5.6 Figures with multiple axis panels

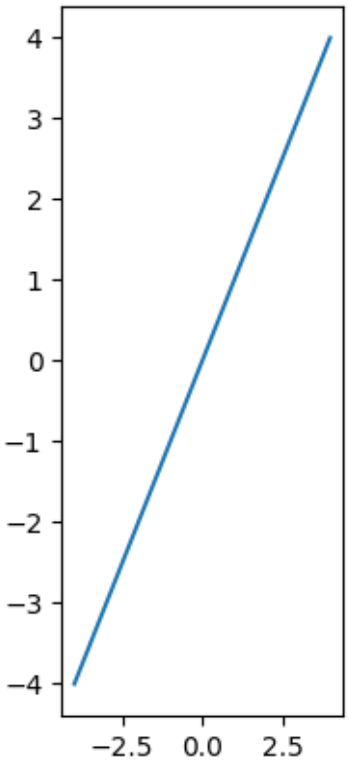
If you open any paper, book, or website, you will see figures with multiple panels showing different but related things. In `pyplot`, this is accomplished with `plt.subplot()`.

(Let's remake the polynomial so they look more interesting.)

[45]:
```
my_x = np.array([-4, -3, -2, -1, 0, 1, 2, 3, 4])

y_lin = my_x
y_quad = my_x**2  # ** is Python for exponentiation
y_cube = my_x**3
```

Now we'll plot our polynomials in separate panels. We'll start with just one.

[46]:
```
plt.subplot(131)
plt.plot(my_x, y_lin, label = 'linear');
```

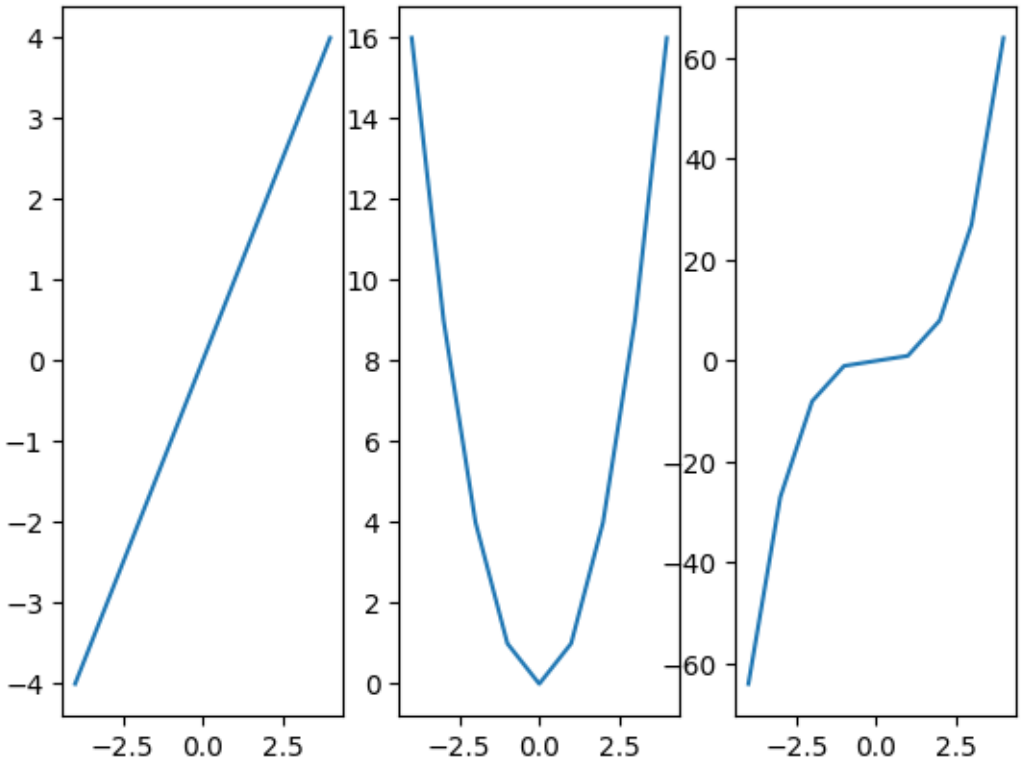


I know it looks funny, but hang on. What happened is that `plt.subplot(131)` made a figure for us. Then, the `131` told it to divide the figure into 1 row by 3 columns of axes panels, and make the 1st axes panel the "active" one.

So the three numbers are ("how many rows" "how many columns" "which one to plot in").

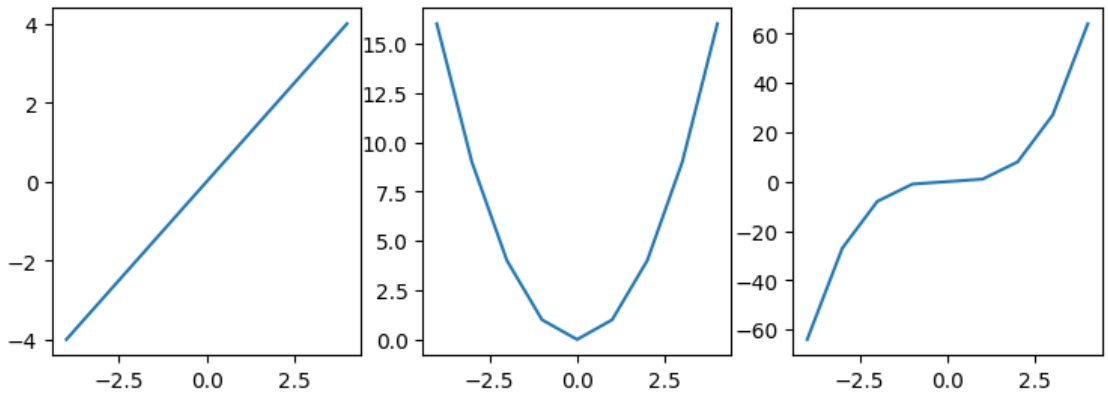
So now let's plot all three of our polynomials.

[47]:
```
plt.style.use('default')

plt.subplot(131)    # make the first (left hand) axes panel the active one
plt.plot(my_x, y_lin, label = 'linear');
#plt.axis('square')

plt.subplot(132)    # make the second panel active
plt.plot(my_x, y_quad, label = 'quadratic');
#plt.axis('square')

plt.subplot(133)   # make the third panel active
plt.plot(my_x, y_cube, label = 'cubic');
#plt.axis('square')
```



It still looks funny – the panels are tall and skinny and figure only takes up part of the page horizontally. This is because some default figure size is being used. But fear not! We can explicitly make a the figure and set its size (in inches) ourselves with the `plt.figure()` command. This way we can make our figure have a rational size and aspect ratio for whatever we're trying to plot. So this time, let's make a figure that's 9"x3" for our plotting.

[48]:
```
plt.style.use('default')

plt.figure(figsize=(9, 3))  # make a wide, short figure

plt.subplot(131)
plt.plot(my_x, y_lin, label = 'linear');

plt.subplot(132)
plt.plot(my_x, y_quad, label = 'quadratic');

plt.subplot(133)
plt.plot(my_x, y_cube, label = 'cubic');
```

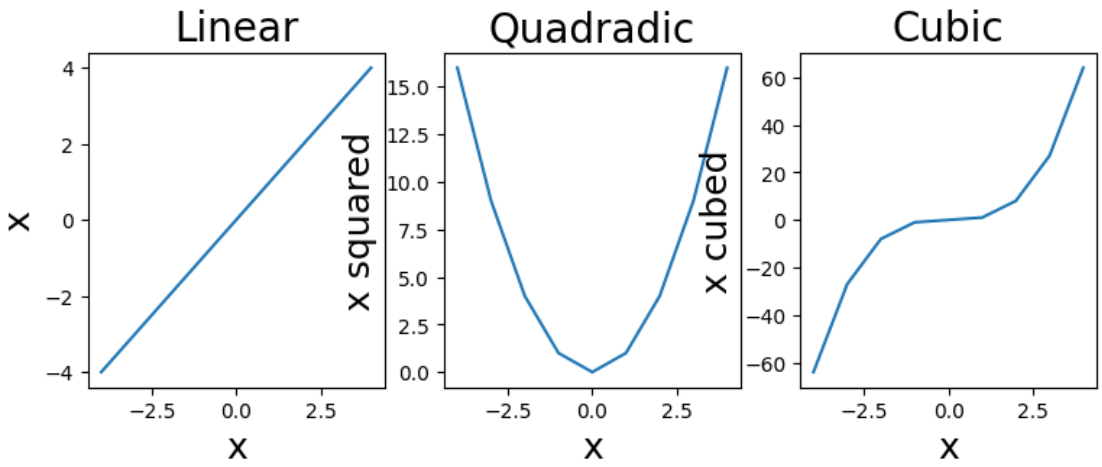


Ah, much better!

One thing that might be a little annoying is that everything else in Python has the row (height) come first, but here we specify the width first in the `figsize` argument. The size argument is this way because that's how it works in the printing world (e.g. 8 1/2" by 11" paper). Oh well, we'll deal.

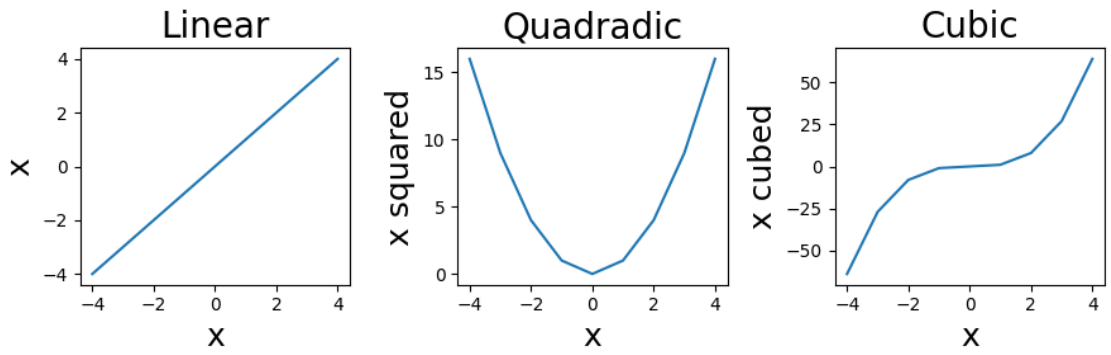
While each plot is active, we can take the opportunity to add axis labels and titles:

[49]:
```
plt.style.use('default')

plt.figure(figsize=(9, 3))

plt.subplot(131)                          # make the first plot active
plt.plot(my_x, y_lin, label = 'linear');  # do the actual plotting
plt.xlabel('x', fontsize = 18);           # and while the plot is active
plt.ylabel('x', fontsize = 18);           # make it fancy like
plt.title('Linear', fontsize = 20,);

plt.subplot(132)
plt.plot(my_x, y_quad, label = 'quadratic');
plt.xlabel('x', fontsize = 18);
plt.ylabel('x squared', fontsize = 18);
plt.title('Quadradic', fontsize = 20,);

plt.subplot(133)
plt.plot(my_x, y_cube, label = 'cubic');
plt.xlabel('x', fontsize = 18);
plt.ylabel('x cubed', fontsize = 18);
plt.title('Cubic', fontsize = 20,);
```



Okay, all good, right? Oh… wait… our y axis labels are weird, like there's not enough room in the figure to spread the axes panels out. We could play around with making the figure wider, but there is also a function, `plt.tight_layout()` which attempts to arrange everything for us with no weird clipping or overlapping. So let's try the exact same code with that at the end:

[50]:
```
plt.style.use('default')

plt.figure(figsize=(9, 3))

plt.subplot(131)                            # make the first plot active
plt.plot(my_x, y_lin, label = 'linear');    # do the actual plotting
plt.xlabel('x', fontsize = 18);             # and while the plot is active
plt.ylabel('x', fontsize = 18);
plt.title('Linear', fontsize = 20,);

plt.subplot(132)
plt.plot(my_x, y_quad, label = 'quadratic');
plt.xlabel('x', fontsize = 18);
plt.ylabel('x squared', fontsize = 18);
plt.title('Quadradic', fontsize = 20,);

plt.subplot(133)
plt.plot(my_x, y_cube, label = 'cubic');
plt.xlabel('x', fontsize = 18);
plt.ylabel('x cubed', fontsize = 18);
plt.title('Cubic', fontsize = 20,);

plt.tight_layout()
```

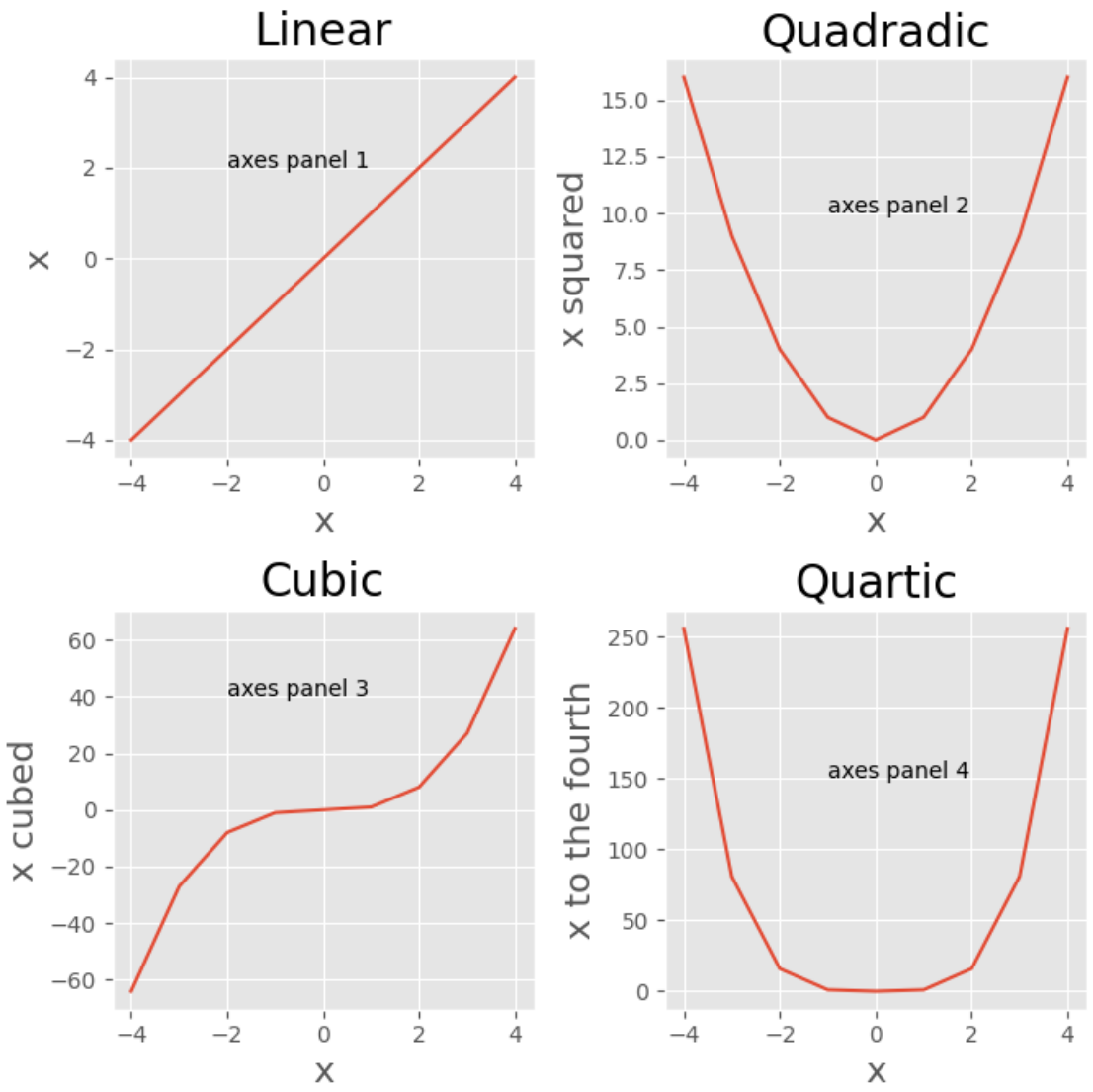


Sweet!

Finally, the subplots – the axes panels – are numbered "row wise", i.e. left-to-right, top to bottom. So if we wanted to write a self-contained piece of code plot four polynomials in a 2x2 grid, it would look like this:

[51]:
```
# compute our polynomials
my_x = np.array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
y_lin = my_x
y_quad = my_x**2  # ** is Python for exponentiation
y_cube = my_x**3
y_quart = my_x**4

# set our style
plt.style.use('ggplot')
axisLabelSize = 16
titleSize = 20

# make a big square figure
plt.figure(figsize=(7, 7))

# Now make each of the plots!

# first is top left
plt.subplot(221)
plt.plot(my_x, y_lin, label = 'linear');
plt.xlabel('x', fontsize = axisLabelSize);
plt.ylabel('x', fontsize = axisLabelSize);
plt.title('Linear', fontsize = titleSize);
plt.text(-2, 2, 'axes panel 1')

# then top right
plt.subplot(222)
plt.plot(my_x, y_quad, label = 'quadratic');
plt.xlabel('x', fontsize = axisLabelSize);
```

```
plt.ylabel('x squared', fontsize = axisLabelSize);
plt.title('Quadradic', fontsize = titleSize,);
plt.text(-1, 10, 'axes panel 2')

# third is bottom left
plt.subplot(223)
plt.plot(my_x, y_cube, label = 'cubic');
plt.xlabel('x', fontsize = axisLabelSize);
plt.ylabel('x cubed', fontsize = axisLabelSize);
plt.title('Cubic', fontsize = titleSize,);
plt.text(-2, 40, 'axes panel 3')

# then bottom right
plt.subplot(224)
plt.plot(my_x, y_quart, label = 'quartic');
plt.xlabel('x', fontsize = axisLabelSize);
plt.ylabel('x to the fourth', fontsize = axisLabelSize);
plt.title('Quartic', fontsize = titleSize,);
plt.text(-1, 150, 'axes panel 4')

# finally, make everything automagically fit
plt.tight_layout()
```



Note that we snuck in an additional useful goodie, which is the `plt.text()` function. It allows you to add text (in axis-axis coordinates) to your figure!

So that does it for our overview of plotting the `pyplot` way! It may seem like a lot right now, but it's really not. Here's a handy summary.

1.5.7 Summary of pyplot plotting functions

Cast, in order of appearance (all are prefixed with `plt.` assuming you have imported `matplotlib.pyplot as plt`)

function	use
plot()	make a plot
xlabel()	add x label (can set font size)

function	use
ylabel()	add y label (can set font size)
title()	add a title (can set font size)
legend()	add a legend (can set position)
style.use()	use a plot style
style.available	list available styles
subplot()	make a plot matrix and set active plot
figure()	make a figure (can specify size)
tight_layout()	try to make everything fit neatly
text()	add text in x,y plotting coordinates

Now we shall briefly look at plotting in "object oriented" style.

Complete the following exercise.

```
x = np.random.randn(100,1)
y_A = x**2
y_B = y_A**2
y_C = y_B**2
```

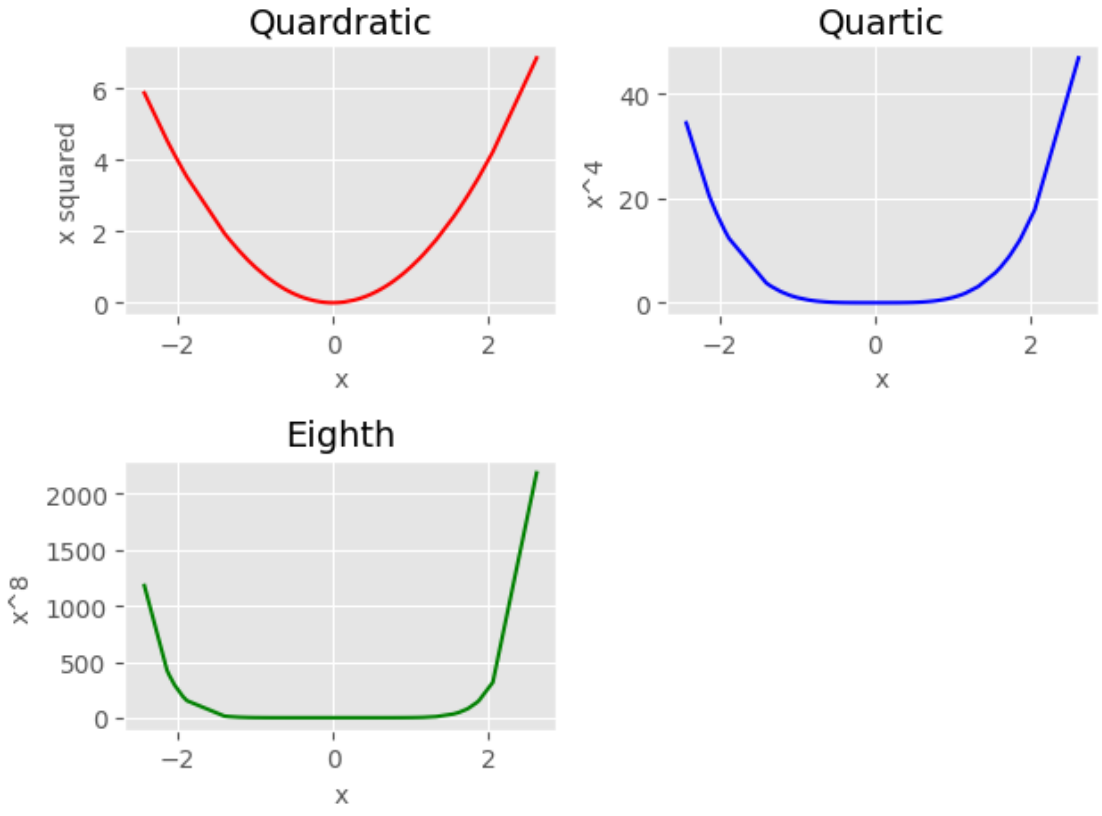
- given the data above:
  - Make 3 plots, one per y values
  - use subplot to plot the 3 plots in 2x2 grid of plots
  - set the style to one of the seaborn styles
  - set the color of each line to a different color say `red`, `blue` and `green`
  - set axis labels, and plot titles

Note that most of what asked to do here is in the previous cells. The only more tricky thing will be to figure out how to find out which styles are `available`, remeber sometimes you can use `plt.<TAB>`, sometimes `plt.style.<TAB>` and see which options are `available`

[Use the `code` cell below to implement the plot]

[61]:
```
# data
x = np.random.randn(100,1)
x = np.sort(x, axis = 0) #sort array for line graph
y_A = x**2
y_B = y_A**2
y_C = y_B**2

#style
plt.style.use('seaborn-v0_8-deep')
axislabelsize = 10
titlesize = 14

#subplot 1
plt.subplot(221)
plt.plot(x, y_A, 'r', label = 'squared');
plt.xlabel('x', fontsize = axislabelsize);
```

```
plt.ylabel('x squared', fontsize = axislabelsize);
plt.title('Quardratic', fontsize = titlesize)

#subplot 2
plt.subplot(222)
plt.plot(x, y_B, 'b', label = 'quartic');
plt.xlabel('x', fontsize = axislabelsize);
plt.ylabel('x^4', fontsize = axislabelsize);
plt.title('Quartic', fontsize = titlesize)
#subplot 3
plt.subplot(223)
plt.plot(x, y_C, 'g', label = 'eighth');
plt.xlabel('x', fontsize = axislabelsize);
plt.ylabel('x^8', fontsize = axislabelsize);
plt.title('Eighth', fontsize = titlesize)

# fit
plt.tight_layout()
```



1.6 Plotting using object oriented (OO) methods

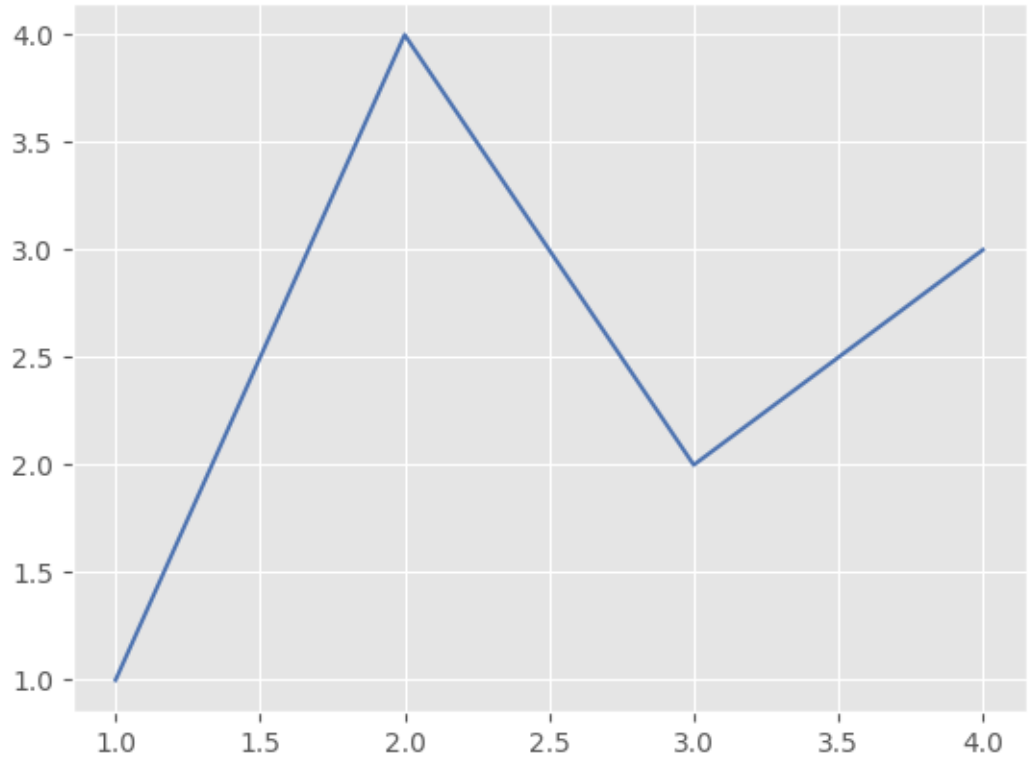
Now, let's go back to our OO plotting method. The difference between plotting using `pyplot` and using the object oriented approach is that, with the latter, we use the ability of two kinds of objects, **figures** and **axes**, to do stuff. The basic steps are:

- Make a figure and one or more axes with plt.subplots() (note it's plural!!)
- use the ability of an axes to do stuff like plotting
- (sometimes) use the figure's ability to do stuff

1.6.1 A simple object oriented plot

As usual, a simple example will make this more clear. Here's one lifted straight from the matplotlib documentation:

[62]:
```
fig, ax = plt.subplots()  # Create a figure containing a single axes.
ax.plot([1, 2, 3, 4], [1, 4, 2, 3]);  # Plot some data on the axes.
```



The call to `plt.subplots()` made a figure with a single axes panel in it. It returned a "handle" to the figure named `fig`, and a handle to the axes panel named `ax`.

We then used the axes panel's name, `ax`, to make the axes panel plot data in itself.

Note: we could have named these anything we wanted – Velma & Daphnie, Shaggy & Scooby, Bugs & Daffy, whatever – but Python / Data Science convention is to name the figure "fig" and the axes

"ax". Booorrriiing, but handy for writing code that others, including future you, can easily read.

1.6.2 Adding stuff

[63]:
```
x = np.linspace(0, 2, 100)  # Make new x values - again!
```

Create the figure and the axes panel. Note that even in the OO-style, we use `.pyplot.figure` to create these!

[64]:
```
# make the figure, fig, and the axes panel, ax
fig, ax = plt.subplots()

# now use ax's skills to plot stuff
ax.plot(x, x, label='linear')         # Plot a line
ax.plot(x, x**2, label='quadratic')   # and a parabola
ax.plot(x, x**3, label='cubic')       # and a cubic curvy thingie

# and now use ax's skills add stuff
ax.set_xlabel('x values')             # Add an x-label
ax.set_ylabel('f(x)')                 # and a y-label
ax.set_title("Plot made OO style!")   # and a title
ax.legend();                          # and, finally, a legend
```

Notice that, *in terms of process*, the only really differences are:

- we call `subplots()` (instead of `subplot()`), which makes
  - a figure named "fig"
  - an axes panel named "ax"
- we plot with `ax.plot()` instead of `plt.plot()`
- we use `ax.set_...` to do xlabel, ylabel, and title, and
- `ax.legend()` to make the legend

Happily, all of the `ax.` functions work just like their `plt.` counterparts.

1.6.3 OO Figures with multiple axis panels

So if the OO method of making figures is pretty much the same as the pyplot method of making figures (except for some annoying differences in terminology), what's the point in having both? Why would you use one vs. the other? The short answer is that:

- pyplot is easier – you can make simple figures with less code
- the OO method is more powerful and flexible for working with complicated figures

Let's return to our four panel figure example above for a quick illustration of this. We'll break up the code so it's a bit more digestable.

Compute the polynomials.

[65]:
```
# compute our polynomials
my_x = np.array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
y_lin = my_x
y_quad = my_x**2
y_cube = my_x**3
y_quart = my_x**4
```

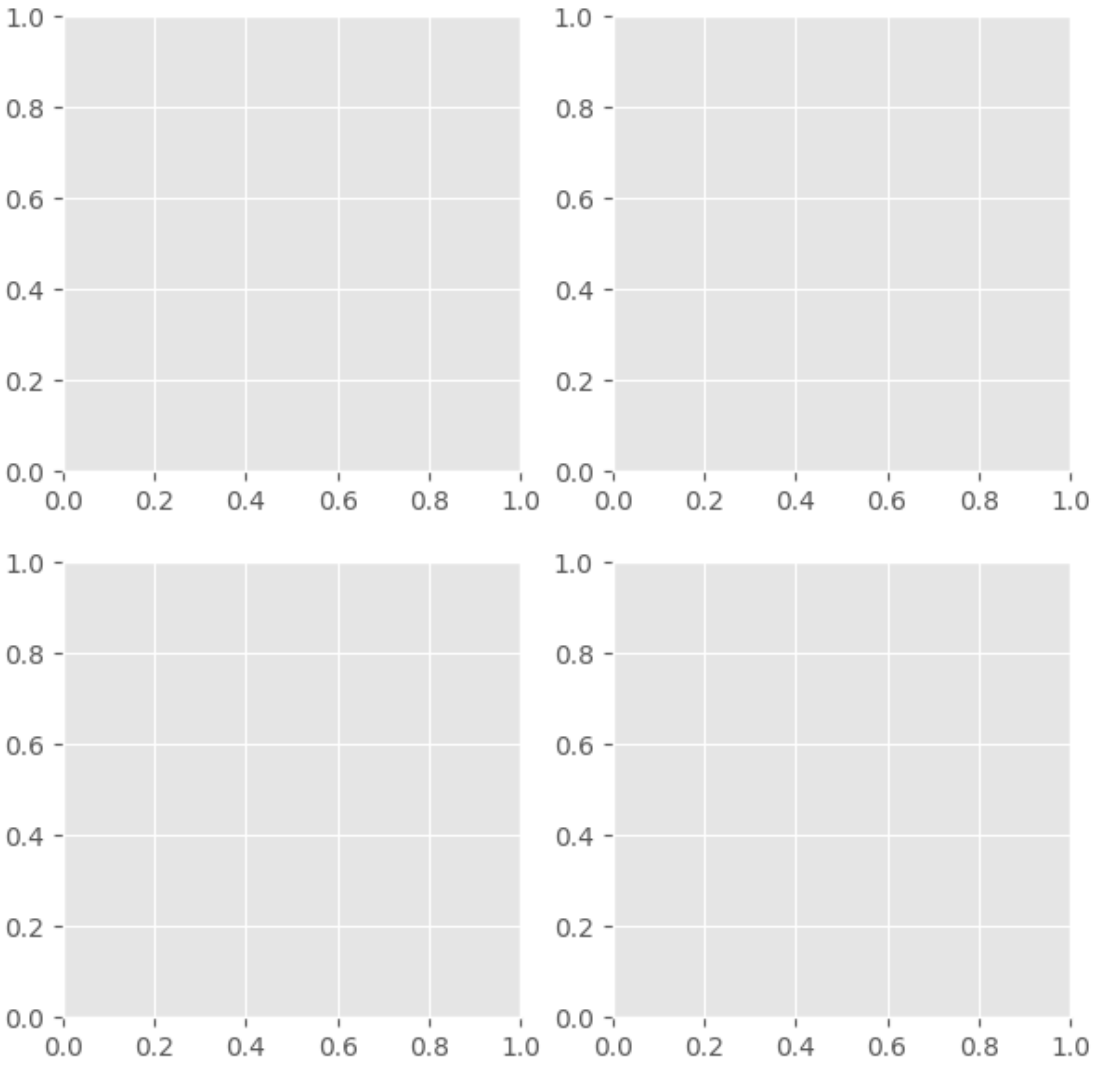
Next, we'll make sure we're stylin'

[66]:
```
# set our style
plt.style.use('ggplot')
axisLabelSize = 16
titleSize = 20
```

Now, here is where the difference begins! We will call `plt.subplots()` and specify the how many rows and columns of axes panels we want. We will still name our figure "fig".

But instead of getting back a single handle to an axes panel that we would name "ax", we are going to get back an array of handles that, by convention, we'll name "axs".

[70]:
```
# make a big square figure
fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(7, 7))
```



So what is "axs"? It is an array, like a numpy array, but contains the "handles" to each of our plots. Like this:

[[upper left plot, upper right plot],

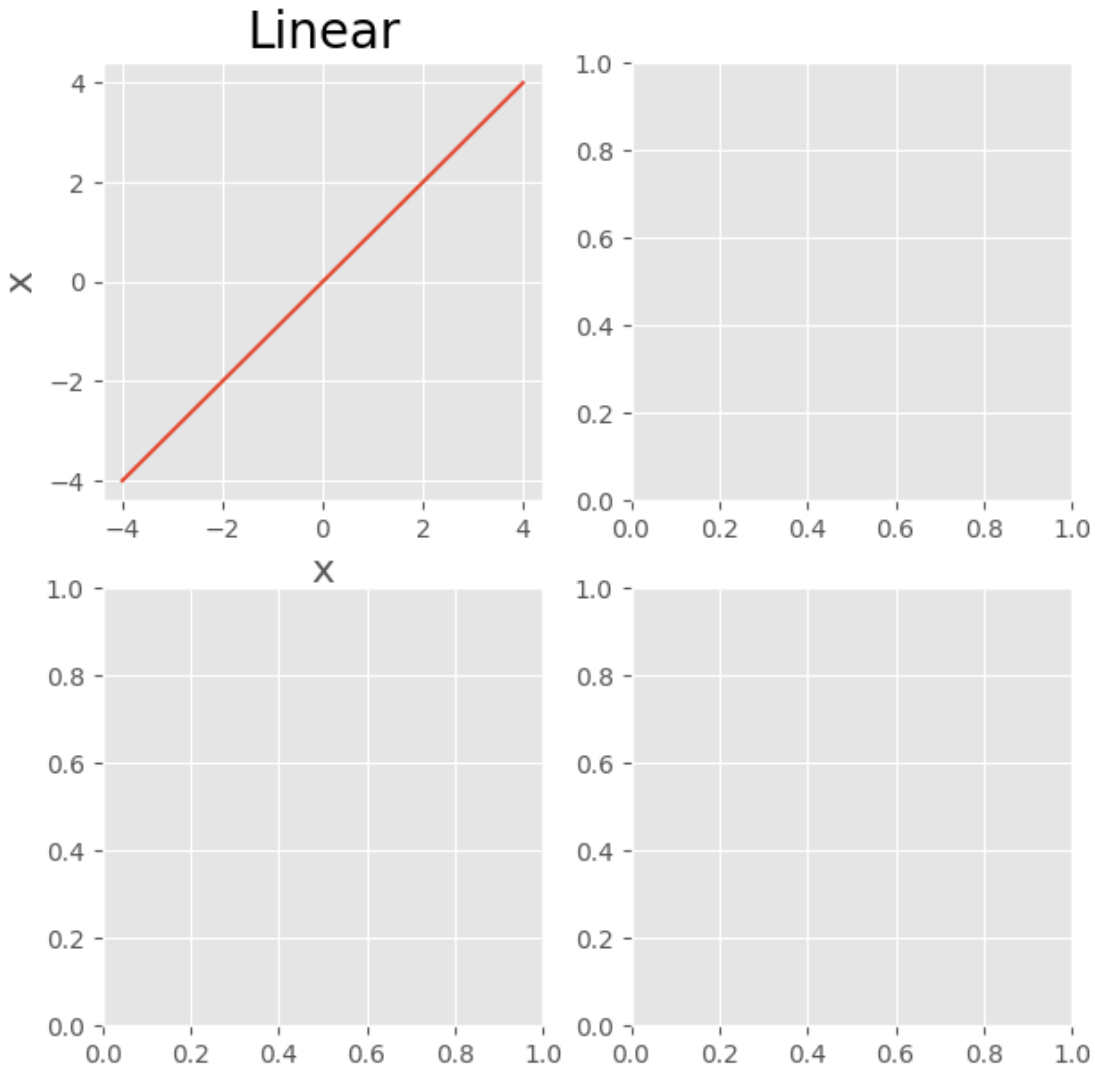
[lower left plot, lower right plot]]

So instead of saying `ax.plot()` like we did above, we would say `axs[0,0].plot()` to plot in the upper left axes panel, `axs[0,1].plot()` to plot in the upper right axes panel, etc.

So let's make the first subplot! (We have to re-make the figure or there will be nowhere for us to plot.)

[71]:
```
fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(7, 7))

axs[0,0].plot(my_x, y_lin, label = 'linear');
```

```
axs[0,0].set_xlabel('x', fontsize = axisLabelSize);
axs[0,0].set_ylabel('x', fontsize = axisLabelSize);
axs[0,0].set_title('Linear', fontsize = titleSize);
```

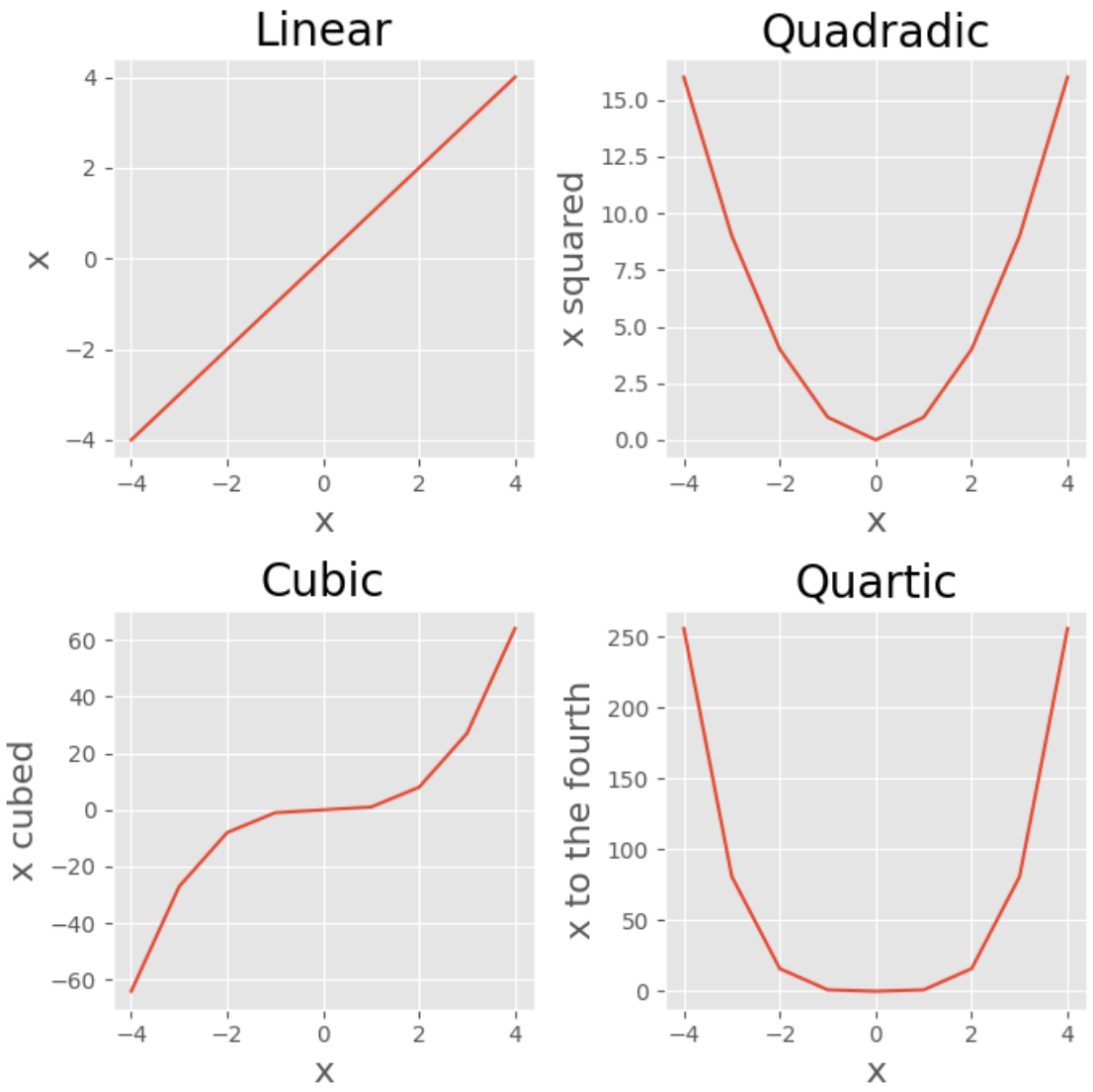


If you think about it, this actually more clear and straightforward than `plt.subplot(221)`, etc. `axs[0,0]` makes it immediately clear that your are plotting in the [first row, first column] of your figure.

Now let's make the full figure:

[68]:
```
# make the figure
fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(7, 7))

# Now make each of the plots!
```

```
# first is top left
# plt.subplot(221) we don't neet this anymore
axs[0,0].plot(my_x, y_lin, label = 'linear');
axs[0,0].set_xlabel('x', fontsize = axisLabelSize);
axs[0,0].set_ylabel('x', fontsize = axisLabelSize);
axs[0,0].set_title('Linear', fontsize = titleSize);

# then top right
axs[0,1].plot(my_x, y_quad, label = 'quadratic');
axs[0,1].set_xlabel('x', fontsize = axisLabelSize);
axs[0,1].set_ylabel('x squared', fontsize = axisLabelSize);
axs[0,1].set_title('Quadradic', fontsize = titleSize,);

# third is bottom left
axs[1,0].plot(my_x, y_cube, label = 'cubic');
axs[1,0].set_xlabel('x', fontsize = axisLabelSize);
axs[1,0].set_ylabel('x cubed', fontsize = axisLabelSize);
axs[1,0].set_title('Cubic', fontsize = titleSize,);

# then bottom right
axs[1,1].plot(my_x, y_quart, label = 'quartic');
axs[1,1].set_xlabel('x', fontsize = axisLabelSize);
axs[1,1].set_ylabel('x to the fourth', fontsize = axisLabelSize);
axs[1,1].set_title('Quartic', fontsize = titleSize,);

# finally, make everything automagically fit
fig.tight_layout()  # notice that now it's fig. instead of plt.
```



1.6.4 Figure-level titles and labels

And now we can use a little more of the power of the OO method to make the figure a bit more appealing. Notice that

- the x axis labels are redundant, one overall label would be better
- a global y axis label, like "f(x)" would be cool
- the figure could use an overall title, like "Polynomials!" or something.

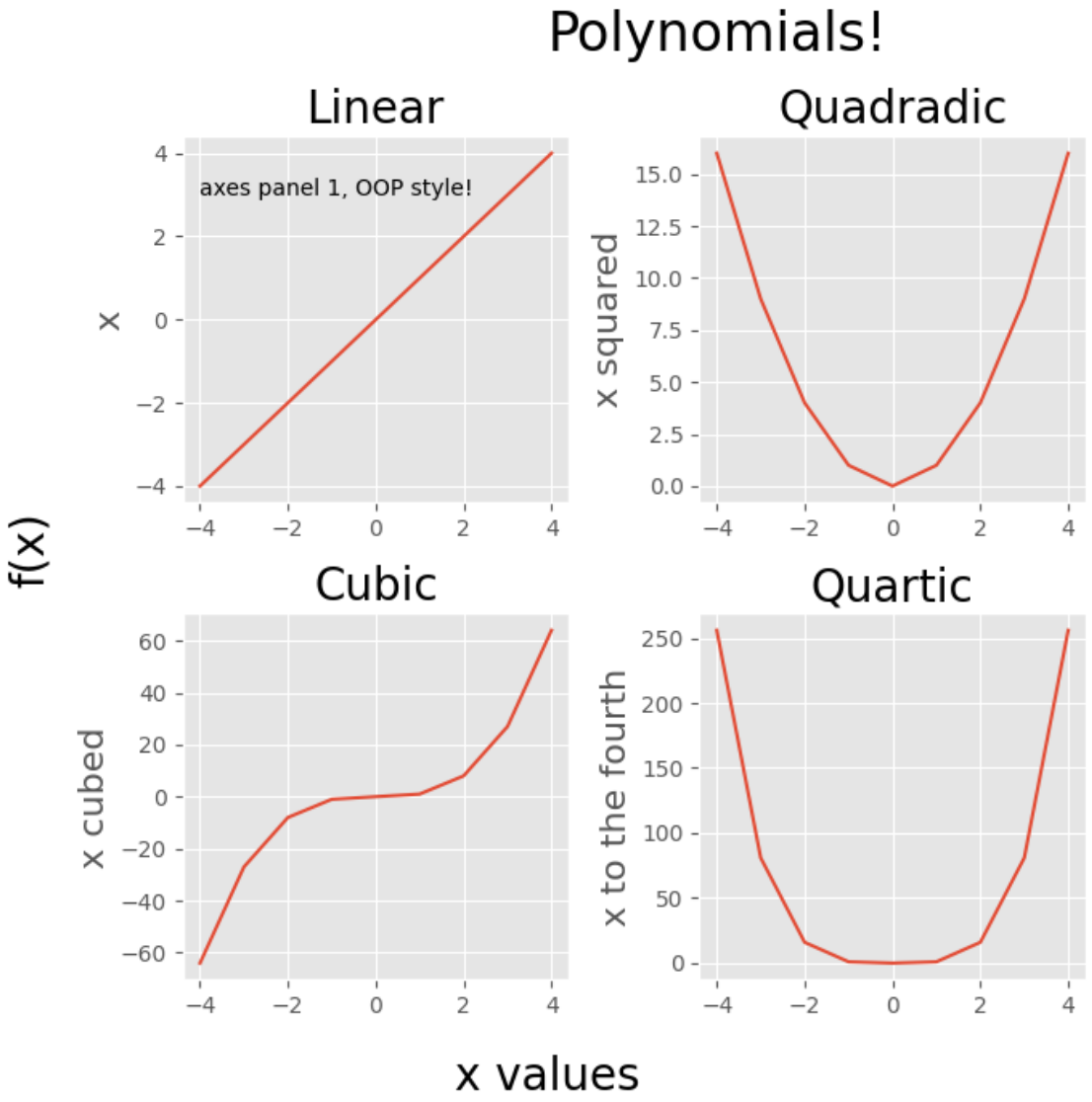
We can do all these things because our figure named "fig" knows how to do stuff to! We'll use

- fig.suptitle() to make a "supra title"
- fig.supylabel() to make a "supra y label", and
- fig.supxlabel() to make a "supra x label", and

We'll do this all in one cell of code, but the only thing that's different from above is the little chunk at line 20 where we do the figure level stuff – the "supra stuff"!

[69]:
```
# compute our polynomials
my_x = np.array([-4, -3, -2, -1, 0, 1, 2, 3, 4])
y_lin = my_x
y_quad = my_x**2  # ** is Python for exponentiation
y_cube = my_x**3
y_quart = my_x**4

# set our style
plt.style.use('ggplot')
axisLabelSize = 16
titleSize = 20

# make a big square figure
fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(7, 7))

# set the "supra stuff" for the figure
fig.suptitle('Polynomials!', ha='left', fontsize=24)
fig.supylabel('f(x)', fontsize=20)
fig.supxlabel('x values', fontsize=20)

# Now make each of the plots!

# first is top left
# plt.subplot(221) we don't neet this anymore
axs[0,0].plot(my_x, y_lin, label = 'linear');
axs[0,0].set_ylabel('x', fontsize = axisLabelSize);
axs[0,0].set_title('Linear', fontsize = titleSize);
axs[0,0].text(-4, 3, 'axes panel 1, OOP style!')

# then top right
axs[0,1].plot(my_x, y_quad, label = 'quadratic');
axs[0,1].set_ylabel('x squared', fontsize = axisLabelSize);
axs[0,1].set_title('Quadradic', fontsize = titleSize,);

# third is bottom left
axs[1,0].plot(my_x, y_cube, label = 'cubic');
axs[1,0].set_ylabel('x cubed', fontsize = axisLabelSize);
axs[1,0].set_title('Cubic', fontsize = titleSize,);

# then bottom right
axs[1,1].plot(my_x, y_quart, label = 'quartic');
axs[1,1].set_ylabel('x to the fourth', fontsize = axisLabelSize);
axs[1,1].set_title('Quartic', fontsize = titleSize,);
```

```
# finally, make everything automagically fit
fig.tight_layout()
```



And there we go! We can do all the same stuff using the OO approach to making plots, but allows us to

- address the axes panels in a more transparent way – axs[row, col] vs. subplot(rcn)
- put global labels and titles on our figures

1.7 Which method do I choose?

It's largely up to you. A good rule of thumb is

- use pyplot for simple, one panel figures

- use the OO method for multi-panel figures

That having been said, some people feel that "If I *have* to use the OO method for some figures, why not just use it for *every* figure?"

It's your call!

Complete the following exercise.

- Describe in your own words the two methods by which matplot lib works

[Use the `markdown` cell below to answer]

There are two methods for plotting. Object oriented `OO` which sets each of the properties directly to each object in a figure. It allows us to create a figure and the axes panels explicitly. On the other hand, the `plt` method allows us to open figures using `pyplot` and implicitly set axes.

- Describe the difference between the `OO` and `plt` approach to using `matplotlib`

[Use the `markdown` cell below to answer]

The `OO` aproach explicitly creates figures and axes and therefore allows for multiplotting to be addresses more transparently (`axs[row, col]`) as well as letting us set global labels. The `plt` approach is much more implicit, with everything being set inside the `plt` object. For multipanelled plots we would need to use a more convoluted way (`subplot(rcn)`). The `plt` approach allows for simple plots to be made with less code but becomes complicated for multi-panel figures.

- Make a new plot using your method of choice. The plot should plot at least 30 data points in `x` transofrmed using 3 of the math operators introduced in the previous tutorials. Label the axis, the figure, the colors of the lines, add titles to the figure and a legend to the plotted data. Also customize the style of the plot and font size of all text.

[Use the `markdown` cell below to implement your code]

[37]:
```
# data
dat = 30
x = np.sort(np.random.randn(dat))
x_a = x**3
x_b = x/3
x_c = x + 3

# set style
plt.style.use('tableau-colorblind10')

# plot graph
plt.plot(x, x_a, 'g', label = 'cubed');
plt.plot(x, x_b, 'om', label = 'multipled');
plt.plot(x, x_c, 'y', label = "added six");

# labels
plt.xlabel('x', fontsize = 16);
plt.ylabel('f(x)', fontsize = 16);
plt.title('woop woop', fontsize = 18);
```

```
plt.legend();
```