Matplotlib

August 25, 2024

1 Introduction to Matplotlib

Matplotlib is a popular Python library used for creating static, interactive, and animated visualizations in Python. It is widely used in data science, engineering, and scientific research for plotting data in a variety of chart types, including line plots, scatter plots, bar charts, histograms, and more.

Here are some key features of Matplotlib:

1. **Versatile Plotting**: Matplotlib supports a wide range of plots and charts, from basic line and bar charts to more complex visualizations like 3D plots and polar charts.
2. **Customization**: It allows extensive customization of plots, including colors, labels, markers, and legends. You can also customize the layout and appearance of plots to suit specific needs.
3. **Integration**: Matplotlib integrates well with other Python libraries like NumPy, Pandas, and SciPy, making it a powerful tool for visualizing data from these sources.
4. **Interactive Plots**: While primarily known for static plots, Matplotlib can also create interactive plots when used in conjunction with other libraries like `mpld3` or `plotly`.
5. **Publication-Quality**: It produces high-quality plots suitable for publication in academic journals, with support for various output formats, including PNG, PDF, and SVG.
6. **Matplotlib's Pyplot**: A submodule called `pyplot` is often used for creating simple plots quickly, offering a MATLAB-like interface.

Matplotlib allows you to create reproducible figures programmatically. Let's learn how to use it! Before continuing this lecture, I encourage you just to explore the official Matplotlib web page: Matplotlib

1.1 Installation

If you are using our environment, its already installed for you. If you are not using our environment (not recommended), you'll need to install matplotlib first with either:

`%conda install matplotlib`

or

`%pip install matplotlib`

[1]:
```
%pip install matplotlib
```

```
Requirement already satisfied: matplotlib in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(3.9.2)
Requirement already satisfied: contourpy>=1.0.1 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (4.53.1)
Requirement already satisfied: kiwisolver>=1.3.1 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (1.4.5)
Requirement already satisfied: numpy>=1.23 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (2.0.1)
Requirement already satisfied: packaging>=20.0 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (24.1)
Requirement already satisfied: pillow>=8 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (10.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from matplotlib) (2.9.0.post0)
Requirement already satisfied: six>=1.5 in
c:\users\vishwajeet\appdata\local\programs\python\python312\lib\site-packages
(from python-dateutil>=2.7->matplotlib) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

2 Import Matplotlib

[3]:
```
import matplotlib.pyplot as plt
import numpy as np
```

2.1 Basic Array Plot

Let's walk through a very simple example using two numpy arrays. You can also use lists, but most likely you'll be passing numpy arrays or pandas columns (which essentially also behave like arrays).

The data we want to plot:

[4]:
```
x = np.arange(0,10)
y = 2*x
```

[5]:
```
print(x)
print(y)
```

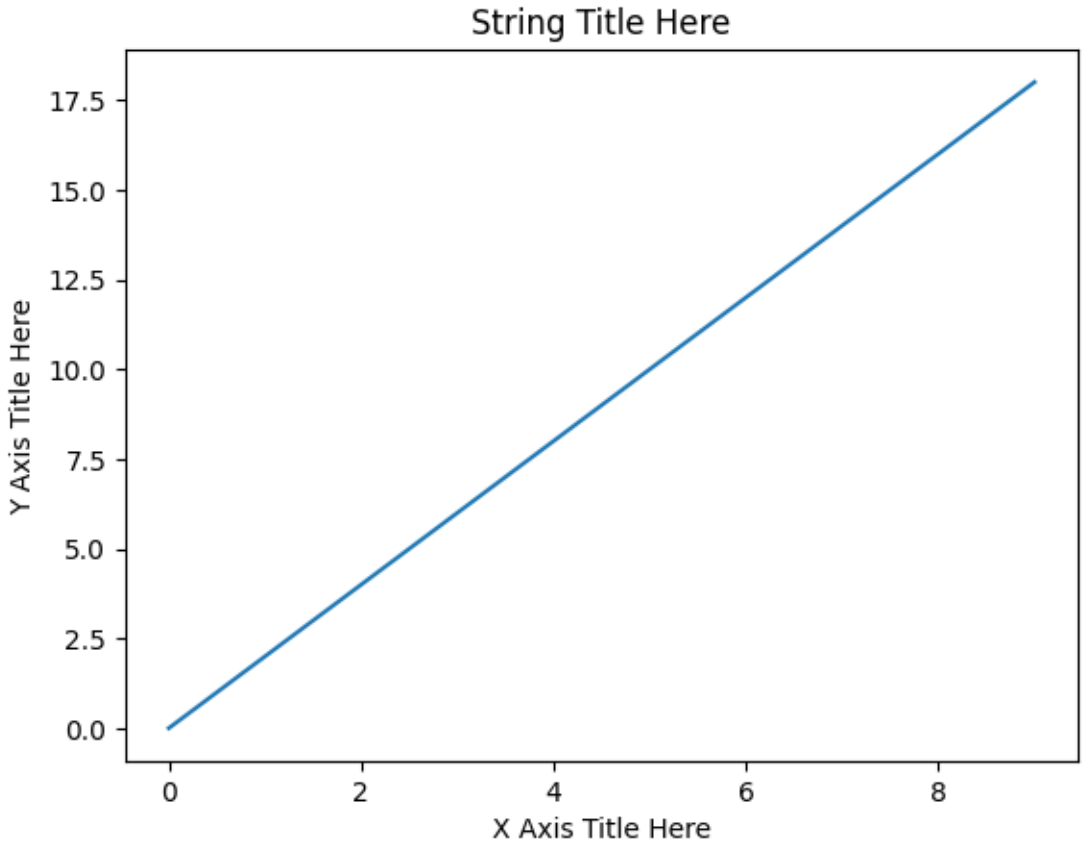
```
[0 1 2 3 4 5 6 7 8 9]
[ 0  2  4  6  8 10 12 14 16 18]
```

3 Using Matplotlib with plt.plot() function calls

3.1 Basic Matplotlib Commands

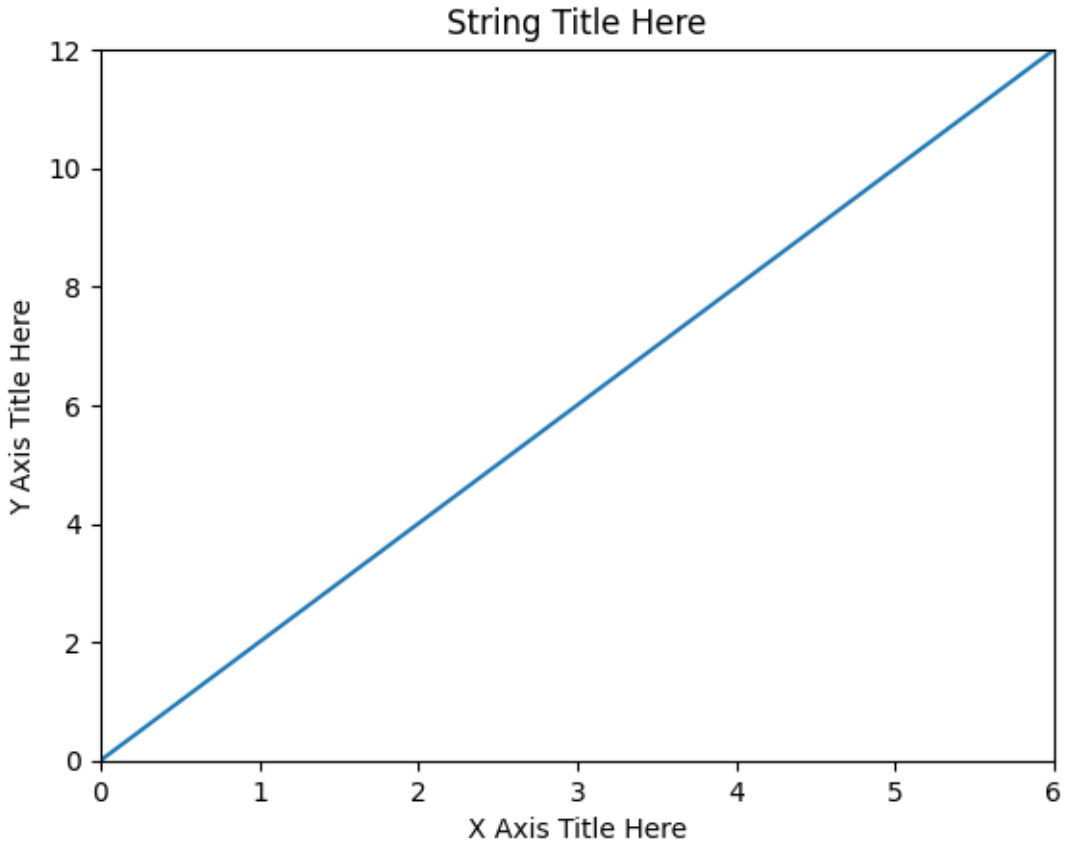
We can create a very simple line plot using the following (I encourage you to pause and use Shift+Tab along the way to check out the document strings for the functions we are using).

[7]:
```
plt.plot(x, y)
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.show()
```



3.2 Editing more figure parameters

[8]:
```
plt.plot(x, y)
plt.xlabel('X Axis Title Here')
plt.ylabel('Y Axis Title Here')
plt.title('String Title Here')
plt.xlim(0,6) # Lower Limit, Upper Limit
plt.ylim(0,12) # Lower Limit, Upper Limit
plt.show() # Required for non-jupyter users , but also removes Out[] info
```



3.3 Exporting a plot

[9]:
```
help(plt.savefig)
```

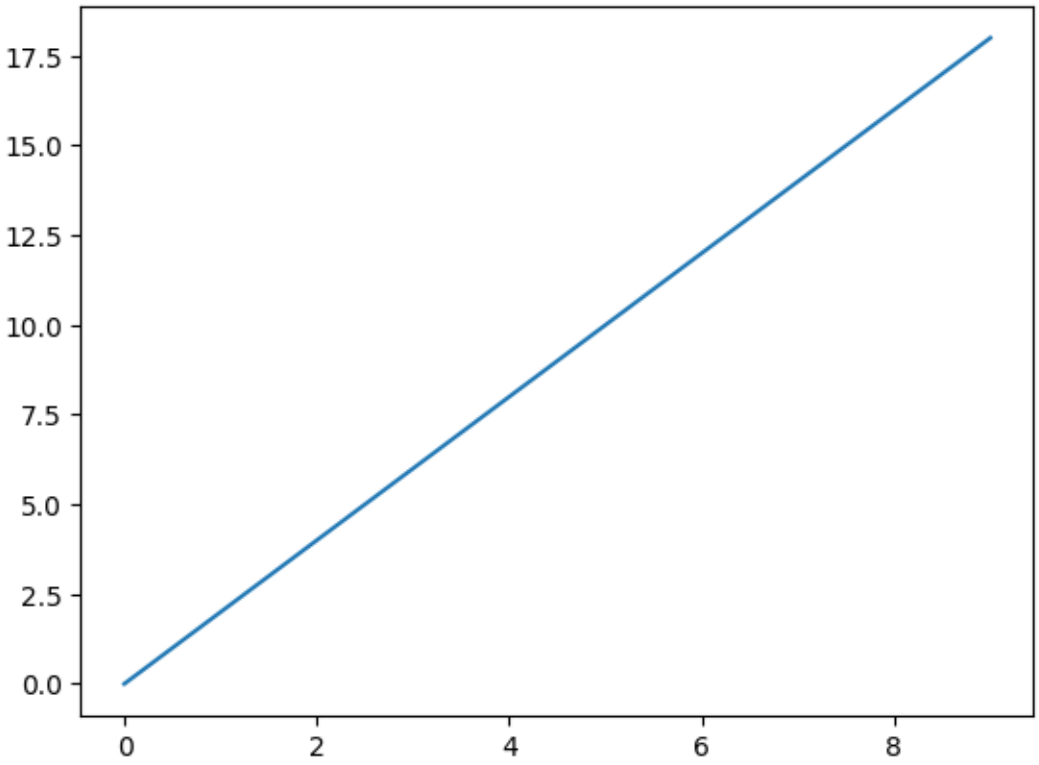
```
Help on function savefig in module matplotlib.pyplot:

savefig(*args, **kwargs) -> 'None'
    Save the current figure as an image or vector graphic to a file.
```

```
Call signature::

  savefig(fname, *, transparent=None, dpi='figure', format=None,
          metadata=None, bbox_inches=None, pad_inches=0.1,
          facecolor='auto', edgecolor='auto', backend=None,
          **kwargs
         )

The available output formats depend on the backend being used.

Parameters
----------
fname : str or path-like or binary file-like
    A path, or a Python file-like object, or
    possibly some backend-dependent object such as
    `matplotlib.backends.backend_pdf.PdfPages`.

    If *format* is set, it determines the output format, and the file
    is saved as *fname*.  Note that *fname* is used verbatim, and there
    is no attempt to make the extension, if any, of *fname* match
    *format*, and no extension is appended.

    If *format* is not set, then the format is inferred from the
    extension of *fname*, if there is one.  If *format* is not
    set and *fname* has no extension, then the file is saved with
    :rc:`savefig.format` and the appropriate extension is appended to
    *fname*.

Other Parameters
----------------
transparent : bool, default: :rc:`savefig.transparent`
    If *True*, the Axes patches will all be transparent; the
    Figure patch will also be transparent unless *facecolor*
    and/or *edgecolor* are specified via kwargs.

    If *False* has no effect and the color of the Axes and
    Figure patches are unchanged (unless the Figure patch
    is specified via the *facecolor* and/or *edgecolor* keyword
    arguments in which case those colors are used).

    The transparency of these patches will be restored to their
    original values upon exit of this function.

    This is useful, for example, for displaying
    a plot on top of a colored background on a web page.

dpi : float or 'figure', default: :rc:`savefig.dpi`
```

```
        The resolution in dots per inch.  If 'figure', use the figure's
        dpi value.

    format : str
        The file format, e.g. 'png', 'pdf', 'svg', … The behavior when
        this is unset is documented under *fname*.

    metadata : dict, optional
        Key/value pairs to store in the image metadata. The supported keys
        and defaults depend on the image format and backend:

        - 'png' with Agg backend: See the parameter ``metadata`` of
          `~.FigureCanvasAgg.print_png`.
        - 'pdf' with pdf backend: See the parameter ``metadata`` of
          `~.backend_pdf.PdfPages`.
        - 'svg' with svg backend: See the parameter ``metadata`` of
          `~.FigureCanvasSVG.print_svg`.
        - 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

        Not supported for 'pgf', 'raw', and 'rgba' as those formats do not
support
        embedding metadata.
        Does not currently support 'jpg', 'tiff', or 'webp', but may include
        embedding EXIF metadata in the future.

    bbox_inches : str or `.Bbox`, default: :rc:`savefig.bbox`
        Bounding box in inches: only the given portion of the figure is
        saved.  If 'tight', try to figure out the tight bbox of the figure.

    pad_inches : float or 'layout', default: :rc:`savefig.pad_inches`
        Amount of padding in inches around the figure when bbox_inches is
        'tight'. If 'layout' use the padding from the constrained or
        compressed layout engine; ignored if one of those engines is not in
        use.

    facecolor : :mpltype:`color` or 'auto', default: :rc:`savefig.facecolor`
        The facecolor of the figure.  If 'auto', use the current figure
        facecolor.

    edgecolor : :mpltype:`color` or 'auto', default: :rc:`savefig.edgecolor`
        The edgecolor of the figure.  If 'auto', use the current figure
        edgecolor.

    backend : str, optional
        Use a non-default backend to render the file, e.g. to render a
        png file with the "cairo" backend rather than the default "agg",
        or a pdf file with the "pgf" backend rather than the default
        "pdf".  Note that the default backend is normally sufficient.  See
```

```
        :ref:`the-builtin-backends` for a list of valid backends for each
        file format.  Custom backends can be referenced as "module://…".

    orientation : {'landscape', 'portrait'}
        Currently only supported by the postscript backend.

    papertype : str
        One of 'letter', 'legal', 'executive', 'ledger', 'a0' through
        'a10', 'b0' through 'b10'. Only supported for postscript
        output.

    bbox_extra_artists : list of `~matplotlib.artist.Artist`, optional
        A list of extra artists that will be considered when the
        tight bbox is calculated.

    pil_kwargs : dict, optional
        Additional keyword arguments that are passed to
        `PIL.Image.Image.save` when saving the figure.

    Notes
    -----

    .. note::

        This is the :ref:`pyplot wrapper <pyplot_interface>` for
`.Figure.savefig`.
```

[10]:
```
plt.plot(x,y)
plt.savefig('sample.png')
```



4 Matplotlib Figure Object

Import the `matplotlib.pyplot` module under the name `plt` (the tidy way):

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

4.0.1 Matplotlib Object Oriented Method

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

[11]:
```
import numpy as np
```

[12]:
```
a = np.linspace(0,10,11)
b = a ** 4
```

[13]:
```
print(a)
print(b)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[0.000e+00 1.000e+00 1.600e+01 8.100e+01 2.560e+02 6.250e+02 1.296e+03
 2.401e+03 4.096e+03 6.561e+03 1.000e+04]
```

[14]:
```
x = np.arange(0,10)
y = 2 * x
```

[15]:
```
print(x)
print(y)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0  2  4  6  8 10 12 14 16 18]
```

4.1 Creating a Figure

The main idea in using the more formal Object Oriented method is to create figure objects and then just call methods or attributes off of that object. This approach is nicer when dealing with a canvas that has multiple plots on it.

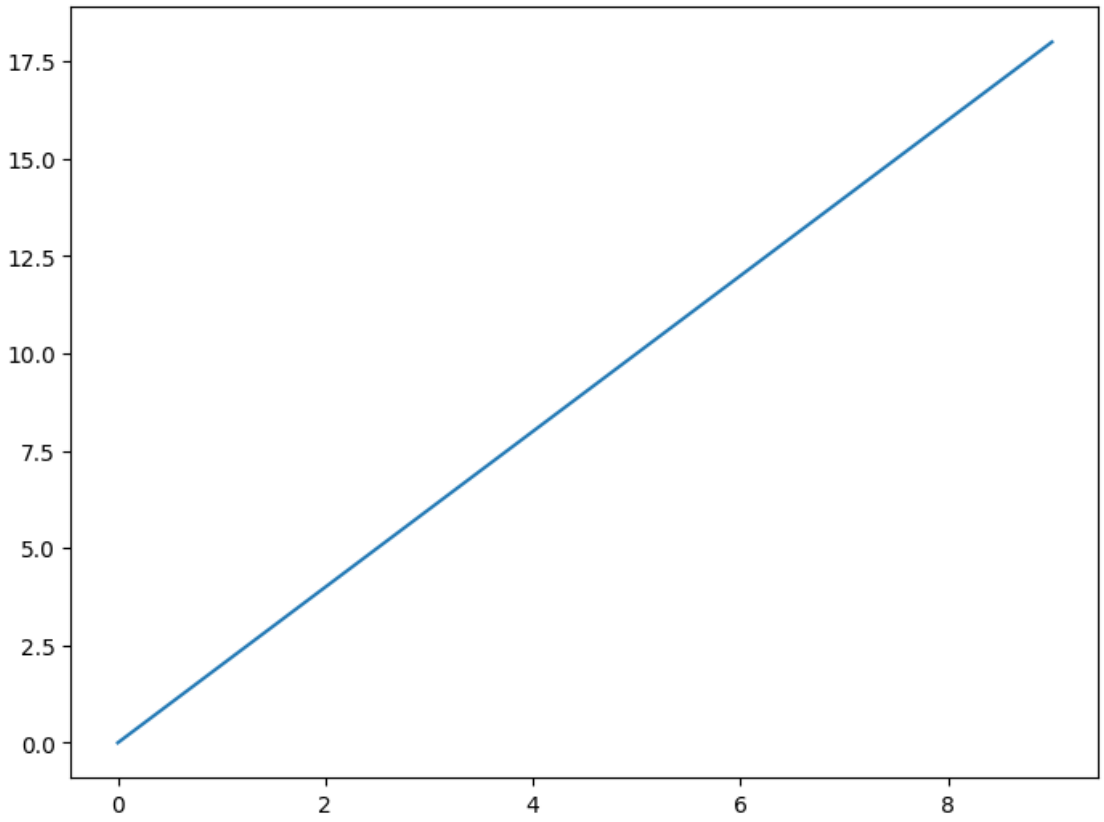
[16]:
```
# Creates blank canvas
fig = plt.figure()
```

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

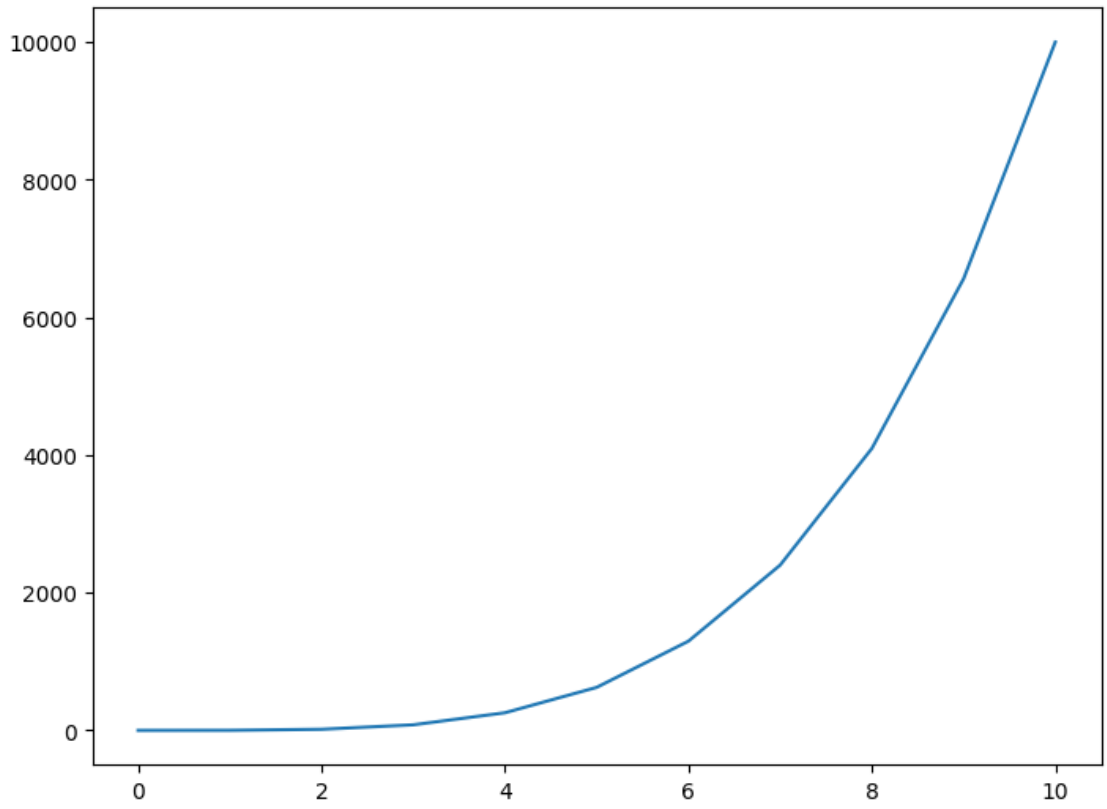
NOTE: ALL THE COMMANDS NEED TO GO IN THE SAME CELL!

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

[17]:
```
# Create Figure (empty canvas)
fig = plt.figure()

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

# Plot on that set of axes
axes.plot(x, y)

plt.show()
```



[18]:
```
# Create Figure (empty canvas)
fig = plt.figure()

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

# Plot on that set of axes
axes.plot(a, b)

plt.show()
```



4.2 Adding another set of axes to the Figure

So far we've only seen one set of axes on this figure object, but we can keep adding new axes on to it at any location and size we want. We can then plot on that new set of axes.

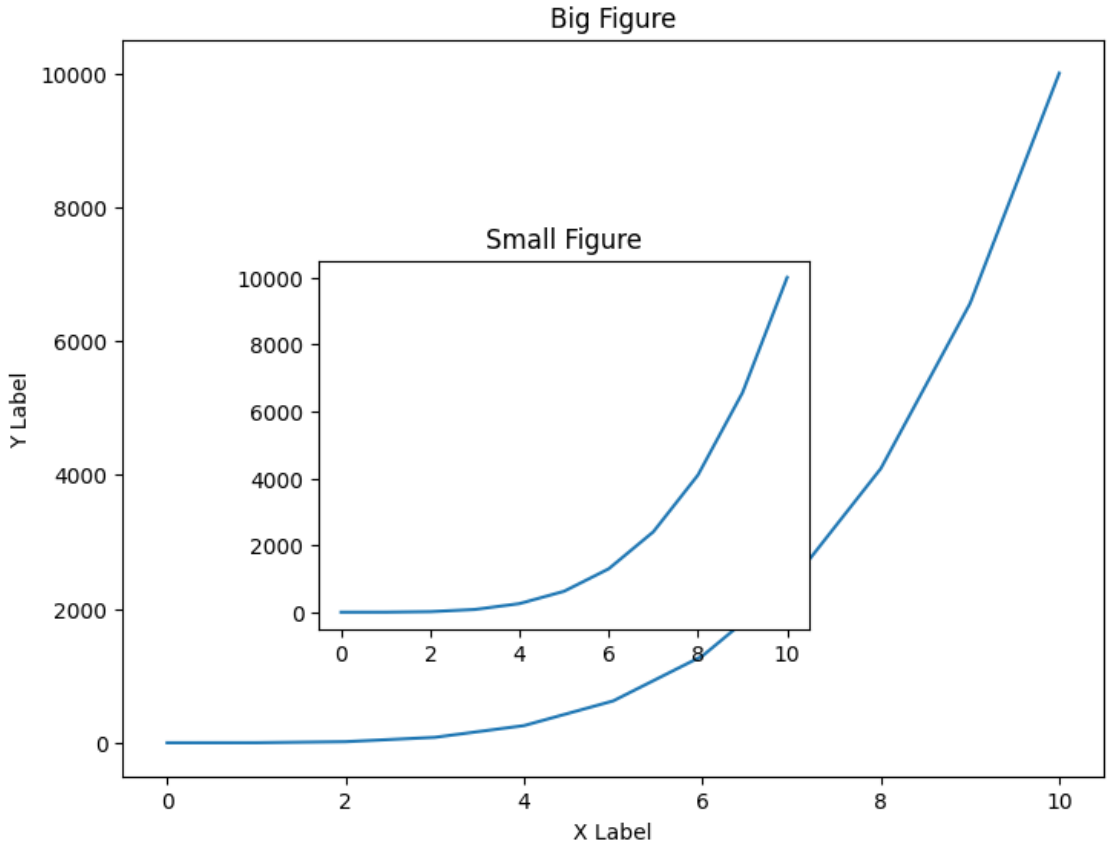
[19]:
```
type(fig)
```

[19]:
```
matplotlib.figure.Figure
```

Code is a little more complicated, but the advantage is that we now have full control of where the plot axes are placed, and we can easily add more than one axis to the figure. Note how we're plotting a,b twice here

[20]:
```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0, 0, 1, 1]) # Large figure
axes2 = fig.add_axes([0.2, 0.2, 0.5, 0.5]) # Smaller figure

# Larger Figure Axes 1
axes1.plot(a, b)
```

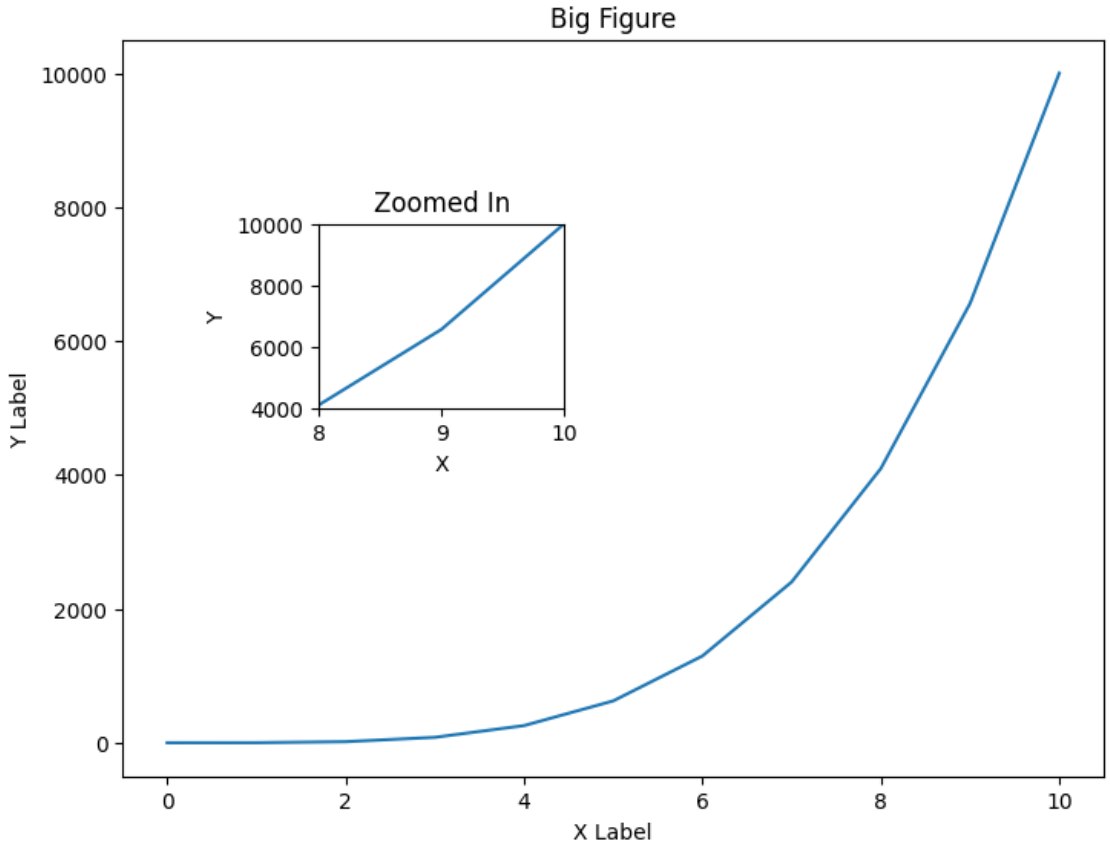
```
# Use set_ to add to the axes figure
axes1.set_xlabel('X Label')
axes1.set_ylabel('Y Label')
axes1.set_title('Big Figure')

# Insert Figure Axes 2
axes2.plot(a,b)
axes2.set_title('Small Figure');
```



Let's move the small figure and edit its parameters.

[21]:
```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0, 0, 1, 1]) # Large figure
axes2 = fig.add_axes([0.2, 0.5, 0.25, 0.25]) # Smaller figure

# Larger Figure Axes 1
axes1.plot(a, b)
```

```
# Use set_ to add to the axes figure
axes1.set_xlabel('X Label')
axes1.set_ylabel('Y Label')
axes1.set_title('Big Figure')

# Insert Figure Axes 2
axes2.plot(a,b)
axes2.set_xlim(8,10)
axes2.set_ylim(4000,10000)
axes2.set_xlabel('X')
axes2.set_ylabel('Y')
axes2.set_title('Zoomed In');
```

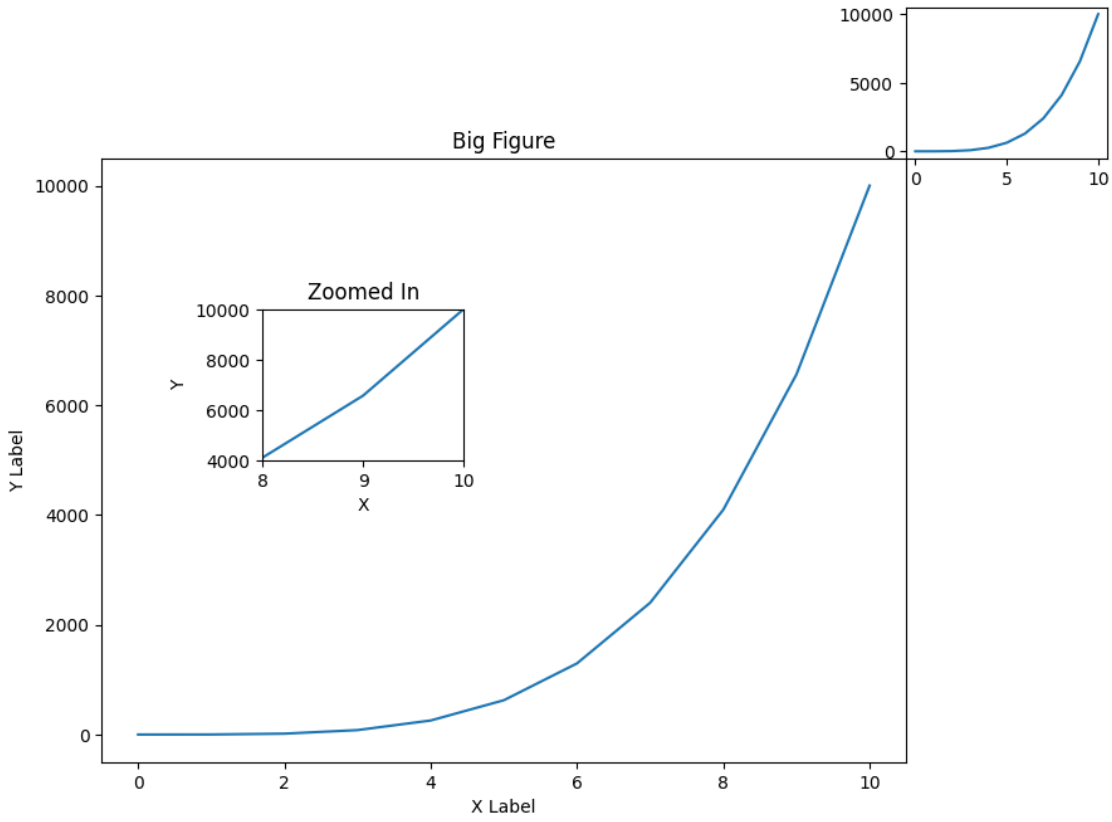


You can add as many axes on to the same figure as you want, even outside of the main figure if the length and width correspond to this.

[22]:
```
# Creates blank canvas
fig = plt.figure()

axes1 = fig.add_axes([0, 0, 1, 1]) # Full figure
```

```
axes2 = fig.add_axes([0.2, 0.5, 0.25, 0.25]) # Smaller figure
axes3 = fig.add_axes([1, 1, 0.25, 0.25]) # Starts at top right corner!

# Larger Figure Axes 1
axes1.plot(a, b)

# Use set_ to add to the axes figure
axes1.set_xlabel('X Label')
axes1.set_ylabel('Y Label')
axes1.set_title('Big Figure')

# Insert Figure Axes 2
axes2.plot(a,b)
axes2.set_xlim(8,10)
axes2.set_ylim(4000,10000)
axes2.set_xlabel('X')
axes2.set_ylabel('Y')
axes2.set_title('Zoomed In');

# Insert Figure Axes 3
axes3.plot(a,b)
```

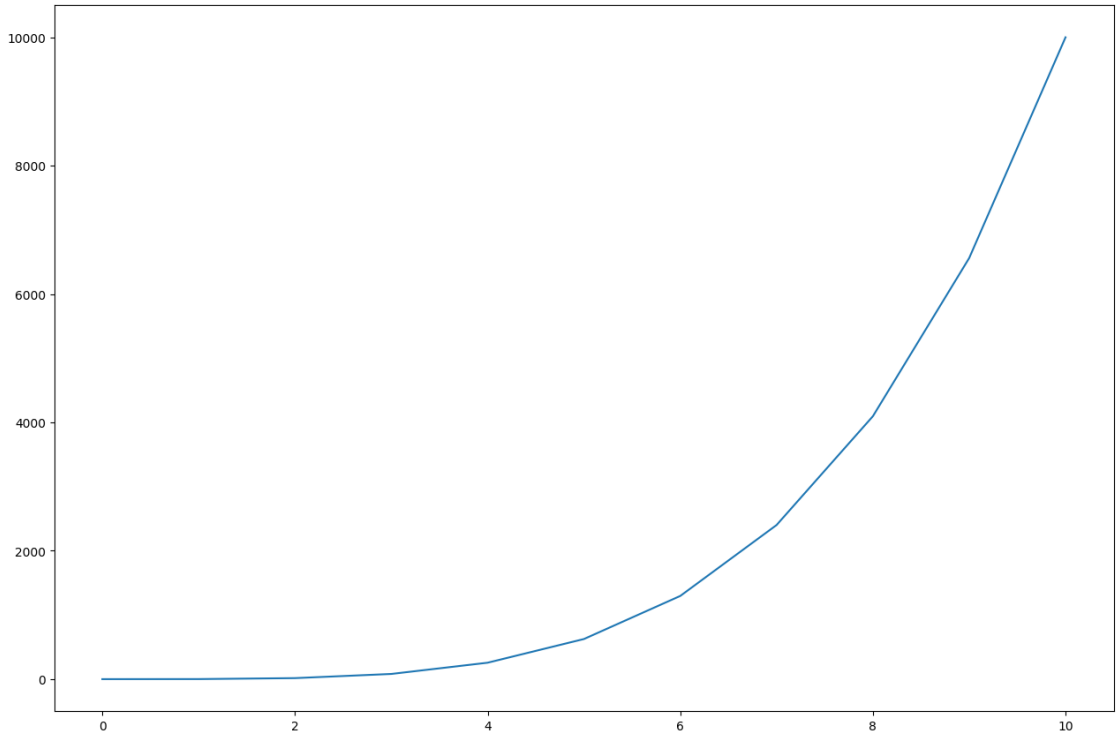
[22]: [<matplotlib.lines.Line2D at 0x1a3d17ee060>]



4.3 Figure Parameters

[23]:
```
# Creates blank canvas
fig = plt.figure(figsize=(12,8),dpi=100)

axes1 = fig.add_axes([0, 0, 1, 1])

axes1.plot(a,b)
```

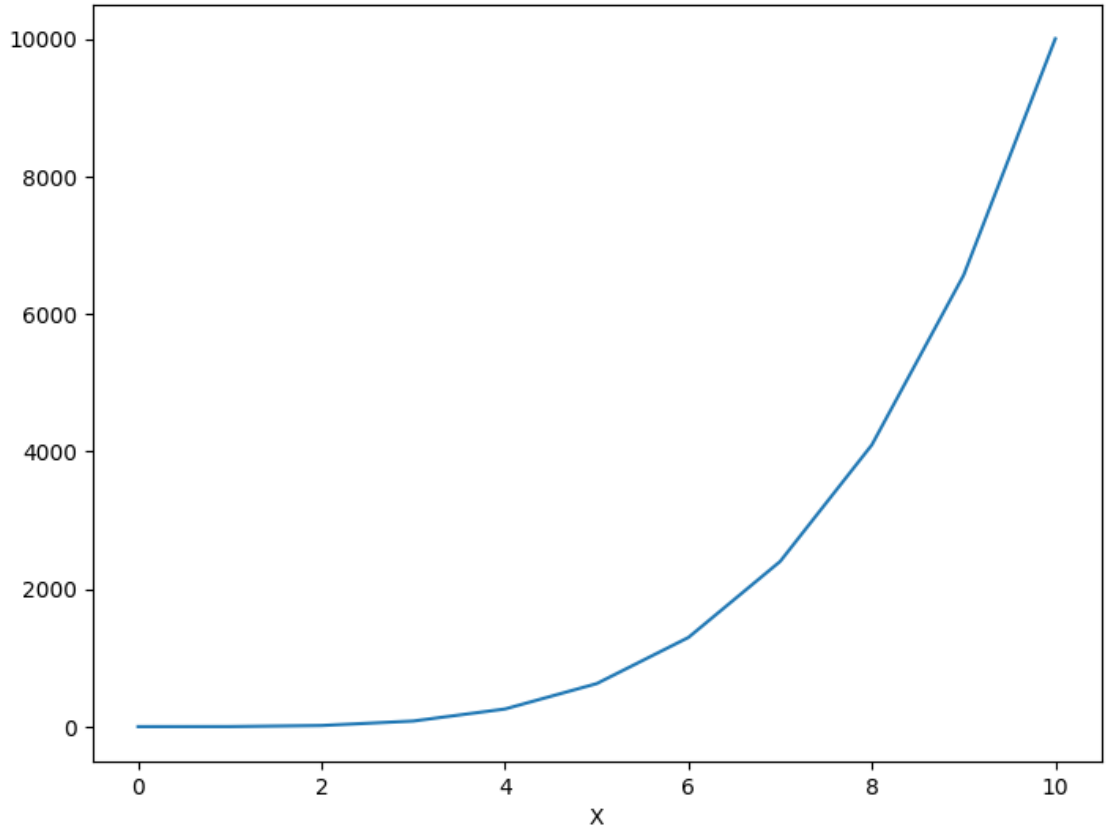
[23]:
```
[<matplotlib.lines.Line2D at 0x1a3d05d55e0>]
```



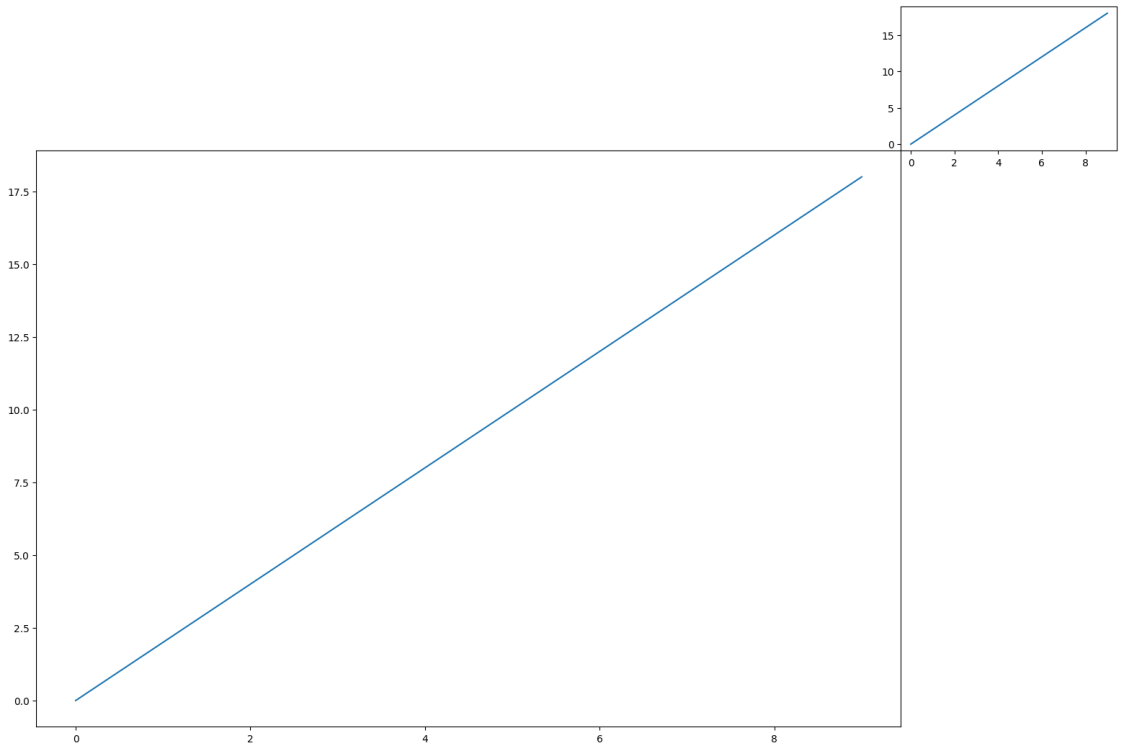
4.4 Exporting a Figure

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

axes1 = fig.add_axes([0, 0, 1, 1])

axes1.plot(a,b)
axes1.set_xlabel('X')
```

```
# bbox_inches ='tight' automatically makes sure the bounding box is correct
fig.savefig('figure.png',bbox_inches='tight')
```



[25]:
```
# Creates blank canvas
fig = plt.figure(figsize=(12,8))

axes1 = fig.add_axes([0, 0, 1, 1]) # Full figure
axes2 = fig.add_axes([1, 1, 0.25, 0.25]) # Starts at top right corner!

# Larger Figure Axes 1
axes1.plot(x,y)

# Insert Figure Axes 2
axes2.plot(x,y)

fig.savefig('test.png',bbox_inches='tight')
```



5 Matplotlib Sub Plots

Import the `matplotlib.pyplot` module under the name `plt` (the tidy way):

[26]:
```
import matplotlib.pyplot as plt
```

5.1 The Data

[27]:
```
import numpy as np
```

[28]:
```
a = np.linspace(0,10,11)
b = a ** 4
```

[29]:
```
print(a)
print(b)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
[0.000e+00 1.000e+00 1.600e+01 8.100e+01 2.560e+02 6.250e+02 1.296e+03
 2.401e+03 4.096e+03 6.561e+03 1.000e+04]
```

[30]:
```
x = np.arange(0,10)
y = 2 * x
```

[32]:
```
print(x)
print(type(x))

print(y)
print(type(y))
```

```
[0 1 2 3 4 5 6 7 8 9]
<class 'numpy.ndarray'>
[ 0  2  4  6  8 10 12 14 16 18]
<class 'numpy.ndarray'>
```

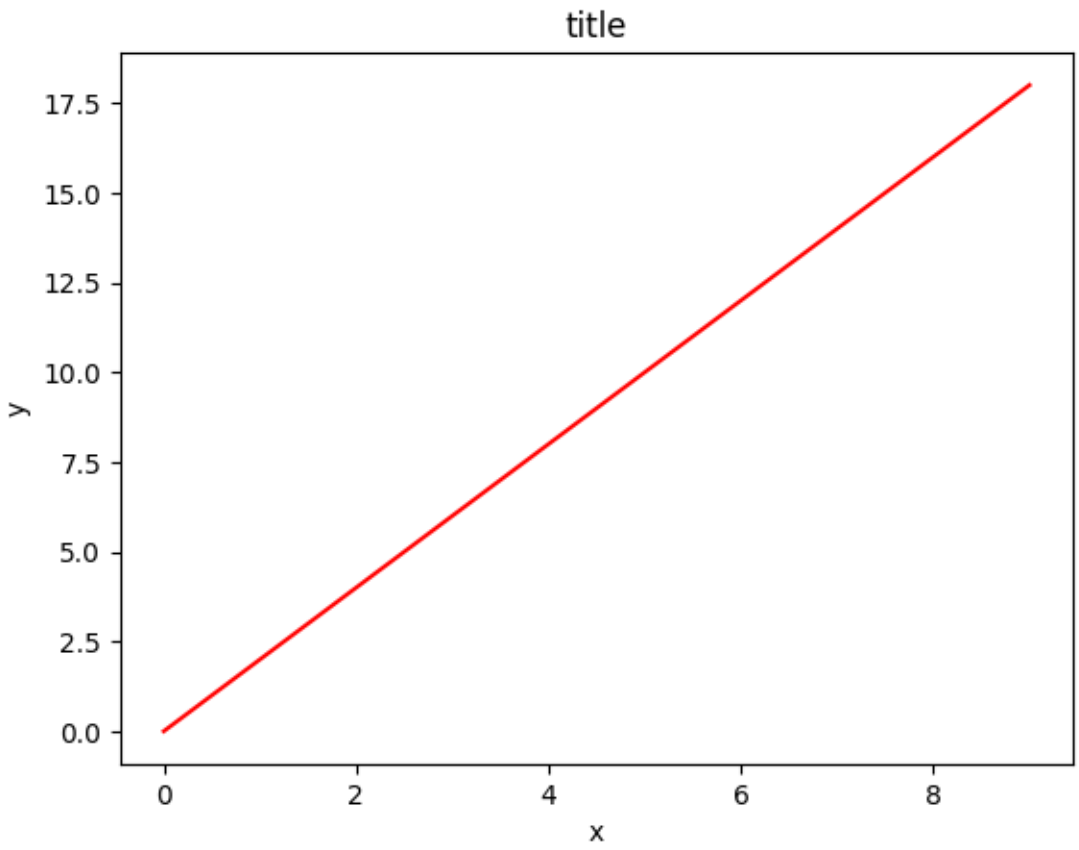
6 plt.subplots()

NOTE: Make sure you put the commands all together in the same cell as we do in this notebook and video!

The plt.subplots() object will act as a more automatic axis manager. This makes it much easier to show multiple plots side by side.

Note how we use tuple unpacking to grba both the Figure object and a numpy array of axes:

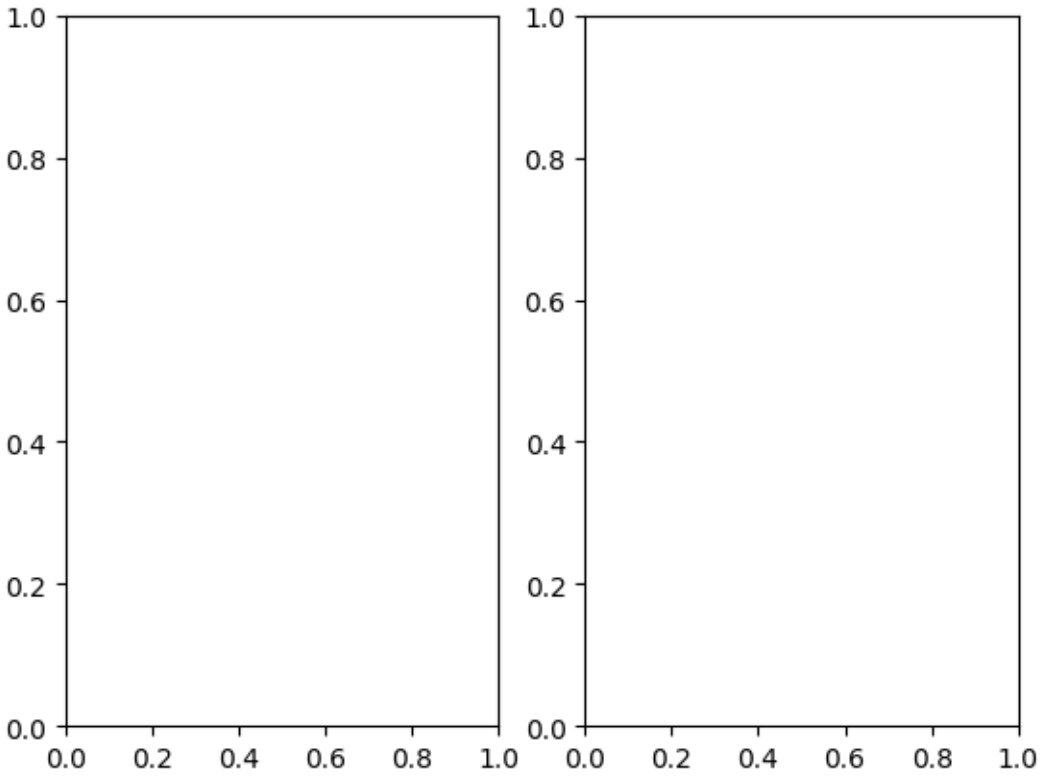
[33]:
```
# Use similar to plt.figure() except use tuple unpacking to grab fig and axes
fig, axes = plt.subplots()

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



6.1 Adding rows and columns

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

[35]:
```
# Empty canvas of 1 by 2 subplots
fig, axes = plt.subplots(nrows=1, ncols=2)
```



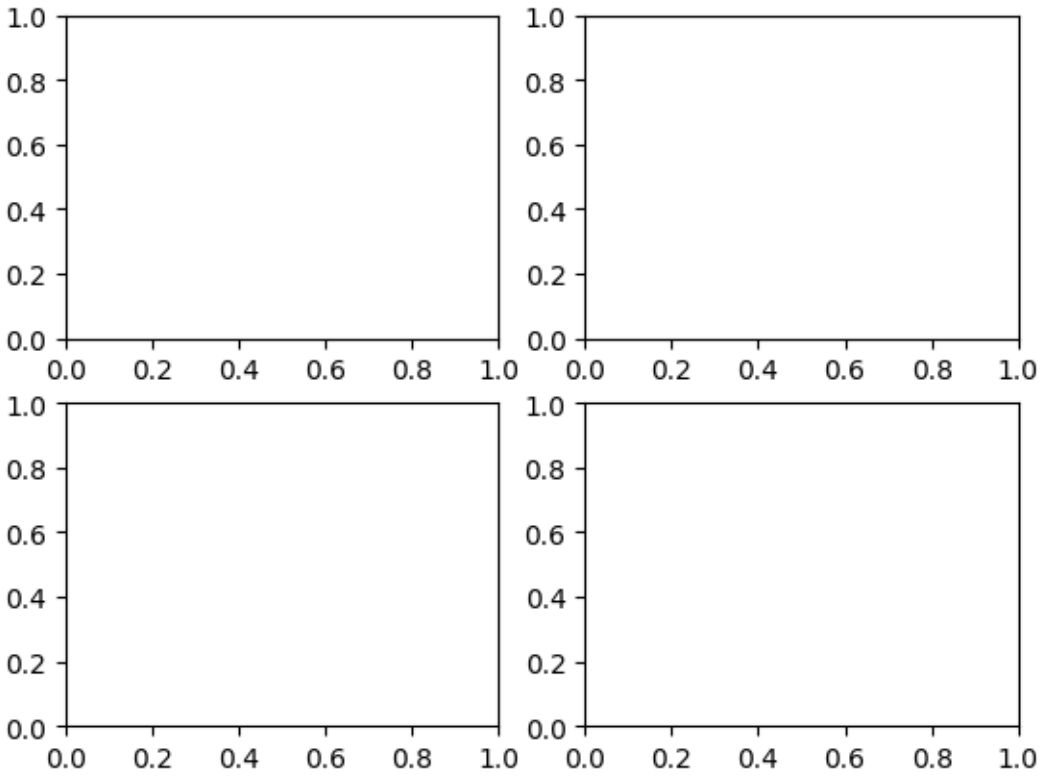
[36]:
```
# Axes is an array of axes to plot on
axes
```

[36]:
```
array([<Axes: >, <Axes: >], dtype=object)
```

[37]:
```
axes.shape
```

[37]:
```
(2,)
```

[38]:
```
# Empty canvas of 2 by 2 subplots
fig, axes = plt.subplots(nrows=2, ncols=2)
```



[39]:
```
axes
```

[39]:
```
array([[<Axes: >, <Axes: >],
       [<Axes: >, <Axes: >]], dtype=object)
```

[40]:
```
axes.shape
```

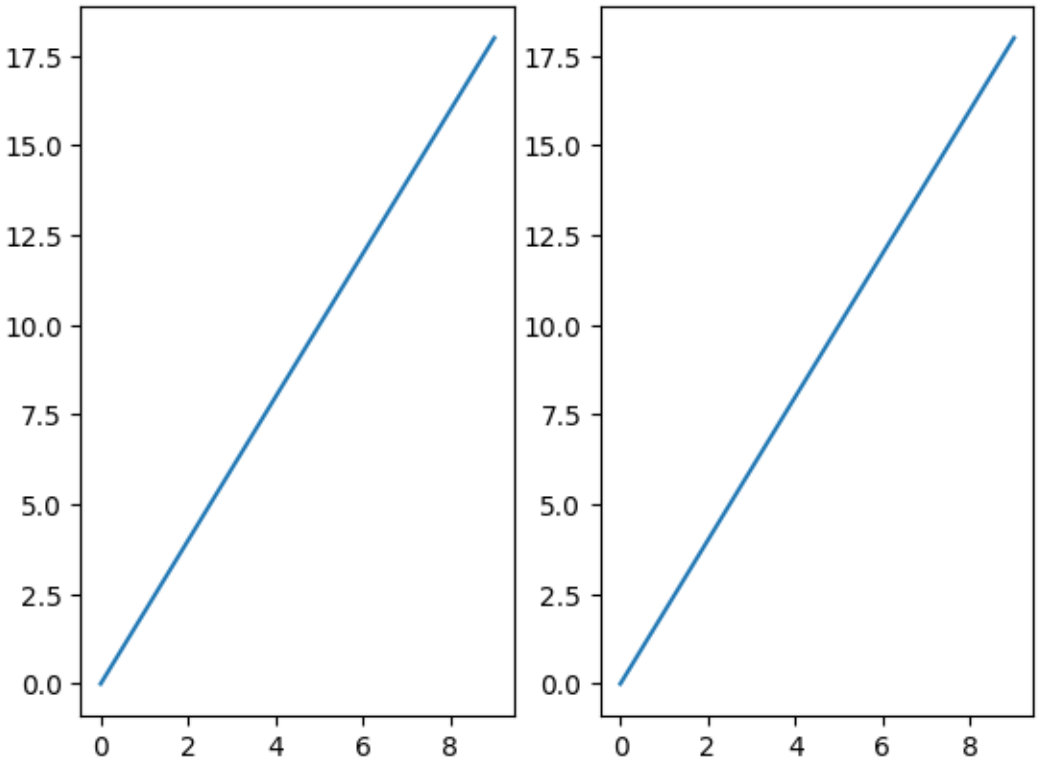
[40]:
```
(2, 2)
```

6.2 Plotting on axes objects

Just as before, we simple .plot() on the axes objects, and we can also use the .set_ methods on each axes.

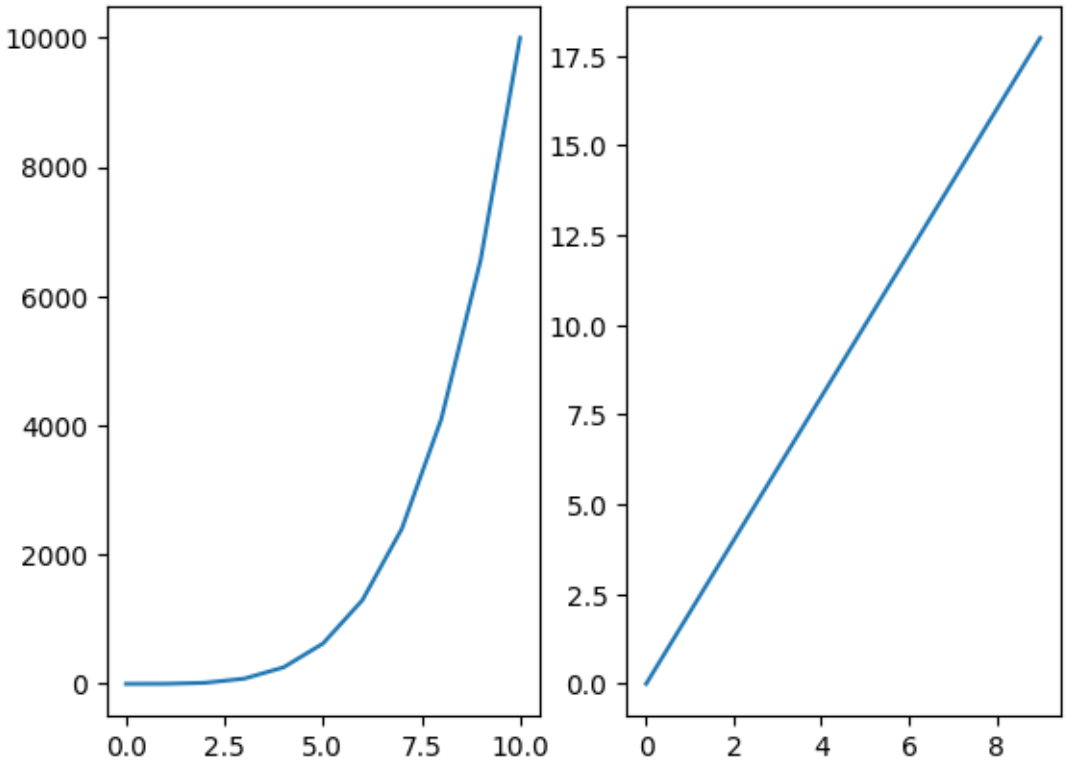
Let's explore this, make sure this is all in the same cell:

[41]:
```
fig,axes = plt.subplots(nrows=1,ncols=2)

for axe in axes:
    axe.plot(x,y)
```



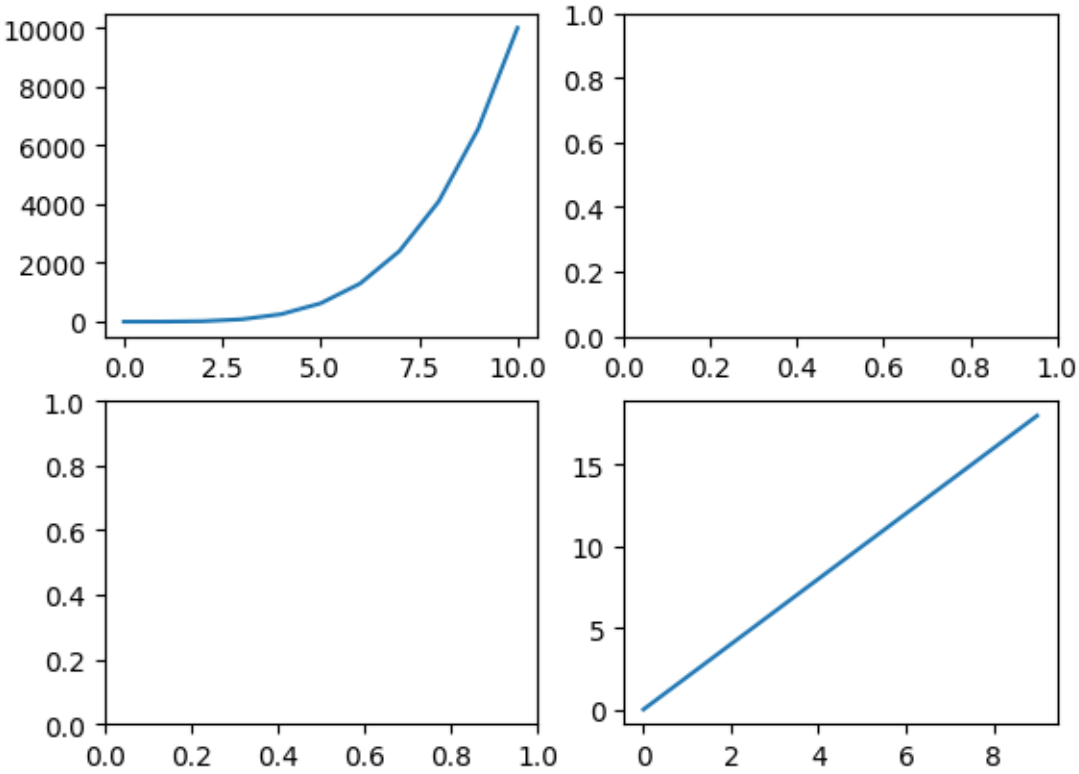
[42]:
```
fig,axes = plt.subplots(nrows=1,ncols=2)

axes[0].plot(a,b)
axes[1].plot(x,y)
```

[42]: [<matplotlib.lines.Line2D at 0x1a3d068a6c0>]



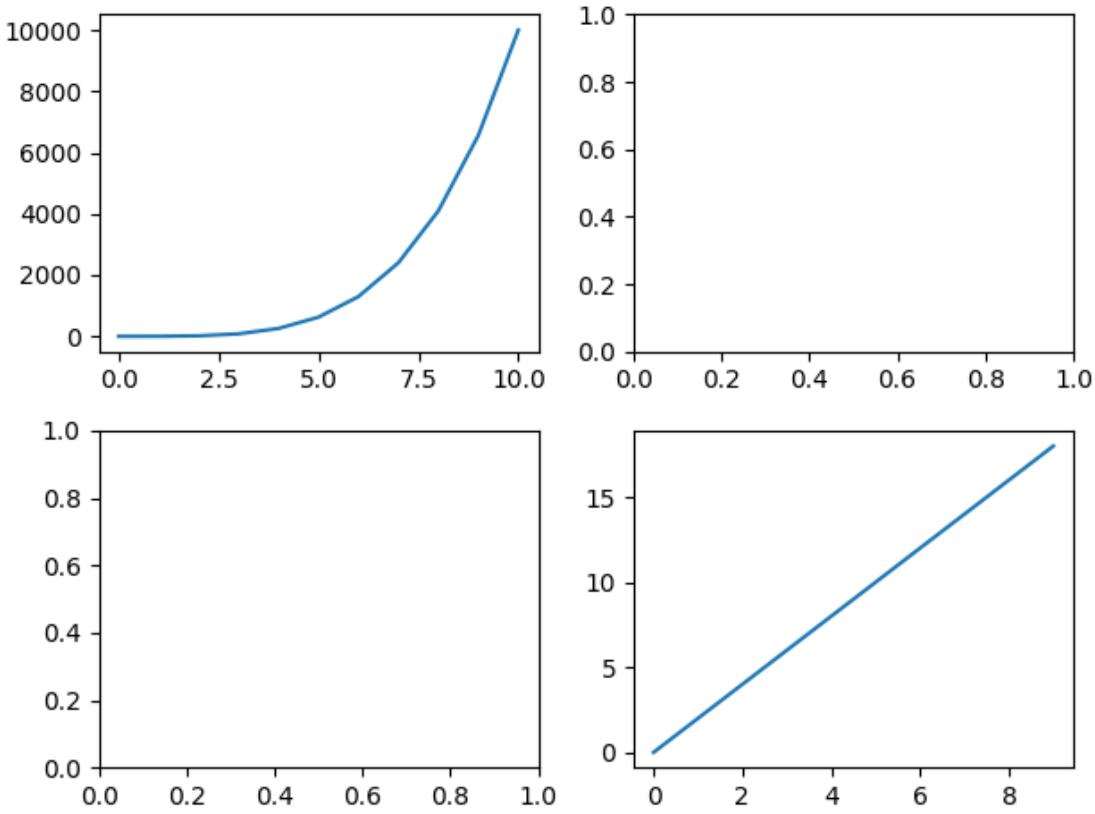
[43]:
```
# NOTE! This returns 2 dimensional array
fig,axes = plt.subplots(nrows=2,ncols=2)

axes[0][0].plot(a,b)
axes[1][1].plot(x,y)
```

[43]: [<matplotlib.lines.Line2D at 0x1a3d0586ab0>]



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

[44]:
```
# NOTE! This returns 2 dimensional array
fig,axes = plt.subplots(nrows=2,ncols=2)

axes[0][0].plot(a,b)
axes[1][1].plot(x,y)

plt.tight_layout()
```

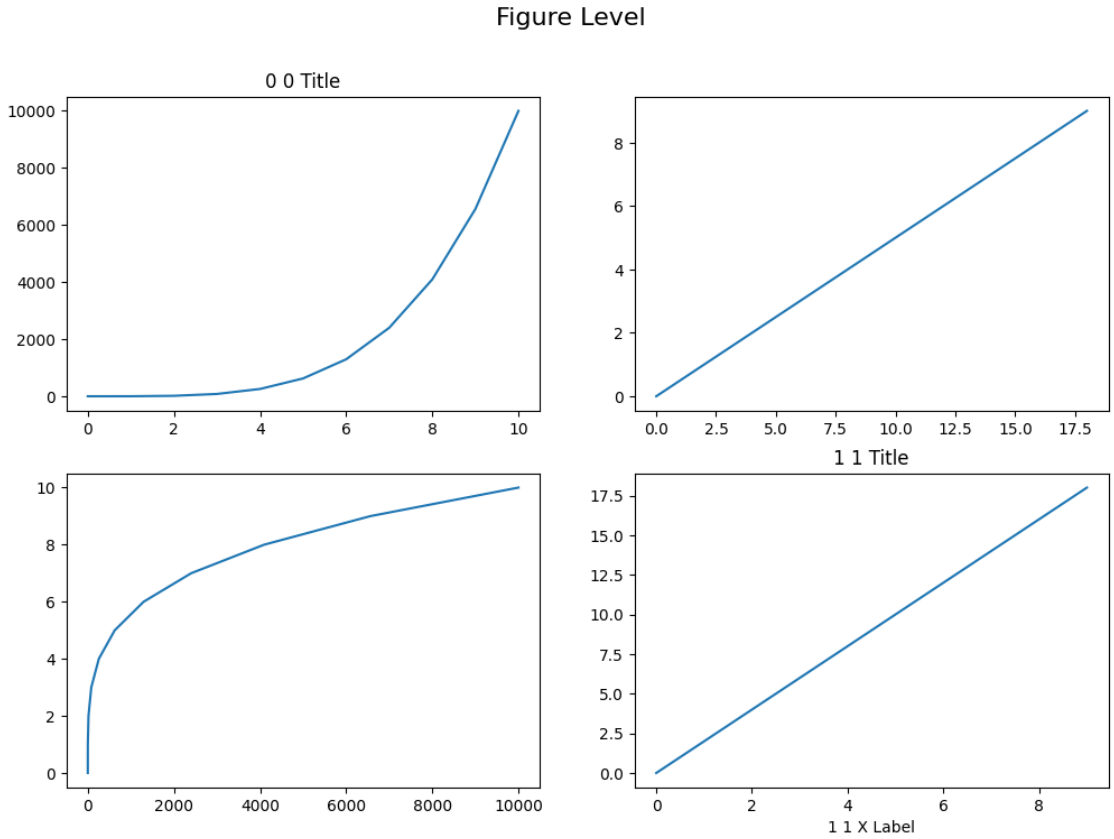


6.3 Parameters on subplots()

Recall we have both the Figure object and the axes. Meaning we can edit properties at both levels.

[45]:
```
fig,axes = plt.subplots(nrows=2,ncols=2,figsize=(12,8))

# SET YOUR AXES PARAMETERS FIRST

# Parameters at the axes level
axes[0][0].plot(a,b)
axes[0][0].set_title('0 0 Title')


axes[1][1].plot(x,y)
axes[1][1].set_title('1 1 Title')
axes[1][1].set_xlabel('1 1 X Label')

axes[0][1].plot(y,x)
axes[1][0].plot(b,a)

# THEN SET OVERALL FIGURE PARAMETERS
```

```
# Parameters at the Figure level
fig.suptitle("Figure Level",fontsize=16)


plt.show()
```



6.3.1 Manual spacing on subplots()

Use .subplots_adjust to adjust spacing manually.

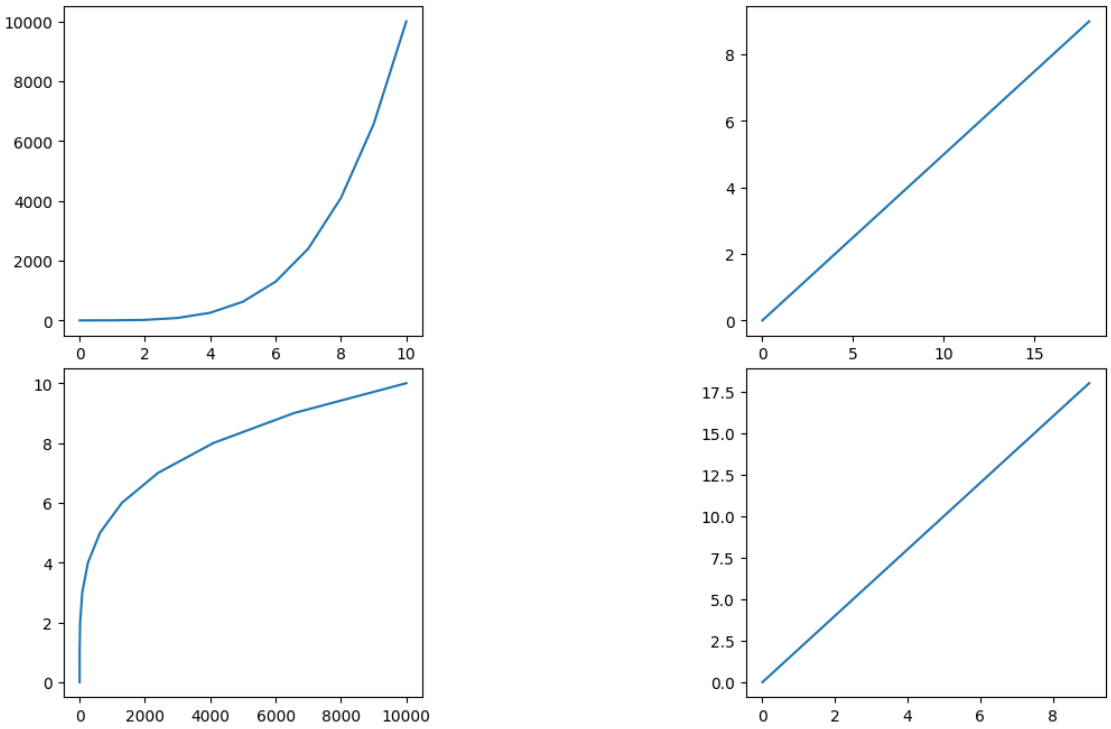
Full Details Here: https://matplotlib.org/3.2.2/api/_as_gen/matplotlib.pyplot.subplots_adjust.html

Example from link:

- left = 0.125 # the left side of the subplots of the figure
- right = 0.9 # the right side of the subplots of the figure
- bottom = 0.1 # the bottom of the subplots of the figure
- top = 0.9 # the top of the subplots of the figure
- wspace = 0.2 # the amount of width reserved for space between subplots, # expressed as a fraction of the average axis width

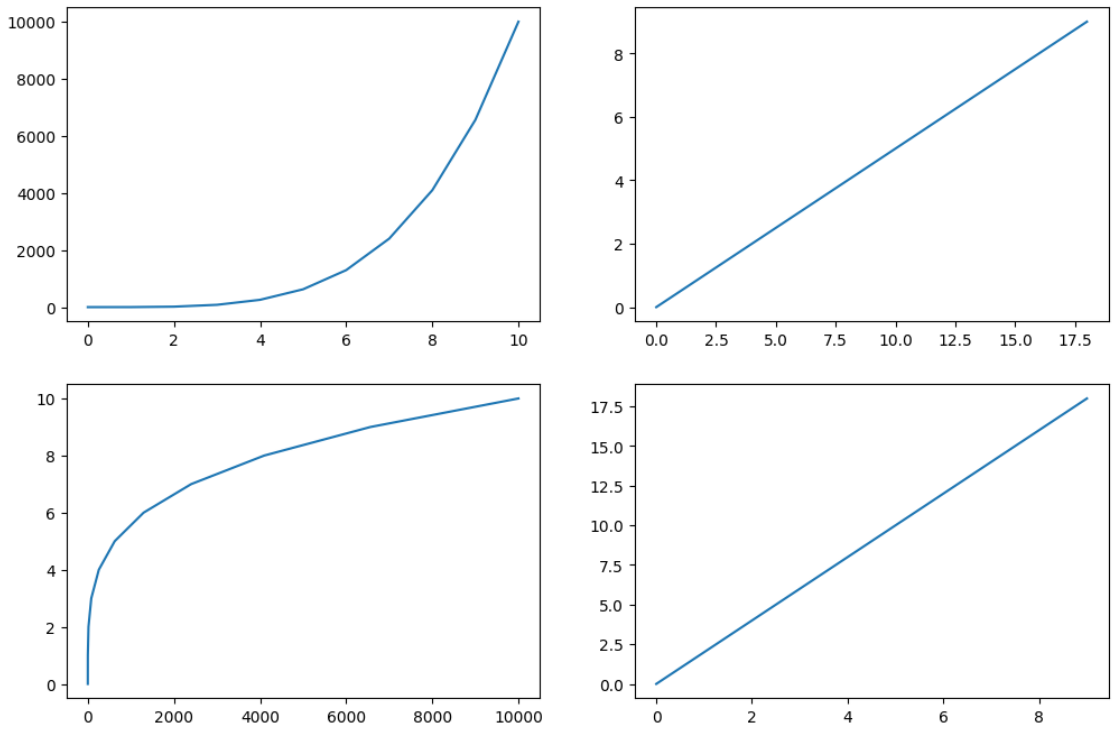
- hspace = 0.2 # the amount of height reserved for space between subplots, # expressed as a fraction of the average axis height

[46]:
```
fig,axes = plt.subplots(nrows=2,ncols=2,figsize=(12,8))

# Parameters at the axes level
axes[0][0].plot(a,b)
axes[1][1].plot(x,y)
axes[0][1].plot(y,x)
axes[1][0].plot(b,a)

# Use left,right,top, bottom to stretch subplots
# Use wspace,hspace to add spacing between subplots
fig.subplots_adjust(left=None,
    bottom=None,
    right=None,
    top=None,
    wspace=0.9,
    hspace=0.1,)

plt.show()
```



6.4 Exporting plt.subplots()

[47]:
```
# NOTE! This returns 2 dimensional array
fig,axes = plt.subplots(nrows=2,ncols=2,figsize=(12,8))

axes[0][0].plot(a,b)
axes[1][1].plot(x,y)
axes[0][1].plot(y,x)
axes[1][0].plot(b,a)

fig.savefig('subplots.png',bbox_inches='tight')

plt.show()
```



7 Matplotlib Styling

Import the `matplotlib.pyplot` module under the name `plt` (the tidy way):

[48]:
```
import matplotlib.pyplot as plt
```

7.1 The Data

[49]:
```
import numpy as np
```

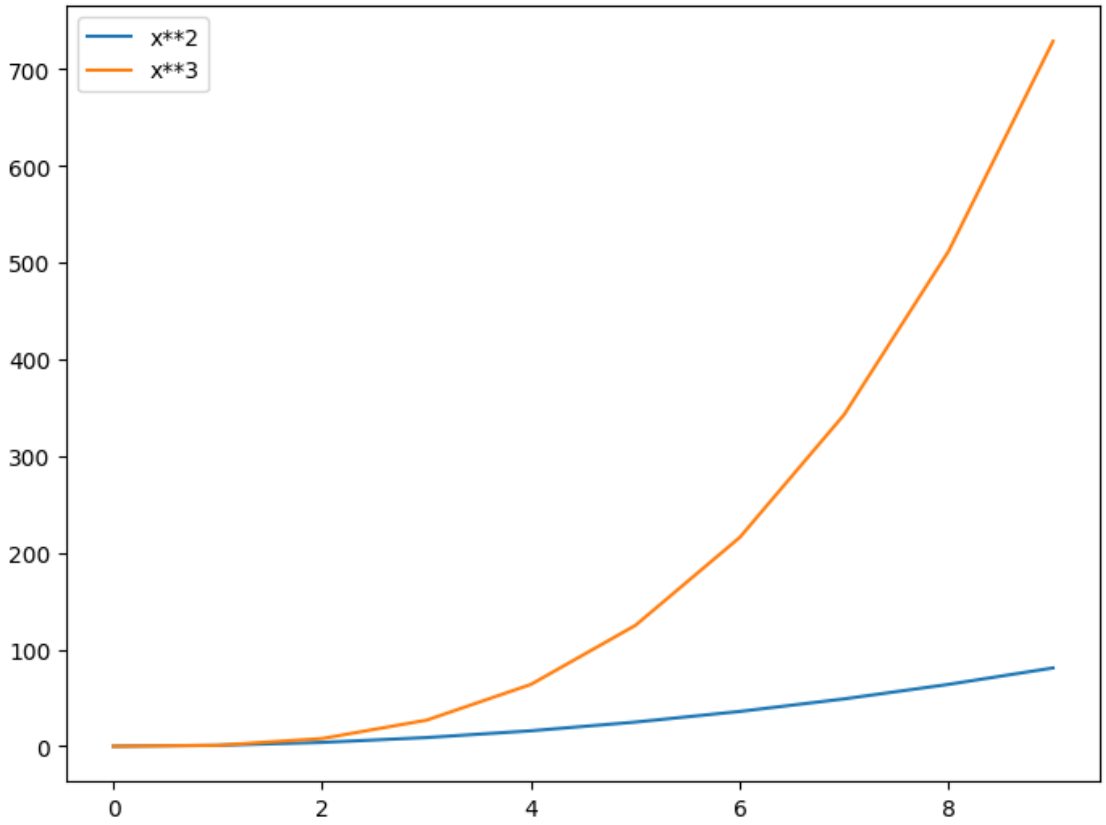
[50]:
```
x = np.arange(0,10)
y = 2 * x
```

7.2 Legends

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

[51]:
```
fig = plt.figure()

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

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

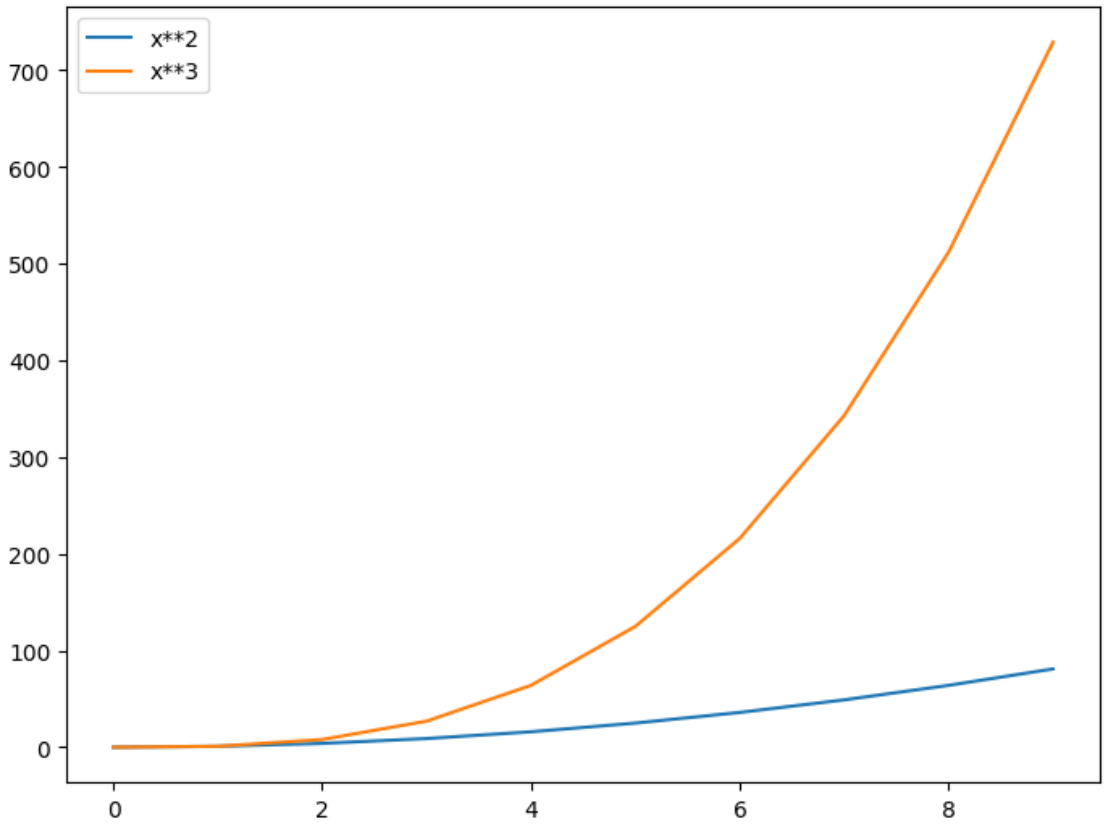
[51]:
```
<matplotlib.legend.Legend at 0x1a3d2bb2d50>
```



Notice how legend could potentially overlap some of the actual plot!

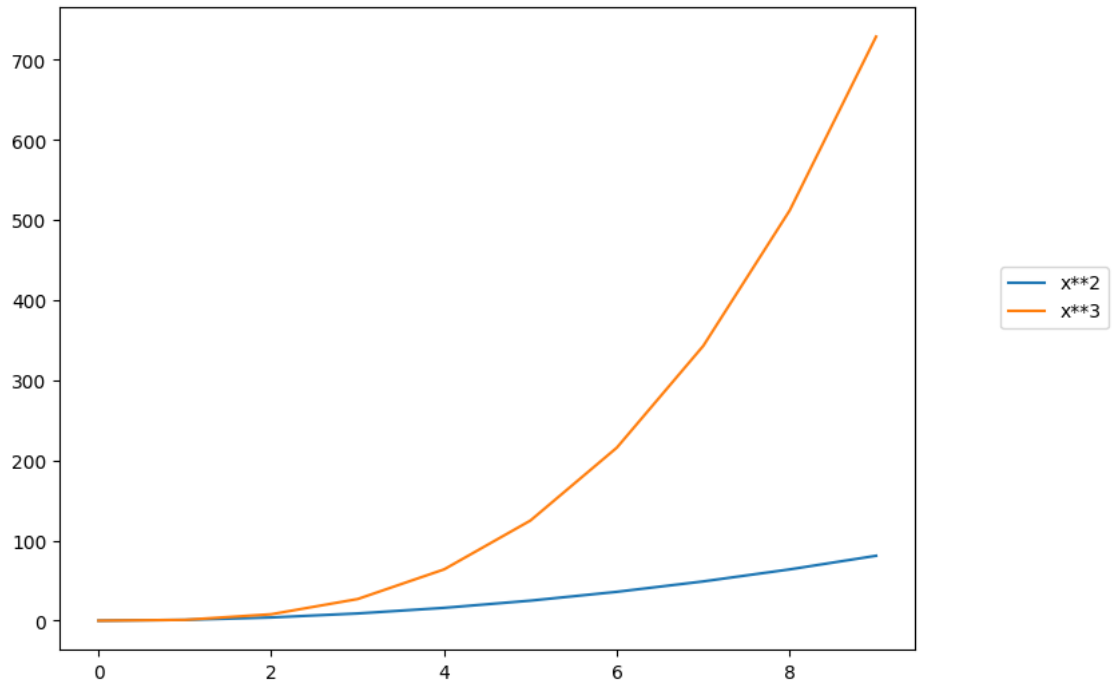
The **legend** function takes an optional keyword argument **loc** that can be used to specify where in the figure the legend is to be drawn. The allowed values of **loc** are numerical codes for the various places the legend can be drawn. See the documentation page for details. Some of the most common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

[52]:
```
# Lots of options....

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

# .. many more options are available

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

[52]:


[53]:
```
ax.legend(loc=(1.1,0.5)) # manually set location
fig
```

[53]:



7.3 Setting colors, linewidths, linetypes

Matplotlib gives you *a lot* of options for customizing colors, linewidths, and linetypes.

There is the basic MATLAB like syntax (which I would suggest you avoid using unless you already feel really comfortable with MATLAB). Instead let's focus on the keyword parameters.

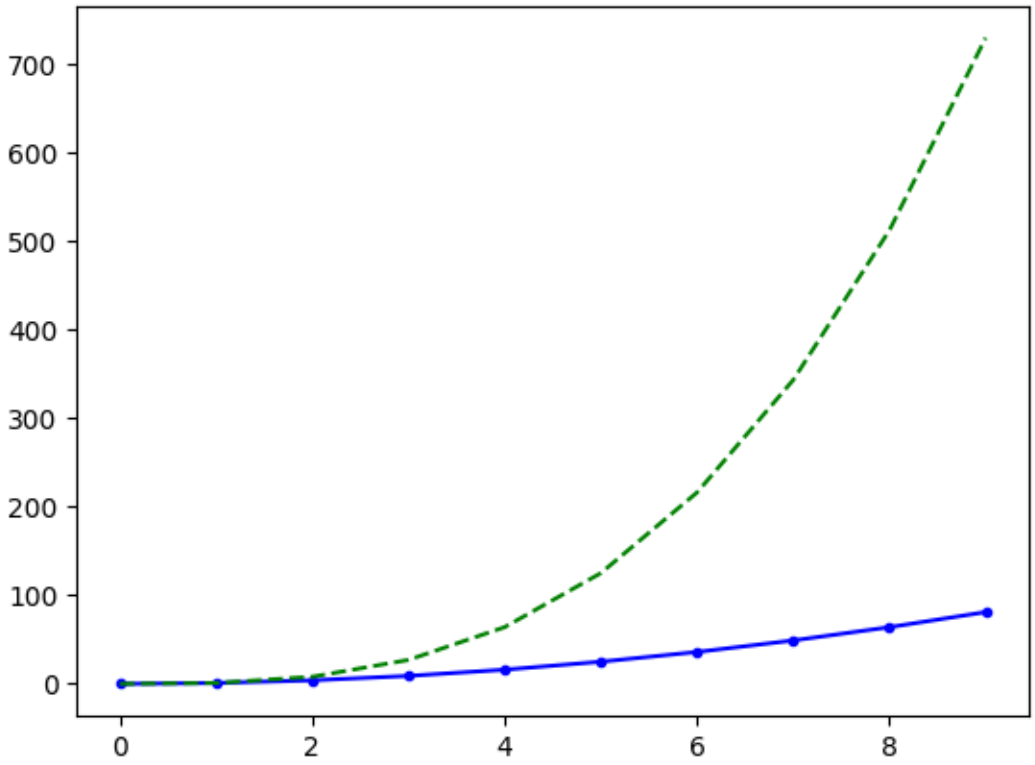
7.3.1 Quick View:

7.3.2 Colors with MatLab like syntax

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

[54]:
```
# MATLAB style line color and style
fig, ax = plt.subplots()
ax.plot(x, x**2, 'b.-') # blue line with dots
ax.plot(x, x**3, 'g--') # green dashed line
```

[54]:
```
[<matplotlib.lines.Line2D at 0x1a3d2a6d9d0>]
```



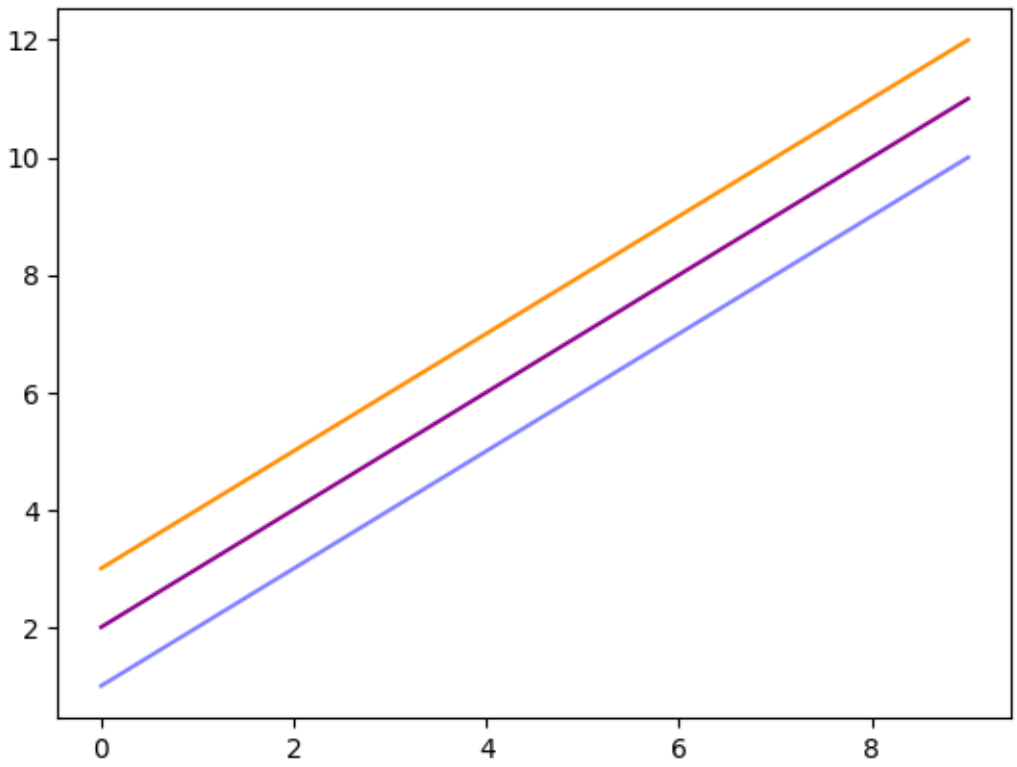
7.4 Suggested Approach: Use keyword arguments

7.4.1 Colors with the color parameter

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

[55]:
```
fig, ax = plt.subplots()

ax.plot(x, x+1, color="blue", alpha=0.5) # half-transparant
ax.plot(x, x+2, color="#8B008B")         # RGB hex code
ax.plot(x, x+3, color="#FF8C00")         # RGB hex code
```

[55]:
```
[<matplotlib.lines.Line2D at 0x1a3d22f7950>]
```



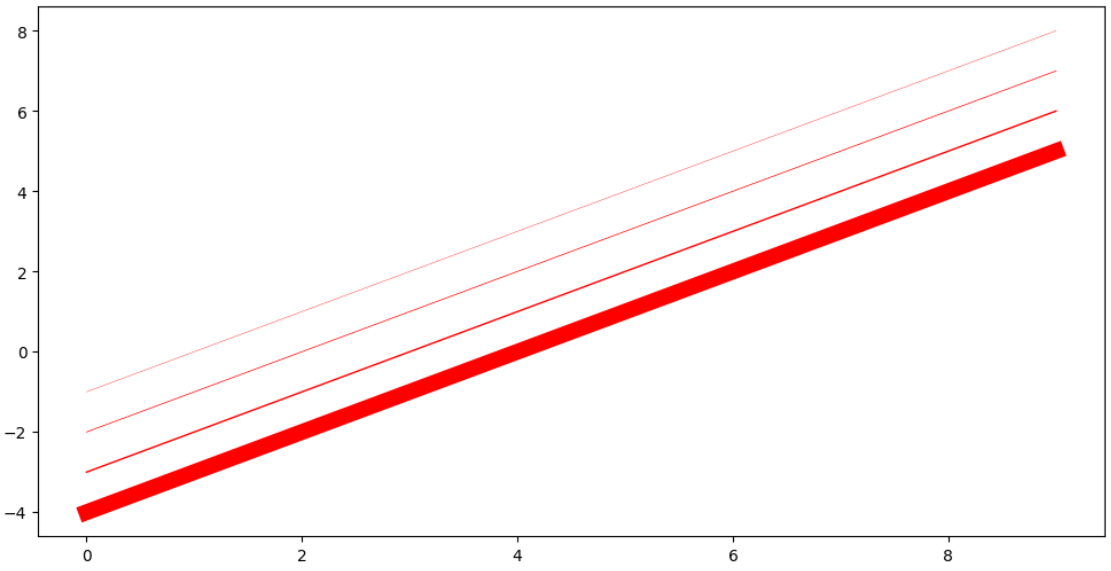
7.5 Line and marker styles

7.6 Linewidth

To change the line width, we can use the `linewidth` or `lw` keyword argument.

[56]:
```
fig, ax = plt.subplots(figsize=(12,6))

# Use linewidth or lw
ax.plot(x, x-1, color="red", linewidth=0.25)
ax.plot(x, x-2, color="red", lw=0.50)
ax.plot(x, x-3, color="red", lw=1)
ax.plot(x, x-4, color="red", lw=10)
```

[56]:
```
[<matplotlib.lines.Line2D at 0x1a3d1a2bad0>]
```

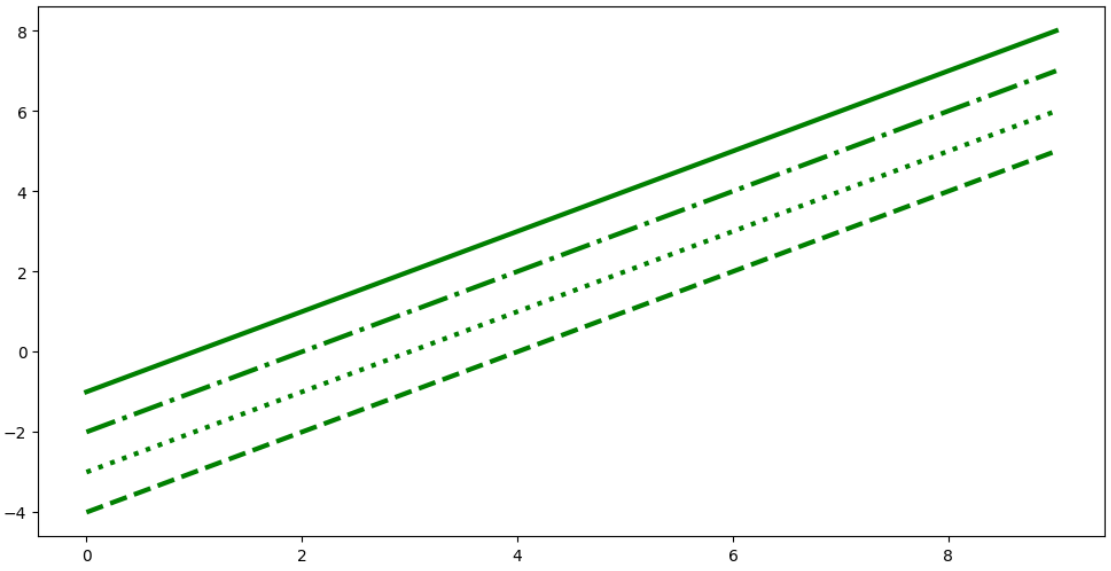


7.7 Linestyles

There are many linestyles to choose from, here is the selection:

[57]:
```
# possible linestype options '--', '-', '-.', ':', 'steps'
fig, ax = plt.subplots(figsize=(12,6))

ax.plot(x, x-1, color="green", lw=3, linestyle='-') # solid
ax.plot(x, x-2, color="green", lw=3, ls='-.') # dash and dot
ax.plot(x, x-3, color="green", lw=3, ls=':') # dots
ax.plot(x, x-4, color="green", lw=3, ls='--') # dashes
```

[57]:
```
[<matplotlib.lines.Line2D at 0x1a3d0390ad0>]
```



7.8 Custom linestyle dash

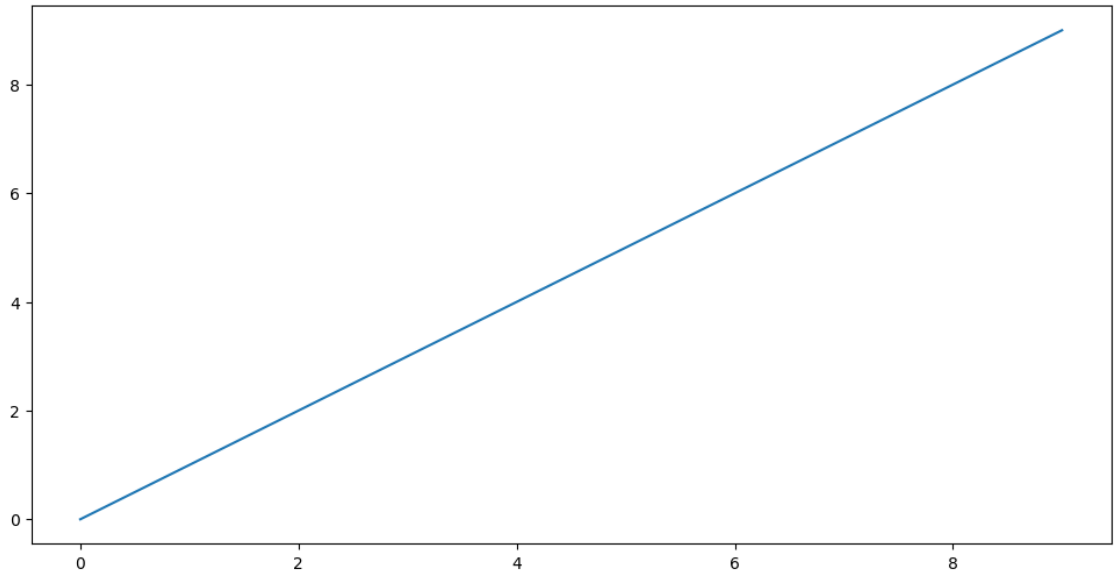
The dash sequence is a sequence of floats of even length describing the length of dashes and spaces in points.

For example, (5, 2, 1, 2) describes a sequence of 5 point and 1 point dashes separated by 2 point spaces.

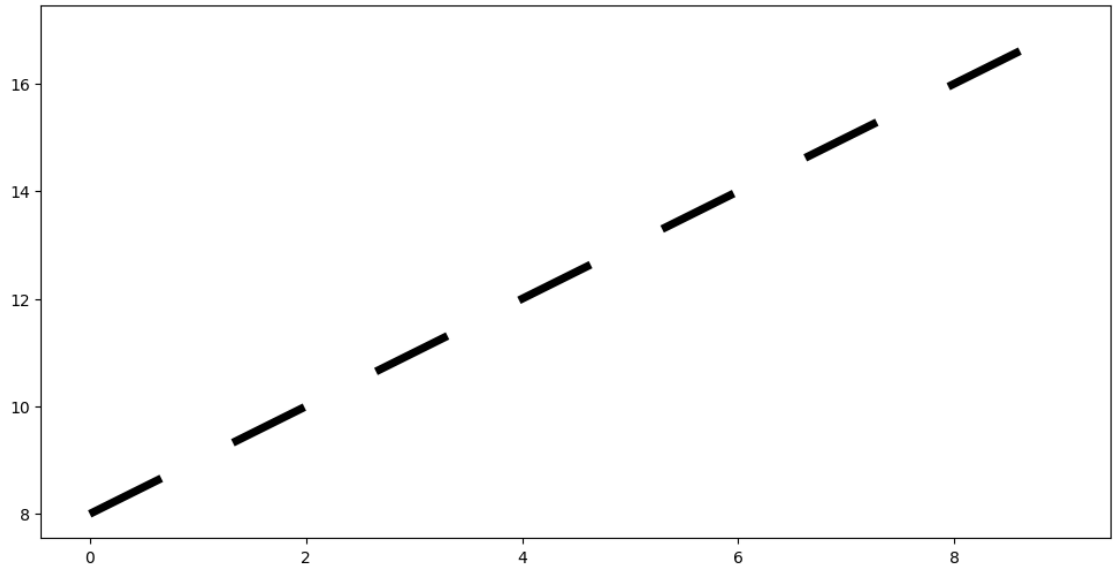
First, we see we can actually "grab" the line from the .plot() command

[58]:
```
fig, ax = plt.subplots(figsize=(12,6))

lines = ax.plot(x,x)

print(type(lines))
```

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

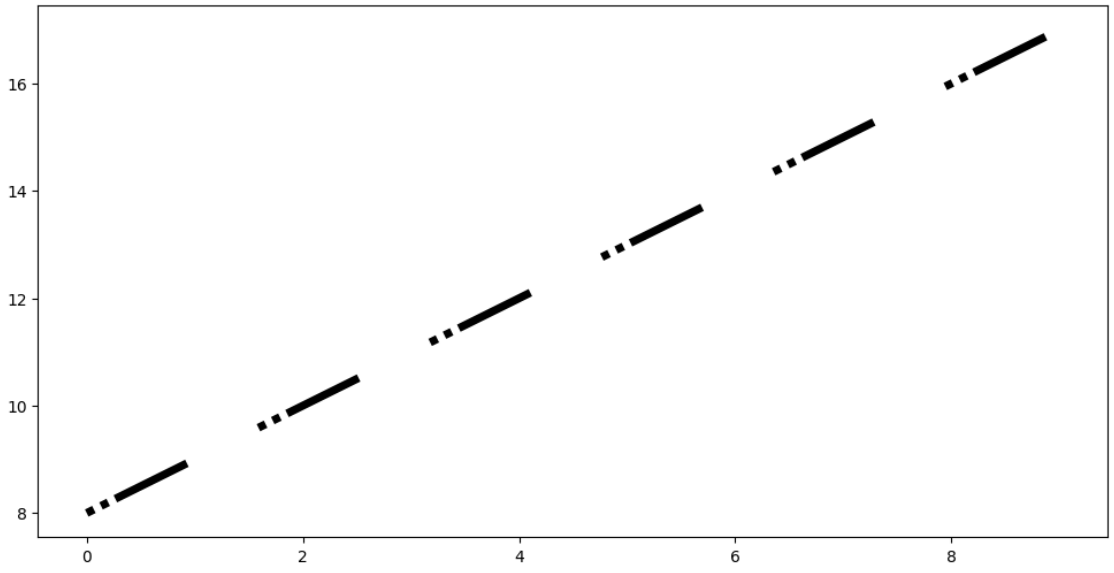


[59]:
```
fig, ax = plt.subplots(figsize=(12,6))
# custom dash
lines = ax.plot(x, x+8, color="black", lw=5)
lines[0].set_dashes([10, 10]) # format: line length, space length
```



[60]:
```
fig, ax = plt.subplots(figsize=(12,6))
# custom dash
lines = ax.plot(x, x+8, color="black", lw=5)
```

```
lines[0].set_dashes([1, 1,1,1,10,10]) # format: line length, space length
```



7.9 Markers

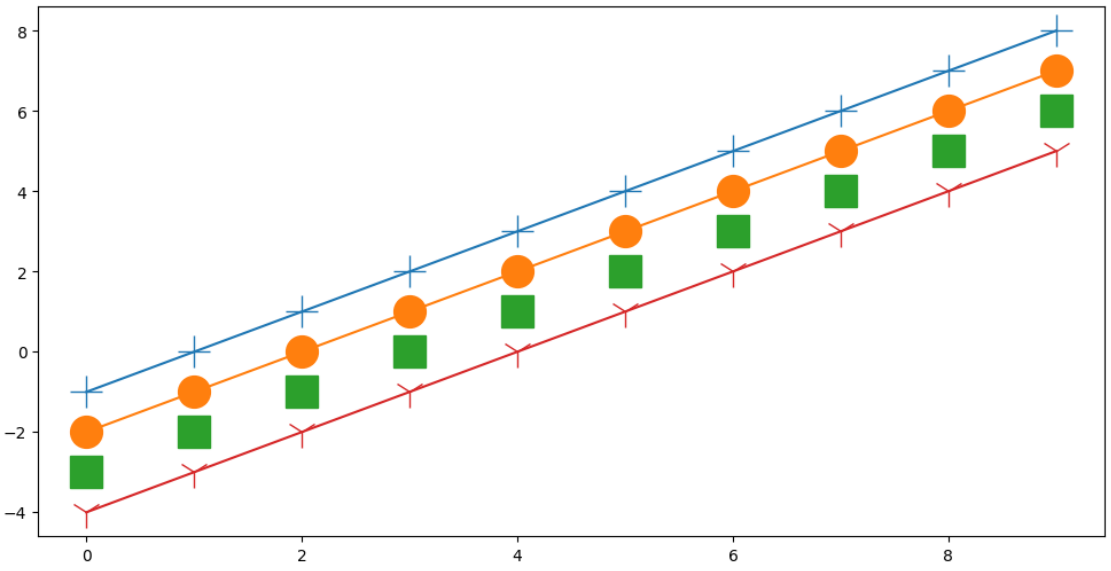
We've technically always been plotting points, and matplotlib has been automatically drawing a line between these points for us. Let's explore how to place markers at each of these points.

7.9.1 Markers Style

Huge list of marker types can be found here: https://matplotlib.org/3.2.2/api/markers_api.html

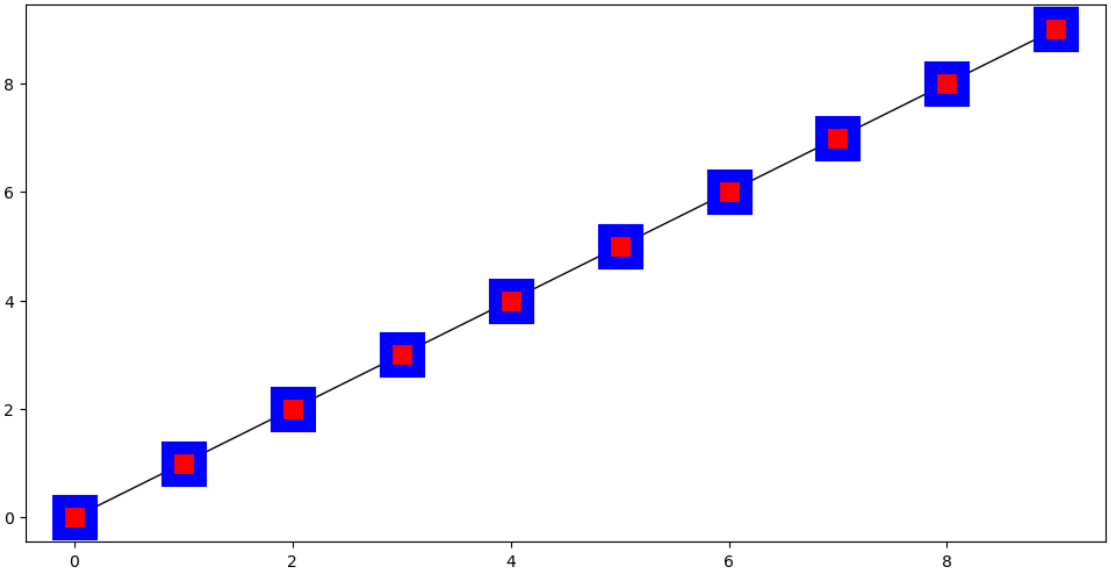
[61]:
```
fig, ax = plt.subplots(figsize=(12,6))

# Use marker for string code
# Use markersize or ms for size

ax.plot(x, x-1,marker='+',markersize=20)
ax.plot(x, x-2,marker='o',ms=20) #ms can be used for markersize
ax.plot(x, x-3,marker='s',ms=20,lw=0) # make linewidth zero to see only markers
ax.plot(x, x-4,marker='1',ms=20)
```

[61]:
```
[<matplotlib.lines.Line2D at 0x1a3ce19a270>]
```



7.10 Custom marker edges, thickness,size,and style

[62]:
```
fig, ax = plt.subplots(figsize=(12,6))

# marker size and color
ax.plot(x, x, color="black", lw=1, ls='-', marker='s', markersize=20,
        markerfacecolor="red", markeredgewidth=8, markeredgecolor="blue");
```

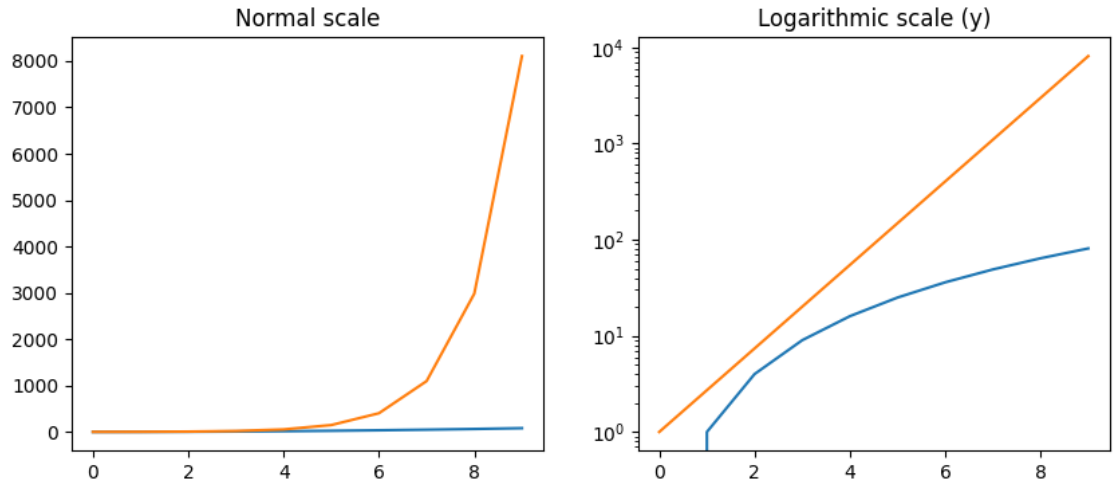


8 Advanced Matplotlib Commands

8.1 Logarithmic scale

It is also possible to set a logarithmic scale for one or both axes. This functionality is in fact only one application of a more general transformation system in Matplotlib. Each of the axes' scales are set seperately using `set_xscale` and `set_yscale` methods which accept one parameter (with the value "log" in this case):

[63]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,4))

axes[0].plot(x, x**2, x, np.exp(x))
axes[0].set_title("Normal scale")

axes[1].plot(x, x**2, x, np.exp(x))
axes[1].set_yscale("log")
axes[1].set_title("Logarithmic scale (y)");
```

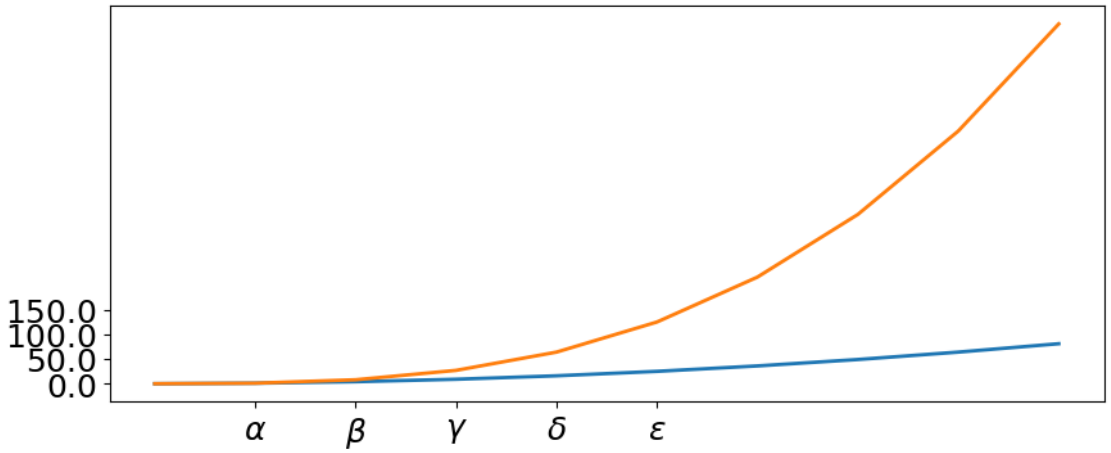


8.2 Placement of ticks and custom tick labels

We can explicitly determine where we want the axis ticks with `set_xticks` and `set_yticks`, which both take a list of values for where on the axis the ticks are to be placed. We can also use the `set_xticklabels` and `set_yticklabels` methods to provide a list of custom text labels for each tick location:

[64]:
```
fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(x, x**2, x, x**3, lw=2)

ax.set_xticks([1, 2, 3, 4, 5])
```

```
ax.set_xticklabels([r'$\alpha$', r'$\beta$', r'$\gamma$', r'$\delta$',␣
 ↪r'$\epsilon$'], fontsize=18)

yticks = [0, 50, 100, 150]
ax.set_yticks(yticks)
ax.set_yticklabels(["$%.1f$" % y for y in yticks], fontsize=18); # use LaTeX␣
 ↪formatted labels
```

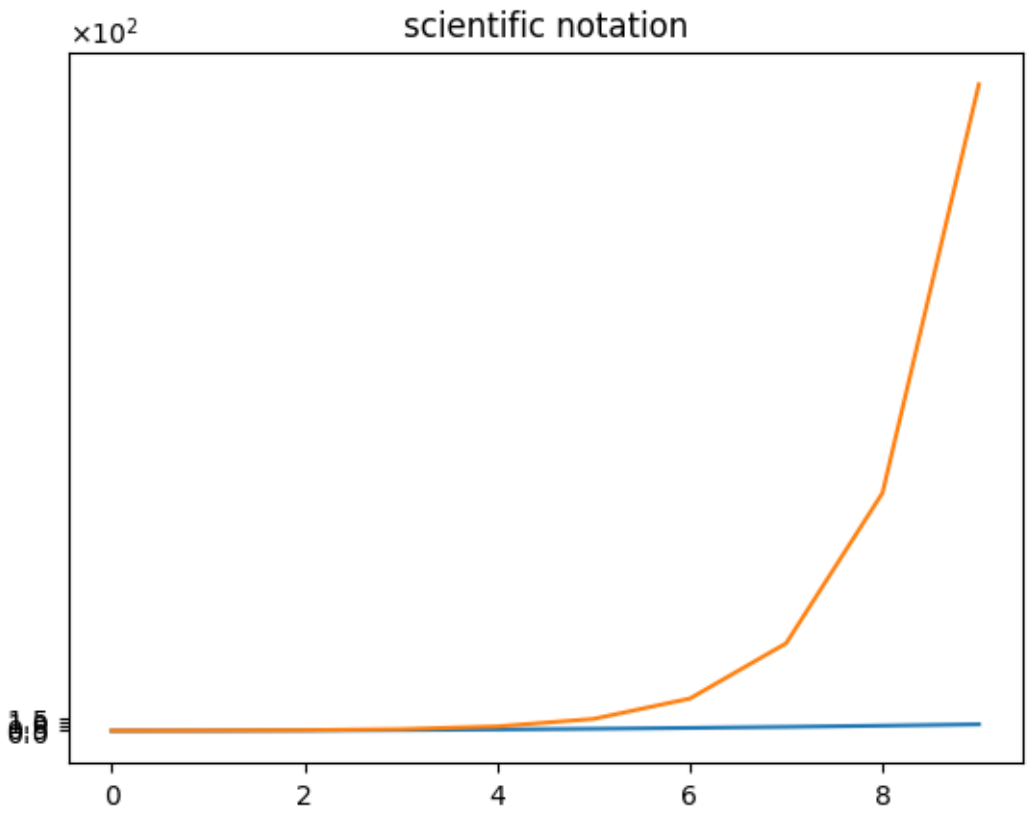


There are a number of more advanced methods for controlling major and minor tick placement in matplotlib figures, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html for details.

8.3 Scientific notation

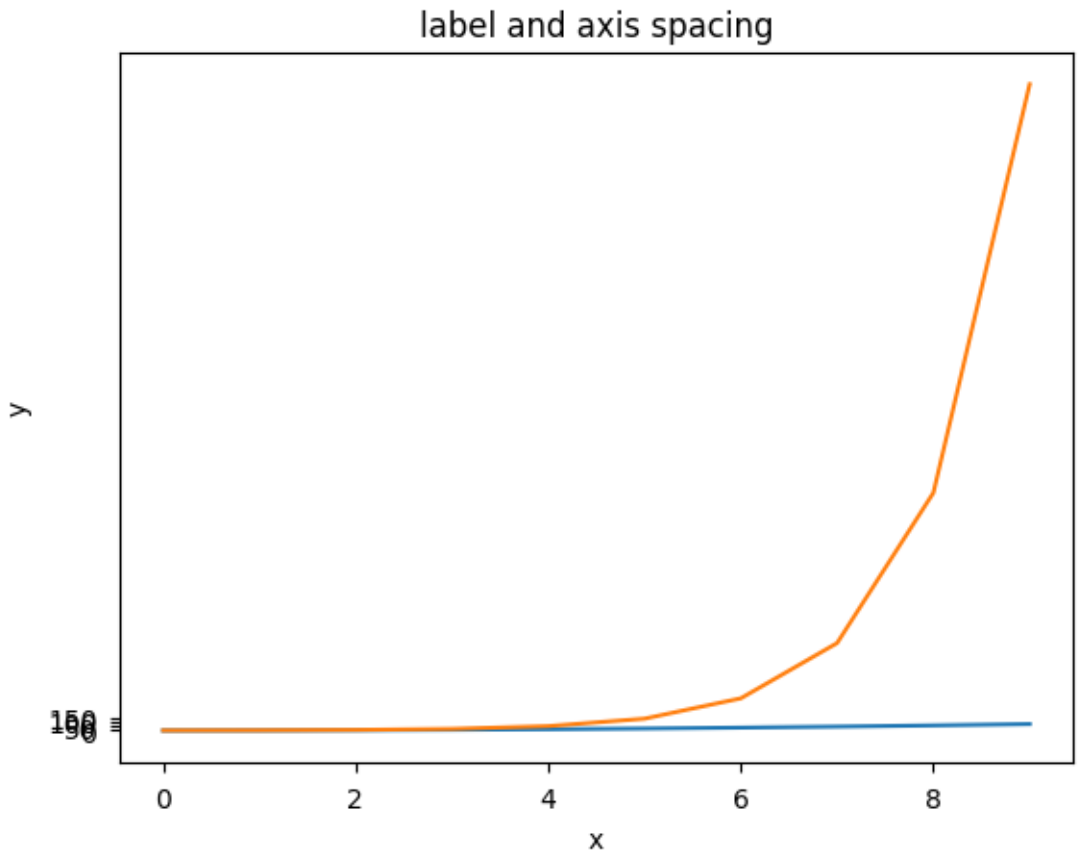
With large numbers on axes, it is often better use scientific notation:

[65]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_title("scientific notation")

ax.set_yticks([0, 50, 100, 150])

from matplotlib import ticker
formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
ax.yaxis.set_major_formatter(formatter)
```



8.4 Axis number and axis label spacing

[67]:
```
import matplotlib
# distance between x and y axis and the numbers on the axes
matplotlib.rcParams['xtick.major.pad'] = 5
matplotlib.rcParams['ytick.major.pad'] = 5

fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("label and axis spacing")

# padding between axis label and axis numbers
ax.xaxis.labelpad = 5
ax.yaxis.labelpad = 5

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

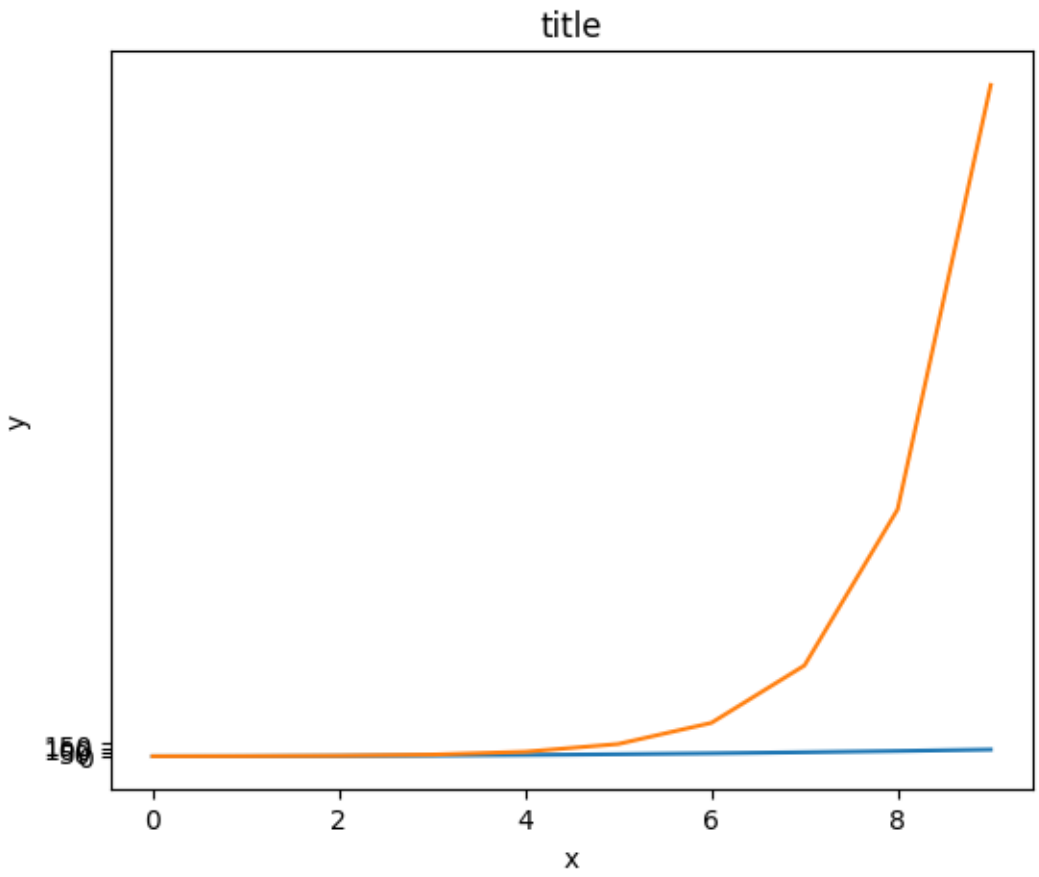


[68]:
```
# restore defaults
matplotlib.rcParams['xtick.major.pad'] = 3
matplotlib.rcParams['ytick.major.pad'] = 3
```

8.5 Axis position adjustments

Unfortunately, when saving figures the labels are sometimes clipped, and it can be necessary to adjust the positions of axes a little bit. This can be done using `subplots_adjust`:

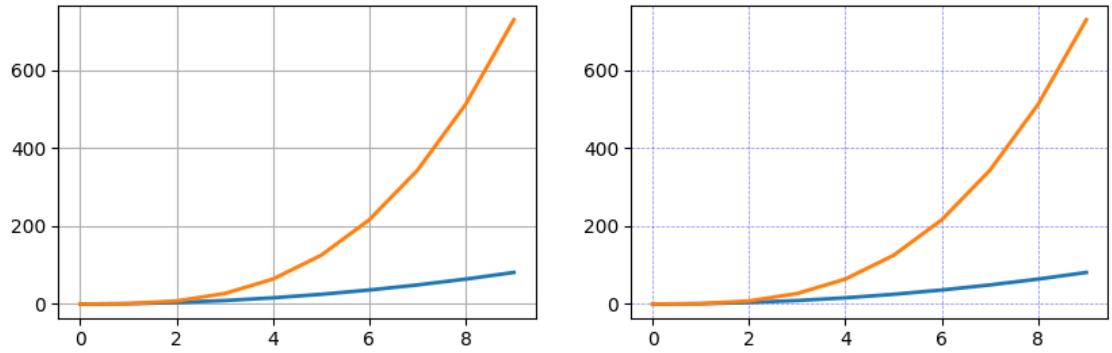
[69]:
```
fig, ax = plt.subplots(1, 1)

ax.plot(x, x**2, x, np.exp(x))
ax.set_yticks([0, 50, 100, 150])

ax.set_title("title")
ax.set_xlabel("x")
ax.set_ylabel("y")

fig.subplots_adjust(left=0.15, right=.9, bottom=0.1, top=0.9);
```



8.6 Axis grid

With the `grid` method in the axis object, we can turn on and off grid lines. We can also customize the appearance of the grid lines using the same keyword arguments as the `plot` function:

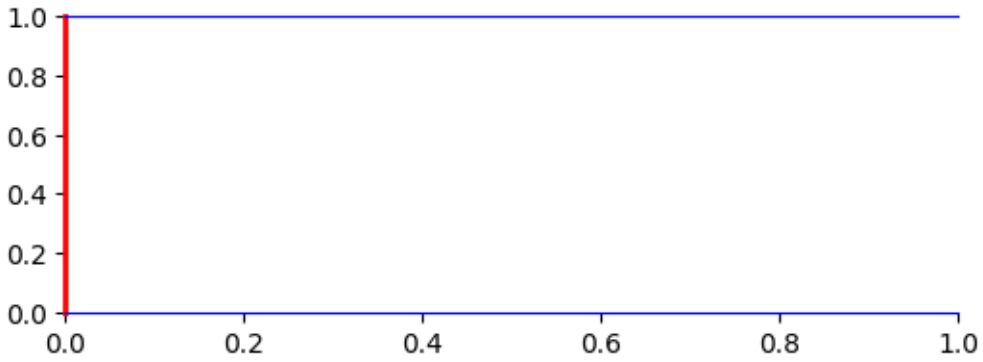
[70]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)

# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
```



8.7 Axis spines

We can also change the properties of axis spines:

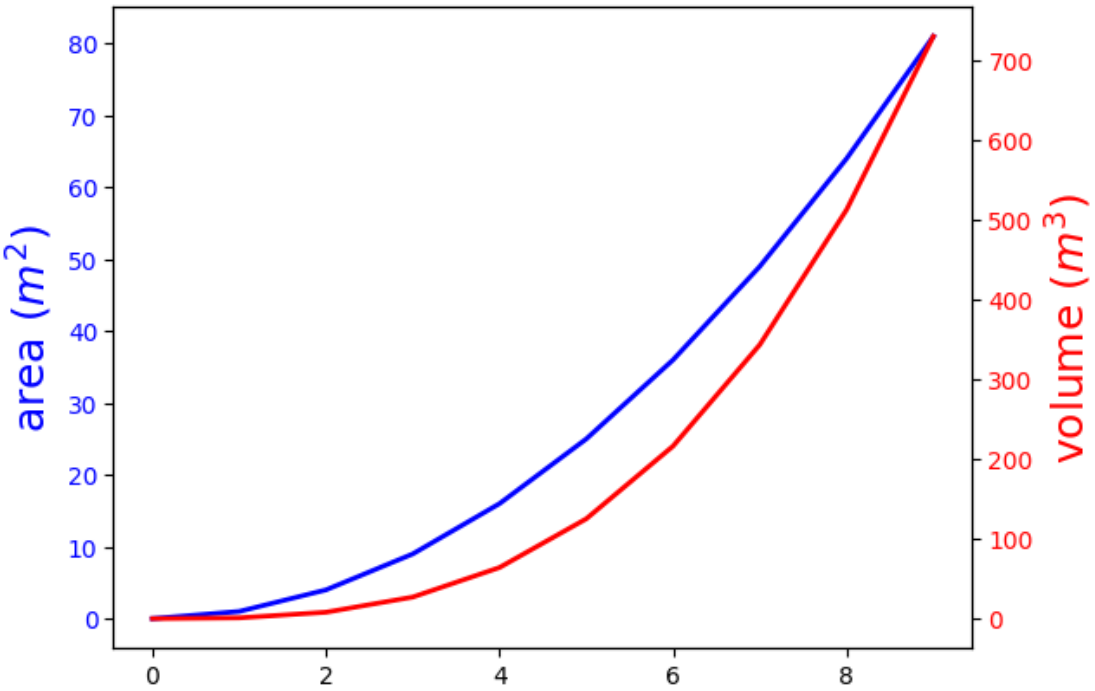
[71]:
```
fig, ax = plt.subplots(figsize=(6,2))

ax.spines['bottom'].set_color('blue')
ax.spines['top'].set_color('blue')

ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)

# turn off axis spine to the right
ax.spines['right'].set_color("none")
ax.yaxis.tick_left() # only ticks on the left side
```



8.8 Twin axes

Sometimes it is useful to have dual x or y axes in a figure; for example, when plotting curves with different units together. Matplotlib supports this with the `twinx` and `twiny` functions:

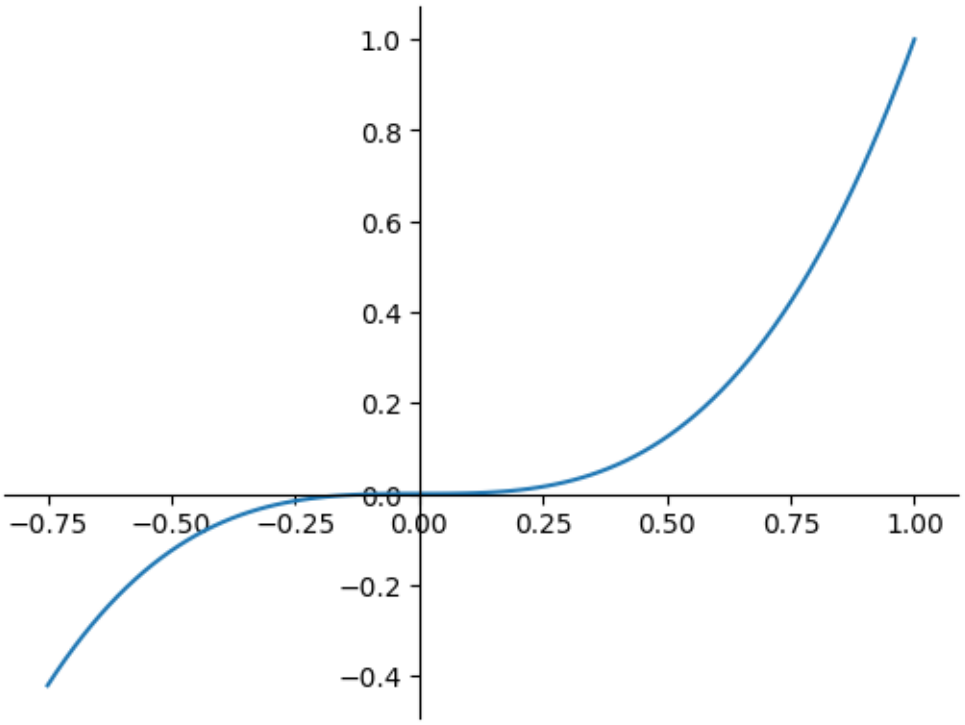
[72]:
```
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")
```



8.9 Axes where x and y is zero

[73]:
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
```

```
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```



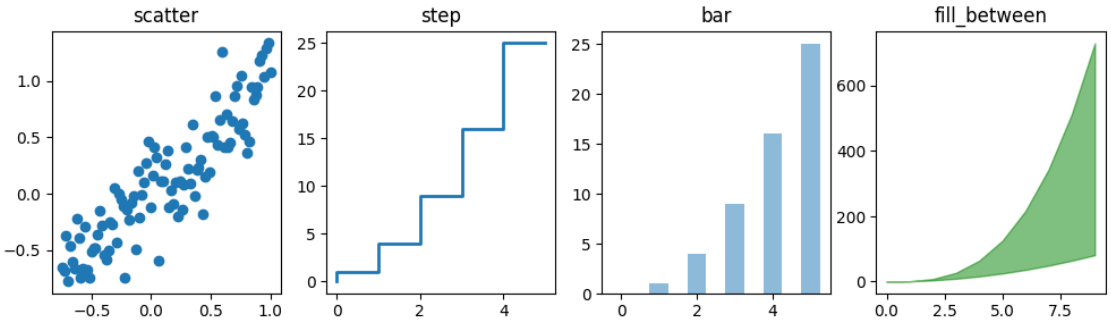
8.10 Other 2D plot styles

In addition to the regular `plot` method, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of available plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

[74]:
```
n = np.array([0,1,2,3,4,5])
```

[75]:
```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*np.random.randn(len(xx)))
axes[0].set_title("scatter")

axes[1].step(n, n**2, lw=2)
axes[1].set_title("step")

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
axes[2].set_title("bar")
```

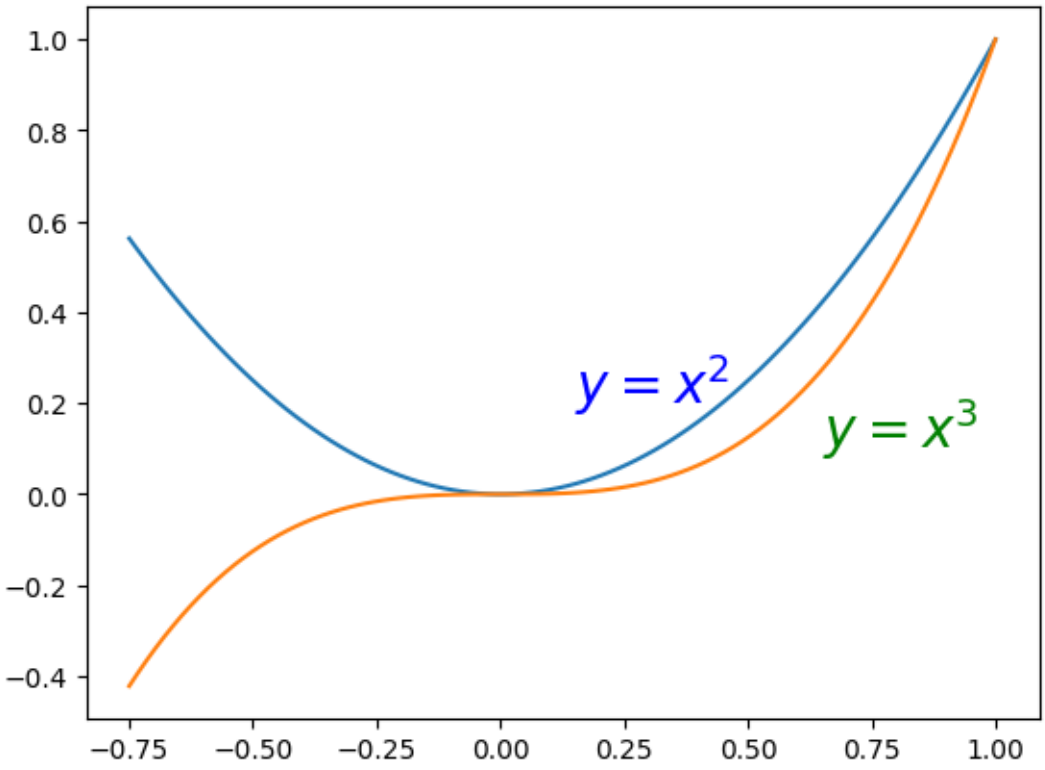
```
axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
axes[3].set_title("fill_between");
```



8.11 Text annotation

Annotating text in matplotlib figures can be done using the `text` function. It supports LaTeX formatting just like axis label texts and titles:

[76]:
```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)

ax.text(0.15, 0.2, r"$y=x^2$", fontsize=20, color="blue")
ax.text(0.65, 0.1, r"$y=x^3$", fontsize=20, color="green");
```

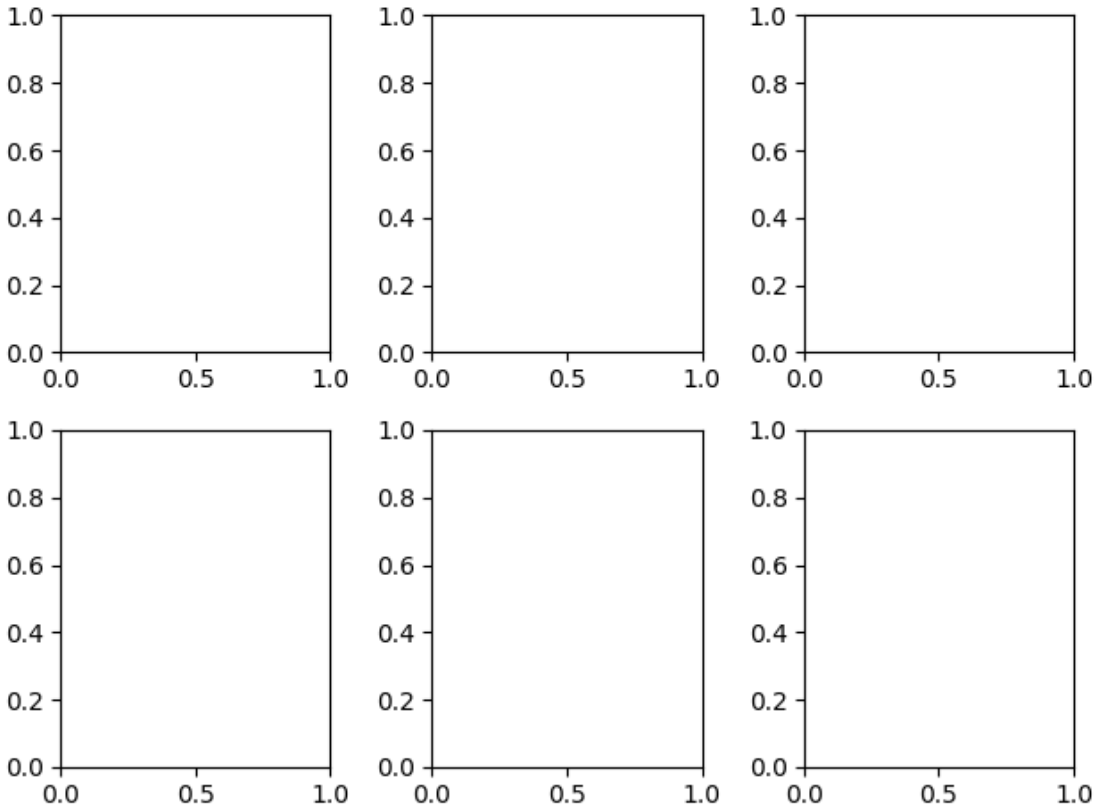


8.12 Figures with multiple subplots and insets

Axes can be added to a matplotlib Figure canvas manually using `fig.add_axes` or using a sub-figure layout manager such as `subplots`, `subplot2grid`, or `gridspec`:

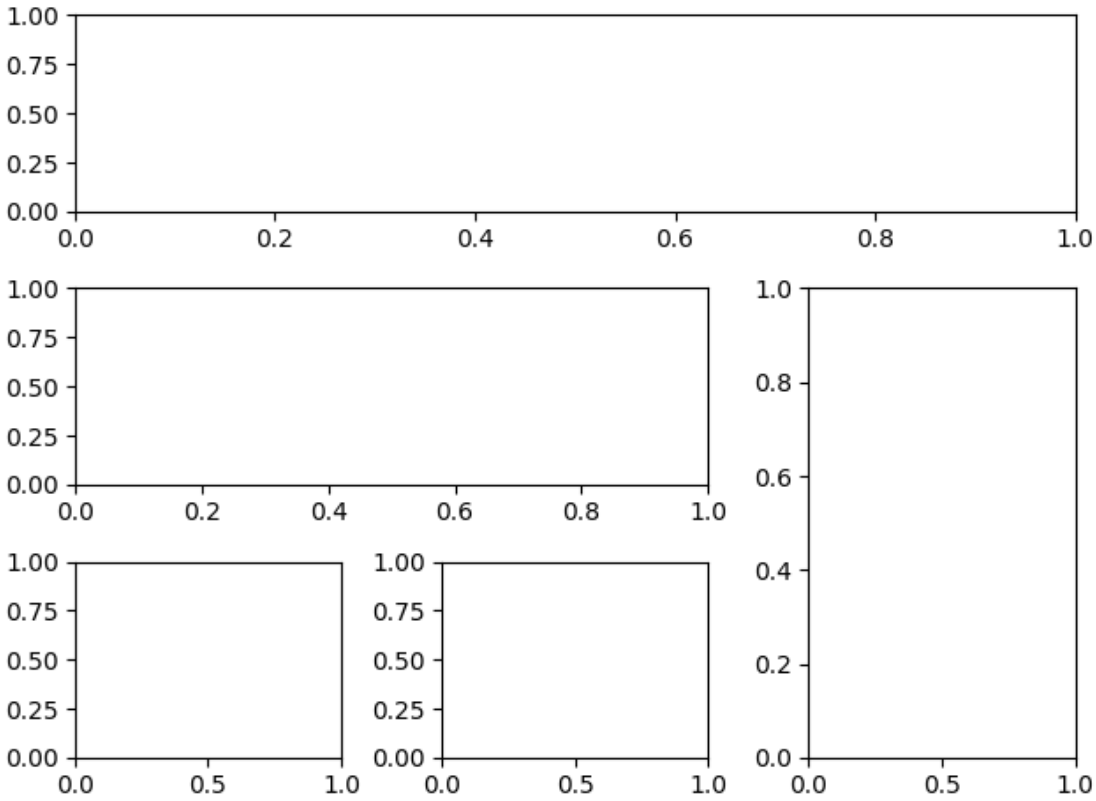
8.12.1 subplots

[77]:
```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```



8.13 subplot2grid

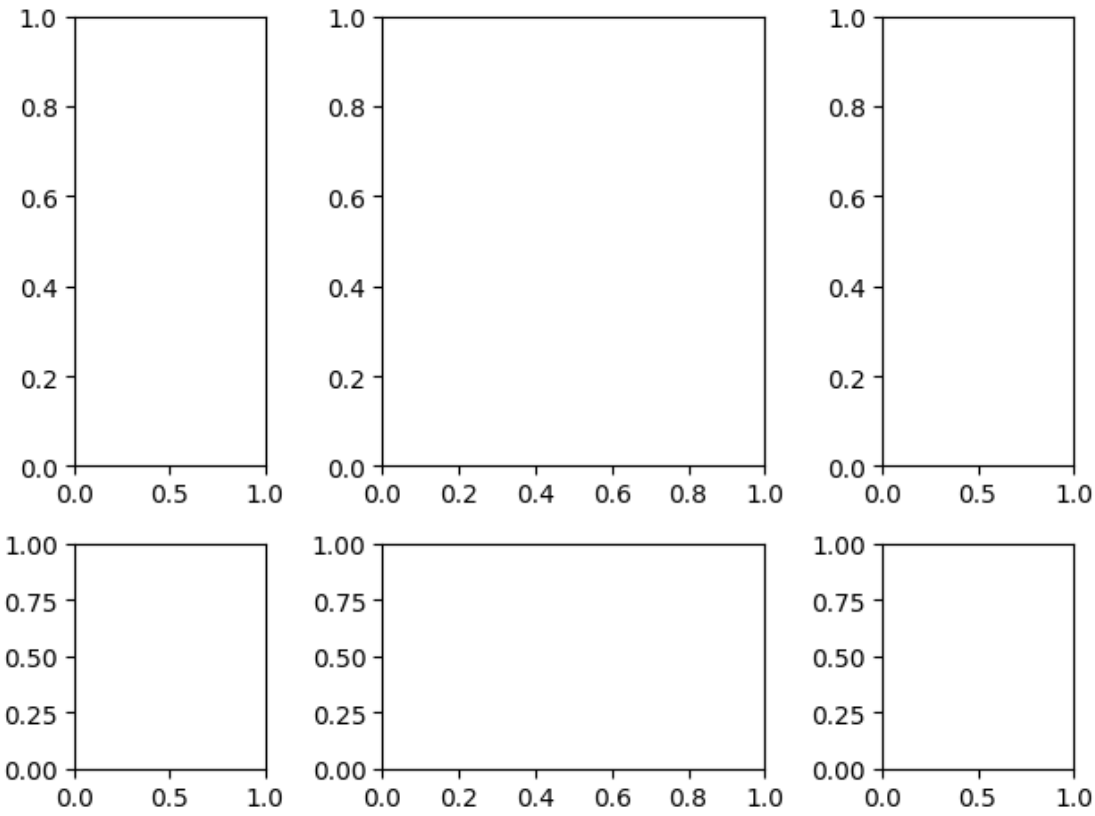
[78]:
```
fig = plt.figure()
ax1 = plt.subplot2grid((3,3), (0,0), colspan=3)
ax2 = plt.subplot2grid((3,3), (1,0), colspan=2)
ax3 = plt.subplot2grid((3,3), (1,2), rowspan=2)
ax4 = plt.subplot2grid((3,3), (2,0))
ax5 = plt.subplot2grid((3,3), (2,1))
fig.tight_layout()
```



8.14 gridspec

[79]:
```
import matplotlib.gridspec as gridspec
```

[80]:
```
fig = plt.figure()

gs = gridspec.GridSpec(2, 3, height_ratios=[2,1], width_ratios=[1,2,1])
for g in gs:
    ax = fig.add_subplot(g)

fig.tight_layout()
```

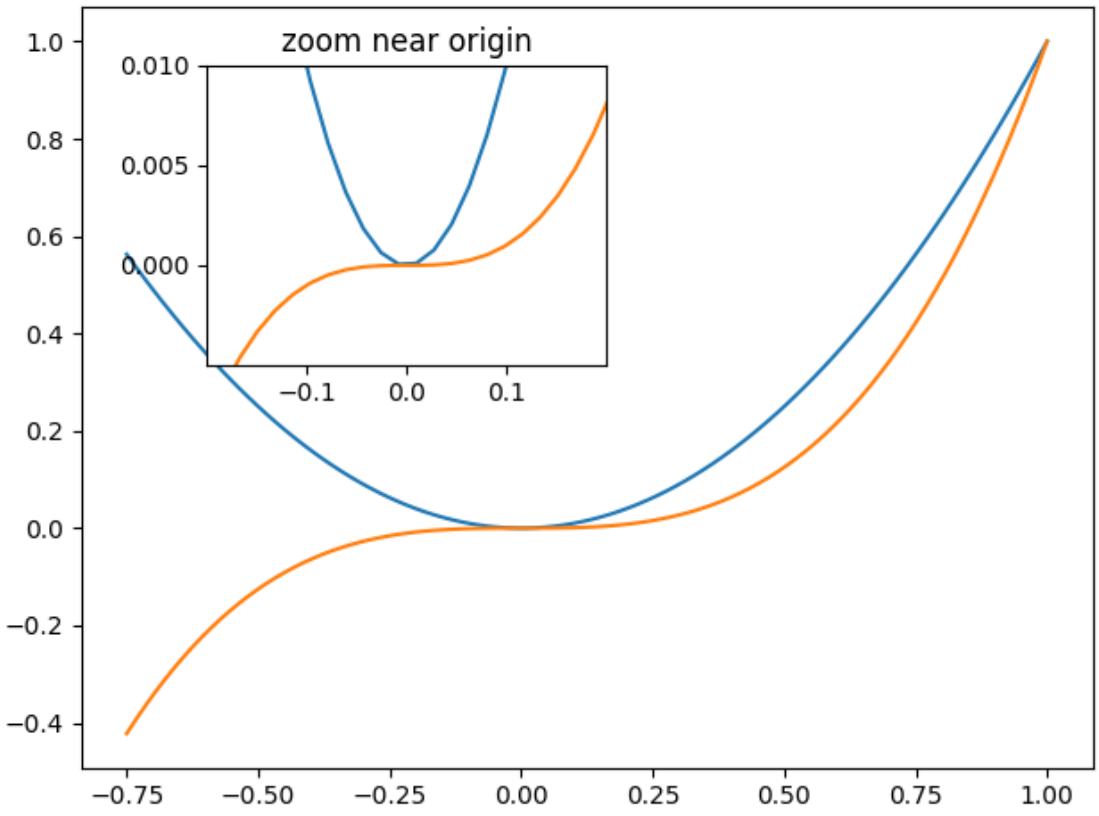


8.15 add_axes

Manually adding axes with `add_axes` is useful for adding insets to figures:

[81]:
```
fig, ax = plt.subplots()

ax.plot(xx, xx**2, xx, xx**3)
fig.tight_layout()

# inset
inset_ax = fig.add_axes([0.2, 0.55, 0.35, 0.35]) # X, Y, width, height

inset_ax.plot(xx, xx**2, xx, xx**3)
inset_ax.set_title('zoom near origin')

# set axis range
inset_ax.set_xlim(-.2, .2)
inset_ax.set_ylim(-.005, .01)

# set axis tick locations
inset_ax.set_yticks([0, 0.005, 0.01])
```

```
inset_ax.set_xticks([-0.1,0,.1]);
```



8.16 Colormap and contour figures

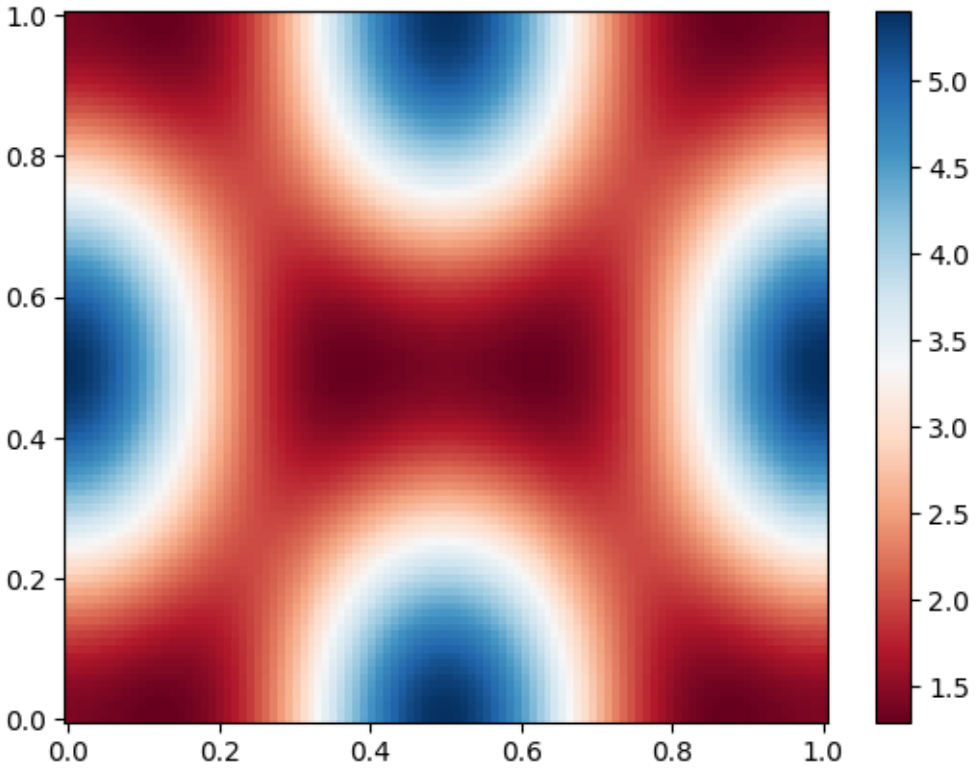
Colormaps and contour figures are useful for plotting functions of two variables. In most of these functions we will use a colormap to encode one dimension of the data. There are a number of predefined colormaps. It is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see: http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

[82]:
```
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p) * np.cos(phi_m) - alpha * np.
 ↪cos(phi_ext - 2*phi_p)
```

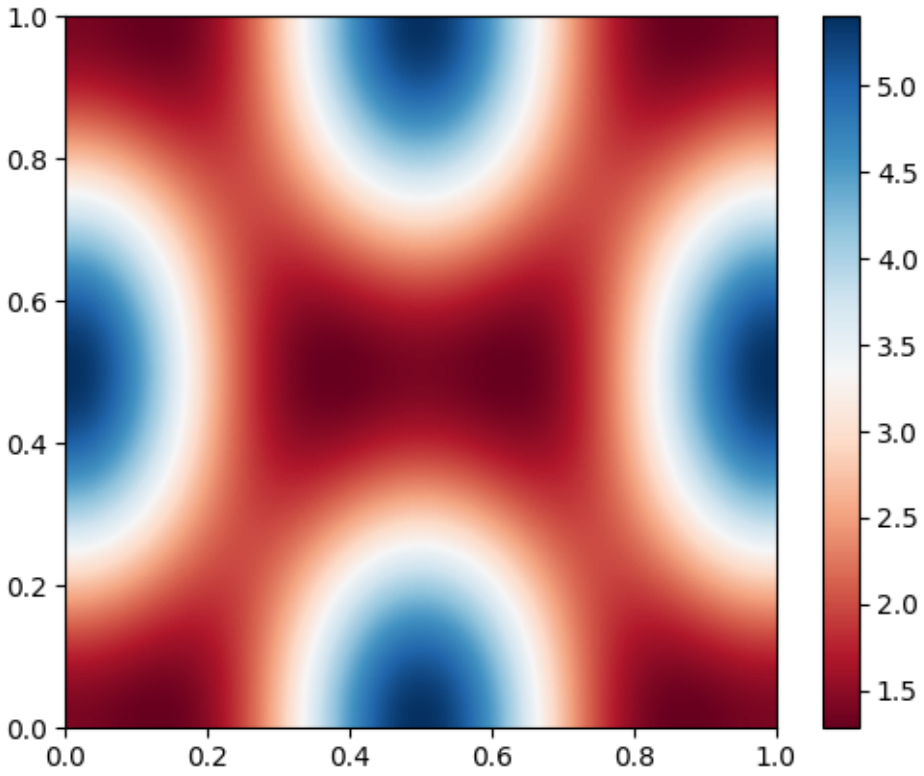
[83]:
```
phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

8.17 pcolor

[84]:
```
fig, ax = plt.subplots()

p = ax.pcolor(X/(2*np.pi), Y/(2*np.pi), Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).
  ↪min(), vmax=abs(Z).max())
cb = fig.colorbar(p, ax=ax)
```

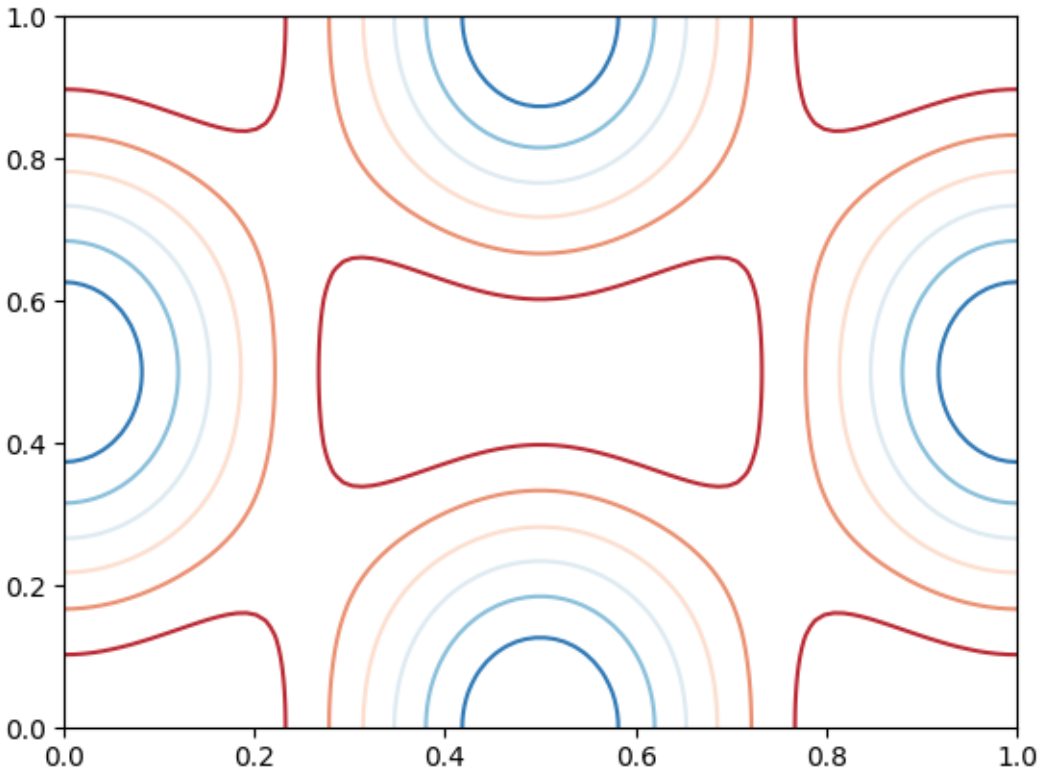


[85]:
```
fig, ax = plt.subplots()

im = ax.imshow(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).
  ↪max(), extent=[0, 1, 0, 1])
im.set_interpolation('bilinear')

cb = fig.colorbar(im, ax=ax)
```



8.18 contour

[86]:
```
fig, ax = plt.subplots()

cnt = ax.contour(Z, cmap=matplotlib.cm.RdBu, vmin=abs(Z).min(), vmax=abs(Z).
↪max(), extent=[0, 1, 0, 1])
```



8.19 3D figures

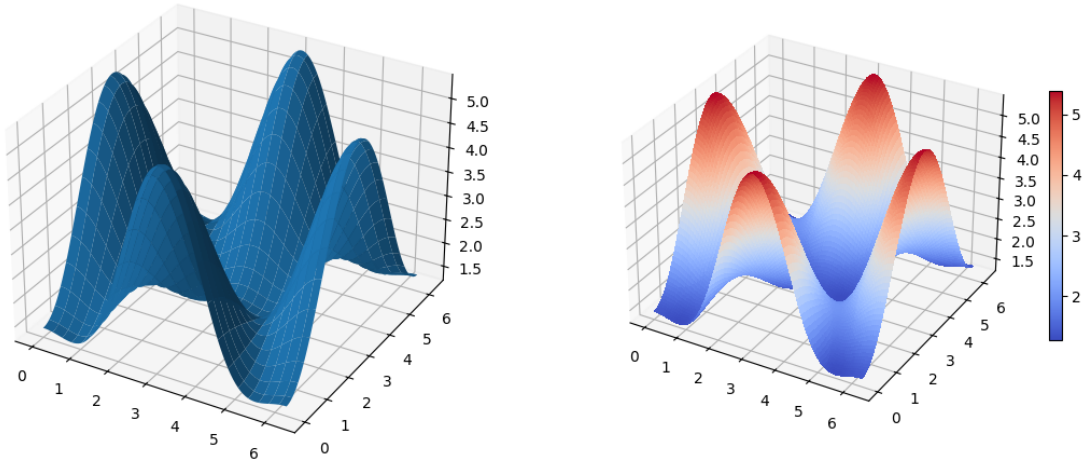
To use 3D graphics in matplotlib, we first need to create an instance of the `Axes3D` class. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes; or, more conveniently, by passing a `projection='3d'` keyword argument to the `add_axes` or `add_subplot` methods.

[87]:
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

8.19.1 Surface plots

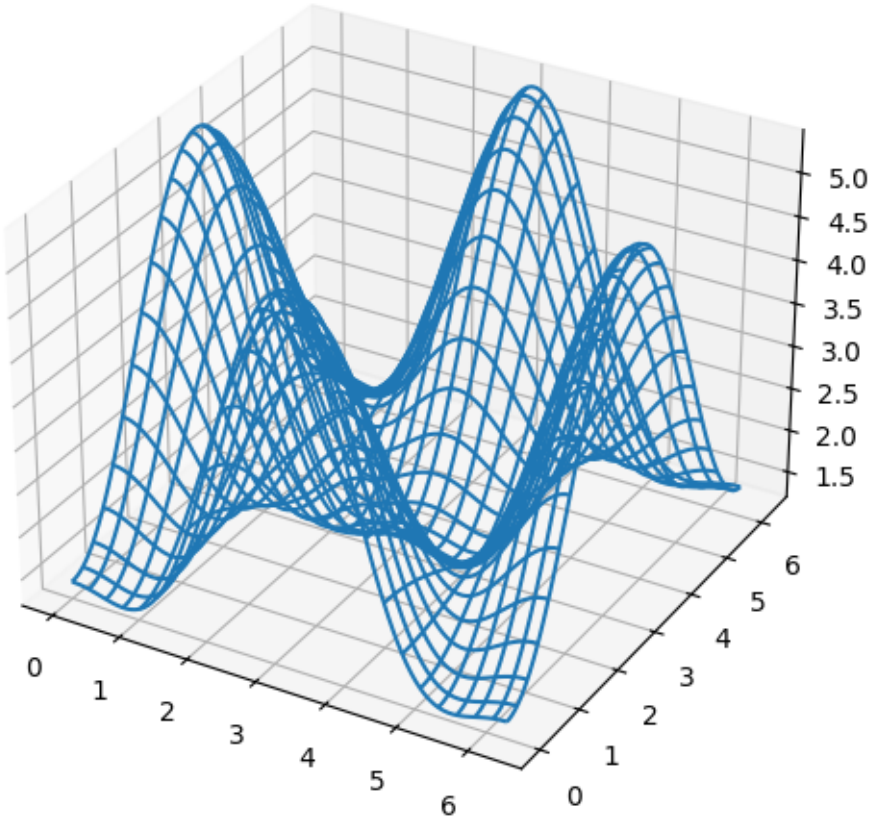
[88]:
```
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance because of the projection='3d' keyword␣
 ↪argument to add_subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
```

```
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=matplotlib.cm.coolwarm,␣
 ↪linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```



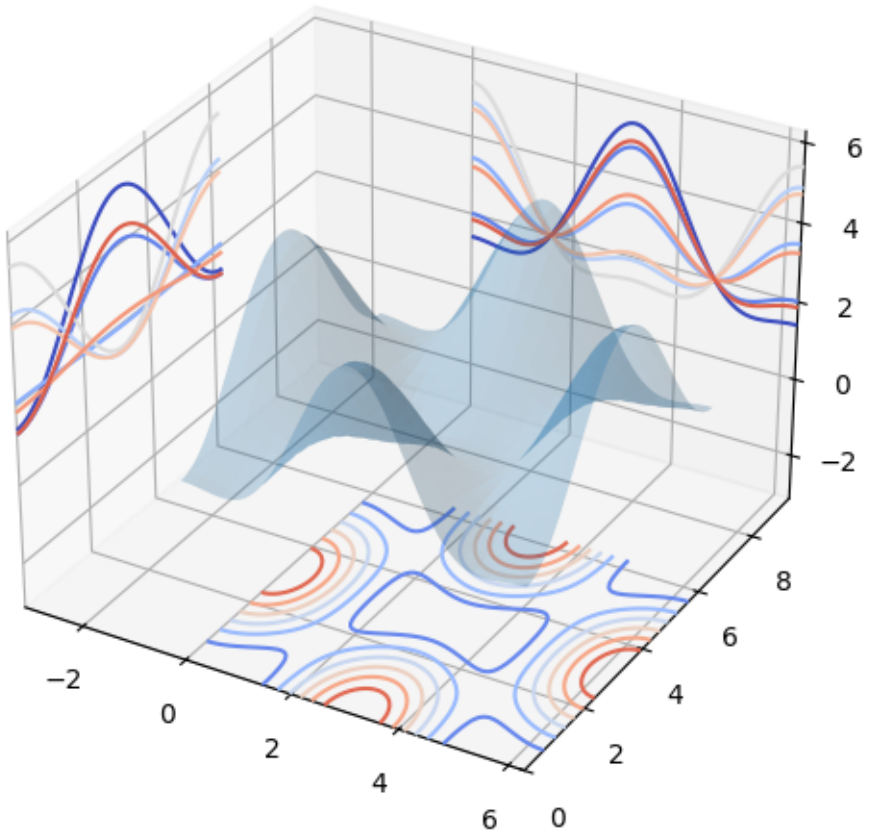
Wire-frame plot

[89]:
```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1, 1, 1, projection='3d')

p = ax.plot_wireframe(X, Y, Z, rstride=4, cstride=4)
```



8.19.2 Coutour plots with projections

[90]:
```
fig = plt.figure(figsize=(8,6))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-np.pi, cmap=matplotlib.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*np.pi, cmap=matplotlib.cm.
 ↪coolwarm)

ax.set_xlim3d(-np.pi, 2*np.pi);
ax.set_ylim3d(0, 3*np.pi);
ax.set_zlim3d(-np.pi, 2*np.pi);
```



9 Happy Programming!