DSiP-5-Matplotlib-Course_New (1)

September 28, 2023

1 Scientific Programming in Python (SPiP) - 2D and 3D plotting with Matplotlib

[1]:
```
# ipython magick command:
# This line loads matplotlib package for ipython notebook
# and configures matplotlib to show figures embedded in the notebook,
# instead of poping up a new window.
%matplotlib inline
```

1.1 1. Introduction

Matplotlib is a 2D and 3D graphics library for generating scientific figures.

Figures are controlled *programmatically*, i.e. you can script it, ensure reproducibility and re-use.

Advantages

- Easy to get started
- Support for LaTeX formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

More information at the Matplotlib web page: http://matplotlib.org/

1.2 1.1. Getting Started

In IPython notebook via magic commands

[2]:
```
# ipython magic command:
# This line loads matplotlib package for ipython notebook
# and configures matplotlib to show figures embedded in the notebook,
# instead of poping up a new window.
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

In Python modules via import of pylab (which also imports other modules like scipy)

[3]:
```
from pylab import *
```

or only mathplotlib plot via:

[4]:
```
import matplotlib.pyplot as plt
```

1.3 2. The Plotting API

1.3.1 2.1. Using mathplotlib in global namespace

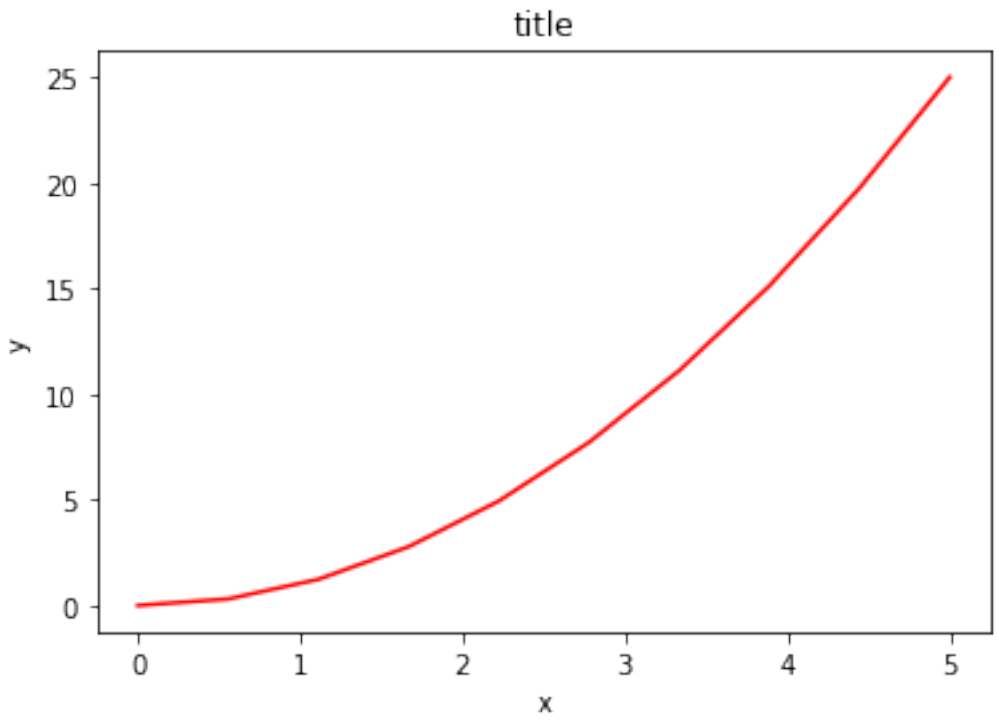
- The easiest way to get started with plotting using matplotlib is often to use the MATLAB-like API provided by matplotlib.
- It is designed to compatible with MATLAB's plotting functions, so it is easy to get started with if you are familiar with MATLAB.
- It maintains a global object against which all painting commands are done. The global object can be switched with the `figure()` function
- To use this API from matplotlib, we need to include the symbols in the `pylab` module (imports all functions in global namespace):

[5]:
```
from pylab import *
```

Example - plotting a simple figure

[6]:
```
# the function to plot
x = linspace(0, 5, 10) # create a numpy array going from 0 to 5 with step size c
y = x ** 2
```

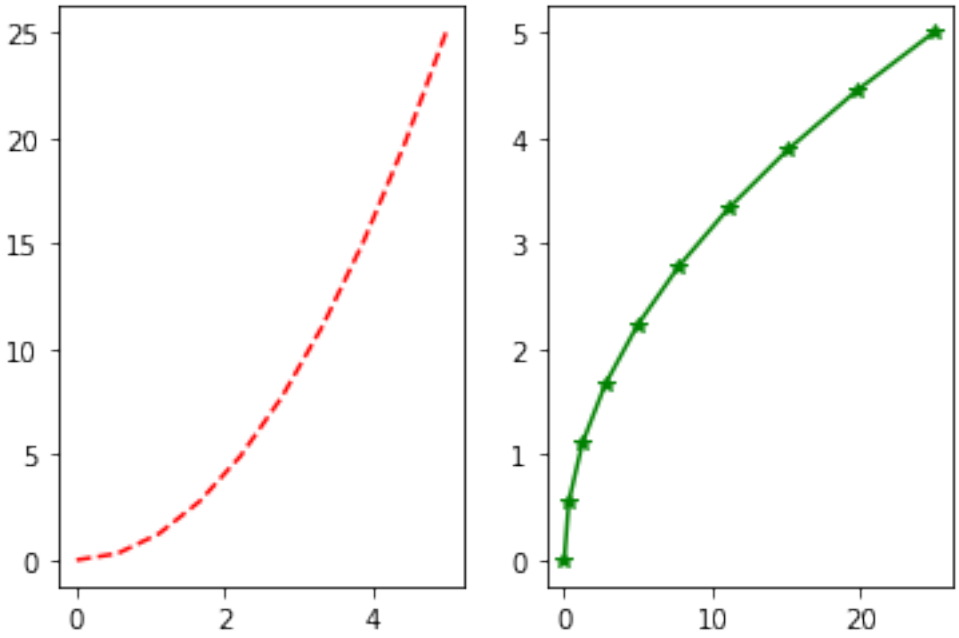
[7]:
```
figure() #intantiates a new figure object (= canvas for drawing)
plot(x, y, 'r') # plot points and connect them as red lines
xlabel('x') # set the x label
ylabel('y') # set the y label
title('title') #set the title of the figure
show() # now draw the figure
```



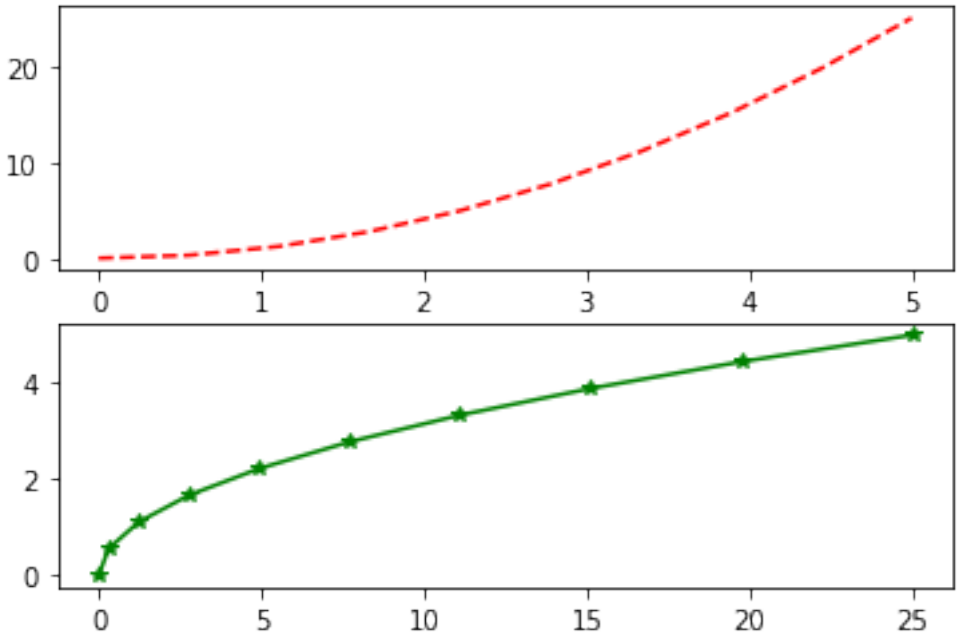
Example 2 - creating subplots

- The plotting canvas can be split into subplots using the `subplot` command
- 'subplot(nrows, ncols, plot_number)"
  - `nrows` - number of rows in the plot figure
  - `ncols` - number of cols in the plot figure
  - `plot_number`- the plot which should be activated
    - *plot_number* starts at 1, increments across rows first an has a maximum of `nrows * ncols`.
- see `help(subplot)` for details

[8]:
```
subplot(1,2,1) # 1 row, 2 columns. select first subplot
plot(x, y, 'r--') #plot in red dashed lines
subplot(1,2,2) # 1 row, 2 columns. select second subplot
plot(y, x, 'g*-'); #plot in green line with marking points
```



[9]:
```
subplot(2,1,1) # 1 row, 2 columns. select first subplot
plot(x, y, 'r--') #plot in red dashed lines
subplot(2,1,2) # 1 row, 2 columns. select second subplot
plot(y, x, 'g*-'); #plot in green line with marking points
```



Comments on MATLAB-style API **Pro** * Similar to MATLAB, so easy to get started by those who know the API * Minimum coding overhead * Nice in interactive explorations

Contra * Keep track of the state of the figure in your head * Can become complex for larger figures * No ojbect oriented view

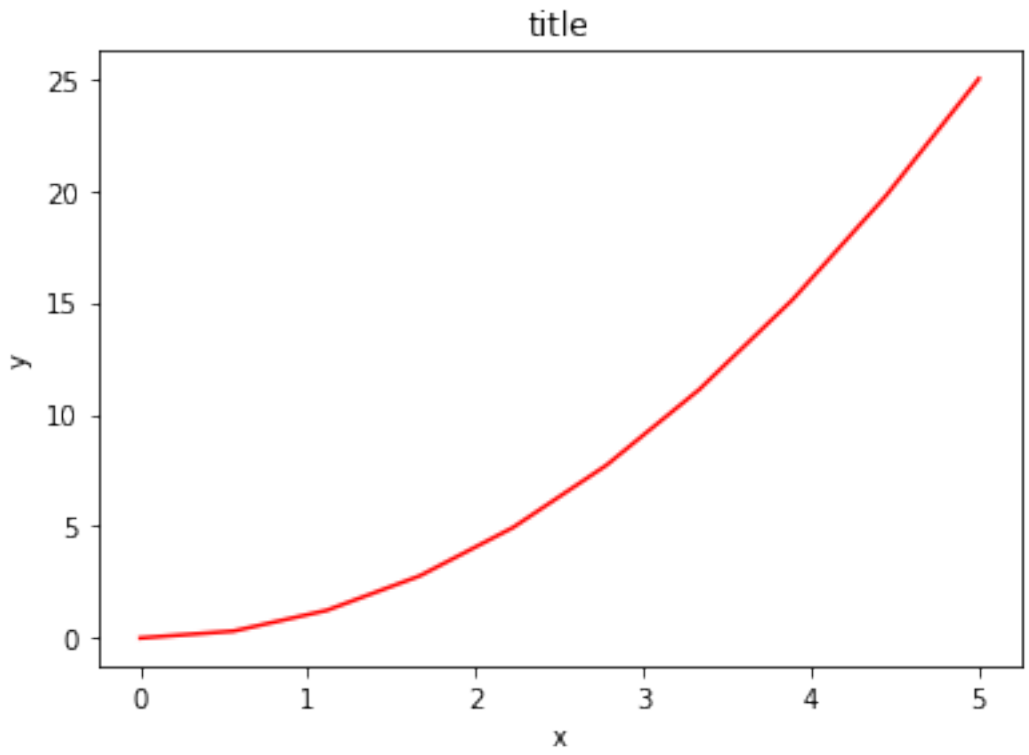
1.3.2 2.2 Using mathplotlib via objects

Design Decision

- Every figure becomes an object
- Invoke commands against the ojbect to alter state/draw items

Approach * You start by creating a figure object (instance of `Figure` class) * A figure consists of `axes`, where new axes can be added via the `add_axes` method in the `Figure` class

[10]:
```
import matplotlib.pyplot as plt # import the plot object as factory for plots
fig = plt.figure() #create new figure object

axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # left, bottom, width, height (range␣
↪0 to 1)

axes.plot(x, y, 'r') # plot red line

axes.set_xlabel('x') # set xlabel
axes.set_ylabel('y') # set ylabel
axes.set_title('title'); # set title
```

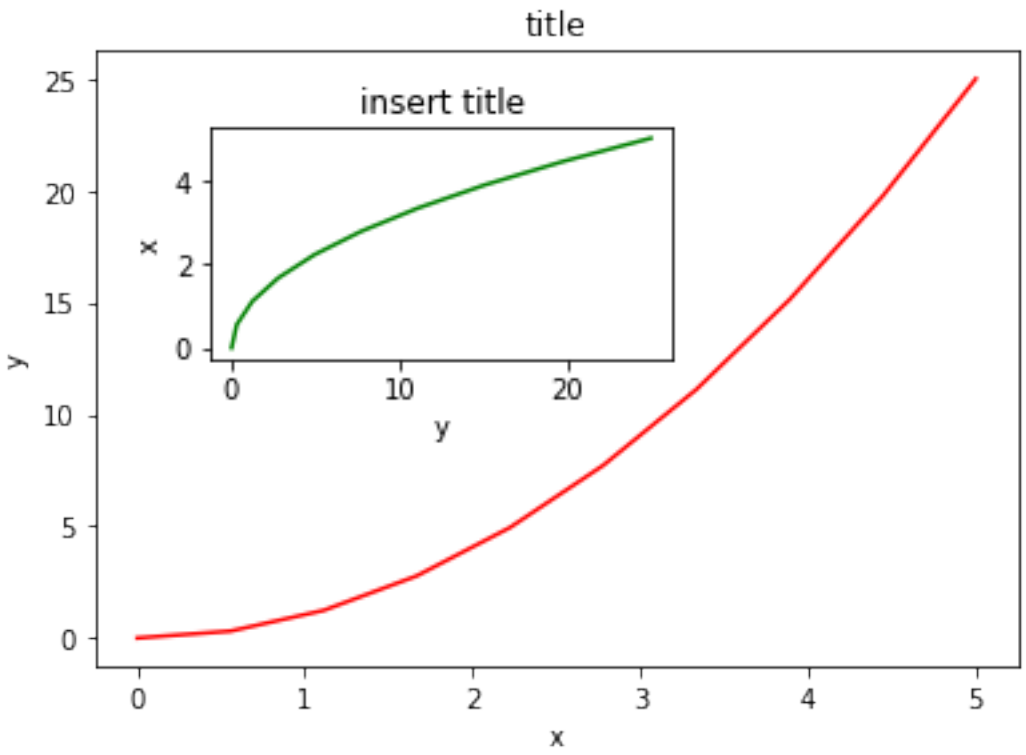


Although a little bit more code is involved, the advantage is that we now have full control of where the plot axes are place, and we can easily add more than one axis to the figure.

Adding axes

You can add axes (=sub plots) anywhere by specifying their bounding box `[left, bottom, width, height]`

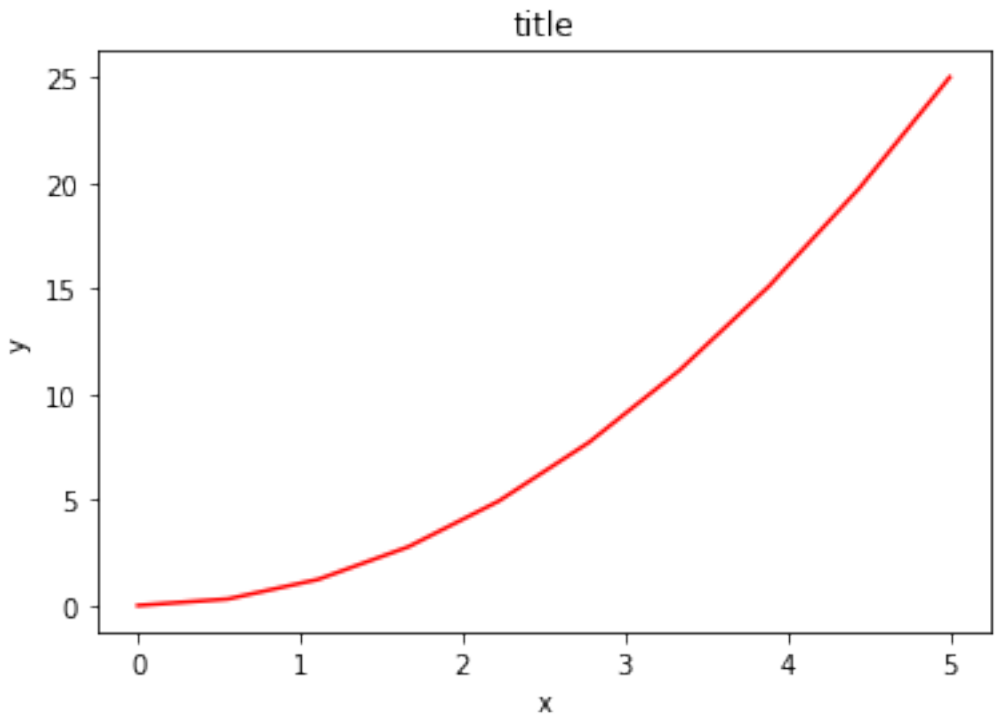
[11]:
```
fig = plt.figure()

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

# main figure
axes1.plot(x, y, 'r')
axes1.set_xlabel('x')
axes1.set_ylabel('y')
axes1.set_title('title')

# insert
axes2.plot(y, x, 'g')
axes2.set_xlabel('y')
axes2.set_ylabel('x')
axes2.set_title('insert title');
```



** Default Axes**

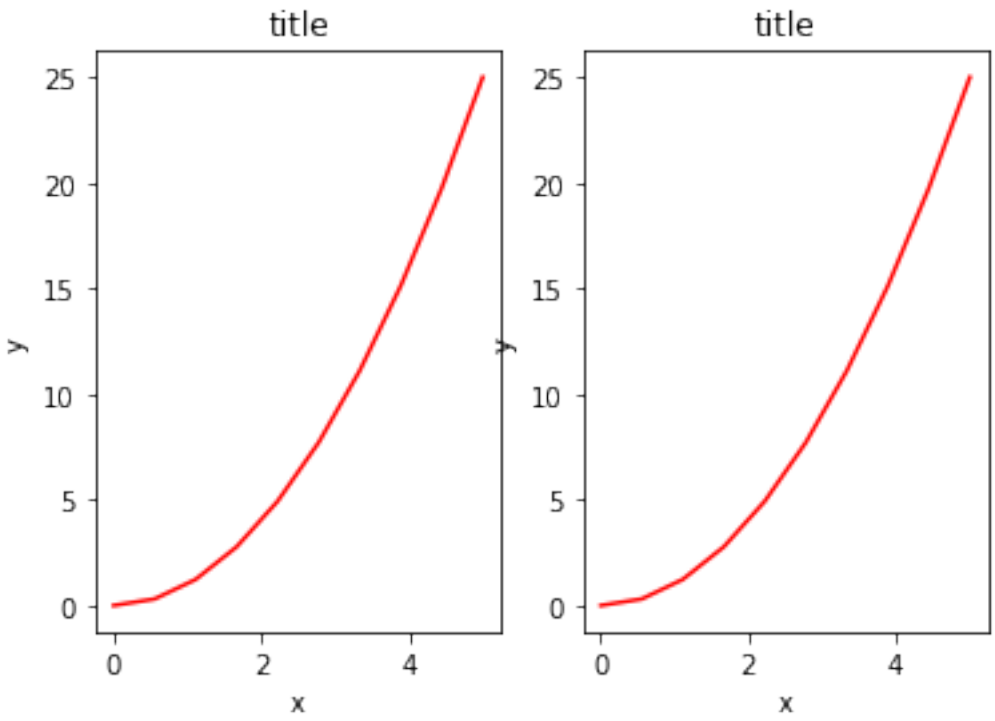
If we don't care to be explicit about where our plot axes are placed in the figure canvas, then we can use one of the many axis layout managers in matplotlib. The most used one is `subplots`, which can be used like this:

[12]:
```
fig, axes = plt.subplots()

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



Creating Subplots

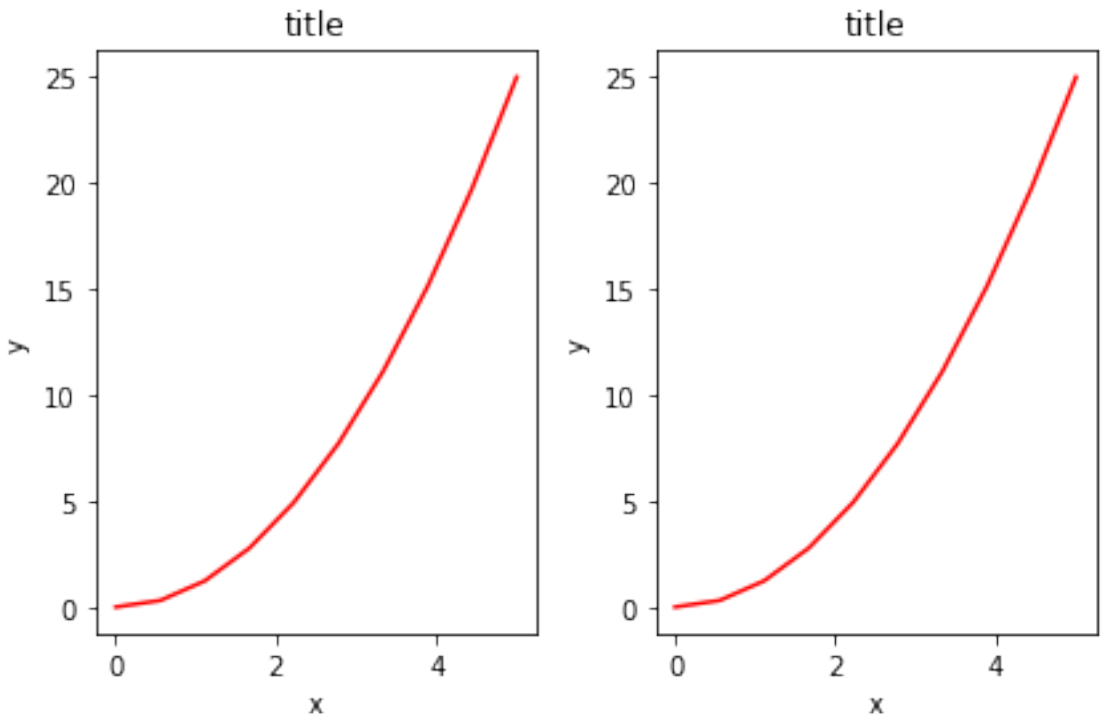
[13]:
```
fig, axes = plt.subplots(nrows=1, ncols=2)

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



** Remove Overlapping **

- Use the `fig.tight_layout` method, to automatically adjusts the positions of the axes on the figure canvas so that there is no overlapping content:

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

for ax in axes:
    ax.plot(x, y, 'r')
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_title('title')

fig.tight_layout()
```



2.2.2. Layout and Formatting Figure size, aspect ratio and DPI

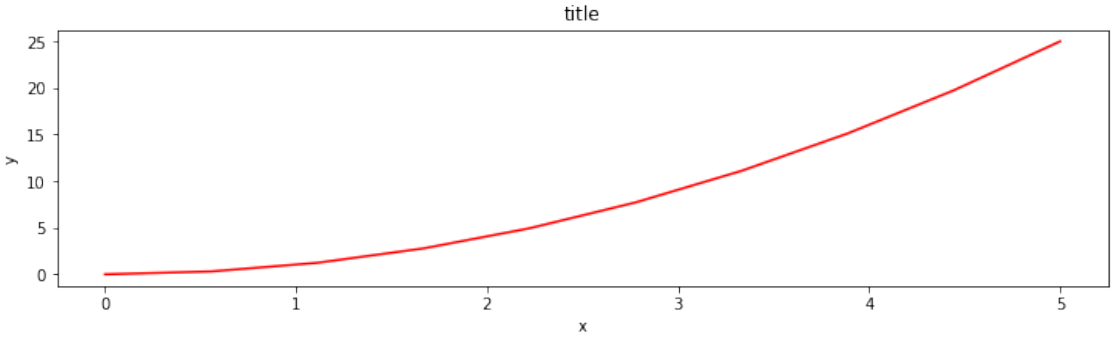
- Figures can have different aspect ratios and dots-per-inch (DPI)
- Set when creating `Figure` object using the `figsize` and `dpi` keyword arguments
- `figsize` is a tuple with width and height of the figure in inches,
- `dpi` is the dot-per-inch (pixel per inch). To create a figure with size 800 by 400 pixels we can do:

[15]:
```
fig = plt.figure(figsize=(8,4), dpi=100)
```

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

The same arguments can also be passed to layout managers, such as the `subplots` function.

[16]:
```
fig, axes = plt.subplots(figsize=(12,3))

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



** Saving figures **

To save a figure a file we can use the `savefig` method in the `Figure` class.

[17]:
```
fig.savefig("filename.png")
```

Here we can also optionally specify the DPI, and chose between different output formats.

[18]:
```
fig.savefig("filename.png", dpi=200)
```

[19]:
```
fig.savefig("filename.svg")
```

1.3.3 What formats are available and which ones should be used for best quality?

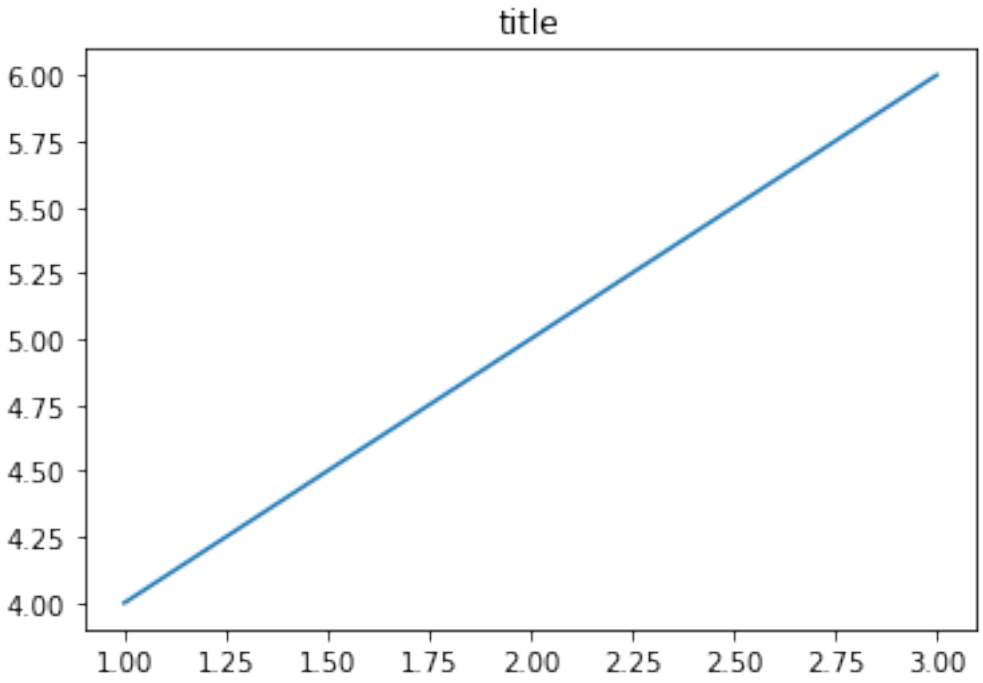
Matplotlib can generate high-quality output in a number formats, including PNG, JPG, EPS, SVG, PDF. For scientific papers, use PDF whenever possible (compile LaTeX documents with `pdflatex`, which can include PDFs using the `includegraphics` command).

Legends, labels and titles Now that we covered the basics of how to create a figure canvas and adding axes instances to the canvas, let's look at how decorate a figure with titles, axis labels and legends:

Figure titles

A title can be added to each axis instance in a figure. To set the title use the `set_title` method in the axes instance:

[20]:
```
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots()
x = [1, 2, 3]
y = [4, 5, 6]
ax.plot(x, y)
ax.set_title("title")
plt.show()
```



Axis labels

Similarly, using the methods `set_xlabel` and `set_ylabel` we can set the labels of the X and Y axes:

[21]:
```
ax.set_xlabel("x")
ax.set_ylabel("y")
```

[21]:
```
Text(3.200000000000003, 0.5, 'y')
```

Legends

Legends to curves in a figure can be added in two ways. First method is to use the `legend` method of the axis object and pass a list/tuple of legend texts for the curves that have previously been added:

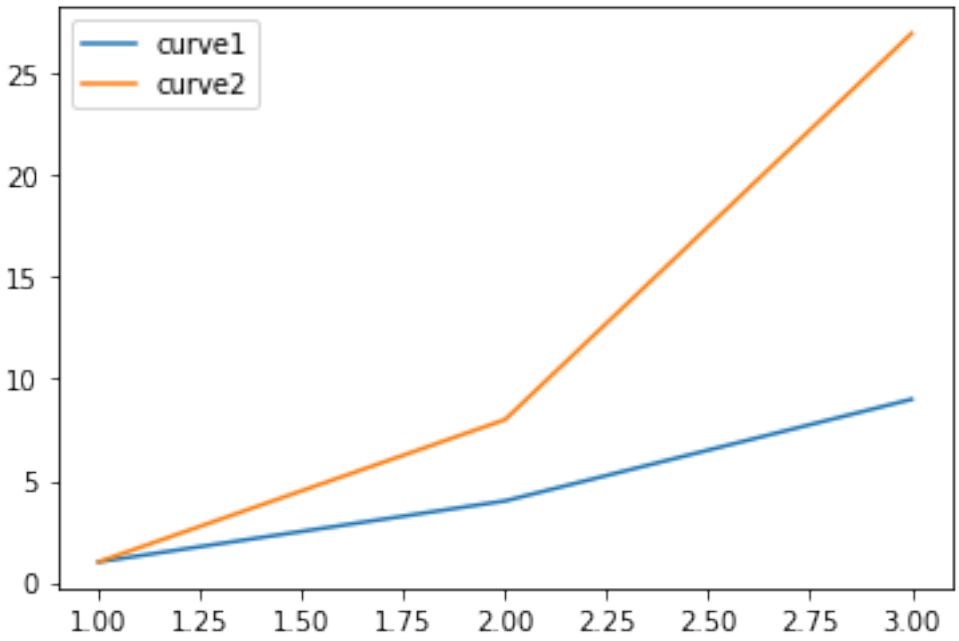
[22]:
```
ax.legend(["curve1", "curve2", "curve3"]);
```

The method described above follow the MATLAB API. It is somewhat prone to errors and unflexible if curves are added to or removed from the figure (resulting in wrong label being used for wrong curve).

A better method is to use the `label="label text"` keyword argument when plots a other objects are added to the figure, and then using the `legend` method without arguments to add the legend:

[23]:
```
# Create a new list with squared values of x
x_squared = [xi ** 2 for xi in x]
```

```
# Create a new list with cubed values of x
x_cubed = [xi ** 3 for xi in x]

# Plot the data
plt.plot(x, x_squared, label="curve1")
plt.plot(x, x_cubed, label="curve2")
plt.legend()

# Show the plot
plt.show()
```



The advantage with this method is that if curves are added or removed from the figure, the legend is automatically updated accordingly.

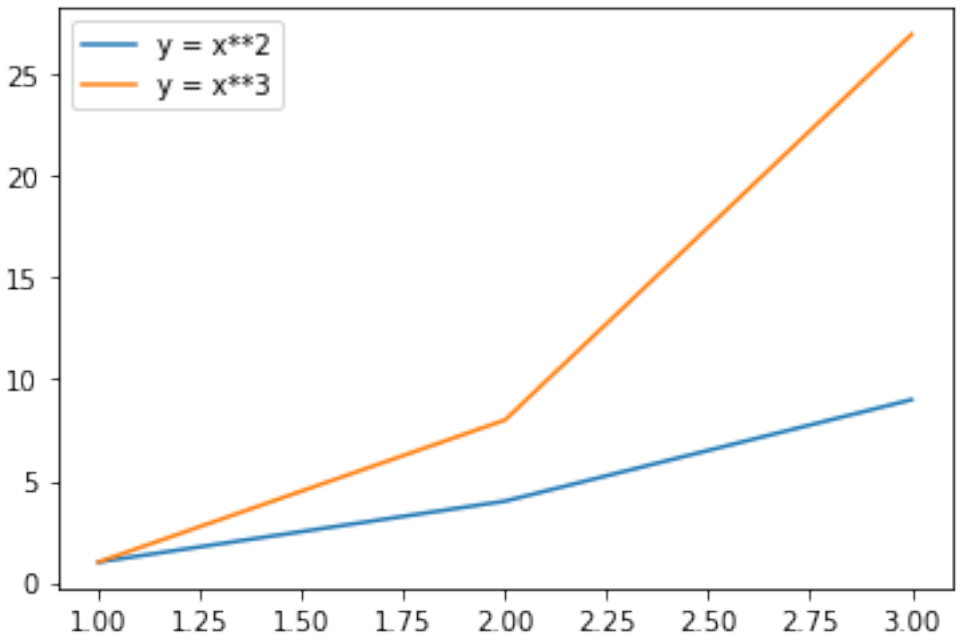
The `legend` function takes and optional keywork argument `loc` that can be used to specify where in the figure the legend is to be drawn. The allowed values of `loc` are numerical codes for the various places the legend can be drawn. See http://matplotlib.org/users/legend_guide.html#legend-location for details. Some most common alternatives are:

[24]:
```
ax.legend(loc=0) # let matplotlib decide the optimal location
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
```

```
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
No handles with labels found to put in legend.
```

[24]:
```
<matplotlib.legend.Legend at 0x22b3a71a460>
```

[25]:
```
# Create a new list with squared values of x
x_squared = [xi ** 2 for xi in x]

# Create a new list with cubed values of x
x_cubed = [xi ** 3 for xi in x]

# Plot the data
plt.plot(x, x_squared, label="y = x**2")
plt.plot(x, x_cubed, label="y = x**3")
plt.legend()

# Show the plot
plt.show()
ax.set_title('title')
ax.legend(loc=2); # upper left corner
```




```
No handles with labels found to put in legend.
```

Formatting text: LaTeX, fontsize, font family

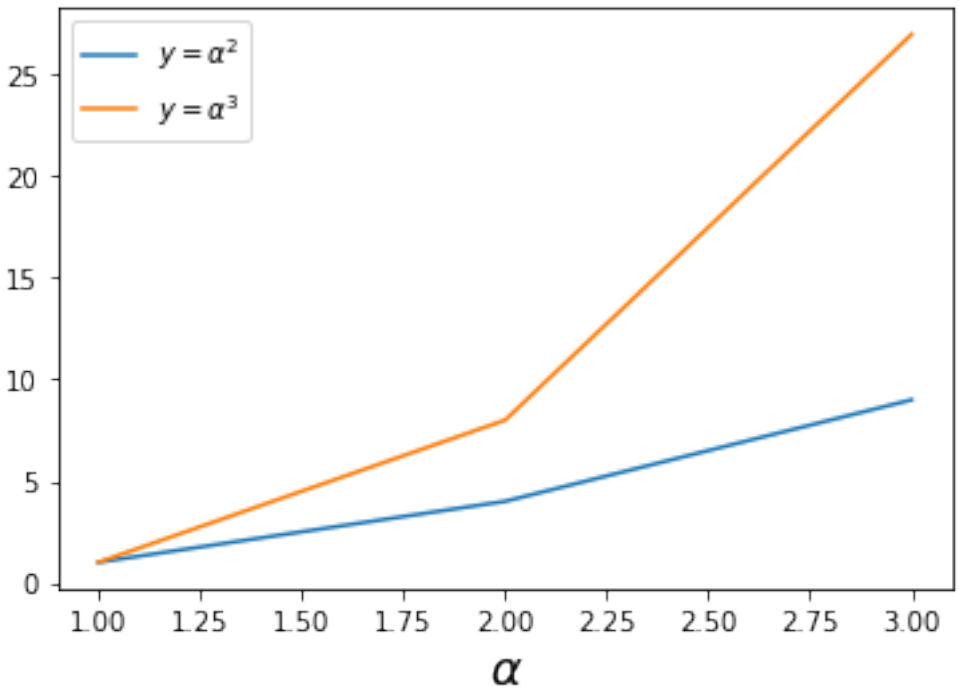
- Improve text to improve readability.
- We can use LaTeX formatted text and adjust font properties (size, font familiy, bold etc.)

Latex Support

- use dollar signs encapsulate LaTeX in any text (legend, title, label, etc.). For example, `"$y=x^3$"`.
- However, we need to escape \ for Latex commands
- Solution: use raw text strings
  - `r"String"`
  - e.g. `r"\alpha"` or `r'\alpha'` instead of `"\alpha"` or `'\alpha'`.

Example - Formatting Layout

[26]:
```
# Create a new list with squared values of x
x_squared = [xi ** 2 for xi in x]

# Create a new list with cubed values of x
x_cubed = [xi ** 3 for xi in x]

# Plot the data
plt.plot(x, x_squared, label=r"$y = \alpha^2$") # use latex equations as raw␣
 ↪strings
plt.plot(x, x_cubed, label=r"$y = \alpha^3$") # use latex equations as raw␣
 ↪strings
plt.xlabel(r'$\alpha$', fontsize=18)

# Add a legend
plt.legend()

# Show the plot
plt.show()
ax.set_title('title')
ax.legend(loc=2); # upper left corner
```



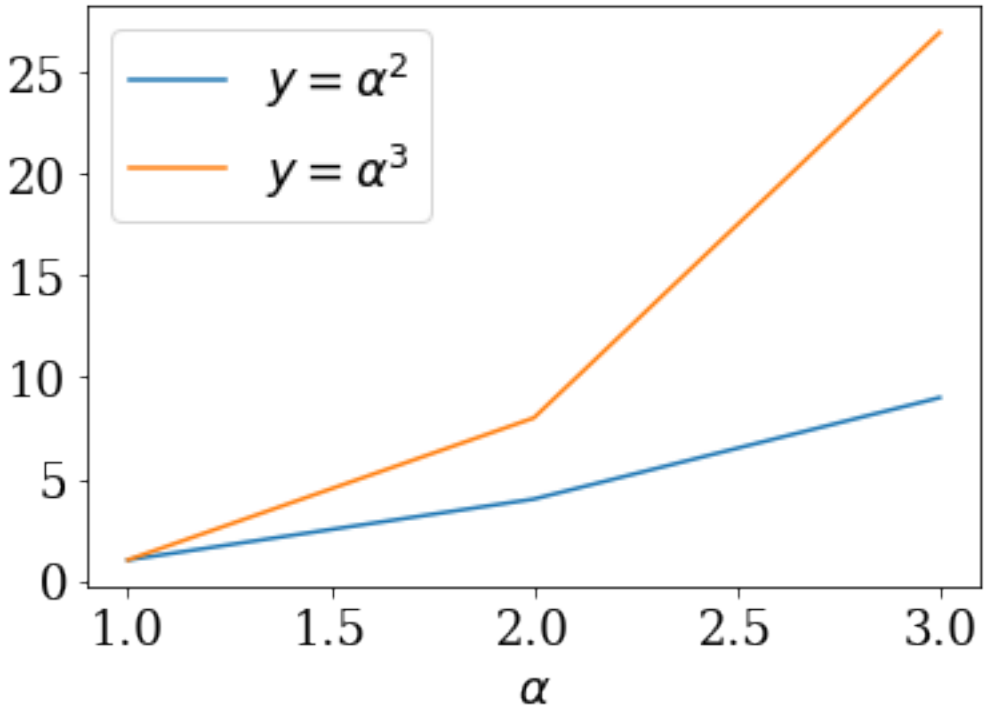

```
No handles with labels found to put in legend.
```

Updateing Font Size We can also change the global font size and font family, which applies to all text elements in a figure (tick labels, axis labels and titles, legends, etc.):

[28]:
```
# Update the matplotlib configuration parameters:
plt.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

[29]:
```
# Create a new list with squared values of x
x_squared = [xi ** 2 for xi in x]

# Create a new list with cubed values of x
x_cubed = [xi ** 3 for xi in x]

# Plot the data
plt.plot(x, x_squared, label=r"$y = \alpha^2$") # use latex equations as raw␣
 ↪strings
plt.plot(x, x_cubed, label=r"$y = \alpha^3$") # use latex equations as raw␣
 ↪strings
plt.xlabel(r'$\alpha$', fontsize=18)

# Add a legend
plt.legend()

# Show the plot
```

```
plt.show()
ax.legend(loc=2); # upper left corner
```

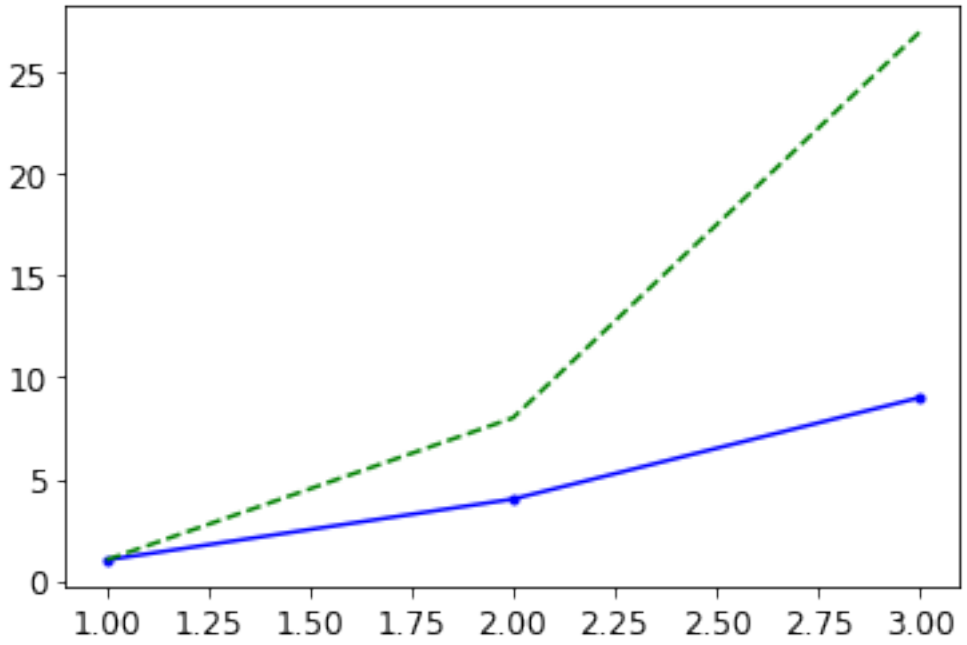



```
No handles with labels found to put in legend.
```

[30]:
```
# Update the matplotlib configuration parameters:
plt.rcParams.update({'font.size': 12, 'font.family': 'sans'})
```

[31]:
```
# MATLAB style line color and style
fig, ax = plt.subplots()
ax.plot(x, np.array(x)**2, 'b.-') # blue line with dots
ax.plot(x, np.array(x)**3, 'g--') # green dashed line
```

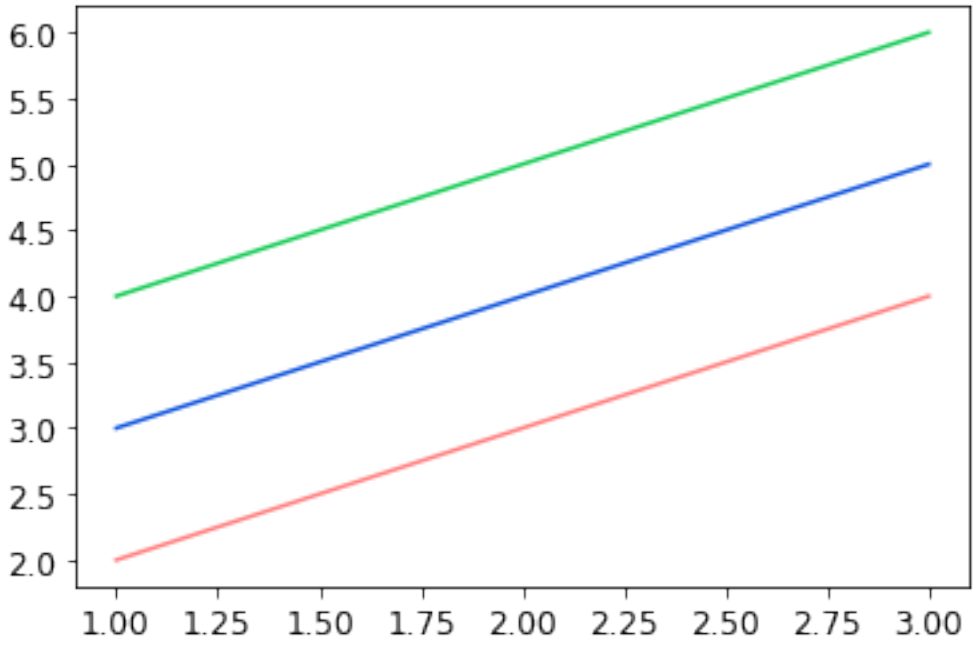
[31]:
```
[<matplotlib.lines.Line2D at 0x22b3a42b760>]
```



In matplotlib we can also define colors by their name or RGB hex codes, and optionally provide an alpha value, using the `color` and `alpha` keyword arguments:

[32]:
```
fig, ax = plt.subplots()

ax.plot(x, np.array(x) + 1, color="red", alpha=0.5) # half-transparant red
ax.plot(x, np.array(x) + 2, color="#1155dd")        # RGB hex code for a bluish␣
 ↪color
ax.plot(x, np.array(x) + 3, color="#15cc55")        # RGB hex code for a␣
 ↪greenish color
```

[32]:
```
[<matplotlib.lines.Line2D at 0x22b3a471190>]
```

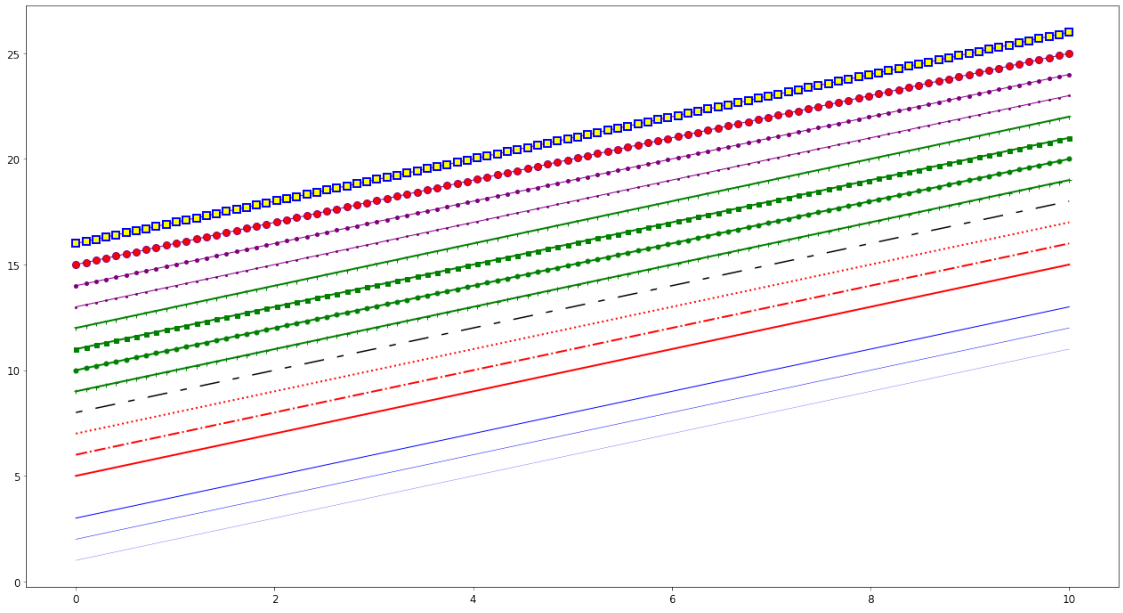


Line and marker styles

To change the line width we can use the `linewidth` or `lw` keyword argument, and the line style can be selected using the `linestyle` or `ls` keyword arguments:

[85]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)

fig, ax = plt.subplots(figsize=(22, 12))

ax.plot(x, np.array(x) + 1, color="blue", linewidth=0.25)
ax.plot(x, np.array(x) + 2, color="blue", linewidth=0.50)
ax.plot(x, np.array(x) + 3, color="blue", linewidth=1.00)

# Possible linestype options: '-', '--', '-.', ':', 'steps'
ax.plot(x, np.array(x) + 5, color="red", lw=2, linestyle='-')
ax.plot(x, np.array(x) + 6, color="red", lw=2, ls='-.')
ax.plot(x, np.array(x) + 7, color="red", lw=2, ls=':')

# Custom dash as (offset, on-off pattern)
ax.plot(x, np.array(x) + 8, color="black", lw=1.50, dashes=(5, 10, 15, 10))  #␣
 ↪Define custom dashes here

# Marker symbols with color
```

```
ax.plot(x, x + 9, color="green", lw=2, marker='+', markersize=5)
ax.plot(x, x + 10, color="green", lw=2, marker='o', markersize=5)
ax.plot(x, x + 11, color="green", lw=2, marker='s', markersize=5)
ax.plot(x, x + 12, color="green", lw=2, marker='1', markersize=5)

# Marker size and color
ax.plot(x, x + 13, color="purple", lw=1, linestyle='-', marker='o',␣
 ↪markersize=2)
ax.plot(x, x + 14, color="purple", lw=1, linestyle='-', marker='o',␣
 ↪markersize=4)
ax.plot(x, x + 15, color="purple", lw=1, linestyle='-', marker='o',␣
 ↪markersize=8, markerfacecolor="red")
ax.plot(x, x + 16, color="purple", lw=1, linestyle='-', marker='s',␣
 ↪markersize=8,
        markerfacecolor="yellow", markeredgewidth=2, markeredgecolor="blue")

plt.show()
```



2.2.3 Control over axis apperance The appearance of the axes is an important aspect of a figure that we often need to modify to make a high quality graphics. We need to be able to control where the ticks and labels are placed, modify the font size and possibly the labels used on the axes. In this section we will look at controling those properties in a matplotlib figure.

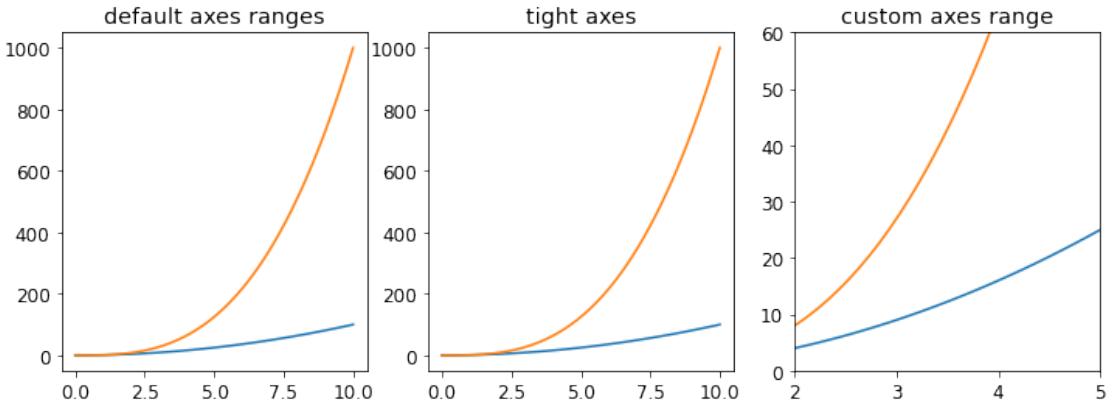
Plot range

The first thing we might want to configure is the ranges of the axes. We can do it using the `set_ylim` and `set_xlim` methods in the axis object, or `axis('tight')` for automatrically getting

"tightly fitted" axes ranges.

[40]:
```
import numpy as np

fig, axes = plt.subplots(1, 3, figsize=(12, 4))

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


axes[1].plot(x, np.array(x)**2, x, np.array(x)**3)
axes[1].axis('tight')
axes[1].set_title("tight axes")

axes[2].plot(x, np.array(x)**2, x, np.array(x)**3)
axes[2].set_ylim([0, 60])
axes[2].set_xlim([2, 5])
axes[2].set_title("custom axes range")
```

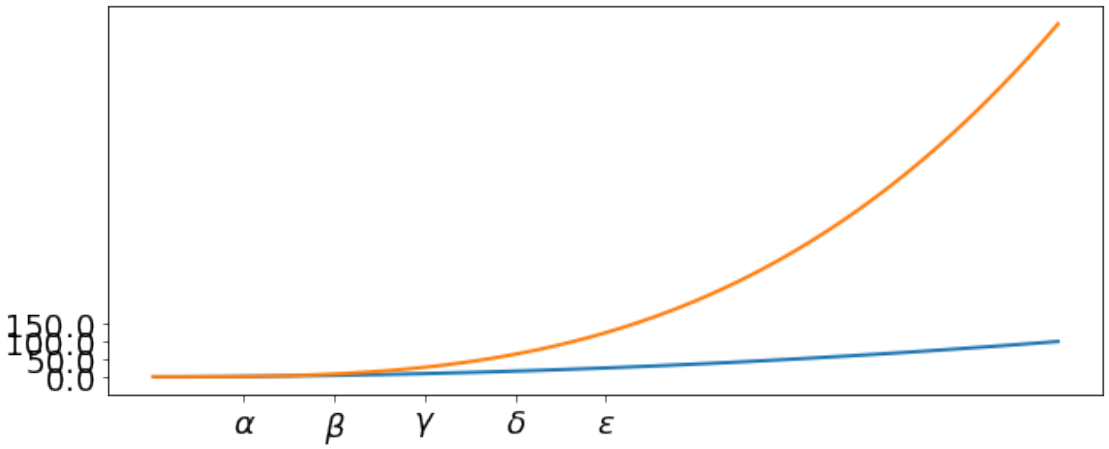
[40]:
```
Text(0.5, 1.0, 'custom axes range')
```



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

[41]:
```
fig, ax = plt.subplots(figsize=(10, 4))

ax.plot(x, np.array(x)**2, x, np.array(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
```

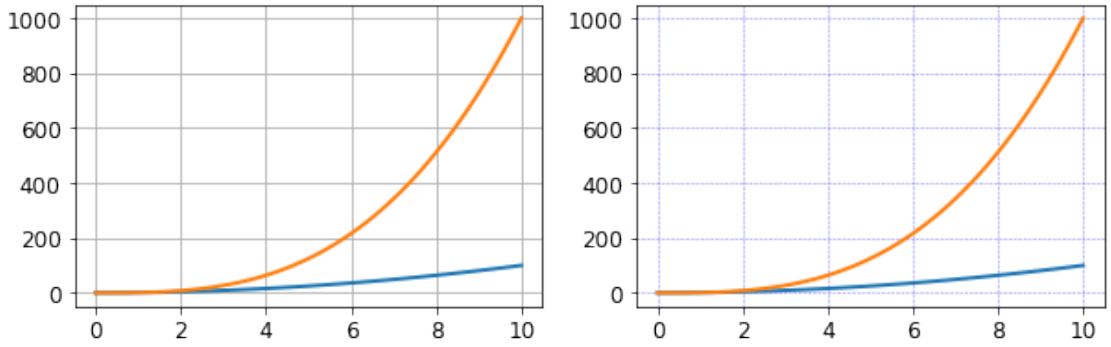


In matplotlib there is a number of more advanced methods for controlling major and minor tick placement, such as automatic placement according to different policies. See http://matplotlib.org/api/ticker_api.html for details.

Grid

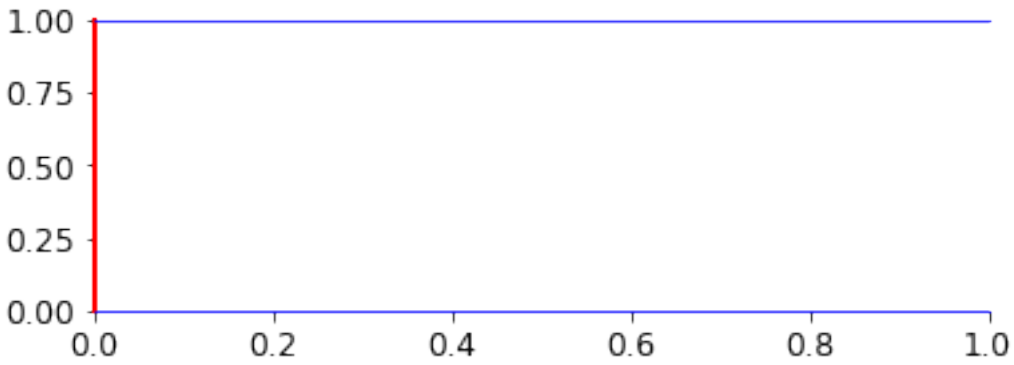
Using the `grid` method in the axis object we can turn on and off grid lines. We can also customize the appearence of the gridlines, using the same keywork arguments as we previously used with the `plot` function.

[42]:
```
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)
```



Axis spines We can also change the properties of the axis spines:

[43]:
```
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
```

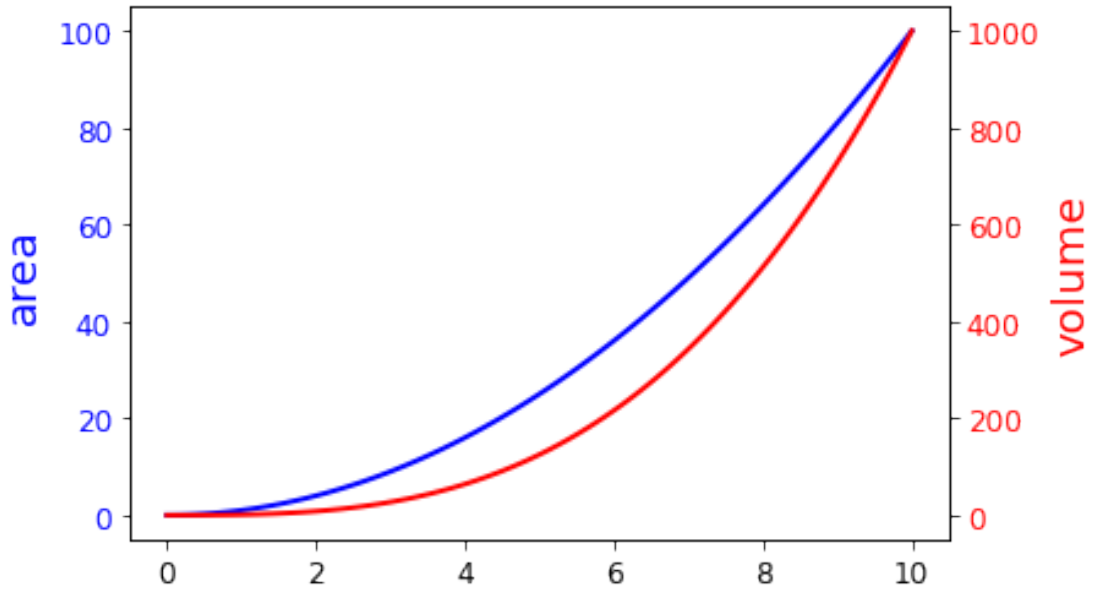


** Twin Axes **

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

[45]:
```
import matplotlib.pyplot as plt
import numpy as np
```

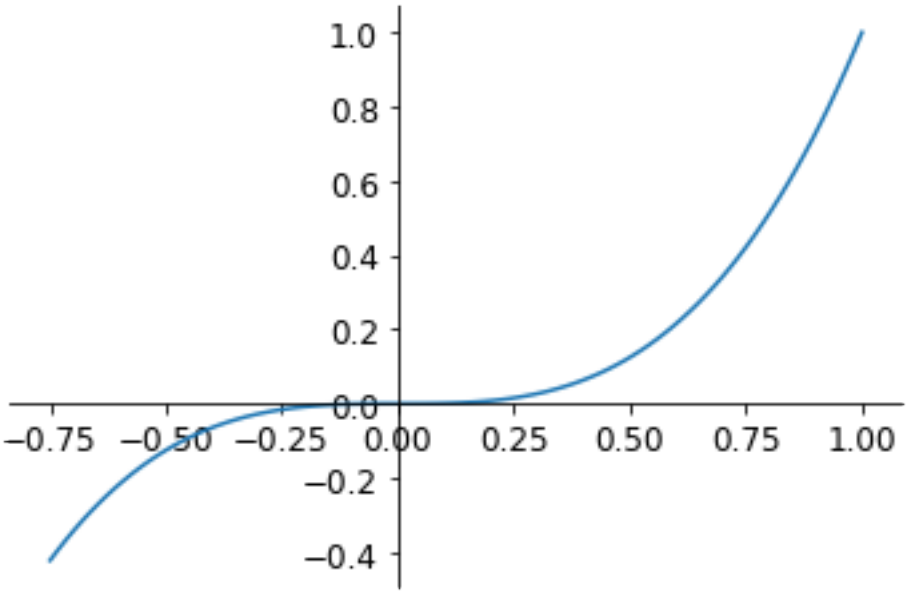
```
x = np.linspace(0, 10, 100)

fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_ylabel(r"area", 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", fontsize=18, color="red")
for label in ax2.get_yticklabels():
    label.set_color("red")

plt.show()
```



1.3.4 Axes where x and y is zero

[46]:
```
fig, ax = plt.subplots()

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

ax.xaxis.set_ticks_position('bottom')
```

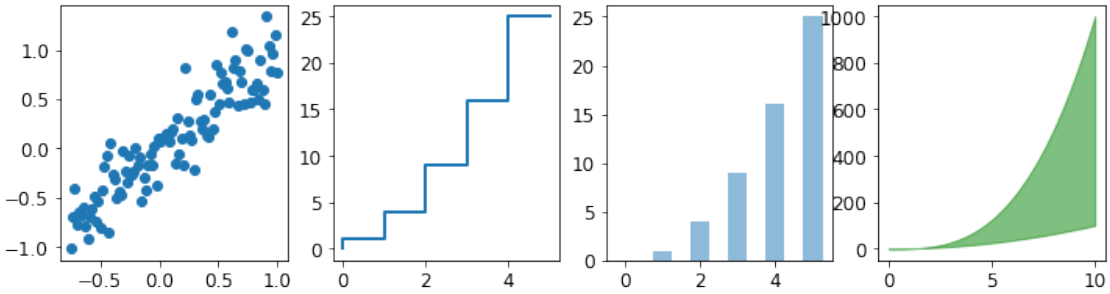
```
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

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);
```



2.2.3. Other 2D plot styles In addition to the function `plot`, there are a number of other functions for generating different kind of plots. See the matplotlib plot gallery for a complete list of avaiable plot types: http://matplotlib.org/gallery.html. Some of the more useful ones are show below:

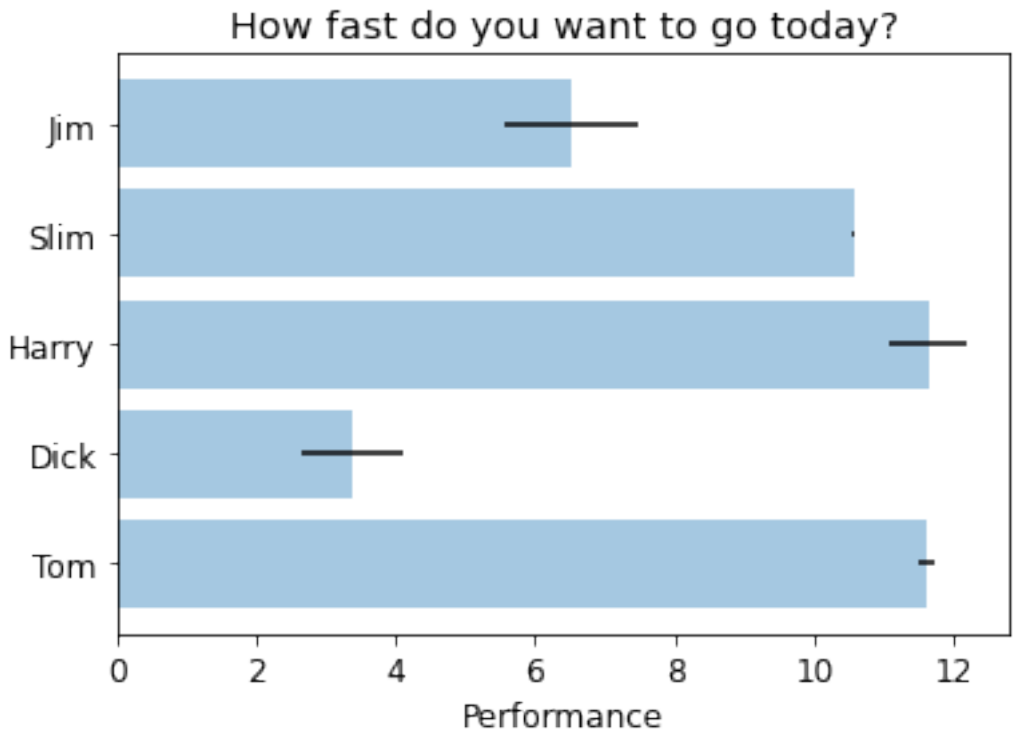
[47]:
```
n = array([0,1,2,3,4,5])
```

[48]:
```
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(xx, xx + 0.25*randn(len(xx)))

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

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

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



Bar Charts

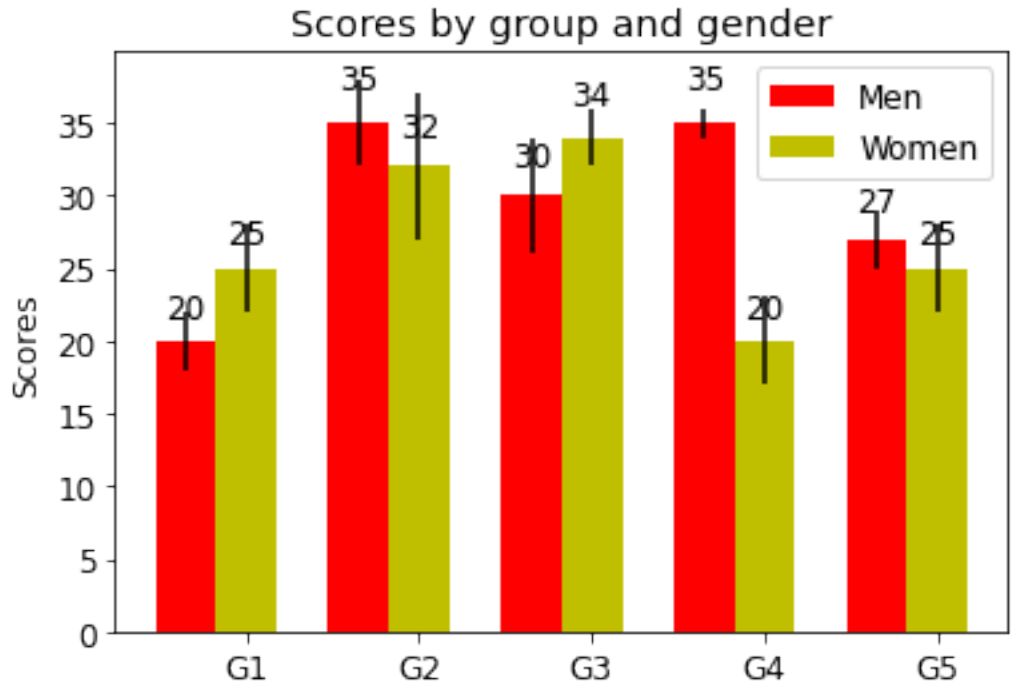
[49]:
```
# Example data
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

plt.barh(y_pos, performance, xerr=error, align='center', alpha=0.4)
plt.yticks(y_pos, people)
plt.xlabel('Performance')
plt.title('How fast do you want to go today?')

plt.show()
```



** Bar Charts with Standard Deviation **

[50]:
```
N = 5
menMeans = (20, 35, 30, 35, 27)
menStd =   (2, 3, 4, 1, 2)

ind = np.arange(N)  # the x locations for the groups
width = 0.35       # the width of the bars

fig, ax = plt.subplots()
rects1 = ax.bar(ind, menMeans, width, color='r', yerr=menStd)

womenMeans = (25, 32, 34, 20, 25)
womenStd =   (3, 5, 2, 3, 3)
rects2 = ax.bar(ind+width, womenMeans, width, color='y', yerr=womenStd)

# add some
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
ax.set_xticks(ind+width)
ax.set_xticklabels( ('G1', 'G2', 'G3', 'G4', 'G5') )

ax.legend( (rects1[0], rects2[0]), ('Men', 'Women') )
```

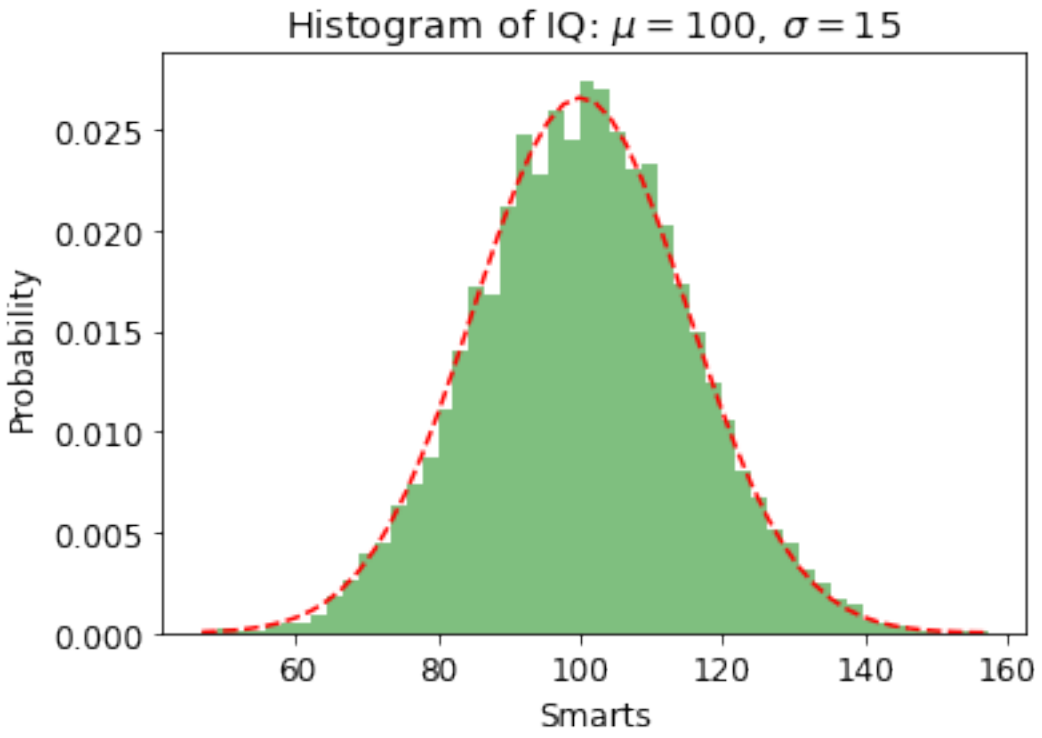
```
def autolabel(rects):
    # attach some text labels
    for rect in rects:
        height = rect.get_height()
        ax.text(rect.get_x()+rect.get_width()/2., 1.05*height, '%d'%int(height),
                ha='center', va='bottom')

autolabel(rects1)
autolabel(rects2)

plt.show()
```



** Histogram**

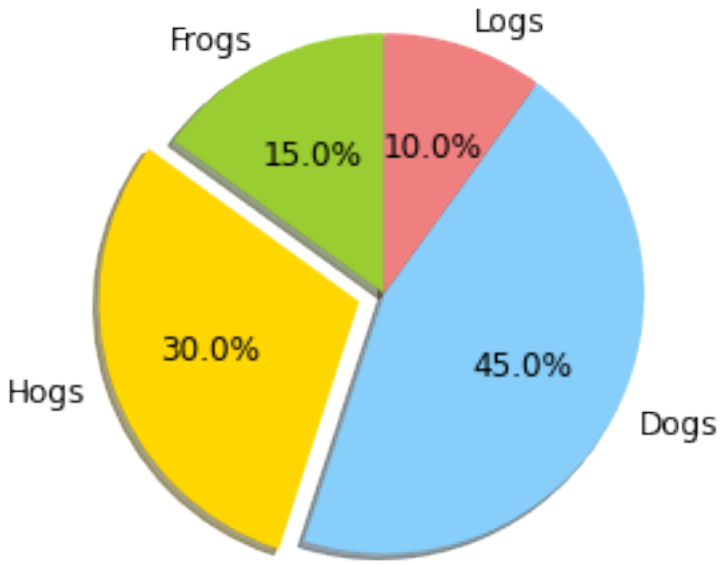
[53]:
```
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import norm  # Import norm from scipy.stats

# Example data
mu = 100  # Mean of distribution
sigma = 15  # Standard deviation of distribution
x = mu + sigma * np.random.randn(10000)
```

```
num_bins = 50
# The histogram of the data
n, bins, patches = plt.hist(x, num_bins, density=1, facecolor='green', alpha=0.
↪5)  # Use density instead of normed

# Calculate the PDF for the normal distribution
y = norm.pdf(bins, mu, sigma)

plt.plot(bins, y, 'r--')
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'Histogram of IQ: $\mu=100$, $\sigma=15$')  # Define title on a␣
↪single line

# Tweak spacing to prevent clipping of ylabel
plt.subplots_adjust(left=0.15)
plt.show()
```



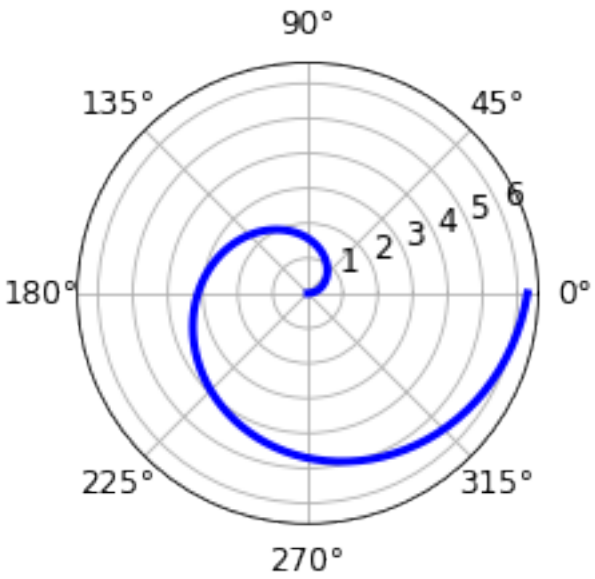
Pie Charts

[54]:
```
# The slices will be ordered and plotted counter-clockwise.
labels = 'Frogs', 'Hogs', 'Dogs', 'Logs'
sizes = [15, 30, 45, 10]
colors = ['yellowgreen', 'gold', 'lightskyblue', 'lightcoral']
explode = (0, 0.1, 0, 0) # only "explode" the 2nd slice (i.e. 'Hogs')

plt.pie(sizes, explode=explode, labels=labels, colors=colors,
        autopct='%1.1f%%', shadow=True, startangle=90)
# Set aspect ratio to be equal so that pie is drawn as a circle.
plt.axis('equal')
plt.show()
```



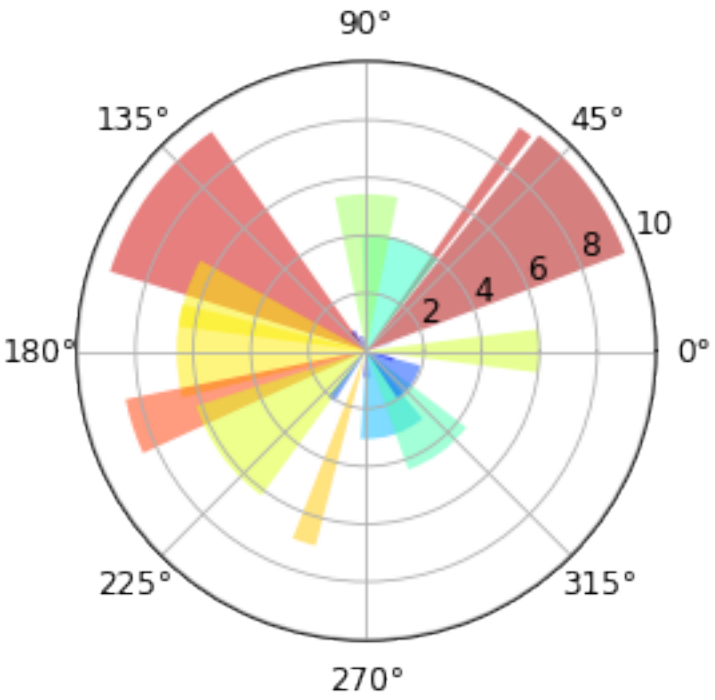
Polar Plots Plot data in polar coordinates as line chart

[55]:
```
# polar plot using add_axes and polar projection
fig = plt.figure()
ax = fig.add_axes([0.0, 0.0, .6, .6], polar=True)
t = linspace(0, 2 * pi, 100)
ax.plot(t, t, color='blue', lw=3);
```



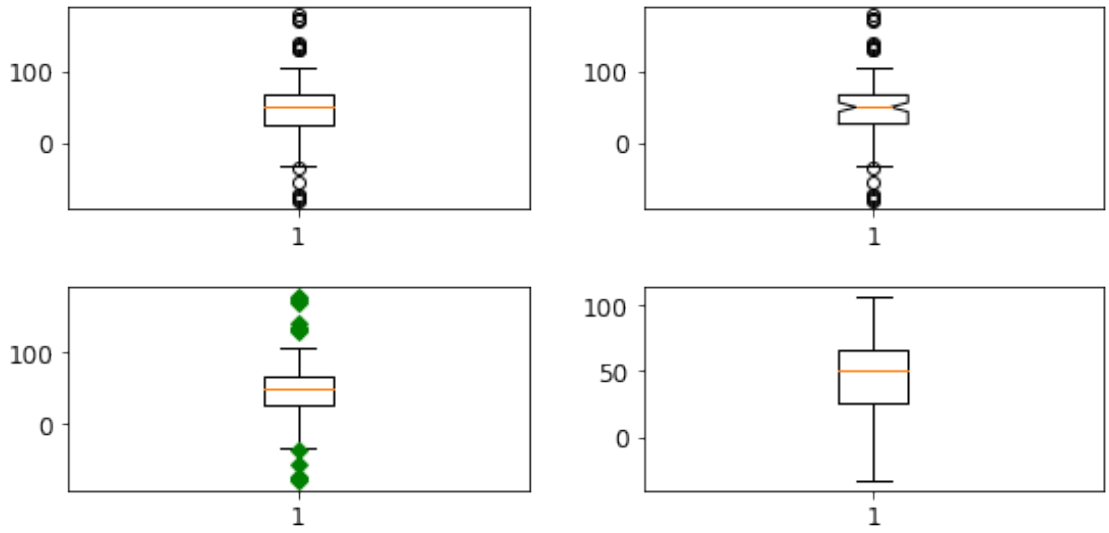
** Polar Plot Bar Chart **

[56]:
```
N = 20
#polar bars have 3 channels (excluding color)
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N) #height of a bar
width = np.pi / 4 * np.random.rand(N) # opening angle of a bar

ax = plt.subplot(111, polar=True)
bars = ax.bar(theta, radii, width=width, bottom=0.0)

# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)

plt.show()
```



Box Plots

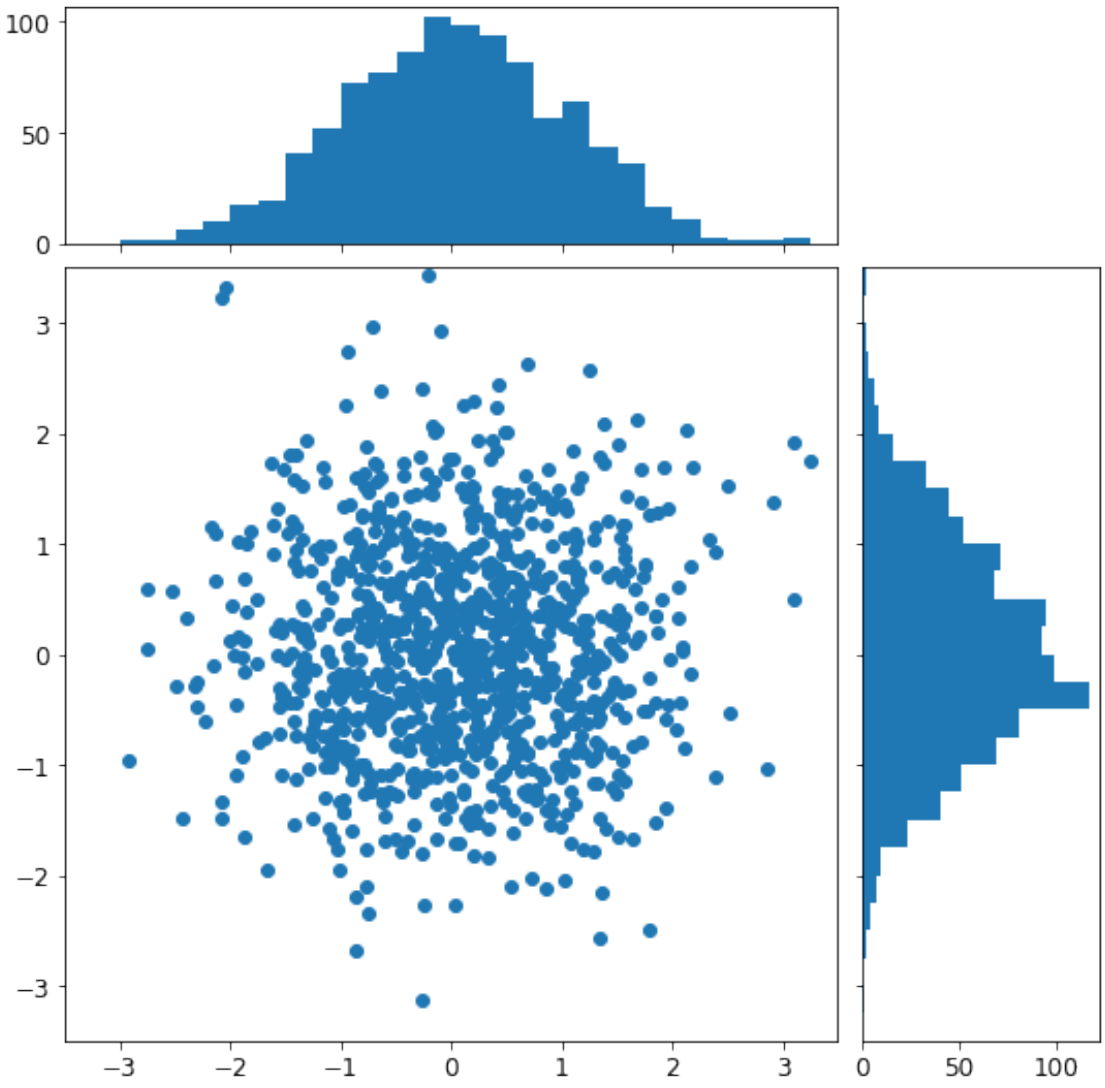
[58]:
```
# fake up some data
spread= rand(50) * 100
center = ones(25) * 50
flier_high = rand(10) * 100 + 100
flier_low = rand(10) * -100
data =concatenate((spread, center, flier_high, flier_low), 0)

fig, axes = plt.subplots(nrows=2,ncols=2,figsize=(8,4))
fig.tight_layout()
# basic plot
axes[0,0].boxplot(data)
# notched plot
axes[0,1].boxplot(data,1)
# change outlier point symbols
axes[1,0].boxplot(data,0,'gD')
# don't show outlier points
axes[1,1].boxplot(data,0,'')

plt.show()
```



** Scatter Plot with Marign Distribution **

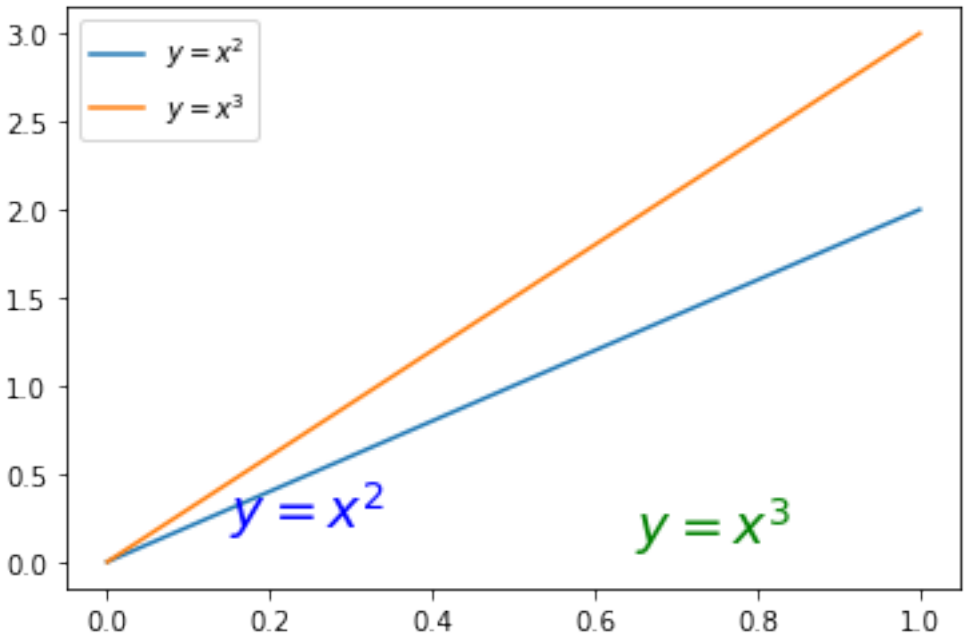
[59]:
```
from matplotlib.ticker import NullFormatter

# the random data
x = np.random.randn(1000)
y = np.random.randn(1000)

nullfmt   = NullFormatter()         # no labels

# definitions for the axes
left, width = 0.1, 0.65
bottom, height = 0.1, 0.65
bottom_h = left_h = left+width+0.02

rect_scatter = [left, bottom, width, height]
rect_histx = [left, bottom_h, width, 0.2]
rect_histy = [left_h, bottom, 0.2, height]

# start with a rectangular Figure
plt.figure(1, figsize=(8,8))

axScatter = plt.axes(rect_scatter)
axHistx = plt.axes(rect_histx)
axHisty = plt.axes(rect_histy)

# no labels
axHistx.xaxis.set_major_formatter(nullfmt)
axHisty.yaxis.set_major_formatter(nullfmt)
```

```
# the scatter plot:
axScatter.scatter(x, y)

# now determine nice limits by hand:
binwidth = 0.25
xymax = np.max( [np.max(np.fabs(x)), np.max(np.fabs(y))] )
lim = ( int(xymax/binwidth) + 1) * binwidth

axScatter.set_xlim( (-lim, lim) )
axScatter.set_ylim( (-lim, lim) )

bins = np.arange(-lim, lim + binwidth, binwidth)
axHistx.hist(x, bins=bins)
axHisty.hist(y, bins=bins, orientation='horizontal')

axHistx.set_xlim( axScatter.get_xlim() )
axHisty.set_ylim( axScatter.get_ylim() )

plt.show()
```



[1]:
```
import matplotlib.pyplot as plt
import numpy as np

xx = np.linspace(0, 1, 100)  # Define the x values

fig, ax = plt.subplots()

ax.plot(xx, xx*2, label=r"$y=x^2$")
ax.plot(xx, xx*3, label=r"$y=x^3$")

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")
```

```
ax.legend()  # Add a legend to label the lines

plt.show()
```



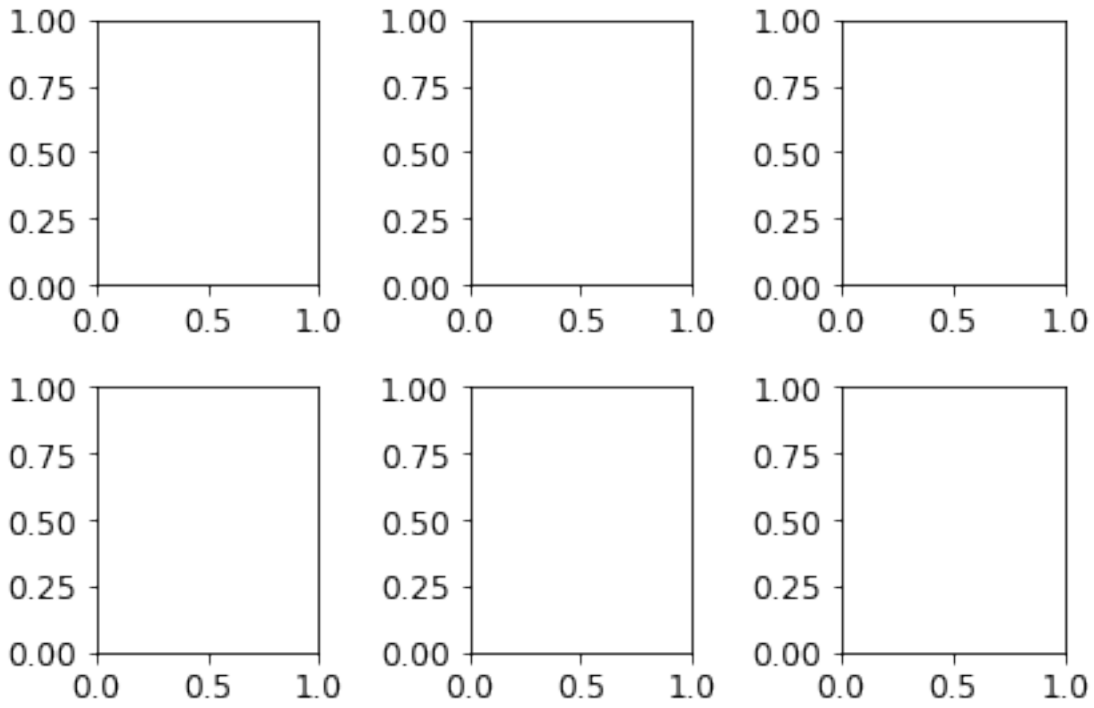
[ ]:
```
fig. ax = plt.subplots()
color=" blue")
ax.plot(xx, XX* *2,
XX, xx**3)
ax.text(e.15, e. 2, r Y
r"y A"
ax.text(e.65, e. 1,
fontsize=2e,
fontsize=2ê,
color='
' green") ;
```

1.4 Figures with multiple subplots and insets

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

1.4.1 subplots

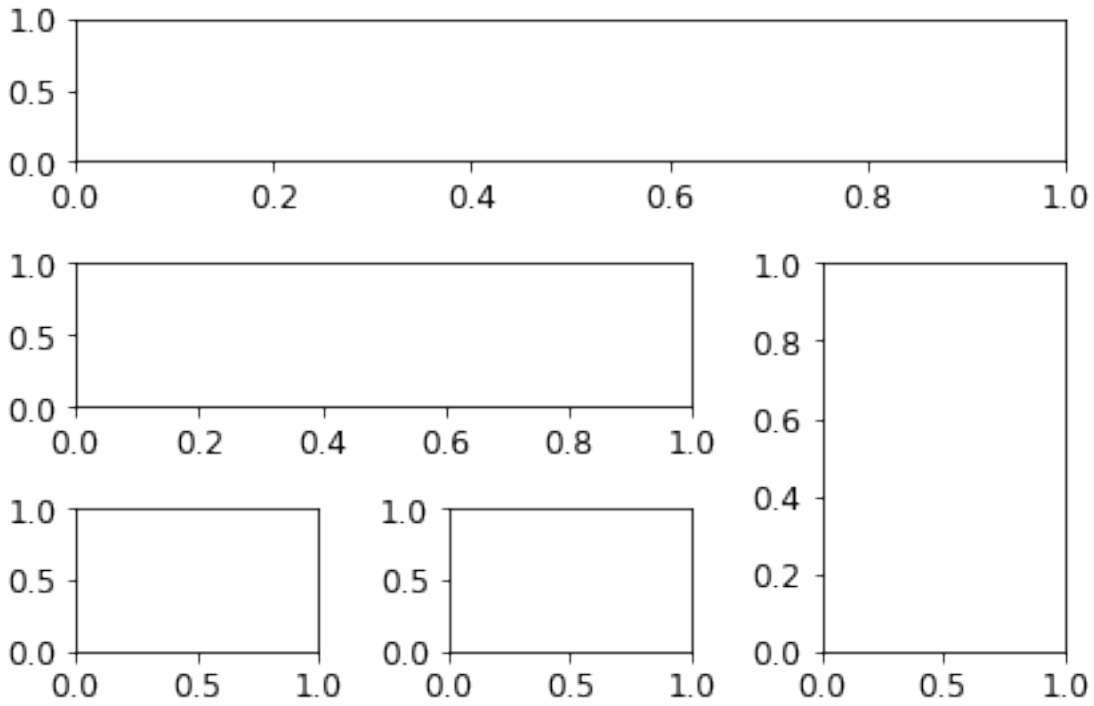
[61]:
```
fig, ax = plt.subplots(2, 3)
fig.tight_layout()
```



subplot2grid

Create subplots of different size

[62]:
```
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()
```

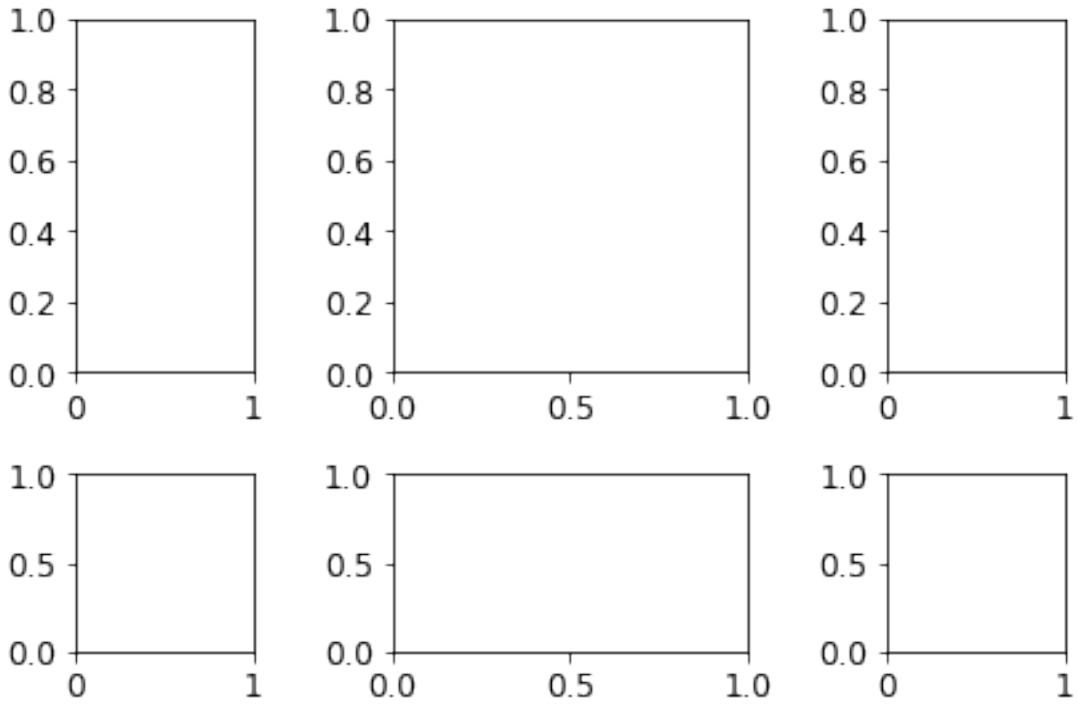


gridspec

Similar to `subplots2grid`, but with specifying a more regular layout

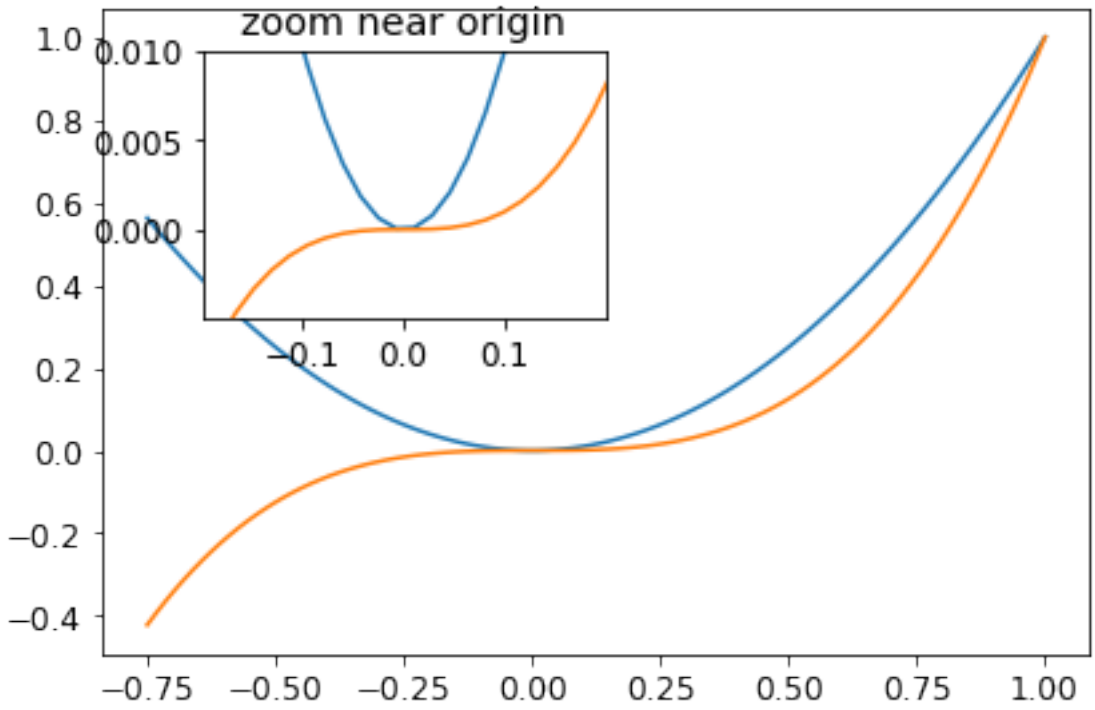
[63]:
```
import matplotlib.gridspec as gridspec
```

[64]:
```
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()
```



1.4.2 add_axes

[65]:
```
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]);
```



1.5 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 is a number of predefined colormaps, and it is relatively straightforward to define custom colormaps. For a list of pre-defined colormaps, see: http://www.scipy.org/Cookbook/Matplotlib/Show_colormaps

pcolor

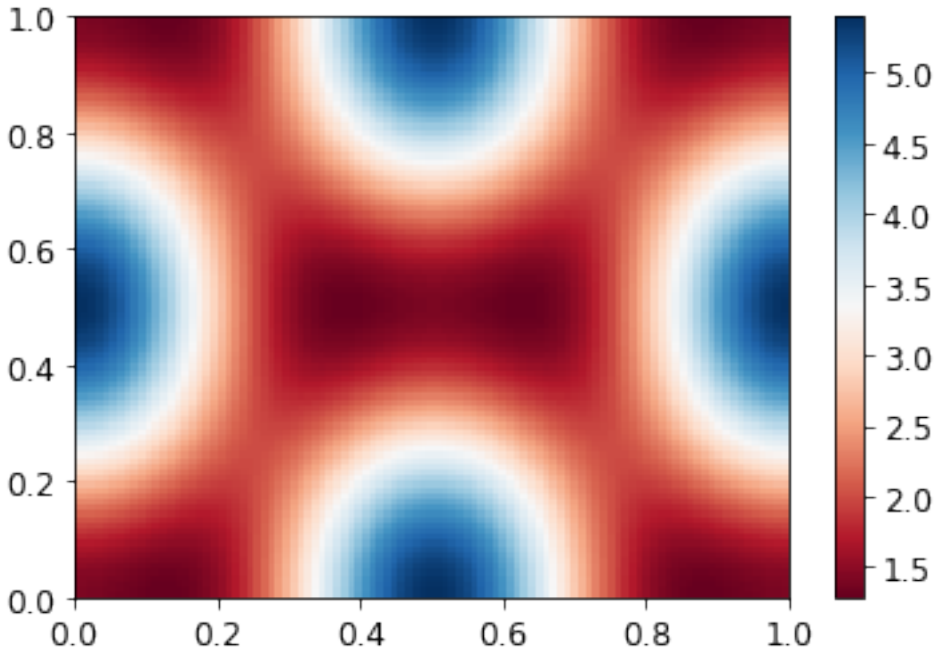
Create a pseudocolor plot of a 2-D array. Can be very slow for large 2D arrays.

[66]:
```
alpha = 0.7
phi_ext = 2 * pi * 0.5

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

[67]:
```
phi_m = linspace(0, 2*pi, 100)
phi_p = linspace(0, 2*pi, 100)
X,Y = meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
```

[68]:
```
fig, ax = plt.subplots()

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

```
C:\Users\PC\AppData\Local\Temp/ipykernel_9588/2727627212.py:3:
MatplotlibDeprecationWarning: shading='flat' when X and Y have the same
dimensions as C is deprecated since 3.3.  Either specify the corners of the
quadrilaterals with X and Y, or pass shading='auto', 'nearest' or 'gouraud', or
set rcParams['pcolor.shading'].  This will become an error two minor releases
later.
  p = ax.pcolor(X/(2*pi), Y/(2*pi), Z, cmap=cm.RdBu, vmin=abs(Z).min(),
vmax=abs(Z).max())
```

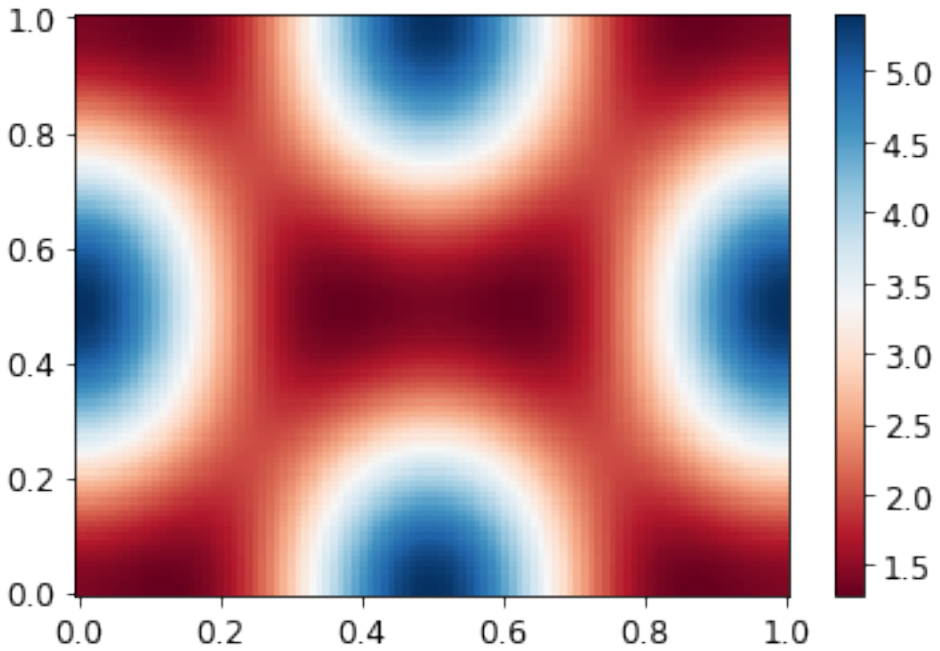


imshow

Display an image on the axes.

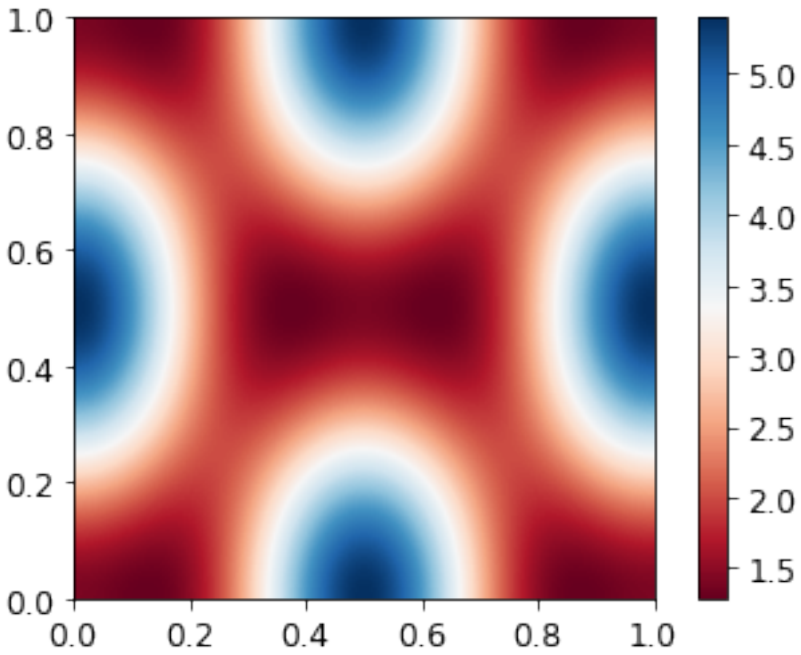
[69]:
```
import matplotlib.pyplot as plt
import numpy as np

# Assuming you have defined X, Y, and Z

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

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

plt.show()
```



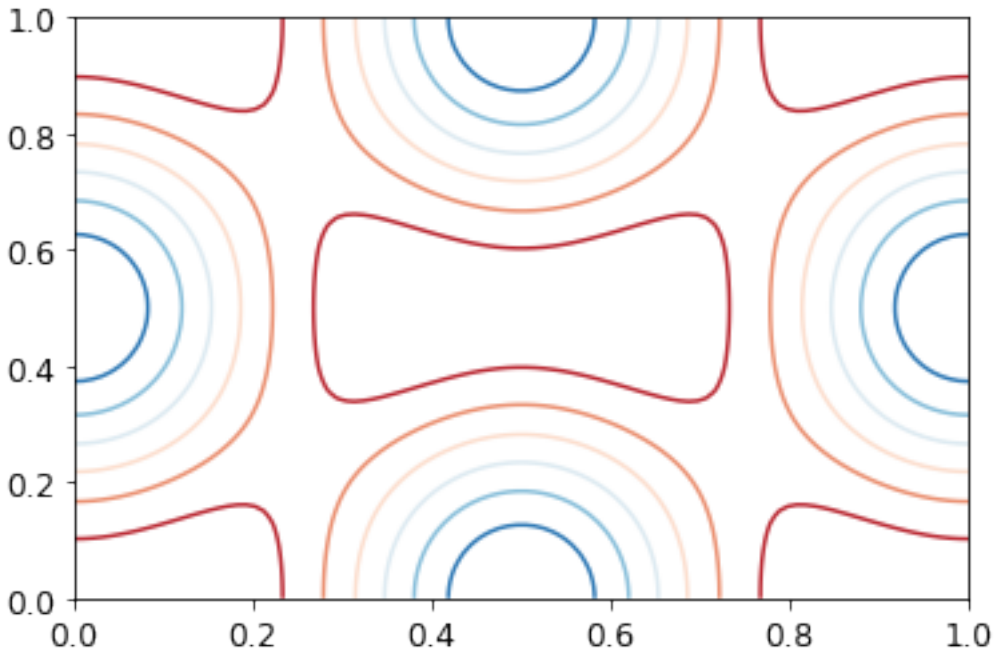
Note that images can be interpolated and that indexig might vary. For interpolation in smaller arrays use `interpolation="nearest"`

[70]:
```
fig, ax = plt.subplots()

im = imshow(Z, cmap=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)
```



1.5.1 contour

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

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



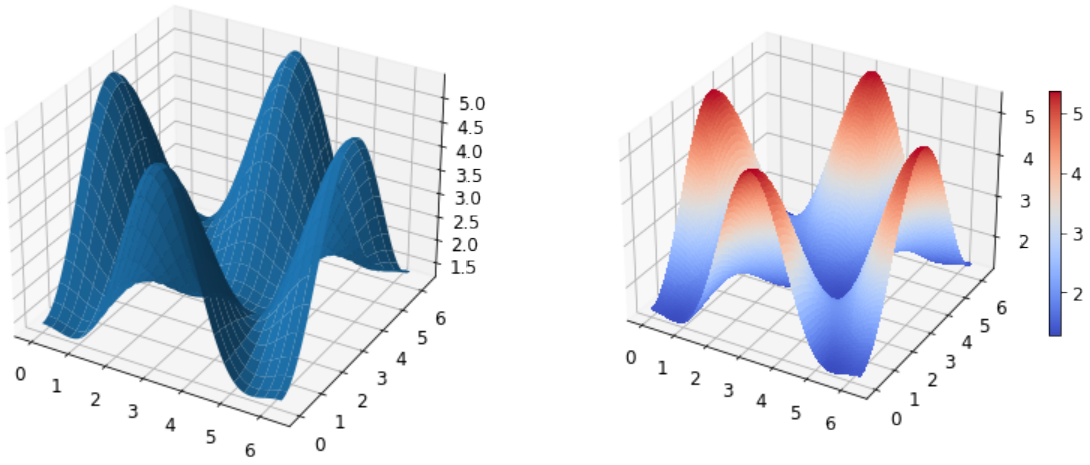
3D figures

To use 3D graphics in matplotlib, we first need to create an axes instance of the class Axes3D. 3D axes can be added to a matplotlib figure canvas in exactly the same way as 2D axes, but a conventient way to create a 3D axis instance is to use the projection='3d' keyword argument to the add_axes or add_subplot functions.

[72]:
```
from mpl_toolkits.mplot3d.axes3d import Axes3D
```

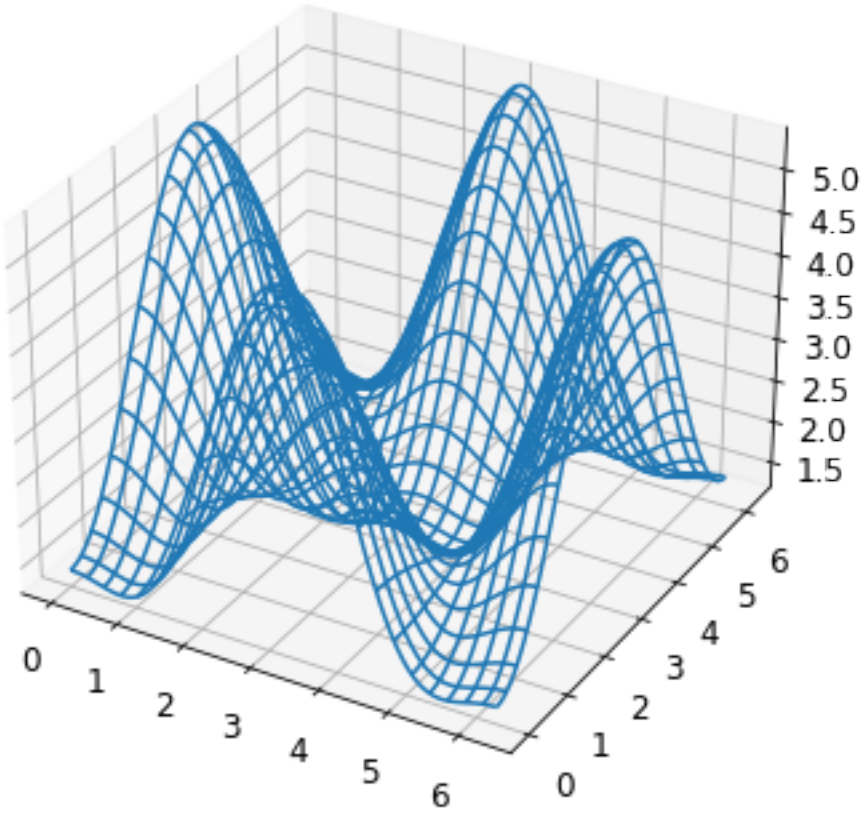
Surface plots

[73]:
```
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=cm.coolwarm,␣
 ↪linewidth=0, antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```



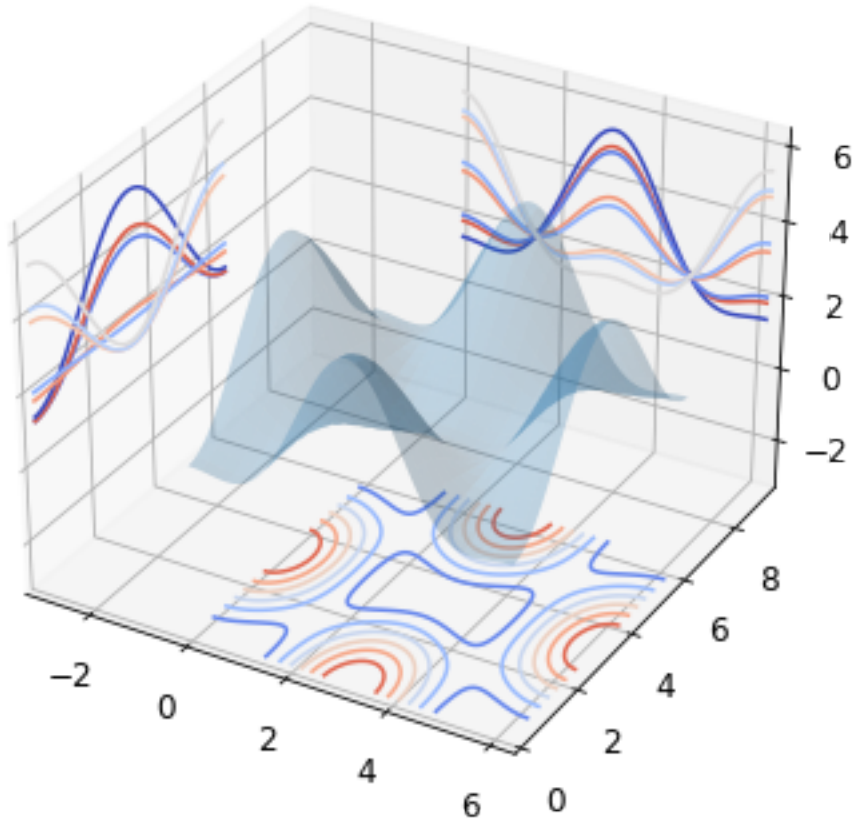
1.5.2 Wire-frame plot

[74]:
```
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)
```



1.5.3 Coutour plots with projections

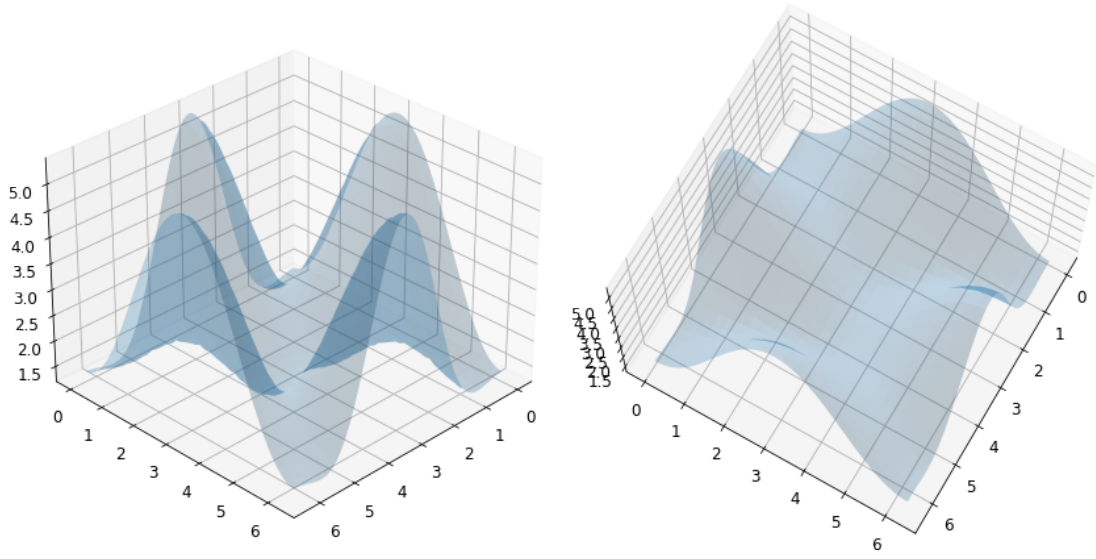
[75]:
```
fig = plt.figure(figsize=(8,6))

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=-pi, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-pi, cmap=cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*pi, cmap=cm.coolwarm)

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



Change the view angle

We can change the perspective of a 3D plot using the `view_init` function, which takes two arguments: the elevation and the azimuth angles (unit degrees)

[76]:
```
fig = plt.figure(figsize=(12,6))

ax = fig.add_subplot(1,2,1, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(30, 45)

ax = fig.add_subplot(1,2,2, projection='3d')
ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
ax.view_init(70, 30)

fig.tight_layout()
```



Animations

Matplotlib also includes a simple API for generating animations for sequences of figures. Using the FuncAnimation function we can generate a movie file from a sequences of figure. The function takes the following arguments: fig a figure canvas, func is a function that we provide which updates the figure, init_func is a function we provide to setup the figure, frame is the number of frames to generate, blit tells the animation function to only update parts of the frame that has changed (giving better smoother animations):

def init(): # setup figure

def update(frame_counter): # update figure for new frame

anim = animation.FuncAnimation(fig, update, init_func=init, frames=200, blit=True)

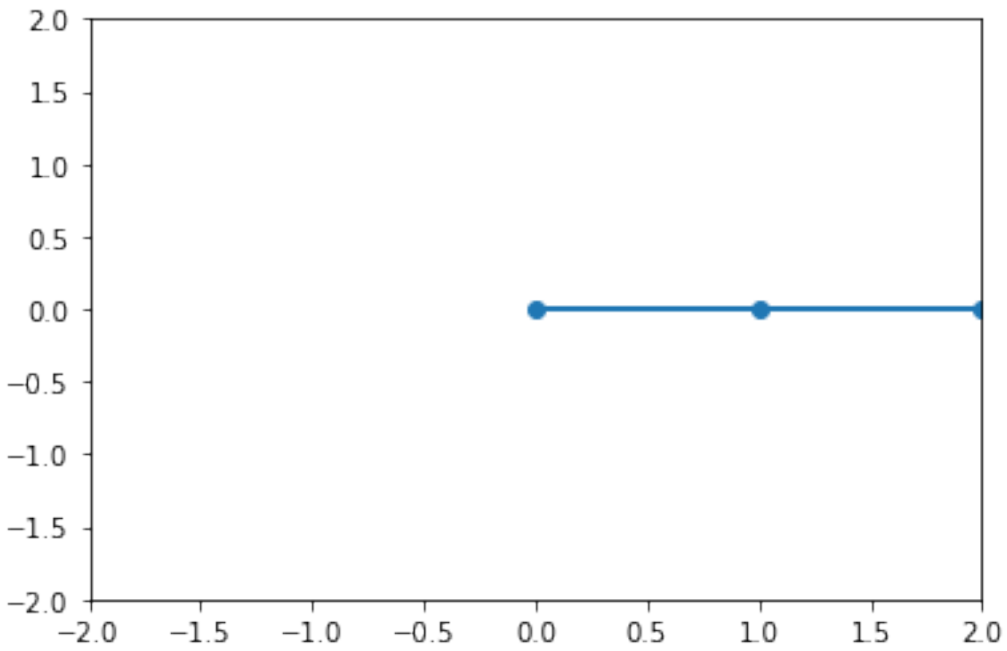
anim.save('animation.mp4', fps=30) # fps = frames per second To use the animation features in matplotlib we first need to import the module matplotlib.animation:

[77]:
```
from matplotlib import animation
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

# Constants
g = 9.81  # Acceleration due to gravity (m/s^2)
L1 = 1.0  # Length of the first pendulum arm (m)
L2 = 1.0  # Length of the second pendulum arm (m)
m1 = 1.0  # Mass of the first pendulum bob (kg)
m2 = 1.0  # Mass of the second pendulum bob (kg)
```

```
# Initial conditions [theta1, omega1, theta2, omega2]
initial_conditions = [np.pi / 2, 0, np.pi / 2, 0]

# Time parameters
t0 = 0
tf = 20  # Total simulation time (s)
dt = 0.05  # Time step (s)
t = np.arange(t0, tf, dt)

# Function to compute the derivatives of the state variables
def derivatives(state, t):
    theta1, omega1, theta2, omega2 = state

    # Equations of motion for the double pendulum
    delta_theta = theta2 - theta1
    denom1 = (m1 + m2) * L1 - m2 * L1 * np.cos(delta_theta) ** 2
    denom2 = (L2 / L1) * denom1

    dtheta1_dt = omega1
    domega1_dt = ((m2 * L1 * omega1 ** 2 * np.sin(delta_theta) * np.
 ↪cos(delta_theta) +
                  m2 * g * np.sin(theta2) * np.cos(delta_theta) +
                  m2 * L2 * omega2 ** 2 * np.sin(delta_theta) -
                  (m1 + m2) * g * np.sin(theta1)) / denom1)
    dtheta2_dt = omega2
    domega2_dt = ((-(m2 + m1) * L1 * omega1 ** 2 * np.sin(delta_theta) * np.
 ↪cos(delta_theta) +
                  (m1 + m2) * g * np.sin(theta1) * np.cos(delta_theta) -
                  (m1 + m2) * L2 * omega2 ** 2 * np.sin(delta_theta) -
                  (m1 + m2) * g * np.sin(theta2)) / denom2)

    return [dtheta1_dt, domega1_dt, dtheta2_dt, domega2_dt]

# Use scipy's odeint to solve the differential equations
from scipy.integrate import odeint
states = odeint(derivatives, initial_conditions, t)

# Extract theta1 and theta2
theta1_values, _, theta2_values, _ = states.T

# Convert to Cartesian coordinates
x1 = L1 * np.sin(theta1_values)
y1 = -L1 * np.cos(theta1_values)
x2 = x1 + L2 * np.sin(theta2_values)
y2 = y1 - L2 * np.cos(theta2_values)

# Create a function to update the animation
```

```
def update(frame):
    line.set_data([0, x1[frame], x2[frame]], [0, y1[frame], y2[frame]])
    return line,

# Create a figure and axis
fig, ax = plt.subplots()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)

# Create a line to represent the double pendulum
line, = ax.plot([], [], 'o-', lw=2)

# Create the animation
animation = FuncAnimation(fig, update, frames=len(t), blit=True)

# Display the animation (note: this may not work in some environments)
plt.show()

# Save the animation as an MP4 video (uncomment this line if needed)
# animation.save('double_pendulum_animation.mp4', writer='ffmpeg', fps=30)
```



Note that when we use an interactive backend, we need to call plt.show() to make the figure appear on the screen.

1.5.4 Further reading

- http://www.matplotlib.org - The project web page for matplotlib.
- https://github.com/matplotlib/matplotlib - The source code for matplotlib.
- http://matplotlib.org/gallery.html - A large gallery that showcase what kind of plots matplotlib can create. Highly recommended!
- http://www.loria.fr/~rougier/teaching/matplotlib - A good matplotlib tutorial.

1.6 Versions

%version_information numpy, scipy, matplotlib