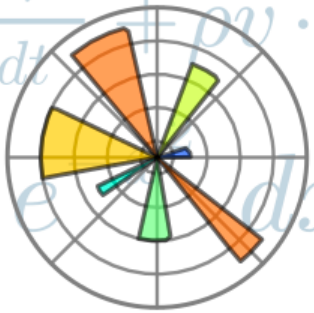
## Goals for Today:

Matplotlib:

- Syntax
- plot()
- Basic plot functions
- “Current” Concept
- Subplots



matplotlib

# Matplotlib - Overview

Matplotlib is the most common data visualization library for python.

There are several other newcomers that are ascendant (some even built on top of matplotlib), but matplotlib is likely here to stay and is capable of doing everything you will need for this course.

We will primarily utilize the matplotlib sub-module called `pyplot`. This is an interface to MATLAB style plotting functions that will familiar to many of the students in this course.

Thus in all of the following examples we will assume the following import call has been performed:

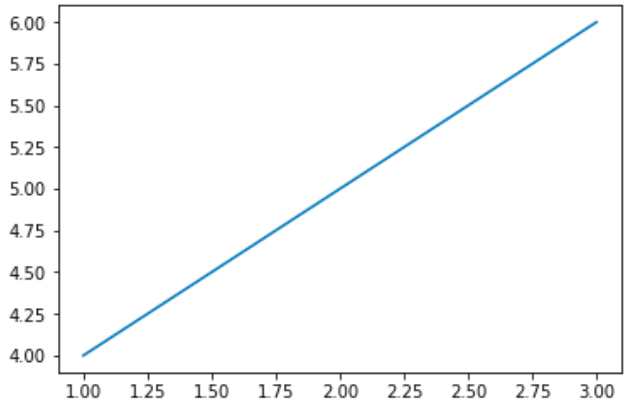
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

Matplotlib does not require numpy or pandas, but it is capable of plotting data in their classes, so we'll import them as well.

# The `plt.plot()` function

Making your first plot in matplotlib can be accomplished with just two function calls:

```
plt.plot([1,2,3],[4,5,6])
plt.show()
```




Here we have passed the `plt.plot()` function data for `x` (first) and data for `y` (second) and used the `plt.show()` function to display the result.

**A couple important things are illustrated by this example:**

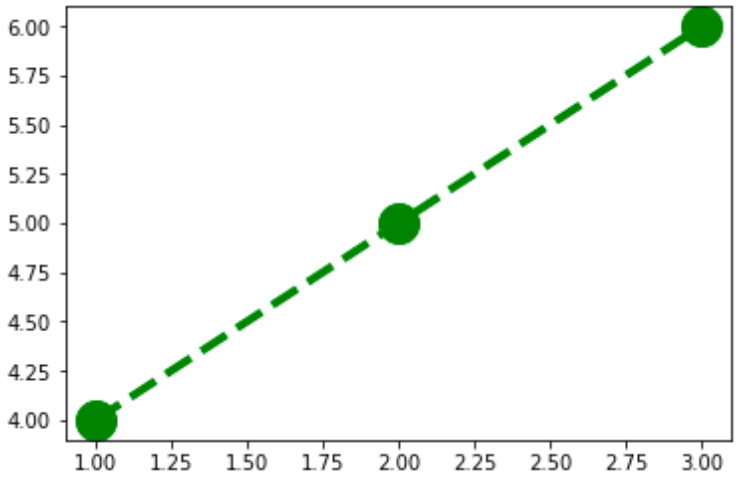
**(1)** Every change to the plot (including displaying or saving it) is done through a pyplot function or by modifying a figure/axis attribute.

**(2)** matplotlib tries to keep things simple and so it automatically guessed sensible x and y ranges for the plot and default colors.

# The `plt.plot()` function

Every component of the plot can be modified; many through the `plt.plot()` arguments:

```
plt.plot([1,2,3],[4,5,6],color='green', marker='o',\
         linestyle='dashed',linewidth=4.0,markersize=20)
plt.show()
```




Some things can only be changed using additional functions or methods of the figure objects.

You'll see many examples in this lecture, but you will need to familiarize yourself with matplotlib documentation to fine-tune things.

# Current figure and axis concepts

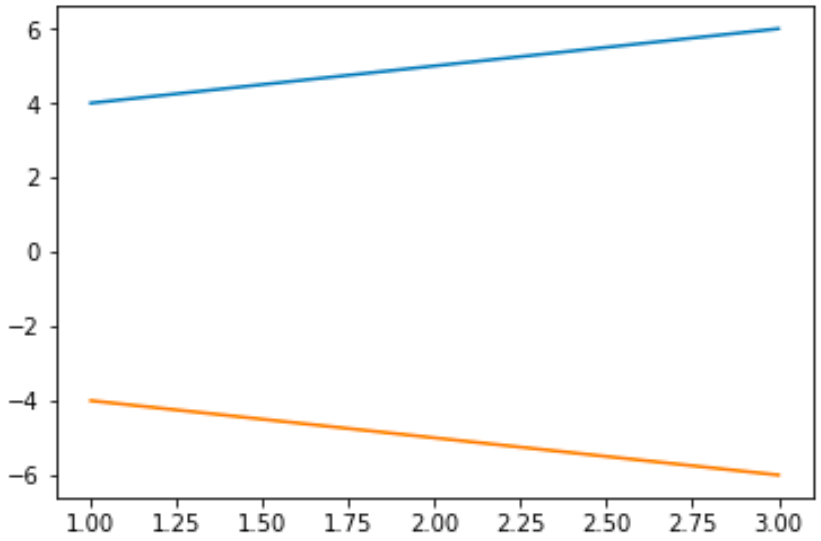
Matplotlib keeps track of the **current figure** and **current axes**.

By default these are set to the last of each that were created.

This is how matplotlib functions like `plt.plot()` know where to put things.

For example:

```
plt.plot([1,2,3],[4,5,6])
plt.plot([1,2,3],[-4,-5,-6])
plt.show()
```

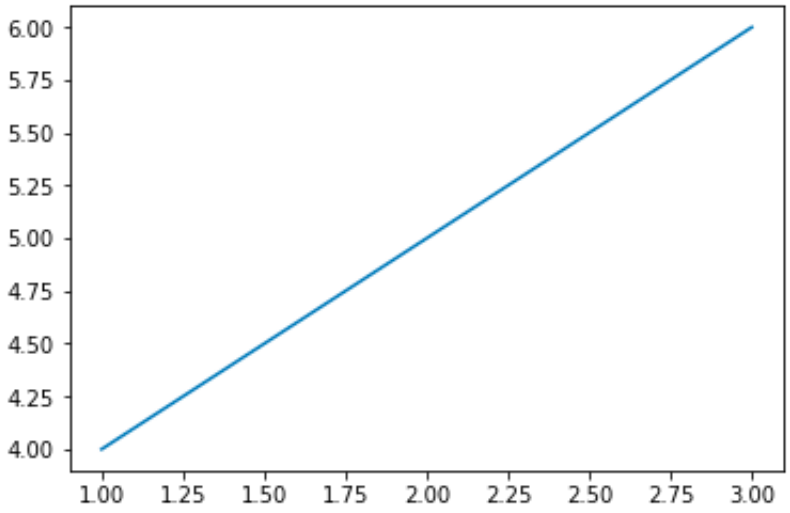


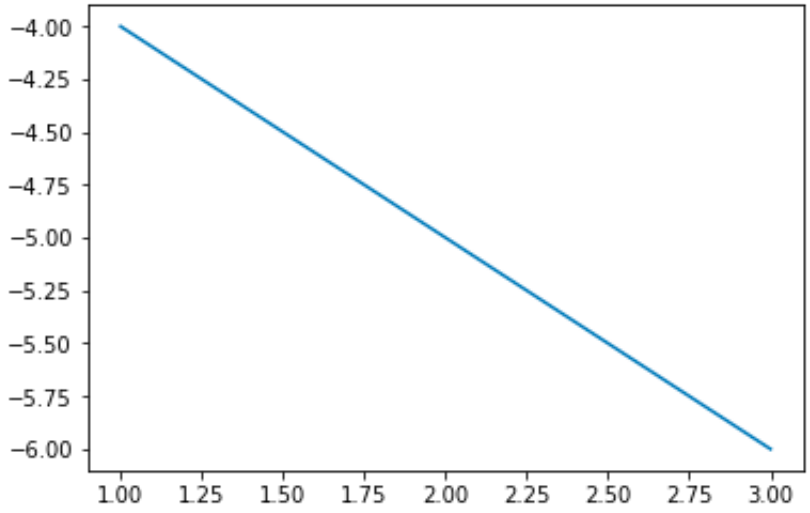

This example results in two lines in the same figure/set of axes. Unless we create a new figure, `matplotlib.pylab` functions will continue adding/modifying the **current figure and axes**.

# Current figure and axis concepts

Compare the previous example with the following:

```
plt.figure() # Explicitly creates a new figure
plt.plot([1,2,3],[4,5,6])
plt.figure() # Explicitly creates a new figure
plt.plot([1,2,3],[-4,-5,-6])
plt.show() # Shows all figures initialized by plt
```



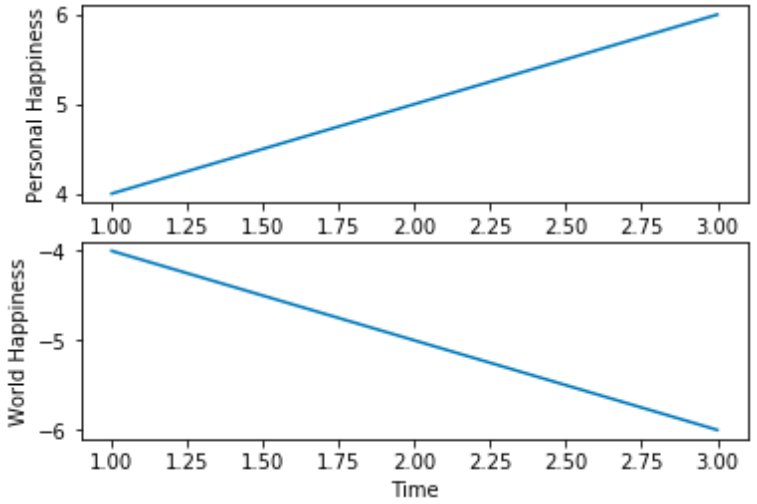




Since we explicitly create new figures between plt.plt() calls, the lines are on separate figures.

# Current figure and axis concepts

The **current axes** and **current figure** were effectively the same thing in the previous examples. This is not the case when we make subplots:

```
ax1 = plt.subplot(211)
ax1.plot([1,2,3],[4,5,6])
ax1.set_xlabel("Time")
ax1.set_ylabel("Personal Happiness")
ax2 = plt.subplot(212)
ax2.plot([1,2,3],[-4,-5,-6])
ax2.set_xlabel("Time")
ax2.set_ylabel("World Happiness")
plt.show()
```




The `subplot()` function specifies `numrows, numcols, plot_number` where `plot_number` ranges from `1` to `numrows*numcols`. When we call the subplot function we create numrows*numcols subplots inside a single figure.

The subplot function creates `numrows*numcols` axes inside a single figure.

We can address each set of axes individually by assigning them to pointers (e.g., `ax1` and `ax2` in the example). The axes have their own attributes and methods (`ax.set_xlabel()` etc.) that can be used to configure the plot.

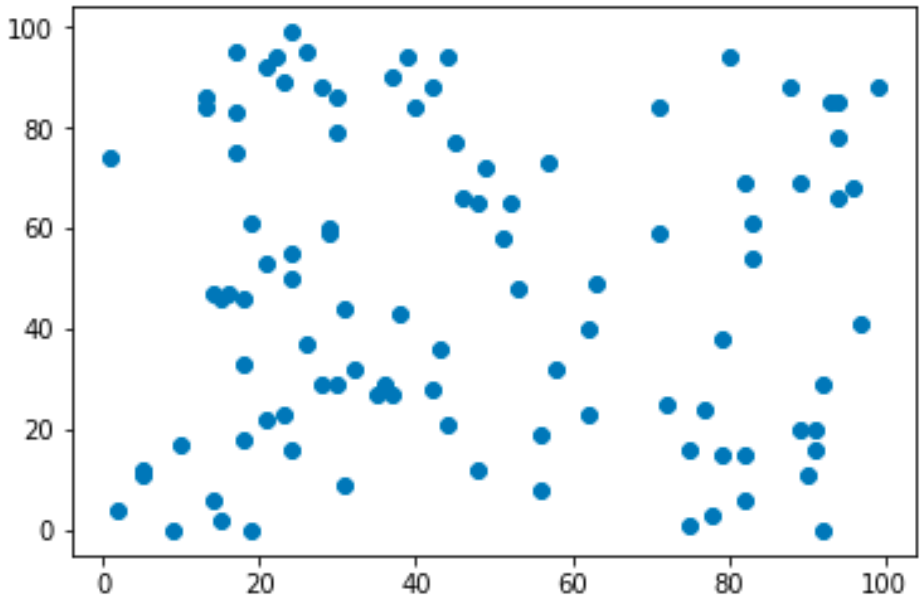
# Basic Plot Types – `plt.scatter()`

matplotlib.pyplot comes with many elementary plotting functions for different types of figures.

A common subset includes `.plot()`, `.scatter()`, `.bar()`, `.hist()` for lineplots, scatterplots, bar charts, and histograms, respectively.

```
np.random.seed(15422) # for reproducibility
plt.scatter(np.random.randint(0,100,size=100),np.random.randint(0,100,size=100))
plt.show()
```

A similar effect can be achieved with `plt.plot()` using markers and `linestyle=None`

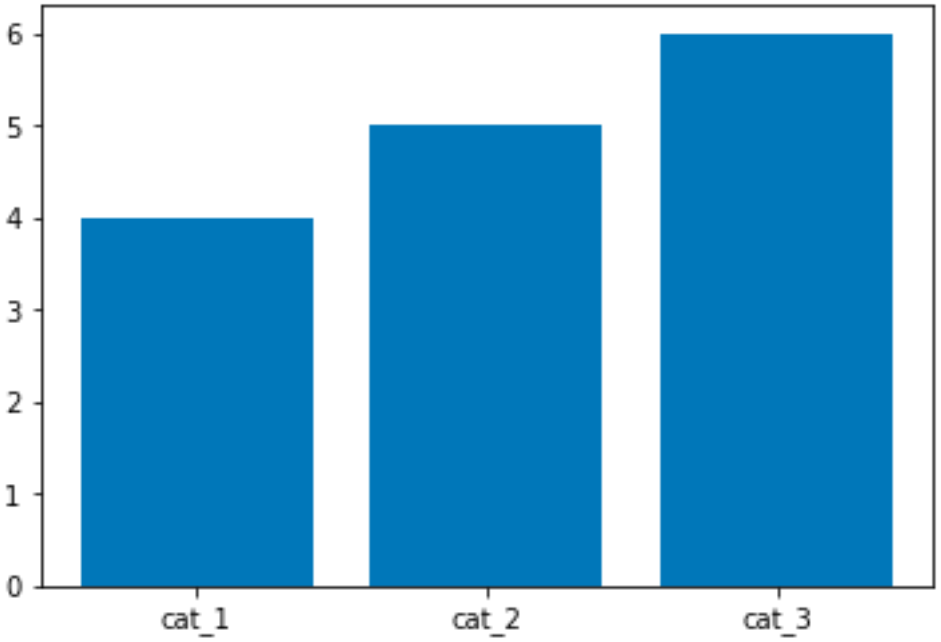


# Basic Plot Types – `plt.bar()`

matplotlib.pyplot comes with many elementary plotting functions for different types of figures.

A common subset includes `.plot()`, `.scatter()`, `.bar()`, `.hist()` for lineplots, scatterplots, bar charts, and histograms, respectively.

```
plt.bar(["cat_1","cat_2","cat_3"],[4,5,6])
plt.show()
```



# Basic Plot Types – `plt.hist()`

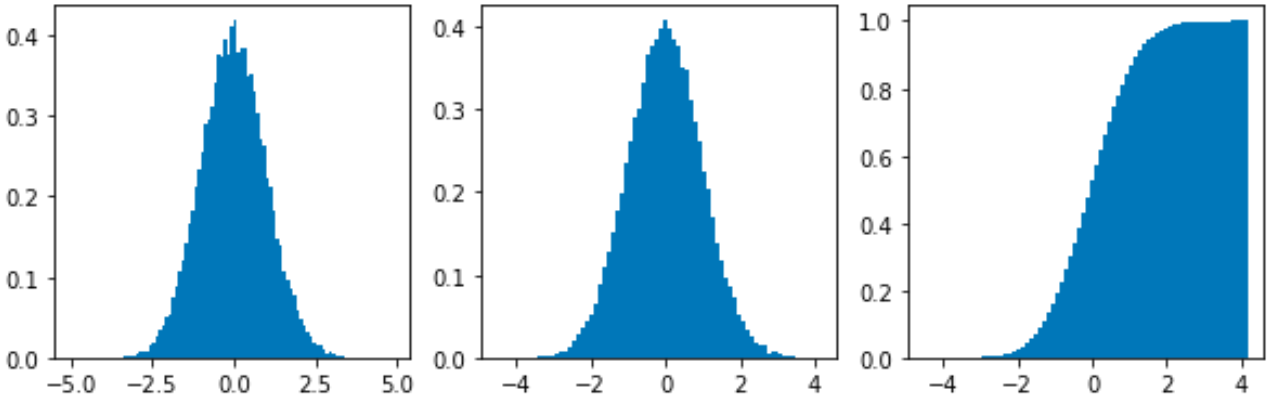
matplotlib.pyplot comes with many elementary plotting functions for different types of figures.

A common subset includes `.plot()`, `.scatter()`, `.bar()`, `.hist()` for lineplots, scatterplots, bar charts, and histograms, respectively.

```
plt.figure(figsize=(10, 3))
np.random.seed(15422) # for reproducibility
data = np.random.normal(size=50000)
ax1 = plt.subplot(131)
n, bins, patches = ax1.hist(data,bins=np.arange(-5,5,0.1),density=1)
ax2 = plt.subplot(132)
n, bins, patches = ax2.hist(data,density=1,bins='rice')
ax3 = plt.subplot(133)
n, bins, patches = ax3.hist(data,density=1,bins='rice',cumulative=True)
plt.show()
```

`bins` can be set manually or with several built-in algorithms.

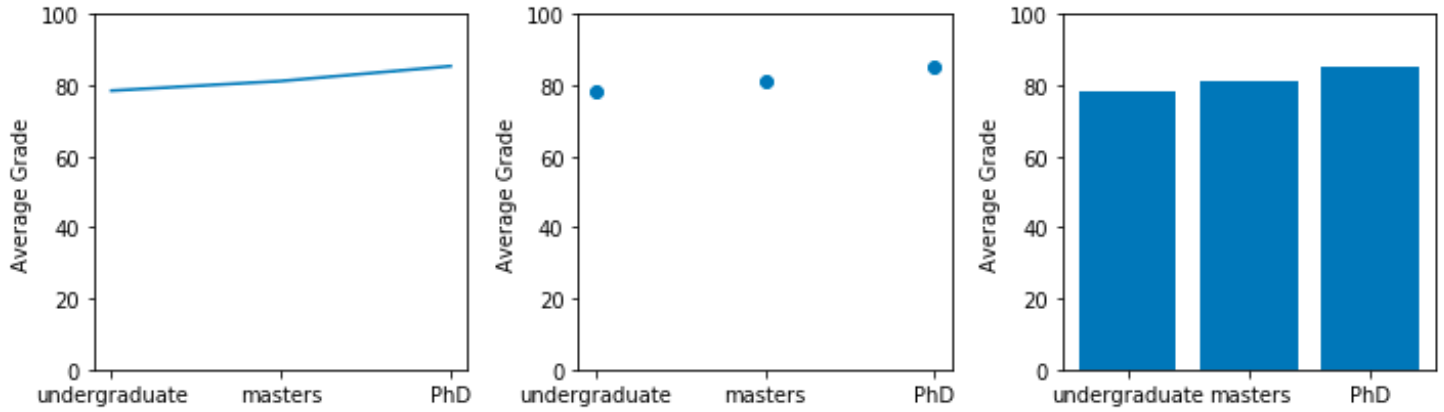
`density` controls normalization.



# Plotting Categorical Data

Matplotlib is flexible in plotting catgorical vs numerical data. We can simply supply categorical data along either axis:

```
plt.figure(figsize=(10, 3))
ax1 = plt.subplot(131)
ax1.plot(["undergraduate","masters","PhD"],[78.5,81.2,85.4])
ax1.set_ylim(0,100)
ax1.set_ylabel("Average Grade")
ax2 = plt.subplot(132)
ax2.scatter(["undergraduate","masters","PhD"],[78.5,81.2,85.4])
ax2.set_ylabel("Average Grade")
ax2.set_ylim(0,100)
ax3 = plt.subplot(133)
ax3.bar(["undergraduate","masters","PhD"],[78.5,81.2,85.4])
ax3.set_ylabel("Average Grade")
ax3.set_ylim(0,100)
plt.tight_layout()
plt.show()
```

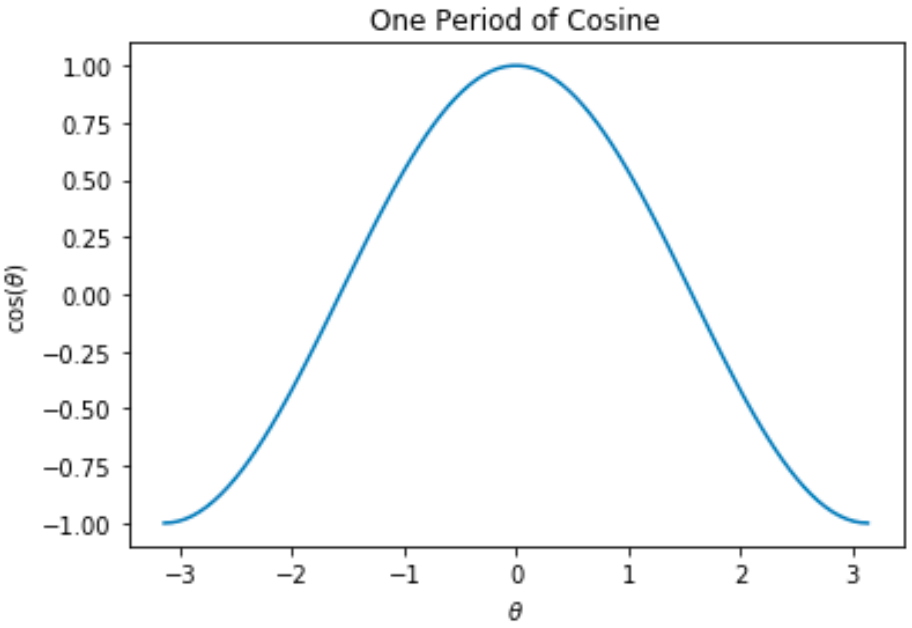


# Formatting - Labels

You can add an x label, y label, and title to your plot with the `plt.xlabel()`, `plt.ylabel()`, and `plt.title()` functions, respectively:

```
x = np.linspace(-np.pi,np.pi,100)
plt.plot(x,np.cos(x))
plt.xlabel(r"$\theta$")
plt.ylabel(r"cos($\theta$)")
plt.title("One Period of Cosine")
plt.show()
```

In this example we have used a "raw" string (with prefix `r" "`) to enable latex style specification of special symbols.

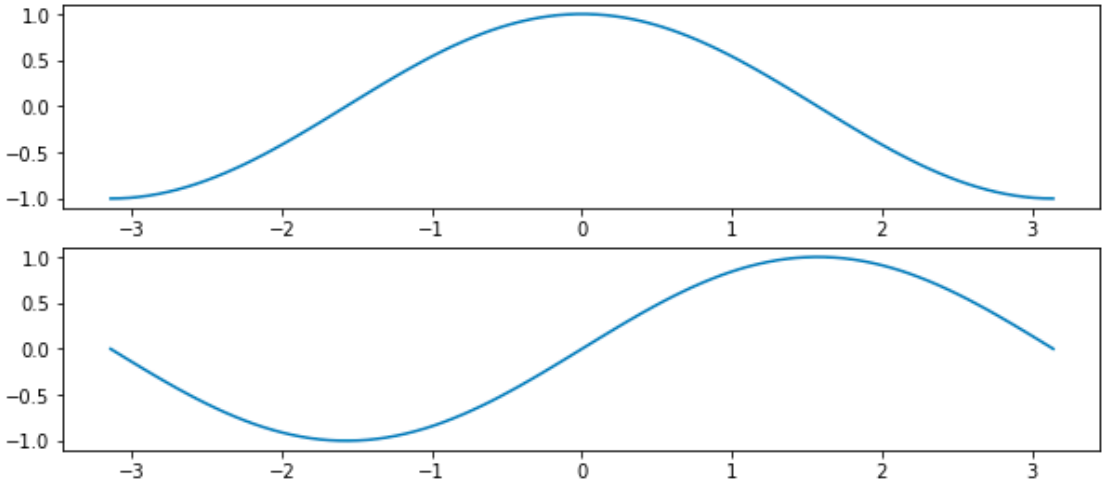


# Formatting - Resizing

You can change the dimensions of your figures by using the `plt.figure(figsize=(height,width))` function where `height` and `width` are specified in inches by default.

You can call this function before plotting to create a new figure, followed by plotting functions like `plt.plot()` which will be rendered on the figure:

```
x = np.linspace(-np.pi,np.pi,100)
plt.figure(figsize=(10,2))
plt.plot(x,np.cos(x))
plt.figure(figsize=(10,2))
plt.plot(x,np.sin(x))
plt.show()
```



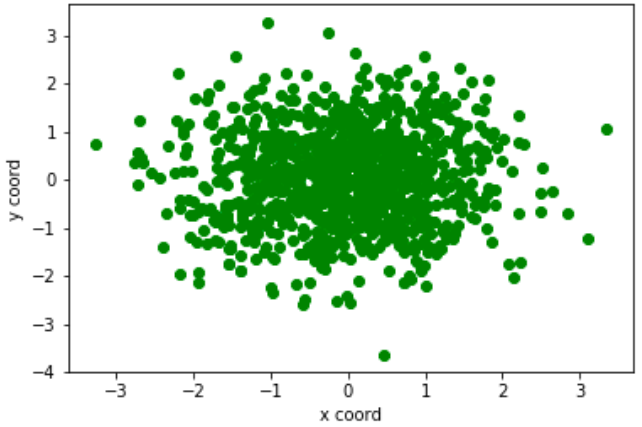
# Formatting - Colors

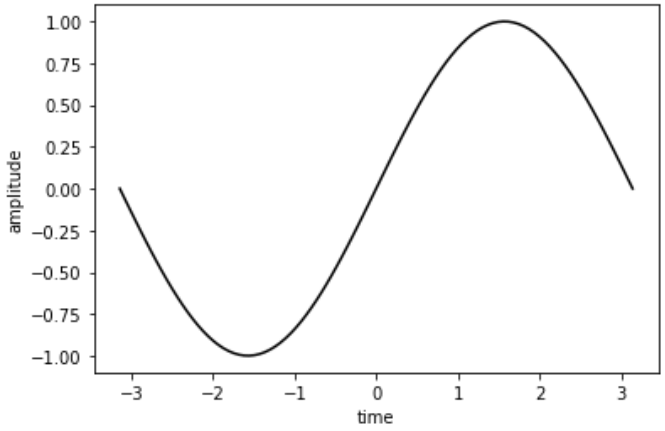
All of the built in plotting functions provide ways of modifying the color via function arguments.

A number of built-in colors are aliased to strings (e.g., "b" means default blue color and "k" means black):

```
# Scatter plot with green markers
a = np.random.normal(size=(1000,2))
plt.scatter(a[:,0],a[:,1],c="g")
plt.xlabel("x coord")
plt.ylabel("y coord")
plt.show()

# Lineplot with a black line
x = np.linspace(-np.pi,np.pi,100)
plt.plot(x,np.sin(x),color="k")
plt.xlabel("time")
plt.ylabel("amplitude")
plt.show()
```

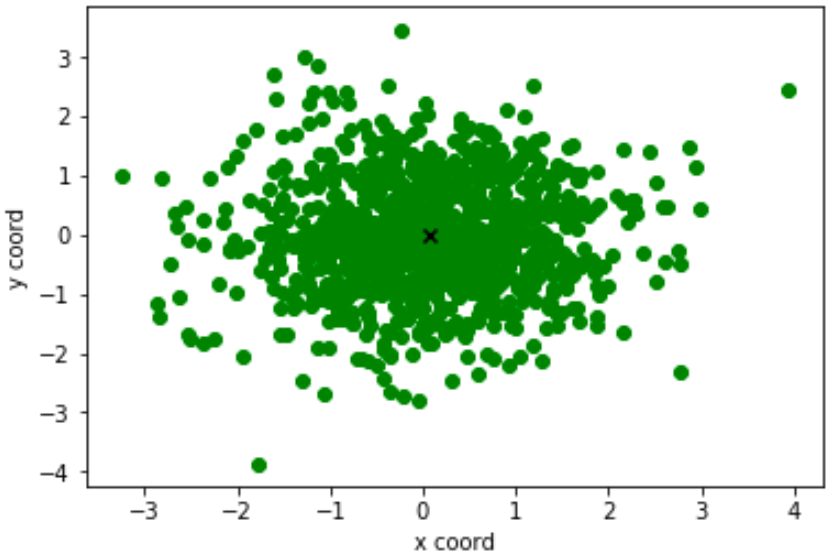


# Formatting - Colors

If you want to change the color of certain quantities, it is usually the most straightforward to use separate plotting calls with the corresponding color supplied:

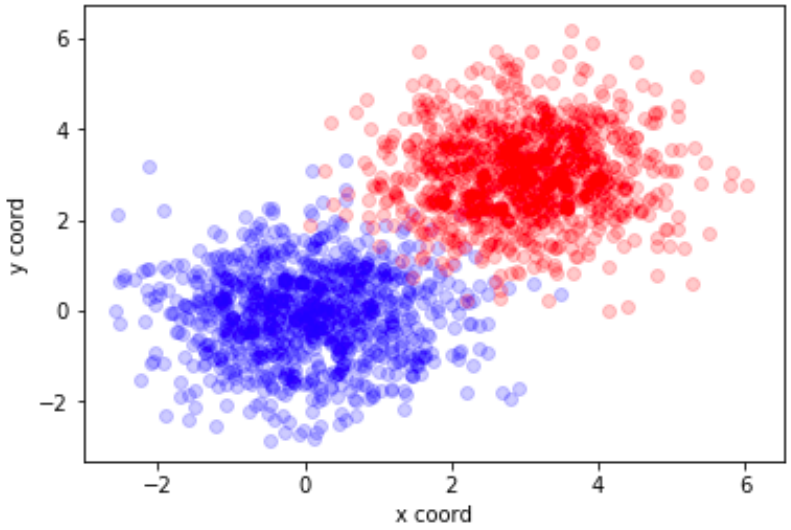
```
# Scatter plot with green markers and a black x for mean
a = np.random.normal(size=(1000,2))
plt.scatter(a[:,0],a[:,1],c="g") # scatter green
plt.scatter(a[:,0].mean(),a[:,1].mean(),c='k',marker="x")
plt.xlabel("x coord")
plt.ylabel("y coord")
plt.show()
```



# Formatting - Colors

The `alpha` argument can be used to set the transparency of objects (`alpha=1.0` is opaque, default):

```
# Scatter plot with transparency
np.random.seed(915432) # for reproducibility
a = np.random.normal(size=(1000,2))
b = np.random.normal(loc=3.0,scale=1.0,size=(1000,2))
plt.scatter(a[:,0],a[:,1],c="b",alpha=0.2) # scatter blue
plt.scatter(b[:,0],b[:,1],c="r",alpha=0.2) # scatter red
plt.xlabel("x coord")
plt.ylabel("y coord")
plt.show()
```

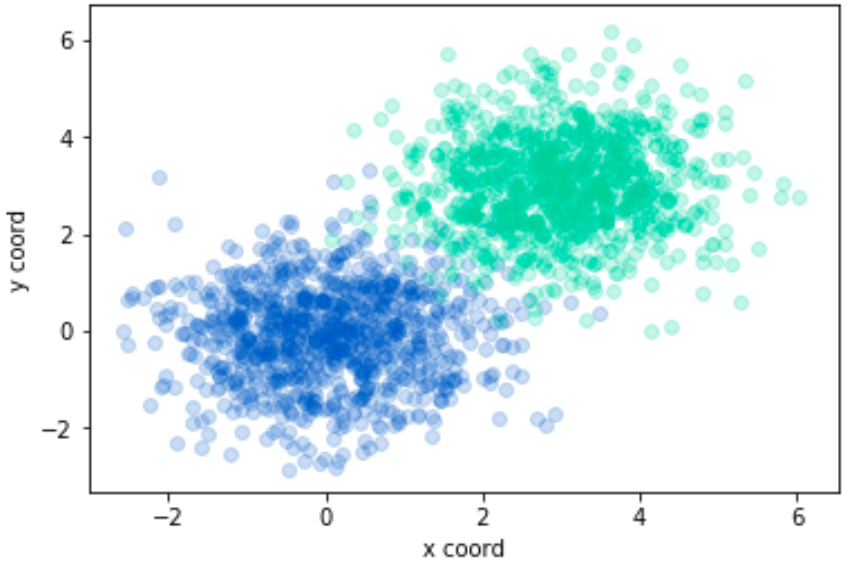


# Formatting - Colors

You can also use specific rgba tuples to set the color and transparency of your choice:

```
# Scatter plot with transparency
np.random.seed(915432) # for reproducibility
a = np.random.normal(size=(1000,2))
b = np.random.normal(loc=3.0,scale=1.0,size=(1000,2))
plt.scatter(a[:,0],a[:,1],c=np.array([[0.05,0.35,0.75,0.2]])) # scatter blue
plt.scatter(b[:,0],b[:,1],c=np.array([(0.05,0.8,0.6,0.2)])) # scatter green
plt.xlabel("x coord")
plt.ylabel("y coord")
plt.show()
```

**Note:** we use a 2D array with a single value to suppress a warning, but just putting a single value will also work.



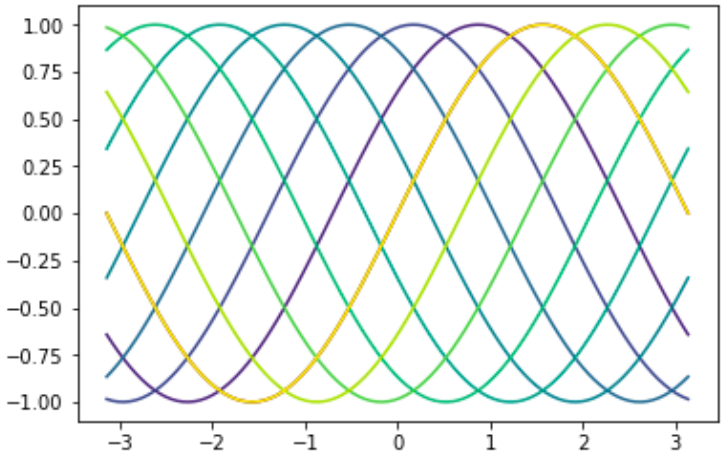
# Formatting - Colors

You can also use built in colormaps to select colors using the `cm` submodule of matplotlib. The simplest way to access colors is to use the `cm.colormap_name(value)` method.

`colormap_name` can be any of the many built-in colormaps (e.g., `viridis` for continuous or `RdBu` for divergent).

```
from matplotlib import cm
colors = [ cm.viridis(x) for x in np.linspace(0,1,10) ] # list of colors
x = np.linspace(-np.pi,np.pi,100)
for count_i,i in enumerate(np.linspace(0,2.0*np.pi,10)):
  plt.plot(x,np.sin(x+i),color=colors[count_i]) # plot with current color
plt.show()
```

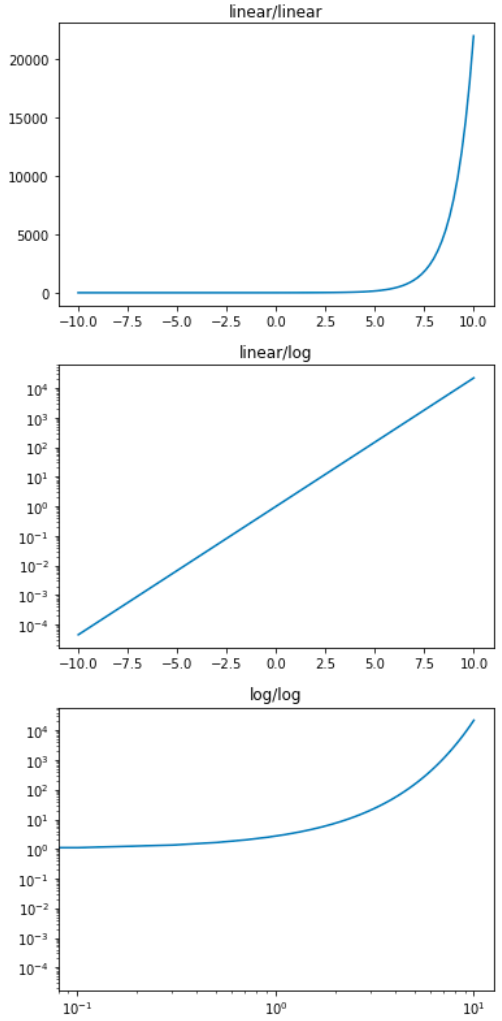
By default the colormap is normalized to values between 0 and 1, so you could call it with `cm.viridis(0.5)` and the midpoint color would be returned.



# Formatting – Axis Scaling

The `plt.xscale()` and `plt.yscale()` functions can be used to change a plot from linear (`'linear'`, default) scaling to logarithmic (`'log'`):

```
x = np.linspace(-10,10,100)
plt.figure()
plt.plot(x,np.exp(x))
plt.title('linear/linear')
plt.figure()
plt.plot(x,np.exp(x))
plt.yscale('log')
plt.title('linear/log')
plt.figure()
plt.plot(x,np.exp(x))
plt.xscale('log')
plt.yscale('log')
plt.title('log/log')
plt.show()
```



# Example – Connecting with Numpy

Let's finish the demonstration of matplotlib by working through a short linear regression example using numpy. This will plant the seed for our later lecture on more sophisticated regression while also reinforcing what we recently learned about numpy.

I have distributed a data file (`data.txt`) with the in class notebook. The file contents look like this:

```
x             y
-1.555556      1.053130
 1.717172      4.239924
 0.101010      0.594050
-1.070707      0.629858
-0.585859      0.405727
 1.757576      4.810278
-1.838384      1.556490
 1.030303      2.069449
 0.222222      0.685768
 1.878788      5.017124
-0.949495      0.567816
-0.464646      0.377200
…
```

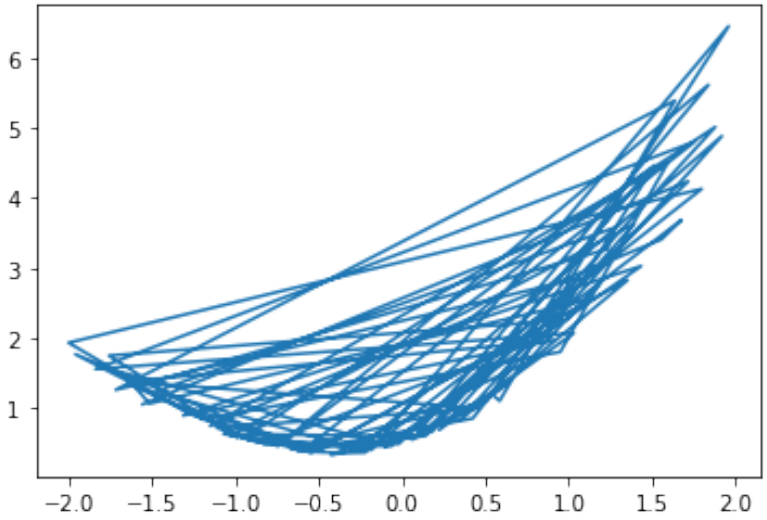
**Note:**
Space-delimited
Two-columns
A one-line header

# Example – Connecting with Numpy

Let's finish the demonstration of matplotlib by working through a short linear regression example using numpy. This will plant the seed for our later lecture on more sophisticated regression while also reinforcing what we recently learned about numpy.

Let's read the data into our notebook using `np.loadtxt` and plot it:

```
data = np.loadtxt('data.txt',skiprows=1)
plt.figure()
plt.plot(data[:,0],data[:,1])
plt.show()
```



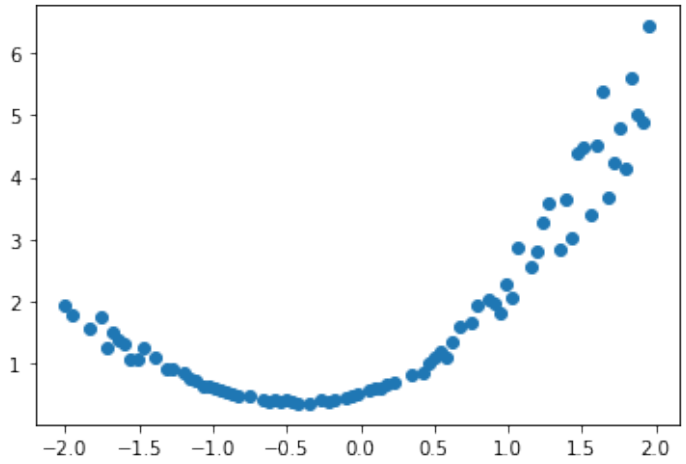

**Note:**
Data isn't sorted by x. This creates the zigzag effect.

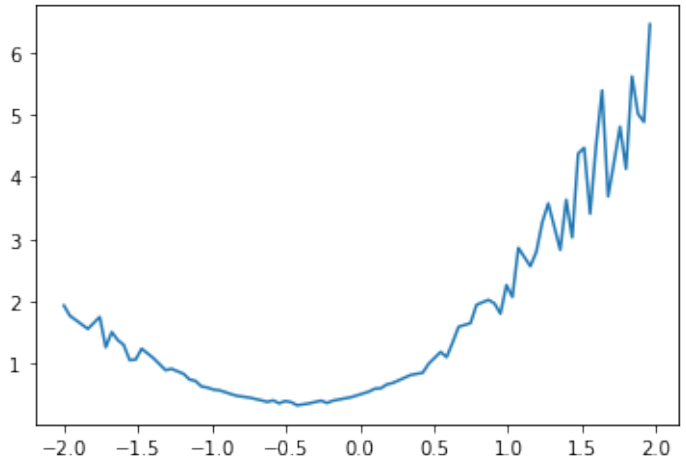
# Example – Connecting with Numpy

We can fix this by either plotting the scatter of the data, or using `np.argsort()`:

```
plt.figure()
plt.scatter(data[:,0],data[:,1]) # Easy solution, just use the points, no
lines
plt.show()

inds = np.argsort(data[:,0])
plt.plot(data[inds,0],data[inds,1]) # Sort the (x,y) pairs by the x values
plt.show()
```




Our data looks like it exhibits a parabolic dependence. Let's use this opportunity to refresh on linear least-squares regression and fit a cubic polynomial to our data.

# Example – Connecting with Numpy

In linear regression we parameterize a model of the form:

$$\mathbf{y}_p = \beta_0 + \mathbf{x}_1\beta_1 + \ldots + \mathbf{x}_n\beta_n = \beta\mathbf{x}$$

*Implied "1" in first element of **x** matrix

The least squares solution corresponds to selecting parameters such that the square of the residuals is minimized:

$$\arg\min_{\beta} \sum_i \left(y_i - \mathbf{x}_i\beta\right)^2$$

This can be solved by taking the derivative and solving for the extremum. Or more concisely:

$$\mathbf{x}\beta = \mathbf{y} \xrightarrow{*} \beta = \left(\mathbf{x}^{\mathrm{T}}\mathbf{x}\right)^{-1}\mathbf{x}^{\mathrm{T}}\mathbf{y}$$

*For tall rank deficient matrices $\mathbf{x}^{-1}$ doesn't exist but the pseudo-inverse $(\mathbf{x}^{\mathrm{T}}\mathbf{x})^{-1}$ yields the least squares solution.

# Example – Connecting with Numpy

Adapting this to our current problem (two ways):

```
# Make our X matrix
X = np.ones([len(data),4])
X[:,1] = data[:,0]
X[:,2] = data[:,0]**(2.0)
X[:,3] = data[:,0]**(3.0)

# Calculate our parameters using the lstsq() function
beta_1 = np.linalg.lstsq(X.T.dot(X),X.T.dot(data[:,1]))[0] # Check docs to see all of the returns
print("parameters for method 1: {}".format(beta_1))

# Calculate our parameters using the inverse
beta_2 = np.dot(np.dot(np.linalg.inv(X.T.dot(X)),X.T),data[:,1])
print("parameters for method 2: {}".format(beta_2))

# Make y predictions for our two models
x_grid = np.linspace(min(data[:,0]),max(data[:,0]),100) # some x values that span the data range.
y_1 = beta_1[0] + beta_1[1]*x_grid + beta_1[2]*x_grid**(2.0) + beta_1[3]*x_grid**(3.0)
y_2 = beta_2[0] + beta_2[1]*x_grid + beta_2[2]*x_grid**(2.0) + beta_2[3]*x_grid**(3.0)

# Plot our predictions against the data
plt.figure(figsize=(5,5))
plt.scatter(data[:,0],data[:,1],marker='o',label="data")
plt.plot(x_grid,y_1,label="lstsq()")
plt.plot(x_grid,y_1,label="inv()")
plt.legend()
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

```
parameters for method 1: [0.54831474 0.78128642 0.84947248 0.04891569]
parameters for method 2: [0.54831474 0.78128642 0.84947248 0.04891569]
```

# Example – Connecting with Numpy

Adapting this to our current problem (two ways):

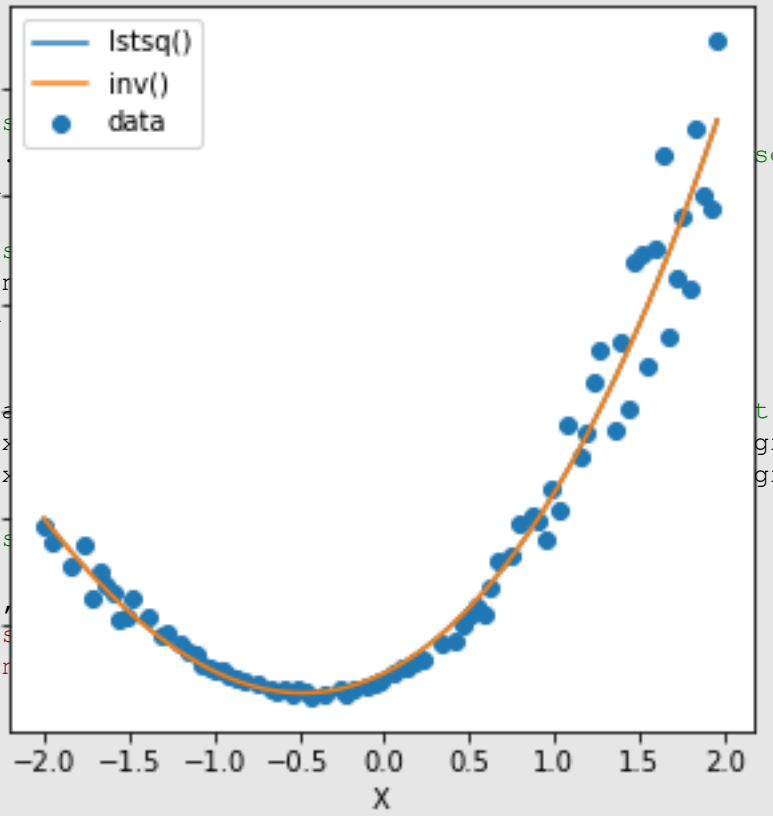
```
# Make our X matrix
X = np.ones([len(data),4])
X[:,1] = data[:,0]
X[:,2] = data[:,0]**(2.0)
X[:,3] = data[:,0]**(3.0)

# Calculate our parameters us
beta_1 = np.linalg.lstsq(X.T.                                    see all of the returns
print("parameters for method_

# Calculate our parameters us
beta_2 = np.dot(np.dot(np.lin
print("parameters for method

# Make y predictions for our
x_grid = np.linspace(min(data                                    t span the data range.
y_1 = beta_1[0] + beta_1[1]*x                                    grid**(3.0)
y_2 = beta_2[0] + beta_2[1]*x                                    grid**(3.0)

# Plot our predictions agains
plt.figure(figsize=(5,5))
plt.scatter(data[:,0],data[:,
plt.plot(x_grid,y_1,label="ls
plt.plot(x_grid,y_1,label="in
plt.legend()
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```




```
parameters for method 1: [0.54831474 0.78128642 0.84947248 0.04891569]
parameters for method 2: [0.54831474 0.78128642 0.84947248 0.04891569]
```