# Python Packages for Machine Learning

## TOC

## Numpy

NumPy is short for **Num**erical **Py**thon:

- Important package for Scientific Computing
- N-dimensional array object
- More efficient than python objects
- Also has Mathematical Functions
- Documentation (https://docs.scipy.org/doc/numpy/)

Let's get started by importing the module

```
In [1]: # Import NumPy
        import numpy as np
```

## N-Dimensional Arrays

N-dimensional arrays (ndarray), typically just called arrays, are the core data structure in Numpy. Let's get started by creating some ndarrays

### 1-D Arrays

We can convert a list of sequence of data to a numpy array as follows:

```
In [2]: test_array = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
        test_array

Out[2]: array([1., 2., 3., 4., 5.])
```

We can check the shape of the array via the following attribute; it returns a tuple describing the shape.

```
In [3]: test_array.shape

Out[3]: (5,)
```

### 2D Arrays

Arrays can be multi dimensional. Below we create an array with 2 rows and 5 columns.

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

Out[4]: array([[ 1,  2,  3,  4,  5],
               [ 6,  7,  8,  9, 10]])

In [5]: two_dim_test_array.shape

Out[5]: (2, 5)
```

We can also check out the **ndim** attribute to find the number of dimensions in our array

```
In [6]: two_dim_test_array.ndim

Out[6]: 2
```

## Datatypes in NumPy Arrays

Numpy arrays datatypes are homogenous. Datatypes available include:

- int8, uint8
- int16, uint16
- int32, uint32
- int64, uint64
- float16
- float32
- plus more!

Below we check the data type of an array and change it:

```
In [7]:  test_array
Out[7]:  array([1., 2., 3., 4., 5.])
```

```
In [8]:  test_array.dtype
Out[8]:  dtype('float64')
```

Data types can also be specified at creation time.

```
In [9]:  test_array_128 = np.array(test_array, dtype='float128')
```

```
In [10]: test_array_128.dtype
Out[10]: dtype('float128')
```

## Creating Numpy Arrays with Builtin Functions

Numpy has built in functions which automatically create arrays.

Below we demo one option, `np.arange` , which is essentially the same as range in Python

`np.arange(start, stop, step_size)`

```
In [11]: test_array = np.arange(1.0, 18., 2.3184)
         print(test_array)

         test_array.shape

         [ 1.      3.3184  5.6368  7.9552 10.2736 12.592  14.9104 17.2288]
Out[11]: (8,)
```

## Reshaping arrays

We can reshape arrays as follows

```
In [12]: print(test_array.reshape(4,2))
```

```
[[ 1.      3.3184]
 [ 5.6368  7.9552]
 [10.2736 12.592 ]
 [14.9104 17.2288]]
```

```
In [13]: print(len(test_array.shape))
```

```
1
```

Note, the `reshape()` function itself does not modify original array. Need to assign the returned value to a variable.

```
In [14]: test_array = test_array.reshape(4,2)
```

```
In [15]: test_array.shape
```

```
Out[15]: (4, 2)
```

## Broadcasting arrays

Below we demo some of the broadcasting rules for numpy arrays. To get started lets create a few arrays

```
In [16]: array1=np.array([1,2,3,4,5,6,7,8])
         array2=2* np.arange(1,9,1)
         array2
```

```
Out[16]: array([ 2,  4,  6,  8, 10, 12, 14, 16])
```

### Combining Two 1D Arrays

```
In [17]: array1 + array2
```

```
Out[17]: array([ 3,  6,  9, 12, 15, 18, 21, 24])
```

```
In [18]: array1 / array2
```

```
Out[18]: array([0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5])
```

### Broadcasting for Scalars and Arrays

In [19]:
```
2*array1
```
Out[19]:
```
array([ 2,  4,  6,  8, 10, 12, 14, 16])
```

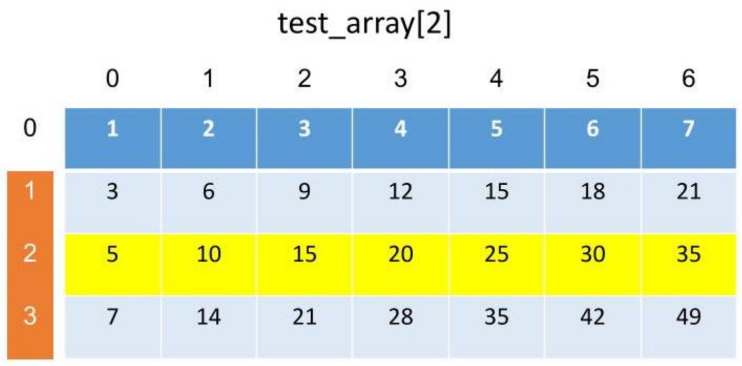
In [20]:
```
array1 > 3
```
Out[20]:
```
array([False, False, False,  True,  True,  True,  True,  True])
```

FYI, there are some interesting broadcasting rules for multidimensional arrays. Check out the [docs (https://numpy.org/devdocs/user/basics.broadcasting.html?highlight=shape)](https://numpy.org/devdocs/user/basics.broadcasting.html?highlight=shape) for more information.

## Slicing Arrays

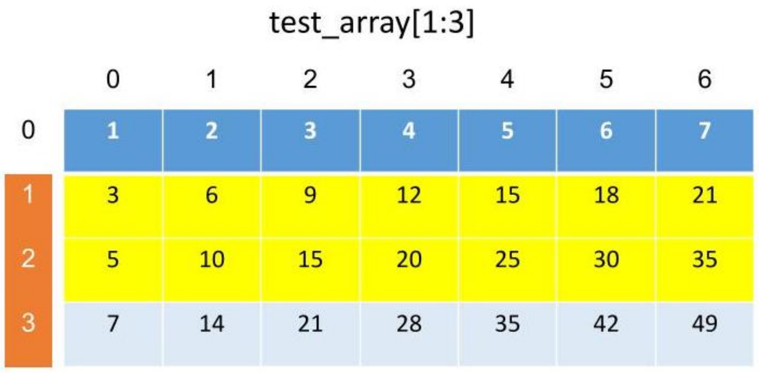
Below we demo a few ways to slice arrays. See the figures above the code for visualization of output found.

In [21]:
```
array = np.arange(1,8,1)
test_array = np.array([array,3*array,5*array,7*array])
test_array
```
Out[21]:
```
array([[ 1,  2,  3,  4,  5,  6,  7],
       [ 3,  6,  9, 12, 15, 18, 21],
       [ 5, 10, 15, 20, 25, 30, 35],
       [ 7, 14, 21, 28, 35, 42, 49]])
```



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 3 | 6 | 9 | 12 | 15 | 18 | 21 |
| 2 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| 3 | 7 | 14 | 21 | 28 | 35 | 42 | 49 |

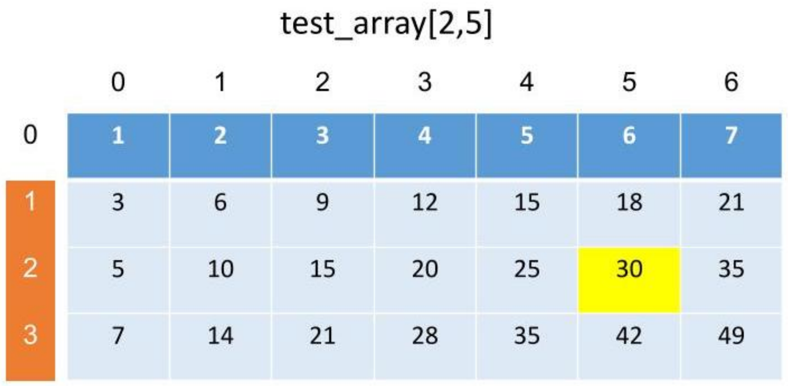
In [22]:
```
print(test_array[2])
```
```
[ 5 10 15 20 25 30 35]
```



test_array[1:3]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 3 | 6 | 9 | 12 | 15 | 18 | 21 |
| 2 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| 3 | 7 | 14 | 21 | 28 | 35 | 42 | 49 |

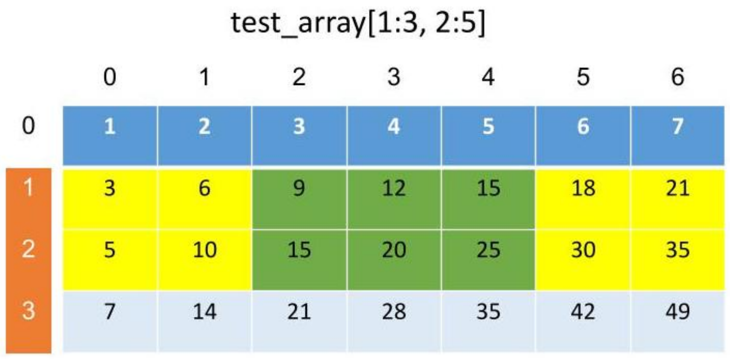
In [23]:
```
print(test_array[1:3])
```
```
[[ 3  6  9 12 15 18 21]
 [ 5 10 15 20 25 30 35]]
```



test_array[2,5]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 3 | 6 | 9 | 12 | 15 | 18 | 21 |
| 2 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| 3 | 7 | 14 | 21 | 28 | 35 | 42 | 49 |

In [24]:
```
print(test_array[2,5])
```
```
30
```



test_array[1:3, 2:5]

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 3 | 6 | 9 | 12 | 15 | 18 | 21 |
| 2 | 5 | 10 | 15 | 20 | 25 | 30 | 35 |
| 3 | 7 | 14 | 21 | 28 | 35 | 42 | 49 |

In [25]:
```
print(test_array[1:3, 2:5])
```
```
[[ 9 12 15]
 [15 20 25]]
```

# Matplotlib

Matplotlib has two styles interfaces when using the library:

- explicit "axes" interface
  - gives user full control over the figure
  - harder to use, but gives user more control
- implicit "pyplot" interface
  - matplotlib infers what user wants
  - easier to use, but give user less control

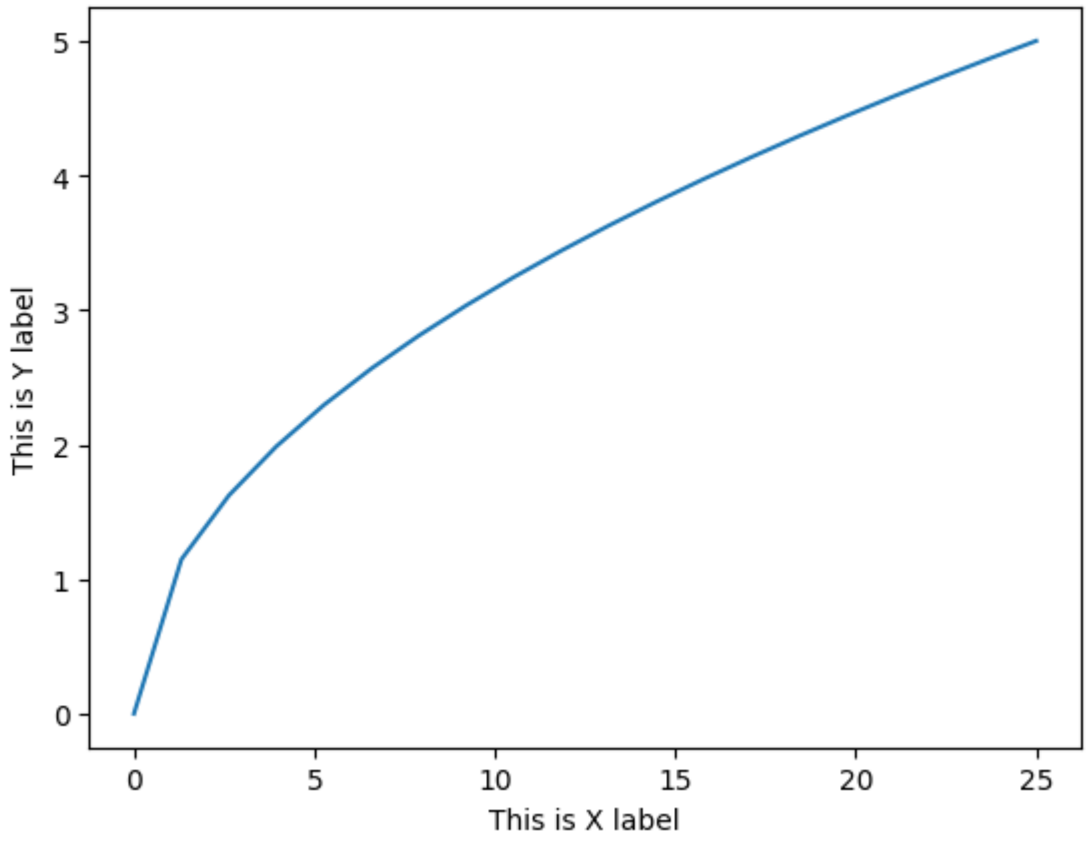
Read the docs (https://matplotlib.org/devdocs/users/explain/api_interfaces.html) for more information

## pyplot interface

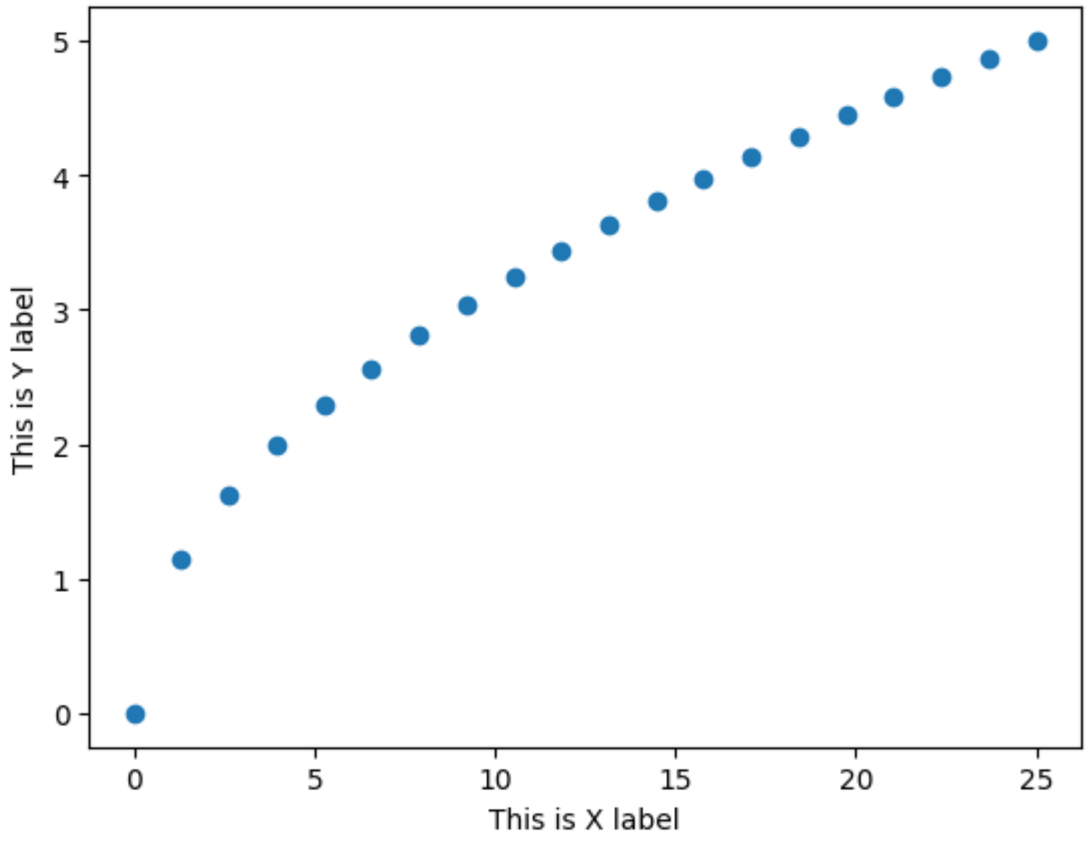
In [26]:
```
# For plots inline in notebook
%matplotlib inline

# Import matplotlib
import matplotlib.pyplot as plt
```

In the demo below we create several plots with the same data. Note that in the various demos we just change a few parameters to change things like the color of the graph, scatter versus plot, etc.

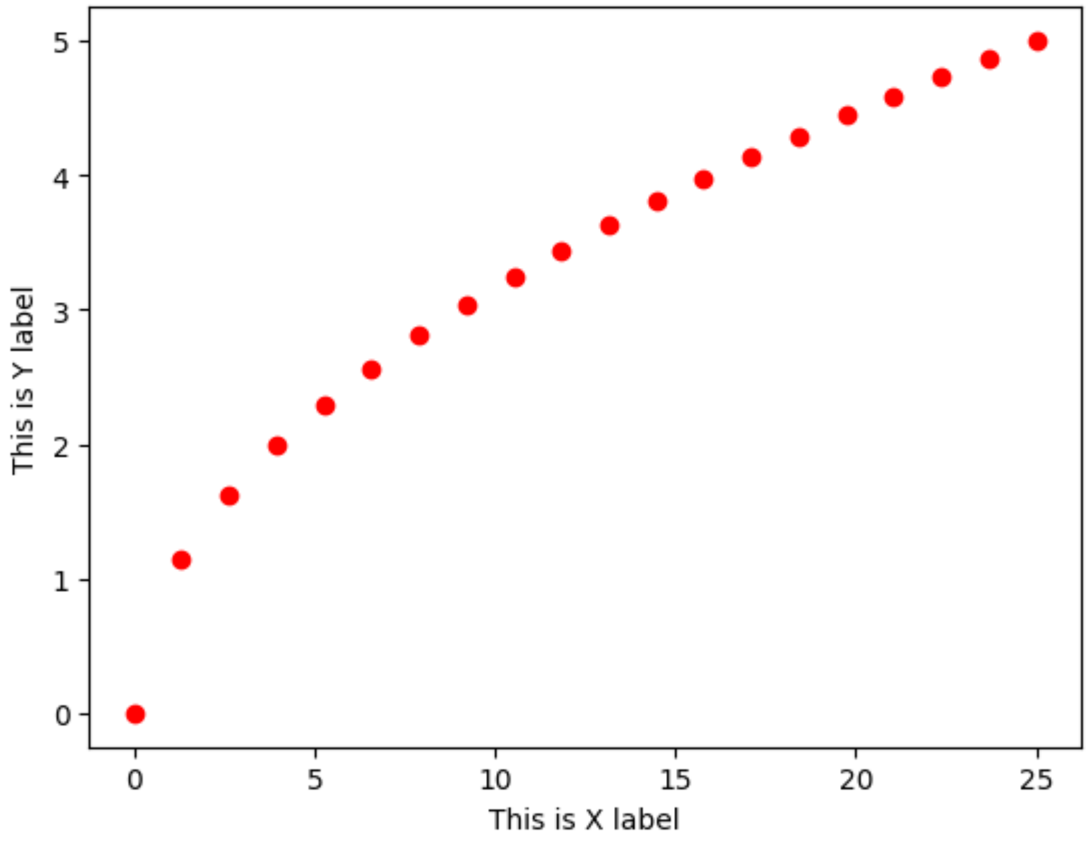
In [27]:
```
# Setup X and Y values
x = np.linspace(0,25, 20)
y = np.sqrt(x)

# Create plot
plt.xlabel('This is X label')
plt.ylabel('This is Y label')
plt.plot(x,y);
```



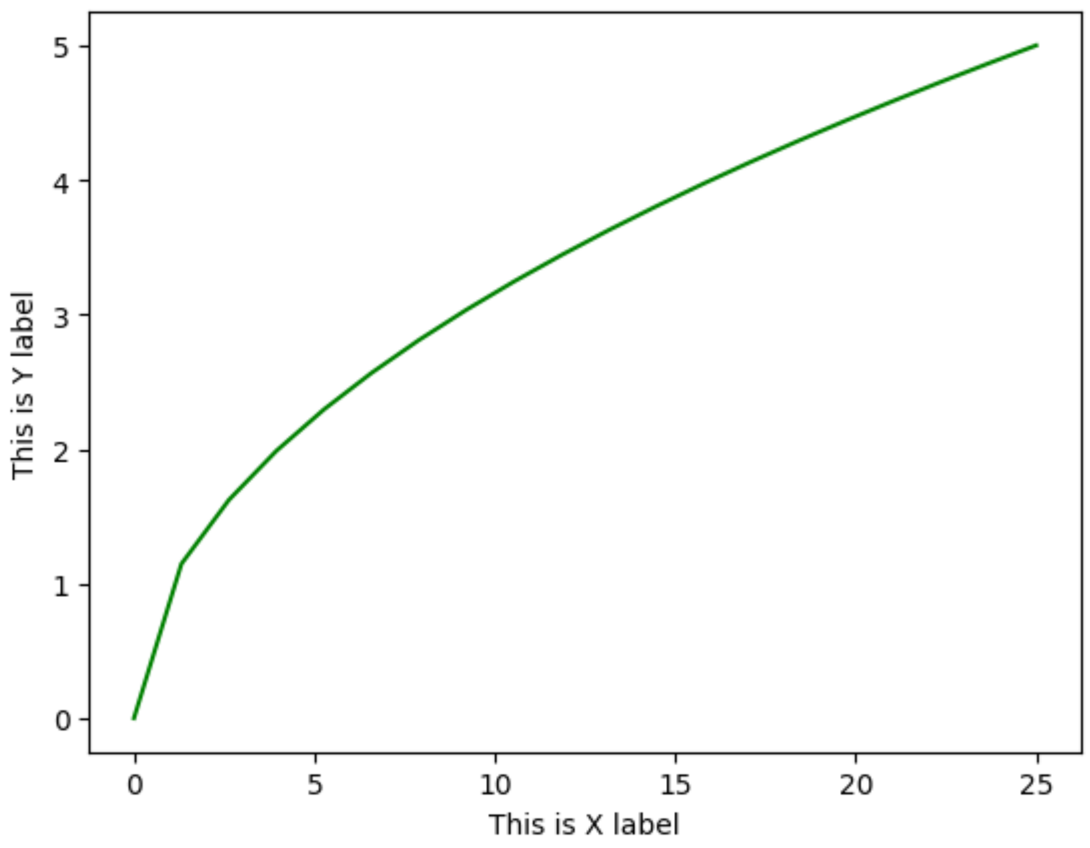
In [28]:
```
# Blue circles
plt.xlabel('This is X label')
plt.ylabel('This is Y label')
plt.plot(x,y,'o');
```



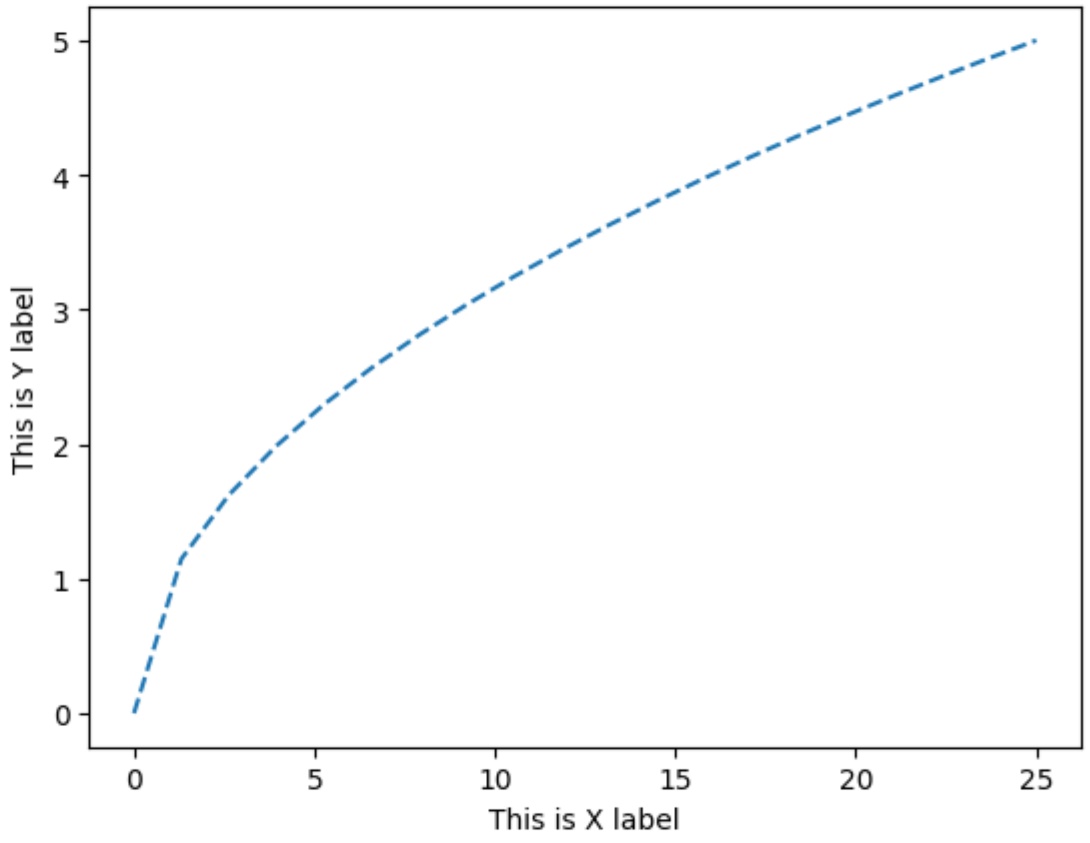
In [29]:
```
# Red circles
plt.xlabel('This is X label')
plt.ylabel('This is Y label')
plt.plot(x,y,'ro');
```



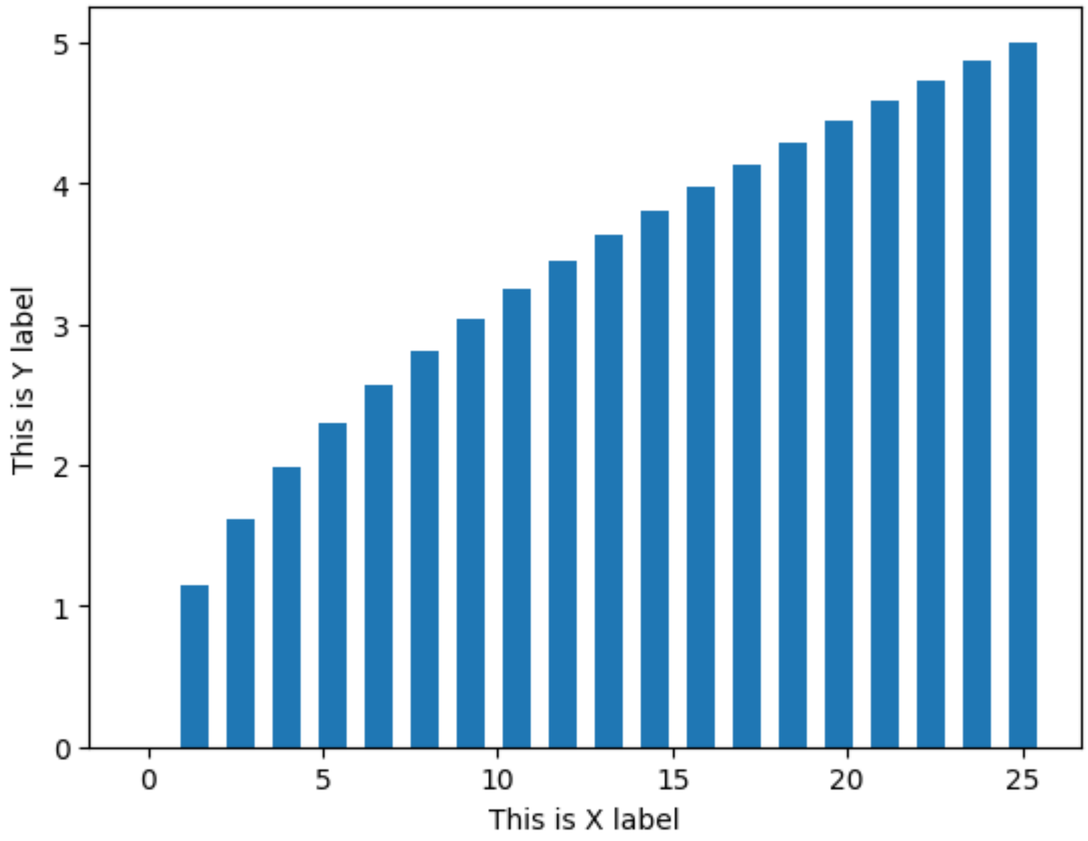
In [30]:
```
# Green Lines
plt.xlabel('This is X label')
plt.ylabel('This is Y label')
plt.plot(x,y, 'g-');
```



```
In [31]: # Dotted Lines
         plt.xlabel('This is X label')
         plt.ylabel('This is Y label')
         plt.plot(x,y,'--');
```



In [32]:
```
# Bar Plot (histogram)
plt.xlabel('This is X label')
plt.ylabel('This is Y label')
plt.bar(x,y);
```

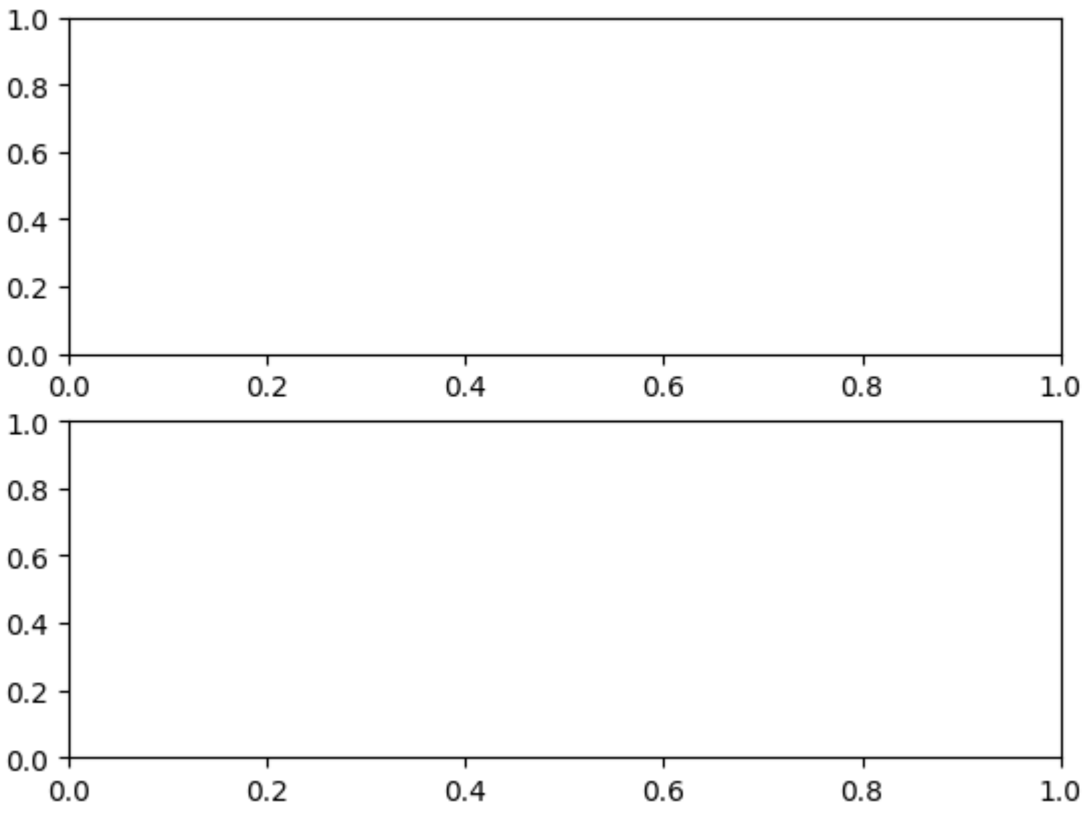


## Axes interface

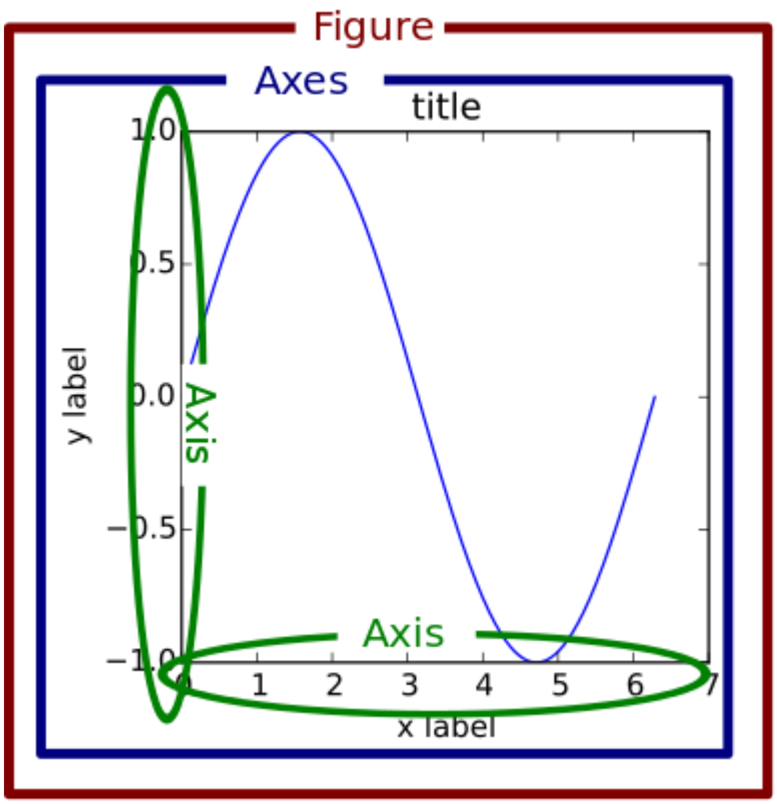
In the demo below we will create two plots in one figure using the axes interface. The two plots will show a sine and cosine wave

In [33]:
```
#get data for plot demo
x=np.arange(0,1,0.025)
y1=10*np.cos(2*np.pi*5*x)
y2=10*np.sin(2*np.pi*5*x)
```

In [34]:
```
fig, axs = plt.subplots(2,1)
```

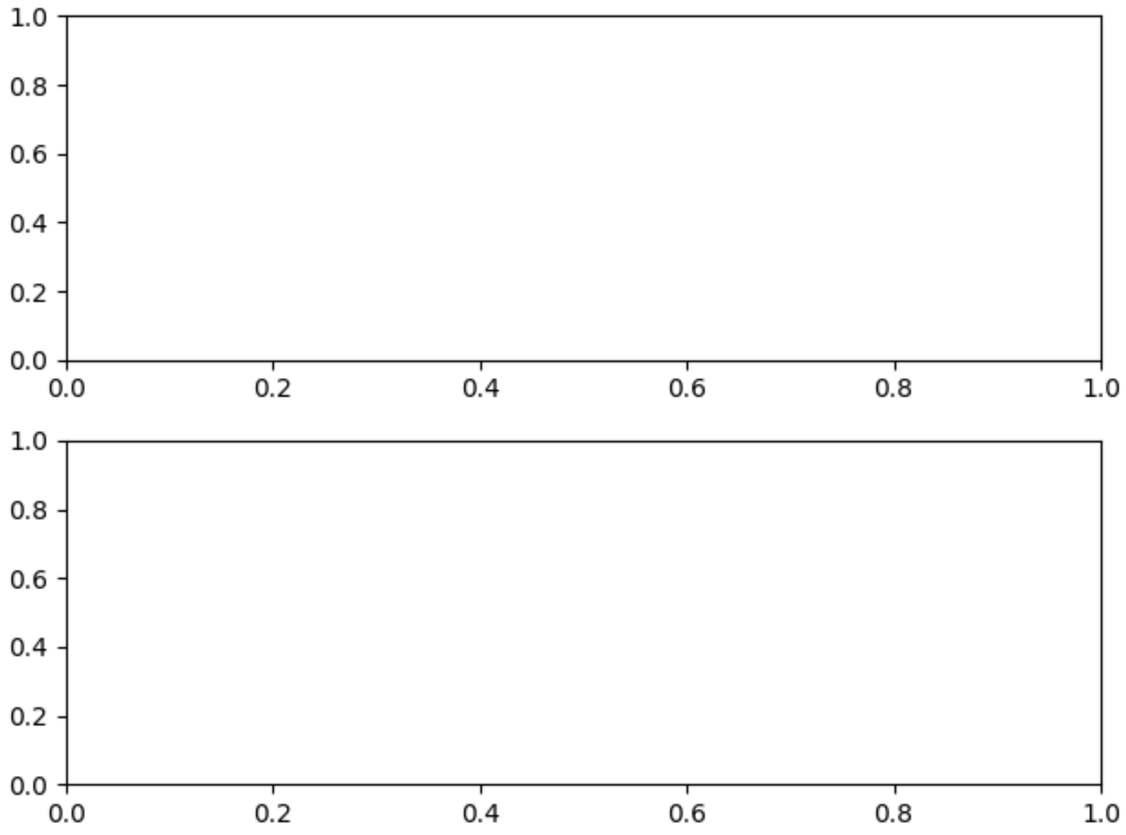


Above we instantiated a fig and axs object used to control a figure. The diagram below highlights what these two objects control:



`.tight_layout()` is an example of a fig method.

In [35]:
```
fig, axs = plt.subplots(2,1)
fig.tight_layout()
```

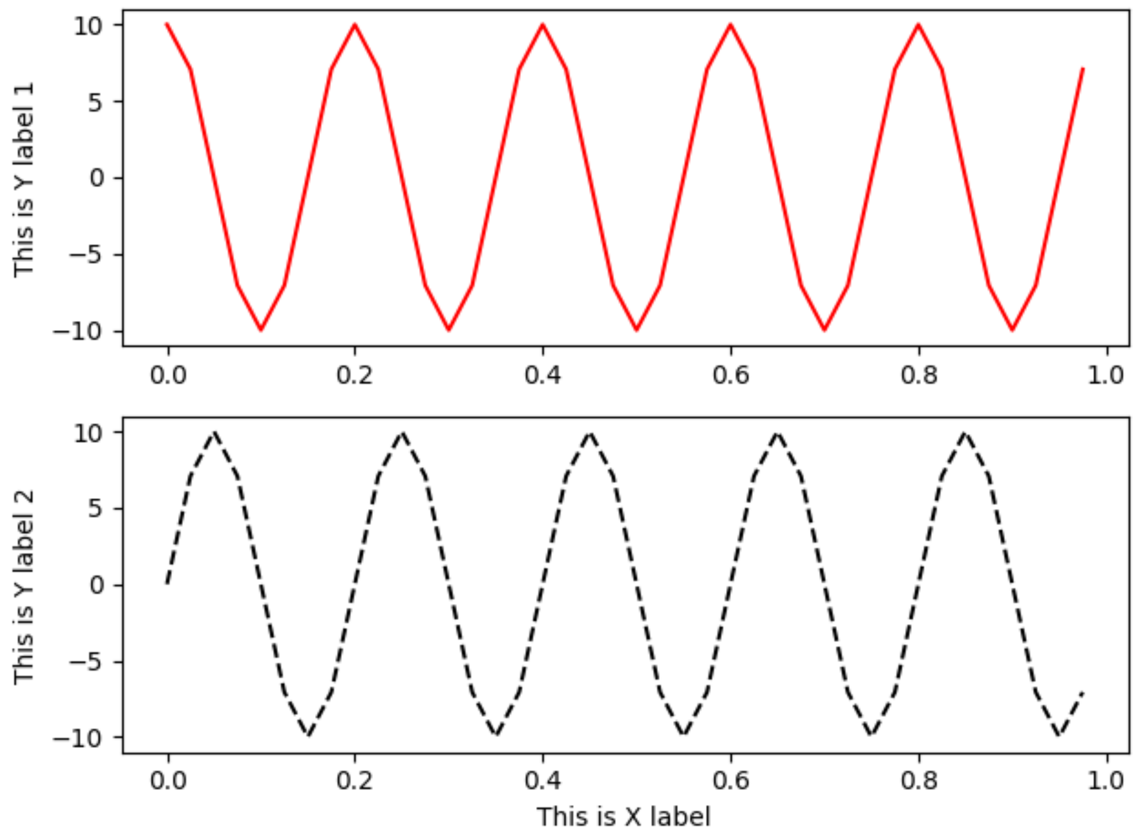


Note that since there are multiple figures in this plot, there are multiple axes objects we can control.

In [36]:
```
axs.shape,axs
```

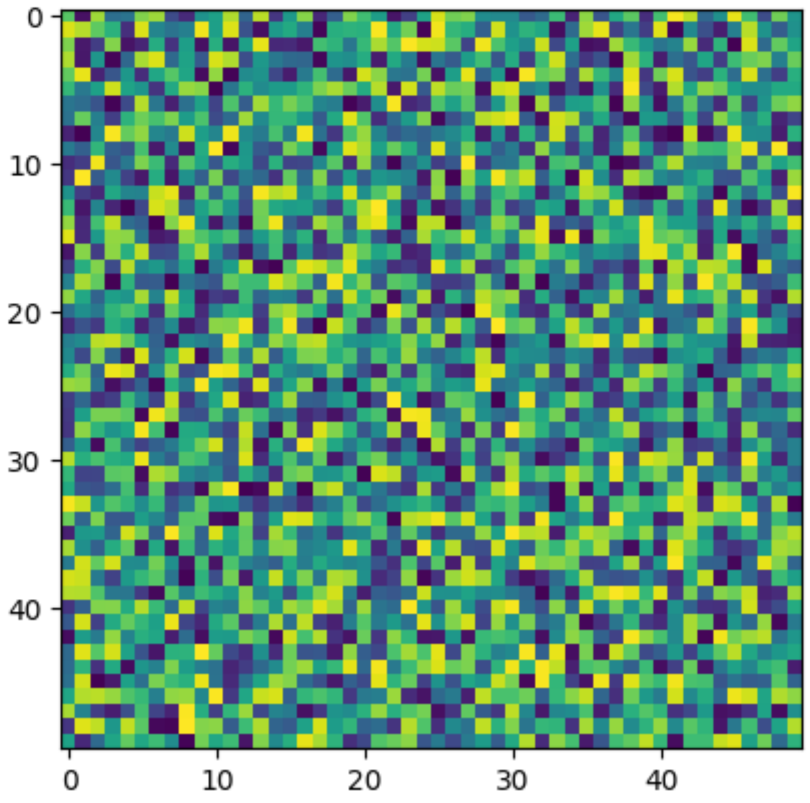
Out[36]:
```
((2,), array([<Axes: >, <Axes: >], dtype=object))
```

In [37]:
```python
fig, axs = plt.subplots(2,1)
axs[0].plot(x,y1,'r-')
axs[1].plot(x,y2,'k--')
axs[1].set_xlabel('This is X label')
axs[0].set_ylabel('This is Y label 1')
axs[1].set_ylabel('This is Y label 2')
fig.tight_layout();
```



In [38]:
```python
random_image = np.random.rand(50,50)
```

In [39]:
```
fig,ax=plt.subplots()
ax.imshow(random_image);
```

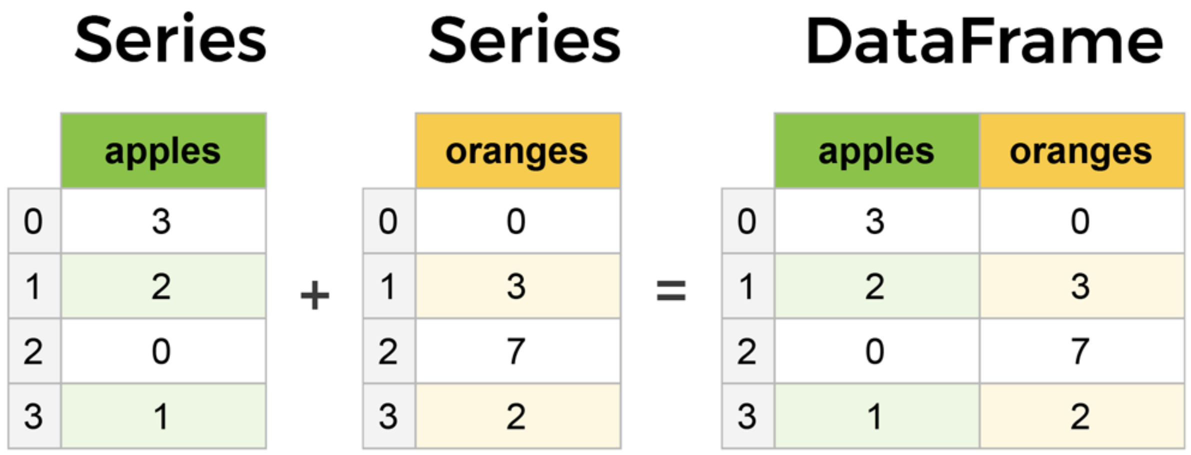


## Pandas

- Python package that builds upon numpy arrays
- Useful in data analysis
- Common manipulations of arrays
- [Pandas Documentation (http://pandas.pydata.org/)](http://pandas.pydata.org/)

Two main data structures in Pandas are:

- Series (1D array; column of a table)
  - Useful for sequence data like time series
- Dataframes (2D array where each column has its own data type; table of data)
  - Useful for spreadsheet like data



Below we will cover how to create and manipulate both data structures.

### Series

In [40]:
```
import pandas as pd
from pandas import Series, DataFrame
```

We can create a series from a python list with data.

In [41]:
```
test_series = Series([1, 2, 4, -7, -10 , 20])
test_series
```

Out[41]:
```
0      1
1      2
2      4
3     -7
4    -10
5     20
dtype: int64
```

Pandas is built on top of numpy. We can use the `.values` attribute to convert the series into a numpy array.

```
In [42]: test_series.values
```

```
Out[42]: array([  1,   2,   4,  -7, -10,  20])
```

Unlike arrays, we can customize the index and access elements via the new index.

```
In [43]: test_series = Series([1, 2, 4, -7, -10, 20], index=["x", "y", "a",
         "d", "e", "c"])
         test_series
```

```
Out[43]: x     1
         y     2
         a     4
         d    -7
         e   -10
         c    20
         dtype: int64
```

```
In [44]: test_series['d']
```

```
Out[44]: -7
```

We can slice the index by passing a list of indicies.

```
In [45]: test_series[['x', 'd', 'e']]
```

```
Out[45]: x     1
         d    -7
         e   -10
         dtype: int64
```

Scalar multiplication distributes through the series.

```
In [46]: test_series*2
```

```
Out[46]: x     2
         y     4
         a     8
         d   -14
         e   -20
         c    40
         dtype: int64
```

We can also create a Series using a dictionary that specifies the index (key).

In [47]:
```
test_dictionary = dict({"one":1, "two":2, "three":3, "four":4})
test_series = Series(test_dictionary)

print(test_series)
```

```
one      1
two      2
three    3
four     4
dtype: int64
```

If we add in an additional index with no value associated with it, pandas will and a NaN Value.

In [48]:
```
#data with null/NA values
test_index = ["one", "two", "three", "four", "five"]
test_series = Series(test_dictionary, index=test_index)
print(test_series)
```

```
one      1.0
two      2.0
three    3.0
four     4.0
five     NaN
dtype: float64
```

We can find where null values do or do not exist in a Series with `isnull()` and `notnull()`

In [49]:
```
pd.isnull(test_series)
```

Out[49]:
```
one      False
two      False
three    False
four     False
five      True
dtype: bool
```

In [50]:
```
pd.notnull(test_series)
```

Out[50]:
```
one       True
two       True
three     True
four      True
five     False
dtype: bool
```

We can sum series. Let's try it out.

In [51]:
```
test_series
```

Out[51]:
```
one      1.0
two      2.0
three    3.0
four     4.0
five     NaN
dtype: float64
```

In [52]:
```
test_series_2 = Series({"one":1, "two":2, "three":3, "five":5})
test_series_2
```

Out[52]:
```
one      1
two      2
three    3
five     5
dtype: int64
```

In [53]:
```
test_series + test_series_2
```

Out[53]:
```
five     NaN
four     NaN
one      2.0
three    6.0
two      4.0
dtype: float64
```

In [ ]:
```

```

## Dataframes

We can create dataframes from multiple python datatypes. Let's start with data stored in a dictionary where the key is a column name and the values are list of data.

In [54]:
```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
```

In [55]:
```
test_data_frame = DataFrame(data)
test_data_frame
```

Out[55]:

| | state | year | pop |
|---|---|---|---|
| **0** | Ohio | 2000 | 1.5 |
| **1** | Ohio | 2001 | 1.7 |
| **2** | Ohio | 2002 | 3.6 |
| **3** | Nevada | 2001 | 2.4 |
| **4** | Nevada | 2002 | 2.9 |

We can change the order of the columns by passing in the order.

In [56]:
```
# Change column sequence
test_data_frame = DataFrame(data, columns=['year', 'state', 'pop'])
test_data_frame
```

Out[56]:

| | year | state | pop |
|---|---|---|---|
| **0** | 2000 | Ohio | 1.5 |
| **1** | 2001 | Ohio | 1.7 |
| **2** | 2002 | Ohio | 3.6 |
| **3** | 2001 | Nevada | 2.4 |
| **4** | 2002 | Nevada | 2.9 |

If we pass a column name that does not exist in our data, the column will become NaN.

In [57]:
```
test_data_frame = DataFrame(data, columns=['year', 'state', 'po
p','debt'])
test_data_frame
```

Out[57]:

| | year | state | pop | debt |
|---|---|---|---|---|
| **0** | 2000 | Ohio | 1.5 | NaN |
| **1** | 2001 | Ohio | 1.7 | NaN |
| **2** | 2002 | Ohio | 3.6 | NaN |
| **3** | 2001 | Nevada | 2.4 | NaN |
| **4** | 2002 | Nevada | 2.9 | NaN |

We can also specify index, or row names.

```
In [58]: # Change row names just like in Series
         test_data_frame = DataFrame(data, columns=['year', 'state', 'pop'],
                                     index=['row one', 'row two', 'row thre
         e', 'row four', 'row five'])
         test_data_frame
```

Out[58]:

| | year | state | pop |
|---|---|---|---|
| **row one** | 2000 | Ohio | 1.5 |
| **row two** | 2001 | Ohio | 1.7 |
| **row three** | 2002 | Ohio | 3.6 |
| **row four** | 2001 | Nevada | 2.4 |
| **row five** | 2002 | Nevada | 2.9 |

I can use a list of column names to query a dataframe for specific columns

```
In [59]: test_data_frame[['year', 'state']]
```

Out[59]:

| | year | state |
|---|---|---|
| **row one** | 2000 | Ohio |
| **row two** | 2001 | Ohio |
| **row three** | 2002 | Ohio |
| **row four** | 2001 | Nevada |
| **row five** | 2002 | Nevada |

I can assign values to columns in multiple ways

```
In [60]: test_data_frame['debt'] = 25
         test_data_frame
```

Out[60]:

| | year | state | pop | debt |
|---|---|---|---|---|
| **row one** | 2000 | Ohio | 1.5 | 25 |
| **row two** | 2001 | Ohio | 1.7 | 25 |
| **row three** | 2002 | Ohio | 3.6 | 25 |
| **row four** | 2001 | Nevada | 2.4 | 25 |
| **row five** | 2002 | Nevada | 2.9 | 25 |

In [61]:
```
# Assigning a column a vector value
test_data_frame['debt'] = np.arange(1,6,1)
test_data_frame
```

Out[61]:

| | year | state | pop | debt |
|---|---|---|---|---|
| **row one** | 2000 | Ohio | 1.5 | 1 |
| **row two** | 2001 | Ohio | 1.7 | 2 |
| **row three** | 2002 | Ohio | 3.6 | 3 |
| **row four** | 2001 | Nevada | 2.4 | 4 |
| **row five** | 2002 | Nevada | 2.9 | 5 |

In [62]:
```
# pass a series
new_data = Series([1.2, 2.4, 5.7],index=['row two', 'row three', 'r
ow five'])
test_data_frame['debt'] = new_data
test_data_frame
```

Out[62]:

| | year | state | pop | debt |
|---|---|---|---|---|
| **row one** | 2000 | Ohio | 1.5 | NaN |
| **row two** | 2001 | Ohio | 1.7 | 1.2 |
| **row three** | 2002 | Ohio | 3.6 | 2.4 |
| **row four** | 2001 | Nevada | 2.4 | NaN |
| **row five** | 2002 | Nevada | 2.9 | 5.7 |

In [63]:
```
# Row shuffle
test_data_frame.reindex(['row five', 'row four', 'row three', 'row
two', 'row one'])
```

Out[63]:

| | year | state | pop | debt |
|---|---|---|---|---|
| **row five** | 2002 | Nevada | 2.9 | 5.7 |
| **row four** | 2001 | Nevada | 2.4 | NaN |
| **row three** | 2002 | Ohio | 3.6 | 2.4 |
| **row two** | 2001 | Ohio | 1.7 | 1.2 |
| **row one** | 2000 | Ohio | 1.5 | NaN |

We can do mathematical operations like summing rows and columns

In [64]:
```
test_data_frame.sum() # sum columns
```

Out[64]:
```
year                          10006
state    OhioOhioOhioNevadaNevada
pop                            12.1
debt                            9.3
dtype: object
```

In [65]:
```
test_data_frame.sum(axis=1,numeric_only=True) # sum rows, skips ove
r nonnumeric cells
```

Out[65]:
```
row one      2001.5
row two      2003.9
row three    2008.0
row four     2003.4
row five     2010.6
dtype: float64
```

We can also join two dataframes together.

In [66]:
```
data_frame_1 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a',
'b'],
                          'data1': range(7)})
data_frame_1
```

Out[66]:

| | lkey | data1 |
|---|---|---|
| **0** | b | 0 |
| **1** | b | 1 |
| **2** | a | 2 |
| **3** | c | 3 |
| **4** | a | 4 |
| **5** | a | 5 |
| **6** | b | 6 |

In [67]:
```
data_frame_2 = DataFrame({'rkey': ['a', 'b', 'd'],
                          'data2': range(3)})
data_frame_2
```

Out[67]:

| | rkey | data2 |
|---|---|---|
| **0** | a | 0 |
| **1** | b | 1 |
| **2** | d | 2 |

In [68]:
```
pd.merge(data_frame_1, data_frame_2,left_on='lkey', right_on='rke
y')
```

Out[68]:

| | lkey | data1 | rkey | data2 |
|---|---|---|---|---|
| **0** | b | 0 | b | 1 |
| **1** | b | 1 | b | 1 |
| **2** | b | 6 | b | 1 |
| **3** | a | 2 | a | 0 |
| **4** | a | 4 | a | 0 |
| **5** | a | 5 | a | 0 |

# Scikit-Learn

Scikit-learn is a machine-learning library for Python. It includes a variety of machine learning learning algorithms for classification, regression, clustering, etc.

Below we demos some of the basics functionality of scikit-learn.

## Simple Linear Regression

We will start with a demo of building a simple linear regression model. Let's start by importing the needed libraries from sklearn.

In [69]:
```
from sklearn.linear_model import LinearRegression
```

If you get an error message you may need to install sklearn. You can do that by uncommenting the line below.

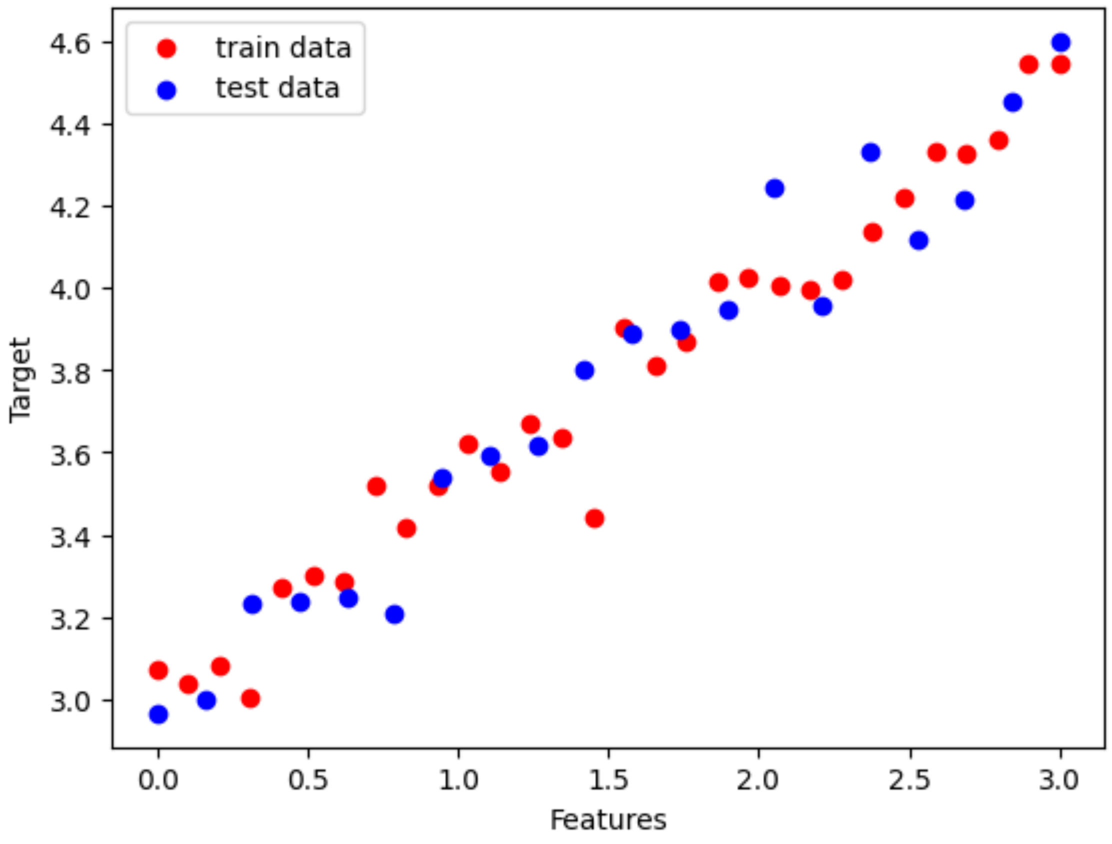
In [70]:
```
#! pip3 install --user scikit-learn
```

Next, lets use numpy to create random data that has a linear relationship where the slope is 0.5 and the y-intercept is 3. Two data sets are built for training and testing purposes.

In [71]:
```
slope = 0.5
intercept = 3

X = np.linspace(0,3,30)
y = slope * X + intercept + 0.1*np.random.normal(size=len(X))
X_test = np.linspace(0,3,20)
y_test = slope * X_test + intercept + 0.1*np.random.normal(size=len
(X_test))
```

Below we plot this data:

In [72]:
```
fig,ax = plt.subplots()
ax.scatter(X,y,c='r',label='train data')
ax.scatter(X_test,y_test,c='b',label='test data')
ax.set_xlabel('Features')
ax.set_ylabel('Target')
ax.legend();
```



Next we will need to instantiate the object for the model we want to build. In this first demo, we will use linear regression.

In [73]:
```
reg = LinearRegression()
```

Next we fit the model to the training data by calling the method 'fit()' and passing in the training data.

In [74]:
```
reg.fit(X.reshape(-1,1),y)
```

Out[74]:
```
▾ LinearRegression
LinearRegression()
```

Once our model is fit we can view attributes associated with the linear regression model such as the slope and intercept:

```
In [75]: reg.coef_ # slope
```

```
Out[75]: array([0.49351207])
```

```
In [76]: reg.intercept_ # intercept
```

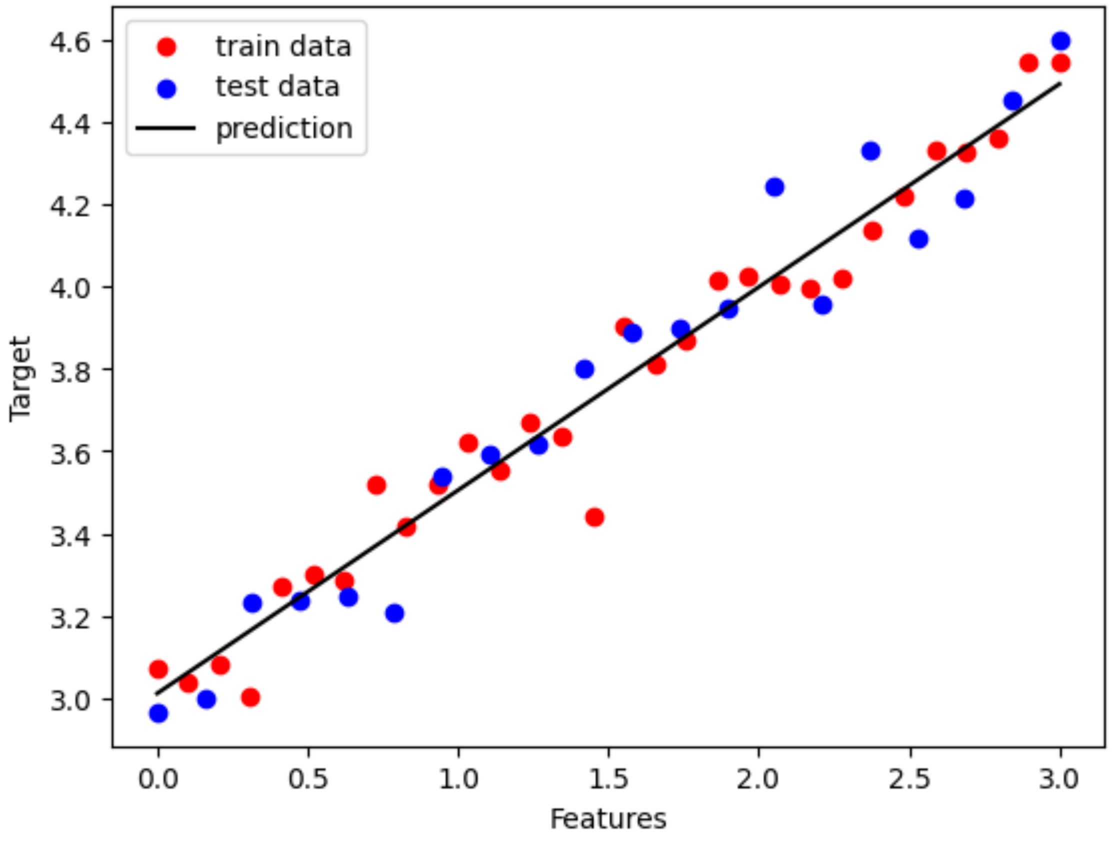
```
Out[76]: 3.0112976439906336
```

Finally we can use our model to make predictions by using the `predict()` method.

```
In [77]: reg.predict(np.array([[1.5]]))
```

```
Out[77]: array([3.75156575])
```

```
In [78]: fig,ax = plt.subplots()
         ax.scatter(X,y,c='r',label='train data')
         ax.scatter(X_test,y_test,c='b',label='test data')
         ax.plot(X,reg.predict(X.reshape(-1,1)),'k-',label='prediction')
         ax.set_xlabel('Features')
         ax.set_ylabel('Target')
         ax.legend();
```



scikit-learn also has many helpful functions for evaluating models. Below we demo code for computing the $R^2$ value of our linear regression model.

```
In [79]: from sklearn.metrics import r2_score

         yhat = reg.predict(X_test.reshape(-1,1))
         print('The R squared valued is {}'.format(r2_score(y_test,yhat)))

         The R squared valued is 0.9527890474531866
```

## Random Forest

The syntax used to build other models is very similar to linear regression. To demo this, we use the same random data generated above and build a Random Forest model.

```
In [80]: from sklearn.ensemble import RandomForestRegressor
```

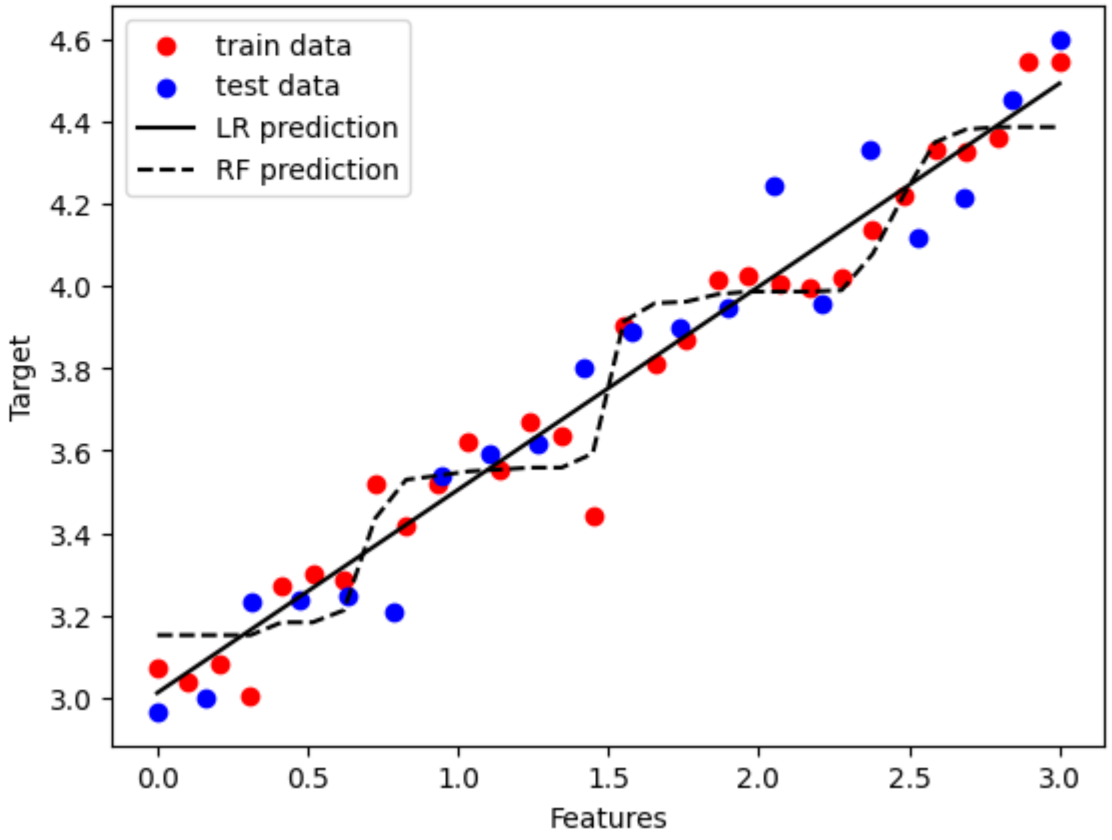
```
In [81]: regr = RandomForestRegressor(max_depth=2, random_state=0)
```

```
In [82]: regr.fit(X.reshape(-1,1),y)
```

```
Out[82]: ▾              RandomForestRegressor
         RandomForestRegressor(max_depth=2, random_state=0)
```

```
In [83]: regr.predict(np.array([[1.5]]))

Out[83]: array([3.68353137])
```

In [84]:
```
fig,ax = plt.subplots()
ax.scatter(X,y,c='r',label='train data')
ax.scatter(X_test,y_test,c='b',label='test data')
ax.plot(X,reg.predict(X.reshape(-1,1)),'k-',label='LR prediction')
ax.plot(X,regr.predict(X.reshape(-1,1)),'k--',label='RF predictio
n')
ax.set_xlabel('Features')
ax.set_ylabel('Target')
ax.legend();
```



In [85]:
```
yhat = regr.predict(X_test.reshape(-1,1))
print('The R squared valued is {}'.format(r2_score(y_test,yhat)))
```

```
The R squared valued is 0.8948817894481444
```

## Dask

Dask is a python library for parallel computation. Dask is built on top of the already existing python data science ecosystem including the popular libraries such as pandas, numpy, and sklearn making it easy to use for python users, while also allowing users to scale up their code to leverage high performance computing environments. Dask has high and low level interfaces:

- High level collections: Array, Bags, and Dataframes that mimic the python ecosystem (pandas, numpy, etc.) but also parallelizes the code for data that doesn't fit in memory.
- Low level schedulers: task schedulers that allow users to leverage HPC environments.

To learn more about Dask checkout this dask tutorial (https://github.com/dask/dask-tutorial).

```
In [ ]:
```