Numpy

Learning Objectives:

By the end of this lesson, all learners will be able to:

- create a numpy array according to the needs
- implement slicing and indexing for accessing elements from an array
- interpret vectorized operations for faster computation
- discuss various matrix operations and broadcasting in numpy
- discuss various matplotlib terms such as plot, axes, title, grid, legend, markers, etc
- interpret various types of visualizations in matplotlib
- select among various types of plot for a particular task
- use subplot to plot different plots in figure

Introduction to Numpy

Numpy (acronym for Numerical Python) is a package for scientific computing in Python. Numpy can be used to store all the data in the same structure.

Unlike `list` , Numpy array cannot store heterogeneous data structures. It provides a high-performance multidimensional array object and tools for working with these arrays as well as enables performing complex mathematical operations on them, including support for linear algebra, Fourier transformation, and random number generation.

Installing NumPy

To install NumPy using `pip` , run the following command in your terminal:

```
pip install numpy
```

Importing NumPy

We can import NumPy, into our scripts, using the following line.

```
import numpy
```

The general convention when using NumPy is to use the alias `np` . Make it a habit to use this alias since almost everyone imports NumPy as `np` . This aliasing makes your code compatible with much of the Python community.

In [ ]:

```
import numpy as np
```

Creating a NumPy array

The simplest way to create a NumPy array is using the function `np.array()` . This function can create NumPy arrays out of most basic types and iterables.

In [ ]:

```
#create a simple 1D array
arr = np.array([1,2,3])
arr
```

Out[5]:

```
array([1, 2, 3])
```

In [ ]:

```
#create a 2D array
arr = np.array([(1,2,3),(4,5,6)])
arr
```

Out[6]:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [ ]:

```
#create a 3D array
arr = np.array([[(1,2,3),(4,5,6)],[(3,2,1),(4,5,6)]])
arr
```

Out[7]:

```
array([[[1, 2, 3],
        [4, 5, 6]],

       [[3, 2, 1],
        [4, 5, 6]]])
```

In [ ]:

```
# Create an array of linear sequence starting 0 and ending at 9
np.arange(0, 10)
```

Out[8]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [ ]:

```
# create an array of linear sequence starting at 0 and ending at 8 with difference
np.arange(0, 10, 2)
```

Out[6]:

```
array([0, 2, 4, 6, 8])
```

In [ ]:

```
# Create an array of ten values which are evenly spaced between 0 and 1
np.linspace(0, 1, 10)
```

Out[9]:

```
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

In [ ]:

```
# Create a 3x3 array filled with 4
np.full((3, 3), 4)
```

Out[10]:

```
array([[4, 4, 4],
       [4, 4, 4],
       [4, 4, 4]])
```

Basic array attributes in Numpy

The basic attributes of numpy array are shape, size, dtype, ndim, where shape gives the size of each dimension, size gives the number of elements, dtype gives the data type, and ndim gives the number of dimensions.

In [ ]:

```
print(arr.shape)
print(arr.size)
print(type(arr))
print(arr.ndim)
```

```
(2, 2, 3)
12
<class 'numpy.ndarray'>
3
```

Data types in Numpy

We can also set the data type of elements within a numpy array. The standard data types in NumPy are listed below:

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (-128 to 127)
`int16`	Integer (-32768 to 32767)
`int32`	Integer (-2147483648 to 2147483647)
`int64`	Integer (-9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`.
`float16`	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`.
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

Source Table: https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html (https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html)

While creating an array, it can be specified using a string or by using a numpy object.

In [ ]:

```
np.zeros(10, dtype=np.int32)
```

Out[12]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

In [ ]:

```
np.zeros(20, dtype='int32')
```

Out[13]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
      dtype=int32)
```

Ones and zeros

NumPy also has functions that create arrays filled with ones or zeros.

In [ ]:

```
np.zeros([2, 3], dtype=np.int8) # create an array of shape 2x3 filled with zeros
```

Out[14]:

```
array([[0, 0, 0],
       [0, 0, 0]], dtype=int8)
```

In [ ]:

```
np.ones([3, 1, 2], dtype=np.float16)
```

Out[15]:

```
array([[[1., 1.]],

       [[1., 1.]],

       [[1., 1.]]], dtype=float16)
```

Random Arrays

You can create random arrays using many different types of random distributions using NumPy.

In [ ]:

```
# random array sampled using the standard normal distribution
np.random.standard_normal([3, 3])
```

Out[16]:

```
array([[-1.14875701, -0.03381262,  1.25858138],
       [ 1.07079641,  0.82863406,  0.55626595],
       [-0.82220452, -0.07673879,  1.22177841]])
```

NumPy provides many ways to create random arrays.

In [ ]:

```
# Create a 5x5 array of uniformly distributed random values between 0 and 1
np.random.random((5, 5))
```

Out[17]:

```
array([[0.85022256, 0.24616792, 0.11767452, 0.58315524, 0.05069959],
       [0.94657031, 0.6987091 , 0.10488648, 0.26263047, 0.75505316],
       [0.89937258, 0.10663218, 0.07974195, 0.04643394, 0.35376335],
       [0.57277991, 0.6792985 , 0.7045467 , 0.37402172, 0.27982386],
       [0.41558808, 0.20579072, 0.52319   , 0.45363379, 0.22307787]])
```

In [ ]:

```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

Out[18]:

```
array([[8, 1, 2],
       [7, 2, 6],
       [4, 1, 7]])
```

If we run the cells which generated the random numbers again, it will produce a different result. To produce the same result on each run, we can use np.random.seed, as shown below:

In [ ]:

```
np.random.seed(0) ; np.random.rand(4)
```

Out[19]:

```
array([0.5488135 , 0.71518937, 0.60276338, 0.54488318])
```

If we run the above cell again, the result will be same.

Numpy Indexing and Slicing

Contents of the NumPy array can be accessed and modified through indexing and slicing. They give the powerful capability to select the data for further analysis. These are the most common operations you need to know by heart.

Numpy Indexing

Numpy indexing allows accessing elements from the entire NumPy array. We use indexing to retrieve a single value. Indexing can be done by using an array as an index. We retrieve the value per the position. You can see it below:

```
array[position]
```

> Get the number or slice at `position` in `array` .

In [ ]:

```
array = np.arange(10)
array[2] # retrieves an element with position 3
```

Out[20]:

```
2
```

Negative Indexing

This type of indexing allows accessing elements from the end of the NumPy array. The last element is indexed as -1 (not -0).

In [ ]:

```
array[-2] # retieves second last element
```

Out[21]:

```
8
```

Numpy Slicing

To retrieve the collections of value, we use slicing.

```
array[start:end]
```

> THe start position is inclusive whereas the end position is exclusive

In [ ]:

```
array[3:6] # outputs the fourth element but excludes the seventh element
```

Out[22]:

```
array([3, 4, 5])
```

We can also assign the step size while retrieving an element from the array.

```
array[start:end:step]
```

> The element are retrieved from start to an end with a step size of the step.

In [ ]:

```
array[2:8:2]
```

Out[23]:

```
array([2, 4, 6])
```

In [ ]:

```
array[8:2:-1] # negative step size retrieves the element from back side
```

Out[24]:

```
array([8, 7, 6, 5, 4, 3])
```

Slicing a Two Dimensional Array

Slicing is similar to the slicing of the one-dimensional array. Here, we have to select the range of values to for both row and column that we want to extract.

```
array[row_range, column_range]
```

> retrieves the elements which fall into the specified range in an array

In [ ]:

```
n_array=np.random.rand(3, 3)
print(n_array)
print(n_array[0:2, 0:2]) # retrieves the first two elements in a row and first two
```

```
[[0.4236548  0.64589411 0.43758721]
 [0.891773   0.96366276 0.38344152]
 [0.79172504 0.52889492 0.56804456]]
[[0.4236548  0.64589411]
 [0.891773   0.96366276]]
```

In [ ]:

```
print(n_array[:, 1]) # retrieves only second column
```

```
[0.64589411 0.96366276 0.52889492]
```

In [ ]:

```
print(n_array[1:, ])# retrieves last two rows with all columns
```

```
[[0.891773   0.96366276 0.38344152]
 [0.79172504 0.52889492 0.56804456]]
```

Boolean Indexing

An everyday use case of boolean array indexing is to filter the unwanted values within an array. Here we specify the condition in the form of boolean expression in the index and the elements satisfying the conditions are retrieved.

In [ ]:

```
A = np.array([[42,56,89,65],
              [100,88,42,12],
              [55,42,17,18]])

B=A[A<20] # retrieve elements that are smaller than 20
B
```

Out[28]:

```
array([12, 17, 18])
```

In [ ]:

```
x = np.array([[0, 1], [1, 1], [2, 2]])
rowsum = x.sum(1)
x[rowsum <= 2, :] # only retrieves the first two rows since the sum of the elements
```

Out[29]:

```
array([[0, 1],
       [1, 1]])
```

Numpy Reshaping

This function of the NumPy gives a new shape to a NumPy array. We can reshape NumPy arrays as long as the original size of the array is the same as the target size.

```
numpy.reshape(array, new_shape, order)
```

> This returns an array of the new size. The order keyword gives the index ordering both for fetching the values from `a`, and then placing the values into the output array.

In [ ]:

```
arr = np.arange(6).reshape((3, 2))
arr
```

Out[30]:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

In [ ]:

```
# Reshaping 2D array to 1D array
import numpy as np
x = np.array([[4,5,8], [5,6,7]])
np.reshape(x, 6)
```

Out[31]:

```
array([4, 5, 8, 5, 6, 7])
```

In [ ]:

```
# Reshaping 3D array to 2D Array
x= np.zeros((2,3,4)) #3D Array
np.reshape(x, (6,4))
```

Out[32]:

```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

Use of -1 as shape parameter

In [ ]:

```
x = np.array([[2,3,4], [5,6,7]])
np.reshape(x, (3, -1)) # Auto-adapt number of columns
```

Out[33]:

```
array([[2, 3],
       [4, 5],
       [6, 7]])
```

Numpy ravel

This numpy function returns the flat contiguous array.You can find it more [here. (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.ravel.html)

In [ ]:

```
x = np.array([[1, 2, 3], [4, 5, 6]])
np.ravel(x)
```

Out[34]:

```
array([1, 2, 3, 4, 5, 6])
```

`ravel()` function returns a flattened view of Numpy array. Any changes made to the flattened array ends up with the same change in the original array.

Matrix Operations and Broadcasting

NumPy provides efficient implementations for many matrix operations, starting from the simple dot product to complex operations like Singular Value Decomposition (SVD). NumPy can use efficient matrix operations libraries, such as different implementations of BLAS and LAPACK, under the hood, which allows you to use your CPU to its maximum potential.

Addition and Subtraction

Add or subtract two NumPy arrays the same way you would add two numbers in NumPy arrays.

In [ ]:

```
import numpy as np
arr_1 = np.array(
    [
        [1, 2],
        [3, 4]
    ]
)

arr_2 = np.array(
    [
        [5, 6],
        [7, 8]
    ]
)
```

In [ ]:

```
print('Addition: \n', arr_1 + arr_2)
print('Subtraction: \n', arr_1 - arr_2)
```

```
Addition: 
 [[ 6  8]
 [10 12]]
Subtraction: 
 [[-4 -4]
 [-4 -4]]
```

Scalar Multiplication

In [ ]:

```
3 * arr_1 # multiplies each element of a by 3
```

Out[41]:

```
array([[ 3,  6],
       [ 9, 12]])
```

Matrix Multiply

To multiply together two matrices, first, make sure their dimensions match up. For example, you can multiply two matrices only if the number of columns in the first matrix is equal to the number of rows in the second.

In [ ]:

```
import numpy as np

arr_1 = np.array(
    [
        [3, 8],
        [3, 3]
    ]
)

arr_2 = np.array(
    [
        [8, 8],
        [3, 3]
    ]
)
```

There are three ways to do this exact thing in NumPy, which can be pretty confusing at first sight. The first way is to use the function `np.matmul()` .

In [ ]:

```
print(np.matmul(arr_1, arr_2))
```

```
[[48 48]
 [33 33]]
```

The other is using `np.dot()` , which is also a first order function in NumPy. This means that you can perform a dot product by calling the `dot()` method of a NumPy array.

In [ ]:

```
print(np.dot(arr_1, arr_2))
print(arr_1.dot(arr_2))
```

```
[[48 48]
 [33 33]]
[[48 48]
 [33 33]]
```

The last way is to use the `matmul` operator, `@` , introduced in Python 3.5.1.

In [ ]:

```
print(arr_1@arr_2)
```

```
[[48 48]
 [33 33]]
```

These methods are not all the same, in any case. If you are curious about the differences between these function, follow the link in the references below.

Transposing a vector

In numpy, we use np.transpose(vec) to transpose a vector.

In [ ]:

```
# Transpose 3 x 3 matrix
mat = np.array([[1, 1, 1],
                [2, 2, 2],
                [3, 3, 3]])
print(mat)
print("The transpose of the above matrix is")
print(mat.T)
%timeit mat.T
```

```
[[1 1 1]
 [2 2 2]
 [3 3 3]]
The transpose of the above matrix is
[[1 2 3]
 [1 2 3]
 [1 2 3]]
The slowest run took 35.80 times longer than the fastest. This could m
ean that an intermediate result is being cached.
10000000 loops, best of 3: 158 ns per loop
```

Aggregations in numpy

Often, the first step, when faced with a large amount of data, is to compute typical summary statistics: mean, standard deviation, sum, product, median, minimum, maximum and so on. NumPy provides fast built-in aggregation functions for working on arrays.

In [ ]:

```
# compute mean, standard deviation, variance, sum and product of data
array1 = np.array([[1, 2, 3], [4, 5, 6]])
print("Mean: ", np.mean(array1))
print("Std: ", np.std(array1))
print("Var: ", np.var(array1))
print("Sum: ", np.sum(array1))
print("Prod: ", np.prod(array1))
```

```
Mean:  3.5
Std:  1.707825127659933
Var:  2.9166666666666665
Sum:  21
Prod:  720
```

In [ ]:

```
# sum all the values in an array
print("Sum: ", np.sum(array1))

# find minimum and maximum values of an array
print("Min: ", np.min(array1), "Max: ", np.max(array1))

# finding argmax along first axis (axis=0) and second axis (axis=1)
print("Argmax along first axis: ", np.argmax(array1, axis=0))
print("Argmax along second axis: ", np.argmax(array1, axis=1))
```

```
Sum:  21
Min:  1 Max:  6
Argmax along first axis:  [1 1 1]
Argmax along second axis:  [2 2]
```

Some other aggregation functions are: np.prod, np.any, np.all, np.median, np.percentile and so on. We can perform all these aggregation functions on both of the axes.

Linear algebra using numpy

Numpy provides the library for linear algebra in 'np.linalg'. In NumPy, the linear algebra functions rely on BLAS and LAPACK. These libraries may be provided by NumPy itself using the C version. Still, whenever possible, highly optimized libraries like OpenBLAS, MKL (TM) and ATLAS that take advantage of specialized processor functionality is preferred. Doc here (https://docs.scipy.org/doc/numpy/reference/routines.linalg.html).

In [ ]:

```
# Taking determinant of a matrix
a = np.array([[3, 4], [1, 2]])
np.linalg.det(a)
```

Out[49]:

```
2.0000000000000004
```

In [ ]:

```
# Taking inverse of a matrix
from numpy.linalg import inv
inv(a)
```

Out[50]:

```
array([[ 1. , -2. ],
       [-0.5,  1.5]])
```

Broadcasting

Not all operations in NumPy need arrays of the same dimensions. For example, if you wanted to add a specific vector to all vectors in a list of vectors, you need not create another array of the same size as the one you want to add to. NumPy handles these cases with broadcasting.

In [ ]:

```
big_list_of_vectors = np.zeros([400, 2])
vector_to_add = np.array([3,2])
```

During addition, `vector_to_add` gets 'broadcasted' to all the vectors in `big_list_of_vectors` . Without broadcasting, the vector `vector_to_add` would have to be a matrix as large as `big_list_of_vectors` , with each row equal to `[3,2]` .

In [ ]:

```
added = big_list_of_vectors + vector_to_add
print(added.shape)
print(added[:10])
```

```
(400, 2)
[[3. 2.]
 [3. 2.]
 [3. 2.]
 [3. 2.]
 [3. 2.]
 [3. 2.]
 [3. 2.]
 [3. 2.]
 [3. 2.]
 [3. 2.]]
```

Note that only the first 10 entries of the resultant vector are shown here.

Multiplication by a scalar is itself possible only due to broadcasting.

Broadcasting is handy because it makes your code short and concise, and it also simplifies many arithmetic operations.

How does NumPy decide if two arrays can be operated on using broadcasting? From the NumPy documentation:

> When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions and works its way forward. Two dimensions are compatible when

- they are equal or
- one of them is 1

NumPy for Faster Computation

Lists are the core library provided by the python core. A list is the Python equivalent of an array, but is resizeable and can contain elements of different types. We use NumPy arrays instead of Lists for faster computation.

The main benefits of using NumPy arrays is smaller memory consumption and better runtime behaviour. Numpy provides an easy and flexible interface for the computation of arrays with data.

Slowness of the Loops

The default operation of Python (CPython) does the operations very slowly. You know that each entry in numbers is an integer, but the interpreter does not know this beforehand. At each iteration through the list numbers, the interpreter has to find out the type of the object number. This process makes Python inherently very slow when working with lists and arrays. When your code involves nested for loops that iterate over large iterables, Python is terribly slow as compared to other statically typed languages.

Illustration of the slowness of the loop in Python

Python makes it easy to measure the execution time of your code: the `%timeit` and `%time` magic measures the time taken to execute a single cell.

%timeit: Time repeated execution of a single statement for more accuracy

In [ ]:

```
# Using NumPy Array
import numpy as np
x = np.arange(10)
%timeit x**2
```

```
The slowest run took 26.22 times longer than the fastest. This could m
ean that an intermediate result is being cached.
1000000 loops, best of 3: 692 ns per loop
```

In [ ]:

```
# Using Python's core Library[List]
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
%timeit [i**2 for i in x]
```

```
100000 loops, best of 3: 2.31 µs per loop
```

Here, from the above two code cells, the best runtime outputs of the respective cells are presented. With the same number of elements and the same operation, the runtime of the numpy array is far better than the loops in the Python core library.

Vectorised Operation

Vectorized Operation in Numpy

NumPy is fast because it uses **vectorized** code. That is, it uses instruction sets like SIMD (Same Instruction Multiple Data), which is a way for CPUs to process data in parallel across many-core efficiently. Vectorized operation in NumPy takes a nested sequence of objects or NumPy arrays as inputs and returns a single NumPy array or a tuple of NumPy arrays.

This vectorized operation is the optimized computation on NumPy arrays.

Since NumPy arrays are homogeneous, an array can only contain a single type of data. So, NumPy can delegate the task of performing mathematical operations on the array's contents to optimized, compiled C code.

For instance, let's consider the task of summing up the integers stored in a NumPy array.

In [ ]:

```
# sum an array, using NumPy's vectorized 'sum' function
%time np.sum(np.arange(10000))
```

```
CPU times: user 443 µs, sys: 7 µs, total: 450 µs
Wall time: 345 µs
```

Out[56]:

```
49995000
```

In [ ]:

```
#sum an array by explicitly looping over the array in Python
import time
start=time.time()
total = 0
for i in np.arange(10000):
  total = i + total

print(time.time()-start)
```

```
0.003955632019043
```

From the above example, it's clear about the convinient nature of the vectorised operation.

Matplotlib

Introduction

Visualizations make it easier and quicker to understand complex problems by providing more straightforward ways to analyze and absorb information. They help in identifying patterns, relationships, and outliers in data. Visualization is necessary for machine learning since it helps to thoroughly understand the given data, which is the foundation for building models.

Matplotlib is the most popular Python 2D plotting library. It helps to generate plots, histograms, bar charts, pie charts, scatterplots, etc. It can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, and web application servers.

Installing Matplotlib

```
python -m pip install -U matplotlib
```

Matplotlib Terms

Plot

Plot is the graph you see in the figure above. Those green and blue lines and red dots, they are each a different plot.

Axes:

Axes are the entire area of a single plot in the figure. It is a class that contains several attributes needed to draw a plot, such as adding a title, giving labels, selecting bin values for different types of plots on each axes, etc.

We can have multiple axes in a single plot, by which we can combine multiple plots into a single figure. However, one Axes can only contain one figure. For e.g., If we want both PDF and CDF curves in the same figure, we can create 2 axes and draw each of them in different axes. Then we can combine them into a single figure.

Axis:

Axes contains two **Axis** objects which correspond to the x-axis and y-axis. These Axis objects contribute to setting the data limit of a figure. Both axes have a title (also referred to as label). In the figure, you can see that the x-axis label is `Voltage(Volts)` You can set the labels using the methods `xlabel()` and `ylabel()` .

```
plt.ylabel('Current (milli-ampere)')
plt.xlabel('Voltage (Volts)')
```

Axis is also responsible for generating the **Ticks** on the graph. Ticks are the marks you can see on the x and y-axis. You can define where they are and their labels using the `xticks()` and `yticks()` methods. For the figure above, we've added the `xticks` as follows:

```
plt.xticks(x)
```

Title:

The title is the text `Example of Linear Regression` you see above the plot in the figure. We can set a title to a plot using the `title()` method.

```
plt.title("Example of Linear Regression")
```

Grid:

When the grid is enabled, a set of horizontal and vertical lines will be added at the background layer of the plot. It can be useful for a rough estimation of value at a particular coordinate, just by looking at the plot.
In the code, it was activated using the `grid()` method and passing `b=True` as a parameter.

```
plt.grid(b=True, which='major', color='#666666', linestyle='dotted')
```

The arguments `which` , `color` and `linestye` help you customize the appearance of the grid.

Legend:

Legends are labeled representation that helps us recognize the different graphs that can sometimes be in a single figure.
In the code, you can see its usage by calling the `legend()` method. The labels are automatically populated using the `label` attribute you provide when creating plots.

```
plt.legend()
```

Markers

Markers can aid in identifying the locations of the data points on the line graph. In the plot above, the markers are added using the `markers()` method. There are various types of markers you can use. You define this with the `marker` attribute when calling the `plot` method. You can read about all the different markers here (https://matplotlib.org/3.1.1/api/markers_api.html).

Size

You can define the figure size using the `plt.figure(figsize=(_,_)))` method. You need to provide the size in inches.

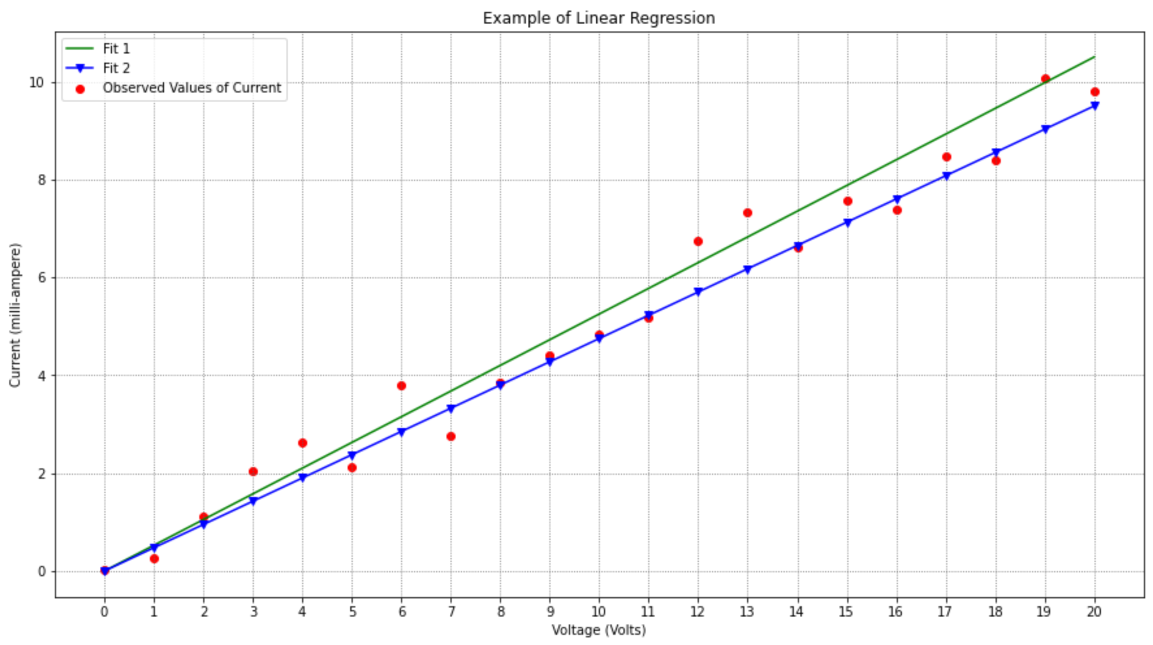
```
plt.figure(figsize=(15,8))
```

An Example Plot

Let us start with an example figure. We are going to use this figure as a reference for the majority of this reading material.

In [ ]:

```python
from matplotlib import pyplot as plt
import numpy as np

#set seed for random number generator
np.random.seed(4)

#Set x values as evenly spaced numbers over certain interval. We take 21 values fro
x=np.linspace(0,20,21)

# example of observed data, random error has been added.
s=x/2+np.random.random((21))-np.random.random((21))

# example of linear fit of observed data
y=np.linspace(0,10.5,21)

# another example of linear fit of observed data
y1=np.linspace(0,9.5,21)

#set size of figure. figure(figsize=(1,1)) would create an inch-by-inch image.
plt.figure(figsize=(15,8))

#set title of the plot
plt.title("Example of Linear Regression")

#set x and y labels
plt.ylabel('Current (milli-ampere)')
plt.xlabel('Voltage (Volts)')

#scatter plot of observed data
s_plot = plt.scatter(x,s, label='Observed Values of Current', color='red')

#plot y versus x as lines
y_plot = plt.plot(y, label='Fit 1', color='green')

#plot y1 versus x as lines and markers
y_plot = plt.plot(y1, label='Fit 2', color='blue',marker='v')

#set the current tick locations of the x-axis
plt.xticks(x)

#configure the grid lines with certain color and dotted linestyle.
plt.grid(b=True, which='major', color='#666666', linestyle='dotted')

#place a legend on the axes.
plt.legend()

#display the figure
plt.show()
```



Two interfaces of matplotlib.

There are two ways you can create plots using matplotlib. "matplotlib" defines them as `state-based interface` and `object-oriented (OO) interface`.

State-based interface is based on MATLAB uses syntaxes similar to it. All the matplotlib code you've seen so far used the state-based interface. All of these tools are encapsulated under the `pyplot` module. This interface is easy to implement and good for quick plotting. However, this interface only keeps track of the current figure you're working on. Any pyplot methods called only affect that current figure.

Object-oriented (OO) interface An object-oriented interface gives you more control over all your plots. There is no current figure, and you can apply any method to any figure at any moment. The plots are created using instances of the `Axes` and `Figure` class. An `Axes` object is used to render visualizations on a `Figure` object.

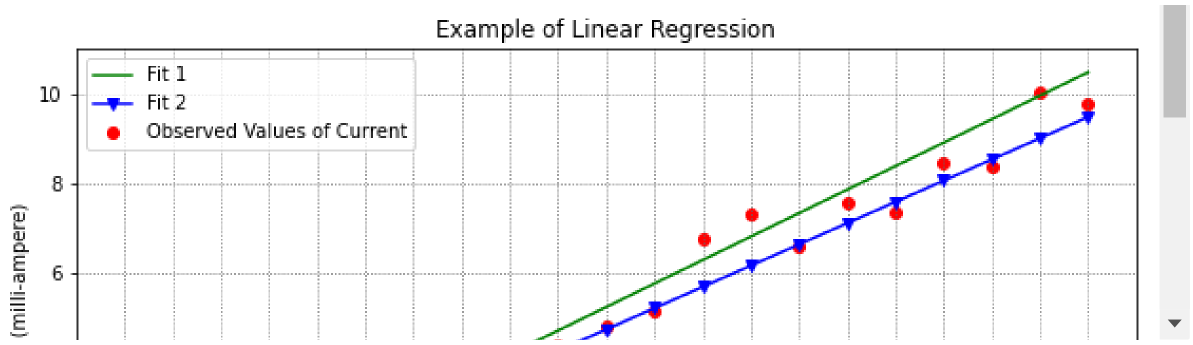
The plot from above in this notebook can be recreated in the following way:

In [ ]:

```python
#set seed for random number generator
np.random.seed(4)

#Set x values as evenly spaced numbers over certain interval.We take 21 values from
x = np.linspace(0,20,21)

# example of observed data, random error has been added.
s = x/2+np.random.random((21))-np.random.random((21))

# example of linear fit of observed data
y = np.linspace(0,10.5,21)

# another example of linear fit of observed data
y1 = np.linspace(0,9.5,21)

# create a figure object
fig = plt.figure(figsize=(10, 5))

# create an axes object in the figure
ax = fig.add_subplot(1, 1, 1)

#set title of the axes object
ax.set_title("Example of Linear Regression")

#set x and y labels
ax.set_ylabel('Current (milli-ampere)')
ax.set_xlabel('Voltage (Volts)')

#scatter plot of observed data
ax.scatter(x,s, label='Observed Values of Current', color='red')

#plot y versus x as lines
ax.plot(y, label='Fit 1', color='green')

#plot y1 versus x as lines and markers
ax.plot(y1, label='Fit 2', color='blue',marker='v')

#set the current tick locations of the x-axis
ax.set_xticks(x)

#configure the grid lines with certain color and dotted linestyle.
ax.grid(b=True, which='major', color='#666666', linestyle='dotted')

#place a legend on the axes.
ax.legend()
```

Out[6]:

```
<matplotlib.legend.Legend at 0x7f4d727ad0f0>
```



Here we've called the `plt.subplots()` method to get an instance of `Figure` and `Axes` classes.

We define the title, x-axis label and y-axis labels using the following:

```
ax.set_title("Example of Linear Regression")
ax.set_ylabel('Current (milli-ampere)')
ax.set_xlabel('Voltage (Volts)')
```

We then add the three plots using:

```
ax.scatter(x,s, label='Observed Values of Current', color='red')
ax.plot(y, label='Fit 1', color='green')
ax.plot(y1, label='Fit 2', color='blue',marker='v')
```

It all seems very similar to the `state-based interface` ; this is because we are plotting something relatively simple here. As the figure gets more and more complex, using the object-oriented approach becomes a must.

Saving matplotlib figures

Saving a figure in matplotlib is very simple. If you're using the state-based interface, you have to use the `savefig()` method of the `pyplot` class.

```
plt.savefig('figure.png')
```

Do note that you have to call the `savefig()` method before the `show()` method.

If you're using the object-oriented interface, you have to use the `savefig()` method of the `Figure` class.

```
fig.savefig('figure.png')
```

Types of Visualization

Matplotlib provides a wide range of plots along with accessible APIs to generate them. Some common ones you may have heard of include:

- Line Plots
- Bar Plots
- Histograms
- Scatter Plots
- Pie Charts

In this notebook, we'll discuss these plots along with some others and describe when you should use what type of plot.

In [ ]:

```
# Common imports
import matplotlib.pyplot as plt
import numpy as np
```

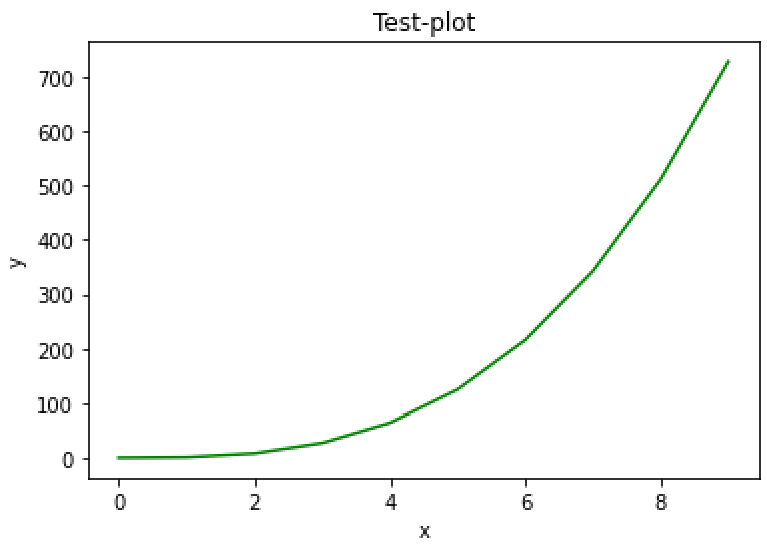
Line plot

A line plot is the simplest type of plot, and probably what first comes to mind when you think of the word "graph." Some common examples of line plots are stock prices, business profit graphs, daily temperature reports. Given any data with an X and Y coordinate, we can create a line plot.

Below you can see a simple line plot. You create line charts by using the `plt.plot()` method.

In [ ]:

```
x=np.arange(10)
y=x**3

plt.title('Test-plot')
plt.xlabel('x')
plt.ylabel('y')

plt.plot(x,y,color='green')
plt.show()
```

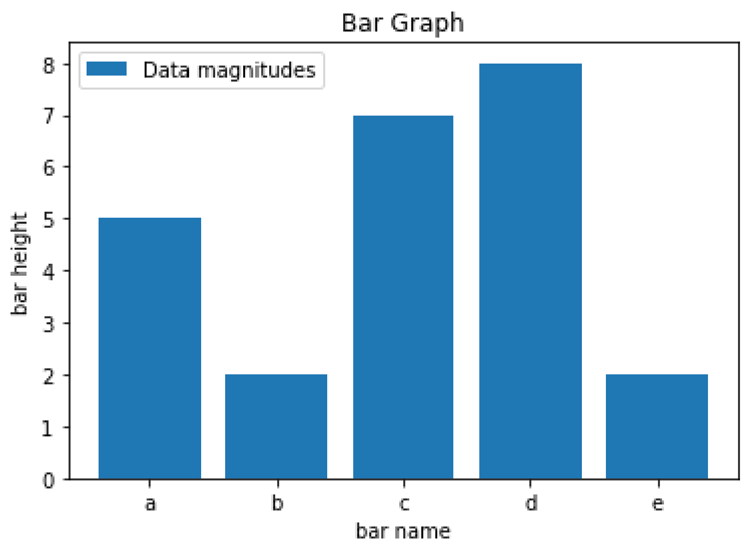


Bar Chart Plotting

A Bar Plots helps us visualize categorical data.
You can create bar plots in matplotlib using the `plt.bar()` method. Let's look at an exmaple.

In [ ]:

```
category = ['a', 'b', 'c', 'd', 'e']
magnitude = [5, 2, 7, 8, 2]

plt.bar(category, magnitude, label= "Data magnitudes")
plt.legend()
plt.xlabel('bar name')
plt.ylabel('bar height')
plt.title('Bar Graph')
plt.show()
```



Histogram

A histogram is a type of bar plot that shows the frequency distribution of a dataset. They help visualize which datapoints occur the most, and which don't.

A histogram divides data into a fixed number of defined intervals called "bins." Each bin contains the count of the data points that fall into that bin.

We plot bins into vertical (or horizontal bars. The height of the bar indicates the frequency of the data in that bin.

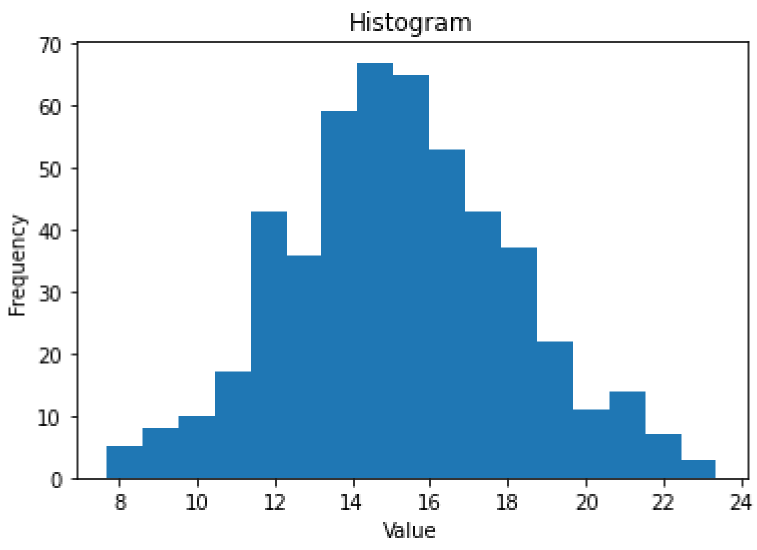
You can create a histogram in matplotlib using the `plt.hist()` method. Below is an example

In [ ]:

```
# creating some data
d = np.random.normal(loc=15, scale=3, size=500)

#plotting

plt.hist(x=d, bins='auto')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram')
```

Out[10]:

```
Text(0.5, 1.0, 'Histogram')
```



The data we've used here is 500 random samples from a normal distribution with mean 15. Because of that, you can see in the plot that most of the data centers around 15.0. Values around the edges, like 25.0, have small frequencies.

Scatter Plot

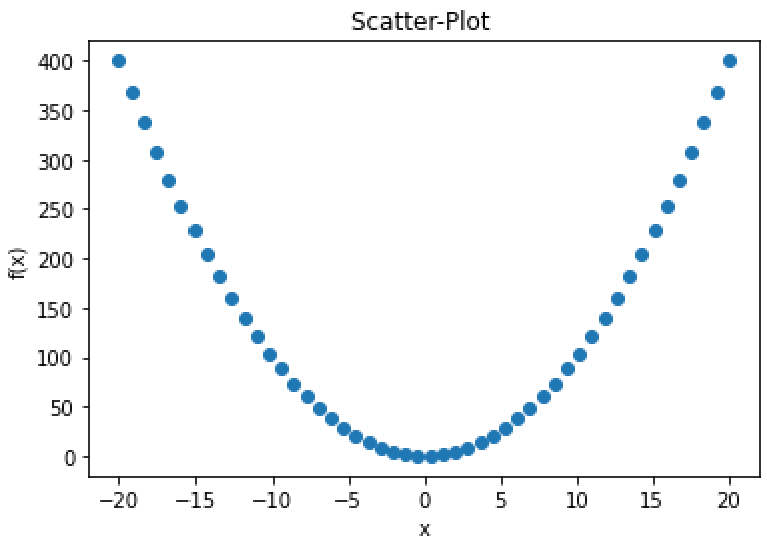
A Scatter plot helps in visualizing 2 different numeric variables with the help of dots. It shows the correlation between the two variables.

Scatter plots are useful to visualize data when using machine learning techniques like regression, where x and y are continuous variables. They also provide visulization on clusters which is handy for clustering and outlier detection.

You can create a scatter plot in matplotlib using the `plt.scatter()` method.

In [ ]:

```
x = np.linspace(-20, 20, 50)
plt.scatter(x, x**2)
plt.title('Scatter-Plot ')
plt.xlabel('x')
plt.ylabel('f(x)')
plt.show()
```



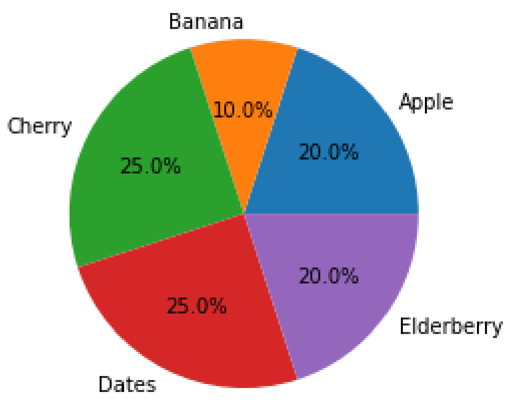
A simple addition you can make to a scatter plot is a line showing the trend of the overall data. You can define the sizes or colors of the dots, which reflects the magnitude of the data.

Pie chart

A pie chart is another one of the frequently used plots. It is a circular graph divided into slices that are proportional to the quantities they represent.
You create pie charts in matplotlib using the `plt.pie()` method.

In [ ]:

```
labels = ['Apple','Banana','Cherry','Dates','Elderberry']
sizes = [20,10,25,25,20]

plt.pie(sizes, labels=labels, autopct='%1.1f%%') #autopct adds the percentage label
plt.show()
```



Images

The computers store an image as a two-dimensional array. Every element of the array represents a pixel. Its values range from 0 to 255, which is 0 for no intensity, and 255 for full intensity. You can use matplotlib's `plt.imshow()` method to plot these images as well.

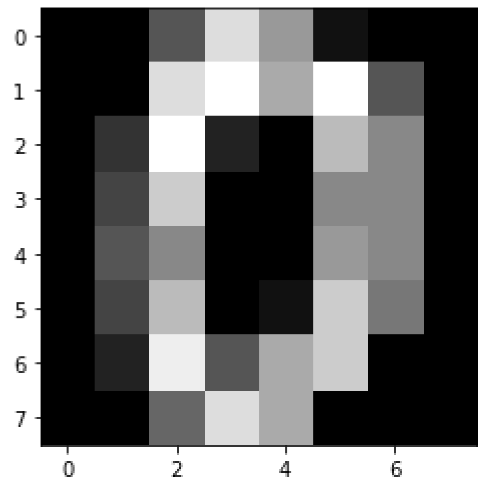
In [ ]:

```
from sklearn.datasets import load_digits
digits = load_digits()
one_digit = digits.data[0].reshape(8,8)
print(one_digit)

plt.imshow(one_digit, cmap='gray')
```

```
[[ 0.  0.  5. 13.  9.  1.  0.  0.]
 [ 0.  0. 13. 15. 10. 15.  5.  0.]
 [ 0.  3. 15.  2.  0. 11.  8.  0.]
 [ 0.  4. 12.  0.  0.  8.  8.  0.]
 [ 0.  5.  8.  0.  0.  9.  8.  0.]
 [ 0.  4. 11.  0.  1. 12.  7.  0.]
 [ 0.  2. 14.  5. 10. 12.  0.  0.]
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

Out[13]:

```
<matplotlib.image.AxesImage at 0x7f4d66b91eb8>
```



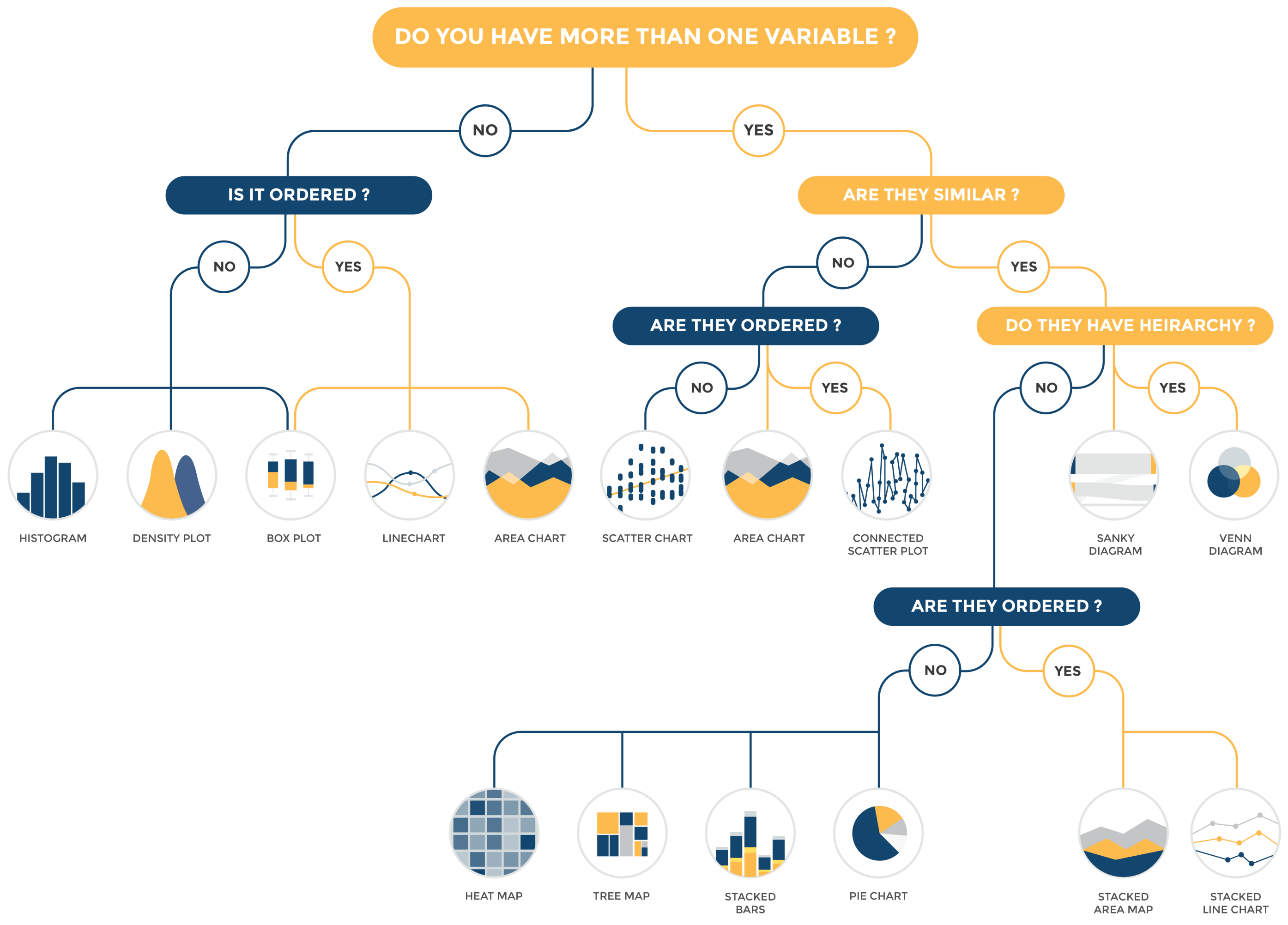
The block of code above first downloads the MNIST dataset of handwritten digits and takes a single sample. You can see that the printed 2-D array correponds to the intensity values in the image below it.

How to choose which plot to use

We've just discussed various types of plots. But one question remains. Under what situations do we use all these kinds of plots? Of course, it all depends on what sort of data we are plotting and what we want to visualize. For example, if you want to show the values of some categorical data, a bar plot is an excellent way to go. But if you want to plot the frequency distribution of some data, you would go for a histogram.

Similarly, there are multitudes of other plots (on top of the ones we've just discussed) that are useful for different kinds of data and for presenting different ideas.

You can see a summary below.



Subplots

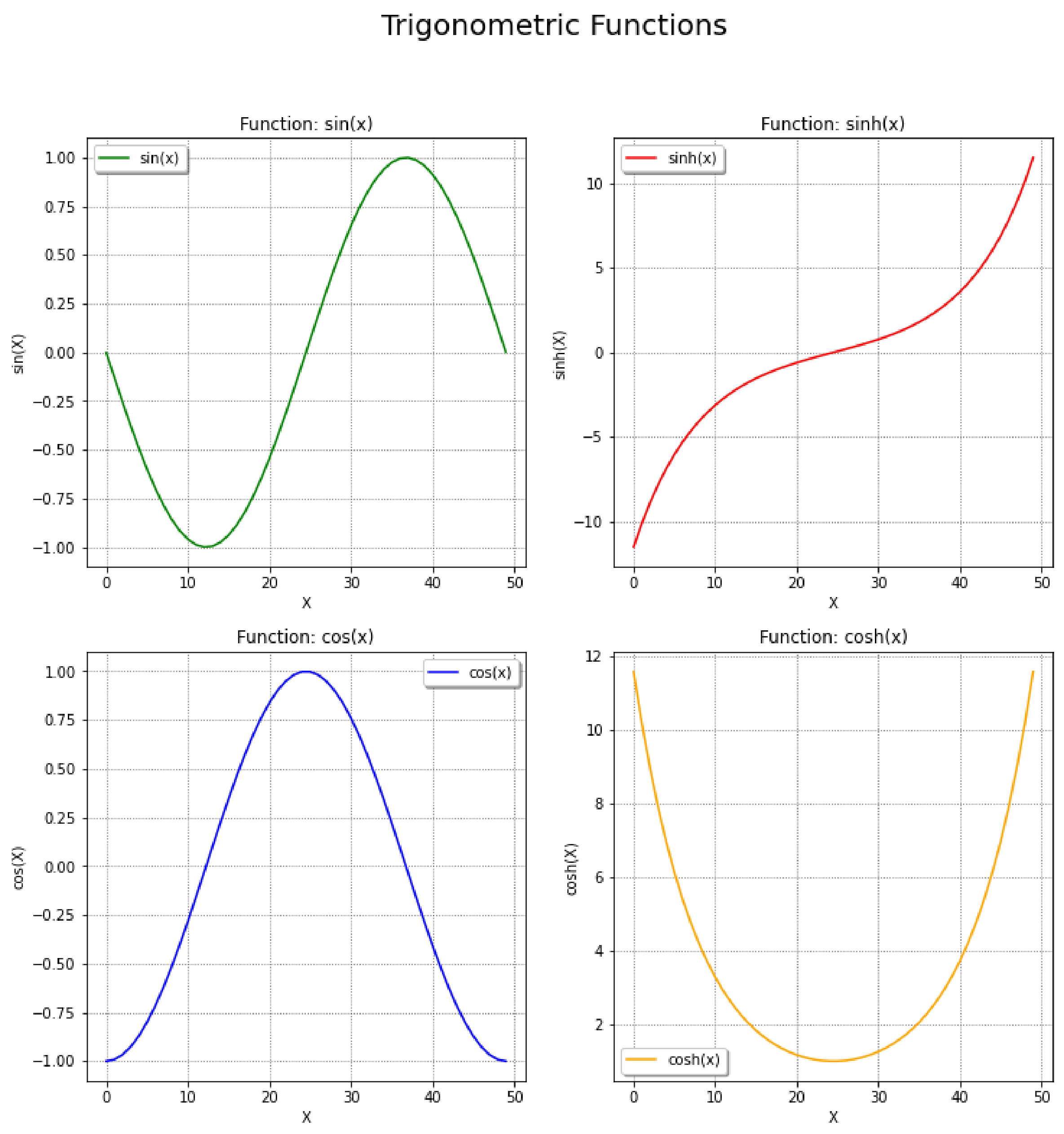
As we've seen from the previous chapters, we can add different plots in the same figure. Creating multiple plots can be very helpful when we are trying to compare and contrast multiple plots at the same time.

Plotting figures separately

We can choose to make different plots for each figure. Lets inspect the code below.

In [ ]:

```python
x=np.linspace(-3.14,3.14,50)

## create a figure and a set of subplots. 4 subplots are created
## and arranged as 2*2 subplots with total size of figure 12*12 inches.
fig, ax = plt.subplots(ncols=2, nrows=2,figsize=(12,12))

# adjust spaces between subplots
fig.subplots_adjust(hspace=0.2,wspace=0.2)

# set overall super title to whole figure
fig.suptitle("Trigonometric Functions",fontsize=20)

# plot trigonometric curve in first subplot
ax[0][0].plot(np.sin(x), label='sin(x)', color='green')

# each subplot can be accessed as [row number][column number]
# set title to first subplot
ax[0][0].set_title("Function: sin(x)")

# set x and y labels in first subplot
ax[0][0].set_ylabel('sin(X)')
ax[0][0].set_xlabel('X')

# set gridlines in first subplot
ax[0][0].grid(b=True, which='major', color='#666666', linestyle='dotted')

# put legends in first subplot
ax[0][0].legend(shadow=True)

# repeat for second sub-plot(see below first subplot)
ax[1][0].plot(np.cos(x), label='cos(x)', color='blue')
ax[1][0].set_title("Function: cos(x)")
ax[1][0].set_ylabel('cos(X)')
ax[1][0].set_xlabel('X')
ax[1][0].grid(b=True, which='major', color='#666666', linestyle='dotted')
ax[1][0].legend(shadow=True)

# repeat for third sub-plot(right side of first subplot)
ax[0][1].plot(np.sinh(x), label='sinh(x)',color='red')
ax[0][1].set_title("Function: sinh(x)")
ax[0][1].set_ylabel('sinh(X)')
ax[0][1].set_xlabel('X')
ax[0][1].grid(b=True, which='major', color='#666666', linestyle='dotted')
ax[0][1].legend(shadow=True)

# repeat for fourth sub-plot (below third subplot)
ax[1][1].plot(np.cosh(x), label='cosh(x)', color='orange')
ax[1][1].set_title("Function: cosh(x)")
ax[1][1].set_ylabel('cosh(X)')
ax[1][1].set_xlabel('X')
ax[1][1].grid(b=True, which='major', color='#666666', linestyle='dotted')
ax[1][1].legend(shadow=True)

# save figure as "figure.png"
fig.savefig('figure.png')
```



Let's go through the code line by line:

1. Set values of x between $-\pi$ and π.
2. The line `fig, ax = plt.subplots(ncols=2, nrows=2,figsize=(12,12))` declares the figure object of size (12, 12) with 4 subplots arranged in $2 * 2$ manner (i.e. 2 rows and 2 columns that can be indexed as `[row][column]` i.e. `[0][0]`, `[0][1]`, `[1][0] and [1][1]`).
3. Adjust spaces between subplots.
4. Create a "supertitle" with font size 20.

5. Plot a sine function in the first subplot with the label `'sin(x)'` and color 'green.'
6. Set a title for the first subplot.
7. Set the x and y labels for the first subplot.
8. Enable the gridlines for the first subplot.
9. Add a legend for the first subplot.
10. Repeat steps 5 to 9 for each of the remaining 3 subplots.

Customizing figure using GridSpec

We can create basic grids using both subplots() and grid space. While using subplots() is more straightforward, the advantage of using `gridspec` is that it helps to create subplots that can span rows and columns using NumPy slice syntax for selection of the part of the "gridspec" each subplot occupies.

```
gridspec = gridspec.GridSpec(nrows, ncols)
```

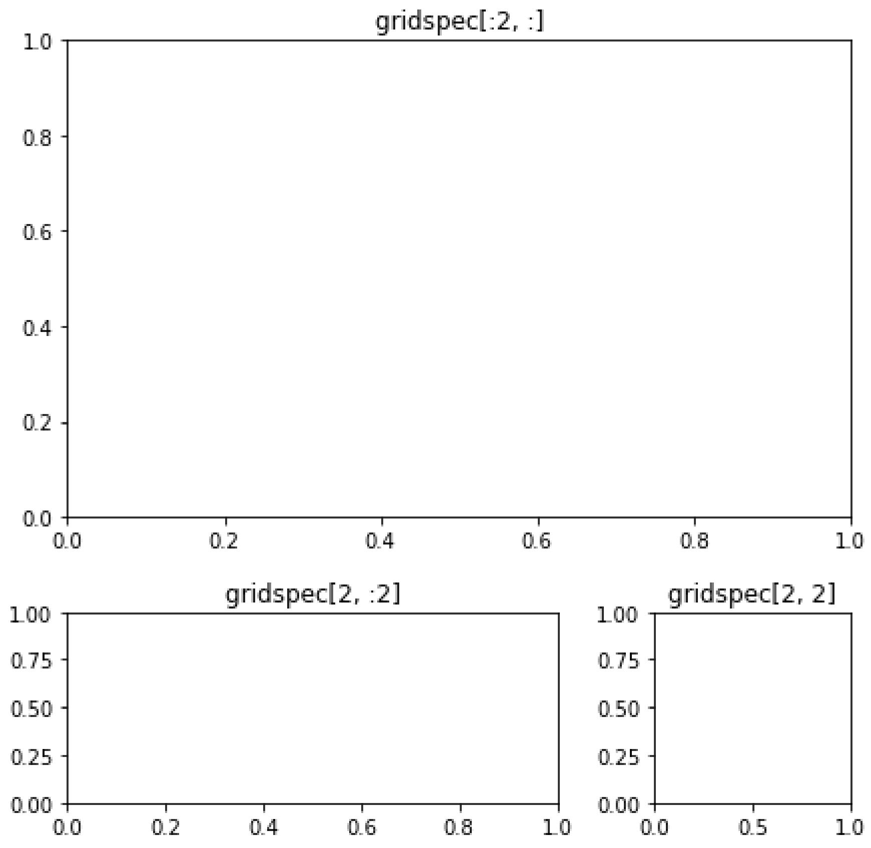
Specifies the geometry of the grid that a subplot resides. Other subplot layout parameters can also be adjusted. You can read more about customizing figures here (https://matplotlib.org/3.1.1/tutorials/intermediate/gridspec.html).

In [ ]:

```
## customizing figure layouts using GridSpec
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(7,7))

##adjust spaces between subplots
fig.subplots_adjust(wspace=0.5, hspace=0.5)

 #specify the geometry of the grid (3*3) where subplot will be placed
gridspec = gridspec.GridSpec(3, 3)

 #add an Axes to the figure as part of a subplot arrangement (topmost axes with siz
fig_ax1 = fig.add_subplot(gridspec[:2, :])
fig_ax1.set_title("gridspec[:2, :]")

 #bottom-left axes with size 1*2
fig_ax2 = fig.add_subplot(gridspec[2, :2])
fig_ax2.set_title("gridspec[2, :2]")

 #bottom-right axes with size 1*1
fig_ax3 = fig.add_subplot(gridspec[2, 2])
fig_ax3.set_title("gridspec[2, 2]")
```

Out[15]:

```
Text(0.5, 1.0, 'gridspec[2, 2]')
```



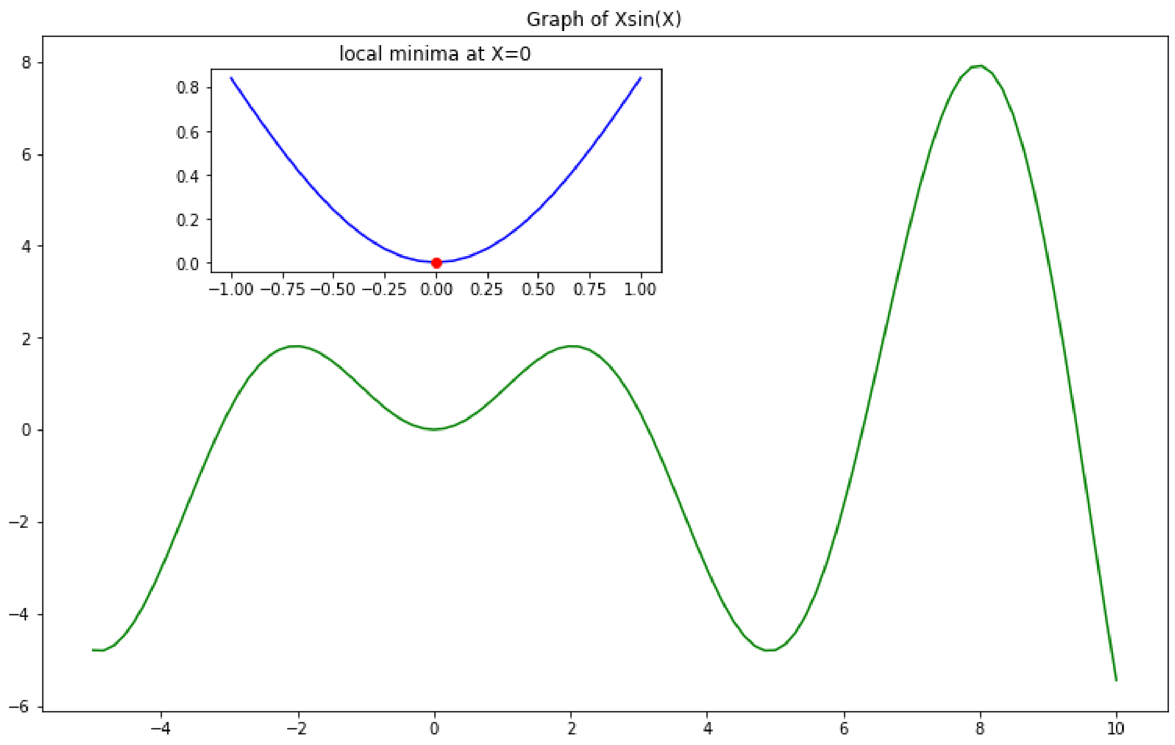
Creating multiple axes within the same subplot

Sometimes, we need to plot graphs together in the same plot. This necessity usually occurs when we have to highlight specific information about a graph or display further information about it.

In the following example, the portion of the graph with local minima is highlighted using different axes.

```
In [ ]:
```

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

## randomly generate data using numpy to plot the graphs.
x=np.linspace(-5, 10, 100)
y=np.linspace(-1,1,25)

#define a figure object of certain size.
fig = plt.figure(figsize=(10,6))

#create an axes from origin
ax1 = fig.add_axes([0, 0, 1, 1])
 #create an axes from (0.15,0.65)
ax2 = fig.add_axes([0.15, 0.65, 0.4, 0.3])

ax1.set_title('Graph of Xsin(X)')

ax1.plot(x,x*np.sin(x),color='green')

ax2.plot(y,y*np.sin(y), color='blue')
#marking the point of local minima
ax2.plot(0,0,'ro')
ax2.set_title('local minima at X=0')

plt.show()
```



Let's go through some of the significant points of the code.

- The line `fig = plt.figure(figsize=(10,6))` defines a figure object of size (10, 6). We can add axes to this object.
- The line `ax1 = fig.add_axes([0, 0, 1, 1])` creates the first axes. The initial numbers `0, 0` tell matplotlib where to start drawing the axes. Since it is (0, 0), it draws from the origin. The next two numbers define the height and width. Here, both are one. This axis eventually draws the green curve.

- The line `ax2 = fig.add_axes([0.15, 0.65, 0.4, 0.3])` defines the second axes. As you can see from the numbers (0.15, 0.65), the axes start slightly to the right and up. This eventually draws the blue curve.
- The line `ax2.plot(0,0,'ro')` plots the point of minima in a smaller graph.