What is SciPy?

SciPy is an open-source Python library which is used to solve scientific and mathematical problems. It is built on the NumPy extension and allows the user to manipulate and visualize data with a wide range of high-level commands. As mentioned earlier, SciPy builds on NumPy and therefore if you import SciPy, there is no need to import NumPy.

NumPy vs SciPy

Both NumPy and SciPy are Python libraries used for used mathematical and numerical analysis. NumPy contains array data and basic operations such as sorting, indexing, etc whereas, SciPy consists of all the numerical code. Though NumPy provides a number of functions that can help resolve linear algebra, Fourier transforms, etc, SciPy is the library that actually contains fully-featured versions of these functions along with many others. However, if you are doing scientific analysis using Python, you will need to install both NumPy and SciPy since SciPy builds on NumPy.

Subpackages in SciPy:

SciPy has a number of subpackages for various scientific computations which are shown in the following table:

Name	Description
cluster	Clustering algorithms
constants	Physical and mathematical constants
fftpack	Fast Fourier Transform routines
integrate	Integration and ordinary differential equation solvers
interpolate	Interpolation and smoothing splines
io	input and Output
linalg	Linear algebra
ndimage	N-dimensional image processing
odr	orthogonal distance regression
optimize	optimization and root-finding routines
signal	Signal processing
sparse	Sparse matrices and associated routines
spatial	Spatial data structures and algorithms
Special	Special functions
stats	Statistical distributions and functions

K-means clustering is a method for finding clusters and cluster centers in a set of unlabelled data. Intuitively, we might think of a cluster as – comprising of a group of data points, whose inter-point distances are small compared with the distances to points outside of the cluster. Given an initial set of K centers, the K-means algorithm iterates the following two steps −

- For each center, the subset of training points (its cluster) that is closer to it is identified than any other center.
- The mean of each feature for the data points in each cluster are computed, and this mean vector becomes the new center for that cluster.

These two steps are iterated until the centers no longer move or the assignments no longer change. Then, a new point **x** can be assigned to the cluster of the closest prototype. The SciPy library provides a good implementation of the K-Means algorithm through the cluster package. Let us understand how to use it.

K-Means Implementation in SciPy

We will understand how to implement K-Means in SciPy.

Import K-Means

We will see the implementation and usage of each imported function.

```
from SciPy.cluster.vq import kmeans,vq,whiten
```

```
from numpy import vstack,array
from numpy.random import rand

# data generation with three features
data = vstack((rand(100,3) + array([.5,.5,.5]),rand(100,3)))
```

Normalize a group of observations on a per feature basis. Before running K-Means, it is beneficial to rescale each feature dimension of the observation set with whitening. Each feature is divided by its standard deviation across all observations to give it unit variance.

Whiten the data

We have to use the following code to whiten the data.

```
# whitening of data
data = whiten(data)
```

Compute K-Means with Three Clusters

Let us now compute K-Means with three clusters using the following code.

```
# computing K-Means with K = 3 (2 clusters)
centroids,_ = kmeans(data,3)
```

The above code performs K-Means on a set of observation vectors forming K clusters. The K-Means algorithm adjusts the centroids until sufficient progress cannot be made, i.e. the change in distortion, since the last iteration is less than some threshold. Here, we can observe the centroid of the cluster by printing the centroids variable using the code given below.

```
print(centroids)
```

```
print(centroids)[ [ 2.26034702  1.43924335  1.3697022 ]
                  [ 2.63788572  2.81446462  2.85163854]
                  [ 0.73507256  1.30801855  1.44477558] ]
```

Assign each value to a cluster by using the code given below.

```
# assign each sample to a cluster
clx,_ = vq(data,centroids)
```

The **vq** function compares each observation vector in the 'M' by 'N' **obs** array with the centroids and assigns the observation to the closest cluster. It returns the cluster of each observation and the distortion. We can check the distortion as well. Let us check the cluster of each observation using the following code.

```
# check clusters of observation
print clx
```

SciPy Constants Package

The **scipy.constants package** provides various constants. We have to import the required constant and use them as per the requirement. Let us see how these constant variables are imported and used.

```
#Import pi constant from both the packages
from scipy.constants import pi
from math import pi

print("sciPy - pi = %.16f"%scipy.constants.pi)
print("math - pi = %.16f"%math.pi)
```

OUTPUT:

```
sciPy - pi = 3.1415926535897931
math - pi = 3.1415926535897931
```

List of Constants Available

Mathematical Constants:

Sr. No.	Constant	Description
1	pi	pi
2	golden	Golden Ratio

Physical Constants

Sr. No.	Constant & Description
1	**c** Speed of light in vacuum
2	**speed_of_light** Speed of light in vacuum
3	**h** Planck constant
4	**Planck** Planck constant h
5	**G** Newton's gravitational constant
6	**e**

	Elementary charge
7	**R** Molar gas constant
8	**Avogadro** Avogadro constant
9	**k** Boltzmann constant
10	**electron_mass(OR) m_e** Electronic mass
11	**proton_mass (OR) m_p** Proton mass
12	**neutron_mass(OR)m_n** Neutron mass