SciPy is a scientific computation library that uses NumPy underneath. It provides more utility functions for optimization, stats and signal processing. SciPy has optimized and added functions that are frequently used in NumPy and Data Science.

```
# Importing SciPy

import scipy
print(scipy.__version__)
```

Constants in SciPy: As SciPy is more focused on scientific implementations, it provides many built-in scientific constants.

```
from scipy import constants

print(constants.pi)
```

```
# List all constants:

l1 = dir(constants)

for e in l1:
  print(e)
```

```
print(constants.minute)
print(constants.hour)
print(constants.day)
print(constants.week)
print(constants.year)
print(constants.Julian_year)
```

Optimizers in SciPy: Optimizers are a set of procedures defined in SciPy that either find the minimum value of a function, or the root of an equation.

NumPy is capable of finding roots for polynomials and linear equations, but it can not find roots for non linear equations, like this one:

x + cos(x)

```
from scipy.optimize import root
from math import cos

def eqn(x):
  return x + cos(x)

myroot = root(eqn, 0)

print(myroot.x)
```

Minimizing a Function

A function, in this context, represents a curve, curves have high points and low points.

High points are called maxima.

Low points are called minima.

The highest point in the whole curve is called global maxima, whereas the rest of them are called local maxima.

The lowest point in whole curve is called global minima, whereas the rest of them are called local minima.

```
from scipy.optimize import minimize

def eqn(x):
  return x**2 + x + 2

mymin = minimize(eqn, 0, method='BFGS')

print(mymin)
```

What is Sparse Data

Sparse data is data that has mostly unused elements (elements that don't carry any information).

It can be an array like this one:

[1, 0, 2, 0, 0, 3, 0, 0, 0, 0, 0, 0]

SciPy has a module, scipy.sparse that provides functions to deal with sparse data.

There are primarily two types of sparse matrices that we use:

CSC - Compressed Sparse Column. For efficient arithmetic, fast column slicing.

CSR - Compressed Sparse Row. For fast row slicing, faster matrix vector products

We will use the CSR matrix in this tutorial.

```
# Create a CSR matrix from an array:

import numpy as np
from scipy.sparse import csr_matrix

arr = np.array([0, 0, 0, 0, 0, 1, 1, 0, 2])

print(csr_matrix(arr))
print(csr_matrix(arr).data)
```

```
arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])

print(csr_matrix(arr))
print(csr_matrix(arr).data)
```

```
# Counting nonzeros with the count_nonzero() method:
arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])

print(csr_matrix(arr).count_nonzero())
```

```
# Removing zero-entries from the matrix with the eliminate_zeros() method:
arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])

mat = csr_matrix(arr)

mat.eliminate_zeros()
print(mat)
```

```
# Eliminating duplicate entries with the sum_duplicates() method:

arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])

mat = csr_matrix(arr)
mat.sum_duplicates()

print(mat)
```

```
# Converting from csr to csc with the tocsc() method:

arr = np.array([[0, 0, 0], [0, 0, 1], [1, 0, 2]])
newarr = csr_matrix(arr).tocsc()
print(newarr)
```

Working with Graphs

Graphs are an essential data structure. SciPy provides us with the module scipy.sparse.csgraph for working with such data structures.

```
# Adjacency Matrix Representation

from scipy.sparse.csgraph import connected_components
from scipy.sparse import csr_matrix

arr = np.array([
  [0, 1, 2],
  [1, 0, 0],
  [2, 0, 0]
])
```

```
])

newarr = csr_matrix(arr)

print(connected_components(newarr))


# Use the dijkstra method to find the shortest path in a graph from one element to another.

from scipy.sparse.csgraph import dijkstra
from scipy.sparse import csr_matrix

arr = np.array([
  [0, 1, 2],
  [1, 0, 0],
  [2, 0, 0]
])

newarr = csr_matrix(arr)

print(dijkstra(newarr, return_predecessors=True, indices=0))


# Use the floyd_warshall() method to find shortest path between all pairs of elements.

import numpy as np
from scipy.sparse.csgraph import floyd_warshall
from scipy.sparse import csr_matrix

arr = np.array([
  [0, 1, 2],
  [1, 0, 0],
  [2, 0, 0]
])

newarr = csr_matrix(arr)

print(floyd_warshall(newarr, return_predecessors=True))


# The bellman_ford() method can also find the shortest path between all pairs of elements, but this method can handle negative weights as wel

import numpy as np
from scipy.sparse.csgraph import bellman_ford
from scipy.sparse import csr_matrix

arr = np.array([
  [0, -1, 2],
  [1, 0, 0],
  [2, 0, 0]
])

newarr = csr_matrix(arr)

print(bellman_ford(newarr, return_predecessors=True, indices=0))


# The depth_first_order() method returns a depth first traversal from a node.

import numpy as np
from scipy.sparse.csgraph import depth_first_order
from scipy.sparse import csr_matrix

arr = np.array([
  [0, 1, 0, 1],
  [1, 1, 1, 1],
  [2, 1, 1, 0],
  [0, 1, 0, 1]
])

newarr = csr_matrix(arr)

print(depth_first_order(newarr, 1))


# The breadth_first_order() method returns a breadth first traversal from a node.

import numpy as np
from scipy.sparse.csgraph import breadth_first_order
from scipy.sparse import csr_matrix

arr = np.array([
  [0, 1, 0, 1],
  [1, 1, 1, 1],
  [2, 1, 1, 0],
  [0, 1, 0, 1]
])

newarr = csr_matrix(arr)

print(breadth_first_order(newarr, 1))
```

Working with Spatial Data

Spatial data refers to data that is represented in a geometric space. E.g. points on a coordinate system.

```
# A Triangulation of a polygon is to divide the polygon into multiple triangles with which we can compute an area of the polygon.

import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt

points = np.array([
  [2, 4],
  [3, 4],
  [3, 0],
  [2, 2],
  [4, 1]
])

simplices = Delaunay(points).simplices

plt.triplot(points[:, 0], points[:, 1], simplices)
plt.scatter(points[:, 0], points[:, 1], color='r')

plt.show()
```

```
# A convex hull is the smallest polygon that covers all of the given points.

import numpy as np
from scipy.spatial import ConvexHull
import matplotlib.pyplot as plt

points = np.array([
  [2, 4],
  [3, 4],
  [3, 0],
  [2, 2],
  [4, 1],
  [1, 2],
  [5, 0],
  [3, 1],
  [1, 2],
  [0, 2]
])

hull = ConvexHull(points)
hull_points = hull.simplices

plt.scatter(points[:,0], points[:,1])
for simplex in hull_points:
  plt.plot(points[simplex,0], points[simplex,1], 'k-')

plt.show()
```

Distance Matrix

There are many Distance Metrics used to find various types of distances between two points in data science, Euclidean distsance, cosine distsance etc.

```
# Euclidean
from scipy.spatial.distance import euclidean
```

```
p1 = (1, 0)
p2 = (10, 2)

res = euclidean(p1, p2)

print(res)
```

```
#Manhattan
from scipy.spatial.distance import cityblock

p1 = (1, 0)
p2 = (10, 2)

res = cityblock(p1, p2)

print(res)
```

```
# Cosine
from scipy.spatial.distance import cosine

p1 = (1, 0)
p2 = (10, 2)

res = cosine(p1, p2)

print(res)
```

```
# Hamming
from scipy.spatial.distance import hamming

p1 = (True, False, True)
p2 = (False, True, True)

res = hamming(p1, p2)

print(res)
```

SciPy Interpolation

Interpolation is a method for generating points between given points.

For example: for points 1 and 2, we may interpolate and find points 1.33 and 1.66.

```
# 1D Interpolation

from scipy.interpolate import interp1d
import numpy as np

xs = np.arange(10)
ys = 2*xs + 1

interp_func = interp1d(xs, ys)

newarr = interp_func(np.arange(2.1, 3, 0.1))

print(newarr)
```

✓ 0s completed at 2:45 PM