Name: Prem Vilas Dhanawade

Github link:https://github.com/premdhanawade/Python/upload/mai (https://github.com/premdhanawade/Python/upload/main/Py

SciPy Tutorial

- SciPy Home
- SciPy Intro
- SciPy Getting Started
- SciPy Constants
- SciPy Optimizers
- SciPy Sparse Data
- SciPy Graphs
- SciPy Spatial Data
- SciPy Matlab Arrays
- SciPy Interpolation
- SciPy Significance Tests

SciPy Tutorial

- SciPy is a scientific computation library that uses NumPy underneath.
- SciPy stands for Scientific Python.

SciPy Introduction

What is SciPy?

- SciPy is a scientific computation library that uses NumPy underneath.
- SciPy stands for Scientific Python.
- It provides more utility functions for optimization, stats and signal processing.
- Like NumPy, SciPy is open source so we can use it freely.
- SciPy was created by NumPy's creator Travis Olliphant.

Why Use SciPy?

- If SciPy uses NumPy underneath, why can we not just use NumPy?
- SciPy has optimized and added functions that are frequently used in NumPy and Data Science.

Which Language is SciPy Written in?

- SciPy is predominantly written in Python, but a few segments are written in C.

Where is the SciPy Codebase?

- The source code for SciPy is located at this github repository https://github.com/scipy/scipy (https://github.com/scipy/scipy)

Checking SciPy Version

- The version string is stored under the **version** attribute.

In [1]:
```
import scipy

print(scipy.__version__)
```
```
1.5.2
```

SciPy Constants

- As SciPy is more focused on scientific implementations, it provides many built-in scientific constants.
- These constants can be helpful when you are working with Data Science.

In [2]:
```
from scipy import constants

print(constants.pi)
```
```
3.141592653589793
```

In [3]:
```python
from scipy import constants

print(dir(constants))
```

```
['Avogadro', 'Boltzmann', 'Btu', 'Btu_IT', 'Btu_th', 'ConstantWarning', 'G', 'J
ulian_year', 'N_A', 'Planck', 'R', 'Rydberg', 'Stefan_Boltzmann', 'Wien', '__al
l__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__nam
e__', '__package__', '__path__', '__spec__', '_obsolete_constants', 'acre', 'al
pha', 'angstrom', 'arcmin', 'arcminute', 'arcsec', 'arcsecond', 'astronomical_u
nit', 'atm', 'atmosphere', 'atomic_mass', 'atto', 'au', 'bar', 'barrel', 'bbl',
'blob', 'c', 'calorie', 'calorie_IT', 'calorie_th', 'carat', 'centi', 'codata',
'constants', 'convert_temperature', 'day', 'deci', 'degree', 'degree_Fahrenhei
t', 'deka', 'dyn', 'dyne', 'e', 'eV', 'electron_mass', 'electron_volt', 'elemen
tary_charge', 'epsilon_0', 'erg', 'exa', 'exbi', 'femto', 'fermi', 'find', 'fin
e_structure', 'fluid_ounce', 'fluid_ounce_US', 'fluid_ounce_imp', 'foot', 'g',
'gallon', 'gallon_US', 'gallon_imp', 'gas_constant', 'gibi', 'giga', 'golden',
'golden_ratio', 'grain', 'gram', 'gravitational_constant', 'h', 'hbar', 'hectar
e', 'hecto', 'horsepower', 'hour', 'hp', 'inch', 'k', 'kgf', 'kibi', 'kilo', 'k
ilogram_force', 'kmh', 'knot', 'lambda2nu', 'lb', 'lbf', 'light_year', 'liter',
'litre', 'long_ton', 'm_e', 'm_n', 'm_p', 'm_u', 'mach', 'mebi', 'mega', 'metri
c_ton', 'micro', 'micron', 'mil', 'mile', 'milli', 'minute', 'mmHg', 'mph', 'mu
_0', 'nano', 'nautical_mile', 'neutron_mass', 'nu2lambda', 'ounce', 'oz', 'pars
ec', 'pebi', 'peta', 'physical_constants', 'pi', 'pico', 'point', 'pound', 'pou
nd_force', 'precision', 'proton_mass', 'psi', 'pt', 'short_ton', 'sigma', 'slin
ch', 'slug', 'speed_of_light', 'speed_of_sound', 'stone', 'survey_foot', 'surve
y_mile', 'tebi', 'tera', 'test', 'ton_TNT', 'torr', 'troy_ounce', 'troy_pound',
'u', 'unit', 'value', 'week', 'yard', 'year', 'yobi', 'yotta', 'zebi', 'zepto',
'zero_Celsius', 'zetta']
```

Metric (SI) Prefixes:

- Return the specified unit in meter (e.g. centi returns 0.01)

In [4]:
```python
from scipy import constants

print(constants.yotta)    #1e+24
print(constants.zetta)    #1e+21
print(constants.exa)      #1e+18
print(constants.peta)     #1000000000000000.0
print(constants.tera)     #1000000000000.0
print(constants.giga)     #1000000000.0
print(constants.mega)     #1000000.0
print(constants.kilo)     #1000.0
print(constants.hecto)    #100.0
print(constants.deka)     #10.0
print(constants.deci)     #0.1
print(constants.centi)    #0.01
print(constants.milli)    #0.001
print(constants.micro)    #1e-06
print(constants.nano)     #1e-09
print(constants.pico)     #1e-12
print(constants.femto)    #1e-15
print(constants.atto)     #1e-18
print(constants.zepto)    #1e-21
```

```
1e+24
1e+21
1e+18
1000000000000000.0
1000000000000.0
1000000000.0
1000000.0
1000.0
100.0
10.0
0.1
0.01
0.001
1e-06
1e-09
1e-12
1e-15
1e-18
1e-21
```

Binary Prefixes:

- Return the specified unit in bytes (e.g. kibi returns 1024)

In [5]:

```python
from scipy import constants

print(constants.kibi)    #1024
print(constants.mebi)    #1048576
print(constants.gibi)    #1073741824
print(constants.tebi)    #1099511627776
print(constants.pebi)    #1125899906842624
print(constants.exbi)    #1152921504606846976
print(constants.zebi)    #1180591620717411303424
print(constants.yobi)    #1208925819614629174706176
```

```
1024
1048576
1073741824
1099511627776
1125899906842624
1152921504606846976
1180591620717411303424
1208925819614629174706176
```

Mass:

- Return the specified unit in kg (e.g. gram returns 0.001)

In [6]:
```
from scipy import constants

print(constants.gram)        #0.001
print(constants.metric_ton)  #1000.0
print(constants.grain)       #6.479891e-05
print(constants.lb)          #0.45359236999999997
print(constants.pound)       #0.45359236999999997
print(constants.oz)          #0.028349523124999998
print(constants.ounce)       #0.028349523124999998
print(constants.stone)       #6.3502931799999995
print(constants.long_ton)    #1016.0469088
print(constants.short_ton)   #907.1847399999999
print(constants.troy_ounce)  #0.031103476799999998
print(constants.troy_pound)  #0.37324172159999996
print(constants.carat)       #0.0002
print(constants.atomic_mass) #1.66053904e-27
print(constants.m_u)         #1.66053904e-27
print(constants.u)           #1.66053904e-27
```

```
0.001
1000.0
6.479891e-05
0.45359236999999997
0.45359236999999997
0.028349523124999998
0.028349523124999998
6.3502931799999995
1016.0469088
907.1847399999999
0.031103476799999998
0.37324172159999996
0.0002
1.6605390666e-27
1.6605390666e-27
1.6605390666e-27
```

Angle:

- Return the specified unit in radians (e.g. degree returns 0.017453292519943295)

In [7]:
```
from scipy import constants

print(constants.degree)     #0.017453292519943295
print(constants.arcmin)     #0.0002908882086657216
print(constants.arcminute)  #0.0002908882086657216
print(constants.arcsec)     #4.84813681109536e-06
print(constants.arcsecond)  #4.84813681109536e-06
```

```
0.017453292519943295
0.0002908882086657216
0.0002908882086657216
4.84813681109536e-06
4.84813681109536e-06
```

Time:

- Return the specified unit in seconds (e.g. hour returns 3600.0)

In [8]:
```
from scipy import constants

print(constants.minute)      #60.0
print(constants.hour)        #3600.0
print(constants.day)         #86400.0
print(constants.week)        #604800.0
print(constants.year)        #31536000.0
print(constants.Julian_year) #31557600.0
```

```
60.0
3600.0
86400.0
604800.0
31536000.0
31557600.0
```

Length:

- Return the specified unit in meters (e.g. nautical_mile returns 1852.0)

In [9]:
```python
from scipy import constants

print(constants.inch)               #0.0254
print(constants.foot)               #0.30479999999999996
print(constants.yard)               #0.9143999999999999
print(constants.mile)               #1609.3439999999998
print(constants.mil)                #2.5399999999999997e-05
print(constants.pt)                 #0.00035277777777777776
print(constants.point)              #0.00035277777777777776
print(constants.survey_foot)        #0.3048006096012192
print(constants.survey_mile)        #1609.3472186944373
print(constants.nautical_mile)      #1852.0
print(constants.fermi)              #1e-15
print(constants.angstrom)           #1e-10
print(constants.micron)             #1e-06
print(constants.au)                 #149597870691.0
print(constants.astronomical_unit) #149597870691.0
print(constants.light_year)         #9460730472580800.0
print(constants.parsec)             #3.0856775813057292e+16
```

```
0.0254
0.30479999999999996
0.9143999999999999
1609.3439999999998
2.5399999999999997e-05
0.00035277777777777776
0.00035277777777777776
0.3048006096012192
1609.3472186944373
1852.0
1e-15
1e-10
1e-06
149597870700.0
149597870700.0
9460730472580800.0
3.085677581491367e+16
```

Pressure:

- Return the specified unit in pascals (e.g. psi returns 6894.757293168361)

In [10]:
```python
from scipy import constants

print(constants.atm)         #101325.0
print(constants.atmosphere)  #101325.0
print(constants.bar)         #100000.0
print(constants.torr)        #133.32236842105263
print(constants.mmHg)        #133.32236842105263
print(constants.psi)         #6894.757293168361
```

```
101325.0
101325.0
100000.0
133.32236842105263
133.32236842105263
6894.757293168361
```

Area:

- Return the specified unit in square meters(e.g. hectare returns 10000.0)

In [11]:
```python
from scipy import constants

print(constants.hectare) #10000.0
print(constants.acre)    #4046.8564223999992
```

```
10000.0
4046.8564223999992
```

Volume:

- Return the specified unit in cubic meters (e.g. liter returns 0.001)

In [12]:
```
from scipy import constants

print(constants.liter)            #0.001
print(constants.litre)            #0.001
print(constants.gallon)           #0.0037854117839999997
print(constants.gallon_US)        #0.0037854117839999997
print(constants.gallon_imp)       #0.00454609
print(constants.fluid_ounce)      #2.9573529562499998e-05
print(constants.fluid_ounce_US)   #2.9573529562499998e-05
print(constants.fluid_ounce_imp)  #2.84130625e-05
print(constants.barrel)           #0.15898729492799998
print(constants.bbl)              #0.15898729492799998

```

```
0.001
0.001
0.0037854117839999997
0.0037854117839999997
0.00454609
2.9573529562499998e-05
2.9573529562499998e-05
2.84130625e-05
0.15898729492799998
0.15898729492799998
```

Speed:

- Return the specified unit in meters per second (e.g. speed_of_sound returns 340.5)

In [13]:
```
from scipy import constants

print(constants.kmh)            #0.2777777777777778
print(constants.mph)            #0.44703999999999994
print(constants.mach)           #340.5
print(constants.speed_of_sound) #340.5
print(constants.knot)           #0.5144444444444445
```

```
0.2777777777777778
0.44703999999999994
340.5
340.5
0.5144444444444445
```

Temperature:

- Return the specified unit in Kelvin (e.g. zero_Celsius returns 273.15)

In [14]:
```
from scipy import constants

print(constants.zero_Celsius)      #273.15
print(constants.degree_Fahrenheit) #0.5555555555555556
```

```
273.15
0.5555555555555556
```

Energy:

- Return the specified unit in joules (e.g. calorie returns 4.184)

In [15]:
```

from scipy import constants

print(constants.eV)             #1.6021766208e-19
print(constants.electron_volt) #1.6021766208e-19
print(constants.calorie)        #4.184
print(constants.calorie_th)     #4.184
print(constants.calorie_IT)     #4.1868
print(constants.erg)            #1e-07
print(constants.Btu)            #1055.05585262
print(constants.Btu_IT)         #1055.05585262
print(constants.Btu_th)         #1054.3502644888888
print(constants.ton_TNT)        #4184000000.0
```

```
1.602176634e-19
1.602176634e-19
4.184
4.184
4.1868
1e-07
1055.05585262
1055.05585262
1054.3502644888888
4184000000.0
```

Power:

- Return the specified unit in watts (e.g. horsepower returns 745.6998715822701)

In [16]:
```
from scipy import constants

print(constants.hp)         #745.6998715822701
print(constants.horsepower) #745.6998715822701
```

```
745.6998715822701
745.6998715822701
```

Force:

- Return the specified unit in newton (e.g. kilogram_force returns 9.80665)

In [17]:
```python
from scipy import constants

print(constants.dyn)              #1e-05
print(constants.dyne)             #1e-05
print(constants.lbf)              #4.4482216152605
print(constants.pound_force)      #4.4482216152605
print(constants.kgf)              #9.80665
print(constants.kilogram_force)   #9.80665
```

```
1e-05
1e-05
4.4482216152605
4.4482216152605
9.80665
9.80665
```

SciPy Optimizers

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.

Optimizing Functions

- Essentially, all of the algorithms in Machine Learning are nothing more than a complex equation that needs to be minimized with the help of given data.

Roots 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)
- For that you can use SciPy's optimze.root function.
- This function takes two required arguments:
- fun - a function representing an equation.
- x0 - an initial guess for the root.
- The function returns an object with information regarding the solution.
- The actual solution is given under attribute x of the returned object:

In [18]:
```python
from scipy.optimize import root
from math import cos

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

myroot = root(eqn, 0)

print(myroot.x)
```

```
[-0.73908513]
```

In [19]:
```python
print(myroot)
```

```
    fjac: array([[-1.]])
     fun: array([0.])
 message: 'The solution converged.'
    nfev: 9
     qtf: array([-2.66786593e-13])
       r: array([-1.67361202])
  status: 1
 success: True
       x: array([-0.73908513])
```

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.

Finding Minima

- We can use scipy.optimize.minimize() function to minimize the function.
- The minimize() function takes the following arguments:

```
                -fun - a function representing an equation.

                 -x0 - an initial guess for the root.

                 -method - name of the method to use. Legal values:
                       'CG'
                       'BFGS'
                       'Newton-CG'
                       'L-BFGS-B'
                       'TNC'
                       'COBYLA'
                       'SLSQP'

                 -callback - function called after each iteration of optimiza
         tion.

                  -options - a dictionary defining extra params:

                       {
                            "disp": boolean - print detailed description
                            "gtol": number - the tolerance of the error
                                  }
```

In [20]:

```python
from scipy.optimize import minimize

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

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

print(mymin)
```

```
     fun: 1.75
hess_inv: array([[0.50000001]])
     jac: array([0.])
 message: 'Optimization terminated successfully.'
    nfev: 8
     nit: 2
    njev: 4
  status: 0
 success: True
       x: array([-0.50000001])
```

SciPy Sparse Data

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]

- Sparse Data: is a data set where most of the item values are zero.
- Dense Array: is the opposite of a sparse array: most of the values are not zero.
- In scientific computing, when we are dealing with partial derivatives in linear algebra we will come across sparse data.

How to Work With Sparse Data

- 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.

In [1]:
```
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))
```

```
  (0, 5)	1
  (0, 6)	1
  (0, 8)	2
```

From the result we can see that there are 3 items with value.

The 1. item is in row 0 position 5 and has the value 1.

The 2. item is in row 0 position 6 and has the value 1.

The 3. item is in row 0 position 8 and has the value 2.

Sparse Matrix Methods

- Viewing stored data (not the zero items) with the data property:

In [2]:
```python
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).data)
```

```
[1 1 2]
```

In [3]:
```python
#Counting nonzeros with the count_nonzero() method:

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).count_nonzero())
```

```
3
```

In [4]:
```python
#Removing zero-entries from the matrix with the eliminate_zeros() method:

import numpy as np
from scipy.sparse import csr_matrix

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

mat = csr_matrix(arr)
mat.eliminate_zeros()

print(mat)
```

```
  (1, 2)	1
  (2, 0)	1
  (2, 2)	2
```

In [5]:
```python
#Eliminating duplicate entries with the sum_duplicates() method:

import numpy as np
from scipy.sparse import csr_matrix

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

mat = csr_matrix(arr)
mat.sum_duplicates()

print(mat)
```

```
  (1, 2)	1
  (2, 0)	1
  (2, 2)	2
```

In [6]:
```python
#Converting from csr to csc with the tocsc() method:

import numpy as np
from scipy.sparse import csr_matrix

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

newarr = csr_matrix(arr).tocsc()

print(newarr)
```

```
  (2, 0)	1
  (1, 2)	1
  (2, 2)	2
```

SciPy Graphs

Working with Graphs

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

Connected Components

- Find all of the connected components with the connected_components() method.

In [7]:
```python
import numpy as np
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))
```

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

Dijkstra

- Use the dijkstra method to find the shortest path in a graph from one element to another.
- It takes following arguments:
  - return_predecessors: boolean (True to return whole path of traversal otherwise False).

- indices: index of the element to return all paths from that element only.
- limit: max weight of path.

In [8]:
```python
import numpy as np
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))
```

```
(array([0., 1., 2.]), array([-9999,     0,     0]))
```

Floyd Warshall

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

In [9]:
```python
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))
```

```
(array([[0., 1., 2.],
       [1., 0., 3.],
       [2., 3., 0.]]), array([[-9999,     0,     0],
       [    1, -9999,     0],
       [    2,     0, -9999]]))
```

Bellman Ford

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

In [10]:
```python
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))
```

```
(array([ 0., -1.,  2.]), array([-9999,     0,     0]))
```

Depth First Order

- The depth_first_order() method returns a depth first traversal from a node.
- This function takes following arguments:
  - the graph.
  - the starting element to traverse graph from.

In [11]:
```python
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))
```

```
(array([1, 0, 3, 2]), array([    1, -9999,     1,     0]))
```

Breadth First Order

- The breadth_first_order() method returns a breadth first traversal from a node.
- This function takes following arguments:
  - the graph.
  - the starting element to traverse graph from.

In [12]:
```python
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))
```

```
(array([1, 0, 2, 3]), array([    1, -9999,     1,     1]))
```

SciPy Spatial Data

Working with Spatial Data

- Spatial data refers to data that is represented in a geometric space.
- E.g. points on a coordinate system.
- We deal with spatial data problems on many tasks.
- E.g. finding if a point is inside a boundary or not.
- SciPy provides us with the module scipy.spatial, which has functions for working with spatial data.

Triangulation

- A Triangulation of a polygon is to divide the polygon into multiple triangles with which we can compute an area of the polygon.
- A Triangulation with points means creating surface composed triangles in which all of the given points are on at least one vertex of any triangle in the surface.
- One method to generate these triangulations through points is the Delaunay() Triangulation.

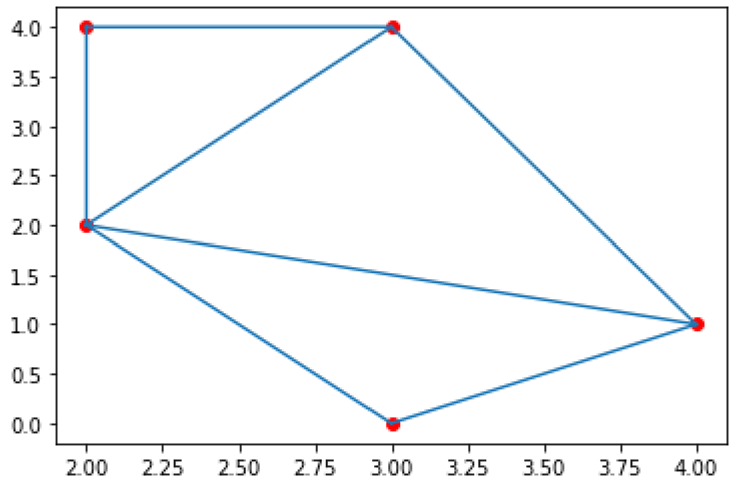
In [13]:
```python
import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt

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()
```



Convex Hull

- A convex hull is the smallest polygon that covers all of the given points.
- Use the ConvexHull() method to create a Convex Hull.

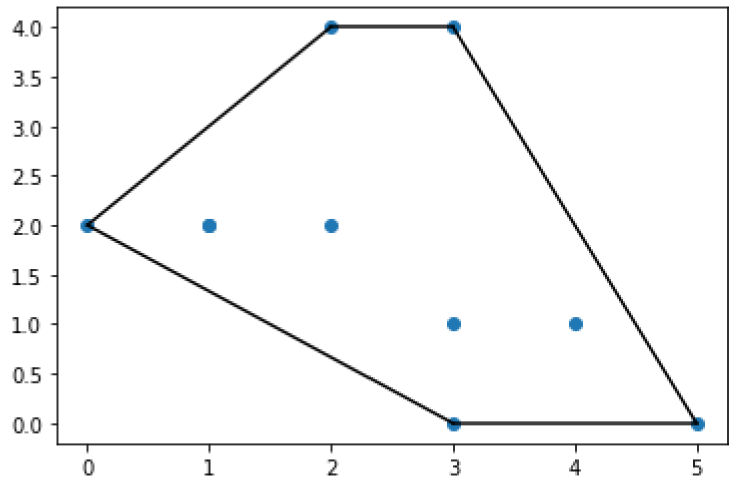
In [14]:
```python
import numpy as np
from scipy.spatial import ConvexHull
import matplotlib.pyplot as plt

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()
```



KDTrees

- KDTrees are a datastructure optimized for nearest neighbor queries.
- E.g. in a set of points using KDTrees we can efficiently ask which points are nearest to a certain given point.
- The KDTree() method returns a KDTree object.
- The query() method returns the distance to the nearest neighbor and the location of the neighbors.

In [15]:
```
from scipy.spatial import KDTree

points = [(1, -1), (2, 3), (-2, 3), (2, -3)]

kdtree = KDTree(points)

res = kdtree.query((1, 1))

print(res)
```

```
(2.0, 0)
```

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.
- The distance between two vectors may not only be the length of straight line between them, it can also be the angle between them from origin, or number of unit steps required etc.
- Many of the Machine Learning algorithm's performance depends greatly on distance metrices. E.g. "K Nearest Neighbors", or "K Means" etc.
- Let us look at some of the Distance Metrices:

Euclidean Distance

- Find the euclidean distance between given points.

In [16]:
```
from scipy.spatial.distance import euclidean

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

res = euclidean(p1, p2)

print(res)
```

```
9.219544457292887
```

Cityblock Distance (Manhattan Distance)

- Is the distance computed using 4 degrees of movement.
- E.g. we can only move: up, down, right, or left, not diagonally.

In [17]:
```python
from scipy.spatial.distance import cityblock

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

res = cityblock(p1, p2)

print(res)
```

```
11
```

Cosine Distance

- Is the value of cosine angle between the two points A and B.

In [18]:
```python
from scipy.spatial.distance import cosine

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

res = cosine(p1, p2)

print(res)
```

```
0.019419324309079777
```

Hamming Distance

- Is the proportion of bits where two bits are difference.
- It's a way to measure distance for binary sequences.

In [19]:
```python
from scipy.spatial.distance import hamming

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

res = hamming(p1, p2)

print(res)
```

```
0.6666666666666666
```

SciPy Matlab Arrays

Working With Matlab Arrays

- We know that NumPy provides us with methods to persist the data in readable formats for Python. But SciPy provides us with interoperability with Matlab as well.
- SciPy provides us with the module scipy.io, which has functions for working with Matlab arrays.

Exporting Data in Matlab Format

- The savemat() function allows us to export data in Matlab format.
- The method takes the following parameters:
    - filename - the file name for saving data.
    - mdict - a dictionary containing the data.
    - do_compression - a boolean value that specifies whether to compress the result or not. Default False.

In [20]:
```python
from scipy import io
import numpy as np

arr = np.arange(10)

io.savemat('arr.mat', {"vec": arr})
```

Import Data from Matlab Format

- The loadmat() function allows us to import data from a Matlab file.
- The function takes one required parameter:
    - filename - the file name of the saved data.
- It will return a structured array whose keys are the variable names, and the corresponding values are the variable values.

In [21]:
```python
from scipy import io
import numpy as np

arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9,])

# Export:
io.savemat('arr.mat', {"vec": arr})

# Import:
mydata = io.loadmat('arr.mat')

print(mydata)
```

```
{'__header__': b'MATLAB 5.0 MAT-file Platform: nt, Created on: Sat Jul 10 13:1
8:44 2021', '__version__': '1.0', '__globals__': [], 'vec': array([[0, 1, 2, 3,
4, 5, 6, 7, 8, 9]])}
```

In [22]:
```
print(mydata['vec'])
```

```
[[0 1 2 3 4 5 6 7 8 9]]
```

In [23]:
```
# Import:
mydata = io.loadmat('arr.mat', squeeze_me=True)

print(mydata['vec'])
```

```
[0 1 2 3 4 5 6 7 8 9]
```

SciPy Interpolation

What is 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.
- Interpolation has many usage, in Machine Learning we often deal with missing data in a dataset, interpolation is often used to substitute those values.
- This method of filling values is called imputation.
- Apart from imputation, interpolation is often used where we need to smooth the discrete points in a dataset.

How to Implement it in SciPy?

- SciPy provides us with a module called scipy.interpolate which has many functions to deal with interpolation:

1D Interpolation

- The function interp1d() is used to interpolate a distribution with 1 variable.
- It takes x and y points and returns a callable function that can be called with new x and returns corresponding y.

In [24]:
```python
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)
```

```
[5.2 5.4 5.6 5.8 6.  6.2 6.4 6.6 6.8]
```

Spline Interpolation

- In 1D interpolation the points are fitted for a single curve whereas in Spline interpolation the points are fitted against a piecewise function defined with polynomials called splines.
- The UnivariateSpline() function takes xs and ys and produce a callable funciton that can be called with new xs.

In [26]:
```python
from scipy.interpolate import UnivariateSpline
import numpy as np

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

interp_func = UnivariateSpline(xs, ys)

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

print(newarr)
```

```
[5.62826474 6.03987348 6.47131994 6.92265019 7.3939103  7.88514634
 8.39640439 8.92773053 9.47917082]
```

Interpolation with Radial Basis Function

- Radial basis function is a function that is defined corresponding to a fixed reference point.
- The Rbf() function also takes xs and ys as arguments and produces a callable function that can be called with new xs.

In [27]:
```python
from scipy.interpolate import Rbf
import numpy as np

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

interp_func = Rbf(xs, ys)

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

print(newarr)
```

```
[6.25748981 6.62190817 7.00310702 7.40121814 7.8161443  8.24773402
 8.69590519 9.16070828 9.64233874]
```

SciPy Statistical Significance Tests

What is Statistical Significance Test?

- In statistics, statistical significance means that the result that was produced has a reason behind it, it was not produced randomly, or by chance.
- SciPy provides us with a module called scipy.stats, which has functions for performing statistical significance tests.
- Here are some techniques and keywords that are important when performing such tests:

Hypothesis in Statistics

- Hypothesis is an assumption about a parameter in population.

Null Hypothesis

- It assumes that the observation is not stastically significant.

Alternate Hypothesis

- It assumes that the observations are due to some reason.
- Its alternate to Null Hypothesis.
- Example:

```
 - For an assessment of a student we would take:

        - "student is worse than average" - as a null hypothesis, an
d:

        - "student is better than average" - as an alternate hypothe
sis.
```

One tailed test

- When our hypothesis is testing for one side of the value only, it is called "one tailed test".
- Example:

```
- For the null hypothesis:

    - "the mean is equal to k", we can have alternate hypothesis:

    - "the mean is less than k", or:

    - "the mean is greater than k"
```

Two tailed test

- When our hypothesis is testing for both side of the values.
- Example:

```
  - For the null hypothesis:

        - "the mean is equal to k", we can have alternate hypothesi
s:

        - "the mean is not equal to k"
```

- In this case the mean is less than, or greater than k, and both sides are to be checked.

Alpha value

- Alpha value is the level of significance.
- Example:

```
- How close to extremes the data must be for null hypothesis to be r
ejected.
```

- It is usually taken as 0.01, 0.05, or 0.1.

P value

- P value tells how close to extreme the data actually is.
- P value and alpha values are compared to establish the statistical significance.
- If p value <= alpha we reject the null hypothesis and say that the data is statistically significant. otherwise we accept the null hypothesis.

T-Test

- T-tests are used to determine if there is significant deference between means of two variables. and lets us know if they belong to the same distribution.
- It is a two tailed test.
- The function ttest_ind() takes two samples of same size and produces a tuple of t-statistic and p-value.

In [28]:

```
import numpy as np
from scipy.stats import ttest_ind

v1 = np.random.normal(size=100)
v2 = np.random.normal(size=100)

res = ttest_ind(v1, v2)

print(res)
```

```
Ttest_indResult(statistic=-0.8617882600948075, pvalue=0.3898465343113394)
```

In [30]:

```

res = ttest_ind(v1, v2).pvalue

print(res)
```

```
0.3898465343113394
```

KS-Test

- KS test is used to check if given values follow a distribution.
- The function takes the value to be tested, and the CDF as two parameters.
- A CDF can be either a string or a callable function that returns the probability.
- It can be used as a one tailed or two tailed test.
- By default it is two tailed. We can pass parameter alternative as a string of one of two-sided, less, or greater.

In [31]:
```python
import numpy as np
from scipy.stats import kstest

v = np.random.normal(size=100)

res = kstest(v, 'norm')

print(res)
```

```
KstestResult(statistic=0.06291507250733158, pvalue=0.7998971593110737)
```

Statistical Description of Data

- In order to see a summary of values in an array, we can use the describe() function.
- It returns the following description:

```
    - number of observations (nobs)
    - minimum and maximum values = minmax
    - mean
    - variance
    - skewness
    - kurtosis
```

In [32]:
```python
import numpy as np
from scipy.stats import describe

v = np.random.normal(size=100)
res = describe(v)

print(res)
```

```
DescribeResult(nobs=100, minmax=(-2.0680392565513896, 2.028184894953221), mean=
0.053198925270453, variance=0.9346339490663652, skewness=0.09399856063273393, k
urtosis=-0.5300569187974569)
```

Normality Tests (Skewness and Kurtosis)

- Normality tests are based on the skewness and kurtosis.
- The normaltest() function returns p value for the null hypothesis:
- "x comes from a normal distribution".

Skewness:

- A measure of symmetry in data.
- For normal distributions it is 0.
- If it is negative, it means the data is skewed left.

- If it is positive it means the data is skewed right.

Kurtosis:

- A measure of whether the data is heavy or lightly tailed to a normal distribution.
- Positive kurtosis means heavy tailed.
- Negative kurtosis means lightly tailed.

In [33]:

```python
import numpy as np
from scipy.stats import skew, kurtosis

v = np.random.normal(size=100)

print(skew(v))
print(kurtosis(v))
```

```
-0.11313903999157268
-0.13182898079547245
```

In [34]:

```python
import numpy as np
from scipy.stats import normaltest

v = np.random.normal(size=100)

print(normaltest(v))
```

```
NormaltestResult(statistic=3.4403845179793002, pvalue=0.17903172414406068)
```

In [ ]:

```

```