In [ ]:
```
                  "SciPy  Module"
```

SciPy library:->is a scientific computaion library that uses NumPy underneath

SciPy stands for scientific python

It provides more utility functions for optimisation,stats and signal processing.

SciPy is a open sourse so we can use it freely.

SciPy created by NumPy's creator travis Olliphant

In [ ]:
```
"Uses Of SciPy:"
```

If SciPy uses NumPy Underneath,Why can we not just use NumPy?

SciPy has Optimised added functions that are frequently used in NumPy and Data Science.

It is also used in fourier transforms and signal processing and filtering data sets and Convolution:

In [ ]:
```
Which Language is SciPy Written in?
```

SciPy is predominently written in Python,but a few segements are written in C.

In [ ]:
```
"Installation of SciPy"
```

we have already Python and PIP installed on our system ,then installation of SciPy is very easy.

command:--->pip install scipy

If this command fails, then use a Python distribution that already has SciPy installed like, Anaconda, Spyder etc.

In [6]:
```
pip install scipy
```

```
Requirement already satisfied: scipy in c:\users\gadam\anaconda3\lib\site-packages (1.13.1)
Requirement already satisfied: numpy<2.3,>=1.22.4 in c:\users\gadam\anaconda3\lib\site-packages (from scipy) (1.
26.4)
Note: you may need to restart the kernel to use updated packages.
```

In [ ]:
```
"import SciPy"
```

Once our SciPy is installed ,import the SciPy modules you want to use in your applications by adding the 'from scipy import module' statement.

In [13]:
```
# ex:
# let's find how many cubic liters in one liter:

from scipy import constants

print(constants.liter)
```

```
0.001
```

In [ ]:
```
"constants"
```

SciPy offers a set of mathematical constants,one of them is 'liter' which returns 1 liter as cubic meters.

In [ ]:
```
"checking SciPy version"
```

In [17]:
```
import scipy
print(scipy.__version__)
```

```
1.13.1
```

In [ ]:
```
"Constants in SciPy"
```

As sciPy is more focused on scientific implementations ,it provides many built in scientific constants.

These constants helps when we are working with data scinece.

In [ ]:
```
PI is an example of a scietific constant.
```

In [20]:
```
from scipy import constants
print(constants.pi)
```

```
3.141592653589793
```

In [ ]:
```
"constants Units"
```

A list of all units under the constants module can be seen using the dit() function.

In [25]:
```
from scipy import constants
dir(constants)
```

Out[25]:
```
['Avogadro',
 'Boltzmann',
 'Btu',
 'Btu_IT',
 'Btu_th',
 'ConstantWarning',
 'G',
 'Julian_year',
 'N_A',
 'Planck',
 'R',
 'Rydberg',
 'Stefan_Boltzmann',
 'Wien',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_codata',
 '_constants',
 '_obsolete_constants',
 'acre',
 'alpha',
 'angstrom',
 'arcmin',
 'arcminute',
 'arcsec',
 'arcsecond',
 'astronomical_unit',
 '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_Fahrenheit',
 'deka',
 'dyn',
 'dyne',
 'e',
 'eV',
 'electron_mass',
 'electron_volt',
 'elementary_charge',
 'epsilon_0',
 'erg',
 'exa',
 'exbi',
 'femto',
 'fermi',
 'find',
 'fine_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',
'hectare',
'hecto',
'horsepower',
'hour',
'hp',
'inch',
'k',
'kgf',
'kibi',
'kilo',
'kilogram_force',
'kmh',
'knot',
'lambda2nu',
'lb',
'lbf',
'light_year',
'liter',
'litre',
'long_ton',
'm_e',
'm_n',
'm_p',
'm_u',
'mach',
'mebi',
'mega',
'metric_ton',
'micro',
'micron',
'mil',
'mile',
'milli',
'minute',
'mmHg',
'mph',
'mu_0',
'nano',
'nautical_mile',
'neutron_mass',
'nu2lambda',
'ounce',
'oz',
'parsec',
'pebi',
'peta',
'physical_constants',
'pi',
'pico',
'point',
'pound',
'pound_force',
'precision',
'proton_mass',
'psi',
'pt',
'quecto',
'quetta',
'ronna',
'ronto',
'short_ton',
'sigma',
'slinch',
'slug',
'speed_of_light',
'speed_of_sound',
'stone',
'survey_foot',
```

```
'survey_mile',
'tebi',
'tera',
'test',
'ton_TNT',
'torr',
'troy_ounce',
'troy_pound',
'u',
'unit',
'value',
'week',
'yard',
'year',
'yobi',
'yocto',
'yotta',
'zebi',
'zepto',
'zero_Celsius',
'zetta']
```

In [ ]:
```
"Units Categories"
```

In [ ]:
```
1.Metric
2.Binary
3.Mass
4.Angle
5.Time
6.Length
7.Pressure
8.Volume
9.Speed
10.Temparature
11.Energy
12.Power
133.Force
```

In [ ]:
```
"Metric(SI) Prefixes"
```

In [ ]:
```
return the specified unit in meter (e.g centi return 0.01)
```

In [33]:
```
#here we have some metric units
from scipy import constants

print(constants.yotta)
print(constants.zetta)
print(constants.exa)
print(constants.peta)
print(constants.tera)
print(constants.giga)
print(constants.mega)
print(constants.kilo)
print(constants.hecto)
print(constants.deka)
print(constants.deci)
print(constants.centi)
print(constants.milli)
print(constants.micro)
print(constants.nano)
print(constants.pico)
print(constants.femto)
print(constants.atto)
print(constants.zepto)
```

```
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
```

In [ ]:
```
"Binary Prefixes:"
```

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

In [36]:
```
from scipy import constants

print(constants.kibi)
print(constants.mebi)
print(constants.gibi)
print(constants.tebi)
print(constants.pebi)
print(constants.exbi)
print(constants.zebi)
print(constants.yobi)
```

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

In [ ]:
```
"Mass":

Return the specified in KG(e.g gram returns 0.001)
```

In [42]:
```
print(constants.gram)
print(constants.metric_ton)
print(constants.grain)
print(constants.lb)
print(constants.pound)
print(constants.oz)
print(constants.ounce)
print(constants.stone)
print(constants.long_ton)
print(constants.short_ton)
print(constants.troy_ounce)
print(constants.troy_pound)
print(constants.carat)
print(constants.atomic_mass)
print(constants.m_u)
print(constants.u)
```

```
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
```

In [ ]:
```
"Angle"
```

```
Formula
1Deg × π/180 = 0.01745Rad

return the specified unit in radians(e.g degree returns 0.017453292519943295)
```

In [45]:
```
print(constants.degree)
print(constants.arcmin)
print(constants.arcminute)
print(constants.arcsec)
print(constants.arcsecond)
```

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

In [ ]:
```
"Time"

return the specified unit in seconds(e.g hour returns 3600)
```

In [47]:
```
print(constants.minute)
print(constants.hour)
print(constants.day)
print(constants.week)
print(constants.year)
print(constants.Julian_year)
```

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

In [ ]:
```
"length"
Return the specified unit in meters
(e.g. nautical_mile returns 1852.0)
```

In [49]:
```
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
```

In [ ]:
```
"Pressure" Pressure (P) = Force (F) / Area (A)
```

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

In [54]:
```
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
```

In [ ]:
```
"Area"

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

In [56]:
```
from scipy import constants

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

```
10000.0
4046.8564223999992
```

In [ ]:
```
"Volume:"

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

In [58]:
```
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
```

In [ ]:
```
"Speed:"

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

In [60]:
```
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
```

In [ ]:
```
"Temperature:"
1C=273K
Return the specified unit in Kelvin (e.g. zero_Celsius returns 273.15)
```

In [62]:
```
from scipy import constants

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

```
273.15
0.5555555555555556
```

In [ ]:
```
"Energy:"
```

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

In [64]:
```
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
```

In [ ]:
```
"Power:"
Return the specified unit in watts (e.g. horsepower returns 745.6998715822701)
```

In [66]:
```
from scipy import constants

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

```
745.6998715822701
745.6998715822701
```

In [ ]:
```
"Force"

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

In [68]:
```
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
```

In [ ]:
```
"OptimiZers in SciPy"
```

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

In [ ]:
```
"Optimising Functions"
```

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

In [ ]:
```
"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 optimise.root function

this functions takes two required arguements,

fun-a function representing an equation

x0-an initial guess for the root.

the functions returns an object with information regarding the solution.

The actual solution is given under attribute x of the returned object:

In [6]:
```
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]
```

```
C:\Users\gadam\AppData\Local\Temp\ipykernel_14892\2989272106.py:4: DeprecationWarning: Conversion of an array wi
th ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your a
rray before performing this operation. (Deprecated NumPy 1.25.)
  return x+cos(x)
```

Note: The returned object has much more information about the solution.

In [9]:
```
print(myroot)
# Print all information about the solution (not just x which is the root)
```

```
message: The solution converged.
success: True
 status: 1
    fun: [ 0.000e+00]
      x: [-7.391e-01]
 method: hybr
   nfev: 9
   fjac: [[-1.000e+00]]
      r: [-1.674e+00]
    qtf: [-2.668e-13]
```

In [ ]:
```
"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,wheareas the rest of them are called local manima.

In [ ]:
```
"Finding Minima"
```

We can use scipy.optimise.minimize() function to minimize the function

The minimize() function takes the following arguements.

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'
    '
```

callback-->function called after each iteration optimization.

options-> a dictionary defining extra params:

{

```
     "disp": boolean - print detailed descriptio
     n
     "gtol": number - the tolerance of the err
     or
```

}SLSQP'

In [19]:
```
from scipy.optimize import minimize

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

mymin=minimize(eqn,0,method='BfGS')
print(mymin)
```

```
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 1.75
       x: [-5.000e-01]
     nit: 2
     jac: [ 0.000e+00]
hess_inv: [[ 5.000e-01]]
    nfev: 8
    njev: 4
```

In [21]:
```
from scipy.optimize import minimize

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

mymin=minimize(eqn,0,method='CG')
print(mymin)
```

```
message: Optimization terminated successfully.
success: True
 status: 0
    fun: 1.75
      x: [-5.000e-01]
    nit: 2
    jac: [ 2.980e-08]
   nfev: 10
   njev: 5
```

In [31]:
```
from scipy.optimize import root
import math

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

myroot=root(eqn,0)
print(myroot.x)
print(myroot)
```

```
[-0.73908513]
 message: The solution converged.
 success: True
  status: 1
     fun: [ 0.000e+00]
       x: [-7.391e-01]
  method: hybr
    nfev: 9
    fjac: [[-1.000e+00]]
       r: [-1.674e+00]
     qtf: [-2.668e-13]
```

```
C:\Users\gadam\AppData\Local\Temp\ipykernel_14892\3238690985.py:5: DeprecationWarning: Conversion of an array wi
th ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your a
rray before performing this operation. (Deprecated NumPy 1.25.)
  return x+cos(x)
```

In [ ]:
```
"SciPy 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:

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

In [ ]:
```
"Sparse Data:" is the data set where most of the item values are zero.

"Dense Array:" is the opposite of a sparse array: most of the item values are not zero.
```

In [ ]:
```
"How to Work With Sparse Data"
```

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

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

"CSE"-->Compressed Sparse Column. For efficient arithematic,fast column slicing

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

We will use the CSR matrxix in this .

In [ ]:
```
"CSR Matrix"
```

In [ ]:
```
We can create CSR matrix by passing an array into function
'scipy.sparse.csr_matrix()'
```

In [38]:
```
import numpy as np
from scipy.sparse import csr_matrix

arr=np.array([0,0,8,0,0,1,0,0,0,2])
print(csr_matrix(arr))
```

```
  (0, 2)        8
  (0, 5)        1
  (0, 9)        2
```

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

The 8. item is in row 0 position 5 and has the value 1.

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

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

In [ ]:
```
"Sparse Matrix Methods"
```

viewing the data(not the zero items) with the 'data' property

In [3]:
```
import numpy as np
from scipy.sparse import csr_matrix

arr=np.array([[0,7,0],[0,0,1],[0,1,1]])
print(csr_matrix(arr).data)
```

```
[7 1 1 1]
```

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

In [5]:
```
import numpy as np
from scipy.sparse import csr_matrix

arr=np.array([[0,7,0],[0,0,1],[0,1,1]])
print(csr_matrix(arr).count_nonzero())
```

```
4
```

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

In [54]:
```
import numpy as np
from scipy.sparse import csr_matrix

arr=np.array([[0,7,0],[0,0,1],[0,1,1]])
mat=csr_matrix(arr)
mat.eliminate_zeros()
print(mat)

index -through deleting
```

```
  (0, 1)        7
  (1, 2)        1
  (2, 1)        1
  (2, 2)        1
```

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

In [56]:
```
import numpy as np
from scipy.sparse import csr_matrix

arr=np.array([[0,7,0],[0,0,1],[0,1,1]])
mat=csr_matrix(arr)
mat.sum_duplicates()
print(mat)
# Eliminating duplicates by adding them:
```

```
(0, 1)        7
(1, 2)        1
(2, 1)        1
(2, 2)        1
```

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

In [1]:
```
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
```

Note: Apart from the mentioned sparse specific operations, sparse matrices support all of the operations that normal matrices support e.g. reshaping, summing, arithemetic, broadcasting etc.

In [ ]:
```
"SciPy Graphs"
```

In [ ]:
```
"Working with Graphs"
```

Graphs are essential in data structure.

SciPy provides us with module scipy.sparse.csgraph for working with such data structures.

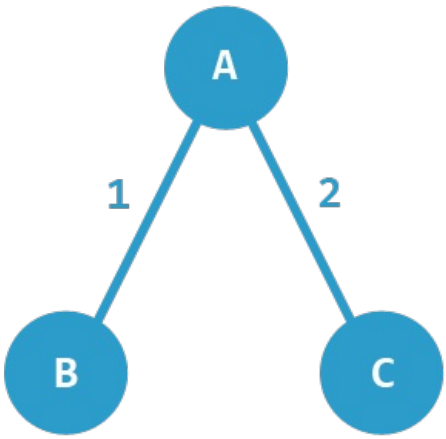
In [ ]:
```
"Adjacency Matrix"
```

Adjacency matrix is a 'nxn' matrix where n is the number of elements in a graph.

And the values represents the connection between the elements.

In [19]:
```
from PIL import Image
Image.open('scipy_graph.png')
```

Out[19]:



For a graph like this,with elements A,B and C ,the connections are:

A&B are connected with weight 1. A&C are connected with weight 2. C&B is not connected

In [ ]:
```
The Adjacency Matrix would look like this:

   A B C
A:[0 1 2]
B:[1 0 0]
C:[2 0 0]
```

Adjacency Matrix:It is a path between two nodes.

matrix is a 2 dimentional or nXn dimentions we have

Represents the conncetion between the nodes in the matrix form

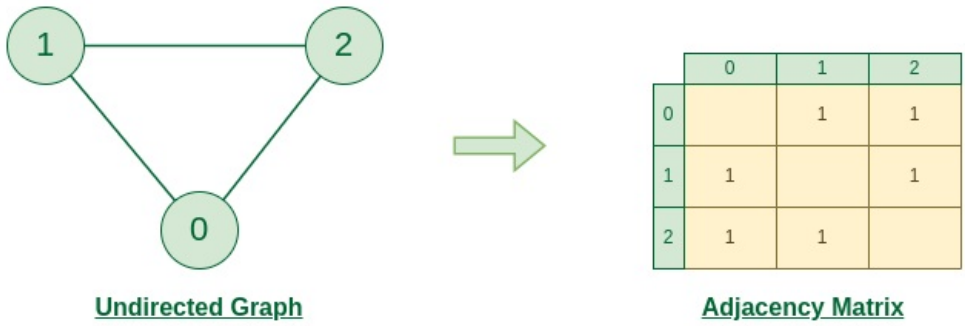
here row and column represents the nodes of the graph .

If edge is present we can give '1' otherwise take it as 0.

here we have weighted and unweighted graphs means having some values in the edge is weight.

In [6]:
```
from PIL import Image
Image.open('adjacency.png')
```

Out[6]:

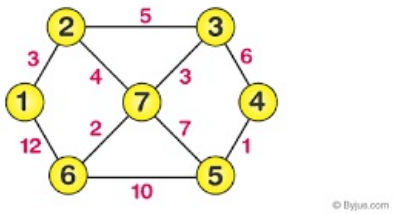


Undirected Graph | Adjacency Matrix

Graph Representation of Undirected graph to Adjacency Matrix

In [8]:
```
Image.open('adjaweighted.png')
```

Out[8]:



In [ ]:
```
"Connected Components"
```

In [ ]:
```
1.every node is connected by paths is called connected component.
2.if the no vertices is connected with super graph it treated as another graph
```

Find all of the connected components with the 'connceted_componenets()'

In [32]:
```
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]))
```

In [ ]:
```
# Output:
   A B C
A:[0 1 2]
B:[1 0 0]
C:[2 0 0]

>the graph has three nodes A ,B and C

>for A to B with weight 1.
>for A to C with weight 2.
>for B to A with weight 1.(since the graph is undirected).
> for c to A with weight 2.(since the graph is undirected).

Since all nodes are conncted through edges the graph is fully connected,
meaning there is only one conncted_component.therefore the output will be
1.'(1,array([0,0,0],dtype=int32))'

> '1' indicates there is 1 conncted component
>array([0, 0, 0]) shows that all nodes (0, 1, and 2) belong to the same connected component labeled as 0.
```

In [28]:
```
"Dijkstra algorithm"
```

Use the 'dijkstra' alogorithm to find the shortest path between two elements in a graph with non-negative edge weights.

It takes following arguements:

1.'return_predecessors': boolean(True to return the whole path of the traversal otherwise False).

2.'indices':index of the element to return all paths from that element only.

3.limit:max weight of path.

In [3]:
```
# Find the shortest path from element 1 to 2:
import numpy as np
from scipy.sparse.csgraph import dijkstra
from scipy.sparse import csr_matrix

graph=np.array([
    [0,1,2],
    [1,0,0],
    [2,0,0]])
newarr=csr_matrix(graph)
print(dijkstra(newarr,return_predecessors=True,indices=0))
```

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

output

A B C A:[0 1 2] B:[1 0 0] C:[2 0 0] 1.this matrix represents a graph with 3 nodes(A,B,C) .Each entry(i,j) in the matrix indicates the weight of the edge between nodes 'i' and 'j'.

2.for exapmle ,graph[A,B]=1 means there is an edge from node A to node B with weight 1.

3.If there is no direct edge between two nodes ,the corresponding entry in the adjacency matrix is 0.

4.Convert to Sparse Matrix:

newarr=csr_matrix(graph) #it takes the values having non-zeroes

.convert the dense adjacency matrix 'graph' into a sparse matrix format ('csr_matrix') ,which is more memory-efficiant for graphs with many zero entries(i.e.graphs that are not fully conncted)

5.Compute Shortest Paths Using Dijkstra's Algorithm:

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

*dijkstra function computes the shortest paths from a source node to all other nodes in the graph.

*newarr is the sparse matrix representation of the graph.

*return_predecessors=True indicates that the function should return an array of predecessors for each node, which helps in reconstructing the shortest path.

*indices=0 specifies that node 0 is the source node from which the shortest paths are calculated.

*Let's break down the code you provided to explain how Dijkstra's algorithm is applied using the

scipy.sparse.csgraph.dijkstra function.

Code Explanation Import Necessary Libraries:

python Copy code import numpy as np from scipy.sparse.csgraph import dijkstra from scipy.sparse import csr_matrix numpy: Provides support for large, multi-dimensional arrays and matrices. dijkstra from scipy.sparse.csgraph: Computes the shortest path using Dijkstra's algorithm. csr_matrix from scipy.sparse: Efficiently stores a sparse matrix in memory using the Compressed Sparse Row format. Define the Adjacency Matrix:

Output Explanation

(distances, predecessors) = dijkstra(newarr, return_predecessors=True, indices=0)

The output of the dijkstra function is a tuple containing two arrays:

Distances Array:

This array represents the shortest path distances from the source node (node 0) to all other nodes. Predecessors Array:

This array shows the predecessor of each node in the shortest path tree. A predecessor is a node that comes immediately before a given node in the path from the source node. .

[[0, 1, 2], [1, 0, 0], [2, 0, 0]] distances Array:

[0. 1. 2.] The distances are [0, 1, 2], which means: The distance from node 0 to itself is 0. The shortest distance from node 0 to node 1 is 1 (direct path with weight 1). The shortest distance from node 0 to node 2 is 2 (direct path with weight 2). predecessors Array:

[-9999 0 0] The predecessors are [-9999, 0, 0], which means: Node 0 has no predecessor (-9999 is used to indicate this since it is the source). The predecessor of node 1 is node 0 (the shortest path from node 0 to node 1 is direct). The predecessor of node 2 is also node 0 (the shortest path from node 0 to node 2 is direct).

In [ ]:
```
"Floyd Warshall"
```

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

In [49]:
```
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]]))
```

In [ ]:
```
distance matrix:
[[0. 1. 2.]
 [1. 0. 3.]
 [2. 3. 0.]]
Distance from node 0 to itself is 0.
Distance from node 0 to node 1 is 1.
Distance from node 0 to node 2 is 2.
Distance from node 1 to node 2 is 3 (going through node 0 with weights 1 + 2).
Distance from node 2 to node 1 is 3 (going through node 0 with weights 2 + 1).

[[-9999  0  0]
 [ 1 -9999  0]
 [ 2  0 -9999]]
The entry -9999 is used to indicate that a node is its own predecessor (diagonal elements) or there is no path.
predecessors[0, 1] = 0 means to reach node 1 from node 0, you come directly from node 0.
predecessors[1, 2] = 0 means to reach node 2 from node 1, you go through node 0.
```

In [ ]:
```
"Bellman Ford"
```

The 'bellman_ford()' method can also find the shortest path between sourse node to all other nodes of elements ,but this can handle negative weights as well.

In [52]:
```
# Find shortest path from element 1 to 2 with given graph with a negative weight:

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]))
```

In [ ]:
```
Distances Array:
[ 0. -1.  2.]
The distance from node 0 to itself is 0.
The shortest distance from node 0 to node 1 is -1, using the direct edge with a weight of -1.
The shortest distance from node 0 to node 2 is 2, using the direct edge with a weight of 2.

Predecessors Array:

[-9999     0     0]
-9999 for node 0 indicates it is the source node.
```

```
0 for node 1 means that node 1's predecessor in the shortest path is node 0.
0 for node 2 means that node 2's predecessor in the shortest path is also node 0.
```

In [ ]:
```
"Depth First Order"
```

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

This function takes follwing arguments:

1.the graph 2.the starting element to traverse graph from

In [54]:
```
# Traverse the graph depth first for given adjacency matrix:
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]))
```

In [ ]:
```
#output
print(depth_first_order(newarr, 1))

depth_first_order performs a depth-first traversal starting from the specified node (1 in this case).
The function returns two arrays:
Traversal Order: The order in which nodes are visited during the DFS.
Predecessors: For each node, the index of its predecessor in the depth-first tree. If a node has no predecessor

(array([1, 0, 3, 2], dtype=int32), array([-9999,  1,  1,  0], dtype=int32))
array([1, 0, 3, 2], dtype=int32): This is the order in which the nodes were visited during the DFS starting fro

Node 1 is visited first (starting node).
Node 0 is visited next.
Node 3 is visited after node 0.
Node 2 is the last to be visited.
array([-9999, 1, 1, 0], dtype=int32): This is the predecessors array.

-9999 for node 1 indicates it is the root node for this DFS.
1 for node 0 indicates that node 0 was first reached from node 1.
1 for node 2 indicates that node 2 was reached from node 1.
0 for node 3 indicates that node 3 was reached from node 0.
```

In [ ]:
```
"Breadth First Order"
```

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

This function takes following arguments:

1.the graph. 2.the starting element to traverse graph from.

In [56]:
```
# Traverse the graph breadth first for given adjacency matrix:

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]))
```

In [ ]:
```
(array([1, 0, 2, 3], dtype=int32), array([-9999,  1,  1,  1], dtype=int32))
array([1, 0, 2, 3], dtype=int32): This is the order in which the nodes were visited during the BFS starting fro

Node 1 is visited first (starting node).
```

```
Node 0 is visited next.
Node 2 is visited after node 0.
Node 3 is the last to be visited.
array([-9999, 1, 1, 1], dtype=int32): This is the predecessors array.

-9999 for node 1 indicates it is the root node for this BFS.
1 for nodes 0, 2, and 3 indicates that these nodes were reached directly from node 1.
```

In [ ]:

In [ ]:
```
"SciPy Spatial Data"
```

Spatail data refers to data that is represented in a geomatric space.

geometric space--> It is set of rules having direction,distance,sometimes curvature also.

E.g points on a coordinates 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.

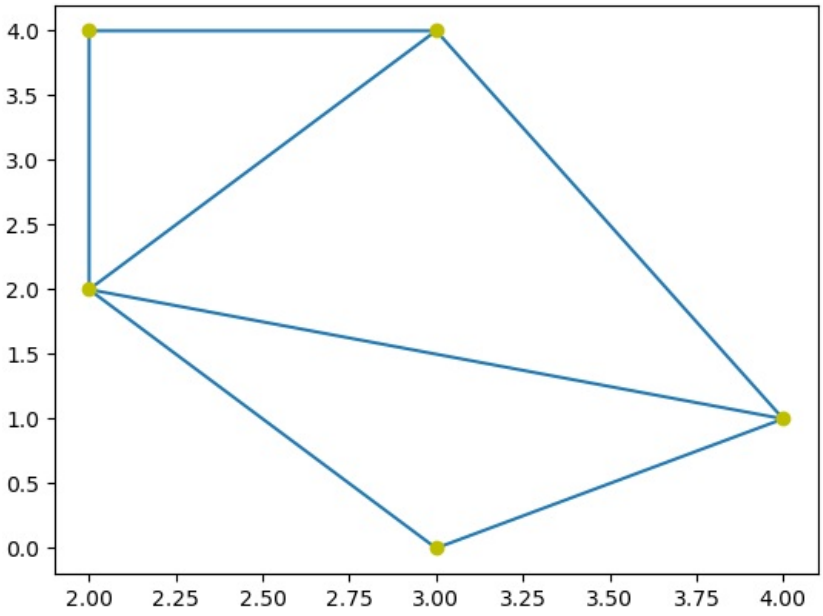
In [ ]:
```
"Trianglation"
```

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

A Triangulation with points means creating surface composed triangles in which all of the given points are on atleast one vertex of any triangle in the surface.

One method to generate these triangulations through points is the Delaunay() Triangulation.

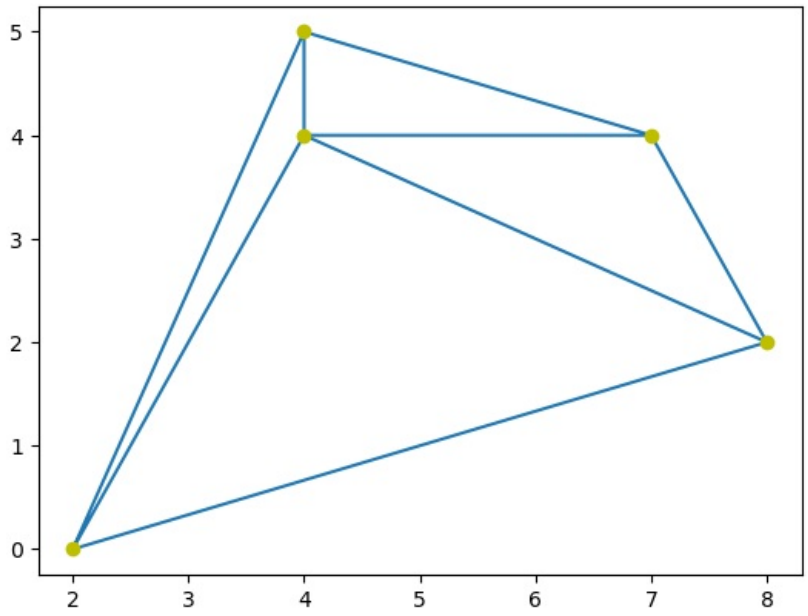
In [31]:
```
import numpy as np
from scipy.spatial import Delaunay
import matplotlib.pyplot as plt

points=np.array([
    [2,4],
    [3,4],
    [3,0],
    [2,2],
    [4,1]])#[x,y]
simplices=Delaunay(points).simplices
plt.triplot(points[:,0],points[:,1],simplices)# Plots the Delaunay triangulation as a series of triangles.
plt.scatter(points[:,0],points[:,1],color='y')#plt.scatter: Plots the original points as red dots on the graph.
plt.show()
```



In [43]:
```
points=np.array([
    [4,4],
    [7,4],
    [2,0],
    [8,2],
    [4,5]])
simplices=Delaunay(points).simplices
```

```
print(simplices)
plt.triplot(points[:,0],points[:,1],simplices)# Plots the Delaunay triangulation as a series of triangles.
plt.scatter(points[:,0],points[:,1],color='y')#plt.scatter: Plots the original points as red dots on the graph.
plt.show()
```

```
[[3 0 2]
 [0 4 2]
 [1 0 3]
 [0 1 4]]
```



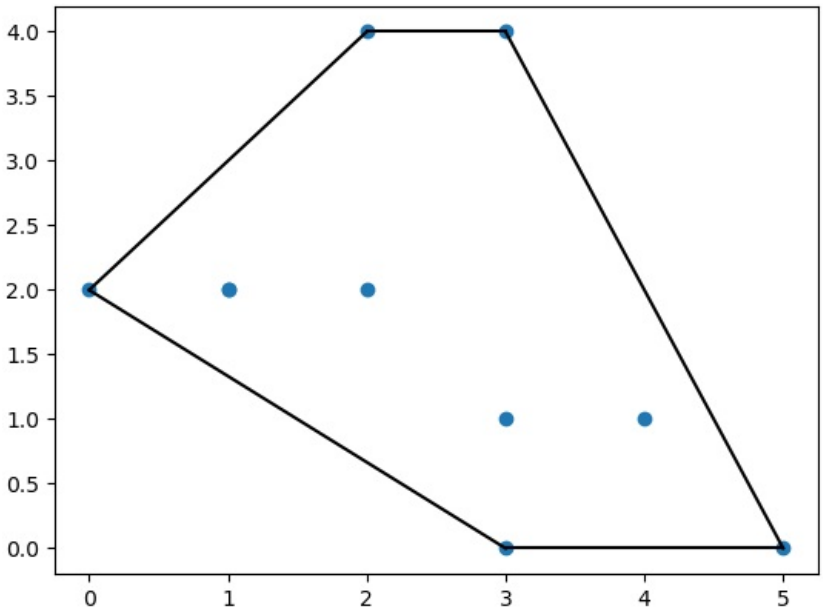
In [ ]:
```
[[2 4 3]
 [1 2 0]
 [3 1 0]
 [3 4 1]]
Explanation
[[2, 4, 3], [1, 2, 0], [3, 1, 0], [3, 4, 1]]:
Each sub-array represents a triangle in the Delaunay triangulation.
The numbers in each sub-array correspond to the indices of the input points that form the vertices of the triang
For instance, the first triangle [2, 4, 3] consists of the points at indices 2, 4, and 3 in the points array.
```

In [ ]:
```
Note:The simplices property creates a generalisation of the triangle notation.
```

In [ ]:
```
"Convex Hull"
```

In [ ]:
```
A ConvexHull is the smallest polygon that covers all of the given points.

Use the 'ConvexHull()' mathod to create a Convex Hull.
```

In [59]:
```
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-') #k stands for black line and - stands for solid line

plt.show()
# for simplex in hull_points: Iterates over each edge of the convex hull.
# points[simplex, 0] and points[simplex, 1]: Extracts the x and y coordinates of the endpoints of each edge.
# plt.plot(..., 'k-'): Plots each edge as a black line ('k' stands for black and '-' specifies a solid line).
```



In [ ]:
```
"kdtrees"
```

1.KDTrees are a datastructure optimised for nearest neighbour queries.

E.g.. in a set of points unsing KDTrees we can efficiantly ask which points are nearest to the certain point.

2.The KDTrees() method returns a KDTree object.

3.the qeury() method returns the distance to the nearest neighbour and the location of the neighbours

In [15]:
```
# Find the nearest neighbour to the point(1,1)

from scipy.spatial import KDTree
points=[(1,-1),(2,3),(-2,3),(2,-3)]
kdtree=KDTree (points)
print(kdtree)
res=kdtree.query((1,1))
print(res)
```

```
<scipy.spatial._kdtree.KDTree object at 0x000001294CAC2D50>
(2.0, 0)
```

2.o is the nearest neighbour and 0 is the index of (1,-1) which is nearest neighbour

2.0: This is the Euclidean distance between the query point (1, 1) and its nearest neighbor in the dataset. In this case, the nearest point is (1, -1). The distance is calculated as: distance=root((1-1))^2+(1-(-1))^2=root(0+4)=2.0

0: This is the index of the nearest neighbor in the original list of points. The index 0 corresponds to the point (1, -1) in the points list.

In [ ]:
```

```

In [ ]:
```
"Distance Matrix"
```

*there many distance matrics used to find various types of distances between two points in data science , 1.Euclidean Distance 2.cosine distance

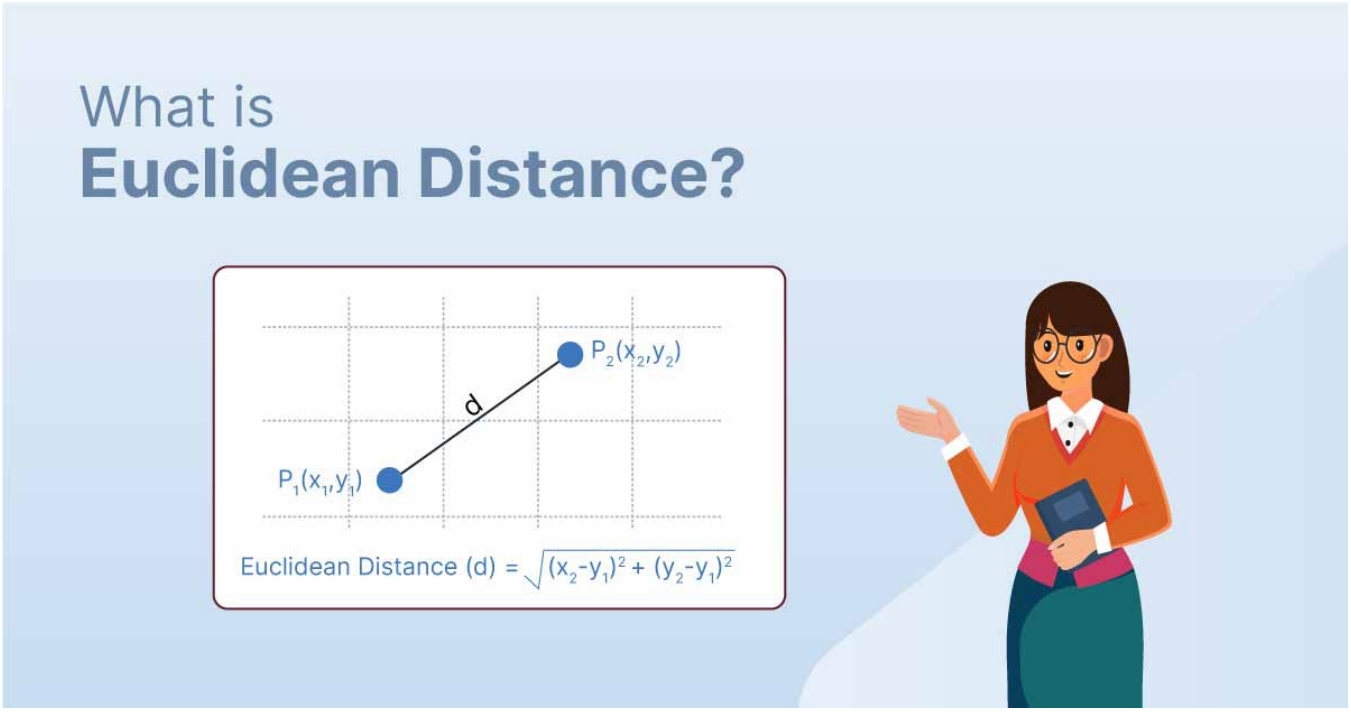
the distance between two vectors may not only be the length of straight line between them,it can also be the angle between them from origin or number of unit steps required etc.

Many of Machine Learning algorithm's performance depends greatly on distance metrices E.g "K Nearest Neighbours" or " K Means" etc.

In [ ]:
```
"1.Euclidean Distance"
```

In [26]:
```
from PIL import Image
Image.open('euclidean.jpg')
```

Out[26]:



In [ ]:
```
Find the euclidean distance between given points.
```

In [20]:
```
from scipy.spatial.distance import euclidean
p1=(1,0)
p2=(10,2)
res=euclidean(p1,p2)
print(res)
```

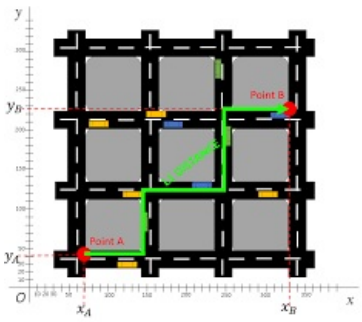
```
9.219544457292887
```

In [ ]:
```
"2.Cityblock Distance (Manhattan Distance)"
```

The Cityblock distance also known as Manhattan distance ortaxicab distnace,Is a measure of distance in a grid-like path,such as the layout of streets in acity where you can only travel along the grid lines.Unlike Euclidean distance,which is the shortest stright_line distance between two points,the Manhattan distance is the sum of the absolute differences between the coordinates of two points.

In [31]:
```
Image.open('manhattan.png')
```

Out[31]:



In [ ]:
```
Formula for Cityblock (Manhattan) Distance
Given two points P1=(x1,y1) and p2=(x2,y2) in a @d space, the Mahnattan distance between thease ponits is calcu

Manhattan Distance=|x2-x1|+|y2-y1|
In higher dimentions for points p1=(x1,x2,x3--xn) and p2=(y1,y2,y3..yn) the formula is

Mahanttan Distance=  i=1∑n |xi−yi|
```

It is the distnace computed using 4 degrees of movement.

E.g we can only move:up,down,right,left,not diagonnaly.

In [29]:
```
# Find the cityblock distance between given points.

from scipy.spatial.distance import cityblock
p1=(1,0)
p2=(10,3)
result=cityblock(p1,p2)
print(result)
```

```
12
```

In [ ]:
```
"Cosine Distance"
```

Cosine Distance is a measure used to determine how different two vectors are in a multidimetional space ,based onthe cosine of the angle between them.It is derived from the'cosine similarity',which measures the cosine of the angle between two non-zero vectors in an inner product space.

Cosine similarity is often used in high-dimentional spaces to measure how similar two vectors are regradless of their magnitude,focusing instead on their direction.This is particularly usefull in feild like text mining information retreival and recommender systems where you care about the orientaion of vectors rather than their size.

In [ ]:
```
Cosine Similarity Formula.

Given two vectors A=(a1,a2,a3,...an) and B=(b1,b2...bn) the cosine similarity is defined as:

Cosine Similarity=cos(0)=  A.B
                          ___________
                         ||A|| ||B||
where:
* A.B is dot product of vectors A and B.
*||A|| is the magnitude (length) of vector A.
*||B|| is the magnitude(length) of vector B.

Dot Product and Magnitudes
Dot Product:
         A.B= i=1∑n ai.bi
Magnitude of vector.
        ||A||=rooti=1∑n ai^2
        ||B||=root i=1∑n bi^2


Cosine Distance
Cosine distance is defined as:
           Cosine Distance=1-Cosine Similarity
```

In [ ]:
```

```

In [ ]:
```
Is the value of cosine angle between the points A and B.
```

In [37]:
```
# Find the cosine distance between given two points .
from scipy.spatial.distance import cosine

p1=(1,0)
p2=(10,2)
result=cosine(p1,p2)
print(result)
```

```
0.019419324309079777
```

In [50]:
```
Image.open('cosine.png')
here A=(1,0) and B=(10,2)
```

Out[50]:

1. Calculate the dot product:

$$\mathbf{A} \cdot \mathbf{B} = (1 \times 10) + (0 \times 2) = 10$$

2. Calculate the magnitudes:

$$\|\mathbf{A}\| = \sqrt{(1^2 + 0^2)} = \sqrt{1} = 1$$

$$\|\mathbf{B}\| = \sqrt{(10^2 + 2^2)} = \sqrt{100 + 4} = \sqrt{104} \approx 10.198$$

3. Calculate the cosine similarity:

$$\text{Cosine Similarity} = \frac{10}{1 \times 10.198} \approx 0.980$$

4. Calculate the cosine distance:

$$\text{Cosine Distance} = 1 - 0.980 = 0.020$$

In [ ]:
```
"Hamming Distance"
```

Is the proportion of bits where two bits are different. It's a way to measure distance for binary sequeneces.

It is used to measure the difference between two strings of equal length. It counts the number of positions at which the corresponding sysmbols(charcaters,bits etc.) are differet.

This wideli used in information theory ,coding thoery and computer scienece.

In [ ]:
```
Example of Hamming Distance:

Considered two strings:
1.String A:'1101'
2.string B:'1001'

To calculete the Hamming distance between these two strigs compare them position by position.

1. 1 vs 1 (same)
2. 1 vs 0 (different)
3. 0 vs 0 (same)
4. 1 vs 1 (same)
    Here, the Hamming distance is 1 because there is only one position where the strings differ (second positio
```

In [ ]:
```
Applications of Hamming Distance:
1.Error Detection and Correction:
2.Genetics: In bioinformatics, it measures the genetic distance between DNA sequences.
3.Cryptography: It's used to determine how different two cryptographic keys or ciphertexts are.
4.Machine Learning: It can be used as a similarity measure between binary vectors.
```

In [2]:
```
from scipy.spatial.distance import hamming
p1=(True,False,True)
p2=(False,True,True)
result=hamming(p1,p2)
print(result)
```

```
0.6666666666666666
```

In [12]:
```
from scipy.spatial.distance import hamming
p1=('vasu','srinu',True)
p2=('lucky',True,True)
result=hamming(p1,p2)
print(result)
```

```
0.6666666666666666
```

In [ ]:
```
"Spatial Matlab Arrays"
```

We know that NumPy provides us with methods to persist(store) the data in readble 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.

In [ ]:
```
"Exporting Data in Matlab format."
```

The 'savemat()' function allows us to export data in Matlab format.

The method takes the following parameters:

1.filenanme-->the file name for saving the data.

2.mdict--> a dictionary conataining the data.

3.do_compression-> a boolean value that specifies whether to compress the result or not.Default False.

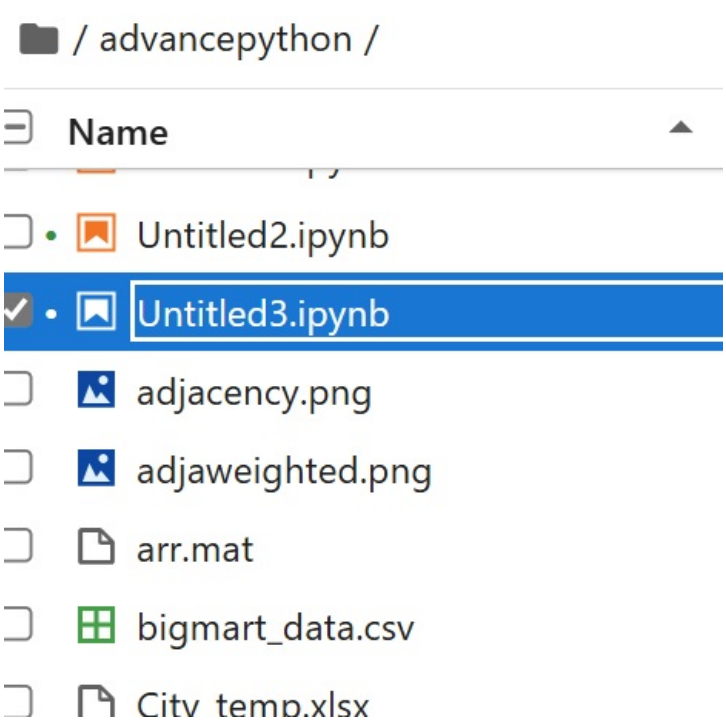
In [16]:
```
# Export the following array as variable name "vec" to a mat file:
from scipy import io
import numpy as np

arr=np.arange(10)
io.savemat('arr.mat',{"vec":arr}) # arr.mat--> file name and {'vec':arr} --> dict data.
```

Note: The example above saves a file name "arr.mat" on your computer.

To open the file, check out the "Import Data from Matlab Format" example below:

In [18]:
```
from PIL import Image
Image.open('mat.png')
```

Out[18]:



In [ ]:
```
"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 structured array whose keys are the variable names and the corresponding values are the variable values.

In [52]:
```
from scipy import io
import numpy as np

mydata=io.loadmat('arr.mat') #previously we created one matfile
mydata
```

Out[52]:
```
{'__header__': b'MATLAB 5.0 MAT-file Platform: nt, Created on: Sat Aug 31 17:09:52 2024',
 '__version__': '1.0',
 '__globals__': [],
 'srinu': array(['array  ', 'vasu   ', 'srinu  ', 'lakshmi', 'bhavya '], dtype='<U7')}
```

In [29]:
```
a=np.array(["array","vasu","srinu","lakshmi","bhavya"])
io.savemat('arr.mat',{"srinu":a})
result=io.loadmat('arr.mat')
result
```

Out[29]:
```
{'__header__': b'MATLAB 5.0 MAT-file Platform: nt, Created on: Sat Aug 31 17:09:52 2024',
 '__version__': '1.0',
 '__globals__': [],
 'srinu': array(['array  ', 'vasu   ', 'srinu  ', 'lakshmi', 'bhavya '], dtype='<U7')}
```

Use the variable name "vec" to display only the array from the matlab data:

In [33]:
```
print(mydata['vec'])
print(result['srinu'])
```

```
[[0 1 2 3 4 5 6 7 8 9]]
['array  ' 'vasu   ' 'srinu  ' 'lakshmi' 'bhavya ']
```

Note: We can see that the array originally was 1D, but on extraction it has increased one dimension.

In order to resolve this we can pass an additional argument squeeze_me=True:

In [56]:
```
mydata=io.loadmat('arr.mat',squeeze_me=True)
print(mydata['srinu'])
```

```
['array  ' 'vasu   ' 'srinu  ' 'lakshmi' 'bhavya ']
```

In [ ]:
```
"SciPy Interpolation"
```

In [ ]:
```
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 fiiling values is called imputation.

Apart from imputation,interpolation is often used where we need to smooth 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:

In [ ]:
```
"1D Interpolation"
```

The function 'interp1d()' is used to interpolate a distribution with 1 variable .

It takes x and y points and return a collable function that can be called with new x and returns corresponding y.

In [64]:
```
# For given xs and ys interpolate values from 2.1, 2.2... to 2.9:
from scipy.interpolate import interp1d
import numpy as np

xs=np.arange(10)#0 to 9.
ys=2*xs+1#(eg.2*1+1=3)[1, 3, 5, 7, 9, 11, 13, 15, 17, 19].

interp_func=interp1d(xs,ys)
# This creates a function interp_func that can interpolate new y-values
# for given x-values based on the data points (xs, ys).
newarr=interp_func(np.arange(2.1,3,0.1))
 # new array of x-values: [2.1, 2.2, 2.3, ..., 2.9].
newarr
```

Out[64]:
```
array([5.2, 5.4, 5.6, 5.8, 6. , 6.2, 6.4, 6.6, 6.8])
```

In [100…
```
# output
from PIL import Image
Image.open('inter1 (1).png')
```

Out[100…

1. **Original Data**:
   - `xs` (x-values): `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
   - `ys` (y-values): Calculated as `2*xs + 1`, resulting in `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]`.

   These represent a linear relationship: `y = 2x + 1`.
2. **Interpolation Function**:
   - The interpolation function `interp_func` was created using `scipy.interpolate.interp1d` with `xs` and `ys` as inputs. This function allows us to estimate y-values for any x-values based on the original linear relationship.

In [102…
```
Image.open('inter1 (2).png')
```
Out[102…

3. **Generating New X-Values**:
   - `np.arange(2.1, 3, 0.1)` produces a new array of x-values: `[2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9]`.
4. **Interpolation Calculation**:
   - The line `newarr = interp_func(np.arange(2.1, 3, 0.1))` calculates the interpolated y-values for these new x-values.

Explanation of Each Interpolated Y-Value

Given the formula $y = 2x + 1$, the interpolation estimates the y-values for each x-value in the range from `2.1` to `2.9`. Let's compute each value:

- For x = 2.1:

$$y = 2 \times 2.1 + 1 = 4.2 + 1 = 5.2$$

In [104…
```
from PIL import Image
Image.open('inter1 (3).png')
```

Out[104…

- For x = 2.2:

$$y = 2 \times 2.2 + 1 = 4.4 + 1 = 5.4$$

- For x = 2.3:

$$y = 2 \times 2.3 + 1 = 4.6 + 1 = 5.6$$

- For x = 2.4:

$$y = 2 \times 2.4 + 1 = 4.8 + 1 = 5.8$$

- For x = 2.5:

$$y = 2 \times 2.5 + 1 = 5.0 + 1 = 6.0$$

- For x = 2.6:

$$y = 2 \times 2.6 + 1 = 5.2 + 1 = 6.2$$

- For x = 2.7:

$$y = 2 \times 2.7 + 1 = 5.4 + 1 = 6.4$$

In [74]:
```
xs=np.arange(9)
ys=2*xs+1
interp_func=interp1d(xs,ys) #new interpolated y-values
new=interp_func(xs)
new
```

Out[74]:
```
array([ 1.,  3.,  5.,  7.,  9., 11., 13., 15., 17.])
```

Note: that new xs should be in same range as of the old xs, meaning that we can't call interp_func() with values higher than 10, or less than 0.

In [ ]:
```
"Spline Interpolation"--> Is a type polynomial function
```

In 1D interpolation the points are fitted for a single curve wheareas in Spline interpolation the points are fitted against a piecewise of function defined with polynomials clled splines.

The UnivariateSpline() function takes xs and ys and produce a callable function that can be called with new xs.

Piecewise function: A function that has different defination for different ranges.

In [107…
```
# Find univariate spline interpolation for 2.1, 2.2... 2.9 for the following non linear points:

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]
```

In [ ]:
```
"Interpolation with Radial Basis Function"
```

Radial basis fucnction 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 [110…
```
# Interpolate following xs and ys using rbf and find values for 2.1, 2.2 ... 2.9:
```

```
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]
```

In [ ]:
```
"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:

In [ ]:
```
1.Hypothesis in Statistics
```

Hypothesis is an assumption about a parameter in population

In [ ]:
```
2.Null Hypothesis
```

It assumes that the obseravation is not statistically significant

In [ ]:
```
3.Alternate Hypothesis
```

It assumes that the observations are due to some reason.

It's alternate to Null Hypothesis.

In [ ]:
```
Example:

For an assessment of a student we would take:

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

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

In [ ]:
```
"One tailed test"
```

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

Example:

1. "the mean is equal to k,we can have alternative hypothesis".

2."the mean is less than k",or: "the mean is greater than k"

In [ ]:
```
"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 alternative hypothesis:

"the mean is not equal to k"

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

In [ ]:
```
"Alpha Value"
```

Alpha value is level of significance.

Example:

How close to extreams the data must be for null hypothesis to be rejected .

It is useally taken as 0.01,0.05, or 0.1.

In [ ]:
```
"P value"
```

P value tells us how close to extreme the data actually is.

P value and alpha values are compared to established the statical significance.

```
If P value<=alpha we reject null hypothesis and say that the data is statically
significant.otherwisw we accept the  null hypothesis.
```

In [ ]:
```
"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 two tailed test.

the function 'ttest_int()' takes two samples of same size and produces a tuple of t-static and p-value.

In [13]:
```
import numpy as np
from scipy.stats import ttest_ind
v1=np.random.normal(size=100)
v2=np.random.normal(size=100)
result=ttest_ind(v1,v2)
print(result)
```

```
TtestResult(statistic=1.6470822567652497, pvalue=0.10112727626716299, df=198.0)
```

In [ ]:
```
If you want only the p-value use the pvalue property
```

In [15]:
```
result=ttest_ind(v1,v2).pvalue
print(result)
```

```
0.10112727626716299
```

In [ ]:
```
"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.

In [ ]:
```
CDF:-->A CDF can be either a string or a collable 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 parameters alternative as a string of one of two-sided ,less or greater.

In [27]:
```
import numpy as np
from scipy.stats import kstest
v=np.random.normal(size=100)
result=kstest(v,'norm')
print(result)
```

```
KstestResult(statistic=0.08794564652099934, pvalue=0.3986911611525127, statistic_location=0.8343057598132788, st
atistic_sign=-1)
```

In [ ]:
```
"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:

1. number of observations

2.minimum and maximum values=minmax

3.mean

4.varience

5.skewness

6.kurtosis

In [34]:
```
import numpy as mp
from scipy.stats import describe
```

```
v=np.random.normal(size=100)
result=describe(v)
result
```

Out[34]:
```
DescribeResult(nobs=100, minmax=(-1.91137892299633, 2.043792164347562), mean=0.03232947360253127, variance=0.86
69724466600112, skewness=0.11370248983562124, kurtosis=-0.5912089122235802)
```

In [ ]:
```
"Normality tests(skewness and kurtosis)"
```

Normality tests are based on these skwness and kurtosis.

the 'normaltest()' function returns p value for the null hypothesis:

"x comes from a normal distribution"

In [ ]:
```
"Skewness"
```

A measure of symmetry in Data.

For normal ditribution it is 0.

If it is negative ,it means the data is skewed left.

If it positive it means the data is skewed right.

In [ ]:
```
"Kurtosis"
```

A measure of weather the data is heavy or lightly tailed to a normal distribution.

Positive kurtosis means heavy tailed.

Negative kurtosis means lightly tailed.

In [44]:
```
# Find the Skewness and kurtosis of values in an array.
import numpy as np
from scipy.stats import skew,kurtosis

v=np.random.normal(size=100)
print(skew(v))
print(kurtosis(v))
```

```
0.15619564189913282
-0.2868576086362653
```

In [48]:
```
# Find if the data comes from a normal distribution:
from scipy.stats import normaltest

v=np.random.normal(size=100)
print(normaltest(v))
```

```
NormaltestResult(statistic=1.5813916775900996, pvalue=0.45352910231531174)
```

In [54]:
```
from scipy import constants
constants.liter
```

Out[54]:
```
0.001
```

In [56]:
```
dir(constants)
```

Out[56]:
```
['Avogadro',
 'Boltzmann',
 'Btu',
 'Btu_IT',
 'Btu_th',
 'ConstantWarning',
 'G',
 'Julian_year',
 'N_A',
 'Planck',
 'R',
 'Rydberg',
 'Stefan_Boltzmann',
 'Wien',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
```

```
'__package__',
'__path__',
'__spec__',
'_codata',
'_constants',
'_obsolete_constants',
'acre',
'alpha',
'angstrom',
'arcmin',
'arcminute',
'arcsec',
'arcsecond',
'astronomical_unit',
'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_Fahrenheit',
'deka',
'dyn',
'dyne',
'e',
'eV',
'electron_mass',
'electron_volt',
'elementary_charge',
'epsilon_0',
'erg',
'exa',
'exbi',
'femto',
'fermi',
'find',
'fine_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',
'hectare',
'hecto',
'horsepower',
'hour',
'hp',
'inch',
'k',
'kgf',
'kibi',
'kilo',
'kilogram_force',
'kmh',
'knot',
```

```
'lambda2nu',
'lb',
'lbf',
'light_year',
'liter',
'litre',
'long_ton',
'm_e',
'm_n',
'm_p',
'm_u',
'mach',
'mebi',
'mega',
'metric_ton',
'micro',
'micron',
'mil',
'mile',
'milli',
'minute',
'mmHg',
'mph',
'mu_0',
'nano',
'nautical_mile',
'neutron_mass',
'nu2lambda',
'ounce',
'oz',
'parsec',
'pebi',
'peta',
'physical_constants',
'pi',
'pico',
'point',
'pound',
'pound_force',
'precision',
'proton_mass',
'psi',
'pt',
'quecto',
'quetta',
'ronna',
'ronto',
'short_ton',
'sigma',
'slinch',
'slug',
'speed_of_light',
'speed_of_sound',
'stone',
'survey_foot',
'survey_mile',
'tebi',
'tera',
'test',
'ton_TNT',
'torr',
'troy_ounce',
'troy_pound',
'u',
'unit',
'value',
'week',
'yard',
'year',
'yobi',
'yocto',
'yotta',
'zebi',
'zepto',
'zero_Celsius',
'zetta']
```

In [ ]:

In [ ]:

In [ ]: