# SciPy in Python: How to use SciPy for Scientific Computations [WITH EXAMPLES]

SciPy is a popular library built on top of Numpy extensions with a mixture of functions and mathematical algorithms, that solves scientific, mathematical, and engineering problems. It uses a high-level Python command that visualizes and manipulates data. One important use of SciPy is that it transforms Python sessions into an environment that is efficient for data processing and system prototyping.

SciPy has a myriad of functions that can be called to do different mathematical and scientific computations. It cannot be used without the knowledge of NumPy because it operates on dimensional arrays of the NumPy library. SciPy is preferred to NumPy not only because it is an extension of NumPy, but also because most recent data science features are present in SciPy rather than NumPy.

Let’s begin by listing out some of these functions or sub-packages. They include:

1. **constants**: This is used for physical and mathematical constants.
2. **integrate**: This is used to solve Ordinary differential equations (ODE) and Integration problems.
3. **cluster**: This is used to cluster algorithms.
4. **fftpack**: This is used for fast Fourier transform routines.
5. **ndimage**: This is used for N-dimensional image processing.
6. **io**: This is used for input and output operations.
7. **odr**: This is used for orthogonal distance regression.
8. **linalg**: This is used for linear regression analysis.
9. **spatial**: This is used for spatial data structures and algorithms.
10. **sparse**: This is used for sparse matrices and associated routines.
11. **interpolate**: This is used for interpolating and smoothing splines.
12. **stats**: This is used for statistical distributions and functions.
13. **special**: This is used for special functions.

By the end of this tutorial, you will learn how to use scipy to do the most common mathematical and scientific computations using some of the above functions/sub-packages. We begin by installing scipy into your machine.

## Installing SciPy on your Machine

SciPy can be installed in Windows, Linux, Mac.

For Windows, we use pip.

Python3 -m pip install --user numpy scipy

For Linux,

sudo apt-get install python-scipy python-numpy

For Mac,

sudo port install py35-scipy py35-numpy

NB: If you have Jupyter Notebook installed on your PC (and by extension Anaconda), you would not need to perform the above steps as scipy comes pre-installed alongside Anaconda.

Next, we will discuss some of the features of SciPy used for scientific computation.

## File Input/Output package

This package has a variety of functions working with different file formats. Some of these file formats are Wave, Matrix market, TXT, binary format, CSV, and Matlab. One of the regularly used file formats is Matlab and we shall look at an example provided below.

#import necessary libraries

import numpy as np

from scipy import io as sio

#Then, we create a 3x3 dimensional one's array

array = np.ones((3,3))

#store the array in a file using the savemat method

sio.savemat('sample.mat',{'array': array})

#get the data from the stored file using the loadmat method

data = sio.loadmat('sample.mat', struct\_as\_record=True)

#print the output

data['array']

OUTPUT

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

[1., 1., 1.],

[1., 1., 1.]])

## Special Function Package

As earlier said, scipy.special is that package that contains various functions in mathematics. These functions are exponential, cubic root, lambert, kelvin, parabolic cylinder, Bessel, gamma, permutation and combinations, log sum amongst others. For a further description in the Python console; you can use the following command

#First, we import the scipy function

import scipy as scipy

help(scipy.special)

OUTPUT

Help on package scipy.special in scipy:

NAME

scipy.special

DESCRIPTION

========================================

Special functions (:mod:`scipy.special`)

========================================

.. module:: scipy.special

Nearly all of the functions below are universal functions and follow broadcasting and automatic array-looping rules. Exceptions are noted.

.. seealso::

`scipy.special.cython\_special` -- Typed Cython versions of special functions

## Cubic Root Function

This function finds the cube root of numbers. The syntax is written on the Python console as scipy.special.cbrt(x). For example,

#First, we import the scipy cubic root function

from scipy.special import cbrt

#Then, we say for example that we want to find the cube root of 125, 27, and 100 using the cbrt() function

cuberoot = cbrt([125, 27, 1000])

#Then, we print the result

print('The cuberoot of 125, 27 and 1000 are: ', end='')

print(cuberoot, end=' ')

print('respectively')

OUTPUT

The cuberoot of 125, 27 and 1000 are: [ 5. 3. 10.] respectively

## Exponential Function

This function computes the 10\*\*x of an element. For example

from scipy.special import exp10

#Then, we find the exp10 function and then give it a value

exponent= exp10(7)

#Then, we print the result

print('The 10th exponent of 7 is: ', end='')

print(int(exponent))

OUTPUT

The 10th exponent of 7 is: 10000000

## Permutations and Combinations

For combination, the function is scipy.special.comb(N,k). For example,

#First, we import the SciPy library with the combination package

from scipy.special import comb

#Them, we find the combinations of 6 and 3 using comb(N,k)

combination\_result = comb(6,3, exact = False, repetition = True)

#Then, we print the output

print('6C3 = ', end='')

print (combination\_result)

OUTPUT

6C3 = 56.0

For Permutation

#First, we import the SciPy library with the permutation package

from scipy.special import perm

#Them, we find the permutation of 6 and 3 using comb(N,k)

permutation\_result = perm(6,3, exact = True)

#Then, we print the output

print('6P3 = ', end='')

print(permutation\_result)

OUTPUT

6P3 = 120

## Linear Algebra with SciPy

It implements both the BLAS and ATLAS LAPACK libraries but it performs faster than both of them. For example, to calculate the determinant of a 2-dimensional matrix.

#First, we import the SciPy library with linear algebra package

from scipy import linalg

#Also, we import numpy library

import numpy as np

#Then, we define the 2x2 matrix

square\_matrix = np.array([ [4,6], [4,8]])

#Then, we pass the values to the det() function to find the determinant of the matrix

linalg.det(square\_matrix)

OUTPUT

7.999999999999998

Also, for the Inverse matrix; the syntax is scipy.linalg.inv(). For example

#First, we import the SciPy library with linear algebra package

from scipy import linalg

#Also, we import numpy library

import numpy as np

#Then, we define the 2x2 matrix

square\_matrix = np.array([ [4,6], [4,8]])

#Then, we pass the values to the inv() function to find the inverse of the matrix

linalg.inv(square\_matrix)

OUTPUT

array([[ 1. , -0.75],

[-0.5 , 0.5 ]])

For Eigenvalues and Eigenvectors which is the most common problem to solve using linear algebra. The syntax is scipy.linalg.eig(). For example,

#import the SciPy library with linear algebra package

from scipy import linalg

#Also, we import numpy library

import numpy as np

# define the 2 by 2 matrix

array = np.array([ [4,6], [4,8]])

#Then, we pass the values to the function to find the eigenvectors and eigenvalues of the matrix

eigen\_value, eigen\_vector = linalg.eig(array)

#Then, we get the eigenvalues

print('The eigenvalues are: ')

print (eigen\_value)

print()

#Also, we get the eigenvectors

print('The eigenvectors are: ')

print (eigen\_vector)

OUTPUT

The eigenvalues are:

[ 0.70849738+0.j 11.29150262+0.j]

The eigenvectors are:

[[-0.8767397 -0.6354064 ]

[ 0.48096517 -0.7721779 ]]

## Discrete Fourier Transform DFT

This is a technique used in mathematics to convert spatial data to frequency data. The algorithm used for computing DFT is called the Fast Fourier Transform FFT. FFT is used for multidimensional arrays. For example, we use a wave with a periodic function.

#Import the necessary libraries including matplotlib

%matplotlib inline

from matplotlib import pyplot as plt

import numpy as np

#Then, we assume a Frequency in terms of Hertz

frequency =10

#Also, we assume a Sample rate

frequency\_sample = 50

time = np.linspace(0, 10, 120, endpoint = False )

amplitude = np.sin(frequency\_sample \* 2 \* np.pi \* time)

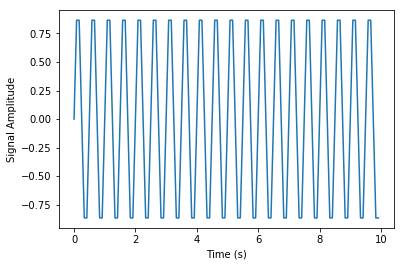
plt.plot(time, amplitude)

plt.xlabel ('Time (s)')

plt.ylabel ('Signal Amplitude')

plt.show()

OUTPUT



From this, the frequency is 10 Hz and the period is 1/10 seconds. With this wave, we can apply DFT.

from scipy import fftpack

A = fftpack.fft(amplitude)

frequency = fftpack.fftfreq(len(amplitude)) \* frequency\_sample

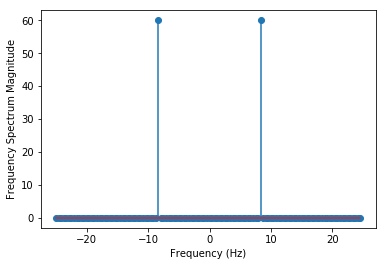
plt.stem(frequency, np.abs(A))

plt.xlabel('Frequency (Hz)')

plt.ylabel('Frequency Spectrum Magnitude')

plt.show()

OUTPUT



From the output above, the result is a one-dimensional array, and also the input where there are complex values is zero except for two points. We use the DFT to visualize the magnitude of the signal.

## Optimization and fit in SciPy

The syntax is scipy.optimize. This provides a meaningful algorithm that is used to minimize curve fittings, multidimensional or scalar, and roof fitting. For example, we look at Scalar function,

#import necessary libraries

import matplotlib.pyplot as plt

from scipy import optimize

import numpy as np

#define a scalar function to be minimized

def function(a):

return a\*5 + 2 \* np.cos(a)

plt.plot(a, function(a))

plt.show()

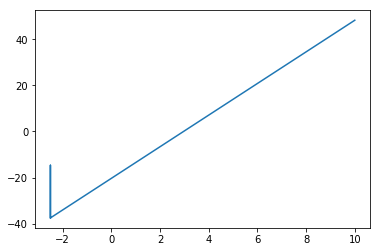
print('The optimized function is: ')

print('------------------------------------')

#use BFGS algorithm for optimization

optimize.fmin\_bfgs(function, 0)

OUTPUT



The optimized function is:

------------------------------------

Optimization terminated successfully.

Current function value: -581537857.204429

Iterations: 2

Function evaluations: 51

Gradient evaluations: 17

array([-1.16307572e+08])

The optimization is done with the example above with the help of the gradient descent algorithm taken from the initial point.

## Nelder-Mead Algorithm

It provides the most direct form of minimization for a fair behaved function. It is not used for gradient evaluations because it takes longer. For example,

#First, we import the necessary libraries

import numpy as np

from scipy.optimize import minimize

#Then, we define the scalar function

def scaler\_function(x):

return 2\*(1 + x[0])\*\*3

#minimize the scalar function

optimize.minimize(scaler\_function, [1, -1], method="Nelder-Mead")

OUTPUT

final\_simplex: (array([[-5.88348348e+43, -2.93385718e+43],

[-3.48931950e+43, -1.73998365e+43],

[-2.06941187e+43, -1.03193268e+43]]), array([-4.07318008e+131, -8.49673760e+130, -1.77243698e+130]))

fun: -4.073180080259595e+131

message: 'Maximum number of function evaluations has been exceeded.'

nfev: 401

nit: 204

status: 1

success: False

x: array([-5.88348348e+43, -2.93385718e+43])

## Image Processing with SciPy

The syntax is scipy.ndimage. The n in ndimage stands for n dimensional image. It is a submodule of SciPy used mainly in image associated operations. These image processing can provide geometric transformation, image segmentation, features extraction and classification. A package called MISC Package contains prebuilt images used in manipulating images. For example,

#import required libraries including the misc package

from scipy import misc

from matplotlib import pyplot as plt

import numpy as np

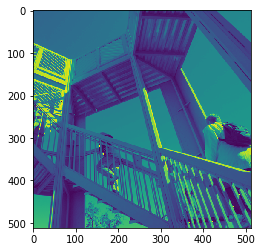
#Then, we get ascent image of panda from misc package

panda = misc.ascent()

#Then, we show the image of face

plt.imshow(panda);

OUTPUT



To flip down the image, we input to the Python console,

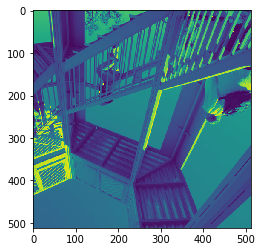
#To flip Down using scipy misc.face image

flip\_down = np.flipud(misc.ascent())

plt.imshow(flip\_down)

plt.show()

OUTPUT



To rotate images using SciPy,,

#import the necessary libraries

from scipy import ndimage, misc

from matplotlib import pyplot as plt

image = misc.ascent()

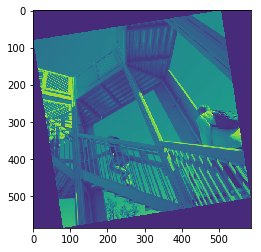
#rotation function of scipy for image – image rotated 9 degree

rotate\_image = ndimage.rotate(image, 9)

plt.imshow(rotate\_image)

plt.show()

OUTPUT



## Integration with SciPy- Numerical Integration

The syntax is scipy.integrate which consists of single integration, double, multiple, Romberg quadrate, Simpson’s, and Trapezoidal rule. Let's say we wish to integrate

#To perform single integration with scipy. integrate function

from scipy import integrate

# define the function f(x)

function = lambda x : x\*\*3

#single integration with a = 1 & b = 2

result = integrate.quad(function, 5 , 1)

print('The result is: ', end='')

print(result)

OUTPUT

The result is: (-156.0, 1.7319479184152442e-12)

The first value is the integration while the second value is the estimated error is integral since it is a numerical computation. As seen, the error value is substantially low.

CONCLUSION

In this tutorial, we have introduced all that there is to know under the concept of SciPy including the various sub-features of SciPy and how to apply all of them. If you have any questions, feel free to leave them in the comment section and I’d do my best to answer them.