Python NumPy, Scipy, Sympy

Panchatcharam M

NumPy

- NumPy is the core library for scientific computing in python
- High-performance multi-dimensional array object and tools for working with these arrays
- Familiar with MATLAB??

- A NumPy array is a grid of values, all of the same type
- indexed by a tuple of nonnegative integers
- number of dimensions is the rank of the array
- the *shape* of an array is a tuple of integers giving the size of the array along each dimension.

```python
import numpy as np

a = np.array([1, 2, 3])    # Create a rank 1 array
print(type(a))             # Prints "<class 'numpy.ndarray'>"
print(a.shape)             # Prints "(3,)"
print(a[0], a[1], a[2])    # Prints "1 2 3"
a[0] = 5                   # Change an element of the array
print(a)                   # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b.shape)                     # Prints "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0])   # Prints "1 2 4
```

```
import numpy as np

a = np.zeros((2,2))   # Create an array of all zeros
print(a)              # Prints "[[ 0.  0.]
                      #          [ 0.  0.]]"
b = np.ones((1,2))    # Create an array of all ones
print(b)              # Prints "[[ 1.  1.]]"
c = np.full((2,2), 7)  # Create a constant array
print(c)               # Prints "[[ 7.  7.]
                       #          [ 7.  7.]]"
d = np.eye(2)         # Create a 2x2 identity matrix
print(d)              # Prints "[[ 1.  0.]
                      #          [ 0.  1.]]"
e = np.random.random((2,2))  # Create an array filled with random values
print(e)                     # Might print "[[ 0.91940167  0.08143941]
                             #               [ 0.68744134  0.87236687]]"
```

Slicing

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print(a[0, 1])   # Prints "2"
b[0, 0] = 77     # b[0, 0] is the same piece of data as a[0, 1]
print(a[0, 1])   # Prints "77
```

```
# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

```python
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))  # Prints "[2 2]"
```

```
import numpy as np
# Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])

print(a)  # prints "array([[ 1,  2,  3],
          #                [ 4,  5,  6],
          #                [ 7,  8,  9],
          #                [10, 11, 12]])"
# Create an array of indices
b = np.array([0, 2, 0, 1])
# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  6  7 11]"
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10

print(a)  # prints "array([[11,  2,  3],
          #                [ 4,  5, 16],
          #                [17,  8,  9],
          #                [10, 21, 12]])
```

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
                     # this returns a numpy array of Booleans of the same
                     # shape as a, where each slot of bool_idx tells
                     # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                     #          [ True  True]
                     #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

```
import numpy as np

x = np.array([1, 2])    # Let numpy choose the datatype
print(x.dtype)          # Prints "int64"

x = np.array([1.0, 2.0])    # Let numpy choose the datatype
print(x.dtype)              # Prints "float64"

x = np.array([1, 2], dtype=np.int64)    # Force a particular datatype
print(x.dtype)
```

```python
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))
```

```
# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])
# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

```python
import numpy as np

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

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
import numpy as np

x = np.array([[1,2], [3,4]])
print(x)    # Prints "[[1 2]
            #          [3 4]]"
print(x.T)  # Prints "[[1 3]
            #          [2 4]]"

# Note that taking the transpose of a rank 1 array does nothing:
v = np.array([1,2,3])
print(v)    # Prints "[1 2 3]"
print(v.T)  # Prints "[1 2 3]"
```

SciPy

- NumPy/SciPy – numerical and scientific function libraries.
- Collection of mathematical algorithms
- NumPy extension
- Interactive python session by providing the user with high-level commands
- Manipulating and visualizing data
- Data processing
- System prototyping like MATLAB, SciLab

Methods for Integrating Functions given a function object:

Method	Explanation
`quad`	General purpose integration
`dblquad`	General purpose double integration
`tplquad`	General purpose triple integration
`fixed_quad`	Integrate function f(x) using Gaussian quadrature of order n
`quadrature`	Integrate with given tolerance using Gaussian quadrature
`romberg`	Integrate function using Romberg integration

Methods for Integrating Functions given a fixed samples:

Method	Explanation
`trapz`	Use trapezoidal rule to compute integral from samples
`cumtrapz`	Use trapezoidal rule to cumulatively compute integral
`simps`	Use Simpson's rule to compute integral from samples
`romb`	Use Romberg Integration to compute integral from (2**k + 1) evenly-spaced samples

Simple Integral example

$$\int_a^b \sin x \, dx$$

scipy.integrate!

Methods for Integrating Functions given a fixed samples:

```
#Integration
import numpy as np
import scipy.integrate
print(scipy.integrate.quad(np.sin,0,np.pi))
print(scipy.integrate.quad(np.sin,-np.inf,np.inf))
#Integration Sampling
x=np.linspace(0,np.pi,10000000)
y=np.sin(x)
print(scipy.integrate.trapezoid(x,y))
```

https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html

```
from scipy import constants
print(constants.liter)
for i,j in constants.physical_constants.items():
    print(i,j)
```

Linear Algebra

```
import numpy as np
from scipy import linalg
A = np.array([[1, 2], [3, 4]])
b = np.array([[5], [6]])
print("A: ",A)
print("b: ",b)
print("Inverse: of ",linalg.inv(A))
print("Solution Ax=b",np.linalg.solve(A,b))
print("Determinant of A: ", linalg.det(A))
print("Column Sum norm: ",linalg.norm(A,1))
print("Row sum norm: ",linalg.norm(A,np.inf))
```

Linear Algebra

```
#1D Interpolation
from scipy.interpolate import interp1d
import numpy as np
x = np.linspace(0, 10, num=21, endpoint=True)
y = np.exp(-x**2/9.0)
f = interp1d(x, y)
f2 = interp1d(x, y, kind='cubic')
xnew = np.linspace(0, 10, num=51, endpoint=True)
import matplotlib.pyplot as plt
plt.plot(x, y, 'o', xnew, f(xnew), '-', xnew, f2(xnew), '--')
plt.legend(['data', 'linear', 'cubic'], loc='best')
plt.show()
```

Submodules	Description
scipy.special	Special Functions
scipy.fft	Fast Fourier Transforms
scipy.signal	Signal Processing
scipy.csgraph	Compressed Sparse Graph
scipy.spatial	Spatial Data structures, Delaunay, Simplices
scipy.stats	Statistics
scipy.ndimage	Multidimensional Image Processing
scipy.io	Input Output
scipy.sparse.linalg	Eigenvalues for large sparse matrices

SymPy

Algebraic Expression

```
import sympy as sp
x=sp.symbols('x')
expr=(x+1)**2
print(sp.expand(expr))
expr1=(x+1)(x+3)
print(sp.expand(expr1))
expr2=x**3+3*x**2+3*x+1
print(sp.factor(expr2))
```

Differentiate

```
import sympy as sp
from sympy import *
print(limit(sin(x)/x,x,0))
print(limit(x*sp.exp(-x),x,oo))
print(limit((1+1/x)**x,x,oo))
```

Differentiate

```
import sympy as sp
x=sp.symbols('x')
expr=x**2+5*x+4
print(sp.diff(expr,x))
expr=sp.sin(x)
print(sp.diff(expr1,x))
expr2=sp.exp(x**2)
print(sp.diff(expr2,x))
expr3=sp.sin(x)+sp.cos(x)*sp.exp(x)
print(sp.diff(expr3,x))
```

$$P(x) = x^2 + 5x + 4 \Rightarrow P'(\mathrm{x}) = 2\mathrm{x} + 5$$

$$f(x) = \sin(x), f'(x) = \cos(x)$$

$$g(x) = e^{x^2} \Rightarrow g'(x) = 2xe^{x^2}$$

$$h(x) = \sin(x) + \cos(x)\, e^x \Rightarrow h'(x) = \cos(x) + e^x(\cos(x) - \sin(x))$$

Partial Derivatives

```
import sympy as sp
from sympy import *
expr=y*z-sp.log(z)-x-y
print(sp.diff(expr,x))
print(sp.diff(expr,y))
print(sp.diff(expr,z))
```

Higher Order Derivatives

```
import sympy as sp
x=sp.symbols('x')
expr=x**2+5*x+4
print(sp.diff(expr,x,x))
expr2=sp.exp(x**2)
print(sp.diff(expr2,x,x))
expr3=sp.sin(x)+sp.cos(x)*sp.exp(x)
print(sp.diff(expr3,x,x))
expr4=x**6
print(sp.diff(expr4,x,4))
```

```
m,n,a,b=symbols('m n a b')
expr=(a*x+b)**m
print(sp.diff(expr,x,m))
```

Higher Partial Derivatives

```
import sympy as sp
from sympy import *
expr=y*z-sp.log(z)-x-y
print(sp.diff(expr,y,z))
print(sp.diff(expr,z,z))
expr1=sp.exp(x*y*z)
print(sp.diff(expr1,x,y,z))
print(sp.diff(expr1,x,y,2,z,3))
```

Higher Partial Derivatives

```
import sympy as sp
from sympy import *
expr=y*z-sp.log(z)-x-y
print(sp.diff(expr,y,z))
print(sp.diff(expr,z,z))
expr1=sp.exp(x*y*z)
print(sp.diff(expr1,x,y,z))
print(sp.diff(expr1,x,y,2,z,3))
```

Differentiate

```
import sympy as sp
from sympy import *
expr=sp.exp(x**2)
print(sp.integrate(expr),(x,-oo,oo))
print(sp.integrate(sp.sin(x**2), (x, -oo, oo)))
print(sp.integrate(sin(x),(x,a,b)))
print(sp.integrate(x**2,(x,a,b)))
print(sp.integrate(sp.exp(-x)*sp.sin(x),(x,0,sp.pi/2)))
```

Differentiate

```
import sympy as sp
from sympy import *
expr=sp.exp(x**2)
print(sp.integrate(expr),(x,-oo,oo))
print(sp.integrate(sp.sin(x**2), (x, -oo, oo)))
```

Simplify

```
import sympy as sp
from sympy import *
x,y,z=symbols('x y z')
expr = x*y + x - 3 + 2*x**2 - z*x**2 + x**3
print(collect(expr, x))
```

Solvers

```
import sympy as sp
from sympy import *
print(solveset(Eq(x**2,1),x))
print(solveset(Eq(x**2-1,0),x))
print(solveset(x**2+1,x,domain=S.Reals))
print(solveset(x**2+1,x))
```

Linear Solvers

```
import sympy as sp
from sympy import *
print(linsolve((x+y+z-1,x+y+2*z-3),(x,y,z))))
print(linsolve((x+y+z-1,x+y+2*z-3,x-y+z-3),(x,y,z)))
```

Nonlinear Solvers

```
import sympy as sp
from sympy import *
print(nonlinsolve([x*y-1,x-2],x,y))
print(nonlinsolve([sp.exp(x)-sp.sin(y),1/y-3],x, y))
```

Differential Equation Solvers

```
import sympy as sp
from sympy import *
f,g=symbols('f g', cls=Function)
print(dsolve(f(x).diff(x,x)+f(x),f(x)))
print(dsolve(Eq(g(x).diff(x,x)+2*g(x).diff(x)+g(x),sin(x))))
```

End of Python NumPy, SciPy, SymPy