Python Training

A basic overview

Introduction to numpy

Libraries

- Numpy: supports numerical and array operations
- Scipy:
- Pandas: supports data manipulation and analysis
- Visualization libraries: matplotlib, seaborne, bokeh, plotly, gmplot, and many others provide support for charts and graphs

numpy arrays

- The basic numpy data structure is an array
- An array is a sequential collection of "like" objects
- unlike python lists, numpy arrays contain objects of the same type
  - this makes indexed access faster
  - and more memory efficient
- numpy arrays are mutable
- numpy are optimized for matrix operations
  - much faster than lists
- numpy provides random number support

Introduction to NumPy

- Main object is the homogeneous multidimensional array.
- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In Numpy dimensions are called *axes*. The number of axes is *rank*.
- E.G.
- The coordinates of a point in 3D space [3,4,5] is an array of rank 1, because it has one axis. That axis has a length of 3.
- In the example below, the array has rank 2 (it is 2-dimensional). The first dimension (axis) has a length of 2, the second dimension has a length of 3.

```
[[ 1., 0., 0.],
 [ 0., 1., 2.]]
```

- Numpy's array class is called ndarray.
- It is also known by the alias array.
- Note : numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality.

Important Attribute of ndarray

- **ndarray.ndim**
  - The number of axes (dimensions) of the array.
  - In the Python world, the number of dimensions is referred to as *rank*.
- **ndarray.shape**
  - The dimensions of the array.
  - This is a tuple of integers indicating the size of the array in each dimension.
  - For a matrix with *n*rows and *m* columns, shape will be (n,m).
  - The length of the shape tuple is therefore the rank, or number of dimensions,ndim.
- **ndarray.size**
  - The total number of elements of the array. This is equal to the product of the elements of shape.

Important Attribute of ndarray

- **ndarray.dtype**
  - an object describing the type of the elements in the array.
  - One can create or specify dtype's using standard Python types.
  - Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- **ndarray.itemsize**
  - the size in bytes of each element of the array.
  - E.G

    item type -> float64 -> itemsize 8 (=64/8)

    complex32 -> itemmsize 4 (=32/8).

  It is equivalent to ndarray.dtype.itemsize.
- **ndarray.data**
  - the buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

Create numpy array

- a = np.array([1,2,3,4]) # RIGHT

- b = np.array([(1.5,2,3), (4,5,6)])
- c = np.array([[1,2], [3,4]], dtype=complex)
- c = np.array([[1,2], [3,4]], dtype=complex)
- #create array of size 3 X 4 initialize with zero
- np.zeros((3,4))

- # dtype can also be specified, #create array of size 3 X 4 initialize with ones
- np.ones((2,3,4), dtype=np.int16)
- To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists
  - >>> np.empty((2,3)) # uninitialized, output may vary

Error in creating array

- **Wrong way to create array**
  - np.array(1,2,3,4) # WRONG

Print array contents

- b = np.arange(12).reshape(4,3) # 2d array
- >>> print(b)

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
```

- >>> c = np.arange(24).reshape(2,3,4) # 3d array
- c.ndim
- >>> print(c)

```
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Operations on array

```
a = np.array( [20,30,40,50] )
b = np.arange( 4 )
print( b)
 #o/p array([0, 1, 2, 3])
c = a-b
print( c)
#o/p array([20, 29, 38, 47])
 b**2
#o/p array([0, 1, 4, 9])
10*np.sin(a)
#o/p array([ 9.12945251, -9.88031624,          7.4511316 , -
2.62374854])
```

- A = np.array([[1,1],
  [0,1]])
- >>> B = np.array([[2,0],
  [3,4]])
- >>> A*B # elementwise product
  array([[2, 0],
  [0, 4]])
- >>> A.dot(B) # matrix product
  array([[5, 4],
  [3, 4]])
- >>> np.dot(A, B)
  array([[5, 4], # another matrix product
  [3, 4]])

Operations in the array

```
a = np.ones((2,3), dtype=int)
b = np.random.random((2,3))
a *= 3
# o/p---array([[3, 3, 3],
              [3, 3, 3]])

b += a
print( b)
#o/p--- array([[ 3.417022    ,  3.72032449,  3.00011437],
               [ 3.30233257,  3.14675589,  3.09233859]])
a += b                    # b is not automatically converted to integer type
        Traceback (most recent call last):
          ...
        TypeError: Cannot cast ufunc add output from dtype('float64') to
        dtype('int64') with casting rule 'same_kind'
```

```
A=np.arange(12).reshape(6,2)
a.resize((2,6))
Print( a)
```

The main difference is that with eye the diagonal can may be offset, whereas identity only fills the main diagonal.

- a=np.identity(3)
- b=np.eye(3,3)
- # Python Programming illustrating
- # numpy.eye method
- import numpy as np
- # 2x2 matrix with 1's on main diagnol
- b = np.eye(2, dtype = float)
- print("Matrix b : \n", b)

- # matrix with R=4 C=5 and 1 on diagnol
- # below main diagnol
- a = np.eye(4, 5, k = -1)
- print("\nMatrix a : \n", a)
- Output : Matrix b :
- [[1. 0.]
- [0. 1.]]
- Matrix a :
- [[0. 0. 0. 0. 0.]
- [1. 0. 0. 0. 0.]
- [0. 1. 0. 0. 0.]
- [0. 0. 1. 0. 0.]

numpy.eye() in Python
numpy.eye(R, C = None, k = 0, dtype = type <'float'>) : Return a matrix having 1's on the diagonal and 0's elsewhere.
Parameters :

R : Number of rows
C : [optional] Number of columns; By default M = N
k : [int, optional, 0 by default]
Diagonal we require; k>0 means diagonal above main diagonal or vice versa.
dtype : [optional, float(by Default)] Data type of returned array.
Returns :

array of shape, R x C, an array where all elements
are equal to zero, except for the k-th diagonal,
whose values are equal to one.

Generating Random Numbers

Import Numpy
import numpy as np
Generate A Random Number From The Normal Distribution
np.random.normal()
0.5661104974399703
Generate Four Random Numbers From The Normal Distribution
np.random.normal(size=4)
array([-1.03175853, 1.2867365 , -0.23560103, -1.05225393])

Generate Four Random Numbers From The Uniform Distribution

```
np.random.uniform(low=4,high=100,size=4)
array([ 0.00193123,  0.51932356,  0.87656884,
0.33684494])
```

Generate Four Random Integers Between 1 and 100

```
np.random.randint(low=1, high=100, size=4)
array([96, 25, 94, 77])
```

```
a=np.array([12,13,14,1,5])
print(a[1:4])
print(a[:2])

#slicing numpy array
b=np.arange(12).reshape(2,6)
print(b)
print(b[:,:3])
print(b[1,:3])
```

```
#filtering array
a=np.array([12,10,3,4,23,1,5])
n=[]
for i in a:
    if i>10:
        n.append(i)
print(n)

f_arr=a>10
#[true,false,false,false,true,false,false]
print(f_arr)
n1=a[f_arr]
N2=a[a>10 ]
print(n1)
```

```
f_arr=np.logical_or(a>10,a%2==10)
test=a[f_arr]
Print(test)
```

```
#searching element

arr = np.array([1, 12, 3, 4,9, 6, 7, 8])
#all index positions where the given condition is true
x = np.where(arr%3 == 0)

print(x)
```

```
#arrange data in sorted order(in
place sort)
arr.sort()
print(arr)
#arrange data in descending
order(in place sort)
arr[::-1].sort()
print(arr)
```

```
#sort array
print(np.sort(arr))
```

```
#sorted search
arr = np.array([5, 7, 8, 15])

x = np.searchsorted(arr, 9)

print(x)
```