Object Oriented Programming (Using Python)

UNIT- III

Python Libraries:

- Numpy
  - Indexing, Slicing and Iterating
  - QR Decomposition

Prof. R. MADANA MOHANA
Professor, Artificial Intelligence & Data Science
http://rmadanamohana.com/

Indexing, Slicing and Iterating

- One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.
- NumPy indexing is used for accessing an element from an array by giving it an index value that starts from 0.
- Slicing NumPy arrays means extracting elements from an array in a specific range. It obtains a substring, subtuple, or sublist from a string, tuple, or list.
- Negative Slicing index values start from the end of the array.
- Indexing starts from 0 and slicing is performed using indexing.

Indexing an Array

- Indexing is used to access individual elements.
- It is also possible to extract entire rows, columns, or planes from multi-dimensional arrays with numpy indexing.
- Indexing starts from 0.

Element of array	2	3	11	9	6	4	10	12
Index	0	1	2	3	4	5	6	7

Indexing using Index Arrays

- In NumPy arrays, when arrays are used as indexes to access groups of elements, this is called indexing using index arrays.
- NumPy arrays can be indexed with arrays or with any other sequence like a list, etc.

Indexing using Index Arrays

Example:

```
>>> import numpy as np
>>> a=np.arange(1,10,2)
>>> a
array([1, 3, 5, 7, 9])
>>> print("Elements of array: ",a)
Elements of array:  [1 3 5 7 9]
>>> a1=a[np.array([4,0,2,-1,-2])]
>>> print("Indexed Elements of array a: ",a1)
Indexed Elements of array a:  [9 1 5 9 7]
```

Indexing in one dimension Array

One-dimensional arrays can be indexed, sliced and iterated over, much like lists and other Python sequences.

```
>>> import numpy as np
>>> a=np.arange(1,10,2)
>>> a
array([1, 3, 5, 7, 9])
>>> print("Elements of array: ",a)
Elements of array:  [1 3 5 7 9]
>>> a[0]
1
```

Indexing in one dimension Array

```
>>> a
array([1, 3, 5, 7, 9])
>>> a[2]
5
>>> a[4]
9
>>> a[-1]
9
>>> a[-2]
7
```

Indexing in one dimension Array

```
>>> a
array([1, 3, 5, 7, 9])
>>> a[1:3]# slicing - excludes a[3]
array([3, 5])
>>> a[0:4] # slicing - excludes a[4]
array([1, 3, 5, 7])
>>>  a[:4:2]=200  #  from  start  to  position  4,
exclusive, set every 2nd element to 200
>>> a
array([200, 3, 200, 7, 9])
```

Indexing in one dimension Array

```
>>> a
array([1, 3, 5, 7, 9])
>>> a[::-1]  # reversed a
array([9, 7, 5, 3, 1])
>>> for i in a: # iteration
    print(i**2)
1
9
25
49
81
```

Indexing in Multidimensional Arrays

- Multidimensional arrays can have one index per axis.
- These indices are given in a tuple separated by commas.

```
>>> import numpy as np
>>> def f(x, y):
        return 10 * x + y
>>> b = np.fromfunction(f, (5, 4), dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

Indexing in Multidimensional Arrays

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2, 3]
23
>>> b[1,2]
12
>>> b[0,1]
1
```

Indexing in Multidimensional Arrays

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[0:5, 1]  # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[:, 1]    # equivalent to the previous one
array([ 1, 11, 21, 31, 41])
>>> b[1:3, :]  # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

Indexing in Multidimensional Arrays

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[-1]  # The last row. Equivalent to b[-1, :]
array([40, 41, 42, 43])
```

Indexing in Multidimensional Arrays

Iterating over multidimensional arrays is done with respect to the first axis:

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> for row in b:
        print(row)
 [0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

Indexing in Multidimensional Arrays

However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array:

```
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> for element in b.flat:
        print(element)
0
1
2
3
10
..
```

Indexing in Multidimensional Arrays

```
>>> c = np.array([[[  0,  1,  2],
                   [ 10, 12, 13]],
                   [[100, 101, 102],
                   [110, 112, 113]]])
# a 3D array (two stacked 2D arrays)
>>> c
array([[[  0,   1,   2],
        [ 10,  12,  13]],
       [[100, 101, 102],
        [110, 112, 113]]])
```

Indexing in Multidimensional Arrays

```
>>> c = np.array([[[  0,  1,  2],
                   [ 10, 12, 13]],
                   [[100, 101, 102],
                   [110, 112, 113]]])
>>> c.shape
(2, 2, 3)
>>> c[1, ...]  # same as c[1, :, :] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[..., 2]  # same as c[:, :, 2]
array([[  2,  13],
       [102, 113]])
```

QR Decomposition (or Factorization) of a Matrix

- QR factorization of a matrix is the decomposition of a matrix say 'A' into 'A=QR' where Q is orthogonal and R is an upper-triangular matrix.
- We can calculate the QR decomposition of a given matrix with the help of `numpy.linalg.qr()`.

Syntax:

```
numpy.linalg.qr(a, mode='reduced')
```

Parameters:

a: *array_like, shape (..., M, N)*

An array-like object with the dimensionality of at least 2.

mode: *{'reduced', 'complete', 'r', 'raw'}, optional*

The mode is set by default to return the results of the reduced type.

QR Decomposition (or Factorization) of a Matrix

Syntax:

```
numpy.linalg.qr(a, mode='reduced')
```

Parameters:

mode: *{'reduced', 'complete', 'r', 'raw'}, optional*

The mode is set by default to return the results of the reduced type.

- For an input matrix of dimension MxN, the reduced mode results in Q and R matrices of dimensions MxK & KxN respectively.
- If the complete mode is being used for instance, then the Q and R matrices shall be returned with dimensions MxM & MxN respectively.
- Declaring an r mode shall return only the R matrix with KxN dimension.
- If one chooses the raw mode h & tau are returned with the dimensions NxM & K respectively. The array h contains the Householder reflectors that generate q along with r. The tau array contains scaling factors for the reflectors. In the deprecated 'economic' mode only h is returned.

QR Decomposition (or Factorization) of a Matrix

Example-1: Write NumPy program to calculate the QR decomposition of a given matrix.

```
>>> import numpy as np
# Define the input matrix
>>> A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Compute the QR decomposition of the matrix
>>> Q, R = np.linalg.qr(A)
# Print the Q and R matrices
>>> print("Q matrix:")
>>> print(Q)
>>> print("R matrix:")
>>> print(R)
```

QR Decomposition (or Factorization) of a Matrix

Example-1: Write NumPy program to calculate the QR decomposition of a given matrix.

Output:

```
>>> A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
Q matrix:
[[-0.12309149  0.90453403  0.40824829]
 [-0.49236596  0.30151134 -0.81649658]
 [-0.86164044 -0.30151134  0.40824829]]
R matrix:
[[-8.12403840e+00 -9.60113630e+00 -1.10782342e+01]
 [ 0.00000000e+00  9.04534034e-01  1.80906807e+00]
 [ 0.00000000e+00  0.00000000e+00 -8.88178420e-16]]
```

QR Decomposition (or Factorization) of a Matrix

Example-2: Write NumPy program to calculate the QR decomposition of a given matrix.

```
import numpy as np
# Original matrix
matrix1 = np.array([[1, 2, 3], [3, 4, 5]])
print(matrix1)
# Decomposition of the said matrix
q, r = np.linalg.qr(matrix1)
print('\nQ:\n', q)
print('\nR:\n', r)
```

QR Decomposition (or Factorization) of a Matrix

Example-2: Write NumPy program to calculate the QR decomposition of a given matrix.

```
Output:
[[1 2 3]
 [3 4 5]]
Q:
 [[-0.31622777 -0.9486833 ]
 [-0.9486833   0.31622777]]
R:
 [[-3.16227766 -4.42718872 -5.69209979]
 [ 0.         -0.63245553 -1.26491106]]
```