Object Oriented Programming (Using Python)

UNIT- III

Python Libraries:

- Numpy
  - Printing arrays
  - Basic Operations

https://numpy.org/doc/stable/user/quickstart.html#quickstart-array-creation

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

Printing Arrays

- When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:
  - the last axis is printed from left to right,
  - the second-to-last is printed from top to bottom,
  - the rest are also printed from top to bottom, with each slice separated from the next by an empty line.
- One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

Printing Arrays

Example: 1d array

```
>>> import numpy as np
>>> a = np.arange(5) # 1d array
>>> print(a)
[0 1 2 3 4]
>>> a
array([0, 1, 2, 3, 4])
```

Printing Arrays

Example: 2d array

```
>>> import numpy as np
>>> b = np.arange(12).reshape(4, 3) # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> b
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Printing Arrays

Example: 3d array

```
>>> import numpy as np
>>> c = np.arange(24).reshape(2, 3, 4)  # 3d array
>>> 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]]]
```

Printing Arrays

numpy.reshape

Gives a new shape to an array without changing its data.

```
Syntax:
numpy.reshape(a, newshape, order='C')
Parameters:
a: array_like # Array to be reshaped.
newshape:int or tuple of ints
```

The newshape should be compatible with the original shape. If an integer, then the result will be a 1-D array of that length. One shape dimension can be -1. In this case, the value is inferred from the length of the array and remaining dimensions.

Printing Arrays

numpy.reshape

Syntax:

```
numpy.reshape(a, newshape, order='C')
```

Parameters:

order: *{'C', 'F', 'A'}, optional*

Read the elements of a using this index order, and place the elements into the reshaped array using this index order.

'C' means to read / write the elements using C-like index order, with the last axis index changing fastest, back to the first axis index changing slowest.

'F' means to read / write the elements using Fortran-like index order, with the first index changing fastest, and the last index changing slowest.

'A' means to read / write the elements in Fortran-like index order if a is Fortran contiguous in memory, C-like order otherwise.

Printing Arrays

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>> import numpy as np
>>> print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>> print(np.arange(10000).reshape(100, 100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

Printing Arrays

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using set_printoptions.

```
>>>    np.set_printoptions(threshold=sys.maxsize)
# sys module should be imported
```

Basic Operations

- Arithmetic operators on arrays apply *elementwise*.
- A new array is created and filled with the result.

```
>>> import numpy as np
>>> a=np.array([10, 20, 30, 40])
>>> a
array([10, 20, 30, 40])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
```

Basic Operations

```
>>> a=np.array([10, 20, 30, 40])
>>> a
array([10, 20, 30, 40])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
>>> c=a-b
>>> c
array([10, 19, 28, 37])
>>> d=a+b
>>> d
array([10, 21, 32, 43])
```

Basic Operations

```
>>> a=np.array([10, 20, 30, 40])
>>> a
array([10, 20, 30, 40])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([-5.44021111,    9.12945251,  -9.88031624,
7.4511316 ])
```

Basic Operations

```
>>> a=np.array([10, 20, 30, 40])
>>> a
array([10, 20, 30, 40])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
>>> a<30
array([ True,  True, False, False])
>>> a>30
array([False, False, False,  True])
```

Basic Operations

- The product operator * operates elementwise in NumPy arrays.
- The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

```
>>> import numpy as np
>>> A = np.array([[1, 1],
                  [0, 1]])
>>> A
array([[1, 1],
       [0, 1]])
>>> B = np.array([[2, 0],
                  [3, 4]])
>>> B
array([[2, 0],
       [3, 4]])
```

Basic Operations

```
>>> A * B # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)# another matrix product
array([[5, 4],
       [3, 4]])
```

Basic Operations

- Some operations, such as += and *=, act in place to modify an existing array rather than create a new one:

```
>>> import numpy as np
>>> rg = np.random.default_rng(1)# create instance of
default random number generator
>>> a = np.ones((2, 3), dtype=int)
>>> a
array([[1, 1, 1],
       [1, 1, 1]])
>>> b = rg.random((2, 3))
>>> b
array([[0.51182162, 0.9504637 , 0.14415961],
       [0.94864945, 0.31183145, 0.42332645]])
```

Basic Operations

```
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b # b is not automatically converted to integer type
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    a += b # b is not automatically converted to integer type
numpy.core._exceptions._UFuncOutputCastingError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int32') with casting rule 'same_kind'
```

Basic Operations

- When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting):

```
>>> import numpy as np
>>> import math
>>> a = np.ones(3, dtype=np.int32)
>>> a
array([1, 1, 1])
>>> a
array([1, 1, 1])
>>> a.dtype.name
'int32'
```

Basic Operations

```
>>> b = np.linspace(0, math.pi, 3)
>>> b
array([0.        , 1.57079633, 3.14159265])
>>> b.dtype.name
'float64'
>>> c = a + b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
```

Basic Operations

```
>>> d = np.exp(c * 1j)
>>> d
array([          0.54030231+0.84147098j,          -
0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

Basic Operations

- Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class:

```
>>> import numpy as np
>>> a = rg.random((2, 3))
>>> a.sum()
3.105710952998157
>>> a.min()
0.027559113243068367
>>> a.max()
0.8277025938204418
```

Basic Operations

- By default, these operations apply to the array as though it were a list of numbers, regardless of its shape.
- However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

```
>>> import numpy as np
>>> b = np.arange(12).reshape(3, 4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

Basic Operations

```
>>> import numpy as np
>>> b = np.arange(12).reshape(3, 4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b.sum(axis=0)      # sum of each column
array([12, 15, 18, 21])
>>> b.min(axis=1)      # min of each row
array([0, 4, 8])
>>> b.cumsum(axis=1)  # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Universal Functions

- NumPy provides familiar mathematical functions such as sin, cos, and exp.
- In NumPy, these are called "universal functions" (ufunc).
- Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
>>> import numpy as np
```

Universal Functions

```
>>> import numpy as np
>>> B = np.arange(3)
>>> B
array([0, 1, 2])
>>> np.exp(B)
array([1.        , 2.71828183, 7.3890561 ])
>>> np.sqrt(B)
array([0.        , 1.        , 1.41421356])
>>> C = np.array([2., -1., 4.])
>>> C
array([ 2., -1.,  4.])
>>> np.add(B, C)
array([2., 0., 6.])
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

- Return the cross product of two (arrays of) vectors.

Syntax:

```
numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)
```

Universal Functions: cross product

`numpy.cross(a, b, axisa=-1, axisb=-1, axisc=-1, axis=None)`

Parameters:

a: array_like # Components of the first vector(s)

b: array_like # Components of the second vector(s)

axisa: *int, optional*

Axis of a that defines the vector(s). By default, the last axis.

axisb: *int, optional*

Axis of b that defines the vector(s). By default, the last axis.

axisc: *int, optional*

Axis of c containing the cross product vector(s). Ignored if both input vectors have dimension 2, as the return is scalar. By default, the last axis.

axis: *int, optional*

If defined, the axis of a, b and c that defines the vector(s) and cross product(s). Overrides *axisa*, *axisb* and *axisc*.

Returns: *c: ndarray* # Vector cross product(s)

Raises: *ValueError*

When the dimension of the vector(s) in a and/or b does not equal 2 or 3.

Universal Functions: cross product

cross product of two (arrays of) vectors:

Assuming we have vector A with elements (A1, A2, A3) and vector B with elements (B1, B2, B3), we can calculate the cross product of these two vectors as:

```
Cross Product = [(A2*B3) – (A3*B2), (A3*B1) – (A1*B3), (A1*B2) – (A2*B1)]
```

Example:

Vector A: (1, 2, 3)

Vector B: (4, 5, 6)

Cross Product = [(A2*B3) – (A3*B2), (A3*B1) – (A1*B3), (A1*B2) – (A2*B1)]

= [(2*6) – (3*5), (3*4) – (1*6), (1*5) – (2*4)]

= (-3, 6, -3)

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

```
>>> import numpy as np
>>> x = [1, 2, 3]
>>> x
[1, 2, 3]
>>> y = [4, 5, 6]
>>> y
[4, 5, 6]
>>> np.cross(x, y)
array([-3,  6, -3])
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

One vector with dimension 2:

```
>>> import numpy as np
>>> x = [1, 2]
>>> x
[1, 2]
>>> y = [4, 5, 6]
>>> y
[4, 5, 6]
>>> np.cross(x, y)
array([12, -6, -3])
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

Equivalently:

```
>>> import numpy as np
>>> x = [1, 2, 0]
>>> x
[1, 2,0]
>>> y = [4, 5, 6]
>>> y
[4, 5, 6]
>>> np.cross(x, y)
array([12, -6, -3])
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

Both vectors with dimension 2:

```
>>> import numpy as np
>>> x = [1, 2]
>>> x
[1, 2]
>>> y = [4, 5]
>>> y
[4, 5]
>>> np.cross(x, y)
array(-3)
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

Multiple vector cross-products.

Note that the direction of the cross product vector is defined by the right-hand rule:

```
>>> import numpy as np
>>> x = np.array([[1,2,3], [4,5,6]])
>>> y = np.array([[4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[-3,  6, -3],
       [ 3, -6,  3]])
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

Multiple vector cross-products.

The orientation of c can be changed using the axisc keyword.

```
>>> import numpy as np
>>> x = np.array([[1,2,3], [4,5,6]])
>>> y = np.array([[4,5,6], [1,2,3]])
>>> np.cross(x, y, axisc=0)
array([[-3,  3],
       [ 6, -6],
       [-3,  3]])
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

Multiple vector cross-products.

Change the vector definition of x and y using *axisa* and *axisb*.

```
>>> import numpy as np
>>> x = np.array([[1,2,3], [4,5,6], [7, 8, 9]])
>>> y = np.array([[7, 8, 9], [4,5,6], [1,2,3]])
>>> np.cross(x, y)
array([[ -6,  12,  -6],
       [  0,   0,   0],
       [  6, -12,   6]])
```

Universal Functions: cross product

cross product of two (arrays of) vectors:

Example program: *Vector cross-product*

Multiple vector cross-products.

Change the vector definition of x and y using *axisa* and *axisb*.

```
>>> import numpy as np
>>> x = np.array([[1,2,3], [4,5,6], [7, 8, 9]])
>>> y = np.array([[7, 8, 9], [4,5,6], [1,2,3]])
>>> np.cross(x, y, axisa=0, axisb=0)
array([[-24,  48, -24],
       [-30,  60, -30],
       [-36,  72, -36]])
```

Universal Functions: Others

all, any, apply_along_axis, argmax, argmin, argsort, average, bincount, ceil, clip, conj, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, invert, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sort, std, sum, trace, transpose, var, vdot, vectorize, where

Refer:

https://numpy.org/doc/stable/user/quickstart.html#quickstart-array-creation