Math Basics in Python

www.huawei.com

HUAWEI

math and scipy Libraries

- The **math** library, a standard library of Python, provides some common mathematical functions; the **numpy** library, a numerical calculation and expansion library of Python, is mainly used to handle issues such as linear algebra, random number generation, and Fourier Transform; the **scipy** library is used to deal with issues such as statistics, optimization, interpolation, and bonus points.

ceil Implementation

- The value of **ceil(*x*)** is the minimum integer greater than or equal to *x*. If *x* is an integer, the returned value is *x*.

```
>>> import math
>>> import numpy as np
>>> math.ceil(4.01)
# Output:
>>> 5
# Code:
>>> math.ceil(4.99)
# Output:
>>> 5
```

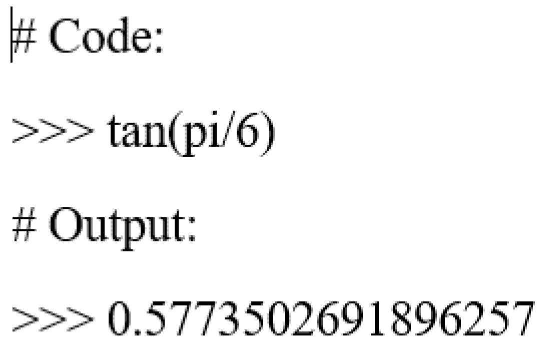
Cos Implementation

- The **cos(*x*)** parameter is the cosine of ***x***, where ***x*** must be a radian (**math.pi/4** is a radian, indicating an angle of 45 degrees).

```
# Code:

>>> math.cos(math.pi/4)

# Output:

>>> 0.7071067811865476

# Code:

>>> math.cos(math.pi/3)

# Output:

>>> 0.5000000000000001
```

Tan Implementation

- The **tan(*x*)** parameter returns the tangent value of *x* (radian).



```
# Code:

>>> tan(pi/6)

# Output:

>>> 0.5773502691896257
```

degrees Implementation

- The **degrees(*x*)** parameter converts *x* from a radian to an angle.

```
# Code:

>>> math.degrees(math.pi/4)

# Output:

>>> 45.0

# Code:

>>> math.degrees(math.pi)

# Output:

>>> 180.0
```

exp, fabs, and Factorial Implementations

- The **exp(*x*)** parameter returns **math.e**, that is, the *x* power of **2.71828**.
- The **fabs(*x*)** parameter returns the absolute value of *x*.
- The **factorial(*x*)** parameter is the factorial of *x*.

```
# Code:
>>> math.exp(1)
# Output:
>>> 2.718281828459045
```

```
# Code:
>>> math.fabs(-0.003)
# Output:
>>> 0.003
```

```
# Code:
>>> math.factorial(3)
# Output:
>>> 6
```

fsum and fmod Implementations

- The **fsum(iterable)** summarizes each element in the iterator.
- The **fmod(x, y)** parameter obtains the remainder of *x/y*. The value is a floating-point number.

```
# Code:
>>> math.fsum([1,2,3,4])
# Output:
>>>10
```

```
# Code:
>>> math.fmod(20,3)
# Output:
>>>2.0
```

log and sqrt Implementations

- The **log([*x*, base])** parameter returns the natural logarithm of *x*. By default, **e** is the base number. If the **base** parameter is fixed, the logarithm of *x* is returned based on the given **base**. The calculation formula is **log(x)/log(base)**.
- The **sqrt(*x*)** parameter indicates the square root of *x*.

```
# Code:

>>> math.log(10)

# Output:

>>> 2.302585092994046
```

```
# Code:

>>> math.sqrt(100)

# Output:

>>>10.0
```

pi, pow, trunc Implementations

- The **pi** parameter is a numeric constant, indicating the circular constant.
- The **pow(*x*, *y*)** parameter returns the *x* to the *y*th power, that is, *x****y*.
- The **trunc(*x*:Real)** parameter returns the integer part of *x*.

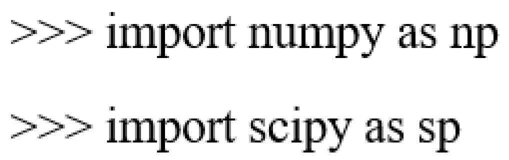
```
# Code:
>>> math.pi
# Output:
>>> 3.141592653589793
```

```
# Code:
>>> math.pow(3,4)
# Output:
>>> 81.0
```

```
# Code:
>>> math.trunc(6.789)
# Output:
>>> 6
```

Linear Algebra

- Linear algebra is a mathematical branch widely used in various engineering technical disciplines. Its concepts and conclusions can greatly simplify the derivation and expression of AI formulas. Linear algebra can simplify complex problems so that we can perform efficient mathematical operations.
- In the context of deep learning, linear algebra is a mathematical tool that provides a technique that helps us to operate arrays at the same time. Data structures like vectors and matrices can store numbers and rules for operations such as addition, subtraction, multiplication, and division.
- The numpy is a numerical processing module based on Python. It has powerful functions and advantages in processing matrix data. As linear algebra mainly processes matrices, this section is mainly based on the numpy. This section also uses the mathematical science library scipy to illustrate equation solution.



```
>>> import numpy as np

>>> import scipy as sp
```

Tensor Implementation (1)

- Generate a two-dimensional tensor whose elements are **0** and two dimensions are **3** and **4**.

```
>>> import math
>>> import numpy as np
# Code:
>>> np.zeros((3,4))
# Output:
>>> np.array([[ 0.,   0.,   0.,   0.],
             [ 0.,   0.,   0.,   0.],
             [ 0.,   0.,   0.,   0.]])
```

Tensor Implementation (2)

- Generate a three-dimensional random tensor whose three dimensions are **2**, **3**, and **4** respectively.

```
# Code:
>>> np.random.rand(2,3,4)
# Output:
>>> array([[[ 0.93187582,   0.4942617 ,   0.23241437,   0.82237576],
            [ 0.90066163,   0.30151126,   0.89734992,   0.56656615],
            [ 0.54487942,   0.80242768,   0.477167  ,   0.6101814 ]],

           [[ 0.61176321,   0.11454075,   0.58316117,   0.36850871],
            [ 0.18480808,   0.12397686,   0.22586973,   0.35246394],
            [ 0.01192416,   0.5990322 ,   0.34527612,   0.424322   ]]])
```

Identity Matrix Implementation

- Identity matrix is a square array whose diagonal elements are all **1** and other elements are **0**.

```
# Code:
>>> np.eye(4)
# Output:
>>> array([[ 1.,   0.,   0.,   0.],
           [ 0.,   1.,   0.,   0.],
           [ 0.,   0.,   1.,   0.],
           [ 0.,   0.,   0.,   1.]])
```

reshape Operation (1)

- There is no ‘reshape’ operation in mathematics, but it is a very common operation in the operation libraries such as the numpy and TensorFlow. The reshape operation is used to change the dimension number of a tensor and size of each dimension.
- For example, a 10x10 image is directly saved as a sequence containing 100 elements. After the system reads the image, you can transform the image from 1x100 to 10x10 through the reshape operation. The example is as follows:

```
# Code:

# Generate a vector that contains integers from 0 to 11.

>>> x = np.arange(12)

>>> x

# Output:

>>> array([ 0,   1,   2,   3,   4,   5,   6,   7,   8,   9, 10, 11])

# View the array size.

>>> x.shape

# Output:

>>>(12,)
```

reshape Operation (2)

# Convert the matrix x into a two-dimensional matrix, where the first dimension of the matrix is **1**.

```
>>> x = x.reshape(1,12)
>>> x
# Output:
>>> array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11]])
>>> x.shape
# Output:
>>>(1, 12)
# Convert x to a 3x4 matrix.
>>> x = x.reshape(3,4)
>>> x
# Output:
>>> array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11]])
```

Transposition Implementation (1)

- The transposition of vectors and matrices is exchanging the row and column. For the transposition of the tensors in three dimensions and above, you need to specify the transposition dimension.

```
# Code:

## Generate a vector x containing five elements and transposes the vector.

>>> x = np.arange(5).reshape(1,-1)

>>> x

array([[0, 1, 2, 3, 4]])

>>> x.T

array([[0],

       [1],

       [2],

       [3],

       [4]])
```

Transposition Implementation (2)

```
## Generate a 3x4 matrix and transpose the matrix.
>>> A = np.arange(12).reshape(3,4)
>>> A
# Output:
>>> array([[ 0,   1,   2,   3],
           [ 4,   5,   6,   7],
           [ 8,   9, 10, 11]])
>>> A.T
# Output:
>>> array([[ 0,   4,   8],
           [ 1,   5,   9],
           [ 2,   6, 10],
           [ 3,   7, 11]])
```

```
## Generate a 2x3x4 tensor.
>>> B = np.arange(24).reshape(2,3,4)
>>> B
# Output:
>>> array([[[ 0,   1,   2,   3],
            [ 4,   5,   6,   7],
            [ 8,   9, 10, 11]],

           [[12, 13, 14, 15],
            [16, 17, 18, 19],
            [20, 21, 22, 23]]])
```

Transposition Implementation (3)

```
## Transpose the 0 and 1 dimensions of B.
>>> B.transpose(1,0,2)
# Output:
>>> array([[[ 0,  1,  2,  3],
         [12, 13, 14, 15]],

        [[ 4,  5,  6,  7],
         [16, 17, 18, 19]],

        [[ 8,  9, 10, 11],
         [20, 21, 22, 23]]])
```

Matrix Multiplication

- To multiply the matrix A and matrix B, the column quantity of A must be equal to the row quantity of B.

```
# Code:
>>> A = np.arange(6).reshape(3,2)
>>> B = np.arange(6).reshape(2,3)
>>> A
# Output:
>>> array([[0, 1],
           [2, 3],
           [4, 5]])
```

```
>>> B
# Output:
>>> array([[0, 1, 2],
           [3, 4, 5]])
# Matrix multiplication
>>> np.matmul(A,B)
# Output:
>>> array([[ 3,   4,   5],
           [ 9, 14, 19],
           [15, 24, 33]])
```

Matrix Corresponding Operation

- Matrix corresponding operations are operations for tensors of the same shape and size. For example, adding, subtracting, multiplying, and dividing the elements with the same position in two tensors.

```
# Code:
# Matrix creation
>>> A = np.arange(6).reshape(3,2)
# Matrix multiplication
>>> A*A
# Output:
>>> array([[ 0,   1],
           [ 4,   9],
           [16, 25]])
```

```
# Matrix addition
>>> A + A
# Output:
>>> array([[ 0,   2],
           [ 4,   6],
           [ 8, 10]])
```

Inverse Matrix Implementation

- Inverse matrix implementation is applicable only to square matrices.

```
# Code:

>>> A = np.arange(4).reshape(2,2)

>>> A

# Output:

>>> array([[0, 1],

           [2, 3]])

>>> np.linalg.inv(A)

# Output:

>>> array([[-1.5,   0.5],

           [ 1. ,   0. ]])
```

Eigenvalue and Eigenvector

- Obtain the eigenvalue and eigenvector of a matrix.

```
# Code:
>>> import numpy as np # Introduce the numpy module.
>>> x= np.diag(1, 2, 3) # Write the diagonal matrix x.
>>> x                        #Output the diagonal matrix x.
# Output:
>>> array([[1,0,0],
[0,2,0],
[0,0,3]])
```

```
>>> a,b= np.linalg.eig(x)     # Assign the eigenvalue to a,
                              #and corresponding eigenvector to b.
>>> a
# Feature values: 1, 2, 3
# Output:
>>> array([1.,2.,3.])
>>> b
# Eigenvector
# Output:
>>> array([1.,0.,0.],
[0.,1.,0.],
[0.,0.,1.])
```

Determinant

- Obtain the determinant of a matrix.

```
# Code:
>>> E = array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
>>> F = array([[-1,   0,   1],
              [ 2,   3,   4],
              [ 5,   6,   7]])
# Output:
>>> np.linalg.det(E)
>>> 6.661338147750940 2e-16
# Output:
>>> np.linalg.det(F)
>>> 2.664535259100367e-15
```

Singular Value Decomposition (1)

- Create a matrix and perform singular value decomposition on the matrix.

```
# Code:
dataMat = [[1,1,1,0,0],
           [2,2,2,0,0],
           [1,1,1,0,0],
           [5,5,5,0,0],
           [1,1,0,2,2]]
>>> dataMat = mat(dataMat)
>>> U,Simga,VT = linalg.svd(dataMat)
>>> U
```

```
# Output:
>>> matrix([[ -1.77939726e-01,  -1.64228493e-02,   1.80501685e-02,
          9.53086885e-01,  -3.38915095e-02,   2.14510824e-01,
          1.10470800e-01],
        [ -3.55879451e-01,  -3.28456986e-02,   3.61003369e-02,
         -5.61842993e-02,  -6.73073067e-01,  -4.12278297e-01,
          4.94783103e-01],
        [ -1.77939726e-01,  -1.64228493e-02,   1.80501685e-02,
         -2.74354465e-01,  -5.05587078e-02,   8.25142037e-01,
          4.57226420e-01],
        [ -8.89698628e-01,  -8.21142464e-02,   9.02508423e-02,
         -1.13272764e-01,   2.86119270e-01,  -4.30192532e-02,
         -3.11452685e-01]])
```

Singular Value Decomposition (2)

```
>>> Simga
# Output:
>>>array([   9.72140007e+00,     5.29397912e+00,     6.84226362e-01,
               1.52344501e-15,     2.17780259e-16])
>>> VT
# Output:
>>> matrix([[ -5.81200877e-01,    -5.81200877e-01,    -5.67421508e-01,
               -3.49564973e-02,    -3.49564973e-02],
             [    4.61260083e-03,     4.61260083e-03,    -9.61674228e-02,
                 7.03814349e-01,     7.03814349e-01],
             [ -4.02721076e-01,    -4.02721076e-01,     8.17792552e-01,
                 5.85098794e-02,     5.85098794e-02],
             [ -7.06575299e-01,     7.06575299e-01,    -2.22044605e-16,
                 2.74107087e-02,    -2.74107087e-02],
             [    2.74107087e-02,    -2.74107087e-02,     2.18575158e-16,
                 7.06575299e-01,    -7.06575299e-01]])
```

Linear Equation Solving

- Solving a linear equation is simple because it requires only one function (**scipy.linalg.solve**).
- An example is finding the solution of the following system of non-homogeneous linear equations:

```
3x_1 + x_2 - 2x_3 = 5

x_1 - x_2 + 4x_3 = -2

2x_1 + 3x_3       = 2.5

# Code:

>>> from scipy.linalg import solve

>>> a = np.array([[3, 1, -2], [1, -1, 4], [2, 0, 3]])

>>> b = np.array([5, -2, 2.5])

>>> x = solve(a, b)

>>> x

# Output:

>>> [0.5 4.5 0.5]
```