BMI3 2.1 *Workshop 1*
Numpy, Pandas,
Advanced coding practices in python

Wanlu Liu

ZJU-UoE Institute

Wanluliu@intl.zju.edu.cn

2022/09/20

BMI3 Week2 2022

Learning Objectives

- Employ basic usage in Numpy
- Practice basic usage in Pandas
- Sketch advanced python programming skills

NumPy Basics

What & Why Numpy?

What is Numpy?

- NumPy is the primary array programming library for the Python language.

Why Numpy?

- Array programming provides a powerful, compact and expressive syntax for accessing, manipulating and operating on data in vectors, matrices and higher-dimensional arrays.
- It has an essential role in research analysis pipelines in fields as diverse as physics, chemistry, astronomy, geoscience, biology, psychology, materials science, engineering, finance and economics.
- For example, in astronomy, NumPy was an important part of the software stack used in the discovery of gravitational waves1 and in the first imaging of a black hole.

Harris et al., 2020, Nature
https://doi.org/10.1038/s41586-020-2649-2

Arrays and Vectorized Computation

1. The NumPy ndarray: A multidimensional array object
   - Creating ndarrays
   - Data types for ndarrays
   - Operations between Arrays and Scalars
   - Basic Indexing and Slicing
   - Bollean Indexing
   - Fancy Indexing
   - Transposing Arrays and Swapping Axes
2. Universal Functions: Fast Element-wise Array Functions
3. Data Processing Using Arrays
   - Expressing Conditional Logic as Array Operations
   - Mathematical and Statistical Methods
   - Methods for Boolean Arrays
   - Sorting
   - Unique and Other Set Logic

ndarray

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large data sets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements:

```
In [8]: data
Out[8]:
array([[ 0.9526, -0.246 , -0.8856],
       [ 0.5639,  0.2379,  0.9104]])
```

```
In [9]: data * 10
Out[9]:
array([[ 9.5256, -2.4601, -8.8565],
       [ 5.6385,  2.3794,  9.104 ]])
```

```
In [10]: data + data
Out[10]:
array([[ 1.9051, -0.492 , -1.7713],
       [ 1.1277,  0.4759,  1.8208]])
```

```
>>> data.shape
(2,3)
```

```
>>> data.dtype
dtype('float64')
```

1. Creating ndarrays

```
>>> import numpy as np

>>> data1=[6,7.5,8,0,1]
>>> arr1=np.array(data1)
>>> arr1
array([6. , 7.5, 8. , 0. , 1. ])
```

```
>>> data2=[[1,2,3,4],[5,6,7,8]]
>>> arr2=np.array(data2)
>>> arr2
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
>>> arr2.ndim
2
>>> arr2.shape
(2, 4)
```

```
>>> np.zeros(10)
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

>>> np.zeros((3,6))
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
>>> np.ones((2,4))
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.]])
>>> np.arange(6)
array([0, 1, 2, 3, 4, 5])

>>> np.empty((2,3,2))
array([[[0.00000000e+000,             nan],
        [1.69759663e-313, 0.00000000e+000],
        [0.00000000e+000, 0.00000000e+000]],


       [[0.00000000e+000, 0.00000000e+000],
        [0.00000000e+000, 0.00000000e+000],
        [0.00000000e+000, 0.00000000e+000]]])
```

1. Creating ndarrays

Table 4-1. Array creation functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list.
`ones, ones_like`	Produce an array of all 1's with the given shape and dtype. `ones_like` takes another array and produces a ones array of the same shape and dtype.
`zeros, zeros_like`	Like `ones` and `ones_like` but producing arrays of 0's instead
`empty, empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`eye, identity`	Create a square N x N identity matrix (1's on the diagonal and 0's elsewhere)

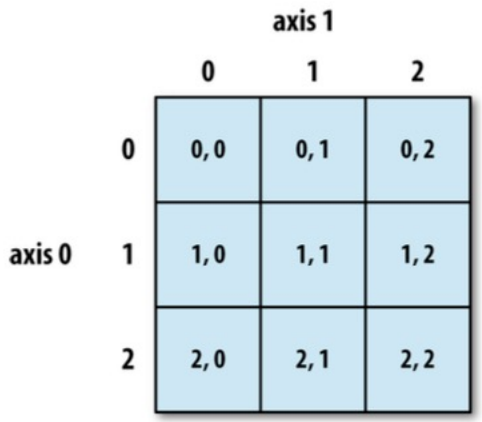
2. Data Types & Array Operations

```
>>> import numpy as np

>>> arr1=np.array([1,2,3],dtype=np.float64)
>>> arr1.dtype
dtype('float64’)

>>> arr2=np.array([1,2,3],dtype=np.int32)
>>> arr2.dtype
dtype('int32’)

>>> float_arr2 = arr2.astype(np.float64)
>>> float_arr2.dtype
dtype('float64')
```

```
>>> arr=np.array([[1.,2.,3.],[4.,5.,6.]])
>>> arr*arr
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
>>> arr-arr
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> 1/arr
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])
>>> arr**0.5
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

3. Basic indexing and Slicing

```
>>> arr=np.arange(10)
>>> arr
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> arr[5]
5
>>> arr[5:8]
array([5, 6, 7])
>>> arr[5:8]=12
>>> arr
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])

>>> arr_slice=arr[5:8]
>>> arr_slice[1]=12345
>>> arr
array([ 0, 1, 2, 3, 4, 12, 12345, 12, 8, 9])
>>> arr_slice[:]=64
>>> arr
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```



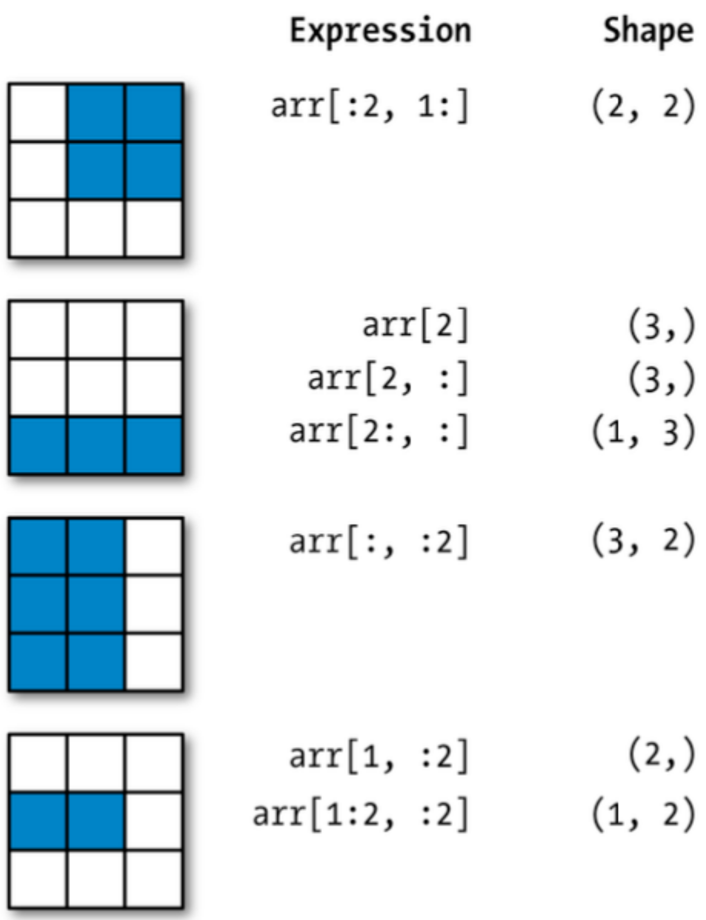

```
>>> arr2d=np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> arr2d[2]
array([7, 8, 9])
>>> arr2d[0][2]
3
>>> arr2d[0,2]
3
```

Try in your local python3 5 mins

3. Basic indexing and Slicing

```
>>> arr3d=np.array([[[1,2,3],[4,5,6]],[[7,8,9],[10,11,12]]])
>>> arr3d
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
>>> arr3d[1,0]
array([7, 8, 9])
>>> old_values = arr3d[0].copy()
>>> arr3d[0] = 42
>>> arr3d
array([[[42, 42, 42],
        [42, 42, 42]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
>>> arr3d[0] = old_values
>>> arr3d
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
```

```
>>> arr2d[:2,1:]
array([[2, 3],
       [5, 6]])
>>> arr2d[:2]
array([[1, 2, 3],
       [4, 5, 6]])
>>> arr2d[1,:2]
array([4, 5])
>>> arr2d[:,:1]
array([[1],
       [4],
       [7]])
```



4. Indexing： Boolean indexing

```
>>> names=np.array(['Bob','Joe','Will','Bob','Will','Joe'])
>>> data=np.random.randn(6,4)
>>> data
array([[-1.31348091,  0.10054471,  0.07040437, -1.63034692],
       [-0.26162686, -0.22517532,  2.08676258,  0.53789589],
       [ 0.05368436,  0.63623913,  0.48809193,  1.91152569],
       [-1.52996805, -2.22178153, -1.3306811 ,  0.10372846],
       [ 0.11923038, -0.36110603, -2.15844436,  1.08896468],
       [-0.46882034, -0.34377829, -0.16985646, -1.05410313]])
>>> names == 'Bob'
array([ True, False, False,  True, False, False])
>>> data[names == 'Bob']
array([[-0.92921052, -0.66745431,  1.48920967, -0.56851344],
       [-0.08092076, -1.23412985, -0.52527274, -0.39371382]])
>>> data[names == 'Bob', 2:]
array([[ 1.48920967, -0.56851344],
       [-0.52527274, -0.39371382]])
>>> data[names == 'Bob', 3]
array([-0.56851344, -0.39371382])
>>> names != 'Bob'
array([False,  True,  True, False,  True,  True])
>>> mask = (names == 'Bob') | (names == 'Will')
>>> mask
array([ True, False,  True,  True,  True, False])
>>> data[mask]
array([[-0.92921052, -0.66745431,  1.48920967, -0.56851344],
       [ 0.24824364, -0.130276  ,  0.18911803, -0.51220018],
       [-0.08092076, -1.23412985, -0.52527274, -0.39371382],
       [ 1.26515624,  0.63606059, -0.29542638, -0.61737474]])
```

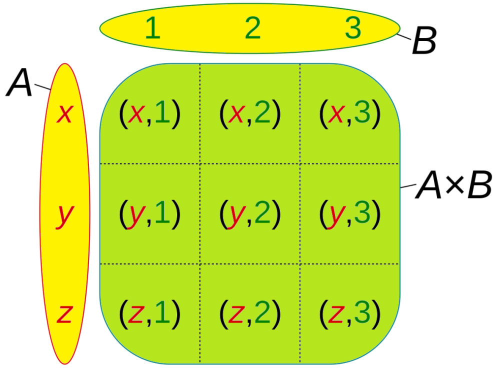
```
>>> data[data < 0] = 0
>>> data
array([[0.        , 0.        , 1.48920967, 0.        ],
       [0.43251166, 0.47011375, 0.        , 1.5109498 ],
       [0.24824364, 0.        , 0.18911803, 0.        ],
       [0.        , 0.        , 0.        , 0.        ],
       [1.26515624, 0.63606059, 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        ]])
>>> data[names != 'Joe'] = 7
>>> data
array([[7.        , 7.        , 7.        , 7.        ],
       [0.43251166, 0.47011375, 0.        , 1.5109498 ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [7.        , 7.        , 7.        , 7.        ],
       [0.        , 0.        , 0.        , 0.        ]])
```

4. Indexing： Fancy indexing

```
>>> arr=np.empty((8,4))
>>> for i in range(8):
...     arr[i]=i
...
>>> arr
array([[0., 0., 0., 0.],
       [1., 1., 1., 1.],
       [2., 2., 2., 2.],
       [3., 3., 3., 3.],
       [4., 4., 4., 4.],
       [5., 5., 5., 5.],
       [6., 6., 6., 6.],
       [7., 7., 7., 7.]])
>>> arr[[4,3,0,6]]
array([[4., 4., 4., 4.],
       [3., 3., 3., 3.],
       [0., 0., 0., 0.],
       [6., 6., 6., 6.]])
>>> arr[[-3,-5,-7]]
array([[5., 5., 5., 5.],
       [3., 3., 3., 3.],
       [1., 1., 1., 1.]])
```

```
>>> arr=np.arange(32).reshape((8,4))
>>> arr
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],
       [24, 25, 26, 27],
       [28, 29, 30, 31]])
>>> arr[[1,5,7,3],[0,3,1,2]]
array([ 4, 23, 29, 14])
>>> arr[[1, 5, 7, 2]][:, [0, 3, 1,
2]]
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
>>> arr[np.ix_([1,5,7,2],[0,3,1,2])]
array([[ 4,  7,  5,  6],
       [20, 23, 21, 22],
       [28, 31, 29, 30],
       [ 8, 11,  9, 10]])
```

np.ix_: Cartesian Product of the two 1D array index, then map to original array



5. Transposing Arrays and Swapping Axes

```
>>> arr
array([[1, 2, 2],
       [1, 3, 3]])
>>> arr.T
array([[1, 1],
       [2, 3],
       [2, 3]])
>>> np.dot(arr.T,arr)
array([[ 2,  5,  5],
       [ 5, 13, 13],
       [ 5, 13, 13]])
```

Dot Product

$$\begin{bmatrix} a & b \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \end{bmatrix}$$

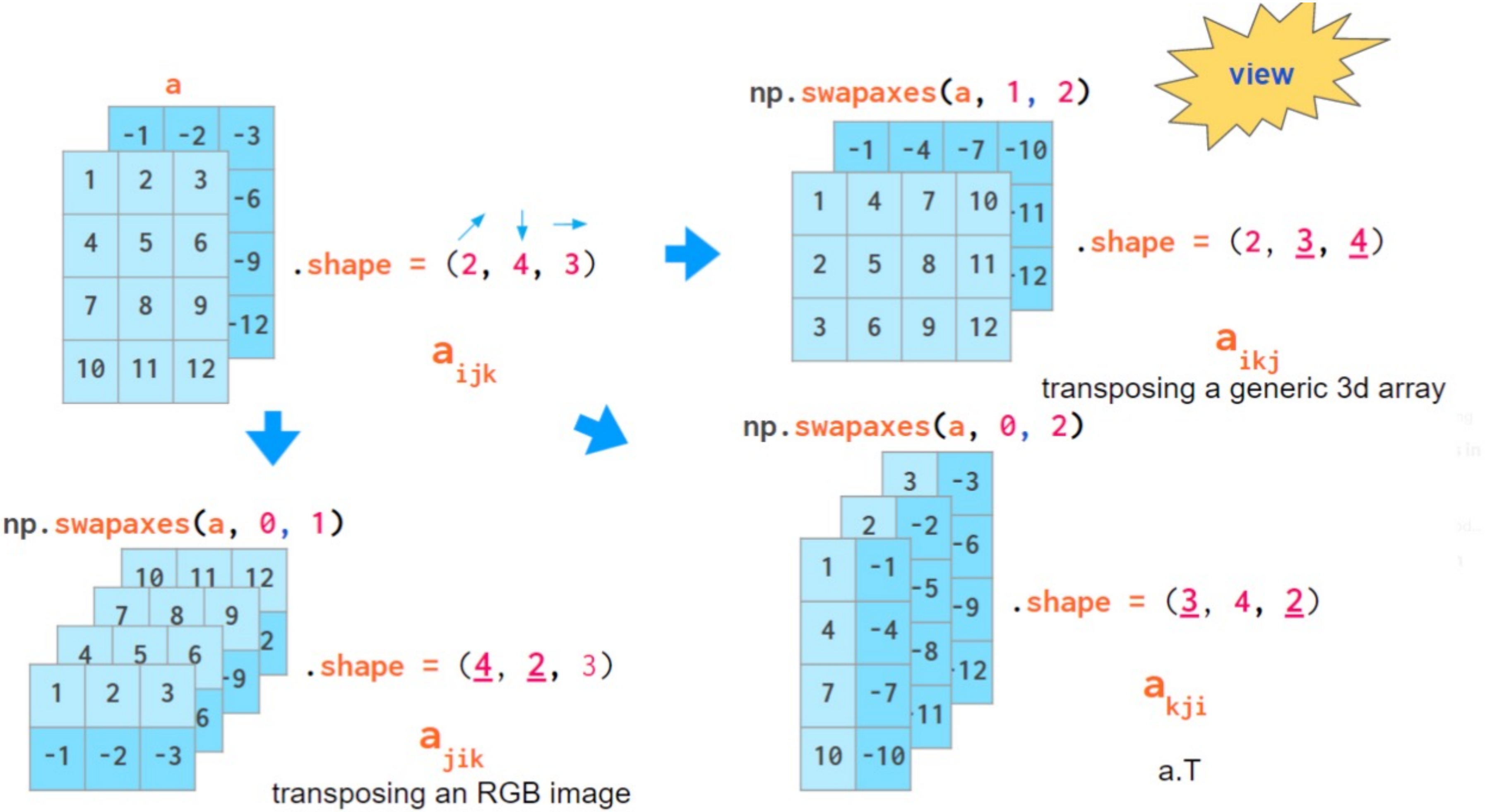
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \\ cx+dy \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 \\ 2 & 3 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 2 & 2 \\ 1 & 3 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 5 & 5 \\ 5 & 13 & 13 \\ 5 & 13 & 13 \end{bmatrix}$$

```
>>> arr = np.arange(16).reshape((2, 2, 4))
>>> arr
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
>>> arr.transpose((1, 0, 2))
array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
>>> arr
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
>>> arr.swapaxes(1,2)
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

5. Transposing Arrays and Swapping Axes



https://betterprogramming.pub/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d

6. Universal Functions: Fast Element-wise Array Functions

A universal function, or *ufunc*, is a function that performs elementwise operations on data in ndarrays.

Unary functions

```
>>> arr=np.arange(10)
>>> arr
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.sqrt(arr)
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
>>> np.exp(arr)
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

Binary functions

```
>>> x=np.random.randn(8)
>>> y=np.random.randn(8)
>>> x
array([-1.1459086 , -0.53527117,  0.56717833,  0.52651762,  0.51670925,
       -1.23571199,  0.08633214, -0.06281685])
>>> y
array([-1.40673137, -1.45959043,  0.41703775,  1.64817168,  1.11252536,
        0.01460472,  0.54736766,  1.07187065])
>>> np.maximum(x,y)
array([-1.1459086 , -0.53527117,  0.56717833,  1.64817168,  1.11252536,
        0.01460472,  0.54736766,  1.07187065])
```

6. Universal Functions: Fast Element-wise Array Functions

Table 4-3. Unary ufuncs

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating point, or complex values. Use `fabs` as a faster alternative for non-complex-valued data
`sqrt`	Compute the square root of each element. Equivalent to `arr ** 0.5`
`square`	Compute the square of each element. Equivalent to `arr ** 2`
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
`ceil`	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
`floor`	Compute the floor of each element, i.e. the largest integer less than or equal to each element
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate array
`isnan`	Return boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-NaN) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise. Equivalent to `-arr`.

Table 4-4. Binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum. `fmax` ignores `NaN`
`minimum, fmin`	Element-wise minimum. `fmin` ignores `NaN`
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators `>, >=, <, <=, ==, !=`
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation. Equivalent to infix operators `&`, `|`, `^`

7. Expressing Conditional Logic as Array Operations

```
>>> xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
>>> yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
>>> cond = np.array([True, False, True, True,
False])
>>> result = [(x if c else y)
...     for x,y,c in zip(xarr,yarr,cond)]
>>>
>>> result
[1.1, 2.2, 1.3, 1.4, 2.5]
>>> result = np.where(cond, xarr, yarr)
>>> result
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

```
>>> a=zip(xarr,yarr,cond)
>>> print(list(a))
[(1.1, 2.1, True), (1.2, 2.2, False), (1.3,
2.3, True), (1.4, 2.4, True), (1.5, 2.5,
False)]
```

tuples

```
>>> arr=np.random.randn(4,4)
>>> arr
array([[-0.98246205,  0.58904665,  1.36283387,  0.13804487],
       [ 0.30578553,  0.02511597, -0.15518313,  2.8678029 ],
       [-1.13660487,  0.44994736, -0.99470787,  0.30267681],
       [ 2.51073658, -1.38738111,  0.16085164,  0.35192243]])
>>> np.where(arr>0,2,arr)
array([[-0.98246205,  2.        ,  2.        ,  2.        ],
       [ 2.        ,  2.        , -0.15518313,  2.        ],
       [-1.13660487,  2.        , -0.99470787,  2.        ],
       [ 2.        , -1.38738111,  2.        ,  2.        ]])
```

8. Mathematical and Statistical Methods

```
>>> arr=np.random.randn(5,4)
>>> arr.mean()
0.0782421137909385З
>>> np.mean(arr)
0.0782421137909385З
>>> arr.sum()
1.5648422758187706
>>> arr.mean(axis=1)
array([-0.13839187, -0.35952662,
0.1747072 ,  0.45639111,  0.25803076])
>>> arr.sum(1)
array([-0.5535675 , -1.4381065 ,
0.6988288 ,  1.82556443,  1.03212305])
>>> arr.sum(0)
array([-0.20100136,  1.81332301, -
0.17813535,  0.13065598])
>>> arr.mean(1)
array([-0.13839187, -0.35952662,
0.1747072 ,  0.45639111,  0.25803076])
>>> arr.mean(0)
array([-0.04020027,  0.3626646 , -
0.03562707,  0.0261312 ])
```

Table 4-5. Basic array statistical methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
`mean`	Arithmetic mean. Zero-length arrays have NaN mean.
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
`min, max`	Minimum and maximum.
`argmin, argmax`	Indices of minimum and maximum elements, respectively.
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

9. Methods for Boolean Arrays, Sorting

```
>>> arr=np.random.randn(100)
>>> (arr>0).sum()
55
>>> bools=np.array([False,False,True,False])
>>> bools.any() #check for one or more values in an array is True
True
>>> bools.all() #check if every value is True
False
```

```
>>> arr=np.random.randn(8)
>>> arr
array([ 0.35604446,  1.6343853 ,  0.35655344, -1.02420942, -
1.20978584,1.43061948,  0.85080542, -0.35147079])
>>> arr.sort()
>>> arr
array([-1.20978584, -1.02420942, -0.35147079,  0.35604446,
0.35655344,0.85080542,  1.43061948,  1.6343853 ])
>>> arr=np.random.randn(5,3)
>>> arr
array([[ 0.46008956, -1.55225313,  0.08296969],
       [-1.28802708, -1.19972313,  0.14697091],
       [ 1.03209696,  0.49719421,  0.57570749],
       [ 0.19051936,  0.8154763 , -0.30915162],
       [-0.7155836 , -2.34356893, -0.68590218]])
```

```
>>> arr.sort(1)
>>> arr
array([[-1.55225313,  0.08296969,
0.46008956],[-1.28802708, -
1.19972313,  0.14697091],[
0.49719421,  0.57570749,
1.03209696],[-0.30915162,
0.19051936,  0.8154763 ],[-
2.34356893, -0.7155836 , -
0.68590218]])
```

10. Unique and Other Set Logic

```
>>> names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
>>> np.unique(names)
array(['Bob', 'Joe', 'Will'], dtype='<U4')
>>> ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
>>> np.unique(ints)
array([1, 2, 3, 4])
>>> sorted(set(names))
['Bob', 'Joe', 'Will']
>>> values = np.array([6, 0, 0, 3, 2, 5, 6])
>>> np.in1d(values, [2, 3, 6])
array([ True, False, False,  True,  True, False,  True])
```

Table 4-6. Array set operations

Method	Description
unique(x)	Compute the sorted, unique elements in x
intersect1d(x, y)	Compute the sorted, common elements in x and y
union1d(x, y)	Compute the sorted union of elements
in1d(x, y)	Compute a boolean array indicating whether each element of x is contained in y
setdiff1d(x, y)	Set difference, elements in x that are not in y
setxor1d(x, y)	Set symmetric differences; elements that are in either of the arrays, but not both

PTA Numpy Practice

2022 BMI3 Week2.1 - Session 1-Numpy basics

</> 编程题 8

标号	标题	分数
7-1	[NumPy]Sort array by column or rows	10
7-2	[NumPy]Get the unique elements	10
7-3	[NumPy]The count of non zero	10
7-4	[NumPy]Reverse an array	10
7-5	[NumPy]Comparison	10
7-6	[NumPy]Multiply the values of two given vectors	10
7-7	[NumPy]Sum	10
7-8	[NumPy]Numbers of rows and columns	10

Getting Started with pandas

What & Why Pandas?

What is Pandas?

pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

Why Pandas?

- Excellent representation of data
- Efficient (less coding done, more work accomplished)
- Can handle huge data
- Extensive feature set
- Flexibility of data and easy customization

Arrays and Vectorized Computation

1. Introduction to pandas Data Structures
   - Series
   - DataFrame
   - IndexObjects
2. Essential Functionality
   - Reindexing
   - Dropping entries from an axis
   - Indexing, selection, and filtering
   - Arithmetic and data alignment
   - Function application and mapping
   - Sorting and ranking
3. Unique Values, value counts, membership
4. Handling Missing Data
5. Hierarchical Indexing

1. Series

```
>>> from pandas import Series, DataFrame
>>> import pandas as pd
>>> obj=Series([4,7,-5,3])
>>> obj
0    4
1    7
2   -5
3    3
dtype: int64
>>> obj.values
array([ 4,  7, -5,  3])
>>> obj.index
RangeIndex(start=0, stop=4, step=1)
>>> obj2=Series([4,7,-5,3], index=['d','b','a','c'])
>>> obj2
d    4
b    7
a   -5
c    3
dtype: int64
>>> 'b' in obj2
True
>>> 'e' in obj2
False
```

```
>>> obj2.index
Index(['d', 'b', 'a', 'c'], dtype='object')
>>> obj2['a']
-5
>>> obj2['d']
4
>>> obj2[['c','a','d']]
c    3
a   -5
d    4
dtype: int64
>>> obj2[obj2>0]
d    4
b    7
c    3
dtype: int64
>>> obj2*2
d     8
b    14
a   -10
c     6
dtype: int64
>>> np.exp(obj2)
d      54.598150
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

1. Series

```
>>> sdata = {'Ohio': 35000, 'Texas': 71000,
'Oregon': 16000, 'Utah': 5000}
>>> obj3 = Series(sdata)
>>> obj3
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
>>> states = ['California', 'Ohio', 'Oregon',
'Texas']
>>> obj4 = Series(sdata, index=states)
>>> obj4
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
>>> pd.isnull(obj4) #opposite: pd.notnull()
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
>>> obj3
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
>>> obj4
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
>>> obj3+obj4
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
>>> obj4.name = 'population'
>>> obj4.index.name = 'state'
>>> obj4
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

2. DataFrame

```
>>> sdata = {'Ohio': 35000, 'Texas': 71000,
'Oregon': 16000, 'Utah': 5000}
>>> obj3 = Series(sdata)
>>> obj3
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
>>> states = ['California', 'Ohio', 'Oregon',
'Texas']
>>> obj4 = Series(sdata, index=states)
>>> obj4
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
>>> pd.isnull(obj4) #opposite: pd.notnull()
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

```
>>> obj3
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
>>> obj4
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
>>> obj3+obj4
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
>>> obj4.name = 'population'
>>> obj4.index.name = 'state'
>>> obj4
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

2. DataFrame

```
>>> data = {'state': ['Ohio', 'Ohio', 'Ohio',
'Nevada', 'Nevada'], 'year': [2000, 2001, 2002,
2001, 2002],
... 'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
>>> frame = DataFrame(data)
>>> frame
    state  year  pop
0    Ohio  2000  1.5
1    Ohio  2001  1.7
2    Ohio  2002  3.6
3  Nevada  2001  2.4
4  Nevada  2002  2.9
>>> DataFrame(data, columns=['year', 'state',
'pop'])
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
>>> frame2 = DataFrame(data, columns=['year',
'state', 'pop', 'debt'],index=['one', 'two',
'three', 'four', 'five'])
```

```
>>> frame2
       year   state  pop debt
one    2000    Ohio  1.5  NaN
two    2001    Ohio  1.7  NaN
three  2002    Ohio  3.6  NaN
four   2001  Nevada  2.4  NaN
five   2002  Nevada  2.9  NaN
>>> frame2.columns
Index(['year', 'state', 'pop', 'debt'], dtype='object')
>>> frame2['state']
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
Name: state, dtype: object
>>> frame2.year
one      2000
two      2001
three    2002
four     2001
five     2002
Name: year, dtype: int64
>>> frame2['debt']=16.5
>>> frame2
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
```

2. DataFrame

```
>>> frame2['debt'] = np.arange(5.)
>>> frame2
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
>>> val = Series([-1.2, -1.5, -1.7],
index=['two', 'four', 'five'])
>>> frame2['debt'] = val
>>> frame2
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
>>> frame2['eastern'] = frame2.state == 'Ohio'
>>> frame2
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
```

```
>>> del frame2['eastern']
>>> frame2.columns
Index(['year', 'state', 'pop', 'debt'],
dtype='object')
>>> frame2
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
>>> pop = {'Nevada': {2001: 2.4, 2002: 2.9},'Ohio':
{2000: 1.5, 2001: 1.7, 2002: 3.6}}
>>> frame3 = DataFrame(pop)
>>> frame3
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2000     NaN   1.5
>>> frame3.T
        2001  2002  2000
Nevada   2.4   2.9   NaN
Ohio     1.7   3.6   1.5
>>> DataFrame(pop, index=[2001, 2002, 2003])
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

2. DataFrame

```
>>> pdata = {'Ohio': frame3['Ohio'][:-1],'Nevada':
frame3['Nevada'][:2]}
>>> DataFrame(pdata)
      Ohio  Nevada
2001   1.7     2.4
2002   3.6     2.9
>>> frame3.index.name = 'year'; frame3.columns.name
= 'state'
>>> frame3
state  Nevada  Ohio
year
2001      2.4   1.7
2002      2.9   3.6
2000      NaN   1.5
>>> frame3.values
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
>>> frame2.values
array([[2000, 'Ohio', 1.5, nan],
       [2001, 'Ohio', 1.7, -1.2],
       [2002, 'Ohio', 3.6, nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7]], dtype=object)
```

Table 5-1. Possible data inputs to DataFrame constructor

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

3. Index Objects

Table 5-2. Main Index objects in pandas

Class	Description
Index	The most general Index object, representing axis labels in a NumPy array of Python objects.
Int64Index	Specialized Index for integer values.
MultiIndex	"Hierarchical" index object representing multiple levels of indexing on a single axis. Can be thought of as similar to an array of tuples.
DatetimeIndex	Stores nanosecond timestamps (represented using NumPy's datetime64 dtype).
PeriodIndex	Specialized Index for Period data (timespans).

Table 5-3. Index methods and properties

Method	Description
append	Concatenate with additional Index objects, producing a new Index
diff	Compute set difference as an Index
intersection	Compute set intersection
union	Compute set union
isin	Compute boolean array indicating whether each value is contained in the passed collection
delete	Compute new Index with element at index i deleted
drop	Compute new index by deleting passed values
insert	Compute new Index by inserting element at index i
is_monotonic	Returns True if each element is greater than or equal to the previous element
is_unique	Returns True if the Index has no duplicate values
unique	Compute the array of unique values in the Index

```
>>> obj = Series(range(3), index=['a', 'b', 'c'])
>>> index = obj.index
>>> index
Index(['a', 'b', 'c'], dtype='object')
>>> index[1:]
Index(['b', 'c'], dtype='object')
>>> index[1] = 'd'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/wanluliu/opt/miniconda3/lib/python3.8/site-packages/pandas/core/indexes/base.py", line 5035, in __setitem__
    raise TypeError("Index does not support mutable operations")
TypeError: Index does not support mutable operations
>>> index = pd.Index(np.arange(3))
>>> obj2 = Series([1.5, -2.5, 0], index=index)
>>> obj2.index is index
True
>>> frame3
state  Nevada  Ohio
year
2001      2.4   1.7
2002      2.9   3.6
2000      NaN   1.5
>>> 'Ohio' in frame3.columns
True
>>> 2003 in frame3.index
False
```

4. Reindexing & Dropping entries

```
>>> obj = Series([4.5, 7.2, -
5.3, 3.6], index=['d', 'b',
'a', 'c'])
>>> obj
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
>>> obj2 = obj.reindex(['a',
'b', 'c', 'd', 'e'])
>>> obj2
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
>>> obj.reindex(['a', 'b',
'c', 'd', 'e'], fill_value=0)
a   -5.3
b    7.2
c    3.6
d    4.5
e    0.0
dtype: float64
```

```
>>> obj = Series(np.arange(5.),
index=['a', 'b', 'c', 'd', 'e'])
>>> new_obj = obj.drop('c')
>>> new_obj
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
>>> obj.drop(['d', 'c'])
a    0.0
b    1.0
e    4.0
dtype: float64
>>> data =
DataFrame(np.arange(16).reshape((4, 4)),
... index=['Ohio', 'Colorado', 'Utah',
'New York'],
... columns=['one', 'two', 'three',
'four'])
>>> data.drop(['Colorado', 'Ohio'])
           one  two  three  four
Utah         8    9     10    11
New York    12   13     14    15
```

```
>>> data.drop('two',
... axis=1)
          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15
>>> data.drop(['two',
'four'], axis=1)
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

5. Indexing, selection, and filtering

```
>>> obj = Series(np.arange(4.),
... index=['a', 'b', 'c', 'd'])
>>> obj['b']
1.0
>>> obj[1]
1.0
>>> obj[[1, 3]]
b    1.0
d    3.0
dtype: float64
>>> obj[obj < 2]
a    0.0
b    1.0
dtype: float64
>>> obj['b':'c']
b    1.0
c    2.0
dtype: float64
>>> obj['b':'c'] = 5
>>> obj
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

```
>>> data = DataFrame(np.arange(16).reshape((4,
4)),index=['Ohio', 'Colorado', 'Utah', 'New
York'],columns=['one', 'two', 'three', 'four'])
>>> data
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
>>> data['two']
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
>>> data[['three', 'one']]
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
>>> data[:2]
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
```

```
>>> data[data['three'] > 5]
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
>>> data < 5
            one    two  three   four
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False
>>> data[data < 5] = 0
>>> data
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
>>> data.loc['Colorado', ['two',
'three']]
two      5
three    6
Name: Colorado, dtype: int64
```

6. Arithmetic and data alignment

```
>>> s1 = Series([7.3, -2.5, 3.4, 1.5],
index=['a', 'c', 'd', 'e'])
>>> s2 = Series([-2.1, 3.6, -1.5, 4, 3.1],
index=['a', 'c', 'e', 'f', 'g'])
>>> s1
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
>>> s2
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
>>> s1+s2
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

```
>>> df1=DataFrame(np.arange(9.).reshape((3,
3)), columns=list('bcd'),index=['Ohio',
'Texas', 'Colorado'])
>>> df2=DataFrame(np.arange(12.).reshape((4,
3)), columns=list('bde'),index=['Utah', 'Ohio',
'Texas', 'Oregon'])
>>> df1
          b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
>>> df2
        b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
>>> df1+df2
          b   c     d   e
Colorado  NaN NaN   NaN NaN
Ohio      3.0 NaN   6.0 NaN
Oregon    NaN NaN   NaN NaN
Texas     9.0 NaN  12.0 NaN
Utah      NaN NaN   NaN NaN
```

6. Arithmetic and data alignment

```
>>> s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
>>> s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
>>> s1
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
>>> s2
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
>>> s1+s2
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

```
>>> df1=DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),index=['Ohio', 'Texas', 'Colorado'])
>>> df2=DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),index=['Utah', 'Ohio', 'Texas', 'Oregon'])
>>> df1
          b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0
>>> df2
        b    d     e
Utah    0.0  1.0   2.0
Ohio    3.0  4.0   5.0
Texas   6.0  7.0   8.0
Oregon  9.0  10.0  11.0
>>> df1+df2
          b    c    d     e
Colorado  NaN  NaN  NaN   NaN
Ohio      3.0  NaN  6.0   NaN
Oregon    NaN  NaN  NaN   NaN
Texas     9.0  NaN  12.0  NaN
Utah      NaN  NaN  NaN   NaN
```

```
>>>df1=DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
>>>df2=DataFrame(np.arange(20.).reshape((4, 5)), columns=list('abcde'))
>>> df1+df2
      a     b     c     d    e
0   0.0   2.0   4.0   6.0  NaN
1   9.0  11.0  13.0  15.0  NaN
2  18.0  20.0  22.0  24.0  NaN
3   NaN   NaN   NaN   NaN  NaN
>>> df1.add(df2, fill_value=0)
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0  11.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
>>>df1.reindex(columns=df2.columns, fill_value=0)
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

7. Function application and mapping

```
>>> frame = DataFrame(np.random.randn(4, 3),
columns=list('bde'),index=['Utah', 'Ohio',
'Texas', 'Oregon'])
>>> f=lambda x: x.max() - x.min()
>>> f
<function <lambda> at 0x7fa14a3cee50>
>>> frame.apply(f)
b     1.899410
d     2.466276
e     2.651629
dtype: float64
>>> frame.apply(f, axis=1)
Utah      1.873209
Ohio      0.729609
Texas     0.722888
Oregon    2.200572
dtype: float64
>>> x = lambda a : a + 10
>>> print(x(5))
15
```

Lambda function:

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Syntax

`lambda` *arguments* `:` *expression*

```
>>> x = lambda a : a + 10
>>> print(x(5))
15
```


https://www.w3schools.com/python/python_lambda.asp

7. Function application and mapping

```
>>> def f(x):
...     return Series([x.min(), x.max()], 
index=['min', 'max'])
...
>>> frame.apply(f)
           b         d         e
min -0.79347 -1.210088 -0.487200
max  1.10594  1.256188  2.164429
>>> format = lambda x: '%.2f' % x
>>> frame.applymap(format)
            b      d      e
Utah    -0.62   1.26   0.62
Ohio    -0.09   0.55   0.64
Texas   -0.79  -1.21  -0.49
Oregon   1.11  -0.04   2.16
>>> frame['e'].map(format)
Utah       0.62
Ohio       0.64
Texas     -0.49
Oregon     2.16
Name: e, dtype: object
```

map function:

- The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax

```
map(function, iterables)
```

```
>>> def myfunc(n):
...     return len(n)
>>> x=map(myfunc, ('apple','banana','cherry'))
```


https://www.w3schools.com/python/ref_func_map.asp

8. Sorting and Ranking

```
>>> obj = Series(range(4), index=['d', 'a',
'b', 'c'])
>>> obj.sort_index()
a    1
b    2
c    3
d    0
dtype: int64
>>> frame =
DataFrame(np.arange(8).reshape((2, 4)),
index=['three', 'one'],columns=['d', 'a',
'b', 'c'])
>>> frame.sort_index()
       d  a  b  c
one    4  5  6  7
three  0  1  2  3
>>> frame.sort_index(axis=1)
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

```
>>> frame = DataFrame({'b': [4, 7, -3, 2],
'a': [0, 1, 0, 1]})
>>> frame
   b  a
0  4  0
1  7  1
2 -3  0
3  2  1
>>> frame.sort_values(by='b')
   b  a
2 -3  0
3  2  1
0  4  0
1  7  1
>>> frame.sort_values(by=['a','b'])
   b  a
2 -3  0
0  4  0
3  2  1
1  7  1
>>> frame.rank()
     b    a
0  3.0  1.5
1  4.0  3.5
2  1.0  1.5
3  2.0  3.5
```

```
>>> frame.rank(axis=1)
     b    a
0  2.0  1.0
1  2.0  1.0
2  1.0  2.0
3  2.0  1.0
```

9. Unique Values, Value Counts, and Membership

```
>>> obj = Series(['c', 'a', 'd', 'a', 'a',
'b', 'b', 'c', 'c'])
>>> uniques = obj.unique()
>>> uniques
array(['c', 'a', 'd', 'b'], dtype=object)
>>> obj.value_counts()
c    3
a    3
b    2
d    1
dtype: int64
>>> pd.value_counts(obj.values, sort=False)
c    3
a    3
d    1
b    2
dtype: int64
>>> mask = obj.isin(['b', 'c'])
```

```
>>> mask
0     True
1    False
2    False
3    False
4    False
5     True
6     True
7     True
8     True
dtype: bool
>>> obj[mask]
0    c
5    b
6    b
7    c
8    c
dtype: object
```

10. Missing data

```
>>> string_data = Series(['aardvark',
'artichoke', np.nan, 'avocado'])
>>> string_data
0     aardvark
1    artichoke
2          NaN
3      avocado
dtype: object
>>> string_data.isnull()
0    False
1    False
2     True
3    False
dtype: bool
```

```
>>> from numpy import nan as NA
>>> data = Series([1, NA, 3.5, NA, 7])
>>> data.dropna()
0    1.0
2    3.5
4    7.0
dtype: float64
>>> data[data.notnull()]
0    1.0
2    3.5
4    7.0
dtype: float64
>>> data = DataFrame([[1., 6.5, 3.], [1.,
NA, NA], [NA, NA, NA], [NA, 6.5, 3.]])
>>> cleaned = data.dropna()
>>> data
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
2  NaN  NaN  NaN
3  NaN  6.5  3.0
>>> cleaned
     0    1    2
0  1.0  6.5  3.0
>>> data.dropna(how='all')
     0    1    2
0  1.0  6.5  3.0
1  1.0  NaN  NaN
3  NaN  6.5  3.0
```

11. Hierarchical Indexing

- *Hierarchical indexing* is an important feature of pandas enabling you to have multiple (two or more) index *levels* on an axis.
- Somewhat abstractly, it provides a way for you to work with **higher dimensional** data in a **lower dimensional** form.

```
>>> data =
Series(np.random.randn(10),
...      index=[['a', 'a', 'a', 'b',
'b', 'b', 'c', 'c', 'd', 'd'],
...      [1, 2, 3, 1, 2, 3, 1, 2, 2,
3]])
>>> data
a  1   -0.072390
   2   -0.189012
   3   -2.484889
b  1    0.477828
   2   -0.091900
   3   -0.818407
c  1    0.132319
   2   -0.262038
d  2    0.744729
   3   -1.462555
dtype: float64
```

```
>>> data.index
MultiIndex([('a', 1),
            ('a', 2),
            ('a', 3),
            ('b', 1),
            ('b', 2),
            ('b', 3),
            ('c', 1),
            ('c', 2),
            ('d', 2),
            ('d', 3)],
           )
>>> data['b']
1    0.477828
2   -0.091900
3   -0.818407
dtype: float64
>>> data['b':'c']
b  1    0.477828
   2   -0.091900
   3   -0.818407
c  1    0.132319
   2   -0.262038
dtype: float64
>>> data.loc[['b', 'd']]
b  1    0.477828
   2   -0.091900
   3   -0.818407
d  2    0.744729
   3   -1.462555
dtype: float64
```

```
>>> data[:, 2]
a   -0.189012
b   -0.091900
c   -0.262038
d    0.744729
dtype: float64
>>> data.unstack()
          1         2         3
a -0.072390 -0.189012 -2.484889
b  0.477828 -0.091900 -0.818407
c  0.132319 -0.262038       NaN
d       NaN  0.744729 -1.462555
>>> data.unstack().stack()
a  1   -0.072390
   2   -0.189012
   3   -2.484889
b  1    0.477828
   2   -0.091900
   3   -0.818407
c  1    0.132319
   2   -0.262038
d  2    0.744729
   3   -1.462555
dtype: float64
```

PTA Pandas Practice

2022 BMI3 Week2.1 - Session 2 - Pandas basics

编程题 5

标号	标题
7-1	[Pandas] Sort Dataframe
7-2	[Pandas]Repetitive barcode
7-3	[Pandas] Average Score
7-4	[Pandas] First Names Only
7-5	[Pandas] Good Grades and Favorite Colors

Advanced Python Programming

1. Filter & reduce

- The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not.

Syntax:

filter(*function*, *iterable*)

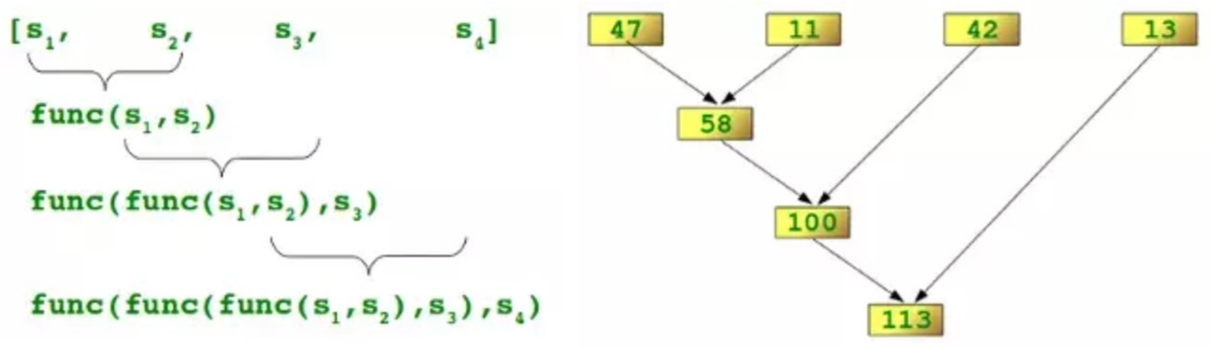
```
>>> fibonacci =
[0,1,1,2,3,5,8,13,21,34,55]
>>> odd_numbers = list(filter(lambda x:
x % 2, fibonacci))
>>> print(odd_numbers)
[1, 1, 3, 5, 13, 21, 55]
>>> even_numbers = list(filter(lambda
x: x % 2 == 0, fibonacci))
>>> print(even_numbers)
[0, 2, 8, 34]
```

- continually applies the function func() to the sequence seq. It returns a single value.

  Syntax:

reduce(func, seq)

```
>>> import functools
>>> functools.reduce(lambda x,y: x+y,
[47,11,42,13])
113
```



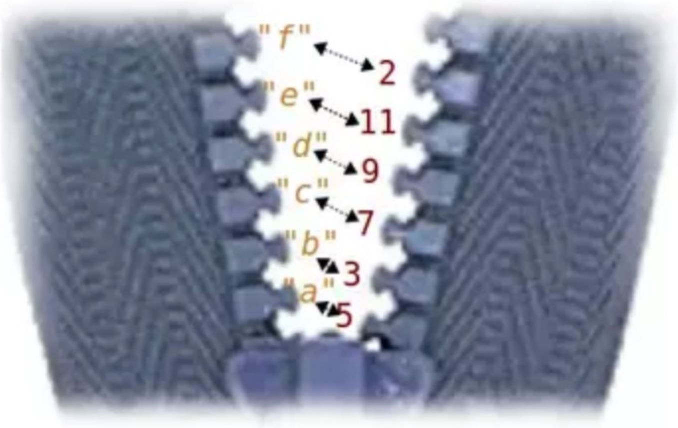
https://python-course.eu/advanced-python/lambda-filter-reduce-map.php

2. Zip

- The zip() function returns a zip object, which is an iterator of tuples where the first item in each passed iterator is paired together, and then the second item in each passed iterator are paired together etc.
- If the passed iterators have different lengths, the iterator with the least items decides the length of the new iterator.

Syntax:

zip(iterator1, iterator2, iterator3 ...)




```
>>> a_couple_of_letters = ["a", "b", "c",
"d", "e", "f"]
>>> some_numbers = [5, 3, 7, 9, 11, 2]
>>> print(zip(a_couple_of_letters,
some_numbers))
<zip object at 0x7fa14a4494c0>
>>> for t in zip(a_couple_of_letters,
some_numbers):
...     print(t)
...
('a', 5)
('b', 3)
('c', 7)
('d', 9)
('e', 11)
('f', 2)
```

https://www.w3schools.com/python/ref_func_zip.asp
https://python-course.eu/advanced-python/zip-tutorial.php

3. List Comprehension

- List comprehension is an elegant way to define and create lists in Python.
- List comprehension is a complete substitute for the lambda function as well as the functions map(), filter() and reduce().

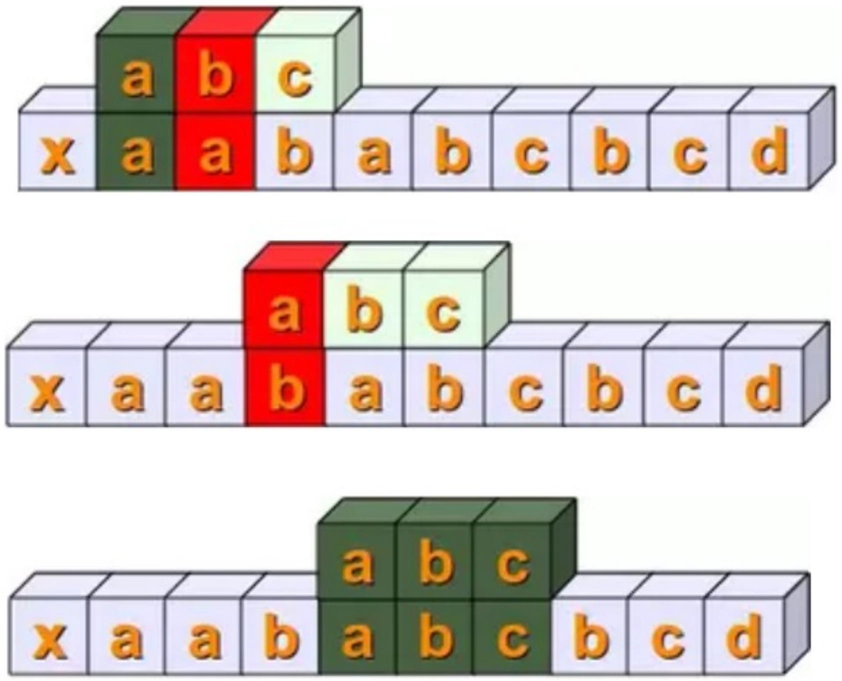
```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius ]
>>> print(Fahrenheit)
[102.56, 97.7, 99.14, 100.03999999999999]
```

```
>>> [(x,y,z) for x in range(1,30) for y in range(x,30) for z
in range(y,30) if x**2 + y**2 == z**2]
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15,
17), (9, 12, 15), (10, 24, 26), (12, 16, 20), (15, 20, 25),
(20, 21, 29)]
```

https://python-course.eu/advanced-python/list-comprehension.php

4. Regular Expression

- The term "regular expression", sometimes also called regex or regexp, has originated in theoretical computer science.
- In theoretical computer science, they are used to define a language family with certain characteristics, the so-called **regular languages.**




```
>>> import re
>>> x = re.search("cat", "A cat and a rat can't
be friends.")
>>> print(x)
<re.Match object; span=(2, 5), match='cat'>
```

```
>>> x = re.search("cow", "A cat and a rat can't
be friends.")
>>> print(x)
None
```

https://python-course.eu/advanced-python/regular-expressions.php

4. Regular Expression

1. Square brackets, "[" and "]": include a character class.
2. caret (^): beginning of a string
3. the dollar sign ($): end of a string
4. \d Matches any decimal digit; equivalent to the set [0-9].
5. \D The complement of \d. It matches any non-digit character; equivalent to the set [^0-9].
6. \s Matches any whitespace character; equivalent to [\t\n\r\f\v].
7. \S The complement of \s. It matches any non-whitespace character; equiv. to [^ \t\n\r\f\v].
8. \w Matches any alphanumeric character; equivalent to [a-zA-Z0-9_]. With LOCALE, it will match the set [a-zA-Z0-9_] plus characters defined as letters for the current locale.
9. \W Matches the complement of \w.
10. \b Matches the empty string, but only at the start or end of a word.
11. \B Matches the empty string, but not at the start or end of a word.
12. \\ Matches a literal backslash.

```
>>> import re
>>> line = "He is a German called Mayer."
>>> if re.search(r"M[ae][iy]er", line):
...     print("I found one!")
I found one!
```

https://python-course.eu/advanced-python/regular-expressions.php

5. Decorator

- A decorator in Python is any callable Python object that is used to modify a function or a class.
    - Function decorator
    - Class decorator
- A reference to a function "func" or a class "C" is passed to a decorator and the decorator returns a modified function or class.
- The modified functions or classes usually contain calls to the original function "func" or class "C".

```
>>> def now():
...     print('2022-9-22')
>>> f = now
>>> f()
2022-9-22
>>> now.__name__
'now'
>>> f.__name__
'now'
>>> def log(func):
...     def wrapper(*args, **kw):
...         print('call %s():' % func.__name__)
...         return func(*args, **kw)
...     return wrapper
```

```
>>> @log
... def now():
...     print('2022-9-22')
...
>>> now()
call now():
2022-9-22
```

```
>>> now=log(now)
```

https://python-course.eu/advanced-python/decorators-decoration.php

Other suggested Python packages in bioinformatics

- SciPy: Fundamental algorithms for scientific computing in python
  - https://scipy.org/
- IPython: Interative computing
  - https://ipython.org/
- Matplotlib: Visualization with Python
  - https://matplotlib.org/
- Biopython: Biological computation
  - https://biopython.org/
- Scanpy: single-cell analysis in python
  - https://scanpy.readthedocs.io/en/stable/
- sci-kit-learn: machine learning in python
  - https://scikit-learn.org/stable/

Learning Objectives - Summary

- Employ basic usage in Numpy
  - Ndarray, dataTypes, arrayOperation, Indexing/Slicing, Boolean indexing, FancyIndexing, TranposingArray and Swapping Axes, Universal Functions, Conditional Logic, Mathmatical/Statistical Methods, BooleanArrays, Unique/SetLogic
- Practice basice usage in Pandas
  - Series, DataFrame, IndexObjects, Reindexing, Dropping entries, Indexing/Selection/Filtering, Arithmetic and data alignment, Function application and mapping, Sorting/Ranking, Unique/Count/Membership, MissingData, Hierarchical indexing
- Sketch advanced python programming skills
  - Lambda Function, Map, Filter, Reduce, Zip, List Comprehension, Regular Expression, Decorator

Don’t forget homework & Code4Fun practice

2022 BMI3 Week 2 - Homework

</> 编程题 5

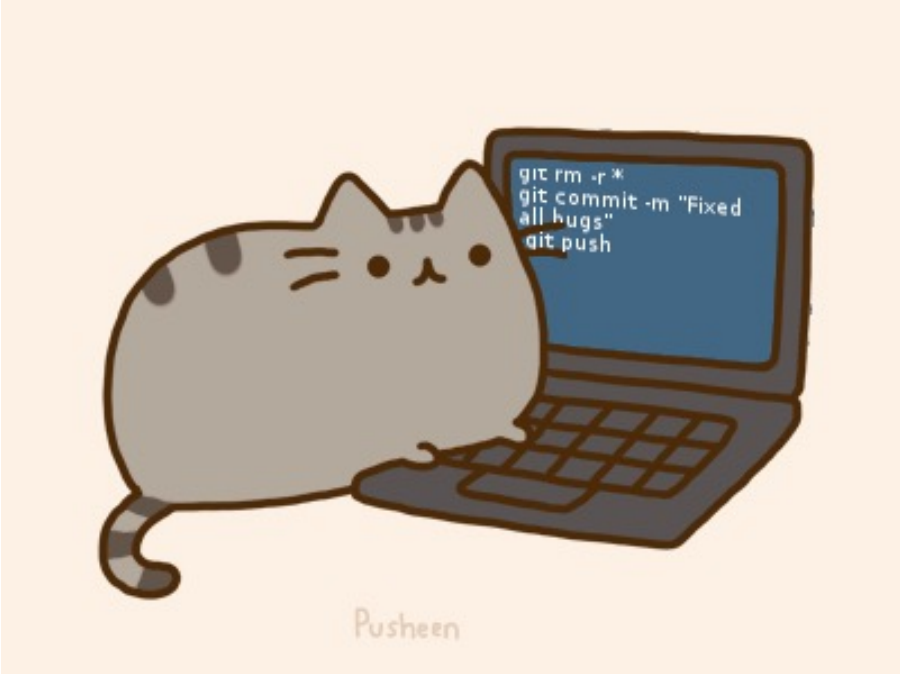
标号	标题
7-1	Partition List
7-2	Rotate List
7-3	Sliding Window Maximum
7-4	Valid Parentheses
7-5	Binary Tree Preorder Traversal

2022 BMI3 - Code for Fun

</> 编程题 22

7-12	H-Index II
7-13	Binary Search
7-14	Path With Minimum Effort
7-15	heap sort
7-16	Radix Sort
7-17	Escape the Spreading Fire
7-18	N-Queens
7-19	Tree Sort
7-20	First Unique Character in a String
7-21	Binary Tree Inorder Traversal
7-22	Remove Duplicate Letters


PRACTICE MAKES
PERFECT
memegenerator.net


git rm -r *
git commit -m "Fixed
all hugs"
git push
Pusheen