pandas numpy常用指令和用法

1如何数据分析

panda

总结

创建：array, arange, zeros...

选取: 使用slicing, **[: :]，[行，列]，[condition]**

组合：concatenate, vstack, hstack, reshape

分割：split, vsplit, hsplit

概念

entry: 也是元素

numpy 一个一维数组中，什么叫entry?

在 NumPy 或其他数组和矩阵相关的术语中，**entry**（有时也叫作元素）指的是数组中的单个值。在一维数组中，entry 就是数组中的每个数值或数据点。

例如，给定以下一维数组：

python 複製程式碼

```python
import numpy as np

arr = np.array([10, 20, 30, 40])
```

这个数组包含 4 个 entry，它们分别是：`10`，`20`，`30`，`40`。每个 entry 都是数组中的一个元素，可以通过数组的索引来访问它们。

Numpy

numpy解决了什么问题?

Let's illustrate the need for specialised arrays with an example. Consider we have 2 variables with tabular data read from a file. For simplicity, we will call them `tbl_1` and `tbl_2`. We need to perform some basic arithmetic operations like `tbl_1 + tbl_2`.

If we proceed to use lists, we need to ensure the following conditions are met before we can perform the operations:

1. data in the lists are compatible with the operation being performed,
2. we must keep track of dimensions (shape) of the table,
3. the size (shape) of the lists match, and
4. we must also recognise and handle missing values.

From the above requirements, it is clear that handling data for computation in Pythonic collections like lists is tedious. We would have to iterate through all elements to ensure all the conditions are met. Only then can we proceed with the computation. This is where NumPy comes in handy.

shape

Understanding 2D Arrays

In a 2D array, data is organized into both **rows** and **columns**, much like a table or a spreadsheet. Each element of the array is placed at the intersection of a row and a column. This structure allows for more complex data arrangements than a simple 1D array.

For example, if you have 20 elements, you can arrange them in various ways:

- 4 rows, each containing 5 elements (4x5),
- 10 rows, each containing 2 elements (10x2),
- 5 rows, each containing 4 elements (5x4).

In all these cases, the array holds the same total number of elements (20), but the layout or **shape** of the array varies. **Shape** refers to the number of rows and columns an array contains. For instance, an array with a shape of (4, 5) has 4 rows and 5 columns.

This idea of shape is crucial when working with 2D arrays because it helps you visualize and manipulate data. In contrast, a 1D array (which is a simpler structure) can be thought of as a single row of elements, with only one dimension to work with. In NumPy, both 1D and 2D arrays are common, but understanding how to use and reshape 2D arrays opens up a lot of possibilities for data analysis.

对于n维

N-dimensional arrays (ndarrays)

NumPy is not limited to just 1D or 2D arrays—it also supports **N-dimensional arrays**, known as **ndarrays**. These arrays can have any number of dimensions, allowing you to work with more complex datasets.

All NumPy arrays have a **shape** property, which indicates the dimensionality of the array. The shape of an array is represented as a tuple, where each element of the tuple corresponds to the size of the array along one dimension. The length of the tuple itself denotes the number of dimensions of the array.

For example:

- A 1D array with 5 elements might have a shape of `(5,)`.
- A 2D array with 3 rows and 4 columns will have a shape of `(3, 4)`.

For higher dimensions, consider a **3D array** with a shape of `(2, 3, 3)`. This means:

- The array has **2 blocks** (the first dimension),
- Each block contains **3 rows** (the second dimension),
- Each row has **3 elements** (the third dimension).

Thus, understanding the **shape** of an array is key to navigating its structure, whether you're working with 1D vectors, 2D matrices, or more complex multidimensional arrays.

block, row and elements.

numpy的运算符

universial function (unfunc)

When you use NumPy's **universal functions** (also called **ufuncs**), you are performing **vectorised operations**. These are designed to operate on entire arrays or large chunks of data at once, which greatly improves performance. Behind the scenes, these functions are written in low-level languages like C, enabling them to run much faster than standard Python loops.

For example, rather than using a `for` loop to add two arrays element by element, you can simply add the arrays directly with a single statement, and NumPy will handle the operation efficiently. This results in cleaner, more readable code, while also dramatically improving speed.

In short, **vectorisation** eliminates the need for explicit loops by applying fast, optimized mathematical operations across entire arrays, making your computations both simpler and faster.

Vectorised arithmetic operators (ufunc)

NumPy's universal functions (ufuncs) provide vectorised implementations of arithmetic functions. Use these whenever you need to do operations over large data sets in arrays, instead of using a for loop.

Operator	unfunc	Description
`+`	`np.add`	Addition
`-`	`np.subtract`	Subtraction
`-`	`np.negative`	Unary negation (e.g -5)
`*`	`np.multiply`	Multiplication
`/`	`np.divide`	Division
`**`	`np.power`	Exponentiation
`//`	`np.floor_divide`	Integer division
`%`	`np.mod`	Modulo (division remainder)

```
In [7]: np.add(d2,d22)

Out[7]: array([[ 4,  6,  8],
               [ 7,  9, 11]])
```

```
In [8]: np.add(d2,10)

Out[8]: array([[11, 12, 13],
               [12, 13, 14]])
```

```
In [10]: d2+100

Out[10]: array([[101, 102, 103],
                [102, 103, 104]])
```

```
In [11]: d2+d22

Out[11]: array([[ 4,  6,  8],
                [ 7,  9, 11]])
```

other ufunc functions

Other ufunc Functions

NumPy has many other functions. Some of these are the NaN-safe version. This means that NumPy will ignore missing values when applying the functions. The following table lists other functions in ufuncs.

Function	Nan-safe Version	Description
`np.all()`	Not available	Evaluate whether all elements are true
`np.any()`	Not available	Evaluate whether any elements are true
`np.argmax()`	`np.nanargmax()`	Find index of maximum value
`np.argmin()`	`np.nanargmin()`	Find index of minimum value
`np.max()`	`np.nanmax()`	Find maximum value
`np.mean()`	`np.nanmean()`	Compute mean of elements
`np.median()`	`np.nanmedian()`	Compute median of elements
`np.min()`	`np.nanmin()`	Find minimum value
`np.percentile()`	`np.nanpercentile()`	Compute rank-based statistics of elements
`np.prod()`	`np.nanprod()`	Compute product of elements
`np.std()`	`np.nanstd`	Compute standard deviation
`np.sort()`	Not available	return a sorted copy of an array
`np.sum()`	`np.nansum()`	Compute sum of elements
`np.transpose()`	Not available	Permute the dimensions of an array
`np.var()`	`np.nanvar()`	Compute variance

常用方法

- **arange(start,stop,step)**

```
output10=np.arange(10,161,10)
print(output10)
output10=output10.reshape(2,2,2,2)
output10
```

```
[ 10  20  30  40  50  60  70  80  90 100 110 120 130 140 150 160]
```

Out[21]:
```
array([[[[ 10,  20],
         [ 30,  40]],

        [[ 50,  60],
         [ 70,  80]]],


       [[[ 90, 100],
         [110, 120]],

        [[130, 140],
         [150, 160]]]])
```

Slicing

对于一维数组，一个[start,stop,step]去分

对于二维数组，[**rows , cols**]

start, stop, step

```
In [22]: a1=np.arange(20)
         a1
Out[22]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
                17, 18, 19])

In [25]: #start:stop:step
         a1[0::2]
Out[25]: array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])

In [26]: a1[0:5]
Out[26]: array([0, 1, 2, 3, 4])

In [27]: a1[0:5:2]
Out[27]: array([0, 2, 4])

In [28]: #做个逆
         a1[10::-1]
Out[28]: array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1,  0])
```

例子

Create a python program that has a 1-D array with 100 elements. It should contain the numbers 0, 2, 4, 6, all the way to 198 in sequence. Assign this array to `output9`.

```
[17]: import numpy as np

      # Write your solution here

      # YOUR CODE HERE
      output9=np.arange(200)[::2]
      output9

t[17]: array([  0,   2,   4,   6,   8,  10,  12,  14,  16,  18,  20,  22,  24,
               26,  28,  30,  32,  34,  36,  38,  40,  42,  44,  46,  48,  50,
               52,  54,  56,  58,  60,  62,  64,  66,  68,  70,  72,  74,  76,
               78,  80,  82,  84,  86,  88,  90,  92,  94,  96,  98, 100, 102,
              104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126, 128,
              130, 132, 134, 136, 138, 140, 142, 144, 146, 148, 150, 152, 154,
              156, 158, 160, 162, 164, 166, 168, 170, 172, 174, 176, 178, 180,
              182, 184, 186, 188, 190, 192, 194, 196, 198])
```

2d array

记住start: stop: step即可，这是核心

2d array slicing

```
In [29]: a1=np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
         a1

Out[29]: array([[ 1,  2,  3,  4],
                [ 5,  6,  7,  8],
                [ 9, 10, 11, 12]])
```

```
In [33]: #第二行第二列
         #行与列
         a1[1,1]

Out[33]: 6
```

```
In [36]: a1[1,1:3]

Out[36]: array([6, 7])
```

```
In [37]: a1[:2]

Out[37]: array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
```

```
In [38]: a1[:2,1:3]

Out[38]: array([[2, 3],
                [6, 7]])
```

:表示取所有rows。

1. initializes two 2-D arrays as depicted below.

$$a1 = \begin{bmatrix} 48 & 75 & 75 \\ 15 & 2 & 46 \end{bmatrix}, a2 = \begin{bmatrix} 78 & 15 & 87 \\ 0 & 61 & 9 \end{bmatrix}$$

2. multiplies the two arrays together, then slice the result to only assign the last column to `output20` (note: an n-row, 1-column array automatically converts into a 1-row, n-column array, leave your answer like this).

```
# Write your solution here

# YOUR CODE HERE
a1=np.array([[48,75,75],[15,2,46]])
a2=np.array([[78,15,87],[0,61,9]])
aa=a1*a2
aa
```

```
array([[3744, 1125, 6525],
       [   0,  122,  414]])
```

```
output20=aa[:,2]
output20
```

```
array([6525,  414])
```

Reshape

Reshaping arrays

Sometimes you may need to change the shape of an array you have created. For example, if you add a new column of data. You can do this using the `reshape()` function. You can also use `reshape()` to create a 2-dimensional array from 1D data.

The `reshape()` function takes the number of rows and columns to use for the 2D array.

In [25]:

```
# Using re-shape function and arrange to create a two-dimensional array from 1D data

#Create a 1D array
a1d = np.arange(0,12)
print ('a1d before reshape ', a1d)

# Reshape data to a 3x4 array
a1d=a1d.reshape(3,4)
print ('a1d after reshape \n', a1d)
```

Out [25]:

```
a1d before reshape  [ 0  1  2  3  4  5  6  7  8  9 10 11]
a1d after reshape
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Concatenate

也就是要么按照row，要么按照col来加在一起。默认以row为参照。

Concatenate arrays

We can combine multiple arrays joining them into one array using:

- `np.concatenate()` Join a sequence of arrays along an existing axis (rows or columns).
  Parameters:
  - A sequence of arrays to concatenate. The arrays must have the same shape, except in the dimension corresponding to axis (rows by default). This means that there can be different numbers of rows, but all of the arrays must have the same number of columns.
  - axis : int, optional. The axis along which the arrays will be joined. Default is 0 (rows). 1 selects columns.

▸ View examples

- `np.vstack()` Stack arrays in sequence vertically (row wise). Parameters:
  - Arrays to concatenate. The arrays must have the same shape along all but the first axis (rows). 1D arrays must have the same length.

▸ View example

- `np.hstack()` Stack arrays in sequence horizontally (column wise). Parameters
  - Arrays to concatenate. The arrays must have the same shape along all but the second axis (columns), except 1D arrays which can be any length.

▸ View example

默认：axis in 0 row

```
In [4]: a2d=a2d.reshape(4,3)
        a2d

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

```
In [5]: b2d=np.array([[1,2,3],[4,5,6]])
        b2d

Out[5]: array([[1, 2, 3],
               [4, 5, 6]])
```

```
In [7]: ab=np.concatenate([a2d,b2d])
        ab

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

---

axis in 1 col

axis =1

```
In [9]: b2d

Out[9]: array([[1, 2, 3],
               [4, 5, 6]])
```

```
In [8]: #以列的方式加入。
        abb=np.concatenate([b2d,b2d],axis=1)
        abb

Out[8]: array([[1, 2, 3, 1, 2, 3],
               [4, 5, 6, 4, 5, 6]])
```

```
In [ ]:
```

若以列来concatenate，则看行数row是否一致

若以行来concatenate，则看列数col是否一致。

Out [31]:

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-108-39705cdff0a6> in <module>
      3 v1 =np.array([[1, 2],[3,4]])
      4 matrix = np.array([[4, 5, 6],[7, 8, 9]])
----> 5 np.concatenate([matrix, v1])

ValueError: all the input array dimensions except for the concatenation axis must match exactly
```

In [32]:

<

```
# But you could concatenate on columns (both v1 and matrix have two rows)
np.concatenate([matrix, v1],axis=1)
```

Out[32]:

```
array([[4, 5, 6, 1, 2],
       [7, 8, 9, 3, 4]])
```

vstack

合并行row

与concatenate的不同：vstack能联合一维与二维数组。

使用concatenate时的问题

vstack

```
In [17]: a1d=np.array([[1,2,3],[1,2,3]])
         a1dd=np.array([[4,5,6],[7,8,9]])
         con=np.concatenate([a1d,a1dd])
         con
```

```
Out[17]: array([[1, 2, 3],
                [1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In [18]: a1d=np.array([1,2,3]) #一维
         a1dd=np.array([[4,5,6],[7,8,9]])#二维，不能直接concatenate
         con=np.concatenate([a1d,a1dd])
         con
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[18], line 3
      1 a1d=np.array([1,2,3])
      2 a1dd=np.array([[4,5,6],[7,8,9]])
----> 3 con=np.concatenate([a1d,a1dd])
      4 con

ValueError: all the input arrays must have same number of dimensions, but the array at index 0 has 1 dimension(s) and the array
at index 1 has 2 dimension(s)
```

一维与二维数组的concatenate

```
In [19]: np.vstack([a1d,a1dd])
```

```
Out[19]: array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

```
In [20]: np.vstack([a1dd,a1d])
```

```
Out[20]: array([[4, 5, 6],
                [7, 8, 9],
                [1, 2, 3]])
```

hstack

合并列col。

```
In [21]: matrix=np.array([[1,2,3],[4,5,6]])
         matrix
Out[21]: array([[1, 2, 3],
                [4, 5, 6]])

In [22]: m2=np.array([[1],[2]])
         m2
Out[22]: array([[1],
                [2]])

In [24]: np.hstack([matrix,m2])
Out[24]: array([[1, 2, 3, 1],
                [4, 5, 6, 2]])
```

Split

一维数组

- **等分**

```
In [32]: a1=np.arange(6)
         a1
Out[32]: array([0, 1, 2, 3, 4, 5])

In [35]: #等分
         b1,b2=np.split(a1,2)

In [36]: print(b1,b2)
         [0 1 2] [3 4 5]
```

- **多段分割**

多段分割，传入的是index的断点位置。

In [41]:
```
a1
```

Out[41]:
```
array([0, 1, 2, 3, 4, 5])
```

In [40]:
```
#多段分割
#[a,b]，a指示第一个分割在哪个index前停止，b指示第二个分割段在哪个index前停止
a,b,c=np.split(a1,[2,5])
print(a,b,c)
```

```
[0 1] [2 3 4] [5]
```

In [42]:
```
a2=np.arange(12)
a2
```

Out[42]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [44]:
```
a,b,c,d,e=np.split(a2,[2,5,8,10])
print(a,b,c,d,e)
```

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

二维数组

- **分割**

二维数组

In [57]:
```
m1=np.array([[1,2,3,4],[3,4,5,6],[5,6,7,8],[6,7,8,9]])
m1
```

Out[57]:
```
array([[1, 2, 3, 4],
       [3, 4, 5, 6],
       [5, 6, 7, 8],
       [6, 7, 8, 9]])
```

In [61]:
```
m11,m22=np.split(m1,[1])#split the first row
```

In [62]:
```
m11
```

Out[62]:
```
array([[1, 2, 3, 4]])
```

In [63]:
```
m22
```

Out[63]:
```
array([[3, 4, 5, 6],
       [5, 6, 7, 8],
       [6, 7, 8, 9]])
```

```
Out[57]: array([[1, 2, 3, 4],
                [3, 4, 5, 6],
                [5, 6, 7, 8],
                [6, 7, 8, 9]])
```

```
In [71]: m11,m22,m33=np.split(m1,[1,2])#split the first row
```

```
In [72]: m11
Out[72]: array([[1, 2, 3, 4]])
```

```
In [73]: m22
Out[73]: array([[3, 4, 5, 6]])
```

```
In [74]: m33
Out[74]: array([[5, 6, 7, 8],
                [6, 7, 8, 9]])
```

从末尾处，开始分割

```
In [67]: m11,m22=np.split(m1,[-1])#split the first row
```

```
In [68]: m11
Out[68]: array([[1, 2, 3, 4],
                [3, 4, 5, 6],
                [5, 6, 7, 8]])
```

```
In [69]: m22
Out[69]: array([[6, 7, 8, 9]])
```

- **等分**

```
In [64]: m111,m222=np.split(m1,2)#以行为基础，分割为2份

In [65]: m111
Out[65]: array([[1, 2, 3, 4],
                [3, 4, 5, 6]])

In [66]: m222
Out[66]: array([[5, 6, 7, 8],
                [6, 7, 8, 9]])
```

vsplit

与split一样，不过只对row进行split.

```
In [75]: m1
Out[75]: array([[1, 2, 3, 4],
                [3, 4, 5, 6],
                [5, 6, 7, 8],
                [6, 7, 8, 9]])

In [76]: m11,m22=np.vsplit(m1,2)

In [77]: m11,m22
Out[77]: (array([[1, 2, 3, 4],
                 [3, 4, 5, 6]]),
          array([[5, 6, 7, 8],
                 [6, 7, 8, 9]]))
```

hsplit

In [78]:
```
m1
```

Out[78]:
```
array([[1, 2, 3, 4],
       [3, 4, 5, 6],
       [5, 6, 7, 8],
       [6, 7, 8, 9]])
```

In [86]:
```
h1,h2=np.hsplit(m1,[1])
```

In [87]:
```
h1,h2
```

Out[87]:
```
(array([[1],
        [3],
        [5],
        [6]]),
 array([[2, 3, 4],
        [4, 5, 6],
        [6, 7, 8],
        [7, 8, 9]]))
```

```
In [78]: m1

Out[78]: array([[1, 2, 3, 4],
                [3, 4, 5, 6],
                [5, 6, 7, 8],
                [6, 7, 8, 9]])

In [88]: h1,h2=np.hsplit(m1,2)

In [89]: h1,h2

Out[89]: (array([[1, 2],
                 [3, 4],
                 [5, 6],
                 [6, 7]]),
          array([[3, 4],
                 [5, 6],
                 [7, 8],
                 [8, 9]]))
```

Copy

数组是易变的。

Copy arrays

Arrays present the same problem as lists when you assign an array to another array. If you modify either of the arrays the other array is modified as well. This behaviour occurs as NumPy arrays are mutable objects. This can be avoided by using the `copy()` method or `np.array()`, which will give you a new array that has the same values.

In [45]:

```
# ERROR - Copy an array in another array WITHOUT copy()
a2 =np.array([[3, 12, 5, 65],[13, 90, 2, 49], [35, 79, 1, 8]])
print('a2 = \n',a2)

a2Slice= a2[:2,:2]
print('\n a2Slice = \n',a2Slice)

# What happens if we modified a position of the array?
a2Slice[0, 0] = -10
print('\n a2Slice after modified= \n',a2Slice)
print('\n a2 is modified too! =\n', a2)
```

```
a2 =
 [[ 3 12  5 65]
 [13 90  2 49]
 [35 79  1  8]]

 a2Slice =
 [[ 3 12]
 [13 90]]

 a2Slice after modified=
 [[-10  12]
 [ 13  90]]

 a2 is modified too! =
 [[-10  12   5  65]
 [ 13  90   2  49]
 [ 35  79   1   8]]
```

个人例子

```
In [90]: m1
```

```
Out[90]: array([[1, 2, 3, 4],
                [3, 4, 5, 6],
                [5, 6, 7, 8],
                [6, 7, 8, 9]])
```

```
In [91]: mm=m1[:2,:2]
         mm
```

```
Out[91]: array([[1, 2],
                [3, 4]])
```

```
In [92]: mm[0,0]=-100
         mm
```

```
Out[92]: array([[-100,    2],
                [   3,    4]])
```

```
In [93]: m1
```

```
Out[93]: array([[-100,    2,    3,    4],
                [   3,    4,    5,    6],
                [   5,    6,    7,    8],
                [   6,    7,    8,    9]])
```

使用了copy之后，这复制后的数组就是独立的个体了。

```
In [94]: mm=m1[:2,:2].copy()
         mm
```

```
Out[94]: array([[-100,    2],
                [   3,    4]])
```

```
In [96]: mm[0,0]=1000
         mm,m1
```

```
Out[96]: (array([[1000,    2],
                 [   3,    4]]),
          array([[-100,    2,    3,    4],
                 [   3,    4,    5,    6],
                 [   5,    6,    7,    8],
                 [   6,    7,    8,    9]]))
```

其他使用情况

何获得任意维度数组中的第一个元素?

使用ravel()

要在任意维度的 NumPy 数组中获取第一个元素，无论数组的维度是多少，都可以使用 `numpy.ravel()` 函数将数组展平成一维，然后获取第一个元素。`numpy.ravel()` 不会改变原始数组的形状，只是返回展平后的视图或副本。

代码示例如下：

```python
import numpy as np

# 创建一个任意维度的 NumPy 数组
arr = np.array([[1, 2, 3], [4, 5, 6]])

# 使用 ravel() 展平数组并获取第一个元素
first_element = np.ravel(arr)[0]

print(first_element)
```

输出将是：

```
1
```

如何使得多维数组reshape成一维?

使用reshape(-1)或ravel。

```
In [60]: output18=np.sort(ab.ravel())[::-1]
         output18
Out[60]: array([11,  9,  7,  7,  6,  5,  5,  4,  3,  3,  2,  1,  1,  0])
```

如何filtering with condition?

在slice的基础上，filtering

只不过是[ ]中，写上条件。

```
array
```

Out[21]:
```
array([ 7, 59, 42, 13, 67, 19, 26, 59, 99, 97, 77,  1, 36, 49, 10, 51, 41,
       73, 33, 79, 19, 34, 84, 11, 41, 75])
```

In [26]:
```
#numpy
aa=array[array %3==0] # divide evenly by 3
print(aa)
aa=aa/6
#series
series_7=pd.Series(aa)
series_7
```

```
[42 99 36 51 33 84 75]
```

Out[26]:
```
0     7.0
1    16.5
2     6.0
3     8.5
4     5.5
5    14.0
6    12.5
dtype: float64
```