CS112
Introduction to Python Programming

Session 06: NumPy Arrays

Shengwei Hou
Ph.D., Assistant Professor
Department of Ocean Science and Engineering
Fall 2022

SUSTech Southern University of Science and Technology

明德求是 日新自强
VIRTUE | TRUTH | ADVANCE

Contents

- Tuple review
- Set review
- NumPy array creation
- Array attributes
- Array placeholder content
- Indexing & slicing
- Boolean indexing
- Mathematical operations
- Stacking & splitting
- Broadcasting

01

Tuples & Sets

Tuples

- A tuple is a finite sequence of ordered, immutable, and heterogeneous items that are of fixed size enclosed in `()`.
- Tuples are immutable, individual items of a tuple are addressed in the same way as those of lists, but the elements cannot be changed
- The + operator can be used to concatenate tuples, and the * operator can be used to repeat a tuple
- The presence or absence of an item in a tuple can be tested using the `in` and `not in` membership operators
- Comparison operators like <, <=, >, >=, `==` and `!=` can also be used to compare tuples
- `sorted()` function returns ~~a sorted copy~~ of the tuple as a list while leaving the original tuple untouched
- If an item in a tuple is mutable (like a list), then this item can be changed.

Tuples

- Tuples can be used as `key:value` pairs to build dictionaries, which is achieved by nesting tuples within tuples
- The method `items()` in a dictionary returns `dict_items()` that can be converted to a list of tuples where each tuple corresponds to a `key:value` pair of the dictionary
- The `count()` method counts the number of times the item has occurred in the tuple and returns it
- The `index()` method searches for the given item from the start of the tuple and returns its first appearance index
- Tuple unpacking requires that there are as many variables on the left side of the equals sign as there are items in the tuple
- The `zip()` function returns a sequence of tuples, where the i-th tuple contains the i-th element from each of the iterables.

Sets

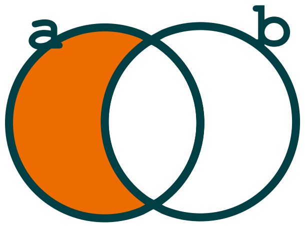
- A set is an unordered collection with no duplicate items
- Primary uses of sets include membership testing and eliminating duplicate entries
- Curly braces `{}` or the `set()` function can be used to create sets with a comma-separated list of items
- To create an empty set, you must use `set()` but NOT `{}`, as the latter creates an empty dictionary
- Indexing is not possible in sets, since set items are unordered
- The presence or absence of an item in a tuple can be tested using the `in` and `not in` membership operators
- Set operations (`-`, `|`, `&`, `^`) can also be done with set methods
- A frozenset is basically the same as a set, except that it is immutable

Sets

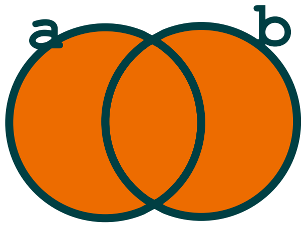
SUSTech Southern University of Science and Technology

```
>>> a =
set('abracadabra')
>>> a
{'a', 'c', 'd', 'b',
'r'}
>>> b = set('alacazam')
>>> b
{'a', 'm', 'c', 'z',
'l'}

>>> a - b
{'r', 'd', 'b'}
>>> a | b
{'a', 'm', 'c', 'd',
'b', 'z', 'l', 'r'}
>>> a & b
{'a', 'c'}
>>> a ^ b
{'m', 'd', 'z', 'l',
'r', 'b'}
```

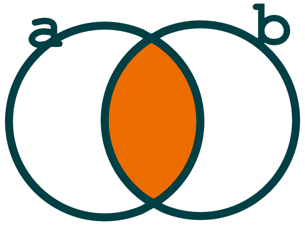
```
>>> set1 = {"a", "b", "e",
"f", "g"}
>>> set2 = {"a", "e", "c",
"d"}

>>> set2.difference(set1)
{'c', 'd'}

>>> set2.intersection(set1)
{'a', 'e'}

>>> set2.union(set1)
{'a', 'c', 'd', 'b', 'e', 'f',
'g'}

>>>
set2.symmetric_difference(set
1)
{'c', 'g', 'd', 'b', 'f'}
```



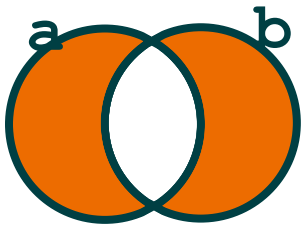
Difference
a-b



Union
a|b (A U B)



Intersection
a&b (A ∩ B)



Symmetric difference
a^b (A △ B)

02

NumPy Arrays

NumPy array

- The NumPy (Numerical Python) array is the real workhorse of data structures for scientific and engineering applications
- The NumPy array, formally called `ndarray` in NumPy documentation, is similar to a list but where all the elements should be of the same type (homogeneous)
- NumPy array can be used as a multi-dimensional container to store generic data
- The elements of a NumPy array are usually numbers, but can also be Booleans, strings, or other objects
- When the elements are numbers, they must all be of same type, e.g., they might be all integers or all floating numbers
- Each dimension of an array has a length which is the total number of elements in that direction. The size of an array is the total number of elements contained in an array
- The size of NumPy arrays are fixed; once created it cannot be changed again

Array creation

- The `array` function can convert a list or tuple to a NumPy array:

```
import numpy as np
>>> a = [0, 0, 1, 4]
>>> b = np.array(a)
>>> b
array([0, 0, 1, 4])
>>> c = (0, 0, 1, 4)
>>> d = np.array(c)
>>> d
array([0, 0, 1, 4])
>>> type(b)
<class 'numpy.ndarray'>
>>> d.dtype
dtype('int32’)  # or dtype(‘int64’), depending on your operating system architecture
```

`b` and `d` are integer arrays, as they are created from lists of integers

Array creation

SUSTech Southern University of Science and Technology

- The `array` function can convert a list or tuple to an array:

```
>>> e = np.array([1, 4., -2, 7])
>>> e
array([ 1.,  4., -2.,  7.])
>>> type(e)
<class 'numpy.ndarray'>
>>> e.dtype
dtype('float64')
```

- `e` is a floating point array even though only one of the elements of the list from which it was made was a floating point number
- The array function automatically promotes all the numbers to the type of the most general entry in the list

Array creation

- The `array` function, converts a list to an array:

```
>>> f = ["abc", 1.2, 2]
>>> g = np.array(f)
>>> g
array(['abc', '1.2', '2'], dtype='<U32')
>>> type(g)
<class 'numpy.ndarray'>
>>> g.dtype
dtype('<U32')  # a little-endian 32 character Unicode string.
```

- Endianness (字节顺序，又称端序) is the order or sequence of bytes of a word of digital data in computer memory. Endianness is primarily expressed as big-endian (BE, denoted as ‘>’) or little-endian (LE, denoted as‘<’).
- A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest. A little-endian system, in contrast, stores the least-significant byte at the smallest address.

When the elements of a list are made up of numbers and strings, all the elements become strings when an array is formed from the list

Array creation

- The `array` function, converts a list to an array:

```
>>> a2d = np.array([[1,2,3], [4,5,6]])
>>> a2d
array([[1, 2, 3],
       [4, 5, 6]])
>>> b2d = np.array(((1,2,3), (4,5,6)))
>>> b2d
array([[1, 2, 3],
       [4, 5, 6]])
>>> afloat = np.array([1,2,3,4], dtype = np.float64)
>>> afloat
array([1., 2., 3., 4.])
>>> afloat.dtype
dtype('float64')
```

Array attributes

SUSTech Southern University of Science and Technology

```
>>> arr_att = np.array([[10, 20, 30], [14, 12, 16]])
>>> arr_att.ndim                        # Number of array dimensions
2
>>> arr_att.shape                       # Tuple of array dimensions
(2, 3)            2 by 3 matrix
>>> arr_att.size                        # Number of elements in the array
6
>>> arr_att.dtype                       # Data-type of the array’s elements
dtype('int32’)   # or dtype(‘int64’)
>>> arr_att.itemsize                    # Length of one array element in bytes
4                # or 8
```

Array placeholder content

- Using the NumPy `zeros` and `ones` function to create arrays where all the elements are either zeros or ones
- They each take one mandatory argument, the number of elements in the array, and one optional argument that specifies the data type of the array. If unspecified, the data type is a float:

```
>>> np.zeros(6)
array([0., 0., 0., 0., 0., 0.])
>>> np.ones(5)
array([1., 1., 1., 1., 1.])
>>> np.ones(5, dtype=int)
array([1, 1, 1, 1, 1])
>>> np.zeros((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
Docstring:
zeros(shape, dtype=float, order='C', *, like=None)

Return a new array of given shape and type, filled with zeros.

Parameters
----------
shape : int or tuple of ints
    Shape of the new array, e.g., ``(2, 3)`` or ``2``.
dtype : data-type, optional
    The desired data-type for the array, e.g., `numpy.int8`.  Default is
    `numpy.float64`.
order : {'C', 'F'}, optional, default: 'C'
    Whether to store multi-dimensional data in row-major
    (C-style) or column-major (Fortran-style) order in
    memory.
```

Array placeholder content

SUSTech Southern University of Science and Technology

```
>>> np.empty((2,3))
array([[0., 0., 0.],
       [0., 0., 0.]])
>>> np.full((3,3),2)
array([[2, 2, 2],
       [2, 2, 2],
       [2, 2, 2]])
>>> np.eye(2,2)
array([[1., 0.],
       [0., 1.]])
>>> np.identity(2)
array([[1., 0.],
       [0., 1.]])
>>> np.random.random((2,2))    # size=(2,2)
array([[0.6, 0.4],
       [0.1, 0.2]])
```

```
Signature: np.full(shape, fill_value, dtype=None, order='C', *, like=None)
Docstring: Return a new array of given shape and type, filled with `fill_value`.

Parameters
----------
shape : int or sequence of ints
    Shape of the new array, e.g., ``(2, 3)`` or ``2``.
fill_value : scalar or array_like
    Fill value.
```

```
Signature: np.eye(N, M=None, k=0, dtype=<class 'float'>, order='C', *, like=None)
Docstring: Return a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters
----------
N : int
  Number of rows in the output.
M : int, optional
  Number of columns in the output. If None, defaults to `N`.
```

```
Signature: np.identity(n, dtype=None, *, like=None)
Docstring: Return the identity array.

The identity array is a square array with ones on
the main diagonal.

Parameters
----------
n : int
    Number of rows (and columns) in `n` x `n` output.
dtype : data-type, optional
    Data-type of the output.  Defaults to ``float``.
```

Array placeholder content

SUSTech Southern University of Science and Technology

- Using the NumPy `linspace` or `logspace` functions
- The `linspace` function creates an array of N (num) evenly spaced points between a starting point and an ending point. The form of the function is `linspace(start, stop, num)`. If the third argument num is omitted, then num=50:

```
>>> np.linspace(0, 10, 5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
```

- The `logspace` function produces evenly spaced points on a logarithmically spaced scale. The form of the function is `logspace(start, stop, num)`. The start and stop refer to a power of 10, i.e., the array starts at 10^{start} and ends at 10^{stop}:

```
>>> np.set_printoptions(precision=1)
>>> np.logspace(1, 3, 5)
array([  10. ,   31.6,  100. ,  316.2, 1000. ])
```

Array placeholder content

- Using the NumPy `arange` function
- The form of the function is `arange(start, stop, step)`. If the third argument is omitted, ~~`step=1`~~. If the first argument is omitted, then ~~`start=0`~~:

```
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])
>>> np.arange(0., 10, 2)
array([0., 2., 4., 6., 8.])
>>> np.arange(0, 10, 1.5)
array([0. , 1.5, 3. , 4.5, 6. , 7.5, 9. ])
```

- The `arange` function produces points evenly spaced between 0 and 10 exclusive of the final point
- In general `arange` produces an integer array if the arguments are all integers; if any one of the arguments is a float, the generated array would be a float

Array indexing & slicing

- One-dimensional arrays can be indexed and sliced the same way as strings and lists, i.e., array indexes are 0-based:

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> a[2]
2
>>> a[2:4]
array([2, 3])
>>> a[:4:2]
array([0, 2])
>>> a[:4:2] = -9
>>> a
array([-9,  1, -9,  3,  4])
>>> a[::-1]
array([ 4,  3, -9,  1, -9])
```

Array indexing & slicing

- Multi-dimensional arrays can be indexed and sliced per axis:

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

`a[][]` also works for NumPy array

```
>>> lower_axes = a[1, :]
>>> lower_axes
array([5, 6, 7, 8])
>>> lower_axes.ndim
1
>>> same_axes = a[1:2, :]
>>> same_axes
array([[5, 6, 7, 8]])
>>> same_axes.ndim
2
>>> a[:, 1]
array([ 2,  6, 10])
>>> a[:, 1:2]
array([[ 2],
       [ 6],
       [10]])
```

list in list

→ dim=2

Array indexing & slicing

SUSTech Southern University of Science and Technology

```
>>> for row in a:
...     print(row)
...
[1 2 3 4]
[5 6 7 8]
[ 9 10 11 12]
>>> a = np.array([[1, 2], [3, 4], [5, 6]])
>>> a
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> a[[0, 1, 2], [0, 1, 0]]
array([1, 4, 5]) # (0, 0), (1, 1), (2, 0)
```

```
>>> for each_element in a.flat:
...     print(each_element)
...
1
2
3
4
5
6
7
8
9
10
11
12
```

Boolean indexing

SUSTech Southern University of Science and Technology

```
>>> a = np.linspace(-1., 5, 5)
>>> a
array([-1. ,  0.5,  2. ,  3.5,  5. ])
>>> a[a>1.]
array([2. , 3.5, 5. ])
>>> a>1.
array([False, False,  True,  True,  True])
>>> a[a>1.] = 1.
>>> a
array([-1. ,  0.5,  1. ,  1. ,  1. ])
>>> a.size
5
>>> b = np.linspace(0., 10, a.size)
>>> b
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
>>> b[a==1] = 3
>>> b
array([0. , 2.5, 3. , 3. , 3. ])
>>> a==1
array([False, False,  True,  True,  True])
```

Boolean indexing is very useful for reassigning values of an array that meet some criteria

Mathematical operations

- Basic mathematical functions perform element-wise operation on arrays

```
>>> a = np.linspace(-1., 5, 5)
>>> a
array([-1. ,  0.5,  2. ,  3.5,  5. ])
>>> a*6
array([-6.,  3., 12., 21., 30.])
>>> a/5
array([-0.2,  0.1,  0.4,  0.7,  1. ])
>>> a**3
array([ -1. ,   0.1,   8. ,  42.9, 125. ])
>>> a + 4
array([3. , 4.5, 6. , 7.5, 9. ])
>>> a - 10
array([-11. ,  -9.5,  -8. ,  -6.5,  -5. ])
>>> (a+3)*2
array([ 4.,  7., 10., 13., 16.])
>>> np.sin(a)
array([-0.8,  0.5,  0.9, -0.4, -1. ])
```

Mathematical operations

SUSTech Southern University of Science and Technology

```
>>> b = 5*np.ones(5)
>>> b
array([5., 5., 5., 5., 5.])
>>> b += 4
>>> b
array([9., 9., 9., 9., 9.])
>>> a = np.linspace(-1., 5, 7)
>>> a
array([-1.,  0.,  1.,  2.,  3.,  4.,  5.])
>>> np.log(a)
__main__:1: RuntimeWarning: divide by zero encountered in log
__main__:1: RuntimeWarning: invalid value encountered in log
array([ nan, -inf,  0. ,  0.7,  1.1,  1.4,  1.6])
```

Mathematical operations

SUSTech Southern University of Science and Technology

```
>>> a = np.array([34., -12, 5.])
>>> b = np.array([68., 5.0, 20.])
>>> a + b
array([102.,  -7.,  25.])
>>> a - b
array([-34., -17., -15.])
>>> a*b
array([2312.,  -60.,  100.])
>>> a/b
array([ 0.5, -2.4,  0.2])
```

```
np.add(a, b)
np.subtract(a, b)
np.multiply(a, b)
np.divide(a, b)
```

- These operations with arrays are called *vectorized* operations because the entire array, or "vector," is processed as a unit.
- ~~Vectorized~~ operations are much faster than processing each element of an array one by one.
- Writing code that takes advantage of these kinds of vectorized operations is almost always preferred to other means of accomplishing the same task

Speed test


```
#%% for loop
import numpy as np
import time
a = np.linspace(0, 32, 10000000) # 10 million
startTime = time.process_time()
for i in range(len(a)):
    a[i] = a[i]*a[i]
endTime = time.process_time()
print(f"Run time = {endTime-startTime} seconds")
```


```
import numpy as np
import time
a = np.linspace(0, 32, 10000000) # 10 million
startTime = time.process_time()
for i in range(len(a)):
    a[i] = a[i]*a[i]
endTime = time.process_time()
print(f"Run time = {endTime-startTime} seconds")
```


```
Run time = 2.77072 seconds
```

```
#%% vectorized operations
import numpy as np
import time
a = np.linspace(0, 32, 10000000) # 10 million
startTime = time.process_time()
a = a*a
endTime = time.process_time()
print(f"Run time = {endTime-startTime} seconds")
```


```
import numpy as np
import time
a = np.linspace(0, 32, 10000000) # 10 million
startTime = time.process_time()
a = a*a
endTime = time.process_time()
print(f"Run time = {endTime-startTime} seconds")
```


```
Run time = 0.007834000000000785 seconds
```

Mathematical operations

```
>>> a = np.array( [20, 30, 40, 50] )
>>> np.sin(a)
array([ 0.9, -1. ,  0.7, -0.3])
>>> np.cos(a)
array([ 0.4,  0.2, -0.7,  1. ])
>>> np.tan(a)  # tangent: np.sin(a)/np.cos(a)
array([ 2.2, -6.4, -1.1, -0.3])
>>> a = np.array([-1.7, -1.5, -0.2, 0.2])
>>> np.floor(a)
array([-2., -2., -1.,  0.])
>>> np.ceil(a)
array([-1., -1., -0.,  1.])
```

```
>>> np.sqrt([1,4,9])
array([1., 2., 3.])
>>> np.max([[2, 3, 4], [1, 5, 2]])
5
>>> np.min([[2, 3, 4], [1, 5, 2]])
1
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
```

0	1	1
0	5	5
0	6	

Axes in NumPy are defined for arrays with more than one dimension.
A 2-dimensional array has two corresponding axes: the first running vertically downwards across rows (axis=0), and the second running horizontally across columns (axis=1).

Stacking & splitting


```
>>> a = np.array([[3, 1, 2], [8, 7, 9]])
>>> b = np.array([[2, 4, 6], [5, 4, 8]])
>>> np.vstack((a, b))
array([[3, 1, 2],
       [8, 7, 9],
       [2, 4, 6],
       [5, 4, 8]])
>>> np.hstack((a, b))
array([[3, 1, 2, 2, 4, 6],
       [8, 7, 9, 5, 4, 8]])
>>> c = np.hstack((a, b))
>>> np.hsplit(c, 3)
[array([[3, 1],
       [8, 7]]),
 array([[2, 2],
       [9, 5]]),
 array([[4, 6],
       [4, 8]])]
>>> np.vsplit(c, 2)
[array([[3, 1, 2, 2, 4, 6]]),
 array([[8, 7, 9, 5, 4, 8]])]
```

3	1	2
8	7	9

2	4	6
5	4	8

3	1	2
8	7	9
2	4	6
5	4	8

3	1	2	2	4	6
8	7	9	5	4	8

3	1
8	7

2	2
9	5

4	6
4	8

3	1	2	2	4	6

8	7	9	5	4	8

Shape & reshape


```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[0., 8., 9., 1.],
       [2., 9., 0., 6.],
       [9., 4., 1., 6.]])
>>> a.shape
(3, 4)
>>> a.ravel()
array([0., 8., 9., 1., 2., 9., 0., 6., 9., 4., 1., 6.])
>>> a.reshape(4,3)
array([[0., 8., 9.],
       [1., 2., 9.],
       [0., 6., 9.],
       [4., 1., 6.]])
```

Signature: np.ravel(a, order='C')
Docstring:
Return a contiguous flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

Signature: np.reshape(a, newshape, order='C')
Docstring:
Gives a new shape to an array without changing its data.

Parameters
----
a : array_like
    Array to be reshaped.
newshape : int or tuple of ints
    The new shape 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.

Broadcasting

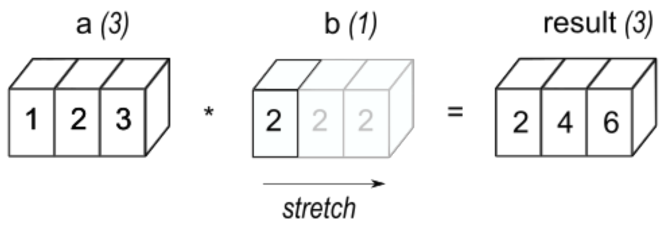
SUSTech Southern University of Science and Technology

- NumPy operations are usually done on pairs of arrays on an element-by-element basis.

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([2.,  4.,  6.])
```

- Broadcasting rule relaxes this constraint when the arrays’ shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([2.,  4.,  6.])
```



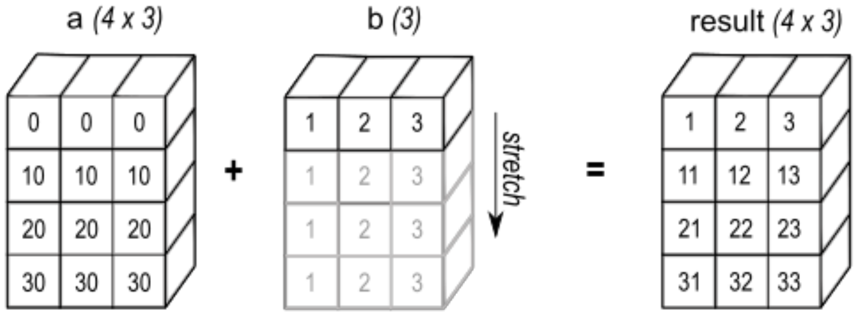
The stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies so that broadcasting operations are as memory and computationally efficient as possible.

Broadcasting

SUSTech Southern University of Science and Technology

- The term broadcasting describes how NumPy treats arrays with different shapes during arithmetic operations:

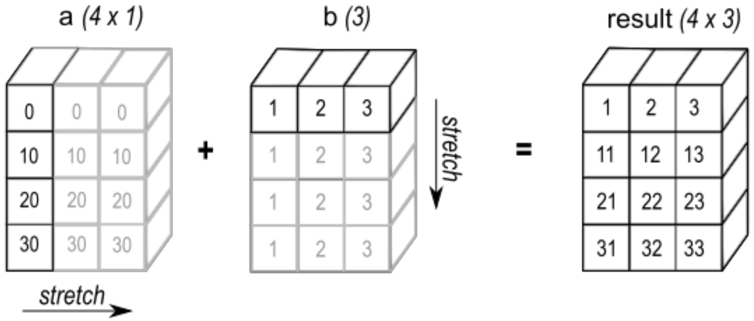
```
>>> a = np.array([[ 0.0,  0.0,  0.0],
                  [10.0, 10.0, 10.0],
                  [20.0, 20.0, 20.0],
                  [30.0, 30.0, 30.0]])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a + b
array([[  1.,   2.,   3.],
       [11.,  12.,  13.],
       [21.,  22.,  23.],
       [31.,  32.,  33.]])
```



A one dimensional array added to a two dimensional array results in broadcasting if number of 1-d array elements matches the number of 2-d array columns.

```
>>> array_1 = np.ones([4, 5])
>>> array_2 = np.arange(5)
>>> array_1
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
>>> array_2
array([0, 1, 2, 3, 4])
>>> array_1 + array_2
array([[1., 2., 3., 4., 5.],
       [1., 2., 3., 4., 5.],
       [1., 2., 3., 4., 5.],
       [1., 2., 3., 4., 5.]])
```

- Broadcasting allows NumPy functions to deal in a meaningful way with input arrays that do not have exactly the same shape
- Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes
- The original array is not affected



Broadcasting

- **Rule 1** → If two input arrays do not have the same number of dimensions, a copy of the smaller array will be repeatedly padded so both the arrays have the same number of dimensions
- **Rule 2** → If the shape of two input arrays does not match, then the array with a shape of "1" along a particular dimension is stretched to match the shape of the array having the largest shape along that dimension. The value of the array element is assumed to be the same along that dimension for the "broadcast" array. After application of the broadcasting rules, the sizes of all arrays must match
- **Rule 3** → If the above two rules are not met, a `ValueError` exception is thrown, indicating that the arrays have incompatible shapes

Broadcasting

SUSTech Southern University of Science and Technology

```
>>> array_1 = np.random.random(4).reshape([4,1])
>>> array_2 = np.arange(4)
>>> array_1
array([[0.46136448],
       [0.9978799 ],
       [0.48440598],
       [0.16054045]])
>>> array_2
array([0, 1, 2, 3])
>>> array_1 + array_2
array([[0.46136448, 1.46136448, 2.46136448, 3.46136448],
       [0.9978799 , 1.9978799 , 2.9978799 , 3.9978799 ],
       [0.48440598, 1.48440598, 2.48440598, 3.48440598],
       [0.16054045, 1.16054045, 2.16054045, 3.16054045]])
```

Broadcasting

```
>>> array_1 = np.random.random([2, 3])
>>> array_2 = np.ones(5)
>>> array_1.shape
(2, 3)
>>> array_2.shape
(5,)
>>> array_1 + array_2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (2,3)
(5,)
```

Array & List

SUSTech Southern University of Science and Technology

- Lists are part of the core Python programming language; arrays are a part of the numerical computing package NumPy
- The elements of a NumPy array must all be of the same type, whereas the elements of a Python list can be of completely different types
- Arrays allow Boolean indexing; lists do not
- NumPy arrays support “vectorized” operations like element-by-element addition and multiplication
- Adding one or more additional elements to a NumPy array creates a new array and destroys the old one. Therefore, it can be very inefficient to build up large arrays by appending elements one by one. By contrast, elements can be added to a list without creating a whole new list

Exercise

Exercise 1

SUSTech Southern University of Science and Technology

- Create a 10 x 10 arrays of zeros and then "frame" it with a border of ones:

Exercise 2

SUSTech Southern University of Science and Technology

- Create a 5 × 5 array with its (i,j)-entry equal to $i+j$:

```
A=np.arange(5)
B=A.reshape([5,1])
C=A+B
```

Exercise 3

- Create a random 3×5 array using the `np.random.rand(3, 5)` function and compute: the sum of all the entries, the sum of the rows and the sum of the columns:

Exercise 4

- A magic square is a matrix all of whose row sums, column sums and the sums of the two diagonals are the same. Check if the following square matrix is a magic matrix:

```
A=np.array([[17, 24, 1, 8, 15],
[23, 5, 7, 14, 16],
[ 4, 6, 13, 20, 22],
[10, 12, 19, 21, 3],
[11, 18, 25, 2, 9]])
```

Exercise 5

- The following two arrays `y`, and `t` are respectively the position vs. time of a falling object, say a ball. Please calculate the average velocity as a function of time:

```
y = np.array([0., 1.3, 5. , 10.9, 18.9, 28.7, 40.])
t = np.array([0., 0.49, 1. , 1.5 , 2.08, 2.55, 3.2])
```