Numpy Basics

Import Numpy package

In [1]:
```
import numpy as np
```

Define a numpy ndarray

In [2]:
```
arr  = np.arange(1,6)
```

Numpy Array Output

In [3]:
```
arr
```

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

In [4]:
```
print(arr)
```

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

Ndarray Operation

for exmaple， list [1,2,3,4,5] for each element to add 1

- Use python list to add one for each element

In [5]:
```
# Method 1: Loop
lst = [1,2,3,4,5]
for i in range(len(lst)):
    lst[i] = lst[i] + 1
print(lst)
```

```
[2, 3, 4, 5, 6]
```

```
In [6]: # Python Expression
        lst1 = [1,2,3,4,5]
        lst1 = [i + 1 for i in lst1 ]
        print(lst1)
```

```
[2, 3, 4, 5, 6]
```

- Use numpy ndarray

```
In [7]: arr = np.arange(1,6)
        arr + 1
        arr
```

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

The reason we use Numpy for calculation is that Numpy is written by C programming language, the array operation will be more time efficient and optimized. For example, we calculate the sum of intergers up to 10000

- Use Python and magic function "%timeit" to see the average time of 7 runs of calculating the sum of intergers up to 10000

```
In [8]: a = list(range(10000))
        %timeit sum(a)
```

```
56.6 µs ± 368 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

-Use Numpy to Calculate the sum of intergers up to 10000

```
In [9]: b = np.array(a)
        %timeit np.sum(b)
```

```
5.15 µs ± 52.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Ndarry object

ndarray = n dimensional array.

The ndarray object is a multi-dimensional array used to store elements of the same type. Each element in the ndarray has a memory area with the same storage size.

One dimensional array：[1,2,3,4,5] (Vector)

Two dimensional array：[[1,2,3], [4,5,6]] （Matrix）

Three dimensioanl array：[[[1,2,3], [4,5,6]], [[7,8,9], [10,11,12]]] (Tensor)

Generate ndarray

Generate a one-dimensional array

```
In [10]: import numpy as np
         a1 = [1,2,3,4,5]
         arr1 = np.array(a1)
         arr1
```

```
Out[10]: array([1, 2, 3, 4, 5])
```

Generate a two-dimensional array (A matrix)

```
In [11]: a2 = [ [1,2,3],
             [4,5,6]
              ]
         arr2 = np.array(a2)
         arr2
```

```
Out[11]: array([[1, 2, 3],
                [4, 5, 6]])
```

Generate a three-dimensional array (A tensor)

```
In [12]: a3 = [
             [[1,2,3],
             [4,5,6]],
               [[7,8,9],
                [10,11,12]]
         ]
         arr3 = np.array(a3)
         arr3
```

```
Out[12]: array([[[ 1,  2,  3],
                 [ 4,  5,  6]],

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

Property of ndarray

Property Name	Explanation
ndarray.shape	the shape of a ndarray
ndarray.ndim	the dimension of a ndarray
ndarray.size	the number of elements in a ndarray
ndarray.itemsize	ndarray the size of each element, the unit is Byte
ndarray.dtype	the type of each element

Look at the shape of a ndarray

```
In [13]: arr2.shape
```

```
Out[13]: (2, 3)
```

```
In [14]: arr3.shape
```

```
Out[14]: (2, 2, 3)
```

Look at the dimension of a ndarray

```
In [15]: arr2.ndim
```

```
Out[15]: 2
```

```
In [16]: arr3.ndim
```

```
Out[16]: 3
```

Look at the number of elements in a ndarray

```
In [17]: arr2.size
```

```
Out[17]: 6
```

```
In [18]: arr3.size
```

```
Out[18]: 12
```

Look at the size of each element

```
In [19]: arr2.itemsize
```

```
Out[19]: 4
```

```
In [20]: arr3.itemsize
```

```
Out[20]: 4
```

Look at the type of each element

```
In [21]: arr2.dtype
```

```
Out[21]: dtype('int32')
```

Ndarry element type

Type Name	Description	Abbreviation
np.bool	Bool Type (1 Byte)	“b”
np.int8\16\32\64	Signed 8\16\32\64-bit Integers	“i1、i2、i4、i8”
np.uint 8\16\32\64	Unsigned 8\16\32\64-bit Intergers	“u1、u2、u4、u8”
np.float16	Half-precision floating 16-bit	‘f2’
np.float32	Single-precision floating 32-bit	‘f4’

Type Name	Description	Abbreviation
np.float64	Double-precision floating 64-bit	‘f8’
np.complex64\128	Complex number，32/64-bit floating for real/imaginary part	‘C8、C16’
np.string_	String	‘S’
np.unicode	Unicode type	‘U’

Data Type

Compare types and size

In [22]:
```
# Set the ndarray data type as int8 (1 byte)
a4 = np.array([1, 2, 3, 4, 5], dtype = np.int8)
a4
```

Out[22]:
```
array([1, 2, 3, 4, 5], dtype=int8)
```

In [23]:
```
# Look at the size the element: 1 byte
a4.itemsize
```

Out[23]:
```
1
```

In [24]:
```
# Set the ndarry data type as float64 (8 bytes)
a5 = np.array([1,2,3,4,5], dtype = np.float64)
a5
```

Out[24]:
```
array([1., 2., 3., 4., 5.])
```

In [25]:
```
a5.itemsize
```

Out[25]:
```
8
```

In [26]:
```
lst = list("abcd")
a6 = np.array(lst)  # Unicode configration should be U type
print(a6.dtype, a6.itemsize)
```

```
<U1 4
```

```
In [27]: lst2 = list("abcd")           # np.string configration should be S type, 1 string in the list 1 byte
         a7 = np.array(lst, dtype = np.string_)
         print(a7.dtype, a7.itemsize)
```

```
|S1 1
```

```
In [28]: a8 = np.array(["SLM", "JPM", "BOA","CITI", "CAPITAL ONE"], dtype = np.string_) # string 11 byte because of len(Capital 
         print(a8.dtype, a8.itemsize)
```

```
|S11 11
```

Revise the type

```
In [29]: a9 = np.array([1.5, 2, 3], dtype = np.float64)
         a9.dtype, a9.itemsize
```

```
Out[29]: (dtype('float64'), 8)
```

Now revise it to int32, using numpy function "astype"

```
In [30]: a10 = a9.astype(np.int32)
         a10, a10.dtype, a10.itemsize
```

```
Out[30]: (array([1, 2, 3]), dtype('int32'), 4)
```

- astype(type)

type: provide specific data types that you would like me to modify.

> int32 --> float64: Space is sufficient, no problem
>
> float64 --> int32: The space is not sufficient, the decimal part will be truncated.
>
> string_ --> float64: If the string array represents only numbers, it is also possible to use astype to convert it to a numerical data type.

tolist() Covert the ndarray to Python list

```
In [31]: a11 = np.array([[1,2,3], [4,5,6]])
         lst4 = a11.tolist()
         lst4, type(lst4)  # See the type of a Python list

Out[31]: ([[1, 2, 3], [4, 5, 6]], list)
```

The data type is propagated downstream.

For ndarray, all elements inside must be of the same data type. If not, it will automaticly downward propagation, in the order int $\longrightarrow$ float $\longrightarrow$ str

```
In [32]: a12 = np.array([1,2,3,4,5])
         a12.dtype   # all intergers, data type should be intergers for all

Out[32]: dtype('int32')
```

```
In [33]: a13 = np.array([1, 2.5, 3, 4, 5])
         a13.dtype  # One element changed to 2.5 the float, all data type should be changed to float

Out[33]: dtype('float64')
```

```
In [34]: a14 = np.array([1, 2.5, 3, 4, 5, 'SLM'])
         a14.dtype    # add one more element string, all data type converted into unicode.

Out[34]: dtype('<U32')
```

a14*2 # UFuncTypeError, but if we converted a14 into np.object_, it can be multiplied and the data type is python object.

```
In [35]: a15 = np.array([1, 2.5, 3, 4, 5, 'SLM'], dtype = np.object_)
         a15*2
         a15, a15.dtype

Out[35]: (array([1, 2.5, 3, 4, 5, 'SLM'], dtype=object), dtype('O'))
```

Create Numpy Ndarray

Create numpy ndarray based on existing data

array

Syntax: array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

```
In [36]:
a16 = [1,2,3,4,5]
a17 = [[1,2,3]
      ,[4,5,6]]
```

```
In [37]:
arr4 = np.array(a16)
arr5 = np.array(a17)
print("arr4:", arr4)
print("arr5:", arr5)
```

```
arr4: [1 2 3 4 5]
arr5: [[1 2 3]
 [4 5 6]]
```

Through list, tuple and the mix of list tuple to create a ndarray

```
In [38]:
x =  [1,2,3,4]    # List
x =  (1,2,3,4)    # Tuple
x =  [(1,2,3),(4,5,6)] # Mix of List and tuple
a = np. array(x)
print (a,type(a))
```

```
[[1 2 3]
 [4 5 6]] <class 'numpy.ndarray'>
```

asarray

Syntax: asarray(a, dtype=None, order=None)

```
In [39]:
a = np. array([[1,2,3],[4,5,6]])
a1 = np. array(a)    # To generate a new array from an existing array
a2 = np. asarray(a)  # It does not actually create a new array, but rather acts as an index to the original a."
a2
```

```
Out[39]:
array([[1, 2, 3],
       [4, 5, 6]])
```

- Difference between array and asarray

Both array and asarray can convert structured data into an ndarray, but the main difference is that when the data source is already an ndarray, array will still make a copy and occupy new memory, whereas asarray will not.

In [40]:
```python
print('a1 is a?', a1 is a)
print('a2 is a?', a2 is a)
```

```
a1 is a? False
a2 is a? True
```

Create arrays based on their shape or values.

Function Name	Description
np.ones	To generate an array filled with ones
np.ones_like	To generate an array filled with ones that has the same shape as a given array
np.zeros	To generate an array filled with zeros
np.zeros_like	To generate an array filled with zeros that has the same shape as a given array
np.empty	To generate an empty array of a given shape without initializing its values
np.empty_like	To generate an empty array of the same shape as a given array without initializing its values
np.full	To generate an array of a specified shape, data type, and with a specific value
np.full_like	To generate an array of the same shape as a given array, but filled with a specific value
np.eye, np.identity	To generate an N x N identity matrix (i.e., a feature matrix with ones on the diagonal and zeros elsewhere)

Generate an array filled with ones

Syntax:

- ones(shape, dtype=None, order='C')
- ones_like(a, dtype=None, order='K', subok=True, shape=None)

In [41]:
```python
a = np.ones((3, 4),dtype = int)
a
```

```
Out[41]: array([[1, 1, 1, 1],
                [1, 1, 1, 1],
                [1, 1, 1, 1]])
```

```
In [42]: x = [[1, 2, 3], [4, 5, 6]]
         b = np.ones_like(x)
         b
```

```
Out[42]: array([[1, 1, 1],
                [1, 1, 1]])
```

Generate an array filled with zeros

Syntax:

- zeros(shape, dtype=float, order=‘C’)
- zeros_like(a, dtype=None, order='K', subok=True, shape=None)

```
In [43]: a = np.zeros((3,4), dtype = np.float64)
         a
```

```
Out[43]: array([[0., 0., 0., 0.],
                [0., 0., 0., 0.],
                [0., 0., 0., 0.]])
```

```
In [44]: x =  [[1, 2, 3], [4, 5, 6]]
         b = np.zeros_like(x)
         b
```

```
Out[44]: array([[0, 0, 0],
                [0, 0, 0]])
```

Generate an array of a specified shape, data type, and with a specific value

Syntax:

- full(shape, fill_value, dtype=None, order='C')
- full_like(a, fill_value, dtype=None, order='K', subok=True)

```
In [45]: a = np. full((3,4),7)
```

```
a
x = [[1,2,3],[4,5,6]]
a = np. full_like(x,7)
a
```

Out[45]:
```
array([[7, 7, 7],
       [7, 7, 7]])
```

Generate an empty array of a given shape without initializing its values

Syntax:

- empty(shape, dtype=float, order=‘C’)
- empty_like(prototype, dtype=None, order='K', subok=True, shape=None)

In [46]:
```
a = np. empty((3,4))
print(a, a.itemsize)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]] 8
```

In [47]:
```
x = [[1,2,3],[4,5,6]]
b = np. empty_like(x)
b #Note that the values in the array are not guaranteed to be 0 or any other specific value;
  # they are whatever happened to be in the memory locations that were allocated for the array.
```

Out[47]:
```
array([[-1686753424,         705,   923192876],
       [-2147481151,           0,           1]])
```

Generate an identity matrix

np.eye() and np.identity() difference

np.identity syntax: np.identity(n, dtype=None)

np.eye syntax: np.eye(N, M=None, k=0, dtype=<type ‘float'>)

- np.identity can only create square matrices.

- np.eye can create rectangular matrices, and the k parameter can be adjusted to shift the position of the diagonal of 1's. A k value of 0 places the diagonal in the center of the matrix, a k value of 1 shifts the diagonal one position up, a k value of 2 shifts the diagonal two positions up, and so on. Similarly, a k value of -1 shifts the diagonal one position down. If the absolute value of k is too large, the diagonal will shift completely out of the matrix, resulting in a matrix full of zeros.

In [48]:
```
a = np.identity(4)
a
```

Out[48]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

In [49]:
```
b = np.eye(3,4)
b
```

Out[49]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.]])
```

In [50]:
```
a = np.eye(3)
b = np.eye(3, k = 1)
print('No shift: \n', a)
print('One shift to the right: \n', b)
```

```
No shift: 
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
One shift to the right: 
 [[0. 1. 0.]
 [0. 0. 1.]
 [0. 0. 0.]]
```

In [51]:
```
c = np.eye(5,6, k = -2)
c
```

Out[51]:
```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.]])
```

Generate or extract a diagonal array

Syntax: diag(v, k=0)

If the input v is a one-dimensional array, the diag function returns a matrix with the input array as its diagonal elements.

If the input v is a two-dimensional matrix, the diag function returns a one-dimensional array containing the diagonal elements of the input matrix.

In [52]:
```
a = np.arange(1, 4)
b = np.arange(1, 10).reshape(3, 3)
print('a: \n', a)
print('np.diag(a): \n', np.diag(a))
print('b: \n', b)
print('np.diag(b): \n', np.diag(b))
```

```
a: 
 [1 2 3]
np.diag(a): 
 [[1 0 0]
 [0 2 0]
 [0 0 3]]
b: 
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
np.diag(b): 
 [1 5 9]
```

Create an array based on a numerical range

Generate a sequence of numbers within a specified range.

Syntax: np.arange(start, stop, step, dtype = None)

In [53]:
```
# Defining an arithmetic sequence with a starting value of 10, ending value of 20, and a step size of 2.
a = np. arange(10,21,2) # Left closed right open
a
```

```
Out[53]: array([10, 12, 14, 16, 18, 20])
```

Generate a arithmetic sequence.

Syntax: np.linspace(start, stop, num = 50, endpoint = True, retstep = False, dtype = None)

```
In [54]: a= np.linspace(0, 100, 11)
         a
```

```
Out[54]: array([  0.,  10.,  20.,  30.,  40.,  50.,  60.,  70.,  80.,  90., 100.])
```

Generate a geometric sequence.

Syntax: np.logspace(start, stop, num = 50, endpoint = True, base = 10.0, dtype = None)

```
In [55]: # Defining a geometric sequence based 10 (10^0, 10^1, 10^2)
         a = np. logspace(0, 2, 3)
         a
```

```
Out[55]: array([  1.,  10., 100.])
```

```
In [56]: # efining a geometric sequence based 2
         a = np.logspace(0, 9, 10, base = 2)
         a
```

```
Out[56]: array([  1.,   2.,   4.,   8.,  16.,  32.,  64., 128., 256., 512.])
```

Accessing array elements:

Use indexing or slicing.

For slicing, use the syntax obj[:, :], where the first colon represents the rows and the second colon represents the columns.

Both indexing and slicing start counting from zero.

Accessing Array Elements by Indexing

Numerical Indexing

```
In [57]: import numpy as np
         arr = np.arange(1,6)
         print(arr)
         # Print the second element
         arr[1]
```

```
[1 2 3 4 5]
Out[57]: 2
```

- Three-dimensional array indexing:

To index a specific element of a three-dimensional array, use the following syntax: arr[i, j, k], where i, j, and k are the indices of the first, second, and third dimensions, respectively.

```
In [58]: a1 = np.array([ [[1,2,3],[4,5,6]], [[12,3,34],[5,6,7]]])
         print(a1)
         # Get the element with first dimension, first row, second column element.
         a1[0, 0, 1]
```

```
[[[ 1  2  3]
  [ 4  5  6]]

 [[12  3 34]
  [ 5  6  7]]]
Out[58]: 2
```

Boolean indexing

It refers to the practice of selecting elements from an array based on a set of Boolean conditions. This is done by passing a Boolean array of the same shape as the original array to index it. Only the elements corresponding to True values in the Boolean array are selected.

```
In [59]: # 1-dimensionl array
         arr = np.arange(7)
         print(arr)
         booling1 = np.array([True, False, False, True, True, False, False])
```

```
# extract the elements in the target array "arr" that correspond
#to the positions of a boolean array "booling1" using boolean indexing.
# Get the element which in boolings equals to True
arr[booling1]
```

```
[0 1 2 3 4 5 6]
```

```
Out[59]: array([0, 3, 4])
```

```
In [60]: arr = np.arange(1,29).reshape((7,4))
         arr
```

```
Out[60]: array([[ 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]])
```

```
In [61]: # define a boolean array booling1 with True at indices 0, 3, and 4, and False elsewhere,
         # and use it to extract corresponding rows from the target array.
         booling1 = np.array([True, False, False, True, True, False, False])
         arr[booling1]
```

```
Out[61]: array([[ 1,  2,  3,  4],
                [13, 14, 15, 16],
                [17, 18, 19, 20]])
```

```
In [62]: arr = np.arange(1,29).reshape((7,4))
         arr
```

```
Out[62]: array([[ 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]])
```

```
In [63]: names = np.array(['A','B','A','C','D','E','A'])
         names == "A"
```

```
Out[63]: array([ True, False,  True, False, False, False,  True])
```

```
In [64]: arr[names == "A"]
```

```
Out[64]: array([[ 1,  2,  3,  4],
                [ 9, 10, 11, 12],
                [25, 26, 27, 28]])
```

```
In [65]: arr[names != "A"]
```

```
Out[65]: array([[ 5,  6,  7,  8],
                [13, 14, 15, 16],
                [17, 18, 19, 20],
                [21, 22, 23, 24]])
```

```
In [66]: # Find the element greater than 15
         # Method 1
         arr[arr>15]
```

```
Out[66]: array([16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28])
```

```
In [67]: # Method 2
         arr[np.where(arr>15)]
```

```
Out[67]: array([16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28])
```

Fancy indexing

Fancy indexing is a term used in NumPy to describe using integer arrays as indices, which means taking values from the target array based on the values of the index array as the index of a certain axis of the target array.

For using a one-dimensional integer array as an index, if the target is a one-dimensional array, the result of indexing is the corresponding element at the position; if the target is a two-dimensional array, then it is the row corresponding to the index.

```
In [68]: # 1D ndarray
         arr = np.array(['zero','one','two','three','four'])
         index = [1, 4]  # Define a list of indices.

         # Extract corresponding elements from an array based on a list of indices
         arr[index]
```

```
Out[68]: array(['one', 'four'], dtype='<U5')
```

```
In [69]: # 2D ndarray
arr = np.empty((8,4), dtype = np.int32)
for i in range(8):
    arr[i] = i
print(arr)

# Extract the elements from the 5th, 4th, 1st, and 7th rows
row = [4, 3, 0, 6]
arr[row]
```

```
[[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]]
```

```
Out[69]: array([[4, 4, 4, 4],
       [3, 3, 3, 3],
       [0, 0, 0, 0],
       [6, 6, 6, 6]])
```

```
In [70]: # Extract the elements from the 6th, 4th, and 2nd rows
# Same meaning of extract the 3rd, 4th, 7th in reverse order, so index can be negative numbers
arr[[-3,-4,-7]]
```

```
Out[70]: array([[5, 5, 5, 5],
       [4, 4, 4, 4],
       [1, 1, 1, 1]])
```

When two integer arrays are used as indices, the result is to retrieve the values of the corresponding indices along the corresponding axis in order. It is important to note that these two integer arrays should have the same shape or one of them should be a one-dimensional array of length 1 (related to NumPy's Broadcasting mechanism). For example, using [1,3,5] and [2,4,6] as integer arrays for indexing, for a two-dimensional array, the elements corresponding to the coordinates (1,2), (3,4), and (5,6) will be retrieved.

```
In [71]: arr = np.arange(42).reshape(6,7)
print(arr)
arr[[1,3,5],[2,4,6]]
```

```
[[ 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 32 33 34]
 [35 36 37 38 39 40 41]]
```

Out[71]:
```
array([ 9, 25, 41])
```

Slicing

Slicing in ndarray objects works similar to slicing in Python lists. The ndarray array can be indexed based on 0 to n, and the slice object can be created using the built-in slice function by setting the start, stop, and step parameters to extract a new array from the original array.

1D ndarray slicing

In [72]:
```python
arr = np.arange(1,6)
print(arr)
# Get the 3rd to 5th elements.
arr[2:4]
```

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

Out[72]:
```
array([3, 4])
```

In [73]:
```python
# To get the last two elements
arr[-2:]
# A colon before the index represents the starting position,
# while a colon after represents the ending index.
# When no specific value is given, it represents all elements.
```

Out[73]:
```
array([4, 5])
```

2D ndarray slicing

In [74]:
```python
arr = np.arange(1,21).reshape(4,5)
print(arr)
```

```
# Get all the elments on the third row
arr[2,:]
```

```
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

```
Out[74]: array([11, 12, 13, 14, 15])
```

```
In [75]: # Get all the element for the 1st column
arr[:, 0]
```

```
Out[75]: array([ 1,  6, 11, 16])
```

```
In [76]: #First row, the 2nd column to the 4th column
arr[0, 1:4]
```

```
Out[76]: array([2, 3, 4])
```

- Replace the elements in x with the elements from y at corresponding positions.

```
In [77]: x = np.array([1, 2, 3, 4, 5])
y = np.array([6, 7, 8])
x, y  # output should be like [6,7,8,4,5]
```

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

```
In [78]: x[0 :len(y)] = y
x
```

```
Out[78]: array([6, 7, 8, 4, 5])
```

Iterate an array

```
In [79]: import numpy as np
a = np.arange(12).reshape(3,4)
a
```

```
Out[79]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])
```

Iterate the rows

```
In [80]: for row in a:
             print("Rows are like: ", row)
```

```
Rows are like:  [0 1 2 3]
Rows are like:  [4 5 6 7]
Rows are like:  [ 8  9 10 11]
```

Iterate the columns

```
In [81]: b = a.transpose()
         b
         for column in b:
             print("Columns are like", column)
```

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

Iterate each element

Use flat property

The flat property is an element iterator for arrays, which can be used to iterate over all the elements of a multi-dimensional array as if it were a one-dimensional array. However, it can only be iterated over once.

```
In [82]: a = np.arange(12).reshape(3,4)
         b = a.flat  #flat is a property, iterator can only be looped once
         print(a,b)

         for item in b:
             print(item, end = ",")
         print()
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]] <numpy.flatiter object at 0x000002C1AE746350>
0,1,2,3,4,5,6,7,8,9,10,11,
```

In [83]:
```
a[0][0] = 100  # Revise the element in a
print("After revise: \n", a, b)
# After iterating through b once,
#it won't display again because b is an iterator and can only be iterated once.
#If you want to see the values of b again, you need to get b again by using b = a.flat.
for item in b:
    print(item)  # No results from this loop, because b is a iterator from flat property
```

```
After revise: 
 [[100   1   2   3]
 [  4   5   6   7]
 [  8   9  10  11]] <numpy.flatiter object at 0x000002C1AE746350>
```

Use flatten() function

The main difference between them is that flatten() creates a new flattened copy of the original array, while flat returns an iterator that can be used to iterate the original array in a flattened manner without creating a new array.

Flatten() returns a new flattened array, while flat returns a flattened view of the original array. Additionally, flat can only be used to iterate the array once, while flatten() creates a new array that can be used multiple times.

In [84]:
```
a = np.arange(12).reshape(3,4)
for item in a.flatten():
    print(item, end = ",")
```

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

In [85]:
```
a[0][0] = 100  # Revise the element in a
for item in a.flatten():
    print(item, end = ",")
```

```
100,1,2,3,4,5,6,7,8,9,10,11,
```

Use nditer() function

Syntax: nditer(op, flags=None, op_flags=None, op_dtypes=None, order='K', casting='safe', op_axes=None, itershape=None, buffersize=0)

The nditer function provides a flexible way to iterate over one or more arrays and access their elements.

```
In [86]: for item in np.nditer(a):
             print(item, end = ",")
```

```
100,1,2,3,4,5,6,7,8,9,10,11,
```

If two arrays are broadcastable, nditer() can iterate over them simultaneously.

```
In [87]: x = np.arange(16).reshape(4,4)
         print(x)
         y = np.arange(1,5)
         print(y)
         for x, y in np.nditer([x,y]):
             print(f"{x}:{y}", end = ",")
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[1 2 3 4]
0:1,1:2,2:3,3:4,4:1,5:2,6:3,7:4,8:1,9:2,10:3,11:4,12:1,13:2,14:3,15:4,
```

Use ndenumerate() function

syntax: ndenumerate(arr)

Multidimensional index iterator. Returns an iterator yielding pairs of array coordinates and values.

```
In [88]: z = np.arange(9).reshape(3,3)
         print(z)
         for index, value in np.ndenumerate(z):
             print(index, value)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
(0, 0) 0
(0, 1) 1
(0, 2) 2
(1, 0) 3
(1, 1) 4
(1, 2) 5
(2, 0) 6
(2, 1) 7
(2, 2) 8
```

Numpy Operation/Manipulation

Change the shape of a ndarray

reshape function

resize function

shape property

Flat the ndarray

ravel function

flatten function

Difference between flatten and ravel

Change the dimension

Increase the dimension

```
[:, np.newaxis] # vertical direction to add a dimension
[np.newaxis, :] # horizontal direction to increase a dimension
```

Reduce the dimension

```
np.squeese(a, axis = None)
```

Transpose of a ndarray

```
transpose(a, axes = None)
```

In [89]:
```
import numpy as np
a = [[1,2,3,4],
    [5,6,7,8]]
arr = np.array(a)
print('Before transpose:\n',arr)
# print('After transpose:\n',arr.transpose())
print('After transpose:\n',arr.T)
```

```
Before transpose:
 [[1 2 3 4]
 [5 6 7 8]]
After transpose:
 [[1 5]
 [2 6]
 [3 7]
 [4 8]]
```

Remove the duplicated in a ndarray

Syntax:

```
unique(ndarray, return_index = False, return_inverse = False, return_counts = False, axis = None)
```

- the indices of the input array that give the unique values
- the indices of the unique array that reconstruct the input array

- the number of times each unique value comes up in the input array

```
In [90]: # One dimensional
         arr1 = np.array([1,2,3,3,3,4,5])
         np.unique(arr1)

Out[90]: array([1, 2, 3, 4, 5])
```

```
In [91]: # Two dimensional
         arr2 = np.array([[1, 2, 3, 4],[3, 4, 5, 6]])
         np.unique(arr2)  # return a one dimensional array

Out[91]: array([1, 2, 3, 4, 5, 6])
```

```
In [92]: a = np.array([1, 2, 6, 4, 2, 3, 2])
         values, counts = np.unique(a, return_counts = True)
         print(values)
         print(counts)
         np.repeat(values, counts)  # original order not preserved

         [1 2 3 4 6]
         [1 3 1 1 1]

Out[92]: array([1, 2, 2, 2, 3, 4, 6])
```

Concatenate the arrays

https://numpy.org/doc/stable/reference/generated/numpy.vstack.html#numpy-vstack

Split the arrays

https://numpy.org/doc/stable/reference/generated/numpy.split.html

Manipulate the elements of ndarray

Revise values in 1-D ndarray

```
In [93]: arr = np.arange(1,6)
         print(arr)

         #  Assigne the value 100 to the index position 1.
         arr[1] = 100

         # Slicing to revise values
         arr[2:4] = 0
         print('After revise: \n',arr)
```

```
[1 2 3 4 5]
After revise: 
 [  1 100   0   0   5]
```

Revise values in 2-D ndarray

```
In [94]: a = np.array([[1,2,3],
                       [4,5,6],
                       [7,8,9]])
         print('Before revise: \n',a)

         # Revise the entire line
         a[1] = 100 # Revise all the elements in the second line to 100.
         print('After revise: \n',a)
         # Revise the entire column
         a[:,1] = 100 # Revise all the element in the second column to 100.
         a
```

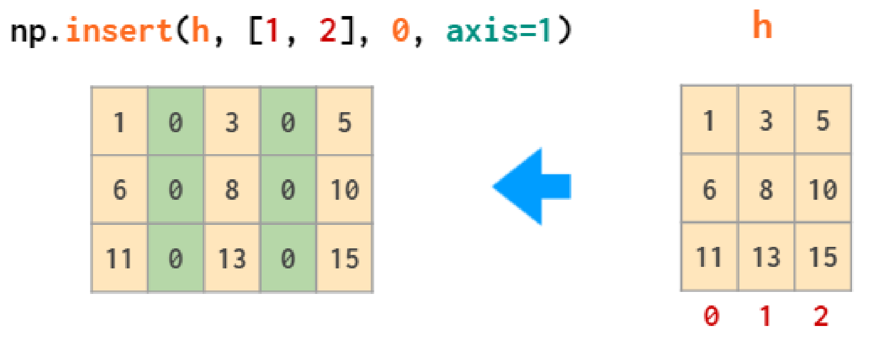
```
Before revise: 
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
After revise: 
 [[  1   2   3]
 [100 100 100]
 [  7   8   9]]
```

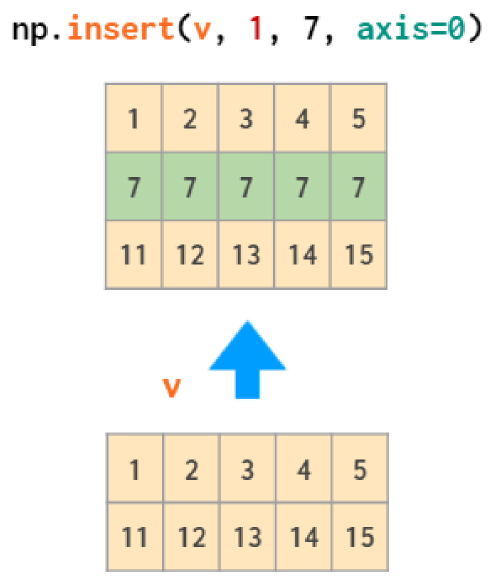
```
Out[94]: array([[  1, 100,   3],
                [100, 100, 100],
                [  7, 100,   9]])
```

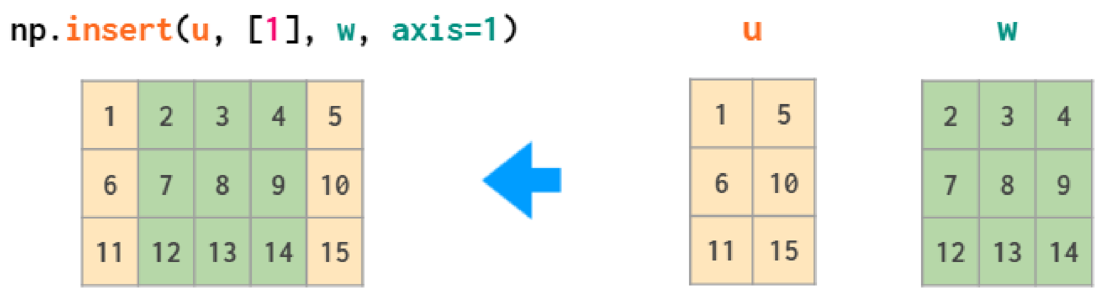
Insert elements

Syntax

```
numpy.insert(arr, obj, values, axis = None)
```



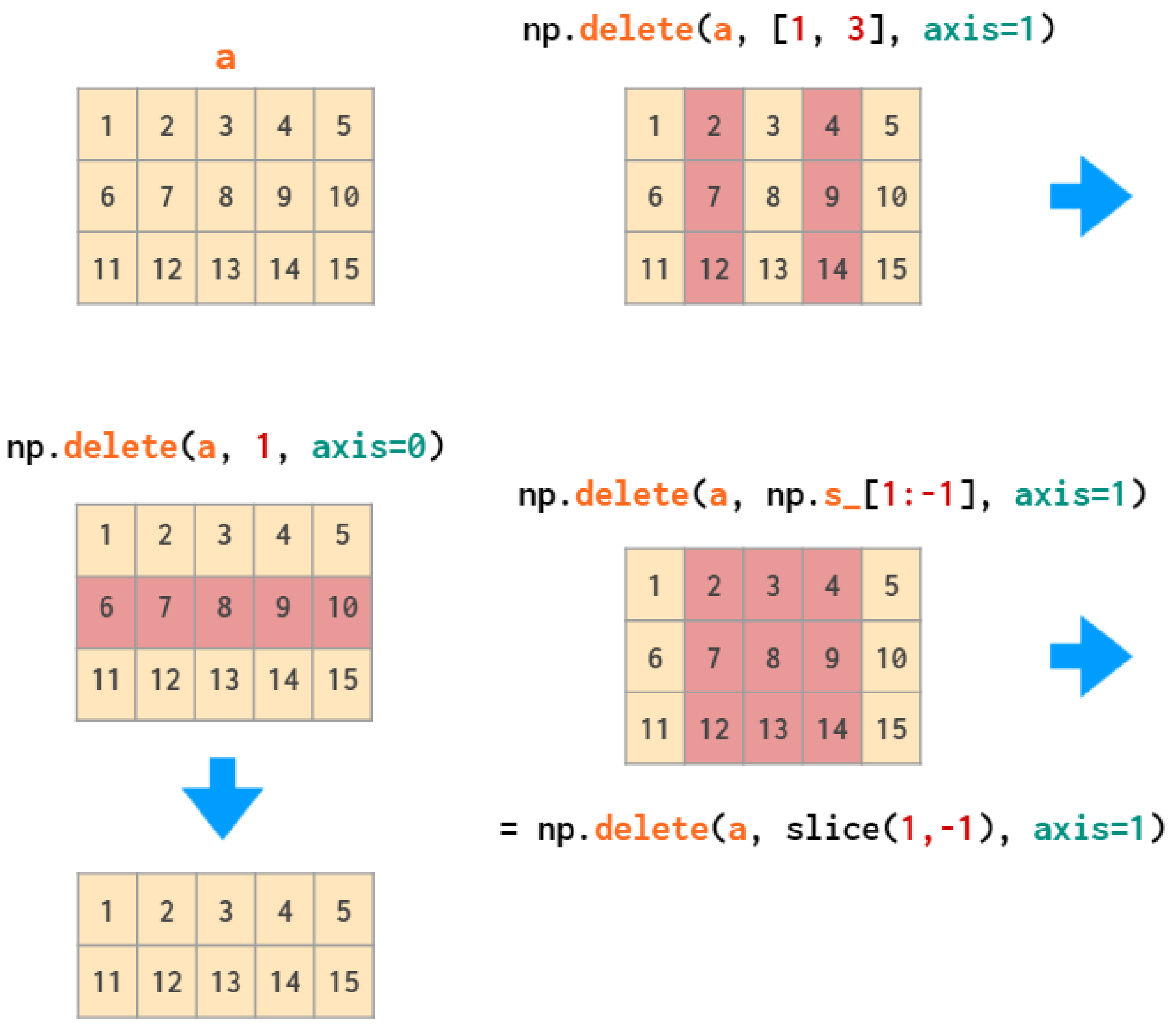




Delete elements

Syntax

```
numpy.delete(arr, obj, axis = None)
```

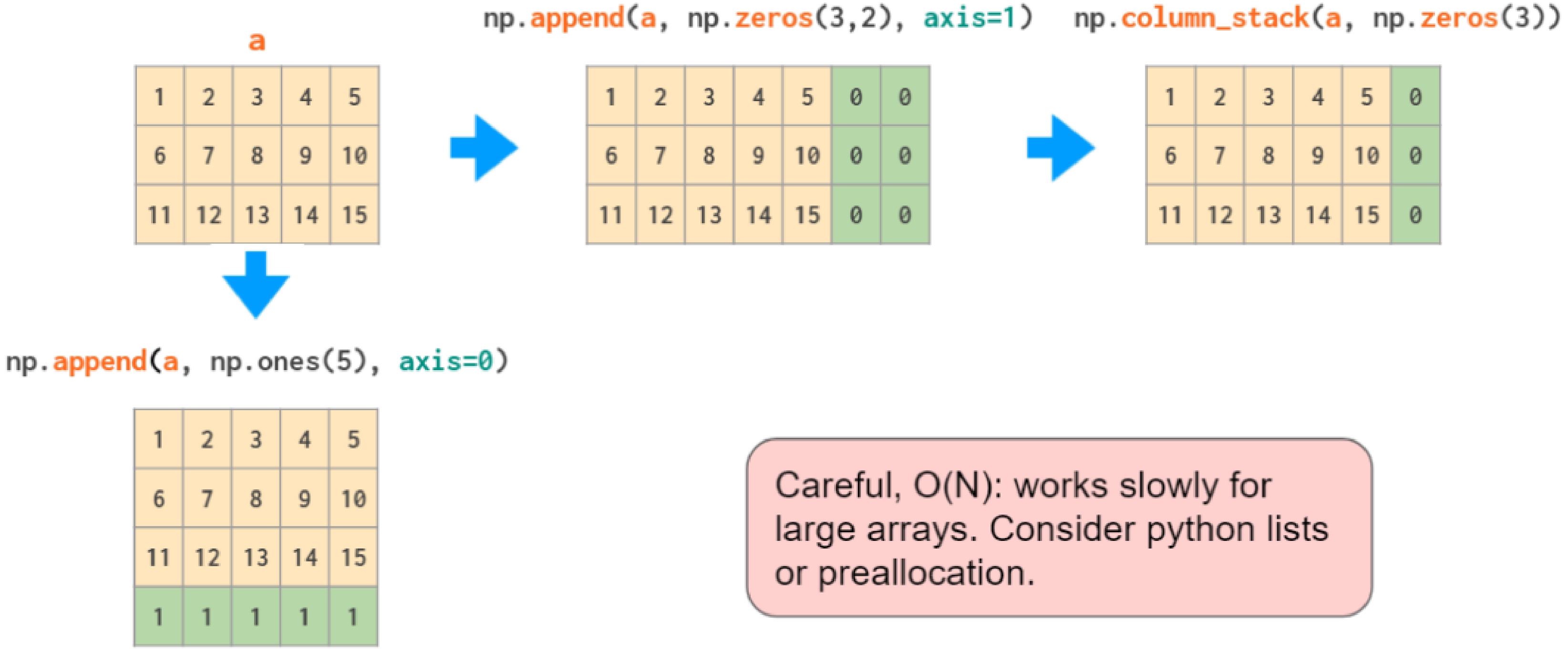


```
np.delete(a, [1, 3], axis=1)
np.delete(a, 1, axis=0)
np.delete(a, np.s_[1:-1], axis=1)
= np.delete(a, slice(1,-1), axis=1)
```

Append elements

Syntax

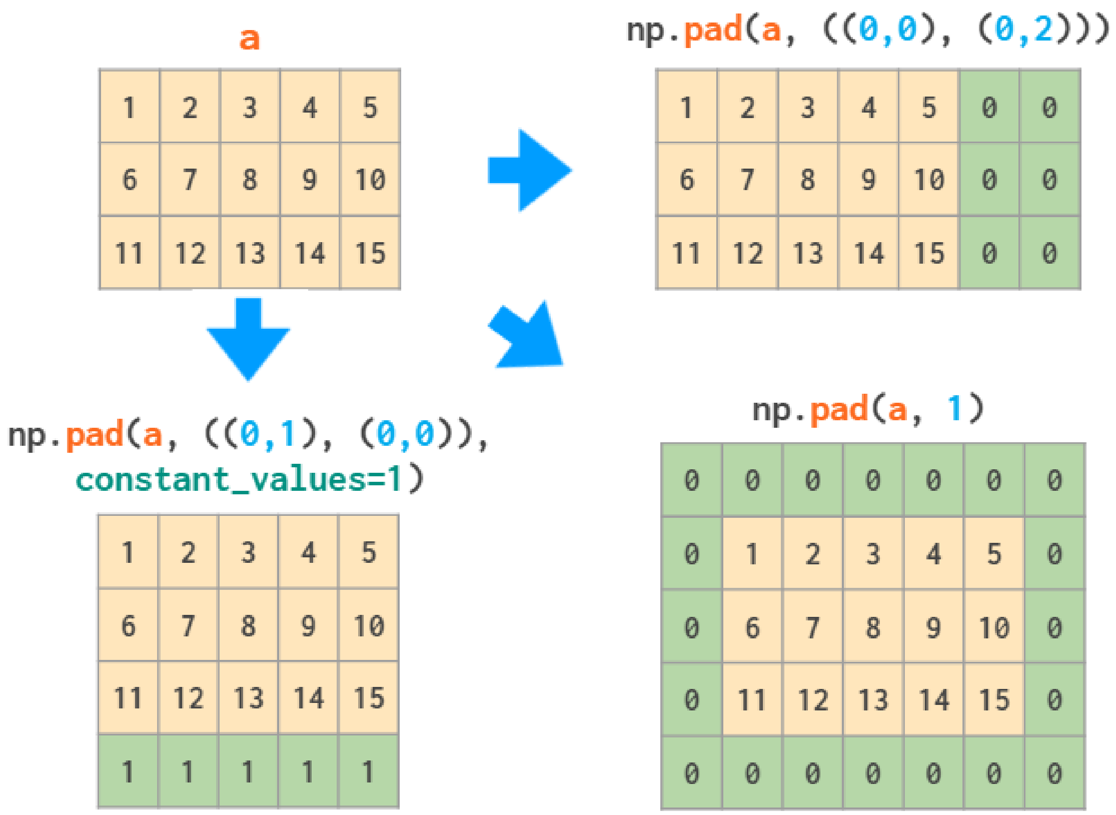
```
numpy.append(arr, values, axis = None)
```


a
np.append(a, np.zeros(3,2), axis=1)
np.column_stack(a, np.zeros(3))
np.append(a, np.ones(5), axis=0)
Careful, O(N): works slowly for large arrays. Consider python lists or preallocation.

Pad an array

Syntax

```
numpy.pad(array, pad_width, mode ='constant', **kwargs)
```



https://numpy.org/doc/stable/reference/generated/numpy.pad.html#numpy.pad

Numpy random module

Very frequently used in data analysis for the initiation of parameters, split the dataset, sampling for statistical analysis.

Basic random numbers

Generate random floats

random.rand

Syntax

`random.rand(d0, d1, ..., dn)`
Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).

```
In [95]: np.random.rand(3,2)
```

```
Out[95]: array([[0.32063791, 0.0578348 ],
                [0.41875906, 0.98370444],
                [0.17956525, 0.38298839]])
```

random_sample

Syntax

`random.random_sample(size = None)`
Return random floats in the half-open interval [0.0, 1.0).

```
In [96]: print(np.random.random_sample())
         print(type(np.random.random_sample())) # type is float
```

```
0.9356446384740765
<class 'float'>
```

```
In [97]: # Generate a 3X2 array of random numbers from [-5, 0):
         5 * np.random.random_sample((3, 2)) - 5
```

```
Out[97]: array([[-3.44797115, -4.25507564],
                [-0.68198836, -3.31844429],
                [-4.77222953, -3.4435931 ]])
```

random.random

Syntax

`random.random(size = None)`
Return random floats in the half-open interval [0.0, 1.0)

Generate random integers

random.randint Syntax

`random.randint(low, high = None, size = None, dtype = int)`
Return random integers from low (inclusive) to high (exclusive).

Return random integers from the “discrete uniform” distribution of the specified dtype in the “half-open” interval [low, high). If high is None (the default), then results are from [0, low).

In [98]:
```
a = np.random.randint(10,50,size = (3,2))  # random integers between [10, 50), 3 by 2 matrix
a
```

Out[98]:
```
array([[16, 49],
       [39, 18],
       [29, 30]])
```

Random sampling

random.choice

Syntax

`random.choice(a, size = None, replace = True, p = None)`
Generates a random sample from a given 1-D array.

In [99]:
```
data = np.arange(10)
np.random.choice(data, (3,3))  #first argument represent the given 1-D array, second tuple indicates the size of random
```

Out[99]:
```
array([[4, 7, 1],
       [4, 2, 1],
       [5, 0, 6]])
```

In [100…
```
np.random.choice(10, (3,3)) # we can just use 10 to represent np.arange(10)
```

Out[100]:
```
array([[7, 2, 5],
       [5, 5, 5],
       [9, 2, 1]])
```

In [101…
```
# Set the elements to be non-duplicated
np.random.choice(data, (3,3), replace = False)
```

```
Out[101]: array([[8, 7, 4],
                 [1, 0, 3],
                 [5, 9, 6]])
```

The probabilities associated with each entry in a. If not given, the sample assumes a uniform distribution over all entries in a.

```
In [102…  np.random.choice(3, 10, p = [0.3, 0.2, 0.5]) # p is the probablity for each element to be chosen in np.arange(3)
```

```
Out[102]: array([0, 2, 2, 1, 2, 1, 2, 1, 1, 2])
```

```
In [103…  aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
          np.random.choice(aa_milne_arr, 5, p = [0.5, 0.1, 0.1, 0.3])
```

```
Out[103]: array(['pooh', 'rabbit', 'pooh', 'pooh', 'Christopher'], dtype='<U11')
```

Distribution random numbers

Normal distribution

random.randn

Syntax

`random.randn(d0, d1, ..., dn)`
Return a sample (or samples) from the “standard normal” distribution.

```
In [104…  a = np.random.randn(2, 3)
          a
```

```
Out[104]: array([[ 0.63902525,  0.6956892 , -0.30469091],
                 [ 0.75911813,  0.28144317,  2.99302878]])
```

standard_normal

Syntax

`random.standard_normal(size = None)`
Draw samples from a standard Normal distribution (mean=0, stdev=1).

In [105…
```
# Two-by-four array of samples from the normal distribution with mean 3 and standard deviation 2.5:
3 + 2.5 * np.random.standard_normal(size = (2, 4))
```

Out[105]:
```
array([[ 6.65877064, -2.29735772, -3.33286083,  2.12467372],
       [ 2.03799283, -3.17244283,  2.65564999,  1.79798669]])
```

Normal Distribution

Syntax

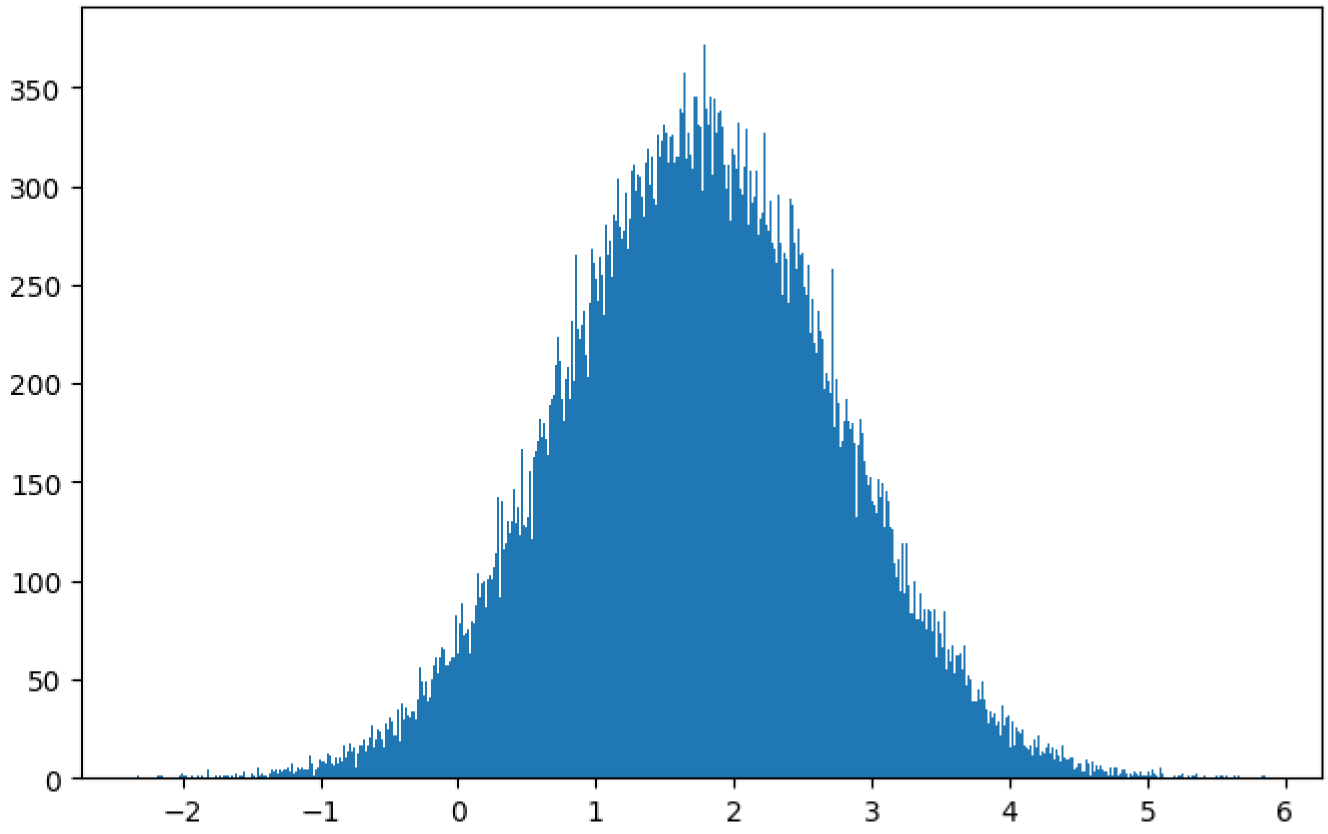
`random.normal(loc = 0.0, scale = 1.0, size = None)`

Draw random samples from a normal (Gaussian) distribution.

In [106…
```
%matplotlib inline
from matplotlib import pyplot as plt

x = np.random.normal(1.75, 1, 100000) # mu = 1.75, sigma = 1 Gaussian

plt.figure(figsize = (8, 5), dpi = 100)
plt.hist(x, bins = 1000)
plt.show()
```



Uniform Distribution

Syntax

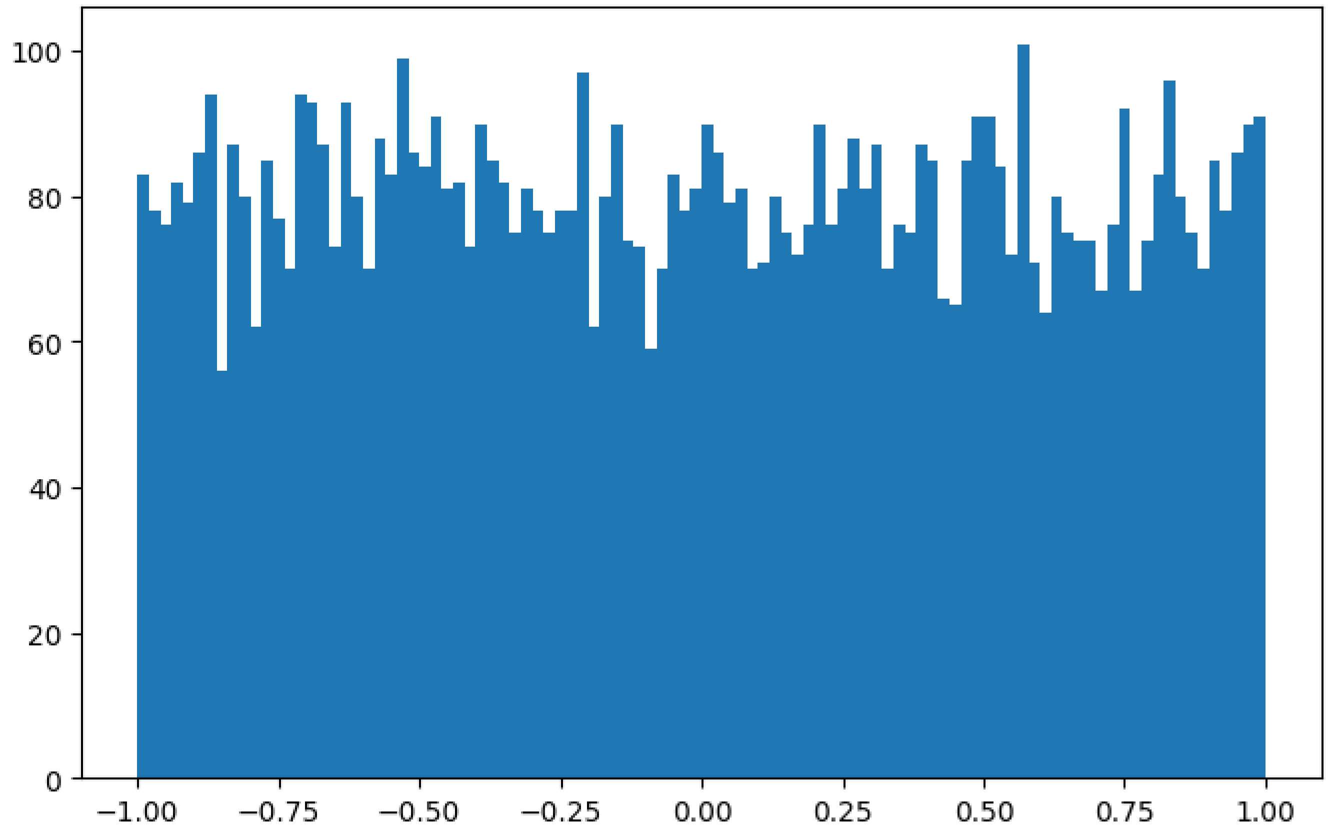
`random.uniform(low = 0.0, high = 1.0, size = None)`
Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by uniform.

In [107…

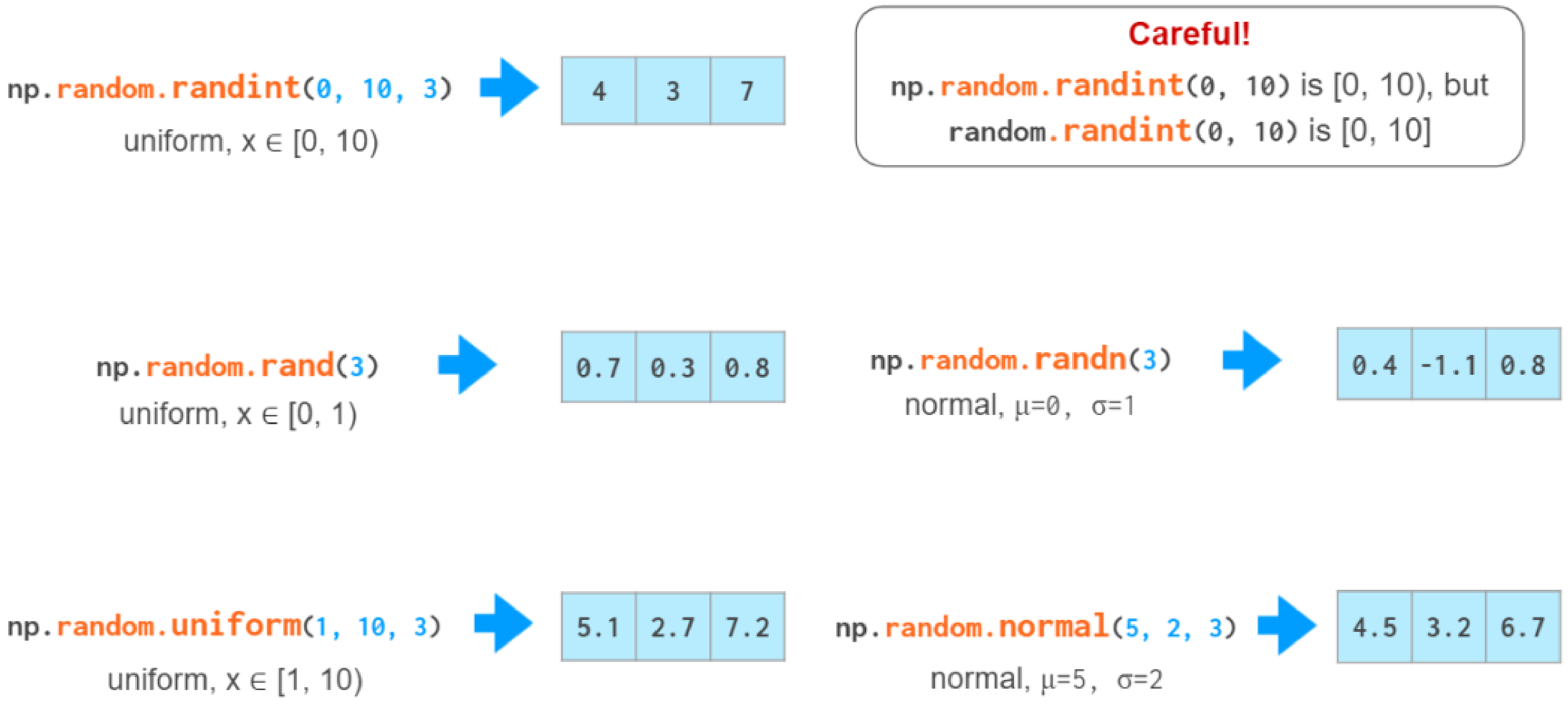
```
x1 = np.random.uniform(-1, 1, 8000)
plt.figure(figsize = (8,5), dpi = 100)
plt.hist(x1, bins = 100)
plt.show
```

```
Out[107]: <function matplotlib.pyplot.show(close=None, block=None)>
```



Summary

- One dimensional



```
np.random.randint(0, 10, 3)  →  [4 | 3 | 7]
```
uniform, x ∈ [0, 10)

Careful!
`np.random.randint(0, 10)` is [0, 10), but
`random.randint(0, 10)` is [0, 10]

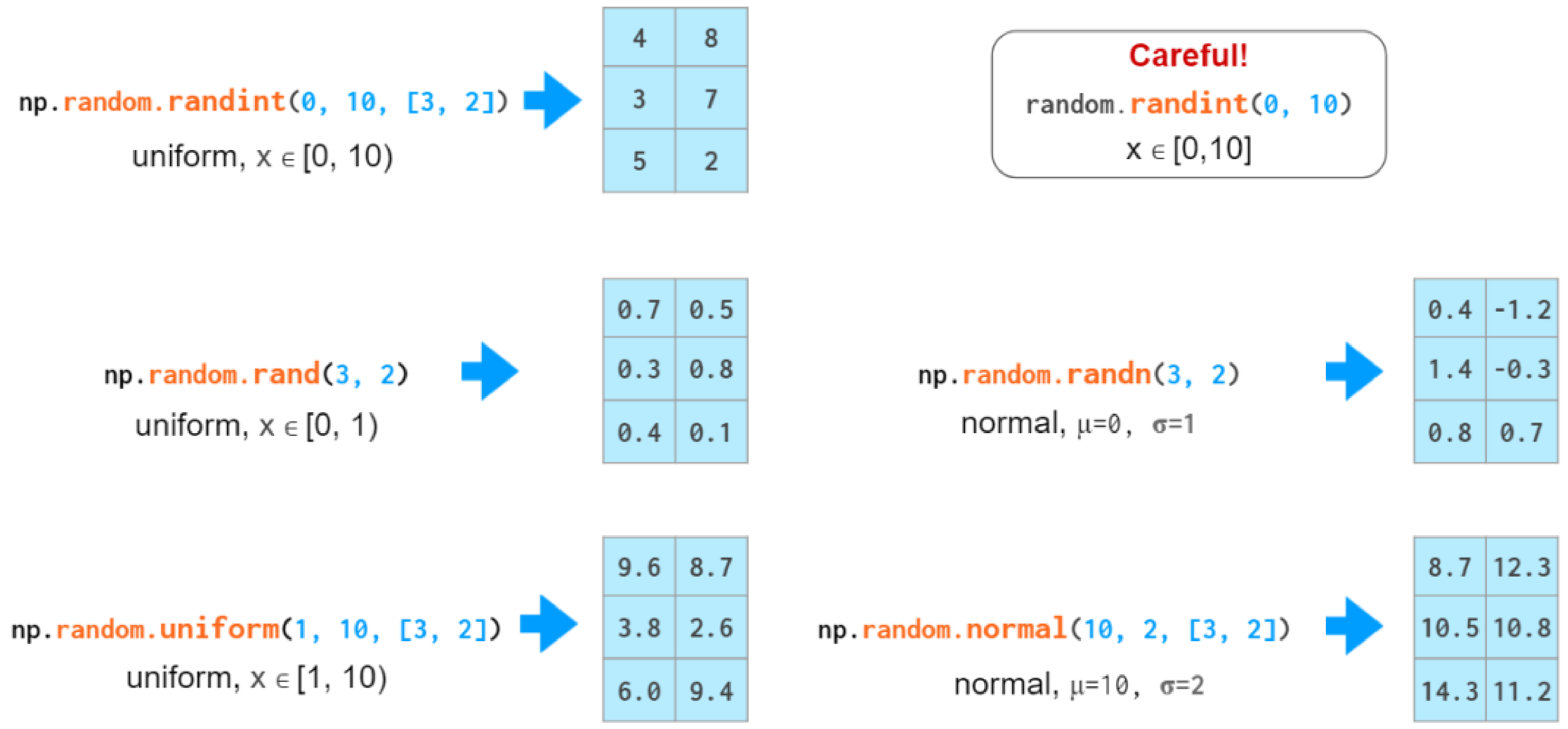
```
np.random.rand(3)  →  [0.7 | 0.3 | 0.8]
```
uniform, x ∈ [0, 1)

```
np.random.randn(3)  →  [0.4 | -1.1 | 0.8]
```
normal, $\mu=0$, $\sigma=1$

```
np.random.uniform(1, 10, 3)  →  [5.1 | 2.7 | 7.2]
```
uniform, x ∈ [1, 10)

```
np.random.normal(5, 2, 3)  →  [4.5 | 3.2 | 6.7]
```
normal, $\mu=5$, $\sigma=2$

- Two dimensional



```
np.random.randint(0, 10, [3, 2])
```
uniform, x ∈ [0, 10)

4	8
3	7
5	2

Careful!
```
random.randint(0, 10)
```
x ∈ [0,10]

```
np.random.rand(3, 2)
```
uniform, x ∈ [0, 1)

0.7	0.5
0.3	0.8
0.4	0.1

```
np.random.randn(3, 2)
```
normal, μ=0, σ=1

0.4	-1.2
1.4	-0.3
0.8	0.7

```
np.random.uniform(1, 10, [3, 2])
```
uniform, x ∈ [1, 10)

9.6	8.7
3.8	2.6
6.0	9.4

```
np.random.normal(10, 2, [3, 2])
```
normal, μ=10, σ=2

8.7	12.3
10.5	10.8
14.3	11.2

Random sequences of array

random.shuffle(x)

random.permutation(x)

random.seed

To generate the same random numbers

Numpy Date

In [108…
```
# create datetime64 objective
date64 = np.datetime64('2023-05-10 03:13:10')
date64
```

Out[108]:
```
numpy.datetime64('2023-05-10T03:13:10')
```

In [109…
```
# seperate time from datatime64 date
dt64 = np.datetime64(date64, 'D')
dt64
```

Out[109]:
```
numpy.datetime64('2023-05-10')
```

In [110…
```
# Add any times in terms of seconds, mins by using np.timedelta

tenminutes = np.timedelta64(10, 'm')  # 10 mins
tenseconds = np.timedelta64(10, 's')  # 10 seconds
tennanoseconds = np.timedelta64(10, 'ns')  # 10 ns

print('Add 10 days: ', dt64 + 10)  # add 10 days
print('Add 10 minutes: ', dt64 + tenminutes)  # add 10 mins
print('Add 10 seconds: ', dt64 + tenseconds)    # add 10 seconds
print('Add 10 nanoseconds: ', dt64 + tennanoseconds)  # add 10 ns
```

```
Add 10 days:  2023-05-20
Add 10 minutes:  2023-05-10T00:10
Add 10 seconds:  2023-05-10T00:00:10
Add 10 nanoseconds:  2023-05-10T00:00:00.000000010
```

In [111…
```
# TO get yesterday dates, today dates and tomorrow
yesterday = np.datetime64('today','D') - np.timedelta64(1,'D')
today = np.datetime64('today','D')
tommorow = np.datetime64('today','D') + np.timedelta64(1,'D')
print(today)
```

```
2023-05-10
```

In [112…
```
dates = np.arange(np.datetime64('2023-05-01'), np.datetime64('2023-05-31'))
print(dates)
```

```
['2023-05-01' '2023-05-02' '2023-05-03' '2023-05-04' '2023-05-05'
 '2023-05-06' '2023-05-07' '2023-05-08' '2023-05-09' '2023-05-10'
 '2023-05-11' '2023-05-12' '2023-05-13' '2023-05-14' '2023-05-15'
 '2023-05-16' '2023-05-17' '2023-05-18' '2023-05-19' '2023-05-20'
 '2023-05-21' '2023-05-22' '2023-05-23' '2023-05-24' '2023-05-25'
 '2023-05-26' '2023-05-27' '2023-05-28' '2023-05-29' '2023-05-30']
```

In [113…

```python
dates = np.arange(np.datetime64('2020-01'), np.datetime64('2023-12'))
print(dates)
```

```
['2020-01' '2020-02' '2020-03' '2020-04' '2020-05' '2020-06' '2020-07'
 '2020-08' '2020-09' '2020-10' '2020-11' '2020-12' '2021-01' '2021-02'
 '2021-03' '2021-04' '2021-05' '2021-06' '2021-07' '2021-08' '2021-09'
 '2021-10' '2021-11' '2021-12' '2022-01' '2022-02' '2022-03' '2022-04'
 '2022-05' '2022-06' '2022-07' '2022-08' '2022-09' '2022-10' '2022-11'
 '2022-12' '2023-01' '2023-02' '2023-03' '2023-04' '2023-05' '2023-06'
 '2023-07' '2023-08' '2023-09' '2023-10' '2023-11']
```

In [114…

```python
# Convert the date format to string using np.datetime_as_string()
a = np.datetime_as_string(dt64)
print(f"date is {a}, type is {type(a)}")
```

```
date is 2023-05-10, type is <class 'numpy.str_'>
```

In [115…

```python
# np.datetime64 converted into datetime.date
import datetime
print('Before: ',type(dt64))
dt = dt64.tolist()
print('After: ',type(dt))
```

```
Before:  <class 'numpy.datetime64'>
After:  <class 'datetime.date'>
```

In [116…

```python
# To get datatime object day, month, year that is very convinient
print('Year: ', dt.year)
print('Day of month: ', dt.day)
print('Month of year: ', dt.month)
print('Day of Week: ', dt.weekday())
```

```
Year:  2023
Day of month:  10
Month of year:  5
Day of Week:  2
```

Numpy Mathematic Operation

function	Description
sqrt()	square root
sin()、cos()	Trigonometric Identities
abs()	Absolute Values
dot()	Dot product
log()、logl()、log2()	logarithmic
exp()	Exponential
cumsum()、cumproduct()	Cum sum and cum product
sum()	Sum
mean()	Mean
median()	Median
std()	Std
var()	Variance
corrcoef()	Person linear coefficient

np.floor(), np.ceil(), np.round()

Solve(a,b)

Calculate the linear system equation.

$x + y + z = 6$

$2y + 5z = -4$

$2x + 5y - z = 27$

```
In [117…
a = [[1,1,1],[0,2,5],[2,5,-1]]
b = [6,-4,27]
print (np.linalg.solve(a,b))
```

```
[ 5.  3. -2.]
```

Inverse of a matrix

Logic Operation : all(), any(), where()

Statistical Related

```
In [ ]:
```