NumPy

- Jake VanderPlas. 2016. *Python Data Science Handbook: Essenti*
  Tools for Working with Data. O'Reilly Media, Inc.
- Chapter 2 - Introduction to NumPy
- **https://github.com/jakevdp/PythonDataScienceHandbo**
  (https://github.com/jakevdp/PythonDataScienceHandb

NumPy provides:

- Memory-efficient N-dimensional arrays with fast, vectorized operations.
- Mathematical functions for linear algebra, statistics, random number generation, etc.
- Element-wise operations between arrays of different but compatible shapes (**broadcasting**).
- Advanced array indexing
- Foundation for many other scientific Python libraries (SciPy, Pandas, Matplotlib, Scikit-learn)

In [1]:

```
import numpy as np
np.__version__
```

Out[1]:

```
'1.26.4'
```

In [2]:

```
# type TAB to get the numpy namespace
#np.
```

Data Types

NumPy provides an alternative implementation for numerical arrays, improving the performance of data-driven computation compared to standard Python built-in lists.

Python Integers vs C Integers

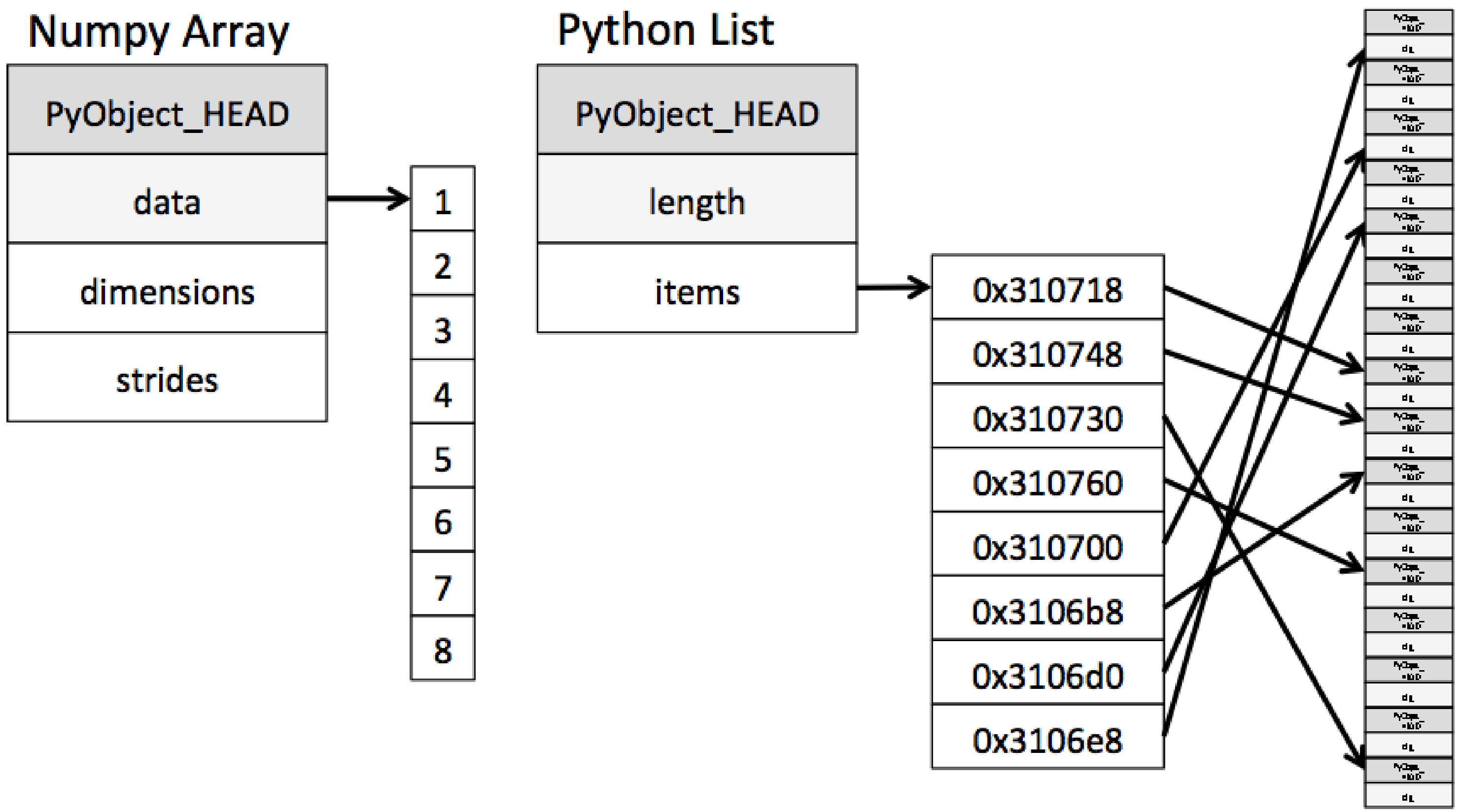
- Python `int`s are *complex* objects (written in C)
  - Dynamically-typed language
  - Almost infinite integer arithmetic precision

```
struct _longobject {
long ob_refcnt;           # reference count
PyTypeObject *ob_type;    # type of the variable
size_t ob_size;           # size of the following data memb
long ob_digit[1];         # integer value encoded into a lo
};
```

- C language integers (char, short, int, long, long...) are simple refer
  a position in memory whose bytes encode an integer value.

Python Lists vs NumPy Arrays

- Python `lists` are *complex* objects (much more than `ints`)
    - `lists` are heterogeneous
    - Different object types are different sizes
    - `lists` contain an array with references to each object
    - Contain a pointer to a block of pointers, each of which points to a Python object
- *Standard* Numpy arrays are homogeneous.
    - Contain a single pointer to one contiguous block of data.


Numpy Array
PyObject_HEAD
data
dimensions
strides
1
2
3
4
5
6
7
8
Python List
PyObject_HEAD
length
items
0x310718
0x310748
0x310730
0x310760
0x310700
0x3106b8
0x3106d0
0x3106e8

NumPy Arrays

Creating Arrays from Python Lists

- `np.array(some_list)` → create an (homogeneous) array
- `np.array(some_list, dtype=<data type>)` → create an array of a given type

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

In [4]:
```
np.array([1, 4, 2, 5, 3], dtype='float32')
```
Out[4]:
```
array([1., 4., 2., 5., 3.], dtype=float32)
```

If types do not match, NumPy will upcast if possible:

In [5]:

```
np.array([1, 4, 2, 5.9, 3])
```

Out[5]:

```
array([1. , 4. , 2. , 5.9, 3. ])
```

Nested lists result in multi-dimensional arrays

In [6]:

```
np.array([[ 0,  1,  2,  3], [10, 11, 12, 13], [20, 21, 22, 23]])
```

Out[6]:

```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23]])
```

NumPy Standard Data Types

when constructing an array, the data type (`dtype`) argument can be specified using:

- string → `dtype='float32'`
- NumPy object → `dtype=np.float32`

In [7]:

```
np.array([1, 4, 2, 5, 3], dtype='float32')
```

Out[7]:

```
array([1., 4., 2., 5., 3.], dtype=float32)
```

In [8]:

```
np.array([1, 4, 2, 5, 3], dtype=np.float32)
```

Out[8]:

```
array([1., 4., 2., 5., 3.], dtype=float32)
```

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa

Data type	Description
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats

Creating Arrays from Scratch

Default values of `dtype` (most of times):

- Integers → `int64`
- Reals (floating point) → `float64`

- `np.zeros(10)` → a length-10 array filled with zeros

In [9]:

```
np.zeros(10)
```

Out[9]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

In [10]:

```
np.zeros(10, dtype='int64')
```

Out[10]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [11]:

```
np.zeros(10, dtype='float32')
```

Out[11]:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

In [12]:

```
np.zeros(10, dtype='int32')
```

Out[12]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0], dtype=int32)
```

- `np.eye(3)` → a 3x3 identity matrix

In [13]:

```
np.eye(3)
```

Out[13]:

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

- `np.ones((3,5))` → a 3x5 array filled with ones

In [14]:

```
np.ones((3,5))
```

Out[14]:

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

- `np.full((3,5), 1.8)` → a 3x5 array filled with `1.8`

In [15]:

```
np.full((3,5), 1.8)
```

Out[15]:

```
array([[1.8, 1.8, 1.8, 1.8, 1.8],
       [1.8, 1.8, 1.8, 1.8, 1.8],
       [1.8, 1.8, 1.8, 1.8, 1.8]])
```

- `np.empty((3,5))` → an *uninitialized* 3x5 array

In [16]:

```
np.empty((3,5))
```

Out[16]:

```
array([[1.8, 1.8, 1.8, 1.8, 1.8],
       [1.8, 1.8, 1.8, 1.8, 1.8],
       [1.8, 1.8, 1.8, 1.8, 1.8]])
```

- `np.random.random((3,5))` → a 3x5 array of uniformly distributed random values in the half-open interval $[0, 1)$

In [17]:

```
np.random.random((3,5))
```

Out[17]:

```
array([[0.8900825 , 0.46183266, 0.25763459, 0.87569671, 0.44428327],
       [0.96104856, 0.29987648, 0.71443116, 0.97085317, 0.41884659],
       [0.24140015, 0.00879754, 0.51899084, 0.38203133, 0.33946914]])
```

- `np.random.seed(int)` → sets a seed for random number generation (**provides reproducibility**)

In [18]:

```
np.random.seed(43)
np.random.random((3,5))
```

Out[18]:

```
array([[0.11505457, 0.60906654, 0.13339096, 0.24058962, 0.32713906],
       [0.85913749, 0.66609021, 0.54116221, 0.02901382, 0.7337483 ],
       [0.39495002, 0.80204712, 0.25442113, 0.05688494, 0.86664864]])
```

- `np.random.normal(10, 2, (3,5))` → a 3x5 array of normally distributed random values with $\mu = 10$ and $\sigma = 2$

In [19]:

```
np.random.normal(10, 2, (3, 5))
```

Out[19]:

```
array([[ 9.06239973,  7.247008  ,  8.96229899,  9.54920845, 10.41643253
       [10.40961276,  9.83619328, 10.67376611,  9.85909104, 11.18066954
       [ 8.80587096, 10.48544469,  9.38402231, 11.21093121,  7.00980231
```

- `np.random.randint(-7, 3, (3, 5))` → a 3x5 array of random integers in the half-open interval $[-7, 3)$

In [20]:

```
np.random.randint(-7, 3, (3, 5))
```

Out[20]:

```
array([[-2, -2, -2,  1,  0],
       [-4, -1, -4, -5, -3],
       [-5,  0, -4, -1,  1]])
```

- `np.arange(4, 20, 2)` → an array filled with a linear sequence in the range $[4, 20)$ and step 2

In [21]:

```
np.arange(4, 20, 2)
```

Out[21]:

```
array([ 4,  6,  8, 10, 12, 14, 16, 18])
```

- `np.linspace(1, 7, 5)` → an array of five values evenly spaced in the range $[1, 7]$

In [22]:

```
np.linspace(1, 7, 5)
```

Out[22]:

```
array([1. , 2.5, 4. , 5.5, 7. ])
```

More on NumPy Arrays

- Array dimensions
- Array attributes
- Array indexing
- Array slicing
- Reshaping of arrays
- Array concatenation and splitting

Array dimensions

NumPy arrays can have any number of dimensions:

- 0-dimensional arrays → **scalars** or **rank-0 tensor**
- 1-dimensional arrays → **vectors** or **rank-1 tensor**
- 2-dimensional arrays → **matrices** or **rank-2 tensor**
- 3-dimensional arrays → **tensors** or **rank-3 tensor**
- ...
- N-dimensional arrays → **tensors** or **rank-N tensor**

In [23]:

```
np.random.randint(1,10,(2,2,2,3,6))
```

Out[23]:

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

         [[2, 6, 9, 1, 4, 7],
          [5, 7, 8, 5, 6, 9],
          [5, 8, 5, 3, 7, 8]]],


        [[[9, 9, 2, 9, 6, 6],
          [9, 7, 9, 4, 5, 3],
          [4, 2, 5, 3, 3, 2]],

         [[5, 6, 6, 1, 4, 5],
          [8, 2, 3, 8, 4, 2],
          [2, 4, 7, 6, 9, 1]]]],



       [[[[6, 1, 9, 7, 8, 9],
          [9, 8, 2, 5, 2, 7],
          [4, 1, 2, 8, 9, 4]],

         [[3, 1, 4, 3, 1, 3],
          [7, 8, 5, 5, 7, 1],
          [5, 7, 1, 5, 7, 4]]],


        [[[7, 8, 9, 2, 5, 6],
          [2, 9, 1, 3, 3, 8],
          [7, 6, 1, 5, 1, 9]],

         [[9, 2, 9, 2, 7, 1],
```

```
[3, 2, 6, 1, 6, 6],
[9, 4, 3, 1, 6, 6]]]]])
```

Array attributes

Given a NumPy array `x`:

- `x.ndim` → number of dimensions
- `x.shape` → a tuple of size `x.ndim` containing the size of each dimension
- `x.size` → total size of the array
- `x.dtype` → data type of the array

In [24]:

```
x = np.random.rand(3, 4)
print(x)
print(f'{x.ndim=}  {x.shape=}  {x.size=}  {x.dtype=}')
```

```
[[0.9973726  0.48601552 0.32129159 0.973439  ]
 [0.99871854 0.4452131  0.0447457  0.44706112]
 [0.7101748  0.62607133 0.67064977 0.97191886]]
x.ndim=2  x.shape=(3, 4)  x.size=12  x.dtype=dtype('float64')
```

Array indexing

One dimensional arrays are indexed as Python lists:

In [25]:

```
x = np.arange(100,110)
x[3]
```

Out[25]:

```
103
```

Negative indexes are valid as with Python lists:

In [26]:

```
x[-1],x[-10]
```

Out[26]:

```
(109, 100)
```

Multi-dimensional arrays are indexed with a comma-separated tuple of indices

In [27]:

```
x = np.random.rand(3,4)
print(x)
print(x[0,3], x[-1,0])
```

```
[[0.66197124 0.0914502  0.58973101 0.21158101]
 [0.79507563 0.35030598 0.50243648 0.70353915]
 [0.17162768 0.60105944 0.16003461 0.8265784 ]]
0.21158100640626754 0.17162767591793848
```

Values can be modified using any of the above index notation.

In [28]:

```
x = np.arange(10)
print(x)
x[-1] = 20
print(x)
```

```
[0 1 2 3 4 5 6 7 8 9]
[ 0  1  2  3  4  5  6  7  8 20]
```

NumPy arrays have fixed types. If you set a float value to an element of an int array, the value is truncated.

In [29]:

```
print(x)
x[-1] = 17.321
print(x)
```

```
[ 0  1  2  3  4  5  6  7  8 20]
[ 0  1  2  3  4  5  6  7  8 17]
```

Array slicing

NumPy slicing syntax follows that of the standard Python list

- `x[start:stop:step]`
- If omitted, default values apply:
    - `start=0`
    - `stop=size_of_dimension`
    - `step=1`

In [30]:

```
x = np.arange(10)
print(f'{x[:5]=}')
print(f'{x[5:]=}')
print(f'{x[4:7]=}')
print(f'{x[::2]=}')
print(f'{x[::-1]=}')
print(f'{x[5::-2]=}')
```

```
x[:5]=array([0, 1, 2, 3, 4])
x[5:]=array([5, 6, 7, 8, 9])
x[4:7]=array([4, 5, 6])
x[::2]=array([0, 2, 4, 6, 8])
x[::-1]=array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
x[5::-2]=array([5, 3, 1])
```

In [31]:

```
x = np.random.rand(3,5)
print(f'{x=}')
print(f'{x[1,:]=}')
print(f'{x[1]=}') # equivalent
print(f'{x[:,1]=}')
print(f'{x[:2,3:]=}')
print(f'{x[0,::-1]=}')
print(f'{x[::-1,1]=}')
```

```
x=array([[0.33270163, 0.54669719, 0.8307106 , 0.96906706, 0.32753498],
       [0.08837382, 0.96348912, 0.98930344, 0.91907709, 0.85048569],
       [0.49369274, 0.46861656, 0.55895281, 0.1069022 , 0.89873709]])
x[1,:]=array([0.08837382, 0.96348912, 0.98930344, 0.91907709, 0.850485
x[1]=array([0.08837382, 0.96348912, 0.98930344, 0.91907709, 0.85048569]
x[:,1]=array([0.54669719, 0.96348912, 0.46861656])
x[:2,3:]=array([[0.96906706, 0.32753498],
       [0.91907709, 0.85048569]])
x[0,::-1]=array([0.32753498, 0.96906706, 0.8307106 , 0.54669719, 0.3327
x[::-1,1]=array([0.46861656, 0.96348912, 0.54669719])
```

Unlike Python list slices, array slices return **views** rather than copies:

In [32]:

```
x = np.random.rand(3,5)
print(f'{x=}')
y = x[1,::2]
print(f'{y=}')
print('-'*40)
y[0] = 0
print(f'{y=}')
print(f'{x=}')
```

```
x=array([[0.61514721, 0.99204581, 0.6925764 , 0.35473485, 0.17346336],
       [0.04616806, 0.32678971, 0.35579443, 0.47938721, 0.65897812],
       [0.08594221, 0.29644561, 0.20342314, 0.65215974, 0.57715637]])
y=array([0.04616806, 0.35579443, 0.65897812])
----------------------------------------
y=array([0.        , 0.35579443, 0.65897812])
x=array([[0.61514721, 0.99204581, 0.6925764 , 0.35473485, 0.17346336],
       [0.        , 0.32678971, 0.35579443, 0.47938721, 0.65897812],
       [0.08594221, 0.29644561, 0.20342314, 0.65215974, 0.57715637]])
```

Explicit copies of arrays or subarrays (slices) can be created:
`x.copy()`

In [33]:

```
x = np.random.rand(3,5)
print(f'{x=}')
y = x[1,::2].copy()
print(f'{y=}')
print('-'*40)
y[0] = 0
print(f'{y=}')
print(f'{x=}')
```

```
x=array([[0.68180214, 0.21692997, 0.02226975, 0.03897417, 0.45671379],
       [0.8776101 , 0.53915719, 0.52549328, 0.54422628, 0.51641898],
       [0.88635236, 0.4101944 , 0.81952181, 0.832217  , 0.59418099]])
y=array([0.8776101 , 0.52549328, 0.51641898])
----------------------------------------
y=array([0.        , 0.52549328, 0.51641898])
x=array([[0.68180214, 0.21692997, 0.02226975, 0.03897417, 0.45671379],
       [0.8776101 , 0.53915719, 0.52549328, 0.54422628, 0.51641898],
       [0.88635236, 0.4101944 , 0.81952181, 0.832217  , 0.59418099]])
```

Reshaping of arrays

- `reshape(a, newshape)` → a reshaped **view** of an array
- `a.reshape(newshape)` → a reshaped **view** of an array

In [34]:

```
np.reshape(np.arange(10), (2, 5))
```

Out[34]:

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

In [35]:

```
np.arange(10).reshape((2, 5))
```

Out[35]:

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

In [36]:

```
a = np.arange(10)
b = a.reshape((2,5))
c = b.reshape((1,10))
b[0,0] = -8
print(f'{a=}\n{b=}\n{c=}')
```

```
a=array([-8,  1,  2,  3,  4,  5,  6,  7,  8,  9])
b=array([[-8,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9]])
c=array([[-8,  1,  2,  3,  4,  5,  6,  7,  8,  9]])
```

`np.newaxis`, when used in Array slicing adds a new dimension (does a reshape):

In [37]:

```
from numpy import newaxis
a = np.arange(4)     # vector
b = a[newaxis,:]  # row vector  a.reshape((1,4))
c = a[:,newaxis]  # column vector a.reshape((4,1))
a[0] = -8
print(f'{a=}\n{b=}\n{c=}')
```

```
a=array([-8,  1,  2,  3])
b=array([[-8,  1,  2,  3]])
c=array([[-8],
       [ 1],
       [ 2],
       [ 3]])
```

Array concatenation and splitting

- Array **concatenation** → combine multiple arrays into one
  - `np.concatenate`, `np.vstack` and `np.hstack`
- Array **splitting** → split a single array into multiple arrays
  - `np.split`, `np.vsplit` and `np.hsplit`

- `concatenate((a1, a2, ...), axis=0, out=None, dtype=None, casting="same_kind")`
    - `(a1, a2, ...)` → array sequence
    - `axis` → the axis along which the arrays will be joined
    - arrays must have the dimension corresponding to `axis`
    - arrays must have the same shape, except in the dimension corresponding to `axis`

In [38]:

```
a = np.arange(0,4)
b = np.arange(4,8)
c = np.arange(8,12)
d = np.concatenate((a,b,c))
print(f'{a=}\n{b=}\n{c=}\n{d=}')
```

```
a=array([0, 1, 2, 3])
b=array([4, 5, 6, 7])
c=array([ 8,  9, 10, 11])
d=array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [39]:

```
# ERROR: arrays a,b and c does not have dimension 1
#np.concatenate((a,b,c), axis=1)
```

In [40]:

```
a2,b2,c2 = a[newaxis,:],b[newaxis,:],c[newaxis,:]
print(f'{a2=}\n{b2=}\n{c2=}')
```

```
a2=array([[0, 1, 2, 3]])
b2=array([[4, 5, 6, 7]])
c2=array([[ 8,  9, 10, 11]])
```

In [41]:

```
np.concatenate((a2,b2,c2))
```

Out[41]:

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

In [42]:

```
np.concatenate((a2,b2,c2), axis=0)
```

Out[42]:

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

In [43]:

```
np.concatenate((a2,b2,c2), axis=1)
```

Out[43]:

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

- `vstack` and `hstack` - 2 (or more) dimensional arrays
  - `vstack((a1, a2, ...))` == `concatenate((a1, a2, ...), axis=0)`
  - `hstack((a1, a2, ...))` == `concatenate((a1, a2, ...), axis=1)`

In [44]:

```
a = np.arange(0,8).reshape((2,4))
b = np.arange(8,16).reshape((2,4))
c = np.hstack((a,b))
d = np.concatenate((a,b), axis=1)
e = np.vstack((a,b))
f = np.concatenate((a,b), axis=0)
print(f'{a=}\n{b=}\n{c=}\n{d=}\n{e=}\n{f=}')
```

```
a=array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
b=array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
c=array([[ 0,  1,  2,  3,  8,  9, 10, 11],
       [ 4,  5,  6,  7, 12, 13, 14, 15]])
d=array([[ 0,  1,  2,  3,  8,  9, 10, 11],
       [ 4,  5,  6,  7, 12, 13, 14, 15]])
e=array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
f=array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

- `vstack` and `hstack` - 1 dimensional arrays
  - `vstack((a1, a2, ...))` == `concatenate((a1[newaxis,:], a2[newaxis,:], ...))`
  - `hstack((a1, a2, ...))` == `concatenate((a1, a2, ...))`

In [45]:

```
a = np.arange(0,4)
b = np.arange(4,8)
c = np.hstack((a,b))
d = np.concatenate((a,b))
e = np.vstack((a,b))
f = np.concatenate((a[newaxis,:],b[newaxis,:]))
print(f'{a=}\n{b=}\n{c=}\n{d=}\n{e=}\n{f=}')
```

```
a=array([0, 1, 2, 3])
b=array([4, 5, 6, 7])
c=array([0, 1, 2, 3, 4, 5, 6, 7])
d=array([0, 1, 2, 3, 4, 5, 6, 7])
e=array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
f=array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

- `split(a, split_points, axis=0)`
    - `a` → the array to split
    - `split_points`
        - N (int) → array is divided into N equal arrays (or fails)
        - integer seq → split indexes (points)
    - `axis` → the axis along which to split

In [46]:

```
a = np.arange(42).reshape((6,7))
upper,center,lower = np.split(a,3)
print(f'{a=}\n{upper=}\n{center=}\n{lower=}')
```

```
a=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, 32, 33, 34],
       [35, 36, 37, 38, 39, 40, 41]])
upper=array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13]])
center=array([[14, 15, 16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25, 26, 27]])
lower=array([[28, 29, 30, 31, 32, 33, 34],
       [35, 36, 37, 38, 39, 40, 41]])
```

In [47]:

```
a = np.arange(42).reshape((6,7))
# ERROR: array split does not result in an equal division
#upper,lower = np.split(a,2,axis=1)
```

In [48]:

```
a = np.arange(36).reshape((6,6))
upper,center,lower = np.split(a,(1,3))
print(f'{a=}\n{upper=}\n{center=}\n{lower=}')
```

```
a=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, 32, 33, 34, 35]])
upper=array([[0, 1, 2, 3, 4, 5]])
center=array([[ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17]])
lower=array([[18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35]])
```

In [49]:

```
a = np.arange(0,14).reshape((2,7))
left,center,right = np.split(a,(1,3), axis=1)
print(f'{a=}\n{left=}\n{center=}\n{right=}')
```

```
a=array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13]])
left=array([[0],
       [7]])
center=array([[1, 2],
       [8, 9]])
right=array([[ 3,  4,  5,  6],
       [10, 11, 12, 13]])
```

- `vsplit` and `hsplit` - 2 (or more) dimensional arrays
    - `vsplit(a,ii)` == `split(a, ii, axis=0)`
    - `hsplit(a,ii)` == `split(a, ii, axis=1)`
- `hsplit` - 1 dimensional arrays
    - `hsplit(a,ii)` == `split(a, ii)`

In [50]:

```
a = np.arange(16).reshape((4,4))
b,c = np.vsplit(a,2)
d,e = np.split(a,2, axis=0)
print(f'{a=}\n{b=}\n{c=}\n{d=}\n{e=}')
```

```
a=array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
b=array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
c=array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
d=array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
e=array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [51]:

```
a = np.arange(16).reshape((4,4))
b,c = np.hsplit(a,2)
d,e = np.split(a,2, axis=1)
print(f'{a=}\n{b=}\n{c=}\n{d=}\n{e=}')
```

```
a=array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
b=array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
c=array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
d=array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
e=array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

In [52]:

```
a = np.arange(16)
b,c = np.hsplit(a,2)
d,e = np.split(a,2)
print(f'{a=}\n{b=}\n{c=}\n{d=}\n{e=}')
```

```
a=array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15
b=array([0, 1, 2, 3, 4, 5, 6, 7])
c=array([ 8,  9, 10, 11, 12, 13, 14, 15])
d=array([0, 1, 2, 3, 4, 5, 6, 7])
e=array([ 8,  9, 10, 11, 12, 13, 14, 15])
```

UFuncs - Universal Functions

Function designed to perform element-wise operations on arrays. Main features:

- **Speed and Efficiency**: Instead of looping over arrays to perform the *same* computation on each element, use single highly optimized functions.
- **Broadcasting**: support for different but *compatible* shape arrays.
- **Type casting**: support for different data types

Arithmetic operations

In [53]:

```
x = np.arange(8)
print("     x =", x)
print(" x + 5 =", x + 5)
print(" x - 5 =", x - 5)
print("    -x =", -x)
print(" x * 2 =", x * 2)
print(" x / 2 =", x / 2)
print("x // 2 =", x // 2)
print(" x % 2 =", x % 2)
print("x ** 2 =", x ** 2)
```

```
     x = [0 1 2 3 4 5 6 7]
 x + 5 = [ 5  6  7  8  9 10 11 12]
 x - 5 = [-5 -4 -3 -2 -1  0  1  2]
    -x = [ 0 -1 -2 -3 -4 -5 -6 -7]
 x * 2 = [ 0  2  4  6  8 10 12 14]
 x / 2 = [0.  0.5 1.  1.5 2.  2.5 3.  3.5]
x // 2 = [0 0 1 1 2 2 3 3]
 x % 2 = [0 1 0 1 0 1 0 1]
x ** 2 = [ 0  1  4  9 16 25 36 49]
```

Each arithmetic operator is in fact a wrapper around an specific function built into NumPy

Operator	Equivalent ufunc	Description
`+`	`np.add`	Addition (e.g., `1 + 1 = 2`)
`-`	`np.subtract`	Subtraction (e.g., `3 - 2 = 1`)
`-`	`np.negative`	Unary negation (e.g., `-2`)
`*`	`np.multiply`	Multiplication (e.g., `2 * 3 = 6`)
`/`	`np.divide`	Division (e.g., `3 / 2 = 1.5`)
`//`	`np.floor_divide`	Floor division (e.g., `3 // 2 = 1`)
`**`	`np.power`	Exponentiation (e.g., `2 ** 3 = 8`)
`%`	`np.mod`	Modulus/remainder (e.g., `9 % 4 = 1`)

Trigonometric functions

In [54]:

```
theta = np.linspace(0, np.pi, 3)
print(f'{x=}')
print(f'{np.sin(theta)=}')
print(f'{np.cos(theta)=}')
print(f'{np.tan(theta)=}')
x = [-1, 0, 1]
print(f'{np.arcsin(x)=}')
print(f'{np.arccos(x)=}')
print(f'{np.arctan(x)=}')
```

```
x=array([0, 1, 2, 3, 4, 5, 6, 7])
np.sin(theta)=array([0.0000000e+00, 1.0000000e+00, 1.2246468e-16])
np.cos(theta)=array([ 1.000000e+00,  6.123234e-17, -1.000000e+00])
np.tan(theta)=array([ 0.00000000e+00,  1.63312394e+16, -1.22464680e-16]
np.arcsin(x)=array([-1.57079633,  0.        ,  1.57079633])
np.arccos(x)=array([3.14159265, 1.57079633, 0.        ])
np.arctan(x)=array([-0.78539816,  0.        ,  0.78539816])
```

More mathematical functions

- `abs()` , `np.absolute()` or `np.abs()`

In [55]:

```
x = np.array([-2, -1, 0, 1, 2])
print(f'{abs(x)=}')
print(f'{np.absolute(x)=}')
print(f'{np.abs(x)=}')
```

```
abs(x)=array([2, 1, 0, 1, 2])
np.absolute(x)=array([2, 1, 0, 1, 2])
np.abs(x)=array([2, 1, 0, 1, 2])
```

- `np.exp()` , `np.exp2()` and `np.power()`

In [56]:

```
x = np.arange(3)
print("     x =", x)
print("   e^x =", np.exp(x))
print("   2^x =", np.exp2(x))
print("3^x =", np.power(3, x))       # keeps int dtype
print("3.14^x =", np.power(3.14, x))
```

```
     x = [0 1 2]
   e^x = [1.         2.71828183 7.3890561 ]
   2^x = [1. 2. 4.]
3^x = [1 3 9]
3.14^x = [1.     3.14   9.8596]
```

- `np.log()` , `np.log2()` and `np.log10()`

In [57]:

```
x = np.array([1,2,np.e,10])
print("       x =", x)
print(" log2(x) =", np.log2(x))
print("  log(x) =", np.log(x))
print("log10(x) =", np.log10(x))
```

```
      x = [ 1.          2.          2.71828183 10.        ]
 log2(x) = [0.         1.         1.44269504 3.32192809]
  log(x) = [0.         0.69314718 1.         2.30258509]
log10(x) = [0.         0.30103    0.43429448 1.        ]
```

- np.sqrt()

In [58]:

```
x = np.linspace(0,100,5)
print("      x =", x)
print("sqrt(x) =", np.sqrt(x))
```

```
      x = [  0.   25.   50.   75.  100.]
sqrt(x) = [ 0.          5.          7.07106781  8.66025404 10.        ]
```

- np.round()

In [59]:

```
x = np.linspace(0,9.3,7)
print("       x =", x)
print("round(x) =", np.round(x))
```

```
       x = [0.   1.55 3.1  4.65 6.2  7.75 9.3 ]
round(x) = [0. 2. 3. 5. 6. 8. 9.]
```

Aggregations Functions

- Takes an array and returns a single scalar (or *smaller* array)
- sum, median, max, min, ...

- `np.sum(a)` or `a.sum()`
- **DO NOT** use built-in `sum` function

In [60]:

```
n = 100000
x = np.arange(1,n+1)
sum(x) == np.sum(x) == x.sum() == n*(n+1)/2
```

Out[60]:

```
True
```

In [61]:

```
#%timeit sum(x)
#%timeit np.sum(x)
#%timeit x.sum()
```

- `np.min(a)` or `a.min()` and `np.max(a)` or `a.max()`
- **DO NOT** use built-in `min` or `max` functions

In [62]:

```
x = np.random.rand(100000)
min(x) == np.min(x) == x.min()
```

Out[62]:

```
True
```

In [63]:

```
#%timeit min(x)
#%timeit np.min(x)
#%timeit x.min()
```

More aggregations functions

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements

Function Name	NaN-safe Version	Description
`np.any`	N/A	Evaluate whether any elements are true

Multi dimensional aggregates

- By default, the aggregate over the entire array.
- An additional argument (`axis`) specifying the axis along which the aggregate is computed

In [64]:

```
a = np.arange(20).reshape((4,5))
print(f'{a=}')
print(f'{a.sum()=}')
print(f'{a.sum(axis=0)=}')
print(f'{a.sum(axis=1)=}')
```

```
a=array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
a.sum()=190
a.sum(axis=0)=array([30, 34, 38, 42, 46])
a.sum(axis=1)=array([10, 35, 60, 85])
```

Broadcasting

The ability to apply binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.

In [65]:

```
a = np.arange(10)
b = a[::-1]
print(f'{  a = }')
print(f'{  b = }')
print(f'{a+b = }')
print(f'{a+3 = }') # broadcasting
```

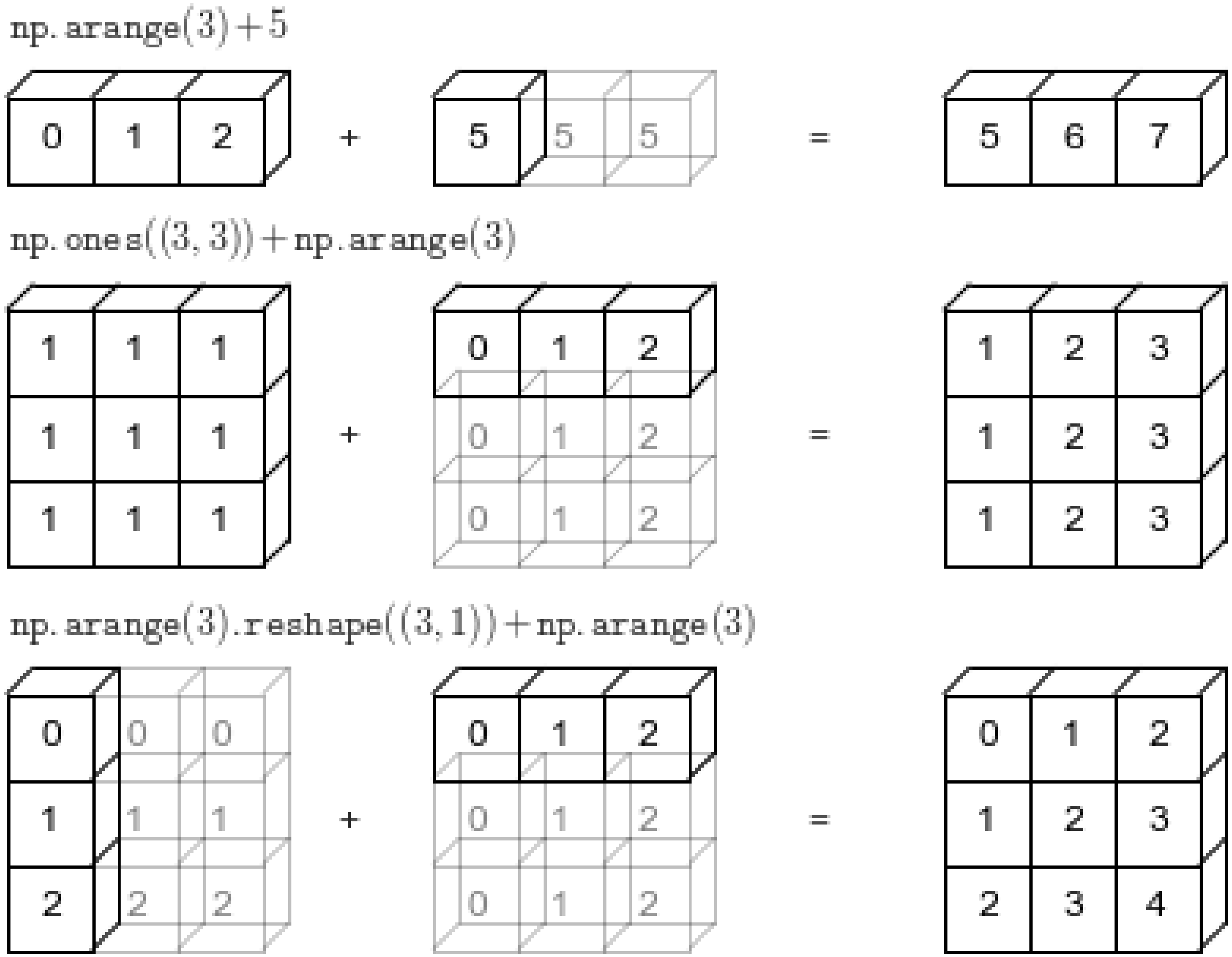
```
  a = array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  b = array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
a+b = array([9, 9, 9, 9, 9, 9, 9, 9, 9, 9])
a+3 = array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [66]:

```
a = np.array([10,20,30,40,50])
a2 = a.reshape((1,5))
b = np.arange(10).reshape(2,5)
c = np.array([[-1],[-10]])
print(f'{   a = }  {a2 = }')
print(f'{   b = }')
print(f'{   c = }')
print(f'{ a+a = }')
print(f'{ a+3 = }') # broadcasting
print(f'{ a+b = }') # broadcasting
print(f'{ a+c = }') # broadcasting
print(f'{a2+c = }') # broadcasting
```

```
   a = array([10, 20, 30, 40, 50])  a2 = array([[10, 20, 30, 40, 50]])
   b = array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
   c = array([[ -1],
       [-10]])
 a+a = array([ 20,  40,  60,  80, 100])
 a+3 = array([13, 23, 33, 43, 53])
 a+b = array([[10, 21, 32, 43, 54],
       [15, 26, 37, 48, 59]])
 a+c = array([[ 9, 19, 29, 39, 49],
       [ 0, 10, 20, 30, 40]])
a2+c = array([[ 9, 19, 29, 39, 49],
       [ 0, 10, 20, 30, 40]])
```

Broadcasting - A visual example



Broadcasting Rules

Given two arrays `x` and `y` with dimensions $(d_1^x, \ldots, d_n^x)$ and $(d_1^y, \ldots, d_m^y)$

1. if $n \neq m$, the shape of the one with fewer dimensions is padded with *ones* on its left side.
    - $n < m \rightarrow (1_1, \ldots, 1_{m-n}, d_1^x, \ldots, d_n^x)$
    - $n > m \rightarrow (1_1, \ldots, 1_{n-m}, d_1^y, \ldots, d_m^y)$
    - After the padding, shapes of x and y are $(d_1^x, \ldots, d_k^x)$ and $(d_1^y, \ldots, d_k^y)$, where $k = \max(n, m)$
2. If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched (replicated) to match the other shape.
    - After the stretching, $\forall i \;\; d_i^x = 1 \iff d_i^y = 1$
3. If in any dimension the sizes disagree ($\exists i \; where \; d_i^x \neq d_i^y$), an error is raised
    - If no error is raised, shape of x and y is $(d_1, \ldots, d_k)$
4. All dimensions match, and shape of result is $(d_1, \ldots, d_k)$

Example 1

In [67]:
```
M = np.ones((2, 3))
a = np.arange(3)
```

- `M.shape == (2, 3)`
- `a.shape == (3,)`

- Rule1: pad (increase number of dimensions)
  - `M.shape` → `(2, 3)`
  - `a.shape` → `(1, 3)`
- Rule2: stretch (`1` → `j`)
  - `M.shape` → `(2, 3)`
  - `a.shape` → `(2, 3)`
- Rule3: check dimensions → **OK**

In [68]:
```
M + a
```

Out[68]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Example 2

In [69]:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

- `M.shape == (3, 1)`
- `a.shape == (3,)`

- Rule1: pad (increase number of dimensions)
    - `M.shape` → `(3, 1)`
    - `a.shape` → `(1, 3)`
- Rule2: stretch (`1` → `j`)
    - `M.shape` → `(3, 3)`
    - `a.shape` → `(3, 3)`
- Rule3: check dimensions → **OK**

In [70]:

```
a + b
```

Out[70]:

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

Example 3

In [71]:

```
M = np.ones((3, 2))
a = np.arange(3)
```

- `M.shape == (3, 2)`
- `a.shape == (3,)`

- Rule1: pad (increase number of dimensions)
    - `M.shape` → `(3, 2)`
    - `a.shape` → `(1, 3)`
- Rule2: stretch (`1` → `j`)
    - `M.shape` → `(3, 3)`
    - `a.shape` → `(3, 3)`
- Rule3: check dimensions → **ERROR**

In [72]:

```
# ERROR: could not broadcast
#M + a
```

Booleans and Masking

- Comparison operators/ufuncs obtain element wise booleans that result in boolean arrays
- Boolean arrays (**masks**) can be used to do selective computations on arrays.
- Highly optimized vectorized operations

In [73]:

```
x = np.arange(7)
mask = x%3==0
print(f'{        x = }')
print(f'{     mask = }')
print(f'{  x[mask] = }')
print(f'{x[x%3==0] = }')
x[x%3==0] = -10
print(f'{        x = }')
```

```
        x = array([0, 1, 2, 3, 4, 5, 6])
     mask = array([ True, False, False,  True, False, False,  True])
  x[mask] = array([0, 3, 6])
x[x%3==0] = array([0, 3, 6])
        x = array([-10,   1,   2, -10,   4,   5, -10])
```

Each comparison operator is in fact a wrapper around an specific function built into NumPy

Operator	Equivalent ufunc
`>`	`np.greater`
`<`	`np.less`
`>=`	`np.greater_equal`
`<=`	`np.less_equal`
`==`	`np.equal`
`!=`	`np.not_equal`

Python's bitwise logic operators (`&`, `|`, `^`, and `~`)

- Element-wise boolean operations (similar to arithmetic operators)

In [74]:

```
x = np.arange(7)
mask = (x%3==0) & (x<5)
print(f'{              x = }')
print(f'{           mask = }')
print(f'{        x[mask] = }')
print(f'{x[(x%3==0)&(x<5)] = }')
```

```
              x = array([0, 1, 2, 3, 4, 5, 6])
           mask = array([ True, False, False,  True, False, False, Fa
        x[mask] = array([0, 3])
x[(x%3==0)&(x<5)] = array([0, 3])
```

Each logic operator is in fact a wrapper around an specific function built into NumPy:

Operator	Equivalent ufunc
`&`	`np.bitwise_and`
`|`	`np.bitwise_or`
`^`	`np.bitwise_xor`
`~`	`np.bitwise_not`

- `np.any(a)` or `a.any()` → aggregated `True` | `False`
- `np.all(a)` or `a.all()` → aggregated `True` | `False`
- `axis` parameter → aggregation along a particular axis

In [75]:

```
x = np.arange(10).reshape((2,5))
print(f'{x=}')
print(f'{x>7=}')
print(f'{np.any(x>7)=}')
print(f'{(x>7).any()=}')
print(f'{np.all(x>7)=}')
print(f'{(x>7).all()=}')
print(f'{np.any(x>7, axis=0)=}')
print(f'{np.any(x>7, axis=1)=}')
```

```
x=array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
x>7=array([[False, False, False, False, False],
       [False, False, False,  True,  True]])
np.any(x>7)=True
(x>7).any()=True
np.all(x>7)=False
(x>7).all()=False
np.any(x>7, axis=0)=array([False, False, False,  True,  True])
np.any(x>7, axis=1)=array([False,  True])
```

Boolean arrays can be summed

- `True==1`, `False==0`
- `np.sum(a)` or `a.sum()` → aggregated `True` | `False`
- `axis` parameter → aggregation along a particular axis
- **DO NOT** use built-in `sum` function

In [76]:

```
x = np.arange(10).reshape((2,5))
print(f'{x=}')
print(f'{x>7=}')
print(f'{np.sum(x>7)=}')
print(f'{(x>7).sum()=}')
print(f'{np.sum(x>7, axis=0)=}')
print(f'{np.sum(x>7, axis=1)=}')
```

```
x=array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
x>7=array([[False, False, False, False, False],
       [False, False, False,  True,  True]])
np.sum(x>7)=2
(x>7).sum()=2
np.sum(x>7, axis=0)=array([0, 0, 0, 1, 1])
np.sum(x>7, axis=1)=array([0, 2])
```

Fancy Indexing

- Use arrays of indices to access multiple array elements
- Returns a **copy** of the data (vs slicing view)

In [77]:

```
a = np.random.randint(100,size=(10,))
ii1 = [1, 8, 3, 5]
ii2 = np.array([1, 8, 3, 5])
print(f'{     a = }')
print(f'{a[ii1] = }')
print(f'{a[ii2] = }')
```

```
     a = array([90, 57, 33, 63, 92, 93, 16, 29, 59, 51])
a[ii1] = array([57, 59, 63, 93])
a[ii2] = array([57, 59, 63, 93])
```

Fancy indexing can be used with multi-dimensional arrays

- Use arrays of indices to access multiple array elements
- **Sequential selection**: returns 1-dimensional array

In [78]:

```
a = np.arange(15).reshape((3,5))
ii = [1, 0, 2, 2]
jj = [1, 4, 3, 2]
print(f'{    a = }')
print(f'{a[ii,jj] = }')
```

```
    a = array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
a[ii,jj] = array([ 6,  4, 13, 12])
```

Fancy indexing can be combined with simple indexes

- kind of *index broadcasting*

In [79]:

```
a = np.arange(16).reshape((4,4))
ii = [1, 3, 2]
print(f'{    a = }')
print(f'{a[1,ii] = }')
print(f'{a[ii,2] = }')
```

```
    a = array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
a[1,ii] = array([5, 7, 6])
a[ii,2] = array([ 6, 14, 10])
```

Subarrays can be obtained

- Providing only first dimension fancy indexing → rows
- Combining with slicing → columns

In [80]:

```
a = np.arange(15).reshape((3,5))
ii = [2, 0]
jj = [1, 4, 3]
print(f'{    a = }')
print(f'{a[ii] = }')
print(f'{a[:,jj] = }')
```

```
    a = array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
a[ii] = array([[10, 11, 12, 13, 14],
       [ 0,  1,  2,  3,  4]])
a[:,jj] = array([[ 1,  4,  3],
       [ 6,  9,  8],
       [11, 14, 13]])
```