

09 – Arrays

Bálint Aradi

Scientific Programming in Python (2024)

<https://atticlectures.net/scipro/python-2024/>

Modules

- Declarations (e.g functions) in a separate file
- Better **code structuring**
- Accessible in many scripts/notebooks (**reusability**)
- Module must be **imported** before first use

```
import numpy
```

- Content is **accessed by prefix notation**

module.content

```
grid = numpy.linspace(-1, 1, 3)
grid
array([-1.,  0.,  1.])
```

- Module name prefix can be changed at import

```
import numpy as np
grid = np.linspace(-1, 1, 3)
```

- If only a few entities are needed, they can be **imported directly** and used without prefix

```
from numpy import linspace
linspace(-1, 1, 3)
```

- Importing all entities from a module directly is possible, but **discouraged**

```
from numpy import *
```

Python Standard Library

- Bundled with the Python interpreter
- [Python Standard Library documentation](#)

SciPy stack

- *De facto* standard mathematical / statistical / scientific (third party) modules

Numpy: arrays & basic numerical routines

<https://docs.scipy.org/doc/numpy/>

Scipy: additional mathematical routines

<https://docs.scipy.org/doc/scipy/>

Matplotlib: powerful graphical plotting routines

<https://matplotlib.org/stable/users/index.html>

Third party modules

- Not part of the official Standard Library
- Must be installed additionally to the Python interpreter (Conda, PyPI, packages)

Pandas: data analysis toolkit

<http://pandas.pydata.org/pandas-docs/stable/>

SymPy: symbolic mathematics

<http://docs.sympy.org/latest/index.html>

```
conda install numpy scipy matplotlib pandas sympy
```

Arrays

Array = Multi-dimensional storage of elements with the same type (matrix)

```
import numpy as np
aa = np.array([1, 2, 3])
print(aa)
```

[1 2 3]

```
bb = np.array([[1, 2, 3], [4, 5, 6]])
print(bb)
```

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

Advantages (compared to lists)

- Elements are stored sequentially in memory
 - Fast access
(assuming the right access pattern)
- Optimized functions for array manipulation can speed up algorithm by orders of magnitude

Disadvantages (compared to lists)

- All elements must have the same type
- All strides in a multi-dimensional array must have the same length

Creating arrays

`numpy.array(expression, dtype=type)`

Creates an array of given type and storage order from an expression

```
np.array([1.0, 2.0], dtype=float) → [ 1.  2.]
```

`numpy.empty(shape, dtype=type)`

`numpy.zeros(shape, dtype=type)`

`numpy.ones(shape, dtype=type)`

Array with uninitialized elements / zeros / ones of given shape, type (and storage order)

```
np.ones((2, 3), dtype=float)
[[ 1.  1.  1.]
 [ 1.  1.  1.]
```

- Data type: either intrinsic type (int, float, etc.) or numpy provided one (np.float32, np.float64, etc.)
- Storage order: C or Fortran (row-major / column-major)

Element storage ordering

- Default is **row-major** storage (like in C):

first index grows slowest, last index grows fastest:

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

- Array manipulation is fastest, if elements are accessed in the same sequence as stored in memory (**cache coherence**)

- Optionally, **column-major** storage format (like in Fortran)

first index grows fastest, last index grows slowest:

```
bb = np.array([[1, 2, 3], [4, 5, 6]], order='F')
```

C-storage

	Memory
bb[0,0]	1
bb[0,1]	2
bb[0,2]	3
bb[1,0]	4
..	..



F-storage

	Memory
bb[0,0]	1
bb[1,0]	4
bb[0,1]	2
bb[1,1]	5
..	..



Querying size and shape

- Object variable **size** contains the total **number of elements in the array**

```
bb = np.array([[1, 2, 3], [4, 5, 6]])  
bb.size
```

6

- Object variable **shape** contains a tuple with **number of elements along each axis**

```
bb.shape
```

(2, 3)

- Array **shape can be changed**, provided **size remains constant**
- Order of elements in the memory remains unchanged

```
bb.shape = (3, 2)  
bb
```

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

Array element access

- Elements accessed as with nested lists
- First element indexed by 0
- Indices for multidimensional arrays collected to **index tuple**

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

```
bb[1]      [[1 2 3]  
           [4 5 6] → [4 5 6]  
           [7 8 9]]
```

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

```
bb[1][1] [7 8 9]]
```

- **Slices** analogously to lists as **[from:to:step]**
- Along arbitrary dimensions / axis

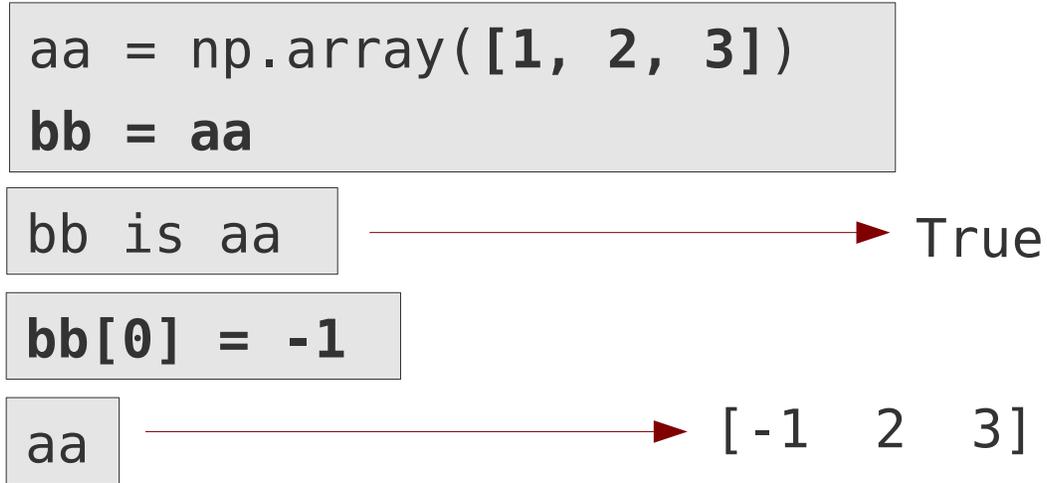
```
bb[0, :]   [[1 2 3] → [1 2 3]  
           [4 5 6]  
           [7 8 9]]
```

```
bb[:, 0]   [[1 2 3] → [1 4 7]  
           [4 5 6]  
           [7 8 9]]
```

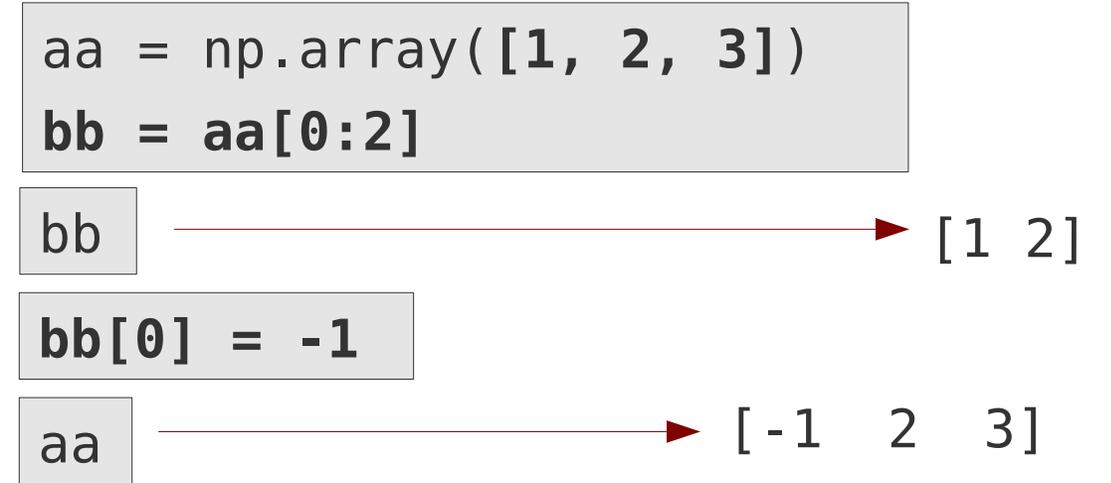
```
bb[0:3:2, 0:2] [[1 2] → [[1 2]  
                [4 5] → [7 8]]  
                [7 8] →
```

Modifying arrays

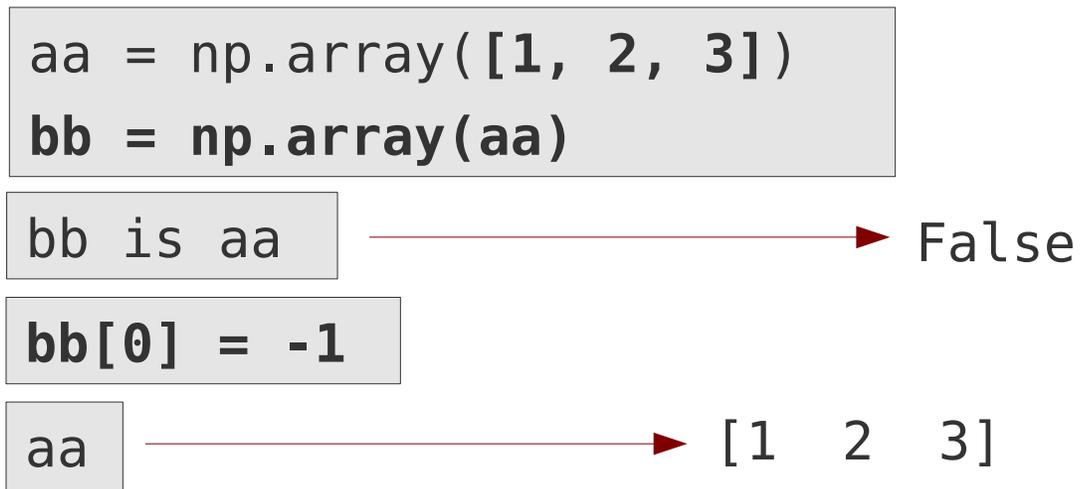
- Arrays are **mutable**, be aware of **side effects**



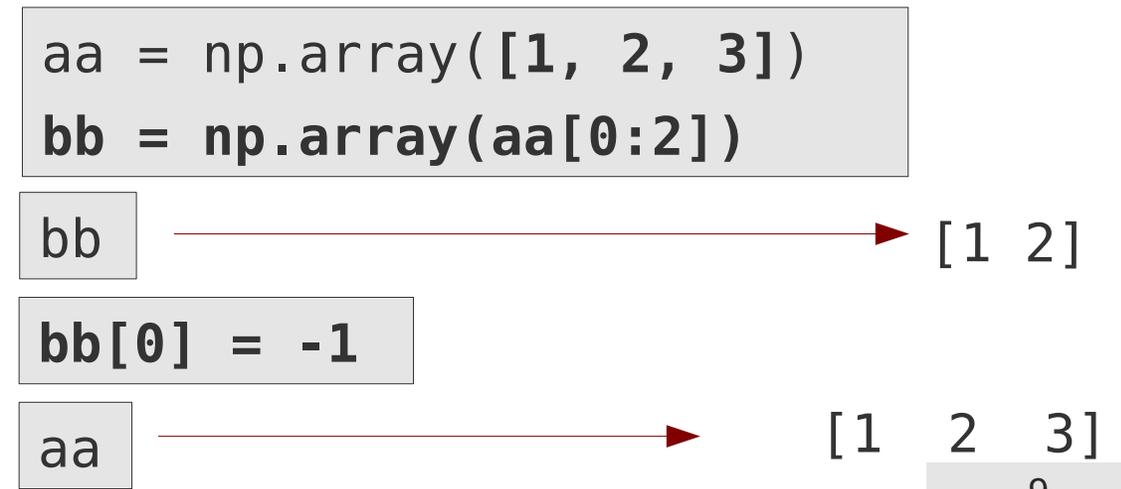
- Arrays slices are pointers/views, no true copies



- True **copy** can be enforced via **array()**



- True **copy** can be enforced via **array()**



Scalar operations with arrays

Arithmetic / logical operators

- Operations applied **elementwise**
- One of the operands must be either a scalar or operands must have identical shapes*
- Result has same shape as operand(s)

```
aa = np.array([1, 2, 3])
bb = np.array([5, 4, 3])
```

```
aa * bb           [5 8 9]
```

```
2.0 * aa         [2. 4. 6.]
```

```
aa**2           [1 4 9]
```

```
aa == bb        [False False True]
```

```
aa < bb         [True True False]
```

* or shapes which can be made identical via **broadcasting**

Universal functions (ufuncs)

- Scalar functions applicable on arrays
- Ufuncs are applied **elementwise**
- Result has same shape as argument

```
aa = np.array([0.0, np.pi / 2,
               -np.pi / 2])
np.sin(aa)
[0.0000e+00  1.0000e+00 -1.0000e+00]
```

- Numpy module contains many **useful ufuncs**
 - Square root function (**sqrt**)
 - Trigonometric functions (**sin, cos, ...**)
 - Rounding functions (**floor, ceil, ...**)
 - etc.

Broadcasting

- If arrays in binary operators have different number of dimensions, numpy tries to **extend the one with less dimensions** by broadcasting:
 - **Extra dimensions inserted before** existing one(s), to equal number of dimensions
 - Array **repeated** along new dimensions to match correct shape
 - Shapes must be compatible!

```
aa = np.array([[1, 2, 3], [4, 5, 6]])
bb = np.array([-2, -3, -4])
aa * bb
```

→

1 2 3	*	-2 -3 -4	→	[[-2 -6 -12]
4 5 6		-2 -3 -4		[-8 -15 -24]]

- With **numpy.newaxis** extra dimension(s) can be **inserted at arbitrary position**

```
cc = np.array([-1, -2])
aa * cc[:, np.newaxis]
```

→

1 2 3	*	-1 -1 -1	→	[[-1 -2 -3]
4 5 6		-2 -2 -2		[-8 -10 -12]]

Matrix multiplication, array reduction

Matrix multiplication

- Arrays can be matrix-multiplied with the **@ operator** (equivalent with **numpy.matmul()**)

```
aa = np.array([[1, 2], [3, 4]])  
bb = np.array([5, 6])
```

```
aa @ bb    [17 39]
```

```
bb @ aa    [23 34]
```

```
bb @ bb    61
```

- numpy.dot()** does also matrix multiplication
- For arrays with more than 2 dimensions, **numpy.dot()** and **numpy.matmul()** / **@** behave differently

Array reduction

- Reduces array to **scalar** via given operation
- numpy.sum()**, **numpy.product()**, etc.

```
vec = np.array([1.0, 2.0, 3.0])  
np.sqrt(np.sum(vec**2)) → 3.7416...
```

- Reduction can be restricted to **selected axis**
- Resulting shape as before but without selected/reduced axis

```
aa = np.array([[1, 2], [3, 4]])  
np.sum(aa, axis=0)
```

```
[ [1, 3],  
  [2, 4] ] → [4 6]
```

Iterating over arrays

Arrays behave in iterations **as (nested) lists**:

- Iteration over 1D-array delivers each element
- Iteration over 2D-array delivers the rows of the 2D-array as 1D-arrays
- :

```
vec = np.array([1, 2, 3])  
for ind, elem in enumerate(vec):  
    print(f"{ind}: {elem}")
```

→ 0: 1
1: 2
2: 3

```
aa = np.array([[1, 2, 3], [4, 5, 6]])  
for ind, row in enumerate(aa):  
    print(f"{ind}: {row}")
```

→ 0: [1 2 3]
1: [4 5 6]