Chapter 5

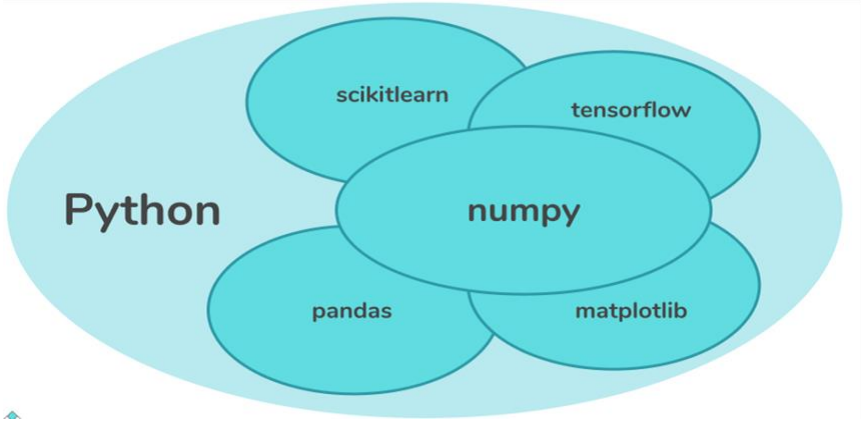
Numeric Computing with Numpy

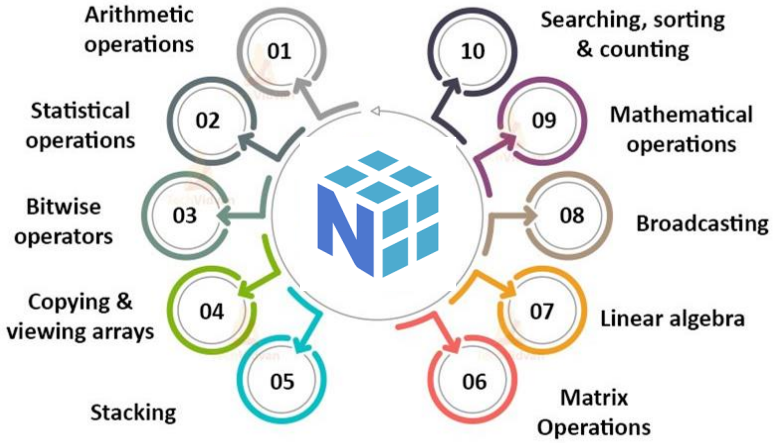
Chapter Content

- Introduction to NumPy
- Why should use Numpy?
- Numpy Array
- Numpy Linear Algebra
- Numpy Matrix Library matlib
- I/O with Numpy

Introduction to NumPy

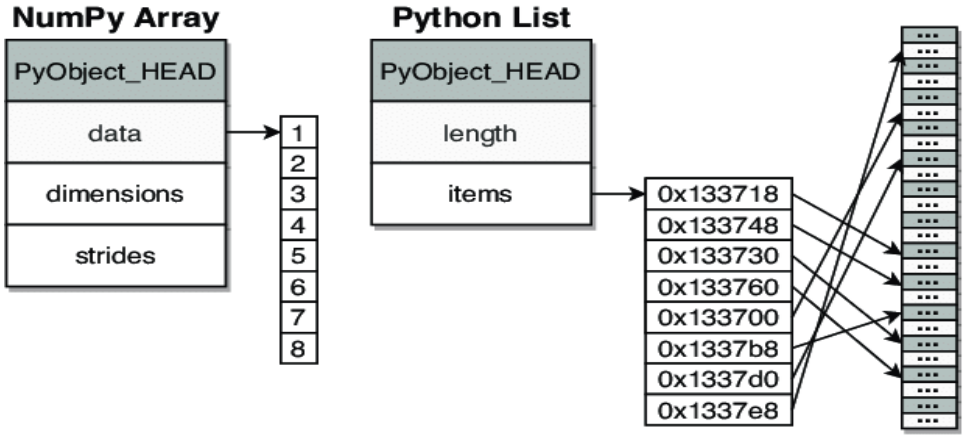
- NumPy is short for Numerical Python
- Array oriented computing
- Efficiently implemented multi-dimensional arrays
- Used for scientific computing.





Why should we use Numpy?

- Convenient interface for working with multi-dimensional array data structures efficiently (ndarray).
- Less memory to store the data.
- High computational efficiency



Numpy Getting Started

- Installing Numpy: pip install numpy
- Import Numpy: import numpy
- Alias of Numpy: import numpy as np
- Check Numpy version: np.__version__

Numpy Data Types

- supports a much greater variety of numerical types than Python does

Boolean	bool_
Integer	int_, intc, intp, int8, int16, int32, int64
Unsigned Integer	uint8, uint16, uint32, uint64
Float	float_, float 16, float32, float 64
Complex	complex_, complex64, complex128

- ndarray (N-Dimensional array)

0-D Array

1

```
np.array(1)
```

1-D Array

1	2	3

Vector

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

2-D Array

Axis 0

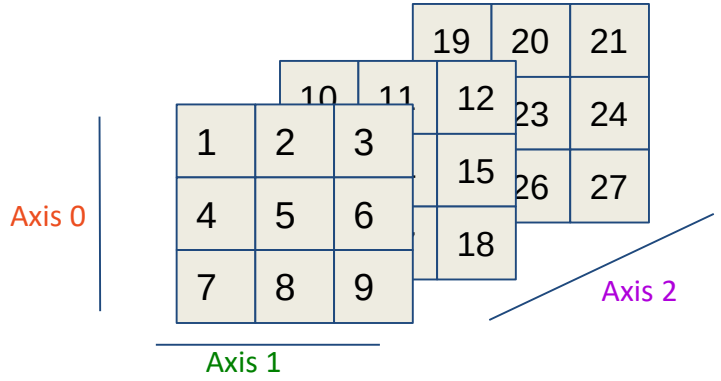
1	2	3
4	5	6
7	8	9

Matrix

Axis 1

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

3-D Array




```
np.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]]])
```

Numpy Array

- Numpy Array Creation
- Numpy Array Indexing
- Numpy Array Slicing
- Numpy Arithmetic Operations
- Numpy Arithmetic Functions
- Numpy Array Manipulation Functions
- Numpy Broadcasting
- Numpy Statistical Operations

Numpy Array Creation

- Using the function array()

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
```

- Converting from lists, tuples.

```
list1 = [1, 2, 3, 4, 5]
arr = np.array(list1)
```

```
tuple1 = (1, 2, 3, 4, 5)
arr = np.array(tuple1)
```

- Using special Numpy functions:
  - empty(shape, dtype)
  - ones(shape, dtype)
  - zeros(shape,dtype)
  - arange(start,stop,step,dtype)
  - random.random(size)
  - full(shape, fill_value,dtype)
  - eyes(nrow,[ncol],dtype)

```
Ex: arr = np.empty(2, dtype=int)
    arr = np.zeros(2, dtype=int)
```

Numpy Array Attributes

ndarray.shape

return the size of array

(3,3)

ndarray.ndim

return number of array dimension

2

ndarray.dtype

return data type of elements in the array

int64

1	2	3
4	5	6
7	8	9

ndarray.size

return number of elements in the array

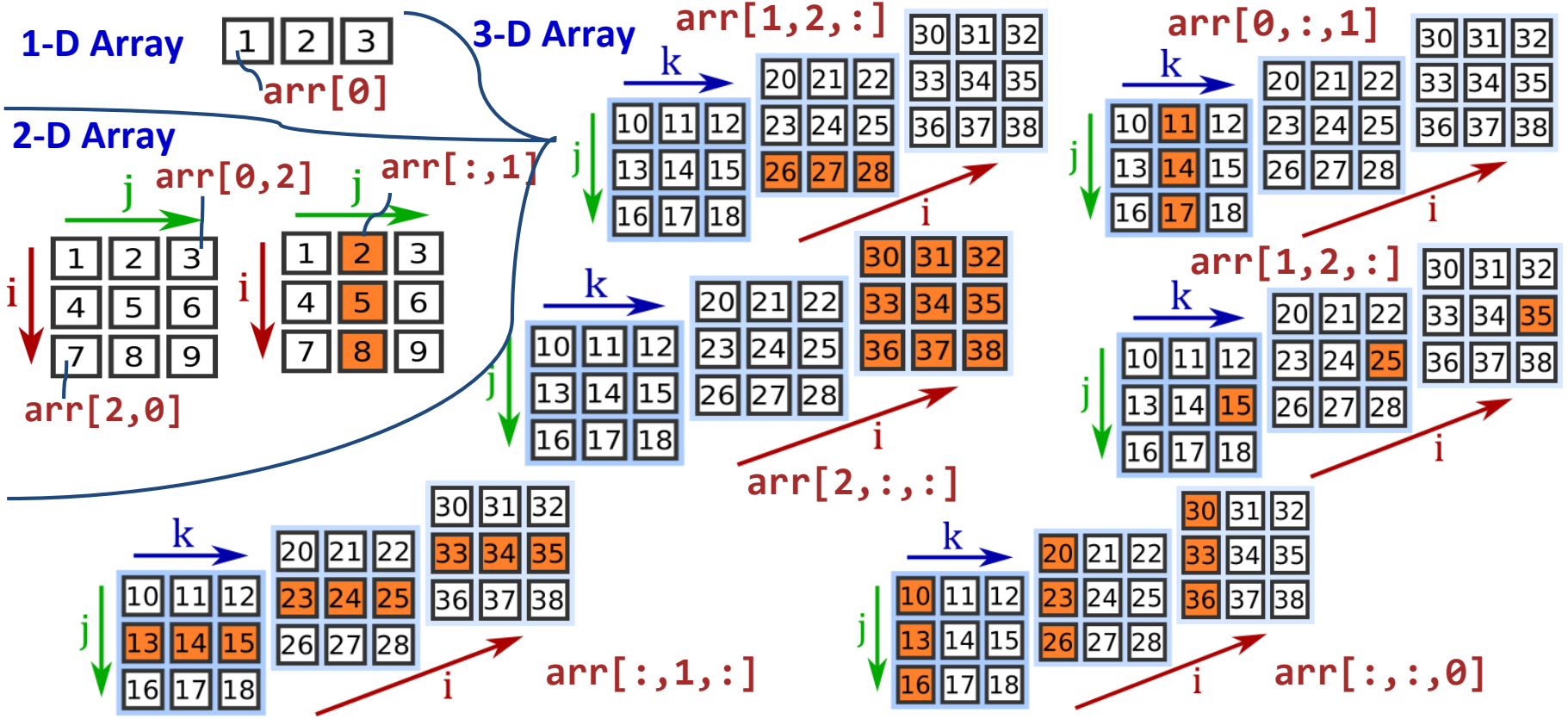
9

ndarray.itemsize

return the size (in bytes) of elements in the array

8

Numpy Array Indexing



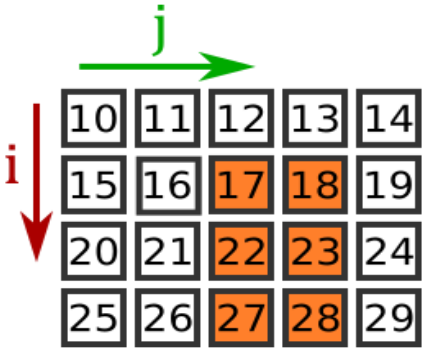
Numpy Array Slicing

1-D Array

`arr[:1]`

`[start:end]`

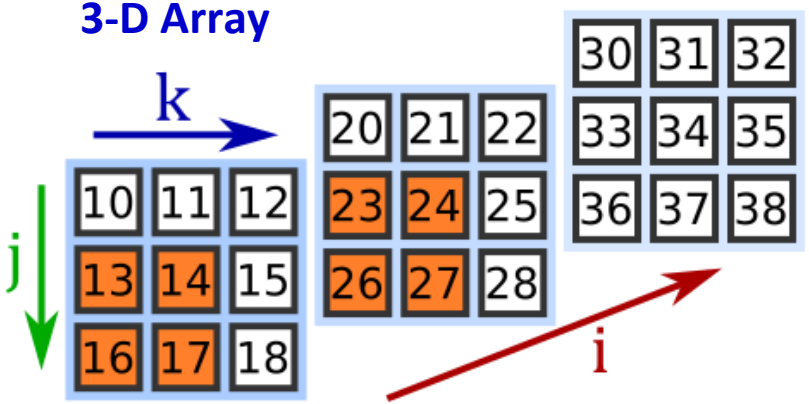
2-D Array



`arr[1:,2:4]`

3-D Array

`arr[:2,1:,:2]`



Numpy Arithmetic Operations

```
import numpy as np
a = np.arange(9, dtype = np.float_).reshape(3,3)
b = np.array([10,10,10])
```

⟹ `[10, 10, 10]`

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

`np.add(a,b)`

⟹
```
[[ 10, 11, 12 ]
 [ 13, 14, 15 ]
 [ 16, 17 18 ]]
```

`np.subtract(a,b)`

⟹
```
[[ 10, 11, 12]
 [ 13, 14, 15]
 [ 16, 17, 18]]
```

`np.multiply(a,b)`

⟹
```
[[-10, -9, -8]
 [-7, -6, -5]
 [-4, -3, -2]]
```

`np.divide(a,b)` ⟹
```
[[ 0, 0.1, 0.2]
 [ 0.3, 0.4, 0.5]
 [ 0.6, 0.7, 0.8]]
```

Numpy Arithmetic Functions

```
import numpy as np
a = np.array([7,3,4,5,1])
b = np.array([3,4,5,6,7])
```

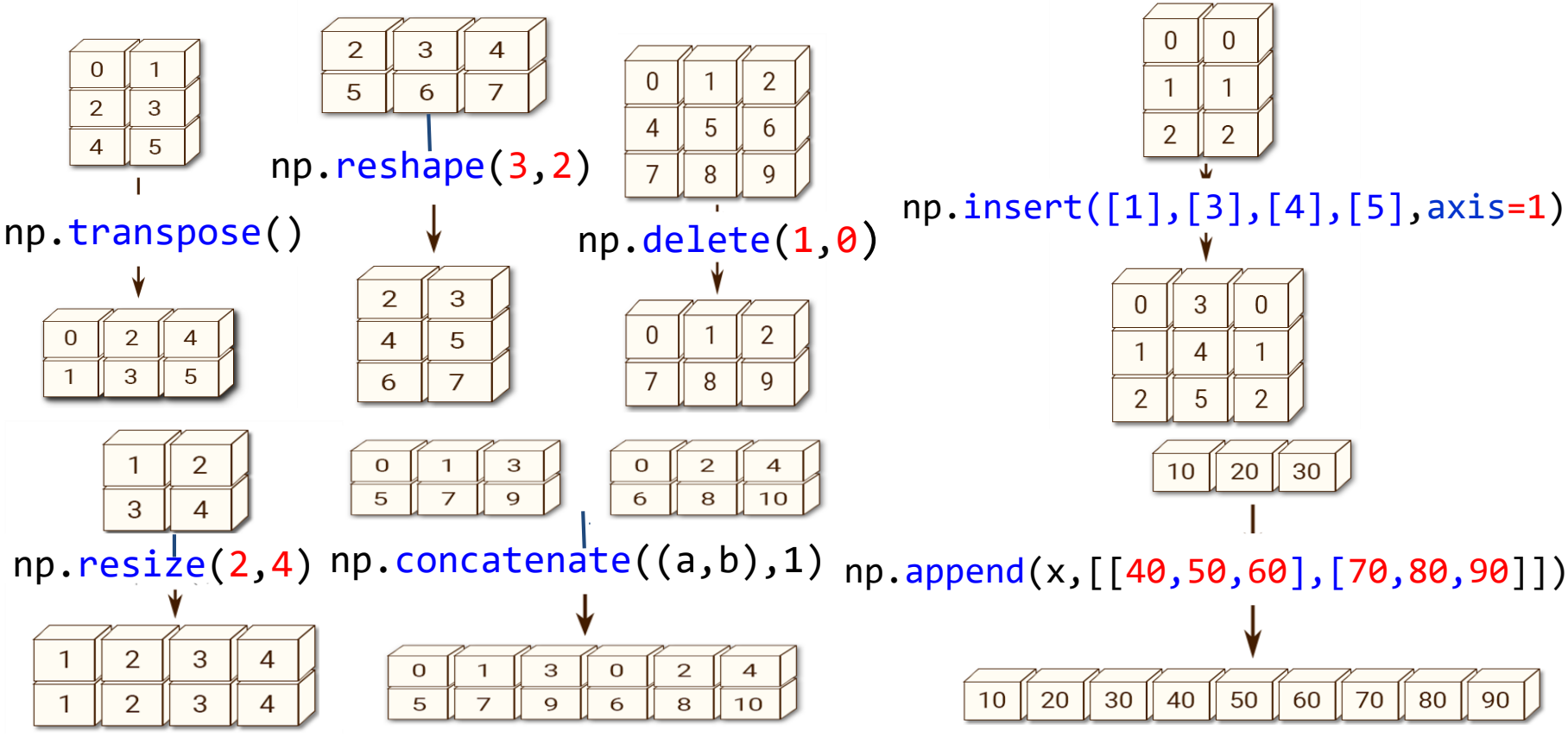
```
np.remainder(a,b)
⇨ [1, 3, 4, 5, 1]
```

```
np.power(a,b)
⇨ [343, 81,1024,15625, 1]
```

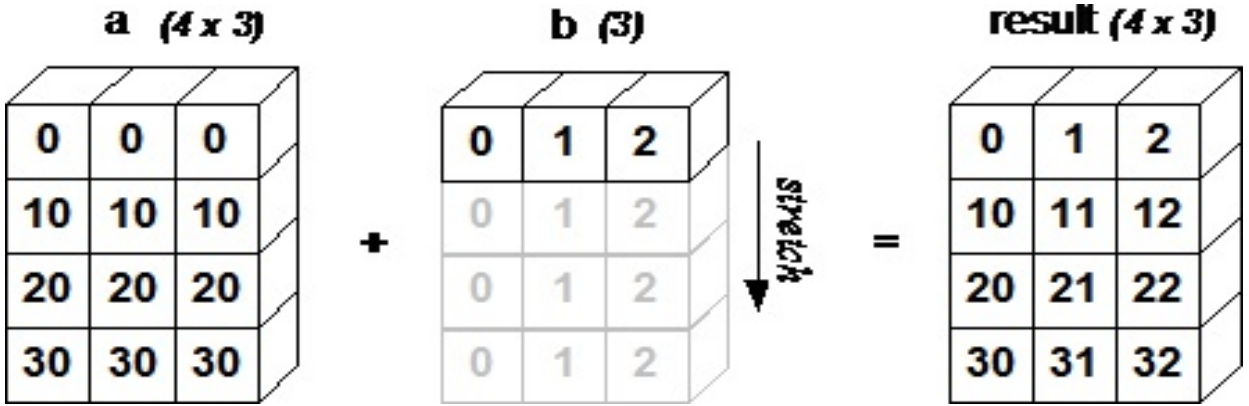
```
np.mod(a,b)
⇨ [1, 3, 4, 5, 1]
```

```
np.reciprocal(a)
⇨ [0, 0, 0, 0, 1]
```

Numpy Array Manipulation Functions

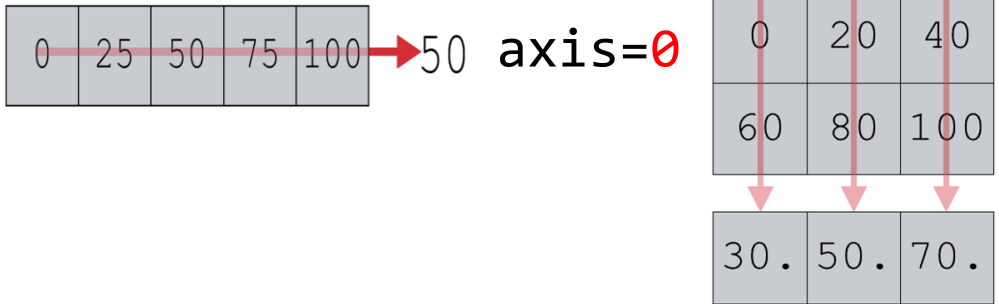


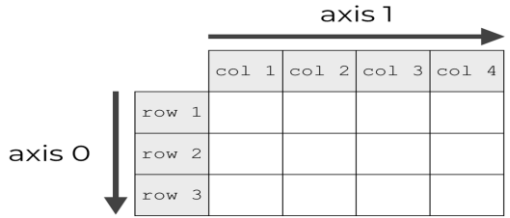
- Broadcasting refers to how numpy treats arrays with different dimension during arithmetic operations which lead to certain constraints, the smaller array is broadcast across the larger array so that they have compatible shapes.

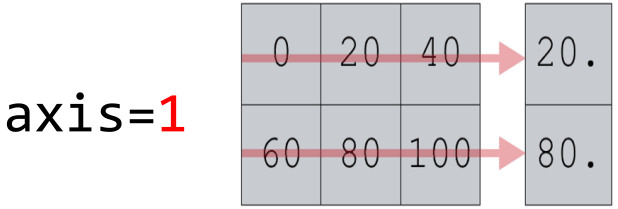


- Broadcasting Rules:
  - If the arrays don’t have the same rank then prepend the shape of the lower rank array with 1s until both shapes have the same length.
  - The two arrays are compatible in a dimension if they have the same size in the dimension or if one of the arrays has size 1 in that dimension.
  - The arrays can be broadcast together iff they are compatible with all dimensions.
  - After broadcasting, each array behaves as if it had shape equal to the element-wise maximum of shapes of the two input arrays.
  - In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

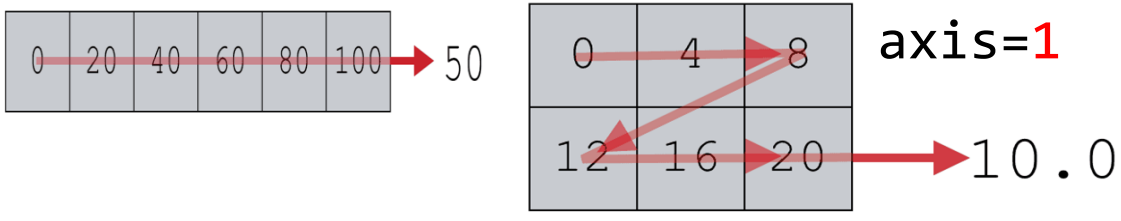
→ np.median(*array,[axis],...*)

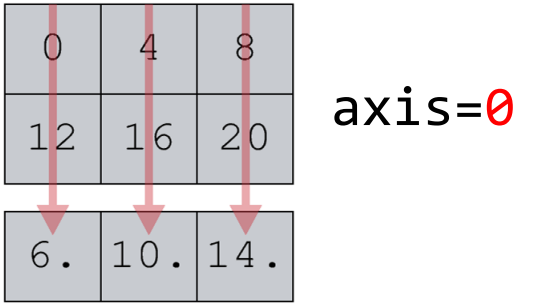






→ np.mean(*array, [axis],...*)

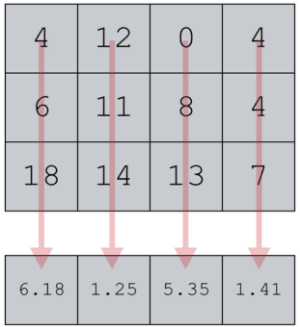




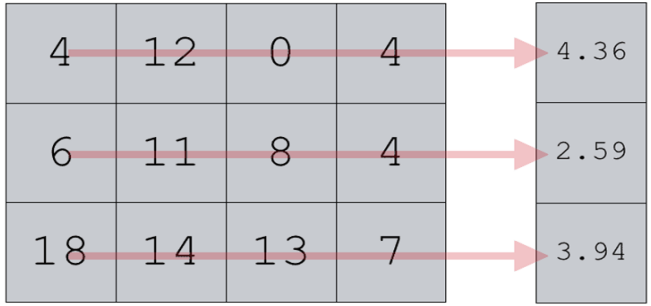
Numpy Statistical Operations

→ `np.std(array,[axis],...)` # standard deviation

axis=0

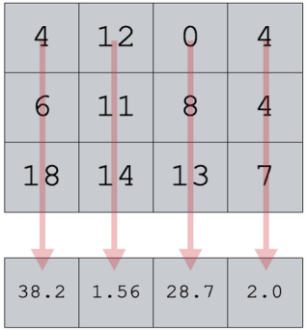


axis=1

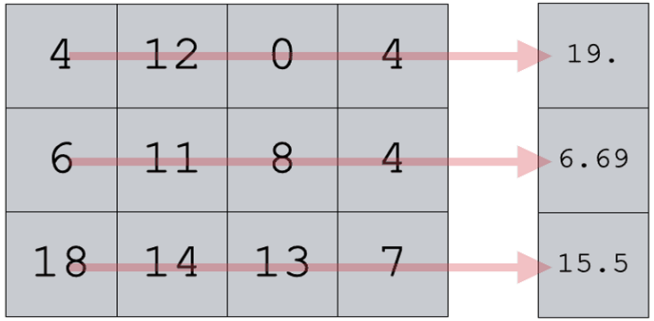


→ `np.var(array, [axis],...)` # variance

axis=0



axis=1



Numpy Linear Algebra

- Linalg : the package in NumPy for Linear Algebra
- dot(): product of two arrays
- vdot(): Complex-conjugating dot product

Example: a = [[1, 0], [0, 1]]

```
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],  [2, 2]])
```

Numpy Linear Algebra

- inner(): product of two arrays
- numpy.inner(a, b, /)
- a, b: array_like
- If a and b are non-scalar, their last dimensions must match

  Returns: out: ndarray
- If a and b are both scalars or both 1-D arrays then a scalar is returned;
- otherwise an array is returned. out.shape = (*a.shape[:-1], *b.shape[:-1])

- outer(): compute the outer product of two vectors
- numpy.outer(a, b, out=None)

Parameters:

- a : (M,) array_like
- First input vector. Input is flattened if not already 1-dimensional.
- b : (N,) array_like
- Second input vector. Input is flattened if not already 1-dimensional.
- out : (M, N) ndarray, optional
- A location where the result is stored

- matmul(): Matrix product of two arrays

numpy.matmul(x1, x2, /, out=None, *, casting='same_kind', order='K', dtype=None, subok=True[, signature, extobj, axes, axis]) = <ufunc 'matmul'>

Parameters: x1, x2: array_like

Input arrays, scalars not allowed.

Out: ndarray, optional

If provide allocation, it must have a shape that matches the signature (n,k),(k,m)->(n,m). If not provided or None, a freshly-allocated array is returned.

- Another linear algebra functions
  - det()
  - inv()
  - trace()
  - rank()

Numpy Matrix Library matlib

- has functions that return matrices instead of ndarray objects.

```
import numpy as np
import numpy.matlib

# with the specified shape and type without initializing entries
mat_e =np.matlib.empty((3, 2), dtype = int)
# filled with 0
mat_zeros = np.matlib.zeros(5, 3)
# filled with 1
mat_ones  = np.matlib.ones(4, 3)
# diagonal elements filled with 1, others with 0
mat_ones  = np.matlib.eye(3,5)
# create square matrix with 0, diagonal filled with 1, others with 0
mat_zeros = np.matlib.identity(5)
# filled with random data
mat_e =np.matlib.empty(3, 2))
```

- What are the I/O functions of NumPy?
- The I/O functions provided by NumPy are:
  - load() and save() functions handle numPy binary files (with npyextension)
  - loadtxt() and savetxt() functions handle normal text files

- numpy.save(): The input array is stored in a disk file with the numpy.save() file and with an npy extension.

Ex:

```
#The input array is stored in a disk file with
the numpy.save() file and with an npy extension.
import numpy as np
a = np.array([1,2,3,4,5])
np.save('outfile',a)
# use load() function to reconstruct
import numpy as np
b = np.load('outfile.npy')
print b
```

```
#The output as:
array([1, 2, 3, 4, 5])
```

- numpy.savetxt() and numpy.loadtxt() functions help in storage and retrieval of the array data in simple text file format.

Ex:

```
import numpy as np
a = np.array([1,2,3,4,5])
np.savetxt('out.txt',a)
b = np.loadtxt('out.txt')
print b
```

The output produced appears as: [1. 2. 3. 4. 5.]