Northeastern University

DS 5110 – Lecture 2
NumPy

Roi Yehoshua

Agenda

- NumPy arrays
- Data types
- Indexing and slicing
- Universal functions
- Broadcasting
- Linear algebra

NumPy

- NumPy (Numerical Python) is the fundamental package for numerical and statistical computing in Python
- Offers a high-performance, richly functional *n*-dimensional array type called **ndarray**
- Supports **vectorization**: a single operation can be carried out on an entire array
- NumPy is written in C to ensure that it runs as fast as possible
- Contains useful linear algebra and statistical analysis methods
- Many Python libraries depend on NumPy

Creating Arrays From Existing Data

- NumPy is typically imported as **np** so that you can access its members with **np.**

```
import numpy as np
```

- NumPy provides various functions for creating arrays
- You can create an array from an existing list by passing it to the function **np.array()**:

```
a = np.array([2, 3, 5, 7, 11])
```

```
print(a)
```

```
[ 2  3  5  7 11]
```

- All the elements in an array must be of the same data type

Creating Two-Dimensional Arrays

- To create a 2D array, you can pass a list of lists:

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

```
print(A)
```

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

Array Attributes

- The array object provides some useful attributes:
  - **dtype** – the type of the elements in the array
  - **ndim** – the array's number of dimensions
  - **shape** – a tuple specifying the size of the array along each dimension

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

```
A.dtype
```

```
dtype('int32')
```

```
A.ndim
```

```
2
```

```
A.shape
```

```
(2, 3)
```

Array Element Type

- The data type of the array is deduced from the type of its elements:

```
a = np.array([1.0, 2.0, 3.0])
a.dtype
```

```
dtype('float64')
```

- You can also explicitly set the data type using the optional **dtype** argument:

```
a = np.array([1, 2, 3], dtype='uint8')
a.dtype
```

```
dtype('uint8')
```

Common NumPy Data Types

Data Type	Description
int8	Integer in a single byte: -128 to 127
int16	Integer in 2 bytes: -32768 to 32767
int32	Integer in 4 bytes: -2147483648 to 2147483647
int64	Integer in 8 bytes: -2^{63} to $2^{63} - 1$
uint8	Unsigned integer in a single byte: 0 to 255
uint16	Unsigned integer in 2 bytes: 0 to 65535
uint32	Unsigned integer in 4 bytes: 0 to 4294967295
uint64	Unsigned integer in 8 bytes: 0 to $2^{64} - 1$
float32	Single-precision, signed float: $\sim 10^{-38}$ to $\sim 10^{38}$ with ~7 decimal digits of precision
float64	Double-precision, signed float: $\sim 10^{-308}$ to $\sim 10^{308}$ with ~15 decimal digits of precision

Filling Arrays with Specific Values

- The functions **zeros**, **ones** and **full** create arrays containing 0s, 1s, or a specified value
  - Their first argument is a tuple of integers specified the desired shape

```
a = np.zeros((3, 3))  # a matrix 3x3 of all zeros
a
```

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

```
b = np.ones(5)  # a 1-D array of all ones
b
```

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

```
c = np.full((2, 2), 7)  # a constant array 2x2 with values = 7
c
```

```
array([[7, 7],
       [7, 7]])
```

Creating Arrays with Random Values

- The module np.random contains functions to generate random arrays
- **np.random.random**(*shape*) creates an array with numbers sampled randomly from the uniform distribution over [0, 1)

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

```
array([[0.53938706, 0.48493127, 0.6644849 ],
       [0.9244017 , 0.12693321, 0.0937883 ],
       [0.11086629, 0.46859553, 0.77811775]])
```

- **np.random.randint**(*low*, [*high*], *size*) creates an array with integer numbers sampled from the interval [*low*, *high*)
  - If *high* is not specified, then the sampled interval is [0, *low*)

```
np.random.randint(1, 10, 5)
```

```
array([3, 1, 7, 2, 3])
```

Creating Arrays with Random Values

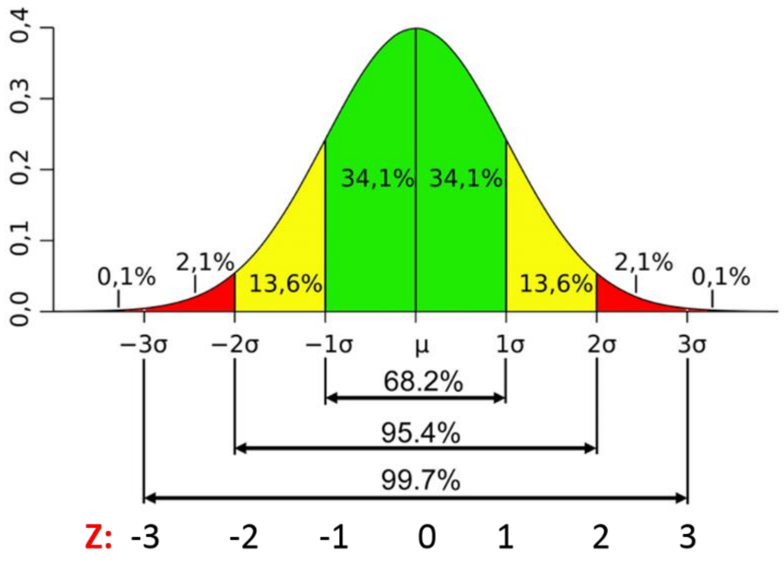
- **np.random.normal**(*loc=0.0, scale=1.0, size*) samples numbers from the normal distribution with mean = *loc* and standard deviation = *scale*

```
np.random.normal(size=(3, 3))
```

```
array([[-0.25329196, -0.0870325 ,  1.77494967],
       [ 1.36233034, -1.04551834, -0.16560166],
       [ 1.04324528, -2.18187512, -1.33373651]])
```

```
np.random.normal(100, 10, size=(3, 3))
```

```
array([[ 71.33579679,  88.46795444, 105.73253249],
       [ 94.35509741, 102.48275919,  86.61720805],
       [ 90.20942587, 103.56148269, 104.07765674]])
```



Creating Arrays from Ranges

- **np.arange**([*start*], *stop*, [*step*]) creates an array from a sequence of numbers
- Its arguments are the same as in Python's range() function
  - *start* – start of the sequence (default = 0)
  - *stop* – end of the sequence (not included in the sequence)
  - *step* – space between values (default = 1)
- In contrast to range() can also generate sequences of floating-point numbers

```
np.arange(5)
```

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

```
np.arange(0.0, 1.0, 0.1)
```

```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

Creating Arrays from Ranges

- **np.linspace**(*start*, *stop*, *num*) creates an array of evenly spaced numbers
  - *start* – start of the sequence
  - *stop* – end of the sequence (included in the sequence)
  - *num* – number of evenly spaced values to generate (default = 50)
- For example to get 10 equals spaced numbers between 0 and π:

```
np.linspace(0, np.pi, 10)
```

```
array([0.        , 0.34906585, 0.6981317 , 1.04719755, 1.3962634 ,
       1.74532925, 2.0943951 , 2.44346095, 2.7925268 , 3.14159265])
```

Reshaping an Array

- You can use the **reshape()** method to change the dimensions of your array
- The new shape should have the same number of elements as the original shape

```
np.arange(1, 13).reshape(3, 4)
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

- One of the dimensions may be specified as -1
  - Its size if inferred from the length of the array and the remaining dimensions

```
np.arange(1, 13).reshape(3, -1)
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Flattening an Array

- **ravel()** can be used to flatten a multidimensional array into one dimension:

```
a = np.arange(1, 10).reshape(3, 3)
a
```

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

```
a.ravel()
```

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

Indexing and Selecting Elements

- In NumPy there are different ways to access or change values in arrays
  - Indexing
  - Slicing
  - Fancy indexing
  - Boolean indexing / masking
  - And combinations thereof

Indexing

- An array is indexed by a tuple of integers, e.g., a[i, j]
  - This is different from Python lists where we used double square brackets a[i][j]

```
a = np.arange(1, 13).reshape(3, 4)
a
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```
a[1, 2]  # row 1, column 2
```

```
7
```

```
a[-1, -1]  # can use negative indexes
```

```
12
```

```
a[1]  # row 1
```

```
array([5, 6, 7, 8])
```

Slicing

- Slicing allows you to select multiple sequential rows/columns
  - To select all the elements along one of the dimensions use colon :

```
a[0:2]   # the first two rows
```

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

```
a[:, 0:2]  # the first two columns
```

```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

```
a[0:2, 0:2]  # the first two rows and columns
```

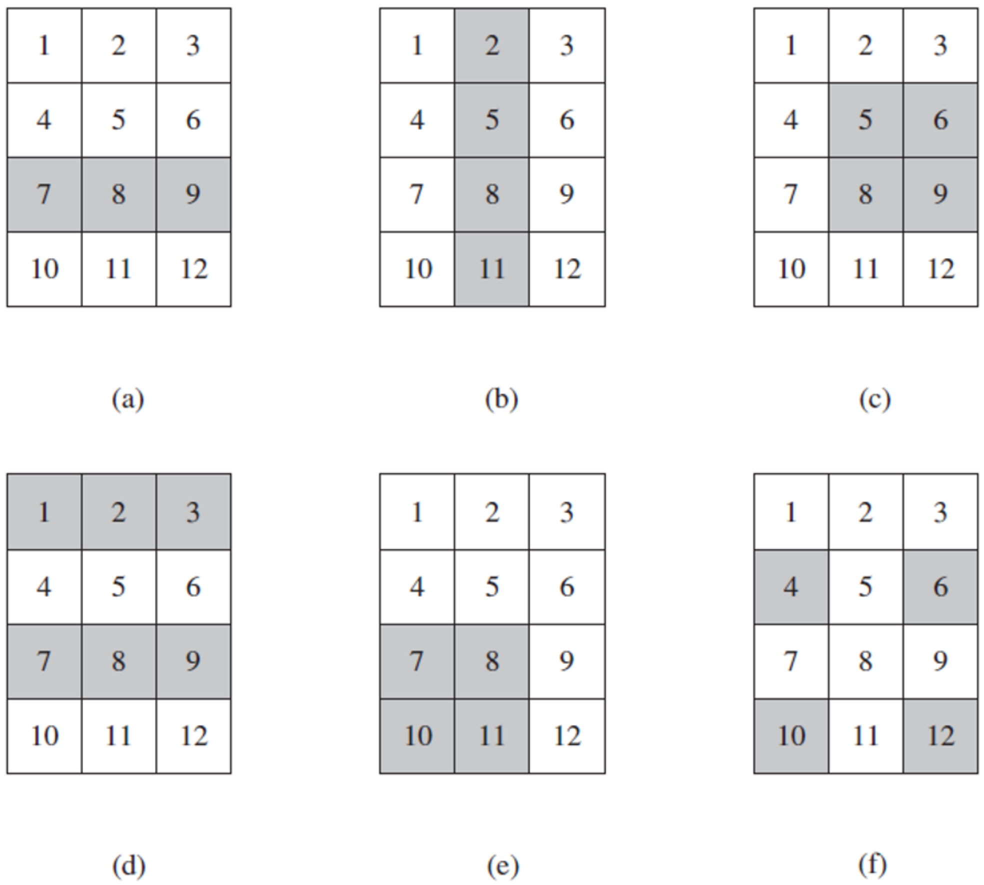
```
array([[1, 2],
       [5, 6]])
```

```
a[1, 1:]  # the second row and second column onwards
```

```
array([6, 7, 8])
```

Class Exercise

- Use slicing to select the highlighted elements from the given matrix:



Fancy Indexing

- You can select multiple non-sequential rows/columns by specifying a list of indexes

```python
a = np.arange(1, 13).reshape(3, 4)
a
```

```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

```python
a[[0, 2]]
```

```
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
```

```python
a[:, [1, 3]]
```

```
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
```

Boolean Indexing (Masking)

- Boolean indexing is used to select elements of the array that satisfy some condition
- For example, to extract only the even numbers from a given array:

```python
a = np.arange(5)
a[a % 2 == 0]
```

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

- The mask creates an array of Boolean values:

```python
a % 2 == 0
```

```
array([ True, False,  True, False,  True])
```

  - The True elements indicate which elements from the array to return
- We can also use Boolean indexing directly:

```python
a[[True, True, False, False, True]]
```

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

Changing Elements

- Changes to a sliced or indexed array are reflected in the original array, e.g.,

```
a = np.arange(10)
a
```

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

```
a[[2, 1, 8, 4]] = 99
a
```

```
array([ 0, 99, 99,  3, 99,  5,  6,  7, 99,  9])
```

```
a[a < 5] = -1
a
```

```
array([-1, 99, 99, -1, 99,  5,  6,  7, 99,  9])
```

Array Operators

- NumPy provides many operators that perform operations on entire arrays
- These operators work **element-wise** (applied to every element in the array)
- For example, you can apply arithmetic operators between arrays and numeric values:

```
a = np.arange(1, 6)
a
```

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

```
a + 10
```

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

```
a * 2
```

```
array([ 2,  4,  6,  8, 10])
```

```
a**3
```

```
array([  1,   8,  27,  64, 125], dtype=int32)
```

Array Operators

- You may also perform arithmetic operations between arrays of the same shape:

```
a = np.arange(1, 6)
a
```

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

```
b = np.linspace(1.1, 5.5, 5)
b
```

```
array([1.1, 2.2, 3.3, 4.4, 5.5])
```

```
a * b
```

```
array([ 1.1,  4.4,  9.9, 17.6, 27.5])
```

Slowness of Loops

Northeastern University

- Vectorized operations execute significantly faster than corresponding list operations

```
def compute_reciprocals(values):
    result = np.empty(len(values))
    for i in range(len(values)):
        result[i] = 1.0 / values[i]
    return result

big_array = np.random.randint(1, 100, size=1000000)
%timeit compute_reciprocals(big_array)
```

```
1.53 s ± 25.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit (1.0 / big_array)
```

```
3.61 ms ± 16.6 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Comparing Arrays

- You can compare arrays with scalars or with other arrays
- Comparisons are performed element-wise and produce an array of Boolean values

```
a = np.arange(1, 6)
a
```

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

```
a < 3
```

```
array([ True,  True, False, False, False])
```

```
2 * a == a**2
```

```
array([False,  True, False, False, False])
```

```
2 * a < a**2
```

```
array([False, False,  True,  True,  True])
```

Comparing Arrays

- Comparing two floating-point arrays with == may give incorrect results
- Instead, use **isclose**(*a*, *b*) to check if the elements are “close” enough to each other

```
x = np.linspace(0, np.pi, 5)
```

```
np.sin(x)**2 == (1 - np.cos(x)**2)
```

```
array([ True, False,  True,  True, False])
```

```
np.isclose(np.sin(x)**2, 1 - np.cos(x)**2)
```

```
array([ True,  True,  True,  True,  True])
```

comparing with == gives incorrect results

Logical Operations

- You can use the operators &, | and ~ to perform *and, or, not* between Boolean arrays

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

```
(a > 1) & (a < 4)
```

```
array([False,  True,  True, False, False])
```

Parentheses are required here because of operator precedence rules

```
(a < 2) | (a > 4)
```

```
array([ True, False, False, False,  True])
```

```
~(a > 3)
```

```
array([ True,  True,  True, False, False])
```

Universal Functions

- NumPy offers many **universal functions** (**ufuncs**) that perform various element-wise operations on arrays
- For example, we can calculate the square root of an array's values using **np.sqrt()**

```
a = np.array([1, 4, 9, 16, 25, 36])
```

```
np.sqrt(a)
```

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

- Some of the ufuncs are called when you use operators like + or *
  - e.g., a + b is equivalent to **np.add**(a, b)

Universal Functions

- Math
  - add, subtract, multiply, divide, remainder, exp, log, sqrt, power, and more
- Trigonometry
  - sin, cos, tan, hycot, arcsin, arccos, arctan, and more
- Bit manipulation
  - bitwise_and, bitwise_or, bitwise_xor, invert, left_shift, right_shift
- Comparison
  - greater, greater_equal, less, less_equal, equal, not_equal, logical_and, logical_or, and more
- Floating point
  - floor, ceil, isinf, isnan, fabs, trunc, and more
- For a full list see https://numpy.org/doc/stable/reference/ufuncs.html

Example: Trigonometric Functions

- NumPy provides some useful trigonometric functions:

```
theta = np.linspace(0, np.pi, 3)
theta
```

```
array([0.        , 1.57079633, 3.14159265])
```

```
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])
```

Example: Exponents and Logarithms

- Another common type of operation are exponentials and logarithms:

```
x = [1, 2, 3]
np.exp(x)
```

```
array([ 2.71828183,  7.3890561 , 20.08553692])
```

```
x = [1, 2, 4, 10]
np.log(x)  # ln(x)
```

```
array([0.        , 0.69314718, 1.38629436, 2.30258509])
```

```
np.log2(x)
```

```
array([0.        , 1.        , 2.        , 3.32192809])
```

```
np.log10(x)
```

```
array([0.        , 0.30103   , 0.60205999, 1.        ])
```

Aggregations

- Aggregation functions allow you to get various statistics about your data
  - e.g., sum(), min(), max(), mean(), var(), std()
- By default these functions ignore the shape and use all elements in the calculations

```
grades = np.array([[87, 96, 70], [92, 87, 80],
                   [84, 67, 90], [100, 81, 92]])
grades
```

```
array([[ 87,  96,  70],
       [ 92,  87,  80],
       [ 84,  67,  90],
       [100,  81,  92]])
```

```
grades.max()
```

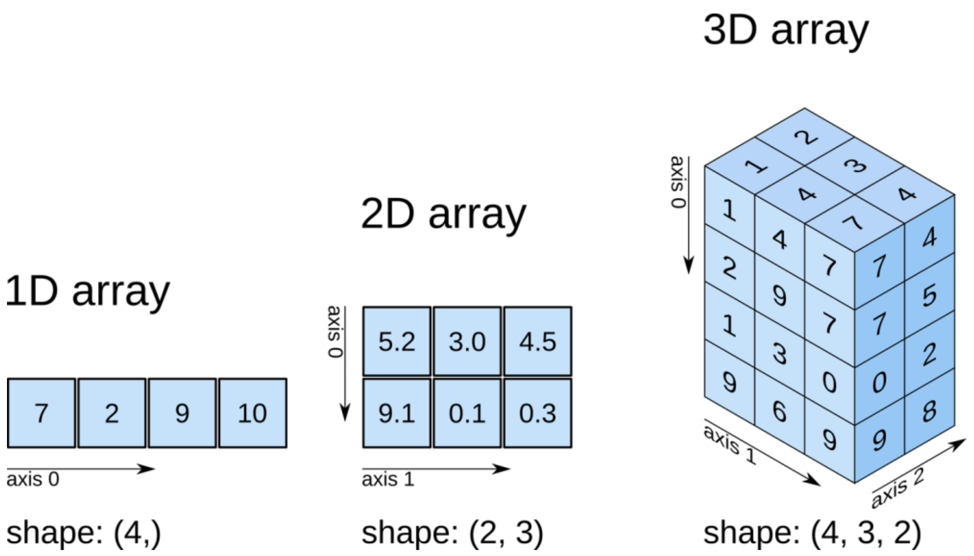
```
100
```

```
grades.mean()
```

```
85.5
```

Calculations by Row or Column

- Many aggregation methods can be performed on specific array dimensions
- These methods receive an **axis** keyword argument that specifies which dimension to use in the calculation



Calculations by Row or Column

- For example, to calculate the average grade on each *exam* we can use axis=0:

```
grades.mean(axis=0)
```

```
array([90.75, 82.75, 83.  ])
```

- To calculate the average grade of each student we can use axis=1:

```
grades.mean(axis=1)
```

```
array([84.33333333, 86.33333333, 80.33333333, 91.        ])
```

Missing Values

- **np.nan** (“not a number”) represents a missing value or the outcome of a calculation that is not well-defined (e.g., 0/0)

```
a = np.array([1, np.nan, 3, 4])
a
```

```
array([ 1., nan,  3.,  4.])
```

- Do not test nans for equality (np.nan == np.nan is False), instead use **np.isnan()**

```
np.isnan(a)
```

```
array([False,  True, False, False])
```

- NaN is a bit like a virus – the result of any arithmetic with nan will also be a nan:

```
a + 1
```

```
array([ 2., nan,  4.,  5.])
```

```
a.sum()
```

```
nan
```

Other Aggregation Functions

- NumPy provides many other aggregation functions
- Most aggregates have a NaN-safe version that ignores missing values

Function	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 variation
np.var()	np.nanvar()	Compute variance
np.min()	np.nanmin()	Find minimum value
np.max()	np.nanxmax()	Find maximum value
np.argmin()	np.nanargmin()	Find index of minimum value
np.argmax()	np.nanargmax()	Find index of maximum value
np.cumsum()	np.nancumsum()	Compute the cumulative sum of elements

Argmin and Argmax

- Return the indices of the minimum/maximum values along an axis

```
a = np.random.randint(0, 10, (3, 3))
a
```

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

```
a.argmin()
```

```
4
```

```
a.argmin(axis=0)
```

```
array([1, 1, 0], dtype=int64)
```

```
a.argmin(axis=1)
```

```
array([2, 1, 0], dtype=int64)
```

Counting Nonzero Entries

- **np.count_nonzero()** can be used to count the True entries in a Boolean array
- The counting can also be done along rows or columns by using the `axis` argument

```
a = np.random.randint(100, size=(3, 4))
a
```

```
array([[22, 72, 90, 92],
       [ 0, 71, 77, 12],
       [98, 25, 68, 13]])
```

```
np.count_nonzero(a % 2 == 0)
```

```
8
```

```
np.count_nonzero(a % 2 == 0, axis=0)
```

```
array([3, 1, 2, 2], dtype=int64)
```

```
np.count_nonzero(a % 2 == 0, axis=1)
```

```
array([4, 2, 2], dtype=int64)
```

Truth Value Testing

- To quickly check whether any or all Boolean values are True, use **np.any()** or **np.all()**:

```
a = np.random.randint(100, size=(3, 4))
a
```

```
array([[87, 99, 81, 78],
       [84, 53, 81, 73],
       [68, 51, 43, 47]])
```

```
# are there any values less than 10?
np.any(a < 10)
```

```
False
```

```
# are all values greater than 20?
np.all(a > 20)
```

```
True
```

```
# are all values in each row greater than 50?
np.all(a > 50, axis=1)
```

```
array([ True,  True, False])
```

Class Exercise

- Write a function that gets a 2D NumPy array and a number, and returns how many rows in the array contain the given number
- For example, given the array

```
[[5 2 7 6]
 [8 3 0 3]
 [1 7 3 7]
 [0 2 1 2]]
```

and the number 7, the function should return 2

Sorting an Array

- **np.sort()** returns a sorted version of the array without modifying it:

```
a = np.array([3, 1, 4, 2, 5])
np.sort(a)
```

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

```
a
```

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

- To sort the array in-place, you can instead use the **sort()** method of arrays:

```
a.sort()
a
```

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

Sorting an Array

- You can sort along specific rows or columns of a 2D array using the `axis` argument

```
a = np.random.randint(10, size=(4, 6))
a
```

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

```
# Sort each column of a
np.sort(a, axis=0)
```

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

```
# Sort each row of a
np.sort(a, axis=1)
```

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

Sorting an Array

- Although Python has built-in sort() and sorted() functions for lists, NumPy's **np.sort()** function turns out to be much more efficient:

```
big_array = np.random.randint(1, 100, size=1000000)
%timeit sorted(big_array)
```

```
308 ms ± 5.53 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit np.sort(big_array)
```

```
30.7 ms ± 906 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

- By default np.sort() uses quicksort, though mergesort and heapsort are also available

Sorting an Array

- **np.argsort()** returns the *indexes* that would sort an array rather than the sorted elements themselves:

```
a = [7, 3, 10, 2, 8]
np.argsort(a)
```

```
array([3, 1, 0, 4, 2], dtype=int64)
```

Searching

- **np.where**(*condition*) returns the indices of all the elements that satisfy the condition:

```
a = np.array([7, 2, 5, 1, 4, 6, 3])
np.where(a > 4)
```

```
(array([0, 2, 5], dtype=int64),)
```

- In case of a 2D array, it returns a tuple of the row and the column indices of the elements that satisfy the condition:

```
a = np.arange(9).reshape(3, 3)
a
```

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

```
np.where(a > 4)
```

```
(array([1, 2, 2, 2], dtype=int64), array([2, 0, 1, 2], dtype=int64))
```

Unique Values

- To find the unique values in an array, you can use **np.unique()**:

```
a = np.array([1, 2, 6, 4, 2, 3, 2, 6])
np.unique(a)
```

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

- If you set the argument *return_counts* to True, the function will also return the number of times each unique item appears in the array:

```
values, counts = np.unique(a, return_counts=True)
counts
```

```
array([1, 3, 1, 1, 2], dtype=int64)
```

Adding Elements to an Array

- **np.append**(*arr*, *values*, *axis=None*) appends values to the end of an array
  - *values* must be of the correct shape (the same shape as *arr*, excluding *axis*)
  - *axis* specifies the axis along which to insert *values*

```
a = np.array([1, 2, 3])
np.append(a, [4, 5])
```

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

```
b = np.arange(9).reshape(3, 3)
b
```

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

```
np.append(b, [[9, 10, 11]], axis=0)
```

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

Joining Arrays

- NumPy provides several functions to merge two arrays
- **np.vstack**(*tuple of arrays*) stacks the arrays vertically (row-wise)

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

np.vstack((a, b))
```

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

- **np.hstack**(*tuple of arrays*) stacks the arrays horizontally (column-wise)

```
np.hstack((a, b))
```

```
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

Broadcasting

- Broadcasting allows operations to be performed on arrays of different shapes
- The smaller array is “broadcast” across the larger array to make them compatible
- For example, we can multiply a 2×3 matrix by a one-dimensional array of 3 elements:

```python
a = np.arange(1, 7).reshape(2, 3)
a
```

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

```python
b = np.array([2, 4, 6])
```

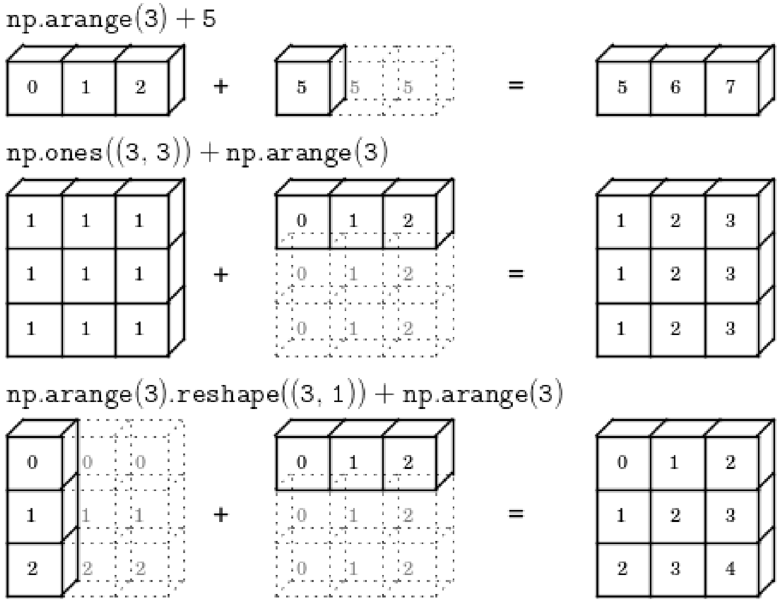
```python
a * b
```

```
array([[ 2,  8, 18],
       [ 8, 20, 36]])
```

- NumPy treats array b as if it were the matrix [[2, 4, 6], [2, 4, 6]]

Broadcasting Rules

- Rule 1: If the two arrays have different numbers of dimensions, the shape of the smaller array is padded with ones on its left side
- Rule 2: The arrays are compatible if all their dimensions are equal or one of them is 1
- Rule 3: If the arrays are compatible and their shapes don't match in one of the dimensions, the array with dimension of 1 is stretched to match the other array



Broadcasting Rules

- When two arrays have shapes that don't support broadcasting, a ValueError occurs:

```
a = np.arange(1, 7).reshape(3, 2)
b = np.arange(3)

a + b
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-24-323d07b6b10f> in <module>
      2 b = np.arange(3)
      3
----> 4 a + b

ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```

- The shapes of the arrays are: a.shape = (3, 2), b.shape = (3,)
- By rule 1, we first pad the shape of *a* with ones: a.shape -> (1, 3)
- a and b are incompatible, since their second dimensions are different and none of them is 1

Class Exercise

- Predict the result of the following operations:

```
x = np.arange(4)
xx = x.reshape(4, 1)
y = np.ones(5)
z = np.ones((3, 4))

print(xx + y)
print(x + z)
print(x + y)
```

Matrices

- NumPy contains special methods to create matrices of specific types
- **np.eye**(*N*) creates an identity matrix of size $N \times N$

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

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

- **np.diag**(*v*) creates a diagonal matrix from the 1-D array *v*:

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

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

Transpose of a Matrix

- The transpose of a matrix results from "flipping" the rows and columns

$$(A^T)_{ij} = A_{ji}$$

- In NumPy, you can use the attribute **.T** to get the transposed matrix

```
A = np.arange(9).reshape(3, 3)
A
```

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

```
A.T
```

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

Dot Product and Matrix Mulitplication

- **np.dot(*a*, *b*)** computes the dot product of arrays *a* and *b*
  - If both *a* and *b* are 1-D arrays, it is the **dot product** of vectors
  - If *a* or *b* is a 2-D array, it is **matrix multiplication**, but using *a* @ *b* is preferred

```
x = np.array([1, 2])
y = np.array([3, 4])

np.dot(x, y)  # 1 * 3 + 2 * 4
```

```
11
```

```
A = np.array([[4, 1], [2, 2]])
B = np.array([[1, 0], [0, 1]])

A @ B   # matrix multilpication
```

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

```
A @ x   # matrix-vector multiplication
```

```
array([6, 6])
```

Linear Algebra

- The **numpy.linalg** module contains additional functions for working with matrices
- For example, **np.linalg.inv**() computes the inverse of a matrix:

```
a = np.array([[1, 2], [3, 4]])
np.linalg.inv(a)
```

```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

  - If the matrix is not invertible, then a LinAlgError exception is raised

- Full list of functions: https://numpy.org/doc/stable/reference/routines.linalg.html

System of Linear Equations

- The set of linear equations:

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}
$$

- Can be expressed as the matrix equation $A\mathbf{x} = \mathbf{b}$:

$$
A\mathbf{x} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \mathbf{b}
$$

- This system has a unique solution if only if A is non-singular
- In this case, we can write the solution as $\mathbf{x} = A^{-1}\mathbf{b}$

System of Linear Equations

- **np.linalg.solve**(*A*, *b*) computes the solution of the linear matrix equation *Ax* = *b*
  - If no unique solution exists (for nonsquare or singular matrix *A*), a LinAlgError is raised
- For example, let's find a solution to the following system of equations:

$$
\begin{aligned}
&3x - 2y = 8 \\
&-2x + y - 3z = -20 \\
&4x + 6y + z = 7
\end{aligned}
$$

```
A = np.array([[3, -2, 0],
              [-2, 1, -3],
              [4, 6, 1]])
b = np.array([8, -20, 7])
np.linalg.solve(A, b)
```

```
array([ 2., -1.,  5.])
```

```
np.linalg.inv(A) @ b
```

```
array([ 2., -1.,  5.])
```

The Determinant

- The **determinant** of a square matrix $A \in \mathbb{R}^{n \times n}$ is a function denoted by $|A|$ or det(A)
- In the case of a 2 × 2 matrix the determinant is computed by:

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

- The general recursive formula for the determinant is:

$$\begin{aligned} |A| &= \sum_{i=1}^{n} (-1)^{i+j} a_{ij} |A_{\backslash i, \backslash j}| \quad (\text{for any } j \in 1, \ldots, n) \\ &= \sum_{j=1}^{n} (-1)^{i+j} a_{ij} |A_{\backslash i, \backslash j}| \quad (\text{for any } i \in 1, \ldots, n) \end{aligned}$$

  - $A_{\backslash i, \backslash j}$ is the matrix that results from deleting the *i*th row and *j*th column from A

The Determinant

- For example, let's compute the determinant of the matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 5 \\ 5 & 6 & 0 \end{pmatrix}$$

$$|A| = 1 \cdot (1 \cdot 0 - 5 \cdot 6) - 2 \cdot (0 \cdot 0 - 5 \cdot 5) + 3 \cdot (0 \cdot 6 - 1 \cdot 5) = -30 + 50 - 15 = 5$$

- In Python, you can use the function **np.linalg.det()** to compute the determinant:

```
A = np.array([[1, 2, 3],
              [0, 1, 5],
              [5, 6, 0]])
```

```
np.linalg.det(A)
```

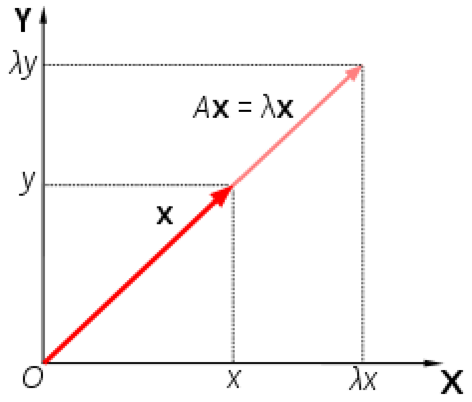
```
4.999999999999995
```

Eigenvalues and Eigenvectors

- Given a square matrix $A \in \mathbb{R}^{n\times n}$, we say that $\lambda \in \mathbb{C}$ is an **eigenvalue** of A and $\mathbf{x} \in \mathbb{C}^n$ is its corresponding **eigenvector** if

$$A\mathbf{x} = \lambda\mathbf{x}, \quad \mathbf{x} \neq 0$$

- Intuitively, this definition means that multiplying A by the vector $\mathbf{x}$ results in a new vector that points in the same direction as $\mathbf{x}$, but scaled by a factor λ



- An eigenvector with its associated eigenvalue is called an **eigenpair**

Eigenvalues and Eigenvectors

- To find the eigenpairs of a matrix *A*, we can rewrite the equation above as follows:

$$(A - \lambda I)\mathbf{x} = 0, \quad \mathbf{x} \neq 0$$

- But $(A - \lambda I)\mathbf{x} = 0$ has non-zero solution to **x** if and only if $A - \lambda I$ is singular, i.e.,

$$|A - \lambda I| = 0$$

- We can use the definition of the determinant to expand this expression into a (very large) polynomial in λ, where λ will have degree *n*
- This polynomial is often called the **characteristic polynomial** of the matrix *A*

Eigenvalues and Eigenvectors: Example

- **np.linalg.eig**(*A*) computes the eigenvalues and eigenvectors of the matrix *A*
  - The eigenvectors are returned as normalized column vectors

```
A = np.array([[2, 1, 0],
              [1, 2, 1],
              [0, 1, 2]])
```

```
eigen_vals, eigen_vecs = np.linalg.eig(A)
eigen_vals
```

```
array([3.41421356, 2.        , 0.58578644])
```

```
eigen_vecs
```

```
array([[-5.00000000e-01,  7.07106781e-01,  5.00000000e-01],
       [-7.07106781e-01,  4.05925293e-16, -7.07106781e-01],
       [-5.00000000e-01, -7.07106781e-01,  5.00000000e-01]])
```