lektion5.nbconvert

November 11, 2026

Einführung in NumpPy Teil 2

1 Broadcasting

2 Beispiel zur Verwendung von Index-Arrays.

3 Verwende boolsche Ausdrücke bei der Definition von Matrizen

4 Achtung

5 Universal functions (ufunc)

1 Einführung in NumPy

1.1 Broadcasting

+, -, * , %, / ** wirken eintragsweise, **wobei die *singleton* Dimension expandiert wird**.

singleton: fehlende Dimenson oder Dimension der Länge 1.

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

[2]:
```
A = np.array([[1, 2], [3, 4]])
v = np.array([-5, 7])
display(A, v) # das etwas übersichtlicher
```

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

```
array([-5,  7])
```

[ ]:
```

```

[3]:
```
# Das erste Element von v wird zu jeder zu jedem Element der ersten Spalte␣
 ↪addiert
A + v
```

[3]:
```
array([[-4,  9],
       [-2, 11]])
```

[4]:
```
#' Das erste Element von v wird ...
A * v
```

[4]:
```
array([[ -5,  14],
       [-15,  28]])
```

[5]:
```
A**v
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 1
----> 1 A**v

ValueError: Integers to negative integer powers are not allowed.
```

[ ]:
```

```

[6]:
```
# zu jedem Eintrag wird 1 addiert
A + 1
```

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

[7]:
```
# jeder Eintrag wird durch 3 dividiert
A / 3
```

[7]:
```
array([[0.33333333, 0.66666667],
       [1.        , 1.33333333]])
```

numpy macht hier einen sogenannten 'upcast', versucht also ganze Zahlen in Gleitkommazahlen (float) umzuwandeln und gibt als Ergebnis ein Array mit floats zurück.

[8]:
```
# 3 wird durch jeden Eintrag dividiert
3 / A
```

[8]:
```
array([[3.  , 1.5 ],
       [1.  , 0.75]])
```

Broadcastingregel (aus https://numpy.org/doc/stable/user/basics.broadcasting.html)

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Arrays do not need to have the same number of dimensions.

[9]:
```
DreiArray = np.array(np.arange(3 * 4 * 5)).reshape(3, 4, 5)
DreiArray.shape
```

[9]:
```
(3, 4, 5)
```

[10]:
```
ZweiArray = np.array(np.arange(3 * 4)).reshape(3, 4)
```

[11]:
```
DreiArray + ZweiArray
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[11], line 1
----> 1 DreiArray + ZweiArray

ValueError: operands could not be broadcast together with shapes (3,4,5) (3,4)
```

[12]:
```
DreiArray_neu = np.array(np.arange(3 * 4 * 5)).reshape(5, 3, 4)
DreiArray_neu.shape
```

[12]:
```
(5, 3, 4)
```

[13]:
```
EinsArray = np.array(np.arange(5)).reshape(5, 1, 1)
```

[14]:
```
#DreiArray_neu + ZweiArray + EinsArray
```

[15]:
```
EinsArray
```

[15]:
```
array([[[0]],

       [[1]],

       [[2]],

       [[3]],

       [[4]]])
```

[16]:
```
np.squeeze(EinsArray)
```

[16]:
```
array([0, 1, 2, 3, 4])
```

1.2 Beispiel zur Verwendung von Index-Arrays.

Erzeuge einen Vektor, der die Gegendiagonale einer Matrix enthält

[17]:
```
A = np.arange(16).reshape(4, 4)
A
```

[17]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[18]:
```
i = np.arange(4)
j = i[::-1] # view von i in umgekehrter Reihenfolge
j, i
```

[18]:
```
(array([3, 2, 1, 0]), array([0, 1, 2, 3]))
```

[19]:
```
b = A[i, j]
b
```

[19]:
```
array([ 3,  6,  9, 12])
```

1.3 Verwende boolsche Ausdrücke bei der Definition von Matrizen

[20]:
```
# Bsp: Setze alle negativen Matrixeinträge auf Null
A
B = A - 2*A.T
B
```

[20]:
```
array([[  0,  -7, -14, -21],
       [  2,  -5, -12, -19],
       [  4,  -3, -10, -17],
       [  6,  -1,  -8, -15]])
```

[21]:
```
B[B < 0] = 0
B
```

[21]:
```
array([[0, 0, 0, 0],
       [2, 0, 0, 0],
       [4, 0, 0, 0],
       [6, 0, 0, 0]])
```

elementweises logisches “oder” und “und”

[22]:
```
B = A - A.T
B
```

[22]:
```
array([[ 0, -3, -6, -9],
       [ 3,  0, -3, -6],
       [ 6,  3,  0, -3],
       [ 9,  6,  3,  0]])
```

[23]:
```
# Welche Elemente von B sind kleiner als -5 ODER größer als 6?
index_array = np.logical_or(B < -5, B > 6)
index_array
```

[23]:
```
array([[False, False,  True,  True],
       [False, False, False,  True],
       [False, False, False, False],
       [ True, False, False, False]])
```

[24]:
```
# Welche Elemente von B sind kleiner als 4 UND größer als 0?
display(B, np.logical_and(B < 4, B > 0))
```

```
array([[ 0, -3, -6, -9],
       [ 3,  0, -3, -6],
       [ 6,  3,  0, -3],
       [ 9,  6,  3,  0]])
```

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

[25]:
```
B[index_array] = 100 # index_array = np.logical_or(B < -5, B > 6)
B
```

[25]:
```
array([[  0,  -3, 100, 100],
       [  3,   0,  -3, 100],
       [  6,   3,   0,  -3],
       [100,   6,   3,   0]])
```

1.4 Achtung

NumPy macht kein Ducktyping.

… duck typing is an application of the duck test—"If it walks like a duck and it quacks like a duck, then it must be a duck" https://en.wikipedia.org/wiki/Duck_typing

[26]:
```
a = np.array([1, -2])
```

[27]:
```
a.dtype
```

[27]:
```
dtype('int64')
```

[28]:
```
a[0] = 1. + 1j
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 1
----> 1 a[0] = 1. + 1j

TypeError: int() argument must be a string, a bytes-like object or a real␣
 ↪number, not 'complex'
```

[29]:
```
ac = np.array([1, -2.25], dtype='complex')
ac
```

[29]:
```
array([ 1.  +0.j, -2.25+0.j])
```

[30]:
```
ac[0] = 10. + 10j
```

[31]:
```
a + ac
```

[31]:
```
array([11.  +10.j, -4.25 +0.j])
```

[32]:
```
np.sqrt(a)
```

```
/tmp/ipykernel_232813/2835837605.py:1: RuntimeWarning: invalid value encountered
in sqrt
  np.sqrt(a)
```

[32]:
```
array([ 1., nan])
```

[33]:
```
np.sqrt(a.astype('complex'))
```

[33]:
```
array([1.+0.j        , 0.+1.41421356j])
```

Die NumPy-Funktionen in emath (power, exp, log, und inverse trigonometrische Funktionen) passen den Ausgabedatentyp an, wenn der Ausgabedatentyp vom Eingabedatentyp abweicht.

[34]:
```
np.emath.sqrt(a)
```

[34]:
```
array([1.+0.j        , 0.+1.41421356j])
```

[35]:
```
?np.emath
```

1.5 Universal functions (ufunc)

Die universellen Funktionen aus NumPy (sin, cos, tan, arcsin, arccos, sinh, cosh, arcsinh, … exp, log, log2, log10, sqrt, …) wirken elementweise, wenn sie auf NumPy Arrays angewandt werden. https://numpy.org/doc/stable/reference/ufuncs.html

[36]:
```
x = np.linspace(0, 5 * np.pi, 10)  # 10 äquidistante Zahlen zwischen 0 und 5 pi
display(x, np.sin(x))
```

```
array([ 0.        ,  1.74532925,  3.4906585 ,  5.23598776,  6.98131701,
        8.72664626, 10.47197551, 12.21730476, 13.96263402, 15.70796327])
```

```
array([ 0.00000000e+00,  9.84807753e-01, -3.42020143e-01, -8.66025404e-01,
        6.42787610e-01,  6.42787610e-01, -8.66025404e-01, -3.42020143e-01,
        9.84807753e-01,  6.12323400e-16])
```

[37]:
```
?np.expm1
```

[38]:
```
x = np.linspace(0, 5* np.pi, 1000)
```

[39]:
```
%%timeit
y = np.sin(x)
```

```
11.4  s ± 24.5 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[40]:
```
%%timeit
y = np.empty_like(x)
for k, xx in enumerate(x):
    y[k] = np.sin(xx)
```

```
839  s ± 29  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```