# Języki i Biblioteki Analizy Danych

## Laboratorium 9.: NumPy

*mgr inż. Zbigniew Kaleta*

### Podstawowe informacje

NumPy jest biblioteką do obliczeń numerycznych w Pythonie (hence the name), a konkretnie obliczeń macierzowych (choć zawiera też moduł związany z rozkładami prawdopodobieństwa).

NumPy jest napisany w C, dzięki czemu jest bardzo wydajny oraz pozwala osiągnąć korzyści z przetwarzania współbieżnego (wielowątkowego)...

... ale nic za darmo: ograniczają nas typy danych dostępne w C, np. ograniczony zakres inta. W C tylko ma 64 bity a w python nieograniczony.

Python posiada modul do programowania współbierznego, ale jest on ułony, bo jest coś takiego jak GIL Global intrepeter log. W jednej chwili tylko jeden wątek jest wykonywany.

NumPy leży u podstaw Pandasa, sklearn'a i wielu innych bibliotek związanych z analizą danych i sztuczną inteligencją.

```
import numpy as np
```

### 101 sposobów utworzenia macierzy

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

```
[1 3 2]
```

```
A = np.full((4, 4), 2)
print(A)
```

```
[[2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]
 [2 2 2 2]]
```

```
np.ones((4, 4))  # <- please note the parenthesis: matrix size needs to be a tuple
```

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

```
np.zeros((4,4))
```

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

```
M = np.eye(4) # tworzący macierz jednostkową
print(M)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

```
np.arange(10)  # arange to array range, nie mylić z arrange
```

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

```
np.empty((4, 4)) # jak potrzebujemy jakąś macierz do zapisu. Nie
obchodzi nas jej zawartość
```

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

### Indeksowanie macierzy

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

```
print(A[1, 2])  # można też A[1][2], tylko po co?
A[2, 2] = 102
print(A)
```

```
6
[[  0   1   2   3]
 [  4   5   6   7]
 [  8   9 102  11]
 [ 12  13  14  15]]
```

```
A[1, :]
```

```
array([4, 5, 6, 7])
```

```
A[1]  # mniej czytelne
```

```
array([4, 5, 6, 7])
```

```
A[:, 1]
```

```
array([ 1,  5,  9, 13])
```

```
A[1:3, 1] = [100, 200]
A
```

```
array([[  0,   1,   2,   3],
       [  4, 100,   6,   7],
       [  8, 200, 102,  11],
       [ 12,  13,  14,  15]])
```

```
A[1, 4]  # analogicznie jak w przypadku list
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In [17], line 1
----> 1 A[1, 4]

IndexError: index 4 is out of bounds for axis 1 with size 4
```

```
A[1, :]
```

```
array([  4, 100,   6,   7])
```

```
A[1:2, :]  # please note the difference in number of brackets in output
```

```
array([[  4, 100,   6,   7]])
```

```
A[..., 1]
```

```
array([  1, 100, 200,  13])
```

### Rozmiar i kształt macierzy

```
print(M.size) # całkowita liczba elementów
print(M.ndim) # liczba wymiarów
print(M.shape) # wielkość w każdym wymiarze
```

```
16
2
(4, 4)
```

```
M.reshape((2, 8)) # zmiana wielkości macierzy
```

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

```
M.reshape((2, -1)) # -1 oznacza, że Python sam się domyśli
```

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

```
M # jakbym chciał zachować macierz po przekształceniu to trzeba zapisać do zmiennej
```

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

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

```
M.reshape((3, 3)) # wszystkie elemeny orginalnej macierzy muszą się
zmieścić w nowej macierzy
```

```
---------------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Cell In [29], line 1
----> 1 M.reshape((3, 3))

ValueError: cannot reshape array of size 16 into shape (3,3)
```

```
M.reshape((4, 2, 2))
```

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

       [[0., 1.],
        [0., 0.]],

       [[0., 0.],
        [1., 0.]],

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

### Typy danych

- int
- int0, int8, int16, int32, int64
- uint0, uint8, uint16, uint32, uint64
- float
- float_, float16, float32, float64, float128
- complex
- complex64, complex128, complex256
- bool
- bool_, bool8
- longlong, longfloat, longdouble, longcomplex

```
print(np.int0 is np.int64)
print(np.float_ is np.float64)
print(np.bool_ is np.bool8)
```

```
True
True
True
```

```
np.int is int
```

```
C:\Users\Radosław\AppData\Local\Temp\ipykernel_19380\1713624351.py:1:
DeprecationWarning: `np.int` is a deprecated alias for the builtin
`int`. To silence this warning, use `int` by itself. Doing this will
not modify any behavior and is safe. When replacing `np.int`, you may
wish to use e.g. `np.int64` or `np.int32` to specify the precision. If
you wish to review your current use, check the release note link for
additional information.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  np.int is int

True
```

```
v[1] = 2**64 # ograniczenie C, dostajemy błąd przepełnienia
print(v)
```

```
---------------------------------------------------------------------
-----
OverflowError                             Traceback (most recent call
last)
Cell In [34], line 1
----> 1 v[1] = 2**64 # ograniczenie C, dostajemy błąd przepełnienia
      2 print(v)

OverflowError: Python int too large to convert to C long
```

```
v.dtype # jaki typ danych przechowuje macierz
```

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

```
np.full((2, 2), 3, dtype=np.complex64)
```

```
array([[3.+0.j, 3.+0.j],
       [3.+0.j, 3.+0.j]], dtype=complex64)
```

```
v = np.array([1, 3, "hello"])
print(v.dtype) # znjaduje wspólny typ danych. Wynik struktura unicode
11 znaków
print(v)
```

```
<U11
['1' '3' 'hello']
```

```
v[1] = 4 # konwersja jest możliwa, więc działa
print(v)
```

```
['1' '4' 'hello']
```

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

```
[1 3 2]
```

```
v[1] = "hello" # konwersja nie jest możliwa, błąd podczas próby
konwersji hello na int
print(v)
```

```
---------------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Cell In [42], line 1
----> 1 v[1] = "hello" # konwersja nie jest możliwa, błąd podczas
próby konwersji hello na int
      2 print(v)

ValueError: invalid literal for int() with base 10: 'hello'
```

### Operacje na macierzach

```
M # dla przypomienia
```

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

```
A # dla przypomienia
```

```
array([[  0,   1,   2,   3],
       [  4, 100,   6,   7],
       [  8, 200, 102,  11],
       [ 12,  13,  14,  15]])
```

```
M + 1
```

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

```
2 * M
```

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

```
M + A
```

```
array([[  1.,   1.,   2.,   3.],
       [  4., 101.,   6.,   7.],
       [  8., 200., 103.,  11.],
       [ 12.,  13.,  14.,  16.]])
```

```
M * A # element po elemencie jak wyżej dodawanie
```

```
array([[  0.,   0.,   0.,   0.],
       [  0., 100.,   0.,   0.],
       [  0.,   0., 102.,   0.],
       [  0.,   0.,   0.,  15.]])
```

```
M.dot(A) # mnożenie macierzowe
```

```
array([[  0.,   1.,   2.,   3.],
       [  4., 100.,   6.,   7.],
       [  8., 200., 102.,  11.],
       [ 12.,  13.,  14.,  15.]])
```

```
M @ A # to samo co wyżej
```

```
array([[  0.,   1.,   2.,   3.],
       [  4., 100.,   6.,   7.],
       [  8., 200., 102.,  11.],
       [ 12.,  13.,  14.,  15.]])
```

```
A.T # transpozycja macierzy
```

```
array([[  0,   4,   8,  12],
       [  1, 100, 200,  13],
       [  2,   6, 102,  14],
       [  3,   7,  11,  15]])
```

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

```
M + B # B dodała sie do każdego wiersza
```

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

```
M + B.T # B.T dodała sie do każdej kolumny. Zachdzi tutaj Broadcasting
czyli takie uzupełnianie wierszy, koulumn do doprawnego dadawania itp.
# Dla zainteresowanych Broadcasting:
https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
```

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

Operacje z przypisaniem też działają.

```
A[0, 1] = 0
```

```
A == M  # nie używać w if'ach, element po elemencie
```

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

```
       [False, False, False, False],
       [False, False, False, False]])

if A == M:
    print("Są równe")
```

```
---------------------------------------------------------------------------
-----
ValueError                                Traceback (most recent call
last)
Cell In [63], line 1
----> 1 if A == M:
      2     print("Są równe")

ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()
```

```
print(np.all(A == M), np.any(A == M)) # all sprawdza czy wszystkie
elementy są równe, any sprawdza czy jakikolwiek element jest równe
```

```
False True
```

```
print((A == M).all(), (A == M).any())
```

```
False True
```

### Wydajność

```
from itertools import product # implementuje iloczyn kartezjański
zbiorów, umożliwa zapisanie w kompaktowy sposób pętli for
import time

SIZE = 10000 # tworzenie macierzy 10000x10000
X = np.arange(SIZE**2).reshape((SIZE, SIZE))
Y = np.arange(SIZE**2).reshape((SIZE, SIZE))
print(X)
```

```
[[       0        1        2 ...     9997     9998     9999]
 [   10000    10001    10002 ...    19997    19998    19999]
 [   20000    20001    20002 ...    29997    29998    29999]
 ...
 [99970000 99970001 99970002 ... 99979997 99979998 99979999]
 [99980000 99980001 99980002 ... 99989997 99989998 99989999]
 [99990000 99990001 99990002 ... 99999997 99999998 99999999]]
```

```
start_time = time.time()
for i, j in product(range(SIZE), range(SIZE)):
    X[i, j] *= 2
end_time = time.time()
print(X)
print("Run time = {}".format(end_time - start_time))
```

```
[[        0         2         4 ...     19994     19996     19998]
 [    20000     20002     20004 ...     39994     39996     39998]
 [    40000     40002     40004 ...     59994     59996     59998]
 ...
 [199940000 199940002 199940004 ... 199959994 199959996 199959998]
 [199960000 199960002 199960004 ... 199979994 199979996 199979998]
 [199980000 199980002 199980004 ... 199999994 199999996 199999998]]
Run time = 65.31753635406494
```

```
start_time = time.time()
Y *= 2
end_time = time.time()
print("Run time = {}".format(end_time - start_time))
np.all(X == Y)
```

```
Run time = 0.14299964904785156

False
```

### Ciekawostki

https://docs.scipy.org/doc/numpy/user/quickstart.html#indexing-with-arrays-of-indices

```
palette = np.array( [ [0,0,0],                # black
                      [255,0,0],              # red
                      [0,255,0],              # green
                      [0,0,255],              # blue
                      [255,255,255] ] )       # white

                    # definiujemy paletę kolorów

image = np.array( [ [ 0, 1, 2, 0 ],           # each value corresponds
to a color in the palette
                    [ 0, 3, 4, 0 ]  ] )

                    # definiujemy obrazek

palette[image]                            # the (2,4,3) color image
# indeksując macierz image otrzymujemy macierz 2x4x3, gdzie każdy
element to wektor koloru z palety, czyli obrazek w kolorze RGB
```

```
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],

       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```

### Algebra liniowa

```
A = np.array([[1.0, 2], [3, 4]])
print(np.linalg.det(A)) # wyznacznik macierzy
```

```
-2.0000000000000004
```

```
B = np.linalg.inv(A) # znalezienie macierzy odwrotnej
print(B)
print(A.dot(B))
print(B.dot(A)) # daje macierz jednostkową
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
[[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
[[1.00000000e+00 0.00000000e+00]
 [1.11022302e-16 1.00000000e+00]]
```

```
np.linalg.eig(A) # wartości własne wektorów
```

```
(array([-0.37228132,  5.37228132]),
 array([[-0.82456484, -0.41597356],
        [ 0.56576746, -0.90937671]]))
```

```
dir(np.linalg)
```

```
['LinAlgError',
 '__all__',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_umath_linalg',
 'cholesky',
 'cond',
 'det',
 'eig',
 'eigh',
 'eigvals',
 'eigvalsh',
 'inv',
 'linalg',
 'lstsq',
 'matrix_power',
 'matrix_rank',
 'multi_dot',
 'norm',
```

```
 'pinv',
 'qr',
 'slogdet',
 'solve',
 'svd',
 'tensorinv',
 'tensorsolve',
 'test']
```

### Losowość

https://numpy.org/doc/stable/reference/random/generator.html#distributions

```
dir(np.random.default_rng()) # wiele rozkladów prawdopodobieństwa
```

```
['__class__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__setstate__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '_bit_generator',
 '_poisson_lam_max',
 'beta',
 'binomial',
 'bit_generator',
 'bytes',
 'chisquare',
 'choice',
 'dirichlet',
 'exponential',
 'f',
```

```
 'gamma',
 'geometric',
 'gumbel',
 'hypergeometric',
 'integers',
 'laplace',
 'logistic',
 'lognormal',
 'logseries',
 'multinomial',
 'multivariate_hypergeometric',
 'multivariate_normal',
 'negative_binomial',
 'noncentral_chisquare',
 'noncentral_f',
 'normal',
 'pareto',
 'permutation',
 'permuted',
 'poisson',
 'power',
 'random',
 'rayleigh',
 'shuffle',
 'standard_cauchy',
 'standard_exponential',
 'standard_gamma',
 'standard_normal',
 'standard_t',
 'triangular',
 'uniform',
 'vonmises',
 'wald',
 'weibull',
 'zipf']
```

```
np.random.default_rng().gamma(1, 1, 100)
```

```
array([3.86768128, 0.39868794, 0.75165338, 0.06059661, 1.57045777,
       1.65934985, 0.43837895, 1.91390379, 2.04112768, 3.23627499,
       2.08558501, 1.38988742, 0.1057689 , 0.60220177, 1.15923385,
       2.33728007, 0.51603047, 1.06776412, 0.82483936, 1.28381185,
       0.28104471, 3.95336328, 0.02168778, 1.91204286, 0.14418166,
       1.26364433, 0.25042407, 0.7869649 , 1.32232202, 0.48380512,
       2.02695324, 0.28861596, 2.92851804, 3.47300939, 1.81217327,
       0.34521705, 0.13181208, 0.07517358, 0.00920027, 0.22304721,
       0.41655159, 0.84157866, 3.41389438, 0.8851338 , 0.26092205,
       1.2929252 , 0.8912304 , 0.58886367, 0.24114972, 0.25278624,
       0.21696978, 0.90446983, 0.03468641, 1.06656939, 0.06775867,
       0.56165026, 0.23033269, 2.06433295, 0.17576101, 0.04520967,
       1.5635382 , 6.82197086, 0.7430024 , 1.07920236, 0.38670188,
```

```
2.11615594, 0.34721814, 1.82827638, 2.5977513 , 0.25742607,
0.49772289, 0.51219523, 4.45633838, 2.56482445, 0.7451263 ,
1.9817258 , 0.163821  , 1.1153784 , 0.04345476, 0.41207319,
2.08279036, 0.29632666, 0.06292778, 0.48413542, 1.39420627,
1.83765344, 0.56501914, 0.40763649, 0.59965726, 0.15061398,
0.37730593, 0.13822203, 1.99152938, 2.02786472, 0.4409499 ,
7.78250359, 1.06738034, 1.79325375, 0.06389426, 1.20049699])
```

Lektura dodatkowa:

- http://cs231n.github.io/python-numpy-tutorial/ -> sekcja Numpy
- https://docs.scipy.org/doc/numpy/user/quickstart.html
- https://pythongeeks.org/numpy-in-python/