Politechnika Wrocławska

Programowanie Procduralne
Wykład #7(W#14) – Fizyka

Janusz Andrzejewski

Wstęp

- Moduł `NumPy`
- Atrybuty tablicy
- Metody modułu (niektóre)

Plan czyli czego się będziemy uczyli?

- W#1 – Podsumowanie
- – Typy mutowalne oraz niemutowalne a przekazywanie argumentów do funkcji.
- – Zasięg zmiennych — zmienne lokalne, nielokalne oraz globalne.
- – Argumenty funkcji: pozycyjne, nazwane, domyślne, rozpakowywanie, słownikowe
- – Funkcje: „first citizen object”.
- – Funkcje: funkcje wyższego rzędu.
- – Funkcje anonimowe.
- – Dopełnienia funkcji.
- – Zastosowania funkcji.
- Laboratoria: od teorii do praktyki czyli rozwiązywanie problemów związanych z omówionymi zagadnieniami na wykładzie.

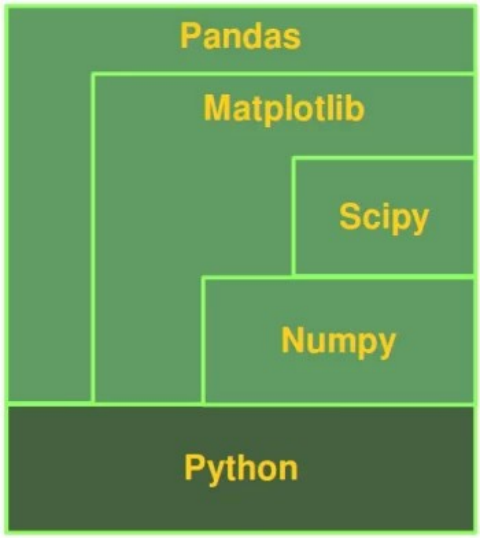
NumPy

- Akronim od ,,Numeric Python'' albo ,,Numerical Python''
- Wzbogaca Pythona w dodatkowe struktury danych – jedno i wielowymiarowe tablice oraz macierze
- obiekty przechowywane w tablicy oraz macierzach wszystkie muszą być tego **samego typu**
- Zalety:
  - Obliczenia oparte na tablicach
  - Wydajne i efektywne implementacje tablic
  - Zaprojektowany do obliczeń naukowych
  - Dostarcza wiele funkcji matematycznych operujących na tablicach

NumPy & przyjaciele

- Notacja obiektowa oraz notacja infixowa („matematyczna”)
- Przyjaciele:
  - `SciPy` (Scientific Python) – oparty na NumPy moduł, rozszerza o funkcje do minimalizacji, aproksymacji, FFT i wiele innych
  - `Matplotlib` – moduł przeznaczony do robienia wykresów 2D oraz 3D
  - `Pandas` – najmłodszy z modułów, przeznaczony do obróbki i analizy danych

NumPy & przyjaciele



https://www.python-course.eu/numpy.php

```
1 import numpy as np
2 print("Ilość atrybutów i metod:",len(dir(np)))
3 w=tuple(x for x in dir(np) if x[0] !='_')
4 print("Ilość 'jawnych' pól: ",len(w))
5 print("Jawne pola:",w)
```

```
Ilość atrybutów i metod: 599
Ilość 'jawnych' pól:  562
Jawne pola: ('ALLOW_THREADS', 'AxisError', 'BUFSIZE', 'CLIP', 'Com
'ERR_DEFAULT', 'ERR_IGNORE', 'ERR_LOG', 'ERR_PRINT', 'ERR_RAISE',
```

Przykłady:

```
1 import numpy as np
2
3 wektor=np.array((1, 2, 3))
4 print(wektor)
5 print(type(wektor))
```

```
[1 2 3]
<class 'numpy.ndarray'>
```

```
1 wkt=np.array((x**2 for x in range(5)))
2 print(wkt)
3 print(type(wkt))
```

```
<generator object <genexpr> at 0x7f925d785d80>
<class 'numpy.ndarray'>
```

```
1 wkt=np.array(tuple(x**2 for x in range(5)))
2 print(wkt)
3 print(type(wkt))
```

```
[ 0  1  4  9 16]
<class 'numpy.ndarray'>
```

System pomocy

```
1 import numpy as np
2 np.array?
```

```
Docstring:
array(object, dtype=None, *, copy=True, order='K', subok=False, ndmin=0,
      like=None)

Create an array.

Parameters
----------
object : array_like
    An array, any object exposing the array interface, an object whose
    __array__ method returns an array, or any (nested) sequence.
    If object is a scalar, a 0-dimensional array containing object is
    returned.
dtype : data-type, optional
    The desired data-type for the array.  If not given, then the type will
    be determined as the minimum type required to hold the objects in the
    sequence.
copy : bool, optional
```

„Standard np”

- Zalecanym sposobem importu modułu jest:

```
import numpy as np
```

- Tablice w NumPy są obiektami klasy ndarray. Podstawowe atrybuty tej klasy to:

Atrybut	Opis
`shape`	Krotka zawierająca ilość elementów w każdym wymiarze (zwanym też osią) tablicy
`size`	Całkowita ilość elementów w tablicy
`ndim`	Ilość wymiarów albo ilość osi w tablicy
`nbytes`	Ilość bytów na zapamiętanie danych
`dtype`	Typ elementów w tablicy

Atrybuty tablic:

```python
import numpy as np

lista = [[1, 2], [3, 4], [5, 6]]
A=np.array(lista)
print("Lista:", lista)
print("tablica A:\n", A)
print("El. tab A 1, 1:", A[1][1])
print("El. tab A 1, 1:", A[1, 1])
print("Atrybuty tablicy:")
print("Typ A ", type(A))
print("Ile wymiarów ", A.ndim)
print("Kształt ", A.shape)
print("Ilość elementów ", A.size)
print("Typ elementy tab ", A.dtype)
print("Ilość potrzebnej pamięci ", A.nbytes)
```

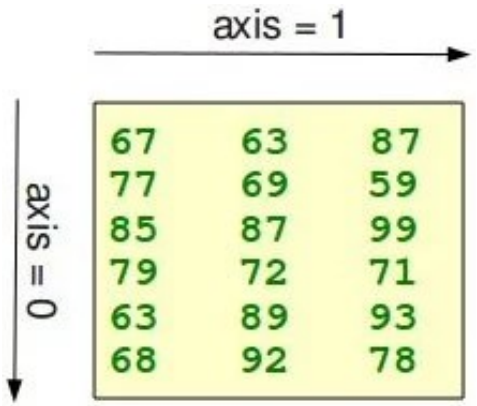
```
Lista: [[1, 2], [3, 4], [5, 6]]
tablica A:
 [[1 2]
 [3 4]
 [5 6]]
El. tab A 1, 1: 4
El. tab A 1, 1: 4
Atrybuty tablicy:
Typ A  <class 'numpy.ndarray'>
Ile wymiarów  2
Kształt  (3, 2)
Ilość elementów  6
Typ elementy tab  int64
Ilość potrzebnej pamięci  48
```

Indeksy w tablicach, podobnie jak w listach zaczynają się od zera !!!

Tablice wielowymiarowe

```
1 x = np.array([ [67, 63, 87],
2                [77, 69, 59],
3                [85, 87, 99],
4                [79, 72, 71],
5                [63, 89, 93],
6                [68, 92, 78]])
7
8 print("Kształt:", np.shape(x))
9 print("x[0, 1]=", x[0, 1])
```

```
Kształt: (6, 3)
x[0, 1]= 63
```




```
1 #Zmiana "kształtu"
2 x.shape=(3, 6)
3 print(x)
```

```
[[67 63 87 77 69 59]
 [85 87 99 79 72 71]
 [63 89 93 68 92 78]]
```

```
1 x.shape=(4, 2)
2 print(x)
```

```
---------------------------------------------------------------
ValueError                                Traceback (most rec
Cell In[17], line 1
----> 1 x.shape=(4, 2)
      2 print(x)

ValueError: cannot reshape array of size 18 into shape (4,2)
```

Tablice wielowymiarowe

```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]],
             order='F')

print(x)
print("Kształt:", np.shape(x))
```

```
[[67 63 87]
 [77 69 59]
 [85 87 99]
 [79 72 71]
 [63 89 93]
 [68 92 78]]
Kształt: (6, 3)
```

```
#Zmiana "kształtu"
x.shape=(3, 6)
print(x)
```

```
---------------------------------------------------------------------------
---------
AttributeError                            Traceback (most recent
all last)
Cell In[27], line 2
      1 #Zmiana "kształtu"
----> 2 x.shape=(3, 6)
      3 print(x)

AttributeError: Incompatible shape for in-place modification. Use
`.reshape()` to make a copy with the desired shape.
```

```
y=x.reshape(3, 6)
print(y)
```

```
[[67 63 87 77 69 59]
 [85 87 99 79 72 71]
 [63 89 93 68 92 78]]
```

```
y[0, 0]=1
print(x[0,0], y[0,0])
```

```
67 1
```

Typy elementów tablicy(wbudowane)

dtype	Warianty	Opis
`int`	`int8, int16, int32, int64`	Liczby całkowite
`uint`	`uint8, uint16, uint32, uint64`	Nieujemne liczby całkowite
`bool`	`bool`	True lub False, typ logiczny
`float`	`float16, float32, float64, float128`	Liczby rzeczywiste
`complex`	`complex64, complex128, complex256`	Liczby zespolone

Typy kompatybilne z wbudowanymi wewnętrznymi typami danych w Pythonie **z wyjątkiem uint**

Typy dostępne przy tworzeniu tablic w NumPy

Cyfry przy typach danych oznaczają ilość bitów potrzebnych do zapamiętania.

`dtype` jest obiektem klasy `numpy.dtype`, i można go inicjować do o wiele bardziej skomplikowanych typów danych:
https://numpy.org/doc/stable/reference/arrays.dtypes.html

Przykłady:

```
1 Bi=np.array([1, -2, 3], dtype=int)
2 print("Tab int: ", Bi)
3 Bf=np.array([1, -2, 3], dtype=float)
4 print("Tab float: ", Bf)
5 Bc=np.array([1, -2, 3], dtype=complex)
6 print("Tab cmpl: ", Bc)
```

```
Tab int:  [ 1 -2  3]
Tab float:  [ 1. -2.  3.]
Tab cmpl:  [ 1.+0.j -2.+0.j  3
```

```
1 print("zajetość Bc: ", Bc.nbytes)
2 Bc128=np.array([1, -2, 3], dtype=np.complex128)
3 print("zajetość Bc128: ", Bc128.nbytes)
4 Bc64=np.array([1, -2, 3], dtype=np.complex64)
5 print("zajetość Bc64: ", Bc64.nbytes)
6 Bc256=np.array([1, -2, 3], dtype=np.complex256)
7 print("zajetość Bc256: ", Bc256.nbytes)
```

```
zajetość Bc:  48
zajetość Bc128:  48
zajetość Bc64:  24
zajetość Bc256:  96
```

Zero-wymiarowa tablica

```
1 #Zero wymiarowa tablica
2 Zero=np.array(2, dtype=float)
3 print("Tab Zero:", Zero)
4 print("Type ", type(Zero))
5 print("Ile wymiarów ", Zero.ndim)
6 print("Kształt ", Zero.shape)
7 print("Ilość elementów ", Zero.size)
8 print("Typ elementy tab ", Zero.dtype)
9 print("Ilość potrzebnej pamięci ", Zero.nbytes)
```

```
Tab Zero: 2.0
Type  <class 'numpy.ndarray'>
Ile wymiarów  0
Kształt  ()
Ilość elementów  1
Typ elementy tab  float64
Ilość potrzebnej pamięci  8
```

Zero-wymiarowa tablica

```
1 Zero=Zero+2.1
2 print("Zero ", Zero)
```

```
Zero  4.1
```

```
1 print("Zero[0]:", Zero[0])
```

```
---------------------------------------------
IndexError                                Tra
Cell In[5], line 1
----> 1 print("Zero[0]:", Zero[0])

IndexError: invalid index to scalar variable.
```

Tworzenie tablic

Nazwa funkcji	Opis
`np.array`	Tworzy tablice na podstawie list, krotek, innej tablicy, itp.
`np.zeros`	Tworzy tablicę wypełnioną zerami wykorzystując podane wymiary oraz typ elementów
`np.ones`	Tworzy tablicę wypełnioną jedynkami wykorzystując podane wymiary oraz typ elementów
`np.diag`	Tworzy tablicę w której na diagonali są zdane elementy oraz pozadiagonalą są same zera
`np.arange([start,] stop[, step], [, dtype=None])`	Tworzy wektor z elementami które zmieniają się od start do end z krokiem step
`np.linspace(start,stop, num=50, endpoint=True)`	Generuje wektor punktów od start do stop, w której jest num punktów
`np.empty`	Tworzy tablicę w której elementy są niezaincjowane

Więcej funkcji: https://numpy.org/doc/stable/reference/routines.array-creation.html

Przykłady

```python
import numpy as np

#Tablica 2 wymiarowa 2 x 3
A=np.zeros((2, 3))
print(A)
print("Typ el tab ", A.dtype)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
Typ el tab  float64
```

```python
#Tablica 2 wymiarowa 3 x 2
A=np.zeros((3, 2), dtype=np.uint16)
print(A)
print("Typ el tab ", A.dtype)
```

```
[[0 0]
 [0 0]
 [0 0]]
Typ el tab  uint16
```

Przykłady

```
1 A=np.empty((2, 3), dtype=np.uint)
2 print("Typ el tab", A.dtype)
3 print(A)
```

```
Typ el tab uint64
[[ 4607182418800017408                    0 13835058055282163712]
 [                   0  4613937818241073152                    0]]
```

```
1 A=np.empty((2, 3), dtype=np.complex64)
2 print("Typ el tab", A.dtype)
3 print(A)
```

```
Typ el tab complex64
[[1.e-45+0.j 3.e-45+0.j 4.e-45+0.j]
 [6.e-45+0.j 7.e-45+0.j 8.e-45+0.j]]
```

Przykład: metoda ones

```
1 A=np.ones(4, dtype=int)
2 print("Typ el tab ", A.dtype)
3 print(A)
```

```
Typ el tab  int64
[1 1 1 1]
```

```
1 A=np.ones((4, 4), dtype=float)
2 print("Typ el tab ", A.dtype)
3 print(A)
```

```
Typ el tab  float64
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

Przykład: metoda diag

```
1 wkt=np.array((1, 2, 3))
2 A=np.diag(wkt)
3 print(A)
4 print("Typ elem:", A.dtype)
```

```
[[1 0 0]
 [0 2 0]
 [0 0 3]]
Typ elem: int64
```

```
1 wkt=np.array((1, 2, 3), dtype=float)
2 A=np.diag(wkt)
3 print(A)
4 print("Typ elem:", A.dtype)
```

```
[[1. 0. 0.]
 [0. 2. 0.]
 [0. 0. 3.]]
Typ elem: float64
```

```
1 wkt=np.array((1, 2, 3), dtype=float)
2 A=np.diag(wkt, k=1)
3 print(A)
```

```
[[0. 1. 0. 0.]
 [0. 0. 2. 0.]
 [0. 0. 0. 3.]
 [0. 0. 0. 0.]]
```

```
1 wkt=np.array((1, 2, 3), dtype=float)
2 A=np.diag(wkt, k=-1)
3 print(A)
```

```
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 2. 0. 0.]
 [0. 0. 3. 0.]]
```

Przykład: metoda diag

```
1 A=np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
2 print("Macierz A:\n", A)
3 D=np.diag(A)
4 print(D)
5 print("Kształt: ", D.shape)
```

```
Macierz A:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
[1 5 9]
Kształt:  (3,)
```

Przykład: metoda arange

```
1 B1=np.arange(5)
2 print("Wektor B1:", B1)
```

```
Wektor B1: [0 1 2 3 4]
```

```
1 B2=np.arange(5, dtype=float)
2 print("Wektor B2:", B2)
```

```
Wektor B2: [0. 1. 2. 3. 4.]
```

```
1 B3=np.arange(1, 5, dtype=float)
2 print("Wektor B3:", B3)
```

```
Wektor B3: [1. 2. 3. 4.]
```

```
1 B4=np.arange(1, 5, 0.4, dtype=float)
2 print("Wektor B4:", B4)
```

```
Wektor B4: [1.  1.4 1.8 2.2 2.6 3.  3.4 3.8 4.2 4.6]
```

Przykład: metoda linspace

```
1 C=np.linspace(1, 10)
2 print(C)
```

```
[ 1.         1.18367347  1.36734694  1.55102041  1.73469388  1.91836735
  2.10204082  2.28571429  2.46938776  2.65306122  2.83673469  3.02040816
  3.20408163  3.3877551   3.57142857  3.75510204  3.93877551  4.12244898
  4.30612245  4.48979592  4.67346939  4.85714286  5.04081633  5.2244898
  5.40816327  5.59183673  5.7755102   5.95918367  6.14285714  6.32653061
  6.51020408  6.69387755  6.87755102  7.06122449  7.24489796  7.42857143
  7.6122449   7.79591837  7.97959184  8.16326531  8.34693878  8.53061224
  8.71428571  8.89795918  9.08163265  9.26530612  9.44897959  9.63265306
  9.81632653 10.        ]
```

```
1 C1=np.linspace(1, 10, 11)
2 print(C1)
```

```
[ 1.   1.9  2.8  3.7  4.6  5.5  6.4  7.3  8.2  9.1 10. ]
```

```
1 C2=np.linspace(1, 10, 11, dtype=int)
2 print(C2)
```

```
[ 1  1  2  3  4  5  6  7  8  9 10]
```

```
1 C3=np.linspace(1, 10, 11, False)
2 print(C3)
```

```
[1.         1.81818182 2.63636364 3.45454545 4.27272727 5.09090909
 5.90909091 6.72727273 7.54545455 8.36363636 9.18181818]
```

Przykład: metoda reshape

```
1 B=np.arange(6).reshape(2, 3)
2 print("mac 2x3", B)
```

```
mac 2x3 [[0 1 2]
 [3 4 5]]
```

```
1 B=np.arange(6, dtype=float).reshape(2, 3)
2 print("mac 2x3", B)
```

```
mac 2x3 [[0. 1. 2.]
 [3. 4. 5.]]
```

```
1 B=np.arange(2*3*4, dtype=float).reshape(2, 3, 4)
2 print("mac 2x3x4", B)
```

```
mac 2x3x4 [[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]
  [ 8.  9. 10. 11.]]

 [[12. 13. 14. 15.]
  [16. 17. 18. 19.]
  [20. 21. 22. 23.]]]
```

Przykład: metoda ones_like

```
1 B=np.arange(6, dtype=int).reshape(2, 3)
2 C=np.ones_like(B)
3 print("C:", C)
4 B=np.arange(2*3*4, dtype=float).reshape(2, 3, 4)
5 D=np.ones_like(B)
6 print("D:", D)
```

```
C: [[1 1 1]
 [1 1 1]]
D: [[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

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

Przykład: metoda zeros_like

```
1 B=np.arange(6, dtype=int).reshape(2, 3)
2 C=np.zeros_like(B)
3 print("C:", C)
4 B=np.arange(2*3*4, dtype=float).reshape(2, 3, 4)
5 D=np.zeros_like(B)
6 print("D:", D)
```

```
C: [[0 0 0]
 [0 0 0]]
D: [[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

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

Przykład: metoda fromfunction

```
1  import numpy as np
2
3  fun1int = lambda x: x+1
4  fun1real = lambda x: x+1.0
5  fun1zesp = lambda x: x+1.0+1.0j
6
7  WekI=np.fromfunction(fun1int, (4,))
8  print(WekI)
9  print("WekI:",WekI.dtype)
10 WekInt=np.fromfunction(fun1int, (4,), dtype=int)
11 print(WekInt)
12 print("WekInt", WekInt.dtype)
13 WekR=np.fromfunction(fun1real, (4,))
14 print(WekR)
15 print("WekR:", WekR.dtype)
16 WekZ=np.fromfunction(fun1zesp, (4,))
17 print(WekZ)
18 print("WekZ:", WekZ.dtype)
```

```
[1. 2. 3. 4.]
WekI: float64
[1 2 3 4]
WekInt int64
[1. 2. 3. 4.]
WekR: float64
[1.+1.j 2.+1.j 3.+1.j 4.+1.j]
WekZ: complex128
```

Przykład: metoda fromfunction

```
1 fun2D = lambda x, y: x+4*y
2 fun3D = lambda i, j, k : i*j*k
3
4 MacA=np.fromfunction(fun2D, (3, 4))
5 print(MacA)
6 print("MacA:", MacA.dtype)
7 Mac3=np.fromfunction(fun3D, (2, 3, 4), dtype=int)
8 print(Mac3)
9 print("Mac3:", Mac3.dtype)
```

```
[[ 0.  4.  8. 12.]
 [ 1.  5.  9. 13.]
 [ 2.  6. 10. 14.]]
MacA: float64
[[[0 0 0 0]
  [0 0 0 0]
  [0 0 0 0]]

 [[0 0 0 0]
  [0 1 2 3]
  [0 2 4 6]]]
Mac3: int64
```

Wycinki - tablica 1 wymiarowa

Wyrażenie	Opis
`a[m]`	Element m-ty tablicy 1-wymiarowej (wektora) a. Indeksy zaczynają się od zera.
`a[-m]`	Wybiera m-ty element od końca
`a[m:n]`	Wybiera elementy od m(włącznie) do n (wyłącznie)
`a[:] lub a[0:-1]`	Wszystkie elementy wektora
`a[:n]`	Wybiera elementy od zera do n(wyłącznie)
`a[n:] lub a[n:-1]`	Wybiera elementy od n-tego(włącznie) do końca
`a[m:n:p]`	Wybiera elementy od m(włącznie) do n(wyłącznie) z krokiem p
`a[::-1]`	Wybiera elementy w odwrotnej kolejności

Uwaga:
Takie same sposoby „wycinania” można wykorzystywać dla każdego z wymiarów z osobna.
Takie same zasady wycinania są także dla list oraz krotek !!!

Przykłady: wycinki

```
1 fun2D = lambda x, y: x+4*y
2
3 MacA=np.fromfunction(fun2D, (3, 4))
4 print(MacA)
5 print("MacA:", MacA.dtype)
```

```
[[ 0.  4.  8. 12.]
 [ 1.  5.  9. 13.]
 [ 2.  6. 10. 14.]]
MacA: float64
```

```
1 print("Kol 0 MacA ", MacA[:, 0])
2 print("Wiersz 1 MacA ", MacA[1, :])
```

```
Kol 0 MacA  [0. 1.]
Wiersz 1 MacA  [ 1.  5.  9. 13.]
```

```
1 print("Górny lewy rog MacA \n", MacA[:2, :2])
```

```
Górny lewy rog MacA
 [[0. 4.]
 [1. 5.]]
```

```
1 print("Co drugi el MacA \n", MacA[::2, ::2])
```

```
Co drugi el MacA
 [[ 0.  8.]
 [ 2. 10.]]
```

Wygląd tablicy

Wygląd (views) tablicy – jest to część tablicy otrzymana poprzez operację wycięcia. Wygląd odnosi się do oryginalnej tablicy, w związku z czym zmiana jakiegoś elementy w wyglądzie zmienia odpowiedni element w tablicy oryginalnej i odwrotnie.

```
1 f2 = lambda x, y : x+10*y
2 A=np.fromfunction(f2, (6, 6))
3 print("Mac A:\n", A)
```

```
Mac A:
 [[ 0. 10. 20. 30. 40. 50.]
 [ 1. 11. 21. 31. 41. 51.]
 [ 2. 12. 22. 32. 42. 52.]
 [ 3. 13. 23. 33. 43. 53.]
 [ 4. 14. 24. 34. 44. 54.]
 [ 5. 15. 25. 35. 45. 55.]]
```

```
1 ##wyglad - tab B
2 B=A[1:5, 1:5]
3 print("Mac B: \n", B)
```

```
Mac B:
 [[11. 21. 31. 41.]
 [12. 22. 32. 42.]
 [13. 23. 33. 43.]
 [14. 24. 34. 44.]]
```

Wygląd tablicy

```
Mac A:
 [[ 0. 10. 20. 30. 40. 50.]
 [ 1. 11. 21. 31. 41. 51.]
 [ 2. 12. 22. 32. 42. 52.]
 [ 3. 13. 23. 33. 43. 53.]
 [ 4. 14. 24. 34. 44. 54.]
 [ 5. 15. 25. 35. 45. 55.]]
```

```
1 B[:,:]=0
2 print("Ponownie mac B\n", B)
```

```
Ponownie mac B
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
1 print("Mac A \n", A)
```

```
Mac A
 [[ 0. 10. 20. 30. 40. 50.]
 [ 1.  0.  0.  0.  0. 51.]
 [ 2.  0.  0.  0.  0. 52.]
 [ 3.  0.  0.  0.  0. 53.]
 [ 4.  0.  0.  0.  0. 54.]
 [ 5. 15. 25. 35. 45. 55.]]
```

Operacje na tablicach

Operacje na tablicach w NumPy wykonywane są element tablicy po elemencie tablicy (elementwise operations).

```
1 w=np.array((1, 2, 3, 4))
2 w+=2
3 print(w)
```

```
[3 4 5 6]
```

```
1 w=np.array((1, 2, 3, 4, 5, 6))
2 w[2:3]+=10
3 print(w)
```

```
[ 1  2 13  4  5  6]
```

```
1 m=np.array((1, 2, 3, 4, 5, 6)).reshape(2, 3)
2 m+=10
3 print(m)
```

```
[[11 12 13]
 [14 15 16]]
```

W ten sam sposób zachowują się operacje arytmetyczne: +; +=; -; -=; *; *=; /; /=; //(dzielenie całkowitoliczbowe); //=; **(potęgowanie); **=;

Funkcje element po elemencie

Funkcje:	Opis:
np.cos, np.sin, np.tan	Funkcje trygonometryczne
np.arccos, np.arcsin. np.arctan	Odwrotne funkcje trygonometryczne
np.cosh, np.sinh, np.tanh	Funkcje hiperboliczne
np.arccosh, np.arcsinh, np.arctanh	Odwrotne funkcje hiperboliczne
np.sqrt	Pierwiastek kwadratowy
np.exp	Funkcja wykładnicza
np.abs	Wartość bezwzględna
np.log, np.log2, np.log10	Logarytm o podstawie odpowiednio:e, 2, 10

Przykład:

```
1  import numpy as np
2
3  def wekX(pocz=-1, kon=1, ile=11):
4      return np.linspace(pocz, kon, ile)
5
6  def wekSIN(x):
7      y=np.sin(np.pi*x)
8      ## Zaokrąglam do 4 cyfr
9      return np.round(y, decimals=4)
10
11 x=wekX()
12 wS=wekSIN(x)
13 print("Sinusy:", wS)
14 w=wS*np.sqrt(2)
15 print("Bojowy sinus",w)
```

```
Sinusy: [-0.     -0.5878 -0.9511 -0.9511 -0.5878  0.      0.5878  0.9511  0.9511
  0.5878  0.    ]
Bojowy sinus [-0.         -0.83127473 -1.34505852 -1.34505852 -0.83127473  0.
  0.83127473  1.34505852  1.34505852  0.83127473  0.        ]
```

Zabawne indeksowanie

```
1 x=np.arange(1, 7)
2 print("Wekt x:",x)
3 ## Lista indeksow
4 li=[3, 1, 5]
5 y=x[li]
6 print("Wekt y:",y)
```

```
Wekt x: [1 2 3 4 5 6]
Wekt y: [4 2 6]
```

```
1 y[0]=-1
2 print("Wekt x:", x)
3 print("Wekt y:", y)
```

```
Wekt x: [1 2 3 4 5 6]
Wekt y: [-1  2  6]
```

```
1 ## indeksowanie poprzez wekt
2 wekt=np.array((1, 2, 3))
3 z=x[wekt]
4 print("Wekt z:", z)
```

```
Wekt z: [2 3 4]
```

Zabawne indeksowanie

```
1 x=np.linspace(0, 10, 11)
2 print("Wekt x: ",x)
3 print("Gdzie x>5:", x>5)
4 y=x>5
5 print("Wekt o elem >5:", x[y])
6 print("Albo to samo, krócej", x[x>5])
```

```
Wekt x:  [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
Gdzie x>5: [False False False False False False  True  True  True  True  True]
Wekt o elem >5: [ 6.  7.  8.  9. 10.]
Albo to samo, krócej [ 6.  7.  8.  9. 10.]
```

```
1 z=x[x>5]
2 z[0]=-10
3 print("Wekt z:",z)
4 print("Wekt x:", x)
```

```
Wekt z: [-10.   7.   8.   9.  10.]
Wekt x: [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

Dziękuję za uwagę