

```
import numpy as np                # Alias nazwy numpy
tab1Da = np.array( [1, 2, 3, 4, 5] ) # Jednowymiarowa, typ domyślny
print("tab1Da:", tab1Da)
print("Wycinek tab1Da[-3, -1]:", tab1Da[-3:-1])
print("tab1D[1]=", tab1Da[1])      # Jednowymiarowa, typ domyślny, seria liczb od 1 do 9
tab1Db = np.arange(1,10)
print("tab1Db:", tab1Db)
# Jednowymiarowa, typ 'napis Unicode', tj. seria napisów od "1" do "9":
tab1Dc = np.array( np.arange(1,10), dtype='U')
print("tab1Dc:", tab1Dc)
print("Typ danych w tablicy tab1Dc to:", type(tab1Dc[5]) )
```

```
tab2Da = np.array( [ [1, 2, 3], [4, 5, 6] ] ) # Dwuwymiarowa
tab2Db = np.array( [ [7, 8, 9], [10, 11, 12] ] ) # jw.
print("Tablica 2D:\n", tab2Da)
print("tab2Da[1, 2]=", tab2Da[1, 2]) # Trzeci element z drugiego wiersza (liczymy
                                     # od zera!)
print("tab2Da[1][2]=", tab2Da[1][2]) # Alternatywna składnia
```

```
tab3Da = np.array( [ [[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]] ] )
tab3Db = np.array( [tab2Da, tab2Db] ) # Taka sama zawartość jak dla tab3Da!
print("Tablica 3D:\n", tab3Da)
print("Tablica 3D:\n", tab3Db)
```

Jeśli uruchomisz program, to ujrzysz następujące wyniki (nieco je przeformatowałem w książce w celu zmniejszenia rozmiaru wydruku):

```
tab1Da: [1 2 3 4 5]
Wycinek tab1Da[-3, -1]: [3 4]
tab1D[1]= 2
tab1Db: [1 2 3 4 5 6 7 8 9]
tab1Dc: ['1' '2' '3' '4' '5' '6' '7' '8' '9']
Typ danych w tablicy tab1Dc to: <class 'numpy.str_'>
Tablica 2D:
[[ 1 2 3]
 [ 4 5 6]]
tab2Da[1, 2]=6
tab2Da[1][2]=6
Tablica 3D:
[[ [1 2 3] [4 5 6]]
 [ [7 8 9] [10 11 12]]]
```

Hm, na razie wszystko wygląda dość podobnie jak w przypadku list zwykłych i list złożonych Pythona, no może z wyjątkiem metody `arange()`, która pozwala wypełnić nowo utworzoną tablicę pewną sekwencją liczbową (do tematu wypełniania tablic danymi zaraz wrócimy).

Typy danych oferowane przez NumPy są zbudowane na podstawie typów podstawowych znanych z języka C i możesz nawet doprecyzować wybór pożądanego typu bazowego oferującego pożądaną precyzję, stosując konstrukcje podobne do:

```
t = np.array( np.arange(1,10), dtype=np.float16).
```

na wykładnik, 10 na mantysę³. Im więcej bitów przeznaczamy na kodowanie liczby, tym więcej wartości możemy zakodować lub odczytać.

Inne popularne typy prezentowane w powyższej formie to np.:

- `int16` — liczba całkowita ze znakiem (od -32 768 do 32 767),
- `int32` — liczba całkowita ze znakiem (od -2 147 483 648 do 2 147 483 647),
- `int64` — liczba całkowita ze znakiem (od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807),
- `uint8` — liczba całkowita bez znaku (od 0 do 255),
- `uint16` — liczba całkowita bez znaku (od 0 do 65 535),
- `uint32` — liczba całkowita bez znaku (od 0 do 4 294 967 295),
- `uint64` — liczba całkowita bez znaku (od 0 do 18 446 744 073 709 551 615),
- `float16` — liczba zmiennoprzecinkowa małej precyzji: bit znaku, 5 bitów na wykładnik, 10 bitów na mantysę,
- `float32` — liczba zmiennoprzecinkowa o zwykłej precyzji: bit znaku, 8 bitów na wykładnik, 23 bity na mantysę,
- `float64` — liczba zmiennoprzecinkowa o podwójnej precyzji: bit znaku, 11 bitów na wykładnik, 52 bity na mantysę,
- `complex64` — liczba zespolona złożona z dwóch 32-bitowych składowych zmiennoprzecinkowych,
- `complex128` — liczba zespolona złożona z dwóch 64-bitowych składowych zmiennoprzecinkowych.

Funkcje tablicowe NumPy

Istnieje wiele metod wspomagających operacje na obiektach NumPy. Część z nich możesz poznać, analizując zestawienie z tabeli 8.9.

Co ciekawe, możliwe jest też łączenie liczb (lub uogólniając, wyrażeń arytmetycznych) z tabelami NumPy przy użyciu klasycznych operatorów typu + (dodawanie), - (odejmowanie) itp. Przykładowo, pomnożenie tablicy NumPy przez liczbę `x` zwróci tablicę, w której każda wartość zostanie przez nią pomnożona!

Inne przykłady:

```
import numpy as np
t = np.array( [5, 10, 15, 20] )
print("t + 5 =", t + 5)      # [5 10 15 20]
print("t - 5 =", t - 5)      # [10 15 20 25]
print("t * 2 =", t * 2)      # [10 20 30 40]
print("t / 2 =", t / 2)      # [2.5 5. 7.5 10]
print("t // 2 =", t // 2)     # [2 5 7 10]
```

³ Wy tłumaczenie tych pojęć znajdziesz w artykule: <https://pl.wikipedia.org/wiki/Liczba>.

TABELA 8.9. Podstawowe funkcje i operatory operujące na tablicach NumPy

METODA/FUNKCJA	PRZYKŁAD
min()	Wyszukiwanie wartości minimalnej. Przykład: <pre>import numpy as np t=np.array([[3, 9, 1], [-2, 2, 6]]) # Dwuwymiarowa print(t.min()) # Wypisze: -2</pre>
max()	Wyszukiwanie wartości maksymalnej. W nawiązaniu do tablicy z poprzedniego przykładu: t.max() zwróci 9.
sum()	Suma wszystkich wartości w tablicy, t.sum() zwróci 19.
mean()	Średnia z wszystkich wartości w tablicy. Wywołanie t.mean() zwróci 3.1666666666666665.
sort()	Sortuje zawartość macierzy, używając szybkiego algorytmu o nazwie Quicksort, i zwraca nową, posortowaną: <pre>import numpy as np t1=np.array([[3, 9, 1], [-2, 2, 6]]) # Dwuwymiarowa t2=np.array([3, 9, 1, -2, 2, 6]) # Jednowymiarowa tsort1=np.sort(t1) # Zwróci: [[1 3 9], [-2 2 6]] tsort2=np.sort(t2) # Zwróci: [-2 1 2 3 6 9]</pre> Zwróć uwagę na sposób sortowania — w tablicach 2D sortowane są każde wiersze osobno. Tablice można sortować też „w miejscu”, wówczas modyfikacji ulegnie bieżący obiekt. Aby tak zrobić, wywołaj sortowanie jako metodę, np. t1.sort().
+	Jeśli pamiętasz zajęcia z matematyki, to na pewno ucieszysz się, że na wartościach zawartych w tablicach (macierzach) NumPy można wykonywać klasyczne operacje matematyczne, takie jak dodawanie, odejmowanie, mnożenie, potęgowanie, po prostu używając standardowych operatorów w wyrażeniach odnoszących się do tabel:
-	
*	
**	
(operatory zastosowane pomiędzy dwoma lub więcej tablicami)	<pre>import numpy as np a=np.array([[3, 9, 1], [-2, 2, 6]]) # Dwuwymiarowa b=np.array([[2, 2, 2], [4, 0, 1]]) # Dwuwymiarowa a+b # Zwróci: [[5 11 3], [2 2 7]] a-b # Zwróci: [[1 7 -1], [-6 2 5]] a*b # Zwróci: [[6 18 2], [-8 0 6]] a**b # Zwróci: [[9 81 1], [16 1 6]]</pre>

```
print("-t      = ", -t)          # [-5 -10 -15 -20]
print("t ** 2 = ", t ** 2)      # [25 100 225 400]
print("t % 2 = ", t % 2)       # [1 0 1 0]
```

Oprócz możliwości opisanych wcześniej w bibliotece znajdziesz bogaty zestaw funkcji realizujących złożone operacje matematyczne przeprowadzane na elementach tablicy w sposób hurtowy i zwracające wyniki zawarte w tablicach o takich samych wymiarach jak tablice zawierające dane wejściowe — nazywane są one unfunc od ang. *universal functions*.

Funkcje te są bardzo wydajne, gdyż nie wymagają znanego z list Pythona powolnego przechodzenia przez wiersze (i ewentualnie kolumny) w celu przeliczenia wartości każdej komórki danych (pętle for).

Przykład:

```
import numpy as np
mojekaty = np.array( [0, 30, 45, 60, 90, 180, 270, 360] )
sinusy=np.sin(mojekaty*np.pi/180) # Konwersja z kątów 0...360 stopni na radiany
cosinusy=np.cos(mojekaty*np.pi/180) # jw.
tangensy=np.tan(mojekaty*np.pi/180) # jw.
```

Oto kilka pozostałych klasycznych funkcji często używanych w praktyce przetwarzania danych (składnia użycia jak w przykładzie wyżej):

```
t1 = np.array([5, 10, 15, 20])
t2 = np.array([2, 1,  2, 4])
t3 = np.array([9, 16, 8, 81])
print(np.negative(t1)) # Negacja: [-5 -10 -15 -20]
print(np.add(t1, t2))  # Dodawanie: [7 11 17 24]
print(np.subtract(t1, t2)) # Odejmowanie: [5 10 15 20]
print(np.multiply(t1, t2)) # Mnożenie: [14 11 34 96]
print(np.divide(t1, t2))   # Dzielenie: [3.5 11. 8.5  6]
print(np.power(t1, t2))   # Potęgowanie: [25 10 225 160000]
print(np.sqrt(t3))        # Pierwiastek kwadratowy: [3.  4. 2.82842712 9]
```

Lista funkcji jest bardzo długa i jeśli musisz odszukać inne, to zachęcam do zapamiętania następujących, bezpośrednich odsyłaczy do dokumentacji NumPy: <https://numpy.org/doc/stable/numpy-ref.pdf> lub <https://numpy.org/doc/stable/reference/generated/numpy.ufunc.html>.

W praktyce bardzo przydatne są metody pozwalające na utworzenie tablicy i wypełnienie jej pożądaną sekwencją liczbową lub wartościami losowymi (*numpy1.py*):

```
import numpy as np
t0 = np.linspace(0, 10, 5) # Przedział od 0 do 10 podzielony na 5 wartości
print("t0=", t0)
t1 = np.arange(1,11)        # Jednowymiarowa, typ domyślny, seria liczb od 1 do 10
t2 = np.arange(1,11, 3)     # Jednowymiarowa, typ domyślny, seria liczb od 1 do 10, co 3
t3 = np.zeros(9)            # Jednowymiarowa, rozmiar 9, wypełniona zerami (float)
# Jednowymiarowa, rozmiar 9, wypełniona 1 (tu: int, typ domyślny to numpy.float64):
t4 = np.ones(9, dtype='i' )
# Jednowymiarowa, rozmiar 4, wypełniona wartościami pseudolosowymi z przedziału [0.0, 1.0):
t5= np.random.random(4)
# Jednowymiarowa, rozmiar 9, wypełniona wartościami pseudolosowymi int (przedział [5,10):
t6= np.random.randint(5,10, 9)
# Tablica 3x5, wypełniona losowymi wartościami z przedziału [0, 3):
t7= np.random.randint(3, size=(3, 5))
print("t1=", t1)
t1=t1.reshape( (2,5) )      # Zmiana kształtu z 1D na 2D (konkretnie 2x5) (*)
print("t1 po reshape(2,5)=", t1)
print("t2=", t2)
```



```
print("t3=", t3)
print("t4=", t4)
print("t5=", t5)
print("t6=", t6)
print("t7=", t7)
```

Jeśli uruchomisz ten program, to ujrzysz następujące wyniki (nieco je przeformatowałem w książce w celu zwiększenia czytelności wydruku):

```
t0= [0.  2.5  5.   7.5 10.]
t1= [1  2  3  4  5  6  7  8  9 10]
t1 po reshape(2,5)= [ [1  2  3  4  5]
                       [6  7  8  9 10] ]

t2= [1  4  7 10]
t3= [0. 0. 0. 0. 0. 0. 0. 0. 0.]
t4= [1 1 1 1 1 1 1 1 1]
t5= [0.20019794 0.59383447 0.03634617 0.50479277]
t6= [5 6 7 8 9 6 8 8 8]
t7= [ [1 2 0 2 2]
      [0 1 2 1 0]
      [2 0 1 1 2] ]
```

Zmiany układu i rozmiaru tablic NumPy

W poprzednim listingu zazaczyłem wiersz (*). Zajrzyj tam teraz.

Zwróć szczególną uwagę na wiersze:

```
t1 = np.arange(1,11) # Wstępna deklaracja tablicy jednowymiarowej (seria liczb od 1 do 10)
t1=t1.reshape( (2,5) ) # Zmiana kształtu z 1D na 2D (konkretnie 2x5) (*)
```

Pierwszy wiersz tworzy klasyczną, jednowymiarową tablicę, wstępnie wypełnioną wartościami od 1. do 10.:

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

Instrukcje zawarte w drugim wierszu zamieniają naszą tablicę na jej wersję dwuwymiarową (2x5, dwa wiersze po 5 wartości):

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

Ponieważ metoda reshape (pol. *przekształcenie*, od ang. *shape*, czyli *kształt* lub *sylwetka*) zwraca *nową* tablicę, z identyczną zawartością, ale w zmodyfikowanym układzie, to w celu posługiwania się poprzednim identyfikatorem zmiennej konieczne było przypisanie `t1=t1.reshape((2,5))`.

Pomimo wyartykułowanego w ramce ostrzeżenia w NumPy możliwe jest jednak zmodyfikowanie rozmiaru w górę (czyli powiększenie tabeli). Ponieważ jednak dane w tablicach nie rodzą się z powietrza, to powinno być oczywiste, że takie sztuczne powiększenie będzie zawierało stare dane plus pewne puste miejsca. Komputery jednak nie rozumieją pojęcia pustki i metoda powiększająca tablicę, która nazywa się `resize()`, przy zmienianiu rozmiaru wypełni puste miejsca po prostu zerami (dla tablicy liczbowej, kontynuacja *numpy1.py*):

```
t8 = np.arange(1,11) # Jednowymiarowa, typ domyślny, seria liczb od 1 do 10
print("t8=", t8)
t8.resize(4,5) # Rozszerzenie z 10 elementów na 4x5, czyli 20
print("t8 po resize (4,5) =", t8)
```

Wynik będzie następujący (nieco przeformatowałem układ wyświetlanych informacji w książce w celu zwiększenia czytelności wydruku):

```
t8= [1  2  3  4  5  6  7  8  9 10]
t8 po resize (4,5) = [ [1  2  3  4  5]
                       [6  7  8  9 10]
                       [0  0  0  0  0]
                       [0  0  0  0  0] ]
```

Odwrotnością metody reshape() jest metoda ravel(), która spłaszcza *N*-wymiarową tablicę NumPy:

```
t=np.array([ [-2, 1, 7], [4, -5, 9], [2, 0, 3] ])
print(t)
print("Spłaszczanie wierszami:", t.ravel(order='C')) # Tryb domyślny, parametr można # ominąć
print("Spłaszczanie kolumnami:", t.ravel(order='F'))
```

Wynik:

```
[ [-2  1  7]
  [ 4 -5  9]
  [ 2  0  3] ]
Spłaszczanie wierszami: [-2  1  7  4 -5  9  2  0  3]
Spłaszczanie kolumnami: [-2  4  2  1 -5  0  7  9  3]
```

Jak się okazuje, to nie są wszystkie możliwości, jakie oferuje biblioteka NumPy.

Bardzo interesujące efekty daje rozszerzanie tablic horyzontalnie lub pionowo, czyli tak naprawdę tworzenie nowych tablic z już wcześniej utworzonych.

Brzmi to groźnie, ale sprawa jest dość prosta, co pokażę na przykładach.

Rozszerzanie i transpozycje

Po zapoznaniu się z podstawami pójdźmy teraz o krok dalej w naszych ćwiczeniach z NumPy — pozwolą nam one docenić niesamowitą giętkość oferowanych przez nią tablic.



Biblioteka NumPy pozwoli na zmiany kształtu za pomocą metody reshape() pod warunkiem zgodności *liczby elementów*. Przykładowo, tablicy zawierającej 5 liczb nie można zamienić metodą reshape() na macierz o rozmiarze 4x6 (24 elementy). „Kształt” tablicy NumPy możesz zawsze zdekodować, odczytując w notacji z kropką wartość atrybutu shape.