1. Wektory – wprowadzenie

Czym jest wektor w python?

Wektor w prostym terminie można uznać za jednowymiarową tablicę. W odniesieniu do Pythona wektor to jednowymiarowa tablica list.

NumPy

Klasa NumPy ndarray służy do reprezentowania zarówno macierzy, jak i wektorów. Wektor to tablica z pojedynczym wymiarem (nie ma różnicy między wektorami wierszowymi i kolumnowymi), podczas gdy macierz odnosi się do tablicy o dwóch wymiarach.

Tworzenie wektorów – python

1. Importujemy bibliotekę NumPy.
2. Wykorzystując funkcję np.array() wpisujemy dane do naszego wektora.

```
import numpy as np

my_vector1 = np.array([1,2,3,4,5])
print("My vector: ", my_vector1)
```

```
My vector:  [1 2 3 4 5]
```

Innym sposobem na stworzenie wektora jest wykorzystanie funkcji np.arange(), która posiada dokładnie określoną ilość elementów z jakiegoś konkretnego przedziału. Funkcja działa podobnie jak range() przyjmując start, stop i step, i zwraca jednowymiarową tablicę liczb zdefiniowanych tymi parametrami.

```
[6]  import numpy as np

     my_vector2 = np.arange(5)
     my_vector3 = np.arange(0, 10, 2)

     print("My vector 2: ", my_vector2)
     print("My vector 3: ", my_vector3)
```

```
My vector 2:  [0 1 2 3 4]
My vector 3:  [0 2 4 6 8]
```

Jeszcze innym sposobem na tworzenie wektora jest użycie funkcji np.linspace(), w której spośród trzech parametrach definiujemy również start i stop, ostatni parametr jednak określa na ile równo oddalonych od siebie wartości wydzielić z przedziału [start, stop]. Bardzo przydatna metoda przy generowaniu zbioru wartości potrzebnego do zbudowania wykresu.

```
import numpy as np

my_vector4 = np.linspace(1,np.pi,4)
print("My vector 4: ", my_vector4)
```

```
My vector 4:  [1.         1.71386422 2.42772844 3.14159265]
```

Operacje na wektorach

Jedna z podstawowych operacji na wektorach jest dodawanie lub usuwanie elementów z listy. Do wstawiania nowych wartości na końcu wektora służy funkcja np.append(), w której jako zmienne podajemy wektor oraz wartość, która ma zostać dodana.

```
import numpy as np

my_vector4 = np.arange(5)

new_vector = np.append(my_vector4,[1000])
print("New vector with 1000 at the end: ", new_vector)
```

```
New vector with 1000 at the end:  [   0    1    2    3    4 1000]
```

Natomiast wykorzystując funkcję np.insert(), możemy wpisać nowe wartości w dowolnym miejscu(przez podanie indeksu)

```
[21] import numpy as np

     my_vector4 = np.arange(5)

     new_vector_insert = np.insert(my_vector4, 1, 8888)
     print("Vector with value 8888 at the index 1: ", new_vector_insert)
```

```
Vector with value 8888 at the index 1:  [   0 8888    1    2    3    4]
```

Aby usunąć konkretną wartość z wektora, należy użyć funkcji np.delete()

```
import numpy as np

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

index = 0
arr2 = np.delete(arr, index)
print("Based array:",arr, "Array with removed value[0]: ", arr2 )
```

```
Based array: [1 2 3 4 5] Array with removed value[0]:  [2 3 4 5]
```

Działania na wektorach

- Dodawanie
  Podczas dodawania dwóch wektorów należy pamiętać, że muszą posiadać one taki sam wymiar. Co za tym indzie: jeśli mamy wektor [1 2 3 4] , który jest 4-elementowy, nie możemy dodać do niego wektora np. [1 2], ponieważ mają one inny rozmiar.

```
[24] import numpy as np

     vec1 = np.array([1, 2, 3, 4, 5])
     vec2 = np.array([1, 2, 3, 4, 5])

     adding1 = np.add(vec1,vec2)
     adding2 = vec1+vec2

     print("After adding uusing function: ", adding1)

     print("After adding without function: ", adding2)
```

```
     After adding uusing function:  [ 2  4  6  8 10]
     After adding without function:  [ 2  4  6  8 10]
```

- Odejmowanie
  Przy odejmowaniu wektorów postępujemy analogicznie jak w przypadku dodawania. Możemy użyć funkcji wbudowanej lub znaku „ - ”

```
[26] import numpy as np

     vec1 = np.array([1, 2, 3, 4, 5])
     vec2 = np.array([5, 4, 3, 2, 1])

     subtract1 = np.subtract(vec1,vec2)
     subtract2 = vec1-vec2

     print("After subtracting using function: ",subtract1)

     print("After subtracting without function: ", subtract2 )
```

```
     After subtracting using function:  [-4 -2  0  2  4]
     After subtracting without function:  [-4 -2  0  2  4]
```

- Mnożenie przez skalar

```
import numpy as np

vec1 = np.array([1, 2, 3, 4, 5])
scalar = 3
result = vec1 * scalar
print(result)
```

```
[ 3  6  9 12 15]
```

- Mnożenie i dzielenie

```
[29] import numpy as np

     vec1 = np.array([1, 2, 3, 4, 5])
     vec2 = np.array([0, 2, 3, 0, 2])

     multiply1 = np.multiply(vec1,vec2)
     multiply2 = vec1*vec2

     print("After multipled using function: ",multiply1)

     print("After multipled without function: ", multiply2 )
```

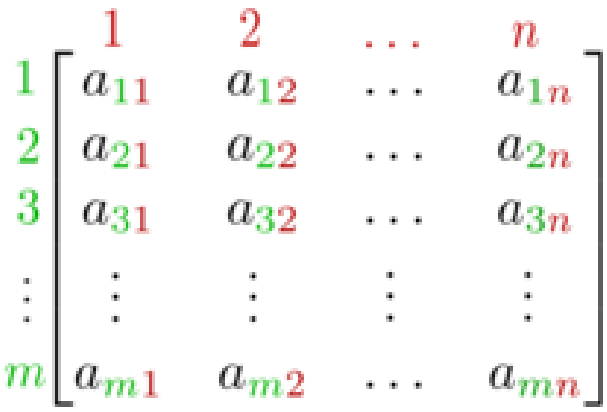
```
After multipled using function:  [ 0  4  9  0 10]
After multipled without function:  [ 0  4  9  0 10]
```

```
import numpy as np

vec1 = np.array([1, 2, 3, 4, 15])
vec2 = np.array([1, 2, 3, 2, 5])

divide1 = np.divide(vec1,vec2)
divide2 = vec1/vec2

print("After divided using function: ",divide1)

print("After divided without function: ", divide2 )
```

```
After divided using function:  [1. 1. 1. 2. 3.]
After divided without function:  [1. 1. 1. 2. 3.]
```

2. Macierze

Macierz to dwuwymiarowa struktura danych, w której liczby są ułożone w wiersze i kolumny.



Macierze w python

Python nie ma wbudowanego typu dla macierzy. Możemy jednak traktować listę listy jako macierz.

```
import numpy as np

matrix1 = np.array([[1, 2, 3, 4, 15], [2, 4, 6, 8, 10]])
print(matrix1)
```

```
[[ 1  2  3  4 15]
 [ 2  4  6  8 10]]
```

```
import numpy as np

x = np.array([1,2,3,4,5,6,7,8,9]).reshape(3,3)
x
```

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

Oprócz obiektów typu `array` istnieje wyspecjalizowany obiekt `matrix`, dla którego operacje `*` (mnożenie) oraz `**-1` (odwracanie) są określone w sposób właściwy dla macierzy (w przeciwieństwu do operacji elementowych dla obietków `array`).

```
import numpy as np

m = np.array([1,2,3,4,5,6,7,8,9]).reshape(3,3)

X = np.matrix(m)
X
```

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

Poruszanie się po macierzach w python

W celu odniesienia się do konkretnego elementu macierzy musimy znać jej „współrzędne” tj. w jakim miejscu listy znajduję się interesująca nas wartość.

Skoro Macierz jest tzw. listą w liście, to analogicznie przeszukiwać ją możemy tak samo jak listy.

```
[[ 1  2  3 ],
  [0][0] [0][1] [0][2]

 [ 4  5  6 ],
   [1][0] [1][1] [1][2]

 [ 7  8  9 ]]
   [2][0] [2][1] [2][2]
```

Mając powyższą macierz, widzimy, że chcąc „wyjąć” wartość 8 z macierz, należy odwołać się do miejsca, w której się znajduje tj. [2][1]. Odczytać zapis możemy w sposób:

Wartość 8 w powyższej macierzy znajduje się w trzeciej w kolejności liście (ponieważ indeksy listy zaczynają się od 0) na miejscu drugim.

```python
import numpy as np

x = np.array([1,2,3,4,5,6,7,8,9]).reshape(3,3)

first_value= x[0][0]
middle_value = x[1][1]
last_value = x[-1][-1]

print("first element in matrix", first_value)
print("middle element in matrix", middle_value)
print("last element in matrix", last_value)
```

```
first element in matrix 1
middle element in matrix 5
last element in matrix 9
```

Macierz diagonalna

W python w prosty sposób możemy otrzymać macierz diagonalną. Biblioteka NumPy posiada wbudowaną funkcję np.diag(), dzięki której szybko otrzymujemy :

```python
[47] import numpy as np


     a = np.diag((1,2,3,4))
     a
```

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

Inne gotowe funkcje NumPy na tworzenie macierzy

Klasa NumPy posiada dodatkowo funkcje na tworzenie macierzy wypełnionej samymi zerami (macierz zerowa) lub wypełnionej jednynkami.

```python
import numpy as np


zeros = np.zeros((3,3))
print(zeros)

ones = np.ones((3,3))
print(ones)
```

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

Zmiana wartości elementów w istniejącej macierzy

Aby zmienić wartość konkretnego elementu w macierzy, należy odwołać się do niego po indeksie, po czym zadeklarować nową wartość.

```
x = zeros.copy()
x[0][2]=6

print(x)
```

```
[[0. 0. 6.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Dodawanie i odejmowanie macierzy

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Powyższa wizualizacja dodawania macierzy mówi nam:

- Dodajemy wartości odpowiednio po indeksach, które również umieszczamy na wskazanej przez nie pozycji.

Należy pamiętać, że aby operacje dodawania lub odejmowania macierzy mogło zaistnieć musimy posiadać macierze tych samych wymiarów.

3. Tensory

Tensor można opisać jako n-wymiarową tablicę liczbową , którą można nazwać macierzą uogólnioną. Może to być macierz 0-D (pojedyncza liczba), macierz 1-D (wektor), macierz 2-D lub dowolna struktura o wyższych wymiarach.

Tensowy w Python

Tensor jest identyfikowany przez trzy parametry, a mianowicie rangę, kształt i rozmiar. O liczbie wymiarów tensora mówi się, że jest jego rangą. Mówi się, że liczba kolumn i wierszy, które ma tensor, jest jego kształtem. Mówi się, że typ danych przypisany do elementów tensora jest jego typem.

Tworzenie tensorów w python

Tensory w Python można tworzyć na kilka sposobów. Jednym z nich jest użycie biblioteki NumPy (np.array()), innym rozwiązaniem jest wykorzystanie biblioteki TensorFlow.

Tensorflow

Tensor można nazwać centralnym typem danych Tensorflow. Dzieje się tak, ponieważ tensory są podstawowymi składnikami obliczeń w ramach Tensorflow. Jak sama nazwa wskazuje, Tensorflow to framework, który obejmuje definiowanie i wykonywanie obliczeń z wykorzystaniem tensorów.

Przykład implementacji tensorów w bibliotece TensorFlow:

```
# using the constant() function
import tensorflow as tf

t1 = tf.constant([1, 2, 3])
t2 = tf.constant([[1.1, 2.2, 3.3], [4, 5, 6]])
t3 = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
t4 = tf.constant(["String_one", "String_two", "String_three"])

print(t1)
print("\n")

print(t2)
print("\n")

print(t3)
print("\n")

print(t4)
```

ZADANIA DO WYKONANIA

Zadanie 1

Stwórz wektor [1 4 5 6 2 1 5 6 7 0]

a) **Usuń wartość o indeksie [5]**
b) **Dodaj element, o wartości 8 na końcu wektora**
c) **Dla wszystkich nieparzystych indeksów wektora dodaj +2 do wartości istniejącej**
d) **Odwróć wektor tj. indeks ostatni staje się pierwszym itd…**

Zadanie 2

Wyjaśnij i pokaż na przykładzie czym różni się mnożenie wektora przez skalar powstałego za pomocą funkcji np.array od listy zawierającej tak samo jak wektor numpy ciąg liczb.

Zadanie 3

Stwórz macierz:

$$\begin{bmatrix} 3 & 5 & 0 \\ 2 & 6 & 1 \\ 3 & 8 & 9 \end{bmatrix}$$

a) **Zmień wartość pierwszego elementu macierzy na -2**
b) **Zmień wartość elementu położonego w drugim wierszu, drugiej kolumnie na 44**
c) **Zmień wartość ostatniego elementu macierzy na 0**

Zadanie 4

Mając do dyspozycji macierz z zadania 3, zmień elementy o indeksach parzystych na 0.

Zadanie 5

Wykorzystując pętlę for w Python wykonaj następujące działanie na macierzach:

$$\begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 6 \\ 4 & 5 & 5 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 5 \\ 2 & 1 & 6 \\ 0 & 3 & 0 \end{bmatrix} = ?$$

Zadanie 6

Wykorzystując bibiotekę TensorFlow stwórz tensor o wymiarach 4x4. Dokładnie opisz otrzymane dane.