Wyklad10

December 13, 2019

1 NumPy - DIY

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

[2]:
```
x_np = np.array([3, 1, 2]) #dwa przykładowe wektory
y_np = np.array([1, 2, 3])
```

[3]:
```
print(x_np+y_np) #przykladowa operacja na wektorach
```

```
[4 3 5]
```

[4]:
```
print(x_np*y_np)
```

```
[3 2 6]
```

[5]:
```
np.linalg.norm(x_np)  #norma wektora x_np
```

[5]:
```
3.7416573867739413
```

[6]:
```
from math import sqrt
sqrt(x_np.dot(x_np)) #sqrt(3*3+1*1+2*2)=sqrt(14)
```

[6]:
```
3.7416573867739413
```

A jak zrobić to samemy?

[7]:
```
x_list = [3, 1, 2] #metoda oparta na listach
y_list = [1, 2, 3]
```

[8]:
```
from math import sqrt

def vector_add(v1,v2):
    if not len(v1)==len(v2):
        print("Wektory maja rozna dlugosc")
        return 0
    else:
        return [v1[i]+v2[i] for i in range(len(v1))]
```

```
def vector_times(v1,v2):
    if not len(v1)==len(v2):
        print("Wektory maja rozna dlugosc")
        return 0
    else:
        return [v1[i]*v2[i] for i in range(len(v1))]

def vector_norm(v1):
    try:
        v3=sum(vector_times(v1,v1))
        return sqrt(v3)
    except:
        return 0
```

[9]:
```
print(vector_add(x_list,y_list)) #zamiast x_list+y_list: [3, 1, 2, 1, 2, 3]
```

```
[4, 3, 5]
```

[10]:
```
print(vector_times(x_list,y_list)) #zamiast x_list*y_list
```

```
[3, 2, 6]
```

[11]:
```
print(vector_norm(x_list))
```

```
3.7416573867739413
```

A co z błędami?

[12]:
```
print(vector_add([1,2,3],[1,2,3,4]))
```

```
Wektory maja rozna dlugosc
0
```

[13]:
```
print(vector_times([1,2,3],[1,2,'a'])) # niby poprawnie,ale...
```

```
[1, 4, 'aaa']
```

[14]:
```
print(vector_norm([1,2,'a'])) #norma już nie dziala
```

```
0
```

[15]:
```
print(vector_times([1,2,3],[1,2,3j]))
print(vector_norm([1,2,3j]))
```

```
[1, 4, 9j]
0
```

[16]:
```
from math import sqrt

def vector_add2(v1,v2):
    if not len(v1)==len(v2):
        print("Wekroty maja rozna dlugosc")
        return 0
    else:
        return [v1[i]+v2[i] for i in range(len(v1))]


def vector_times2(v1,v2):
    if not len(v1)==len(v2):
        print("Wektory maja rozna dlugosc")
        return 0
    elif len(list(filter(lambda x:type(x)==str,v1+v2))):  #funkcja filter() -␣
↪warto znać
        print("Zle wektory")
        return 0
    else:
        return [v1[i]*v2[i] for i in range(len(v1))]


def vector_norm2(v1):
    try:
        v1c=list(map(lambda i:complex(i).conjugate(),v1)) #funkcja map() - warto␣
↪znać
        v3=sum(vector_times2(v1,v1c)).real
        return sqrt(v3)
    except:
        print("Zly wektor")
        return 0
```

[17]:
```
print(vector_times2([1,2,3],[1,2,'a']))
```

```
Zle wektory
0
```

[18]:
```
print(vector_times2([1,2,3],[1,2,3j]))
print(vector_norm2([1,2,3j]))
```

```
[1, 4, 9j]
3.7416573867739413
```

Kilka uwag:

[19]:
```
list(filter(lambda x:type(x)==str,[1,2,'a']))
```

[19]:
```
['a']
```

[20]:
```
list(map(lambda i:complex(i).conjugate(),[1,2,3]))
```

[20]:
```
[(1-0j), (2-0j), (3-0j)]
```

Można też troche ładniej

[21]:
```
class Wektor:
    """Trójwymiarowy wektor"""
    def __init__(self, x=0.0, y=0.0, z=0.0): #wazne: __init__ i self
        self.x = x
        self.y = y
        self.z = z
```

[22]:
```
x_vector=Wektor(3,1,2) #wywolanie funkcji __init__(x_vector,3,1,2)
```

[23]:
```
type(x_vector)
```

[23]:
```
__main__.Wektor
```

[24]:
```
print(x_vector.x,x_vector.y,x_vector.z)
```

```
3 1 2
```

[25]:
```
y_vector=Wektor(z=3,x=1,y=2)
print(y_vector.x)
```

```
1
```

[26]:
```
z_vector=Wektor(z=3.)
print(z_vector.x,z_vector.y,z_vector.z)
```

```
0.0 0.0 3.0
```

init jest jedną z metod specjalnych klasy. Pozostałe metody dostępne są w: https://docs.python.org/3/reference/datamodel.html#special-method-names

[27]:
```
class Wektor:
    """Trójwymiarowy wektor"""

    def __init__(self, x=0.0, y=0.0, z=0.0): #wazne: __init__ i self

        if complex in [type(x),type(y),type(z)]:
            type_vec=complex
        elif float in [type(x),type(y),type(z)]:
            type_vec=float
        else:
            type_vec=int
        #------------------------
```

```
        try:
            self.x = type_vec(x)
            self.y = type_vec(y)
            self.z = type_vec(z)
        except:
            print("Zle dane wejsciowe")

    def __str__(self):  # "nieformalna" reprezentacja obiektu: zwraca string
        return "Wektor: {}, {}, {}".format(self.x, self.y, self.z)

    def __repr__(self):  # "oficjalna" reprezentacja obiektu - zwraca string
        return "[{}, {}, {}]".format(self.x, self.y, self.z)
```

[28]:
```
x_vector=Wektor(3,1,2)
y_vector=Wektor(z=3,x=1,y=2)
z_vector=Wektor(z=3.)
```

[29]:
```
print(z_vector)  #odwolanie do __str__
```

```
Wektor: 0.0, 0.0, 3.0
```

[30]:
```
z_vector #odwolanie do __repr__
```

[30]:
```
[0.0, 0.0, 3.0]
```

[31]:
```
Wektor(3,1,2.)
```

[31]:
```
[3.0, 1.0, 2.0]
```

[32]:
```
Wektor(3,1,2.j)
```

[32]:
```
[(3+0j), (1+0j), 2j]
```

[ ]:
```
Wektor(3,1,'2.')
```

[34]:
```
class Wektor:
    """Trójwymiarowy wektor"""

    def __init__(self, x=0.0, y=0.0, z=0.0): #wazne: __init__ i self

        if complex in [type(x),type(y),type(z)]:
            type_vec=complex
        elif float in [type(x),type(y),type(z)]:
            type_vec=float
        else:
            type_vec=int
        #------------------------
```

```
        try:
            self.x = type_vec(x)
            self.y = type_vec(y)
            self.z = type_vec(z)
        except:
            print("Zle dane wejsciowe")

    def __str__(self):  # "nieformalna" reprezentacja obiektu: zwraca string
        return "Wektor: {}, {}, {}".format(self.x, self.y, self.z)

    def __repr__(self):  # "oficjalna" reprezentacja obiektu - zwraca string
        return "[{}, {}, {}]".format(self.x, self.y, self.z)

    def __add__(self, w):  # operator dodawania
        return Wektor(self.x + w.x, self.y + w.y, self.z + w.z)


x_vector=Wektor(3,1,2)
y_vector=Wektor(z=3,x=1,y=2)
z_vector=Wektor(z=3.)
```

[35]:
```
x_vector+y_vector #rownowazne x_vector.__add__(y_vector)
```

[35]:
```
[4, 3, 5]
```

[36]:
```
class Wektor:
    """Trójwymiarowy wektor"""

    def __init__(self, x=0.0, y=0.0, z=0.0): #wazne: __init__ i self

        if complex in [type(x),type(y),type(z)]:
            type_vec=complex
        elif float in [type(x),type(y),type(z)]:
            type_vec=float
        else:
            type_vec=int
        #------------------------
        try:
            self.x = type_vec(x)
            self.y = type_vec(y)
            self.z = type_vec(z)
        except:
            print("Zle dane wejsciowe")

    def __str__(self):  # "nieformalna" reprezentacja obiektu: zwraca string
        return "Wektor: {}, {}, {}".format(self.x, self.y, self.z)
```

```
    def __repr__(self):  # "oficjalna" reprezentacja obiektu - zwraca string
        return "[{}, {}, {}]".format(self.x, self.y, self.z)

    def __add__(self, w):  # operator dodawania
        return Wektor(self.x + w.x, self.y + w.y, self.z + w.z)

    def __mul__(self, w):          # operator mnożenia
        return Wektor(self.x*w.x, self.y*w.y, self.z*w.z)

x_vector=Wektor(3,1,2)
y_vector=Wektor(z=3,x=1,y=2)
z_vector=Wektor(z=3.)
```

[37]:
```
print(x_vector)
print(y_vector)
x_vector*y_vector #rownowazne x_vector.__mul__(y_vector)
```

```
Wektor: 3, 1, 2
Wektor: 1, 2, 3
```

[37]: [3, 2, 6]

[38]:
```
class Wektor:
    """Trójwymiarowy wektor"""

    def __init__(self, x=0.0, y=0.0, z=0.0): #wazne: __init__ i self

        if complex in [type(x),type(y),type(z)]:
            type_vec=complex
        elif float in [type(x),type(y),type(z)]:
            type_vec=float
        else:
            type_vec=int
        #------------------------
        try:
            self.x = type_vec(x)
            self.y = type_vec(y)
            self.z = type_vec(z)
        except:
            print("Zle dane wejsciowe")

    def __str__(self):  # "nieformalna" reprezentacja obiektu: zwraca string
        return "Wektor: {}, {}, {}".format(self.x, self.y, self.z)

    def __repr__(self):  # "oficjalna" reprezentacja obiektu - zwraca string
        return "[{}, {}, {}]".format(self.x, self.y, self.z)
```

```
    def __add__(self, w):  # operator dodawania
        return Wektor(self.x + w.x, self.y + w.y, self.z + w.z)

    def __mul__(self, w):          # operator mnożenia
        if type(w) == type(self):  # dla wektora - iloczyn składowych wektorow␣
↪self i w
            return Wektor(self.x*w.x, self.y*w.y, self.z*w.z)
        else:                      # dla liczby - mnożenie składowych wektora␣
↪self i liczby
            return Wektor(w*self.x, w*self.y, w*self.z)

x_vector=Wektor(3,1,2)
y_vector=Wektor(z=3,x=1,y=2)
z_vector=Wektor(z=3.)
```

[39]:
```
print(x_vector)
print(y_vector)
```

```
Wektor: 3, 1, 2
Wektor: 1, 2, 3
```

[40]:
```
x_vector*10
```

[40]:
```
[30, 10, 20]
```

[41]:
```
10*x_vector
```

```
      ␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call last)

       <ipython-input-41-b43b69b8e1e6> in <module>
    ----> 1 10*x_vector


       TypeError: unsupported operand type(s) for *: 'int' and 'Wektor'
```

[42]:
```
class Wektor:
    """Trójwymiarowy wektor"""

    def __init__(self, x=0.0, y=0.0, z=0.0): #wazne: __init__ i self

        if complex in [type(x),type(y),type(z)]:
            type_vec=complex
```

```
        elif float in [type(x),type(y),type(z)]:
            type_vec=float
        else:
            type_vec=int
        #------------------------
        try:
            self.x = type_vec(x)
            self.y = type_vec(y)
            self.z = type_vec(z)
        except:
            print("Zle dane wejsciowe")

    def __str__(self):  # "nieformalna" reprezentacja obiektu: zwraca string
        return "Wektor: {}, {}, {}".format(self.x, self.y, self.z)

    def __repr__(self):  # "oficjalna" reprezentacja obiektu - zwraca string
        return "[{}, {}, {}]".format(self.x, self.y, self.z)

    def __add__(self, w):  # operator dodawania
        return Wektor(self.x + w.x, self.y + w.y, self.z + w.z)

    def __mul__(self, w):           # operator mnożenia
        if type(w) == type(self):  # dla wektora - iloczyn składowych wektorow␣
↪self i w
            return Wektor(self.x*w.x, self.y*w.y, self.z*w.z)
        else:                       # dla liczby - mnożenie składowych wektora␣
↪self i liczby
            return Wektor(w*self.x, w*self.y, w*self.z)

    def __rmul__(self, w):  # mnożenie z prawej
        return self.__mul__(w)

    def vec_conj(self):
        return Wektor(complex(self.x).conjugate(), complex(self.y).conjugate(),␣
↪complex(self.z).conjugate())

    def Norma(self):  # mnożenie z prawej
        nv=self.__mul__(self.vec_conj())
        return sqrt((nv.x+nv.y+nv.z).real)



x_vector=Wektor(3,1,2)
y_vector=Wektor(z=3,x=1,y=2)
z_vector=Wektor(z=3.)
```

[43]:
```
10*x_vector
```

[43]:
```
[30, 10, 20]
```

[44]:
```
(x_vector*(1+0j)).vec_conj()
```

[44]:
```
[(3-0j), (1-0j), (2-0j)]
```

[45]:
```
x_vector.Norma()
```

[45]:
```
3.7416573867739413
```

A co dalej?

[46]:
```
class WektorN:
    """Trójwymiarowy wektor"""
    def __init__(self, *args):
        setattr(self, "xlen", len(args))
        for idx, item in enumerate(args):
            setattr(self, "x{}".format(idx), item)

    def __str__(self):  # "nieformalna" reprezentacja obiektu: zwraca string
        wynik="Wektor: "
        for i in range(self.xlen):
            wynik+="{}, ".format(eval("self.x{}".format(i)))
        return wynik

    def __repr__(self):  # "nieformalna" reprezentacja obiektu: zwraca string
        wynik="["
        for i in range(self.xlen-1):
            wynik+="{}, ".format(eval("self.x{}".format(i)))
        wynik+="{}]".format(eval("self.x{}".format(i+1)))
        return wynik
```

[47]:
```
x_WN=WektorN(3,1,2,5,5,6)
```

[48]:
```
print(x_WN.xlen)
```

```
6
```

[49]:
```
print(x_WN)
```

```
Wektor: 3, 1, 2, 5, 5, 6,
```

[50]:
```
x_WN
```

[50]:
```
[3, 1, 2, 5, 5, 6]
```

[51]:
```
x_WN.x4
```

[51]: 5

[ ]:
```
cdn...
```