Wstęp do Sztucznej Inteligencji - rok akademicki 2022/2023

Przed rozpoczęciem pracy z notatnikiem zmień jego nazwę zgodnie z wzorem: `NrAlbumu_Nazwisko_Imie_PoprzedniaNazwa`.

Przed wysłaniem notatnika upewnij się, że rozwiązałeś wszystkie zadania/ćwiczenia.

Temat: Wprowadzenie do języka Python cz. I

Zapoznaj się z treścią niniejszego notatnika czytając i wykonując go komórka po komórce. Wykonaj napotkane zadania/ćwiczenia.

Dokumentacja Python 3 (https://docs.python.org/3/).

Komentarze

```
# to jest komentarz jednolinijkowy, interpreter pomija linie
zaczynające się od '#'
# Wykonaj komórkę (Shift+Enter) i zobacz że nic się nie wykona

"""
To jest komentarz wielolinijkowy, wszystko pomiędzy trzema znakami
cudzysłowia (alternatywnie trzema apostrofami)
traktowane jest jako łańcuch znaków. Jeśli nie zostanie on przypisany
do żadnej zmiennej to będzie traktowany jako
komentarz. Często taki sposób wykorzystywany jest do dokumentowania
kodu.
"""
```

Funkcja `print()`

Funkcja `print()` przyjmuje dowolną liczbę argumentów, dowolnego typu (no nie do końca dowolnego) i ma też parę parametrów/opcji, ale o tym innym razem.

Funkcja `print` jest tak zwaną funkcją wbudowaną, inne dostępne i czasami przydatne funkcje wbudowane https://docs.python.org/3/library/functions.html.

```
print('Hello')
print('Hello',1,3,[1,3],2+4j)
print('Hello',1,3,[1,3],2+4j, sep=';',end=':')
```

Podstawowe typy danych

Python jest językiem dynamicznie typowanym, co oznacza, że nie ma potrzeby jawnego określania typu danej zmiennej, interpreter zrobi to za nas sam. Co więcej, Python jest językiem w pełni obiektowy, w którym również zmienne są obiektami.

Typy liczbowe: `int`, `float`, `complex`.

```
# liczba całkowita: int
a = 2    # przypisujemy zmiennej 'a' liczbę całkowitą 2
print(a) # wydruk zmiennej a
type(a)  # wbudowana funkcja type() zwraca typ danego obiektu
```

Przykładowe metody obiektu `int`. Wciskając po napisaniu kropki `Tab` wyświetlone zostaną dostępne pola i metody danego obiektu.

```
a.bit_length()

# liczba zmiennoprzecinkowa: float
b = 2.5
print(b)
type(b)

b.is_integer()

# liczba zespolona: complex
z = 2+4j # j jednostka urojona, piszemy z prawej strony bez spacji
print(z)
type(z)

z.imag
```

Typ `str` (łańcuch znaków).

Domyślnym kodowaniem znaków w Python 3 jest `utf8`.

```
# tekst (string) definiujemy w apostrofach lub cudzysłowach
napis1 = 'wstęp do sztucznej inteligencji,'
napis2 = "wstęp do sztucznej inteligencji."
print(napis1, napis2)
print(type(napis1))
type(napis2)
```

Domyślnie rozpoznawane są również znaki specjalne takie jak `\n`, `\t` itp. Aby potraktować znaki specjalne jako zwykłe znaki trzeba dodać jeszcze jeden znak `\` lub dodać przed apostrofem `r`.

```
print('wstęp do\tsztucznej\ninteligencji')
```

```
print('wstęp do\\tsztucznej\\ninteligencji')
print(r'wstęp do\tsztucznej\ninteligencji')
```

Wbudowana funkcja `len` zwracająca długość łańcucha znaków.

```
print(napis1)
len(napis1)
```

Z łańcucha znaków możemy wybierać dowolny element za pomocą metody `__getitem__` lub prościej za pomocą nawiasów `[]`.

```
s = 'napisssss'
print(s.__getitem__(1))  #numeracja elementów od 0
print(s[1])
```

Obiekt klasy `str` jest obiektem niemodyfikowalnym, tzn. nie można przypisać nowego znaku na danej pozycji.

```
s[2] = 'g'
```

Nie można również podać indeksu większego niż `ilość znaków - 1`.

```
s[100]
```

Slicing ("plasterkowanie")

```
# wybieranie fragmentu
print(s[1:6])
# wybieranie znaków z krokiem
print(s[1:6:2])
# ostatnie element ciągu znaków
print(s[-1])
# od tyłu
print(s[-4:-2])
```

Typ `tuple` (krotka)

Krotka to zestawienie kilku obiektów w jeden obiekt. Mogą to być obiekty różnych typów.

```
a = (1, 2, 3, 4)
b = (1, 3.5, 'a', a)
print(a)
print(b)
print(len(a))
type(a)
```

Elementy krotki można rozpakować do pojedynczych zmiennych. Bardzo przydatne przy definiowaniu funkcji zwracających wiele obiektów.

```
# rozpakowywanie krotki
a,b,c = (1, 'a', 'abc')
d = (1, 'a', 'abc')
print(a)
print(b)
print(c)
print(d)
```

Podobnie jak łańcuch znaków, krotka jest obiektem niemodyfikowalnym.

```
k = (1, 'a')
print(k[0])
k[0]='b'
```

Typ `list` (lista)

Lista może zawierać obiekty dowolnych typów i w odróżnieniu do krotki jej elementy można modyfikować.

```
l = [1, 2, '3', [1, 3]]
print(l)
print(len(l))
type(l)

print(l)
print(l[0])
l[0] = 'nowe_0'
print(l)
```

Uwaga: Kopiowanie list jak również wszystkich innych obiektów w języku Python odbywa się poprzez referencje tzn. nie tworzony jest nowy obiekt a jedynie nadawana jest tak jakby nowa etykieta to tego samego obiektu.

```
l = [1,2,4]
p = l
print(p,l)

p[1] = -5
print(p,l)
```

Aby skopiować tworząc nowy obiekt należy użyć metody `copy()`.

```
l = [1,2,4]
p = l.copy()
print(p,l)
p[1]= 100
print(p,l)
```

Zamiast `copy` można alternatywnie użyć `[:]` co oznacza kopiowanie element po elemencie.

```
l = [1,2,3]
p[:] = l[:]
print(p,l)
p[0] = -100
print(p,l)
```

Dodawanie nowych obiektów do listy (rozszerzanie).

```
p = [1,2,3]
# dodawanie na końcu
p.append('nowy_k')
p

# dodawanie w konkretnym miejscu
p.insert(1,'nowy_i') # podajemy miejsce wstawienia
p

# rozrzerzanie o więcej niż jeden element
p.extend([10,32])
p
```

Typ `set` (zbiór).

```
z = {1,2,3}
type(z)
```

Tworzenie zbioru z listy (unikalne wartości).

```
p = [1,2,3,1,3,5,1]
print(p)
sp = set(p)
sp
```

Typ `dic` (Słownik, {klucz:wartość}).

```
d = {'Nr': 1234, 'Imię': 'Jack', 'Nazwisko': 'Sparrow',}
print(d)
type(d)

# do elementów słownika odwołujemy się poprzez klucze
print(d['Nr'])
print(d['Imię'])
print(d['Nazwisko'])

# klucze w słowniku
d.keys()
```

```
# dodawanie nowego pola w słowniku
d['Ocena'] = 2.0
d
```

Typ `bool` (prawda, fałsz).

```
prawda = True
falsz = False
type(prawda)
```

Operatory arytmetyczne (`+, -, *, /, //, **, %`)

```
# dwie liczby typu int
a = 2
b = 5
print(a+b)
print(a-b)
print(a*b)
print(a/b) # rzutuje na float
print(a//b) # int
print(a**b) # potęgowanie
print(a%b) # reszta z dzielenia, modulo
```

Operatory arytmetyczne definiowane są również dla prawie wszystkich typów danych nie tylko liczbowych. Co ważne, Python jest silnie typowany co oznacza m.in., że nie można dowolnie łączyć typów w operacjach arytmetycznych.

```
2.5 + 'a'
```

Jednak dla niektórych typów działa tak zwana konwersja niejawna (rzutowanie na obiekt klasy "nadrzędnej")

```
a = 2.5 + (2-3j)
print(a)
type(a)
```

Biblioteka matematyczna `math`.

W Python wbudowane są niektóre podstawowe finkcje matematyczne, taki jak `abs`, `min`, `max`, `sum`, `pow`. Inne znaleźć można w bibliotece (module) `math`. Import bibliotek (modułów, pakietów) w języku Python dokonuje się za pomocą słowa kluczowego `import`.

```
print(min(1,2,-4,2))
print(max(2, 1.4, 3, 5.5))
abs(-4)

# aby wczytać pakiet math wpisujemy
import math
```

```
# możemy zaimportowac tylko wybrane funkcje
from math import sin, cos

math.log(1)

sin(math.radians(90))
```

Zadanie 1:

Sprawdź i opisz działanie operatorów arytmetycznych dla typu `list`.

YOUR ANSWER HERE

Zadanie 2:

Sprawdź i opisz działanie operatorów arytmetycznych dla typu `str`.

YOUR ANSWER HERE

Operatory logiczne

Operatory porównania: `==`, `>`, `>=`, `<`, `<=`, `!=`.

Spójniki logiczne: `and`, `or`.

Zaprzeczenie: `not`.

Zawsze zwracany jest typ `bool` (True, False).

```
print(2==3)
print(2!=3)
print(2<3)
a = 4.5
b = 4
print(not 4<=5)
print(a>b and a<b)
print(a>b and a<b or 3>2)  # (a>b and a<b) or 3>2
```

Porównywać można dowolne obiekty (dla których zdefiniowane są odpowiednie operatory) np. listy.

```
print([1,3,4]==[3,6,7])
print([1,3,4]==[1,3,4])
print([1,3,4]<[3,6,7])
print([1,3,4]<[1,1,7]) #tzw. relacja słownikowa, sprawdza pierwszy
element, jezeli równe to dopiero nastepny
```

Operatory `in`, `not in` oraz `is`, `is not`.

```
print(1 in [1,2,3])
print('a' in 'informatyka')
print('a' not in 'informatyka')
```

```
True
True
False
```

```
a = 2
b = 3
c = 2
print(a is b)
print(a is c)
print(b is not c)
d = None  # None oznacza 'nic', pusty obiekt
print(d is None)
```

```
False
True
True
True
```

Instrukcje sterujące

Instrukcje: `if`, `if else`, `if elif else`.

Składnia języka Python nie posiada sprecyzowanych początków i końców dla funkcji czy bloków kodu oraz żadnych nawiasów służących do zaznaczania, gdzie funkcja lub blok się zaczyna, a gdzie kończy. Jedynym separatorem jest dwukropek `:` i `wcięcia kodu`. Wstawiając wcięcie zaczynamy blok, a kończymy go zmniejszając wielkość wcięcia do poprzedniej wartości. Nie ma żadnych nawiasów, klamer czy słów kluczowych. Oznacza to, że białe znaki (spacje itp.) mają znaczenie i ich stosowanie musi być konsekwentne.

```
a=2
b=3
if a<b:  # jeżeli warunek True to wykonaj instrukcje w bloku
(instrukcje z wcięciami (tab))
    print(a+b)

a=4
b=3
if a<b:  # gdy warunek prawdziwy
    print(a+b)
else:    # gdy warunek fałszywy
    print(a-b)

a=10
b=8
```

```
if a<b:  # gdy warunek prawdziwy
    print(a+b)
elif a<2*b: # kolejny warunek gdy poprzedni nie jest spełniony
    print(a*b)
else:    # gdy warunek prawdziwy
    print(a-b)

# warunkiem może być dowolne wyrażenie zwracające True lub False
a = 10
if a < 20 and a >5:
    print(a,a,a, end=' ')

# to samo możemy zapisać prościej
if 5 < a < 20:
    print(a,a,a)
```

```
10 10 10 10 10 10
```

Pętla `while`

```
i=0
while i<10:  #sprawdź warunek, jeżeli True wykonuj instrukcje w bloku
i znowu sprawdź warunek
    print(i, end=' ')
    i=i+1  # można krócej i+=1, Uwaga: nie ma operatora i++
```

Pętla `for`

Pętla `for` w języku Python działa jak pętla `foreach` w niektórych innych językach i składa się z dwóch słów kluczowych `for` oraz `in`.

```
for i in [1,2,6,7]:
    print(i, end=' ')
```

Iterować można nie tylko po listach, ale po wszystkich obiektach "iterowalnych" czyli posiadających metodę `__iter__`, np. łańcuch znaków, słownik itp.

```
for litera in 'informatyka':
    print(litera, end=' ')

dic = {'imie':'Jack', 'nazwisko': 'Sparrow'}
for key in dic:
    print(key, end=' ')
```

`range`, `enumerate`

Przydatnym obiektem przy używaniu pętli jest obiekt `range`, który w języku Python 3 (inaczej niż w Python 2) zwraca generator. Więcej o generatorach https://docs.python.org/3/glossary.html#term-generator.

```
range(100)  # generator przedziału liczb całkowitych od 0 do 99 (100 elementow)

range(5,80)  # generator przedziału liczb całkowitych od 5 do 79

range(5,80,2)  # generator przedziału liczb całkowitych od 5 do 79 z krokiem 2

for el in range(5,99,3):
    print(el, end=' ')
```

Równie przydatne jest słowo kluczowe `enumerate`, które dodaje do obiektu iterowanego pozycje elementów.

```
enumerate(['a','b','c'])  # z listy ['a','b','c'] otrzymujemu obiekt klasy enumerate postaci [(0,'a'), (1,'b'), (2,'c')]

abc = ['a','b','c']
for index, element in enumerate(abc):
    print(index, element)

# możliwe jest również coś takiego
abc = [[1,1],[2,1],[3,1]]
for x, y in abc:
    print(x,y)
```

Słowa kluczowe `break`, `continue`

```
# przerywanie pętli while
i=0
while True:
    print(i, end=' ')
    i+=1
    if i>10:
        break

# pomijanie iteracji w pętli while
i=0
while True:
    if i % 3 == 0:
        i+=1
        continue
    print(i, end=' ')
    i+=1
    if i>10:
        break

# przerwanie pętli for
for i in range(100):
    print(i*i, end=' ')
```

```
    if i*i>4000:
        break

# pomijanie iteracji w pętli for
for i in range(100):
    if 5 < i < 25:
        continue
    print(i*i, end=' ')
    if i*i>4000:
        break
```

Funkcje

Funkcje definiuje się słowem kluczowym `def`, a wywołuje poprzez jej nazwę `nazwa_funkcji()`.

```
def przywitanie():
    print('Hello')

przywitanie()
```

Funkcja może zwracać wartości, służy do tego słowo kluczowe `return`. Można zwracać dowolną liczbę wartości, dowolnego typu. Tak na prawdę funkcja zawsze coś zwraca, jezeli nie użyjemy w funkcji `return` to funkcja zwróci obiekt `None`.

```
def funkcja():
    return 2

a = funkcja()
print(a)

def funkcja2():
    return 1,2,3,4

a=funkcja2() # przypisanie zwracanych wartości do krotki
print(a)

x,y,z,t = funkcja2()
print(x,y,z,t)
```

Argumenty funkcji

Funkcje mogą przyjmować dowolną liczbę argumetów, argumentem może być dowolnego typu, również inne funkcje.

```
# funkcja zwracające normę euklidesa wektora (lista)
def norm(wektor):
    suma = 0
    for x in wektor:
```

```
        suma+=x**2
    norma = suma**(1/2.0)
    return norma

norm([1,2,3])

# fuckcja zwracająca p-tą normę, dla p=2 jest to norma euklidesa
def pnorm(wektor, p):
    suma = 0
    for x in wektor:
        suma+=x**p
    norma = suma**(1.0/p)
    return norma

pnorm([1,2,3], 2)
```

Argumenty funkcji zdefiniowane jak powyżej to argumety wymaganlne i przy wywoływaniu trzeba je podać i dokładnie tyle ile trzeba.

```
# na mało argumentów
pnorm([1,2,3])

# za dużo argumentów
pnorm([1,2,3],3,4)
```

Argumenty domyślne

Można jednak definiować tzw. argumenty domyślne, dla których w przypadku nie podania wartości przyjęte zostaną zdefiniowane wartości domyślne. Argumenty podawane są porzez ich nazwy więc można je podawać w dowolnej kolejności.

```
# p-ta norma
def pnorm(wektor, p=2, pp=True):
    suma = 0
    for x in wektor:
        suma+=x**p
    norma = suma**(1.0/p)
    if pp:
        res = 'Norma {0}-ta wektora {1} to {2}'.format(p, wektor,
norma)
        print(res)
    return norma

print(pnorm([1,2,3]))
print(pnorm([1,2,3],3))
print(pnorm([1,2,3],3,False))
print(pnorm([1,2,3],pp=False))
print(pnorm([1,2,3],p=3,pp=False))
```

Argumenty `*args, **kargs`

Ponieważ funkcje mogą przyjmować bardzo wiele argumentów, aby wyszystkich nie wypisywać, w definicji funkcji używa się często argumentów `*args` oraz `**kargs`. Pierszy znich odpowiada argumentom podawanym wprost (argumenty takie umieszczane są w krotce `args`), a drugi argumantom podawanym poprzez nazwy (argumenty takie umieszczane są w słowniku `kargs`).

```
def pnorm2(*args, **kargs):
    wektor = args[0]
    p = args[1]
    pp = kargs['pp']
    suma = 0
    for x in wektor:
        suma+=x**p
    norma = suma**(1.0/p)
    if pp:
        res = 'Norma {0}-ta wektora {1} to {2}'.format(p, wektor,
norma)
        print(res)
    return norma

pnorm2([1,2,3],3,pp=True)
```

Argumentami funkcji mogą być również inne funkcje, a także mogą odwoływać się do samej siebie (rekurencja)

```
def suma(a,b):
    return a+b

def roznica(a,b):
    return a-b

def dzialanie(x,y,f):
    return f(x,y)

dzialanie(2, 3, suma)
```

Funkcja `lambda`

Słowo kluczowe `lambda` pozwala na zdefiniowanie jednolinijkowej funkcji.

```
dzialanie(2, 6, lambda x, y: x+y-2*y)
```

Zadanie 3:

- Napisz program (lub funkcję), który wczytuje od użytkownika współczynniki a, b, c równania $a x^2+b x+c=0$ i wypisuje rozwiązania rzeczywiste tego równania lub informację o braku rozwiązań. Wartości współczynników możesz podawać wprost w

kodzie np. `a=1` itp. lub wykorzytsać metodę `input()`, która pozwala na wprowadzenie wartości z klawiatury w trakcie działania programu.

- Napisz drugą wersję tego programu, która pozwoli na otrzymywanie również rozwiązań zespolonych.

```
import math
def solution():
    a = float(input('Podaj współczynnik a: '))
    b = float(input('Podaj współczynnik b: '))
    c = float(input('Podaj współczynnik c: '))
    wsp = (b**2) - (4*a*c)
    delta = wsp**(1/2)
    if type(delta) == complex:
        return 2
    if delta > 0:
        solution1 = (-b - delta)/(2*a)
        solution2 = (-b + delta)/(2*a)
        return solution1, solution2
    elif delta == 0:
        solution1 = (-b)/(2*a)
        return solution1
    else:
        return None

x = solution()
print(x)


1.0
```

Odwzorowania listowe (List comprehension)

- czyli szybkie tworzenie list.

Klasyczne tworzenie listy za pomocą pętli `for`:

```
tab = []
for i in range(50):
    tab.append(i**2)
print(tab)
```

To samo możemy uzyskać w jednej linijce:

```
tab2 = [i**2 for i in range(50)]
print(tab2)
```

Obiektem po którym iterujemu może być dowolny obiekt iterowalny np. lista, zbiór, słownik itp. Możemy również generować w ten sposób listy wielowymiarowe:

```
tab2d = [[i,j] for i in [1,2,3,4,5] for j in [1,2,3,4,5]]
print(tab2d)
```

Powyższa konstrukcja może zawierać również instrukcje warunkowe:

```
#tylko liczby podzielne przez 13 z listy tab2
tab3 = [x for x in tab2 if x % 13 == 0]
print(tab3)
```

Uniwersalna forma dopuszczająca konstrukcję `if else`:

```
#liczby podzielne przez 13 zostaw bez zmian, a pozostałe zastąp zerem
tab4 = [x if x % 13 == 0 else 0 for x in tab2]
print(tab4)
```

Praca z plikami tekstowymi

Do pracy z plikami/folderami przydatny moze okazać sie moduł `os` (https://docs.python.org/3/library/os.html). Np. aby sprawdzić scieżkę do folderu, w którym aktualnie pracujemy należy wykonać polecenie:

```
import os
os.getcwd()
```

Otwieranie pliku tekstowego:

```
open('sciezka\nazwa_pliku','tryb')
```

Tryby otwierania pliku:

`w` - do zapisu (jeżli plik istnieje to go nadpiszemy)

`r` - do odczytu

`a` - tryb dopisywania

Do każdego trybu można dodać litere `b` co oznacza pracę w trybie binarnym.

Zapis do pliku:

```
# otwarcie pliku do zapisu
f = open('plik.txt','w')
# zapisujemy w pliku 5 lini tekstu
f.write('linia 1\n')
f.write('linia 2\n')
f.write('linia 3\n')
f.write('linia 4\n')
f.write('linia 5\n')
```

```
# zamykamy plik
f.close()
```

Czytanie z pliku:

```
# otwarcie pliku do odczytu
f = open('plik.txt','r')
# czytanie całego pliku
tekst = f.read()
# zamykamy plik
f.close()
print(tekst)

# otwarcie pliku do odczytu
f = open('plik.txt','r')
# czytanie linia po lini
print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())
print(f.readline())  # pusty string
print(f.readline())  # pusty string
# zamykamy plik
f.close()

# otwarcie pliku do odczytu
f = open('plik.txt','r')
# czytanie linia po linii pętla for
for line in f:
    print(line)
# zamykamy plik
f.close()

# otwarcie pliku do edycji (dopisywanie)
f = open('plik.txt','a')
f.write('linia 6\n')
f.write('linia 7\n')
f.close()

f = open('plik.txt','r')
tekst = f.read()
f.close()
print(tekst)
```

Aby nie musieć zawsze pamiętać o zamykaniu pliku można użyć słowa kluczowego `with`, wówczas po wyjściu z bloku `with` plik automatycznie zostanie zamknięty.

```
with open('plik.txt','r') as f:
    tekst = f.read()
print(tekst)
f.read()  #error, plik już jest zamknięty
```

Praca z plikami csv. Moduł `csv`

(https://docs.python.org/3/library/csv.html).

```
# import modulu csv
import csv

# zapis do pliku csv
with open('plik.csv','w', newline='') as f:
    csvwriter = csv.writer(f, delimiter=';')  #csv rozdzielany
średnikiem, domyslnie przecinek
    csvwriter.writerow(['row_1_col_1', 'row_1_col_2','row_1_col_3'])
    csvwriter.writerow(['row_2_col_1', 'row_2_col_2','row_2_col_3'])
    csvwriter.writerow(['row_3_col_1', 'row_3_col_2','row_3_col_3'])
# aby zobaczyć efekt należy podgladnąć plik.csv w folderze roboczym

# odczyt z pliku csv
with open('plik.csv','r', newline='') as f:
    csvreader = csv.reader(f, delimiter=';')
    for row in csvreader:
        for col in row:
            print(col)
```

Klasy czyli obiektowość w Python

(https://docs.python.org/3/tutorial/classes.html).

Klasy w Python definiuje się tak samo jak funkcję z tą różnicą, że używamy słowa kluczowego `class`.

```
class NazwaKlasy(klasy, po_ktorych, dziedziczymy):
```

Metody zaczynające i kończące się `__` to tzw. metody magiczne (specjalne)
(https://docs.python.org/3/reference/datamodel.html#special-method-names)

W poniższym przykładzie klasy Wektor została wykorzystana metoda `__init__` oraz `__add__` do nadpisania operatora dodawania `+`.

```
#Każda nowa klasa dzidziczy po klasie 'object'.
class Wektor(object):
    # metoda specjalna, inicjalizacja wykonywana przy tworzeniu
obiektu (coś jak konstruktor), nie musi jej być.
    def __init__(self, lista=[]):
```

```
        self.values = lista     #pola naszej klasy
        self.dim = len(lista)   #słowo kluczowe `self` odnosi się do
naszego obiektu (coś jak `this` w C++)

    # metody klasy
    def dl(self):  #dlugosc wektora
        s=0
        for x in self.values:
            s+=x**2
        return s**(1/2.0)

    # inne metody specjalne np. operatory
    # dodawanie wektorów
    def __add__(self, w):
        assert (self.dim == w.dim), 'Wymiary muszą się zgadzać'
        return Wektor([x+y for (x,y) in zip(self.values, w.values)])

# tworzenie obiektów klasy Wektor
w1 = Wektor([1,2,3])
w2 = Wektor([4,5,6])

print(w1.values)
print(w2.values)

print(w1.dim)
print(w2.dim)

w3 = w1 + w2

w3.values

w3.dl()

isinstance(w3, Wektor)

w4 = Wektor([1,2,3,4])
w5 = w3 + w4
```

Zadanie 4:

Utwórz klasę nadrzędną `Figura` z odpowiednimi metodami pozwalającymi na obliczenie i wyświetlenie pola powierzchni oraz obwodu figury. Klasa `Figura` powinna zawierać jedynie puste definicje tych metod, można wykorzystać słowo kluczowe `pass`.

Utwórz klasy `Trojkat`, `Prostokat`, `Kwadrat` dziedziczące po klasie `Figura`. Inicjalizacja obiektów danej klasy powinna odbywać się poprzez podanie długości boków. Klasy te powinny posiadać odpowiednie pola oraz nadpisywać metody z klasy `Figura`. Funkcja inicjalizująca powinna sprawdzać czy z podanych boków można zbudować daną figurę. To rzucania wyjątków można użyć słowa kluczowego `raise` lub jak w przykładzie wcześniej `assert`.

```
import math

class Figura:
    def countArea():
        pass
    def countPerimiter():
        pass
    def printAll():
        pass

class Trojkat(Figura):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
    def countArea(self):
        s = (self.a + self.b + self.c)/2
        return math.sqrt(s*(s-self.a)*(s-self.b)*(s-self.c))
    def countPerimiter(self):
        return self.a+self.b+self.c
    def printAll(self):
        print(self.countArea(), self.countPerimiter())

class Prostokat(Figura):
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def countArea(self):
        return (self.a)*(self.b)
    def countPerimiter(self):
        return 2*self.a + 2*self.b
    def printAll(self):
        print(self.countArea(), self.countPerimiter())

class Kwadrat(Figura):
    def __init__(self, a):
        self.a = a
    def countArea(self):
        return a**2
    def countPerimiter(self):
        return 4*a
    def printAll(self):
        print(self.countArea(), self.countPerimiter())

k1 = Kwadrat(1)
k1.printAll()
```

```
100 40
```

© Katedra Informatyki, Politechnika Krakowska