Laboratorium 9 (Programowanie 2 – 410-KS1-2PRO3)
Kierunek kognitywistyka i komunikacja
Programowanie obiektowe (OOP) w Pythonie - Klasy c.d.
Wyrażenia regularne

Klasy

1. **(dziedziczenie c.d.) Uzupełnij następujący kod:**

```
class Zwierze:
    def __init__(self, name, age):
    # przypisz odpowiednie wartości polom trzymającym imię i wiek
zwierzaka
    pass

def przedstaw_sie(self):
    # dodaj wypisywanie imienia i wieku
    pass


class Pies(Zwierze):
    # przedefiniujemy metodę przedstaw_sie
    # dodają dodatkowo komunikat/wiadomość: 'Jestem psem!'
    def przedstaw_sie(self):
        pass


class Kot(Zwierze):
    # niech kot przedstawia się inaczej:
    # nie korzysta wcale z metody nadklasy,
    # ale wypisuje swój komunikat, innej treści, chociaż zawierający
wszystkie informacje
    def przedstaw_sie(self):
        pass
```

2. **Stwórz kilka innych podklas klasy *Zwierze* np. Mysz, Chomik, Szczur itd. i niech każda z nich raz przedstawia się korzystając z metod nadklasy, dodaje dodatkową informację do metody z nadklasy lub też nie korzysta z metody nadklasy. Następnie utwórz po jednej instancji każdej podklasy.**
3. **(sprawdzanie, czy dana klasa jest podklasą) Dla stworzonych klas użyj funkcji *issubclass(klasa1, klasa2)* do sprawdzenia czy *klasa1* jest podklasą klasy *klas2*.**
4. **(sprawdzanie, czy dana instancja należy do klasy) Dla stworzonych klas i instancji użyj funkcji *isinstance(instancja, klasa)* do sprawdzenia czy instancja *instancja* jest elementem klasy *klasa*.**
5. **(Praca domowa) Stwórz klasę *DomoweZoo* (klasa ma zawierać listę zwierząt żyjących w danym domu). Konstruktor klasy musi mieć postać:**

```
def __init__(self):
    self.domownicy = []
```

W klasie mają być 3 metody: *dodaj, usun, wypisz_zwierzaki.* O ile implementacja dwóch pierwszych metod jest oczywista (metoda dodaje lub usuwa zwierzaka z listy), to metoda ostatnia ma informować ile zwierząt jest w domu i wszystkie zwierzęta mają się przedstawiać. W przypadku, gdy nie ma zwierząt w domu metoda powinna wyświetlić komunikat: *W domu nie ma ani jednego zwierzęcia.* Korzystając z wcześniej utworzonych instancji przetestuj stworzony kod.

6. **(metody abstrakcyjne z pakietu abc) Przeanalizuj następujący kod**

```
from abc import ABC, abstractmethod
class bird(ABC):
    def __init__(self, species, speed):
        self.species = species
        self.speed = speed

    def fly(self):
        print(f'Tu {self.species}. Startuję, i zaraz osiągnę prędkość
{self.speed}.')

    @abstractmethod
    def make_a_sound(self):
        pass

class eagle(bird):
    def __init__(self, speed):
        super().__init__('orzeł', speed)

    def hunt(self):
        print(f'Tu {self.species}. Rozpocząłem polowanie.')

    def make_a_sound(self):
        print('wrrrr.')

class penguin(bird):
    def __init__(self, speed):
        super().__init__('pingwin', speed)

    def slide(self):
        print(f'Tu {self.species}. Rozpocząłem ślizg.')

    def fly(self):
        print(f'Tu {self.species}. Przykro mi, ale nie latam.')

    def make_a_sound(self):
        print('kwiiiii.')
```

7. **Używając metod abstrakcyjnych zmodyfikuj definicję klas *Zwierze*, *Pies* i *Kot*. Metoda *przedstaw_sie* ma być metodą abstrakcyjną!**
8. **Korzystając z metod abstrakcyjnych zdefiniuj klasę *FiguraGeometryczna*, w której konstruktor będzie miał jeden atrybut *nazwa*, będą dwie abstrakcyjne metody *pole* i *obwód* oraz metoda instancji do przedstawiania się. Następnie zdefiniuj trzy klasy potomne *Kolo* (z 3 atrybutami: współrzędnymi środka i promieniem), *Trojkat* (z 3 atrybutami: bokami dokładniej długościami boków) oraz *Prostokat* (z 2 atrybutami: dwoma bokami). Dla klasy *Prostokat* zdefiniuj klasę potomną *Kwadrat*. Przetestuj swój kod.**
9. **(wielokrotne dziedziczenie). Przeanalizuj następujący kod**

```
class A:
    def hello(self): print('To ja klasa A')

class B:
    def hello(self): print('To ja klasa B')

class C1(B,A):
    pass
```

```
class C2(A, B):
    pass

c1 = C1()
c1.hello()
c2 = C2()
c2.hello()
```

Co zaobserwowałaś/zaobserwowałeś?

10. **Zaimplementuj klasy: *Pojazd* i *Aramata*. Klasa *Pojazd* ma zawierać atrybuty *predkosc*, *rodzaj_napedu* oraz metody *jedz* i *zatrzymaj_sie*. Natomiast klasa *Armata* ma zawierać atrybuty *kaliber* i *szybkostrzelnosc* oraz metodę *strzelam*. W oparciu o te klasy utwórz klasę potomną *Czolg*. Następnie utwórz instancje *Rudy* oraz sprawdź, czy Twój czołg jeździ i strzela.**
11. **(dekoratory – *property* – ustawianie i pobieranie atrybutów klasy) Przeanalizuj następujący kod (jest on częścią rozbudowanego rozwiązania polecenia 8):**

```
class Trojkat(FiguraGeometryczna):
    def __init__(self, a, b, c):
        self.__a = a
        self.__b = b
        self.__c = c
        super().__init__('trójkąt')

    @property
    def boki(self):
        return self.__a, self.__b, self.__c

    @boki.setter
    def boki(self, value):
        if len(value) != 3:
            raise TypeError('Dane muszą być listą lub krotką o
długości 3!')
        else:
            self.__a = value[0]
            self.__b = value[1]
            self.  c = value[2]

    def obwod(self):
        return self.__a + self.__b + self.__c

    def pole(self):
        p = self.obwod()/2
        return math.sqrt(p*(p-self.__a)*(p-self.__b)*(p-self.__c))
```

Jeżeli masz wykonane polecenie 8, to spróbuj ustawić długości boków samodzielnie oraz korzystając z metody *boki*.

12. **Pakiet *re* (polecenia)**

match()	***Dopasowanie wzorca na początku łańcucha.***	***Match***
search()	***Dopasowanie w dowolnym miejscu łańcucha.***	***Match***
findall()	***Dopasowanie wszystkich, niezachodzących dopasowań.***	***lista***
finditer()	***Dopasowanie wszystkich, niezachodzących dopasowań.***	***iterator***
split()	***Pocięcie łańcucha w miejscach dopasowań.***	***lista***
sub()	***Zamiana dopasowania wzorca na inny łańcuch.***	***Łańcuch znaków***

13. **Zapoznaj się z następującymi symbolami stosowanymi w tworzeniu wyrażeń regularnych**

+	***Jedno lub więcej dopasowań.***
*****	***Zero lub więcej dopasowań.***

?	*Zero lub jedno dopasowanie.*
{n}	*dokładnie n dopasowań.*
{n,m}	*miedzy n a m dopasowań.*
{n,}	*n lub więcej, dopasowań.*
{,m}	*m lub mniej dopasowań, z 0 włącznie.*
^	*początek napisu*
$	*koniec napisu*
..|..	*alternatywne wzorce*
[..]	*alternatywne wzorce*
[^..]	

wzorzec(?=X)	*po wzorcu powinien występować X*
wzorzec(?!X)	*po wzorcu nie powinien występować X*
(?<=X)wyrażenie	*X poprzedza wyrażenie.*
(?<!X)wyrażenie	*X nie poprzedza wyrażenia.*

Znaki specjalne i zakresy znaków

\s	*biały znak*
\S	*nie-biały znak*
\d	*cyfra*
\D	*nie-cyfra*
\w	*znaki alfanumeryczne (litery i cyfry) oraz _*
\W	*znaki nie-alfanumeryczne i nie _*
\b	*początek lub koniec „słowa" - formalnie, \b oznacza granicę między \w i \W*
\B	*nie początek lub koniec „słowa"*
[a-z]	*małe litery*
[A-Z]	*wielkie litery*
[0-9]	*cyfry*

Uwagi:

- **Jeśli znaki +, * czy ? mają w wyrażeniu oznaczać konkretnie te znaki, to zależy je poprzedzić znakiem ucieczki **
- **Dopasowania z użyciem znaków * oraz + jest zachłanne, co oznacza, że dopasowuje tak wiele znaków, jak to tylko możliwe. Dodanie znaku ?, zmienia to zachowanie, na leniwe - dopasowane jest tak mało znaków, jak to możliwe.**
- **Następuje zmiana znaczenie znaku ? w zależności od kontekstu.**

14. Zapoznaj się z stronami:

- **https://ggoralski.gitlab.io/python-wprowadzenie/czesc_ii/2_01-regex/**
- **https://www.programiz.com/python-programming/regex**
-

15.