Programowanie Obiektowe
Instrukcja do laboratorium 2

1 Zasady oceniania

Zadanie	Ocena
Punkt i Prosta (3.1)	3,0
Prostokąt (3.2)	3,5
Notatka i Notatnik (3.3)	4,0
Pracownik (3.4)	4,5
Player (3.5)	5,0

Warunkiem uzyskania oceny za zadanie n jest wykonanie poprzednich zadań.

W przypadku nie oddania zadania w terminie (tydzień po zajęciach), uzyskana ocena jest zmniejszana o 0,5 za każdy tydzień opóźnienia.

Plik (wkleić cały kod do jednego pliku, podpisać poszczególne zadania za pomocą komentarzy) ze zrobionymi zadaniami proszę przesłać na platformie Moodle. Plik musi mieć nazwe `numeralbumu_lab1.py`. **Plik musi być plikiem tekstowym z rozszerzeniem `.py`**.

UWAGA: Termin oddania zadania jest ustawiony w systemie moodle. W przypadku nie oddania zadania w terminie, uzyskana ocena będzie zmniejszana o 0,5 za każdy zaczęty tydzień opóźnienia. Zadania oddawane później niż miesiąc po terminie ustawionym na moodle są oddawane i rozliczane w trybie indywidualnym na zajęciach lub po umówieniu się z prowadzącym.

UWAGA: W przypadku wysłania zadania w formie niezgodnej z opisem w instrukcji prowadzący zastrzega prawo do wystawienia oceny negatywnej za taką pracę. Przykład: wysłanie `.zip` lub `.pdf` tam, gdzie był wymagany plik tekstowy z rozszerzeniem `.py`.

2 Materiał pomocniczy

2.1 Obiekt

Obiekt jest pewną strukturą, która łączy dane i funkcje przetwarzające te dane. Klasy są szablonami, z których są tworzone obiekty. Przykładowo, zdefiniujemy następującą klasę:

```
1  class Klasa:
2      # Zmienna klasy, która będzie dostępna bezpośrednio
3      # z klasy oraz z każdego obiektu tej klasy
4      zmienna_klasy = 'to jest zmienna klasy Klasa'
5
6      # Konstruktor obiektu (wywoływany przy tworzeniu obiektu
7      def __init__(self):
8          # Zmienna obiektu
9          self.zmienna = 'to jest zmienna obiektu'
10
11     def funkcja(self):
12         # Metoda wypisująca pewien ciąg
13         print(f'Wartość zmiennej to "{self.zmienna}"')
```

Kolejny listing pokazuje jak możemy stworzyć obiekt tej klasy, oraz w jaki sposób uzyskujemy i zmieniamy wartości atrybutów które ten obiekt posiada.

```
>>> ob = Klasa()
>>> ob.zmienna_klasy
'to jest zmienna klasy Klasa'
>>> ob.zmienna
'to jest zmienna obiektu'
>>> ob.funkcja()
Wartość zmiennej to "to jest zmienna obiektu"
>>> ob.zmienna = 'nowa wartosc'
>>> ob.funkcja()
Wartość zmiennej to "nowa wartosc"
```

2.2 Co to jest `self`?

`self` jest odniesieniem do obiektu klasy. Każda metoda klasy, przeprowadzająca pewne działania na obiekcie posiada `self` jako pierwszy argument(przykład: w konstruktorze `__init__` przypisujemy do `self` pewne zmienne). Przykładowo dodawanie możemy zrealizować w sposób następujący:

```
class ExampleAdd:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def add_two_variable(self):
        return self.a + self.b


def add_two_variable(a, b):
    return a + b

>>> obj = ExampleAdd(2,4)
>>> obj.add_two_variable()
6
>>> add_two_variable(2,4)
6
```

W przypadku użycia funkcji *add_two_variable* zawsze jesteśmy zobligowani podać zmienne, natomiast w przypadku metody *add_two_variable* pochodzącej z klasy **ExampleAdd** wystarczy podać do konstruktora raz zmienne i ją wywoływać. Bez użycia `self` nie jesteśmy w stanie ustalić modyfikatorów dostępu pól oraz metod klasy. Słowo kluczowe `self` jest słowem ogólnie przyjętym, dlatego też można (co jest niewskazane) użyć dowolnego słowa.

2.3 Modyfikatory dostępu

Modyfikatory dostępu jak sama nazwa wskazuje określają prawa do dostępu obiektu. Większość języków zorientowanych obiektowo (takich jak Java, C++) używa słów kluczowych tj. **public**, **private** oraz **protected**. W przypadku języka Python używa się podkreśleń przed nazwą obiektu (np. pola lub metody). Brak podkreślenia oznacza, że pole (lub metoda) jest publiczne, czyli jest ono dostępne dla każdej operacji wykonywanej w klasie lub poza nią. W przypadku jednego podkreślenia pole (lub metoda) uzyskuje atrybut chroniony. Oznacza to, że klasa udostępnia swoje pole (lub metodę) dla klas dziedziczących po niej oraz dla operacji wykonywanych w klasie. Dwa podkreślenia definiują pole (lub metodę) prywatne. Oznacza to, że można jedynie zmodyfikować dany atrybut jedynie w obrębie klasy. W przypadku próby modyfikacji takiego atrybutu z zewnątrz powinien pojawić się błąd. Poniżej został zdefiniowany przykład pokazujący zastosowanie modyfikatorów dostępu w klasie.

```
class Example:
    def __init__(self, a, b, c):
```

```
        self.pole_publiczne = a
        self._pole_chronione = b
        self.__pole_prywatne = b

    def metoda_publiczna(self):
        return self.pole_publiczne

    def _metoda_chroniona(self):
        return self._pole_chronione

    def __metoda_prywatna(self):
        return self.__pole_prywatne
```

Za pomocą listingu 2.3 przedstawiono przykład pokazujący jak modyfikatory dostępu wpływają na dostępność atrybutów klasy. W tym przykładzie użyto klasy z poprzedniego listingu.

Przykład:

```
>>> obj = Example(1, 2, 3)
>>> obj.pole_publiczne
1
>>> obj._pole_chronione
2
>>> obj.__pole_prywatne
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Example' object has no attribute '__pole_prywatne'
```

2.4 Metody magiczne

Metody magiczne to takie metody klasy, które są zdolne do implementowania pewnych operacji. Służą one do przeciążania operatorów. Lista metod magicznych wraz z ich opisem umieszczona jest na stronie z dokumentacją oraz stronie z wyróżnionymi operatorami.

Przykładowe metody magiczne:

- `__add__(self, b)` - arytmetyczna operacja dodawania (przeciążenie operatora '+')
- `__mul__(self, b)` - arytmetyczna operacja mnożenia (przeciążenie operatora '*')
- `__eq__(self, b)` - operacja porównania obiektu (przeciążenie operatora '==')
- `__len__(self)` - długość obiektu
- `__str__(self)` - ciąg znaków reprezentujących obiekt
- `__repr__(self)` - reprezentacja obiektu

Przykład różnicy między `__repr__(self, b)`, a `__str__(self, b)`:

```
>>> import numpy as np
>>> vec = np.array([1,2,3])
>>> repr(vec)
'array([1, 2, 3])'
>>> str(vec)
'[1 2 3]'
```

2.5 Przeciążenia operatorów

Przeciążenie operatorów jest operacją pozwalającą na przechwytywanie obiektów danej instancji w celu wykonania na nich działania. W przypadku braku przeciążenia operatora wystąpi błąd.

Przykład:

```
>>> obj = Example(1, 2, 3)
>>> obj_other = Example(4, 5, 6)
>>> obj + obj_other
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Example' and 'Example'
```

W celu dodania dwóch obiektów tej samej klasy powinniśmy przeciążyć operator dodawania. Poniżej zamieszczono klasę posiadającą jedno pole w której przeciążono operator dodawania:

```
class ExampleAdding:
    def __init__(self, a):
        self.pole_publiczne = a

    def __add__(self, other):
        return ExampleAdding(self.pole_publiczne + other.pole_publiczne)
```

Za pomocą listingu 2.5 przedstawiono przykładowe dodawanie dwóch obiektów w których przeciążono operator dodawania:

```
>>> obj = ExampleAdding(4)
>>> obj_other = ExampleAdding(8)
>>> obj_created = obj + obj_other
>>> obj_created.pole_publiczne
12
```

2.6 Dekoratory

Dekorator jest obiektem do którego można się odnieść jak do funkcji. Inaczej mówiąc dekorator dodaje nową funkcjonalność do funkcji, która jest przekazywana jako argument. Definiujemy go przy pomocy znaku @.

Przykład:

```
>>> def example_decorator(obj):
...     return obj
>>> @example_decorator
... def function_with_decorator():
...     print("Func with decorator")
```

Dekorator `@property` jest dekoratorem umożliwiającym dostanie się do pola, które posiada modyfikator dostępu chroniony lub prywatny. Dostęp do takiego pola będzie potrzebny gdy będziemy chcieli się do niego dostać w jakiś sposób.

Przykład:

```
class Example:
    def __init__(self, a):
        self.__a = a

    @property
    def a(self):
        print('Pobieramy wartość a')
        return self.__a
```

Poniżej zaprezentowano przykład w którym pole a o modyfikatorze dostępu prywatny zostało wyświetlone dzięki dekoratorowi `@property`. W przypadku pokazania wartości atrybutu nie występuje błąd. Jednakże przy próbie zmiany wartości atrybutu wyskakuje błąd, przez wzgląd na zadany modyfikator dostępu.

```
>>> ob = Example(3)
>>> ob.a
Pobieramy wartość a
3
>>> ob.a = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

W celu dostania się do atrybutu chronionego lub prywatnego można powołać się na następujące metody dekoratora `@property`:

- **Getter** - pozwala na uzyskanie dostępu dla danego pola (`@property`)
- **Seter** - pozwala na zmianę wartości danego pola (`@zmienna.setter`)
- **Deleter** - pozwala na usunięcie danego pola (`@zmienna.deleter`)

Przykład zdefiniowania setera oraz gettera dla instancji `Example` pola a z modyfikatorem dostępu prywatny.

```
class Example:
    def __init__(self, a):
        self.__a = a

    @property
    def a(self):
        print('Pobieramy wartość a')
        return self.__a

    @a.setter
    def a(self, new_a):
        print('Ustawiamy wartość a')
        self.__a = new_a
```

Poniżej został przedstawiony przykład użycia gettera oraz setera dekoratora `@property`. Jak widać dzięki użytym metodom dekoratora możemy dostać się do zabezpieczonego pola a w celu ustawienia jego nowej wartości oraz wyświetlenia jego wartości.

```
>>> ob = Example(3)
>>> ob.a
Pobieramy wartość a
3
>>> ob.a = 5
Ustawiamy wartość a
>>> ob.a
Pobieramy wartość a
5
```

2.7 Klasa Vector

Do przedstawienia ww. funkcjonalności została zdefiniowana klasa wektor posiadająca dwa pola prywatne a oraz b. Posiada ona przeciążenie operatora dodawania oraz porównania. Dodatkowo zostały zaimplementowane przeciążenia metod takich jak `str()`, `repr()` oraz `abs()`, które odpowiadają odpowiednio za wyświetlenie ciągu znaków reprezentujących obiekt, reprezentację obiektu oraz wartość bezwzględną obiektu (w tym przypadku długość wektora). Ponadto uwzględniono hermetyzację pól a oraz b, gdzie pozwolono przy pomocy dekoratora `@property` jedynie na wyświetlenie wartości obu pól.

```python
class Vector:
    def __init__(self, a, b):
        self.__a = a
        self.__b = b

    @property
    def a(self):
        return self.__a

    @property
    def b(self):
        return self.__b

    def __str__(self):
        return f'Vector: ({self.a}, {self.b})'

    def __repr__(self):
        return f'Vector({self.a}, {self.b})'

    def __add__(self, oth):
        return Vector(self.a + oth.a, self.b + oth.b)

    def __abs__(self):
        return (self.a**2 + self.b**2) ** 0.5

    def __eq__(self, oth):
        return self.a == oth.a and self.b == oth.b
```

Poniżej znajduje się przykład wykorzystania wektora (stworzenie, dodawanie itd.). Wszystkie operacje zostały opisane w listingu za pomocą komentarzy.

```
>>> v1 = Vector(1, 2)
>>> v2 = Vector(3, 4)
>>> print(v1.a, v1.b) # Możemy pobrać tylko do odczytu wartości atrybutów
1 2
>>> print(v1) # Możemy teraz wypisać nasz wektor
Vector: (1, 2)
>>> v1 # Również po prostu w konsoli (wtedy działa __repr__)
Vector(1, 2)
>>> v1 + v2 # Działa dodawanie obiektów naszej klasy
Vector(4, 6)
>>> abs(v1) # Działa też obliczenie wartości bezwzględnej wektora
2.23606797749979
>>> v1 == v2 # Możemy też porównywać wektory
False
```

3 Zadania do wykonania

3.1 Punkt i Prosta

Utwórz dwie klasy: jedna klasa ma reprezentować punkt na płaszczyźnie i posiadać współrzędne x,y tego punktu jak atrybuty i argumenty konstruktora. Druga klasa ma reprezentować prostą i posiada takie atrybuty jak a i b, definiujące przebieg prostej o równaniu $y = ax + b$.

Dodaj do klasy Punkt metodę, która przyjmuje jako argument obiekt klasy Prosta i zwraca True albo False w zależności od tego leży nasz punkt na tej prostej czy nie.

Dodaj do klasy Prosta metodę która znajdzie miejsce w którym ta prosta przecina oś X albo wypisze informacje że taki punkt nie istnieje.

Przykład:

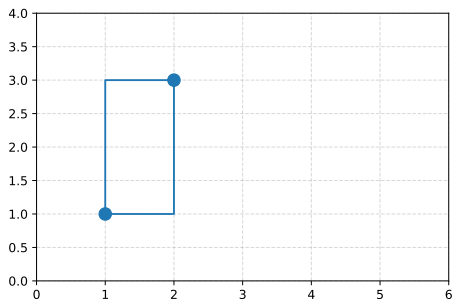
```
>>> p = Punkt(3, 6)
>>> pr = Prosta(2, 0)
>>> p.nalezy_do(pr)
True
>>> pr.miejsce_zerowe()
0
```

3.2 Prostokąt

Utwórz klasę reprezentującą prostokąt. Konstruktor tej klasy musi przyjmować dwa obiekty klasy Punkt jako argumenty. Dwa punkty definiujące prostokąt muszą leżeć na przekątnej. Należy zaimplementować metodę, która obliczy boki prostokąta na podstawie punktów podanych do konstruktora. Klasa ma posiadać metody do obliczenia pola i obwodu prostokąta. Klasa ma także posiadać metodę która narysuje zdefiniowany prostokąt za pomocą biblioteki `matplotlib` i zaznaczy punkty którymi został zdefiniowany (patrz Rysunek 1).

Przykład:

```
>>> p1 = Punkt(1, 1)
>>> p2 = Punkt(2, 3)
>>> prost = Prostokat(p1, p2)
>>> prost.pole()
2
>>> prost.obwod()
6
>>> prost.rysuj()
```



Rysunek 1: Przykładowy wynik działania metody `rusyj()`

3.3 Notatka i Notatnik

Utworzyć klasy Notatka (Note) i Notatnik (Notebook). Klas notatki przechowuje autora, treść i czas stworzenia (autor i treść są podawane jako argumenty konstruktora, a czas jest pobierany i zapisywany przy tworzeniu obiektu). Konstruktor klasy Notatnik nie przyjmuje żadnych argumentów, lecz tworzy pustą listę do której będą dodawane obiekty klasy Notatka. Klasa Notatnika musi posiadać implementacje metod, pozwalających: dodać nową notatkę, dodać istniejącą notatkę, sprawdzić ile jest dodanych notatek, wyświetlić wszystkie dodane notatki. Dodatkowo musi być obsłużona sytuacja kiedy notatnik jest pusty.

Przykład:

```
>>> nb = Notebook()
>>> nb.dodaj_nowa("Bartek", "Dokończyć instrukcje")
>>> nb.wyswietl_wszystko()
```

```
Masz takie notatki:
1. Bartek: "Dokończyć instrukcje" o godzinie 22:18
>>> n1 = Note("Andrii", "Sprawdzić instrukcje ")
>>> nb.dodaj(n1)
>>> nb.wyswietl_wszystko()
Masz takie notatki:
1. Bartek: "Dokończyć instrukcje" o godzinie 22:18
2. Andrii: "Sprawdzić instrukcje" o godzinie 22:20
```

Podpowiedź: do reprezentacji czasu można użyć modułu `datetime`.
Dokumentacja modułu `datetime`: https://docs.python.org/3/library/datetime.html
Przykład:

```
>>> import datetime
>>> t = datetime.datetime.now()
>>> t
datetime.datetime(2021, 4, 8, 22, 39, 46, 274407)
>>> t.hour
22
>>> t.minute
27
```

3.4 Pracownik

Stwórz klasę "Pracownik"w Pythonie, która będzie posiadać atrybuty "imie", "nazwisko"oraz "stanowisko". Klasa powinna zawierać publiczną metodę "przedstaw_sie()", która wyświetli imię i nazwisko pracownika oraz jego stanowisko.

Dodatkowo, klasa powinna posiadać atrybut chroniony "_id_pracownika", który będzie nadawany automatycznie przy tworzeniu nowego obiektu i będzie unikalny dla każdego pracownika.

Podpowiedź: użyć `id(self)`: https://www.programiz.com/python-programming/methods/built-in/id

Ostatecznie, klasa powinna mieć prywatne pole "__pensja", które będzie przechowywać informację o wynagrodzeniu pracownika oraz prywatną metodę "__zmien_pensje(self, nowa_pensja)", która będzie umożliwiała zmianę wynagrodzenia tylko z poziomu klasy.

Przykład:

```
>>> pracownik1 = Pracownik("Jan", "Kowalski", "Inżynier")
>>> pracownik1.przedstaw_sie()
Cześć, nazywam się Jan Kowalski i pracuję na stanowisku Inżynier.
>>> pracownik2 = Pracownik("Anna", "Nowak", "Specjalista ds. marketingu")
>>> pracownik2.przedstaw_sie()
Cześć, nazywam się Anna Nowak i pracuję na stanowisku Specjalista ds. marketingu.
>>> print(f"ID pracownika 1: {pracownik1._id_pracownika}")
ID pracownika 1: 2532711240704
>>> print(f"ID pracownika 2: {pracownik2._id_pracownika}")
ID pracownika 2: 2532712894912
>>> pracownik1.podwyzka(1000)
>>> print(f"Wynagrodzenie pracownika 1: {pracownik1.get_pensja()}")
Wynagrodzenie pracownika 1: 1000
```

3.5 Player

Stwórz klasę "Player"w Pythonie, która będzie przechowywać informacje o graczach w grze. Klasa powinna mieć pola publiczne "nick"oraz chronione "_health"oraz "_score". Klasa powinna zawierać metody publiczne "attack(enemy)"oraz "heal()", które będą umożliwiać odpowiednio atakowanie przeciwnika oraz leczenie siebie. Metoda ataku powinna zmniejszać zdrowie przeciwnika, a metoda leczenia zwiększać zdrowie gracza.

Pole "_health"powinno być prywatne, aby uniemożliwić bezpośrednią zmianę wartości z zewnątrz klasy. Do odczytu wartości pola "_health"powinna zostać utworzona metoda prywatna o nazwie "_get_health()", która

będzie zwracać wartość pola "_health". Do zapisu wartości pola "_health"powinna zostać utworzona metoda prywatna o nazwie "_set_health()", która będzie zapisywać wartość pola "_health".

Aby ułatwić odczyt wartości pola "_health", powinna zostać utworzona właściwość o nazwie "health", dekoratorem @property oraz @health.setter. Właściwość ta powinna korzystać z metod prywatnych "_get_health()"oraz "_set_health()".

Dodatkowo, klasa "Player"powinna mieć pole publiczne "level", które będzie przechowywać poziom gracza. Pole to powinno zostać utworzone za pomocą dekoratora @property, w taki sposób, że poziom gracza będzie wyznaczany na podstawie wartości pola "_score". Np. jeśli "_score"będzie większe od 100, to poziom gracza będzie równy 2, a jeśli "_score"będzie większe od 200, to poziom gracza będzie równy 3, itd.

Przykład:

```
# Tworzenie obiektów klasy Player
>>> player1 = Player("John")
>>> player2 = Player("Mike")

# Odczytanie początkowych wartości pól obiektów
>>> print(f"{player1.nick}: health={player1.health}, level={player1.level}")
John: health=100, level=1
>>> print(f"{player2.nick}: health={player2.health}, level={player2.level}")
Mike: health=100, level=1

# Atakowanie przeciwnika
>>> player1.attack(player2)
>>> print(f"{player1.nick} attacked {player2.nick}")
John attacked Mike

# Odczytanie zmienionych wartości pól obiektów
>>> print(f"{player1.nick}: health={player1.health}, level={player1.level}")
John: health=100, level=1
>>> print(f"{player2.nick}: health={player2.health}, level={player2.level}")
Mike: health=90, level=1

# Uzdrawianie gracza
>>> player2.heal()
>>> print(f"{player2.nick} healed himself")
Mike healed himself

# Odczytanie zmienionych wartości pól obiektów
>>> print(f"{player1.nick}: health={player1.health}, level={player1.level}")
John: health=100, level=1
>>> print(f"{player2.nick}: health={player2.health}, level={player2.level}")
Mike: health=100, level=2

# Zmiana wartości pola health przy użyciu właściwości health
>>> player1.health = 80
>>> print(f"{player1.nick} health value changed to {player1.health}")
John health value changed to 80

# Odczytanie zmienionych wartości pól obiektów
>>> print(f"{player1.nick}: health={player1.health}, level={player1.level}")
John: health=80, level=1
>>> print(f"{player2.nick}: health={player2.health}, level={player2.level}")
Mike: health=100, level=2
```

Alternatywa:

Alternatywnie można zaimplementować metody do wyświetlania wartości obiektu (metoda magiczna str) oraz wyświetlanie komunikatów przy zmianie wartości atrybutu health, wywołaniu metody heal() oraz wywołaniu metody

attack().

Przykład:

```
# Tworzenie obiektów klasy Player
>>> player1 = Player("John")
>>> player2 = Player("Mike")

# Odczytanie początkowych wartości pól obiektów
>>> print(player1)
John: health=100, level=1
>>> print(player2)
Mike: health=100, level=1

# Atakowanie przeciwnika
>>> player1.attack(player2)
John attacked Mike

# Odczytanie zmienionych wartości pól obiektów
>>> print(player1)
John: health=100, level=1
>>> print(player2)
Mike: health=90, level=1

# Uzdrawianie gracza
>>> player2.heal()
Mike healed himself

# Odczytanie zmienionych wartości pól obiektów
>>> print(player1)
John: health=100, level=1
>>> print(player2)
Mike: health=100, level=2

# Zmiana wartości pola health przy użyciu właściwości health
>>> player1.health = 80
John health value changed to 80

# Odczytanie zmienionych wartości pól obiektów
>>> print(player1)
John: health=80, level=1
>>> print(player2)
Mike: health=100, level=2
```