Moduł 8 - Wprowadzenie do programowania obiektowego.

1. Klasa i obiekt.

Klasa jest definicją pewnej struktury danych i zachowania, które chcemy zamodelować w postaci zbioru cech (właściwości) oraz metod (zachowanie). Ta definicja podobnie jak definicja funkcji nie powoduje wykonania żadnego kodu związanego z tą definicją, dzieje się to dopiero w momencie stworzenia obiektu klasy. Obiekt, nazywany również instancją klasy, jest tworzony w momencie wywołania konstruktora danej klasy.

Podobnie jak w wielu innych językach programowania zorientowanych obiektowo tak i w Pythonie istnieje klasa bazowa dla wszystkich obiektów - jest to klasa o nazwie `object`.

Listing 1

```
# definicja pustej klasy
class Player:
    pass

p1 = Player()


print(type(p1))
# czy klasa Player jest klasą potomną klasy object
print(issubclass(Player, (object)))
# jest też jednocześnie instancją klasy Player oraz object
print(isinstance(p1, (object, Player)))
```

Deklarowanie właściwości klasy może odbywać się na kilka sposobów. Najbardziej popularnym i uznawanym za dobrą praktykę jest deklaracja w konstruktorze klasy (patrz listing 4). Konstruktor to nic innego jak specjalna metoda o nazwie `__init__`, która jest wywoływana w momencie tworzenie nowej instancji (kopii) obiektu. W Pythonie możliwe jest również deklarowanie właściwości w metodzie (listing 3) lub nawet poza definicją klasy (listing 2), co uznawane jest nierzadko za złą praktykę.

Listing 2

```
class Player:
    pass

p1 = Player()
p1.nickname = 'bonzo'
print(p1.nickname)

p2 = Player()
print(p2.nickname) # błąd !
```

Listing 3

```
# deklaracja właściwości w metodzie - takie podejście ma niewątpliwą wadę, gdyż
taka właściwość
# może nie zostać zainicjalizowana (jej wartość) przed jej wywołaniem.
class Player:

    def set_nickname(self, nickname):
        self.nickname = nickname

p1 = Player()
# print(p1.nickname) # błąd !
p1.set_nickname('alias')
print(p1.nickname)
```

Listing 4

```
class Player:

    def __init__(self, nickname):
        self.nickname = nickname

# lub też inicjalizacja właściwości wartością domyślną w konstruktorze
class Player:

    def __init__(self):
        self.nickname = ''

    def set_nickname(self, nickname):
        self.nickname = nickname
```

2. Dziedziczenie.

Dziedziczenie jest kolejnym paradygmatem programowania obiektowego, który pozwala na określenie pewnej hierarchii obiektów (rodzic <- potomek lub inaczej klasa bazowa <- klasa pochodna). Ma to wiele zalet, a jedną z głównych jest to, że możemy tworzyć nowe klasy, bardziej szczegółowe na podsatwie ogólnych typów, które dla wszystkich typów potomnych udostępniają zbiór podobnych cech (właściwości) i zachowań (metody). Zmniejsza to ilość kodu niezbędnego do napisania, ale też pozwala na wykorzystanie innego paradygmatu - polimorfizmu (o tym w kolejnym labie). Mechanizm dziedziczenia wykorzystywaliśmy już wielokrotnie przy tworzeniu dowolnej instancji obiektu na innej klasie niż klasa `object`. Poniżej przykład wykorzystania dziedziczenia.

Listing 5

```
class Osoba:

    def __init__(self, imie, nazwisko):
```

```
        self.imie = imie
        self.nazwisko = nazwisko

    def przedstaw_sie(self):
        return "{} {}".format(self.imie, self.nazwisko)


class Pracownik(Osoba):

    def __init__(self, imie, nazwisko, pensja):
        Osoba.__init__(self, imie, nazwisko)
        # lub
        # super().__init__(imie, nazwisko) # super() oznacza klasę bazową, ale
przy
        # wielodziedziczeniu nie będzie możliwe jej wykorzystanie w ten sposób
        self.pensja = pensja

    def przedstaw_sie(self):
        return "{} {} i zarabiam {}".format(self.imie, self.nazwisko, self.pensja)


class Menadzer(Pracownik):

    def przedstaw_sie(self):
        return "{} {}, jestem menadżerem i zarabiam {}".format(self.imie,
self.nazwisko, self.pensja)


jozek = Pracownik("Józek", "Bajka", 2000)
adrian = Menadzer("Adrian", "Mikulski", 12000)

print(jozek.przedstaw_sie())
print(adrian.przedstaw_sie())
```

3. Przesłanianie metod.

Przesłanianie metod to dość powszechna praktyka w programowaniu obiektowym, która pozwala na zmianę zachowania obiektu poprzez zmianę implementacji jednej z istniejących metod obiektu nadrzędnego. Wielokrotnie wykorzystywaliśmy ten mechanizm np. poprzez wywołanie metody `print(zmienna)` gdzie `zmienna` może być dowolnego typu, a na wyjściu zawsze otrzymywaliśmy typ `str` tuż przed jej wypisaniem na standardowe wyjście. "Pod spodem" wywoływana jest metoda `__str__()` danego obiektu, która może być przesłonięta na dowolnym etapie grafu dziedziczenia.

> Bardziej szczegółowo zostało to opisane w rozdziale 3 oficjalnej dokumentacji: https://docs.python.org/3/reference/datamodel.html

Poniżej przykład bez przesłaniania metody `__str__()`, która jednak została odziedziczona po klasie `object`.

***Listing* 6**

```
class Player:
    pass

p1 = Player()
print(p1)
# przykładowe wyjście
# <__main__.Player object at 0x00000242B8772AD0>
```

Jeżeli chcemy jednak zmienić domyślną postać łańcuchową dla naszej własnej klasy, to przesłaniamy metodę `__str__()`.

Listing 7

```
class Player:

    def __init__(self, nickname):
        self.nickname = nickname

    def __str__(self):
        return f'Player with nickname "{self.nickname}"'

p1 = Player('dyzio')
print(p1)
# wyjście
# Player with nickname "dyzio"
```

Zadania

Zadanie 1
Zadeklaruj klasę `Point` o właściwościach `x` oraz `y` (oba typ `int`). Obie właściwości powinny posiadać domyślną wartość równą zero (wywołanie konstruktora bez podania wartości inicjalizujących).

Zadanie 2
Przesłoń w klasie `Point` metodę `__str__()` tak, aby zwracała tekst Point(x, y) gdzie x i y przedstawia aktualną wartość tych właściwości.

Zadanie 3
Sprawdź, czy możliwe jest pomnożenie obiektu typu `Point` przez liczbę całkowitą. Zaimplementuj taką możliwość poprzez przesłonięcie metody `__mul__()`. Przetestuj działanie.

Zadanie 4
Przesłoń w klasie `Point` metodę `__eq__()` odpowiedzialną za porównywanie tego obiektu (self) z innym (other) w sensie tożsamości. Wartość `True` zwracana jest tylko wtedy, gdy jest to dokładnie ten sam typ obiektu (czyli Point) oraz wartości `x` i `y` są identyczne dla obu obiektów.

Zadanie 5
Stwórz klasę o nazwie `Polygon` i zdefiniuj właściwość `points` typu `list`, która będzie docelowo

przechowywała obiekty typu `Point`. Inicjalizuj pustą listę w konstruktorze. Zdefiniuj również metodę `add_point(point: Point)`, która będzie dodawała punkt do listy.

Zadanie 6

W klasie `Polygon` przesłoń metodę `__str__()` tak, aby wypisanie `Polygon` wyglądało mniej więcej tak: Polygon[Point(2, 3), Point(1,1), ...].

Zadanie 7

W klasie `Polygon` przesłoń metodę `__getitem__()`, tak aby możliwe było zwrócenie pojedynczego punktu (item to int) oraz wycinka (item to slice). W tej metodzie obsłuż wyjątek `TypeError` jeżeli nie jest to int lub slice.