Laboratorium 5

Elementy programowania obiektowego w języku Python cz. 1

Zagadnienia

- **Klasy i instancje klas**
- **Atrybuty i metody klasy**
- **Metody statyczne**
- **Tworzenie właściwości za pomocą dekoratorów**
- **Dokumentowanie kodu**

Klasy i instancje

Programowanie obiektowe (ang. Object-Oriented Programming, OOP) jest jednym z najważniejszych paradygmatów programowania, który ma na celu ułatwienie tworzenia, zarządzania i rozszerzania oprogramowania. W języku Python, programowanie obiektowe odgrywa kluczową rolę, umożliwiając tworzenie skomplikowanych struktur danych, reprezentujących obiekty i ich zachowanie. W programowaniu obiektowym, oprogramowanie jest organizowane wokół obiektów, które są instancjami klas. Klasa jest szablonem lub wzorcem definiującym cechy i zachowanie obiektów. Każdy obiekt jest konkretnej klasy i może posiadać atrybuty (zmienne) oraz metody (funkcje), które operują na tych atrybutach. Definicja klasy w języku Python wygląda następująco:

```
class Car:
    def __init__ (self, brand, model, prod_year, horsepower, price):
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.price = price

    def isLuxury(self):
        return self.price >= 5e5
```

Powyższa klasa Car posiada specjalną funkcję __init__, której zadaniem jest zainicjowanie atrybutów obiektu po jego utworzeniu (konstruktor klasy). Wewnątrz tej funkcji znajdują się definicje atrybutów klasy takich jak marka, model, rok produkcji, moc i cena. Dodatkowo, klasa posiada metodę isLuxury, która pozwala sprawdzić, czy dany pojazd jest luksusowy w zależności od warunku logicznego. Metody odnoszące się do instancji klasy przyjmują jako argument obiekt self, który oznacza instancję, na rzecz której metoda jest wywoływana. Utworzenie instancji klasy Car wraz z wywołaniem metody isLuxury zaprezentowano poniżej:

```
car1 = Car("Skoda", "Superb", 2012, 140, 8e4)
car2 = Car("Ferrari", "F12", 2010, 650, 1e6)
print(f"Samochod {car1.brand} {car1.model} " + ("jest " if car1.isLuxury()
else "nie jest" ) + " luksusowy")
print(f"Samochod {car2.brand} {car2.model} " + ("jest " if car2.isLuxury()
else "nie jest" ) + " luksusowy")
```

Listę atrybutów dla danego obiektu można wyświetlić korzystając z funkcji dir:

```
print(dir(car1))
print("\n" + "-"*100 + "\n")
print(dir(Car))
```

Atrybuty i metody klasy

Warto zwrócić uwagę, że jako atrybuty dla instancji klasy traktowane są również metody oraz metody specjalne. Dodatkowo, klasa sama w sobie również jest pewnego rodzaju obiektem. Oznacza to, że możliwe jest definiowanie atrybutów dla klasy:

```
class Car:

    car_counter = 0
    def __init__ (self, brand, model, prod_year, horsepower, price):
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.price = price
        Car.car_counter += 1

    def isLuxury(self):
        return self.price >= 5e5

print(Car.car_counter)
car1 = Car("Skoda", "Superb", 2012, 140, 8e4)
print(car1.car_counter)
car2 = Car("Ferrari", "F12", 2010, 650, 1e6)
print(car2.car_counter)
```

W powyższym przykładzie widać, że do atrybutów klasy możliwy jest dostęp również z poziomu instancji danej klasy. Potwierdza to również wywołanie funkcji dir dla klasy oraz jej obiektów. Możliwe jest również definiowanie metod wywoływanych na rzecz klasy, a nie jej instancji. Takie metody należy opatrzyć dekoratorem @classmethod, a jako argument przekazać klasę, czyli słowo kluczowe cls. Są one przydatne na przykład do tworzenia obiektów klasy na podstawie danych wejściowych w określonej formie:

```
class Car:

    car_counter = 0
    def __init__ (self, brand, model, prod_year, horsepower, price):
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.price = price

    def isLuxury(self):
        return self.price >= 5e5
```

```
    @classmethod
    def carFromText(cls,txt):
        car = cls(*txt.split(","))
        return car

txt = "Fiat,Uno,2000,65,2000"
car = Car.carFromText(txt)
print(car.brand,car.model,car.prod_year,car.price,sep=" ")
```

Metody statyczne

Dla klas możliwe jest również definiowanie metod, których wywołanie nie jest w żaden sposób powiązane ani z klasą ani z obiektem. Metody, które umieszczone są w klasie tylko ze względu na ich logiczne bądź funkcjonalne powiązanie z klasą nazywamy metodami statycznymi i definiujemy następująco:

```
class Car:

    def __init__ (self, brand, model, prod_year, horsepower, price):
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.price = price

    @staticmethod
    def convert_kw_hp(kw):
        return int(round(kw*1.36,0))

car_brand="Seat"
car_model="Ibiza"
car_prod_year=2004
car_kw_power=59
car_price=7500
car1=Car(car_brand,car_model,car_prod_year,Car.convert_kw_hp(car_kw_power),
car_price)
print(car1.brand,car1.model,car1.prod_year,car1.horsepower,car1.price,sep="
")
```

Jak widzimy, metoda statyczna convert_kw_hp może być wywołana bez tworzenia instancji klasy.

Tworzenie właściwości za pomocą dekoratorów

W przeciwieństwie do języków takich jak C++, klasy w języku Python nie obsługują domyślnie mechanizmu enkapsulacji. Możliwy jest zatem dostęp do atrybutów instancji klasy oraz dowolna ich modyfikacja:

```
class Car:
    def __init__(self,brand,model,prod_year,horsepower,price):
        self.brand = brand
```

```
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.price = price

    def __str__(self):
        return (f"Samochod: {self.brand} {self.model}, rocznik:
{self.prod_year}, moc: {self.horsepower}" f" cena {self.price} zl")


car1=Car("Toyota","Avensis",2008,120,18000)
print(car1)
car1.price = -200
print(car1)
```

Funkcja __str__ w Pythonie jest specjalną metodą, która definiuje sposób, w jaki obiekt jest reprezentowany jako string. Jest ona wywoływana w trakcie użycia funkcji str() na obiekcie lub wydrukowania go za pomocą print(). W wielu przypadkach, możliwość modyfikacji atrybutów bez kontroli jest niepożądana - w powyższym przykładzie zmieniono cenę samochodu na ujemną. Aby zapobiec tego typu sytuacjom, a jednocześnie zachować wysoką elastyczność pisanego kodu, jaką stara się nam zapewnić Python, wystarczy skorzystać z ukrywania atrybutów klasy. Można tego dokonać poprzez dodanie do nazwy atrybutu prefiksu w postaci podwójnego podkreślnika:

```
class Car:
    def __init__(self,brand,model,prod_year,horsepower,price):
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.__price = price

    def __str__(self):
        return (f"Samochod: {self.brand} {self.model}, rocznik:
{self.prod_year}, moc: {self.horsepower}" f" cena {self.__price} zl")


car1=Car("Toyota","Avensis",2008,120,18000)
print(car1.__price)
```

Czy to oznacza, że atrybutu tak skonstruowanego nie da się zmienić? Żeby odpowiedzieć na to pytanie przyjrzyjmy się wywołaniu funkcji vars (funkcja vars wyświetla słownik zawierający atrybuty obiektu wraz z ich wartościami) z argumentem w postaci obiektu typu Car:

```
car1=Car("Toyota","Avensis",2008,120,18000)
print(car1)
print(vars(car1))
```

Okazuje się, że atrybut price nadal funkcjonuje, jednak otrzymał dodatkowo przedrostek w postaci '_NazwaKlasy'. Tak naprawdę, nie ukryliśmy całkowicie atrybutu price, a jego wartość nadal można zmodyfikować posługując się właśnie tą nową nazwą:

```
car1._Car__price = -200
print(car1)
```

W języku Python przyjęło się jednak, że zmienne ukryte w ten sposób są częścią niepublicznego interfejsu klasy i nie powinny być modyfikowane. Jest to jedynie pewna umowa pomiędzy programistami korzystającymi z tak napisanych klas. Zapewnienie odpowiedniego interfejsu publicznego powinno mieć miejsce przy użyciu metod zwracających (getterów) i ustawiających (setterów) wartości atrybutów klasy. Atrybut obiektu klasy, który powinien być dostępny publicznie ale tylko poprzez dedykowany interfejs nazywamy właściwością (ang. property) i może on być zdefiniowany następująco:

```
class Car:
    def __init__(self,brand,model,prod_year,horsepower,price):
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.__price = price

    def __str__(self):
        return (f"Samochod: {self.brand} {self.model}, rocznik:
{self.prod_year}, moc: {self.horsepower}" f" cena {self.__price} zl")

    def __getPrice(self):
        return self.__price
    def __setPrice(self,price):
        if price>0:
            self.__price = price
        else:
            raise ValueError("Price must be greater than 0!")

    price = property(__getPrice, __setPrice, None)

car1=Car("Toyota","Avensis",2008,120,18000)
print(car1)
car1.price=100
print(car1)
car1.price=-100
```

Funkcja property jest wbudowana w Pythonie i służy do tworzenia właściwości w klasach. Przyjmuje do czterech argumentów: getter (do pobierania wartości), setter (do ustawiania wartości), deleter (do usuwania wartości), String dokumentacyjny (opcjonalny). Powyżej widać, że poprzez właściwość opracowany został interfejs umożliwiający zwracanie i modyfikowanie wartości atrybutu price, jednak na określonych warunkach. Bardziej czytelnym sposobem definiowania właściwości jest wykorzystanie dekoratorów:

```
class Car:
    def __init__(self,brand,model,prod_year,horsepower,price):
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
        self.horsepower = horsepower
        self.__price = price
```

```
    def __str__(self):
        return (f"Samochod: {self.brand} {self.model}, rocznik:
{self.prod_year}, moc: {self.horsepower}" f" cena {self.__price} zl")

    @property
    def price(self):
        return self.__price
    @price.setter
    def price(self,price):
        if price>0:
            self.__price = price
        else:
            raise ValueError("Price must be greater than 0!")

car1=Car("Toyota","Avensis",2008,120,18000)
print(car1)
car1.price=100
print(car1)
car1.price=-100
```

Dokumentowanie klas

Podobnie jak w przypadku funkcji, tak i klasy posiadają możliwość dokumentowania poprzez tzw. docstrings:

```
class Car:
    """
    A class representing a car.

    Attributes:
        brand (str): The brand of the car.
        model (str): The model of the car.
        prod_year (int): The production year of the car.
        horsepower (int): The horsepower of the car.
        price (float): The price of the car.
    """

    def __init__(self, brand, model, prod_year, horsepower, price):
        """
        Initializes a new instance of the Car class.

        Args:
            brand (str): The brand of the car.
            model (str): The model of the car.
            prod_year (int): The production year of the car.
            horsepower (int): The horsepower of the car.
            price (float): The price of the car.
        """
        self.brand = brand
        self.model = model
        self.prod_year = prod_year
```

```
        self.horsepower = horsepower
        self.price = price

    @staticmethod
    def convert_kw_hp(kw):
        """
        Converts power from kilowatts (kW) to horsepower (hp).

        Args:
            kw (float): The power in kilowatts.

        Returns:
            int: The power in horsepower, rounded to the nearest integer.
        """
        return int(round(kw * 1.36, 0))
```

Zadania

1. Utwórz klasę Book, która będzie miała atrybuty title, author, year i genre. Następnie utwórz kilka instancji (przynajmniej 3) tej klasy i wyświetl ich atrybuty.
2. Utwórz klasę Library, która będzie miała atrybut klasowy total_books zliczający liczbę książek oraz metodę add_book dodającą książki do biblioteki. Dodaj do biblioteki utworzone obiekty Book z poprzedniego zadania i wyświetl liczbę książek w bibliotece.
3. Napisz klasę MathCalculations z metodą statyczną, która oblicza sumę liczb naturalnych od 1 do n. Dodaj drugą metodę klasową, która oblicza silnię liczby n.
4. Utwórz klasę Rectangle z atrybutami width i height (za pomocą dekoratorów zabezpiecz właściwości przed wartościami niedodatnimi). Dodaj właściwość area, która będzie obliczać pole prostokąta.
5. Dodaj dokumentację do każdej klasy.