
	Politechnika Bydgoska im. J. J. Śniadeckich Wydział Telekomunikacji, Informatyki i Elektrotechniki		
Przedmiot	Skryptowe języki programowania		
Prowadzący	mgr inż. Martyna Tarczewska		
Temat	Obiektowość Pythona		
Student			
Nr ćw.	4	Data wykonania	
Ocena		Data oddania spr.	

1. Cel ćwiczenia

Celem ćwiczenia jest poznanie koncepcji programowania obiektowego w języku *Python* oraz wykonanie zadań utrwalających przyswojoną wiedzę. **Funkcje i zmienne stworzone podczas zajęć powinny być odpowiednio otypowane z użyciem biblioteki typing. Kod powinien być napisany w języku angielskim.**

2. Informacje podstawowe

2.1. Klasy

Klasy w Pythonie funkcjonują dokładnie w taki sam sposób, jak to się dzieje w innych obiektowych językach programowania. Klasę można rozumieć jako złożony typ danych, jako szablon dla elementów. Klasa definiuje zbiór atrybutów (zmiennych) oraz funkcji. Klasy mogą podlegać dziedziczeniu.

Sensem programowania obiektowego (w ogóle i w Pythonie) jest lepsza reprezentacja rzeczywistości i ponowne wykorzystanie napisanego już kodu. Kiedy rozejrzemy się po sali laboratoryjnej, nie widzimy napisów, zmiennych liczbowych czy znakowych. Widzimy poszczególne obiekty – biurko, komputer, innych Studentów. Dopiero te obiekty mogą być opisane zmiennymi (np. wysokość, liczba rdzeni, imię). Każdy z rzeczywistych obiektów będzie się też inaczej zachowywał. To zachowanie będą opisywać funkcje. Funkcje zwykle nazywa się od czasowników.

```
from typing import List

class Student:
    def __init__(self):
        self.name = ""
        self.last_name = ""
        self.marks = []
    def give_name(self, name: str, last_name: str) -> None:
        self.name = name
        self.last_name = last_name
    def give_mark(self, mark: int) -> None:
        self.marks.append(mark)
    def get_marks(self) -> List[int]:
```

```
        return self.marks
    def say_hello(self) -> None:
        print("Hello! I'm " + self.name + " " + self.last_name)
```

W powyższej definicji klasy istotne jest kilka rzeczy. Pierwsza to inicjalizacja (funkcja oznaczona słowem `init`). W niej definiuje się zadania, które mają zostać wykonane w momencie tworzenia (inicjalizacji) obiektu (w Pythonie INSTANCJI). W tym przypadku zostaną utworzone 2 zmienne napisowe (puste) i jedna zmienna przechowująca listę (również pusta).

Następnie zapisano kolejne funkcje tej klasy, które opisują możliwe zachowanie instancji. Funkcje te mogą przyjmować różne argumenty (np. w funkcji *give_name* mamy dwa parametry typu string). Mamy również parametr `self`. Oznacza on, że ta funkcja będzie działała dla tego konkretnego obiektu, na rzecz którego została wywołana. Ten zapis jest równoważny *this* znanemu z Javy.

2.2. Instancje klasy

Każda instancja klasy posiada wszystkie zmienne zdefiniowane w klasie oraz ma dostęp do wszystkich funkcji, które klasa definiuje. Poniżej widać sposób tworzenia instancji klasy oraz wywołanie różnych metod na rzecz tej instancji. Zwraca uwagę brak słowa `self`. Wywołanie funkcji na rzecz obiektu odbywa się za pomocą kropki.

Funkcję można też wywołać przez podanie nazwy klasy i poszerzenie listy parametrów o instancję, na rzecz której wywołujemy funkcję.

```
s = Student()
s.give_name("Jane", "Doe")
s.give_mark(5) # wywołanie sposób 1
Student.give_mark(s, 3) # wywołanie sposób 2
print(s.get_marks())
s.say_hello()
```

2.3. Dziedziczenie

Dziedziczenie to podstawowy mechanizm każdego języka obiektowego. Jest to przejęcie atrybutów i funkcji z klasy nadrzędnej do nowej, podrzędnej klasy.

Dziedziczenie w praktyce dobrze wyobrazić sobie na pojazdach zmechanizowanych. Możemy napisać klasę opisującą pojazd (każdy pojazd jeździ, ma tablicę rejestracyjną, właściciela). Z klasy pojazd mogą dziedziczyć inne klasy, np. motocykl i samochód – będą one zupełnie inaczej realizowały niektóre funkcje (np. motocyklem nie będziemy poruszać się w czwórkę), ale oba te pojazdy nadal jeżdżą, mają tablice i właścicieli. Przy tworzeniu drzewa dziedziczenia można powiedzieć sobie samochód (klasa podrzędna) JEST pojazdem (klasa nadrzędna).

```
class Vehicle:
    def get_sound(self):
        print("vehicle's brum brum")

class Car(Vehicle):
```

```
def __init__(self, owner: str, table: str):
    self.owner = owner
    self.table = table

def get_sound(self) -> None:
    print("car's brum brum")

def get_owner(self) -> str:
    return self.owner
```

Dziedziczenie może odbywać się po więcej niż jednej klasie. Trzeba jednak uważać w przypadku deklarowania takiej samej funkcji dla dwóch klas, z których dziedziczy kolejna klasa. Taka problemowa sytuacja zachodzi w przykładzie poniżej. Należy zastanowić się (lub sprawdzić), która z metod zostanie wywołana w przypadku wywołania funkcji *get_sound* dla instancji klasy *thing*.

```
class Item:
    def get_sound(self) -> None:
        print("item's sound")

class Element:
    def get_sound(self) -> None:
        print("element's sound")

class Thing(Element, Item) -> None:
    def say_hello(self):
        print("hello!")
```

2.3. Tworzenie instancji

W momencie tworzenia nowego obiektu korzystamy z nawiasu. W tym nawiasie podajemy wszystkie parametry, którymi chcemy zainicjalizować obiekt. Można nie przekazać żadnej wartości. W ciele klasy możemy zdefiniować inicjalizator (`__init__`). Będzie on wywołany w momencie tworzenia obiektu.

2.4. Hermetyzacja

Pojęcie to polega na ukryciu niektórych elementów przed innymi klasami. W Pythonie możemy sprawić, aby jakaś zmienna była widoczna poza klasą. Przyjęto, że w takim wypadku należy zacząć nazwę funkcji od podkreślenia i korzystać z niej tylko do wywoływania wewnątrz metod klasy.

2.5. Atrybuty klasy

Jeśli chcemy, aby jakaś zmienna lub funkcja dotyczyła nie konkretnego obiektu, a wszystkich obiektów jednocześnie, należy zadeklarować zmienną w klasie jak podano na poniższym przykładzie (zmienna *quantity* to liczba zwiększająca się za każdym razem, gdy tworzymy nową instancję klasy *student*).

```
class Student:
```

```
quantity = 0

def __init__(self):
    self.name = ""
    self.last_name = ""
    self.marks = []
    Student.quantity += 1
```

2.6. Przeciążanie operatorów

Przeciążanie to mechanizm pozwalający na zmianę dotychczasowego sposobu działania jakiegoś operatora lub funkcji. Pierwszym przykładem przeciążania jest własna definicja funkcji `__init__`, czyli przeciążenie konstruktora. Ponadto można stworzyć własne sposoby porównywania instancji jednej klasy.

Przeciążanie operatorów nie jest jednak bardzo często używane, gdyż może prowadzić do błędów i stworzenia nieczytelnego kodu.

Może się to okazać pomocne, kiedy chcemy porównywać dwie instancje lub je szeregować. Jeśli zadeklarujemy np. funkcję `lt`, to będziemy mogli używać operatora `<` między dwiema instancjami.

```
class Student:

    def __init__(self, name: str, last_name: str, index: int):
        self.name = name
        self.last_name = last_name
        self.index = index

    def __repr__(self) -> str:
        return (self.name + " " + self.last_name)

    def __str__(self) -> str:
        return (self.last_name + " " + self.name)

    def __eq__(self, o: Student) -> bool: #funkcja do sprawdzania równości (==)
        return self.index == o.index

    def __ne__(self, o: Student) -> bool: #funkcja zastępująca !=
        return self.index != o.index

    def __lt__(self, o: Student) -> bool: #funkcja zastępująca <
        return self.index < o.index

    def __gt__(self, o: Student) -> bool: #funkcja zastępująca >
        return self.index > o.index

s1 = Student('Joe', 'Doe', 111111)
print(repr(s1))

s2 = Student('Jane', 'Key', 222222)
```

```
print(str(s2))
if (s1 == s2):
    print("objects equal!")
else:
    print("not equal..")
```

2.7. Słowo kluczowe *super*

Aby odwołać się do funkcji klasy bazowej możemy skorzystać ze słowa kluczowego *super*.

```
class Skoda_fabia(Car):
    def __init__(self, owner: str, table: str):
        print("skodafabia constructor")
        super().__init__(owner, table)
```

3. Przebieg ćwiczenia

3.1. Zadanie 1.

Skopiować kod z punktów 2.1 i 2.2. Dodać dwie kolejne funkcje do klasy *student*: funkcję obliczającą średnią z otrzymanych ocen i funkcję nadającą studentowi numer indeksu. Zmodyfikować funkcję *sayHello* tak, aby podawała imię, nazwisko i numer indeksu.

3.2. Zadanie 2.

Skopiować kod z punktu 2.3, który jest przykładem dziedziczenia. Utworzyć instancję klasy *vehicle* i instancję klasy *car*. Sprawdzić, która funkcja *getSound* wywoła się dla poszczególnych obiektów. Wywołać funkcję *getOwner* dla obu obiektów – czy nie powoduje to błędów? Jeśli tak, poprawić błędy.

3.3. Zadanie 3.

Rozszerzyć działanie klasy *student* o licznosc (patrz. 2.5). Utworzyć kilka instancji klasy *student* i sprawdzić licznosc dla każdej z nich. Sprawdzić dostęp do tej zmiennej przez poszczególne obiekty oraz przez klasę (np. *s1.quantity* i *student.quantity*).

3.4. Zadanie 4.

Utworzyć funkcję do użytku wewnętrznego w klasie *student*. Funkcja ma sprawdzać, czy w zmiennych *name* i *last_name* nie ma pustych stringów (powinna zwracać *True* lub *False*). Wywołać tę funkcję w funkcji *sayHello* i sprawdzić działanie. Spróbować wywołać tę funkcję spoza klasy.

3.5. Zadanie 5.

Skopiować kod dotyczący dziedziczenia po wielu klasach. Sprawdzić jego działanie poprzez wywołanie metody *getSound* dla obiektów typu *item*, *element* i *thing*. Zamienić kolejność klas bazowych w definicji klasy *thing* i ponownie uruchomić kod. Jaka jest różnica?

3.6. Zadanie 6.

Zapoznać się z częścią 2.6. wstępu teoretycznego. Stworzyć własne wersje funkcji porównujących `__lt__` (<), `__gt__` (>), `__eq__` (==), `__le__` (<=), `__ge__` (>=) oraz `__ne__` (!=). Można skorzystać z którejs z wcześniej utworzonych klas (np. *student*).

3.7. Zadanie 7.

W części 2.6. podano też przeciążanie funkcji `__repr__` i `__str__`. Sprawdzić działanie klasy *student* bez przeciążania tych dwóch funkcji i z przeciążaniem.

3.8. Zadanie 8.

Sprawdzić działanie funkcji `dict`, np. przez polecenie: `print (s.__dict__)`.

3.9. Zadanie 9.

Wykorzystać którąś z wcześniej utworzonych klas. Dodać inicjalizator. Dodać klasę dziedziczącą i stworzyć kolejny inicjalizator. Skorzystać z `super` i zbadać kolejność wykonywania się inicjalizatorów.

3.10. Zadanie 10.

Wybrać jedną z utworzonych funkcji i przy pomocy biblioteki `pytest` utworzyć po trzy testy sprawdzające różne scenariusze działania wybranej funkcji. Wyniki testów zamieścić w sprawozdaniu.

4. Sprawozdanie

Sprawozdanie z laboratorium powinno zawierać:

- wypełnioną tabelę z początku instrukcji,
- kody programów będących rozwiązaniami wszystkich zadań wraz z komentarzami,
- demonstrację działania programu,
- odpowiedzi na pytania zawarte w zadaniach,
- wnioski.