Laboratorium 9 — Ćwiczenia w programowaniu obiektowym. Tworzenie własnych typów danych.

Paradygmaty programowania obiektowego

Hermetyzacja zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Tylko własne metody obiektu są uprawnione do zmiany jego stanu. W przypadku tego języka założenie to jest mocno osłabione, w związku z umownym podejściem do metod i pól prywatnych. Mimo to język Python udostępnia bardzo elastyczny mechanizm, znany chociażby z języka *C#*, oparty na właściwościach (składają się one z akcesorów - specjalnych metod pozwalających na zwrócenie zmiennych, oraz opcjonlanych mutatorów pozwalający na nadanie nowych wartości) na prostsze zarządzanie zmiennymi prywatnymi (bez wykorzystani metod `get()`, `set()`).

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

    @property
    def a(self):
        return self.__a
    @a.setter
    def a(self, value):
        if(value < 0):
            self.__a = 0
        else:
            self.__a = value

pr = Przyklad(5)
print(pr.a)
pr.a = -10
print(pr.a)
```

Jak widzimy w powyższym przykładzie mimo, że „a” jest metodą, to traktowana jest jak zwykłe pole. Użytkownik korzystający z klasy nie widzi, że tak naprawdę operuje na zmiennej prywatnej „__a”, a każda próba przypisania nowej wartości jest kontrolowana przez napisany w setterze kod.

Zadanie 1 Zamień wszystkie pola klasy `Osoba` (z poprzednich zajęć) na właściwości. Pamiętaj, aby settery ustawić jedynie dla wagi i wzrostu, pozostałe pola mają pozostać tylko do odczytu. W setterach wagi i wzrostu upewnij się, że nie wprowadzono liczby ujemnej, jeśli tak to wstaw 0. Dodaj także właściwość wiek, która zastąpi metodę obliczającą aktualny wiek.

Dziedzicznie pozwala na utworzenie obiektu bardziej szczegółowego na podstawie ogólnego (np. na podstawie klasy `Figura`, możemy utworzyć klasy `Prostokat`, `Kolo`, czy `Trojkat`, które posiadają pewne wspólne cechy z klasą bazową). Dzięki zastosowaniu dziedziczenia, jeśli pojawi się potrzeba zmodyfikowania funkcji wspólnej dla wszystkich potomków, nie będzie konieczne modyfikowania jej dla każdej klasy z osobna. Istotną cechą dziedziczenia w języku Python jest konieczność ręcznego uruchomienia konstruktora klasy bazowej, o ile w klasie potomnej utworzyliśmy nowy konstruktor (domyślnie się tak nie dzieje).

```
class Bazowa:
    def __init__(self):
        print('konstruktor 1')

    def metoda1(self):
        print('metoda1')

    def metoda2(self):
        print('metoda2')

class Potomek(Bazowa):
    def __init__(self):
        super().__init__() # wywolanie konstruktora klasy bazowej
        print('konstruktor 2')

    # przeciazenie metody
    def metoda2(self):
        super().metoda2() # przeciazajac metode mamy mozliwosc wywolania
            oryginalu
        print('metoda 2 - potomek')

pot = Potomek()
pot.metoda1();
pot.metoda2()
```

Zadanie 2 Twoim zadaniem będzie stworzenie klasy `Pracownik`, będącej potomkiem klasy `Osoba`. Do nowej klasy należy dodać dwa nowe pola: `miejsce zatrudnienia` (pole obowiązkowe) oraz `pensja` (opcjonalne) i uwzględnić je w konstruktorze nowej klasy oraz odpowiednie właściwości dla nowych pól (pamiętaj, że pensja nie może być ujemna oraz, że oba pola powinny móc zmieniać wartości). Przeciąż metody `__str__` oraz `__repr__`, aby dodatkowo wyświetlały miejsce zatrudnienia oprócz oryginalnych danych.

Polimorfizm (wielopostaciowość) pozwala traktować różnorodne dane w ten sam sposób. Przykładowo pozwala on na zastosowanie klas potomnych wszędzie tam, gdzie mogłaby zostać użyta klasa bazowa. Pozwala to na uogólnienie pewnych czynności. Prostym przykładem może być lista obiektów typu klasyfikator mimo, że w liście mogą zostać umieszczone klasyfikatory różnego typu (np. MLP, SVM, AdaBoost), to dzięki temu, że posiadają wspólny zestaw metod odziedziczony po klasie bazowej, możliwe jest wywołanie w pętli wszystkich metod dostępnych dla klasy bazowej, np. metody `classify`. Przykład dla powyższych klas:

```
a = [Bazowa(), Potomek(), Potomek()]
for obj in a:
    obj.metoda2()
    print('------')
```

W efekcie otrzymamy:

```
metoda2
------
metoda2
metoda 2 - potomek
------
metoda2
metoda 2 - potomek
```

Jak widać dla obiektów klasy potomnej wywołana została przeciążona wersja metody 2.

Zadanie 3 Utwórz zbiór (`set`) o nazwie `baza`. Dodaj do niego 5 obiektów typu `Osoba` i 3 obiekty typu `Pracownik`. Wykorzystując jedną pętlę, wyświetl reprezentacje (`repr(obiekt)`) wszystkich dodanych obiektów.

Abstrakcja Zdarza się, że klasa bazowa ma być tworem czysto abstrakcyjnym i sama w sobie nie ma być używana, lub też niektóre z jej metod muszą zostać zaimplementowane w klasach po niej dziedziczących (zakładamy, że o ile dana metoda nie dotyczy samej klasy bazowej, to w klasach potomnych jej obecność jest wymagana). Przykładem takiej abstrakcyjnej klasy może być klasa `Figura`, która sama w sobie nic nie reprezentuje, ale może stanowić pewnego rodzaju interfejs dla klas powstałych na jej bazie.

```
from abc import ABC, abstractmethod
class Figura(ABC):
    @abstractmethod
    def oblicz_pole(self):
        pass

    @abstractmethod
    def typ_figury(self):
        pass

    def print(self):
        print('Figura:', self.typ_figury(), ', Pole:', self.oblicz_pole())

# fig = Figura() --- zwroci blad, poniewaz klasa jest abstrakcyjna
```

Jak widzimy aby utworzyć taką klasę musimy zaimportować odpowiednie moduły. Klasa abstrakcyjna w Pythonie musi dziedziczyć po klasie ABC. Dodatkowo nie da się utworzyć obiektu dla takiej klasy.

```
class Kwadrat(Figura):
    def __init__(self, a):
        self.a = a

    def oblicz_pole(self):
        return self.a**2

    def typ_figury(self):
        return 'Kwadrat'

kw = Kwadrat(5)
kw.print()
```

Zadanie 4 Dodaj dwie kolejne figury dziedziczące po klasie abstrakcyjnej `Figura`: `Kolo`, `Trojkat`. Utwórz po jednym obiekcie i wywołaj dla każdego z nich odziedziczoną metodę `print()`.

Mechanizm wyjątków W przypadku, gdy tworzymy kod z myślą o ponownym wykorzystaniu jego fragmentów lub udostępnieniu możliwe jest, że utworzona przez nas funkcje zostaną wywołane nieprawidłowo. Takie wywołanie może zmienić prawidłowy przebieg wykonywanego programu do tego stopnia, że należy przerwać jego wykonanie lub obsłużyć zaistniałą sytuację. Przykładem takiego zdarzenia może być próba otwarcia nieistniejącego pliku. Z myślą o takich sytuacjach stworzony został mechanizm wyjątków, który w momencie natrafienia na taką sytuację generuje wyjątek, zapamiętuje stan bieżącego programu i przechodzi do jego obsługi lub przerywa wykonanie.

Obsługa wyjątku:

```
try:
    x = int(input("Podaj liczbe: "))
except ValueError:    # oczekiwanie na wyjatek konkretnego typu
    print("Wprowadzono nie poprawne dane")
except: # oczekiwanie na dowolny wyjatek
    print("?")
finally: # ponizszy kod wykonana sie niezaleznie od wystapienia wyjatku
    print("")
```

Zgłoszenie wyjątku:

```
try:
    raise Exception('wystapil wyjatek') # zgloszenie ogolnego wyjatku
except Exception as inst:  # przypisanie instancja wyjatku
    print(inst)
```

Przykładowe wyjątki:

```
NameError # odwolanie do nieistniejacej zmiennej
ValueError # niepoprawna wartosc
ZeroDivisionError # dzielenie przez zero
TypeError # nieprawdilowy typ danych
```

Nowy wyjątek:

```
class NowyWyjatek(Exception):
    pass
```

Zadanie 5 Stosując funkcję *isinstance* upewnij się, że dane przekazywane do konstruktorów figur są liczbami całkowitymi lub zmiennoprzecinkowymi w przeciwnym wypadku zwróć wyjątek *TypeError*.

Kopiowanie klasy W związku z brakiem możliwości przeciążania metod, Python nie posiada możliwości utworzenia konstruktora kopiującego. Alternatywą jest przeciążenie metod `__copy__` oraz `__deepcopy__` odpowiadających za możliwość wykonania płytkiej kopi klasy oraz głębokiej. Metody te dostępne są po załadowaniu modułu `copy`

Płytka kopia klasy:

```
from copy import copy
class simpleClass:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return "a: " + str(self.a)

    def __copy__(self):
        # tworzymy instancje klasy bez wywolania __init__()
        cls = self.__class__
        result = cls.__new__(cls)
        # kopiujemy wszystkie pola klasy
        result.__dict__.update(self.__dict__)
        # (kopie parametrow mozna wykonac takze recznie)
        # result.a = self.a
        return result

a = simpleClass(3)
b = copy(a)
print(a)
print(b)
```

Głęboka kopia klasy:

```
from copy import copy, deepcopy
class complexClass:
    def __init__(self, a):
        self.a = [a] # lista jak pole klasy

    def __str__(self):
        return "a: " + str(self.a) + " id: " + str(id(self.a))

    def __copy__(self):
        cls = self.__class__
        result = cls.__new__(cls)
        result.__dict__.update(self.__dict__)
        return result

    def __deepcopy__(self, memo):
        cls = self.__class__
        result = cls.__new__(cls)
        # tworzymy glebokie kopie kazdego pola
        memo[id(self)] = result # zapobiega nieskonczonej petli
        for k, v in self.__dict__.items():
            setattr(result, k, deepcopy(v, memo))
        # lub recznie
        # result.a = deepcopy(self.a)
        return result

a = complexClass(3)
b = copy(a)
c = deepcopy(a)
print(a)
print(b) # identyczne id obiektu listy
print(c) # lista jest nowym obiektem
```

`memo[id(self)] = result` — linia ta zapobiega powstaniu rekurencji w przypadku, gdy któraś ze zmiennych posiada referencje na kopiowany obiekt.

Wersja alternatywna — istnieje też podejście oparte o metodę klasową (poniżej wariant płytki):

```
class simpleClassV2:
    def __init__(self, a):
        self.a = a

    def __str__(self):
        return "a: " + str(self.a)

    @classmethod
    def makeCopy(cls, obj):
        result = cls.__new__(cls)
        result.__dict__.update(obj.__dict__)
        return result

a = simpleClassV2(3)
b = simpleClassV2.makeCopy(a)
print(a)
print(b)
```