Zaawansowane Techniki Obiektowe

Gdzie jesteśmy? Opanowaliśmy fundamenty OOP: klasy, dziedziczenie, polimorfizm i enkapsulację.

Cel na dziś: Przećwiczenie zaawansowanych technik, które czynią nasze klasy bardziej potężnymi, eleganckimi i "pythonicznymi".

Plan Działania:

- Przeciążanie operatorów (__lt__).
- Deskryptory
- dataclasses - zwięzła definicja klas danych.
- Własne wyjątki.
- Pełne porównywanie (@total_ordering).
- Emulacja kontenerów.
- Własne iteratory.
- Pułapki i Optymalizacje.
- Podsumowanie.

Zadanie 1: Przeciążanie operatorów w praktyce

Problem: Nasze obiekty mogą być sortowane **(sorted(lista_graczy)** rzuca błąd).

Rozwiązanie: __lt__(self, other) Wywoływana przy self < other (używana przez sorted()).

Zadanie: Rozbuduj klasy Gracz i Wojownik z poprzednich zajęć.

Sortowanie (__lt__):

- W klasie **Gracz,** zaimplementuj **__lt__(self, other).**
- Niech zwraca **True,** jeśli **hp** naszego gracza **(self)** jest mniejsze niż hp drugiego gracza **(other).**

Przetestuj:

- Stwórz listę kilku **Graczy** i posortuj ją za pomocą **sorted().**

Omówienie Zadania 1: Przeciążanie operatorów w praktyce

Cel: Analiza rozwiązania i utrwalenie, jak metody specjalne pozwalają na przeciążanie operatorów.

Zadanie 2: Deskryptory, czyli jak działa @property

Problem:

- Jak działa **@property**? Jak stworzyć reużywalny komponent z logiką walidacji, który można przypisać do wielu atrybutów?

Rozwiązanie: Protokół Deskryptora.

- Deskryptor to obiekt przypisany do atrybutu klasy, który zarządza dostępem do danych w instancjach tej klasy.
- **__get__(self, instance, owner):** Wywoływana przy odczycie atrybutu.
- **__set__(self, instance, value):** Wywoływana przy zapisie atrybutu.

Zadanie: Stwórz deskryptor **NieujemnaLiczba**, który zapewni, że atrybut nigdy nie będzie miał wartości ujemnej.

a. Stwórz klasę **NieujemnaLiczba** (nasz deskryptor).

b. Zaimplementuj **__set_name__(self, owner, name):** ta specjalna metoda jest wywoływana automatycznie i zapisuje nazwę atrybutu (**name**) w prywatnym atrybucie deskryptora, np. self._nazwa_atrybutu.

c. Zaimplementuj **__get__:** ma pobierać wartość z wewnętrznego słownika instancji (**instance.__dict__)** używając zapisanej nazwy atrybutu.

d. Zaimplementuj **__set__:**

   - Sprawdź, czy value jest < 0. Jeśli tak, ustaw wartość na 0.
   - Zapisz wartość w wewnętrznym słowniku instancji (instance.__dict__).

e. Zmodyfikuj klasy **Wojownik** i **Mag** z poprzednich zajęć, aby używały deskryptora:

```
class Wojownik(Gracz):
    sila = NieujemnaLiczba() # Zamiast zwykłego atrybutu
    # ... reszta klasy ...

class Mag(Gracz):
    mana = NieujemnaLiczba() # Nowy atrybut
    # ... reszta klasy ...
```

f. Przetestuj: Stwórz wojownika i maga, spróbuj przypisać im ujemne wartości **sila** i **mana**, a następnie sprawdź, czy wartości zostały poprawnie ustawione na 0.

Omówienie Zadania 2 - Metody Klasy i Statyczne

Cel: Analiza rozwiązania i utrwalenie różnic między metodą instancji, metodą klasy i metodą statyczną.

Kluczowe wnioski:

- **Metoda instancji (metoda(self, ...)):** Operuje na konkretnym obiekcie. Najczęstszy typ.
- **Metoda klasy (@classmethod metoda(cls, ...)):** Operuje na klasie. Używana jako "fabryka" obiektów.
- **Metoda statyczna (@staticmethod metoda(...)):** Zwykła funkcja wewnątrz klasy. Używana jako funkcja pomocnicza.

Zadanie 3: dataclasses – Mniej kodu, więcej danych

Problem: Pisanie klas, które głównie przechowują dane, wymaga dużo powtarzalnego kodu (tzw. boilerplate):

- **__init__(self, x, y): self.x = x; self.y = y**
- **__repr__(self): return f"Punkt(x={self.x}, y={self.y})"**
- **__eq__(self, other): return self.x == other.x and self.y == other.y**

Rozwiązanie (od Pythona 3.7): Moduł dataclasses

- Automatycznie generuje za nas metody **__init__, __repr__, __eq__** i inne!
- Wystarczy zadeklarować pola i ich typy.

Składnia:

```
from dataclasses import dataclass

@dataclass
class NazwaKlasy:
    atrybut1: typ
    atrybut2: typ = wartosc_domyslna
```

Zadanie: Przepisz prostą klasę **Punkt** na **dataclass**.

a. Zaimportuj **dataclass** z modułu **dataclasses**.

b. Stwórz nową klasę **PunktData**.

c. Udekoruj ją za pomocą **@dataclass**.

d. Wewnątrz klasy, zdefiniuj atrybuty **x** i **y** wraz z ich typami (**int**).

e. Przetestuj:

- Stwórz instancję: **p1 = PunktData(10, 20).**
- Wyświetl ją: **print(p1)** (zobaczysz ładny **__repr__).**
- Porównaj dwa obiekty: **p2 = PunktData(10, 20), print(p1 == p2)** (powinno być **True**).

Omówienie Zadania 3: dataclasses

Cel: Analiza rozwiązania i utrwalenie korzyści płynących z użycia dataclasses.

Zadanie 4: Własne wyjątki

Problem: Co, jeśli w naszej grze postać z 0 HP próbuje wykonać akcję? Zwykły **ValueError** nie opisuje dobrze problemu.

Rozwiązanie: Własne, niestandardowe wyjątki.

- Tworzymy własne klasy błędów, które dziedziczą po wbudowanej klasie **Exception**.
- Dzięki temu nasz kod staje się bardziej **semantyczny** (samodokumentujący).
- Pozwala to na precyzyjną obsługę błędów za pomocą **try...except**.

Składnia:

```
class NazwaTwojegoBledu(Exception):
    pass
```

Zadanie: Zaimplementuj obsługę błędu dla postaci, która nie ma już punktów życia.

a. **Stwórz wyjątek:** Stwórz nową klasę **BrakPunktowZyciaError**, która dziedziczy po **Exception**.

b. **Zmodyfikuj Wojownika:** W klasie **Wojownik** dodaj prostą metodę **atakuj().**

c. Wewnątrz **atakuj**, dodaj warunek: jeśli **self.hp <= 0**, podnieś (**raise**) swój nowy wyjątek: raise **BrakPunktowZyciaError("Postać nie może atakować!").**

d. Jeśli postać ma HP, niech metoda po prostu wyświetli komunikat o ataku.

e. **Przetestuj:**

- Stwórz wojownika i ustaw jego HP na 0.
- Wywołaj na nim metodę **atakuj()** i zaobserwuj, że program zatrzymuje się, wyświetlając Twój nowy, czytelny błąd.

Omówienie Zadania 4: Własne Wyjątki

Cel: Analiza rozwiązania i utrwalenie, jak tworzyć i obsługiwać niestandardowe błędy.

Zadanie 5 - Pełne porównywanie z @total_ordering

Problem: W Zadaniu 1 zaimplementowaliśmy **__eq__** i **__lt__.**

Ale co z resztą operatorów: **<=, >, >=?** Czy musimy pisać je wszystkie ręcznie?

Rozwiązanie: Dekorator functools.total_ordering

- Dekorator, który automatycznie generuje brakujące metody porównania.
- Wymagania: Klasa musi zdefiniować **__eq__** oraz jedną z metod: **__lt__, __le__, __gt__,** lub **__ge__.**

Zadanie: Zrefaktoryzuj klasę **Gracz**, aby używała **@total_ordering**.

a. **Zaimportuj dekorator: from functools import total_ordering.**
b. **Udekoruj klasę Gracz:** Umieść **@total_ordering** bezpośrednio nad **class Gracz:**.
c. **Sprawdź implementację:** Upewnij się, że klasa **Gracz** wciąż ma zdefiniowane metody **__eq__** (porównanie po imieniu) i **__lt__** (porównanie po HP).
d. **Przetestuj:**
    - Stwórz dwóch graczy z różnymi wartościami HP.
    - Sprawdź działanie **wszystkich** operatorów porównania: **==, !=, <, <=, >, >=.**
    - Zaobserwuj, że działają, mimo że nie napisaliśmy kodu dla **>, <=,** itd.

Omówienie Zadania 5: Pełne Porównywanie z @total_ordering

Cel: Analiza rozwiązania i utrwalenie, jak dekoratory mogą automatycznie generować kod.

Zadanie 6 - Emulacja kontenerów

Problem: Mamy klasę **Ekwipunek**, która przechowuje przedmioty.

Dostęp do nich jest mało elegancki: **ekwipunek.dodaj_przedmiot(...).**

Chcielibyśmy pisać: **ekwipunek["miecz"] = ...** lub **len(ekwipunek).**

Rozwiązanie: Protokół Kontenera - zestaw metod specjalnych, które pozwalają obiektowi naśladować zachowanie kontenerów.

- **__len__(self)** -> **len(obj)**
- **__getitem__(self, klucz)** -> **obj[klucz]** (odczyt)
- **__setitem__(self, klucz, wartosc)** -> **obj[klucz] = wartosc** (zapis)
- **__delitem__(self, klucz)** -> **del obj[klucz]** (usunięcie)

Zadanie: Stwórz klasę **Ekwipunek**, która będzie zachowywać się jak słownik.

a. W konstruktorze **__init__** stwórz prywatny słownik self._przedmioty = {}.

b. Zaimplementuj **__len__,** aby zwracała liczbę przedmiotów.

c. Zaimplementuj **__setitem__,** aby dodawała parę klucz-wartość do wewnętrznego słownika.

d. Zaimplementuj **__getitem__,** aby pobierała wartość dla danego klucza.

e. Zaimplementuj **__delitem__,** aby usuwała przedmiot o danym kluczu.

f. (Opcjonalnie) Zaimplementuj **__repr__,** aby ładnie wyświetlać zawartość ekwipunku.

g. Przetestuj: Stwórz obiekt **Ekwipunek**, dodaj przedmioty za pomocą składni **[]**, odczytaj je, sprawdź długość za pomocą **len()** i usuń jeden z przedmiotów.

Zadanie 6 - Emulacja kontenerów

Cel: Analiza rozwiązania i utrwalenie, jak metody specjalne mapują się na składnię języka.

Zadanie 7: Własne iteratory

Problem: Nasz Ekwipunek jest iterowalny dzięki **__getitem__,** ale to niejawne. Jak stworzyć własny, kontrolowany proces iteracji?

Rozwiązanie: Protokół Iteratora - obiekt, który wie, jak przechodzić po kontenerze.

1. **Kontener (Iterable):** Implementuje **__iter__(self),** która zwraca iterator.
2. **Iterator:**
   - Implementuje **__iter__(self),** która zwraca self.
   - Implementuje **__next__(self),** która zwraca kolejny element lub podnosi **StopIteration**.

Zadanie: Zmodyfikuj klasę Ekwipunek, aby miała własny, jawny iterator.

a. **Stwórz klasę iteratora:** Stwórz nową klasę **IteratorEkwipunku**.

b. W jej konstruktorze **__init__** przyjmij listę przedmiotów i zainicjuj licznik pozycji na **0**.

c. W **IteratorEkwipunku** zaimplementuj **__next__.** Ma ona zwracać kolejny przedmiot z listy i zwiększać licznik. Gdy licznik przekroczy długość listy, podnieś **StopIteration**.

d. W **IteratorEkwipunku** zaimplementuj **__iter__,** która zwraca **self**.

e. **Zmodyfikuj Ekwipunek:** W klasie **Ekwipunek** zaimplementuj metodę **__iter__,** która tworzy i **zwraca nową instancję IteratorEkwipunku**, przekazując mu listę swoich przedmiotów.

f. **Przetestuj:** Stwórz obiekt **Ekwipunek**, dodaj przedmioty. Użyj pętli **for** i zaobserwuj, że działa. Spróbuj też ręcznie użyć iteratora: **it = iter(plecak), next(it).**

Omówienie Zadania 7: Własne iteratory

Cel: Analiza rozwiązania i utrwalenie różnicy między obiektem iterowalnym a iteratorem.

Zadanie 8: Mutowalny Atrybut Klasy

Cel: Zrozumienie, jak Python obsługuje atrybuty klas i jak unikać niezamierzonego współdzielenia stanu.

Problem: Co się stanie, gdy klasa potomna zmodyfikuje mutowalny atrybut (np. listę) zdefiniowany w klasie bazowej?

Zadanie: Przeanalizuj kod, a następnie uruchom go i sprawdź swoje przewidywania.

```
class Bron:
    # Atrybut KLASY, nie instancji!
    dostepne_ulepszenia = []

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

    def dodaj_ulepszenie(self, ulepszenie):
        self.dostepne_ulepszenia.append(ulepszenie)

    def __repr__(self):
        return f"{self.nazwa} (ulepszenia: “
               f”{self.dostepne_ulepszenia})"

class Miecz(Bron): pass
class Topor(Bron): pass

miecz = Miecz("Stalowy Miecz")
topor = Topor("Krasnoludzki Topór")

print(f"Przed modyfikacją: {miecz}, {topor}")

# Dodajemy ulepszenie TYLKO do miecza
miecz.dodaj_ulepszenie("Ostrzenie")

# Pytanie: Co zostanie wyświetlone poniżej?
print(f"Po modyfikacji: {miecz}, {topor}")
```

Omówienie Zadania 8: Mutowalny Atrybut Klasy

Wynik: Ulepszenie dodane do miecz pojawiło się też w topor!

```
Po modyfikacji: Stalowy Miecz (ulepszenia: ['Ostrzenie']), Krasnoludzki Topór (ulepszenia: ['Ostrzenie'])
```

- Atrybut klasy **dostepne_ulepszenia = []** jest tworzony tylko raz, gdy Python wczytuje definicję klasy **Bron**.
- Wszystkie instancje Bron oraz jej podklas (**Miecz**, **Topor**) **współdzielą tę samą listę w pamięci.**
- Modyfikując listę poprzez jeden obiekt (**miecz**), modyfikujemy ją dla wszystkich.

Zadanie 9 - Optymalizacja: __slots__

Problem:

- Domyślnie każdy obiekt przechowuje swoje atrybuty w słowniku **__dict__.** To elastyczne, ale zużywa pamięć.
- Możemy też dynamicznie dodawać nowe atrybuty (**obj.nowy = 1**), co może prowadzić do błędów (np. literówek).

Rozwiązanie: __slots__

- Specjalna deklaracja w klasie, która rezerwuje stałą ilość miejsca na atrybuty.
- Blokuje tworzenie __dict__ i uniemożliwia dodawanie nowych atrybutów w locie.

Zadanie: Stwórz "lekką" klasę, która nie będzie posiadała __dict__.

a. Stwórz klasę **PunktLekki**.

b. Dodaj do niej deklarację: **__slots__ = ('x', 'y').**

c. Zaimplementuj prosty konstruktor **__init__,** który ustawia **x** i **y**.

d. Przetestuj:

- Stwórz instancję: **p = PunktLekki(1, 2).**
- Sprawdź, czy możesz odczytać atrybuty: **print(p.x)**.
- Spróbuj dodać nowy atrybut: **p.z = 3**. Zaobserwuj **AttributeError**.
- Spróbuj odwołać się do **__dict__: print(p.__dict__)**. Zaobserwuj **AttributeError**.

Omówienie Zadania 9 - Optymalizacja: __slots__

Korzyści z __slots__:

- **Oszczędność pamięci:** Kluczowe przy tworzeniu milionów małych obiektów.
- **Szybszy dostęp do atrybutów.**
- **Bezpieczeństwo:** Chroni przed literówkami i przypadkowym dodawaniem atrybutów.

Koszt: Utrata elastyczności (brak **__dict__** i dynamicznych atrybutów).

Podsumowanie

Co osiągnęliśmy w trakcie dwóch laboratoriów?

- **Fundamenty:** Stworzyliśmy klasy (__init__, __repr__), zbudowaliśmy hierarchię (dziedziczenie, super()) i zabezpieczyliśmy dane (@property).
- **Integracja z językiem:** Nauczyliśmy nasze obiekty reagować na operatory (__eq__, __add__), zachowywać się jak kontenery (__len__) i iterować (__iter__).
- **Nowoczesny kod:** Używaliśmy dataclasses i @total_ordering, aby pisać mniej powtarzalnego kodu.
- **Solidność i elegancja:** Stworzyliśmy własne wyjątki, fabryki (@classmethod) i poznaliśmy zaawansowane pułapki oraz optymalizacje (__slots__).

Gdzie jesteśmy? Potrafimy tworzyć własne, w pełni funkcjonalne typy danych, które elegancko integrują się z językiem Python.

Co dalej? (Wykład W4)

- Znamy już **mechanikę** OOP.
- Następny krok: **Architektura Obiektowa** (Wzorce Projektowe, SOLID).