Wprowadzenie do Programowania Obiektowego

Cel zajęć: Przełożenie teoretycznych koncepcji OOP poznanych na wykładzie na praktyczne, działające klasy i obiekty w Pythonie.

Plan Działania:

Zadanie 1: Od słownika do obiektu (class, __init__).

- Stworzenie pierwszej klasy **Gracz**.
- Implementacja konstruktora i atrybutów instancji.

Zadanie 2: Obiekty, które "mówią" i "działają" (__str__, __repr__).

- Dodanie metod (**pokaz_status**, **otrzymaj_obrazenia**).
- Implementacja metod specjalnych dla czytelnego wyświetlania.

Zadanie 3: Właściwości wspólne i indywidualne (atrybuty klasy).

- Zastosowanie atrybutu klasy do zliczania wszystkich stworzonych graczy.

Zadanie 4: Hierarchia i specjalizacja (Dziedziczenie, super()).

- Stworzenie klas potomnych **Wojownik** i **Mag**.
- Użycie **super()** i nadpisywanie metod.

Zadanie 5: Polimorfizm w akcji.

- Przetwarzanie listy obiektów różnych typów w jednej pętli.

Zadanie 6: Enkapsulacja w praktyce (@property).

- Refaktoryzacja atrybutu hp z użyciem właściwości.

Zadanie 7: Kompozycja (has-a).

- Stworzenie klasy **Ekwipunek** i dodanie jej do **Gracza**.

Zadanie 8 i 9: Zaawansowane Metody.

- Metody klasowe **(@classmethod)** i statyczne **(@staticmethod).**
- Przeciążanie operatorów **(__eq__, __add__).**

Zadanie 1: Od słownika do obiektu

Problem: W podejściu proceduralnym dane i logika są oddzielone. To prowadzi do problemów ze spójnością i organizacją kodu.

Kod "Przed" (Podejście proceduralne):

```
# Dane i logika są oddzielone
gracz1 = {"imie": "Aragorn", "hp": 100}

def pokaz_status(gracz):
    print(f"Gracz: {gracz['imie']}, HP: {gracz['hp']}")

pokaz_status(gracz1)
```

Cel: Zastąpić słownik i luźną funkcję spójną klasą **Gracz.**

Zadanie:

a. Stwórz klasę **Gracz** (pamiętaj o konwencji **PascalCase).**

b. Zdefiniuj w niej konstruktor **__init__,** który przyjmie **imie** i **hp** jako argumenty.

c. Wewnątrz konstruktora, przypisz otrzymane wartości do atrybutów instancji (**self.imie i self.hp**).

d. Stwórz instancję (obiekt) swojej nowej klasy **Gracz,** np. **gracz_obj = Gracz("Aragorn", 100).**

e. Sprawdź, czy możesz uzyskać dostęp do atrybutów obiektu, np. **print(gracz_obj.imie).**

Omówienie Zadania 1 - Tworzenie Pierwszej Klasy

Cel: Analiza poprawnego rozwiązania i utrwalenie kluczowych koncepcji: **class, __init__, self.**

Co dalej? Nasz obiekt przechowuje dane, ale nie potrafi się ładnie "przedstawić" ani niczego "zrobić". Czas dodać mu zachowanie (metody) i inteligencję (**__str__, __repr__**).

Zadanie 2 - Obiekty, które "mówią" i "działają"

Problem: Nasz obiekt **Gracz** jest "niemy" (brzydko się drukuje) i "bierny" (nic nie potrafi zrobić).

Cel: Dodać do klasy **Gracz** zachowanie (metody) oraz "inteligencję" (metody specjalne **__str__ i __repr__).**

Zadanie: Rozbuduj swoją klasę **Gracz** z poprzedniego zadania:

a. **Dodaj metodę pokaz_status(self):**
   - Metoda ta nie przyjmuje żadnych argumentów poza **self**.
   - Ma wyświetlić w konsoli sformatowany komunikat, np. "**Gracz: Aragorn, HP: 100**".

b. **Dodaj metodę otrzymaj_obrazenia(self, ilosc):**
   - Metoda ta modyfikuje stan obiektu.
   - Zmniejsza atrybut **hp** o podaną **ilosc**.
   - Wyświetla komunikat o otrzymanych obrażeniach.

c. **Zaimplementuj metodę specjalną __str__(self):**
   - Ta metoda jest wywoływana przez **print().**
   - Ma zwrócić (a nie drukować!) string, który będzie przyjazną dla użytkownika reprezentacją obiektu, np. **"Gracz Aragorn (HP: 100)".**

d. **Zaimplementuj metodę specjalną __repr__(self):**
   - Ta metoda jest dla programistów (np. do debugowania).
   - Ma zwrócić string, który jest jednoznaczną reprezentacją obiektu, np. "Gracz(imie='Aragorn', hp=100)".

e. **Przetestuj:** Stwórz obiekt, wywołaj na nim **print(),** użyj nowych metod i ponownie wywołaj **print(),** aby zobaczyć, jak zmienił się jego stan.

Omówienie Zadania 2: Metody i Reprezentacja Obiektu

Cel: Analiza poprawnego rozwiązania, utrwalenie roli metod oraz różnicy między **__str__** a **__repr__.**

Kluczowe wnioski:

- **Metody** to funkcje "należące" do obiektu, które operują na jego danych (**self**).
- **__str__ i __repr__** muszą zwracać (**return**) string, a nie go drukować.
- Nasz obiekt jest teraz kompletną "kapsułką" łączącą **dane** (atrybuty) i **zachowanie** (metody).

Zadanie 3 - Właściwości wspólne i indywidualne (atrybuty klasy)

Problem: Jak śledzić dane, które są wspólne dla wszystkich obiektów danej klasy? Np. ilu graczy jest w grze?

Rozwiązanie: Atrybut Klasy

- **Atrybut instancji (self.hp):** Unikalny dla każdego obiektu.
- **Atrybut klasy (Gracz.liczba_graczy):** Współdzielony przez wszystkie obiekty. Definiowany bezpośrednio w klasie.

Zadanie: Zmodyfikuj klasę **Gracz**, aby automatycznie zliczała, ile obiektów tego typu zostało stworzonych.

a. **Dodaj atrybut klasy:** W ciele klasy **Gracz** (ale poza **__init__),** dodaj atrybut liczba_graczy i zainicjalizuj go wartością **0.**

b. **Zmodyfikuj konstruktor:** W metodzie **__init__, po** przypisaniu atrybutów instancji, dodaj linię, która zwiększa atrybut klasy o 1.
   - **Wskazówka:** Odwołuj się do atrybutu klasy przez jej nazwę, np. **Gracz.liczba_graczy += 1.**

c. **Przetestuj:**
   - Wyświetl wartość **Gracz.liczba_graczy** przed stworzeniem jakiegokolwiek obiektu.
   - Stwórz kilka obiektów **Gracz**.
   - Ponownie wyświetl wartość **Gracz.liczba_graczy**, aby zobaczyć, czy licznik działa.a

Omówienie Zadania 3: Atrybuty Klasy (Wspólne vs. Indywidualne)

Cel: Analiza poprawnego rozwiązania i utrwalenie różnicy między atrybutem klasy a atrybutem instancji.

Zadanie 4: Hierarchia i specjalizacja (Dziedziczenie)

Problem: Chcemy stworzyć różne typy graczy (**Wojownik, Mag**), które mają wspólne cechy (imię, hp), ale też unikalne. Jak uniknąć kopiowania kodu z klasy **Gracz**?

Rozwiązanie: Dziedziczenie - tworzenie klas potomnych, które przejmują cechy klasy bazowej.

- **Relacja "jest": Wojownik** jest **Graczem.**

Zadanie:

a. Stwórz nową klasę **Wojownik**, która **dziedziczy** po klasie **Gracz**. Składnia: **class Wojownik(Gracz):.**

b. Zdefiniuj konstruktor **__init__** dla **Wojownika**. Powinien przyjmować **imie**, **hp** oraz dodatkowy atrybut **sila**.

c. Wewnątrz konstruktora **Wojownika**, użyj **super().__init__(imie, hp)**, aby wywołać konstruktor klasy **Gracz** i zainicjować wspólne atrybuty. Nie kopiuj kodu!

d. Po wywołaniu **super(),** dodaj nowy, unikalny atrybut **self.sila = sila.**

e. **Nadpisz (override)** metodę **przedstaw_sie** w klasie **Wojownik**, aby wyświetlała również informację o sile. *Wskazówka: możesz użyć* ***super().przedstaw_sie()*** *aby rozszerzyć, a nie zastąpić, zachowanie rodzica.*

f. Dodaj nową metodę **atak(self)** tylko w klasie **Wojownik**, która wyświetli komunikat o ataku z użyciem siły.

g. **Przetestuj:** Stwórz obiekt **Wojownik**, wywołaj na nim odziedziczone i nowe metody.

Omówienie Zadania 4: Dziedziczenie i Specjalizacja

Cel: Analiza poprawnego rozwiązania, utrwalenie koncepcji dziedziczenia, super() i nadpisywania metod.

Zadanie 5 - Polimorfizm w akcji

Problem: Jak napisać kod, który będzie działał z różnymi, ale powiązanymi obiektami (**Gracz**, **Wojownik**), nie sprawdzając za każdym razem ich typu za pomocą **if/elif/else**?

Rozwiązanie: Polimorfizm (z gr. "wielopostaciowość") i **Duck Typing**.

- Nie obchodzi nas, jakiego **typu** jest obiekt.
- Obchodzi nas tylko to, czy **ma metodę**, którą chcemy wywołać.

Zadanie:

a. Stwórz nową klasę **Mag**, która również dziedziczy po klasie **Gracz**.

b. W konstruktorze **Maga**, używając **super(),** zainicjuj **imie** i **hp**, a także dodaj nowy, unikalny atrybut **mana**.

c. **Nadpisz** metodę **przedstaw_sie** w klasie **Mag**, aby wyświetlała również informację o manie.

d. Stwórz listę o nazwie **druzyna**, która będzie zawierać obiekty różnych typów: jednego zwykłego **Gracza**, jednego **Wojownika** i jednego **Maga**.

e. Napisz **jedną pętlę for**, która przejdzie przez listę **druzyna**.

f. Wewnątrz pętli, dla każdej **postaci**, wywołaj jej metodę **przedstaw_sie().** Zaobserwuj, jak każdy obiekt reaguje inaczej na to samo polecenie.

Omówienie Zadania 5: Polimorfizm w Akcji

Cel: Analiza rozwiązania i utrwalenie, jak polimorfizm pozwala pisać elastyczny i rozszerzalny kod.

Zadanie 6 - Enkapsulacja w praktyce (@property)

Problem: Nasze atrybuty są "publiczne". Każdy może je zmienić z zewnątrz, nawet na bezsensowną wartość: **gracz1.hp = -999**.

Rozwiązanie: Właściwości (@property) - "inteligentne" atrybuty, które pozwalają kontrolować dostęp do wewnętrznego stanu obiektu, zachowując prosty interfejs.

Zadanie: Zrefaktoryzuj klasę **Gracz**, aby chronić atrybut **hp**.

a. W klasie **Gracz**, zmień nazwę atrybutu **self.hp** na **self._hp** w konstruktorze. To jest konwencja oznaczająca atrybut "chroniony".

b. Stwórz nową metodę **hp(self),** która po prostu zwraca **self._hp**.

c. Udekoruj tę metodę za pomocą **@property**. To jest nasz **getter**.

d. Stwórz drugą metodę, również o nazwie **hp(self, nowa_wartosc).**

e. Udekoruj ją za pomocą **@hp.setter**. To jest nasz setter.

f. Wewnątrz settera, zaimplementuj logikę: jeśli **nowa_wartosc** jest ujemna, ustaw **self._hp** na **0**. W przeciwnym razie, ustaw **self._hp** na **nowa_wartosc**.

g. **Przetestuj:** Stwórz obiekt **Gracz**, spróbuj przypisać mu ujemne HP (**gracz.hp = -50**) i sprawdź, czy jego HP faktycznie wynosi **0**.

Omówienie Zadania 6 - Enkapsulacja w praktyce (@property)

Cel: Analiza rozwiązania i utrwalenie, jak właściwości pozwalają kontrolować dostęp do danych, zachowując czysty interfejs.

Zadanie 7 - Kompozycja (Relacja "ma")

Problem: Nasz **Gracz** staje się skomplikowany. Co, jeśli jego ekwipunek potrzebuje własnej logiki (np. limit wagi)? Wpychanie tego do klasy **Gracz** złamałoby Zasadę Jednej Odpowiedzialności.

Rozwiązanie: Kompozycja - budowanie złożonych obiektów z mniejszych, niezależnych obiektów.

- **Dziedziczenie (relacja "jest"): Wojownik jest Graczem.**
- **Kompozycja (relacja "ma"): Gracz ma Ekwipunek.**

Zadanie: Zrefaktoryzuj kod, aby Gracz składał się z obiektu Ekwipunek.

a. Stwórz nową, prostą klasę **Ekwipunek.**

b. W jej konstruktorze **__init__** stwórz atrybut przedmioty jako pustą listę.

c. Dodaj do niej metodę **dodaj_przedmiot(self, przedmiot),** która dodaje przedmiot do listy.

d. Dodaj metodę **pokaz_przedmioty(self),** która wyświetla zawartość ekwipunku.

e. W klasie **Gracz,** w konstruktorze **__init__,** usuń atrybut **self.ekwipunek = []** (jeśli go miałeś/aś).

f. Zamiast tego, stwórz tam instancję nowej klasy: **self.ekwipunek = Ekwipunek().**

g. **Przetestuj:** Stwórz obiekt **Gracz,** a następnie dodaj mu przedmioty, wywołując metodę na jego atrybucie ekwipunku, np. **gracz.ekwipunek.dodaj_przedmiot("Miecz").**

Omówienie Zadania 7: Kompozycja (Relacja "ma")

Cel: Analiza rozwiązania i utrwalenie, jak kompozycja pomaga tworzyć elastyczne i łatwe w utrzymaniu klasy.

Zadanie 8: Metody Klasy i Statyczne

Problem:

- Jak stworzyć alternatywny "konstruktor" (np. do wczytywania z pliku)?
- Gdzie umieścić funkcję pomocniczą, która jest logicznie związana z klasą, ale nie potrzebuje dostępu do danych obiektu (**self**)?

Rozwiązanie:

- **@classmethod**: Metoda, która jako pierwszy argument otrzymuje **klasę (cls),** a nie **instancję (self).** Idealna do tworzenia "fabryk" obiektów.
- **@staticmethod:** Zwykła funkcja "zamknięta" w przestrzeni nazw klasy. Nie otrzymuje ani **self**, ani **cls**. Idealna do funkcji pomocniczych.

Zadanie:

a. **Fabryka Wojowników:** W klasie **Wojownik**, stwórz **@classmethod** o nazwie **stworz_berserkera(cls, imie).** Metoda ta ma zwracać nową instancję **Wojownika** ze stałymi wartościami, np. **hp=80**, **sila=40**.

b. **Walidator Imienia:** W klasie **Gracz**, stwórz **@staticmethod** o nazwie **sprawdz_poprawnosc_imienia(imie).** Metoda ta ma zwracać **True**, jeśli imię nie jest puste i zaczyna się wielką literą, w przeciwnym razie **False**.

c. **Przetestuj:**

- Stwórz berserkera, wywołując metodę na klasie: **berserker = Wojownik.stworz_berserkera("Olaf").**
- Sprawdź, czy działa, wywołując na nim przedstaw_sie().
- Sprawdź działanie walidatora: **Gracz.sprawdz_poprawnosc_imienia("Aragorn").**

Omówienie Zadania 8 - 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 9: Porównywanie i Dodawanie Obiektów

Problem: Nasze obiekty nie potrafią się porównywać (==) ani dodawać (+). Jak nauczyć je zachowywać się jak wbudowane typy?

Rozwiązanie: Metody specjalne. Python wywołuje je "pod spodem", gdy używamy standardowych operatorów.

- **__eq__(self, other):** Wywoływana przy **self == other**.
- **__add__(self, other):** Wywoływana przy **self + other.**

Zadanie: Rozbuduj klasę **Gracz** (i/lub **Wojownik**).

a. Porównywanie (__eq__):

- W klasie **Gracz**, zaimplementuj metodę **__eq__(self, other).**
- Metoda powinna zwracać **True**, jeśli **other** jest również obiektem klasy **Gracz** i ma takie samo **imie**. W przeciwnym razie **False**.
- *Wskazówka: Użyj **isinstance(other, Gracz)** do sprawdzenia typu.*

b. Dodawanie (__add__):

- W klasie **Wojownik**, zaimplementuj metodę **__add__(self, other).**
- "Dodanie" dwóch wojowników powinno tworzyć **nowego** wojownika-fuzję.
- Nowy wojownik powinien mieć:
  - **imie**: połączone imiona (np. "Aragorn i Boromir")
  - **hp**: suma hp obu wojowników
  - **sila**: suma siły obu wojowników
- Metoda musi **zwrócić (return)** ten nowy obiekt.

c. Przetestuj:

- Stwórz dwa identyczne obiekty **Gracz** i sprawdź, czy == działa poprawnie.
- Stwórz dwa obiekty **Wojownik** i "dodaj" je do siebie, a następnie przedstaw nowo powstałą postać.

Omówienie Zadania 9: Porównywanie i Dodawanie Obiektów

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

Podsumowanie

Co dzisiaj osiągnęliśmy?

- **Od słownika do klasy:** Zastąpiliśmy luźne struktury spójnymi obiektami (**class**, **__init__, self**).
- **Ożywione obiekty:** Nauczyliśmy je "mówić" **(__str__, __repr__)** i "działać" (metody).
- **Hierarchia i specjalizacja:** Zbudowaliśmy rodzinę klas (**Gracz**, **Wojownik**, **Mag**) za pomocą **dziedziczenia**.
- **Elastyczność:** Wykorzystaliśmy **polimorfizm**, aby traktować różne obiekty w jednolity sposób.
- **Bezpieczeństwo:** Zabezpieczyliśmy stan obiektu za pomocą **enkapsulacji (@property**).
- **Architektura:** Zobaczyliśmy, jak **kompozycja** pozwala budować złożone obiekty z prostszych.

Gdzie jesteśmy? Potrafimy modelować proste systemy za pomocą klas, implementując kluczowe zasady programowania obiektowego.

Zadania do samodzielnej:

- Rozbudowa hierarchii klas o Łucznika.
- Implementacja walki między dwoma obiektami.
- Stworzenie bardziej złożonej kompozycji (np. Gracz ma Zadanie).