Instrukcja 4

Wprowadzenie do obiektowości

Podstawy tworzenia gier

Programowanie zorientowane obiektowo

Programowanie Zorientowane Obiektowo (object-oriented programming - OOP) to sposób projektowania programu (paradygmat programowania), który skupia się na tworzeniu powtarzalnych, łatwych do zarządzania modułów kodu. Jest to jedna z najpopularniejszych metod projektowania i strukturyzowania kodu w całej dziedzinie programowania.

Podstawowymi jednostkami w Programowaniu Zorientowanym Obiektowo są klasy i obiekty.

Klasa stanowi "szablon", który określa budowę i parametry obiektów, utworzonych bazując na kodzie "szablonu". Obiekty są konkretnymi instancjami klasy. Są one niezależne od siebie, jednak cechują się wspólnymi właściwościami zdefiniowanymi w klasie ("szablonie" pozwalającym tworzyć obiekty).

Przykład

Stosowanie klas i obiektów pozwala na znaczne zmniejszenie ilości napisanego kodu. Wyobraź sobie, że masz do napisania grę, w której będą brać udział różni gracze. Bardzo szybko zauważysz, że Twój kod zaczyna się powtarzać - dla każdego gracza utworzysz przecież takie same zmienne (np. zapisujące położenie na ekranie, liczbę punktów, poziom, itp.). Co więcej, niektóre ze zmiennych będą miał również taką samą wartość (np. każdy z graczy rozpocznie grę z 0 punktów i poziomem 1). Różnice będą występowały tylko w wartościach niektórych zmiennych, charakteryzujących danego użytkownika, takich jak nazwa gracza lub jego kolor.

Możemy powiedzieć, że wszyscy gracze są tworzeni za pomocą jednego szablonu. Klasa Player mogłaby zdefiniować wszystkie zmienne opisujące danego gracza (np. nazwę, kolor, położenie, liczbę punktów i poziom), a także wspólne funkcje (np. zastosowanie tej samej funkcji sprawdzania kolizji, zamiast napisania jej dla każdego z graczy z osobna). Klasa pozwala na tworzenie obiektów. W przypadku gry, obiektami klasy Player będą konkretni gracze, np. player1, player2, itd. Każdy z nich otrzyma inną nazwę, inny kolor i położenie, jednak cały kod do tego konieczny zostanie utworzony za pomocą "szablonu" z klasy Player. Konstruktor pozwoli na utworzenie obiektu poprzez podanie tylko określonych parametrów konkretnego gracza, a pozostała część zostanie wykonana przez klasę.

Właściwości klas

Przypomnij sobie strukturę gry Minecraft. W budowie świata najważniejszą rolę odgrywają bloki. Napisanie kodu dla każdego z bloków w grze byłoby praktycznie niewykonalne. Nie istniałaby również możliwości "postawienia" nowego bloku, ponieważ kod z góry definiowałby położenie i ilość bloków dostępnych na mapie (możliwe byłoby tylko ich "przestawienie"). Obiektowość pozwala rozwiązać te problemy.

Klasy i obiekty

Klasą w grze Minecraft może być "przepis" na określony rodzaj bloku. Na przykład klasa "Blok Drewna" zdefiniuje, w jaki sposób ma wyglądać blok drewna, ile ma wytrzymałości, jakie będzie jego zachowanie (np. Czy działa na niego grawitacja? Czy niszczy go wybuch? Czy jest odporny na wodę? Czy emituje sygnał Redstone?) itp. Utworzone zostaną również zmienne dotyczące położenia (według współrzędnych ze świata X, Y, Z, a w przypadku bloku drewna również jego obrót). Zmienne te zostaną zadeklarowane, jednak klasa nie przypisze im żadnej domyślnej wartości - zrobi to dopiero konstruktor określonego obiektu.

Obiektem klasy "Blok Drewna" będzie konkretny blok drewna postawiony w świecie gry. W grze może istnieć wiele bloków drewna, a każdy z nich zostanie utworzony na podstawie szablonu z klasy "Blok Drewna". Podczas "stawiania" bloku, w kodzie zostanie uruchomiony konstruktor, będącą funkcją tworzącą obiekt. Konstruktor wykorzysta koordynaty w celu zapisania docelowego położenia "stawianego" bloku, a pozostałe wartości (np. teksturę) pozostawi domyślne.

Obiekt odnosi się do pojedynczej instancji należącej do danej klasy. W powyższym przykładzie, typ bloku jest klasą, a bloki drewna znajdujące się w świecie - obiektami danej klasy.

Dziedziczenie

W grze Minecraft niektóre bloki posiadają swoje warianty. W przypadku drewna, bloki mogą mieć różne tekstury: dębu, brzozy, świerka, itd. Zwróć uwagę, że wszystkie właściwości bloków pozostają takie same - czas rozbijania bloku będzie taki sam, niezależnie od tego czy jest to dąb lub akacja (oczywiście zakładając użycie takiego samego narzędzia). Nie ma jednak konieczności tworzenia osobnych klas dla każdego z bloków drewna.

Programowanie Zorientowane Obiektowo umożliwia zastosowanie "dziedziczenia". Jest to utworzenie nowej klasy, która bazuje na klasie już istniejącej. Nowa klasa "odziedziczy" elementy klasy bazowej (takie jak jej zmiennie i funkcje), ale może też wprowadzić w nich modyfikacje lub dodać całkowicie nowe, unikalne właściwości.

Klasa "Blok Drewna" może posłużyć jako klasa bazowa, której parametry zostaną "odziedziczone" przez klasy "Blok Dębu", "Blok Tropikalny", itd.

Klasy pochodne zachowają cechy i właściwości drewna, jednak wprowadzą własną teksturę, inną miniaturkę itemu oraz unikatową nazwę.

Praktyczny przykład - Among Us

Gra Among Us wymaga zaangażowania wielu graczy w tym samym czasie. Niezależnie od typu postaci (Crewmate lub Impostor), podstawowa mechanika wszystkich postaci jest taka sama. W tym celu zastosowana zostanie klasa Player. Dla każdego gracza utworzony zostanie obiekt klasy Player.

```
class Player:
    is_alife = true
```

Skoro klasa to szablon dla obiektów, to każdy obiekt utworzony na powyższym przykładzie będzie pojedynczą instancją klasy Player. Każda instancja klasy Player ma domyślnie ustawioną zmienną is_alife na wartość true, ponieważ podczas tworzenia graczy, każdy z nich jest żywy. Istnieją jednak parametry, które są indywidualne dla każdego z graczy np. kolor lub nazwa użytkownika. Możemy je zdefiniować od razu podczas inicjalizacji nowych obiektów. Służy do tego konstruktor.

```
class Player:
    def __init__(self, name, color, role):
        self.name = name
        self.color = color
        self.role = role
        self.is_alive = True
```

W powyższym przykładzie __init__ oznacza konstruktor klasy Player (init -> inicjalizacja nowego obiektu klasy Player) . Słowo self odnosi się do konkretnej instancji klasy - tej która jest właśnie tworzona.

Parametry klasy to zmienne zdefiniowane w klasie. Przechowują stan obiektu oraz jego właściwości. W powyższym przykładzie każdy obiekt klasy Player posiada swoją własną nazwę (self.name), kolor (self.color), rolę (self.role) oraz status życia (self.is_alive).

UWAGA: Przypomnij sobie w jaki sposób parametry (zmiennie) są przekazywane do funkcji. Konstruktor jest również (rodzajem) funkcji.

W języku Python nie ma konieczności "osobnego" tworzenia zmiennych (tak jak w pierwszym przykładzie kodu ze zmienną is_alife). Zamiast tego, parametry klasy (zmienne) mogą zostać zadeklarowane w konstruktorze. Słowo self pozwala na "rozróżnienie", czy określona nazwa zmiennej należy do tworzonego obiektu klasy lub została przekazana jako parametr konstruktora (w nawiasie powyżej, tak jak w przypadku zmiennych przekazywanych do funkcji).

```
self.name = name
```

Do konstruktora zostaje przekazana nazwa o nazwie name, a następnie jest ona przypisana do konkretnego obiektu klasy i zapisana w self.name.

Utworzony konstruktor pozwala na szybką inicjalizację nowych obiektów klasy Player. Obiekty są tworzone poprzez wywołanie konstruktora klasy, w dokładnie taki sam sposób jak wywołuje się funkcje. Zamiast nazwy funkcji podana zostaje nazwa klasy.

```
player1 = Player("NovaHunter", "Red", "Crewmate")
player2 = Player("CometChaser", "Green", "Crewmate")
player3 = Player("GalacticGem", "Black", "Impostor")
```

Powyższy kod tworzy trzech graczy na podstawie szablonu, którym jest klasa Player. Każdy z nich ma indywidualną nazwę, kolor oraz rolę. Będą oni jednak korzystać z tych samych funkcji, które będą zapisane w klasie.

UWAGA: Utworzony konstruktor posiada 4 zmienne, a podczas inicjalizacji obiektów, do każdego z nich przekazane zostały tylko 3. Jest to spowodowane faktem, że zmienna is_alife otrzymała już w konstruktorze domyślną wartość True, więc nie ma konieczności jej powtarzania. Każdy z graczy na początku jest "żywy".

Obiekty klasy mogą posiadać "wspólne" funkcje, które u każdego z graczy będą działały dokładnie tak samo. Zostają one zdefiniowane wewnątrz klasy. W przypadku obiektowości, funkcje definiujące zachowania obiektów są nazywane metodami.

```
def report_body(self):
        if self.is_alive:
            print(f"{self.name} ({self.color}) has reported a
                  body!")
        else:
            print(f"{self.name} cannot report, they are no
                  longer with us...")
```

Powyższy przykład przedstawia metodę o nazwie report_body. Jej jedynym parametrem jest self, co oznacza odwołanie do własnej instancji obiektu.

```
player2.report_body()
player1.report_body()
```

Metoda może zostać wywołana w taki sam sposób dla każdego z obiektów. Za każdym razem będzie wyświetlała parametry tego obiektu, dla którego ta metoda została wywołana.

Przykład przedstawiony na następnej stronie przedstawia klasę Player z przykładowymi metodami oraz sposobem ich wywołania.

```python
class Player:
    def __init__(self, name, color, role):
        self.name = name
        self.color = color
        self.role = role
        self.is_alive = True

    def report_body(self):
        if self.is_alive:
            print(f"{self.name} ({self.color}) has reported a body!")
        else:
            print(f"{self.name} cannot report, they are no longer with
                  us...")

    def complete_task(self):
        if self.is_alive and self.role == "Crewmate":
            print(f"{self.name} is completing a task.")
        elif not self.is_alive:
            print(f"{self.name} cannot complete tasks, they are no longer
                  with us...")
        else:
            print(f"{self.name} is an Impostor and cannot complete tasks.")

    def be_eliminated(self):
        self.is_alive = False
        print(f"{self.name} has been eliminated.")

# Tworzenie graczy
player1 = Player("NovaHunter", "Red", "Crewmate")
player2 = Player("CometChaser", "Green", "Crewmate")
player3 = Player("GalacticGem", "Black", "Impostor")

# Przykładowe akcje
player1.complete_task()
player2.complete_task()
player1.be_eliminated()
player3.report_body()
player2.complete_task()
```

W grze AmongUs istnieją dwie klasy postaci: Crewmate będący zwykłym graczem (członkiem załogi wykonującym zadania) oraz Impostor, którego zadaniem jest sabotaż. Impostor ma podobne właściwości do zwykłego gracza (taki sam sposób sterowania, właściwości, niektóre metody, itp.), jednak jego możliwości są rozszerzone o dodatkowe funkcje. Obiektowość pozwala na wykorzystanie dziedziczenia do utworzenia nowej klasy Impostor oraz "odziedziczenia" wszystkich parametrów i metod klasy Player.

```
class Impostor(Player):
    def sabotage(self):
        print(f"{self.name} sabotuje stację kosmiczną!")

player4 = Impostor("MoonRider", "Yellow", "Impostor")
player4.sabotage()
player4.report_body()
player4.be_eliminated()
```

W powyższym przykładzie utworzona zostaje klasa Impostor dziedzicząca z klasy Player. Posiada ona wszystkie parametry i metody klasy Player, jednak dodana zostanie do niego nowa metoda o nazwie sabotage. Gracz player4 zostaje utworzony na podstawie tego samego konstruktora co klasa Player (w końcu jest on też "odziedziczony"). Impostor może korzystać z metod klasy Player jak również swojej własnej.

Rozszerzenie

Modyfikacja metod klasy nadrzędnej

Powyższe przykłady posiadały pewne "uproszczenia" w celu obrazowego przedstawienia poszczególnych zagadnień. Dobrym znakiem będzie, jeżeli zauważyłeś, że zastosowanie konstruktora klasy Player w poprzednim przykładzie jest trochę "bez sensu" - gracze posiadają parametr o nazwie role, jednak każdy obiekt klasy Impostor jest "z definicji" Impostorem. Miało to na celu pokazanie, że możliwe jest wykorzystanie konstruktora z klasy nadrzędnej. Mimo tego, nowy obiekt jest klasy Impostor, a nie klasy Player.

Klasy pochodne (takie jak Impostor) mogą odwoływać się do metod klasy nadrzędnej (Player), a także modyfikować i rozszerzać ich zawartość.

OSTRZEŻENIE: W celu lepszego przedstawienia modyfikacji konstruktora, poniższy przykład również nie usuwa roli z klasy Player.

```
class Impostor(Player):
    def __init__(self, name, color):
        super().__init__(name, color, "Impostor")
        self.sabotage_count = 0
```

W powyższym przykładzie utworzony zostaje konstruktor klasy Impostor, będący modyfikacją konstruktora klasy Player. Oznaczenie super() umożliwia umożliwia wywołanie metody _ _init_ _ klasy nadrzędnej Player, przekazując do niej odpowiednie argumenty. Do konstruktora dodany zostaje również parametr sabotage_count, którego wartość domyślnie zostaje ustawiona na zero.

```
player1 = Player("NovaHunter", "Red", "Crewmate")
player2 = Player("CometChaser", "Green", "Crewmate")
player3 = Impostor("GalacticGem", "Black")
player4 = Impostor("MoonRider", "Yellow")
```

Gracze player3 i player4 są inicjalizowani w podobny sposób jak gracze klasy Player, jednak ich rola jest automatycznie ustawiona na "Impostor", posiadają również licznik sabotażu.

Metody specjalne

Język Python poza konstruktorem _ _init_ _ wprowadza również inne metody pozwalające na pracę z obiektami.

```
class Player:
    def __init__(self, name, role):
        self.name = name
        self.role = role
    def __str__(self):
        return f"Player {self.name}, Role: {self.role}"
```

Metoda _ _str_ _ jest wywoływana, gdy obiekt jest konwertowany na ciąg znaków przy pomocy funkcji str() lub jest wyświetlany za pomocą print(). Służy do zwracania czytelnej reprezentacji obiektu, w formie zdefiniowanej przez użytkownika.

```
class Player:
    def __init__(self, name, role):
        self.name = name
        self.role = role

    def __repr__(self):
        return f"Player('{self.name}', ,{self.role}')"
```

Metoda _ _repr_ _ jest bardzo podobna do _ _str_ _. Jest ona stosowana w kontekście debugowania i rozwijania kodu. Powinna zwracać bardziej szczegółową i jednoznaczną reprezentację obiektu, często w formie, która pozwala na dokładne odtworzenie tego obiektu.

```
player = Player("MoonRider")
print(player)              #Metoda __str__
print(repr(player))        #Metoda __repr__
```

Metody _ _rep_ _ i _ _str_ _ mają inne grupy docelowe. _ _str_ _ ma za zadanie dostarczyć informację w sposób przyjazny dla użytkownika końcowego, często nie przekazując wszystkich informacji. _ _repr_ _ jest wykorzystywany do debugowania. Wyświetlenie wszystkich detali dotyczących obiektu pozwala deweloperowi na odtworzenie obiektu w formie 1:1, a przez to dokładniejsze sprawdzenie przypadku w kontekście poszukiwania błędu. Warto uwzględnić w kodzie jedną i drugą opcję.

```
class Inventory:
    def __init__(self, items):
        self.items = items

    def __len__(self):
        return len(self.items)
```

Metoda _ _len_ _ pozwala określić, w jaki sposób obliczyć "długość" lub "rozmiar" obiektu. Pozwala to na używanie funkcji len() bezpośrednio na obiekcie klasy. W powyższym przykładzie, klasa Inventory reprezentującej kolekcję elementów - ekwipunek. Metoda _ _len_ _ pozwala na zwrot informacji ile elementów znajduje się w ekwipunku.

```
my_inventory = Inventory(['sword', 'shield'])
print(len(my_inventory))  #Funkcja zwraca liczbę 2
```

Zadanie

Napisz własną wersję popularnej gry internetowej Agar.io. Wykorzystaj koncepcję obiektowości. Gra powinna obsługiwać minimum 2 użytkowników (np. strzałki i WASD).

Projekt gry:

1. Zdefiniuj klasę bazową, która będzie reprezentowała WSZYSTKIE obiekty rysowane w oknie gry. Do zaimplementowania:
   - Przykładowe parametry, które mogą zostać zdefiniowane w konstruktorze to współrzędne (x i y), rozmiar obiektu (lub punkty) i kolor (losowy lub narzucony "z góry").
   - Metoda odpowiadająca za rysowanie obiektu w oknie gry.
   - Metoda losująca położenie na ekranie.
2. Zdefiniuj klasę gracza, dziedziczącą z klasy bazowej. Do zaimplementowania:
   - Metoda poruszania graczem (weź pod uwagę różne sposoby sterowania, na przykład wybór sposobu w konstruktorze obiektu)
   - Metoda sprawdzająca kolizje z innym graczem i punktami:
     - Zwiększająca rozmiar gracza po "konsumpcji" punktu lub innego gracza (pod warunkiem większego rozmiaru).
     - Zmieniająca położenie gracza oraz resetująca stan punktów (w przypadku kolizji z graczem o większym rozmiarze).
3. Przed pętlą gry:
   - Utwórz obiekty graczy i punktów (do punktów wykorzystaj pętlę, zapisz je w liście (tak, listy mogą zapisywać obiekty, będzie łatwiej sprawdzać kolizje [na przykład za pomocą pętli]).
4. W pętli gry:
   - Sprawdź eventy klawiatury w celu wykonania przesunięcia / zmiany kierunku graczy.
   - Po każdej zmianie położenia powinno nastąpić sprawdzenie kolizji oraz związane z tym akcje.
   - Gracze będą "rosnąć" w nieskończoność, wymyśl coś i rozwiąż ten problem.

ROZSZERZENIE

Zastosuj modyfikację konstruktora klasy bazowej super(). Spraw, aby gra przypominała agar.io i skorzystaj z rysowania okręgów (sprawdzenie kolizji może wymagać obliczenia dystansu między środkami okręgów z twierdzenia Pitagorasa [biblioteka math i math.sqrt]).

DODATKOWA OCENA

Rozbuduj grę według własnego uznania. Im ciekawsza tym lepsza.