Programowanie obiektowe - klasy

Klasy

Zazwyczaj chcemy żeby w naszym kodzie było możliwie mało powtórzeń, całość była jak najbardziej czytelna, a osoby korzystające z naszego programu miały jasne wytyczne czego nasz kod oczekuje.

Jednym z narzędzi w drodze do tego celu są klasy, będące fundamentem podejścia nazywanego programowaniem obiektowym.

Korzystając z tej metody staramy się modelować nasz kod w sposób nieco przypominający to jak zbieramy informacje o otaczającym nas świecie. Np. grupujemy gatunki w podobne kategorie. Pies domowy należy do rodziny psowatych, które z kolei należą do rzędu ssaków drapieżnych.

Mimo tego, że psy różnią się od siebie (ciężko pomylić jamnika z dobermanem) to mają one pewne cechy wspólne, które umożliwiają nam przypisanie ich do tego typu (gatunku).

Podobnie modelując aplikację staramy się znaleźć jakieś elementy wspólne obiektów, które tworzymy i na tej podstawie stworzyć typ grupujący je - służy on jako swego rodzaju szablon do stworzenia nowych obiektów.

Oprócz cech charakterystycznych (atrybutów) klasa zawiera też metody - czyli rzeczy, które dany obiekt może robić.

Jako przykład posłużmy się postacią do gry komputerowej rpg. Każda z nich będzie miała jakieś punkty życia, punkty magii, profesję itd. Żeby nie wymyślać tej struktury za każdym razem możemy zdefiniować odpowiednią klasę:

```
class PostacGracza:
    def __init__(self, imie, profesja):
        self.imie = imie
        self.profesja = profesja
        self.hp = 100
        self.mana = 100
```

```
gracz1 = PostacGracza("Fizban", "mag")
gracz2 = PostacGracza("Taz", "łotrzyk")
```

Obie stworzone postaci (gracz1 i gracz2) zostały zrobione na podstawie tego samego szablonu (klasy `PostacGracza`), ale nie są identyczne - różnią się imieniem i profesją. Co więcej pozostałe ich atrybuty (hp i mana) mimo, że na start mają tę samą wartość są od siebie całkowicie niezależne (są to zupełnie osobne wartości). Czyli np. jeśli w trakcie gry Fizban otrzyma 30 obrażeń to będzie miał on 70 hp, podczas gdy Taz nadal będzie miał 100.

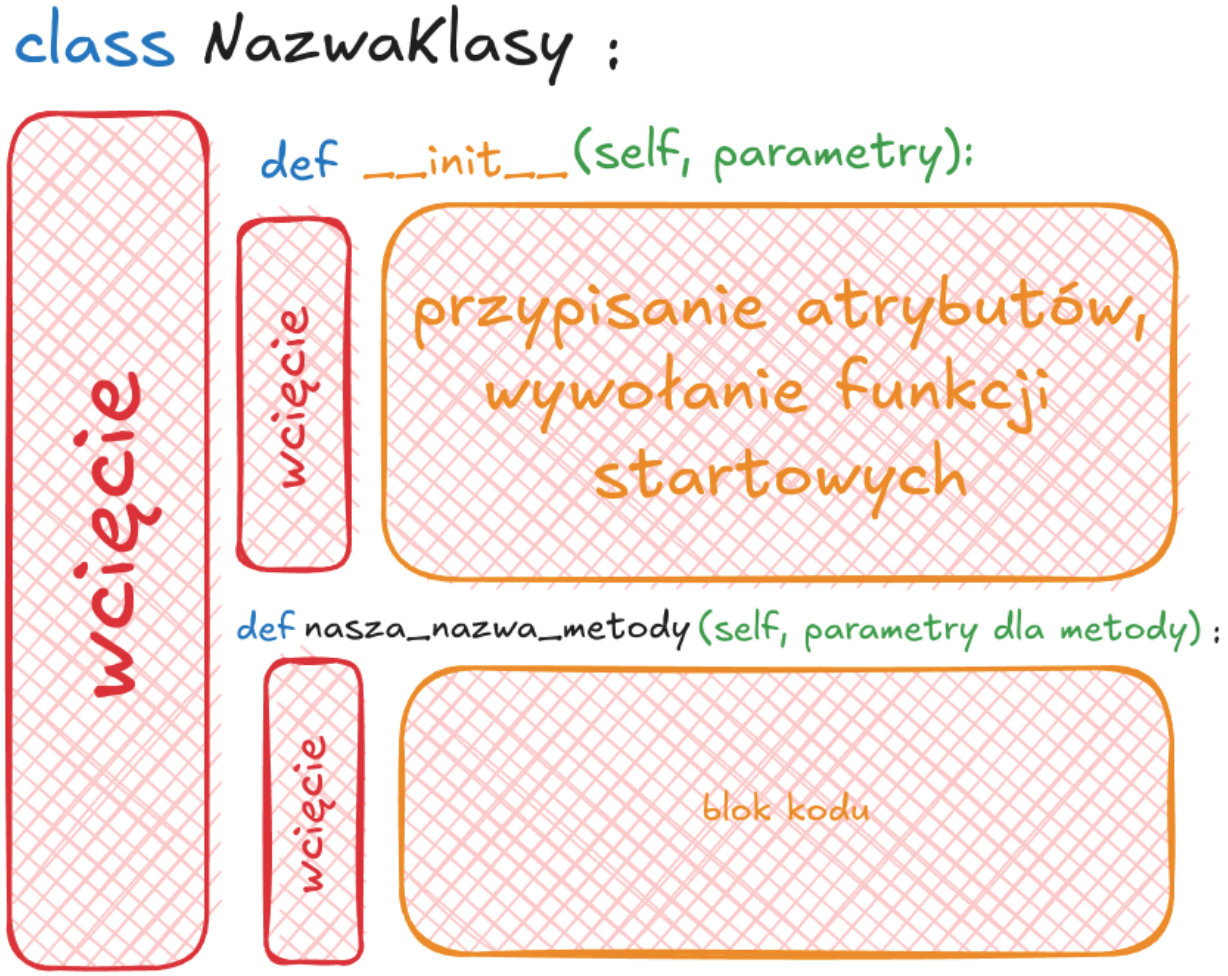
Klasa to nie obiekt

Ważne aby zdawać sobie sprawę, że ten przygotowany "szablon" nie jest równoznaczny z obiektem danego typu. W powyższym przykładzie `PostacGracza` to opis klasy, a `gracz1` to juz obiekt klasy `PostacGracza` ("szablon wypełniony zawartością").

Na tej samej zasadzie np. definicja kota w encyklopedii nie jest naszym kotem Mruczkiem, który właśnie przeszkadza nam w pracy ;) .

Porównując klasy do np. szablonów ankiet, które dajemy komuś do wypełnienia to sam szablon byłby klasą, a uzupełniona, wypełniona przez kogoś ankieta byłaby obiektem.

Tworzenie klasy



Metoda `__init__`

- to sekcja, która jest uruchamiana podczas tworzenia naszego obiektu na podstawie szablonu (klasy). Służy do przypisania obiektowi jego indywidualnych cech (w powyższym przykładzie będzie to np. wpisanie "Fizban" pod pole `imie`), lub wykonanie jakiś czynności na starcie. Czyli mamy "szkielet" i w tym momencie wypełniamy go "mięsem".
- sama definicja tej funkcji przestrzega wcześniejszych zasad np. warto opisywać typy jakie mają być przekazane
- bardzo istotne jest tutaj słowo `self`. Wskazuje one, że przypisujemy indywidualne cechy do tego konkretnego obiektu, nie do całej klasy.

Warto?ci domy?lne

Jeśli do któregoś atrybutu klasy często przypisujemy tę samą wartość możemy oszczędzić użytkownikowi jej wpisywania poprzez zadeklarowanie tej wartości domyślnej. W powyższej klasie mogłoby to wyglądać tak:

```
class PostacGracza:
    def __init__(self, imie: str, profesja: str, hp: int = 100, mana: int = 100):
        self.imie = imie
        self.profesja = profesja
        self.hp = hp
        self.mana = mana

    def powitaj(self):
        print(f"Witaj {self.imie}.")
```

Tym samym chcąc stworzyć nową postać z domyślną wartością dla hp i many wystarczy wpisać:

```
gracz1 = PostacGracza("Taz", "łotrzyk")
```

Oczywiście chcąc dać np. magowi więcej punktów many możemy tę domyślną wartość napisać inną np.

```
gracz2 = PostacGracza("Fizban", "mag", mana=200)
```

Funkcje powi?zane z klas? - metody

- możemy też dodać własne funkcje powiązane z daną klasą, czyli rzeczy, które dany obiekt może robić. Takie funkcje powiązane z klasą nazywamy metodami.
- jeśli chcemy, żeby metoda korzystała z jakiś atrybutów tego konkretnego obiektu (a powinna) ponownie ważne będzie słowo `self`. Jeśli w metodzie nie skorzystamy z atrybutów danej klasy (a tym samym nie pojawi się słowo self wewnątrz niej) to jest dla nas wskazówka, że przypuszczalnie dana funkcja nie jest dobrym materiałem na metodę (i być może dobrze by było zadeklarować ją jako zupełnie osobną funkcję).

```
class PostacGracza:
□def __init__(self, imie, profesja, hp =100, mana = 100):
□□self.imie = imie
□□self.profesja = profesja
□□self.hp = hp
□□self.mana = mana

□def powitaj(self):
□□print(f"Witaj {self.imie}. Twoja profesja to {self.profesja} .")
```

W powyższym przykładzie mamy jedną metodę - `powitanie`. Wykorzystujemy je identycznie jak metody dostarczone razem z klasami dostarczonymi z Pythonem (czyli po nazwie zmiennej

stawiamy kropkę, następnie wpisujemy nazwę metody i korzystamy z nawiasów okrągłych do uruchomienia i ew. przekazania jakiś parametrów). Np. chcąc stworzyć obiekt klasy `PostacGracza` i użyć metody `powitaj` należy np.

```
gracz1 = (imie="Fizban", profesja="mag")


gracz1.powitaj()
```

Dziedziczenie

W poyższym przykładzie z klasą `PostacGracza` jest zauważalny problem - profesja to nic innego jak pole tekstowe. Poza informacją o profesji nie wnosi nic nowego, nie rozszerza możliwości obiektu tej klasy. Czyli np. mag nie dostanie jakiś metod odpowiadających za rzucanie zaklęć, nie dostanie automatycznie większej ilości punktów magii itd.

Jednym z rozwiązań może być zastosowanie mechanizmu tzw. dziedziczenia. Tworzymy nową klasę (np. `Mag`) i oznaczamy, że... odziedziczy ona wszystkie cechy / metody innej (np. `PostacGracza` - tym samym `PostacGracza` jest w tym przykładzie rodzicem :D). Następnie dodajemy nowe atrybuty / metody do tej nowej klasy - tym samym zwiększając dostępne możlwości.

Mogłoby to wyglądać np. tak:

```
class PostacGracza:
□def __init__(self, imie, profesja, hp =100, mana = 100):
□□self.imie = imie
□□self.hp = hp
□□self.mana = mana

□def powitaj(self):
□□print(f"Witaj {self.imie}.")



class Mag(PostacGracza):
    def __init__(self, imie):
        super().__init__(imie, hp=80, mana=200)

    def rzuc_kule_ognia(self):
        print("Rzucono kulę ognia!")
        return 30
```

Klasa Mag rozszerza możliwości dostarczone przez PostacGracza o nową metodę - `rzuc_kule_ognia` oraz zmienia wartości przypisywane do atrybutów hp i mana.

super().__init__()

Funkcja ta służy do aktywowanie atrybutów klasy rodzica. Mówiąc inaczej - służy do uruchominie funkcji `__init__` rodzica. Nie dodajemy w tym wypadku w niej słowa `self` - to jest przyjęte domyślnie.

Alternatywnie możemy zamiast funkcji super użyć nazwy klasy rodzica (np. `PostacGracza`)... ale w tym wypadku nie możemy pominąć słowa self.

```
class Mag(PostacGracza):
    def __init__(self, imie):
        PostacGracza().__init__(self, imie, hp=80, mana=200)

    def rzuc_kule_ognia(self):
        print("Rzucono kulę ognia!")
        return 30
```

Z tego względu sugerowane jest użycie funkcji `super`.

Odno?niki

„3.12.5 Documentation”. Dostęp 18 sierpień 2024. https://docs.python.org/3/.

Wersja #15
Utworzono 2024-10-14 13:25:21 UTC przez Przemek Jeske
Zaktualizowano 2026-04-28 19:29:13 UTC przez Przemek Jeske