Wielodziedziczenie czy kompozycja?

Paweł Gliwny

Dziedziczenie

- **Pozwala na ponowne użycie kodu** między klasami.
- **Klasa potomna** dziedziczy wszystkie funkcjonalności **klasy nadrzędnej**
- Tworzy relację **„jest rodzajem” („is-a”)**.
- Może **rozszerzać funkcjonalność** o dodatkowe:
  - **Atrybuty**
  - **Metody**

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hello, my name is {self.name}")

class Employee(Person):
    def __init__(self, name, age, title):
         Person.__init__(name, age)
         self.title = title

    def charge_position(self, new_title):
         self.title = new_title
```

Konstruktor: Super() zamiast Person()

- `Person().__init__()` – jawne wywołanie metody z klasy bazowej
- `super().__init__()` – dynamiczne wywołanie konstruktora klasy bazowej

```
class Employee(Person):
    def __init__(self, name, age, title):
        Super().__init__(name, age)
        self.title = title

    def charge_position(self, new_title):
        self.title = new_title
```

Przykład

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hello, my name is {self.name}")

class Employee(Person):
    def __init__(self, name, age, title):
        Person.__init__(self, name, age)
        self.title = title
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hello, my name is {self.name}")

class Employee(Person):
    def __init__(self, name, age, title):
        super().__init__(name, age)
        self.title = title
```

Różnica między `NazwaKlasy.__init__()` a `super()` w Pythonie

- **`NazwaKlasy.__init__()`**
  - Wywołuje metodę `__init__()` z klasy nadrzędnej, ale **nie obsługuje wielodziedziczenia**.
  - Musimy ręcznie podać nazwę klasy nadrzędnej.
  - Nie jest dynamiczne – jeśli zmienimy nazwę klasy nadrzędnej, kod wymaga aktualizacji.
- **`super().__init__()`**
  - Automatycznie odwołuje się do klasy nadrzędnej bez potrzeby podawania jej nazwy.
  - Zalecane w nowoczesnym Pythonie, ponieważ ułatwia refaktoryzację i zmiany w hierarchii dziedziczenia.

Wielodziedziczenie w Pythonie

- Klasa może dziedziczyć od wielu klas nadrzędnych.
- Python rozwiązuje konflikty metod za pomocą **MRO (Method Resolution Order)**.
- Kolejność przeszukiwania metod można sprawdzić za pomocą `ClassName.mro()`.
- `super()` w wielodziedziczeniu działa zgodnie z MRO.

```
class Osoba:
    pass

class Pracownik:
    pass

class Menedzer(Osoba, Pracownik):
    pass
```

Wyzwania wielodziedziczenia

- Może prowadzić do **konfliktów nazw metod** (np. jeśli dwie klasy nadrzędne mają metodę o tej samej nazwie).
- **`super()`** działa zgodnie z MRO, więc może nie wywołać wszystkich metod, jeśli nie zostanie poprawnie użyte.
- W niektórych przypadkach **lepszym rozwiązaniem może być kompozycja zamiast wielodziedziczenia**.

Kompozycja

- Kompozycja to technika projektowania, w której **klasa zawiera obiekty innych klas**, zamiast dziedziczyć ich funkcjonalności.
- Jest to alternatywa dla dziedziczenia, szczególnie przydatna, gdy relacja między obiektami nie jest typu **„jest rodzajem” („is-a”)**, ale **„posiada” („has-a”)**.

Kluczowe cechy kompozycji

- **Elastyczność** – klasy mogą być łatwo modyfikowane i łączone w różnych konfiguracjach.
- **Brak problemów z MRO** – unikamy skomplikowanych zależności i konfliktów metod.
- **Łatwiejsza organizacja kodu** – zamiast „zmuszać” klasę do dziedziczenia, możemy po prostu „włożyć” inny obiekt jako atrybut.

Przykład: dziedziczenie vs kompozycja

```
class Silnik:
    def start(self):
        return "Silnik uruchomiony."

class Samochod(Silnik):
    def jedz(self):
        return "Samochód jedzie."
```

```
class Silnik:
    def start(self):
        return "Silnik uruchomiony."

class Samochod:
    def __init__(self, silnik):
        self.silnik = silnik  # Samochód "posiada" silnik

    def jedz(self):
        return "Samochód jedzie."

silnik_v8 = Silnik()
auto = Samochod(silnik_v8)

print(auto.silnik.start())  # "Silnik uruchomiony."
print(auto.jedz())          # "Samochód jedzie."
```

Wielodziedziczenie podsumowanie

- Wielodziedziczenie umożliwia korzystanie z metod i atrybutów wielu klas.
- Python używa MRO, aby ustalić kolejność przeszukiwania metod.
- **`super()`** w wielodziedziczeniu musi być stosowane ostrożnie
- Należy uważać na konflikty metod między klasami nadrzędnymi.

Stosuj wielodziedziczenie, gdy rzeczywiście jest potrzebne – w przeciwnym razie lepszym rozwiązaniem jest kompozycja!