Lekcja 12: Wprowadzenie do Programowania Obiektowego (OOP)

`#lekcja` `#python` `#oop` `#klasy` `#obiekty` `#programowanie_obiektowe`

Witaj w ósmej lekcji! Do tej pory poznaliśmy programowanie strukturalne i funkcyjne. Dziś zrobimy kolejny duży krok i wejdziemy w świat **programowania zorientowanego obiektowo (OOP)**. Jest to paradygmat, czyli sposób myślenia o programowaniu i strukturyzowania kodu, który pozwala modelować byty ze świata rzeczywistego w formie obiektów. Dzięki temu kod staje się bardziej zorganizowany, elastyczny i łatwiejszy w zarządzaniu, zwłaszcza w dużych projektach.

W tej lekcji omówimy:

- Czym są klasy i obiekty.
- Cztery filary OOP: Enkapsulację, Abstrakcję, Dziedziczenie i Polimorfizm.
- Jak dostosowywać działanie klas przez przeciążanie operatorów.
- Czym jest i do czego służy MRO (Method Resolution Order).

Zaczynajmy!

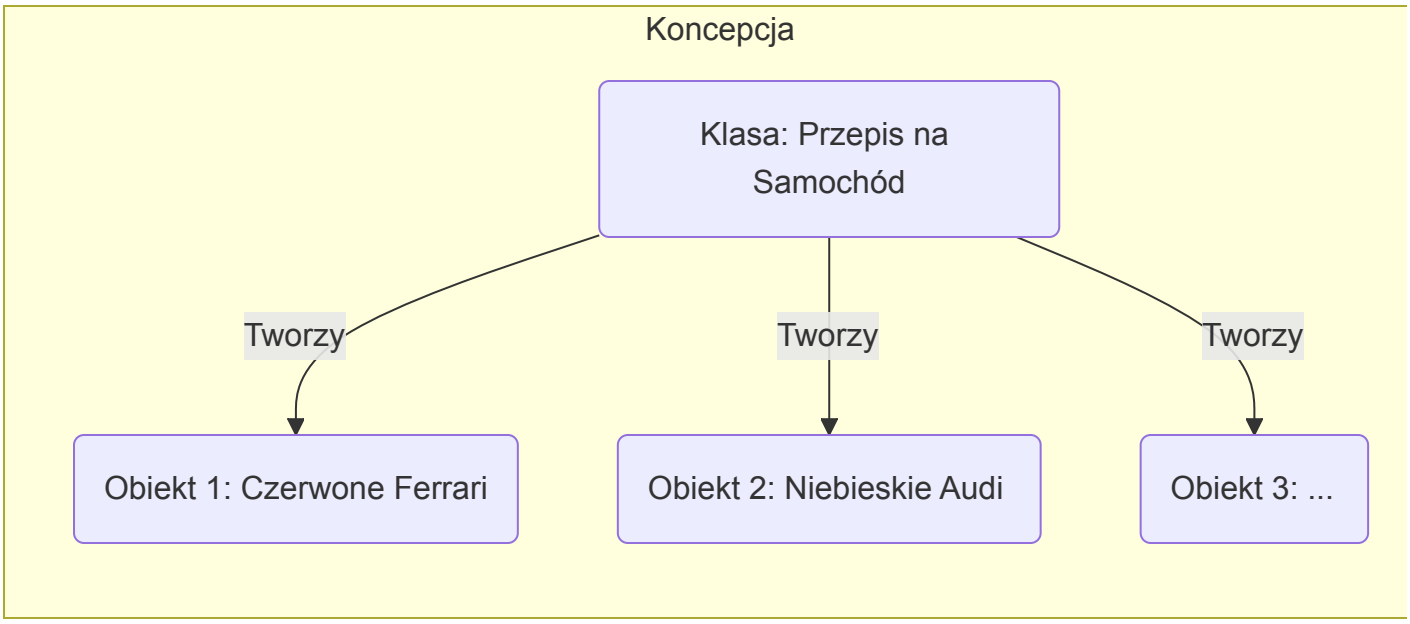
1. Klasy i obiekty: Szablony i ich realizacje

W programowaniu obiektowym wszystko kręci się wokół dwóch kluczowych pojęć: **klasy** i **obiektu**.

> **Definition**
>
> Klasa to szablon, przepis lub plan, według którego tworzone są obiekty. Definiuje ona wspólny zestaw cech (nazywanych atrybutami) oraz zachowań (nazywanych metodami), które będą miały wszystkie obiekty danego typu.
>
> **Obiekt** (lub instancja) to konkretny, istniejący w pamięci komputera egzemplarz stworzony na podstawie klasy. Posiada on wartości atrybutów zdefiniowanych w klasie.

Pomyśl o tym w ten sposób:

- **Klasa:** `PrzepisNaCiasto` – zawiera instrukcje, jakie składniki są potrzebne (atrybuty) i co z nimi zrobić (metody).
- **Obiekt:** `CiastoUpieczoneNaUrodziny` – konkretna realizacja przepisu, istniejące ciasto.



Zobaczmy, jak to wygląda w Pythonie. Stwórzmy prostą klasę `Pies`:

```
# Definicja klasy 'Pies'
# Używamy słowa kluczowego 'class', a nazwy klas zgodnie z konwencją PEP8 piszemy wielką literą (PascalCase)
class Pies:
    # Atrybut klasy - wspólny dla wszystkich obiektów tej klasy
    gatunek = "Canis lupus familiaris"

    # Metoda __init__ to specjalny "konstruktor". Wywołuje się automatycznie przy tworzeniu nowego obiektu.
    # Służy do inicjalizacji atrybutów obiektu.
    # 'self' odnosi się do konkretnego obiektu (instancji), który jest tworzony.
    def __init__(self, imie, wiek):
        # Atrybuty instancji - unikalne dla każdego obiektu
        self.imie = imie
        self.wiek = wiek
        print(f"Stworzono nowego psa o imieniu {self.imie}!")

    # Metoda instancji - operacja, którą obiekt może wykonać.
    def szczekaj(self):
        return "Hau, hau!"

    def przedstaw_sie(self):
        return f"Jestem {self.imie} i mam {self.wiek} lat."

# Tworzenie obiektów (instancji) klasy Pies
azor = Pies("Azor", 5)
burek = Pies("Burek", 3)
```

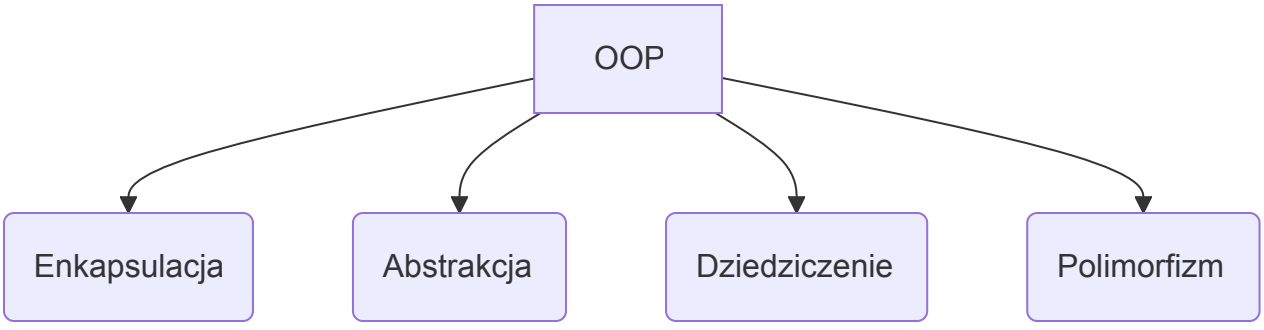
```
# Każdy obiekt ma swoje własne atrybuty
print(f"Imię pierwszego psa: {azor.imie}") # >> Imię pierwszego psa: Azor
print(f"Imię drugiego psa: {burek.imie}") # >> Imię drugiego psa: Burek

# Ale mogą współdzielić atrybuty klasy
print(f"Gatunek Azora: {azor.gatunek}") # >> Gatunek Azora: Canis lupus
familiaris
print(f"Gatunek Burka: {burek.gatunek}") # >> Gatunek Burka: Canis lupus
familiaris

# Wywoływanie metod na obiektach
print(f"{azor.imie} szczeka: {azor.szczekaj()}") # >> Azor szczeka: Hau,
hau!
print(burek.przedstaw_sie()) # >> Jestem Burek i mam 3 lat.
```

Tip

self jest niezwykle ważnym parametrem w metodach instancji. Reprezentuje on sam obiekt, na którym metoda jest wywoływana. Dzięki self mamy dostęp do atrybutów i innych metod tego konkretnego obiektu (np. self.imie).

2. Cztery filary OOP

Programowanie obiektowe opiera się na czterech fundamentalnych zasadach, które pomagają tworzyć dobrze zorganizowany i skalowalny kod.



Enkapsulacja (Hermetyzacja)

Definition

Enkapsulacja to mechanizm łączenia danych (atrybutów) i metod, które na nich operują, w jedną całość (klasę). Pozwala to również na ukrywanie wewnętrznego stanu obiektu i kontrolowanie dostępu do niego z zewnątrz.

Dzięki enkapsulacji użytkownik klasy nie musi martwić się o jej wewnętrzną logikę. Interakcja odbywa się przez zdefiniowany **interfejs** (metody publiczne), a szczegóły implementacji są ukryte.

```
class KontoBankowe:
    def __init__(self, numer_konta, saldo_poczatkowe=0):
        self.numer_konta = numer_konta
        # Użycie podwójnego podkreślenia '__' na początku nazwy atrybutu
        # sugeruje, że jest on "prywatny" i nie powinien być modyfikowany
bezpośrednio z zewnątrz.
        # Python "zmienia" jego nazwę, aby utrudnić przypadkowy dostęp.
        self.__saldo = saldo_poczatkowe

    # Metoda publiczna do wpłacania pieniędzy - jedyny słuszny sposób na
zwiększenie salda
    def wplac(self, kwota):
        if kwota > 0:
            self.__saldo += kwota
            print(f"Wpłacono {kwota} PLN. Nowe saldo: {self.__saldo} PLN.")
        else:
            print("Kwota wpłaty musi być dodatnia.")

    # Metoda publiczna do sprawdzania salda
    def sprawdz_saldo(self):
        return self.__saldo

# Tworzymy obiekt
moje_konto = KontoBankowe("123456789", 1000)

# Interakcja przez publiczne metody
print(f"Saldo początkowe: {moje_konto.sprawdz_saldo()}") # >> Saldo
początkowe: 1000
moje_konto.wplac(500) # >> Wpłacono 500 PLN. Nowe saldo: 1500 PLN.

# Próba bezpośredniej modyfikacji "prywatnego" atrybutu nie zadziała zgodnie
z oczekiwaniami
# i jest złą praktyką!
moje_konto.__saldo = -9999
print(f"Saldo po 'złej' modyfikacji: {moje_konto.sprawdz_saldo()}") # >>
```

```
Saldo po 'złej' modyfikacji: 1500

# Dzieje się tak, ponieważ Python zmienił nazwę atrybutu na
_KontoBankowe__saldo
# print(moje_konto._KontoBankowe__saldo) # To by zadziałało, ale tego NIE
robimy!
```

Abstrakcja

> **Definition**
>
> Abstrakcja polega na ukrywaniu złożonych szczegółów implementacyjnych i pokazywaniu użytkownikowi tylko niezbędnych funkcji. Koncentrujemy się na tym, co obiekt robi, a nie jak to robi.

Pomyśl o pilocie do telewizora. Masz przyciski do zmiany kanału i regulacji głośności. Nie musisz wiedzieć, jakie sygnały elektroniczne są wysyłane, aby to zadziałało. Używasz prostego interfejsu.

```
# Przykład abstrakcji
# Użytkownik tej klasy nie musi wiedzieć, jak dokładnie działa serializacja
do formatu JSON.
# Po prostu wywołuje metodę 'zapisz_do_pliku'.
import json

class Uzytkownik:
    def __init__(self, imie, email):
        self.imie = imie
        self.email = email

    def zapisz_do_pliku(self, sciezka_pliku):
        """Zapisuje dane użytkownika do pliku w formacie JSON."""
        dane = {
            'imie': self.imie,
            'email': self.email
        }
        try:
            with open(sciezka_pliku, 'w', encoding='utf-8') as f:
                # Cała złożoność pracy z JSON jest ukryta wewnątrz tej
metody.
                json.dump(dane, f, ensure_ascii=False, indent=4)
            print(f"Dane użytkownika {self.imie} zapisano w
{sciezka_pliku}")
```

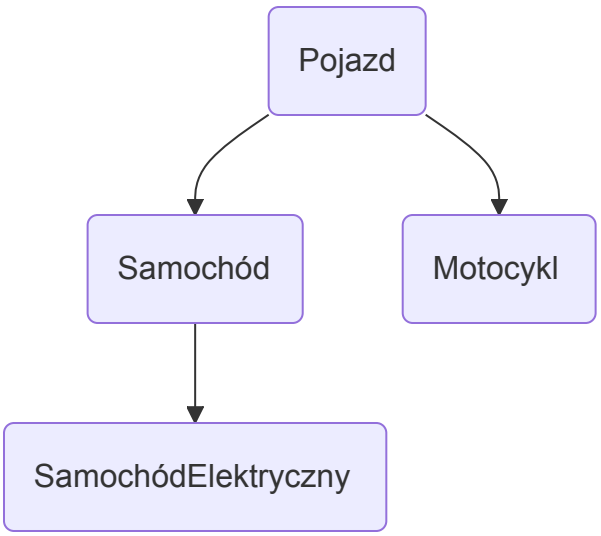
```
        except IOError as e:
            print(f"Wystąpił błąd podczas zapisu do pliku: {e}")

# Użycie klasy jest proste i nie wymaga znajomości modułu json
nowy_user = Uzytkownik("Jan", "jan@kowalski.pl")
nowy_user.zapisz_do_pliku("user_jan.json") # >> Dane użytkownika Jan
zapisano w user_jan.json
```

Dziedziczenie

Definition

Dziedziczenie to mechanizm, który pozwala tworzyć nową klasę (nazywaną klasą potomną lub podklasą) na podstawie istniejącej klasy (klasy bazowej lub nadrzędnej). Klasa potomna przejmuje ("dziedziczy") wszystkie atrybuty i metody klasy bazowej, a także może dodawać nowe lub modyfikować istniejące.

Dziedziczenie promuje reużywalność kodu. Zamiast pisać wszystko od nowa, możemy rozszerzać funkcjonalność istniejących klas.




```
# Klasa bazowa (rodzic)
class Zwierze:
    def __init__(self, imie):
        self.imie = imie

    def jedz(self):
        return f"{self.imie} je."

# Klasa potomna (dziecko), która dziedziczy po klasie Zwierze
```

```python
# W nawiasach podajemy nazwę klasy, po której dziedziczymy.
class Kot(Zwierze):
    # Możemy dodać nową metodę, specyficzną tylko dla kota
    def miaucz(self):
        return "Miau!"

# Kolejna klasa potomna
class Pies(Zwierze):
    # Możemy nadpisać (zmienić) działanie metody odziedziczonej
    def jedz(self):
        return f"{self.imie} chrupie karmę."

    def szczekaj(self):
        return "Hau!"

# Tworzymy obiekty
mruczek = Kot("Mruczek")
reksio = Pies("Reksio")

# Mruczek ma dostęp do metody z klasy Zwierze
print(mruczek.jedz()) # >> Mruczek je.
# Oraz do swojej własnej metody
print(mruczek.miaucz()) # >> Miau!

# Reksio używa swojej własnej, nadpisanej wersji metody jedz()
print(reksio.jedz()) # >> Reksio chrupie karmę.
print(reksio.szczekaj()) # >> Hau!
```

Polimorfizm

Definition

Polimorfizm (z gr. "wielopostaciowość") to zdolność obiektów różnych klas do odpowiadania na to samo wywołanie metody w sposób specyficzny dla swojej klasy.

Innymi słowy, możemy używać tego samego interfejsu (np. wywoływać metodę `mow()`) na obiektach różnych typów, a każdy z nich zachowa się inaczej.

```python
# Kontynuując poprzedni przykład z dziedziczeniem
# Mamy już klasy Zwierze, Kot i Pies.
# Każda z nich ma inną implementację (lub jej brak) metody 'mow'
class Kot(Zwierze):
    def mow(self):
        return "Miau!"
```

```
class Pies(Zwierze):
    def mow(self):
        return "Hau, hau!"

class Krowa(Zwierze):
    def mow(self):
        return "Muuu!"

# Tworzymy listę obiektów różnych klas, ale wszystkie są typu Zwierze
zwierzeta = [
    Kot("Filemon"),
    Pies("Burek"),
    Krowa("Mućka")
]

# Dzięki polimorfizmowi możemy iterować po liście i wywoływać tę samą metodę
'mow()'
# na każdym obiekcie, nie martwiąc się o jego konkretny typ.
# Każdy obiekt "wie", jak ma odpowiedzieć.
for zwierze in zwierzeta:
    print(f"{zwierze.imie} mówi: {zwierze.mow()}")

# Wynik:
# Filemon mówi: Miau!
# Burek mówi: Hau, hau!
# Mućka mówi: Muuu!
```

3. Przeciążanie operatorów i metody specjalne

Python pozwala klasom na definiowanie własnego zachowania dla wbudowanych operatorów, takich jak `+`, `-`, `*`, `==` itd. Odbywa się to poprzez implementację **metod specjalnych** (nazywanych też "magicznymi" lub "dunder" od *double underscore*).

> **Note**
>
> Metody specjalne mają nazwy zaczynające się i kończące podwójnym podkreśleniem, np. **init**, **str**, **add**.

Jedną z najważniejszych już poznaliśmy – to `__init__`, czyli konstruktor. Innym bardzo użytecznym jest `__str__`.

```
class Ksiazka:
    def __init__(self, tytul, autor, strony):
```

```
        self.tytul = tytul
        self.autor = autor
        self.strony = strony

    # Domyślnie, gdy spróbujemy wydrukować obiekt, zobaczymy coś mało
czytelnego,
    # np. <__main__.Ksiazka object at 0x...>

    # Możemy to zmienić, implementując metodę __str__
    def __str__(self):
        # Ta metoda musi zwrócić stringa.
        # Będzie on używany, gdy na obiekcie wywołamy funkcję str() lub
print().
        return f'"{self.tytul}" autorstwa {self.autor}, {self.strony}
stron.'

    # Przeciążmy operator dodawania (+)
    def __add__(self, other):
        # Chcemy, aby dodanie dwóch książek zwróciło sumę ich stron
        if isinstance(other, Ksiazka):
            return self.strony + other.strony
        # Zwracamy specjalną wartość, jeśli typ jest nieobsługiwany
        return NotImplemented


ksiazka1 = Ksiazka("Wiedźmin", "Andrzej Sapkowski", 320)
ksiazka2 = Ksiazka("Lalka", "Bolesław Prus", 780)

# Dzięki __str__ wydruk jest teraz czytelny
print(ksiazka1) # >> "Wiedźmin" autorstwa Andrzej Sapkowski, 320 stron.
print(ksiazka2) # >> "Lalka" autorstwa Bolesław Prus, 780 stron.

# Dzięki __add__ możemy "dodać" do siebie obiekty
suma_stron = ksiazka1 + ksiazka2
print(f"\nSuma stron obu książek: {suma_stron}") # >> Suma stron obu
książek: 1100
```

Info

W kontekście dziedziczenia, gdy klasa potomna definiuje metodę **init**, "przesłania" ona konstruktor klasy bazowej. Aby wywołać również logikę z konstruktora rodzica, używamy funkcji super():

```
class SamochodElektryczny(Samochod):
    def __init__(self, marka, model, zasieg_baterii):
```

```
        # Wywołaj __init__ z klasy nadrzędnej (Samochod), aby ustawiła
markę i model
        super().__init__(marka, model)
        # Dodaj nową logikę specyficzną dla dziecka
        self.zasieg_baterii = zasieg_baterii
```

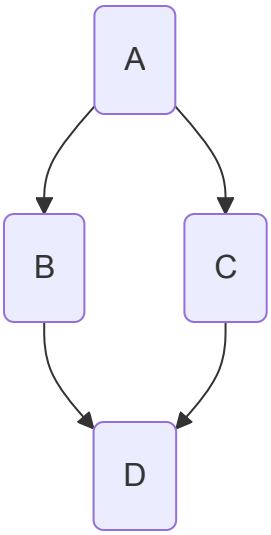
4. MRO: Method Resolution Order

Gdy mamy do czynienia ze złożoną hierarchią dziedziczenia, zwłaszcza z **dziedziczeniem wielokrotnym** (gdy klasa dziedziczy po więcej niż jednej klasie bazowej), Python musi wiedzieć, w jakiej kolejności przeszukiwać klasy nadrzędne w poszukiwaniu metody.

Definition

MRO (Method Resolution Order) to algorytm określający dokładną, liniową kolejność, w jakiej Python przeszukuje hierarchię klas w celu znalezienia metody, która została wywołana.

Jest to szczególnie ważne, aby uniknąć problemów, takich jak "problem diamentu".




```
# Przykład "problemu diamentu"
class A:
    def kim_jestem(self):
        print("Jestem z klasy A")

class B(A):
    def kim_jestem(self):
        print("Jestem z klasy B")
```

```python
class C(A):
    def kim_jestem(self):
        print("Jestem z klasy C")

# Klasa D dziedziczy po B i C. Po której z nich powinna odziedziczyć metodę
'kim_jestem'?
class D(B, C):
    pass

# Tworzymy obiekt klasy D
d = D()
d.kim_jestem() # >> Jestem z klasy B

# Dlaczego B, a nie C? Odpowiedź leży w MRO.
# Możemy sprawdzić MRO dla dowolnej klasy na dwa sposoby:

# 1. Metoda .mro()
print("\nMRO dla klasy D:")
print(D.mro())
# >> MRO dla klasy D:
# >> [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>]

# 2. Funkcja help()
# help(D)

# Zgodnie z MRO, Python szuka metody 'kim_jestem' w kolejności: D -> B -> C
-> A.
# Znajduje ją już w klasie B i tam kończy poszukiwania.
```

> **Tip**
>
> MRO jest obliczane za pomocą algorytmu C3. Nie musisz znać jego szczegółów, ale warto pamiętać, że super() również podąża za MRO, aby wywołać metodę z następnej klasy w kolejce.

🧪 Zadania do samodzielnej pracy

Zadania proste

1. ✏️ Zadanie 1 – Klasa Film

Stwórz klasę Film, która przy tworzeniu obiektu będzie przyjmować tytul, rezyser i rok_produkcji. Dodaj metodę informacje(), która będzie zwracać string z pełnymi informacjami o filmie w formacie: "Tytuł" (rok_produkcji), reżyseria: Reżyser. Stwórz dwa obiekty tej klasy i wydrukuj informacje o nich.

(proste)

2. ✏️ Zadanie 2 – Atrybuty Produkt

   Zdefiniuj klasę Produkt z konstruktorem **init** przyjmującym nazwa, cena i kategoria. Stwórz obiekt tej klasy, a następnie wydrukuj każdy z jego atrybutów w osobnej linii.

   (proste)

3. ✏️ Zadanie 3 – Dziedziczenie Pracownik -> Programista

   Stwórz klasę bazową Pracownik z atrybutami imie i stawka_godzinowa. Dodaj metodę oblicz_pensje(liczba_godzin). Następnie stwórz klasę potomną Programista, która dziedziczy po Pracownik. W klasie Programista dodaj atrybut jezyki_programowania (lista stringów). Stwórz obiekt klasy Programista i wywołaj na nim metodę oblicz_pensje.

   (proste)

4. ✏️ Zadanie 4 – Czytelny Punkt

   Stwórz klasę Punkt do reprezentowania punktu w 2D, z atrybutami x i y. Zaimplementuj metodę **str**, aby print(punkt) wyświetlał współrzędne w formacie (x, y).

   (proste)

5. ✏️ Zadanie 5 – Polimorficzna Figura

   Stwórz klasę bazową Figura z metodą oblicz_pole(), która pass (nic nie robi). Następnie stwórz dwie klasy potomne: Kwadrat (z atrybutem bok) i Kolo (z atrybutem promien). W obu klasach nadpisz metodę oblicz_pole() odpowiednimi wzorami matematycznymi (dla koła przyjmij PI=3.14159). Stwórz listę zawierającą jeden kwadrat i jedno koło, a następnie w pętli wydrukuj pole każdej figury.

   (proste)

Zadania z gwiazdką (challenge)

6. 🧠 Zadanie 6 – Wektor 2D i przeciążanie operatorów

   Stwórz klasę Wektor2D z atrybutami x i y. Przeciąż następujące operatory:

   - `__add__(self, other)`: do dodawania dwóch wektorów (dodajemy odpowiadające sobie współrzędne).
   - `__sub__(self, other)`: do odejmowania wektorów.

- **eq**(self, other): do porównywania, czy dwa wektory są równe (mają te same x i y). Dodatkowo zaimplementuj **str** do ładnego wyświetlania. Przetestuj działanie, tworząc dwa wektory i wykonując na nich wszystkie zaimplementowane operacje. (challenge)

7. 🧠 Zadanie 7 – Enkapsulacja w Telewizorze

   Stwórz klasę Telewizor. Użyj enkapsulacji, aby ukryć następujące atrybuty: **kanal (domyślnie 1),** glosnosc (domyślnie 10), __wlaczony (domyślnie False). Stwórz publiczne metody do zarządzania telewizorem:

   - `wlacz()` i `wylacz()`
   - `zmien_kanal(numer)`: kanał można zmienić tylko, gdy TV jest włączony.
   - `glosniej()` i `ciszej()`: głośność można regulować w zakresie 0-100 i tylko, gdy TV jest włączony.
   - info(): wyświetla aktualny stan (włączony/wyłączony, kanał, głośność). Przetestuj, czy nie da się zmienić kanału na wyłączonym telewizorze lub ustawić głośności powyżej 100. (challenge)

8. 🧠 Zadanie 8 – Hierarchia instrumentów muzycznych

   Zaprojektuj hierarchię klas: Instrument -> Strunowy i Dety. Następnie Gitara (dziedziczy po Strunowy) i Trabka (dziedziczy po Dety). Klasa Instrument powinna mieć metodę graj(), która zwraca ogólny komunikat. Każda kolejna klasa w hierarchii powinna nadpisywać tę metodę, dodając coś od siebie i wywołując wersję z klasy nadrzędnej za pomocą super().graj().

   - `Instrument.graj()` -> `"Wydaje dźwięk."`
   - `Strunowy.graj()` -> `"Wydaje dźwięk. [Szarpnięcie struny]"`
   - Gitara.graj() -> "Wydaje dźwięk. [Szarpnięcie struny] [Akord G-dur]" (challenge)

9. 🧠 Zadanie 9 – Walidacja danych w **init**

   Stwórz klasę RejestracjaUzytkownika. W konstruktorze **init** przyjmuj email i haslo. Wewnątrz konstruktora dodaj walidację:

   - Sprawdź, czy `email` zawiera znak `@`. Jeśli nie, podnieś wyjątek `ValueError` z odpowiednim komunikatem.
   - Sprawdź, czy haslo ma co najmniej 8 znaków. Jeśli nie, podnieś ValueError. Użyj bloku try...except, aby przetestować tworzenie obiektów z poprawnymi i niepoprawnymi danymi. (challenge)

10. 🧠 Zadanie 10 – Eksploracja MRO

    Stwórz następującą, złożoną hierarchię dziedziczenia:

    - `class A`
    - `class B(A)`
    - `class C(A)`

- `class D(B)`
- `class E(C)`
- class F(D, E) Narysuj schemat tej hierarchii w mermaid. Następnie, nie uruchamiając kodu, spróbuj przewidzieć, jakie będzie MRO dla klasy F. Na koniec sprawdź swoją odpowiedź, używając print(F.mro()). (challenge)