# Programowanie obiektowe

Paweł Gliwny

# OOP (ang. *object-oriented programming*)

- **OOP** TO WŁASNOŚĆ JĘZYKA PROGRAMOWANIA, KTÓRA UMOŻLIWIA GRUPOWANIE ZMIENNYCH I FUNKCJI W NOWE TYPY DANYCH, ZWANE **KLASAMI.**
- NA PODSTAWIE KLAS MOŻNA TWORZYĆ **OBIEKTY**.
- DZIĘKI zorganizowaniu **kodu w klasy** można podzielić monolityczny program na mniejsze części, które są łatwiejsze do **zrozumienia** i **debugowania.**

# Podstawowe zasady programowania obiektowego (OOP)

- **Hermetyzacja (ang. *Encapsulation*):**
Grupowanie danych i metod w jednej klasie.
- **Dziedziczenie (ang. *Inheritance*):**
Rozszerzanie funkcjonalności istniejącego kodu.
- **Polimorfizm (ang. *Polymorphism*):**
Tworzenie jednolitego interfejsu dla różnych typów obiektów.

# Czy zawsze klasy są potrzebne?

- Niektóre języki (Java), wymagają zorganizowania całego kodu w klasach, własności **OOP języka Python są opcjonalne**.
- Można korzystać z klas, jeśli ich potrzebujemy, lub zignorować je, jeśli nie są nam potrzebne.
- Główny programista Pythona, Jack Diederich's w referacie *Stop Writing Classes* (PyCon 2012, youtube) wskazuje na wiele przypadków, w których programiści piszą klasy w sytuacjach, gdy lepiej sprawdzałyby się prostsze funkcje lub moduły.
- Programista jednak powinien znać **podstawy** OOP.

# Klasy w python

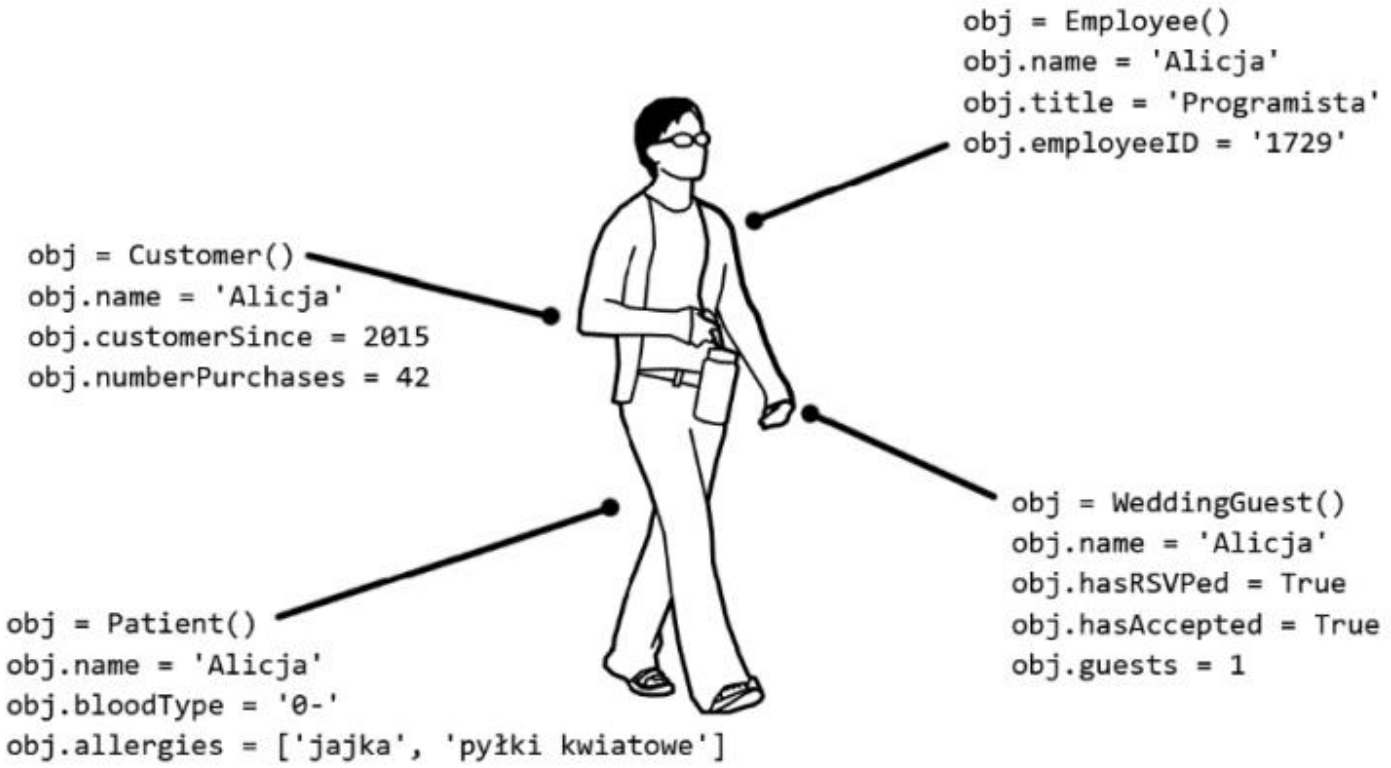
- W Pythonie klasa i typ danych są pojęciami równoznacznymi.
- Podobnie jak w formularz papierowy lub elektroniczny, **klasa jest planem dla obiektów języka Python** (nazywanych również egzemplarzami — ang. *instance*)

# Klasy analogia

- Klasy przypominają pusty **szablon formularza**, a obiekty utworzone na podstawie tych klas są jak wypełnione formularze — tzn. **zawierają właściwe dane** o rodzaju rzeczy, którą reprezentuje formularz

# Klasy, a świat rzeczywisty

- Klasy i obiekty często porównuje się do **modeli** danych elementów w świecie rzeczywistym.
- Nie należy jednak mylić mapy z terenem.
- To, co trafia do **klasy, zależy od tego, co program musi zrobić.**



# Tworzenie prostej klasy: WizCoin

- Metoda __init__() - konstruktor klasu, nigdy nie zawierają instrukcji return
- Używamy **`self`**, aby odwołać się do atrybutów i metod klasy (instancji).

```python
class WizCoin:
    def __init__(self, galleons, sickles, knuts):
        """Tworzy nowy obiekt Wizcoin z monetami: knutami, syklami, galeonami."""
        self.galleons = galleons
        self.sickles = sickles
        self.knuts = knuts
    def value(self):
        """Wartość (w knutach) wszystkich monet w tym obiekcie WizCoin."""
        return (self.galleons * 17 * 29) + (self.sickles * 29) + (self.knuts)
    def weightInGrams(self):
        """Zwraca wagę monet w gramach."""
        return (self.galleons * 31.103) + (self.sickles * 11.34) + (self.knuts * 5.0)
```

**wizcoin.py**

# Tworzenie i korzystania z modułu (klasy)

- Zgodnie z konwencją nazwy modułów (takie jak **wizcoin** w naszym pliku ***wizcoin.py***) są pisane małymi literami, podczas gdy nazwy klas (takie jak **WizCoin**) zaczynają się od wielkich liter.

```
import wizcoin

purse = wizcoin.WizCoin(2, 5, 99)
print(purse)
print('G:', purse.galleons, 'S:', purse.sickles, 'K:', purse.knuts)
print('Całkowita wartość:', purse.value())
print('Waga:', purse.weightInGrams(), 'grama')
```

# Poziom instancji w Pythonie

- **Poziom instancji** oznacza, że dane i metody są związane z konkretnym obiektem (instancją) klasy.
- Każda instancja może mieć inne wartości atrybutów, niezależne od innych obiektów tej samej klasy.

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand  # Atrybut instancji
        self.model = model  # Atrybut instancji
        self.year = year    # Atrybut instancji

    def display_info(self):
        """Metoda instancji, która wypisuje informacje o samochodzie."""
        print(f"Samochód: {self.brand} {self.model}, Rok: {self.year}")

# Tworzenie dwóch różnych instancji klasy Car
car1 = Car("Toyota", "Corolla", 2020)
car2 = Car("Ford", "Focus", 2018)

# Każda instancja ma inne dane
car1.display_info()  # Samochód: Toyota Corolla, Rok: 2020
car2.display_info()  # Samochód: Ford Focus, Rok: 2018
```

# Metoda klasy (`@classmethod`) w Pythonie

- Metoda klasy w Pythonie to metoda, która działa na poziomie klasy, a nie konkretnej instancji obiektu.
- Jest oznaczona dekoratorem **`@classmethod`** i jako pierwszy argument przyjmuje **`cls`**, który odnosi się do samej klasy.

**Cechy metody klasy:**

- Nie ma dostępu do atrybutów instancji (nie używa `self`).
- Może odwoływać się do atrybutów i metod klasy.
- Może być wywoływana zarówno na poziomie klasy, jak i instancji.

# @classmethod – Metody Klasy

- Jest to metoda działająca na poziomie **klasy**, a nie pojedynczej instancji.
- Jest oznaczona dekoratorem **@classmethod**.
- Jako pierwszy argument przyjmuje cls, który odnosi się do samej klasy.
- **Nie może używać atrybutów instancji (self)**.

```python
class MyClass:
    licznik = 0

    def __init__(self):
        MyClass.licznik += 1

    1 usage
    @classmethod
    def ile_obiektow(cls):
        return (f"Liczba utworzonych obiektów:"
                f" {cls.licznik}")
obj1 = MyClass()
obj2 = MyClass()

print(MyClass.ile_obiektow())
```

- **Metody klasy** działają na poziomie klasy, a nie pojedynczych obiektów.
- **Używamy ich do operacji związanych z całą klasą**, np. śledzenia liczby obiektów.

# @classmethod – alternatywny konstruktor

- **Pozwala na alternatywne konstruktory**
- Można mieć tylko jedną metodę `__init__()`
- Używaj metod klasy do tworzenia obiektów
- Użyj `return`, aby zwrócić obiekt
- `cls(...)` wywoła `__init__(...)`

```python
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    @classmethod
    def from_file(cls, filename):
        with open(filename, "r") as f:
            text = f.readlines()
            name = text[0].split()[0]
            salary = text[0].split()[1]
        return cls(name, salary)
```

```python
emp = Employee.from_file("salary_data.txt")
print(emp.name, emp.salary)
```

# Kiedy używać @classmethod

- Alternatywne konstruktory
- Ograniczenie do pojedynczej instancji (obiektu) klasy
  - Połączenia z bazą danych
  - Ustawienia konfiguracyjne

**Do zrobienie zadanie 1**

# Metody magiczne w Pythonie

- **Specjalne (magiczne) metody klasy** pozwalają klasom w Pythonie:
- naśladować typy wbudowane,
- reagować na operacje, np.:
  - dodawanie,
  - mnożenie,
  - sprawdzanie długości,
  - reprezentację jako string,
  - inne operacje wbudowane.

- `__init__`: Konstruktor obiektu, wywoływany przy tworzeniu nowej instancji.
- `__str__:` Zwraca reprezentację obiektu w formie łańcucha znaków, przyjazną dla człowieka.
- `__len__`: Zwraca długość obiektu, umożliwia używanie funkcji `len()` na instancjach klasy.
- `__eq__`: Definiuje zachowanie równości za pomocą operatora `==`.

# Porównywanie obiektów

```
class Customer:
    def __init__(self, name, acc_id):
        self.name = name
        self.acc_id = acc_id

customer1 = Customer("Maryam Azar", 123)
customer2 = Customer("Maryam Azar", 123)

print(customer1 == customer2)  # ❌ False – porównuje referencje w pamięci!
```

- **Choć obiekty zawierają te same dane, wynik to `False`.**

# **Przeciążanie operatorów** (ang. operator overloading)

- Python pozwala na **dostosowanie zachowania operatorów** (+, -, == itp.) dla obiektów własnych klas.
- Przeciążanie operatorów umożliwia np. porównywanie obiektów w sposób logiczny zamiast porównywania ich referencji w pamięci.

```
class Customer:
….
    # Przeciążenie operatora ==
    def __eq__(self, other):
        print("__eq__() jest wywoływane!")
        return (self.acc_id == other.acc_id) and (self.name == other.name)

customer1 = Customer("Maryam Azar", 15)
customer2 = Customer("Maryam Azar", 15)

print(customer1 == customer2)  # ✅ True
```

# Przeciążanie Operatora +

Możemy zdefiniować metodę __add__(), aby określić, jak obiekty danej klasy powinny być sumowane.

```
class BankAccount:
    def __init__(self, owner, balance, acc_id):
        self.owner = owner
        self.balance = balance

    def __add__(self, other):
        return BankAccount(self.owner, self.balance + other.balance)

acc1 = BankAccount("Jan", 5000)
acc2 = BankAccount("Jan", 3000)

acc3 = acc1 + acc2
print(acc3)  # Konto Jan, saldo: 8000 PLN
```

# Wyświetlanie informacji o obiekcie

```
class Customer:
      def __init__(self, name, acc_id):
          self.name = name
          self.acc_id = acc_id
```

- `c1 = Customer("Maryam Azar", 123)`
- print(c1)

<__main__.Customer object at 0x000001DFB4AC85D0>

- a_list = [1, 2, 3]
- print(a_list)

[1, 2, 3]

# Metody specjalne `__str__` i `__repr__`

***__str__*** zwraca przyjazną dla użytkownika reprezentację obiektu.

- Jest wywoływana, gdy używamy ***print()*** lub ***str(obj)***.

***__repr__*** zwraca **dokładną** reprezentację obiektu, często w formie kodu, który można użyć do jego odtworzenia.

- Jest wywoływana przez ***repr(obj)*** oraz w interpreterze Pythona.

```python
class Osoba:
    def __init__(self, imie, wiek):
        self.imie = imie
        self.wiek = wiek

    def __str__(self):
        return f"{self.imie}, lat {self.wiek}"

    def __repr__(self):
        return f"Osoba('{self.imie}', {self.wiek})"

osoba = Osoba( imie: "Paweł", wiek: 30)
print(osoba)       # __str__: Paweł, lat 30
print(repr(osoba)) # __repr__: Osoba('Paweł', 30)
```

__str__()

- print(obj), str(obj)

```
$> print(np.array([1,2,3])
[1 2 3]

$> str(np.array([1 2 3]))
'[1 2 3]'
```

- *Nieformalny, dla użytkowników kończowych*
- *Reprezentacja w postaci string*

__repr__()

- repr(obj), wyświetlanie w konsoli

```
$> repr(np.array([1, 2, 3]))
'array([1,2,3])'

$> np.array([1,2,3])
array([1, 2, 3])
```

- *Formalny, dla developerów*
- ***Repr**oducible **repr**esentation*
- *Zastępstwo dla `print()`*

- Implementacja metody **str**

```
class Customer:
...
  def __str__(self):
      cust_string = f"""
          Customer:
                  Name: {self.name},
                  Id: {self.acc_id}.
                  """
      return cust_string
```

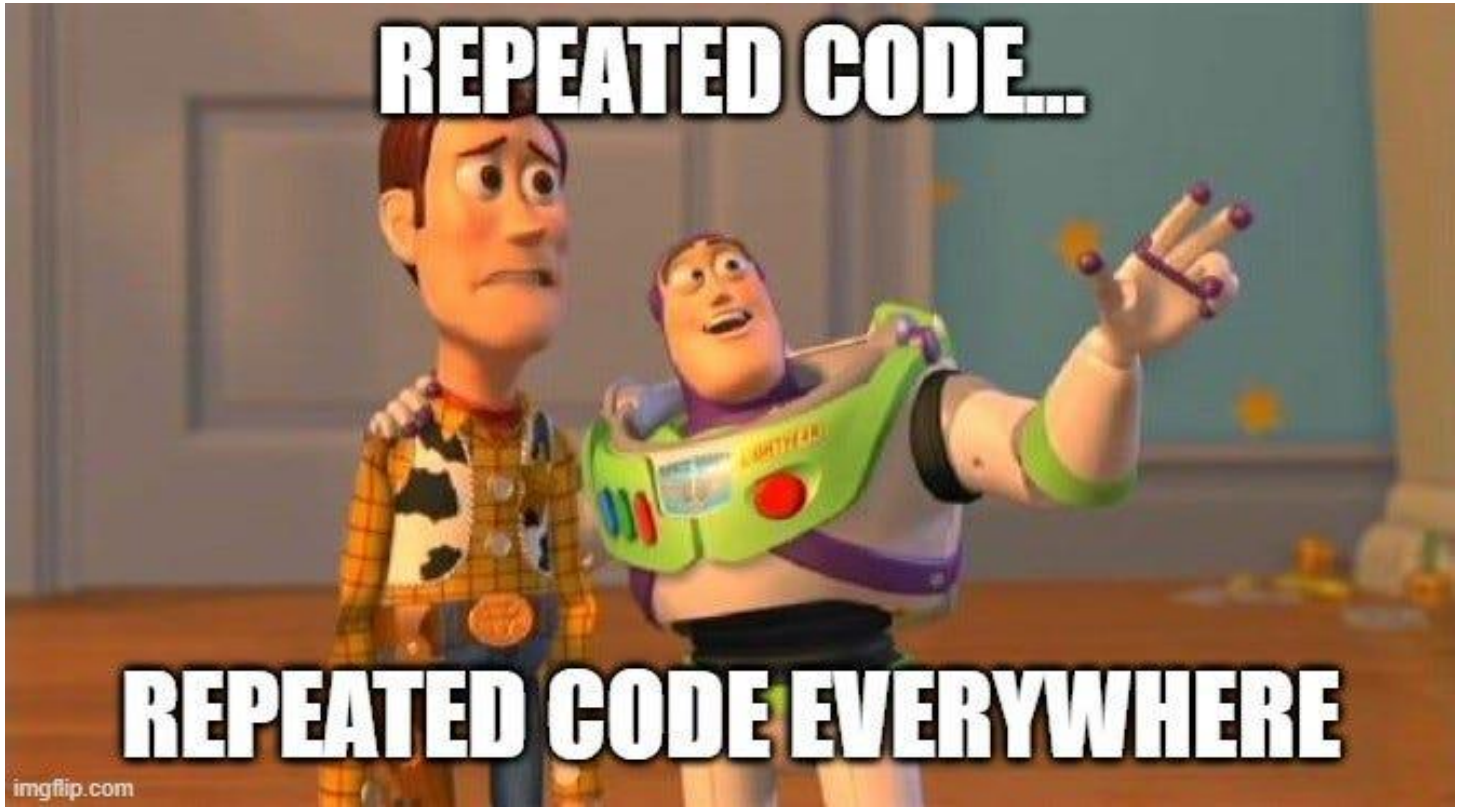
- Implementacja metody **repr**

```
class Customer:
...
  def __repr__(self):
     return f"Customer('{self.name}',
  {self.acc_id})"
```

# Dziedziczenie - idea

- DRY: Don't Repeat Yourself

***Nowa funkcjanolność klasy = stara funkcjanalność klasy + coś extra***



# 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
```

# 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.
  - Obsługuje **wielodziedziczenie**, wywołując metody zgodnie z kolejnością MRO (*Method Resolution Order*).
  - 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**.

# 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!**

# 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."
```

# źródła

- *Programowanie w Pythonie dla średnio zaawansowanych. Najlepsze praktyki tworzenia czystego kodu*, Al Sweigart, Helion
- Kurs o programowaniu obiektowym na platformie DataCamp