Programowanie obiektowe

mgr inż. Bartłomiej Kizielewicz

Zespół Badawczy Inteligentnych Systemów Wspomagania Decyzji
Katedra Sztucznej Inteligencji i Matematyki Stosowanej
Wydział Informatyki
Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Listopad, 2021

Paradygmaty programowania

Co to jest paradygmat?

Paradygmat programowania jest to zbiór mechanizmów jakich używa programista przy pisaniu programu oraz jak napisany program jest wykonywany przez komputer.

Przykładowe paradygmaty

- Programowanie imperatywne
- Programowanie obiektowe
- Programowanie funkcyjne
- Programowanie logiczne

Programowanie obiektowe

W programowaniu obiektowym program to zbiór porozumiewających się ze sobą obiektów, czyli jednostek zawierających pewne dane i umiejących wykonywać na nich pewne operacje.

Ważne cechy

- Ważną cechą jest tu powiązanie danych (czyli stanu) z operacjami na nich (czyli poleceniami) w całość, stanowiącą odrębną jednostkę — obiekt.
- Cechą nie mniej ważną jest mechanizm dziedziczenia, czyli możliwość definiowania nowych, bardziej złożonych obiektów, na bazie obiektów już istniejących.

Przykładowy kod

```
01 |  class Cat:
02 |      species = "Siberian cat"
03 |
04 |      def __init__(self, name, age):
05 |          self.name = name
06 |          self.age = age
07 |
08 |  object = Cat('Keks', 1)
```

Podstawowe pojęcia

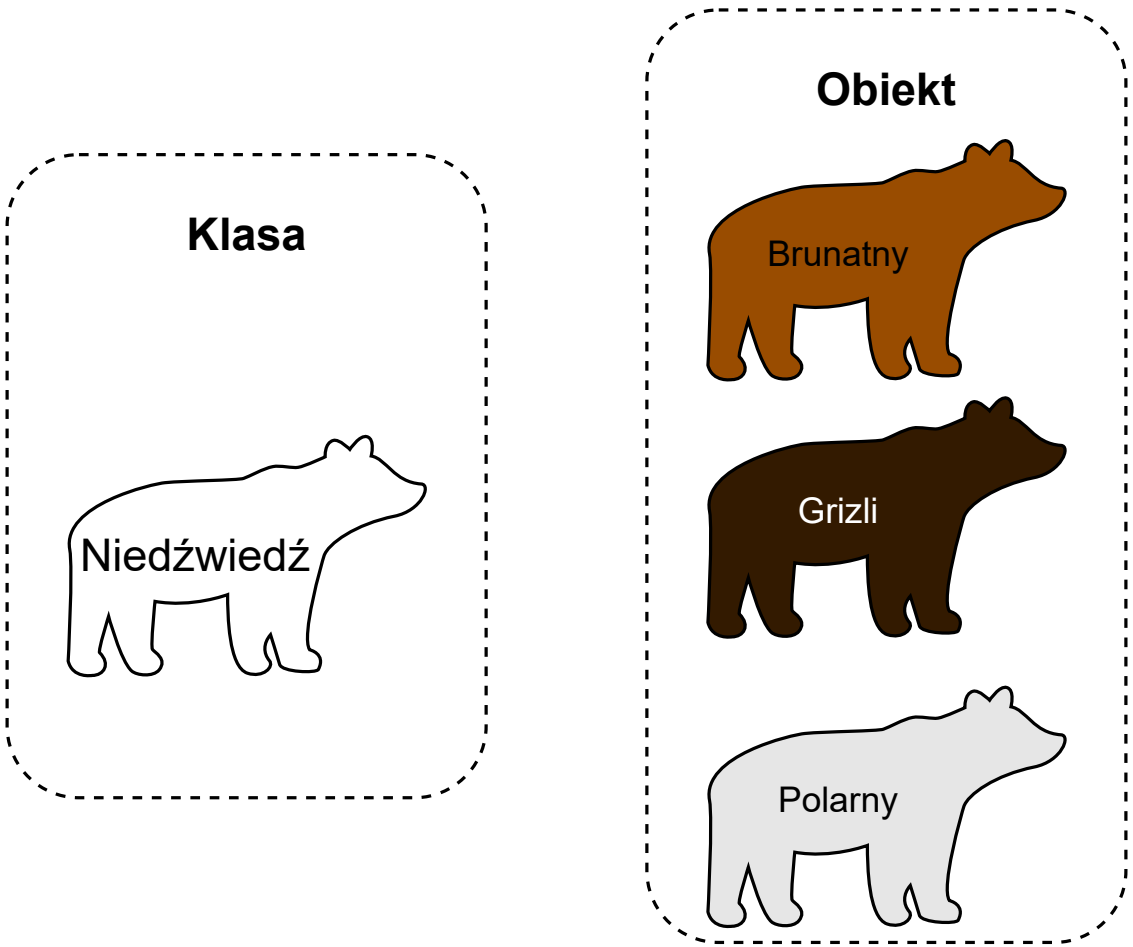
Klasa

Klasa jest planem definiującym naturę przyszłego obiektu. Dzięki niej jesteśmy w stanie organizować informacje o typach danych. Klasy służą do tworzenia i zarządzania nowymi obiektami.

Obiekt

Obiekt to instancja klasy lub podklasy z własnymi metodami lub procedurami klasy oraz zmiennymi danych. Przykładem takiego obiektu może być stół, but, miauczenie lub nawet nazwa koloru.

Klasa oraz obiekt



Atrybut oraz metoda

Atrybut

Atrybutem w programowaniu obiektowym nazywamy zmienną przechowywaną w obiekcie danego typu. Atrybuty to właściwości obiektów, które określają ich cechy lub wartości, które obiekt przechowuje. W przypadku tworzenia nowej instancji klasy, każda z tych zmiennych zostanie zainicjowana wartością domyślną lub wartością przekazaną do konstruktora.

Metoda

Metodą w programowaniu obiektowym nazywamy funkcję zdefiniowana wewnątrz klasy, która może działać na instancjach tej klasy. Metoda ta ma dostęp do atrybutów i metod instancji oraz klasy, w której została zdefiniowana.

Atrybut oraz metoda

Atrybuty

- kolor: brązowy
- rasa: brunatny
- wzorst: 160 cm

Metody

- jedz (pokarm)
- gryź (mięso)
- pij (wodę)
- idź
- biegnij

Konstruktor

Konstruktor

W języku Python konstruktor to specjalna metoda klasy o nazwie `__init__`. Konstruktor jest wywoływany automatycznie podczas tworzenia nowego obiektu klasy i służy do inicjalizacji atrybutów obiektu.

Przykład konstruktora

```
01 |  class NowaKlasa:
02 |
03 |      def __init__(self, v1, v2): # konstruktor
04 |          self.pole1 = v1
05 |          self.pole2 = v2
```

Przykład w języku Python

```
01 |  class Niedzwiedz: # klasa Niedzwiedz
02 |
03 |      # Konstruktor
04 |      def __init__(self, kolor, rasa, wzrost):
05 |          self.kolor = kolor # pole kolor
06 |          self.rasa = rasa
07 |          self.wzrost = wzrost
08 |
09 |      def jedz(self, pokarm): # metoda jedz
10 |          print(f'Niedzwiedz {self.rasa} zjadl {pokarm}')
11 |
12 |  # Tworzenie obiektu
13 |  >>> obiekt = Niedzwiedz('Brazowy', 'Brunatny', 160)
14 |  # Wywolanie metody jedz z obiektu
15 |  >>> obiekt.jedz('jerzyny')
16 |  Niedzwiedz Brunatny zjadl jerzyny
```

Destruktor

Destruktor

W języku Python, destruktor to specjalna metoda klasy o nazwie `__del__`. Destruktor jest używany do definiowania zachowania obiektu w momencie, gdy obiekt jest usuwany z pamięci, czyli gdy nie ma już referencji do niego i mechanizm garbage collection (zarządzania pamięcią) go usuwa.

Przykład destruktora

Przykład destruktora

```
01 |  class NowaKlasa:
02 |      def __init__(self, v1, v2): # konstruktor
03 |          self.pole1 = v1
04 |          self.pole2 = v2
05 |
06 |      def __del__(self): # destruktor
07 |          print(f"Obiekt klasy NowaKlasa usunięty")
08 |
09 |  # Tworzenie obiektu klasy NowaKlasa
10 |  obiekt = NowaKlasa(10, 20)
11 |
12 |  # Usunięcie obiektu
13 |  del obiekt  # Wywołuje destruktor dla obiektu
```

Przykład w języku Python

```
01 |  class Test:
02 |      def __del__(self):
03 |          print('Bye Class')
04 |
05 |  obj = Test()
06 |  obj2 = obj
07 |  del obj
08 |  del obj2
09 |  print('End')
```

self

self

W języku Python, `self` jest konwencjonalną nazwą pierwszego parametru metody, który odnosi się do obiektu, na którym metoda jest wywoływana. `self` reprezentuje instancję klasy, na której jest wywoływana metoda. Dzięki temu, klasa wie, na której konkretnej instancji wywołuje się metodę i może dostępować do atrybutów i metod tej konkretnej instancji.

Ciekawostka

Nazwa "self"pochodzi z konwencji nazewnictwa używanej w języku programowania Smalltalk, na podstawie którego został stworzony Python. W Smalltalku obiekt jest nazywany "self", a metoda jest wywoływana na rzecz tego obiektu. Python przejął tę konwencję i nazwał pierwszy argument metody "self".

*args oraz **kwargs

*args

`*args` to informacja, że parametry przekazane do funkcji, należy 'spakować', do zmiennej typu krotka. Nazwa `args` jest standardową nazwą w tym przypadku, jednak może być dowolna inna nazwa zmiennej np. `*a`.

**kwargs

`**kwargs` to informacja dla Python, że przekazywane parametry, są typu klucz:wartość i należy je 'spakować' oraz umieścić w zmiennej typu słownik.

Przykład *args

```
01 |  def funk(*args):
02 |      print("Liczba przekazanych parametrow:",len(args))
03 |      for arg in args:
04 |          print ("Wartosc:",arg)
05 |
06 |  lista = [1,3,5,6]
07 |
08 |  funk(1,lista,2,'xyz',3)
```

Przykład **kwargs

```
01 |  def funk(**kwargs):
02 |      print("Liczba przekazanych parametrow:",len(kwargs))
03 |
04 |      for key, item in kwargs.items():
05 |          print ("Klucz:", key, "Wartosc:", item)
06 |  funk(a=1,b=2,c=3)
```

Przykład *args oraz **kwargs

```
01 |  def funk(*args, **kwargs):
02 |
03 |      print("Liczba przekazanych parametrow *args:",len(↩
       args))
04 |      for arg in args:
05 |          print ("Args - Wartosc:", arg)
06 |
07 |      print("Liczba przekazanych parametrow **kwargs:",len(↩
       kwargs))
08 |      for key, item in kwargs.items():
09 |          print ("Kwargs - ", "Klucz:", key, "Wartosc:", ↩
       item)
10 |  funk(11,22,['x','y','z'],a=1,b=2,c=3)
```

Metody magiczne

Metody magiczne

Metody magiczne to specjalne metody, których nazwy zaczynają się i kończą podwójnym podkreśleniem, np. `__init__`, `__str__`, `__len__`, `__add__`.

UWAGA!

Deklarowanie własnych metod magicznych w Python jest możliwe i w pewnych przypadkach może być przydatne, jednak nie jest to zalecane ze względu na kilka powodów:

1. Niezgodność z konwencją
2. Nieprzewidywalne zachowanie
3. Niepotrzebne skomplikowanie kodu

Metody magiczne - porównania

- `__eq__(self,other)` – sposób użycia: `x == y`
- `__ne__(self,other)` – sposób użycia: `x != y`
- `__lt__(self,other)` – sposób użycia: `x < y`
- `__le__(self,other)` – sposób użycia: `x <= y`
- `__gt__(self,other)` – sposób użycia: `x > y`
- `__ge__(self,other)` – sposób użycia: `x >= y`

Metody magiczne - konwersja

- `__bool__(self)` – sposób użycia: `bool(x)` – konwersja do odpowiednika logicznego
- `__abs__(self)` – sposób użycia: `abs(x)`
- `__int__(self)` – sposób użycia: `int(x)` – konwersja do typu całkowitego
- `__float__(self)` – sposób użycia: `float(x)` – konwersja do typu rzeczywistego
- `__complex__(self)` – sposób użycia: `complex(x)` – konwersja do typu zespolonego
- `__round__(self,digits)` – sposób użycia: `round(x,digits)` – zaokrąglanie

Metody magiczne - operatory

```
__pos__(self) +x
__neg__(self) -x

__add__(self,other) x + y
__iadd__(self,other) x += y
__radd__(self,other) y + x

__sub__(self,other) x - y
__isub__(self,other) x -= y
__rsub__(self,other) y - x

__mul__(self,other) x * y
__imul__(self,other) x *= y
__rmul__(self,other) y * x

__mod__(self,other) x % y
__imod__(self,other) x %= y
__rmod__(self,other) y % x
```

```
__floordiv__(self,other) x  y
__ifloordiv__(self,other) x = y
__rfloordiv__(self,other) y  x

__truediv__(self,other) x  y
__itruediv__(self,other) x = y
__rtruediv__(self,other) y  x

__pow__(self,other) x ** y
__ipow__(self,other) x **= y
__rpow__(self,other) y ** x

__and__(self,other) x & y
__iand__(self,other) x &= y
__rand__(self,other) y & x

__or__(self,other) x | y
__ior__(self,other) x |= y
__ror__(self,other) y | x
```

Przeciążanie operatorów

Przeciążanie operatorów

Przeciążanie operatorów w języku Python to mechanizm, który umożliwia programiście zmianę sposobu działania wbudowanych operatorów (takich jak np. +, -, *,) dla własnych klas obiektów. Kiedy przeciążamy operator, definiujemy dla naszej klasy, jak ma zachowywać się dany operator w odniesieniu do obiektów tej klasy.

Przykład przeciążania

```
01 |  class Vector:
02 |
03 |      def __init__(self, x, y):
04 |          self.x = x
05 |          self.y = y
06 |
07 |      # Przeciazenie operatora dodawania
08 |      def __add__(self, other):
09 |          # Zwrocenie nowego obiektu klasy
10 |          return Vector(self.x + other.x, self.y + other.y)
11 |
12 |      def __str__(self):
13 |          return f"({self.x}, {self.y})"
```

Przykład przeciążania

```
01 | class MyClass:
02 |     def __init__(self, value):
03 |         self.value = value
04 |
05 |     def __add__(self, other):
06 |         if isinstance(other, MyClass):
07 |             return MyClass(self.value + other.value)
08 |         elif isinstance(other, int):
09 |             return MyClass(self.value + other)
10 |
11 | obj1 = MyClass(5)
12 | obj2 = MyClass(10)
13 | obj3 = obj1 + obj2  # Wywołuje __add__ zdefiniowane w
14 |                     # klasie MyClass
```

Wyjątki związane z przeciążaniem operatorów

- Omówienie sytuacji, w których przeciążanie operatorów może prowadzić do błędów:
  - Nieprawidłowe użycie operatorów dla określonych typów danych.
  - Niedokładna implementacja przeciążania operatorów, prowadząca do nieoczekiwanych wyników.
  - Konflikty między różnymi przeciążeniami operatorów w jednym kontekście.
- Zachowanie ostrożne przy używaniu przeciążania operatorów:
  - Unikanie nadmiernego przeciążania operatorów, co może prowadzić do utraty czytelności kodu.
  - Staranne testowanie przeciążonych operatorów dla różnych przypadków użycia.
  - Uwzględnianie konsystencji w przeciążaniu operatorów w obrębie aplikacji.
  - Dokumentowanie specjalnych metod używanych do przeciążania, aby ułatwić zrozumienie kodu innym programistom.

Przykłady praktyczne

- **Przeciążanie operatora dodawania:**
  - Błąd: Dodawanie obiektów różnych typów, które nie mają sensownej sumy.
  - Ostrożność: Sprawdzenie typów przed wykonaniem operacji dodawania.
- **Przeciążanie operatora indeksowania:**
  - Błąd: Brak obsługi sytuacji, gdy indeks jest poza zakresem.
  - Ostrożność: Dodanie logiki sprawdzającej poprawność indeksu przed dostępem do elementu.
- **Przeciążanie operatora równości:**
  - Błąd: Nieprawidłowa implementacja porównywania dla niestandardowych klas.
  - Ostrożność: Ustalenie jasnych reguł porównywania i testowanie na różnych przypadkach.
- **Przeciążanie operatora mnożenia:**
  - Błąd: Niezamierzone efekty uboczne, gdy operator jest używany w nieprzewidywalny sposób.
  - Ostrożność: Ograniczenie przeciążania do logicznych operacji związanych z danym typem.

Zalety i wady przeciążania operatorów

- **Przewagi przeciążania operatorów:**
  - Zwiększenie czytelności kodu: Przeciążanie operatorów może uczynić kod bardziej zwięzłym i czytelnym, zwłaszcza w przypadku operacji, które są powszechnie stosowane.
  - Naturalność: Pozwala na korzystanie z operatorów w sposób naturalny, podobny do matematycznych operacji.
  - Wsparcie dla interfejsów: Umożliwia implementację interfejsów i koncepcji znanych z matematyki lub innych dziedzin.
- **Ograniczenia przeciążania operatorów:**
  - Nadużycie i utrata czytelności: Nadmierne przeciążanie operatorów może prowadzić do utraty czytelności kodu i utrudniać zrozumienie jego działania.
  - Możliwość błędów: Niewłaściwe przeciążanie operatorów może prowadzić do nieintuicyjnego zachowania i błędów w kodzie.
  - Wzrost skomplikowania: Przeciążanie operatorów może skomplikować kod, zwłaszcza gdy jest używane nadmiernie.

Przykładowe zastosowania w bibliotekach

- **NumPy:**
  - Przeciążanie operatora dodawania (+) dla macierzy:
    - Umożliwia dodawanie macierzy za pomocą operatora +, co jest bardziej zwięzłe niż korzystanie z funkcji `numpy.add()`.
  - Przeciążanie operatora indeksowania (`[]`) dla tablic:
    - Pozwala na wygodne pobieranie i przypisywanie wartości do tablicy, np. `array[i] = value`.
- **Pandas:**
  - Przeciążanie operatora indeksowania (`[]`) dla obiektu `DataFrame`:
    - Ułatwia wybieranie kolumn danych, np. `df['kolumna']`, co jest bardziej czytelne niż korzystanie z metody `df.loc`.
  - Przeciążanie operatora dodawania (+) dla `DataFrame`:
    - Umożliwia dodawanie dwóch ram danych za pomocą operatora +, co może być bardziej intuicyjne.

Enkapsulacja

Enkapsulacja

Enkapsulacja to jeden z podstawowych konceptów programowania obiektowego, który polega na ukryciu szczegółów implementacyjnych obiektu przed innymi częściami programu. W języku Python enkapsulacja może być osiągnięta poprzez wykorzystanie modyfikatorów dostępu: publicznego, chronionego i prywatnego.

Modyfikatory dostępu

Atrybuty publiczne

Atrybuty publiczne są dostępne z dowolnego miejsca w programie, mogą być odczytywane i modyfikowane przez każdy kod, który ma dostęp do obiektu. W przypadku atrybutów publicznych nie używa się podkreślenia.

Atrybuty chronione

Atrybuty chronione są oznaczane pojedynczym podkreśleniem przed nazwą (np. "`_nazwa`"), co informuje programistę, że ten atrybut powinien być traktowany jako chroniony. Atrybuty chronione są dostępne tylko dla obiektów z tej samej klasy lub z klas dziedziczących po tej klasie.

Modyfikatory dostępu

Atrybuty prywatne

Atrybuty prywatne są oznaczane podwójnym podkreśleniem przed nazwą (np. "`__nazwa`"), co informuje programistę, że ten atrybut powinien być traktowany jako prywatny. Atrybuty prywatne są dostępne tylko wewnątrz klasy i nie powinny być bezpośrednio odczytywane lub modyfikowane przez obiekty innych klas.

Przykład modyfikatorów dostępu

```
01 |  class Zwierze:
02 |
03 |      def __init__(self, imie, gatunek):
04 |          self.imie = imie  # atrybut publiczny
05 |          self._gatunek = gatunek  # atrybut chroniony
06 |          self.__wiek = 0  # atrybut prywatny
07 |
08 |  >>> kot = Zwierze("Filemon", "kot")
09 |  >>> print(kot.imie)
10 |  Filemon
11 |  >>> print(kot._gatunek) # niezalecane uzycie
12 |  kot
13 |  >>> print(kot.__wiek)
14 |  AttributeError:'Zwierze' object has no attribute '__wiek'
```

Modyfikatory dostępu dla metod

Metody publiczne

Metody publiczne są dostępne z dowolnego miejsca w programie i mogą być wywoływane przez każdy kod, który ma dostęp do obiektu. Dostęp do metod publicznych nie jest ograniczony.

Metody chronione

Metody chronione są oznaczane pojedynczym podkreśleniem przed nazwą (np. "`_nazwa_metody`"), co informuje programistę, że ta metoda powinna być traktowana jako chroniona. Metody chronione są dostępne tylko dla obiektów z tej samej klasy lub z klas dziedziczących po tej klasie.

Modyfikatory dostępu dla metod

Metody prywatne

Metody prywatne są oznaczane podwójnym podkreśleniem przed nazwą (np. "`__nazwa_metody`"), co informuje programistę, że ta metoda powinna być traktowana jako prywatna. Metody prywatne są dostępne tylko wewnątrz klasy i nie powinny być bezpośrednio wywoływane przez obiekty innych klas.

Przykład modyfikatorów dostępu dla metod

```
01 |  class Zwierze:
02 |
03 |      def __init__(self, imie, gatunek):
04 |          self.imie = imie  # metoda publiczna
05 |          self._gatunek = gatunek  # metoda chroniona
06 |          self.__wiek = 0  # metoda prywatna
07 |
08 |      def podaj_gatunek(self):
09 |          return self._gatunek
10 |
11 |      def __zaktualizuj_wiek(self):
12 |          self.__wiek += 1
13 |
14 |  >>> kot = Zwierze("Filemon", "kot")
15 |  >>> print(kot.podaj_gatunek())
16 |  kot
17 |  >>> kot.__zaktualizuj_wiek()
18 |  AttributeError: 'Zwierze' object has no attribute
19 |  '__zaktualizuj_wiek'
```

Dlaczego ukrywanie metod jest istotne dla enkapsulacji

- **Izolacja implementacji:** Chroniąc metody jako chronione lub prywatne, ukrywamy szczegóły implementacyjne klasy przed innymi częściami programu, co zwiększa modularność.
- **Zabezpieczenie przed nieprawidłowym użyciem:** Ograniczając dostęp do metod tylko dla klasy, unikamy niekontrolowanego modyfikowania stanu obiektów z zewnątrz, co pomaga utrzymać spójność danych.
- **Zwiększenie bezpieczeństwa:** Ukrywanie metod minimalizuje ryzyko błędów wynikających z nieprawidłowego użycia, co zwiększa bezpieczeństwo kodu.
- **Ułatwienie refaktoryzacji:** Ukrywanie implementacji metod ułatwia zmiany kodu wewnątrz klasy bez wpływu na inne części programu, co wspiera refaktoryzację.
- **Zachowanie spójności w interfejsie publicznym:** Ukrywanie metod pozwala na utrzymanie spójności w interfejsie publicznym klasy, co jest korzystne dla stabilności i kompatybilności.

Zasada hermetyzacji

- **Hermetyzacja to zasada enkapsulacji:**
  - Hermetyzacja jest jednym z głównych założeń enkapsulacji w programowaniu obiektowym.
  - Oznacza, że klasy powinny być hermetyzowane, co polega na ukrywaniu implementacji wewnętrznej przed światem zewnętrznym.
- **Dlaczego hermetyzacja jest kluczowa:**
  - **Izolacja implementacji:** Chronienie szczegółów implementacyjnych klasy przed innymi częściami programu.
  - **Ochrona przed nieautoryzowanym dostępem:** Uniemożliwianie bezpośredniego dostępu do wewnętrznych atrybutów i metod.
  - **Zapewnienie spójności:** Utrzymywanie spójności obiektów poprzez kontrolowanie dostępu do ich stanu.

Zasada hermetyzacji c.d.

- **Implementacja hermetyzacji:**
  - **Modyfikatory dostępu:** Korzystanie z modyfikatorów dostępu, takich jak `private` czy `protected`, w językach, które je obsługują.
  - **Ukrywanie implementacji:** Oferowanie jedynie interfejsu publicznego, ukrywając szczegóły implementacyjne.
  - **Zastosowanie właściwości i getterów/setterów:** Kontrolowanie dostępu do atrybutów poprzez metody dostępowe.

Przykład enkapsulacji - Klasa samochodu

- **Cel:** Przedstawienie przykładu enkapsulacji na przykładzie klasy samochodu.
- **Implementacja klasy samochodu:**
  - Stworzenie klasy `Car` z prywatnymi atrybutami, takimi jak prędkość i poziom paliwa.
  - Atrybuty są oznaczone jako prywatne, na przykład za pomocą podwójnego podkreślenia (`__speed`, `__fuel_level`).
- **Dostęp tylko za pomocą interfejsu publicznego:**
  - Metody publiczne, takie jak `get_speed()` i `get_fuel_level()`, umożliwiają dostęp do prywatnych atrybutów.
  - Użytkownik klasy nie ma bezpośredniego dostępu do prywatnych atrybutów, co wprowadza kontrolę nad manipulacją nimi.

Dekorator

Dekorator

Dekorator to funkcja, która przyjmuje inną funkcję jako argument i zwraca nową funkcję z dodatkową funkcjonalnością lub modyfikującą istniejącą funkcję w inny sposób. Dekoratory są zwykle definiowane przy użyciu symbolu "@"w Pythonie, a następnie umieszczone przed definicją funkcji, którą chcemy udekorować.

Przykład dekoratora

```
01 |  def logowanie(func):
02 |      def wrapper(*args, **kwargs):
03 |          print(f"Rozpoczynam funkcje {func.__name__} z ↩
       argumentami {args}, {kwargs}")
04 |          wynik = func(*args, **kwargs)
05 |          print(f"Koncze funkcje {func.__name__}, wynik: {↩
       wynik}")
06 |          return wynik
07 |      return wrapper
08 |
09 |  @logowanie
10 |  def dodaj(a, b):
11 |      return a + b
12 |
13 |  >>> dodaj(2, 3)
14 |  Rozpoczynam funkcje dodaj z argumentami (2, 3), {}
15 |  Koncze funkcje dodaj, wynik: 5
```

Dekorator property

Dekorator property

Dekorator **property** w języku Python to narzędzie, które pozwala na definiowanie metod, które działają jak atrybuty, ale umożliwiają dodanie logiki związaną z odczytem, zapisem lub usunięciem wartości.

Wady i zalety property

- pozwala on na uzyskanie większej kontroli nad zachowaniem atrybutów klasy
- interfejs jest jednolity
- zachowanie czytelnego kodu
- może on wprowadzać pewne opóźnienia w działaniu programu przez wywoływanie gettera i settera

Przykład property

```
01 |  class Prostokat:
02 |      def __init__(self, a, b):
03 |          self.__a = a
04 |          self.__b = b
05 |
06 |      @property
07 |      def a(self):
08 |          return self.__a
09 |
10 |      @property
11 |      def b(self):
12 |          return self.__b
13 |
14 |      @a.setter
15 |      def a(self, value):
16 |          if value <= 0:
17 |              raise ValueError("Za mala dlugosc")
18 |          self.__a = value
19 |
20 |      ...
```

Przykład property

```
01 |         ...
02 |         @b.setter
03 |         def b(self, value):
04 |             if value <= 0:
05 |                 raise ValueError("Za mala dlugosc")
06 |             self.__b = value
07 |
08 |         @property
09 |         def pole(self):
10 |             return self.__a * self.__b
11 |
12 |         @property
13 |         def obwod(self):
14 |             return 2 * self.__a + 2 * self.__b
```

Przykład property

```
01 |  >>> p = Prostokat(2, 3)
02 |  >>> print(p.a)
03 |  2
04 |  >>> print(p.b)
05 |  3
06 |  >>> p.a = 4
07 |  >>> p.b = 5
08 |  >>> print(p.a)
09 |  4
10 |  >>> print(p.b)
11 |  5
12 |  >>> print(p.pole)
13 |  20
14 |  >>> print(p.obwod)
15 |  18
16 |  >>> p.a = 0
17 |  ValueError: Dlugosc boku musi byc wieksza od 0.
```

Przykłady złego użycia enkapsulacji

- **Zbyt silna enkapsulacja może utrudnić zrozumienie kodu:**
  - W niektórych przypadkach nadmierne ukrywanie szczegółów implementacyjnych może prowadzić do utraty jasności i zrozumienia kodu.
  - Przykładowo, ukrywanie prostych operacji za pomocą wielu warstw enkapsulacji może sprawić, że kod stanie się trudny do śledzenia.
- **Ważność zachowania umiaru w enkapsulacji:**
  - Enkapsulacja jest potężnym narzędziem, ale nadmierna ochrona danych i metod może skomplikować kod bez uzyskania znaczącej korzyści.
  - Warto zachować umiar, utrzymując równowagę między ochroną implementacji a zachowaniem czytelności kodu.
  - Enkapsulacja powinna być stosowana tam, gdzie ma sens i przynosi wartość, unikając nadmiernego skomplikowania kodu.

Rodzaje metod w programowaniu obiektowym

Metody instancji

Metody instancji (ang. instance methods): to metody, które działają na instancjach klasy i są wywoływane na konkretnej instancji za pomocą notacji kropkowej. W definicji metody jako pierwszy argument zawsze występuje `self`, co oznacza, że metoda działa na obiekcie, na którym została wywołana.

```
01 |  class Klasa:
02 |      def metoda_instancji(self, argument1, argument2):
03 |          # cialo metody
```

Rodzaje metod w programowaniu obiektowym

Metody klasowe (metody klasy)

Metody klasowe (ang. class methods): to metody, które działają na klasie, a nie na konkretnej instancji. Do definicji takiej metody używamy dekoratora `@classmethod`, a jako pierwszy argument metody przekazujemy obiekt klasy, zwykle nazywany `cls`.

```
01 |  class Klasa:
02 |      @classmethod
03 |      def metoda_klasowa(cls, argument1, argument2):
04 |          # cialo metody
```

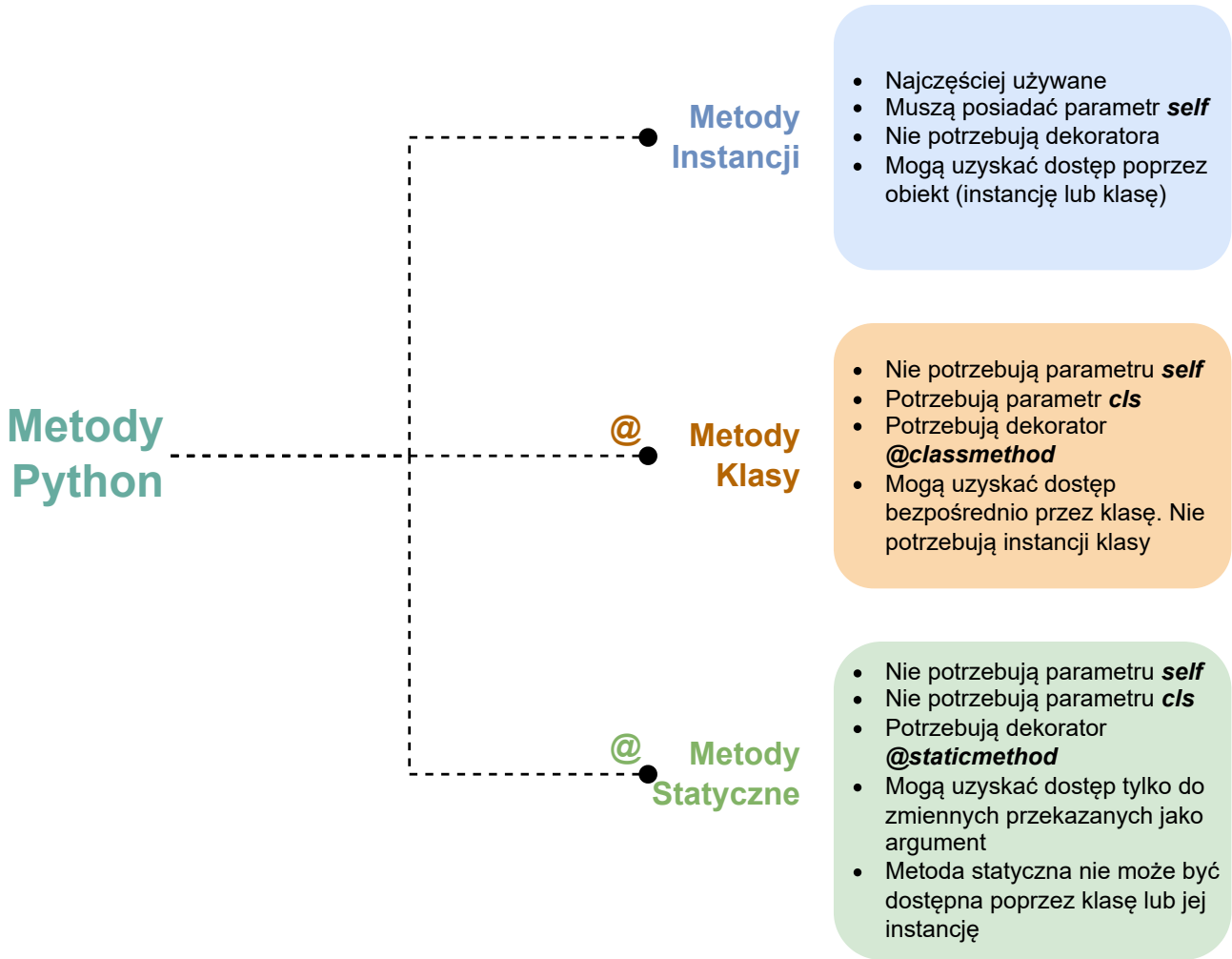
Rodzaje metod w programowaniu obiektowym

Metody statyczne

Metody statyczne (ang. static methods): to metody, które nie działają na klasie ani na instancji, tylko wykonują pewne operacje na przekazanych do nich argumentach. Do definicji takiej metody używamy dekoratora `@staticmethod`. Metoda ta nie przyjmuje argumentu `self` ani `cls`.

```
01 |  class Klasa:
02 |      @staticmethod
03 |      def metoda_statyczna(argument1, argument2):
04 |          # cialo metody
```

Porównanie rodzajów metod



Przykład wykorzystania metody klasy

```
01 |  from datetime import date
02 |  class Klasa:
03 |
04 |      def __init__(self, name, age):
05 |          self.name = name
06 |          self.age = age
07 |
08 |      @classmethod
09 |      def metoda_klasy(cls, name, brithYear):
10 |          return cls(name, date.today().year - birthYear)
11 |
12 |  >>> person1 = Klasa("Jan", 32)
13 |  >>> person2 = Klasa.metoda_klasy("Anna", 1985)
14 |  >>> print(person1.name, person1.age)
15 |  Jan 32
16 |  >>> print(person2.name, person2.age)
17 |  Anna 37
```

Rodzaje pól w programowaniu obiektowym

Pole instancji

Pole instancji (ang. instance variable) - to pole związane z instancją klasy i przechowujące jej dane. Każda instancja klasy ma swoją kopię tego pola i może je modyfikować. Pola instancji są definiowane w metodzie konstruktora klasy.

```
01 |  class Klasa:
02 |      def __init__(self, wartosc):
03 |          self.pole_instancji = wartosc # Pole instancji
```

Rodzaje pól w programowaniu obiektowym

Pole klasy

Pole klasy (ang. class variable) - to pole związane z klasą i przechowujące dane wspólne dla wszystkich instancji klasy. Pole klasy jest zwykle definiowane na poziomie klasy, a nie w metodzie konstruktora. Wszystkie instancje klasy mają dostęp do tego pola i mogą je modyfikować.

```
01 |  class Klasa:
02 |      pole_klasy = 4 # Pole klasy
03 |      def __init__(self, wartosc):
04 |          self.pole_instancji = wartosc # Pole instancji
```

Dziedziczenie

Dziedziczenie

Dziedziczenie to mechanizm programowania obiektowego, który umożliwia tworzenie nowych klas na podstawie istniejących klas. Klasa dziedzicząca (nazywana klasą pochodną, podklasą lub potomną) może odziedziczyć atrybuty i metody z klasy nadrzędnej (nazywanej klasą bazową lub nadrzędną), co pozwala na ponowne wykorzystanie kodu i zmniejszenie ilości powtarzającego się kodu.

Przykład dziedziczenia

```
01 |   class KlasaBazowa:
02 |       # cialo klasy bazowej
03 |
04 |   class KlasaPochodna(KlasaBazowa):
05 |       # cialo klasy pochodnej
```

Funkcja super

Funkcja super

`super()` to wbudowana funkcja w Pythonie, która umożliwia dostęp do metod klasy bazowej z klasy pochodnej. W dziedziczeniu w Pythonie, gdy klasa dziedziczy po innej klasie, `super()` może być użyte do wywołania konstruktora klasy bazowej lub metod zdefiniowanych w klasie bazowej.

Pierwszy sposób użycia

Pierwszy sposób wywołuje konstruktor klasy bazowej z klasy pochodnej. Można to zrobić, używając super() bez argumentów, co jest równoważne wywołaniu konstruktora klasy bazowej z argumentami przekazanymi do konstruktora klasy pochodnej. Przykład:

```
01 |  class KlasaBazowa:
02 |      def __init__(self, arg1, arg2):
03 |          self.arg1 = arg1
04 |          self.arg2 = arg2
05 |
06 |  class KlasaPochodna(KlasaBazowa):
07 |      def __init__(self, arg1, arg2, arg3):
08 |          super().__init__(arg1, arg2)
09 |          self.arg3 = arg3
```

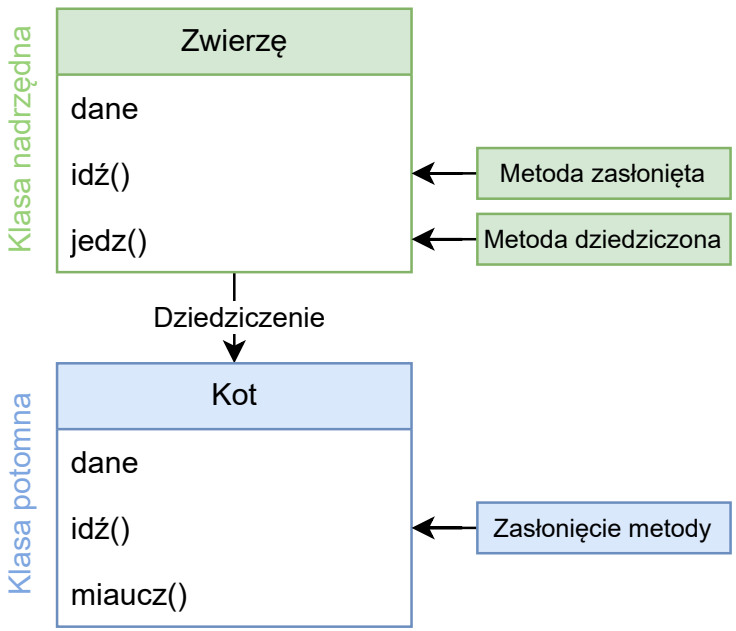
Drugi sposób użycia

Drugi sposób użycia `super()` polega na wywołaniu metody zdefiniowanej w klasie bazowej z klasy pochodnej. W tym przypadku, `super()` jest wywoływane z argumentami, które określają klasę pochodną i obiekt, na którym metoda powinna zostać wywołana. Przykład:

```
01 |  class KlasaBazowa:
02 |      def metoda(self):
03 |          print("Jestem metoda z klasy bazowej")
04 |
05 |  class KlasaPochodna(KlasaBazowa):
06 |      def metoda(self):
07 |          super(KlasaPochodna, self).metoda()
08 |          print("Jestem metoda z klasy pochodnej")
```

Przesłanianie metod (overriding)

Przesłanianie (overriding) to pojęcie związane z dziedziczeniem w programowaniu obiektowym. W Pythonie, gdy klasa dziedziczy po innej klasie, może nadpisywać (przesłaniać) metody tej klasy nadrzędnej, co pozwala na dostosowanie lub modyfikację zachowania tych metod w klasie potomnej.



Przykład

```
01 |  class A:
02 |      def some_method(self):
03 |          print("Metoda z klasy A")
04 |
05 |  class B(A):
06 |      def some_method(self):
07 |          print("Metoda z klasy B")
08 |
09 |  obj = B()
10 |  obj.some_method()  # Wywoła metodę z klasy B
```

Funkcja issubclass

Funkcja issubclass

Funkcja `issubclass()` w Pythonie służy do sprawdzania, czy jedna klasa jest podklasą innej klasy. Funkcja ta przyjmuje dwa argumenty: klasę potomną (lub krotkę klas potomnych) oraz klasę nadrzędną (lub krotkę klas nadrzędnych). Zwraca wartość logiczną `True`, jeśli klasa potomna jest podklasą klasy nadrzędnej lub którejś z jej klas nadrzędnych, a `False` w przeciwnym przypadku.

Przykład użycia

```
01 |  class Zwierze:
02 |      pass
03 |
04 |  class Kot(Zwierze):
05 |      pass
06 |
07 |  class Pies(Zwierze):
08 |      pass
09 |
10 |  >>> print(issubclass(Kot, Zwierze))
11 |  True
12 |  >>> print(issubclass(Pies, Zwierze))
13 |  True
14 |  >>> print(issubclass(Pies, Kot))
15 |  False
```

Modyfikatory dostępu w dziedziczeniu

W Pythonie istnieją trzy modyfikatory dostępu do pól:

- publiczne (brak przedrostka)
- chronione (przedrostek _)
- prywatne (przedrostek __)

W przypadku dziedziczenia, pola publiczne i chronione są dziedziczone i mogą być dostępne z poziomu klasy dziedziczącej. Pola prywatne nie są dziedziczone i nie są bezpośrednio dostępne z klasy dziedziczącej. Jednakże, można uzyskać dostęp do pól prywatnych poprzez użycie metody publicznej w klasie bazowej, która zwraca wartość pola prywatnego.

Przykład

```
01 |  class KlasaBazowa:
02 |      def __init__(self):
03 |          self.publiczne_pole = "Publiczne pole"
04 |          self._chronione_pole = "Chronione pole"
05 |          self.__prywatne_pole = "Prywatne pole"
06 |
07 |
08 |  class KlasaDziedziczaca(KlasaBazowa):
09 |      def __init__(self):
10 |          # wywolanie konstruktora klasy bazowej
11 |          super().__init__()
12 |          # dostepne z poziomu klasy dziedziczacej
13 |          print(self.publiczne_pole)
14 |          # dostepne z poziomu klasy dziedziczacej
15 |          print(self._chronione_pole)
16 |          # pole prywatne nie jest dziedziczone
17 |          # print(self.__prywatne_pole)
```

Dziedziczenie wielokrotne

Dziedziczenie wielokrotne

Dziedziczenie wielokrotne w języku Python polega na dziedziczeniu jednej klasy po więcej niż jednej klasie bazowej. W wyniku dziedziczenia wielokrotnego, klasa dziedzicząca posiada metody i pola zdefiniowane w klasach bazowych. Aby zdefiniować dziedziczenie wielokrotne, należy podać listę klas bazowych oddzielonych przecinkami w nawiasach okrągłych po nazwie klasy dziedziczącej.

Przykład

```
01 |  class KlasaA:
02 |      def metoda_a(self):
03 |          print("Metoda A")
04 |
05 |  class KlasaB:
06 |      def metoda_b(self):
07 |          print("Metoda B")
08 |
09 |  class KlasaC(KlasaA, KlasaB):
10 |      pass
11 |
12 |  >>> obiekt = KlasaC()
13 |  >>> obiekt.metoda_a()
14 |  Metoda A
15 |  >>> obiekt.metoda_b()
16 |  Metoda B
```

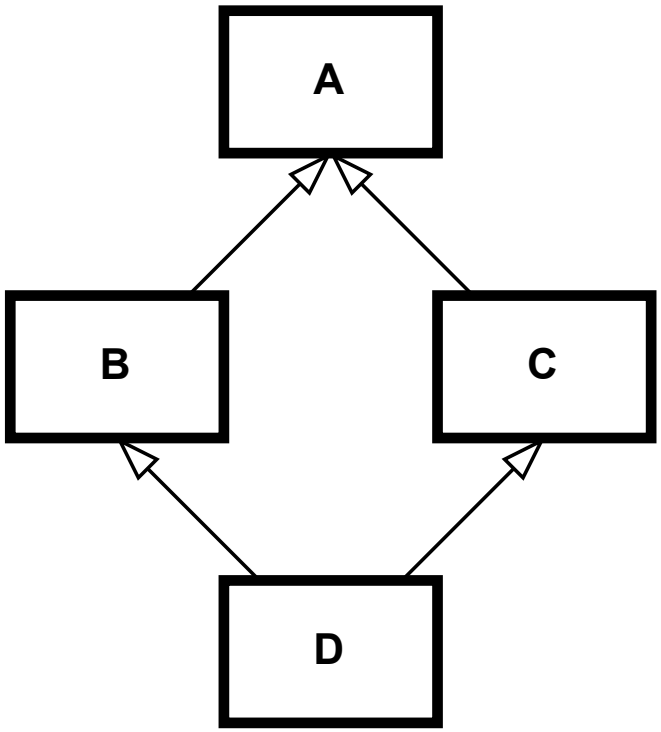
Method Resolution Order (MRO)

Method Resolution Order

Method Resolution Order (MRO) w Pythonie odnosi się do kolejności, w jakiej interpreter Pythona szuka metod w klasach dziedziczących. MRO jest istotny, gdy klasa dziedziczy po wielu klasach nadrzędnych (multiple inheritance). W praktyce oznacza to, że interpreter Pythona najpierw sprawdzi klasę dziedziczącą, a jeśli tam nie znajdzie poszukiwanej metody, to będzie kontynuować przeszukiwanie w klasach nadrzędnych, zgodnie z ustalonym MRO.

Problem diamentu

Problem diamentu to niejednoznaczność powstająca, gdy dwie klasy B i C dziedziczą po A, a klasa D dziedziczy zarówno po B, jak i C. Jeśli istnieje metodę w A, którą B i C zastąpili , a D jej nie zastąpi, to którą wersję metody odziedziczy D: wersję B czy C?



Przykład

```
01 |  class A:
02 |      def foo(self):
03 |          print("Metoda foo w klasie A")
04 |
05 |  class B(A):
06 |      def foo(self):
07 |          print("Metoda foo w klasie B")
08 |
09 |  class C(A):
10 |      def foo(self):
11 |          print("Metoda foo w klasie C")
12 |
13 |  class D(B, C):
14 |      pass
15 |
16 |  obj = D()
17 |  obj.foo()
18 |  print(D.__mro__) # Sprawdź MRO dla klasy D
19 |  print(D.mro()) # Lub
```

Klasa abstrakcyjna

Klasa abstrakcyjna

Klasa abstrakcyjna to taka klasa, której instancji nie można stworzyć. Służy ona do tego aby z niej dziedziczyć oraz implementować te metody, które zostały oznaczone jako abstrakcyjne.

Metoda abstrakcyjna

Metoda abstrakcyjna to taka metoda, która posiada sygnaturę, jednakże nie posiada ona implementacji. Klasa, która zawiera przynajmniej jedną metodę abstrakcyjną jest klasą abstrakcyjną, ze względu na to, że nie można tworzyć obiektów, dla których brakuje implementacji pewnych operacji.

Moduł abc

Moduł abc

Moduł `abc` zapewnia infrastrukturę do definiowania abstrakcyjnych klas bazowych (ABC) w Pythonie. Istnieje kilka modułów podstawowych Pythona, które bazują na implementacji modułu `abc`. Przykładem takiego modułu jest moduł `collections`, w którego skład wchodzi również podmoduł `collections.abc` zawierający implementacji pewnych instancji.

Klasa abc.ABC

Klasa abc.ABC

Klasa `abc.ABC` jest to klasa pomocnicza, której klasa `abc.ABCMeta` jest metaklasą. Za pomocą tej klasy można utworzyć abstrakcyjną klasę bazową, której instancję następnie możemy odziedziczyć w klasach potomnych.

Dekoratory klasy abstrakcyjnej

Dekoratory klasy abstrakcyjnej

W celu nadania cechy metodzie wskazującej na jej abstrakcję, należy posłużyć się jednym z następujących dekoratorów:

- `@abstractmethod`
- `@abstractclassmethod`
- `@abstractstaticmethod`

Uwaga

Dekoratory abstrakcyjne dotyczące metod klasy oraz metod statycznych są przestarzałe od wersji Python 3.3. Dlatego też zaleca się używanie zwykłych dekoratorów metod klasy oraz metod statycznych.

Przykładowa implementacja klasy abstrakcyjnej

```
01 |  from abc import ABC, abstractmethod
02 |
03 |  class AbstractClassExample(ABC):
04 |      def __init__(self, value):
05 |          self.value = value
06 |          super().__init__()
07 |
08 |      @abstractmethod
09 |      def do_something(self):
10 |          pass
```

Przykład dziedziczenia klasy abstrakcyjnej

```
01 |  from abc import ABC, abstractmethod
02 |
03 |  class AbstractClassExample(ABC):
04 |      def __init__(self, value):
05 |          self.value = value
06 |          super().__init__()
07 |
08 |      @abstractmethod
09 |      def do_something(self):
10 |          pass
11 |
12 |  class DoAdd42(AbstractClassExample):
13 |      def do_something(self):
14 |          return self.value + 42
```

Polimorfizm

Polimorfizm w kontekście programowania obiektowego oznacza zdolność różnych obiektów do odpowiadania na te same operacje lub metody w sposób zgodny z ich typem, bez względu na konkretną klasę do której należą. Innymi słowy, różne klasy mogą udostępniać wspólny interfejs lub podobne metody, co umożliwia korzystanie z nich w jednolity sposób.

Przykład:

- Stworzymy hierarchię klas zwierząt.
- Każda klasa ma metodę `make_sound()`, ale różne klasy implementują tę metodę w inny sposób.

Przykład polimorfizmu

```
01 |  class Animal:
02 |      def make_sound(self):
03 |          pass
04 |
05 |  class Dog(Animal):
06 |      def make_sound(self):
07 |          return "Bark!"
08 |
09 |  class Cat(Animal):
10 |      def make_sound(self):
11 |          return "Meow!"
12 |
13 |  def animal_speak(animal):
14 |      return animal.make_sound()
15 |
16 |  dog = Dog()
17 |  cat = Cat()
18 |
19 |  print(animal_speak(dog))  # "Bark!"
20 |  print(animal_speak(cat))  # "Meow!"
```

Polimorfizm z użyciem klas abstrakcyjnych

```
01 | from abc import ABC, abstractmethod
02 |
03 | class Shape(ABC):
04 |     @abstractmethod
05 |     def area(self):
06 |         pass
07 |
08 | class Circle(Shape):
09 |     def __init__(self, radius):
10 |         self.radius = radius
11 |
12 |     def area(self):
13 |         return 3.14 * self.radius**2
14 |
15 | class Rectangle(Shape):
16 |     def __init__(self, width, height):
17 |         self.width = width
18 |         self.height = height
19 |
20 |     def area(self):
21 |         return self.width * self.height
```