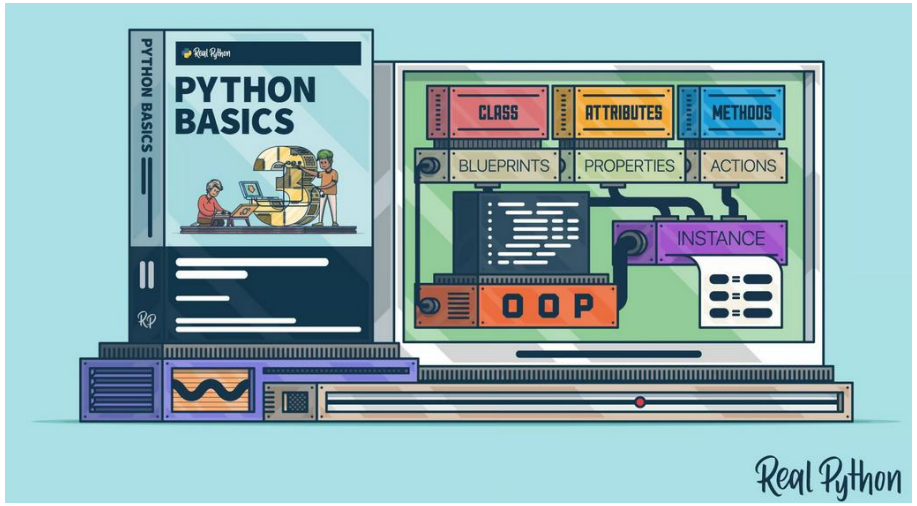


Programowanie obiektowe - Założenia

M@я3k Pųð€£kØ

Programowanie w Pythonie

Założenia/filary programowania obiektowego:

- Abstrakcja
- Hermetyzacja
- Dziedziczenie
- Polimorfizm

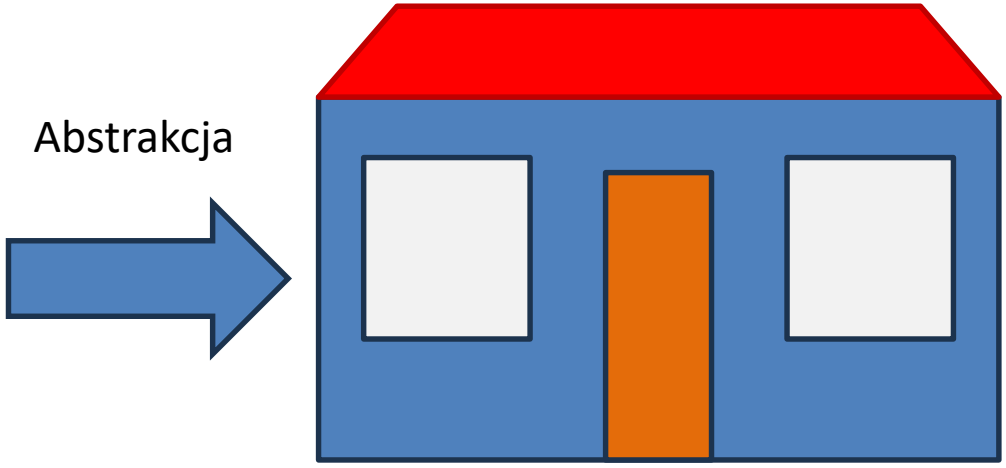
Abstrakcja

- Abstrakcja polega na ukrywaniu lub pomijaniu mało istotnych informacji a skupieniu się na wydobyciu informacji, które są niezmienne i wspólne dla pewnej grupy obiektów.
- Abstrakcja to rodzaj uproszczenia rozpatrywanego problemu (klasy/obiektu).
- Umożliwia skupienie się na ogólnym obrazie bez zagłębiania się w szczegóły.

Abstrakcja

- Umożliwia skupienie się na ogólnym obrazie bez zagłębiania się w szczegóły.
  - Abstrakcja w wypadku klasy `Dom` polega na tym, że trzeba wyodrębnić cechy wspólne dla wszystkich obiektów klasy `Dom`: mają dach, ściany, okna. Stoją na jakiejś działce.
  - Nieistotne dla większości użytkowników jest ułożenie rur i przewodów w ścianach, liczba dachówek w dachu, ilość farby jaką zostały pomalowane ściany, ustawienie mebli.

Abstrakcja

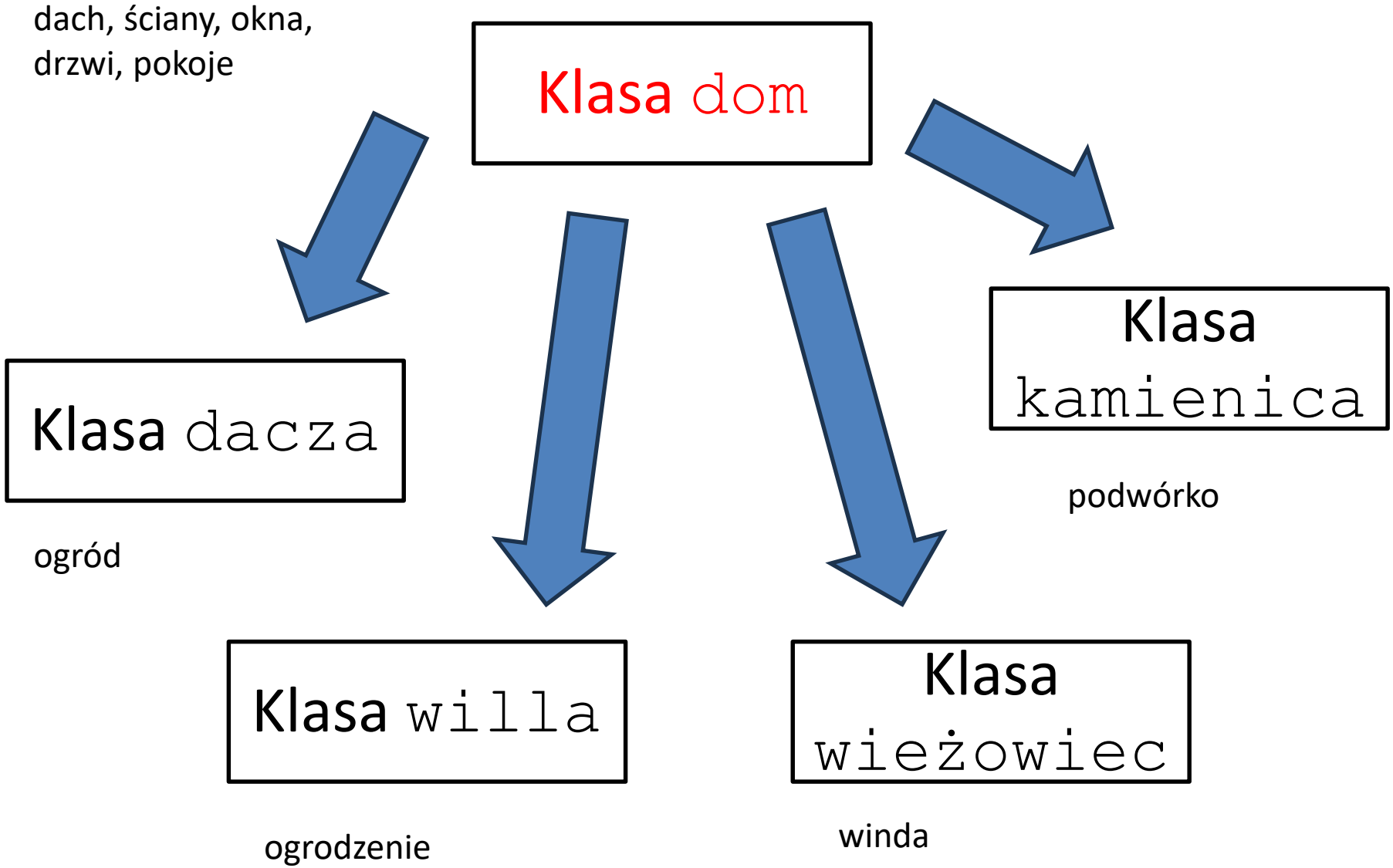


Klasa abstrakcyjna

- Klasa abstrakcyjna jest bazą dla innych klas.
- Zawiera metody wspólne dla klas tworzonych w oparciu o nią (tzw. klas potomnych, tj. dziedziczących z niej), a także nie posiada swoich bezpośrednich reprezentacji (nie tworzy się jej instancji).

- Klasa abstrakcyjna w Pythonie jest szczególnie ważna w kontekście tworzenia interfejsów, ponieważ język ten nie posiada w swojej składni struktur służących bezpośrednio do ich budowy.
  - Za pomocą klasy abstrakcyjnej możliwe jest zdefiniowanie interfejsu, tzn. określenie zbioru wszystkich metod, jakie muszą posiadać klasy dziedziczące.

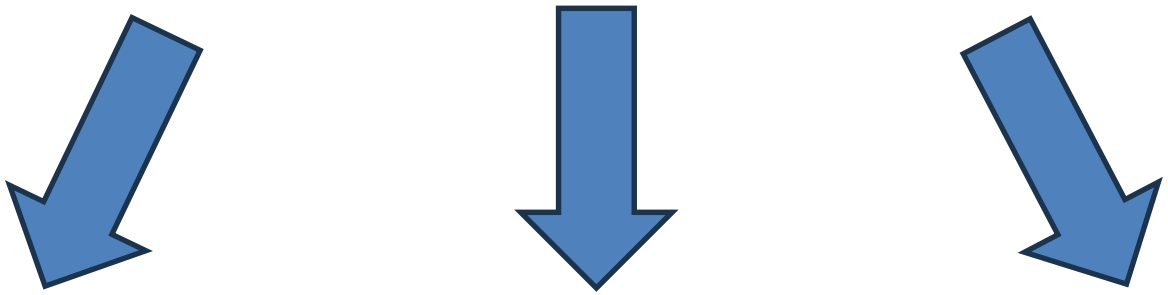
Przykład klasy abstrakcyjnej



Hermetyzacja

- Hermetyzacja (enkapsulacja) to ukrywanie nieistotnych informacji na temat obiektu.
  - zminimalizowanie efektów jego modyfikacji
  - oddzieleniu tego co zawiera i co może zrobić obiekt od tego jak jest zbudowany i jak to robi.
- Hermetyzacja ma za zadanie uniemożliwienie metodom niepowołanym dostępu do wnętrza obiektu i modyfikację pól, które nie powinny być dostępne dla każdego.

Klasy dostępności



Publiczna	Chroniona	Prywatne
• Dostęp mają wszyscy	• Dostęp mają klasy dziedziczące	• Dostęp ma tylko ta klasa

Hermetyzacja w Pythonie

- W języku Python nie ma typowej hermetyzacji obiektowej. Wszystkie pola i metody są dostępne spoza obiektu.
- Zasady nazewnictwa:
  - **Pojedyncze podkreślenie _**
  - Jest wyświetlane,
  - może się zmienić (poprzez `nazwaklasy._element`)
  - środowisko programistyczne nie podpowiada go.
  - **Podwójne podkreślenie __**
  - Nie jest wyświetlane
  - można się do niego odwołać (dla nazwy `__element`) poprzez `nazwaklasy.__element`.
  - środowisko programistyczne nie podpowiada go.

Hermetyzacja - przykład

```
class Dom:
    Powierzchnia = 100
    _Cena = 1000000
    __IloscPokoi = 6

    def wyswietl(self):
        print (self.__IloscPokoi)

~~print (Dom.Cena)~~
print (Dom._Cena)
Dom._Cena = 10
print (Dom._Cena)

~~print (Dom.__IloscPokoi)~~

Willa = Dom ()
Willa.wyswietl ()
```

_Cena Pole prywatne pozwalające zabezpieczyć wartość wewnątrz

_ _IloscPokoi Pole ukrywające wartość wewnątrz

Metoda pozwalająca wyświetlić wartość zabezpieczoną wewnątrz obiektu

Nie można wyświetlić nazwy pola bez _

Wyświetlenie pola prywatnego z _

Pole prywatne z _ można zmodyfikować

Nie można wyświetlić nazwy pola z _ _

Pole prywatne z _ _ można wyświetlić przy użyciu odpowiedniej metody

Dziedziczenie

- Dziedziczenie to mechanizm pozwalający klasie (potomnej) przejmować właściwości (pola) i zachowania (metody) od klasy bazowej.
- Dzięki dziedziczeniu możemy tworzyć nowe klasy w oparciu o już istniejące, bez potrzeby implementowania funkcjonalności już zaimplementowanych w klasach bazowych.

- Klasa potomna dziedziczy po klasie bazowej.
- Zyskuje wszystkie publiczne i chronione składowe klasy bazowej, a także może definiować własne unikalne pola i metody.
- Pozwala na tworzenie logicznych zależności między klasami.
- Zamiast powtarzać ten sam kod, można go umieścić w klasie bazowej i pozwolić, by wiele klas dziedziczyło te same funkcje.

Dziedziczenie

Zalety	Wady
Przejrzystość: Tworzenie hierarchii klas odzwierciedla relacje między pojęciami w modelowanym świecie.	**Złożoność**: Nadmierne dziedziczenie, zwłaszcza wielokrotne, może prowadzić do skomplikowanych hierarchii, co utrudnia testowanie i debugowanie.
Wydajność: Unika się pisania powtarzającego się kodu, co skraca czas tworzenia programu.	**Kruche zależności**: W dużych systemach, modyfikacje klasy bazowej mogą nieoczekiwanie wpłynąć na wszystkie klasy, które po niej dziedziczą.
Elastyczność: Ułatwia modyfikację i rozbudowę oprogramowania, ponieważ można rozszerzyć lub nadpisać metody klasy bazowej.	**Kompozycja**: Alternatywnym podejściem jest kompozycja, czyli budowanie nowych klas z istniejących obiektów. Jest często uważana za bardziej elastyczną i mniej podatną na błędy niż głębokie dziedziczenie, szczególnie w przypadku tworzenia bibliotek i frameworków.

Idea dziedziczenia

Klasa `prostokąt`

- czworobok
- 4 kąty proste
- boki parami równe
- 2 przekątne przecinające się w połowie
- Punkt przecięcia przekątnych środkiem okręgu opisanego na prostokącie
- Posiada dwie osie symetrii
- Przekątna dzieli prostokąt na dwa przystające trójkąty prostokątne

Ponieważ każdy kwadrat jest prostokątem, możliwe jest dziedziczenie

Klasa `kwadrat`

- Wszystkie boki są równe
- Punkt przecięcia przekątnych środkiem okręgu wpisanego w kwadrat
- Posiada cztery osie symetrii.

Dziedziczenie w pythonie

- W Pythonie dziedziczenie implementuje się poprzez umieszczenie nazw klas bazowych w nawiasach okrągłych podczas definicji klasy podrzędnej.

class klasa_podrzędna (klasa_nadrzędna):

```
class kwadrat (prostokąt)
```

- Python obsługuje dziedziczenie pojedyncze (po jednej klasie) i wielokrotne (po wielu klasach, oddzielonych przecinkami).

class klasa_podrzędna (klasa_nadrzędna1, 2, 3):

```
class kwadrat (prostokąt, romb, deltoid)
```

Dziedziczenie w pythonie

- Klasa nadrzędna to klasa, która już istnieje i posiada pewien zestaw cech.
- Klasa podrzędna (potomna) to nowa klasa, która dziedziczy po klasie nadrzędnej, może nadpisywać istniejące cechy lub dodawać nowe.

- `super()`:
- Funkcja, która umożliwia dostęp do metod klasy nadrzędnej z poziomu klasy potomnej, co jest kluczowe, np. przy wywoływaniu konstruktora klasy nadrzędnej `__init__`.

Dziedziczenie - przykład

```
import math
class prostokąt:     #klasa nadrzędna
    def __init__(self, a, b):
        self.a = a
        self.b = b
    def przekatna (self):
        d = math.sqrt (a**2 + b**2)

class kwadrat (prostokąt):   #klasa podrzędna
        #konstruktor klasy kwadrat
    def __init__(self, a):
        # Wywołanie konstruktora klasy nadrzędnej
        super().__init__(a, 1)
        self.a = a
    def przekatna (self):
        # Nadpisanie metody klasy nadrzędnej
        d = self.a * math.sqrt (2)
         print (d)

A = kwadrat (5)      #Utworzenie obiektu A klasy kwadrat
A.przekatna()        #7.071067811865475
```

Dziedziczenie po 2 klasach – przykład cz. 1

```
import math
class prostokąt:     #klasa nadrzędna 1
    def __init__(self, a, b):
        self.a = a
        self.b = b
                # metoda klasy 1
def przekatna (self):
        d = math.sqrt (a**2 + b**2)

class romb:     #klasa nadrzędna 2
    def __init__(self, a):
        self.a = a
                # metoda klasy 2
def obwod (self):
        obw = 4 * a
        print (obw)
```

Dziedziczenie po 2 klasach – przykład cz. 2

```
 #klasa podrzędna po 2 klasach nadrzędnych
class kwadrat (prostokąt, romb):
      #konstruktor klasy kwadrat
    def __init__(self, a):
        # Wywołanie konstruktora klas nadrzędnych
        super().__init__(a, 1)
        self.a = a

    def przekatna (self):
# Nadpisanie metody klasy nadrzędnej
        d = self.a * math.sqrt (2)
        print (d)

A = kwadrat (5)  #Utworzenie obiektu klasy kwadrat
A.przekatna()     #7.071067811865475
A.obwod ()        #metoda klasy 2 - wynik 20
```

Polimorfizm

- Polimorfizm to mechanizm, który pozwala na użycie tej samej nazwy funkcji, metody lub operatora dla obiektów różnych klas, pozwalając na różnorodne działanie bez konieczności znajomości ich dokładnego typu.
- Jedna nazwa może obsługiwać wiele form (typów) obiektów, co czyni kod bardziej elastycznym, łatwiejszym w zarządzaniu i rozwijaniu.

Polimorfizm

- Polimorfizm pozwala traktować różnorodne dane w ten sam sposób, w zależności od kontekstu. W trakcie wykonywania programu automatycznie są znajdowane i interpretowane odpowiednie metody zależne od rodzajów danych.
- Przykładem jest operator dodawania +, który może być wykorzystany z różnymi typami danych np. int, float, bool, complex

```
x = 2 + 5 = 7
x = 2.5 + 3.6 = 6.1
x = True + False
x = complex (5,4) + complex (3,4)
```

Polimorfizm w pythonie

- W Pythonie polimorfizm jest realizowany poprzez zdefiniowanie odpowiedniej metody w klasie.
- Ułatwienie są też listy i inne struktury dynamiczne pozwalające na elastyczne traktowanie pól i metod.

Przykład polimorfizmu

```
class Punkt_2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
                        # Metoda przesun
      def przesun (self, lista):
        self.x = self.x+lista[0]
        self.y = self.y+lista[1]

class Punkt_3D:
    def __init__(self, a, b, c):
        self.x = a
        self.y = b
        self.z = c
                        # Metoda przesun
    def przesun (self, lista):
        self.x = self.x+ lista[0]
        self.y = self.y+ lista[1]
        self.z = self.z+ lista[2]
```

```
        # Funkcja działająca na dowolnym obiekcie, który ma metodę przesun
def przesunięcie (punkt, lista):
        punkt.przesun (lista)

        # lista – dynamiczna struktura danych
lista = [3, 3, 3]

A = Punkt_2D (5,5)
B = Punkt_3D (2,2,2)

# Realizacja funkcji przesuniecie na obiektach różnych klas
przesunięcie (A, lista)
przesunięcie (B, lista)

print (A.x)
print (A.y)
print (B.x)
print (B.y)
print (B.z)
```