Politechnika Wrocławska

Języki Skryptowe

Programowanie obiektowe w Python

Marcin Jodłowiec

2 kwietnia 2024

Politechnika Wrocławska

Agenda

- Abstrakcja
  - Klasy i instancje
  - Właściwości
- Enkapsulacja
- Związki generalizacja-specjalizacja i dziedziczenie
- Abstrakcyjne klasy bazowe: moduł abc
- Polimorfizm dynamiczny a kacze typowanie
- Metody magiczne
- Enumeracje
- Metody instancyjne i klasowe. @staticmethod i @classmethod

Programowanie obiektowe

Klasy i instancje I

- **Klasa** – typ zdefiniowany przez użytkownika opisujący stan i zachowanie instancji.
- **Instancja** – wystąpienie, **obiekt**.
- Weryfikacja typu:
  - `type()`
  - `isinstance()`

```
1  >>> a = 5
2  >>> type(a)
3  <class 'int'>
4  >>> isinstance(a, int)
5  True
6  >>> isinstance(a, str)
7  False
8  >>> isinstance(a, object)
9  True
```

- Python: *everything is an object* – wszystko jest obiektem.

Politechnika Wrocławska

Charakterystyka klas w Pythonie

- Wywołanie klasy (analogicznie do funkcji) powoduje jej **zainstancjonowanie**.
- Klasa może mieć **atrybuty**, tzn. obiekty przypisane do nazw wewnątrz klasy.
- Atrybuty klasy, do których dowiązano funkcje, nazwyane są **metodami**.
- Metody mogą mieć specjalne nazwy pre- i postfiksowane dwoma podkreśleniami (np. `__str__`, `__init__`), które mają specjalne zanaczenie – metody specjalne lub **metody magiczne**.
- Klasa może **dziedziczyć** z jednej lub wielu klas, tzn. delegować do innych klas w celu poszukiwania atrybutów.

Abstrakcja. Definiowanie i instancjonowanie klas I

Instrukcja `class`

- Definiowanie klasy:

```
1  class MyClass:
2      pass
```

- Definiowanie klasy z metodą `__init__`:

```
1  class Animal:
2      def __init__(self, name, species, age):
3          self.name = name
4          self.species = species
5          self.age = age
6
7      def make_sound(self):
8          print("The animal makes a sound.")
```

- Instancjonowanie klasy

```
1  >>> obj = MyClass()
2  >>> burek = Animal("Burek", "dog", 2)
```

Abstrakcja. Definiowanie i instancjonowanie klas II

Instrukcja `class`

- Atrybuty klasowe a instancyjne:

```
1  class Car:
2      number_of_wheels = 4
3
4      def __init__(self, model, year):
5          self.model = model
6          self.year = year
7
8  >>> c1 = Car("Ford Fiesta", 1998)
9  >>> c2 = Car("Fiat 126p", 1980)
10 >>> c1.model
11 'Ford Fiesta'
12 >>> c2.model
13 'Fiat 126p'
14 >>> Car.number_of_wheels = 5
15 >>> c1.number_of_wheels
16 5
17
18 >>> c1.plate = "DW 12345"
```

Abstrakcja. Właściwości I

- Właściwości (ang. *properties*) – atrybuty instancji „zarządzane” przez funkcje.
- Pozwalają na realizację atrybutów wyliczeniowych lub oprogramowanie dodatkowej logiki
- zalecany sposób na eksponowanie publicznych atrybutów danych
- Rodzaje metod:
  - *getter* (obligatoryjny)
  - *setter*
  - *deleter*
- składnia:

```
attrib = property(fget=None, fset=None, fdel=None, doc=None)
```

Abstrakcja. Właściwości II

- Przykład definicji właściwości:

```
1  class Rectangle:
2     def __init__(self, width, height):
3        self.width = width
4        self.height = height
5
6     def area(self):
7        return self.width * self.height
8
9     area = property(area, doc='area of the rectangle')
10
11 rect = Rectangle(10,20)
12 print(rect.area)
```

Abstrakcja. Właściwości III

- Składnia dekoratorowa `@property`, ustawianie *settera*

```
 1  import math
 2
 3  class Rectangle:
 4      def __init__(self, width, height):
 5          self.width = width
 6          self.height = height
 7
 8      @property
 9      def area(self):
10          return self.width * self.height
11
12      @area.setter
13      def area(self, value):
14          scale = math.sqrt(value/self.area)
15          self.width *= scale
16          self.height *= scale
```

Politechnika Wrocławska

Enkapsulacja

- Brak trybów widoczności i enkapsulacji (np. private, protected w C++, Java).
- Mechanizm zamiany nazw (ang. *name mangling*).
- atrybuty postaci `__ident` zamieniane są na `_cname__ident`, gdzie cname to nazwa klasy.

```
1  >>> obj = ClassWithPrivateField(10)
2  >>> obj.__internal
3  Traceback (most recent call last):
4      File "<stdin>", line 1, in <module>
5  AttributeError: 'ClassWithPrivateField' object has no attribute '__internal'
6  >>> obj._ClassWithPrivateField__internal
```

- Konwencja – identyfikatory prywatne można poprzedzać pojedynczym podkreśleniem, ale interpreter zapewnia mechanizmu uniemożliwiającego dostęp.

Związki generalizacja-specjalizacja i dziedziczenie I

- Związek generalizacja-specjalizacja (*nadklasa-podklasa*) – strukturalne wiązanie klas, którego konsekwencją jest **dziedziczenie** stanu i zachowania

```
1  class Animal:
2      def __init__(self, name):
3          self.name = name
4
5      def make_sound(self):
6          print("")
7
8  class Dog(Animal):
9      def make_sound(self):
10         print("Hau!")
11
12 class Cat(Animal):
13     def make_sound(self):
14         print("Miau!")
```

```
1  >>> dog1,cat1 = Dog("Burek"),Cat("Filemon")
2  >>> print(dog1.name)
3  Burek
4  >>> dog1.make_sound()
5  Hau!
6  >>> print(cat1.name)
7  Filemon
8  >>> cat1.make_sound()
9  Miau!
```

Związki generalizacja-specjalizacja i dziedziczenie II

- `issubclass(A, B)` – test, czy klasa `A` jest podklasą `B`
- Kolejność szukania referencji atrybutów: `C.mro()`
  (ang. *Method Resolution Order (MRO)*)

```
>>> class MaineCoon(Cat): pass
...
>>> MaineCoon.mro()
[<class '__main__.MaineCoon'>, <class '__main__.Cat'>, <class '__main__.Animal'>, <
      class 'object'>]
```

- Delegacja do metod klasy bazowej

```
class Derived(Base):
    def __init__(self):
        super().__init__()
        self.anotherattribute = 45
```

```
class Base:
    def __init__(self):
        self.anattribute = 23

class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.anotherattribute = 45
```

Związki generalizacja-specjalizacja i dziedziczenie III

- Wielodziedziczenie (ang. *Multiple inheritance*)

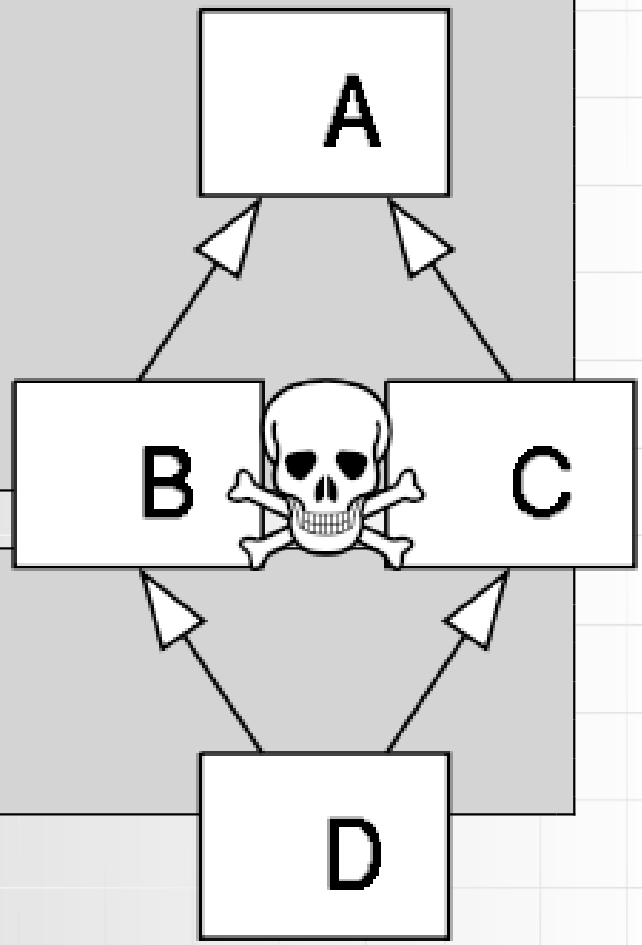
```
1  class Shape:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6  class Rectangle(Shape):
7      def __init__(self, x, y, width, height):
8          super().__init__(x, y)
9          self.width = width
10         self.height = height
11
12 class Circle(Shape):
13     def __init__(self, x, y, radius):
14         super().__init__(x, y)
15         self.radius = radius
16
17 class RoundedRectangle(Rectangle, Circle):
18     def __init__(self, x, y, width, height, radius):
19         Rectangle.__init__(self, x, y, width, height)
20         Circle.__init__(self, x, y, radius)
```

Związki generalizacja-specjalizacja i dziedziczenie IV

- Problem diamentu (ang. *Deadly diamond of death*)

```
1  class A:
2      def met(self):
3          print('A.met')
4  class B(A):
5      def met(self):
6          print('B.met')
7          super().met()
8  class C(A):
9      def met(self):
10         print('C.met')
11         super().met()
12 class D(B,C):
13     def met(self):
14         print('D.met')
15         super().met()
```

```
1  >>> d = D()
2  >>> d.met()
3  D.met
4  B.met
5  C.met
6  A.met
```



Jak działa Method Resolution Order w Pythonie

Algorytm C3

- C3 – algorytm linearyzacji MRO.
- Monotoniczność

  *A MRO jest **monotoniczny** gdy prawdą jest, co następuje: jeżeli C_1 musi poprzedzać C_2 na liście linearyzacji klasy C, to C_1 poprzedza C_2 w linearyzacji dowolnej podklasy C.*

- MRO klasy C szuka metody w klasie C, a następnie w jej nadklasach, od definiowanych lewej do prawej.
- Jeśli nie można znaleźć metody, szuka w klasie `object`, która jest nadklasą wszystkich klas.
- Niektóre hierarchie nie pozwalają na skonstruowanie MRO, np.

```
1  class A: pass
2
3  class B(A): pass
4
5  class C(A,B): pass
```

spowoduje `TypeError: Cannot create a consistent method resolution`, gdyż MRO po monotonizacji byłby niezgodny z regułami dziedziczenia:

$$C \to \cancel{A} \to B \to A$$

Abstrakcyjne klasy bazowe: moduł abc

- Moduł abc – abstrakcyjne klasy bazowe – nieinstancjonowalność

```
>>> import abc
>>> class MyAbstractClass(metaclass=abc.ABCMeta):
...     @abc.abstractclassmethod
...     def my_abstract_classmethod(cls):
...         pass
...
...     @abc.abstractstaticmethod
...     def my_abstract_staticmethod():
...         pass
...
...     @abc.abstractproperty
...     def my_abstract_property(self):
...         pass
...
...     @abc.abstractmethod
...     def my_abstract_method(self):
...         pass
...
>>> a = MyAbstractClass()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class MyAbstractClass with abstract methods
       my_abstract_classmethod, my_abstract_property, my_abstract_staticmethod
```

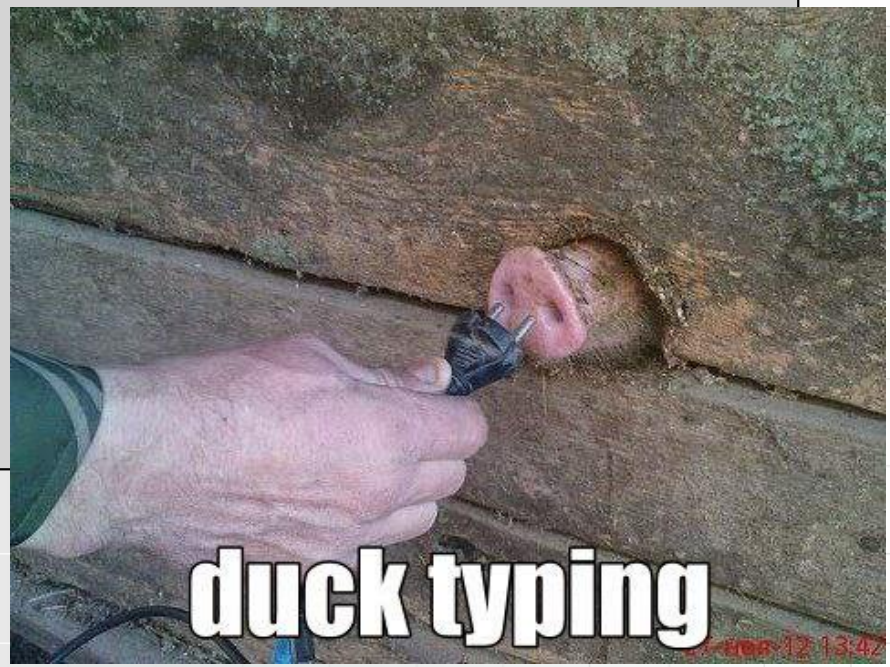
Polimorfizm dynamiczny a kacze typowanie I

- **Polimorfizm** – wielopostaciowość – możliwość wykorzystania różnych obiektów z zachowaniem jednolitego interfejsu
- w statycznych implementacjach języków programowania (m.in. zorientowanych obiektowo) – silnie związany z **Liskov Substitution Principle**
- w Pythonie – kacze typowanie (ang. *duck typing*)
  - „*jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczką*”

Polimorfizm dynamiczny a kacze typowanie II

- Polimorfizm może, ale **nie musi** być związany z dziedziczeniem:

```
1  class Car:
2      def start(self):
3          print("Starting car")
4
5  class Fan:
6     def start(self):
7        print("Starting fan")
8
9  class Garage:
10     def __init__(self, items):
11         self.items = items
12
13     def use(self):
14         for item in self.items:
15            item.start()
16
17 car = Car()
18 fan = Fan()
19
20 garage = Garage([car, fan])
21 garage.use()
```



Magiczne metody: wbudowany typ `Object` I

- `object` – baza wszystkich wbudowanych typów i klas

```
1  >>> class MyClass:
2  ...   pass
3  ...
4  >>> MyClass.__bases__
5  (<class 'object'>,)
6  >>> int.__bases__
7  (<class 'object'>,)
```

- definiuje magiczne (specjalne) metody implementujące domyślną semantykę obiektów

`__init__`, `__new__`, `__del__` – tworzenie, inicjalizacja,finalizacja bezpośrednich instancji obiektu i inicjalizacja jego wartości. Po wywołaniu klasy `C`, python uruchamia `__new__` (konstruktor) z przekazanymi argumentami, a następnie `__init__` (inicjalizator) z tymi samymi argumentami.

Magiczne metody: wbudowany typ `Object` II

Przykład – implementacja singletona z wykorzystaniem metody __new__

```
1  class Singleton:
2      _instance = None
3
4      def __new__(cls, *args, **kwargs):
5          if not cls._instance:
6              cls._instance = super().__new__(cls, *args, **kwargs)
7          return cls._instance
```

`__str__`, `__repr__` – renderowanie instancji jako ciągi znaków. `str` – ciągi znaków przeznaczone do czytania przez czlowieka, `repr` – debugowanie, REPL, etc.

`__lt__`, `__eq__`, `__gt__`, `__le__` ... – kontrola porównywania instancji

Magiczne metody: wbudowany typ `Object` III

`__gettattr__`, `__setattr__`, `__delattr__` – referowanie, wiązanie i odwiązywanie atrybutów instancji

```
1  class Person:
2      def __init__(self, name, age):
3          self.name = name
4          self.age = age
5
6      def __getattr__(self, attr):
7          print(f"Attribute '{attr}' does not exist!")
8
9      def __setattr__(self, attr, value):
10         print(f"Setting attribute '{attr}' to value '{value}'")
11         object.__setattr__(self, attr, value)
```

```
1  >>> person = Person("Karolina", 18)
2  Setting attribute 'name' to value 'Karolina'
3  Setting attribute 'age' to value '18'
4  >>> person.age = 23
5  Setting attribute 'age' to value '23'
6  >>> person.address
7  Attribute 'address' does not exist!
```

Magiczne metody: wbudowany typ `Object` IV

`__call__` – implementacja wywoływania instancji (funktor)

```
1  class Adder:
2      def __init__(self, num):
3          self.num = num
4
5      def __call__(self, x):
6          return self.num + x
```

```
1  >>> adder = Adder(10)
2  >>> adder(21)
3  31
```

`__add__`, `__sub__`, `__mul__`, `__and__`, `__not__`, `...` przeładowywanie operatorów

```
1  class Point:
2      def __init__(self, x, y):
3          self.x = x
4          self.y = y
5
6      def __add__(self, other):
7          return Point(self.x + other.x, self.y + other.y)
```

`...` https://docs.python.org/3/reference/datamodel.html#special-method-names

Metody magiczne dla kontenerów I

- instancja może pełnić rolę kontenera (np. sekwencji, odwzorowania)
- dla zapewnienia użyteczności, kontenery powinny dostarczać metod:
  - `__len__`, `__getitem__`, `__contains__`, `__iter__`
  - w przypadku mutowalnych kontenerów: `__setitem__`, `__delitem__`
- dla klas bazujących na listach i słownikach jako nadklasę warto wykorzystać `collection.UserDict` i `collection.UserList`

```
1  from collections import UserDict
2
3  class MyDict(UserDict):
4      def __setitem__(self, key, value):
5          super().__setitem__(key.lower(), value)
```

```
1  >>> d = MyDict()
2  >>> d['Foo'] = 'bar'
3  >>> print(d) # {'foo': 'bar'}
4  {'foo': 'bar'}
```

Metody magiczne dla kontenerów II

```
1  class Person:
2      def __init__(self, name):
3          self.name = name
4
5
6  class PeopleList:
7      def __init__(self, people):
8          self.people = people
9
10     def __len__(self): return len(self.people)
11
12     def __iter__(self): return iter(self.people)
13
14     def __contains__(self, name):
15         return any(person.name == name for person in self.people)
16
17     def __getitem__(self, key):
18         if isinstance(key, int):
19             return self.people[key]
20         else:
21             filtered_people = [person for person in self.people if person.name == key]
22             if len(filtered_people) == 0:
23                 raise KeyError(f"No person with name '{key}'")
24             else:
25                 return filtered_people[0]
```

Enumeracje I

- Enumeracje – wyliczenia, zbiory powiązanych wartości służące do symbolicznego oznaczania wartości
- Moduł `enum`, klasa `enum.Enum`

```
1  from enum import Enum, auto
2  class Color(Enum):
3      RED = 1
4      GREEN = 2
5      BLUE = 3
6
7  class Size(Enum):
8      XS = auto()
9      S = auto()
10     M = auto()
11     L = auto()
12     XL = auto()
```

- Skrócona składnia

```
1  from enum import Enum
2  Color = Enum('Color', ('RED', 'GREEN', 'BLUE'))
3  Size = Enum('Size', 'XS S M L XL')
```

Enumeracje II

- Flagi – (enum.Flag) – używane do definiowana enumeracji, które mogą być łączone przy użyciu operatorów bitowych (&, |, ^, ~). Bity z wartością 0 oznaczają elementy flagi, które są wyłączone, analogicznie: 1 – włączone. Elementy flagi są kolejnymi potęgami 2.

```
1  from enum import Flag, auto
2
3  class Permissions(Flag):
4      READ = auto()
5      WRITE = auto()
6      EXECUTE = auto()
7
8  my_permissions = Permissions.READ | Permissions.WRITE
9
10 if my_permissions & Permissions.READ:
11     print("Mam uprawnienie do odczytu.")
12
13 my_permissions = my_permissions & ~Permissions.WRITE
```

Metody instancyjne vs metody klasowe i statyczne I

- Metody z pierwszym parametrem `self` są metodami wiązanymi do instancji

```
>>> class Dog:
...     def bark(self): print("Hau hau!")
...
>>> Dog.bark
<function Dog.bark at 0x7f16bb5fed40>
>>> d = Dog()
>>> d.bark
<bound method Dog.bark of <__main__.Dog object at 0x7f16bb65e3b0>>
>>> Dog.bark(d)
```

- `staticmethod` i `classmethod` – metody niewiązane do instancji

Metody instancyjne vs metody klasowe i statyczne II

- `staticmethod` – metoda statyczna
  - metoda wołana na klasie lub dowolnej instacji klasy
  - brak ograniczeń w zakresie parametrów
  - zwykła funkcja dowiązana do atrybutu klasy

```
1  class Animal:
2      def __init__(self, name, species):
3          self.name = name
4          self.species = species
5
6      @staticmethod
7      def is_mammal(species):
8          mammals = ["dog", "cat", "lion", "cow", "sheep", "elephant"]
9          return species in mammals
```

```
1  >>> Animal.is_mammal("dog")
2  True
3  >>> my_pet = Animal("Luna", "dog")
4  >>> my_pet.is_mammal("dog")
5  True
```

Metody instancyjne vs metody klasowe i statyczne III

- `classmethod` – metoda klasowa
  - metoda wołana na klasie lub dowolnej instacji klasy
  - pierwszy parametr dowiązany do klasy (konwencja nazwy `cls`)
  - metoda wywołana z podklasy wiąże z podklasą

```
1  class Base:
2
3      @classmethod
4      def hello(cls):
5          print('Hello from', cls.__name__)
6
7  class Derived(Base):
8     pass
```

```
1  >>> Base.hello()
2  Hello from Base
3  >>> Derived.hello()
4  Hello from Derived
```

Politechnika Wrocławska

To już wszystko na dziś!