Kurs rozszerzony języka Python

Wykład 4.

Marcin Młotkowski

29 października 2024

Plan wykładu

1. Klasy i obiekty
2. Atrybuty klas i obiektów
   - Właściwości (properties)
3. Wyjątki
4. Model obiektowy
   - Obiekty w Pythonie
   - Specjalne atrybuty obiektów
   - Obiekty jako kolekcje
   - Badanie stanu obiektu — refleksje
   - Obiekt uniwersalny

Plan wykładu

Deklaracja klasy

Przykłady

```
class Figura:
    """Pierwsza klasa"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Definicja metody

class Figura, cd. definicji

```
...
    def info(self):
        print(self.x, self.y)

    def zmien(self, x, y):
        self.x = x
        self.y = y
```

Tworzenie obiektów i wywołanie metod

Przykład

```
o = Figura(1, -1)
o.info()
o.zmien(2,3)
o.info()
```

Dziedziczenie

```
class Okrag(Figura):
    """Okrąg"""
    def __init__(self):
        self.x, self.y, self.r = 0, 0, 1

    def info(self):
        print(f"x = {self.x}, y = {self.y}, r = {self.r}")
```

Wywołanie konstruktora z nadklasy

```
    def __init__(self):
        super().__init__(2.0, 3.0)
        # Figura.__init__(self, 2.0, 3.0)
        # super(self.__class__, self).__init__()
```

Metody wirtualne

class Figura

```
    def info(self):
        ...

    def przesun(self, dx, dy):
        self.info()
        self.x, self.y = self.x + dx, self.y + dy
        self.info()
```

Metody wirtualne

class Figura

```
    def info(self):
        ...

    def przesun(self, dx, dy):
        self.info()
        self.x, self.y = self.x + dx, self.y + dy
        self.info()
```

```
okrag = Okrag();
okrag.przesun(10,15)
```

Wielodziedziczenie

```
class Samochod:
    def naprzod(self):
```

```
class Okret:
    def naprzod(self):
```

```
class Amfibia(Samochod, Okret):
```

Wielodziedziczenie

```
class Samochod:
    def naprzod(self):
```

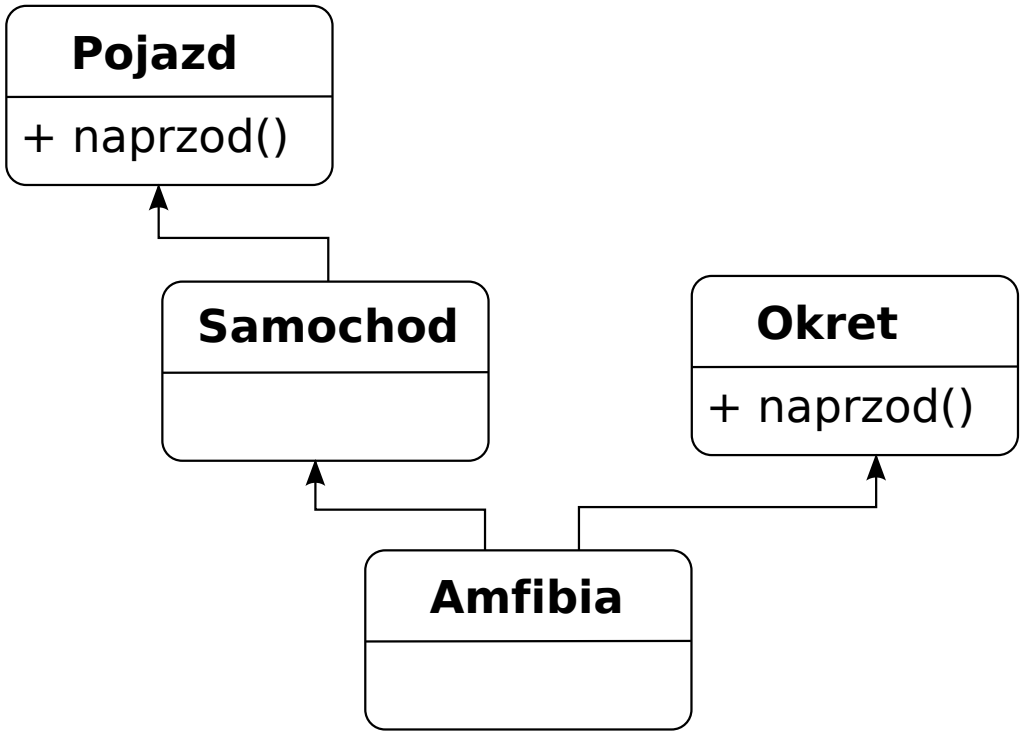
```
class Okret:
    def naprzod(self):
```

```
class Amfibia(Samochod, Okret):
```

Zagadka

```
amf = Amfibia()
amf.naprzod()
```

Rozwiązywanie niejednoznaczności przez Pythona



MRO: method resolution order

W głąb, od lewej do prawej

Modyfikacja wyszukiwania metody

```
super(Samochod, self).naprzod()
```

Równość obiektów

Operatory tożsamości obiektów

is

is not

Równość obiektów

Operatory tożsamości obiektów

is

is not

Popularny idiom

```
if x is None:

if x is not None:
```

Funkcja id()

Operator is korzysta z funkcji id(), która w CPythonie oznacza adres obiektu w pamięci.

Sprawdzanie klasy obiektu

```
isinstance(None, NoneType)
```

Sprawdzanie klasy obiektu

```
isinstance(None, NoneType)
```

Zwraca prawdę również jeśli jest podklasą wskazanej klasy.

Plan wykładu

Pola statyczne klasy

```
class Okrag:
    pi = 3.1415

    def __init__(self):
        self.r = 2.71

    def pole(self):
        return Okrag.pi * self.r **2
```

Pola statyczne klasy

```
class Okrag:
    pi = 3.1415

    def __init__(self):
        self.r = 2.71

    def pole(self):
        return Okrag.pi * self.r **2
```

Odwołanie do pól statycznych klasy

```
print(Okrag.pi)
o = Okrag()
print(o.pi)
```

Pola obiektu

```
class Okrag:
    pi = 3.1415
    self.x, self.y = 0, 0
    def __init__(self):
        self.x, self.y = 0, 0
```

Zmienne

Fakt 1.

Zmienne można dodawać dynamicznie

Zmienne

Fakt 1.

Zmienne można dodawać dynamicznie

Nowa zmienna modułu

modul.nowa_zmienna = 'Nowa zmienna'

Zmienne

Fakt 1.

Zmienne można dodawać dynamicznie

Nowa zmienna modułu

```
modul.nowa_zmienna = 'Nowa zmienna'
```

Nowa zmienna obiektu

```
o = Figura()
o.nowe_pole = "Nowe pole"
```

Zmienne

Fakt 1.

Zmienne można dodawać dynamicznie

Nowa zmienna modułu

```
modul.nowa_zmienna = 'Nowa zmienna'
```

Nowa zmienna obiektu

```
o = Figura()
o.nowe_pole = "Nowe pole"
```

Nowa zmienna

```
pi = 3.1415
```

Zmienne

Fakt 2.

Zmienne można usuwać dynamicznie

Zmienne

Fakt 2.

Zmienne można usuwać dynamicznie

Przykład

```
x = 'x'
del x
```

Zmienne "prywatne"

Zmienną (pseudo)prywatną jest zmienna poprzedzona dwoma podkreśleniami i zakończona co najwyżej jednym podkreśleniem (dotyczy modułów i klas).

Np.

```
__zmiennaPrywatna
```

Metody statyczne i metody klasy

```
class Klasa:
    @staticmethod
    def dodawanie(a, b):
        return a + b

    @classmethod
    def utworz(cls):
        return cls()
```

Te metody mogą być wywoływane przez klasy i obiekty. Argumentem metody klasy jest klasa, w której jest zdefinowana metoda.

Czysta metoda klasy

```
class Klasa:
    def dodawanie(a, b):
        return a + b
```

Metody tego obiektu nie mogą wywołać `dodawanie()` bez kwalifikatora; muszą dać kwalifikator `Klasa.dodawanie`.

Przykład z https://realpython.com/blog/python

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    @classmethod
    def margherita(cls):
        return cls(["mozzarella", "tomatoes"])

    @classmethod
    def prosciutto(cls):
        return cls(["mozzarella", "tomatoes", "ham"])
p = Pizza.margheritta()
```

Klasa Temperatura

Skale temperaturowe: °C, °R, °F, °N, K, ...

```
class Temperature:

    def __init__(self, temperature=0):
        self._temperature = temperature
```

Właściwość Celsius

```
@property
def celsius(self):
    return self._temperature

@celsius.setter
def celsius(self, newtemp):
    self._temperature = newtemp
```

Właściwość Fahrenheit

```
@property
def fahrenheit(self):
    return (self._temperature * 1.8) + 32


@fahrenheit.setter
def fahrenheit(self, newtemp):
    self._temperature = (newtemp - 32)/1.8
```

Przykłady

```
t = Temperature(37)
print(t.fahrenheit)
t.fahrenheit = 0
print(t.celsius)
```

Inne

Klasy abstrakcyjne: moduł `abc`: Abstract Base Classes

Plan wykładu

Wyjątki

- Mechanizm przepływu sterowania

Wyjątki

- Mechanizm przepływu sterowania
- Wyjątki to obiekty

Obsługa wyjątków

```
try:
    f = open("plik"[10] + '.py', 'r')
except IOError:
    print("Błąd wejścia/wyjścia")
except IndexError as x:
    print(x)
except:
    print("Nieznany wyjątek")
finally:
    f.close()
```

Klauzula else

```
try:
    print(2/n)
except:
    print("Nieudane dzielenie")
else:
    print("Udane dzielenie")
```

Zgłaszanie wyjątków

raise

```
raise RuntimeError("Stało się coś strasznego!")
```

Własne obiekty wyjątków

```
class MojWyjatek(Exception):
  pass
```

Plan wykładu

Wszystko jest obiektem.

Wszystko jest obiektem.

```
type(5)
x = 5
x.__class__
x.__bases__
```

Klasa object

Klasa object jest nadklasą wszystkich klas.

Klasa object

Klasa object jest nadklasą wszystkich klas.

```
help(object)
```

Zadanie

Implementacja klasy wektorów `Vector`:

- operatory arytmetyczne

  $v1 = Vector([1, 0, 0])$
  $v2 = Vector([0, 1, 0])$
  $v3 = v1 + v2$

- str(Vector([0, 0 1])): <0, 0, 1>
- len(Vector([0, 0 1])): 3

Implementacja wektorów

```
class Vector:
  def __init__(self, lista):
      self.value = lista
```

Implementacja wektorów

```
class Vector:
  def __init__(self, lista):
      self.value = lista
  def __add__(self, arg):
    res = Vector([x + y for x, y in
          zip(self.value, arg.value)])
    return res
```

Wykorzystanie

```
v1 = Vector([1, 0, 3])
v2 = Vector([0, 2, 0])
print(v1 + v2)
```

Inne standardowe metody

__mul__ — mnożenie
__sub__ — odejmowanie
__truediv__ — dzielenie
__mod__ — reszta z dzielenia

Tak zdefiniowane operatory zachowują standardowe priorytety.

Postać napisowa

```
>>> print(Vector([1,2,3]))
<__main__.Vector instance at 0xb7eabdec>
```

Postać napisowa

```
>>> print(Vector([1,2,3]))
<__main__.Vector instance at 0xb7eabdec>
```

class Vector

```
  def __str__(self):
      return '<'
             + ', '.join((str(x) for x in self.value))
             + '>'
```

Postać napisowa

```
>>> print(Vector([1,2,3]))
<__main__.Vector instance at 0xb7eabdec>
```

class Vector

```
  def __str__(self):
      return '<'
             + ', '.join((str(x) for x in self.value))
             + '>'
```

```
>>> print(Vector([1,2,3]))
<1, 2, 3>
```

Własności kolekcji

Pożądane cechy kolekcji

- Indeksowany dostęp do danych k[4]
- Obsługa poprzez iteratory for–in
- rozmiar kolekcji len

Dostęp indeksowany

Implementacja akcesorów w klasie `Vector`

```
def __getitem__(self, index):
    return self.value[index]

def __setitem__(self, index, value):
    self.value[index] = value
```

Zastosowanie

```
>>> print(v1[k])
>>> v1[k] = k
```

Pozostałe własności kolekcji

Usuwanie elementu za pomoca del

```
def __delitem__(self, index):
    del self.value[index]
```

Pozostałe własności kolekcji

Usuwanie elementu za pomoca del

```
def __delitem__(self, index):
    del self.value[index]
```

Długość kolekcji: len

```
def __len__(self):
    return len(self.value)
```

Równość a identyczność

```
va = Vector([0,0,1])
vb = Vector([0,0,1])

va == vb
va is vb
id(va) == id(vb)
```

Operator porównania ==

```
class Vector:

    def __eq__(self, other):
        if len(self.value) != len(other):
            return False
        for i in range(len(self)):
            if self.value[i] != other.value[i]:
                return False
        return True
```

Wynik

```
va = Vector([0,0,1])
vb = Vector([0,0,1])

va == vb
va is vb
id(va) == id(vb)
```

Stan obiektu/modułu

- 'Napis'.__class__
- Figura.__doc__
- Figura.__dict__
- plik.__file__
- __name__

Słowniki symboli

Zmienne (oraz nazwy funkcji) w czasie działania programu są przechowywane w słowniku.

- dir()
- __dict__

Funkcja standardowa dir()

Co robi dir

Zwraca listę dostępnych nazw. Jeśli nie podano argumentu, to podaje listę symboli w lokalnym słowniku.

Funkcja standardowa dir()

Co robi dir

Zwraca listę dostępnych nazw. Jeśli nie podano argumentu, to podaje listę symboli w lokalnym słowniku.

>>> dir(Vector([1,2,3]))

['__add__', '__cmp__', '__delitem__', '__doc__', '__getitem__', '__init__', '__len__', '__module__', '__setitem__', '__str__', 'iter', 'next', 'value']

>>> Vector([1,2, 3]).__dict__

{'value': [1, 2, 3]}

Przydatność słowników

```
if 'nazwa' in obj.__dict__:
    print (obj.nazwa)
```

```
if "__str__" in dir(obj):
    print (str(obj))
```

Uniwersalny obiekt

Obiekt uniwersalny: ma wszystkie pola i implementuje dowolną metodę.

Implementacja uniwersalnego obiektu

Implementacja klasy

```
class Uniwersalna(object):
```

Implementacja uniwersalnego obiektu

Implementacja klasy

```
class Uniwersalna(object):
```

Implementacja dostępu do atrybutów

```
  def __getattr__(self, name):
      print("Odwołujesz się do atrybutu", name)
      return self

  def __setattr__(self, name, val):
      print(f"Przypisanie {name} wartości {val}")
```

Implementacja uniwersalnego obiektu

Implementacja klasy

```
class Uniwersalna(object):
```

Implementacja dostępu do atrybutów

```
    def __getattr__(self, name):
        print("Odwołujesz się do atrybutu", name)
        return self

    def __setattr__(self, name, val):
        print(f"Przypisanie {name} wartości {val}")
```

Wszystkie metody

```
    def __call__(self, *args):
        print("Wywołano metodę z argumentami", args)
```