Programowanie i projektowanie obiektowe

Obiekty i klasy w Pythonie

Paweł Daniluk

Wydział Fizyki

Jesień 2015

Klasy i obiekty

Klasy w implementacji odpowiadają klasom projektowym.

Obiekty są instancjami klas.

Klasy i obiekty

Klasy w implementacji odpowiadają klasom projektowym.

Obiekty są instancjami klas.

Przynależność do klasy określa zakres odpowiedzialności obiektów.

Każdy obiekt należy do pewnej klasy.

Klasy i obiekty

Klasy w implementacji odpowiadają klasom projektowym.

Obiekty są instancjami klas.

Przynależność do klasy określa zakres odpowiedzialności obiektów.

Każdy obiekt należy do pewnej klasy.

Klasa określa funkcjonalności (metody) obiektów.

Każdy obiekt odpowiada za wartości swoich atrybutów.

Klasy i obiekty

Klasy w implementacji odpowiadają klasom projektowym.

Obiekty są instancjami klas.

Przynależność do klasy określa zakres odpowiedzialności obiektów.

Każdy obiekt należy do pewnej klasy.

Klasa określa funkcjonalności (metody) obiektów.

Każdy obiekt odpowiada za wartości swoich atrybutów.

Metody określone przez klasę odwołują się do atrybutów przechowywanych w obiekcie.

Klasy i obiekty

Klasy w implementacji odpowiadają klasom projektowym.

Obiekty są instancjami klas.

Przynależność do klasy określa zakres odpowiedzialności obiektów.

Każdy obiekt należy do pewnej klasy.

Klasa określa funkcjonalności (metody) obiektów.

Każdy obiekt odpowiada za wartości swoich atrybutów.

Metody określone przez klasę odwołują się do atrybutów przechowywanych w obiekcie.

Czy klasy mogą być obiektami?

Najprostszy obiekt

```
class A:
    pass
```

Klasa `A` nie ma żadnych metod, ale może mieć atrybuty.

```
>>> a=A()
>>> a.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: A instance has no attribute 'x'
>>> a.x=1
>>> a.x
1
>>>
```

Atrybuty

W Pythonie atrybuty obiektów działają podobnie jak zmienne. Tworzymy je przez pierwsze przypisanie.

Atrybuty

W Pythonie atrybuty obiektów działają podobnie jak zmienne. Tworzymy je przez pierwsze przypisanie.

Czy to oznacza, że nie można określić w definicji klasy zakresu odpowiedzialności za przechowywanie danych?

Atrybuty

W Pythonie atrybuty obiektów działają podobnie jak zmienne. Tworzymy je przez pierwsze przypisanie.

Czy to oznacza, że nie można określić w definicji klasy zakresu odpowiedzialności za przechowywanie danych?

Dobre pytanie...

Metody

Metody są definiowane w klasach.

```
class Lew:
    def talk(self):
        print "Jestem lew"
```

```
>>> l=Lew()
>>> l.talk()
Jestem lew
>>>
```

Metody

Metody są definiowane w klasach.

```
class Lew:
    def talk(self):
        print "Jestem lew"
```

Definicja metody jest podobna do definicji funkcji. Pierwszy argument (`self`) jest obligatoryjny...

Metody

Definicja metody jest podobna do definicji funkcji. Pierwszy argument (`self`) jest obligatoryjny...

... i służy do odwoływania się do obiektu, dla którego metoda została wywołana.

```
def setHungry(self, val):
    self.hungry=val

def talkMore(self):
    self.talk()
    if self.hungry:
        print "glodny lew"
```

```
>>> l=Lew()
>>> l.talk()
Jestem lew
>>> l.setHungry(True)
>>> l.talkMore()
Jestem lew
glodny lew
>>>
```

Oczywiście do atrybutu `hungry` możemy również dostawać się bezpośrednio.

Inicjalizacja

Na początku lew jest popsuty.

```
>>> l=Lew()
>>> l.talkMore()
Jestem lew
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in talkMore
AttributeError: Lew instance has no attribute 'hungry'
```

Metoda `__init__`

```
def __init__(self):
    self.hungry=False
```

Dziedziczenie

```
class GroznyLew(Lew):
    def talkMore(self):
        self.talk()
        print "grozny lew,"
        print "spotkac mnie znaczy pech."
        if self.hungry:
            print "Wszystkich zjem, az do dna."
            print "Rety, lepiej nie spotkac lwa."
```

Metoda `talk` jest zdefiniowana jest w klasie `Lew`.

Dziedziczenie

```
>>> gl=GroznyLew()
>>> gl.talk()
Jestem lew
>>> gl.setHungry(True)
>>> gl.talkMore()
Jestem lew
grozny lew,
spotkac mnie znaczy pech.
Wszystkich zjem, az do dna.
Rety, lepiej nie spotkac lwa.
```

Ograniczenia dostępu

W Pythonie nie ma możliwości określania poziomu dostępności metod i atrybutów. Są dobrowolne konwencje

Składowe o nazwach zaczynających się od znaku "_" są uznawane za niedostępne publicznie i/lub zależne od implementacji. W szczególności nie należy zakładać, że w kolejnych wersjach programów pozostaną niezmienone.

Ograniczenia dostępu

Składowe o nazwach zaczynających się od znaków "__" są "prywatne" w specyficzny sposób.

Wewnątrz definicji klasy identyfikator postaci "`__ident`" jest zamieniany na "`_Klasa__ident`''

Ograniczenia dostępu

```
class Lew:
    ...

    def __talkMore(self):
        self.talk()
        if self.hungry:
            print "glodny lew"

    def talkMore(self):
        self.__talkMore()

    def talkSafe(self):
        self.__talkMore()
```

```
class GroznyLew(Lew):
    def __talkMore(self):
        self.talk()
        print "grozny lew,"
        print "spotkac mnie zn
        if self.hungry:
            print "Wszystkich
            print "Rety, lepie

    def talkMore(self):
        self.__talkMore()

    def talkMean(self):
        self.__talkMore()
```

Ograniczenia dostępu

```
>>> l=Lew()
>>> l.talkMore()
Jestem lew
glodny lew
>>> l.talkSafe()
Jestem lew
glodny lew
```

Ograniczenia dostępu

```
>>> gl=GroznyLew()
>>> gl.talkMore()
Jestem lew
grozny lew,
spotkac mnie znaczy pech.
Wszystkich zjem, az do dna.
Rety, lepiej nie spotkac lwa.
>>> gl.talkSafe()
Jestem lew
glodny lew
>>> gl.talkMean()
Jestem lew
grozny lew,
spotkac mnie znaczy pech.
Wszystkich zjem, az do dna.
Rety, lepiej nie spotkac lwa.
>>>
```

Duck typing

Duck typing

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

Na zmienną można przypisać referencję do dowolnego obiektu. Próba odwołania się do nieistniejącej składowej skutkuje błędem wykonania.

Duck typing c.d.

Zalety

- Można używać w jednym kontekście obiektów nie mających wspólnej nadklasy zawierającej wymagane składowe.
- Nie trzeba stosować rzutowań, interfejsów, typów generycznych ani wzorców.
- Łatwiejsze projektowanie.

Duck typing c.d.

Zalety

- Można używać w jednym kontekście obiektów nie mających wspólnej nadklasy zawierającej wymagane składowe.
- Nie trzeba stosować rzutowań, interfejsów, typów generycznych ani wzorców.
- Łatwiejsze projektowanie.

Wady

- Proste błędy (np. literówki), mogłyby zostać wykryte na etapie kompilacji.
- Ścisły system typów zabezpiecza przed podawaniem niewłaściwych argumentów itp.

Python quirks – metoda `__init__`

Metoda `__init__` nie jest konstruktorem.

Jeżeli podczas inicjalizacji podklasy ma zostać wywołana metoda `__init__` z nadklasy, trzeba to zrobić wprost.

```
BaseClass.__init__(self, [args...])
```

Python quirks – metody to funkcje

```
>>> l=Lew()
>>> l.talkMore()
Jestem lew
>>> Lew.talkMore(l)
Jestem lew
>>> Lew.setHungry(l, True)
>>> Lew.talkMore(l)
Jestem lew
glodny lew
>>>
```

Python quirks – argumenty metod/funkcji

Domyślne wartości

```
def __init__(self, hungry=True):
    ...
```

Argumenty nazwane

```
def parrot(voltage, state='a stiff', action='voom',
           type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

```
parrot(1000)                    # 1 positional argument
parrot(voltage=1000)            # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword args
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword args
parrot('a million', 'bereft of life', 'jump')
# 3 positional arguments
parrot('a thousand', state='pushing up the daisies')
# 1 positional, 1 keyword
```