

Programowanie i projektowanie obiektowe

Python od środka

Paweł Daniluk

Wydział Fizyki

Jesień 2016



Zasięgi nazw (ang. *scopes*)

Przestrzeń nazw

Mapowanie nazw (zmiennych) do wartości (obiektów). Jest związane z:

- klasami
- obiektami
- modułami

Dodatkowo występują przestrzenie:

- lokalna (`locals()`)
- globalna (`globals()`)
- wbudowana (`__builtin__`)

Przestrzenie nazw są niezależne.

Zasięgi nazw (ang. *scopes*)

Zasięgi

- Zasięgi są związane z funkcjami.
- Zmienne występujące w funkcji są widoczne w jej leksykalnym obrębie i należą do lokalnej przestrzeni nazw.
- Przypisanie zawsze dotyczy najsilniej zagnieżdżonego zasięgu.
- Zasięg, w którym funkcja została zdefiniowana istnieje wraz z nią.

Zasięgi – przykłady

```
def f():  
    print(x)
```

```
x = 'global'  
f()
```

```
def f():  
    x = 'f'  
    print(x)
```

```
x = 'global'  
print(x)  
f()  
print(x)
```

Zasięgi – przykłady

```
def f():  
    print(x)  
    x = 'f'  
  
x = 'global'  
f()
```

```
def f():  
    print(x)  
  
def g():  
    global x  
    print(x)  
    x = 'g'  
  
x = 'global'  
f()  
g()  
f()
```

Zasięgi – przykłady

```
x = 'global'
def f():
    def g():
        global x
        print(x)
    x = 'f'
    g()

f()
```

```
def f():
    def g():
        nonlocal x
        x = 'g'
    x = 'f'
    g()
    print(x)

x = 'global'
f()
print(x)
```

Implementacja przestrzeni nazw

Przestrzenie nazw są zazwyczaj implementowane przy pomocy słowników. Często słownik dostępny jest przez atrybut `__dict__`.

Funkcje

- `locals()`
- `globals()`
- `dir([object])` – zwraca zmienne w bieżącym zasięgu lub atrybuty dostępne w obiekcie (niekoniecznie własne)

Można zrobić klasę, której obiekty nie mają atrybutu `__dict__`. Służy do tego atrybut `__slots__`.

```
class A(object):  
    __slots__ = ['a1', 'a2']  
Klasa A ma tylko dwa atrybuty a1 i a2.
```

Typy i obiekty

- Każda wartość w Pythonie jest obiektem (nawet liczba).
- Każdy obiekt ma unikalny identyfikator (`id()`)
- Każdy obiekt ma typ (`type()`)

Typy wbudowane w Pythona

- None
- NotImplemented
- Ellipsis
- numbers.Number
 - ▶ numbers.Integral
 - ▶ numbers.Real
 - ▶ numbers.Complex
- Sequences
 - ▶ immutable
 - ★ strings
 - ★ unicode
 - ★ tuples
 - ▶ mutable
 - ★ lists
 - ★ byte arrays

Typy wbudowane w Pythona

- sets
 - ▶ set
 - ▶ frozenset
- mappings
- callable types
 - ▶ user-defined functions
 - ▶ user-defined methods
 - ▶ generator functions
 - ▶ built-in functions
 - ▶ built-in methods
 - ▶ class types (new-style)
 - ▶ classic classes
 - ▶ class instances (czasem)
- modules
- classes
- class instances
- files

Funkcje są obiektami

Funkcje mogą być traktowane jak wartości:

- przypisywane na zmienną,
- przekazywane jako parametry.

Funkcje mogą mieć atrybuty.

```
>>> def f():  
...     pass  
...  
>>> f.atr=1  
>>> f.atr  
1
```

Funkcje są obiektami

Funkcje nazwane

```
def f(x):  
    return x+1
```

Funkcje anonimowe

```
f=lambda x: x+1
```

Czym się różni funkcja od metody?

```
>>> class A():
...     def f(self):
...         pass
...
>>> A.__dict__
'__module__': '__main__', '__doc__': None, 'f': <function f at 0x10e3615f0>
>>> A.f
<unbound method A.f>
>>> A().f
<bound method A.f of <__main__.A instance at 0x10e1fdb90>>
```

- Funkcje są atrybutami klas.
- Pobranie atrybutu, który jest funkcją, skutkuje zwróceniem metody.
- Metoda może być przywiązana lub nie.

Czym się różni funkcja od metody? c.d.

Metody mają atrybuty:

- `im_class` – klasa
- `im_func` – funkcja (atrybut klasy)
- `im_self` – obiekt (określony tylko w metodzie przywiązanej)

Metoda przywiązana

Obiekt, który można wywołać (ma metodę `__call__`). Wywołanie powoduje przekazanie do funkcji parametru `self` oraz pozostałych podanych w wywołaniu metody.

Metoda nieprzywiązana

Metoda nieprzywiązana różni się od funkcji tym, że weryfikuje, czy pierwszy parametr jest instancją klasy, z której metoda pochodzi.

Nowe i stare klasy (*new-style classes*, *classic classes*)

Classic classes

- Niezależne od systemu typów.
- Instancje klas są typu `<type 'instance'>`.

New-style classes

- Włączone w system typów.
- Typ instancji jest równy jej klasie.
- Dziedziczą z klasy `object`.

Możliwości klas w nowym stylu

- metody statyczne i klasowe
- cechy i deskryptory
- lepsze wielodziedziczenie
- metaklasy
- `__slots__`

Wielodziedziczenie (ang. *multiple inheritance*)

Czasami występuje potrzeba opisu klas, które łączą w sobie cechy kilku klas nadrzędnych (np. latająca łódź).

Wielodziedziczenie może powodować straszne problemy (Deadly Diamond of Death). Niezwykle istotne są reguły wyszukiwania metod (*Method Resolution Order* – MRO).

Wielodziedziczenie (ang. *multiple inheritance*)

Czasami występuje potrzeba opisu klas, które łączą w sobie cechy kilku klas nadrzędnych (np. latająca łódź).

Wielodziedziczenie może powodować straszne problemy (Deadly Diamond of Death). Niezwykle istotne są reguły wyszukiwania metod (*Method Resolution Order* – MRO).

Stare i nowe klasy różnią się MRO. W nowych klasach zawsze występuje diament (bo dziedziczą z `object`).

Linearyzacja

Kolejność klas w MRO zachowuje porządek wynikający z dziedziczenia (nadklasa zawsze po podklasie) i kolejności występowania nadklas w definicji klasy.

Wielodziedziczenie c.d.

Niektóre hierarchie nie pozwalają na linearyzację

```
class A(object):  
    def meth(self): return "A"  
class B(object):  
    def meth(self): return "B"  
  
class X(A, B): pass  
class Y(B, A): pass  
  
class Z(X, Y): pass
```

Jak dostać się do nadklasy?

Metoda rozszerzająca zazwyczaj wywołuje metodę z nadklasy.

Problem

```
class A(object):
    def m(self): "save_A's_data"
class B(A):
    def m(self):
        "save_B's_data"
        A.m(self)
class C(A):
    def m(self):
        "save_C's_data"
        A.m(self)
class D(B, C):
    def m(self):
        "save_D's_data"
        B.m(self); C.m(self)
```

Jak dostać się do nadklasy?

Metoda rozszerzająca zazwyczaj wywołuje metodę z nadklasy.

Problem

```
class A(object):
    def m(self): "save_A's_data"
class B(A):
    def m(self):
        "save_B's_data"
        A.m(self)
class C(A):
    def m(self):
        "save_C's_data"
        A.m(self)
class D(B, C):
    def m(self):
        "save_D's_data"
        B.m(self); C.m(self)
```

Wywołanie `D.m()` spowoduje dwukrotne wykonanie `A.m()`.

Jak dostać się do nadklasy?

```
class A(object):
    def m(self): "save_A's_data"
class B(A):
    def _m(self): "save_B's_data"
    def m(self): self._m(); A.m(self)
class C(A):
    def _m(self): "save_C's_data"
    def m(self): self._m(); A.m(self)
class D(B, C):
    def _m(self): "save_D's_data"
    def m(self): self._m(); B._m(self); C._m(self); A.m(self)
```

Jak dostać się do nadklasy?

```
class A(object):
    def m(self): "save_A's_data"
class B(A):
    def _m(self): "save_B's_data"
    def m(self): self._m(); A.m(self)
class C(A):
    def _m(self): "save_C's_data"
    def m(self): self._m(); A.m(self)
class D(B, C):
    def _m(self): "save_D's_data"
    def m(self): self._m(); B._m(self); C._m(self); A.m(self)
```

To rozwiązanie jest poprawne, ale implementacja D musi uwzględnić istnienie klasy A.

Strategia następnej metody

```
class A(object):
    def m(self): "save_A's_data"
class B(A):
    def m(self): "save_B's_data"; super(B, self).m() # C
class C(A):
    def m(self): "save_C's_data"; super(C, self).m() # A
class D(B, C):
    def m(self): "save_D's_data"; super(D, self).m() # B
```

- A. `__mro__` == (A, **object**)
- B. `__mro__` == (B, A, **object**)
- C. `__mro__` == (C, A, **object**)
- D. `__mro__` == (D, B, C, A, **object**)

`super(class, obj)`

- nie musi być nadklasą `class`
- jest nadklasą `obj.__class__`
- `obj` musi być instancją pewnej podklasy `class`

Jak powstają metody?

```
>>> class A(object):
...     def f(self): pass
...
>>> A.__dict__['f']
<function f at 0x10be7f140>
>>> A.f
<unbound method A.f>
>>> A().f
<bound method A.f of <__main__.A object at 0x10bea10d0>>
```

Funkcja ma metodę `__get__(self, obj, objtype=None)`.

```
>>> def g(self):pass
...
>>> g.__get__(None, A)
<unbound method A.g>
>>> g.__get__(A(), A)
<bound method A.g of <__main__.A object at 0x10bea1250>>
```


Jak powstają metody?

Zamiast funkcji możemy mieć dowolny obiekt, który ma metodę `__get__`.

```
class RandFun(object):
    def __init__(self, *args):
        self.flist=args

    def __get__(self, obj, objtype=None):
        return random.choice(self.flist).__get__(obj, objtype)

class A(object):
    def f1(self):
        print "Jestem f1"

    def f2(self):
        print "Jestem f2"

    def g(self, val):
        print "Jestem g: "+str(val)

rf=RandFun(f1, f2)
rfg=RandFun(f1, f2, fg)
```

Deskryptory

Obiekt implementujący co najmniej jedną z metod `__get__()`, `__set__()`, `__delete__()` nazywa się deskryptorem.

```
class RevealAccess(object):
    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print 'Retrieving', self.name
        return self.val

    def __set__(self, obj, val):
        print 'Updating', self.name
        self.val = val

class B(object):
    x = RevealAccess(10, 'var_□"x"')
    y = 5
```

Deskryptory c.d.

Deskryptory pozwalają na łatwą realizację kapsułkowania.

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

```
class C(object):
    def getx(self):
        return self.__x
    def setx(self, value):
        self.__x = value
    def delx(self):
        del self.__x

x = property(getx, setx, delx, "I'm the 'x' property.")
```

Metody statyczne

Przy pomocy deskryptorów można zrealizować metody statyczne.

```
class staticmethod(object):  
    def __init__(self, f):  
        self.f = f  
  
    def __get__(self, obj, objtype=None):  
        return self.f
```

... i klasowe

```
class classmethod(object):
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```