

Programowanie i projektowanie obiektowe

Python od środka

Paweł Daniluk

Wydział Fizyki

Jesień 2014



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
```

Metody statyczne (i klasowe) można tworzyć korzystając z klas `staticmethod` i `classmethod`.

```
class A(object):  
    def f():  
        pass  
    static=staticmethod(f)  
  
    def g(cls):  
        pass  
    g=classmethod(g)
```

@dekoratory c.d.

Ale zazwyczaj stosuje się zapożyczoną z Javy składnię.

```
class A(object):  
    @staticmethod  
    def f():  
        pass  
  
    @classmethod  
    def g(cls):  
        pass
```

@dekoratory c.d.

Napis:

```
@dekorator  
def f(arg):  
    ...
```

oznacza:

```
def f(arg):  
    ...  
f=dekorator(f)
```

Dekorator funkcji jest funkcją, która jako argument bierze funkcję dekorowaną i zwraca zmodyfikowaną funkcję.

Przykład

```
def bread(func):
    def wrapper():
        print "</''''''\>"
        func()
        print "<\_\_\_\_\_\_/>"
    return wrapper

def ingredients(func):
    def wrapper():
        print "#tomatoes#"
        func()
        print "~salad~"
    return wrapper

def sandwich(food="--ham--"):
    print food
```

Przykład

```
sandwich()
#outputs: --ham--
sandwich = bread(ingredients(sandwich))
sandwich()
#outputs:
#</'''''''\>
# #tomatoes#
# --ham--
# ~salad~
#<\ />
```

@dekoratory c.d.

```
@bread
@ingredients
def sandwich ( food="--ham--" ):
    print food
```

```
sandwich()  
#outputs:  
#</''''\>  
# #tomatoes#  
# --ham--  
# ~salad~  
#<\'      />
```

Kolejność ma znaczenie

```
@ingredients
@bread
def strange_sandwich(food="--ham--"):
    print food

strange_sandwich()
#outputs:
##tomatoes#
#</''''''''\>
# --ham--
#<\_____/>
# ~salad~
```

@dekoratory – przekazywanie argumentów

```
def a_decorator_passing_arguments(function_to_decorate):  
    def a_wrapper_accepting_arguments(arg1, arg2):  
        print "I got args! Look:", arg1, arg2  
        function_to_decorate(arg1, arg2)  
    return a_wrapper_accepting_arguments  
  
@a_decorator_passing_arguments  
def print_full_name(first_name, last_name):  
    print "My name is ", first_name, last_name  
  
print_full_name("Peter", "Venkman")  
  
# outputs:  
#I got args! Look: Peter Venkman  
#My name is Peter Venkman
```

Przy dekoratorach stosowanych do dowolnych funkcji używa się `*args`, `**kwargs`.

@dekoratory – przekazywanie argumentów do dekoratora

```
def wrap_in_tag(tag):  
    def factory(func):  
        def decorator(*args, **kwargs):  
            return '<%(tag)s>%(rv)s</%(tag)s>' %  
                ({'tag': tag, 'rv': func(*args, **kwargs)})  
        return decorator  
    return factory  
  
@wrap_in_tag('b')  
@wrap_in_tag('i')  
def say(val):  
    return val  
  
print say('hello')
```

Funkcja `wrap_in_tag` zwraca dekorator, który dokłada do wyniku dekorowanej funkcji zadany tag XML.

Klasy również można dekorować

```
def addID(original_class):  
    orig_init = original_class.__init__  
  
    def __init__(self, id, *args, **kws):  
        self.__id = id  
        self.getId = getId  
        orig_init(self, *args, **kws)  
  
    original_class.__init__ = __init__  
    return original_class  
  
@addID  
class Foo:  
    pass
```

Konstruktory i inicjalizatory

Metoda `__new__`

`__new__` jest metodą statyczną, która jako pierwszy argument bierze klasę obiektu, który ma zostać stworzony. Pozostałe argumenty są przekazywane do inicjalizatora.

Metoda `__init__`

`__init__` jest wywoływana po `__new__`.

Metoda `__call__`

Tworząc klasę (np. `A()`) wywołujemy metodę `__call__`, która wywołuje po kolei `__new__` i `__init__`.

Konstruktory i inicjalizatory - c.d.

Przykład

```
In [1]: class A(object):  
...:     def __init__(self):  
...:         self.x = 1  
...:
```

```
In [2]: a1 = A()
```

```
In [3]: a1.x  
Out[3]: 1
```

```
In [4]: a2 = A.__new__(A)
```

```
In [5]: a2  
Out[5]: <__main__.A at 0x10434b750>
```

Konstruktory i inicjalizatory - c.d.

Przykład

```
In [6]: a2.x
```

```
AttributeError
```

```
Traceback (most recent call last)
```

```
<ipython-input-6-ecef1d046d23> in <module>()  
----> 1 a2.x
```

```
AttributeError: 'A' object has no attribute 'x'
```

```
In [7]: a2.__init__()
```

```
In [8]: a2.x
```

```
Out[8]: 1
```

Metoda `__new__`

```
class GimmeFive(object):  
    def __new__(cls, *args, **kwargs):  
        return 5
```

Zastosowania

- 1 Tworzenie obiektu innej klasy.
- 2 Wykonanie operacji, które nie mogą zależeć od implementacji `__init__` w podklasach.

Metoda `__new__` - przykład

```
class Operator(object):
    def __new__(cls, op, *args, **kwargs):
        if op == '+':
            return super(Operator, cls).__new__(Add)
        elif op == '*':
            return super(Operator, cls).__new__(Mul)

    def __init__(self, op, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

class Add(Operator):
    def eval(self):
        return self.arg1 + self.arg2

class Mul(Operator):
    def eval(self):
        return self.arg1 * self.arg2
```

Metoda `__new__` - przykład

```
In [2]: op.Operator('+', 3, 4)
```

```
Out[2]: <op.Add at 0x1071a5790>
```

```
In [3]: op.Operator('+', 3, 4).eval
```

```
Out[3]: <bound method Add.eval of <op.Add object at 0x1071a5b90>
```

```
In [4]: op.Operator('+', 3, 4).eval()
```

```
Out[4]: 7
```

```
In [5]: op.Operator('*', 3, 4).eval()
```

```
Out[5]: 12
```

Metaklasy (Lasciate ogni speranza)

Klasy są obiektami:

- klasy `type` – new-style classes
- klasy `types.ClassType` – classic classes

Metaklasy (Lasciate ogni speranza)

Klasy są obiektami:

- klasy type – new-style classes
- klasy `types.ClassType` – classic classes

W zasadzie nic nie stoi na przeszkodzie, aby klasy były obiektami dowolnej klasy (metaklasy).

Jaki jest z tego pożytek?

- sianie zamętu i zniechęcenia
- kontrola tworzenia klasy (metody `__new__` i `__init__`)
- kontrola tworzenia obiektów klasy (metoda `__call__`)

Metaklasy c.d.

Metaklasa danej klasy jest określona przez:

- atrybut `__metaclass__`
- metaklasa jednej z klas bazowych
- zmienna globalna `__metaclass__`
- `types.ClassType`

Tworzenie klasy skutkuje wywołaniem metaklasy:

`M(name, bases, dict)`

gdzie:

- `name` – nazwa klasy
- `bases` – klasy bazowe
- `dict` – elementy klasy

Metaklasy c.d.

```
class Meta(type):
    def __init__(cls, name, bases, dict):
        print "Definiowana_klasa:", cls
        print "o_nazwie:", name
        print "dziedziczy_z:", bases
        print "o_elementach:", dict
        type.__init__(cls, name, bases, dict)

    def __call__(cls, *args, **kwargs):
        print "Tworze_obiekt_klasy:", cls
        return type.__call__(cls, *args, **kwargs)
```

Metaklasy c.d.

```
In [10]: class A(object):
.....:     __metaclass__ = Meta
.....:     def m(self):
.....:         print "Jestem_m"
.....:

Definiowana klasa: <class '__main__.A'>
o nazwie: A
dziedziczy z: (<type 'object'>,)
o elementach: {'m': <function m at 0x10e3c0de8>, '__module__':
In [11]: A()
Tworze obiekt klasy: <class '__main__.A'>
Out[11]: <__main__.A at 0x10e3b8e50>
```

Przykład – automatyczne tworzenie properties

```
class autoprop(type):
    def __init__(cls, name, bases, dict):
        super(autoprop, cls).__init__(name, bases, dict)
        props = {}
        for member in dict.keys():
            if member.startswith("_get_")
                or member.startswith("_set_"):
                props[member[5:]] = 1
        for prop in props.keys():
            fget = getattr(cls, "_get_%s" % prop, None)
            fset = getattr(cls, "_set_%s" % prop, None)
            setattr(cls, prop, property(fget, fset))
```

Metaklasy c.d.

Zastosowanie

```
In [29]: class A(object):
.....:     __metaclass__ = autoprop
.....:     def _get_x(self):
.....:         print "_get_x"
.....:         return a.__x
.....:     def _set_x(self, x):
.....:         print "_set_x"
.....:         self.__x = x
.....:
```

```
In [30]: a = A()
```

```
In [31]: a.x = 5
set_x
```

```
In [32]: a.x
get_x
get_x
Out[32]: 5
```