Programowanie i projektowanie obiektowe

Metaprogramowanie (w Pythonie)

Paweł Daniluk

Wydział Fizyki

Jesień 2016

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)
2. instrukcje warunkowe

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)
2. instrukcje warunkowe
3. pętle

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)
2. instrukcje warunkowe
3. pętle
4. funkcje i procedury

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)
2. instrukcje warunkowe
3. pętle
4. funkcje i procedury
5. klasy i moduły

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)
2. instrukcje warunkowe
3. pętle
4. funkcje i procedury
5. klasy i moduły
6. makrodefinicje (takie jak w C – w zasadzie nie należą do języka)

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)
2. instrukcje warunkowe
3. pętle
4. funkcje i procedury
5. klasy i moduły
6. makrodefinicje (takie jak w C – w zasadzie nie należą do języka)
7. wzorce (C++), klasy generyczne (Java)

Wstęp ideologiczny

Języki programowania są jak cebula

1. instrukcje proste (przypisania, wywołania funkcji bibliotecznych, ...)
2. instrukcje warunkowe
3. pętle
4. funkcje i procedury
5. klasy i moduły
6. makrodefinicje (takie jak w C – w zasadzie nie należą do języka)
7. wzorce (C++), klasy generyczne (Java)

Każda kolejna warstwa pozwala skrócić kod programu i ułatwia pracę programisty.

Wstęp ideologiczny

Sytuacja idealna

Język programowania specjalnie dostosowany do rozwiązywanego problemu.

Wstęp ideologiczny

Sytuacja idealna

Język programowania specjalnie dostosowany do rozwiązywanego problemu.

Rozwiązanie

Funkcje i klasy dające możliwie prosty sposób wyrażania operacji na reprezentacjach przetwarzanych informacji.

Wstęp ideologiczny

Sytuacja idealna

Język programowania specjalnie dostosowany do rozwiązywanego problemu.

Rozwiązanie

Funkcje i klasy dające możliwie prosty sposób wyrażania operacji na reprezentacjach przetwarzanych informacji.

Smutna rzeczywistość

Programowanie to w dużej części rzemiosło. Często występują problemy podobne do siebie.

Wstęp ideologiczny

Sytuacja idealna

Język programowania specjalnie dostosowany do rozwiązywanego problemu.

Rozwiązanie

Funkcje i klasy dające możliwie prosty sposób wyrażania operacji na reprezentacjach przetwarzanych informacji.

Smutna rzeczywistość

Programowanie to w dużej części rzemiosło. Często występują problemy podobne do siebie.

Rozwiązanie

Programy do automatycznego generowania innych programów.

Python jest językiem refleksyjnym – wykonywany program jest traktowany jak dane.

Przykładowe funkcje

type(), isinstance(), callable(), dir(), getattr(), eval(), exec()

Można i tak...

```
def f(i):
    def res(self):
        print i

    return res

class A(object):
    for i in range(10):
        exec('f'+str(i)+'=f('+str(i)+')')

A().f1()
A().f3()
```

Metody statyczne i klasowe – przypomnienie

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

    def __get__(self, obj, objtype=None):
        return self.f
```

```
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
```

@dekoratory

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

```
class A(object):
    def f():
        pass
    f=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.

@dekoratory c.d.

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
```

@dekoratory c.d.

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~
#<\______/>
```

@dekoratory c.d.

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 decorator(func):
        def wrapper(*args, **kwargs):
            return '<%(tag)s>%(rv)s</%(tag)s>' %
              ({'tag': tag, 'rv': func(*args, **kwargs)})
        return wrapper
    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]: Operator('+', 3, 4)
Out[2]: <Add at 0x1071a5790>

In [3]: Operator('+', 3, 4).eval
Out[3]: <bound method Add.eval of <Add object at 0x1071a5b90>>

In [4]: Operator('+', 3, 4).eval()
Out[4]: 7

In [5]: 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`

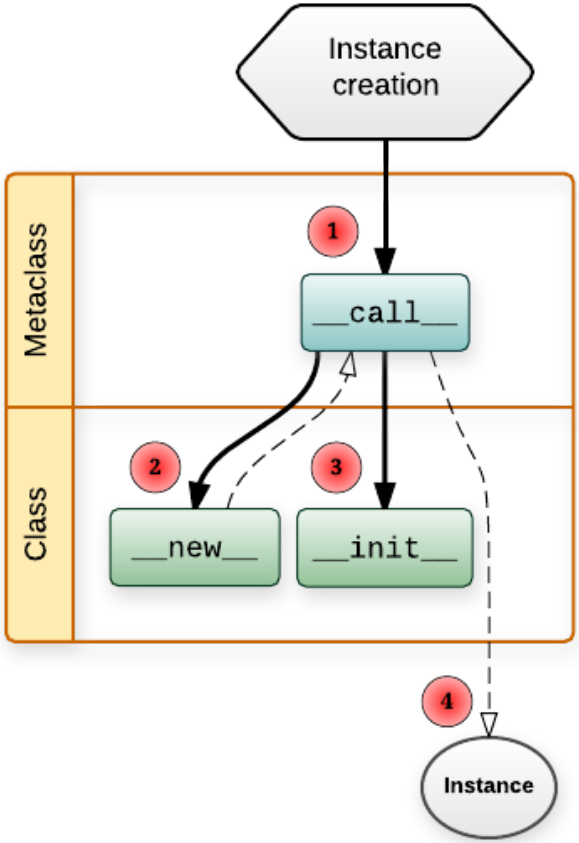
Tworzenie klasy skutkuje wywołaniem metaklasy:

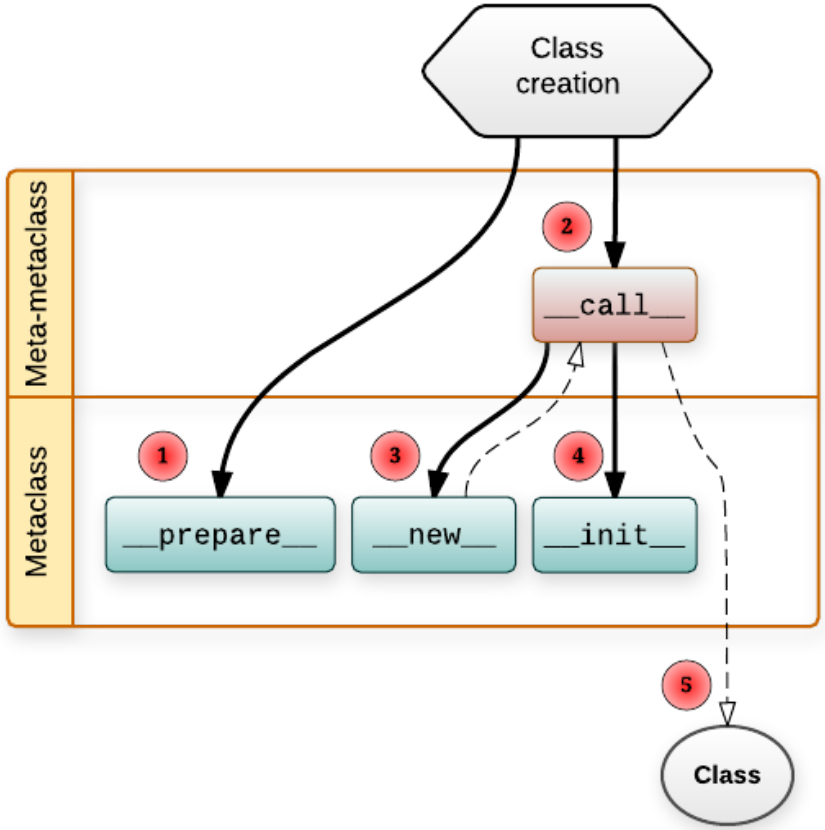
```
M(name, bases, dict)
```

gdzie:

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

Tworzenie (Python 3.0)





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>
```

Metaklasy c.d.

Przykład – automatyczne tworzenie properties

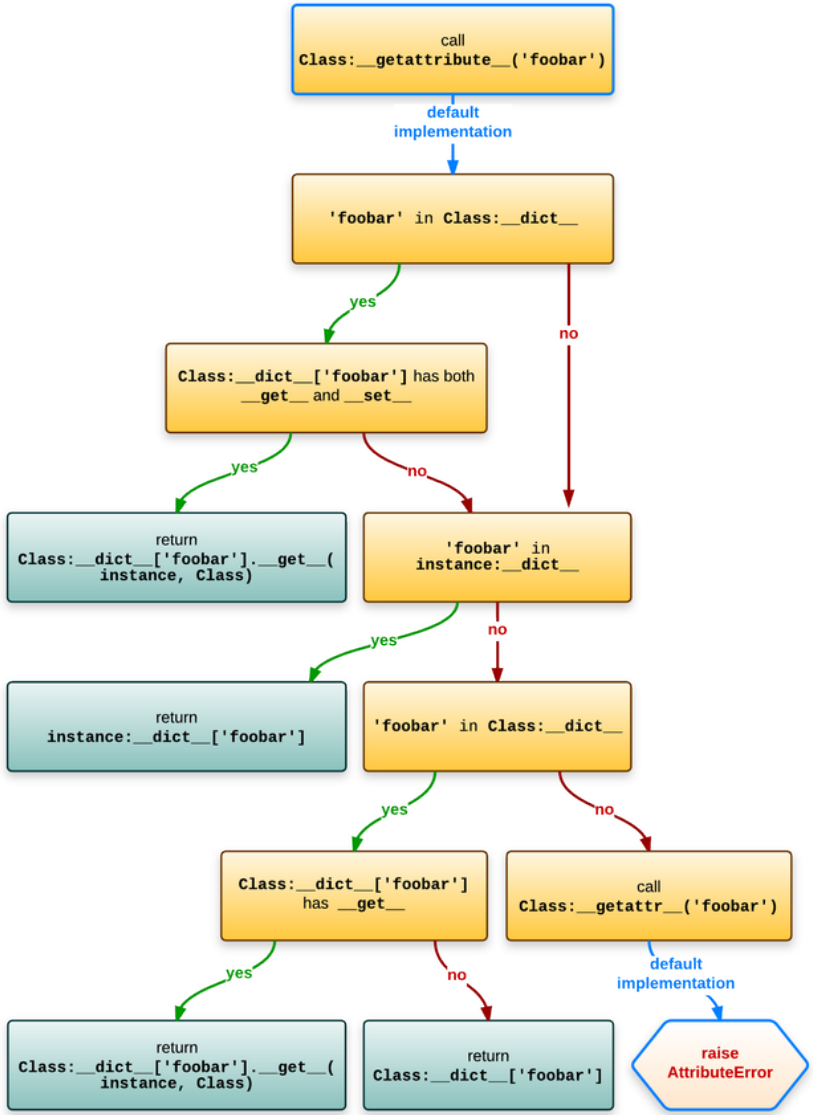
```
class autoprop(type):
    def __init__(cls, name, bases, dict):
        super(autoprop, cls).__init__(name, bases, dict)
        props = {}
        for member in dict.keys():
            if member.startswith("_get_")
                    or member.startswith("_set_"):
                props[member[5:]] = 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.

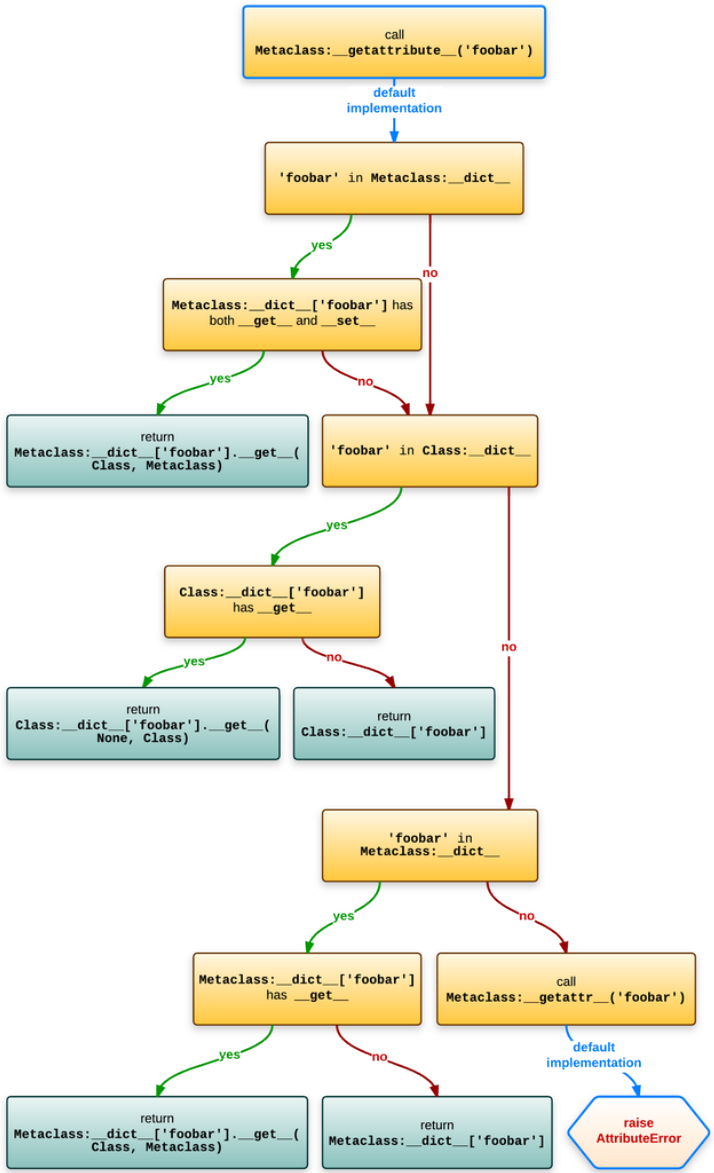
Zastosowanie

```
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
```

Reguły poszukiwania atrybutów instancji (Python 3.0)



Reguły poszukiwania atrybutów klas (Python 3.0)



Metaklasy – potencjalne zastosowania

- Dekoratory, które są dziedziczone
- Automatyczne uzupełnianie klasy o deskryptory
- Rejestr definiowanych klas
- Prawdziwe sigletony