16. Метаобектен протокол

03 декември 2024

Преговор: атрибути

```
dir(foo) -> foo.__dict__

getattr(foo, 'x') -> foo.__getattribute__('x') -> foo.__getattr__('x')

setattr(foo, 'x', 'y') -> foo.__setattr__('x', 'y')

del foo.x -> delattr(foo, 'x') -> foo.__delattr__('x')
```

Мета

- **μετά /mɛ.tə/** (гръцки)
  Представка за положение зад, през, след или отвъд нещо
- **meta /ˈmɛtə/** (английски)
  Отнасящ се до себе си или условностите на жанра си; самореферентен
- **метаобекти**
  Обекти, които описват обекти
- Други примери: метаданни, мета-информация, мета-хумор, метапрограмиране
  (за последното ще разкажем повече следващия път)

Протокол

(от френски: protocole; от латински: protocollum; от старогръцки: πρωτόκολλο) най-общо представлява определено множество от правила в употребление при дадени обстоятелства.

(от Wikipedia)

Метаобектен протокол

Objects are Python’s abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann’s model of a “stored program computer”, code is also represented by objects.)

(из https://docs.python.org/3/reference/datamodel.html)

Метаобектен протокол (2)

- Python взаимства идеи от Lisp
- Предимство: не е нужно да знаеш Python, за да четеш Python
- Недостатък: липсва консистентността в репрезентацията на код/данни
- Прилича на прост начин за reflection
- Може да го срещнете като “MOP”

__dunders__

- Представляват (голяма част от) метаобектния протокол
- Могат да бъдат четени и/или писани и/или предефинирани
- ... и ни позволяват да интроспектираме обекти
- ... или пък да “вмъкнем” наш код на разни места в езика

__dunders__ (2)

Нека:

- Преговорим познатите
- Разгледаме някои от непознатите
- Игнорираме по-апокрифните

Кастове

- `__bool__(self)`
- `__float__(self)`
- `__int__(self)`
- `__str__(self)`

Кастове (2)

```
class Scotsman:
    def __bool__(self):
        print('Converting to bool...')
        return True

if Scotsman():
    print('A True Scotsman!')
```

```
Converting to bool...
A True Scotsman!
```

Репрезентация

- `__repr__(self)`
- `__str__(self)`
- `__doc__`
- `__dir__(self)`

Аритметика

- `__add__(self, a)`
- `__sub__(self, a)`
- `__mul__(self, a)`
- `__div__(self, a)`
- `__floordiv__(self, a)`
- `__truediv__(self, a)`
- `__divmod__(self, a)`
- `__pow__(self, a)`
- `__matmul__(self, a)`

Полиморфизъм

- Код, който се държи различно спрямо (типа на) параметрите си
- Примери от други езици:
  - generics;
  - overloading;
  - multiple dispatch (multimethods);
  - виртуални методи
- Методите в Python синтактично имат само последното
  - ... но не го наричаме така;
  - полиморфизъм само по първия аргумент (self);
  - a.к.a. single dispatch
- Операторите имат нещо като double dispatch

Аритметика (2)

“десни” варианти:

- `__radd__(self, a)`
- `__rsub__(self, a)`
- `__rmul__(self, a)`
- `__rdiv__(self, a)`
- `__rfloordiv__(self, a)`
- `__rtruediv__(self, a)`
- `__rdivmod__(self, a)`
- `__rpow__(self, a)`
- `__rmatmul__(self, a)`
- `NotImplemented`

Аритметика (3)

```
>>> class X:
...     pass
...
>>> x = X()
>>> 1 + x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'X'
>>> (1).__add__(x)
NotImplemented
```

Аритметика (4)

```
>>> class X:
...     def __radd__(self, other):
...         print(f'called with self={self}, other={other}')
...         return self
...
>>> x = X()
>>> 1 + x
called with self=<__main__.X object at 0x7fe4be931070>, other=1
<__main__.X object at 0x7fe4be931070>
```

Аритметика (5)

“in-place” варианти:

- `__iadd__(self, a)`
- `__isub__(self, a)`
- `__imul__(self, a)`
- `__idiv__(self, a)`
- `__ifloordiv__(self, a)`
- `__itruediv__(self, a)`
- `__idivmod__(self, a)`
- `__ipow__(self, a)`
- `__imatmul__(self, a)`

Унарна аритметика

- `__abs__(self)`
- `__pos__(self)`
- `__neg__(self)`

Битова аритметика

- `__and__ / __rand__ / __iand__(self, a)`
- `__or__ / __ror__ / __ior__(self, a)`
- `__xor__ / __rxor__ / __ixor__(self, a)`
- `__rshift__ / __rrshift__ / __irshift__(self, n)`
- `__lshift__ / __rlshift__ / __ilshift__(self, n)`
- `__invert__(self)`

Равенство и хеширане

- `__eq__(self, a)`
- `__ne__(self, a)`
- `__hash__(self)`

Равенство и хеширане (2)

Имат вградено поведение:

- `x == y # x is y`
- `x != y # not (x == y)`
- `hash(x) # някаква_функция(id(x))`
  - към момента на 64 битов OS: `id(x) // 16`
  - не разчитайте поведението на последното да се запази!

Сравнения

- `__lt__(self, a)`
- `__le__(self, a)`
- `__gt__(self, a)`
- `__ge__(self, a)`

Сравнения (2)

```
>>> class Python:
...     def __gt__(self, other):
...         print('Greatest!')
...         return True
...
>>> Python() > 'Java'
Greatest!
True
```

Сравнения (3)

- Double dispatch!
- нямаме десен вариант на `__lt__`
- вместо него се ползва `__gt__`
- ... и обратното
- аналогично за `__ge__`/`__le__`

Сравнения (4)

```
>>> class Python:
...     def __gt__(self, other):
...         print('Greatest!')
...         return True
...
>>> 'Java' < Python()
Greatest!
True
>>> 'Java'.__lt__(Python())
NotImplemented
```

with

- `__enter__(self)`
- `__exit__(self, type, value, traceback)`

Атрибути

- `__dir__(self)`
- `__getattribute__(self, name)`
- `__getattr__(self, name)`
- `__setattr__(self, name, value)`
- `__delattr__(self, name)`

Колекции

- `__len__(self)`
- `__contains__(self, item)`
- `__getitem__(self, i)`
- `__setitem__(self, i, value)`
- `__delitem__(self, i)`

Итератори

- `__iter__(self)`
- `__next__(self)`

Метаатрибути

- `__dict__`
- `__slots__`
- `__class__`
- `__globals__`
- `__name__`

Функции

- `__code__`
- `__call__(self, *args, **kwargs)`

Импортиране на модули

```
import module
```

...съответства на...

```
module = __import__('module')
```

Конструиране

- `__new__(cls, *args, **kwargs)`
- `__init__(self, *args, **kwargs)`

__new__

`__new__` е истинският конструктор на вашите обекти. `__init__` е само инициализатор.

```
class Vector(tuple):
  def __new__(klass, x, y):
    return tuple.__new__(klass, (x, y))

  def __add__(self, other):
    if not isinstance(other, Vector):
      return NotImplemented
    return Vector(self[0] + other[0], self[1] + other[1])
```

Реконструиране

- `__reduce__(self)`
- `__reduce_ex__(self, protocol)`

Method resolution order

Редът за обхождане на базови класове

```
class A(int): pass

class B: pass

class C(A, B, int): pass

C.__mro__ # или C.mro()
# <class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
# <class 'int'>, <class 'object'>
```

Method resolution order (2)

- Използва алгоритъм наречен C3 linearization
- https://en.wikipedia.org/wiki/C3_linearization
- https://dl.acm.org/doi/pdf/10.1145/236337.236343

Други (без конкретен ред)

- `__sizeof__`
- `__weakrefoffset__`
- `__base__`
- `__bases__`
- `__basicsize__`
- `__module__`
- `__package__`
- `__all__`
- `__builtins__`
- `__subclasses__`
- `__subclasshook__`
- `__subclasscheck__`
- `__file__`
- `__format__`
- `__loader__`
- `Ellipsis/...`

Дескриптори

- `__get__(self, instance, owner)`
- `__set__(self, instance, value)`
- `__delete__(self, instance)`

Дескриптори: Теория

```
def __get__(self, instance, owner): ...

def __set__(self, instance, value): ...

def __delete__(self, instance): ...
```

- Викат се върху обект, който бива достъпван като атрибут на друг обект.
- Ако класът `A` има атрибут `foo`, със стойност обект от тип `B`, достъпвайки `A.foo` ще се извика `__get__` на `B`.
- Опитвайки се да го предефинираме, удряме `__set__`.
- Познайте какво става ако опитаме да го изтрием с `del`.

Дескриптори: Практика

```
class B:
    def __get__(self, instance, owner):
        return "You came to the wrong neighborhood, motherflower!"

    def __set__(self, instance, value):
        print("What!? You think you can change my personality just like that!?")

    def __delete__(self, instance):
        print("Can't touch me!")

class A:
    foo = B()

a = A()
print(a.foo)
a.foo = 'bar'
del a.foo
```

Пример - cached_property

```
from functools import cached_property

class Lecturer:
    def __init__(self, name):
        self.name = name

    @cached_property
    def greet(self):
        print("Thinking what to say…")
        return f"Hello, my name is {self.name}"
```

- при първо достъпване кодът се изпълнява
- пресметнатата стойност се кешира в инстанцията (всяка инстанция има нужда от собствен кеш)
- при следващо достъпване се взима от кеша

Пример - cached_property (2)

```
>>> joan = Lecturer("Joan")
>>> joan.greeting
Thinking what to say…
'Hello, my name is Joan'
>>> joan.greeting
'Hello, my name is Joan'
>>> joro = Lecturer("Victor")
>>> joro.greeting
Thinking what to say…
'Hello, my name is Victor'
```

Пример - cached_property (3)

Оригиналната имплементация може да бъде написана и така:

```
def greeting(self):
    print("Thinking what to say…")
    return f"Hello, my name is {self.name}"

class Lecturer:
    def __init__(self, name):
        self.name = name

    greeting = cached_property(greeting)
```

Проста имплементация на cached_property

```
class cached_property:
    def __init__(self, function):
        self.function = function

    def __get__(self, obj, type=None) -> object:
        result = self.function(obj)
        setattr(obj, self.function.__name__, result)
        return result
```

Bound methods

```
>>> increment = (1).__add__
>>> map(increment, [0, 1, 2, 3])
[1, 2, 3, 4]
```

- "Закача" инстанция за метод
- Прилича на частично прилагане на функция
- Единствено `self` може да бъде приложен

Bound methods: Проста имплементација!

```
class MyMethod:

    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        if instance:
            return lambda: self.func(instance)
        else:
            return lambda explicit_instance: self.func(explicit_instance)

class Python:
    name = 'Monty'
    greet = MyMethod(lambda self: 'My name issss %s' % self.name)
```

Bound methods: Проста имплементација!

```
snake = Python()
snake.greet() # 'My name issss Monty'
snake.name = 'Nagini'
Python.greet() # TypeError: <lambda>() takes exactly 1 argument (0 given)
Python.greet(snake) # 'My name issss Nagini'
```

Въпроси?