12. Метаобектен протокол

24 ноември 2022

Питанка

Какво ще изведе следният код:

```
sentence = 'we are humans'
matched = re.match(r'(.*) (.*?) (.*)', sentence)
print(matched.groups())
```

```
('we', 'are', 'humans')
```

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')
```

Метаобектен протокол/метапрограмиране/мета

Отзад напред:

- Какво означава “мета”?
- Какво означава “метапрограмиране”?
- Какво означава “метаобектен протокол”?

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
- Други примери: метаданни, мета-информация

Метапрограмиране

- По-специфично:
  Програми, които пишат програми.

- По-общо:
  Техника, при която програми третират други программи като данни;
  (четене; интроспекция; манипулация; генериране).

Метапрограмиране - примери

- Lisp и приятели
- macros
- template metaprogramming
- reflection

Метапрограмиране - macros

```
#ifdef X
  #include <smthng>
  int x = X;
#else
  #define X 42
  #include <smthng_else>
#endif
```

/* има if-else условия; би могло да има [доста глуповата] рекурсия */
/* ужасен пример; в други езици има по-адекватни макроси */

Метапрограмиране - template metaprogramming

```
template <int N> int fib() { return fib<N-1>() + fib<N-2>(); }

template <> int fib<0>() { return 0; }

template <> int fib<1>() { return 1; }
```

// fib<10> ще генерира функции fib<2> ... fib<10> по време на компилация
// условия под формата на pattern matching; има рекурсия

Метапрограмиране - reflection

```
import java.lang.reflect.Method;

// без reflection
Foo foo = new Foo();
foo.hello();

// интроспекция и извикване със reflection
try {
    Object foo = Foo.class.getDeclaredConstructor().newInstance();
    Method m = foo.getClass().getDeclaredMethod("hello", new Class<?>[0]);
    m.invoke(foo);
} catch (ReflectiveOperationException ignored) {}
```

Метапрограмиране - lisp и приятели

```
'(1 2 3) ; списък с числа
(+ 1 2) ; 1+2
(foo x) ; foo(x)
'(+ 1 2) ; списък с функцията + и числата 1 и 2

; Кодът и данните споделят общ формат!
```

Метапрограмиране - не-примери

Може да използват техники/идеи от метапрограмирането, но технически/терминологично погледнато не са метапрограмиране...

- Оптимизиращи компилатори
- Linters (напр. pycodestyle/pep8)
- Интерпретатори
- Емулатори

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

Код или данни?

```
def fma(a, x, y=0):
    return a*x + y
```

Код или данни? (2)

```
def fma(a, x, y = 0):
    return a.__mul__(x).__add__(y)
```

Код или данни? (3)

```
>>> dir(fma)
['__annotations__', '__call__', '__class__', '__closure__',
'__code__', '__defaults__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__get__',
'__getattribute__', '__globals__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__kwdefaults__', '__le__', '__lt__',
'__module__', '__name__', '__ne__', '__new__', '__qualname__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
>>> fma.__name__
'fma'
>>> fma.__class__
<class 'function'>
>>> fma.__defaults__
(0,)
```

Код или данни? (4)

```
>>> dir(fma.__code__)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'co_argcount', 'co_cellvars', 'co_code', 'co_consts',
'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_kwonlyargcount', 'co_lnotab',
'co_name', 'co_names', 'co_nlocals', 'co_posonlyargcount', 'co_stacksize', 'co_varnames',
'replace']
>>> repr(fma.__code__)
'<code object fma at 0x7f1da9cc1190, file "<stdin>", line 1>'
>>> fma.__code__.co_argcount
3
>>> fma.__code__.co_filename
'<stdin>'
>>> fma.__code__.co_firstlineno
1
>>> fma.__code__.co_stacksize
3
>>> fma.__code__.co_names
('__mul__', '__add__')
>>> fma.__code__.co_code
b'|\x00\xa0\x00|\x01\xa1\x01\xa0\x01|\x02\xa1\x01S\x00'
```

python.exe

Интерпретаторът на Python е програма, която (грубо казано):

- чете кодът на вашата програма
- превръща я в данни
- кешира ги като *__pycache__/*.pyc* върху файловата система
- оценява (изчислява) данните
- обработва __dunder__ атрибутите по по-специални начини

__dunders__

Нека:

- преговорим познатите
- разгледаме някои от непознатите
- игнорираме по-апокрифните

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

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

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
```

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'
```

Кастове

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

Репрезентация

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

Репрезентация

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

Аритметика

- `__add__ / __radd__(self, a)`
- `__sub__ / __rsub__(self, a)`
- `__mul__ / __rmul__(self, a)`
- `__div__ / __rdiv__(self, a)`
- `__floordiv__ / __rfloordiv__(self, a)`
- `__truediv__ / __rtruediv__(self, a)`
- `__divmod__ / __rdivmod__(self, a)`
- `NotImplemented`
- `Ellipsis/...`

Аритметика (2)

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

Битова аритметика

- `__and__(self, a)`
- `__or__(self, a)`
- `__xor__(self, a)`
- `__rshift__(self, n)`
- `__lshift__(self, n)`

Равенство и хеширане

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

Сравнения

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

with

- `__enter__`
- `__exit__`

Атрибути

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

Итератори

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

Колекции

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

Метаатрибути

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

Функции

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

Въпроси?