14. Метаобектен протокол

23 ноември 2023

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 “протокол”?
- 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, мета-хумор

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

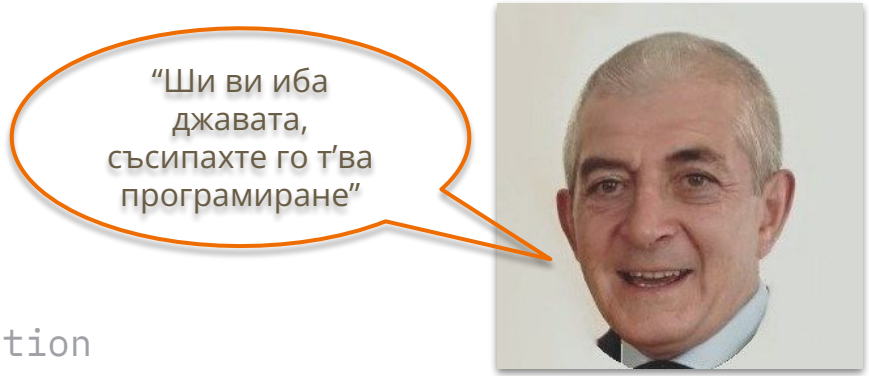
```
#ifdef X
  #include <smthng>
  int x = X;
#else
  #define X 42
  #include <smthng_else>
#endif

/* 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

Технически погледнато използват техники/идеи от метапрограмирането, но терминологично не са метапрограмиране...

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

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

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

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)`
- `__matmul__(self, a)`

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)`
- `__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)`
- `__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 Top:
...     def __gt__(self, other):
...         print('Greatest!')
...         return True
...
>>> 4 < Top()
Greatest!
True
>>> (42).__lt__(Top())
NotImplemented
```

Сравнения (3)

- нямаме десен вариант на `__lt__`
- вместо него се ползва `__gt__`
- аналогично за останалите оператори за сравнение

with

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

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)`

Въпроси?