13. Метапрограмиране

29 ноември 2022

Но първо - още малко метаобектен протокол...

Импортиране на модули

```python
import module
```

…съответства на…

```python
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])
```

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

Метапрограмиране!

Две идеи:

- Работа с класове (типове)
- Работа с функции (методи)

Преговор!

```
isinstance(3, int)            # True
isinstance(3, object)         # True
isinstance(3, str)            # False
isinstance(int, type)         # True
isinstance(3, type)           # False
isinstance('hello', str)      # True
```

Преговор?

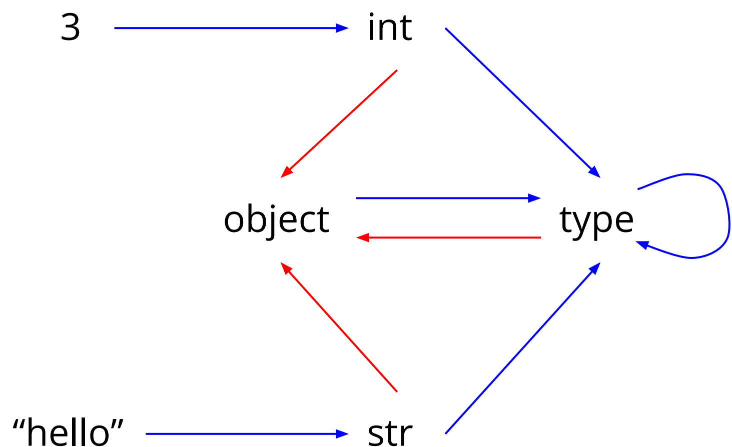
```
issubclass(int, object) # True
issubclass(object, int) # False
issubclass(int, int)    # True
issubclass(3, int)      # TypeError: [...] must be a class
issubclass(int, type)   # False
```

Преговор?!?!?!?!

```
isinstance(type, type)          # True
issubclass(type, type)          # True
isinstance(object, object)      # True
issubclass(object, object)      # True
isinstance(type, object)        # True
issubclass(type, object)        # True
isinstance(object, type)        # True
issubclass(object, type)        # False
```

isintance изобразен

- **issubclass** обхожда **__bases__** и търси съвпадение
- **isinstance(x, t) == issubclass(x.__class__, t)**



метакласове

- всичко в Пайтън е обект, включително и класовете
- всеки обект е инстанция на някакъв клас, включително и класовете
- класовете на класовете си имат специално име - метакласове
- *type* е метаклас на току-що разгледаните типове

Какво всъщност е *type*?

- без аргументи е просто класът type
- с един аргумент type(x) връща типа на x
- с три аргумента се конструира инстанция на type:

```
type(name, bases, dict)
```

Пример за type(name, bases, dict)

```
def init_person(self, name): self.name = name

def say_hi(self): print(f'Hi, My name is {self.name}')

Person = type('Person', (), {
    '__init__': init_person,
    'say_hi': say_hi,
})

Person('George').say_hi()
```

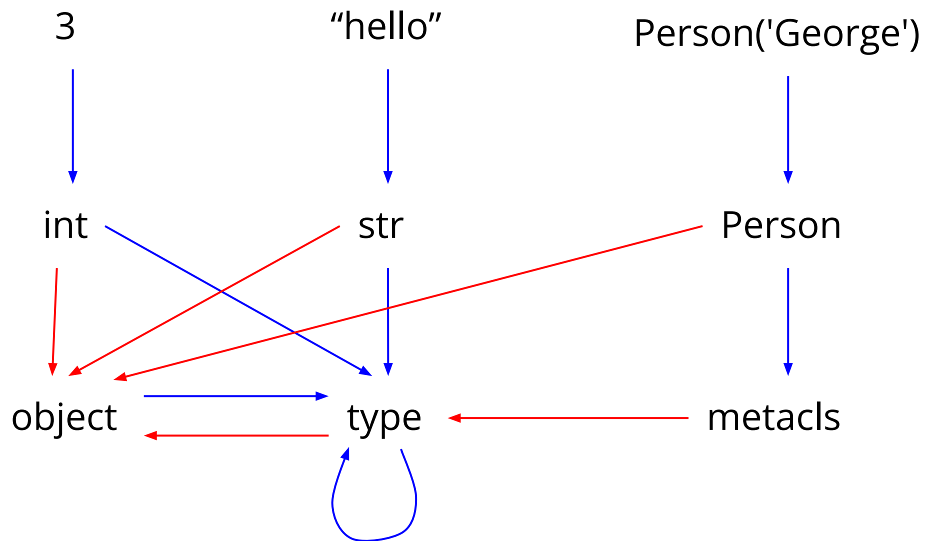
Наследяване от type

```
class metacls(type):
    def __new__(cls, name, bases, dict):
        dict['say_bye'] = lambda self: print('bye')
        return type.__new__(cls, name, bases, dict)

Person = metacls('Person', (), {
    '__init__': init_person,
    'say_hi': say_hi,
})

Person('George').say_bye()
```

metacls изобразен



Синтактична захар

```
class Foo(A, B, C, metaclass=Bar):
    x = 1
    y = 2

# е захар за:
Foo = Bar('Foo', (A, B, C), {'x': 1, 'y': 2})
```

Синтактична захар (2)

```
class Person(metaclass=metacls):
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print(f'Hi, My name is {self.name}')

Person('George').say_bye() # bye
```

Безкористен питон

```
def without_ego(func):
    def wrapped(self, *args, **kwargs):
        old_self = func.__globals__.get('self')
        func.__globals__['self'] = self
        result = func(*args, **kwargs)
        func.__globals__['self'] = old_self
        return result
    wrapped.__name__ = func.__name__
    return wrapped

class selfless(type):
    def __new__(cls, name, bases, attrs):
        for key, value in attrs.items():
            if hasattr(value, '__call__'):
                attrs[key] = without_ego(value)
        return type.__new__(cls, name, bases, attrs)
```

Безкористна нинджа!

```
class Person(metaclass=selfless):
    def __init__(name):
        self.name = name

    def say_hi():
        print(f'Hi, I am {self.name}')

Person("忍者").say_hi()
```

And Now for Something Completely Different

Как да си сглобим код?

- eval - оценява един израз
- exec - оценява парче код
- compile - малко по-сложно е

Избягвайте ги; податливи са на code injection

Примерна функция

```python
from math import pi

def circle_area(r):
    return pi * (r ** 2)

print(circle_area.__code__.co_code)
# b't\x00j\x01|\x00d\x01\x13\x00\x14\x00S\x00'
```

import dis

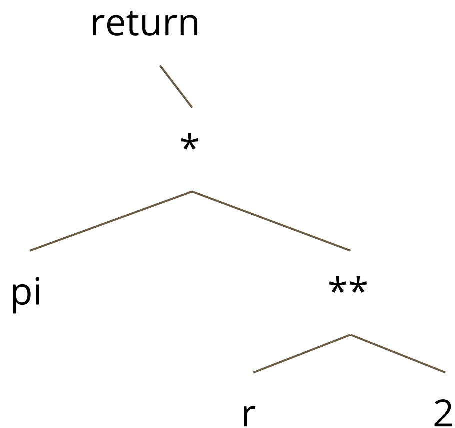
```
>>> import dis
>>> dis.dis(circle_area.__code__.co_code)
          0 LOAD_GLOBAL              0 (0)
          4 LOAD_FAST                0 (0)
          6 LOAD_CONST               1 (1)
          8 BINARY_POWER
         10 BINARY_MULTIPLY
         12 RETURN_VALUE
```

WTF?

- Нарича се байткод
- Не е [точно] асемблер
- Приликата с асемблер не е случайна
- Това са инструкции
- Нека опитаме да разберем какво означава...

Абстрактно синтактично дърво (AST)

- Дървовидна репрезентация на програма
- Тялото на **circle_area** (грубо) изглежда така:



Инфиксен запис

- Обхождаме AST в дълбочина
- Обхождаме в ред ляво-корен-дясно
- Получаваме:
  (return (pi * (r ** 2)))
- Забележка 1: ако знаем приоритетите не са нужни скоби
- Забележка 2: като “нормален” език за програмиране

Префиксен (полски) запис

- Обхождаме AST в дълбочина
- Обхождаме в ред корен-ляво-дясно
- Получаваме:
  (return (* pi (** r 2)))
- Забележка 1: Това е валиден Lisp (Scheme) код:
  (* pi (expt r 2))
- Забележка 2: ако знаем арността не са нужни скоби

Суфиксен (обратен полски) запис

- Обхождаме AST в дълбочина
- Обхождаме в ред ляво-дясно-корен
- Получаваме:
  ((pi (r 2 **) *) return)
- Забележка 1: ако знаем арността не са нужни скоби
- Забележка 2: това е валиден PostScript код:
  PI r 2 exp mul

Байткод!

((pi (r 2 **) *) return) - със скоби
pi r 2 ** * return - без скоби

```
LOAD_GLOBAL 0      ⇔   pi
LOAD_FAST 0        ⇔   r
LOAD_CONST 1       ⇔   2
BINARY_POWER       ⇔   **
BINARY_MULTIPLY    ⇔   *
RETURN_VALUE       ⇔   return
```

Байткод (2)

- Python байкода е обратен полски запис на AST-то
- ... подобно на JVM (Java)
- ... подобно на .NET (C#)
- ... подобно на p-code (Pascal)

import ast

Модул, с който да сглобяваме AST/код

Въпроси?