18. Метапрограмиране

29 април 2026

Преговор?

- `type(b'abcd')`
  `<class 'bytes'>`
- `b'абвг'`
  `# SyntaxError: bytes can only contain ASCII literal characters`

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?!?!?!?!

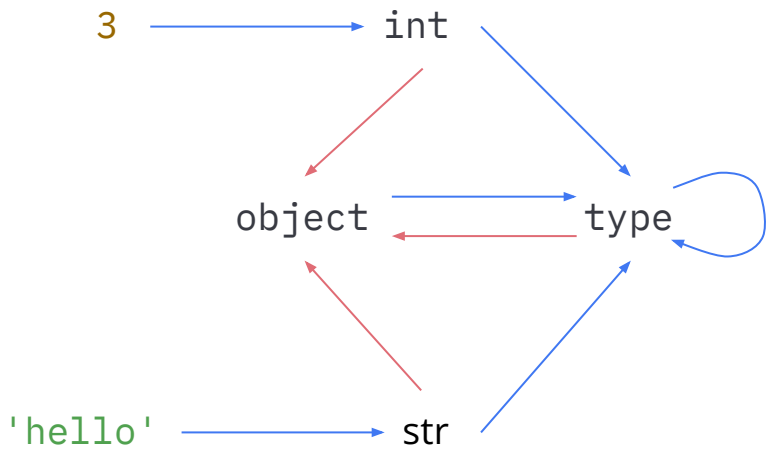
- `isinstance(type, type)`
  `# True`
- `issubclass(type, type)`
  `# True`
- `isinstance(object, object)`
  `# True`
- `issubclass(object, object)`
  `# True`
- `isinstance(type, object)`
  `# True`
- `issubclass(type, object)`
  `# True`
- `isinstance(object, type)`
  `# True`
- `issubclass(object,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

Класове

- Всичко в Пайтън е обект, включително и класовете
- Всеки обект е инстанция на някакъв клас, включително и класовете

isintance изобразен

- `issubclass(t, t) == True` (всеки клас е подклас на себе си)
- В останалите случаи `issubclass` обхожда `__bases__` и търси съвпадение
- `isinstance(x, t) == issubclass(x.__class__, t)`



Метаклас

- Класът на класовете си има специално име - метаклас
- `type` е метакласът на току-що разгледаните типове (`int`, `str`, `object`)
- `type` е също така метакласът на `type`
- ... има и други.

Какво всъщност е *type*?

- Без аргументи е просто класът type
- С аргументи е конструктор
- Пример за `__new__` магия
- 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__(self, name):
    self.name = name

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

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

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

Синтактична захар

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

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

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

Обобщение

- Създаваме класове със синтаксиса `class Something`
- ... или ги създаваме с конструктора `type`
- Второто ни позволява динамични:
  - имена на типове
  - полета на класа
  - базови типове
  - ... и още!
- Програма която създава нови типове докато работи!
- Програма която се пише/дописва сама!?!

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

```
public interface Foo {
    Object bar(Object obj) throws BazException;
}

public class FooImpl implements Foo {
    Object bar(Object obj) throws BazException {
        // ...
    }
}

public class DebugProxy implements java.lang.reflect.InvocationHandler {

    private Object obj;

    public static Object newInstance(Object obj) {
        return java.lang.reflect.Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new DebugProxy(obj));
    }

    private DebugProxy(Object obj) {
        this.obj = obj;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        Object result;
        try {
            System.out.println("before method " + m.getName());
            result = m.invoke(obj, args);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        } catch (Exception e) {
            throw new RuntimeException("unexpected invocation exception: " +
                                       e.getMessage());
        } finally {
            System.out.println("after method " + m.getName());
        }
        return result;
    }
}
```

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

Възможности...

До къде може да стигне Python?

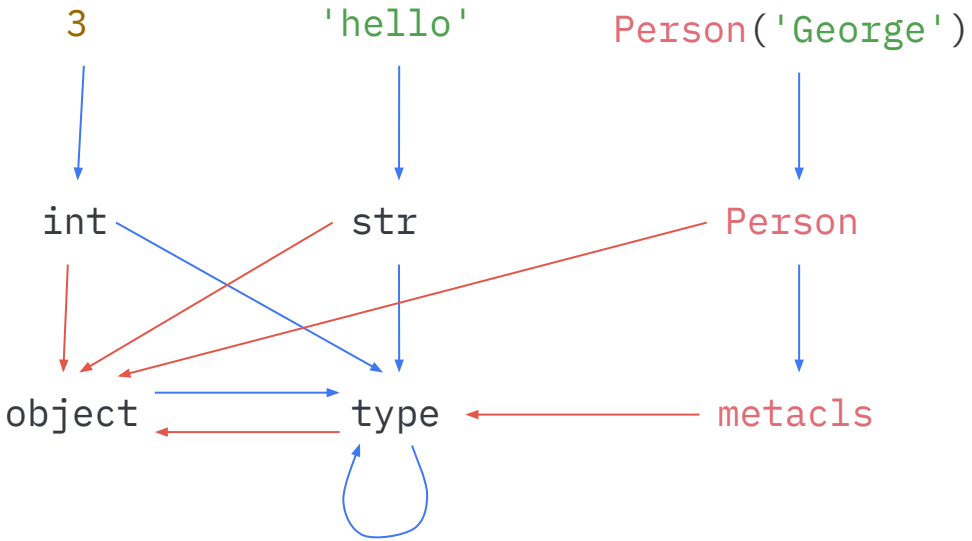
Наследяване от type

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

- Не изглежда типа `Person` да има `say_bye`
- Изглежда като магия!

Пример - enum

- В Python няма синтаксис за енумерации
- ... но можем да си напишем наш с метапрограмиране
- ... или да използваме готовия!

```
>>> from enum import Enum
>>> type(Enum)
<class 'enum.EnumType'>
>>> #   ^^^^^^^^^^^^^ метаклас!
>>> enum.EnumType.__bases__
(<class 'type'>,)
```

Пример - enum (2)

```
>>> class Weekday(Enum):
...     MONDAY = 1
...     TUESDAY = 2
...     WEDNESDAY = 3
...     THURSDAY = 4
...     FRIDAY = 5
...     SATURDAY = 6
...     SUNDAY = 7
...
>>> type(Weekday)         # получваме метаклас и няколко неща наготово
<class 'enum.EnumType'>
>>> Weekday.MONDAY        # стандартно ползване
<Weekday.MONDAY: 1>
>>> Weekday(3)            # търсене по стойност
<Weekday.WEDNESDAY: 3>
>>> Weekday.MONDAY.name   # името като низ
'MONDAY'
>>> Weekday.MONDAY.value # стойност
1
```

Пример - enum (3)

```
>>> Weekday.MONDAY in Weekday
True
>>> "Something" in Weekday
False
>>> 3 in Weekday
True
>>> 8 in Weekday
False
>>> for day in Weekday:
...     print(day)
…
Weekday.MONDAY
Weekday.TUESDAY
Weekday.WEDNESDAY
Weekday.THURSDAY
Weekday.FRIDAY
Weekday.SATURDAY
Weekday.SUNDAY
```

На кого му трябва синтаксис?

- В python няма синтаксис за:
  - properties
  - статични методи
  - абстрактни методи
  - enum
  - статична типизация
  - generics
  - ... и още неща
- ... постигат се чрез комбинация от:
  - декоратори
  - метакласове
  - `__getattr__` и приятели
  - дескриптори
  - други неща в метаобектния протокол
  - анотации

[Типови] анотации

```python
def split_names(full_name: str) -> tuple[str, str] | tuple[str, str, str]:
    names = full_name.split()
    if 2 <= len(names) <= 3:
        return tuple(names)
    raise ValueError(f"Invalid name '{full_name}'")

split_names("Michael Trent Reznor") # ('Michael', 'Trent', 'Reznor')
split_names("David Robert Jones") # ('David', 'Robert', 'Jones')
```

- Няма такова нещо като компилационни грешки (очевидно)
- Проверка с външни инструменти (напр. MyPy, Pytype, Pyright, Pyre)
- Или пък IDE (напр. vscode)
- Разни инструменти очакват това да са типове; Python не ви задължава

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.
- Подобна на `__*attr__` методите, но за нещото вдясно от точката.
- 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'
>>> victor = Lecturer("Victor")
>>> victor.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__
>>> list(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'
```

Абстрактни класове/методи

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def vocalize(self):
        pass

class Dog(Animal):
    def vocalize(self):
        print("Woof!")

class EricBurdon(Animal):
    def vocalize(self):
        print("There is a house in New Orleans, they caaall…")


Dog().vocalize() # Woof!
Animal() # TypeError: Can't instantiate abstract class Animal [...]
```

Абстрактни класове/методи (2)

```
>>> type(Animal)
abc.ABCMeta
>>> abc.ABCMeta.__bases__
(type,)
>>> Animal.__abstractmethods__
frozenset({'vocalize'})
```

Selfless python

```
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()
```

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
- Има очаквания към __dunder__ атрибутите и ги ползва по по-специални начини

Въпроси?