17. Метапрограмиране

05 декември 2024

Преговор?

- `type(b'abcd')`
- `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?!?!?!?!

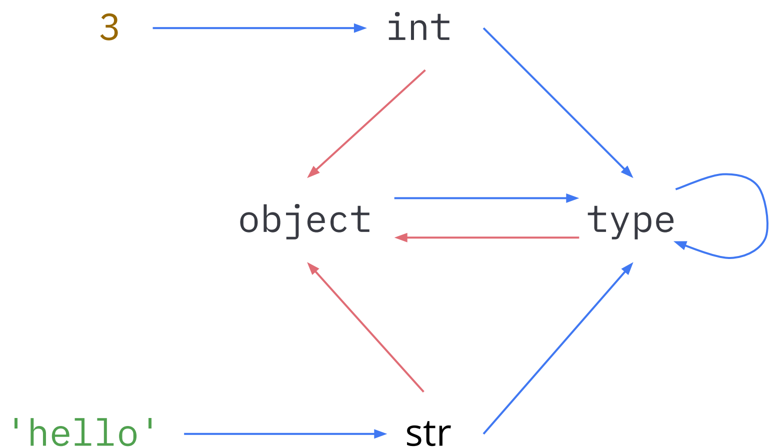
- `isinstance(type, type)`
  `# True`
- `issubclass(type, type)`
  `# True`
- `isinstance(object, object)`
  `# True`
- `issubclass(object, object)`
  `# True`
- `isinstance(type, object)`
  `# True`
- `issubclass(type, object)`
  `# True`
- `isinstance(object, type)`
  `# True`
- `issubclass(object,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

```java
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;
    }
}
```

“Ши ви иба джавата, съсипахте го т’ва програмиране”

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?

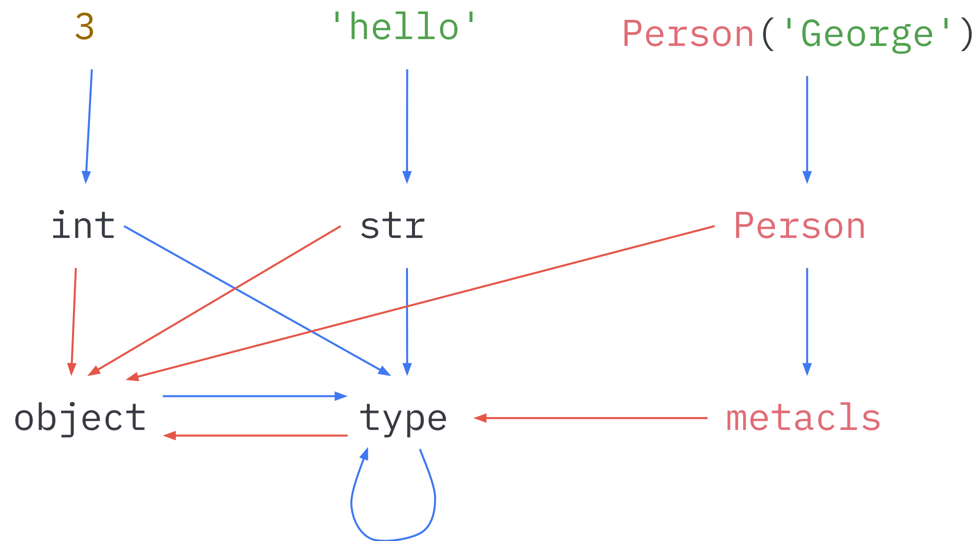
Наследяване от type

```
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

```python
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)

```python
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 кой 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

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'})
```

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)

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()
```

Код или данни?

```
def print_song(artist, song, album = None):
    result = f'The song "{song}"'
    if album:
        result += f' from the album "{album}"'
    result += f' is by {artist}'
    print(result)

print_song("David Bowie", "Earthling", "I'm Afraid of Americans")
print_song("Black Sabbath", "Black Sabbath", "Black Sabbath") # true story
```

Код или данни? (2)

```
>>> dir(print_song)
['__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__']
>>> print_song.__name__
'print_song'
>>> print_song.__class__
<class 'function'>
>>> print_song.__defaults__
(None,)
```

Код или данни? (3)

```
>>> repr(print_song.__code__)
'<code object print_song at 0x000001C0DAA6C7A0, file "<pyshell#51>", line 1>'
>>> print_song.__code__.co_argcount # брой аргументи
3
>>> print_song.__code__.co_filename
'<stdin>'
>>> print_song.__code__.co_firstlineno
1
>>> print_song.__code__.co_stacksize
4
>>> print_song.__code__.co_names
('print',)
>>> print_song.__code__.co_code
b'\x97\x00d\x01|\x01\x9b\x00d\x02\x9d\x03}\x03|\x02r\t|\x03d\x03|\x02\x9b\x00d
\x02\x9d\x03z\r\x00\x00}\x03|\x03d\x04|\x00\x9b\x00\x9d\x02z\r\x00\x00}\x03t\x
01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00|\x03\xa6\x01\x00\x00\xab\x01\x00\x0
0\x00\x00\x00\x00\x00\x00\x01\x00d\x00S\x00'
```

ДИС

```
>>> dis.dis(print_song.__code__.co_code)
          0 RESUME                   0
          2 LOAD_CONST               1
          4 LOAD_FAST                1
          6 FORMAT_VALUE             0
          8 LOAD_CONST               2
         10 BUILD_STRING             3
         12 STORE_FAST               3
         14 LOAD_FAST                2
         16 POP_JUMP_FORWARD_IF_FALSE     9 (to 36)
         18 LOAD_FAST                3
         20 LOAD_CONST               3
         22 LOAD_FAST                2
         24 FORMAT_VALUE             0
         26 LOAD_CONST               2
         28 BUILD_STRING             3
         30 BINARY_OP               13 (+=)
         34 STORE_FAST               3
    >>   36 LOAD_FAST                3
         38 LOAD_CONST               4
         40 LOAD_FAST                0
         42 FORMAT_VALUE             0
         44 BUILD_STRING             2
         46 BINARY_OP               13 (+=)
         50 STORE_FAST               3
         52 LOAD_GLOBAL              1
         64 LOAD_FAST                3
         66 PRECALL                  1
         70 CALL                     1
         80 POP_TOP
         82 LOAD_CONST               0
         84 RETURN_VALUE
```

ДИС 2

```
>>> import dis
>>> dis.dis(print_song)
  1           0 RESUME                   0

  2           2 LOAD_CONST               1 ('The song "')
              4 LOAD_FAST                1 (song)
              6 FORMAT_VALUE             0
              8 LOAD_CONST               2 ('"')
             10 BUILD_STRING             3
             12 STORE_FAST               3 (result)

  3          14 LOAD_FAST                2 (album)
             16 POP_JUMP_FORWARD_IF_FALSE     9 (to 36)

  4          18 LOAD_FAST                3 (result)
             20 LOAD_CONST               3 (' from the album "')
             22 LOAD_FAST                2 (album)
             24 FORMAT_VALUE             0
             26 LOAD_CONST               2 ('"')
```

ДИС 3

```
             28 BUILD_STRING             3
             30 BINARY_OP               13 (+=)
             34 STORE_FAST               3 (result)

5      >>    36 LOAD_FAST                3 (result)
             38 LOAD_CONST               4 (' is by ')
             40 LOAD_FAST                0 (artist)
             42 FORMAT_VALUE             0
             44 BUILD_STRING             2
             46 BINARY_OP               13 (+=)
             50 STORE_FAST               3 (result)

6            52 LOAD_GLOBAL              1 (NULL + print)
             64 LOAD_FAST                3 (result)
             66 PRECALL                  1
             70 CALL                     1
             80 POP_TOP
             82 LOAD_CONST               0 (None)
             84 RETURN_VALUE
```

Инструкции

- Можем да прозиведем **изцяло** нов код
- Можем да сглобяваме нов at runtime
- За упражнение вкъщи!

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

Въпроси?