OOP and Python

Alexander Evgin

13 марта 2020 г.

Объектно-ориентированное программирование

- ООП — не закон, а парадигма (методология, шаблон, набор рекомендаций)
- Большая кодовая база (Smalltalk, 1980)
- Цели:
  - более понятная структура программы
  - приближение структуры кода "к жизни"(интуитивность)
  - компонентный подход
- Реализация не стандартизирована (зависит от языка)

Объектно-ориентированное программирование

0. Абстракция
1. Инкапсуляция
2. Наследование
3. Полиморфизм

Объекты в Python

Объект — "кусок памяти"

- identity (не меняется)
- type (не меняется)
- value (может меняться)

В Python ***всё*** — объекты

- Переменная — ссылка на объект (assignment)
- ***Атрибуты*** объекта — ссылки на другие объекты

Классы

- ***Класс*** — описание собственного типа объекта
- Создание класса — синтаксис объявления класса (но не только! см. метапрограммирование)
- Объект — ***экземпляр*** (instance) класса
- Возможность следовать ООП

Простой класс

```
class Counter:
    """I am a Counter, I count stuff."""

    def __init__(self, initial_count=0):
        self.count = initial_count

    def get(self):
        return self.count

    def increment(self):
        self.count += 1

c = Counter(initial_count=91)
c.increment()
print(c.get())
```

Атрибуты

```
class Counter:
    all_counters = []  # class attribute

    def __init__(self, initial_count=0):
        Counter.all_counters.append(self)
        # no explicit field declaration
        self.count = initial_count


c1 = Counter(92)
c2 = Counter(62)
assert len(Counter.all_counters) == 2
assert c1.all_counters is c2.all_counters
```

__dict__

```
>>> c = Counter(92)
>>> c.__class__
<class '__main__.Counter'>
>>> c.__dict__
{'count': 92}
>>> c.count == c.__dict__["count"]
True
>>> c.__dict__["foo"] = 62
>>> c.foo
62
>>> del c.foo
>>> del c.__dict__["count"] # ~= .pop("count")
>>> vars(c)  # ~= c.__dict__
{}
```

Класс это объект

```
>>> (Counter.__name__, Counter.__doc__, Counter.__module__)
('Counter', 'I am a Counter.', '__main__')
>>> Counter.__bases__
(<class 'object'>,)
>>> Counter.__dict__
mappingproxy({
    'all_counters': [],
    '__init__': <function Counter.__init__ ...>,
    'get': <function Counter.get ...>,
    'increment': <function Counter.increment ...>,
})
```

Класс это statement

```
>>> class Weird:
...     f1, f2 = 0, 1
...     for _ in range(10):
...         f1, f2 = f2, f1 + f2
...
>>> Weird.f1
55
```

__dict__ класса -- результат выполнения тела класса

Attribute search order

Read:

- instance __dict__
- class __dict__ (bases)

Assign:

- instance __dict__

“Copy on write”

Bound Methods

```
>>> class A:
...     def foo(self):
...         pass
...
>>> a = A()
>>> a.foo
<bound method A.foo of <__main__.A object ... >>
>>> A.foo
<function A.foo ...>
>>> a.foo is A.foo
False
```

Functions vs Methods

```
1   class A:
2       y = 1
3
4       def __init__(self):
5           self.x = 0
6
7       def get_x(self):
8           return self.x
9
10  a = A()
11
```

```
>>> type(A.get_x)
<class 'function'>
>>> type(a.get_x)
<class 'method'>
```

@classmethod

```
1   class A:
2       y = 1
3
4       @classmethod
5       def get_y(cls):
6           return cls.y
7
8   a = A()
9
```

```
>>> type(A.get_y)
<class 'method'>
>>>
```

@staticmethod

```
1   class RequestHandler:
2       def other_method(self, addr):
3           addr = self.unwrap_address(addr)
4           ...
5
6       @staticmethod
7       def unwrap_address(addr):
8           unwrapped = [b if b is not None else 0
9                        for b in addr]
10          return unwrapped
11
```

Properties

```
class Counter:
    def __init__(self, initial_count=0):
        self.count = initial_count

    def increment(self):
        self.count += 1

    @property
    def is_zero(self):
        return self.count == 0


c = Counter()
assert c.is_zero  # нет `()`
c.increment()
assert not c.is_zero
```

```
class Temperature:
    def __init__(self, *, celsius=0):
        self.celsius = celsius

    @property
    def fahrenheit(self):
        return self.celsius * 9 / 5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5 / 9

    @fahrenheit.deleter
    def fahrenheit(self):
        del self.celsius


c = Temperature()
c.fahrenheit = 451
assert c.celsius == 232.77777777777777
```

__slots__

```
>>> class A:
...     __slots__ = ["x", "y"]  # экономим память
...
>>> a = A()
>>> a.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__dict__'
>>> a.x = 92
>>> a.z = 92
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'z'
```

Управление доступом

Соглашения и mangling

```
class A:
    def __init__(self):
        self.pub = 92
        self._priv = 62
        self.__mangled = 42


a = A()
assert a.pub == 92
assert a._priv == 62
assert a._A__mangled == 42
```

Наследование

```
class Counter:
    def __init__(self, initial_count=0):
        self.count = initial_count

    def get(self):
        return self.count


class SquaredCounter(Counter):
    def get(self):
        return super().get() ** 2


c = SquaredCounter(91)
assert c.get() == 8281
```

Наследование

```
assert isinstance(c, Counter)
assert issubclass(SquaredCounter, Counter)
assert issubclass(Counter, (str, object))
```

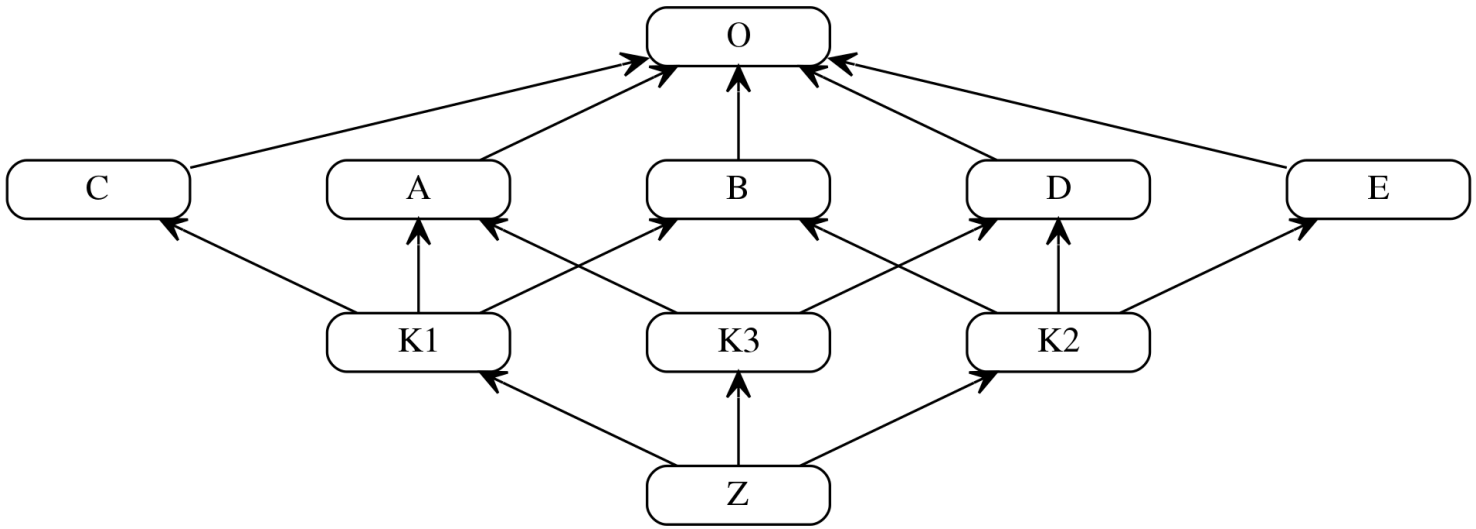
Наследование

```
class A:
    def f(self):
        print("A")

class B:
    def f(self):
        print("B")

class C(A, B):
    pass

C().f()
# A
```

```
class Base:
    def f(self):
        print("Base")

class A(Base):
    def f(self):
        print("A")
        super().f()  # super is dynamic!

class B(Base):
    def f(self):
        print("B")
        super().f()

class C(A, B):
    pass

C().f()
# A
# B
# Base

assert C.mro() == [C, A, B, Base, object]
```

Method Resolution Order (MRO)

C3 superclass linearization



- local precedence order
- monotonicity

Mixin

```
class DoublingMixing:  # !!!
    def increment(self):
        super().increment()
        super().increment()


class DoublingCounter(DoublingMixing, Counter):
    pass


c = DoublingCounter()
assert c.count == 0
c.increment()
assert c.count == 2
```

“Магические” ✨ методы

```
class Counter:
    def __init__(self, initial_count):
        self.count = initial_count

    def __lt__(self, other):
        return self.count < other.count

    def __eq__(self, other):
        return self.count == other.count


c1 = Counter(62)
c2 = Counter(92)
assert c1 < c2
assert (62).__lt__(92)
assert c2 >= c1  # упадёт, нет __ge__
```

“Магические” ✨ методы

```
class Counter:
    def __init__(self, initial_count):
        self.count = initial_count

    def __repr__(self):
        return "Counter({})".format(self.count)

    def __str__(self):
        return "Counted to {}".format(self.count)


c = Counter(92)
assert str(c) == f"{c}" == "Counted to 92"
assert repr(c) == f"{c!r}" == "Counter(92)"
```

“Магические” ✨ методы

```
class Counter:
    def __init__(self, initial_count):
        self.count = initial_count

    def __hash__(self):
        # NB: a == b => hash(a) == hash(b)
        return hash(self.count)

    def __eq__(self, other):
        return self.count == other.count


assert len({Counter(92), Counter(92)}) == 1
```

“Магические” ✨ методы

```
class Counter:
    def __init__(self, initial_count):
        self.count = initial_count

    def __bool__(self):
        return self.count > 0


c = Counter(0)

if not c:
    print("empty")
```

“Магические” ✨ методы

```
class Counter:
    def __init__(self, initial_count):
        self.count = initial_count

    def __add__(self, other):
        if not isinstance(other, int):
            return NotImplemented
        return Counter(self.count + other)

    def __radd__(self, other):
        return self + other


c = Counter(0)

assert (c + 1).count == 1
assert (1 + c).count == 1
```

“Магические” ✨ методы

```
class Identity:
    def __call__(self, x):
        return x


assert Identity()(92) == 92
```

Магические методы

```
class A:
    def __init__(self):
        self.part1 = list(range(10))
        self.part2 = list(range(100, 110))

    def __getitem__(self, key):
        return self.part1[key], self.part2[key]

    def __setitem__(self, key, value):
        assert len(value) == 2
        self.part1[key], self.part2[key] = value[0], value[1]

a = A()
assert a[1] == (1, 101)
a[-1] = 42, 42
assert a[9] == (42, 42)
```

Finally

0. Абстракция
1. Инкапсуляция
2. Наследование
3. Полиморфизм

Q&A