Классы. Продолжение

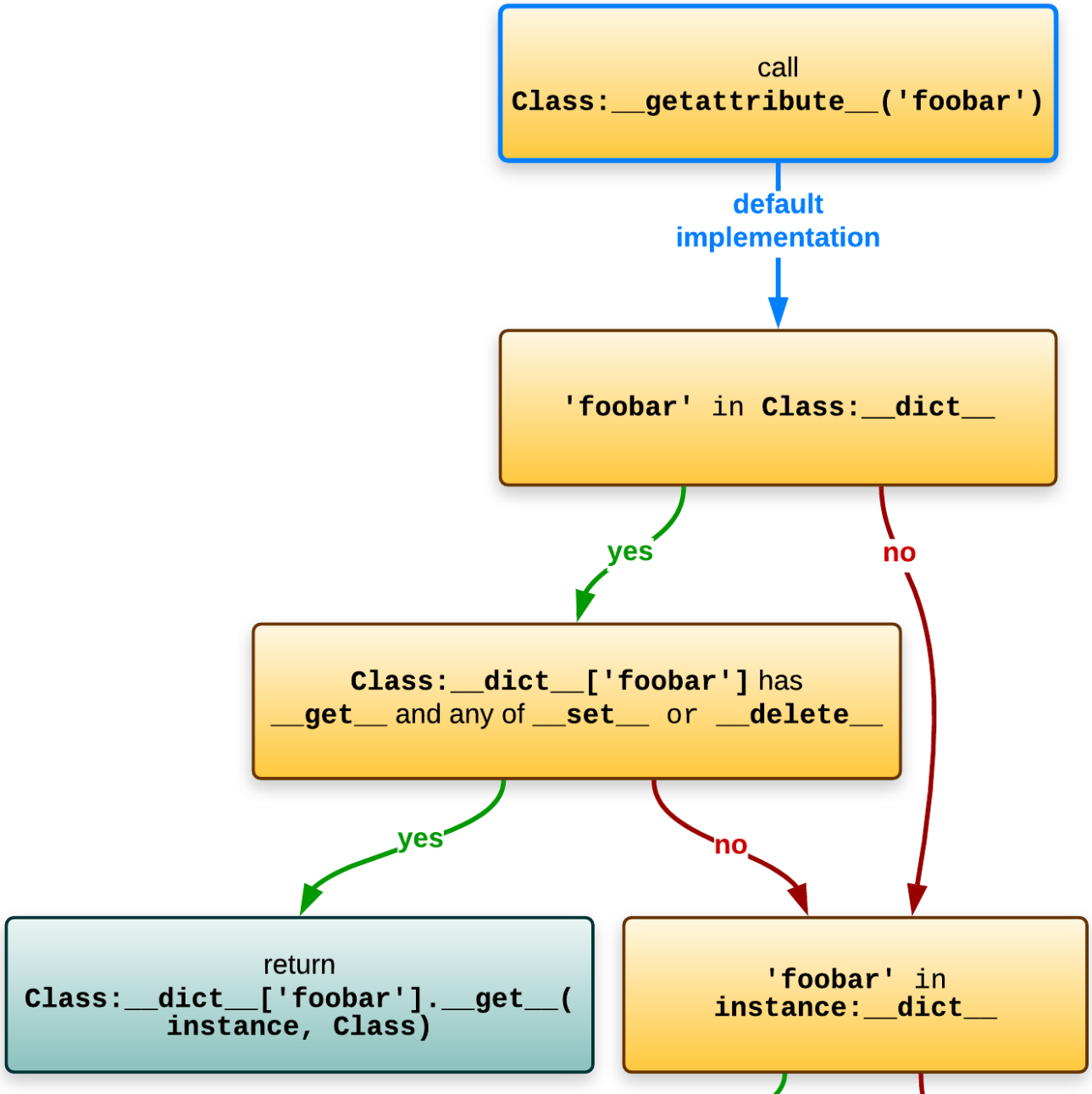
Поиск атрибутов

Вспомним неправду с пары про классы, там говорилось следующее:

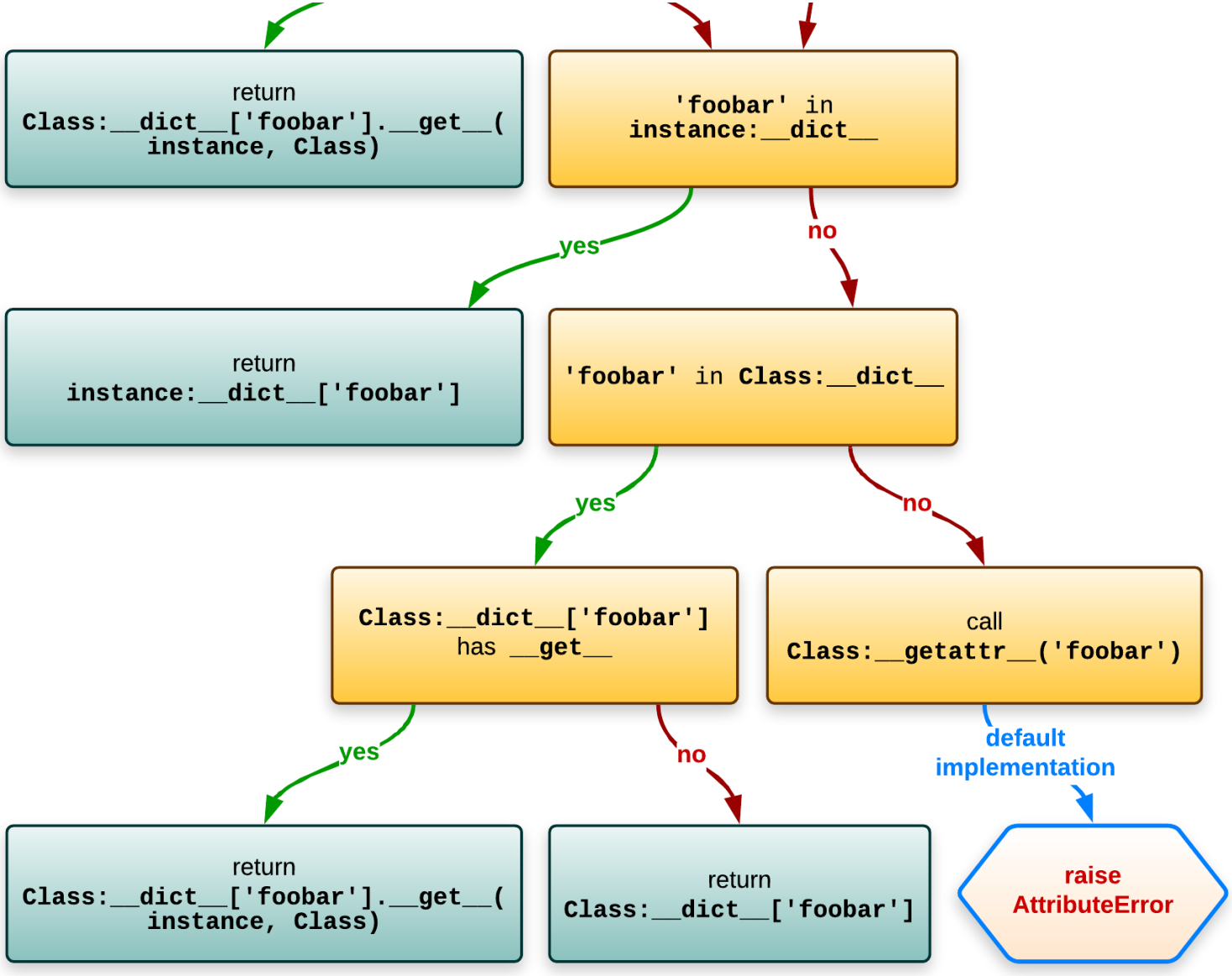
Интерпретатор ищет аттрибут `attr` в следующем порядке:

1. Зовем obj.__class__.__getattribute__ передавая наш объект, по умолчанию он проверяет
   a. Есть ли `attr` в obj.__dict__? Если есть, вернем
   b. Есть ли `attr` в obj.__class__.__dict__? Если есть, вернем
2. Если поиск неуспешный, зовем __getattr__

Поиск атрибутов на самом деле



Поиск атрибутов на самом деле



Дескрипторы

Объект, у которого определен хотя бы один магический метод: `__get__`, `__set__` или `__delete__` называют дескриптором:

Дескрипторы

```
class Desc:
    # instance -- экземпляр класса, в котором нашли дескриптор
    # owner -- сам класс
    def __get__(self, instance, owner):
        return 42

    # instance -- аналогично
    # value -- значение, котором хотим установить
    # def __set__(self, instance, value):
    #    print("set")

    # instance -- аналогично
    # def __delete__(self, instance):
    #    pass
```

Дескрипторы

```
class ClassWithDescriptor:
    forty_two = Desc()

c = ClassWithDescriptor()
c.__dict__['forty_two'] = 45 # Почему?
# c.forty_two = 43 # -- Тоже сломает. Почему?
print(c.forty_two)
# 45
```

Как работают методы?

```
from functools import partial

def foo():
    pass

# если посмотрим на атрибуты обычной функции, то узнаем, что
она является дескриптором
print(dir(foo))
# [ ...,
#   __get__,
#   ...]
# зачем?
```

Как работают методы?

Функция является дескриптором, чтобы при попытке извлечь метод из объекта получать не сам метод, но метод уже связанный с экземпляром.

```
class A:
    pass
def foo(self):
    self.some_value = 50
    return self

a = A()
# из-за того, как происходит поиск, a.foo() раскрывается в
a_foo = foo.__get__(a, A) # получили связанный с a метод
print(a_foo)
# <bound method foo of <__main__.A object at 0x10bbf5040>>
print(a_foo().some_value)
# 50
```

Как работают методы?

Т.е. на самом деле `__get__` мог быть реализован как-то так:

```
from functools import partial
class Method:
    # instance -- экземпляр класса, для которого достали
    # атрибут
    # owner -- класс этого экземпляра
    def __get__(self, instance, owner):
        # связали первый аргумент функции с
        # экземпляром, от которого был вызов
        return partial(self, instance)
```

Код с пары

Пример дескриптора property

staticmethod и classmethod

```
class Config:
    config_type = 'local'
    def __init__(self, name):
        self.name = name

    @staticmethod # static методы не связывают self
    def create_default():
        return Config("")
    # во все class методы передается не self
    @classmethod # но класс self
    def get_type(cls):
        # cls -- класс объекта, от которого вызвали
        # т.е. <class Config> в нашем случае
        return cls.config_type

config = Config.create_default()
print(type(config).get_type())
# print(config.get_type()) # тоже самое
```

Метачто?

> *Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).*
>
> — *Tim Peters*

объекты ← классы ← метаклассы

- Объекты создаются с помощью классов.
- Классы тоже объекты, их тоже создают классы. Но специальные и их называют `метаклассы`

Как создаются объекты?

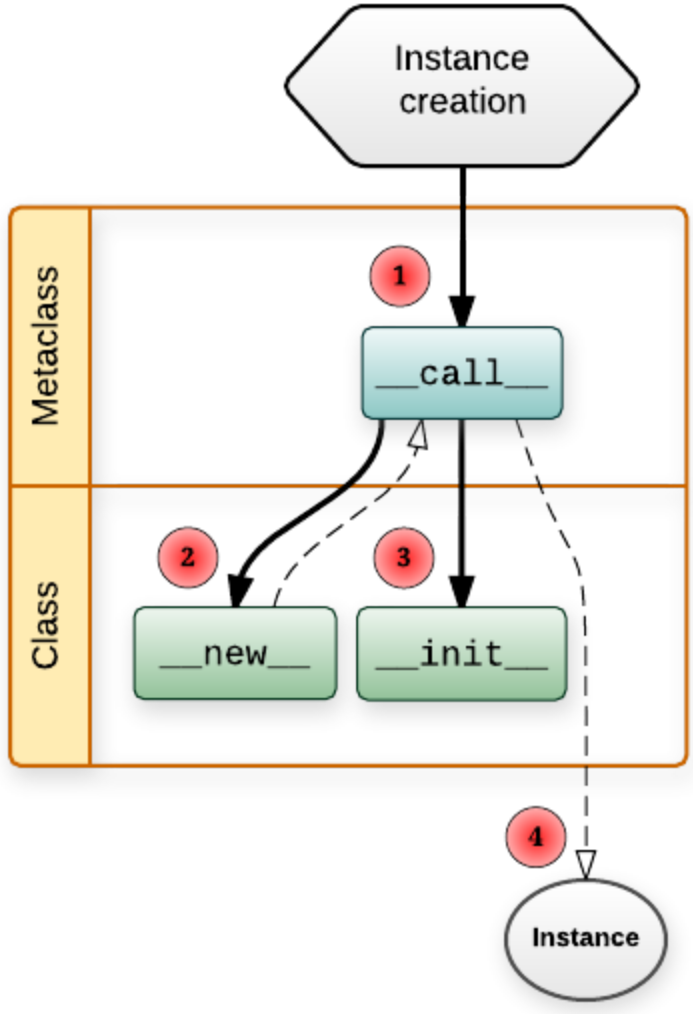
```
class A:
    pass

a = A()
# тоже самое, что и
init = A.__call__
a = init()
```

Чтобы создать новый объект нужно вызвать объект `A`, т.е. взять его метод `__call__` и выполнить.

Метода `__call__` нет у самого `A`, но он есть у метакласса, который его создал.

Как создаются объекты?



Первые попытки

```
class CursedMeta(type): # наследование обязательно!
    def __call__(self, *args, **kwargs):
        return [*args, *kwargs.items()]

class Bar(metaclass=CursedMeta):
    ...
class C(metaclass=CursedMeta):
    ...
class E(Bar, C):
    ...

print(Bar(150, 'creating_bar'))
# [150, 'creating_bar']
print(C(None))
# [None]
print(E(key='value'))
# [('key', 'value')]
```

Парадокс

Все метаклассы наследуются от `type`.

В т.ч. обычные классы создаются с помощью самого `type`:

```
class A:
    ...

print(type(A))
# <class 'type'>
#  ^   ^
# и сам type тоже класс
print(type(type(A)))
# <class 'type'>
```

Но кто создал `type`*?*

Откуда берется __call__

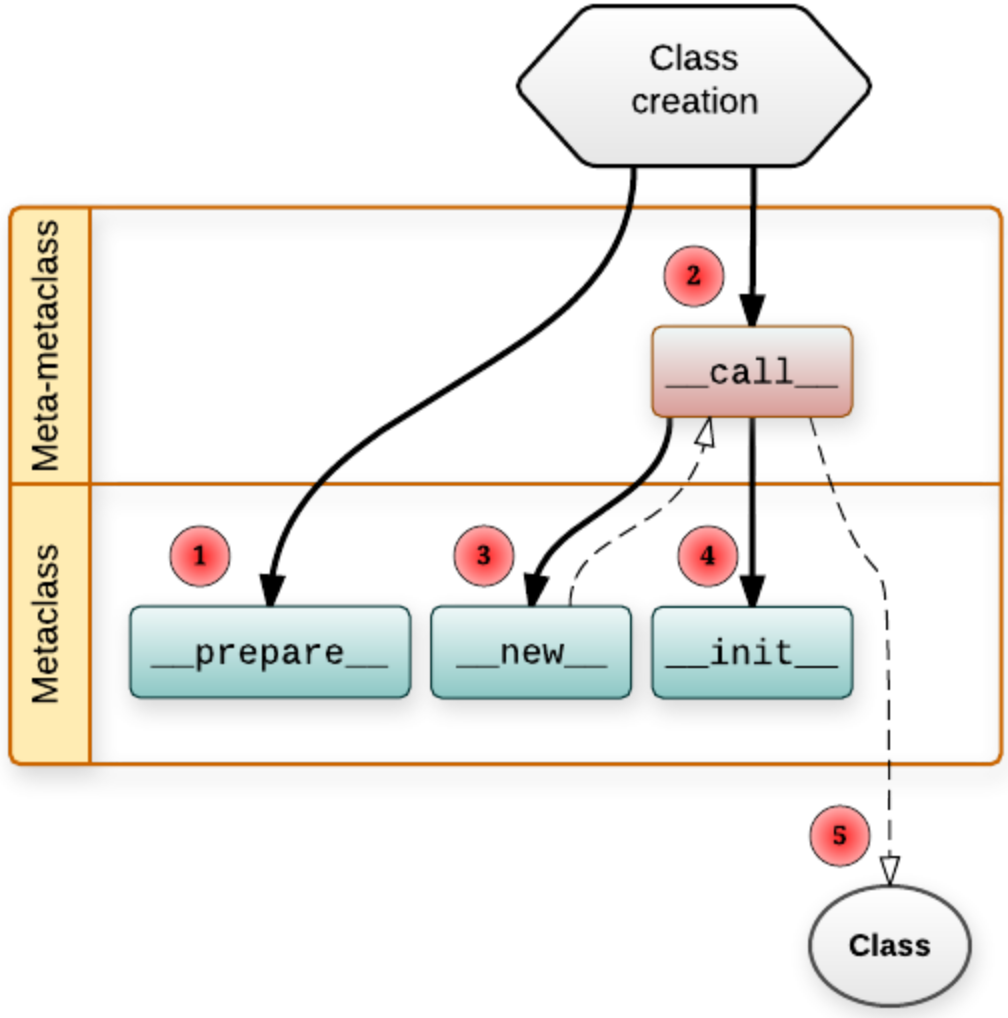
```
class Foo:
    pass

_ = Foo.__call__() # <=> Foo()
# но у Foo все еще нет своего __call__
print('__call__' in dir(Foo))
# False

# класс -- *объект*, созданный с помощью метакласса
# поэтому и разрешение атрибута происходит аналогично
# обычным объектам
# *(с некоторыми оговорками, но сейчас это не важно)*
# _ = <metaclass of Foo>.__dict__['__call__'].__get__(Foo)()
_ = type.__dict__['__call__'].__get__(Foo)()

# достанем класс, в котором __call__ определен:
print(Foo.__call__.__objclass__ is type)
# True
```

Как создаются классы?



Как создаются классы?

```
class Meta(type):
    @classmethod # обязательно classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        """ Должен создавать и возвращать словарь, который
        будет использоваться в качестве __dict__ класса

        mcs -- сам метакласс Meta
        name -- имя класса
        bases -- родители в порядке mro
        kwargs -- аргументы из class, напрмиер arg в коде:
        class A(metaclass=Meta, arg=10)
            """
        return OrderedDict() # бессмысленно с 3.7
    ...
```

Как создаются классы?

```
class Meta(type):
    def __new__(cls, name, bases, attrs, **kwargs):
        """ Создает и возвращает класс.

        cls -- *внезапно* сам метакласс,
        при этом не пишем @classmethod
        attrs -- словарь, который вернул prepare и в
        который положим атрибуты, здесь его можно изменять.
        Например, добавив новые методы или атрибуты и т.д.
            """
        print(attrs) # можно изменять словарь
        return super().__new__(cls, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        # аналогично, но cls уже создан, его нужно
        # инициализировать
        super().__init__(name, bases, attrs)
    ...
```

Как создаются классы?

```
class Meta(type):
    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        return OrderedDict()

    def __new__(cls, name, bases, attrs, **kwargs):
        return super().__new__(cls, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        super().__init__(name, bases, attrs)

    def __call__(self, *args, **kwargs):
        return super().__call__(self, *args, **kwargs)
```

Полезные ссылки

- Код с практики про метаклассы с комментариями.
- Блогпост, из которого были стащены картинки и часть рассказа.