Python2018

compscicenter.ru

aleksey.kladov@gmail.com

Лекция 10

Классы II

Напоминание

```
class A:
    X = 92

    def __init__(self, y):
        self.y = y

    def foo(self):
        print(self.y)


a = A(62)
a.y    # instance __dict__
a.X    # class __dict__
a.foo  # bound method?
```

Протокол Дескрипторов

```
descr.__get__(self, obj, type=None) -> value

descr.__set__(self, obj, value) -> None

descr.__delete__(self, obj) -> None
```

Протокол Дескрипторов*

```
obj.x
```

- Если x есть в obj.__dict__, вернуть obj.__dict__['x']
- Если x есть в Cls.__dict__
    - Если __get__ есть в Cls.__dict__['x'], вернуть
        - Cls.__dict__['x'].__get__(obj, Cls)
    - Вернуть Cls.__dict__['x']

Функции -- Дескрипторы

```
>>> def foo(x): return x
...
>>> foo.__get__
<method-wrapper '__get__' of function object ... >
>>> f = foo.__get__(92, int)
>>> f()
92
>>>
```

Протокол Дескрипторов*

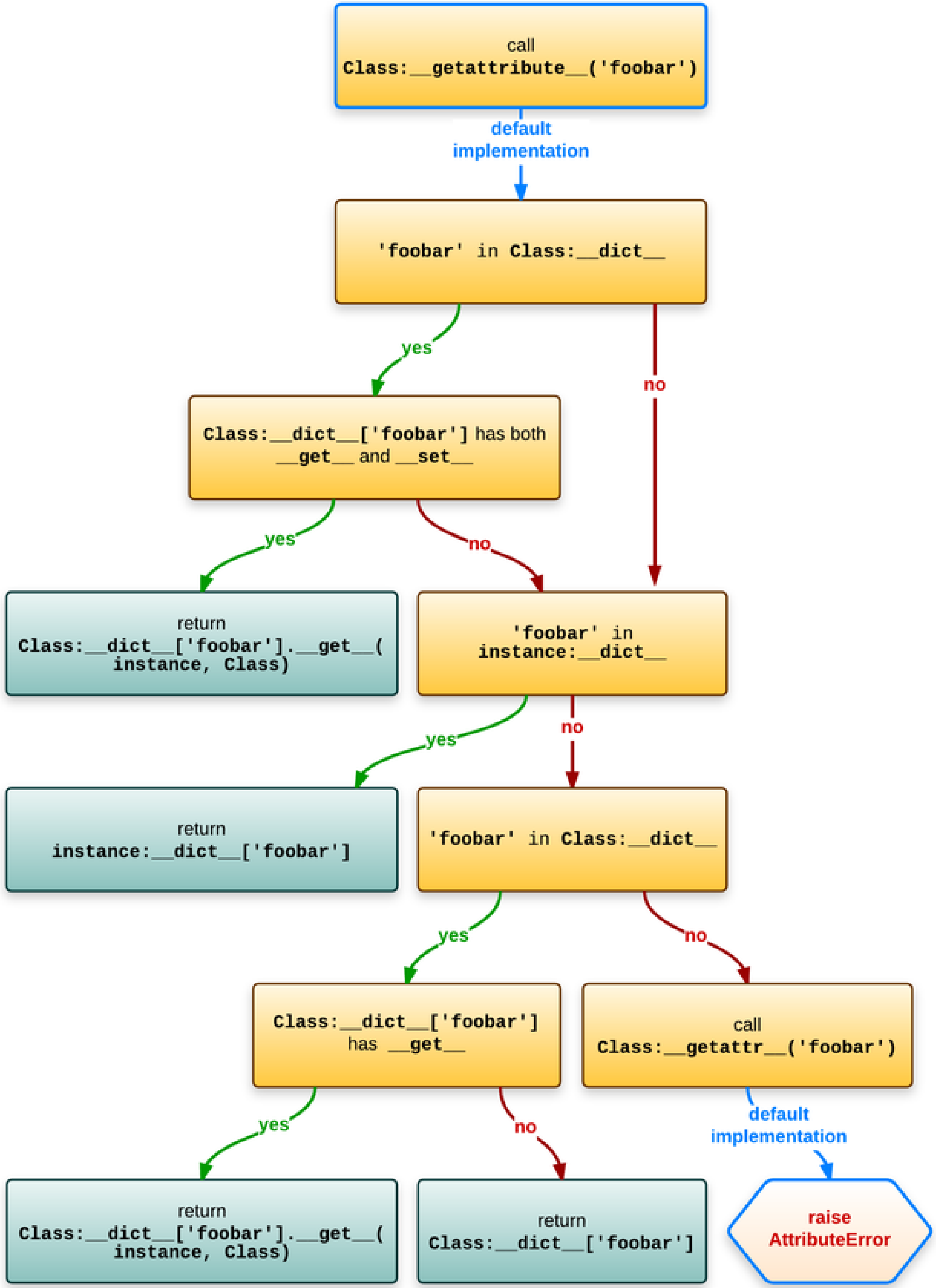
```
obj.x
descr.__get__(self, obj, type=None) -> value

obj.x = value
descr.__set__(self, obj, value) -> None


del obj.x
descr.__delete__(self, obj) -> None
```

Non-Data Descriptors

Если у дескриптора есть только __get__, то obj.__dict__ важнее дескриптора.

Non-Data Descriptors

```python
class cached_property(object):
    def __init__(self, func):
        self.func = func

    def __get__(self, obj, cls):
        value = self.func(obj)
        obj.__dict__[self.func.__name__] = value
        return value
```



call
Class:__getattribute__('foobar')

default implementation

'foobar' in Class:__dict__

yes

no

Class:__dict__['foobar'] has both __get__ and __set__

yes

no

return
Class:__dict__['foobar'].__get__(instance, Class)

'foobar' in instance:__dict__

yes

no

return
instance:__dict__['foobar']

'foobar' in Class:__dict__

yes

no

Class:__dict__['foobar'] has __get__

call
Class:__getattr__('foobar')

yes

no

default implementation

return
Class:__dict__['foobar'].__get__(instance, Class)

return
Class:__dict__['foobar']

raise AttributeError

property

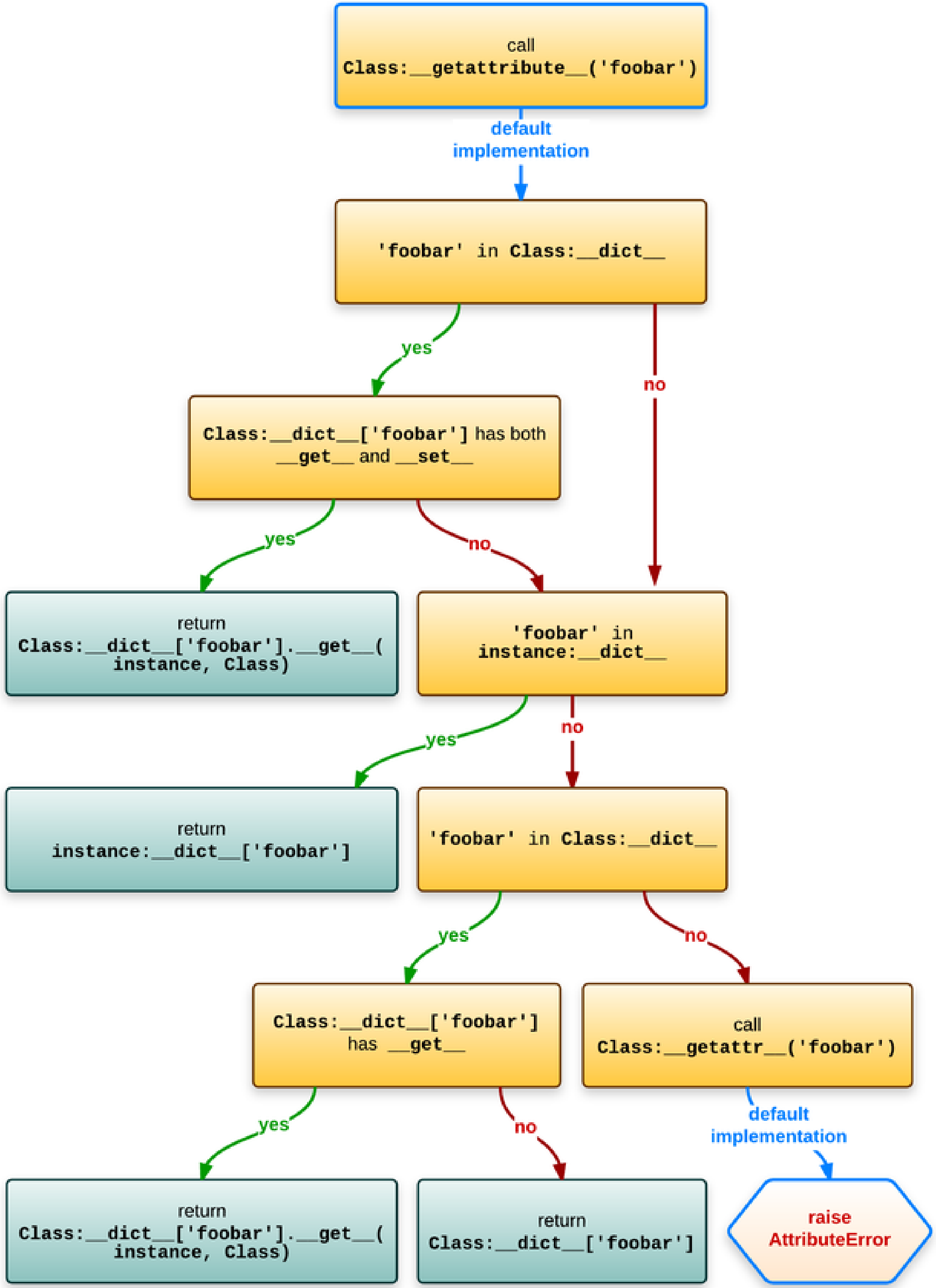
```
class A:
    @property
    def foo(self):
        return self.bar

    @foo.setter
    def foo(self, value):
        self.bar = value


a = A()
a.foo = 92
assert a.foo == 92
```

```python
class property(object):
    def __init__(self, fget=None, fset=None, fdel=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)
```

```python
class property(object):
    def __init__(self, fget=None, fset=None, fdel=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel

    ...

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel)
```

Дескриптор у Класса

```
>>> class A:
...     @property
...     def foo(self):
...         return 92
...
>>> a = A()
>>> a.foo
92
>>> A.foo
<property object at 0x7f90d125e958>
>>> type(A)
<class 'type'>  # <- класс класса -- метакласс
```



call `Class:__getattribute__('foobar')`

`'foobar'` in `Class:__dict__`

`Class:__dict__['foobar']` has both `__get__` and `__set__`

return `Class:__dict__['foobar'].__get__( instance, Class)`

`'foobar'` in `instance:__dict__`

return `instance:__dict__['foobar']`

`'foobar'` in `Class:__dict__`

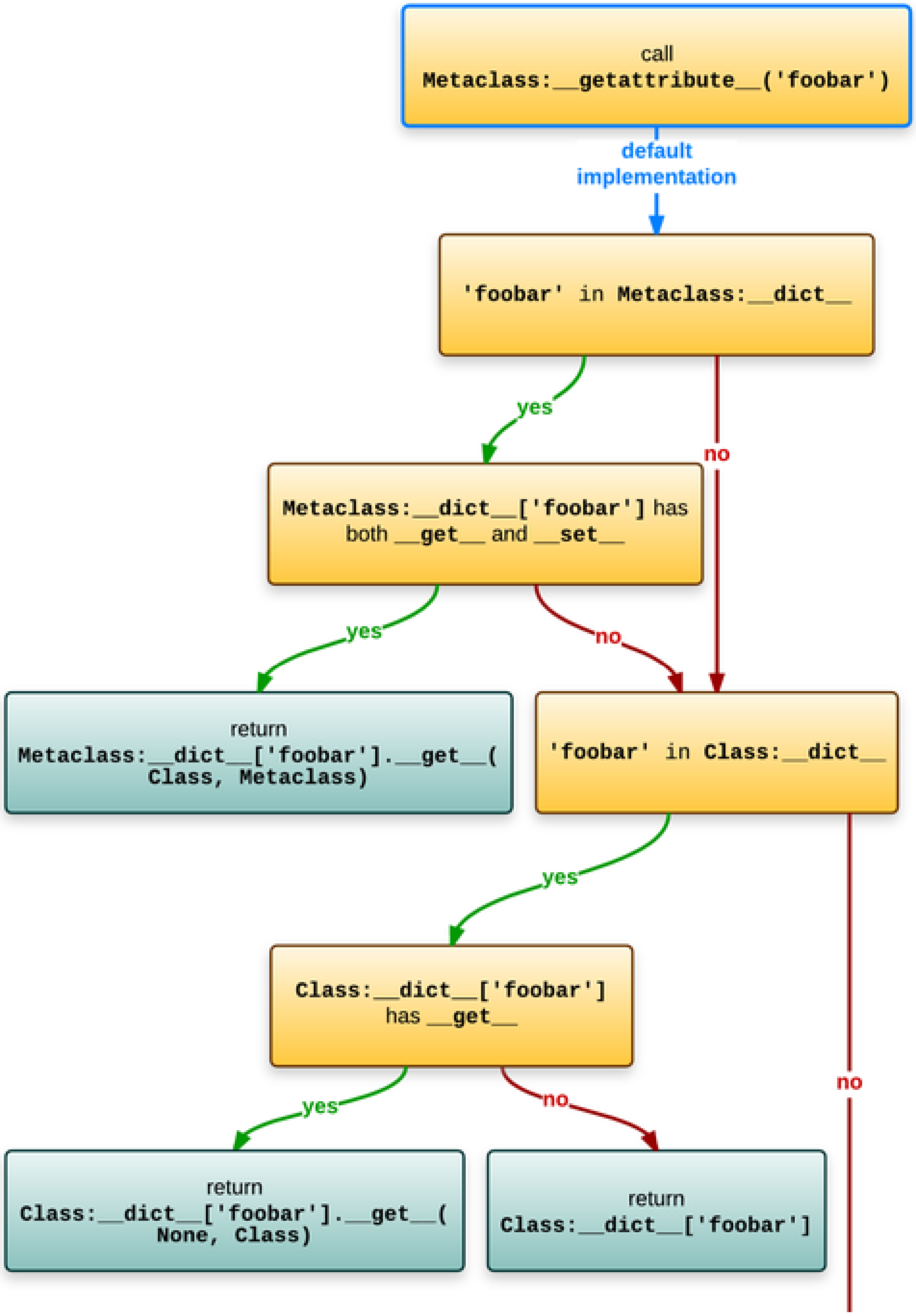
`Class:__dict__['foobar']` has `__get__`

call `Class:__getattr__('foobar')`

return `Class:__dict__['foobar'].__get__( instance, Class)`

return `Class:__dict__['foobar']`

raise AttributeError



call
`Metaclass:__getattribute__('foobar')`

`'foobar' in Metaclass:__dict__`

`Metaclass:__dict__['foobar']` has both `__get__` and `__set__`

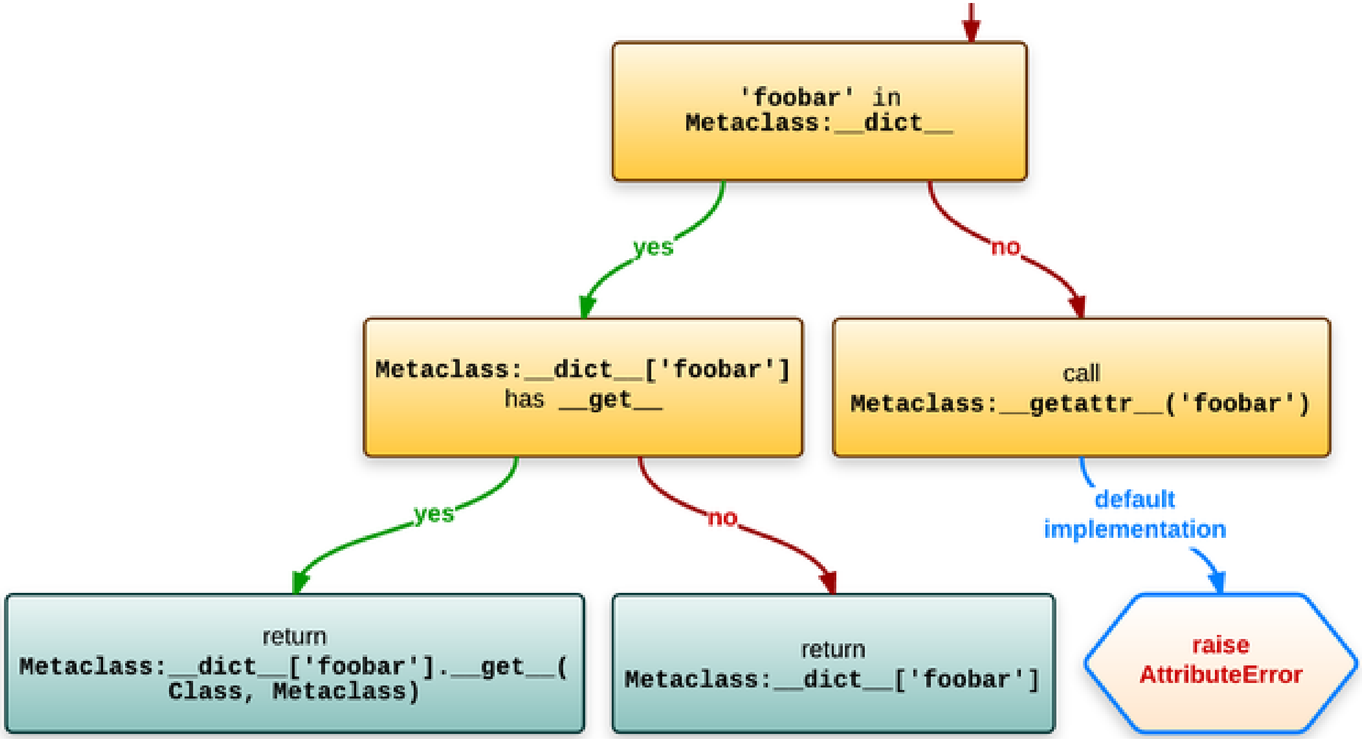
return
`Metaclass:__dict__['foobar'].__get__(Class, Metaclass)`

`'foobar' in Class:__dict__`

`Class:__dict__['foobar']` has `__get__`

return
`Class:__dict__['foobar'].__get__(None, Class)`

return
`Class:__dict__['foobar']`


'foobar' in
Metaclass:__dict__
yes
no
Metaclass:__dict__['foobar']
has __get__
call
Metaclass:__getattr__('foobar')
yes
no
default
implementation
return
Metaclass:__dict__['foobar'].__get__(
Class, Metaclass)
return
Metaclass:__dict__['foobar']
raise
AttributeError

staticmethod

```
class staticmethod(object):
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f


class A:
    @staticmethod
    def foo():
        print("no self")
```

classmethod

```
class classmethod(object):
    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc

class A:
    @classmethod
    def foo(cls):
        print(cls)

class B(A):
    pass

B.foo()  # <class '__main__.B'>
```

magic __call__

```python
class A:
    def __call__(self, *args, **kwargs):
        print("called:", args, kwargs)

a = A()
a(92)  # called: (92,) {}
```

magic __call__

```
a = A()
```

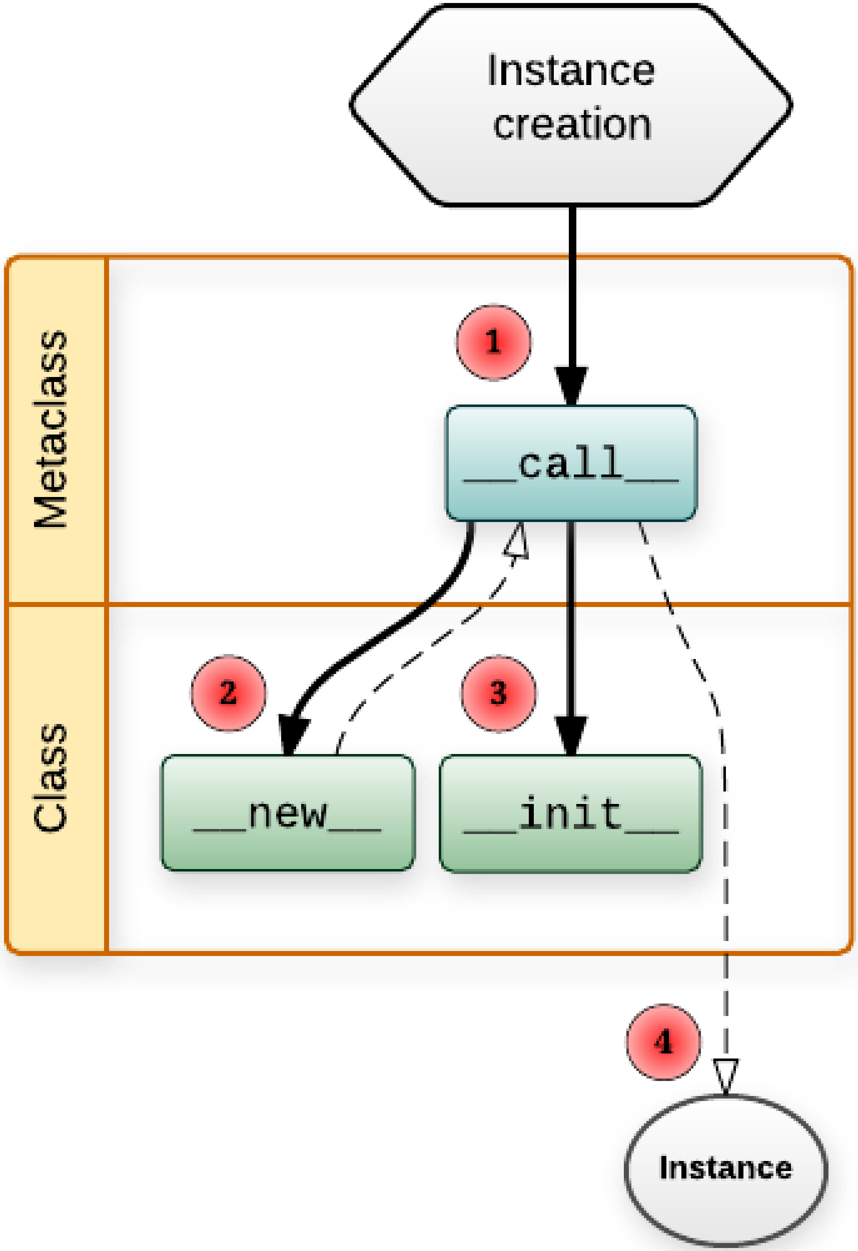
magic __call__

```
a = A()
```

A - callable, потому что type(A) определяет __call__

Конструктор

```
class type:
    def __call__(self, *args, **kwargs):
        # static method
        obj = self.__new__(self, *args, **kwargs)
        if isinstance(obj, self):
            obj.__init__(*args, **kwargs)
        return obj
```


Instance creation
Metaclass
Class
1
__call__
2
__new__
3
__init__
4
Instance

Синглтон

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        print("args, kwargs", args, kwargs)
        if Singleton._instance is None:
            Singleton._instance = super().__new__(cls)
        return Singleton._instance

    def __init__(self, value):
        self.value = value


a = Singleton(92)
b = Singleton(62)
assert a is b
assert a.value == 62
```

Метакласс

Можно ли переопределить __call__ у класса?

Тривиальный Метакласс

```
class Meta(type):
    pass


class A(metaclass=Meta):
    pass

assert type(A) is Meta
```

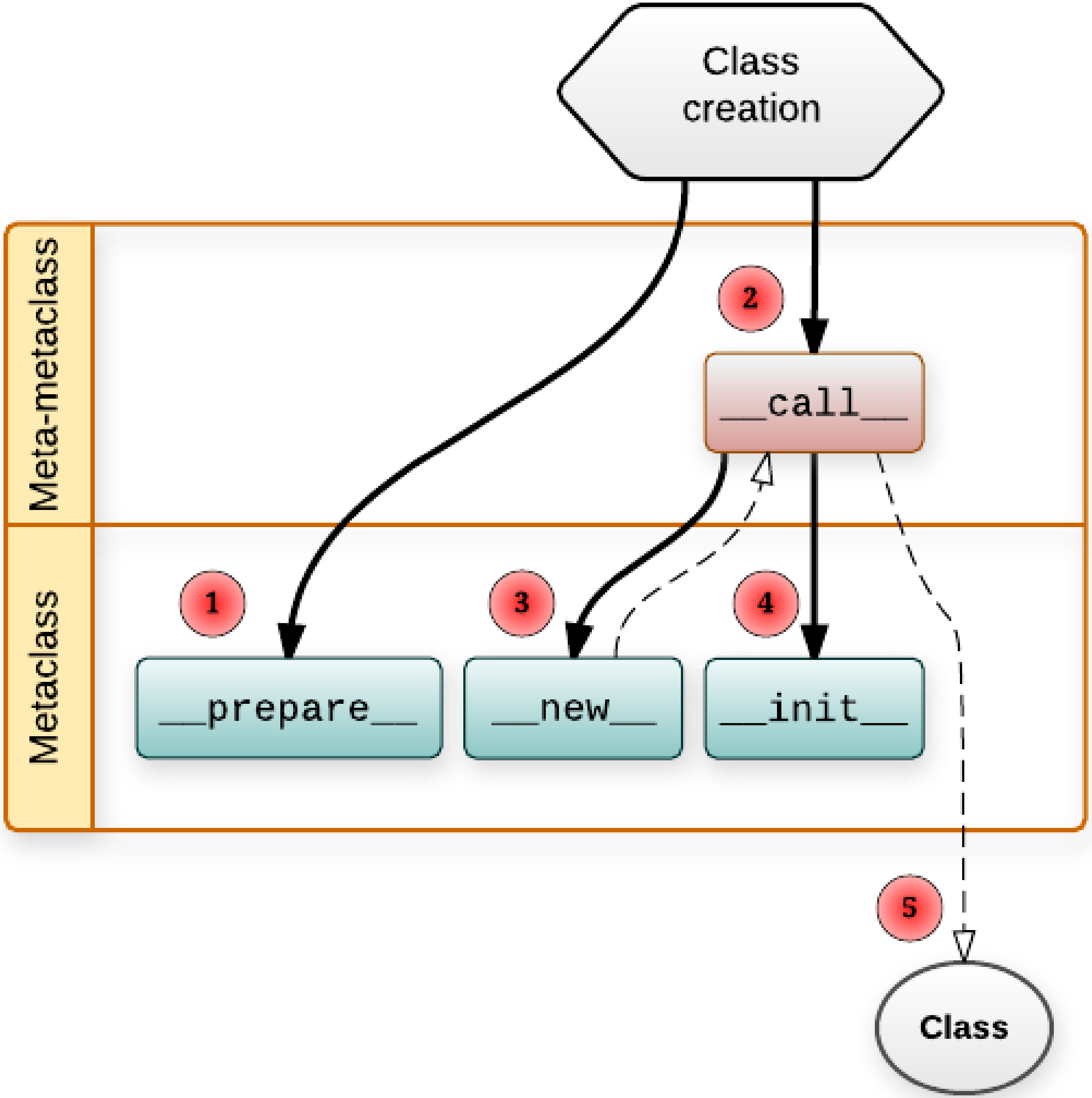
Метакласс

```python
class Foo:
    pass


class Meta(type):
    def __call__(self, *args, **kwargs):
        return Foo()


class A(metaclass=Meta):
    pass

a = A()
assert isinstance(a, Foo)
```

Метакласс

Можно ли настроить процедуру создания класса?

```python
class Base: pass

class A(
    Base,
    foo=92,
    metaclass=lambda *args, **kwargs: print(args, kwargs),
):
    def foo(self): pass


"""
(
    'A',  # name
    (<class '__main__.Base'>,),  # bases
    # cls dict
    {'__module__': '__main__',
     '__qualname__': 'A',
     'foo': <function A.foo at 0x7fb0869f5950>}
)

# kwargs
{'foo': 92}
"""
```


Class creation
Meta-metaclass
Metaclass
2
__call__
1
3
4
__prepare__
__new__
__init__
5
Class

Типичный Метакласс

```python
class Meta(type):
    @classmethod
    def __prepare__(mcs, name, bases, **kwargs):
        # Типичный пример для Python <= 3.7
        return OrderedDict()

    def __new__(mcs, name, bases, attrs, **kwargs):
        # Тут можно сделать что-нибудь интересное
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        # Не интересно: +/- декоратор
        super().__init__(name, bases, attrs)
```

Имя Дескриптора

```python
class IntField:
    def __get__(self, instance, owner):
        return instance.__dict__['x?']

    def __set__(self, instance, value):
        assert isinstance(value, int)
        instance.__dict__['x?'] = value


class A:
    x = IntField()
```

```
class IntField:
    def __set_name__(self, name):
        self._name = name

    def __get__(self, instance, owner):
        return instance.__dict__[self._name]

    def __set__(self, instance, value):
        assert isinstance(value, int)
        instance.__dict__[self._name] = value


class FieldMeta(type):

    def __new__(mcs, name, bases, attrs, **kwargs):
        for k, v in attrs.items():
            if isinstance(v, IntField):
                v.__set_name__(k)
        return super().__new__(mcs, name, bases, attrs)
```

__set_name__ (>= 3.6)

```
class IntField:
    def __set_name__(self, owner, name):
        self.name = name
```

type.__new__ автоматически вызывает __set_name__ у дескрипторов

Выбор Метакласса

- Метакласс может быть только один
- Метаклассы наследуются
- Реальный метакласс -- most derived, или ошибка

Metaclass Conflict

```
T = TypeVar("T")


class Factor(NamedTuple, Generic[T]):
    elements: List[int]
    levels: Mapping[T, int]


def factor(xs: List[T]) -> Factor[T]:
    pass

# TypeError: metaclass conflict:
#   the metaclass of a derived class must be
#   a (non-strict) subclass of the metaclasses
#   of all its bases
```

__init_subclass__ (>= 3.6)

```
class CodeStyleChecker:
    # автоматически classmethod
    def __init_subclass__(cls, ignore_case=None, **kwargs):
        ignore_case = ignore_case or []
        super().__init_subclass__(**kwargs)

        for name in dir(cls):
            if name in ignore_case:
                continue
            assert name == name.lower(), f"bad name: {name}"


class JavaRocks(CodeStyleChecker, ignore_case=["toString"]):
    def toString(self):
        print("JavaRocks")
```

Полезные метаклассы

```
class NamedTupleMeta(type):  # В сокращении

    def __new__(cls, typename, bases, ns):
        types = ns.get('__annotations__', {})
        nm_tpl = collections.namedtuple(
            typename,
            [n for n in types]
        )
        return nm_tpl
```

Полезные метаклассы

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Полезные метаклассы

```
import abc

class Iterable(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def __iter__(self):
        raise NotImplementedError


class Something(Iterable):
    pass

Something()

# TypeError:
#   Can't instantiate abstract class Something
#   with abstract methods __iter__
```

Полезные метаклассы

```python
from collections.abc import Iterable


class Empty:
    def __iter__(self):
        return iter([])


assert isinstance(Empty(), Iterable)
# what the duck?
```

```python
from abc import ABC

# issubclass => __subclasscheck__
# isinstance => __instancecheck__


class MyIterable(ABC):

    @abstractmethod
    def __iter__(self):
        while False:
            yield None


    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyIterable:
            if any("__iter__" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented
```

```python
from collections import deque


class MemorizingDict(dict):
    """A dict which remembers
       a fixed number of last-modified keys."""

    def __init__(self, *args, **kwargs):
        self.history = deque(maxlen=10)
        super().__init__(*args, **kwargs)

    def __setitem__(self, key, value):
        self.history.append(key)
        super().__setitem__(key, value)


d = MemorizingDict({"foo": 42})
d.setdefault("bar", 24)
d["baz"] = 100500
assert len(d.history) == 2  # :( AssertionError
from collections.abc import MutableMapping


class MemorizingDict(MutableMapping):
    def __init__(self, data, **kwargs):
        self.data = dict(data, **kwargs)
        self.history = deque(maxlen=10)

    def __getitem__(self, key):         # ≡ self[key]
        pass

    def __setitem__(self, key, value):  # ≡ self[key] = value
        pass

    def __iter__(self):                 # ≡ iter(self)
        pass

    def __len__(self):                  # ≡ len(self)
        pass
```

```python
from collections import UserDict


class MemorizingDict(UserDict):
    def __init__(self, data=None, **kwargs):
        self.history = deque(maxlen=10)
        super().__init__(data, **kwargs)

    def __setitem__(self, key, value):
        self.history.append(key)
        super().__setitem__(key, value)

    def get_history(self):
        return self.history
```

Что читать в транспорте

- https://docs.python.org/3.7/howto/descriptor.html
- https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/
- https://vorpus.org/blog/timeouts-and-cancellation-for-humans/
- https://vorpus.org/blog/notes-on-structured-concurrency-or-go-statement-considered-harmful/