MIPT
MOSCOW INSTITUTE
OF PHYSICS AND TECHNOLOGY

АТП кафедра Алгоритмов
и Технологий
Программирования

Дескрипторы и мета классы

Емельянов А. А.
login-const@mail.ru

2017

https://github.com/king-menin/python-course

Дескрипторы

Свойства, мотивация

```
class Limited(object):

    def __init__(self, lo, hi, label):
        self.lo = lo
        self.hi = hi
        self.label = label
        self.__val = None

    def get_val(self):
        """Это свойство val."""
        return self.__val

    def set_val(self, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self.__val = value

    def del_val(self):
        self.__val = 'No more'

pass

sv = Limited(1, 1000, "val")
try:
    sv.set_val(2000)
except ValueError as e:
    print(e)

sv.set_val(500)
print(sv.get_val())
sv.del_val()
print(sv.get_val())
```

```
Please, set val with conditions: 1 < val < 1000
500
No more
```

Функция property, «вычисляемое» свойство.

```
class Limited(object):

    def __init__(self, lo, hi, label):
        self.lo = lo
        self.hi = hi
        self.label = label
        self.__val = None

    def get_val(self):
        return self.__val

    def set_val(self, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self.__val = value

    def del_val(self):
        self.__val = 'No more'

    val = property(get_val, set_val, del_val, "Это свойство val.")

pass

sv = Limited(1, 1000, "x")
try:
    sv.val = 2000
except ValueError as e:
    print(e)

sv.val = 500
print(sv.val)
del(sv.val)
print(sv.val)
```

```
Please, set x with conditions: 1 < x < 1000
500
No more
```

Функция property, «вычисляемое» свойство.

- Позволяет использовать методы в качестве свойств объектов — порождает *дескриптор*, позволяющий создавать «вычисляемые» свойства (тип property):

property([fget[, fset[, fdel[, doc]]]]) -> property

- fget : Функция, реализующая возврат значения свойства.
- fset : Функция, реализующая установку значения свойства.
- fdel : Функция, реализующая удаление значения свойства.
- doc : Строка документации для создаваемого свойства. 2.5 Если не задано , будет использовано описание от fget (если оно существует).

Декоратор @property

```
class Limited(object):

    def __init__(self, lo, hi, label):
        self.lo = lo
        self.hi = hi
        self.label = label
        self.__val = None

    @property
    def val(self):
        "Это свойство val."
        return self.__val

    @val.setter
    def val(self, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self.__val = value

    @val.deleter
    def val(self):
        self.__val = 'No more'

pass

sv = Limited(1, 1000, "x")
try:
    sv.val = 2000
except ValueError as e:
    print(e)

sv.val = 500
print(sv.val)
del(sv.val)
print(sv.val)
```

```
Please, set x with conditions: 1 < x < 1000
500
No more
```

Типичный пример использования свойств

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

    @property
    def fullname(self):
        return '{name} {surname}'.format(**self.__dict__)

p = Person('John', 'Doe')
p.fullname
```

```
'John Doe'
```

Дескрипторы, мотивация

```
class Limited(object):

    def __init__(self, lo, hi, label):
        self.lo = lo
        self.hi = hi
        self.label = label
        self.__val = None

    @property
    def val(self):
        "Это свойство val."
        return self.__val

    @val.setter
    def val(self, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self.__val = value

    @val.deleter
    def val(self):
        self.__val = 'No more'

    @property
    def val2(self):
        "Это свойство val2."
        return self.__val2

    @val.setter
    def val(self, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self.__val2 = value

    @val2.deleter
    def val2(self):
        self.__val2 = 'No more'
```

Что-то тут нечисто

```
class Limited(object):

    def __init__(self, lo, hi, label):
        self.lo = lo
        self.hi = hi
        self.label = label
        self.__val = None

    @property
    def val(self):
        "Это свойство val."
        return self.__val

    @val.setter
    def val(self, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self.__val = value

    @val.deleter
    def val(self):
        self.__val = 'No more'

pass

x = Limited(1, 1000, "x")
x.val = 1e2
print(x.val, type(x.val), Limited.val, type(Limited.val))
```

```
100.0 <class 'float'> <property object at 0x00000048ABB17098> <class 'property'>
```

Определение дескриптора

- Дескриптор — это атрибут объекта со связанным поведением (*англ.* binding behavior), т.е. такой, чьё поведение при доступе переопределяется методами протокола дескриптора. Эти методы:
  - __get__
  - __set__
  - __delete__
- Если хотя бы один из этих методов определён для объекта, то он становится дескриптором.

```
class Limited(object):

    def __init__(self, lo, hi):
        pass

    def __get__(self, instance, owner):
        pass

    def __set__(self, instance, value):
        pass

    def __delete__(self, instance):
        pass

pass
```

```
class Summator(object):
    x = Limited(1, 1000)
    y = Limited(0, 1)

summator = Summator()
summator.x = 1e2
summator.x
```

Небольшое исследование

```
class Trace(object):
    def __set__(self, *args):
        print('__set__', args, sep="\n")
    def __get__(self, *args):
        print(self, '__get__', args, sep="\n")
    def __delete__(self, *args):
        print('__delete__', args, sep="\n")

class Test(object):
    attr = Trace()

t = Test()
t.attr = 5
print(t.attr)
del t.attr
```

```
__set__
(<__main__.Test object at 0x00000048ABB11B70>, 5)
<__main__.Trace object at 0x00000048ABB11AC8>
__get__
(<__main__.Test object at 0x00000048ABB11B70>, <class '__main__.Test'>)
None
__delete__
(<__main__.Test object at 0x00000048ABB11B70>,)
```

Небольшое исследование

- Дескриптор определяется только в классе, а не в объекте.

```
class Trace(object):
    def __set__(self, *args):
        print('__set__', args, sep="\n")
    def __get__(self, *args):
        print(self, '__get__', args, sep="\n")
    def __delete__(self, *args):
        print('__delete__', args, sep="\n")

class Test(object):
    attr = Trace()

t = Test()
t.attr = 5
print(t.attr)
del t.attr
```

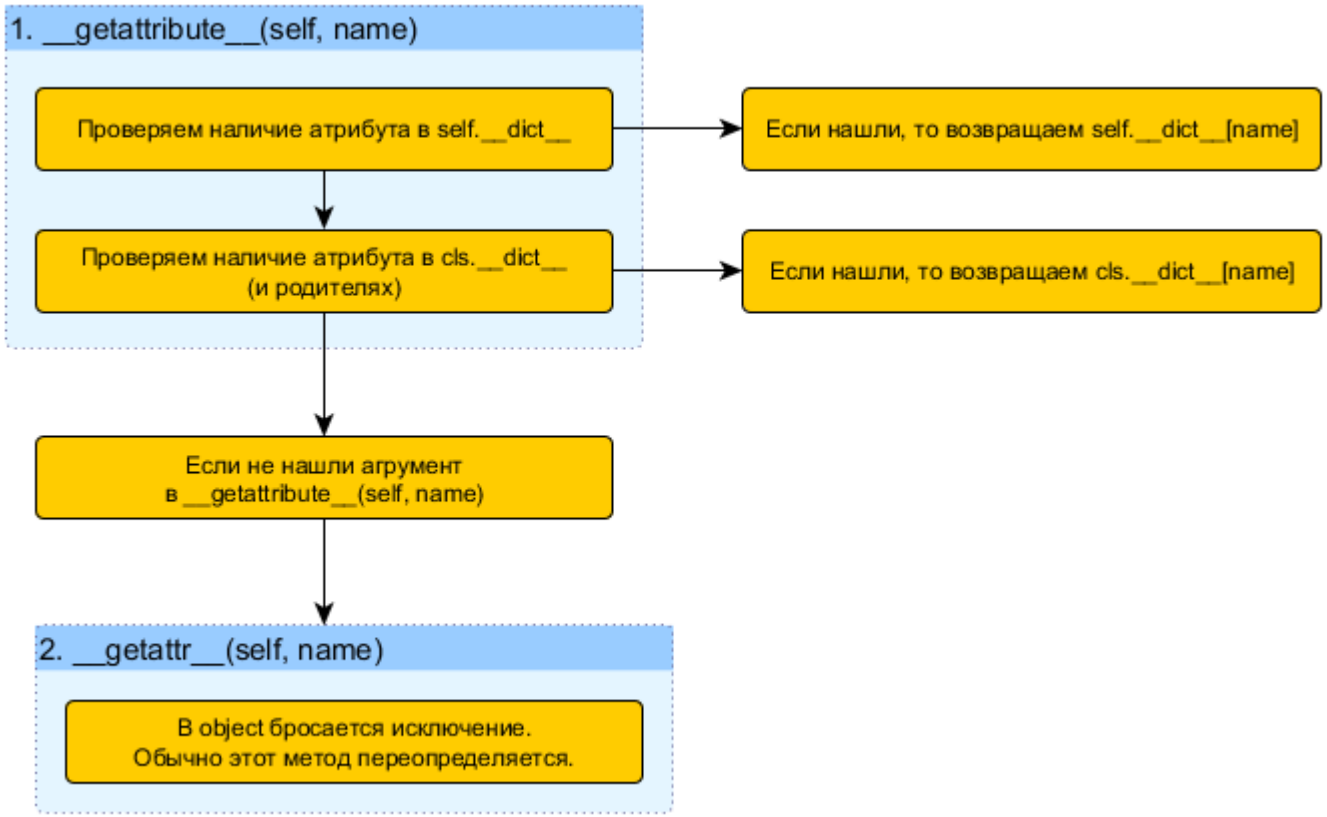
```
__set__
(<__main__.Test object at 0x00000048ABB11B70>, 5)
<__main__.Trace object at 0x00000048ABB11AC8>
__get__
(<__main__.Test object at 0x00000048ABB11B70>, <class '__main__.Test'>)
None
__delete__
(<__main__.Test object at 0x00000048ABB11B70>,)
```

VS

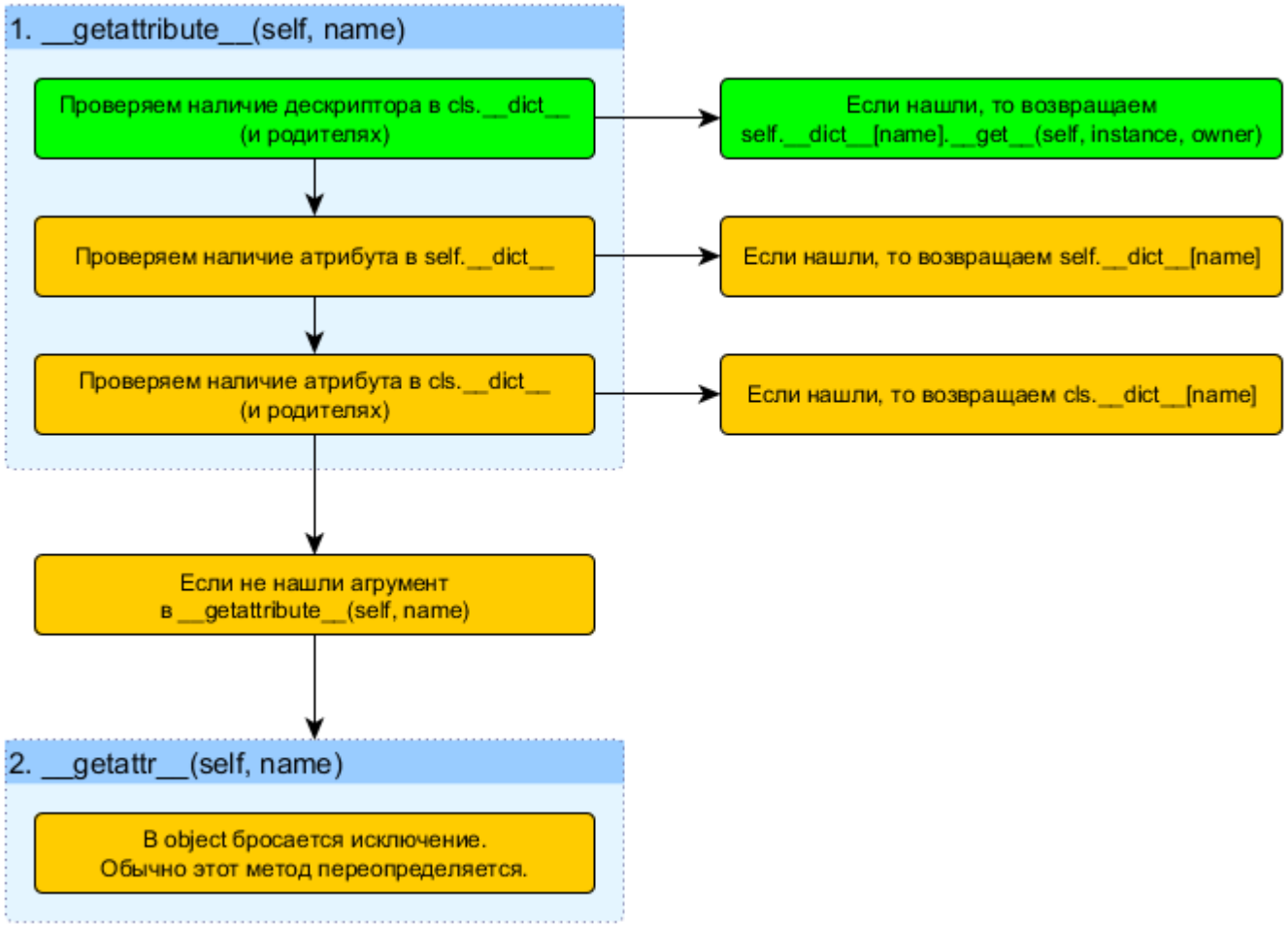
```
class Trace(object):

    def __set__(self, *args):
        print('__set__', args)

    def __get__(self, *args):
        print(self, '__get__', args)

    def __delete__(self, *args):
        print('__delete__', args)
pass


class Test(object):
    def __init__(self):
        self.attr = Trace()

pass


t = Test()
print(t.attr)
```

```
<__main__.Trace object at 0x00000048AF725320>
```

Алгоритм получения атрибута



Алгоритм получения атрибута. Дескрипторы




НУ...
ВЫ МЕНЯ ПОНЯЛИ

Костыль дескриптора 1

```
class Limited(object):

    def __init__(self, lo, hi, label):
        self.lo = lo
        self.hi = hi
        self.label = label

    def __get__(self, instance, owner):
        return instance.__dict__.get(self.label)

    def __set__(self, instance, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        instance.__dict__[self.label] = value

    def __delete__(self, instance):
        instance.__dict__[self.label] = 'No more'

pass

class Summator(object):
    x = Limited(1, 1000, "y")

summator, summator2 = Summator(), Summator()
print(summator.__dict__, summator2.__dict__)
summator.y, summator2.y = 1e2, 5
summator.y, summator2.y
```

```
{} {}
```

```
(100.0, 5)
```

```
summator.__dict__, summator2.__dict__
```

```
({'y': 100.0}, {'y': 5})
```

Плюсы и минусы костыля 1

- **Плюсы**
  - Общий *изменяемый* атрибут с прописанной логикой доступа для всех объектов класса.

- **Минусы**
  - Автоматически добавляет в словарь объекта поле, что может привести к коллизиям.

```
summator.y = 4
summator.x
```

```
4
```

Костыль дескриптора 2

```
from collections import defaultdict


class Limited(object):

    def __init__(self, lo, hi):
        self.lo = lo
        self.hi = hi
        self._values = defaultdict()

    def __get__(self, instance, owner):
        return self._values.get(instance)

    def __set__(self, instance, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self._values[instance] = value

    def __delete__(self, instance):
        self._values[instance] = 'No more'

pass

class Summator(object):
    x = Limited(1, 1000)
    y = Limited(0, 1)

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

summator, summator2 = Summator("summator"), Summator("summator2")
summator.x, summator2.x = 1e2, 5
print(summator.__dict__, summator2.__dict__)
summator.x, summator2.x
```

```
{'name': 'summator'} {'name': 'summator2'}
```

```
(100.0, 5)
```

Плюсы и минусы костыля 2

- **Плюсы**
  - Общий *изменяемый* атрибут с прописанной логикой доступа для всех объектов класса.

- **Минусы**
  - Храним данные объекта в поле класса. Ссылки на значения дескрипторов будут оставаться в словаре, даже после удаления объекта.

```
del(summator)
```

```
list(((k.name, v) for k, v in Summator.__dict__["x"]._values.items()))
```

```
[('summator', 100.0), ('summator2', 5)]
```

Костыль дескриптора 3

```
from weakref import WeakKeyDictionary


class Limited(object):

    def __init__(self, lo, hi):
        self.lo = lo
        self.hi = hi
        self._values = WeakKeyDictionary()

    def __get__(self, instance, owner):
        return self._values.get(instance)

    def __set__(self, instance, value):
        if not (self.lo < value < self.hi):
            raise ValueError("Please, set {label} with conditions: {lo} < {label} < {hi}".format(**self.__dict__))
        self._values[instance] = value

    def __delete__(self, instance):
        self._values[instance] = 'No more'

pass

class Summator(object):
    x = Limited(1, 1000)
    y = Limited(0, 1)

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

summator, summator2 = Summator("summator"), Summator("summator2")
summator.x, summator2.x = 1e2, 5
print(summator.__dict__, summator2.__dict__)
summator.x, summator2.x
```

```
{'name': 'summator'} {'name': 'summator2'}

(100.0, 5)
```

Плюсы и минусы костыля 3

- **Плюсы**
  - Общий *изменяемый* атрибут с прописанной логикой доступа для всех объектов класса.

- **Минусы**
  - Храним данные объекта в поле класса.

Как лучше создавать дескриптор

- Именованный label («Костыль дескриптора 1»).
- Хранить значения в WeakKeyDictionary («Костыль дескриптора 3»).

pikabu.ru

Меняем состояние дескриптора

```
class CallbackProperty(object):
    def __init__(self, default=None):
        self.data = WeakKeyDictionary()
        self.default = default
        self.callbacks = WeakKeyDictionary()

    def __get__(self, instance, owner):
        if instance is None:
            return self
        return self.data.get(instance, self.default)

    def __set__(self, instance, value):
        for callback in self.callbacks.get(instance, []):
            callback(value)
        self.data[instance] = value

    def add_callback(self, instance, callback):
        if instance not in self.callbacks:
            self.callbacks[instance] = []
        self.callbacks[instance].append(callback)

class BankAccount(object):
    balance = CallbackProperty(0)

def low_balance_warning(value):
    if value < 100:
        print("You are now poor")

ba = BankAccount()
BankAccount.balance.add_callback(ba, low_balance_warning)

ba.balance = 5000
print("Balance is %s" % ba.balance)
ba.balance = 99
print("Balance is %s" % ba.balance)
```

```
Balance is 5000
You are now poor
Balance is 99
```

Как работает property?

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    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)

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

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

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

Мета классы

«99% людей не знают, что такое мета классы.
Если вы **сомневаетесь**, нужны они вам или нет. **Они вам не нужны.**»
© Guido van Rossum

Что такое тип (type) в python?

```
class Cls(object):
    pass

type(int), type(Cls), type(type)
```

```
(type, type, type)
```

- Базовый тип. Прародитель других встроенных и пользовательских типов данных. Конструктор для динамических пользовательских типов.

- Использование: type(obj) / type(name, bases, dict)

- Один аргумент:
  obj : Объект, тип которого требуется определить.

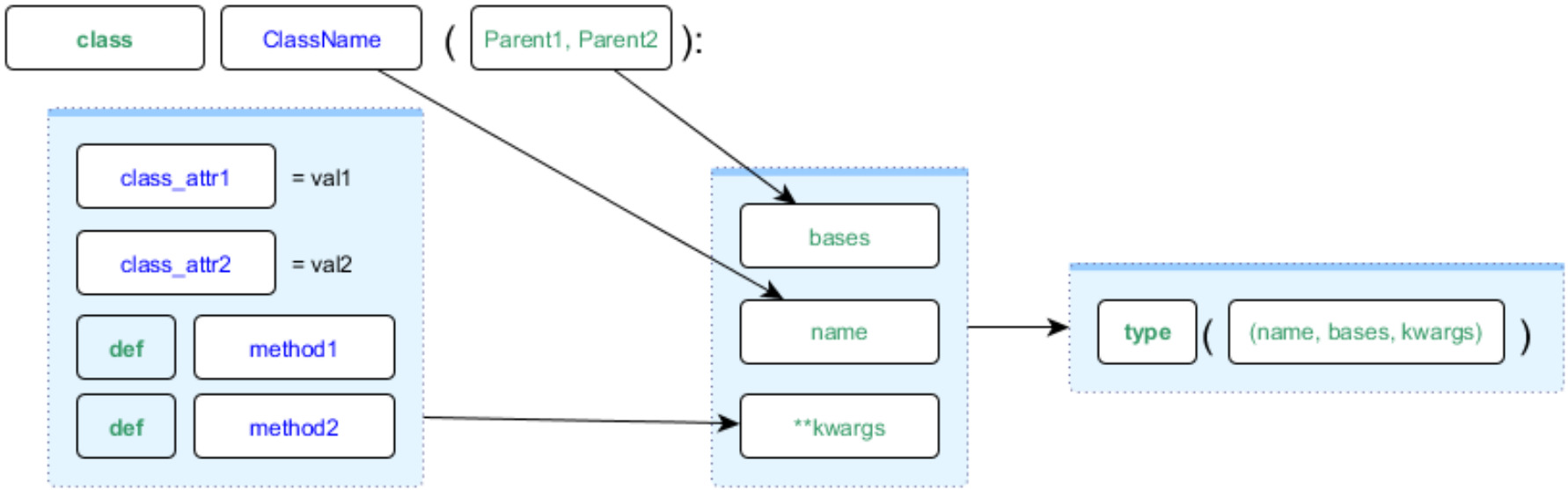
- Три аргумента:
  - name: Имя для создаваемого типа (становится атрибутом __name__);
  - bases: Кортеж с родительскими классами (становится атрибутом __bases__);
  - dict: Словарь, который будет являться пространством имён для тела класса (становится атрибутом __dict__).

Как создается класс?

```
def class_factory(name, bases, **kwargs):
    return type(name, bases, kwargs)

F = class_factory('DeepThought', (object, ), ans=42)
f = F()
print(f.ans, f, type(f), type(F))
```

```
42 <__main__.DeepThought object at 0x00000048ABBCAF60> <class '__main__.DeepThought'> <class 'type'>
```

```
class DeepThought(object):
    ans = 42

pass
```



Мета класс

- Мета класс – это класс, объектами которого являются другие классы.

- По умолчанию в python есть один мета класс type. Он реализует всю логику, которая конструирует классы.

- Наследуясь от type, можно переопределять логику того, как эти классы создаются.

```
class Meta(type):
    pass

def class_factory(name, bases, **kwargs):
    return Meta(name, bases, kwargs)

F = class_factory('DeepThought', (object, ), ans=42)
f = F()
print(f.ans, f, type(f), type(F))
```

```
42 <__main__.DeepThought object at 0x00000048ABBB1128> <class '__main__.DeepThought'> <class '__main__.Meta'>
```

Изменяем поведение мета класса

```
class Meta(type):
    def __init__(cls, name, base, attrs):
        print("__init__ Meta")
        super().__init__(name, base, attrs)
        cls.hola = lambda self: 'qwerty'

def class_factory(name, bases, **kwargs):
    return Meta(name, bases, kwargs)

F = class_factory('DeepThought', (object, ), ans=42)
```

```
__init__ Meta
```

```
f = F()
print(f.ans, f, type(f), type(F))
print(f.hola())
```

```
42 <__main__.DeepThought object at 0x00000048ABBCAB00> <class '__main__.DeepThought'> <class '__main__.Meta'>
qwerty
```

Мета классы, «полная картина»

```
class Meta(type):
    def __new__(mcs, name, bases, attrs, **kwargs):
        print('Meta.__new__(mcs=%s, name=%r, bases=%s, attrs=[%s], **%s)' % (
             mcs, name, bases, ', '.join(attrs), kwargs
         ))
        return super().__new__(mcs, name, bases, attrs)

    def __init__(cls, name, bases, attrs, **kwargs):
        print('Meta.__init__(cls=%s, name=%r, bases=%s, attrs=[%s], **%s)' % (
             cls, name, bases, ', '.join(attrs), kwargs
         ))
        return super().__init__(name, bases, attrs)

    def __call__(cls, *args, **kwargs):
        print('Meta.__call__(cls=%s, args=%s, kwargs=%s)' % (
             cls, args, kwargs
         ))
        return super().__call__(*args, **kwargs)


def class_factory(name, bases, **kwargs):
    return Meta(name, bases, kwargs)

F = class_factory('DeepThought', (object, ), ans=42)
```

```
Meta.__new__(mcs=<class '__main__.Meta'>, name='DeepThought', bases=(<class 'object'>,), attrs=[ans], **{})
Meta.__init__(cls=<class '__main__.DeepThought'>, name='DeepThought', bases=(<class 'object'>,), attrs=[ans], **{})
```

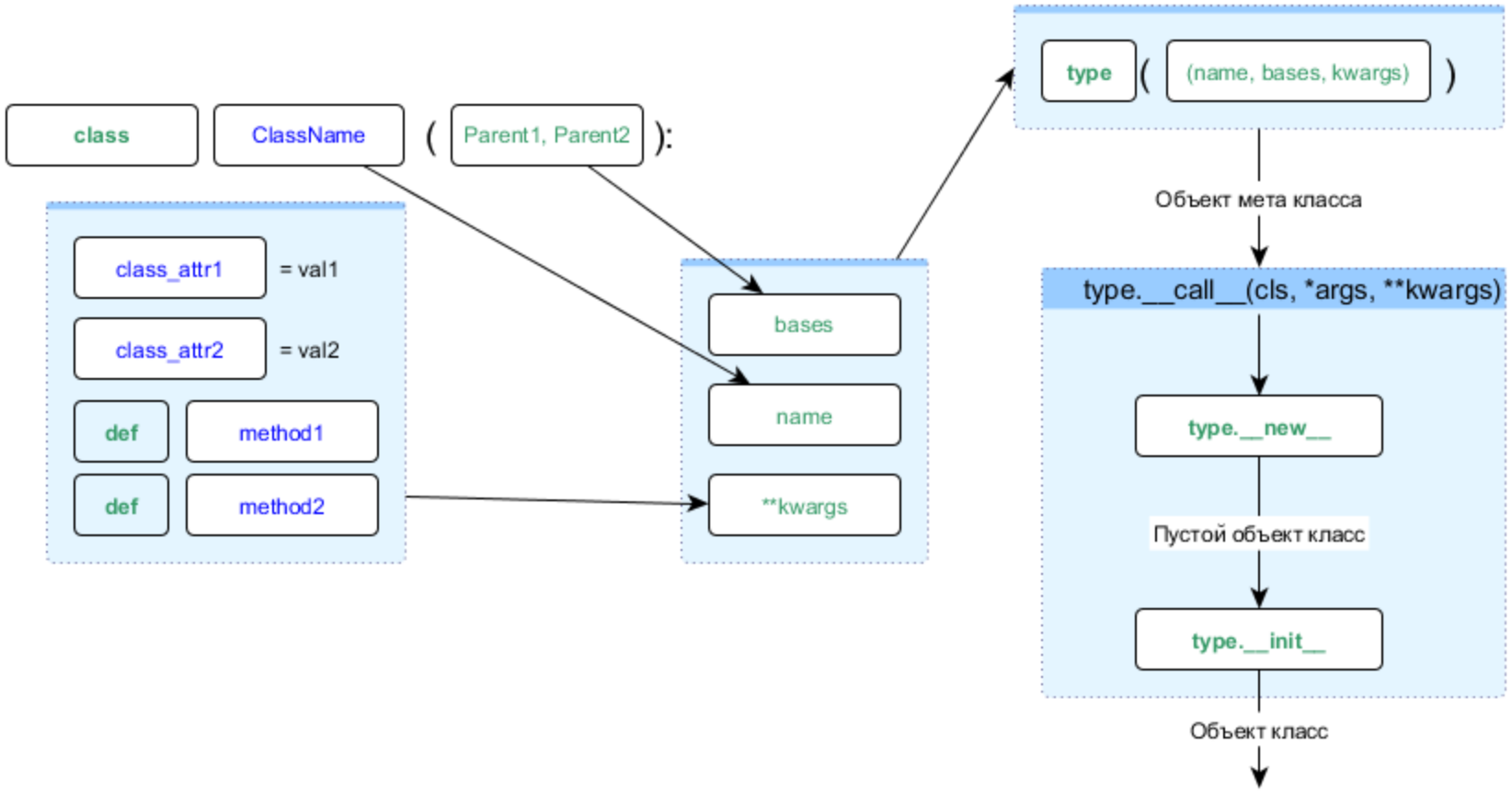
```
f = F()
```

```
Meta.__call__(cls=<class '__main__.DeepThought'>, args=(), kwargs={})
```

```
print(f.ans, f, type(f), type(F))
```

```
42 <__main__.DeepThought object at 0x00000048ABBECAC8> <class '__main__.DeepThought'> <class '__main__.Meta'>
```

Создание класса



Способ задания мета класса без функции

```
class NewMeta(type):
    def __init__(cls, name, base, attrs):
        super().__init__(name, base, attrs)
        cls.f = lambda self: 'qwerty'

class Test(object, metaclass=NewMeta):
    pass
Test
```

```
__main__.Test
```

То же самое, что:

```
class NewMeta(type):
    def __init__(cls, name, base, attrs):
        super().__init__(name, base, attrs)
        cls.f = lambda self: 'qwerty'


def class_factory(name, bases, **kwargs):
    return NewMeta(name, bases, kwargs)

Test = class_factory("Test", (object, ))
Test
```

```
__main__.Test
```

Методы мета класса и методы объекта

```
class NewMeta(type):
    def __init__(cls, name, base, attrs):
        super().__init__(name, base, attrs)
        cls.f = lambda self: 'qwerty'

class Test(object, metaclass=NewMeta):
    pass

t_cls = Test()
```

```
t_cls.f()
```

```
'qwerty'
```

```
Test.f()
```

```
----------------------------------------
TypeError                                 Tr
<ipython-input-367-65c09b92bd2b> in <module>
----> 1 Test.f()

TypeError: <lambda>() missing 1 required pos
```

VS

```
class NewMeta(type):
    def __init__(cls, name, base, attrs):
        super().__init__(name, base, attrs)

    def f(cls):
        return 'qwerty'

class Test(object, metaclass=NewMeta):
    pass
t_cls = Test()
```

```
t_cls.f()
```

```
----------------------------------------
AttributeError                            Tr
<ipython-input-369-1b8805f9ddce> in <module>
----> 1 t_cls.f()

AttributeError: 'Test' object has no attribu
```

```
Test.f()
```

```
'qwerty'
```

Аргумент мета класса - любой callable объект

- В качестве мета класса может выступать любой объект, даже функция, которая может и не возвращать объект класс.

```
class MyPrint(object, metaclass=print):
    pass
```

```
MyPrint (<class 'object'>,) {'__module__': '__main__', '__qualname__': 'MyPrint'}
```

```
type(MyPrint)
```

```
NoneType
```

Пример использования мета классов 1

```
def upper_attr(future_class_name, future_class_parents, future_class_attr):
    uppercase_attr = {}
    for name, val in future_class_attr.items():
        if not name.startswith('__'):
            uppercase_attr[name.upper()] = val
        else:
            uppercase_attr[name] = val
    return type(future_class_name, future_class_parents, uppercase_attr)


class Foo(metaclass=upper_attr):
    bar = 'bip'

f = Foo()
print(hasattr(Foo, 'bar'))
print(hasattr(Foo, 'BAR'))
print(f.BAR)
```

```
False
True
bip
```

Пример использования мета классов 1’

- Тоже самое классом:

```
class UpperAttrMetaclass(type):

    def __new__(cls, name, bases, dct):

        attrs = ((name, value) for name, value in dct.items() if not name.startswith('__'))
        uppercase_attr = dict((name.upper(), value) for name, value in attrs)

        return type.__new__(cls, name, bases, uppercase_attr)
```

```
class Foo(metaclass=UpperAttrMetaclass):
    bar = 'bip'

f = Foo()
print(hasattr(Foo, 'bar'))
print(hasattr(Foo, 'BAR'))
print(f.BAR)
```

```
False
True
bip
```

Пример использования мета классов 2

```
class Descriptor(object):
    def __init__(self):
        #notice we aren't setting the label here
        self.label = None

    def __get__(self, instance, owner):
        print('__get__. Label = {}'.format(self.label))
        return instance.__dict__.get(self.label, None)

    def __set__(self, instance, value):
        print('__set__')
        instance.__dict__[self.label] = value


class DescriptorOwner(type):
    def __new__(cls, name, bases, attrs):
        for attr_name, attr_value in attrs.items():
            if isinstance(attr_value, Descriptor):
                attr_value.label = attr_name
        return super().__new__(cls, name, bases, attrs)


class Foo(object, metaclass=DescriptorOwner):
    x = Descriptor()

f1, f2 = Foo(), Foo()
f1.x, f2.x = 10, 20
print(f1.x, f2.x)
```

```
__set__
__set__
__get__. Label = x
__get__. Label = x
10 20
```

Применения мета классов

- Web фреймворк django (динамическая генерация страниц с контентом)

- protobuf – описываем общим синтаксисом структуру данных. далее в компилятор protoc с помощью модуля в питоне можно послать структуру в форме строки, отправить ее по сети и получить, например класс в C.

- Фреймворки для работы с базой данных, например читать из node.js.


THIS IS
THE END