

# Углубленный Python

## Лекция 4

### Метaprogramмирование, дескрипторы, ABC

Кандауров Геннадий



education

# Напоминание отметиться на портале

+ ОСТАВИТЬ ОТЗЫВ

vk образование

БлогиЛюдиПрограммаВакансииРасписание

Q

VK

Техно

Открыт приём заявок!

чт, 8 сентября

Нет занятий

пт, 9 сентября

18:00 Углубленный Py... с3  
Введение в Python, основные  
понятия, тестирование  
Г. Кандауров

сб, 10 сентября

Нет занятий

вс, 11 сентября

Нет занятий

пн, 12 сентября

Нет занятий

Углубленный Python

↓ 0 ↑

Блог для материалов по курсу "Углубленный Python"

57 читателей, 2 топика

ПодписатьсяСоздать топик

Поиск по авторам, заголовку и тексту топика...

Найти

Добро пожаловать на курс!

Углубленный Python

ИзменитьУдалить

Всем привет и добро пожаловать на курс по углубленному изучению Python!

Прямой эфир

МоиВсе

Геннадий Кандауров час назад  
Углубленный Python → Добро пожаловать  
на курс! 0

Екатерина Черкасова 7 дней назад  
Стажировка → Приглашаем мобильных,  
фронтенд- и бэкэнд-разработчиков на  
Weekend Offer! 0

Дарья Вовченко 9 дней назад  
Углубленный Python → Добро пожаловать  
в образовательные проекты VK  
Образование! 0

Дарья Вовченко 9 дней назад  
Разработка веб-сервисов на

## Квиз про прошлой лекции



# Содержание занятия

1. Классы
2. Дескрипторы
3. Метaproгpаммиpование
4. ABC

# Классы



# Классы: магические методы

```
object.__new__(cls[, ...])
```

Статический метод, создает новый экземпляр класса.

После создание экземпляра вызывается (уже у экземпляра) метод `__init__`.

`__init__` ничего не должен возвращать (кроме `None`), иначе - `TypeError`

```
class Singleton:
```

```
    _instance = None
```

```
    def __new__(cls, *args, **kwargs):
```

```
        if cls._instance is None:
```

```
            cls._instance = super().__new__(cls, *args, **kwargs)
```

```
        return cls._instance
```

# Классы: магические методы

```
object.__set_name__(self, owner, name)
```

Автоматически вызывается при создании класса-владельца `owner`.

Хук вызывается, когда объекту было присвоено имя `name` в этом классе

```
class Attr:
    def __set_name__(self, owner, name):
        print(f"{locals()}=")
        self.name = name

class A:
    x = Attr()  # Automatically calls: x.__set_name__(A, "x")

a = A()
```

# Дескрипторы







*Дескриптор это атрибут объекта со “связанным поведением”, то есть такой атрибут, при доступе к которому его поведение переопределяется методом протокола дескриптора. Эти методы `__get__`, `__set__` и `__delete__`. Если хотя бы один из этих методов определен в объекте , то можно сказать что этот объект дескриптор.*

Раймонд Хеттингер

# Дескрипторы

- Если определен один из методов `__get__`, `__set__` и `__delete__`, объект считается дескриптором.
- Если объект дескриптора определяет `__get__`, `__set__`, то он считается data дескриптором.
- Если объект дескриптора определяет `__get__`, то является non-data дескриптором.

# Дескрипторы

```
class MyDescriptor:
    def __get__(self, obj, objtype):
        print(f"get {obj} cls={objtype}")

    def __set__(self, obj, val):
        print(f"set {val} for {obj}")

    def __delete__(self, obj):
        print(f"delete from {obj}")
```

```
class MyClass:
    field = MyDescriptor()
```

```
>>> inst = MyClass()
>>> MyClass.field
get None cls=<class '__main__.MyClass'>
>>> inst.field
get <__main__.MyClass object ...> cls=<class
'__main__.MyClass'>
>>> inst.field = 1
set 1 for <__main__.MyClass object ...>
>>> del inst.field
delete from <__main__.MyClass object ...>
```

# Дескрипторы

```
>>> class A:
    def foo(self):
        pass
    a = A()
```

```
>>> a.foo.__class__.__get__
<slot wrapper '__get__' of 'method' objects>
```

```
>>> A.__dict__['foo'] # Внутренне хранится как функция
<function foo at 0x00C45070>
```

```
>>> A.foo # Доступ через класс возвращает несвязанный метод
<unbound method A.foo>
```

```
>>> a.foo # Доступ через экземпляр объекта возвращает связанный метод
<bound method A.foo of <__main__.A object at 0x00B18C90>>
```

# Метаклассы

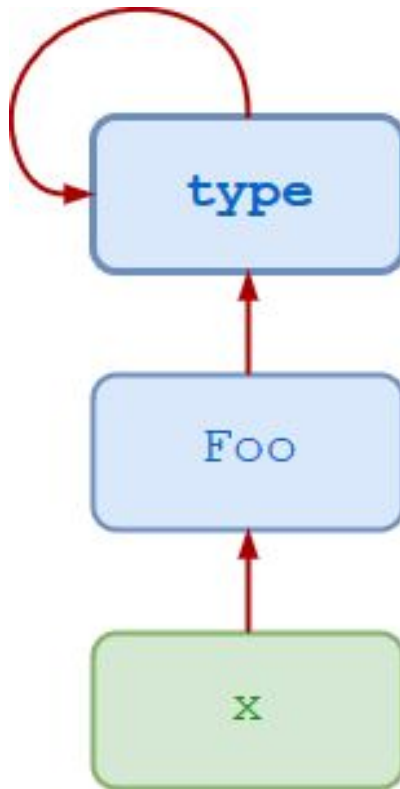
Классы, экземпляры которых  
являются классами



## Метаклассы: type

```
class Foo:  
    pass
```

```
x = Foo()
```



# Метаклассы: `type`

Новые классы создаются с помощью вызова

```
type(<name>, <bases>, <classdict>)
```

`name` – имя класса (`__name__`)

`bases` – базовые классы (`__bases__`)

`classdict` – namespace класса (`__dict__`)

```
MyClass = type("MyClass", (), {})
```

## Метаклассы: type

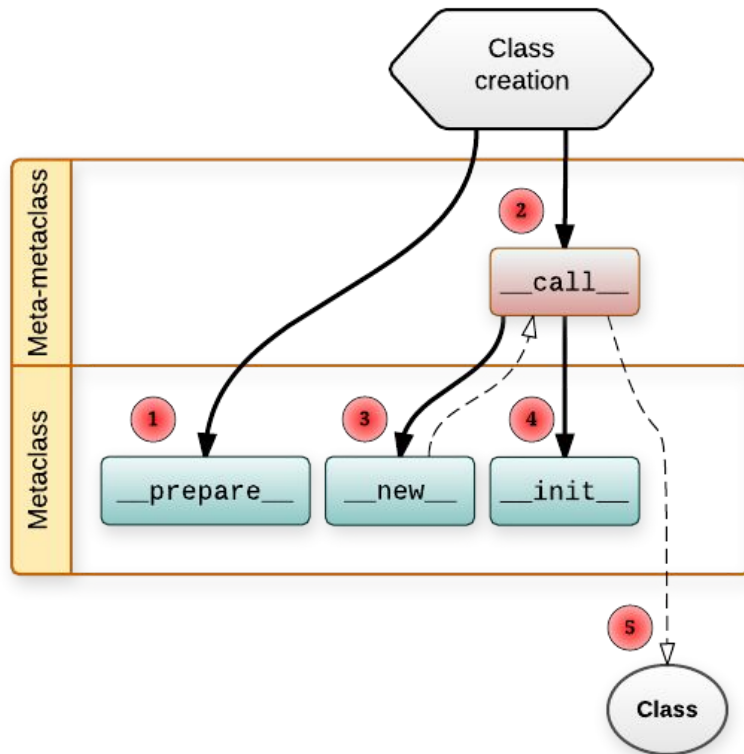
```
>>> Bar = type('Bar', (Foo,), dict(attr=100))
>>> x = Bar()
>>> x.attr
100
>>> x.__class__
<class '__main__.Bar'>
>>> x.__class__.__bases__
(<class '__main__.Foo'>,)

>>> class Bar(Foo):
...     attr = 100
...
>>> x = Bar()
>>> x.attr
100
>>> x.__class__.__bases__
(<class '__main__.Foo'>,)

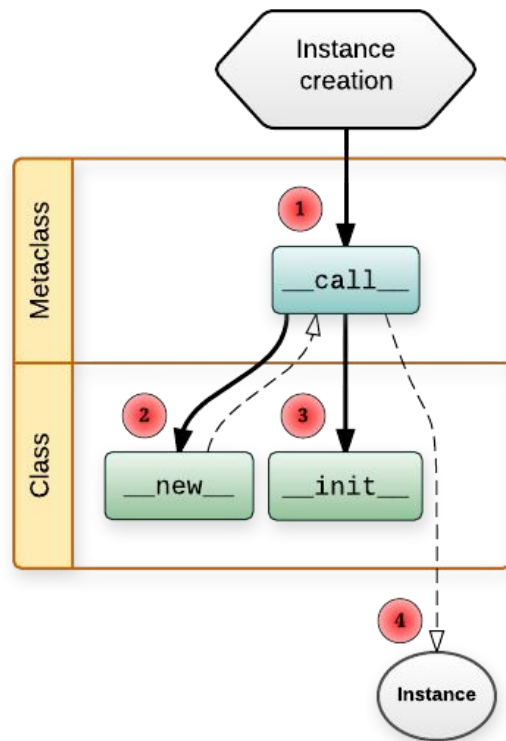
```



# Метаклассы: создание класса



# Метаклассы



# Метаклассы: создание класса

- определяются базовые классы
- определяется метакласс
- подготавливается namespace класса (`__prepare__`)
- выполняется тело класса
- создается класс (`__new__`, `__init__`)

# Метаклассы

```
class AMeta(type):  
    def __new__(mcs, name, bases, classdict, **kwargs):  
        cls = super().__new__(mcs, name, bases, classdict)  
        print('Meta __new__', cls)  
        return cls  
  
    def __init__(cls, name, bases, classdict, **kwargs):  
        super().__init__(name, bases, classdict, **kwargs)  
  
    def __call__(cls, *args, **kwargs):  
        return super().__call__(*args, **kwargs)  
  
    @classmethod  
    def __prepare__(mcs, name, bases, **kwargs):  
        print('Meta __prepare__', **kwargs)  
        return {'b': 2, 'a': 2}
```

# ABC

Добавляем абстракции



# ABC

```
>>> from abc import ABCMeta, abstractmethod
>>> class C(metaclass=ABCMeta):
...     @abstractmethod
...     def abs_method(self):
...         pass
>>> c = C()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class C with abstract methods abs_method

>>> class B(C):
...     def abs_method(self):
...         print("Now a concrete method")
>>> b = B()
>>> b.abs_method()
Now a concrete method
```

# ABC

```
class Hashable(metaclass=ABCMeta):
    __slots__ = ()
    @abstractmethod
    def __hash__(self):
        return 0

    @classmethod
    def __subclasshook__(cls, C):
        if cls is Hashable:
            return _check_methods(C, "__hash__")

        return NotImplemented
```

```
>>> from collections.abc import Hashable
>>> isinstance("123", Hashable) # ???
>>> isinstance({}, Hashable) # ???
```

## Домашнее задание #04

- Реализация метакласса для добавления префикса `custom_` ко всем атрибутам
- Дескрипторы с проверками типов и значений данных
- Тесты
- `flake8` и `pylint` перед сдачей



# Напоминание отметиться на портале Vol 2

+ ОСТАВИТЬ ОТЗЫВ ПОСЛЕ ЛЕКЦИИ

vk образование

БлогиЛюдиПрограммаВакансииРасписание

Q

VK

Техно

Открыт приём заявок!

чт, 8 сентября

Нет занятий

пт, 9 сентября

18:00 Углубленный Py... с3  
Введение в Python, основные  
понятия, тестирование  
Г. Кандауров

сб, 10 сентября

Нет занятий

вс, 11 сентября

Нет занятий

пн, 12 сентября

Нет занятий

Углубленный Python

↓ 0 ↑

Блог для материалов по курсу "Углубленный Python"

57 читателей, 2 топика

ПодписатьсяСоздать топик

Поиск по авторам, заголовку и тексту топика...

Найти

Добро пожаловать на курс!

Углубленный Python

ИзменитьУдалить

Всем привет и добро пожаловать на курс по углубленному изучению Python!

Прямой эфир

МоиВсе

Геннадий Кандауров час назад  
Углубленный Python → Добро пожаловать  
на курс! 0

Екатерина Черкасова 7 дней назад  
Стажировка → Приглашаем мобильных,  
фронтенд- и бэкэнд-разработчиков на  
Weekend Offer! 0

Дарья Вовченко 9 дней назад  
Углубленный Python → Добро пожаловать  
в образовательные проекты VK  
Образование! 0

Дарья Вовченко 9 дней назад  
Разработка веб-сервисов на

Спасибо за  
внимание

