Углубленный Python

Лекция 4

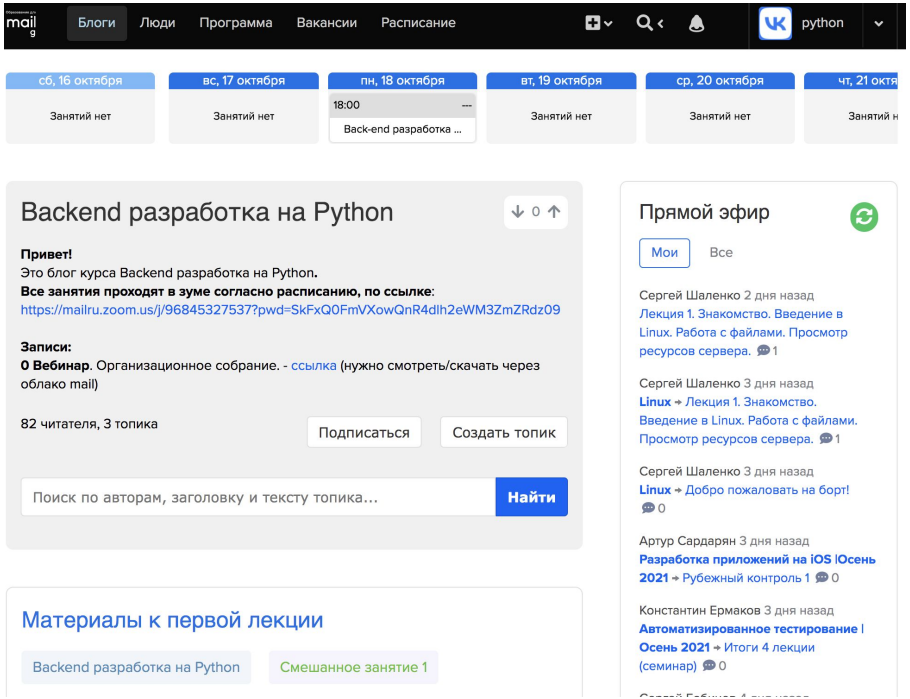
Метапрограммирование, дескрипторы, ABC

Кандауров Геннадий

образование

Напоминание отметиться на портале

+ оставить отзыв после лекции

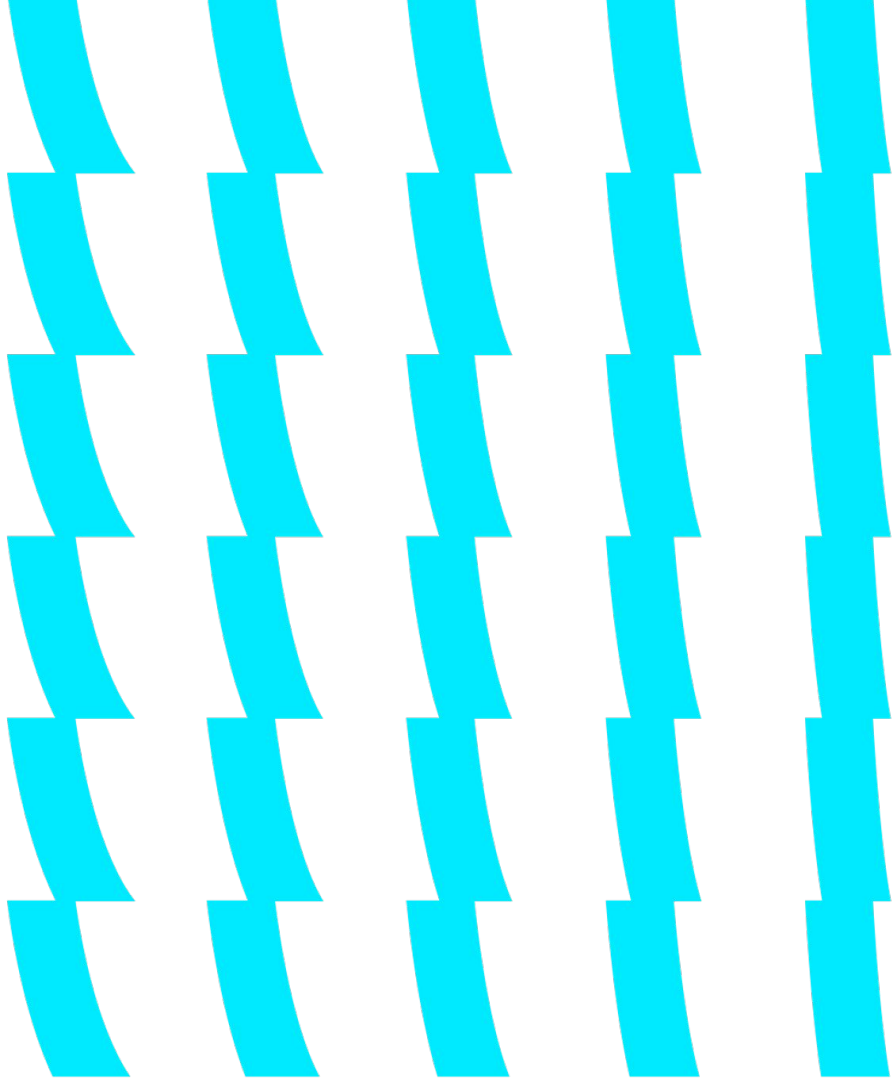


Квиз про прошлой лекции

Содержание занятия

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

Классы



Классы

```
class A:
    @staticmethod
    def print_static():
        print("static")

    @classmethod
    def print_cls(cls):
        print(f"class_method for {cls.__name__}")

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

    def print_offset(self, offset=10):
        print(self.val + offset)

    def __str__(self):
        return f"{self.__class__.__name__}:val={self.val}"
```

Классы: магические методы

__hash__

Вызывается функцией `hash()` и коллекциями, которые построены на основе hash-таблиц. Нужно, чтобы у равных объектов был одинаковый hash.

Если определен метод `__eq__` и не определен `__hash__`, то объект не может быть ключом в hashable коллекции.

```
>>> key1 = (1, 2, 3)
>>> key2 = (1, 2, 3, [4, 5])
>>> s = set()
>>> s.add(key1)  # ???
>>> s.add(key2)  # ???
```

Классы: магические методы

`__slots__`

Позволяет явно указать поля, которые будут в классе.

В случае указания `__slots__` пропадают поля `__dict__` и `__weakref__`.

Используя `__slots__` можно экономить на памяти и времени доступа к атрибутам объекта.

```
class Point:
    __slots__ = ("x", "y")
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

collections.namedtuple

namedtuple(typename, field_names, *, rename=False, defaults=None, module=None)

```
>>> Point = collections.namedtuple("Point", ["x", "y"])
>>> p = Point(11, y=22)  # p = (11, 22)
>>> p[0] + p[1]
33
>>> x, y = p
>>> x, y
(11, 22)
>>> p.x + p.y
33
>>> p._asdict()  # {'x': 1, 'y': 4}
```

Enum

```
class StatusCode(Enum):
    OK = 200
    NOT_FOUND = 404
    ERROR = 500

    @classmethod
    def _missing_(cls, code):
        print("MISSED", code)
        return cls.OK

ok, not_ok = StatusCode(200), StatusCode["ERROR"]
```

Классы: наследование

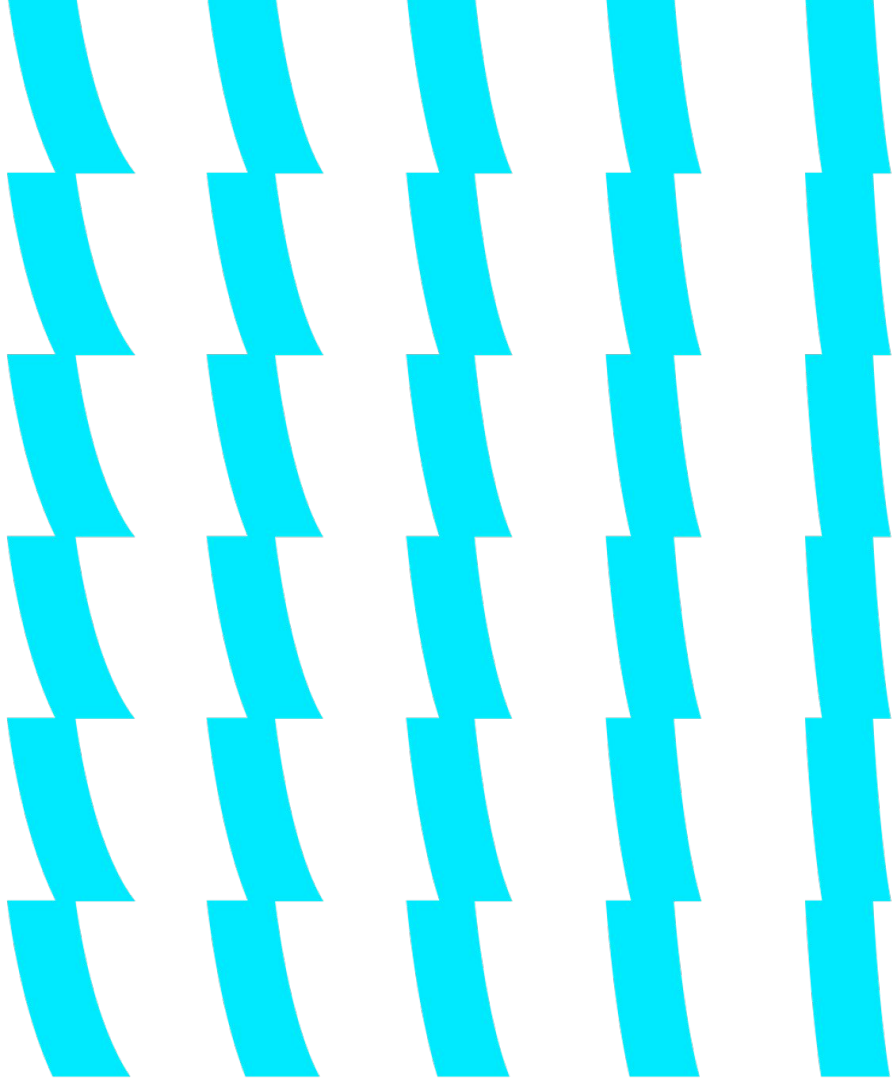
```
class Timing:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def duration(self):
        print("Timing.duration")
        return self.end - self.start


class MinuteTiming(Timing):
    def duration(self):
        print("MinuteTiming.duration")
        seconds = super().duration()
        return seconds / 60
```

```
>>> m = MinuteTiming(1000, 7000)
>>> m.duration()
MinuteTiming.duration
Timing.duration
100.0
```

Классы: __init_subclass__

```
class Timing:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    @classmethod
    def __init_subclass__(cls, **kwargs):
        print("INIT subclass", cls, kwargs)


class MinuteTiming(Timing):
    def duration(self):
        print("MinuteTiming.duration")
        seconds = super().duration()
        return seconds / 60
```

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 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>>
```

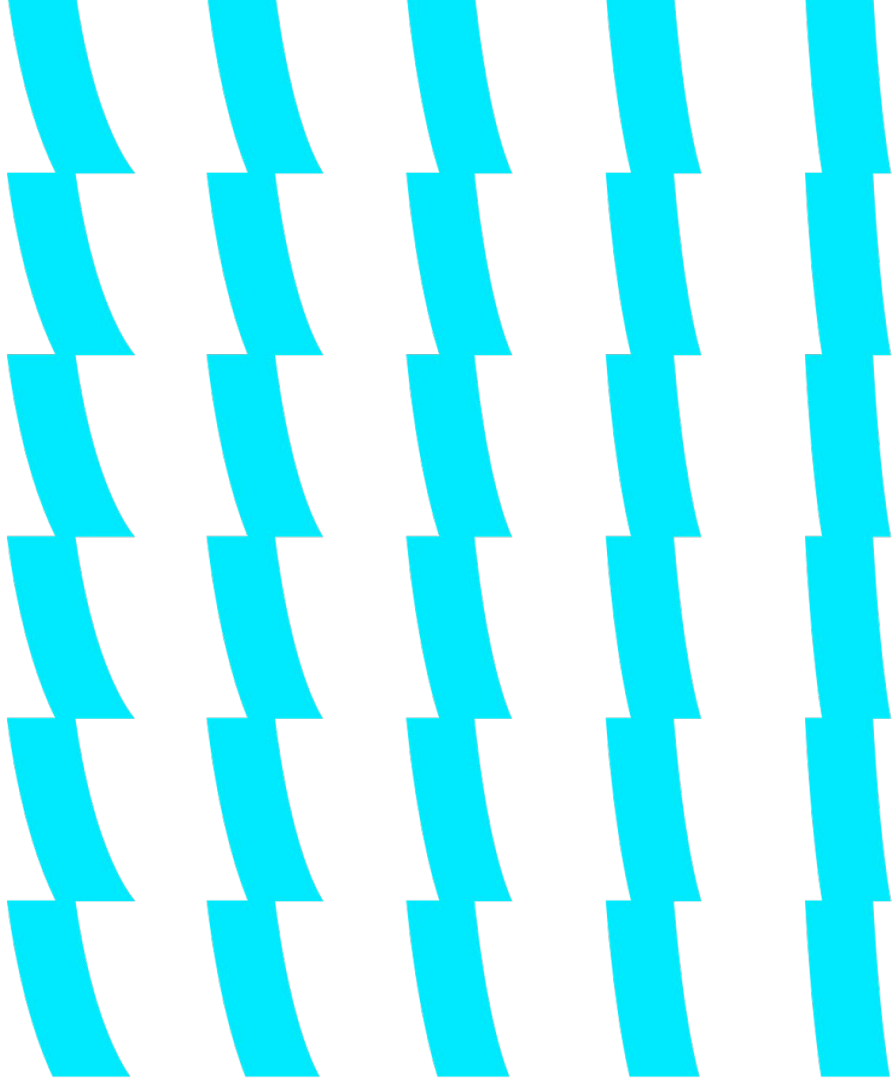
Дескрипторы

```
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 …>
```

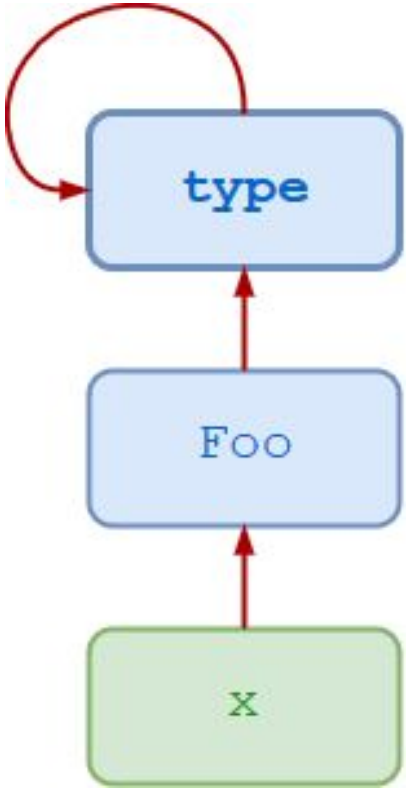
Метаклассы

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__`)

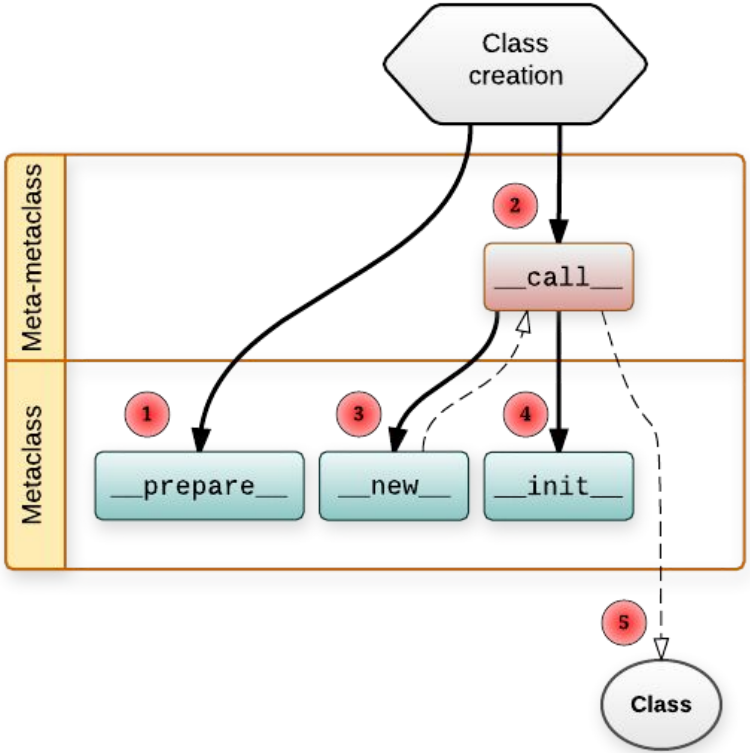
`classdict` – namespace класса (`__dict__`)

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

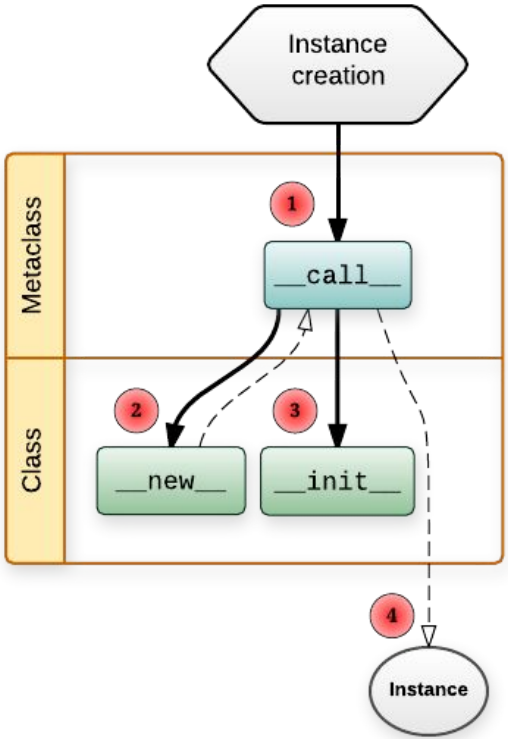
Метаклассы: type

```
>>> 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


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

Метаклассы: создание класса

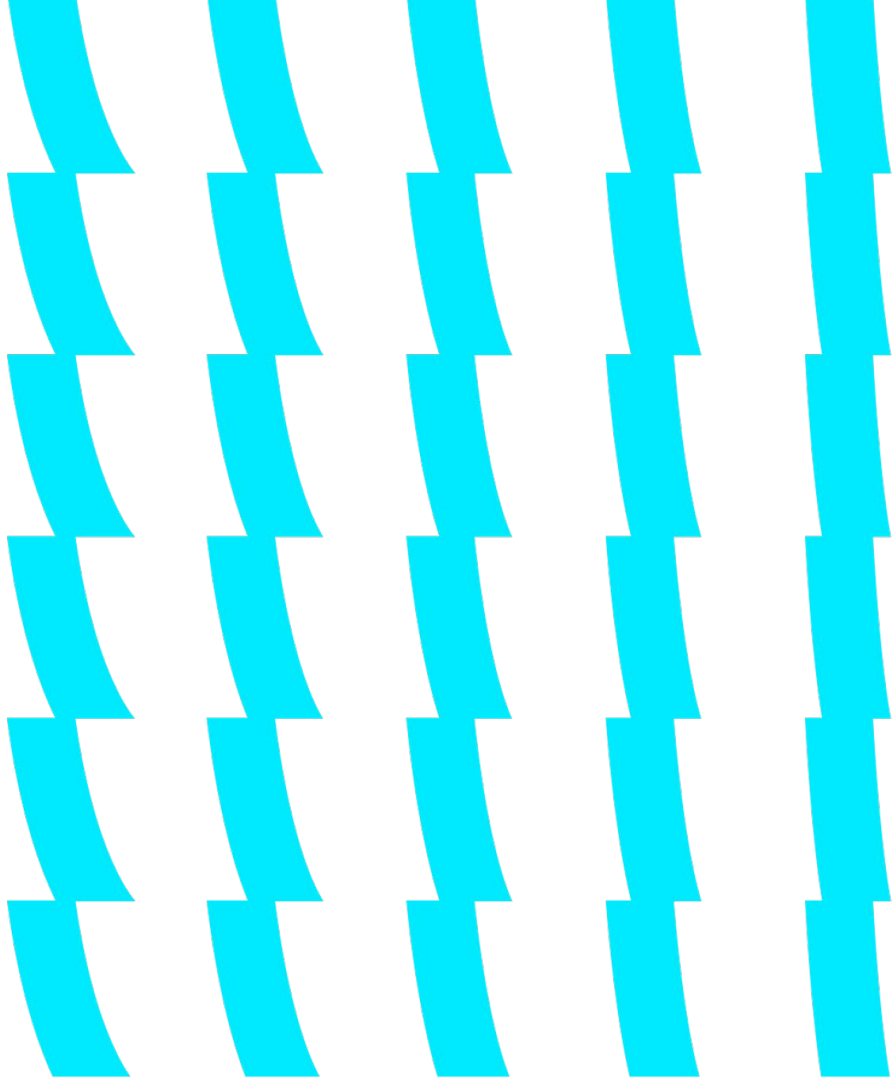
- определяются базовые классы
- определяется метакласс
- подготавливается namespace класса (`__prepare__`)
- выполняется тело класса
- создается класс (`__new__`, `__init__`)

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
>>> 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
```

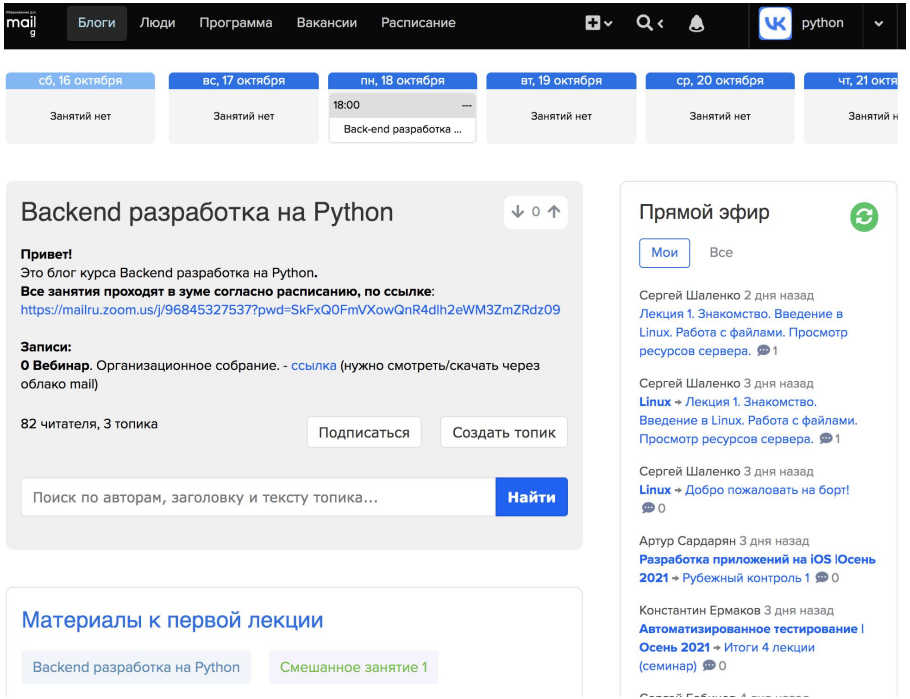
```
>>> from collections.abc import Hashable
>>> isinstance("123", Hashable)  # ???
>>> isinstance({}, Hashable)  # ???
```

Домашнее задание #4

- Реализация метакласса с префиксом custom_
- Дескрипторы с проверками типов и значений данных
- +тесты
- flake8 + pylint перед сдачей

Напоминание отметиться на портале Vol 2

+ оставить отзыв



Спасибо за внимание

образование