GeekBrains

Погружение в Python

Урок 12

ООП. Финал

Содержание урока

План курса

1. Основы Python
2. Простые типы данных
3. Коллекции
4. Функции
5. Итераторы и генераторы
6. Модули
7. Файлы и файловая система
8. Сериализация
9. Декораторы
10. ООП. Начало
11. ООП. Особенности Python
12. ООП. Финал
13. Исключения
14. Основы тестирования
15. Обзор стандартной библиотеки Python

Что будет на уроке сегодня

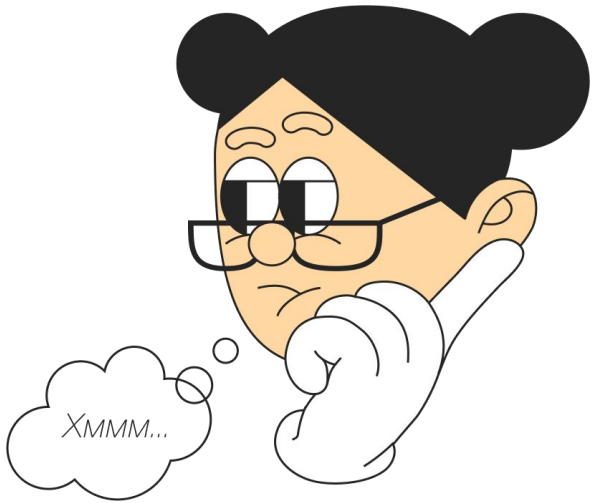
- Разберёмся в превращении объекта в функцию
- Изучим способы создания итераторов
- Узнаем о создании менеджеров контекста
- Разберемся в превращении методов в свойства
- Изучим работу дескрипторов
- Узнаем о способах экономии памяти

Класс как функция

Метод вызова функции __call__

Python позволяет вызывать экземпляр класса как функцию
с передачей аргументов в круглых скобках

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

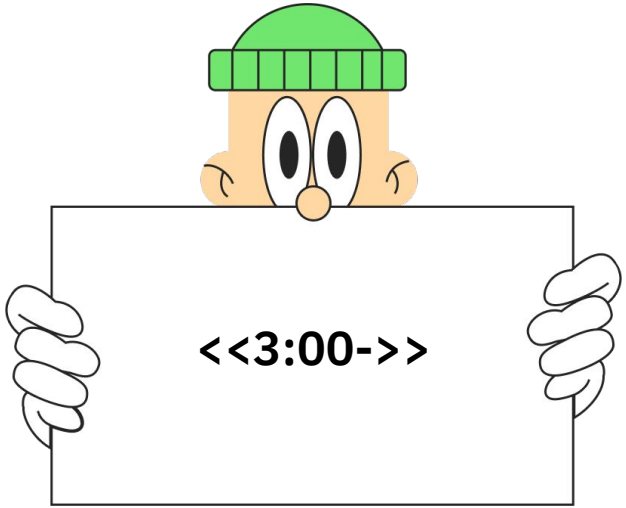


Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Класс как функция

```
class MyClass:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return f'MyClass(a={self.a}, b={self.b})'

    def __call__(self, *args, **kwargs):
        self.a.append(args)
        self.b.update(kwargs)
        return True

x = MyClass([42], {73: True})
y = x(3.14, 100, 500, start=1)
x(y=y)
print(x)
```

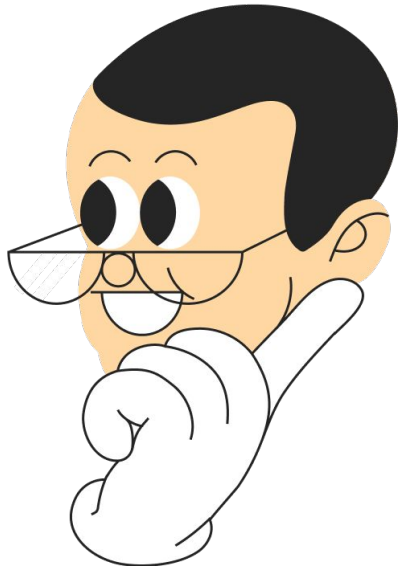


Создаём итераторы

Создаём итераторы

Если экземпляр класса должен итерироваться,
необходимо реализовать пару дандер методов

- **__iter__** — возвращает объект итератор, например self
- **__next__** — возвращает очередное значение. Для завершения итерации вызывает исключение StopIteration

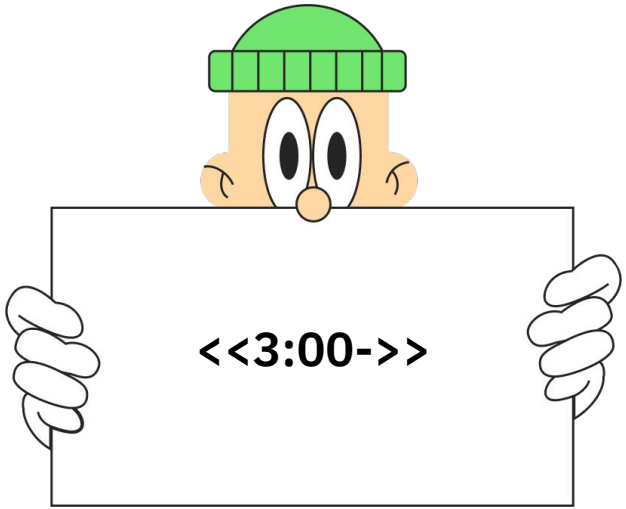


Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Создаём итераторы

```
class Iter:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        for i in range(self.start, self.stop):
            return chr(i)
        raise StopIteration


chars = Iter(65, 91)
for c in chars:
    print(c)
```

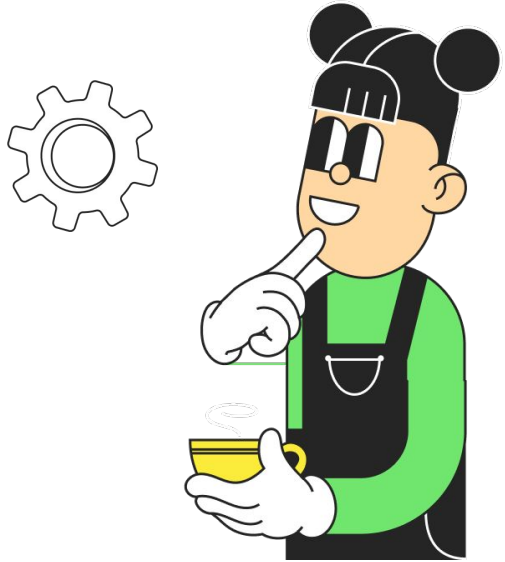


Создаём менеджер контекста with

Создаём менеджер контекста with

Менеджер контекста with запускает два дандер метода. Один в момент вызова менеджера, а второй в момент выхода из внутреннего блока кода.

- **__enter__** — действия при входе в менеджер контекста
- **__exit__** — действия при выходе из менеджера контекста

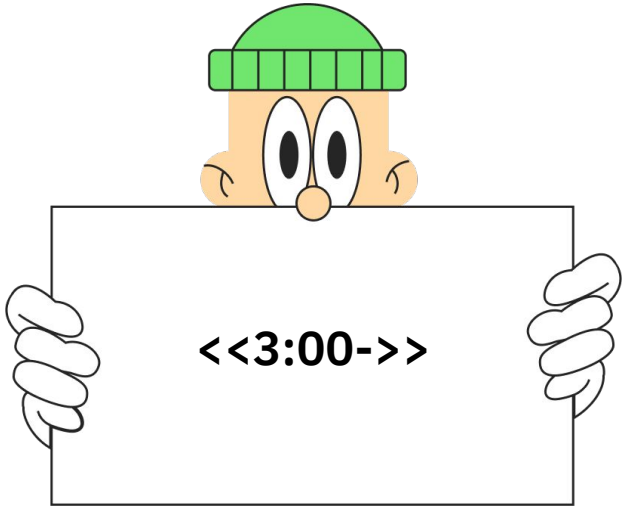


Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Создаём менеджер контекста with

```
class MyCls:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __enter__(self):
        self.full_name = self.first_name +' ' + self.last_name
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.full_name = self.full_name.upper()

x = MyCls('Гвидо ван', 'Россум')
with x as y:
    print(y.full_name)
    print(x.full_name)
print(y.full_name)
print(x.full_name)
```

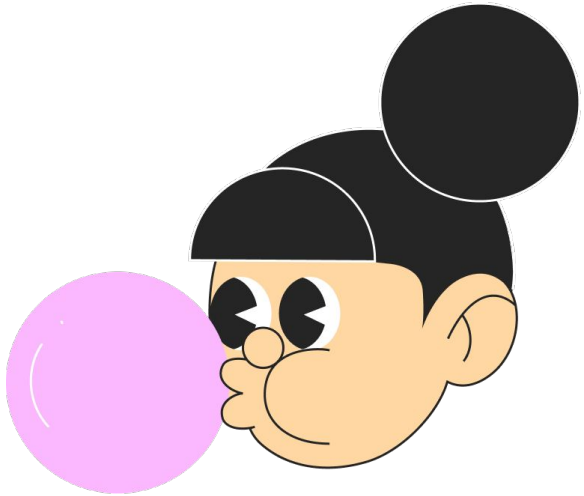


Декоратор @property

Getter

Getter — метод, выдающий себя за свойство

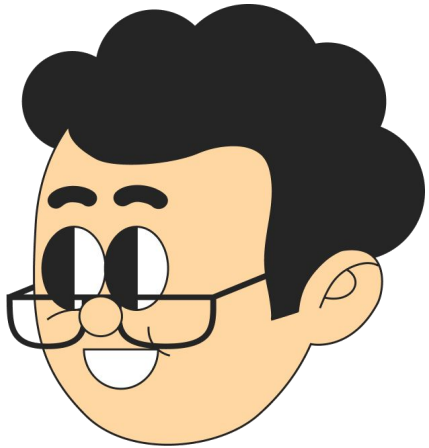
```
class Name:
    ...
        @property
        def param(self):
            ...
            return ...
```



Setter

Setter контролирует изменение защищённого свойства через метод

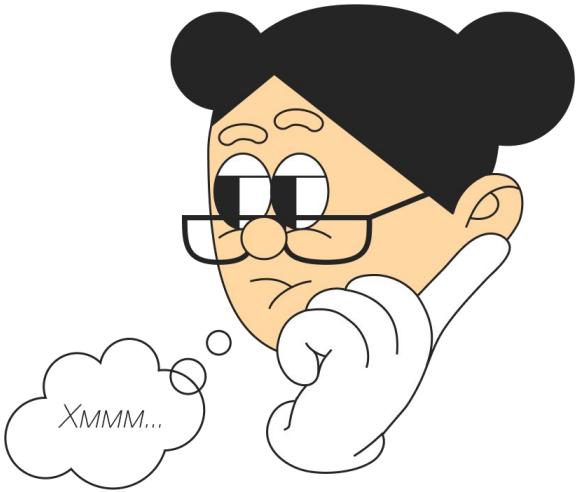
```
class Name:
    ...
        @property
        def param(self):
            ...
            return ...

        @param.setter
        def param(self, value):
            ...
            self._param = value
```



Deleter

Deleter изменяет поведение при попытке удалить свойство командой del

```
class Name:
    ...
        @property
        def param(self):
            ...
            return ...

        @param.deleter
        def param(self):
            ...
            self._param = ...
```

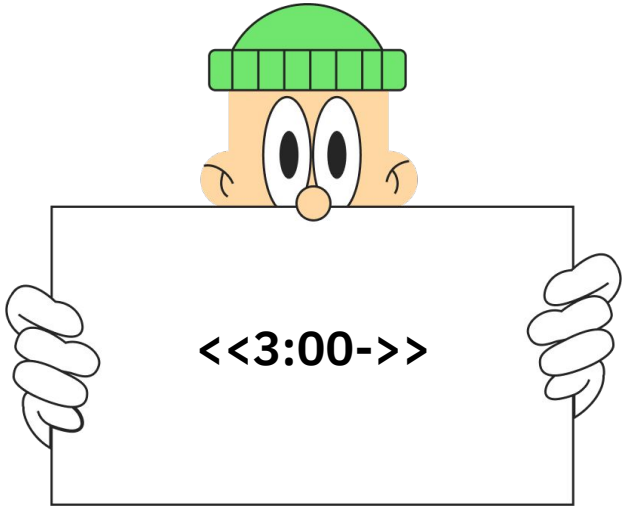


Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Декоратор @property

```
class MyCls:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return self.first_name + ' ' + self.last_name

    @full_name.setter
    def full_name(self, value: str):
        self.first_name, self.last_name, _ = value.split()

x = MyCls('Стивен', 'Хокинг')
print(x.full_name)
x.full_name = 'Гвидо ван Россум'
print(x.full_name)
```

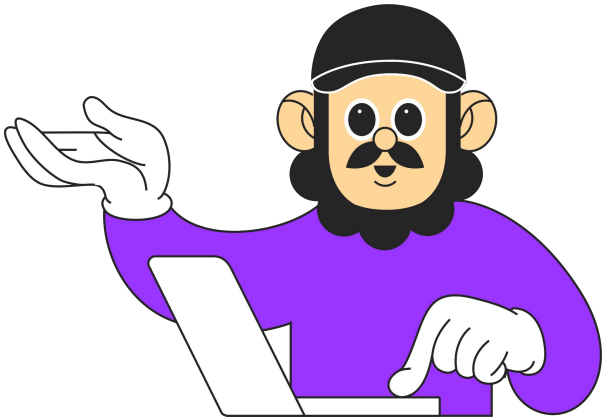


Дескрипторы

Дескриптор

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 метод дескриптор.

Рассмотрим большой практический пример.

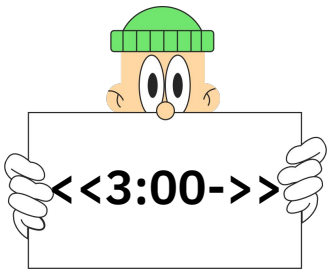


Перед вами несколько строк кода.
Напишите в чат что они вернут
не запуская программу.

У вас 3 минуты.

Дескрипторы

```
class Text:
    def __init__(self, param):
        self.param = param

    def __set_name__(self, owner, name):
        self.param_name = '_' + name

    def __set__(self, instance, value):
        if self.param(value):
            setattr(instance, self.param_name, value)
        else:
            raise ValueError(f'Bad {value}')
```



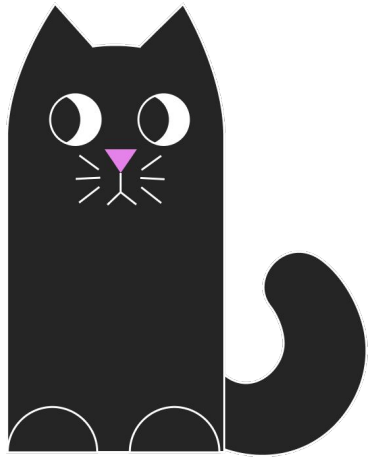

```
class User:
    first_name = Text(str.istitle)
    last_name = Text(str.isupper)

    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __repr__(self):
        return f'Student(age={self.first_name},
                    grade={self.last_name})'


if __name__ == '__main__':
    std_one = User('Гвидо ван', 'Россум')
```

Экономим память

Экономим память

- ✔ **Дандер __dict__** представляет словарь внутри объекта, который хранит в качестве ключей имена атрибутов, а в качестве значений ссылки на них. Словари занимают много места в памяти.

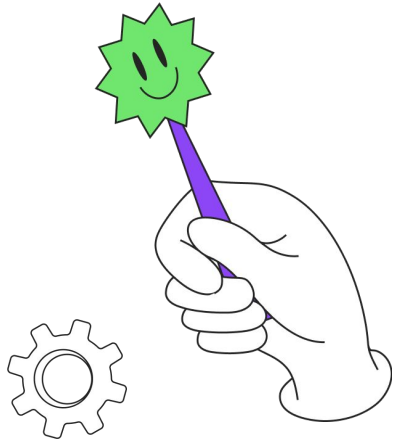
- ✔ **Дандер __slots__** представляет линейный массив, который хранит только перечисленные переменные.



Что делает __slots__

Пять причин использовать слоты из официальной документации

- ✔ Обеспечивает немедленное обнаружение ошибок из-за неправильного написания атрибутов. Допускаются только имена атрибутов, указанные в __slots__
- ✔ Помогает создавать неизменяемые объекты, в которых дескрипторы управляют доступом к закрытым атрибутам, хранящимся в __slots__
- ✔ Экономит память. В 64-битной сборке Linux экземпляр с двумя атрибутами занимает 48 байт со __slots__ и 152 байт без него. Экономия памяти имеет значение только тогда, когда будет создано большое количество экземпляров.
- ✔ Улучшает скорость. По данным на Python 3.10 на процессоре Apple M1 чтение переменных экземпляра выполняется на 35% быстрее со __slots__().
- ✔ Блокирует такие инструменты как functools.cached_property(), которым для правильной работы требуется экземплярный словарь.



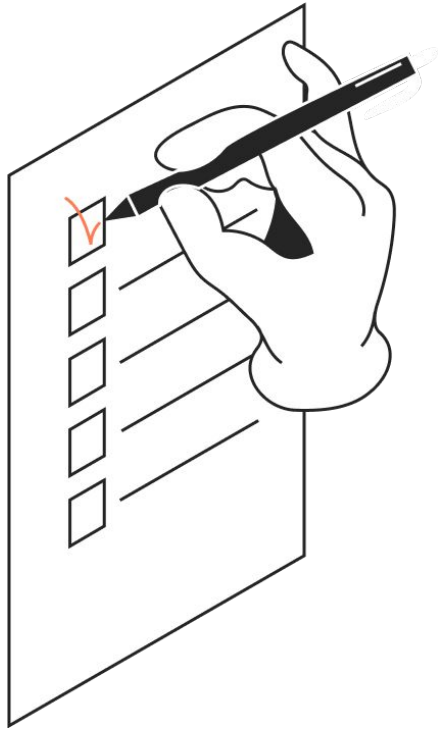
Итоги занятия

На этой лекции мы

- Разобрались в превращении объекта в функцию
- Изучили способы создания итераторов
- Узнали о создании менеджеров контекста
- Разобрались в превращении методов в свойства
- Изучили работу дескрипторов
- Узнали о способах экономии памяти

Задание

Возьмите 1-3 задачи из прошлых занятий и попробуйте перенести переменные и функции в класс, если это не задачи про классы. Добавьте к ним дандер методы из лекции для решения исходной задачи.



Спасибо за внимание