GeekBrains

Лекция 12. ООП. Финал

Погружение в Python

Оглавление

На этой лекции мы 2
Дополнительные материалы к лекции 2
Краткая выжимка, о чём говорилось в предыдущей лекции 2
Термины лекции 3
Подробный текст лекции
1. Класс как функция 3
2. Создаём итераторы 5
3. Создаём менеджер контекста with 8
4. Декоратор @property 11
Getter 12
Setter 13
Deleter 14
Задание 15
5. Дескрипторы 16
Класс-дескриптор Range 19
Контроль имён, __set_name__ 19
Контроль получения значений, __get__ 20
Контроль присвоение значений, __set__ 20
Контроль удаления атрибутов, __delete__ 20
Класс Student 20
6. Экономим память 21
Краткий анонс следующей лекции 24

На этой лекции мы

1. Разберёмся с созданием и удалением классов

2. Узнаем о документировании классов
3. Изучим способы представления экземпляров
4. Узнаем о возможностях переопределения математических операций
5. Разберёмся со сравнением экземпляров
6. Узнаем об обработке атрибутов

Дополнительные материалы к лекции

Руководство по работе с дескриптором
https://docs.python.org/3/howto/descriptor.html

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с созданием и удалением классов
2. Узнали о документировании классов
3. Изучили способы представления экземпляров
4. Узнали о возможностях переопределения математических операций
5. Разобрались со сравнением экземпляров
6. Узнали об обработке атрибутов

Термины лекции

- **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.

Подробный текст лекции

1. Класс как функция

При желании можно заставить класс, а точнее его экземпляры вести себя как функции. После имени экземпляра указываются круглые скобки с параметрами для вызова и экземпляр возвращает ответ. Разберём как это работает.

```
class Number:
    def __init__(self, num):
        self.num = num


n = Number(42)
print(f'{callable(Number) = }')
print(f'{callable(n)      = }')
```

Класс Number имеет метод инициализации для сохранения числа. Мы создали экземпляр класса n и воспользовались встроенной функцией callable. Для класса получили истину, для экземпляра ложь. Функция отвечает на вопрос вызываемый перед нами объект или нет. Вызов класса возможен. Он запускает инициализацию и возвращает экземпляр. Вызвать экземпляр нельзя.

Метод вызова функции __call__

Создадим класс, экземпляры которого можно вызывать. Например для добавления очередного элемента во внутренний словарь класса по типам.

```
from collections import defaultdict


class Storage:
    def __init__(self):
        self.storage = defaultdict(list)

    def __str__(self):
                txt = '\n'.join((f'{k}: {v}' for k, v in
self.storage.items()))
        return f'Объекты хранилища по типам:\n{txt}'
```

```
    def __call__(self, value):
        self.storage[type(value)].append(value)
        return f'К типу {type(value)} добавлен {value}'


s = Storage()
print(s(42))
print(s(72))
print(s('Hello world!'))
print(s(0))
print(s)
```

При создании класса используется продвинутая версия словаря из модуля collections — defaultdict. Словарю передана функция list. При обращении к несуществующему ключу вместо ошибки будет создан ключ и вызвана функция list для создания значения ключа.
Каждый вызов экземпляра добавляет переданный аргумент value в словарь storage и возвращает строку с информацией о выполненном действии.
Последовательно вызывая экземпляр с числами и текстом выводим его не печать и видим содержимое на момент печати.
Плюсом вызова экземпляра является то, что он не удаляется из памяти после вызова как обычная функция. Следовательно экземпляр может накапливать значения, использоваться в технологии мемоизации. Её рассматривали на лекции о декораторах.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
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)
```

2. Создаём итераторы

Список list можно передать в цикл for in для перебора его элементов, итерации. Также итерироваться по списку можно в генераторных выражениях. А можно передать список функции для итерации, например функции all(). У итерируемых объектов много способов использования. Можно ли создать итерируемый объект самому? Да. Если экземпляр класса должен итерироваться, необходимо реализовать пару дандер методов.
Создадим класс экземпляр которого будет выдавать числа Фибоначчи в диапазоне начиная с числа больше или равного start и заканчивая числом меньше stop.

```
class Fibonacci:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
        self.first = 0
        self.second = 1


fib = Fibonacci(20, 100)
for num in fib:  # TypeError: 'Fibonacci' object is not iterable
    print(num)
```

Внутри дандер __init__ запомнили границы start и stop и определили нулевое и первое число Фибоначчи в свойствах first и second соответственно.
Создание экземпляра не вызывает проблем. А попытка получить числа в цикле увенчалась ошибкой. Python сообщил, что объект не итерируемый.

Возврат итератора, __iter__

Для того, чтобы объект стал итерируемым, ему необходимо вернуть объект-итератор. В нашем случае экземпляр класса и есть объект-итератор. Следовательно он должен вернуть сам себя. Для возврата итератора нужно создать дандер метод __iter__.

```
class Fibonacci:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
        self.first = 0
        self.second = 1

    def __iter__(self):
        return self


fib = Fibonacci(20, 100)
for num in fib:   # TypeError: iter() returned non-iterator of type 'Fibonacci'
    print(num)
```

Две строки метода вернули ссылку на самого себя. В результате получаем новую ошибку. Вернулся не итерируемый объект.

Возврат очередного значения, __next__

Как вы помните из лекции об итераторах и генераторах, любая итерация представляет из себя последовательный вызов функции next() с итератором в качестве аргумента.
Для возврата такого значения необходимо определить дандер метод __next__.

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

```
        self.first = 0
        self.second = 1

    def __iter__(self):
        return self

    def __next__(self):
        while self.first < self.stop:
              self.first, self.second = self.second, self.first +
self.second
            if self.start <= self.first < self.stop:
                return self.first
        raise StopIteration


fib = Fibonacci(20, 100)
for num in fib:
    print(num)
```

Итератор отработал как и ожидалось.

- ➢ дандер __next__ создаёт цикл пока число в first не превысит значение stop
- ➢ получаем следующую пару Фибоначчи в first и second
- ➢ если first оказывается внутри диапазона [start, stop), возвращаем очередной элемент
- ➢ обязательным условием для завершения итерации является вызов ошибки StopIteration. Python обрабатывает её как сигнал для завершения итерации и перехода к следующему за циклом коду. Остановки кода по ошибке не будет.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
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)
```

3. 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. Знакомая нам функция open() поддерживает работу с менеджером контекста. При вызове менеджера функция возвращает файловый дескриптор. А при выходе из него закрывает файл. Подобный функционал можно реализовать для любого объекта, где нужны одинаковые действия в начале и в конце. Рассмотрим пример работы с базой данных sqlite.

```
import sqlite3

connection = sqlite3.connect('sqlite.db')
cursor = connection.cursor()
cursor.execute("""create table if not exists users(name,
age);""")
cursor.execute("""insert into users values ('Гвидо', 66);""")
connection.commit()
connection.close()
```

Получение соединения с базой данных и получение курсора из соединения — обязательное начало для работы с базой.
Подтверждение изменений вызовом commit() и закрытие соединения с базой — обязательные действия в конце работы с базой.
Можно держать соединение открытым и подтверждать коммитить изменения после каждого действия с базой. А можно создать менеджер контекста.

- **Действия при входе в менеджер контекста, __enter__**

Создадим класс DB для упрощения работы с базой данных.

```
import sqlite3


class DB:
    def __init__(self, name):
        self.name = name
        self.connection = None
        self.cursor = None

    def __enter__(self):
        self.connection = sqlite3.connect(self.name)
        self.cursor = self.connection.cursor()
        return self.cursor


db = DB('sqlite.db')
with db as cur:  # AttributeError: __exit__
        cur.execute("""create table if not exists users(name,
age);""")
    cur.execute("""insert into users values ('Гвидо', 66);""")
```

Экземпляр класса хранит имя базы, которое задаём один раз при получении экземпляра. Дополнительно запоминаем соединение и курсор. В момент создания экземпляра им присваиваем None.
Дандер __enter__ определяет действия при входе в менеджер контекста. В нашем случае это установление соединения с базой данных и получение курсора. Сам курсор возвращаем в менеджер в переменную после as.
Если запустить код, получим ошибку доступа к атрибуту. Менеджер отказывается работать без указания действий для выхода.

- **Действия при выходе из менеджера контекста, __exit__**

Добавим дандер метод __exit__ и пропишем в нём операции, обязательные при завершении работы с базой данных.

```
import sqlite3


class DB:
    def __init__(self, name):
        self.name = name
        self.connection = None
        self.cursor = None

    def __enter__(self):
        self.connection = sqlite3.connect(self.name)
```

```
        self.cursor = self.connection.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.connection.commit()
        self.connection.close()
        self.cursor = self.connection = None


db = DB('sqlite.db')
with db as cur:
        cur.execute("""create  table  if  not  exists  users(name,
age);""")
    cur.execute("""insert into users values ('Гвидо', 66);""")
```

Внутри __exit__ подтверждаем изменения, закрываем соединения с базой и обнуляем свойства экземпляра. Параметры дандер __exit__ содержат информацию о типе и значении ошибки и трассировку, если она возникла внутри менеджера. Если ошибок не было, все три параметра содержат None.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
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)
```

4. Декоратор @property

На прошлой лекции мы работали с классом треугольник и пометили его свойства защищёнными, добавив символ подчёркивания в начале имени. Но что если доступ к свойству нужен. Хотя бы на чтение. Для этого отлично подойдёт функция декоратор property(). Рассмотрим на более простом и коротком примере.

```
class User:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name


user = User('Стивен')
print(f'{user.name = }')
user.name = 'Славик'    # AttributeError: can't set attribute
'name'
```

Класс User получает имя пользователя и сохраняет его в защищённой переменной экземпляра _name.
Далее создали метод name который возвращает значение из защищённого свойства _name. К методу применён декоратор property. Теперь Python воспринимает name не как имя вызываемого метода, а как название свойства.
При обращении к свойству name получаем результат — имя пользователя. Если же сделать попытку на изменение свойства, получим ошибку.

- **Getter**

Декоратор property позволяет создавать “геттеры”. Это методы, которые выдают себя за свойства, позволяют прочитать результат, но блокируют возможность записи. Рассмотрим другой пример “геттера”.

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

```
        self.last_name = last_name

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'


user = User('Стивен', 'Спилберг')
print(f'{user.first_name      =      }\n{user.last_name           =
}\n{user.full_name  = }')
user.full_name  =  'Стивен  Хокинг'    # AttributeError: can't  set
attribute 'full_name'
user.last_name = 'Хокинг'
print(f'{user.first_name      =      }\n{user.last_name           =
}\n{user.full_name  = }')
```

Теперь у пользователя есть два публичных свойства для имени и фамилии. Кроме того есть свойство (а не метод) для вывода полного имени, т.е. с фамилией. Все три свойства работают на чтение. А вот перезаписать полное имя мы не можем. Зато ничего не мешает изменить фамилию и получить обновлённое полное имя.

- **Setter**

Python позволяет к “геттеру” добавить “сеттер” — метод контролирующий изменение свойства. Добавим пользователю возраст и будем контролировать чтобы новый возраст был больше старого. Например мы вручную обновляем данные раз в 5-10 лет.

```
class User:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self._age = 0

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value > self._age:
```

```
            self._age = value
        else:
            raise ValueError(f'Новый возраст должен быть больше текущего: {self._age}')


user = User('Стивен', 'Спилберг')
user.age = 75
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
print('Прошёл один год.')
user.age = 76
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
print('Прошло несколько лет. Изобретена технология омоложения. Но возраст она не уменьшает.')
user.age = 25   # ValueError: Новый возраст должен быть больше текущего: 76
```

Что получилось:

1. защёщенное свойство _age получает значение ноль при рождении экземпляра, в дандер __init__
2. используя декоратор property создали свойство age для чтения текущего возраста
3. создаём “сеттер” для контроля записи новых значений в свойство _age
    - ➢ применяем декоратор @age.setter. Имя между @ и точкой должно совпадать с именем “геттера”.
    - ➢ методу присваиваем такое же имя как и у свойства и он должен принимать значения помимо self
    - ➢ внутри метода делаем проверку на увеличения возраста
    - ➢ если возраст увеличивается, обновляем свойство _age
    - ➢ если возраст не увеличился вызываем ошибку ValueError и сообщаем её причину
4. В основном коде запросто увеличиваем возраст пользователя, но не можем его уменьшить.

При создании “сеттера” не обязательно вызывать ошибки. В целом внутри может быть прописана любая логика. Например вы работает с финансовой программой и присваиваете новую сумму денег. “Сеттер” будет приводить сумму к типу Decimal перед присваиванием.

- **Deleter**

Помимо “геттера” и “сеттера” можно создать “делейтер”. Он выполняется при вызове команды del для свойства. Один из возможных вариантов использования “делейтера” - заменять значение на какое-то по умолчанию или помечать элемент

скрытым вместо удаления.

```
class User:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self._age = 0

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value > self._age:
            self._age = value
        else:
              raise ValueError(f'Новый возраст должен быть больше
текущего: {self._age}')

    @age.deleter
    def age(self):
        self._age = 0


user = User('Стивен', 'Спилберг')
user.age = 75
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
print('Прошло много лет. Изобретена технология перерождения.')
del user.age
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
```

Создание “делейтера” аналогично “сеттеру”. Также используется декоратор с именем свойства, но после точки пишем deleter. Внутри метода прописываются действия для удаления.

Антипаттерн геттера, сеттера, делейтера

Представленный ниже код является кодам ради кода и не имеет смысла в языке Python. Избегайте подобного. И да, код работает верно. Просто он не делает ничего нового.

```
class BadPattern:
    def __init__(self, x):
        self._x = x

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Все три декоратора ничего не делают. Подобный код в Python должен выглядеть так, без защиты переменной x

```
class GoodPattern:
    def __init__(self, x):
        self.x = x
```

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
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)
```

5. 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 метод дескриптор.

Звучит немного сложно. Так и есть. Дескрипторы не нужны для простых классов. Их польза проявляется при метапрограммировании, создании фреймворков.
Посмотрите на то как в Django создаются модели для работы с базой данных. Пример взят из официальной документации

```
from django.db import models

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

Как и почему работает код, где на уровне класса в обход инициализации создаются два свойства как экземпляры другого класса? Под капотом работают дескрипторы.
Напишем класс, который хранит имя ученика, его возраст, номер класса (от 1 до 11) и номер кабинета, в котором сидит класс.

```
class Student:
    def __init__(self, name, age, grade, office):
        self.name = name
        self.age = age
        self.grade = grade
        self.office = office
```

```
    def __repr__(self):
            return  f'Student(name={self.name},  age={self.age},
grade={self.grade}, office={self.office})'


std1 = Student('Шурик', 7, 1, 12)
print(std1)
```

А теперь внимательно посмотрим на числовые значения.

- ➢ возраст должен быть больше нуля
- ➢ класс должен быть от 1 до 11
- ➢ кабинет должен быть номером в каком-то диапазоне. Предположим, что в нашей школе кабинеты нумеруются от 3 до 42

Ничего не мешает добавить “сеттеры” для каждого из свойств. Декоратор property мы уже прошли. Но если присмотреться, в трёх случаях мы задаём диапазон для целого числа. Дескрипторы позволяют сделать проверку на диапазон один раз и использовать её для всех трёх свойств. Ниже полностью готовый код.

```
class Range:
    def __init__(self, min_value: int = None, max_value: int =
None):
        self.min_value = min_value
        self.max_value = max_value

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

    def __get__(self, instance, owner):
        return getattr(instance, self.param_name)

    def __set__(self, instance, value):
        self.validate(value)
        setattr(instance, self.param_name, value)

    def __delete__(self, instance):
        raise AttributeError(f'Свойство "{self.param_name}" нельзя
удалять')

    def validate(self, value):
        if not isinstance(value, int):
            raise TypeError(f'Значение {value} должно быть целым
числом')
        if self.min_value is not None and value < self.min_value:
            raise ValueError(f'Значение {value} должно быть больше
```

```
или равно {self.min_value}')
        if self.max_value is not None and value >= self.max_value:
            raise ValueError(f'Значение {value} должно быть меньше
{self.max_value}')


class Student:
    age = Range(3, 103)
    grade = Range(1, 11 + 1)
    office = Range(3, 42 + 1)

    def __init__(self, name, age, grade, office):
        self.name = name
        self.age = age
        self.grade = grade
        self.office = office

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


if __name__ == '__main__':
    std_one = Student('Архимед', 12, 4, 29)
    std_other = Student('Аристотель', 2406, 5, 17)   # ValueError:
Значение 2406 должно быть меньше 103
    print(f'{std_one   = }')
    std_one.age = 15
    print(f'{std_one   = }')
    std_one.grade = 11.0   # TypeError: Значение 11.0 должно быть
целым числом
    std_one.office = 73   # ValueError: Значение 73 должно быть
меньше 42
    del std_one.age   # AttributeError: Свойство "_age" нельзя
удалять
    print(f'{std_one.__dict__ = }')
```

Разберём код сверху вниз.

Класс-дескриптор Range

Мы создали дескриптор, реализующий все базовые дандер методы: чтения, записи и удаления.

В первую очередь инициализируем класс с параметрами для минимального и максимального значения. По умолчанию они равны None, любая из границ может быть открытой. Если же значения переданы, будем следовать правилу функции range - левая граница входит, а правая не входит.

- **Контроль имён, __set_name__**

Следующий метод срабатывает при определении имён свойств. В нашем случае это переменных уровня класса, определённые сразу после заголовка класса Student.
Обратите внимание на локальную переменную param_name, которая получает имя создаваемой переменной с символом подчёркивания в начале. Дандер занимается инкапсуляцией за нас.

- **Контроль получения значений, __get__**

Всего одна строчка кода использует функцию getattr() для получения у объекта instance значения для свойства self.param_name, того самого с подчёркиванием в начале. Мы ничего не меняем, а лишь возвращаем значение свойства экземпляра.

- **Контроль присвоение значений, __set__**

Дандер ничего не возвращает. В первой строк вызываем метод validate. Он отвечает за попадание целого числа в диапазон, заданный при инициализации. Вторая строка сработает в том случае, когда валидация пройдена успешно и не вызвала ошибки. В этом случае функция setattr() присваивает у экземпляра instance параметру self.param_name значение value. Это значение стоит справа от знака равно в основном коде.
Рассмотрим подробнее работу метода validate:

- ➢ метод принимает значение value и выполняет ряд проверок с ним
- ➢ если значение не является целым числом, вызываем ошибку TypeError
- ➢ если задана нижняя граница и значение меньше неё, вызываем ошибку ValueError
- ➢ аналогично если задана верхняя граница и значение больше или равно границе, вызываем ошибку ValueError
- ➢ в случае прохождения всех проверок метод ничего не возвращает. Он позволяет выполняться коду дальше

- **Контроль удаления атрибутов, __delete__**

Как понятно из названия метод срабатывает при попытке удалить свойство командой del. В нашем примере вызываем ошибку AttributeError и ничего не удаляем.

Класс Student

Далее создаём класс для хранения информации о студентах. На уровне класса задаём три переменные, которые являются экземплярами класса Range. Для этих свойств будут срабатывать методы дескриптора.

При инициализации студента получаем имя и три уже описанных параметра. Отдельно задаём дандер __repr__ для вывода на печать. Ничего нового тут нет.

В основном блоке кода с лёгкостью создаём первого студента. Все атрибуты прошли проверку. А вот создать второго студента не получилось, его возраст вышел за пределы диапазона. Как видите дескриптор работает уже на этапе инициализации экземпляра.

Если сделать изменение в пределах допустимого, код работает как обычно. А при попытке присвоения значения не проходящего валидацию получаем соответствующую ошибку.

Присмотритесь к ошибке при попытке удалить свойство age. Дескриптор сообщает, что _age нельзя удалить. Т.е. свойства скрыты от нас, но мы можем обращать используя обычные имена, без подчёркивания в начале.

В финальной строке смотрим дандер переменную __dict__ и видим, что наши свойства с диапазонами начинаются с подчёркивания. Результат работы дандер __set_name__.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
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')
```

6. Экономим память

Мы уже несколько раз сталкивались с дандер словарём __dict__. Его предназначение — хранить атрибуты и их значения у каждого объекта Python.

Хранитель атрибутов __dict__
Рассмотрим уже знакомый по прошлой лекции класс Triangle и выведем на печать содержимое __dict__ у экземпляра и у класса.

```
from math import sqrt


class Triangle:
    def __init__(self, a, b, c):
        self._a = a
        self._b = b
        self._c = c

    def __str__(self):
         return f'Треугольник со сторонами: {self._a}, {self._b},
{self._c}'

    def __repr__(self):
        return f'Triangle({self._a}, {self._b}, {self._c})'
```

```
    def __eq__(self, other):
        first = sorted((self._a, self._b, self._c))
        second = sorted((other._a, other._b, other._c))
        return first == second

    def area(self):
        p = (self._a + self._b + self._c) / 2
          _area = sqrt(p * (p - self._a) * (p - self._b) * (p -
self._c))
        return _area

    def __lt__(self, other):
        return self.area() < other.area()

    def __hash__(self):
        return hash((self._a, self._b, self._c))


triangle = Triangle(3, 4, 5)
print(triangle)
print(triangle.__dict__)
print(Triangle.__dict__)
```

Как видите экземпляр хранить лишь три свойства, определённые внутри метода инициализации. За всем остальным он обращается к своему классу.
Что касается класса, его словарь имена и адреса дандеров, методов и даже пустой дандер __doc__, ведь мы не сделали строку документации.
При редкой необходимости можно обращаться к ключам словаря для получения или изменения значений.

Экономия памяти, __slots__
При создании класса можно явно указать перечень имён свойств, которые в нём будут использоваться.

```
class Triangle:
    __slots__ = ('_a', '_b', '_c')

    def __init__(self, a, b, c):
    ...
```

Подобная запись говорит о том, что теперь у нас лишь три свойства. Python не позволит добавить новые.
А при попытке обратится к словарю экземпляра получим ошибку AttributeError: 'Triangle' object has no attribute '__dict__'. Did you mean: '__dir__'?

Коротко о том, что даёт замена изменяемого __dict__ на неизменяемый __slots__?

1. Обеспечивает немедленное обнаружение ошибок из-за неправильного написания атрибутов. Допускаются только имена атрибутов, указанные в __slots__
2. Помогает создавать неизменяемые объекты, в которых дескрипторы управляют доступом к закрытым атрибутам, хранящимся в __slots__
3. 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.
4. Улучшает скорость. По данным на Python 3.10 на процессоре Apple M1 чтение переменных экземпляра выполняется на 35% быстрее со __slots__().
5. Блокирует такие инструменты как functools.cached_property(), которым для правильной работы требуется экземплярный словарь.

Вывод

На этой лекции мы:

1. Разобрались в превращении объекта в функцию
2. Изучили способы создания итераторов
3. Узнали о создании менеджеров контекста
4. Разобрались в превращении методов в свойства
5. Изучили работу дескрипторов
6. Узнали о способах экономии памяти

Краткий анонс следующей лекции

1. Разберёмся с обработкой ошибок в Python
2. Изучим иерархию встроенных исключений
3. Узнаем о способе принудительного поднятия исключения в коде
4. Разберёмся в создании собственных исключений

Домашнее 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.