Серверная разработка ПО

Лекция 3

Четыре основных принципа объектно-ориентированного программирования следующие.

- Абстракция. Моделирование требуемых атрибутов и взаимодействий сущностей в виде классов для определения абстрактного представления системы.

- Инкапсуляция. Скрытие внутреннего состояния и функций объекта и предоставление доступа только через открытый набор функций.

- Наследование. Возможность создания новых абстракций на основе существующих.

- Полиморфизм. Возможность реализации наследуемых свойств или методов отличающимися способами в рамках множества абстракций.

Классы и объекты

В реальной жизни мы пользуемся не переменными или циклами, а объектами (автомобиль, книга, кошка). Взглянем более подробно на объект «кошка». Абсолютно все кошки обладают некоторыми характерными свойствами или параметрами. К ним можно отнести: наличие хвоста, четырех лап, шерсти, усов и т. п. Но это еще не все. Помимо определенных вешних параметров, кошка может выполнять свойственные ей действия: мурлыкать, шипеть, царапаться, кусаться.

Только что мы схематически описали некие общие свойства всех кошек в целом. Подобное описание характерных свойств и действий какого-либо объекта называется классом. То есть мы описали здесь класс: класс «кошки».

Класс - просто набор переменных и функций, которые описывают какой-либо объект; это некая схема, которая описывает объект в целом.
Объект класса – это материальное воплощение некого конкретного элемента из этого класса.

Класс Кошка - это описание ее свойств и действий, он всегда только один. А объектов класса Кошка может быть великое множество.

Для создания класса надо написать ключевое слово class и затем указать имя этого класса. Создадим класс Кошки с именем cat:

class Cat:

Затем нам нужно указать действия, которые будет способен совершать наш класс. Такие действия реализуются в виде функций, которые описываются внутри класса. Функции, описанные внутри класса, называются *методами*.

```
class Cat:

    def purr(self):
        print("Муррр!")

    def hiss (self):
        print("!Шшиш!")

    def scrabble(self):
        print("Цап-царап!")
```

Каждая из трех описанных функций имеет единственный параметр метр self.

Классам нужен способ, чтобы ссылаться на самих себя. В методах класса первый параметр функции по соглашению именуют self, и он представляет собой ссылку на сам объект этого класса. Помещать этот параметр нужно в каждую функцию, чтобы иметь возможность вызвать ее на текущем объекте. Таким образом, параметр self заменяет идентификатор объекта. Все наши кошки (объект cat) умеют мурлыкать (имеют метод purr). Теперь мы хотим, что бы замурлыкал наш конкретный объект Кошка, созданный на основе класса cat. Допустим, мы в Python напишем следующую команду:

```
Cat.purr()
```

Если мы запустим на выполнение эту команду, то станут мурлыкать сразу все кошки и коты на свете. А если мы воспользуемся командой:

```
self .purr ()
```

то мурлыкать станет только та кошка, на которую укажет параметр self, т. е. именно наша.

Давайте теперь в классе Кошки зададим им ряд свойств и, в частности: цвет шерсти, цвет глаз, кличку. А также зададим статический атрибут наименование класса - Name _ Class.

Мы можем создать переменную и занести в нее наименование класса. А также в абсолютно любом классе нужно определить функцию _ ini t _ () . Эта функция вызывается всегда, когда мы создаем реальный объект на основе нашего класса.

Итак, задаем нашим кошкам наименование класса - Name _ Class, и три свойства: цвет шерсти - wool _ color, цвет глаз - eyes _ color, кличку - name

```
class Cat:
Name Class = “Кошки”

# Действия, которые надо выполнять при создании объекта "Кошка"
def __init__ (self, wool_color, eyes_color, name):
        self.wool -color = wool -color
        self.eyes_color = eyes_color
        self.name = name
```

Программный код описания класса Кошка теперь будет выглядеть следующим образом:

```
class Cat:

    name_class = "Кошки"

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


    def purr(self):
        print("Муррр!")

    def hiss (self):
        print("!Шшиш!")

    def scrabble(self):
        print("Цап-царап!")
```

Объекты

Мы создали класс Кошки. Теперь давайте создадим на основе этого класса реальный объект - кошку:

```
my_cat = Cat('Цвет шерсти', 'Цвет глаз', 'Кличка')
```

В этой строке мы создаем переменную my_cat, а затем присваиваем ей объект класса cat. Выглядит это все так, как вызов некоторой функции cat (...). На самом деле так оно и есть. Этой записью мы вызываем метод _ ini t _ () класса cat. Функция _ ini t _ () в нашем классе принимает четыре аргумента: сам объект класса - self, который указывать не надо, а также еще три аргумента, которые затем становятся атрибутами нашей кошки. Для этой кошки зададим следующие атрибуты или свойства: белую шерсть, зеленые глаза и кличку Мурка.

```
my_cat = Cat('Белая', 'Зеленые', 'Мурка')

print("Наименование класса - ", my_cat.Name_Class)
print ( "Вот наша кошка:")
print("Цвет шерсти- ", my_cat.wool_color)
print("Цвет глаз- ", my_cat.eyes_color)
print ("Кличка- ", my_cat.name)
```

Атрибуты кошки можно менять. Например, давайте сменим кличку нашей кошки и поменяем ее цвет (сделаем кота Ваську черного цвета).

```
my_cat = Cat("Серая", "Оранжевые", "Плюшка")

print("Наименование класса - ", my_cat.name_class)
print("Вот наша кошка:")
print("Цвет шерсти- ", my_cat.wool_color)
print("Цвет глаз- ", my_cat.eyes_color)
print ("Кличка- ", my_cat.name)
```

Попросим нашего кота Ваську мяукнуть. Для этого добавим в программу всего одну строку:

my_ cat.purr ( )

Создание классов и объектов на примере автомобиля

```
class Car(object):
# Наименование класса
Name class = "Автомобиль«

def init (self, brand, weight, power):
self.brand = brand # Марка, модель автомобиля
self.weight = weight # Вес автомобиля
self.power = power # Мощность двигателя

# Метод двигаться прямо

def drive(self):
# Здесь команды двигаться прямо
print("Поехали, двигаемся прямо!")

# Метод повернуть направо

def righ(self):
# Здесь команды повернуть руль направо
print("Едем, поворачиваем руль направо!")

# Метод повернуть налево

def left (self):
# Здесь команды повернуть руль налево
print. ( "Едем, поворачиваем руль налево!")

# Метод тормозить

def brake (self):
# Здесь команды нажатия на педаль тормоза
print("Стоп, активируем тормоз")

# Метод подать звуковой сигнал

def Ьeep (self):
# Здесь команды подачи звукового сигнала
print("Подан звуковой сигнал")
```

В приведенном примере мы создали класс Автомобиль (car) и добавили в него три атрибута и пять методов. Атрибутами здесь являются:

self.brand = brand
self.weight = weight
self.power = power

Теперь на основе этого класса создадим объект - автомобиль с именем MyCar и атрибутами.

MyCar = Car('Мерседес', 1200, 250)
priht('Параметры автомобиля, созданного из класса- , MyCar.Narne_class)
print('Марка (модель)- ', MyCar.brand)
print('Вес (кг)- ', MyCar.weight)
print('Мощность двигателя (лс)- ', MyCar.power)

Теперь испытаем созданные в этом классе методы и заставим автомобиль двигаться, т. е. обратимся к соответствующим методам созданного нами объекта MyCar .

MyCar. drive ()
MyCar. righ ()
MyCar.drive ()
MyCar. left ()
MyCar .drive ()
MyCar. Ьеер ( )
MyCar. brake ()

Программные модули

Любой файл с расширением py является модулем. Зачем они нужны? Многие программисты создают приложения с полезными функциями и классами. Другие программисты могут подключать эти сторонние модули и использовать все имеющиеся в них функции и классы, тем самым упрощая себе работу. Например, вам не нужно тратить время и писать свой программный код для работы с матрицами. Достаточно подключить модуль numpy и использовать его функции и классы. Модуль math предоставляет множество методов для работы с числами: синусы, косинусы, переводы градусов в радианы и прочее, и прочее.

Установка модуля

pip install [название_модуля]

Например, для импорта модуля, позволяющего работать с математическими функциями, надо написать:

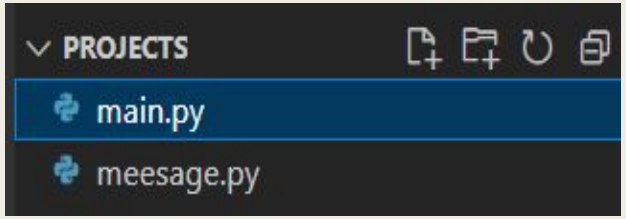
Подключение и использование модуля

Сторонний модуль подключается достаточно просто:

import [название_модуля]

Как использовать модули

Работать с кодом на тысячи строк намного проще, если он разбит на несколько модулей. В таком случае обычно работают с главным файлом, а отдельные функции помещают в разные модули. Для создания модуля необходимо создать собственно файл с расширением *.py, который будет представлять модуль.



Соответственно модуль будет называться message, в нем будет лежать одна функция

```
sms = "Tururu"

def print_message(text):
    print(f"Message: {text}")
```

В основном файле программы используем данный модуль:

```
import message

print(message.sms)
message.print_message("I want to eat")
```

```
Tururu
Message: I want to eat
```

Подключение функциональности модуля в глобальное пространство имен

Другой вариант настройки предполагает импорт с помощью ключевого слова from:

```
from message import print_message

print_message("Cucumber")
```

Установка псевдонимов

При импорте модуля и его функциональности мы можем установить для них псевдонимы. Для этого применяется ключевое слово as, после которого указывается псевдоним.

```
import message as mes

print(mes.sms)
mes.print_message("Cucumber")
```

Инкапсуляция, атрибуты и свойства

По умолчанию атрибуты в классах являются общедоступными, а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.

```
class Person:
    def __init__(self, name):
        self.name = name
        self.age = 1

    def display_info(self):
        print(f"Имя: {self.name}\tВозраст: {self.age}")


tom = Person("Tom")
tom.name = "Человек-паук"
tom.age = -129
tom.display_info()
```

В данном случае можно указать некорректное значиние. Подобное поведение нежелательно, поэтому встает вопрос о контроле за доступом к атрибутам объекта. С данной проблемой тесно связано понятие инкапсуляции. Она предотвращает прямой доступ к атрибутам объект из вызывающего кода. В языке Python скрыть атрибуты класса можно сделав их приватными или закрытыми и ограничив доступ к ним через специальные методы, которые еще называются *свойствами*.

```python
class Person:
    def __init__(self, name):
        self.__name = name
        self.__age = 1

    def set_age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

    def get_age(self):
        return self.__age

    def get_name(self):
        return self.__name

    def display_info(self):
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")


tom = Person("Tom")
tom.display_info()
tom.set_age(-3486)
tom.set_age(25)
tom.display_info()
```

Для создания приватного атрибута в начале его наименования ставится двойной прочерк: self.__name.
К такому атрибуту мы сможем обратиться только из того же класса. Но не сможем обратиться вне этого класса. Например, присвоение значения этому атрибуту ничего не даст: tom.__age = 43.
Потому что в данном случае просто определяется динамически новый атрибут __age, но это он не имеет ничего общего с атрибутом self.__age. А попытка получить его значение приведет к ошибке выполнения (если ранее не была определена переменная __age): print(tom.__age). Однако все же нам может потребоваться устанавливать возраст пользователя из вне. Для этого создаются свойства. Используя одно свойство, мы можем получить значение атрибута: def get_age(self):
Данный метод еще часто называют геттер или аксессор.
Для изменения возраста определено другое свойство: def set_age(self, age):
Данный метод еще называют сеттер или мьютейтор (mutator). Здесь мы уже можем решить в зависимости от условий, надо ли изменять возраст

Аннотации свойств

Выше мы рассмотрели, как создавать свойства. Но Python имеет также еще один - более элегантный способ определения свойств. Этот способ предполагает использование аннотаций, которые предваряются символом @.

Для создания свойства-геттера над свойством ставится аннотация **@property.**

Для создания свойства-сеттера над свойством устанавливается аннотация **имя_свойства_геттера.setter.**

```python
class Person:
    def __init__(self, name):
        self.__name = name
        self.__age = 1

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

    @age.setter
    def age(self, age):
        if 1 < age < 110:
            self.__age = age
        else:
            print("Недопустимый возраст")

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

    def display_info(self):
        print(f"Имя: {self.__name}\tВозраст: {self.__age}")
```

Во-первых, стоит обратить внимание, что свойство-сеттер определяется после свойства-геттера.Во-вторых, и сеттер, и геттер называются одинаково - age. И поскольку геттер называется age, то над сеттером устанавливается аннотация @age.setter. После этого, что к геттеру, что к сеттеру, мы обращаемся через выражение tom.age.

Наследование

Ключевыми понятиями наследования являются подкласс и суперкласс. Подкласс наследует от суперкласса все публичные атрибуты и методы. Суперкласс еще называется базовым (base class) или родительским (parent class), а подкласс - производным (derived class) или дочерним (child class).

Например, у нас есть класс Person, который представляет человека

```python
class Person:

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

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

    def display_info(self):
        print(f"Name: {self.__name} ")
```

```python
class Employee:

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

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

    def display_info(self):
        print(f"Name: {self.__name} ")

    def work(self):
        print(f"{self.name} works")
```

Предположим, нам необходим класс работника, который работает на некотором предприятии. Мы могли бы создать с нуля новый класс, к примеру, класс Employee. Однако класс Employee может иметь те же атрибуты и методы, что и класс Person, так как работник - это человек. Но чтобы не дублировать функционал одного класса в другом, в данном случае лучше применить наследование.

Итак, унаследуем класс Employee от класса Person:

```
class Person:

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

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

    def display_info(self):
        print(f"Name: {self.__name} ")

class Employee(Person):

    def work(self):
        print(f"{self.name} works")

tom = Employee("Tom")
print(tom.name)
tom.display_info()
tom.work()
```

Класс Employee полностью перенимает функционал класса Person, лишь добавляя метод work(). Соответственно при создании объекта Employee мы можем использовать унаследованный от Person конструктор:

tom = Employee("Tom").

И также можно обращаться к унаследованным атрибутам/свойствам и методам:
print(tom.name)

tom.display_info().

Однако, стоит обратить внимание, что для Employee НЕ доступны закрытые атрибуты типа __name. Например, мы НЕ можем в методе work обратиться к приватному атрибуту self.__name:

def work(self):
print(f"{self.__name} works")

Использование архитектурных паттернов
Django

Фреймворк Django реализует архитектурный паттерн Model-View-Template или сокращенно MVT, который по факту является модификацией распространённого в веб-программировании паттерна MVC (Model-View-Controller).

Основные элементы паттерна:

- **URL dispatcher**: при получение запроса на основании запрошенного адреса URL определяет, какой ресурс должен обрабатывать данный запрос.

- **View**: получает запрос, обрабатывает его и отправляет в ответ пользователю некоторый ответ. Если для обработки запроса необходимо обращение к модели и базе данных, то View взаимодействует с ними. Для создания ответа может применять Template или шаблоны. В архитектуре MVC этому компоненту соответствуют контроллеры (но не представления).

- **Model**: описывает данные, используемые в приложении. Отдельные классы, как правило, соответствуют таблицам в базе данных.

- **Template**: представляет логику представления в виде сгенерированной разметки html. В MVC этому компоненту соответствует View, то есть представления.

Когда к приложению приходит запрос, то URL dispatcher определяет, с каким ресурсом сопоставляется данный запрос и передает этот запрос выбранному ресурсу. Ресурс фактически представляет функцию или View, который получает запрос и определенным образом обрабатывает его. В процессе обработки View может обращаться к моделям и базе данных, получать из нее данные, или, наоборот, сохранять в нее данные. Результат обработки запроса отправляется обратно, и этот результат пользователь видит в своем браузере. Как правило, результат обработки запроса представляет сгенерированный html-код, для генерации которого применяются шаблоны (Template).

Какие паттерны применяются:

Абстрактная фабрика
Прототип
Компоновщик
Итератор
Декоратор
Одиночка
Фасад
Адаптер
Команда
Наблюдатель
Стратегия
Шаблонный метод
Состояние
Посредник
Цепочка

Общая организация Django-приложения

- urls.py
  - содержат routes (роуты)
  - задают соответствие между урлами и обработчиками запросов (view)
- views.py
  - содержит view (вид, вьюха)
  - на вход view получает объект представляющий собой запрос
  - на выход отдают объект-ответ
  - объект ответ часто включет в себя сгенерированный с помощью шаблонов (templates) html
- шаблоны (templates)
  - набор html-подобного текста с вкраплениями инструкций на языке Django-шаблонизатора
  - позволяют леко генерировать html по данным
- models.py
  - содержит models (модели)
  - специальные классы описывающие элементы хранимые в базе данных
  - на основе их ORM генерирует методы работы с базой - через них можно искать, создавать, изменять, удалять объекты в базе
- forms.py
  - содержит forms (формы)
  - по декларативному описанию умеют создавать html-формы, а также обрабатывать данные от них на стороне сервера.

База данных

- Перед использованием необходимо определить в настройках базу данных.
- По-умолчанию выбрана SQLite (всё хранится в файле) и с ней проще всего начать работать, если нужно быстро попробовать Django.
- Для более менее серьёзных задач нужно выбирать более подходящую базу данных с нужными возможностями (рекомендуется попробовать PostgreSQL).
- Лучше сразу выбрать ту базу, с которой будете дальше работать, т.к. переход с одной БД на другую может быть нетривиальным.

- **Сервер**
- В общем случае ещё нужно проводить дополнительные настройки сервера и взаимодействия приложения с отдельным веб-сервером.
- Для упрощения разработки в Django встроен простой веб-сервер.
- Он предназначен только для разработки - как самостоятельный веб-сервер он сильно ограничен и использовать его для чего-то серьёзного строго не рекомендуется:
    - мало возможностей
    - не предназначен для нагрузки
    - может быть более уязвимым с точки зрения безопасности

Проекты и приложения

- Выше мы создали простейший пустой проект (project) - по сути, в нём пока есть только конфигурация.
- Проект (в терминах Django) - совокупность конфигурации и множества приложений (app, application) составляющих отдельный веб-сайт.
- Приложение - часть проекта предназначенная для выполнения определённой выделенной цели.
- Проект состоит из приложений.
- Приложение может быть в разных проектах (переиспользоваться).

Структура приложений на Django

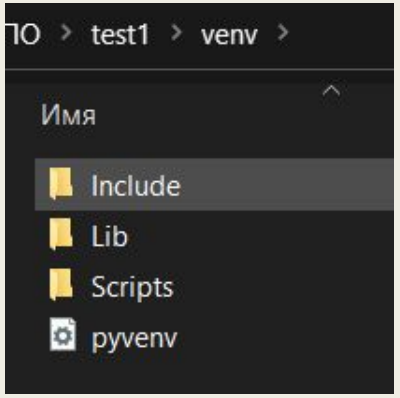
Прежде чем создавать проект на Django, нам необходимо создать окружение. Для этого создаем папку, через нее заходим в cmd. Далее в командной строке пишем :

```
python -m venv venv
```

Далее нам необходимо активировать окружение командой:

```
.\venv\Scripts\activate
```

И если мы все сделали правильно, то в нашей папке должна появится папка venv с нашим окружение.

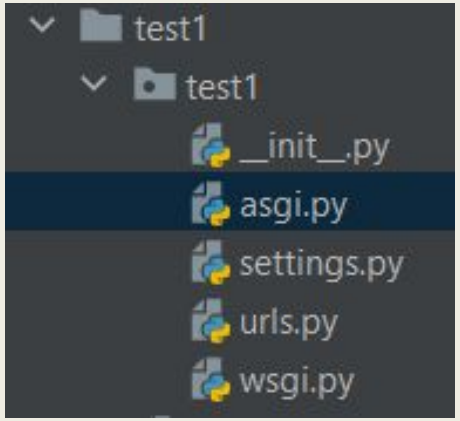


После этого необходимо установить django при помощи известной нам уже команды pip install django.

Установка и создание заготовки проекта

При условии успешно установленного Django создать заготовку для нового проекта можно следующей консольной командой:
django-admin startproject [name]

Результат этого будет выглядеть как-то так:



- test1 (внешний) - корневая директория проекта, её название не важно (можно переименовать)
- manage.py - вспомогательный скрипт, позволяющий взаимодействовать с и управлять созданным проектом
- test1 (вложенный) - директория, которая является Python-пакетом вашего проекта (например, используется как корневая при импортах и т.п.)
- __init__.py - обычный пустой файл (служит для задания Python-пакета)
- settings.py - настройки проекта
- wsgi.py - входная точка для веб-серверов поддерживающих wsgi для работы с проектом.

Запустим встроенный сервер можно следующей командой:

python manage.py runserver

Здесь:

- 0.0.0.0 - ip-адрес (должен быть на текущей машине) по которому будет доступен сервер.
- 127.0.0.1 - только локально
- 0.0.0.0 - все публичные ip-адреса текущего сервера
- xxx.xxx.xxx.xxx - какой-то определённый адрес
- 8000 - номер порта по которому будет происходить соеднинение и работа с сервером.

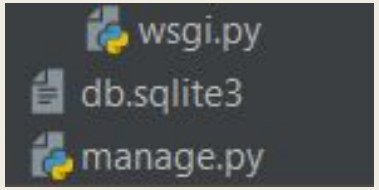
```
(venv) PS C:\srpo\test1> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not w
Run 'python manage.py migrate' to apply them.
January 17, 2023 - 17:14:21
Django version 4.1.5, using settings 'test1.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Щелкните левой кнопкой мыши по ссылке: http://127.0.0.1:8000/

После этого на вашем компьютере откроется веб-браузер и в него будет загружена веб-страница поздравления с успешной установкой Django.

Остановите локальный веб-сервер нажатием комбинации клавиш Ctrl + C в терминале.

В папке также имеется файл db.sqlite3. Как уже отмечалось ранее, это файл с базой данных SQLite.



SQLite - компактная встраиваемая реляционная база данных, исходный код которой передан в общественное достояние. Она является чисто реляционной базой данных. Слово «встраиваемая» означает, что SQLite не использует парадигму «клиентсервер». То есть движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а предоставляет библиотеку, с которой программа компонуется, а движок становится составной частью программы. При этом в качестве протокола обмена используются вызовы функций (API) библиотеки SQLite.

Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа.

В Django база данных SQLite создается автоматически (по умолчанию), формирование именно этой базы данных прописывается в файле конфигурации проекта. Если снова открыть файл settings.py, то в нем можно увидеть следующие строки:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
```

Именно здесь можно переопределить базу данных, с которой будет работать приложение. Чтобы использовать другие системы управления базами данных (СУБд), необходимо установить соответствующий пакет.

СУБД	Пакет	Команда установки
PostgreSQL	psycopg2	pip install psycopg2
MySQL	mysql-python	pip install mysql-python
Oracle	cx_Oracle	pip install cx_Oracle


кажетс
я
конец
лекции