Учреждение образования «Гомельский государственный университет имени Франциска Скорины»

Факультет физики и информационных технологий

Кафедра радиофизики и электроники

СОГЛАСОВАНО
Заведующий кафедрой
радиофизики и электроники
А.С.Руденков
_____ 2023 г.

СОГЛАСОВАНО
Декан факультета физики
и информационных технологий
Д.Л.Коваленко
_____ 2023 г.

# ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

# ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

для специальности
1-98 01 01 Компьютерная безопасность (по направлениям)

Направление специальности
1-98 01 01-02 Компьютерная безопасность
(радиофизические методы и программно-технические средства)

Составитель:
Руденков А.С., заведующий кафедрой радиофизики и электроники, к.т.н., доцент

Рассмотрено на заседании
кафедры радиофизики и электроники
19 апреля 2023 г., протокол № 9

Рассмотрено и утверждено
на заседании научно-методического совета
24.05. 2023 г., протокол № 9

Гомель, 2023

# Учебная программа

Учреждение образования
«Гомельский государственный университет имени Франциска Скорины»

**УТВЕРЖДАЮ**
Ректор
ГГУ имени Ф.Скорины

_____ С.А.Хахомов

2

_____
(дата утверждения)

Регистрационный № УД 2021-135 /уч.

**Модуль «Программирование»:**
**ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ**

Учебная программа учреждения высшего образования
по учебной дисциплине для специальности
1-98 01 01 Компьютерная безопасность (по направлениям)
Направления специальности
1-98 01 01-02 Компьютерная безопасность
(радиофизические методы и программно-технические средства)

2021

Учебная программа составлена на основе образовательного стандарта ОСВО 1-98 01 01-02-2013 и учебного плана учреждения высшего образования от 02.07.2021, регистрационный № Р 98-01-21/УП.

**СОСТАВИТЕЛЬ:**

А.С. Руденков, доцент кафедры радиофизики и электроники учреждения образования «Гомельский государственный университет имени Франциска Скорины», кандидат технических наук, доцент

**РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:**

Кафедрой оптики
(протокол № 11 от 22.06.2021);

Научно-методическим советом ГГУ имени Ф. Скорины
(протокол № 9 от 28.07.2021)

Рецензенты:

1. Трохова П.А. – заведующий кафедрой "Информатика" ГГТУ имени П.О. Сухого" к.т.н., доцент
2. Бычков П.В. – доцент кафедры автоматизированных систем обработки информатики ГГУ имени Ф. Скорины, к.ф.-м.н.,

**РЕЦЕНЗИЯ**

на учебную программу по дисциплине
«Основы объектно-ориентированного программирования» для студентов
специальности 1-98 01 01-02 «Компьютерная безопасность
(радиофизические методы и программно-технические средства)»

Представленная на рецензию учебная программа модуля государственного компонента «Программирование» «Основы объектно-ориентированного программирования» составлена доцентом кафедры радиофизики и электроники Руденковым Александром Сергеевичем для специальности 1-98 01 01-02 «Компьютерная безопасность (радиофизические методы и программно-технические средства)».

В пояснительной записке учебной программы сформулированы цели и задачи учебной дисциплины, имеются указания, что должен знать, уметь и владеть студент при изучении курса «Основы объектно-ориентированного программирования».

В содержании учебного материала приведено достаточное количество разделов и тем для полноценного усвоения учебной дисциплины «Основы объектно-ориентированного программирования». Изучение базовых концепций объектно-ориентированного программирования, сравнительный анализ реализации и использования объектно-ориентированного подхода в различных высокоуровневых языках программирования и фреймворках обеспечит разностороннюю подготовку, необходимую для выбора языка и инструментов разработки в зависимости от поставленных проектных задач.

В учебно-методической карте учебной программы четко отражено распределение аудиторных часов между разделами и темами, отведенных на изучение данной дисциплины.

В программе представлен перечень лабораторных работ, тематика которых логически согласована с содержанием дисциплины. Полученные навыки в ходе их выполнения поспособствуют более качественному усвоению дисциплины «Основы объектно-ориентированного программирования» и формированию практических навыков разработки графических и веб-приложений. Приводится достаточный перечень рекомендуемой основной и дополнительной литературы.

В целом, учебная программа по дисциплине «Основы объектно-ориентированного программирования» соответствует предъявляемым к ней требованиям, образовательному стандарту, учебному плану и рекомендуется в качестве учебной программы. Ее внедрение в учебный процесс будет являться основой подготовки для студентов обучающихся по специальности 1-98 01 01-02 «Компьютерная безопасность (радиофизические методы и программно-технические средства)».

Заведующий кафедрой «Информатика»
УО «Гомельский государственный
технический университет
имени П.О. Сухого»,
к.т.н., доцент Трохова Т.А.

Отдел кадров

Подпись Трохова Т.А. удостоверяю
Инспектор по кадрам

**РЕЦЕНЗИЯ**

на учебную программу по дисциплине
«Основы объектно-ориентированного программирования» для студентов специальности 1-98 01 01-02 «Компьютерная безопасность (радиофизические методы и программно-технические средства)»

Предоставленная на рецензию учебная программа «Основы объектно-ориентированного программирования» составлена Руденковым Александром Сергеевичем – доцентом кафедры радиофизики и электроники Учреждения образования «Гомельский государственный университет имени Франциска Скорины».

Учебная программа дисциплины «Основы объектно-ориентированного программирования» относится к модулю государственного компонента «Программирование», изложена на 15 страницах и содержит пояснительную записку, содержание учебного материала из 16 тем, учебно-методическую карту, 16 лабораторных работ, основную и дополнительную рекомендуемую литературу.

«Основы объектно-ориентированного программирования» является учебной дисциплиной, содержащей основные положения и принципы методологии объектно-ориентированного программирования. Знания, умения и навыки, приобретенные в ходе изучения дисциплины способствуют приобретению базовых профессиональных навыков таких, как: использовать принципы объектно-ориентированного программирования и проектирования в процессе создания приложений; свободно переходить из одной объектно-ориентированной платформы на другую.

В учебно-методической карте учебной программы подробно расписано распределение аудиторных часов между разделами и темами, отведенных на изучение данной дисциплины.

Перечень лабораторных работ программы охватывает основные разделы учебного материала и способствует закреплению навыков и практических знаний студентов. Также приводится достаточный перечень рекомендуемой основной и дополнительной литературы для освоения данной дисциплины.

Учебная программа оригинальна и не содержит сведений, дублирующих учебный материал дисциплины «Основы и методологии программирования».

Все предъявляемые требования к составлению и оформлению учебных программ выполнены. Считаю, что предлагаемая учебная программа дисциплины «Основы объектно-ориентированного программирования» соответствует образовательному стандарту и учебному плану специальности 1-98 01 01-02 «Компьютерная безопасность (радиофизические методы и программно-технические средства)» и может быть рекомендована к утверждению.

Рецензент
доцент кафедры АСОИ
ГГУ имени Ф. Скорины
к.ф.-м.н. П.В. Бычков

Подпіс В. Бочкова
ЗАВЯРАЮ
Начальнік аддзела кадраў
установы адукацыі "Гомельскі дзяржаўны
ўніверсітэт імя Францыска Скарыны"

# ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Учебная дисциплина модуля государственного компонента «Программирование» «Основы объектно-ориентированного программирования» является основой для усвоения материала специальных дисциплин направления специальности 1-98 01 01-02 «Компьютерная безопасность (радиофизические методы и программно-технические средства)», определяет уровень профессиональной подготовки студентов.

**Цель учебной дисциплины:** овладение студентами основами применения объектно-ориентированного программирования и проектирования в современных языках программирования.

**Задачи изучения дисциплины:**

- освоение принципов объектно-ориентированного программирования;
- изучение свойств наследования, полиморфизма и инкапсуляции;
- освоение принципов проектирования классов, модулей и пакетов;
- освоение принципов использования шаблонов (паттернов) проектирования.

Перечень дисциплин, усвоение которых необходимо для изучения данной дисциплины: «Основы и методологии программирования».

В результате изучения учебной дисциплины «Основы объектно-ориентированного программирования» у обучающихся формируются следующие компетенции:

*универсальные компетенции:*

УК-1. Владеть основами исследовательской деятельности, осуществлять поиск, анализ и синтез информации;

УК-2. Решать стандартные задачи профессиональной деятельности на основе применения информационно-коммуникационных технологий;

УК-5. Быть способным к саморазвитию и совершенствованию в профессиональной деятельности;

УК-6. Проявлять инициативу и адаптироваться к изменениям в профессиональной деятельности.

*базовые профессиональные компетенции:*

БПК-2. Строить, анализировать и тестировать алгоритмы и программы решения типовых задач обработки информации с использованием структурного, объектно-ориентированного и иных парадигм программирования.

В результате изучения дисциплины:

*Студент должен знать:*

- основные принципы и концепции объектно-ориентированного программирования;
- свойства полиморфизма, наследования и инкапсуляции;
- принципы представления и структуризации эффективных проектов систем с использованием шаблонов проектирования;
- современное состояние одного из объектно-ориентированных языков.

*Студент должен уметь:*

- создавать программы на основе технологий использования классов с использованием современных систем объектно-ориентированного проектирования;
- свободно переходить из одной объектно-ориентированной платформы на другую;
- использовать возможности классов для разработки проектных решений;
- выбирать и использовать фреймворки с учетом поставленных задач.

*Студент должен владеть:*

- навыками практической работы в современных интегрированных средах разработки;
- методами объектно-ориентированного программирования, анализа и проектирования;
- навыками анализа исходных и выходных данных решаемых задач и формы их представления;
- навыками тестирования и отладки программ.

Дисциплина государственного компонента «Основы объектно-ориентированного программирования» изучается студентами первого курса дневной формы обучения направления специальности 1-98 01 01-02 «Компьютерная безопасность (радиофизические методы и программно-технические средства)».

На изучение учебной дисциплины отводится 204 часов (6 зачетных единиц), в том числе 96 аудиторных, из них на лекции – 32 часа, на лабораторные работы – 64 часа. Форма отчетности – экзамен во 2 семестре. Форма получения высшего образования – дневная.

# СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

## Тема 1. Объектная модель

Понятие объектной модели. Поведение и свойства объекта. Абстракция. Понятие класса и экземпляра класса. Отношение между объектами и классами. Типы связей: ассоциация, обобщение, зависимость. Агрегирование и композиция.

## Тема 2. Базовые концепции объектно-ориентированного программирования

Наследование. Подклассы и суперклассы. Иерархические структуры. Простое и множественное наследование. Интерфейсы и абстрактные классы. Полиморфизм. Перегрузка операций и методов. Инкапсуляция. Ограничение доступа к атрибутам и методам класса. Методика связывания классов.

## Тема 3. Особенности реализации классов в Python

Динамическая типизация. Область видимости и пространство имен. Инструкция class. Построение дерева атрибутов. Словари пространства имен. Порядок поиска в иерархии наследования. Связи между пространствами имен. Использование инструментов интроспекции. Абстрактные суперклассы.

## Тема 4. Методы класса, перегрузка операций

Распространенные методы перегрузки операций. Конструкторы, выражения, деструкторы. Геттеры и сеттеры. Эмуляция защиты атрибутов экземпляров. Методы отображения, выражения вызовов, сравнения и булевские проверки. Функциональные интерфейсы и код, основанный на обратных вызовах.

## Тема 5. Проектирование с использованием классов

Интерфейсы и сигнатуры вызовов. Объектно-ориентированное программирование, наследование и композиция: отношения «является» и «имеет». Делегирование: промежуточные объекты-оболочки. Обобщенные фабрики объектов. Реализация подмешиваемых классов отображения.

## Тема 6. Расширенные возможности классов

Расширение встроенных типов путем внедрения и создания подклассов. Слоты как метод ограничения для объявления атрибутов. Свойства как средства доступа к атрибутам. Статические методы и методы классов. Декораторы и метаклассы. Безопасность на основе декораторов. Кооперативная координация вызовов методов при множественном наследовании.

### Тема 7. Модули и пакеты

Принципы импортирования. Организация поиска модулей. Создание собственных модулей. Пространства имен модулей. Перегрузка модулей (reload). Импортирование пакетов. Относительный и абсолютный импорт. Скрипты настройки и метаданные.

### Тема 8. Основы исключений

Стандартный обработчик исключений. Перехват и генерация исключений. Диспетчеры контекстов. Исключения на основе классов. Реализация классов исключений. Встроенные классы исключений. Проектирование с использованием исключений.

### Тема 9. Управляемые атрибуты

Назначение управляемых атрибутов. Вычисляемые атрибуты. Реализация свойств с помощью декораторов. Использование информации состояния в дескрипторах. Связь между дескрипторами и свойствами. Использование свойств и дескрипторов для проверки достоверности.

### Тема 10. Декораторы

Использование и определение декораторов. Управление вызовами, экземплярами, функциями и классами. Реализация декораторов функций. Реализация декораторов классов. Классы-одиночки. Отслеживание объектных интерфейсов.

### Тема 11. Метаклассы

Назначение и использование метаклассов. Модель метаклассов. Объявление и реализация метаклассов. Наследование и экземпляр. Методы и перегрузка операций в методах метаклассов. Трассировка с помощью метаклассов и декораторов.

### Тема 12. Потоки, процессы и асинхронное программирование

Многопоточное и многопроцессная обработка. Синхронизация потоков. Семафоры. Взаимодействие между процессами. Пул процессов. Планирование времени вызова функций. Асинхронное получение результатов. Параллельное выполнение задач.

### Тема 13. Разработка графических интерфейсов

Событийно-ориентированное программирование: событие, обработчик события, цикл обработки событий. Инструменты для создания графических интерфейсов пользователя (GUI). Общие сведения о GUI в Python. Отслеживание событий. Элементы графического интерфейса (виджеты). Создание и конфигурирование виджета. Менеджер размещения.

## Тема 14. Веб-технологии и инструментальные средства для разработки веб-приложений

Технологии клиентского и серверного программирования. Базовые сведения о HTML: теги для представления текста, списки, таблицы, гиперссылки. Каскадные таблицы стилей (CSS). Обзор возможностей JavaScript.

## Тема 15. Основы работы в фреймворке Django

Структура приложений на Django. Представления и маршрутизация: обработка запросов пользователей. Параметры представлений. Переадресация и отправка пользователю статусных кодов. Шаблоны: создание, использование, передача данных. Использование в формах POST-запросов. Использование полей в формах Django.

## Тема 16. Модели данных в Django

Создание моделей и миграции базы данных. Типы полей. Манипуляция с данными в Django на основе CRUD. Чтение, запись, редактирование и удаление данных. Организация связей между таблицами в модели данных.

# УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА УЧЕБНОЙ ДИСЦИПЛИНЫ

Очная (дневная) форма обучения

| Номер раздела, темы, занятия | Название раздела, темы, занятия; перечень изучаемых вопросов | Количество аудиторных часов | | | | Формы контроля знаний |
|---|---|---|---|---|---|---|
| | | лекции | лабораторные занятия | практические занятия | количество часов УСР | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | **Объектная модель** | 2 | | | | |
| | 1. Понятие объектной модели. Поведение и свойства объекта.<br>2. Абстракция. Понятие класса и экземпляра класса.<br>3. Отношение между объектами и классами.<br>4. Типы связей: ассоциация, обобщение, зависимость. Агрегирование и композиция. | | | | | |
| 2 | **Базовые концепции объектно-ориентированного программирования** | 4 | 2 | | | |
| | 1. Наследование. Подклассы и суперклассы.<br>2. Иерархические структуры. Простое и множественное наследование.<br>3. Интерфейсы и абстрактные классы.<br>4. Полиморфизм. Перегрузка операций и методов.<br>5. Инкапсуляция. Ограничение доступа к атрибутам и методам класса. Методика связывания классов. | | | | | Защита отчетов по лабораторным работам |
| 3 | **Особенности реализации классов в Python** | 2 | 8 | | | |
| | 1. Динамическая типизация. Область видимости и пространство имен.<br>2. Инструкция class. Построение дерева атрибутов. Словари пространства имен.<br>3. Порядок поиска в иерархии наследования. Связи между пространствами имен. 4. Использование инструментов интроспекции.<br>5. Абстрактные суперклассы. | | | | | Защита отчетов по лабораторным работам |

| | | | | | | |
|---|---|---|---|---|---|---|
| 4 | **Методы класса, перегрузка операций** | 2 | 4 | | | |
| | 1. Распространенные методы перегрузки операций.<br>2. Конструкторы, выражения, деструкторы.<br>3. Геттеры и сеттеры. Эмуляция защиты атрибутов экземпляров.<br>4. Методы отображения, выражения вызовов, сравнения и булевские проверки.<br>5. Функциональные интерфейсы и код, основанный на обратных вызовах. | | | | | Защита отчетов по лабораторным работам |
| 5 | **Проектирование с использованием классов** | 2 | 4 | | | |
| | 1. Интерфейсы и сигнатуры вызовов.<br>2. Объектно-ориентированное программирование, наследование и композиция: отношения «является» и «имеет».<br>3. Делегирование: промежуточные объекты-оболочки.<br>4. Обобщенные фабрики объектов. Реализация подмешиваемых классов отображения. | | | | | Защита отчетов по лабораторным работам |
| 6 | **Расширенные возможности классов** | 2 | 4 | | | |
| | 1. Расширение встроенных типов путем внедрения и создания подклассов.<br>2. Слоты как метод ограничения для объявления атрибутов. Свойства как средства доступа к атрибутам.<br>3. Статические методы и методы классов. Декораторы и метаклассы.<br>4. Безопасность на основе декораторов.<br>5. Кооперативная координация вызовов методов при множественном наследовании. | | | | | Защита отчетов по лабораторным работам |
| 7 | **Модули и пакеты** | 2 | 4 | | | |
| | 1. Принципы импортирования. Организация поиска модулей.<br>2. Создание собственных модулей. Пространства имен модулей.<br>3. Перегрузка модулей (reload).<br>4. Импортирование пакетов. Относительный и абсолютный импорт.<br>5. Скрипты настройки и метаданные. | | | | | Защита отчетов по лабораторным работам |
| 8 | **Основы исключений** | 1 | 4 | | | |
| | 1. Стандартный обработчик исключений. Перехват и генерация исключений.<br>2. Диспетчеры контекстов. Исключения на основе классов.<br>3. Реализация классов исключений. Встроенные классы исключений.<br>4. Проектирование с использованием исключений. | | | | | Защита отчетов по лабораторным работам |

| | | | | | | |
|---|---|---|---|---|---|---|
| 9 | **Управляемые атрибуты** | 1 | | | | |
| | 1. Назначение управляемых атрибутов. Вычисляемые атрибуты.<br>2. Реализация свойств с помощью декораторов.<br>3. Использование информации состояния в дескрипторах.<br>4. Связь между дескрипторами и свойствами.<br>5. Использование свойств и дескрипторов для проверки достоверности. | | | | | |
| 10 | **Декораторы** | 2 | 4 | | | |
| | 1. Использование и определение декораторов.<br>2. Управление вызовами, экземплярами, функциями и классами.<br>3. Реализация декораторов функций. Реализация декораторов классов.<br>4. Классы-одиночки.<br>5. Отслеживание объектных интерфейсов. | | | | | Защита отчетов по лабораторным работам |
| 11 | **Метаклассы** | 2 | | | | |
| | 1. Назначение и использование метаклассов. Модель метаклассов.<br>2. Объявление и реализация метаклассов. Наследование и экземпляр.<br>3. Методы и перегрузка операций в методах метаклассов.<br>4. Трассировка с помощью метаклассов и декораторов. | | | | | |
| 12 | **Потоки, процессы и асинхронное программирование** | 2 | 4 | | | |
| | 1. Многопоточное и многопроцессная обработка. Синхронизация потоков.<br>2. Семафоры.<br>3. Взаимодействие между процессами. Пул процессов.<br>4. Планирование времени вызова функций.<br>5. Асинхронное получение результатов. Параллельное выполнение задач. | | | | | Защита отчетов по лабораторным работам |
| 13 | **Разработка графических интерфейсов** | 2 | 10 | | | |
| | 1. Событийно-ориентированное программирование: событие, обработчик события, цикл обработки событий.<br>2. Инструменты для создания графических интерфейсов пользователя (GUI).<br>3. Общие сведения о GUI в Python. Отслеживание событий.<br>4. Элементы графического интерфейса (виджеты).<br>5. Создание и конфигурирование виджета. Менеджер размещения. | | | | | Защита отчетов по лабораторным работам |

| | | | | | | |
|---|---|---|---|---|---|---|
| 14 | **Веб-технологии и инструментальные средства для разработки веб-приложений** | 2 | 4 | | | |
| | 1. Технологии клиентского и серверного программирования.<br>2. Базовые сведения о HTML: теги для представления текста, списки, таблицы, гиперссылки.<br>3. Каскадные таблицы стилей (CSS).<br>4. Обзор возможностей JavaScript.. | | | | | Защита отчетов по лабораторным работам |
| 15 | **Основы работы в фреймворке Django** | 2 | 8 | | | |
| | 1. Структура приложений на Django.<br>2. Представления и маршрутизация: обработка запросов пользователей.<br>3. Параметры представлений. Переадресация и отправка пользователю статусных кодов.<br>4. Шаблоны: создание, использование, передача данных.<br>5. Использование в формах POST-запросов. Использование полей в формах Django. | | | | | Защита отчетов по лабораторным работам |
| 16 | **Модели данных в Django** | 2 | 4 | | | |
| | 1. Создание моделей и миграции базы данных. Типы полей.<br>2. Манипуляция с данными в Django на основе CRUD.<br>3. Чтение, запись, редактирование и удаление данных.<br>4. Организация связей между таблицами в модели данных | | | | | Защита отчетов по лабораторным работам |
| | **ВСЕГО** | **32** | **64** | | | **экзамен** |

Доцент кафедры радиофизики и электроники А.С. Руденков

## ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

*Примерный перечень тем лабораторных работ*

1. Разработка взаимосвязей, создание простейших классов и объектов.
2. Реализация ромбовидного наследования, организация поиска по дереву атрибутов.
3. Абстрактные классы.
4. Создание интерфейсов и делегирование.
5. Разработка и тестирование собственного модуля.
6. Исследованием преимуществ использования исключений при создании экземпляров классов.
7. Декорирование классов и функций.
8. Исследование слотов и свойств для объявления и доступа к атрибутам
9. Исследование асинхронной модели программирования, измерение времени вызова методов классов.
10. Разработка приложения для визуализации математических функций с использованием пакета модулей Tkinter
11. Разработка приложения для моделирования физического процесса с использованием пакета модулей PyGame
12. Разработка HTML-страницы с использование каскадных таблиц стилей.
13. Разработка структуры проекта в фреймворке Django.
14. Организация маршрутизации и представлений.
15. Разработка форм и шаблонов
16. Разработка моделей данных и организация миграции данных.

# РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

## Основная


1. Кьоу, Д. Объектно-ориентированное программирование: учебный курс : пер. с англ. / Д. Кьоу, М. Джеанини . – М., 2005 . – 238 с.
2. Орлов, С.А. Теория и практика языков программирования: учебник / С.А. Орлов. – СПб.: Питер, 2013. – 688 с.
3. Хорев, П.Б. Объектно-ориентированное программирование: учеб. пособие для студентов вузов по направлению «Информатика и вычислительная техника» / П.Б. Хорев. – 4-е изд., стер. – М.: Академия, 2012 . – 447 с.
4. Гниденко, И. Г. Технологии и методы программирования: учебное пособие / И. Г. Гниденко, Ф. Ф. Павлов, Д. Ю. Федоров. – М.: Издательство «Юрайт», 2021. – 235 с.
5. Лафоре, Р. Объектно-ориентированное программирование в С++ : пособие для программистов : пер. с англ. / Р. Лафоре . – 4-е изд., стер . –М., 2006 . – 928 с.
6. Незнанов, А.А. Программирование и алгоритмизация: учебник: для студентов вузов по направлению «Автоматизированные технологии и производства» / А.А. Незнанов. – М.: Академия, 2010 . – 304 с.
7. Языки программирования. Python: учебно-методическое пособие: в 2 ч. / В. В. Иванченко [и др.]. – Минск: БНТУ, 2021. – Ч. 1. – 91 с.
8. Лутц, М. Изучаем Python: в 2 т. / М. Лутц. – СПб.: ООО «Диалектика», 2019. – Т.1. – 832 с.
9. Федоров, Д. Ю. Основы программирования на примере языка Python: учебное пособие / Д. Ю. Федоров. – СПб., 2016. – 176 с.
10. Чернышев, С.А. Основы программирования на Python: учебное пособие для вузов / С.А. Чернышев. – М.: Издательство «Юрайт», 2022. – 286 с.
11. Федоров, Д.Ю. Программирование на языке высокого уровня Python: учебное пособие / Д. Ю. Федоров. – М.: Издательство «Юрайт», 2021. – 3-е изд., перераб. и доп. – 210 с.
12. Постолит, А.В. Python, Django и PyCharm для начинающих. – СПб.: БХВ-Петербург, 2021. – 464 с.
13. Дронов, В.А. Django 3.0. Практика создания веб-сайтов на Python / В.А. Дронов. — СПб.: БХВ-Петербург, 2021. – 704 с.


## Дополнительная


14. Жуков, Р.А. Язык программирования Python: практикум: учебное пособие / Р.А. Жуков. – М.: ИНФРА-М, 2019. – 216 с.
15. Стивенсон, Б. Python. Сборник упражнений / Б. Стивенсон; пер. А. Ю. Гинько. – М.: ДМК Пресс, 2021. – 238 с.
16. Бизли, Д. Python. Книга рецептов / Д. Бизли, Б.К. Джонсон; пер. Б. В. Уварова. – М.: ДМК Пресс, 2019. – 648 с.

# ПРОТОКОЛ СОГЛАСОВАНИЯ УЧЕБНОЙ ПРОГРАММЫ ПО ИЗУЧАЕМОЙ УЧЕБНОЙ ДИСЦИПЛИНЕ С ДРУГИМИ ДИСЦИПЛИНАМИ СПЕЦИАЛЬНОСТИ

| Название дисциплины, с которой требуется согласование | Название кафедры | Предложения об изменениях в содержании учебной программы по изучаемой учебной дисциплине | Решение, принятое кафедрой, разработавшей учебную программу (с указанием даты и номера протокола) |
|---|---|---|---|
| Основы и методологии программирования | кафедра оптики | При изучении классов и методов классов опираться на знания, полученные на занятиях по дисциплине «Основы и методологии программирования» | протокол № ___ от _____ |

# ДОПОЛНЕНИЯ И ИЗМЕНЕНИЯ К УЧЕБНОЙ ПРОГРАММЕ ПО ИЗУЧАЕМОЙ УЧЕБНОЙ ДИСЦИПЛИНЕ

на _____/_____ учебный год

| №№ пп | Дополнения и изменения | Основание |
|---|---|---|
| | | |

Учебная программа пересмотрена и одобрена на заседании
кафедры оптики
(протокол № ____ от _____ 202_ г.)

Заведующий кафедрой оптики
к.ф.-м.н., доцент _____ В.Е. Гайшун

УТВЕРЖДАЮ
Декан факультета физики и информационных технологий
ГГУ им. Ф. Скорины
к.ф.-м.н., доцент _____ Д.Л. Коваленко

# Основы ООП

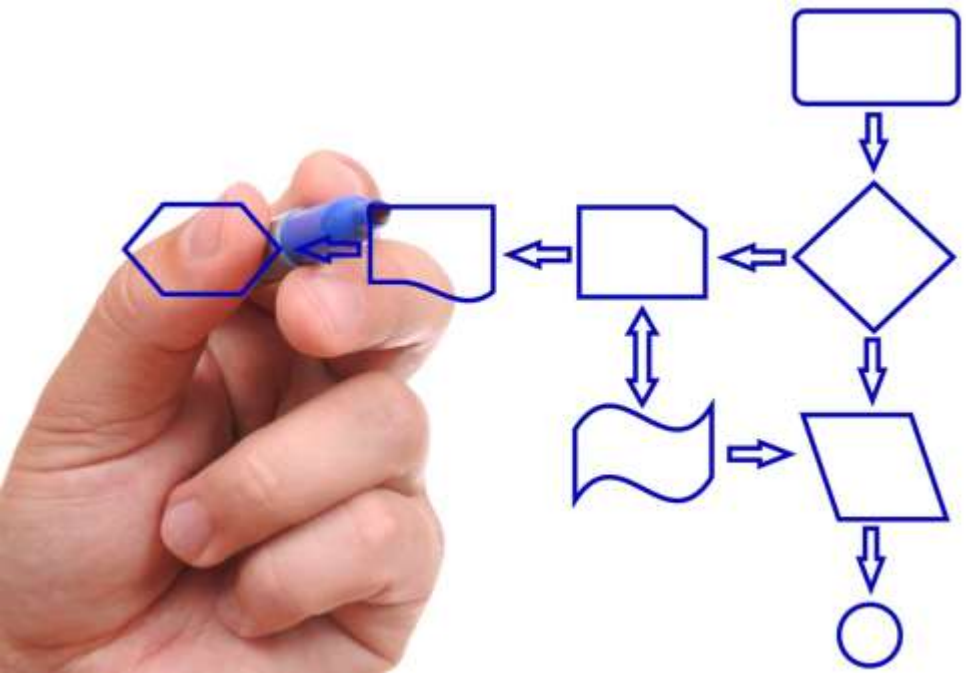
## КУРС ЛЕКЦИЙ

# Основы ООП

## Лекция 1. Объектная модель

На начальном этапе развития компьютерной техники и программирования основными являлись вычислительные задачи. Здесь центральным понятием являлся **алгоритм** – предписание выполнить точно определенную последовательность операций, которая преобразовывает входные данные в результат.

**Программа представлялась как средство реализации некоторого алгоритма.**



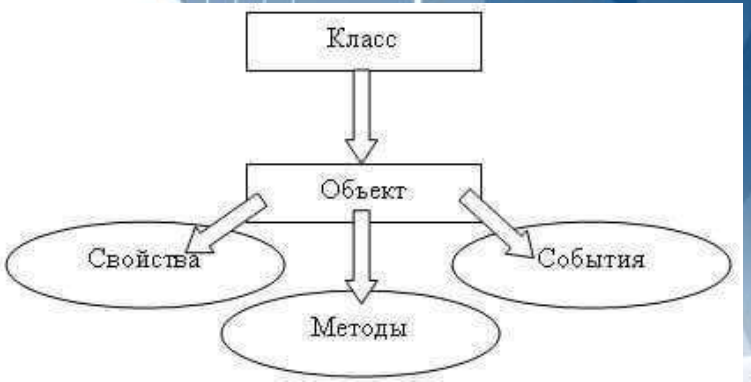
Увеличение сложности программ привело к изменению подходов в программировании. Программы приходилось разделять на все более мелкие фрагменты, которые решали конкретные подзадачи. Основой для такого разбиения стала **процедурная (функциональная) декомпозиция**.

Программа, таким образом, превратилась в совокупность процедур, каждая из которых представляет собой законченную последовательность действий, направленных на решение отдельной задачи. Отдельно выделялась главная процедура, определяющая процесс решения задачи путем вызова в определенном порядке отдельных процедур. Такой подход в методологии создания программ назвали **структурным программированием**.


Одна из основных особенностей такой методологии заключалась в том, что появилась возможность создавать **библиотеки подпрограмм** (процедур), которые можно было бы использовать повторно в различных проектах или в рамках одного проекта.

С течением времени компьютер перестал восприниматься в качестве простого вычислителя, он превратился в среду решения различных прикладных задач обработки и манипулирования данными. На первый план вышли задачи организации простого и удобного человеко-машинного взаимодействия, разработка программ с удобным графическим интерфейсом и создание автоматизированных систем управления.

В этот момент ведущим при разработке программ стал **объектно-ориентированный подход**, который основан на использовании в программах набора моделей объектов, описывающих предметную область решаемой задачи. Такие модели называют **объектными**, или **объектно-информационными**.





Человечество в своей деятельности (научной, образовательной, технологической, культурной) постоянно **создает** и **использует модели** для описания окружающего мира (макет строящегося жилищного комплекса, модель самолета, эскизы картин и т.д.). **Модели** позволяют представить в наглядной форме объекты, взаимосвязи между ними, и процессы.

**Основой** объектно-информационной модели являются **объекты**

**Объект** – это часть окружающей нас действительности, воспринимаемая человеком как единое целое. Объекты могут быть **материальными** (предметы и явления) и **нематериальными** (идеи и образы), например, стол, стул, собака, сердце – это материальные объекты; матрица, теория относительности, законы Ньютона, философское учение Платона – примеры нематериальных объектов.

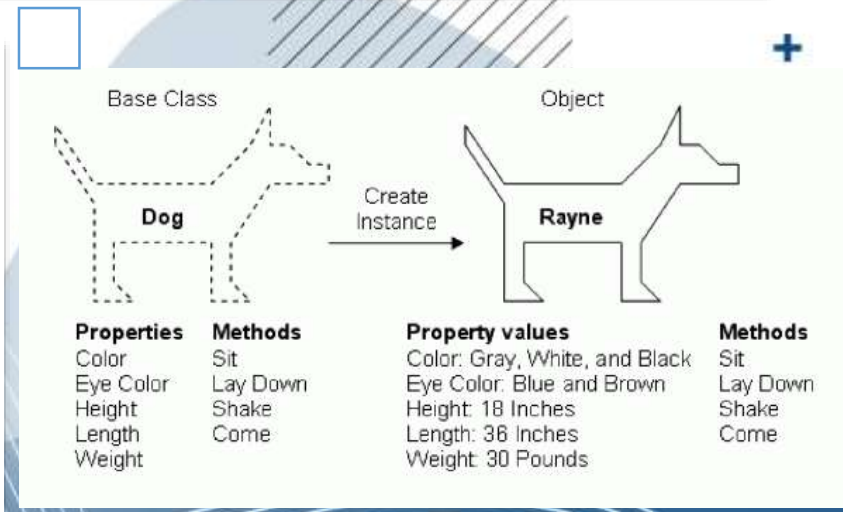
Отдельно можно выделить объекты, которые добавляются в процессе программной реализации и не имеют никакого отношения к окружающей нас реальности – вектор, динамический список, бинарное дерево, хэш-таблица и пр.

Каждый объект характеризуется множеством **свойств**. Информационная модель объекта выделяет из этого множества только некоторые свойства, существенные для решения конкретной задачи, позволяющие отделить этот объект от других. Объекты могут находиться в различных **состояниях**. Состояние объекта характеризуется перечнем всех его свойств и их текущими значениями.

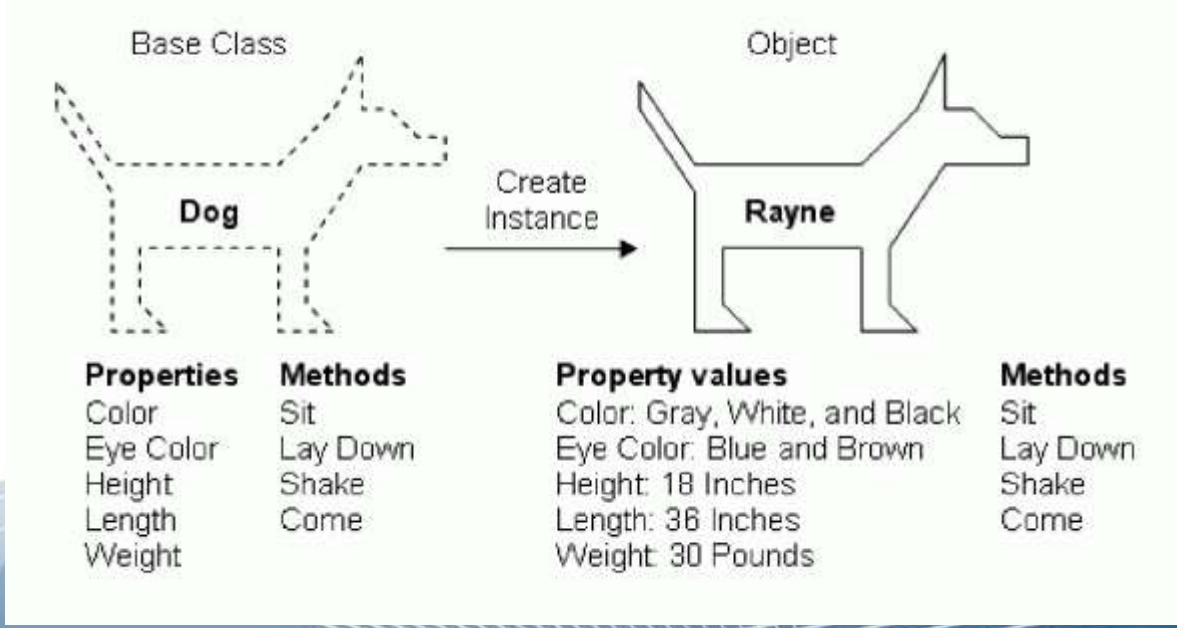
Обычно выделяется свойство (набор свойств), однозначно идентифицирующее объект во множестве объектов того же типа. Это свойство (набор свойств) называют **идентичностью**.

Как правило, объекты не остаются неизменными. Изменение состояния объекта отражается в его модели **изменением значений его свойств**

В объектно-информационной модели отражаются не только свойства, но также и **поведение объекта**. **Поведение объекта** – действия, которые могут выполняться над объектом или которые может выполнять сам объект. Именно поведение объекта определяет его переход от одного состояния в другое.



**Экземпляр класса** – это конкретный предмет или объект, а класс определяет множество объектов с одинаковым набором свойств и поведением. Класс может порождать произвольное число объектов, однако любой объект относится к строго фиксированному классу. **Класс объекта** – это его неявное свойство.



Проектирование объектной модели сводится не только к определению классов, которые описывают предметную область. **Классы** не существуют автономно – они **взаимодействуют между собой**. Поэтому в объектную модель включается также **описание связей** (отношений) между классами.

Наиболее распространенными при описании предметной области модели являются следующие три типа связей – **ассоциация**, **обобщение** и **зависимость**.

**Ассоциацией** называется структурное отношение, показывающее, что объекты одного типа связаны с объектами другого типа. Например, высказывание «студент учится в вузе» определяет ассоциацию между объектами классов «Студент» и «Вуз»

При определении ассоциации указывается, какое количество объектов каждого класса участвует в отношении. Это количество называют **кратностью ассоциации**.

Особым видом ассоциации является **агрегирование** – отношение типа «является частью» («is-part-of»), когда объект-целое состоит из нескольких объектов-частей. Например, высказывание **«группа состоит из студентов»**


Частным случаем агрегирования является **композиция** – отношение, когда время жизни частей и целого совпадают. Примером такой связи является отношение «Вуз-Факультет» – после ликвидации вуза факультеты как самостоятельные единицы существовать не могут

Агрегация «Группа-Студент» не обладает таким свойством – при распределении студентов по специализациям осуществляется переформирование групп (прежние группы упраздняются, новые группы формируются). При этом объекты-студенты не уничтожаются.

**Обобщение** – это отношение между общим классом (суперклассом, родителем) и одной или несколькими его вариациями (подклассами, потомками). Обобщение объединяет классы по их общим свойствам и поведению, что обеспечивает структурирование описания объектов.

Обобщение иногда называют отношениями типа **«является»** («is-a»), имея в виду, что одна сущность (класс «Студент-контрактник») является частным случаем другой, более общей (класс «Студент»).

Обобщение означает, что объекты класса-потомка могут использоваться всюду, где встречаются объекты класса-родителя, но не наоборот. Потомок может быть подставлен вместо родителя. При этом он наследует свойства родителя – его атрибуты и операции. Часто, хотя и не всегда, у потомков есть и свои собственные атрибуты и операции, помимо тех, что существуют у родителя.

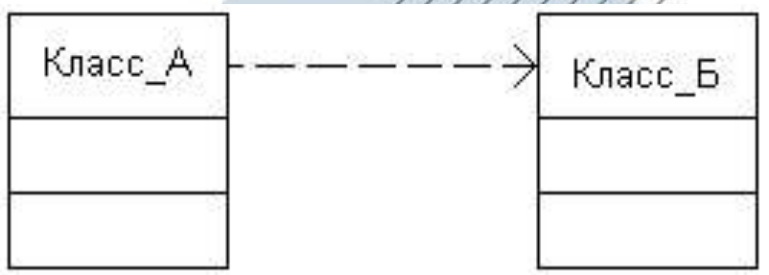
В случаях, когда класс-потомок не содержит собственных атрибутов и операций, но реализация некоторых унаследованных им методов отличается от родительских, определяется отношение типа **«является подобным»** (**«is-like-a»**).

Отношение **зависимости** – это такой тип отношения, при котором изменение в определении одного класса приводит к изменению реализации другого класса.

КВАДРАТ

Например, изменение в классе «Студент» (добавление новых методов, изменение прототипов существующих методов и пр.) может привести к изменениям в классе «Учебная группа».

Чаще всего такая связь возникает в случаях, когда классы находятся в отношении агрегации или когда объекты одного класса являются параметрами методов другого класса.



**Пример**

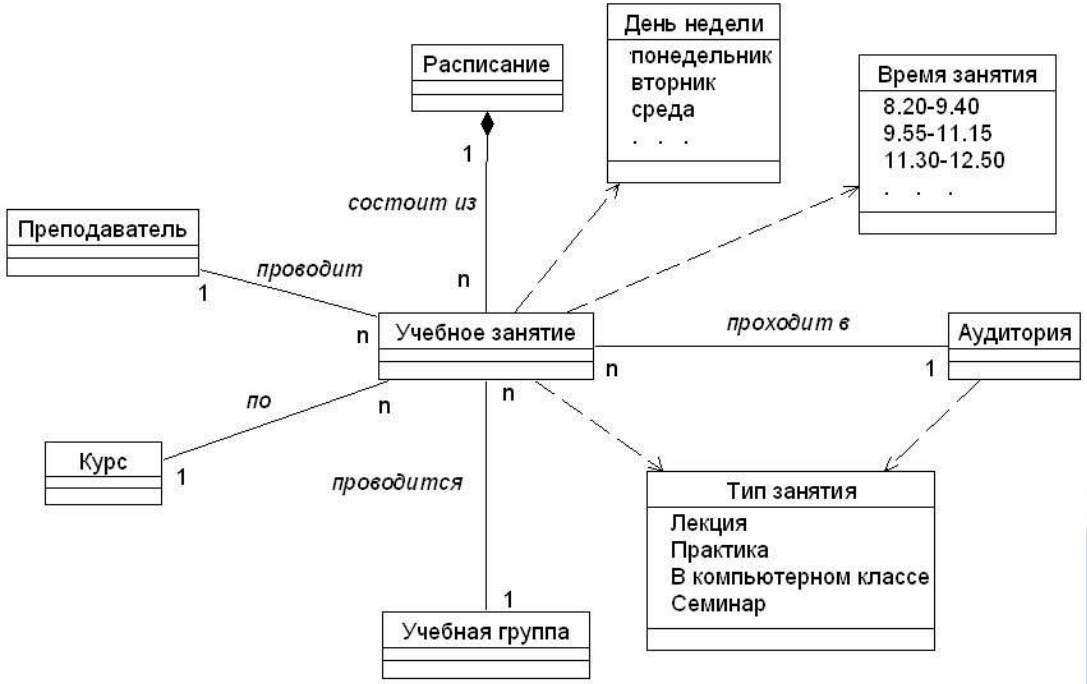
Объектная модель системы «Формирование учебного расписания»

Основные объекты системы **«Формирование учебного расписания»** определяются следующими классами:

- **«Расписание».** Его свойствами является название факультета и список занятий;
- **«Учебное занятие».** Оно задается названием предмета, типом занятия (лекция, практика, консультация и т.д.), преподавателем, который его проводит, учебной группой, днем недели и временем проведения, аудиторией;
- **«Преподаватель»**, проводящий занятие;
- **«Предмет»**, по которому проводятся занятия;
- **«Учебная группа»**, для которой проводится занятие;
- **«Аудитория»**, в которой проводится занятие.

**Пример**

Объектная модель системы «Формирование учебного расписания»




Расписание содержит в себе информацию обо всех учебных занятиях, т.е. определяет отношение типа «часть/целое» с классом «Учебное занятие». Данное отношение является отношением композиции, поскольку учебные занятия не существуют как самостоятельные сущности без расписания.

**Пример**

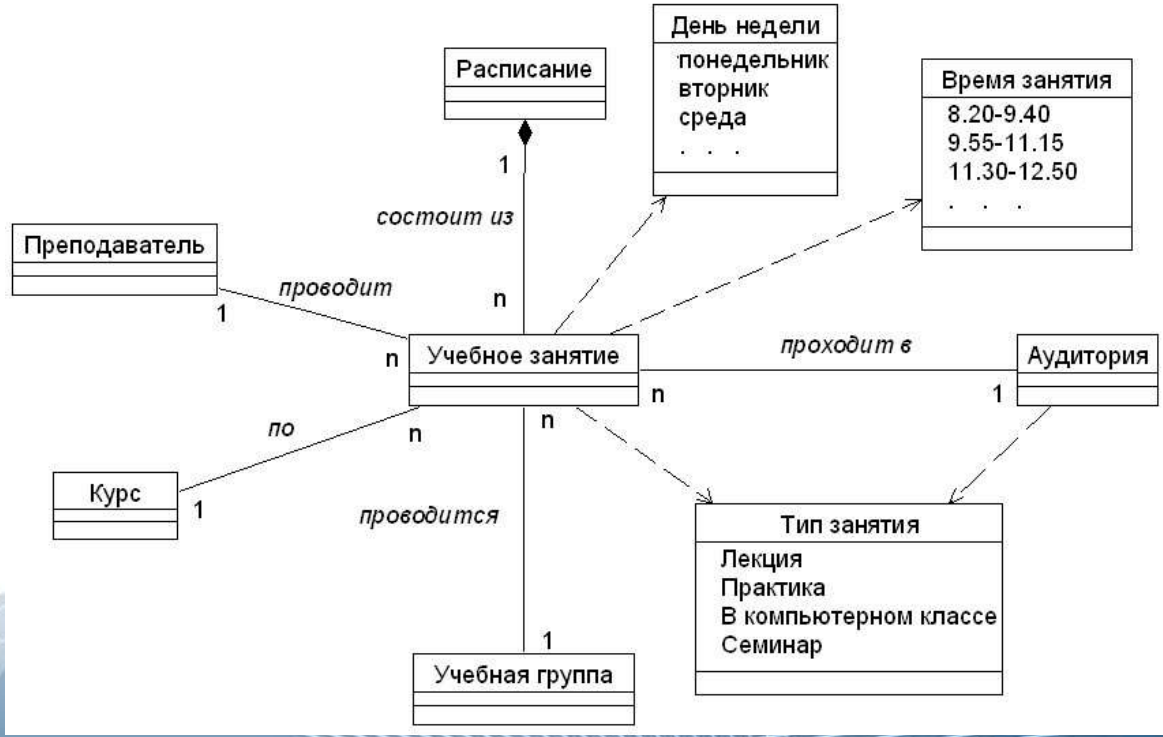
Объектная модель системы «Формирование учебного расписания»

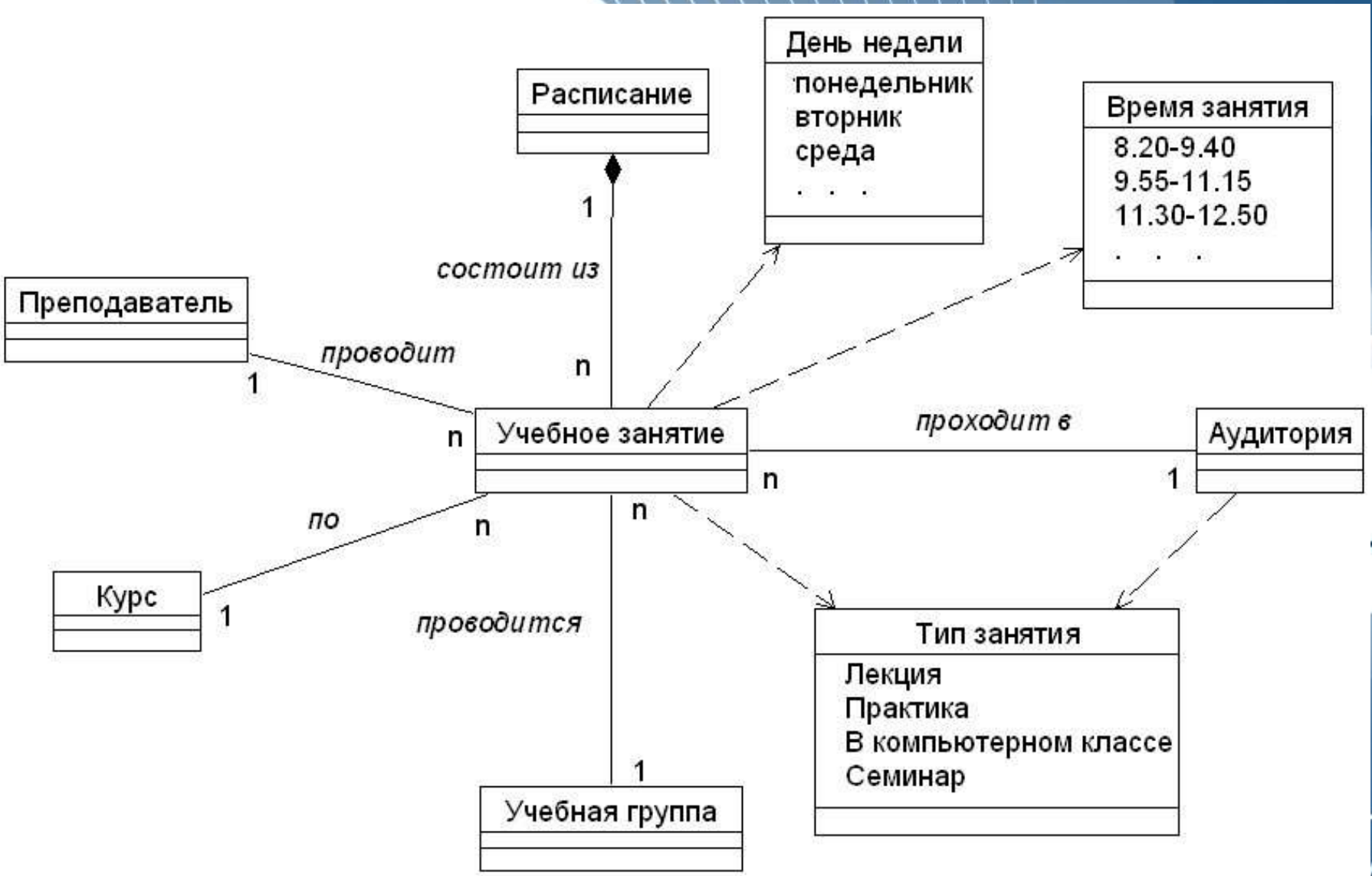
Между классом «Учебное занятие» и оставшимися классами существуют ассоциации. Одно учебное занятие проводит один преподаватель, при этом один преподавать может проводить несколько занятий в разное время. Одно учебное занятие может проводиться для студентов нескольких учебных групп, при этом каждая группа может посещать несколько занятий в разное время. Одно учебное занятие ведется по конкретному предмету, но для каждого предмета может быть несколько занятий в расписании. Каждое учебное занятие проходит только в одной аудитории, но в этой же аудитории в другое время могут проходить другие занятия.

**Пример**

Объектная модель системы «Формирование учебного расписания»

Типы занятий можно выделить в отдельный класс, который будет накладывать ограничение на аудитории, в которых может проводиться занятие (в зависимости от вместительности аудитории или наличии определенной техники). Тем самым, образуется отношение зависимости между классами «Тип занятия» и «Аудитория».




Расписание
День недели
понедельник
вторник
среда
. . .
Время занятия
8.20-9.40
9.55-11.15
11.30-12.50
. . .
1
состоит из
n
Преподаватель
проводит
1
n
Учебное занятие
проходит в
Аудитория
n
1
по
n
n
Курс
1
проводится
Тип занятия
Лекция
Практика
В компьютерном классе
Семинар
1
Учебная группа

# Заключение

Объектная модель более естественная для разработки, поскольку ориентирована на человеческое восприятие мира, а не на компьютерную реализацию

Объектная модель сложной программной системы может быть разделена на объектные модели ее подсистем. Поэтому процесс разработки носит обычно эволюционный характер, что предполагает развитие системы на базе уже разработанных небольших подсистем

Объектная модель в дальнейшем позволяет использовать выразительные возможности объектных и объектно-ориентированных языков программирования, таких как C++, C#, Java и т.д.

# Основы ООП

## Лекция 2. Основные принципы ООП

# Введение

В объектно-ориентированной технологии используется особый подход к разработке программ, основанный на использовании объектных моделей и нескольких базовых концепциях. К этим концепциям относятся **абстрагирование**, **инкапсуляция**, **полиморфизм**, **наследование**.

# Абстрагирование

Любая объектная модель содержит **описание объектов**, необходимых для работы приложения, и их **взаимосвязей**. Любой объект обладает большим количеством различных **свойств**. Каждый человек воспринимает объект по-своему, исходя из того, какие задачи приходится ему решать, работая с этим объектом. В этом случае для описания объекта выделяется некоторое количество его **характеристик**, существенных для решения задачи. К характеристикам объекта относятся его свойства, как с точки зрения его структуры, так и с точки зрения его поведения.

**Например, при приобретении велосипеда покупатель обращает внимание на:**

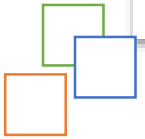
- **структурные характеристики**: возрастная группа велосипедиста (детский, подростковый, взрослый), тип велосипеда (спортивный, прогулочный, горный, шоссейный), размер колес, количество передач, материал, из которого сделан велосипед, фирма-производитель, цвет, стоимость и др.;
- **поведение**: переключение скорости, движение, торможение и др.

# Абстрагирование

Так и в программировании разработчики концентрируют свое внимание на **существенных свойствах**, необходимых для **описания объекта**, и на операциях, которые описывают его **поведение**. В этом и заключается **абстрагирование**.

При **абстрагировании** выделяются те характеристики объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют его концептуальные границы с точки зрения наблюдателя.

**Классы объектной модели** представляют собой **абстракции** сущностей предметной области задачи. Выбор правильного набора абстракций для заданной предметной области представляет собой **главную задачу** объектно-ориентированного проектирования.



**Абстракцию** множества объектов, которые имеют общий набор свойств и обладают одинаковым поведением, называют **классом**. Каждый объект в этом случае рассматривается как **экземпляр соответствующего класса**. Объекты, которые не имеют полностью одинаковых характеристик или не обладают одинаковым поведением, по определению, не могут быть отнесены к одному классу.

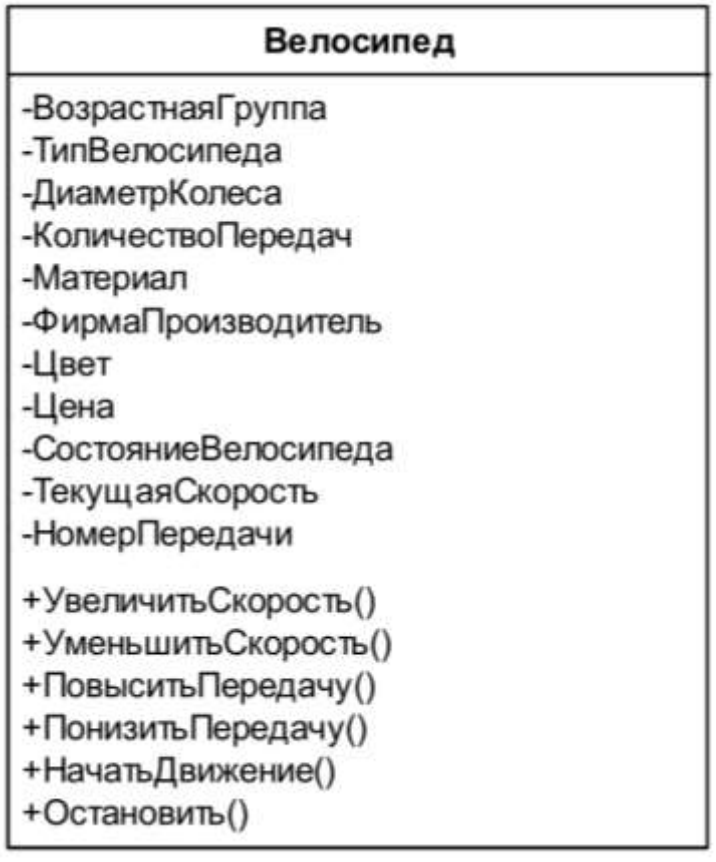
# Абстрагирование

Например, в класс **«Велосипед»** добавим характеристики, которые описывают текущее состояние велосипеда в процессе использования:

- СостояниеВелосипеда (стоит или двигается),
- ТекущаяСкорость,
- НомерПередачи.

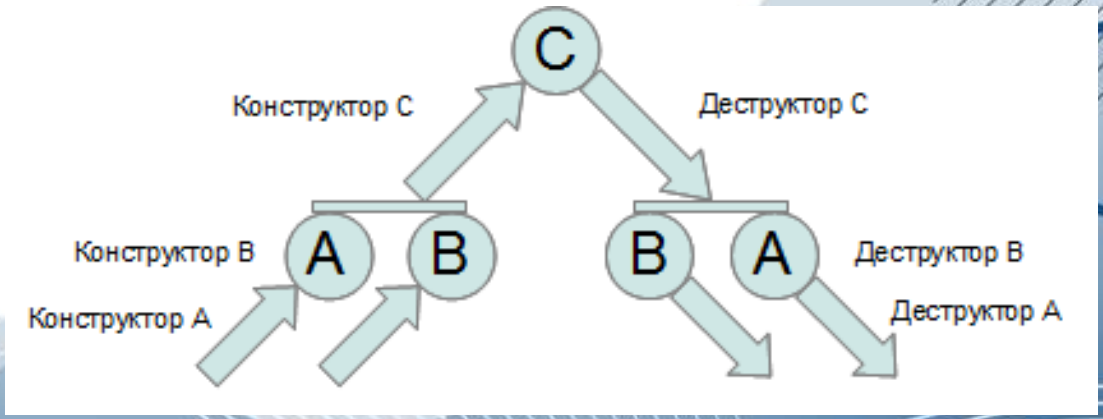
Изменяют эти свойства соответствующие им методы.

Например, метод Остановить() может изменить поле ТекущаяСкорость на значение 0.0 и поле СостояниеВелосипеда на значение false.



# Абстрагирование

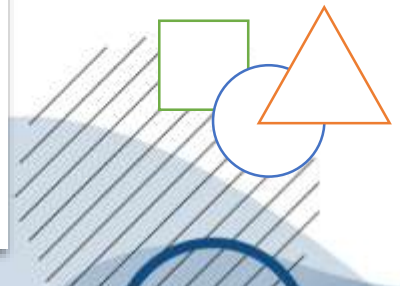
Для создания объектов класса служат специальные методы, которые называют **конструкторами**. Они необходимы для корректной инициализации объекта. Например, при создании матрицы заданных размеров конструктор должен выделить память для хранения элементов этой матрицы. Уничтожением объекта также занимается специальный метод класса, который называют **деструктором**. Его задача – освободить ресурсы, занимаемые объектом (закрыть используемые файлы, соединения с базами данных и пр.).

# Инкапсуляция

Этот термин характеризует **сокрытие** отдельных деталей внутреннего устройства класса от внешних по отношению к нему объектов или пользователей. Действительно, пользователю нет необходимости знать, из каких частей состоит экземпляр класса и каким образом реализовано его поведение. Реализация присущих классу структурных и поведенческих характеристик, является его собственным делом. Более того, отдельные свойства и методы вообще могут быть невидимы за пределами этого класса.

Например, класс **«Матрица» (Matrix)**: этот класс содержит переменные, доступ к которым может быть осуществлен только из методов класса. Такой подход используется для того, чтобы **защитить** переменные от несанкционированного доступа.



Если дать возможность программисту, который будет работать с объектом класса **Matrix**, изменять напрямую значения переменных для хранения размера матрицы, то эти значения могут стать некорректными (отрицательными, очень большими, или не соответствующими действительным размерам). А, выполняя то же самое с помощью методов класса, можно включить в них проверку корректности введенных значений. Для нашего класса таким методом может быть метод ввода матрицы.

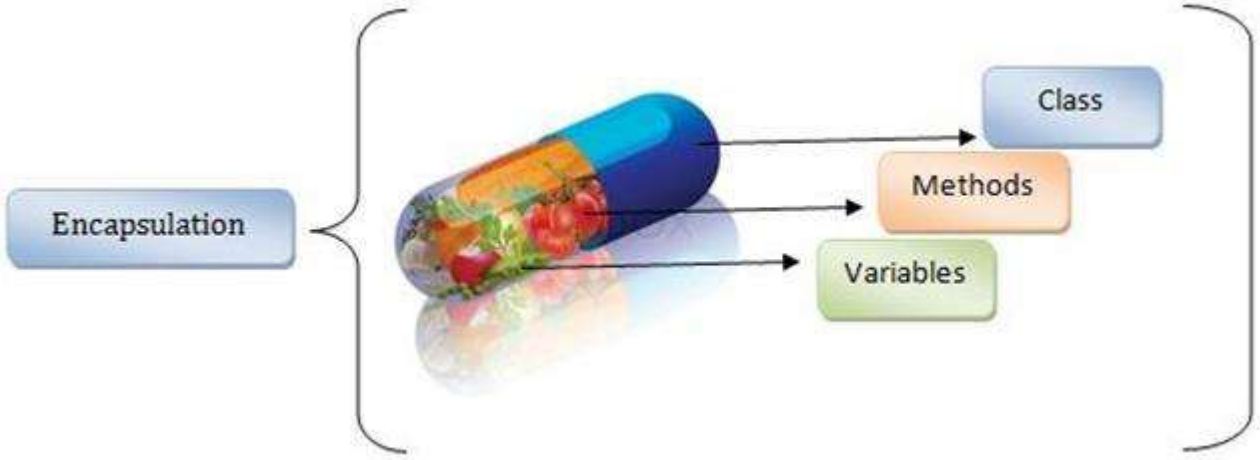
# Инкапсуляция

Помимо структурных характеристик класса **сокрытию подлежит и реализация операций класса**. Для пользователя нет необходимости знать, как реализован тот или иной метод. Надо знать только то, что метод выполняет, как к нему обратиться и как воспользоваться результатом его работы. Например, определитель квадратной матрицы можно вычислить различными способами: методом исключений Гаусса или по формуле Лапласа – разложение по строке или столбцу. Для пользователя не имеет значения, какой алгоритм заложен в методе класса.

Для реализации принципа сокрытия данных в объектно-ориентированных языках программирования явно определяется способ доступа к данным и методам класса. Доступ к элементам класса может быть **открытым (public)**, **защищенным (protected)**, **закрытым (private)**. К закрытым данным и методам можно обращаться только в методах самого класса. Защищенные данные и методы будут доступны для методов класса и классов, которые связаны с исходным отношением «родитель-потомок». К открытым (общедоступным) данным и методам можно обращаться из любого места программы.

# Инкапсуляция

Принципы **абстрагирования и инкапсуляции** используются совместно при разработке классов, дополняя друг друга. Уже на этапе выбора структурных и поведенческих характеристик класса, т.е. при применении принципа абстрагирования, определяется способ доступа к этим свойствам (применяется принцип инкапсуляции). Тем самым определяется внешний интерфейс класса – набор средств, которыми можно пользоваться извне при работе с объектами этого класса. Каждое такое средство определяет некоторое внешнее поведение объекта. Внутренняя же реализация этих средств скрыта от других объектов.



# Наследование

Для любого объектно-ориентированного приложения необходимо определить объекты, которые оно будет содержать. Иногда эти объекты могут быть схожими по своей структуре и поведению, но при этом иметь существенные различия. Также бывают ситуации, когда некоторые объекты являются частями других объектов. Подобные взаимосвязи образуют иерархию объектов – их упорядочивание. Основными видами иерархических структур применительно к объектно-ориентированным приложениям являются структура классов (иерархия «is-a» – «является», «is-like-a» – «является подобным») и структура объектов (иерархия «is-part-of» – «является частью»), которые определяются отношениями агрегации и обобщения.

# Наследование

**Наследование** – это механизм, который позволяет создавать новые классы на основе существующих, используя их структурные и поведенческие характеристики. Новые классы называют **дочерними** (производными или подклассами), а классы, на основе которых происходит наследование, – **родительскими** (базовыми или суперклассами). Кроме наследуемых свойств дочерние классы обладают дополнительными характеристиками, которые и отличают их от родительских.

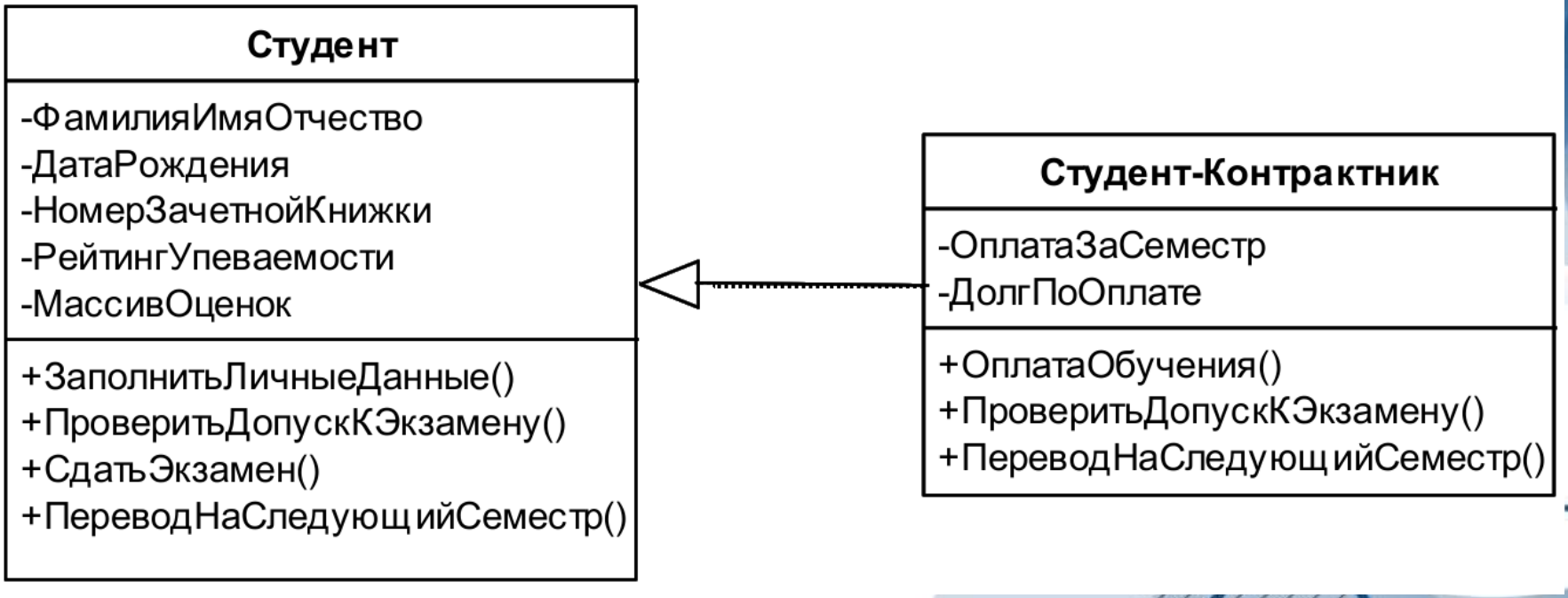
Например, в качестве родительского класса выступает **класс «Студент»**, который наследуется **дочерним классом «Студент-контрактник»**. Данное отношение классов образует иерархию вида «is-a» – «студент-контрактник является студентом». Использование принципа наследования позволяет расширить базовый класс посредством создания производного класса, содержащего дополнительно:

**Новые данные**, которые будут определять дочерний класс. Например, расширяя структурные свойства класса «Студент», добавим переменные для хранения значений общей стоимости обучения за год, внесенной суммы, оставшейся задолженности и, тем самым, образуем структурные характеристики нового класса «Студент-контрактник»

**Новые поведенческие характеристики дочернего класса.** Например, новым поведением объектов класса «Студент-контрактник» может служить внесение оплаты за обучение

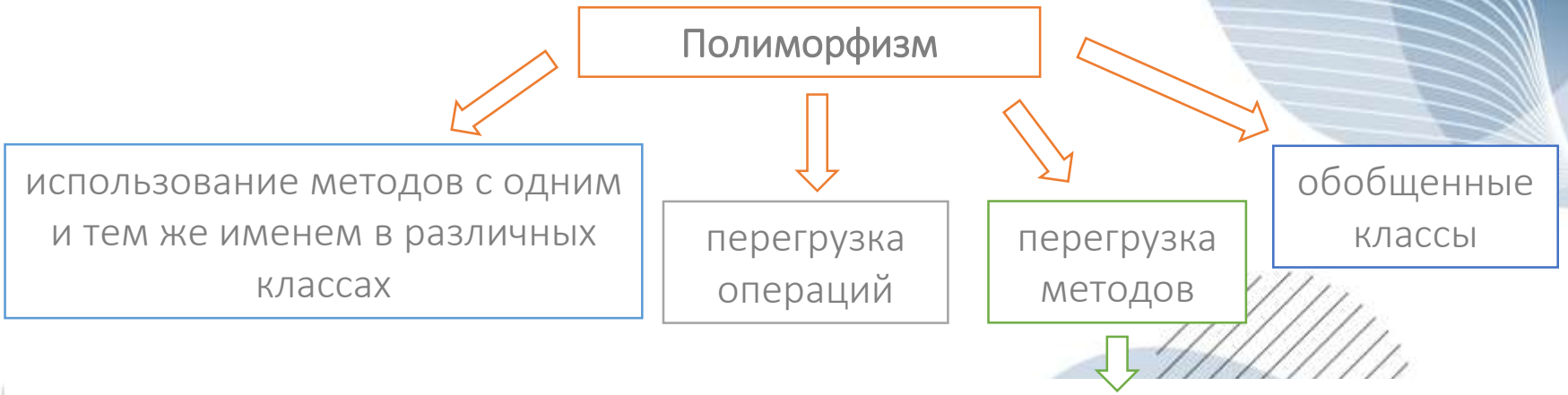
**Переопределенные поведенческие характеристики базового класса**. Например, если для студентов бюджетной формы обучения для допуска к экзамену необходим определенный балл. Это приведет к переопределению в дочернем классе метода базового класса, проверяющего допуск студента к экзамену. Переопределенный метод имеет тот же прототип и скрывает в производном классе метод базового класса.

# Наследование



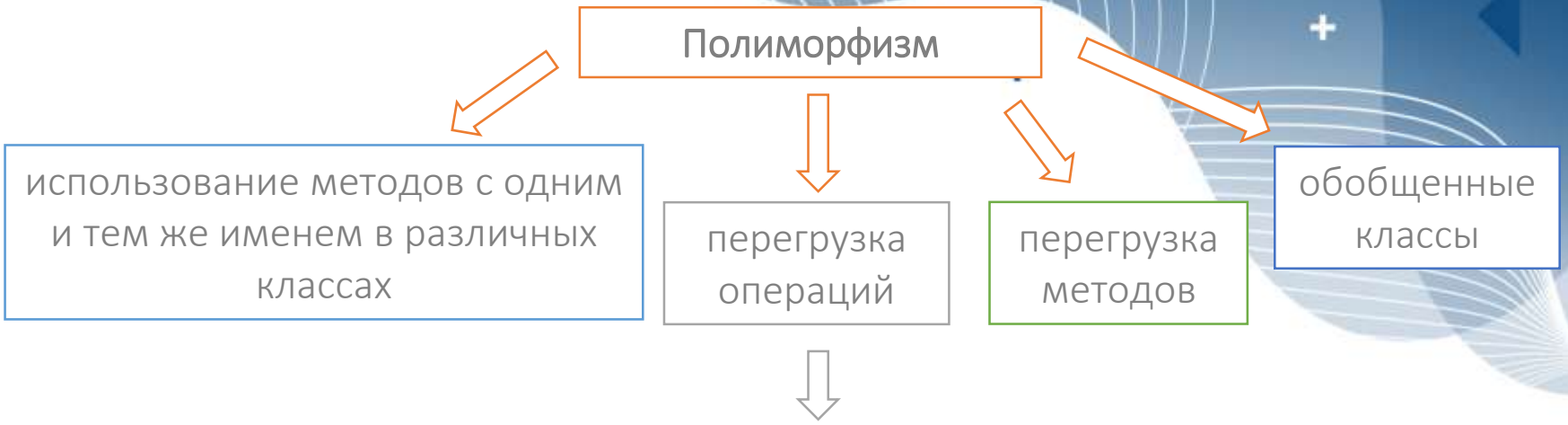
# Полиморфизм

Слово **«полиморфизм»** означает **«имеющий множество форм»**. В программировании под **полиморфизмом** понимают использование одного и того же имени для выполнения различных задач.



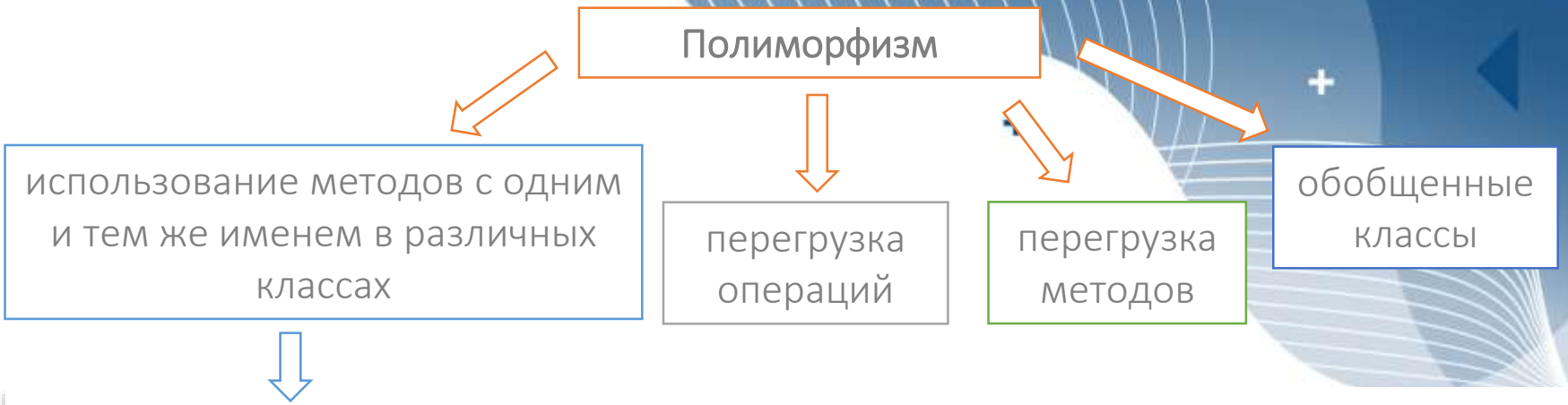

Часто приходится разрабатывать методы, выполняющие **одинаковые действия** с различными типами данных. Например, методы сортировки массивов, содержащих элементы различных типов (целого, вещественного или символьного типов), удобно было бы называть одинаково. Поэтому в языках программирования предусмотрена возможность создавать методы с одинаковыми именами, но различными параметрами. Такие **методы** называются **перегруженными**. Типы возвращаемых значений у них также могут отличаться, однако использование методов, которые отличаются только типом возвращаемого значения недопустимо.

# Полиморфизм




При проектировании классов, характеризующих поведение математических объектов, удобно использовать традиционные математические знаки операций для выполнения соответствующих действий. Например, при сложении двух матриц было бы понятней использовать операцию "+", а не вызывать метод Summ() (тем более, что другой программист может назвать его иным именем). Для таких ситуаций используется **перегрузка операций**.

# Полиморфизм

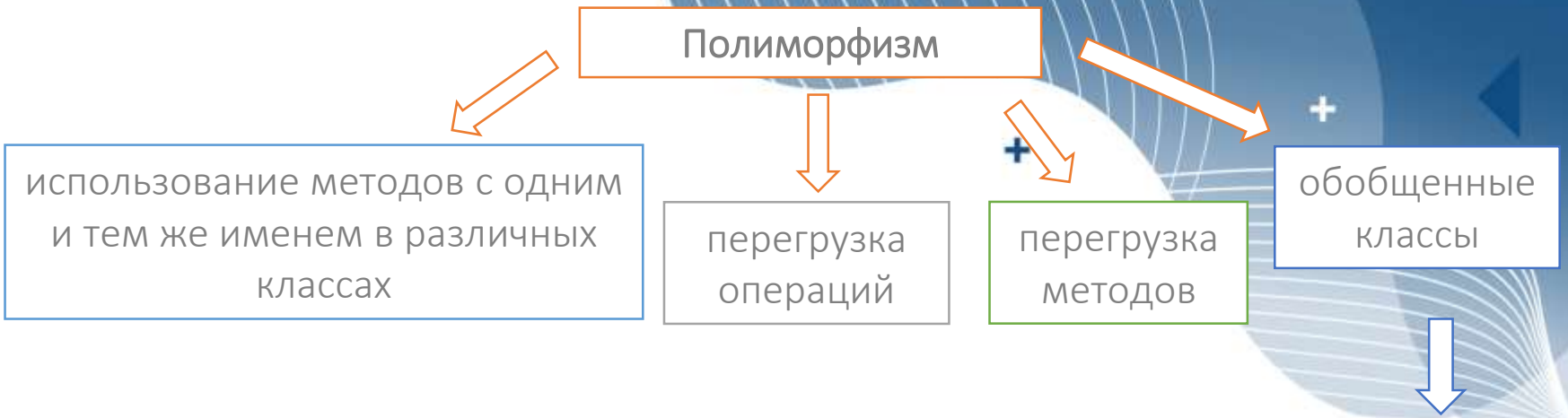



Несколько классов могут иметь **методы с одинаковыми именами**. Это означает, что действия, выполняемые одноименными методами, могут различаться в зависимости от того, к какому из классов относится тот или иной метод. Например, классы «Учебная группа» и «Студент» могут содержать методы для печати информации о соответствующем объекте, которые имеют одинаковое имя, например, **Печать()**. При использовании метода **Печать()** будет выводиться информация о том объекте, для которого он вызван: при вызове метода для объекта класса «Учебная группа» будет печататься список студентов этой группы, а при вызове для объекта класса «Студент» – информация о студенте (ФИО, № зачетной книжки, № группы и др.). Это возможно за счет того, что каждый объект знает, какому классу он принадлежит, что позволяет вызвать метод именно этого класса.

# Полиморфизм


Другой способ использования методов с одинаковыми именами в различных классах основан на понятии **виртуального метода**. Механизм использования виртуальных методов основывается на **возможности хранения** в переменной родительского класса объекта дочернего класса. По умолчанию выбор вызываемого метода осуществляется в соответствии с типом объекта, хранящегося в переменной. Если в родительском классе вызываемый метод был определен как виртуальный, а в дочернем классе он был переопределен, то будет вызываться последний. Например, в одной группе могут учиться и бюджетные, и контрактные студенты. Тогда класс «Учебная группа» может содержать список переменных класса «Студент», некоторые из которых могут хранить объекты класса «Студент-контрактник». В классе «Студент» могут быть определены виртуальные методы (например, ПроверитьДопускКСсесии(), ПереводНаДругойСеместр(),Печать()), которые должны быть переопределены в классе «Студент-контрактник». Тогда в методах класса «Учебная группа» информация о студентах может обрабатываться единообразно (посредством вызова виртуальных методов).

# Полиморфизм




**Обобщенные классы** (шаблоны классов) определяют описание класса для обобщенного типа данных. При обращении к шаблону класса указывается конкретный тип данных (int, double или класс), который подставляется на место обобщенного типа. Таким образом, компилятор генерирует класс для этого конкретного типа. Подставляя разные типы данных, можно создать множество классов на основе шаблона, реализующих единый алгоритм обработки данных. Например, на основе шаблона класса «Матрица» могут создаваться объекты-матрицы, элементами которых являются целые числа, рациональные дроби (объекты класса «Дробь»), массивы и пр.

# Основы ООП

## Лекция 3. Классы Python

# Объекты

В модели ООП языка Python имеются два вида объектов: **объекты классов** и **объекты экземпляров**. Объекты классов обеспечивают стандартное поведение и служат фабриками для объектов экземпляров. Объекты экземпляров являются действительными объектами, обрабатываемыми вашей программой - каждый представляет собой самостоятельное пространство имен, но наследует (т.е. автоматически получает доступ) имена от класса, из которого он был создан. Объекты классов происходят из операторов, а экземпляры - из вызовов; при каждом обращении к классу вы получаете новый экземпляр этого класса. По существу классы являются фабриками для генерирования множества экземпляров.

# class

В результате выполнения оператора **class** мы получаем объект класса. Основные характеристики классов Python:

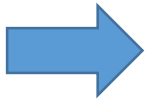
• Оператор **class** создает объект класса и присваивает его имени. В точности как оператор def определения функции оператор **class** является исполняемым. После достижения и запуска он генерирует новый объект класса и присваивает его имени, указанному в заголовке **class**. Также подобно **def** операторы **class** обычно выполняются при первом импортировании файлов, где они находятся.

• Присваивания внутри операторов **class** создают атрибуты классов. Как и в файлах модулей, присваивания на верхнем уровне внутри оператора **class** (не вложенные в def) генерируют атрибуты в объекте класса. Формально оператор **class** определяет локальную область видимости, которая превращается в пространство имен атрибутов для объекта класса подобно глобальной области видимости модуля. После выполнения оператора **class** атрибуты класса доступны посредством уточнения с помощью имени: **объект.имя**.

• Атрибуты класса снабжают объект состоянием и поведением. Атрибуты объекта класса хранят информацию о состоянии и описывают поведение, которое разделяется всеми экземплярами, создаваемыми из класса; операторы **def** определения функций, вложенные внутрь **class**, генерируют методы, которые обрабатывают экземпляры.

# Практика

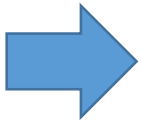
```python
# Объявление пустого класса MyClass
class MyClass:
    pass

# Создание экземпляра класса
obj = MyClass()

# Вывод типа obj
print(type(obj))
```



```
<class '__main__.MyClass'>
```

# Практика

```python
# В Python всё является объектами, в том числе
# и сами классы


# Объявление пустого класса MyClass
class MyClass:
    pass

# Создание экземпляра класса
obj = MyClass()

# Объект obj -- это экземпляр класса MyClass,
# то есть он имеет тип MyClass
print(type(obj))  # <class '__main__.MyClass'>

# MyClass -- это класс, но также он является
# и объектом, экземпляром метакласса type,
# являющегося абстракцией понятия типа данных
print(type(MyClass))  # <class 'type'>

# Соответственно, с классами работать как
# с объектами, например, копировать
AnotherClass = MyClass
print(type(AnotherClass))

# Как видим, теперь AnotherClass -- это то же самое, что и MyClass,
# и obj является и экземпляром класса AnotherClass
print(isinstance(obj, AnotherClass))  # True
```



```
<class '__main__.MyClass'>
<class 'type'>
<class 'type'>
True
```

# Экземпляры

При обращении к объекту класса мы получаем объект экземпляра. Ниже приведен краткий обзор ключевых моментов, касающихся экземпляров класса.

• Обращение к объекту класса как к функции создает новый объект **экземпляра**.

При каждом обращении к классу он создает и возвращает новый объект экземпляра. ***Экземпляры представляют конкретные элементы в предметной области программы.***

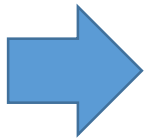
• Каждый объект экземпляра наследует атрибуты класса и получает собственное пространство имен. Объекты экземпляров, созданные из классов, являются новыми пространствами имен. Объекты экземпляров начинают свое существование пустыми, но наследуют атрибуты, имеющиеся в объектах классов, из которых они были сгенерированы.

• Присваивания атрибутам аргумента **self** в методах создают атрибуты для отдельного экземпляра. Внутри функций методов класса первый аргумент (по соглашению называемый **self**) ссылается на обрабатываемый объект экземпляра; присваивания атрибутам аргумента **self** создают либо изменяют данные в экземпляре, но не в классе.

Конечным результатом оказывается то, что классы определяют общие разделяемые данные и поведение плюс генерируют экземпляры. Экземпляры отражают конкретные сущности приложения и хранят собственные данные, которые могут варьироваться от объекта к объекту.

# Практика

```
class FirstClass:
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)

y = FirstClass()
x = FirstClass()

x.setdata("King Arthur")
y.setdata(3.141)

x.display()
y.display()

x.data = "New value"
x.display()
```



```
King Arthur
3.141
New value
```

# Практика

Как и все составные операторы, оператор class начинается со строки заголовка с именем класса, после чего следует тело с одним или несколькими вложенными операторами, (обычно) набранными с отступом. В приведенном примере вложенными операторами являются **def**; они определяют функции, которые реализуют поведение класса, предназначенное для экспортирования.

В примере операторы **def** присваивают объекты функций именам **setdata** и **display** в области видимости оператора **class**, а потому генерируют атрибуты, присоединяемые к классу - **FirstClass.setdata** и **FirstClass.display**. В действительности любое имя, присвоенное на верхнем уровне вложенного блока класса, становится атрибутом этого класса.

Функции внутри класса, как правило, называются **методами**. Они создаются посредством нормальных операторов **def** и поддерживают все, что вам уже известно о функциях (т.е. могут иметь стандартные значения аргументов, возвращать значения, выдавать элементы по запросу и т.п.). Но первый аргумент в функции метода при ее вызове автоматически получает подразумеваемый объект экземпляра - объект, на котором произведен вызов.

Обращаясь к классу таким способом (обратите внимание на круглые скобки), мы генерируем объекты экземпляров, представляющие собой просто пространства имен, которые имеют доступ к атрибутам своих классов. Собственно говоря, в этой точке мы имеем три объекта: два экземпляра и класс.

# Практика

В терминах ООП мы говорим, что экземпляр **x “является” FirstClass**, равно как и **y** - они оба наследуют имена, присоединенные к классу.
Классы и экземпляры связывают объекты пространств имен в дерево классов, в котором ищутся атрибуты при поиске в иерархии наследования. Здесь атрибут **data** находится в экземплярах, но **setdata** и **display** присутствуют в классах, расположенных выше них.
Два экземпляра начинают свое существование как пустые, но имеют ссылки на класс, из которого они были сгенерированы. Если мы уточним экземпляр с помощью имени атрибута, который находится в объекте класса, тогда Python извлечет имя из класса посредством поиска в иерархии наследования (при условии, что он также не присутствует в экземпляре).
Ни x, ни y не имеет собственного атрибута setdata, поэтому чтобы найти его, Python следует по ссылке из экземпляра в класс. Вот и все, что нужно для наследования в Python: оно происходит во время уточнения атрибутов и предусматривает лишь поиск имен в связанных объектах - в данном случае за счет следования по ссылкам.

# Практика

В функции **setdata** класса **FirstClass** передаваемое значение присваивается **self.data**. Внутри метода **self** (имя, по соглашению назначаемое крайнему слева аргументу) автоматически ссылается на обрабатываемый экземпляр (x или y), так что присваивания сохраняют значения в пространствах имен экземпляров, а не класса.

Поскольку классы способны генерировать множество экземпляров, методы должны с помощью аргумента **self** получать обрабатываемый экземпляр. Вызвав метод **display** класса для вывода **self.data**, мы заметим, что он отличается для каждого экземпляра; с другой стороны, само имя **display** одинаковое в x и y, т.к. оно поступает (наследуется) от класса.

В качестве еще одного способа оценить, насколько динамична эта модель, имейте в виду, что мы можем изменять атрибуты в самом классе, присваивая **self** в методах, или класса путем присваивания явному объекту экземпляра.

# Классы настраиваются через наследование

Помимо того, что классы служат фабриками для генерирования множества объектов экземпляров, они так же предоставляют возможность вносить изменения за счет ввода новых компонентов (называемых **подклассами**) вместо изменения существующих компонентов на месте.

Python также позволяет классам быть унаследованными от других классов, открывая возможность создания классов, которые специализируют поведение. Переопределяя атрибуты в подклассах, которые находятся ниже в иерархии, мы переопределяем более общие определения таких атрибутов выше в дереве. По сути, чем ниже мы углубляемся в иерархию, тем более специфическим становится программное обеспечение. Здесь отсутствуют какие-либо параллели с модулями, чьи атрибуты находятся в единственном плоском пространстве имен, которое не поддается настройке.

Экземпляры в Python наследуют от классов, а классы - от суперклассов.

# Классы настраиваются через наследование

Основы механизма наследования атрибутов:

• Суперклассы перечисляются внутри круглых скобок в заголовке class. Чтобы заставить класс наследовать атрибуты от другого класса, просто укажите другой класс внутри круглых скобок в строке заголовка нового оператора class. Класс, выполняющий наследование, обычно называется подклассом, а класс, от которого производится наследование, является его суперклассом.

• Классы наследуют атрибуты от своих суперклассов. Точно так же, как экземпляры наследуют имена атрибутов, определенные в их классах, классы наследуют все имена атрибутов, которые определены в их суперклассах; при доступе к атрибутам Python находит их автоматически, если они не существуют в подклассах.

# Классы настраиваются через наследование

Основы механизма наследования атрибутов:

• Экземпляры наследуют атрибуты от всех доступных классов. Каждый экземпляр получает имена от класса, из которого он сгенерирован, а также от всех суперклассов этого класса. При поиске имени Python проверяет экземпляр, затем его класс и, наконец, все суперклассы.

• Каждая ссылка объект. атрибут инициирует новый независимый поиск. Python выполняет независимый поиск в дереве классов для каждого выражения с из влечением атрибута. Сюда входят ссылки на экземпляры и классы, сделанные за пределами операторов **class** (например, **x.атрибут**), а также ссылки на атрибуты экземпляра аргумента **self** в функциях методов класса. Каждое выражение **self.атрибут** в методе вызывает новый поиск для атрибута в **self** и выше.

• Изменения в логику вносятся за счет создания подклассов, а не модификации суперклассов. Переопределяя имена суперклассов в подклассах ниже в иерархии (дерева классов), подклассы замещают и тем самым настраивают унаследованное поведение.

# Практика

Первым делом мы определим новый класс **SecondClass**, который наследует все имена **FirstClass** и предоставляет одно собственное имя.

```
class SecondClass(FirstClass): # Наследует setdata
    def display(self): # Изменяет display
        print(f'Current value = {self.data}')
```

Класс **SecondClass** определяет метод **display** для вывода в другом формате. За счет определения атрибута с таким же именем, как у атрибута в **FirstClass**, класс **SecondClass** фактически замещает атрибут **display** в своем суперклассе. Вспомните, что поиск в иерархии наследования направлен вверх от экземпляров к подклассам и далее к суперклассам, останавливаясь на первом найденном вхождении имени атрибута. В данном случае, поскольку имя **display** в **SecondClass** будет обнаружено перед таким же именем в **FirstClass**, мы говорим, что **SecondClass** переопределяет **display** из **FirstClass**. Действие по замещению атрибутов путем их переопределения ниже в дереве иногда называют **перегрузкой**.

```
z = SecondClass()
z.setdata(42) # Находит setdata в FirstClass
z.display()
```

```
Current value = 42
```

# Практика

Первым делом мы определим новый класс **SecondClass**, который наследует все имена **FirstClass** и предоставляет одно собственное имя.

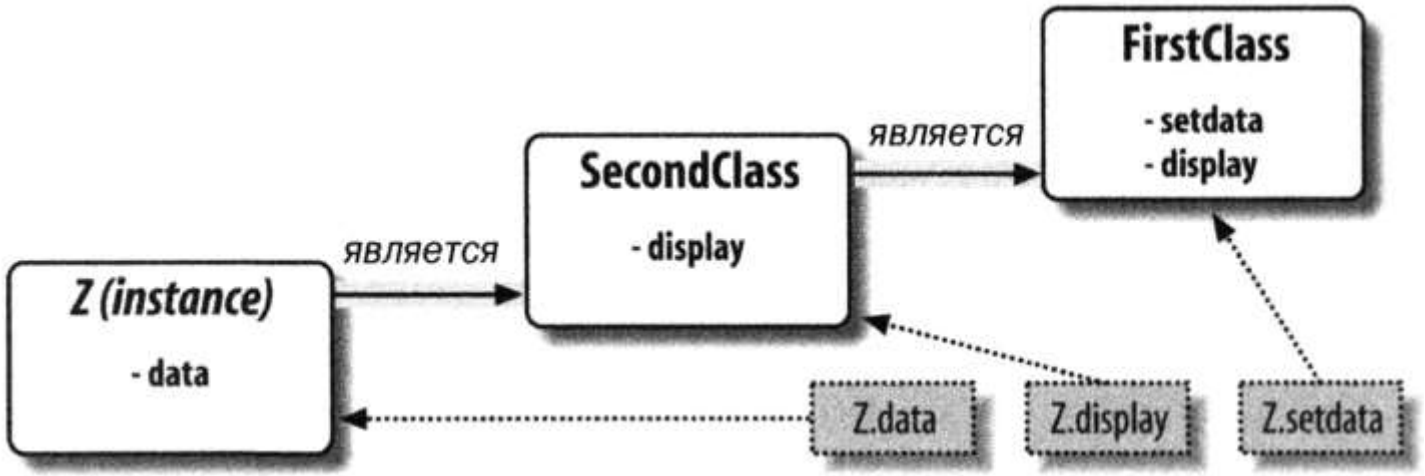
```
class SecondClass(FirstClass): # Наследует setdata
    def display(self): # Изменяет display
        print(f'Current value = {self.data}')
```

Класс **SecondClass** определяет метод **display** для вывода в другом формате. За счет определения атрибута с таким же именем, как у атрибута в **FirstClass**, класс **SecondClass** фактически замещает атрибут **display** в своем суперклассе. Вспомните, что поиск в иерархии наследования направлен вверх от экземпляров к подклассам и далее к суперклассам, останавливаясь на первом найденном вхождении имени атрибута. В данном случае, поскольку имя **display** в **SecondClass** будет обнаружено перед таким же именем в **FirstClass**, мы говорим, что **SecondClass** переопределяет **display** из **FirstClass**. Действие по замещению атрибутов путем их переопределения ниже в дереве иногда называют **перегрузкой**.

```
z = SecondClass()
z.setdata(42) # Находит setdata в FirstClass
z.display()
```

```
Current value = 42
```

# Практика




Специализация: переопределение унаследованных имен за счет их повторного определения в расширениях ниже в дереве классов. Здесь **SecondClass** переопределяет и тем самым настраивает метод **display** для своих экземпляров.

# Практика

Следует отметить, что с именем класса не связано ничего магического. Оно представляет собой всего лишь переменную, которой при выполнении оператора **class** присваивается объект, и на объект можно ссылаться с помощью любого нормального выражения.

В единственном файле модуля может находиться несколько классов - подобно другим операторам в модуле операторы **class** выполняются во время импортирования для определения имен, которые становятся индивидуальными атрибутами модуля. В более общем случае каждый модуль может произвольно смешивать любое количество переменных, функций и классов, причем все имена в модуле ведут себя одинаково.

Как в случае любой другой переменной, мы не можем увидеть класс в файле без предварительного импортирования и его извлечения из включающего файла. Если это кажется непонятным, просто не применяйте одинаковые имена для модуля и класса внутри него. На самом деле по соглашению, принятому в Python, имена классов должны начинаться с буквы верхнего регистра, чтобы сделать их более различимыми.

Кроме того, хотя и классы, и модули являются пространствами имен для присоединения атрибутов, они соответствуют очень разным структурам исходного кода: модуль отражает целый файл, а класс является оператором внутри файла.

# Перегрузка операций

***Перегрузка операций*** позволяет объектам, сознанным из классов, перехватывать и реагировать на операции, которые работают со встроенными типами: сложение, нарезание, вывод, уточнение и т.д. По большей части это просто механизм автоматической диспетчеризации - выражения и другие встроенные операции передают управление реализациям в классах.

Поскольку перегрузка операций заставляет наши объекты действовать подобно встроенным объектам, это способствует созданию более согласованных и легких в изучении объектных интерфейсов, а также делает возможной обработку объектов, основанных на классах, с помощью кода, который написан в расчете на интерфейс встроенного типа.

# Перегрузка операций

Основные идеи, лежащие в основе перегрузки операций:

- Методы, имена которых содержат удвоенные символы подчеркивания (__ x__ ), являются специальными привязками. В классах Python мы реализуем перегрузку операций за счет предоставления особым образом именованных методов для перехвата операций. В языке Python определено фиксированное и неизменяемое отображение каждой операции на метод со специальным именем.
- Такие методы вызываются автоматически, когда экземпляры встречаются во встроенных операциях. Скажем, если объект экземпляра наследует метод __ add__ , то этот метод вызывается всякий раз, когда объект появляется в выражении с операцией +. Возвращаемое значение метода становится результатом соответствующего выражения.
- Классы могут переопределять большинство встроенных операций с типами. Существуют десятки специальных имен методов для перегрузки операций, которые можно перехватывать и реализовывать почти каждую операцию, действующую на встроенных типах. Сюда входят не только операции выражений, но также базовые операции наподобие вывода и создания объектов.
- Для методов перегрузки операций не предусмотрены стандартные реализации и ни один из них не является обязательным. Если класс не определяет или не наследует какой-то метод перегрузки операции, то это просто означает, что соответствующая операция не поддерживается для экземпляров класса. Например, если метод__ add__ отсутствует, тогда выражения + будут приводить к исключениям.

# Практика

Определим подкласс класса SecondClass из предыдущего раздела и реализуем три особым образом именованных атрибута, которые будут автоматически вызываться Python:

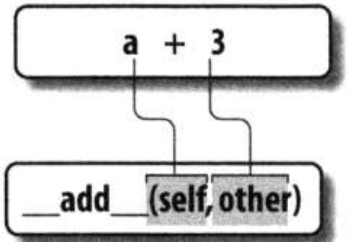
• **__ init__** выполняется, когда создается новый объект экземпляра:
**self** является новым объектом ThirdClass1;

• **add** выполняется, когда экземпляр ThirdClass присутствует в выражении +;

• **str** выполняется, когда объект выводится (формально при его преобразовании в отображаемую строку встроенной функцией str или ее внутренним эквивалентом Python).

# Практика

```
class ThirdClass(SecondClass):
    def __init__(self, value):
        self.data = value

    def __add__(self, other):
        return ThirdClass(self.data + other)

    def __str__(self):
        return f'ThirdClass: {self.data}'

    def mul(self, other):
        self.data *= other
```




```
a = ThirdClass ('abc')
a.display()
print(a)

b = a + 'xyz'
b.display()
print(b)

a.mul(3)
print(a)
```

# Вызывается__init__
# Вызывается унаследованный метод
# __str__ : возвращает отображаемую строку

# __add__ : создает новый экземпляр
# b имеет все методы класса ThirdClass
# __str__ : возвращает отображаемую строку

# mul: изменяет экземпляр на месте

```
Current value = abc
ThirdClass: abc
Current value = abcxyz
ThirdClass: abcxyz
ThirdClass: abcabcabc
```

# Практика

Класс **ThirdClass** “является” **SecondClass**, поэтому его экземпляры наследуют на встроенный метод **display** от класса **SecondClass** из предыдущего раздела. Тем не менее, на этот раз при создании экземпляра ThirdClass передается **аргумент (' abc ’).** Аргумент передается аргументу value **конструктора__ init__** и присваивается здесь атрибуту self.data. Совокупный эффект в том, что ThirdClass организован так, чтобы устанавливать атрибут **data** автоматически во время конструирования, не требуя последующего вызова **setdata**.

Кроме того, объекты **ThirdClass** теперь могут появляться в выражениях + и вызовах **print**. В случае выражения + интерпретатор Python передает объект экземпляра слева аргументу **self** и значение справа аргументу **other** в методе**__ add__** ; любое возвращаемое значение**__ add__** становится результатом выражения +.

При перегрузке операции выражений и другие встроенные операции, выполняемые над классом, отображаются на методы с особыми именами в классе. Такие специальные методы необязательны и могут быть унаследованы обычным образом. В данном случае выражение + запускает метод_ add

# Практика

Для вызова **print** интерпретатор Python передает выводимый объект аргументу **self** в методе__ **str**__ ; любая возвращаемая этим методом строка становится отображаемой строкой для объекта. Благодаря методу__ **str**__ (или его более подходящему двойнику __ **repr**__ , мы можем применять для вывода объектов данного класса нормальный вызов print вместо обращения к специальному методу display.

Особым образом именованные методы, такие как__ **init**__ ,__ **add**__ **и**___**str**__ , наследуются подклассами и экземплярами в точности подобно любым другим именам, присваиваемым в операторе **class**. Если они не реализованы в классе, тогда Python ищет такие имена во всех суперклассах класса, как обычно. Имена методов для перегрузки операций также не являются встроенными или зарезервированными словами; они представляют собой всего лишь атрибуты, которые Python ищет, когда объекты появляются в разнообразных контекстах. Обычно Python вызывает их автоматически, но иногда они могут вызываться также и в коде.

# Практика

Некоторые методы для перегрузки операций, скажем, **__ str__** , требуют результатов, но другие обладают большей гибкостью. Например, обратите внимание на то, как метод**__ add__** создает и возвращает новый объект экземпляра класса **ThirdClass**, вызывая **ThirdClass** с результирующим значением, что в свою очередь запускает **__ init__** для инициализации результатом. Это общее соглашение, которое объясняет, почему переменная b имеет метод display; она тоже является объектом ThirdClass, т.к. именно его возвращает операция + для объектом данного класса.

Метод **mul** изменяет текущий объект экземпляра на месте, заново присваивая атрибут self. Чтобы сделать последнее, мы могли бы перегрузить операцию выражения *, но тогда результат слишком бы отличался от поведения * для встроенных типов, таких как числа и строки, где операция * всегда создает новые объекты. Общая практика требует, чтобы перегруженные операции работали таким же образом, как их встроенные реализации. Однако поскольку перегрузка операций в действительности представляет собой просто механизм диспетчеризации между выражениями и методами, в объектах собственных классов вы можете интерпретировать операции любым желаемым способом.

# Практика

Одним из методов перегрузки, который мы будем часто здесь использовать, является **метод конструктора__ init__** , применяемый для инициализации вновь созданных объектов экземпляров и присутствующий почти в каждом реалистичном классе. Из-за того, что конструктор позволяет классам немедленно заполнять атрибуты в своих новых экземплярах, он удобен практически во всех видах классов, которые вам доведется реализовывать. На самом деле, хотя атрибуты экземпляра в Python не объявляются, обычно легко выяснить, какие атрибуты будет иметь экземпляр, проинспектировав **метод__ init__** его класса.

# Атрибуты

Базовая модель наследования, которую производят классы, очень проста - в действительности она включает в себя всего лишь поиск атрибутов в деревьях связанных объектов. Фактически мы можем создать класс, не содержащий вообще ничего. Следующий оператор создает класс без присоединенных атрибутов, т.е. объект пустого пространства имен

```
class record:
    pass

record.name = 'Bob'
print(record.name)
```

```
Bob
```

Создав с помощью присваивания атрибуты, мы можем извлекать их с использованием обычного синтаксиса.

**Обратите внимание, что прием работает, даже когда еще нет ни одного экземпляра класса; классы сами по себе являются объектами и без экземпляров.** В действительности они представляют собой всего лишь автономные пространства имен; до тех пор, пока у нас есть ссылка на класс, мы в любой момент можем устанавливать либо изменять его атрибуты.

# Атрибуты

Экземпляры начинают свое существование как объекты совершенно пустых пространств имен. Однако поскольку экземпляры запоминают класс, из которого были созданы, они будут извлекать атрибуты, присоединенные нами к классу, через наследование:

```
x = record()
y = record()
```

На самом деле эти экземпляры сами не имеют атрибутов; они просто извлекают атрибут **name** из объекта класса, где он хранится. Если же мы присваиваем атрибуту экземпляра, тогда создается (или изменяется) атрибут в одном объекте, но не в другом - критически важно то, что ссылки на атрибуты инициируют поиск в иерархии наследования, а присваивания атрибутов влияют только на объекты, в которых присваивания выполнялись. Здесь это означает, что x получает собственный атрибут **name**, но **y** по-прежнему наследует атрибут **name**, **присоединенный к классу выше в дереве**:

```
x.name = 'Sue'
print(record.name, x.name, y.name)
```

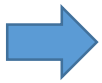
```
Bob Sue Bob
```

Атрибуты объекта пространства имен обычно реализуются как словари, а деревья наследования классов представляют собой (говоря в общем) всего лишь словари, содержащие связи с другими словарями. Если вы знаете, куда смотреть, то сможете увидеть это явно.

# Атрибуты

Здесь словарь пространств имен класса содержит присвоенные ранее атрибуты **name** и **age**, экземпляр **x** имеет собственный атрибут **name**, а экземпляр y все еще пуст. Из-за такой модели атрибут часто может извлекаться либо посредством индексирования словаря, либо с помощью записи атрибута, но только если он присутствует в обрабатываемом объекте. Запись атрибута инициирует поиск в иерархии наследования, но индексирование ищет только в одиночном объекте.

```
print(list(record.__dict__.keys()))
print(list(x.__dict__.keys()))
print(list(y.__dict__.keys()))
```



```
['__module__', '__dict__', '__weakref__', '__doc__', 'name']
['name']
[]
```

Для упрощения поиска в иерархии наследования при извлечении атрибутов каждый экземпляр имеет связь со своим классом, которую создает Python - она называется **__ class__** и ее можно просмотреть:

```
print(x.__class__)
```

```
<class '__main__.record'>
```

Классы также располагают атрибутом **__ bases__** , который является кортежем ссылок на их объекты суперклассов - в данном примере только подразумеваемый корневой класс **object**.

```
print(record.__bases__)
```

```
(<class 'object'>,)
```

# Атрибуты

Главное, на что стоит обратить внимание - модель классов Python чрезвычайно динамична. Классы и экземпляры представляют собой всего лишь объекты пространств имен с атрибутами, создаваемыми на лету через присваивания. Такие присваивания, как правило, происходят внутри записываемых вами операторов **class**, но могут встречаться везде, где имеется ссылка на один из объектов в дереве. Даже методы, которые обычно создаются с помощью операторов **def**, вложенных в **class**, могут быть созданы совершенно независимо от любого объекта класса.

```
def uppername (obj):
    return obj.name.upper()

print(uppername(x))
```

```
SUE
```

Здесь пока еще ничего не связано с классом; **uppername** является простой функцией и может вызываться в данной точке при условии передачи ей объекта **obj** с атрибутом name, значение которого имеет метод **upper**. Экземпляры нашего класса соответствуют ожидаемому интерфейсу и запускают преобразование строк в верхний регистр.
Тем не менее, если мы присвоим эту простую функцию атрибуту нашего класса, она становится методом, допускающим вызов через любой экземпляр, а также через имя самого класса при условии передачи экземпляра вручную.

```
record.method = uppername
print(x.method())
print(record.method(x))
```

# Атрибуты

Обычно классы заполняются посредством операторов **class**, а атрибуты экземпляров создаются с помощью присваиваний атрибутам **self** в функциях методов, еще раз отметим, что поступать так необязательно; **ООП в Python главным образом касается поиска атрибутов в связанных объектах пространств имен**.

# Атрибуты

Классы упаковывают информацию подобно словарям, но могут также умещать в себе логику обработки в форме методов.

```
rec = {}
rec['name'] = 'Bob'
rec['age' ] = 40.5
rec ['jobs' ] = ['dev' , 'mgr' ]
print(rec)
```

```
{'name': 'Bob', 'age': 40.5, 'jobs': ['dev', 'mgr']}
```

Код эмулирует инструменты, похожие на записи в других языках. Тем не менее, существует также множество способов делать то же самое с применением классов. Пожалуй, простейший из них предусматривает замену ключей атрибутами

```
class rec:
    pass
rec.name = 'Bob'
rec.age = 40.5
rec.jobs = ['dev', 'mgr']
```

С помощью оператора class создается объект пустого пространства имен. С течением времени полученный пустой класс заполняется путем присваивания атрибутов класса, как и ранее.

# Атрибуты

Прием работает, но для каждой отличающейся записи будет требоваться новый оператор **class**. Вероятно, более естественно взамен генерировать экземпляры пустого класса для представления каждой отличающейся записи.

```
class rec:
    pass
pers1 = rec()
pers1.name = 'Bob'
pers1.jobs = ['dev']
pers1.age = 40.5
pers2 = rec()
pers2.name = 'Sue'
pers2.jobs = ['dev', 'cto']
```

# Атрибуты

Наконец, мы можем вместо этого создать более развитый класс с целью реализации записи и ее обработки - то, что словари, ориентированные на данные, не поддерживают напрямую.

```
class Person:
    def __init__(self, name, jobs, age=None):
        self.name = name
        self.jobs = jobs
        self.age = age
    def info(self):
        return (self.name, self.jobs)

rec1 = Person('Bob', ['dev', 'mgr'], 40.5)
rec2 = Person('Sue', ['dev', 'cto'])
```

Конструктор придает экземплярам некоторую согласованность, всегда устанавливая атрибуты **name**, **job** и **age**, хотя последний атрибут может не указываться. Вместе методы класса и атрибуты экземпляра образуют пакет, объединяющий данные и логику.

# Резюме

- Классы создаются путем выполнения операторов class; экземпляры создаются за счет обращения к классу, как если бы он был функцией.

- Атрибуты класса создаются путем выполнения присваивания атрибутам объекта класса. Они обычно генерируются присваиваниями верхнего уровня, вложенными внутрь оператора **class** - каждое имя, присвоенное в блоке оператора **class,** становится атрибутом объекта класса. Тем не менее, атрибуты класса можно также создавать путем их присваивания везде, где имеется ссылка на объект класса - даже за пределами оператора **class.**

- Атрибуты экземпляра создаются посредством присваивания значений атрибутам объекта экземпляра. Они обычно создаются в функциях методов класса, реализованных внутри оператора **class**, с помощью присваивания значений атрибутам аргумента **self** (который всегда является подразумеваемым экземпляром). Однако их тоже можно создавать присваиванием везде, где присутствует ссылка на экземпляр, даже за пределами оператора **class**. Обычно все атрибуты экземпляра инициализируются в методе конструктора **__init__**; таким образом, более поздние вызовы методов могут предполагать, что атрибуты уже существуют.

# Резюме

- **self** - это имя, обычно назначаемое первому (крайнему слева) аргументу в функции метода класса; Python автоматически заполняет его объектом экземпляра, который представляет собой подразумеваемый объект вызова метода. Данный аргумент не обязан называться self (хотя соглашение очень строгое); важна его позиция.

- Перегрузка операций реализуется в классе Python посредством особым образом именованных методов; имена начинаются и заканчиваются двумя символами подчеркивания, чтобы сделать их уникальными. Имена не являются встроенными или зарезервированными; Python всего лишь автоматически выполняет их, когда экземпляр встречается в соответствующей операции. Сам Python определяет отображения операций на специальные имена методов.

- Перегрузка операций полезна для реализации объектов, которые имеют сходство со встроенными типами (например, последовательностей или числовых объектов, таких как матрицы), и для имитации интерфейса встроенного типа, ожидаемого порцией кода. Имитация интерфейсов встроенных типов дает возможность передавать экземпляры классов, которые также содержат информацию состояния (т.е. атрибуты, запоминающие данные между вызовами операции). Однако вы не должны применять перегрузку операций, когда будет достаточно простого именованного метода.

# Резюме

- Наиболее часто используется метод конструктора__ **init**__ ; почти каждый класс применяет этот метод для установки начальных значений атрибутов экземпляра и выполнения других задач начального запуска..

- Двумя краеугольными камнями объектно-ориентированного кода Python являются специальный аргумент **self** в функциях методов и метод конструктора __ **init**__ ; зная их, вы должны быть в состоянии читать большинство объектно-ориентированного кода на Python - помимо них это практически пакеты функций. Конечно, поиск в иерархии наследования тоже имеет значение, но self представляет автоматический объектный аргумент, а метод__ **init**__ широко распространен.

# Основы ООП

## Лекция 4. Основы проектирования классов

# Основы написания классов

Построим набор классов, делающих что-то конкретное - регистрацию и обработку сведений о людях. Вы увидите, что компоненты, которые в программировании на Python называются экземплярами и классами, часто способны исполнять такие же роли, как записи и программы в более традиционных терминах.

В частности реализуем два класса:

- **Person** - класс, который создает и обрабатывает сведения о людях;
- **Manager** - настроенная версия класса Person, которая модифицирует унаследованное поведение.

Сохраним наши экземпляры в объектно-ориентированной базе данных shelve, обеспечив их постоянство. В итоге вы сможете применять написанный код в качестве шаблона для формирования полноценной базы данных, реализованной полностью на Python.

# Шаг 1. Создание экземпляров

Первое, что мы хотим делать с помощью класса Person, связано с регистрацией основных сведений о людях - заполнением полей записей, если так понятнее.

В терминологии Python они известны как **атрибуты** объекта экземпляра и обычно создаются путем присваивания значений атрибутам self в функциях методов класса. Нормальный способ предоставления атрибутам экземпляра первоначальных значений предусматривает их присваивание через self в методе конструктора **__ init__** , который содержит код, автоматически выполняемый Python каждый раз, когда создается экземпляр.

```
# Добавление инициализации полей записи
class Person:
    def __init__(self, name, job, pay):  # Конструктор принимает три аргумента
        self.name = name                  # Заполнить поля при создании
        self.job = job                    # self - новый объект экземпляра
        self.pay = pay
```

Обратите внимание, что имена аргументов здесь встречаются дважды. Поначалу код может даже показаться несколько избыточным, но это не так. Например, аргумент **job** представляет собой локальную переменную в области видимости функции **__ init__** , но **self.job** - атрибут экземпляра, в котором передается подразумеваемый объект вызова метода. Они являются двумя разными переменными. За счет присваивания локальной переменной **job** атрибуту **self.job** посредством **self.job=job** мы сохраняем переданное значение **job** в экземпляре для последующего применения.

# Шаг 1. Создание экземпляров

Методом конструктора **__ init__** автоматически вызывается при создании экземпляра и имеет особым образом именованный первый аргумент. В целях демонстрации давайте сделаем аргумент **job** необязательным - он будет получать стандартное значение None, указывающее на то, что создаваемый экземпляр Person представляет человека, который (в текущий момент) не нанят на работу. Поскольку стандартным значением job будет **None**, тогда ради согласованности, вероятно, имеет смысл также установить стандартное значение для **pay** в **0.**

```
class Person:
    def __init__(self, name, job=None, pay=0):  # Конструктор принимает три аргумента
        self.name = name                        # Заполнить поля при создании
        self.job = job                          # self - новый объект экземпляра
        self.pay = pay
```

аргумент **self** заполняется Python автоматически для ссылки на объект экземпляра - присваивание значений атрибутам **self** присоединяет их к новому экземпляру.

# Шаг 1. Создание экземпляров

Программирование на Python в действительности представляет собой вопрос пошагового создания прототипов - вы пишете какой-то код, тестируете его, пишете дополнительный код, снова тестируете и т.д. Поскольку Python обеспечивает нас как интерактивным сеансом, так и практически немедленным учетом изменений в коде, более естественно проводить тестирование по ходу дела, нежели писать крупный объем кода и тестировать его весь сразу.

```
class Person:
    def __init__(self, name, job=None, pay=0):  # Конструктор принимает три аргумента
        self.name = name                        # Заполнить поля при создании
        self.job = job                          # self - новый объект экземпляра
        self.pay = pay


bob = Person('Bob Smith')
sue = Person('Sue Jones', job='dev', pay=100000)
print(bob.name, bob.pay)
print(sue.name, sue.pay)
```

```
Bob Smith 0
Sue Jones 100000
```

# Шаг 1. Создание экземпляров

Формально bob и sue являются объектами пространств имен - подобно всем экземплярам классов каждый из них имеет собственную независимую копию информации о состоянии, созданную классом. Из-за того, что каждый экземпляр класса располагает своим набором атрибутов **self**, классы оказываются естественным инструментом для регистрации сведений для множества объектов. Как и встроенные типы вроде списков и словарей, классы служат своего рода ***фабриками объектов***.
Организация выполнения тестовых операторов в конце файла, только когда файл запускается для тестирования, а не когда он импортируется. Именно для этого предназначена проверка атрибута **__ name__** модуля.

```
class Person:
    def __init__(self, name, job=None, pay=0):  # Конструктор принимает три аргумента
        self.name = name                        # Заполнить поля при создании
        self.job = job                          # self - новый объект экземпляра
        self.pay = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
```

Запуск файла как сценария верхнего уровня тестирует его, потому что **__ name__** является **__ main__** , но импортирование его в качестве библиотеки классов не приводит к выполнению тестов.

# Шаг 2. добавление методов, реализующих поведение

Несмотря на то что классы добавляют дополнительный уровень структуры, в конечном итоге они делают большую часть своей работы, встраивая и обрабатывая основные типы данных наподобие списков и строк; классы в действительности представляют собой лишь незначительное структурное расширение.

```
class Person:
    def __init__(self, name, job=None, pay=0):  # Конструктор принимает три аргумента
        self.name = name                        # Заполнить поля при создании
        self.job = job                          # self - новый объект экземпляра
        self.pay = pay

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.name.split()[-1])  # Извлечение фамилии из объекта
    sue.pay *= 1.10              # Предоставление этому объекту повышения
    print('%.2f' % sue.pay)
```

```
Bob Smith 0
Sue Jones 100000
Smith
110000.00
```

# Шаг 2. добавление методов, реализующих поведение

Задействуем концепцию проектирования программного обеспечения, известную как **инкапсуляция** - помещение операционной логики в оболочку интерфейсов, чтобы код каждой операции был написан только один раз в программе. Тогда если в будущем возникнет необходимость в изменении, то изменять нужно будет только одну копию. Более того, мы можем практически произвольно изменять внутренности одиночной копии, не нарушая работу кода, который ее потребляет.

Выражаясь терминами Python, мы хотим поместить код операций над объектами в методы класса, а не засорять ими всю программу. Фактически это одно из дел, с которыми классы справляются очень хорошо - вынесение кода с целью устранения избыточности и в итоге повышения удобства сопровождения. В качестве дополнительного бонуса помещение операций внутрь методов позволяет применять их к любому экземпляру класса, а не только к тем, где они были жестко закодированы для обработки.

# Шаг 2. добавление методов, реализующих поведение

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))
```

Методы представляют собой нормальные функции, которые при соединяются к классам и предназначены для обработки экземпляров этих классов. Экземпляр является подразумеваемым объектом вызова метода и передается в аргументе **self** автоматически.

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob.name, bob.pay)
    print(sue.name, sue.pay)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.10)
    print(sue.pay)
```

```
Bob Smith 0
Sue Jones 100000
Smith
110000.00
```

# Шаг 2. добавление методов, реализующих поведение

Хранящий величину заработной платы атрибут **pay** в **sue** по-прежнему остается целым числом. Можно использовать:

- round(N, 2)
- тип decimal с фиксированной точностью
- полные числа с плавающей точкой и отображать их с использованием строки формата % . 2f или {0: . 2f}

Мы выводим фамилию объекта **sue** - поскольку логика извлечения фамилии была инкапсулирована в методе, мы можем применять его к любому экземпляру класса. Как мы видели, Python сообщает методу, какой экземпляр обрабатывать, автоматически передавая его в первом аргументе, который обычно называется **self**.

• в первом вызове, **bob.lastName ()**, **bob** является подразумеваемым объектом, передаваемым **self**;
• во втором вызове, **sue.lastName()**, в **self** передается sue.

# Шаг 3. добавление методов, реализующих поведение

В нынешнем виде тестирование все еще менее удобно, чем должно быть - для трассировки объектов нам приходится вручную извлекать и вводить индивидуальные атрибуты (**bob.name**, **sue.pay**). Было бы хорошо, если бы отображение экземпляра всего сразу действительно давало какую-то полезную информацию.

Положение дел легко улучшить, задействовав перегрузку операций - написать код методов в классе, которые перехватывают и обрабатывают встроенные операции, когда выполняются на экземплярах класса. Необходимо использовать вторые по частоте применения в Python методы перегрузки операций после **__ init__** :

метод **__ repr__** , и его двойник **__ str__** .

Формально **__ str__** предпочтительнее **print**, а **___repr__** используется в качестве запасного варианта для данных ролей и во всех остальных контекстах. Хотя можно применить два метода для реализации отличающегося отображения в разных контекстах, написание кода одного лишь**__ repr__** достаточно, чтобы предоставить единственное отображение во всех случаях - вывод с помощью **print**, вложенные появления и эхо-вывод в интерактивной подсказке.

Метод конструктора**__ init__** тоже является перегрузкой операции - он выполняется автоматически во время создания для инициализации нового экземпляра. Тем не менее, конструкторы настолько распространены, что они не выглядят похожими на особый случай. Более специализированные методы вроде **__ repr__** позволяют подключаться к специфическим операциям и обеспечивать специальное поведение, когда объекты используются в таких контекстах.

# Шаг 3. добавление методов, реализующих поведение

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):
        #  return '[Person: % s, % s]' % (self.name, self.pay)
        return f'[Person: {self.name}, {self.pay}]'


if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.10)
    print(sue)
```

Обратите внимание, что в **__ repr__** для построения строки, подлежащей отображению, применяется операция % строкового форматирования либо **f-строки**; для выполнения своей работы классы используют объекты и операции встроенных типов.

# Шаг 3. добавление методов, реализующих поведение

**__ repr__** , если присутствует, то часто используется, чтобы предоставить низкоуровневое отображение объекта как в коде, а метод **__ str__** зарезервирован для более информативного отображения, дружественного к пользователям. Иногда классы предлагают и **__ str__** для дружественного к пользователям отображения, и ___герг__ с добавочными деталями для просмотра разработчиками. Поскольку операция вывода запускает **__ str__** , а интерактивная подсказка автоматически выводит результаты с помощью **__ repr__** , это можно применять для снабжения обеих целевых аудиторий подходящим отображением.

Метод **__ repr__** пригоден в большем количестве случаев отображения, в том числе при вложенных появлениях.

# Шаг 4. настройка поведения за счет создания подклассов

Сейчас наш класс создает экземпляры, снабжает поведением в методах и даже немного перегружает операции, чтобы перехватывать операции вывода в **__ repr__** . Он фактически упаковывает наши данные и логику вместе в единственный автономный программный компонент, облегчая нахождение кода и его изменение в будущем. Разрешая нам инкапсулировать поведение, он также дает возможность выносить такой код во избежание избыточности и связанных затруднений при сопровождении. В качестве следующего шага применения и настройки нашего класса **Person**, расширяя имеющуюся программную иерархию определим подкласс класса **Person** по имени **Manager**, который замещает унаследованный метод **giveRaise** более специализированной версией.

```
class Manager(Person):
```

Существуют два способа написания кода для такой настройки класса Manager: хороший и плохой. Начнем с плохого способа, т.к. он может быть чуть легче для понимания. Плохой способ заключается в том, чтобы вырезать код **giveRaise** из класса **Person** и вставить его в класс **Manager**, после чего модифицировать:

```
class Manager(Person):
    def giveRaise(self, percent, bonus=0.10):
        self.pay = int(self.pay * (1 + percent + bonus))
```

# Шаг 4. настройка поведения за счет создания подклассов

Проблема здесь имеет очень общий характер: всякий раз, когда вы копируете код посредством вырезания и вставки, то по существу **удваиваете** объем работ по сопровождению в будущем.
Несмотря на то что пример является простым и искусственным, он демонстрирует универсальную проблему - всякий раз, когда возникает искушение программировать путем копирования кода, вероятно, стоит поискать более подходящий подход.
**Хороший способ** в Python предусматривает вызов исходной версии напрямую с дополненными аргументами.

```
class Manager(Person):
    def giveRaise(self, percent, bonus=0.10):
        Person.giveRaise(self, percent + bonus)
```

В коде задействован тот факт, что метод класса всегда может быть вызван либо через экземпляр (обычный способ, когда Python автоматически передает экземпляр аргументу **self**), либо через класс (менее распространенная схема, при которой экземпляр должен передаваться вручную).
В случае вызова через имя класса вы должны самостоятельно передавать экземпляр атрибуту self; для кода внутри метода вроде giveRaise аргумент self уже является объектом, на котором произведен вызов, и отсюда экземпляром, подлежащим передаче.

# Шаг 4. настройка поведения за счет создания подклассов

При создании экземпляра **Manager** мы передаем имя, а также необязательную должность и размер заработной платы - поскольку конструктор **__ init__** в классе **Manager** отсутствует, он наследует его от **Person**.

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):
        return f'[Person: {self.name}, {self.pay}]'
```

```
class Manager(Person):
    def giveRaise(self, percent, bonus=0.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)
    tom.giveRaise(0.10)
    print(tom.lastName())
    print(tom)
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

# Шаг 4. настройка поведения за счет создания подклассов

При создании экземпляра **Manager** мы передаем имя, а также необязательную должность и размер заработной платы - поскольку конструктор **__ init__** в классе **Manager** отсутствует, он наследует его от **Person**.

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):
        return f'[Person: {self.name}, {self.pay}]'
```

```
class Manager(Person):
    def giveRaise(self, percent, bonus=0.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)
    tom.giveRaise(0.10)
    print(tom.lastName())
    print(tom)
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
```

# Шаг 4. настройка поведения за счет создания подклассов

Экземпляр **Manager** по имени **tom** получает повышение на 10%, то на самом деле становится обладателем 20% (его оплата возрастает с 50 000 до 60 000), потому что настроенная версия **giveRaise** из **Manager** выполняется только для него. Также обратите внимание, что вывод экземпляра tom как единого целого в конце тестового кода отображает элегантный формат, определенный в методе **__ repr__** класса **Person:** объекты **Manager** получают код **__ repr__** , **lastName** и метода конструктора **__ init__** “бесплатно” от Person благодаря наследованию.

Для расширения унаследованных методов в примерах вызывались исходные методы через имя суперкласса: Person.giveRaise (...).

В Python также имеется встроенная функция **super**, которая позволяет вызывать методы суперкласса более обобщенно. Однако, она шероховато работает с перегрузкой операций в Python и не всегда хорошо сочетается с традиционно реализованным множественным наследованием, где обращения к единственному суперклассу будет недостаточно.

В защиту вызова **super** следует отметить, что с ним тоже связан допустимый сценарий использования (совместная координация одинаково именованных методов в деревьях множественного наследования). Однако он опирается на упорядочение классов MRO (Method Resolution Order — порядок распознавания методов), которое многие считают экзотическим и искусственным; нереалистично предполагает, что будет надежно применяться универсальное развертывание; не полностью поддерживает замещение методов и списки аргументов переменной длины; к тому же для многих использование сценария, редко встречающегося в реальном коде на Python, представляется невразумительным решением.

# Шаг 4. настройка поведения за счет создания подклассов

В приведенном ниже коде **obj** является экземпляром либо **Person**, либо **Manager**, и Python выполняет соответствующий метод **giveRaise** автоматически - нашу исходную версию в классе Person для **bob** и **sue** и настроенную версию в **Manager** для **tom**. Отследите вызовы методов самостоятельно, чтобы увидеть, каким образом Python выбирает правильный метод **giveRaise** для каждого объекта. Это всего лишь демонстрация в работе понятия полиморфизма в Python.

```
if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.10)
    print(sue)
    tom = Manager('Tom Jones', 'mgr', 50000)
    tom.giveRaise(0.10)
    print(tom.lastName())
    print(tom)
    print('--All three--')
    for obj in (bob, sue, tom):
        obj.giveRaise(.10)
        print(obj)
```

```
[Person: Bob Smith, 0]
[Person: Sue Jones, 100000]
Smith Jones
[Person: Sue Jones, 110000]
Jones
[Person: Tom Jones, 60000]
--All three--
[Person: Bob Smith, 0]
[Person: Sue Jones, 121000]
[Person: Tom Jones, 72000]
```

# Шаг 4. настройка поведения за счет создания подклассов

В приведенном ниже коде **obj** является экземпляром либо **Person**, либо **Manager**, и Python выполняет соответствующий метод **giveRaise** автоматически - нашу исходную версию в классе Person для **bob** и **sue** и настроенную версию в **Manager** для **tom**.
Отследите вызовы методов самостоятельно, чтобы увидеть, каким образом Python выбирает правильный метод **giveRaise** для каждого объекта. Это всего лишь демонстрация в работе понятия полиморфизма в Python.
Практический эффект данного кода заключается в том, что экземпляр sue получает дополнительно 10%, но экземпляр tom - 20%, потому что выбор метода giveRaise координируется на основе типа объекта. Как уже известно, полиморфизм является центральной частью гибкости Python. Скажем, передача любого из трех объектов функции, которая вызывает метод **giveRaise**, дала бы такой же результат: в зависимости от типа переданного объекта автоматически выполнилась бы подходящая версия. С другой стороны, вывод выполняет тот же самый метод **__repr__** для всех трех объектов, т.к. он реализован только один раз в **Person**. Класс **Manager** специализирует и применяет код, который мы первоначально написали в Person.

# Шаг 4. настройка поведения за счет создания подклассов

Мы можем добавить в класс **Manager** уникальные методы, которые отсутствуют в **Person**, если экземпляры **Manager** требуют чего-то совершенно другого. Сказанное иллюстрируется в следующем коде, где **giveRaise** переопределяет метод суперкласса с целью его настройки, но **someThingElse** определяет кое-что новое для расширения:

```
class Person:
    def lastName(self): ...
    def giveRaise(self): ...
    def __repr__ (self): ...
class Manager(Person):                  # Наследование
    def giveRaise(self, ...):...       # Настройка
    def someThingElse(self, ...): ...  # Расширение
tom = Manager()
tom.lastName()       # Буквальное наследование
tom.giveRaise()      # Настроенная версия
tom.someThingElse()  # Метод расширения
print(tom)           # Унаследованный перегруженный метод
```

Дополнительные методы наподобие **someThingElse** в показанном коде расширяют существующее программное обеспечение и доступны только для объектов **Manager**, но не **Person**.

# Объектно-ориентированное программирование: основная идея

***Основная идея, лежащая в основе ООП в целом:***

мы программируем путем настройки того, что уже было сделано, а не копирования либо изменения существующего кода. На первый взгляд это не всегда очевидный выигрыш, особенно с учетом требований по написанию добавочного кода классов. Но, в общем, стиль программирования, подразумеваемый классами, может радикально сократить время разработки по сравнению с другими подходами.

- Несмотря на то что мы могли бы просто реализовать класс **Manager** с нуля как новый, независимый код, нам пришлось бы повторно реализовать все линии поведения из **Person**, которые остались такими же в **Manager**.
- Хотя мы могли бы просто изменить существующий код класса Person на месте для удовлетворения требований операции **giveRaise** из **Manager**, это вероятно нарушило бы работу кода в местах, где по-прежнему необходимо исходное поведение **Person**.
- Несмотря на то что мы могли бы просто скопировать класс **Person** полностью, переименовать копию на **Manager** и изменить код операции **giveRaise**, это ввело бы избыточность кода, приводя к удваиванию работы по сопровождению - изменения, вносимые в будущем в класс **Person**, не будут подхватываться автоматически, и их придется вручную распространять на код **Manager**. Как обычно, подход с вырезанием и вставкой в текущий момент может выглядеть быстрым, но он удваивает объем работы в будущем.

Настраиваемые иерархии, которые мы можем строить с помощью классов, обеспечивают гораздо лучшее решение для программного обеспечения, развивающегося с течением времени. Другие инструменты в Python такой режим разработки не поддерживают.

# Шаг 5. настройка конструкторов

Настроим логику конструктора для объектов **Manage**r таким образом, чтобы предоставлять название должности автоматически. В переводе на код нам нужно переопределить метод **__ init__** в классе **Manager** с целью предоставления строки **mg**r. Как и в настройке giveRaise, нам также необходимо выполнять исходный метод **__ init__** из **Person**, вызывая его через имя класса, чтобы он по-прежнему инициализировал атрибуты информации о состоянии наших объектов.

```
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):
        return f'[Person: {self.name}, {self.pay}]'
```

# Шаг 5. настройка конструкторов

```
class Manager(Person):
    def __init__(self, name, pay): # Переопределить конструктор
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=0.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__':
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)  # Название должности не требуется:
    tom.giveRaise(0.10)                # устанавливается классом
    print(tom.lastName())
    print(tom)
```

При расширении конструктора **__init__** применяется та же методика, которую ранее использовали для **giveRaise** - выполняем версию из суперкласса, путем вызова через имя класса напрямую и явно передаем экземпляр **self**. Хотя конструктор имеет странное имя, эффект идентичен. Поскольку нужно также выполнить логику создания экземпляра **Person** (для инициализации атрибутов экземпляра), мы действительно обязаны вызывать ее таким способом; иначе экземпляры не получат присоединенных атрибутов.

# Шаг 5. настройка конструкторов

Подобного рода вызов конструкторов суперклассов из переопределенных версий оказывается весьма распространенным стилем программирования в Python. Сам по себе Python применяет наследование для поиска и вызова только одного метода **__ init__** на стадии конструирования - расположенного ниже всех в дереве классов. Если необходимо, чтобы на стадии конструирования выполнялись методы**__ init__** , находящиеся выше в дереве классов (обычно так и есть), тогда вы должны вызывать их вручную, как правило, через имя суперкласса. Положительный аспект здесь в том, что вы можете явно указывать аргумент для передачи конструктору суперкласса или даже вообще не вызывать его: отказ от вызова конструктора суперкласса позволяет вместо расширения полностью замещать его логику.

# ООП проще чем кажется ☺

Классы в примере реализуют:

- создание экземпляров - заполнение атрибутов экземпляров;
- методы, реализующие поведение - инкапсуляция логики в методах класса;
- перегрузка операций - обеспечение линии поведения для встроенных операций вроде вывода;
- настройка поведения - переопределение методов в подклассах для их специализации;
- настройка конструкторов - добавление логики инициализации к шагам суперкласса.

Большинство перечисленных концепций основаны всего лишь на трех простых идеях: поиск атрибутов в иерархии наследования в форме деревьев объектов, особый аргумент **self** в методах и автоматическое сопоставление перегруженных операций с методами.

# Шаг 5. настройка конструкторов Альтернатива

Существует стиль программирования предусматривающий вложение объектов друг в друга для построения составных объектов. Альтернатива реализует такой прием за счет использования метода для перегрузки операции **__ getattr__**, чтобы перехватывать извлечения неопределенных атрибутов и делегировать выполнение работы внедренному объекту посредством встроенной функции **getattr**.

```
class Manager:
    def __init__ (self, name, pay) :
        self.person = Person(name, 'mgr', pay) # Внедрить объект Person
    def giveRaise(self, percent, bonus=.10):
        self.person.giveRaise(percent + bonus) # Перехватить и делегировать
    def __getattr__ (self, attr):
        return getattr (self.person, attr) # Делегировать все остальные атрибуты
    def __repr__ (self) :
        return str(self.person) # Снова должен быть перегружен
                                # (в Python 3.X)
```

Путем комбинирования указанных инструментов метод **giveRaise** по-прежнему обеспечивает настройку, изменяя аргумент, который передается внедренному объекту. В действительности **Manager** становится уровнем контроллера, передающим вызовы вниз внедренному объекту, а не вверх методам суперкласса. альтернативная версия **Manager** иллюстрирует общий стиль программирования, обычно известный как делегирование - структура на основе составного объекта, которая управляет внедренным объектом и передает ему вызовы методов.

# Шаг 5. настройка конструкторов Альтернатива

Шаблон удалось внедрить в нашем примере, но он потребовал почти в два раза больше кода и он не настолько хорошо подходит для задействованных видов настроек, как наследование.

Тем не менее, внедрение объектов и основанные на нем паттерны проектирования могут очень хорошо подходить, когда внедренные объекты требуют более ограниченного взаимодействия с контейнером, чем подразумевает прямая настройка. Скажем, уровень контроллера, или посредника, подобный альтернативной версии Manager, может оказаться полезным, когда мы хотим адаптировать класс к ожидаемому интерфейсу, который он не поддерживает, либо отследить или проверить достоверность обращений к методам другого объекта.

Кроме того, гипотетический класс **Department**, подобный показанному ниже, мог бы агрегировать другие объекты с целью их трактовки как набора.

# Шаг 5. настройка конструкторов
# Альтернатива

```python
class Person:
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay

    def lastName(self):
        return self.name.split()[-1]

    def giveRaise(self, percent):
        self.pay = int(self.pay * (1 + percent))

    def __repr__(self):
        return f'[Person: {self.name}, {self.pay}]'


class Manager(Person):
    def __init__(self, name, pay): # Переопределить конструктор
        Person.__init__(self, name, 'mgr', pay)
    def giveRaise(self, percent, bonus=0.10):
        Person.giveRaise(self, percent + bonus)
```

# Шаг 5. настройка конструкторов Альтернатива

```
class Department:
    def __init__ (self, *args):
        self.members = list(args)
    def addMember(self, person):
        self.members.append(person)
    def giveRaises(self, percent):
        for person in self.members:
            person.giveRaise(percent)
    def showAll(self):
        for person in self.members:
            print(person)

if __name__ == '__main__' :
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    tom = Manager('Tom Jones', 50000)
    development = Department(bob, sue)  # Внедрить объекты в составной объект
    development.addMember(tom)
    development.giveRaises(0.10)  # Выполняет giveRaise внедренных объектов
    development.showAll()         # Выполняет__repr__ внедренных объектов
```

# Шаг 5. настройка конструкторов Альтернатива

Интересно отметить, что в коде применяются **наследование** и **композиция** - **Department** является составным объектом, который внедряет и управляет другими объектами для агрегирования, но сами внедренные объекты **Person** и **Manager** для настройки используют наследование. В качестве еще одного примера графический пользовательский интерфейс может похожим образом применять наследование для настройки поведения или внешнего вида меток и кнопок, а также композицию для построения более крупных пакетов внедряемых виджетов вроде форм ввода, калькуляторов и текстовых редакторов. Используемая структура классов зависит от объектов, которые вы пытаетесь моделировать - по сути, такая возможность моделирования сущностей реального мира считается одной из сильных сторон ООП.

# Шаг 6: использование инструментов интроспекции

**Интроспекция** - это способность объекта во время выполнения получить тип, доступные атрибуты и методы, а также другую информацию, необходимую для выполнения дополнительных операций с объектом.

Две проблемы, которые должны быть улажены до практического применения классов:

- если вы посмотрите на отображение объектов в той форме, как есть, то заметите, что при выводе экземпляр **tom** класса **Manager** помечается как **Person**. Формально это нельзя считать некорректным, поскольку **Manager** является настроенной и специализированной версией **Person**. Тем не менее, правильнее было бы отображать объект с самым специфическим (т.е. расположенным ниже всех) классом: тем, из которого объект создан.
- текущий формат отображения показывает только атрибуты, которые мы включили в **__ repr__** , что может не учитывать будущие цели. Например, мы пока не в состоянии проверять, что название должности **tom** было корректно установлено в mgr конструктором класса **Manager**, т.к. метод **__ repr__** , реализованный для **Person**, не выводит данное поле. Хуже того, если мы когда-либо расширим или по-другому изменим набор атрибутов, назначаемых нашим объектам в **__ init__** , нам придется не забыть об обновлении также и метода **__ repr__** для отображения новых имен, иначе со временем он утратит синхронизацию.

# Шаг 6: использование инструментов интроспекции

Мы можем решить обе проблемы с помощью инструментов интроспекции Python - специальных атрибутов и функций, которые обеспечивают доступ к внутренностям реализаций объектов. Эти инструменты довольно сложны и, как правило, используются теми, кто создает инструменты для применения другими программистами, а не программистами, которые разрабатывают прикладные приложения**.**

Встроенный атрибут **экземпляр.__ class__** предоставляет ссылку из экземпляра на класс, из которого экземпляр был создан. В свою очередь классы имеют атрибут **__name__** , в точности как модули, и последовательность **__ bases__** , обеспечивающую доступ к суперклассам. Мы можем их здесь применять для вывода имени класса, из которого создавался экземпляр, вместо жестко закодированного имени.

Встроенный атрибут **объект.__ dict__** предоставляет словарь с одной парой “ключ-значение” для каждого атрибута, присоединенного к объекту пространства имен (включая модули, классы и экземпляры). Поскольку это словарь, мы можем извлекать список его ключей, индексировать по ключу, проходить по его ключам и т.д. для обработки всех атрибутов обобщенным образом.

# Шаг 6: использование инструментов интроспекции

Некоторые атрибуты, доступные из экземпляра, могут не храниться в словаре **__ dict__** , если в классе экземпляра определено средство **__ slots__** . Поскольку слоты в действительности принадлежат к классам, а не экземплярам, в любом случае они используются редко. Однако имейте в виду, что некоторые программы могут нуждаться в перехвате исключений из-за отсутствующего словаря **__dict__** либо применять **hasattr** для проверки или getattr со стандартным значением, если их пользователи развернули слоты.

# Шаг 6: использование инструментов интроспекции

Мы можем задействовать эти интерфейсы в суперклассе, который отображает точные имена классов и форматирует все атрибуты экземпляра любого класса. Из-за того, что его перегруженный метод отображения **__repr__** применяет обобщенные инструменты интроспекции, он будет работать на любом экземпляре независимо от набора атрибутов экземпляра. И поскольку он является классом, то автоматически становится универсальным инструментом форматирования: благодаря наследованию его можно соединять с любым классом, в котором желательно использовать такой формат отображения.

# Шаг 6: использование инструментов интроспекции

```
class AttrDisplay:
    """
    Предоставляет наследуемый метод перегрузки отображения, который показывает
    экземпляры с их именами классов и пары имя=значение для каждого атрибута,
    сохраненного в самом экземпляре (но не атрибутов, унаследованных от его классов) .
    Может соединяться с любым классом и будет работать на любом экземпляре.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__ ) :
            attrs.append(f'{key}={getattr(self, key)}')
        return ', '.join(attrs)
    def __repr__(self) :
        return f'[{self.__class__ .__name__}:{self.gatherAttrs()}]'
```

# Шаг 6: использование инструментов интроспекции

```
if __name__ == '__main__':
    class TopTest(AttrDisplay):
        count = 0
        def __init__(self) :
            self.attr1 = TopTest.count
            self.attr2 = TopTest.count + 1
            TopTest.count += 2
    class SubTest(TopTest):
        pass
    X,Y = TopTest(), SubTest()  # Создать два экземпляра
    print(X)                    # Показать все атрибуты экземпляров
    print(Y)                    # Показать имя класса, расположенного ниже всех
```

```
[TopTest:attr1=0, attr2=1]
[SubTest:attr1=2, attr2=3]
```

поскольку класс применяет **__repr__** вместо **___str__** , его отображение используется во всех контекстах, но клиенты также не будут иметь возможности предоставить альтернативное низкоуровневое отображение - они по-прежнему могут добавлять метод **__str__** , но он применяется только к **print** и **str**. В более универсальном инструменте использование взамен**__str__** ограничивает границы отображения, но оставляет клиенту возможность добавления**__ repr__** для вторичного отображения в интерактивной подсказке и во вложенных появлениях.

# Атрибуты экземпляра или атрибуты класса

Изучив результаты **__dict__** вы не увидите атрибутов, унаследованных экземпляром от классов, расположенных выше в дереве. Унаследованные атрибуты класса присоединяются только к классу, они не копируются в экземпляры. Если вы когда-нибудь захотите включить также унаследованные атрибуты, тогда можете подняться по ссылке **__class__** к классу экземпляра, использовать **__dict__** для извлечения атрибутов класса и затем пройти через атрибут **__bases__** класса, чтобы подняться к более высоко расположенным суперклассам, при необходимости повторяя процесс. Если применять простой код, тогда выполнение встроенного вызова **dir** на экземпляре вместо применения **__dict__** и подъема имело бы во многом такой же эффект, т.к. результаты **dir** включают унаследованные имена в отсортированном списке результатов.

```
print(dir(bob))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
 '__weakref__', 'giveRaise', 'job', 'lastName', 'name', 'pay']
```

```
print(list(bob.__dict__.keys()))
```

```
['name', 'job', 'pay']
```

```
print(list(name for name in dir(bob) if not name.startswith ('__')))
```

```
['giveRaise', 'job', 'lastName', 'name', 'pay']
```

# Атрибуты экземпляра или атрибуты класса

В текущем виде подклассам доступны другие методы: либо для прямых вызовов, либо для настройки вышеуказанным образом.

Чтобы свести к минимуму вероятность конфликтов имен, программисты на Python часто снабжают имена методов, не предназначенных для внешнего применения, префиксом в виде одиночного подчеркивания **_gatherAttrs**. Хотя такой прием не обеспечивает полной защиты от неправильного использования, но его обычно достаточно и он представляет собой распространенное соглашение по именованию для внутренних методов класса.

Лучшим, но менее часто применяемым решением было бы использование двух подчеркиваний в начале имени метода**:__gatherAttrs**. Интерпретатор Python расширяет имена с двумя подчеркиваниями с целью включения имени содержащего их класса, которое делает имена по-настоящему уникальными при поиске в иерархии наследования. Такое средство обычно называется **псевдозакрытыми атрибутами класса**.

# Финальная форма

Теперь для применения построенного обобщенного инструмента в наших классах необходимо лишь импортировать его из модуля, скомбинировать с классом верхнего уровня, используя наследование, и избавиться от специфического метода **__repr__** , который мы реализовали ранее. Новый метод перегрузки отображения будет наследован всеми экземплярами **Person**, а также **Manager**; класс **Manager** получает **__repr__** от класса **Person**, который приобретает его от класса **AttrDisplay**, реализованного в другом модуле.

```
class AttrDisplay:
    """
    Предоставляет наследуемый метод перегрузки отображения, который показывает
    экземпляры с их именами классов и пары имя=значение для каждого атрибута,
    сохраненного в самом экземпляре (но не атрибутов, унаследованных от его классов)
    Может соединяться с любым классом и будет работать на любом экземпляре.
    """
    def gatherAttrs(self):
        attrs = []
        for key in sorted(self.__dict__ ) :
            attrs.append(f'{key}={getattr(self, key)}')
        return ', '.join(attrs)
    def __repr__(self) :
        return f'[{self.__class__ .__name__}:{self.gatherAttrs()}]'
```

# Финальная форма

```
# Использовать обобщенный инструмент отображения
from classtools import AttrDisplay
# Комбинирование с классом верхнего уровня
class Person (AttrDisplay) :
    """
    Создает и обрабатывает записи о людях
    """
    def __init__(self, name, job=None, pay=0):
        self.name = name
        self.job = job
        self.pay = pay
    def lastName(self) : # Предполагается, что фамилия указана последней
        return self.name.split()[-1]
    def giveRaise(self, percent): # Процент должен находиться между 0 и 1
        self.pay = int(self.pay * (1 + percent))
```

# Финальная форма

```
class Manager(Person):
    """
    Настроенная версия Person со специальными требованиями
    """
    def __init__(self, name, pay) :
        Person.__init__(self, name, 'mgr', pay) # Название должности подразумевается
    def giveRaise(self, percent, bonus=.10):
        Person.giveRaise(self, percent + bonus)

if __name__ == '__main__' :
    bob = Person('Bob Smith')
    sue = Person('Sue Jones', job='dev', pay=100000)
    print(bob)
    print(sue)
    print(bob.lastName(), sue.lastName())
    sue.giveRaise(0.10)
    print(sue)
    tom = Manager('Tom Jones', 50000)
    tom.giveRaise(0.10)
    print(tom.lastName())
    print(tom)
```

# Финальная форма

Запустив код прямо сейчас, мы увидим все атрибуты наших объектов, а не только те, которые жестко закодированы в исходном методе **__ repr__** . И наша финальная проблема решена: из-за того, что **AttrDisplay** напрямую берет имена классов из экземпляра **self**, каждый объект выводится с именем своего ближайшего (находящегося ниже всех в дереве) класса. Теперь **tom** отображается как объект **Manager**, а не **Person**, и мы можем окончательно удостовериться в корректности заполнения его названия должности в конструкторе **Manager**

```
[Person:job=None, name=Bob Smith, pay=0]
[Person:job=dev, name=Sue Jones, pay=100000]
Smith Jones
[Person:job=dev, name=Sue Jones, pay=110000]
Jones
[Manager:job=mgr, name=Tom Jones, pay=60000]
```

В более широком плане наш класс отображения атрибутов стал универсальным инструментом, который посредством наследования мы можем комбинировать с любым классом, чтобы задействовать определяемый им формат отображения. Кроме того, все его клиенты будут автоматически подхватывать будущие изменения, вносимые в наш инструмент.

# Сохранение в базе данных

Теперь располагаем двухмодульной системой, которая не только реализует первоначальные проектные цели по представлению сведений о людях, но и предлагает универсальный инструмент отображения атрибутов, подходящий для применения в других программах. За счет помещения функций и классов в файлы модулей мы гарантируем естественную поддержку ими многократного использования. И за счет реализации программного обеспечения в виде классов мы заставляем их естественным образом поддерживать расширение.

Тем не менее, хотя наши классы работают, как было запланировано, создаваемые ими объекты не являются настоящими записями базы данных. То есть если мы уничтожим сеанс Python, то наши экземпляры исчезнут - они представляют собой временные объекты в памяти и не сохраняются на более постоянном носителе вроде файла, поэтому не будут доступными при будущих запусках программы. Оказывается, сделать объекты экземпляров постоянными несложно с помощью средства Python под названием постоянство объектов, которое обеспечивает существование объектов после прекращения работы создавшей их программы

# Модуль pickle

Модуль **pickle** является своего рода суперобщим инструментом форматирования и деформатирования объектов: он способен преобразовывать практически произвольный объект Python из памяти в строку байтов, которую позже можно использовать для воссоздания первоначального объекта в памяти. Модуль **pickle** может обрабатывать почти любой создаваемый вами объект - списки, словари, их вложенные комбинации и экземпляры классов. Последние особенно удобны, поскольку они предоставляют данные (атрибуты) и поведение (методы); на самом деле такое сочетание можно считать грубым эквивалентом “записей” и “программ”. Из-за такой универсальности модуля **pickle** он может заменить дополнительный код, который пришлось бы иначе писать для создания и разбора специальных представлений объектов внутри текстовых файлов. Сохраняя полученную посредством **pickle** строку объекта, вы фактически делаете его постоянным: просто загрузите и с помощью **pickle** воссоздайте исходный объект.

# Модуль shelve

Модуль **shelve** предлагает добавочный уровень структуры, который позволяет хранить объекты, обработанные **pickle** по ключу. Модуль **shelve** транслирует объект в строку посредством **pickle** и сохраняет полученную строку под ключом в файле **dbm**; при последующей загрузке **shelve** извлекает строку по ключу и воссоздает исходный объект в памяти с помощью **pickle**. Все это кажется запутанным, но для вашего сценария хранилище **shelve** обработанных посредством **pickle** объектов выглядит подобно словарю - вы индексируете по ключам для извлечения, присваиваете по ключам для сохранения и используете такие словарные инструменты, как **len**, **in** и **dict**, **keys** для получения информации. Хранилища **shelve** автоматически отображают словарные операции на объекты, хранящиеся в файле.

Единственное отличие между хранилищем **shelve** и нормальным словарем в коде связано с тем, что вы обязаны сначала открывать хранилище и закрывать его после внесения изменений. Совокупный эффект в том, что хранилище **shelve** предлагает простую базу данных для сохранения и извлечения собственных объектов Python по ключам, что делает их постоянными между запусками программы. Хранилище **shelve** не поддерживает инструменты запросов, такие как язык SQL, и оно лишено ряда расширенных возможностей, имеющихся в базах данных производственного уровня (вроде подлинной обработки транзакций). Но собственные объекты Python, сохраненные в хранилище **shelve**, могут обрабатываться с привлечением всей мощи языка Python после того, как будут извлечены обратно по ключу.

# Сохранение объектов в базе данных shelve

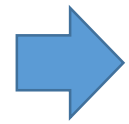
Импортируем наши классы в новый файл, чтобы создать несколько экземпляров, подлежащих сохранению. Ранее для загрузки класса в интерактивной подсказке мы применяли оператор **from**, но на самом деле, как и с функциями и остальными переменными, существуют два способа загрузки класса из файла (имена классов являются переменными, похожими на любые другие, и в этом контексте вообще нет ничего магического).

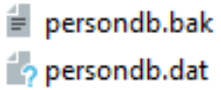
Для загрузки класса в сценарии мы будем использовать оператор **from**, т.к. он требует чуть меньшего объема набора.

Когда у нас есть несколько экземпляров, сохранить их в хранилище **shelve** практически тривиально. Мы всего лишь импортируем модуль **shelve**, открываем новое хранилище с внешним именем, присваиваем объекты по ключам в хранилище и по завершении закрываем хранилище, потому что в него были внесены изменения.

# Сохранение объектов в базе данных shelve

```
from person import Person, Manager  # Загрузить наши классы
bob = Person('Bob Smith')           # Создать объекты для сохранения
sue = Person('Sue Jones', job='dev', pay=100000)
tom = Manager('Tom Jones', 50000)

import shelve
db = shelve.open('persondb')    # Имя файла, в котором хранятся объекты
for obj in (bob, sue, tom):     # Использовать атрибут name объекта в качестве ключа
    db[obj.name] = obj          # Сохранить объект в shelve по ключу
db.close()                      # Закрыть после внесения изменений
```






Обратите внимание на то, как мы присваиваем объекты хранилищу **shelve** с применением собственных имен объектов в качестве ключей. Мы поступаем так ради удобства; в хранилище **shelve** ключом может быть любая строка, в том числе та, которую мы могли бы создать для обеспечения уникальности, используя такие средства, как идентификаторы процессов и отметки времени (библиотечные модули **os** и time). Единственная норма - ключи обязаны быть строками и должны быть уникальными, т.к. мы можем сохранять только один объект на ключ, хотя этот объект может быть списком, словарем или другим объектом, содержащим внутри себя множество объектов. Фактически значения, сохраняемые под ключами, могут быть объектами Python почти любого вида - встроенными типами вроде строк, списков и словарей, а также экземплярами определяемых пользователем классов, вложенными комбинациями всех подобных типов и т.д. Скажем, атрибуты name и job наших объектов могли бы быть вложенными словарями и списками.

# Сохранение объектов в базе данных shelve

При желании можно просмотреть файлы **shelve** в проводнике Windows или в командной оболочке Python, но они представляют собой двоичные файлы хеш-данных, большая часть содержимого которых имеет мало смысла за рамками контекста модуля **shelve.**

Стандартный библиотечный модуль **glob** в Python позволяет получать в коде перечни файлов в каталогах, чтобы проверять наличие в них файлов, и мы можем открывать файлы в текстовом или двоичном режиме для исследования строк и байтов.

```
import glob
print(glob.glob('person*'))
```

```
['person.py', 'persondb.bak', 'persondb.dat', 'persondb.dir']
```

```
print(open('persondb.dir').read())
```

```
'Bob Smith', (0, 80)
'Sue Jones', (512, 92)
'Tom Jones', (1024, 91)
```

```
print(open('persondb.dat', 'rb').read())
```

```
b'\x80\x03cperson\nPerson\nq\x00)\x81q\x01}q\x02(X\x04\
```

# Сохранение объектов в базе данных shelve

Напишем написать еще один сценарий для исследованиея хранилища **shelve**. Поскольку хранилища **shelve** представляют собой объекты Python, содержащие другие объекты Python, мы можем обрабатывать их с помощью нормального синтаксиса Python и режимов разработки.

```
import shelve
db = shelve.open('persondb') # Повторно открыть хранилище shelve
print(len(db))
print(list(db.keys()))       # Ключи являются индексом
bob = db['Bob Smith']        # Извлечь объект bob по ключу
print(bob)                   # Выполняет __repr__ из AttrDisplay
```

```
3
['Bob Smith', 'Sue Jones', 'Tom Jones']
```

```
[Person:job=None, name=Bob Smith, pay=0]
```

```
for key in db:
    print(key, '=>', db[key])  # Проход, извлечение, вывод

for key in sorted(db):         # Проход по сортированным ключам
    print(key, '=>', db[key])
```

```
Bob Smith => [Person:job=None, name=Bob Smith, pay=0]
Sue Jones => [Person:job=dev, name=Sue Jones, pay=100000]
Tom Jones => [Manager:job=mgr, name=Tom Jones, pay=50000]
```

# Сохранение объектов в базе данных shelve

Обратите внимание, что для загрузки либо использования сохраненных объектов импортировать классы **Person** или **Manager** необязательно. Скажем, мы можем свободно вызвать метод **lastName** объекта **bob** и автоматически получить специальный формат отображения, несмотря на отсутствие в области видимости класса **Person**, к которому **bob** относится. Подобное возможно потому, что при сохранении посредством **pickle** экземпляра класса Python записывает его атрибуты экземпляра **self** вместе с именем класса, из которого он был создан, и модуля, где класс находится. Когда объект **bob** позже извлекается из хранилища **shelve** и воссоздается с помощью **pickle**, то Python автоматически повторно импортирует класс и связывает с ним **bob**.

Результатом такой схемы оказывается то, что экземпляры класса автоматически обзаводятся поведением их класса при загрузке в будущем. Мы должны импортировать классы, только чтобы создать новые экземпляры, но не обрабатывать существующие.

# Сохранение объектов в базе данных shelve

- Недостаток в том, что классы и файлы их модулей обязаны быть импортируемыми, когда экземпляр позже загружается. Более формально классы, допускающие обработку посредством pickle, должны записываться на верхнем уровне файла модуля, который доступен в каталоге из пути поиска модулей **sys.path** (и не находиться в модуле **__ main__** самого верхнего файла сценария, если только при использовании они не всегда присутствуют в данном модуле). Из-за такого требования к внешнему файлу модуля в некоторых приложениях принимается решение обрабатывать с помощью **pickle** более простые объекты, подобные словарям или спискам, особенно если они передаются по сети.
- Преимущество в том, что изменения в файле исходного кода класса автоматически подхватываются, когда экземпляры этого класса загружаются снова; часто нет необходимости обновлять сами сохраненные объекты, т.к. обновление кода их класса изменяет поведение.

# Обновление объектов в хранилище shelve

Напишем скрипт, который выводит содержимое базы данных и каждый раз предоставляет повышение одному из сохраненных объектов. Если вы отследите происходящее, то заметите, что мы получаем много возможностей “бесплатно” - вывод наших объектов автоматически задействует универсальный метод перегрузки **__repr__** , и мы предоставляем повышения, вызывая реализованный ранее метод **giveRaise**. Все они “просто работают” для объектов, основанных на модели наследования ООП, даже когда объекты располагаются в файле.

```
import shelve
db = shelve.open('persondb')      # Заново открыть хранилище shelve
                                  # с тем же самым именем файла
for key in sorted(db):            # Проход для отображения объектов из базы данных
    print(key, '\t=>', db[key])   # Выводит в специальном формате
sue = db['Sue Jones']             # Индексация по ключу с целью извлечения
sue.giveRaise(.10)                # Обновление в памяти, используя метод класса
db['Sue Jones'] = sue             # Присваивание по ключу для обновления в хранилище shelve
db.close()                        # Закрытие после внесения обновлений
```

```
Bob Smith    => [Person:job=None, name=Bob Smith, pay=0]
Sue Jones    => [Person:job=dev, name=Sue Jones, pay=100000]
Tom Jones    => [Manager:job=mgr, name=Tom Jones, pay=50000]
```

# Обновление объектов в хранилище shelve

Поскольку сценарий выводит содержимое базы данных при начальном запуске, чтобы увидеть, изменился ли объект, мы должны запустить его, по меньшей мере, дважды. Далее сценарий показан в действии, отображая все записи и увеличивая заработную плату объекта sue каждый раз, когда он запускается (довольно хороший сценарий для sue...).

```
Bob Smith   => [Person:job=None, name=Bob Smith, pay=0]
Sue Jones   => [Person:job=dev, name=Sue Jones, pay=110000]
Tom Jones   => [Manager:job=mgr, name=Tom Jones, pay=50000]
```

```
Bob Smith   => [Person:job=None, name=Bob Smith, pay=0]
Sue Jones   => [Person:job=dev, name=Sue Jones, pay=121000]
Tom Jones   => [Manager:job=mgr, name=Tom Jones, pay=50000]
```

# Применение

***Графические пользовательские интерфейсы***

В том виде, как есть, мы можем обрабатывать базу данных только с помощью основанного на командах интерфейса интерактивной подсказки и сценариев.

Мы могли бы потрудиться над расширением удобства эксплуатации нашей базы данных объектов, добавив настольный графический пользовательский интерфейс для просмотра и обновления ее записей. Графические пользовательские интерфейсы можно строить переносимым образом посредством либо стандартной библиотечной поддержки **tkinter**, либо сторонних инструментальных комплектов, таких как **wxPython** и **PyQt**. Комплект **tkinter** поставляется вместе с Python, позволяет быстро строить простые графические пользовательские интерфейсы, и он идеален для изучения методик программирования интерфейсов подобного рода; wxPython и PyQt сложнее в использовании, но часто в итоге дают интерфейсы более высокого качества.

# Применение

***Веб-сайты***

Хотя графические пользовательские интерфейсы являются удобными и быстрыми, веб-сеть трудно превзойти с точки зрения доступности. Для просмотра и обновления записей мы могли бы также реализовать веб-сайт взамен или в дополнение к графическому пользовательскому интерфейсу и интерактивной подсказке. Веб-сайты можно создавать с помощью либо базовых инструментов для написания сценариев **CGI**, имеющихся в составе Python, либо полномасштабных сторонних веб-фреймворков вроде **Django**, **TurboGears**, **Pylons**, **web2Py**, **Zope,** или **Google App Engine**. В веб-сети ваши данные по-прежнему могут находиться в хранилище shelve, в файле pickle или в другой среде, основанной на Python. Сценарии обработки данных запускаются автоматически на сервере в ответ на запросы из веб-браузеров и других клиентов, и они производят **HTML** разметку для взаимодействия с пользователем, либо напрямую, либо через **API** интерфейсы фреймворка. Системы обогащенных Интернет-приложений (**Rich Internet application** — RIA), такие как **Silverlight** и **pyjamas**, также пытаются объединить взаимодействие, подобное обеспечиваемому графическими пользовательскими интерфейсами, с развертыванием в веб-сети.

# Применение

***Веб-службы***

Хотя веб-клиенты часто способны проводить разбор информации в ответах от веб-сайтов (методика, красочно называемая “анализом экранных данных”), мы могли бы двинуться дальше и обеспечить более прямой способ извлечения записей через веб-сеть посредством интерфейса веб-служб, такого как SOAP или вызовы XML-RPC - API-интерфейсов, поддерживаемых либо самим Python, либо сторонними инструментами с открытым кодом, которые в целом отображают данные в формат XML и обратно с целью передачи. Для сценариев на Python подобного рода API-интерфейсы возвращают данные более непосредственно, чем текст, внедренный в HTML-разметку страницы ответа.

# Применение

***Базы данных***

Если наша база данных становится объемной или важной, тогда со временем мы могли бы перейти от модуля shelve к более полнофункциональному механизму хранения. Им может быть система управления объектно-ориентированными базами данных с открытым кодом **ZODB** или более традиционная система управления реляционными базами данных на основе **SQL**, такая как **MySQL**, **Oracle** или **PostgreSQL**. Сам Python поставляется со встроенным модулем внутрипроцессной системы управления базами данных **SQLite**, но в веб-сети свободно доступны другие варианты с открытым кодом. Например, система **ZODB** похожа на модуль **shelv**e из Python, но лишена множества его ограничений, лучше поддерживает крупные базы данных, параллельные обновления, обработку транзакций и автоматическую сквозную запись при изменениях в памяти (хранилища shelve могут кешировать объекты и сбрасывать их на диск во время закрытия посредством параметра **writeback**, но с этим связаны ограничения). Системы на основе **SQL**, подобные **MySQL**, предлагают инструменты производственного уровня для хранилища в виде базы данных и могут напрямую применяться в сценарии на Python. **Mongo**DB обеспечивает альтернативный подход, предусматривающий хранение документов **JSON**, которые близко напоминают словари и списки Python, но в отличие от данных **pickle** нейтральны к языку.

# Применение

***Средства объектнореляционного отображения***

В случае перевода хранилища на систему правления реляционными базами данных мы не должны приносить в жертву инструменты ООП в Python. Средства объектно-реляционного отображения (**object-relational mapper - ORM**) вроде **SQLObject** и **SQLAlchemy** способны автоматически отображать реляционные таблицы и строки на классы и экземпляры Python и обратно, так что мы можем обрабатывать сохраненные данные с использованием нормального синтаксиса классов Python. Такой подход предлагает альтернативу системам управления объектно-ориентированными базами данных наподобие **shelve** и **ZODB** и задействует мощь реляционных баз данных и модели классов Python

# Основы ООП

## Лекция 5. Детали реализации классов

# Оператор class

Хотя оператор **class** в Python на первый взгляд может выглядеть похожим на инструменты в других языках ООП, при ближайшем рассмотрении выясняется, что он совершенно отличается о того, к чему привыкли программисты. Например, как и в C++, оператор **class** представляет собой главный инструмент ООП в Python, но в отличие от C++ оператор **class** языка Python **class** не является объявлением. Подобно **def** оператор **class** создает объекты и неявно присваивает их - при выполнении он генерирует объект класса и сохраняет ссылку на него в имени, используемом в заголовке. Также подобно **def** оператор **class** относится к настоящему исполняемому коду - ваш класс не существует до тех пор, пока Python не достигнет и не выполнит оператор **class**, который его определяет. Обычно это происходит во время импортирования модуля, содержащего оператор **class**, но не ранее.

Оператор **class** является составным, с телом, состоящим из операторов, которые обычно размещаются с отступом под заголовком. В заголовке внутри круглых скобок после имени класса приводится список суперклассов, разделенных запятыми.

```
class имя(суперкласс,...): # Присваивание имени
    атрибут = значение # Разделяемые данные класса
    def метод(self, ...): # Методы
        self.атрибут = значение # Данные экземпляра
```

Любые присваивания внутри оператора class генерируют атрибуты класса, а особым образом именованные методы перегружают операции; скажем, функция по имени **__init__** вызывается на стадии конструирования объекта экземпляра, если она определена.

# Оператор class

Классы главным образом представляют собой всего лишь пространства имен - т.е. инструменты для определения имен (атрибутов), которые экспортируют данные и логику клиентам. Оператор **class** фактически определяет пространство имен. Как и в файле модуля, вложенные в тело **class** операторы создают атрибуты класса. Когда Python выполняет оператор **class** (не вызов класса), он запускает все операторы в теле от начала до конца. Присваивания, происходящие во время такого процесса, создают имена в локальной области видимости класса, которые становятся атрибутами в ассоциированном объекте класса. По этой причине классы напоминают и модули, и функции.

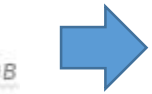
- подобно функциям операторы **class** являются локальными областями видимости, где находятся имена, созданные вложенными присваиваниями;
- подобно именам в модуле имена, присвоенные в операторе class, становятся атрибутами в объекте класса.

Главное отличие классов связано с тем, что их пространства имен формируют также основу наследования в Python; указанные атрибуты, которые не обнаруживаются в объекте класса или экземпляра, извлекаются из других классов.

Поскольку **class** представляет собой составной оператор, в его тело могут вкладываться операторы любого вида, т.е. **print, if, def** и т.д. Все операторы внутри class выполняются при выполнении самого оператора **class** (не тогда, когда позже происходит вызов класса для создания экземпляра). Обычно операторы присваивания внутри оператора class создают атрибуты данных, а вложенные операторы def - атрибуты методов. Однако в общем случае любой вид присваивания значения имени на верхнем уровне оператора **class** создает атрибут с таким именем в результирующем объекте класса.

# Оператор class

```
class SharedData:
    spam =42 # Генерирует атрибут данных

x = SharedData() # Создание двух экземпляров
y = SharedData()
print(x.spam, y.spam)
```



```
42 42
```

Поскольку имени spam присваивается значение на верхнем уровне оператора class, оно присоединяется к классу, а потому будет разделяться всеми экземплярами. Мы можем изменять его через имя класса и ссылаться посредством либо экземпляров, либо самого класса.

```
SharedData.spam = 99
print(x.spam, y.spam)
```

```
99 99
```

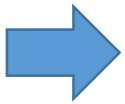
Присваивания значений атрибутам экземпляра создают либо изменяют имена в ***экземпляре, а не в разделяемом классе***. В более общем смысле поиск в иерархии наследования инициируется только при ссылке на атрибуты, не в случае присваивания: присваивание значения атрибуту объекта всегда изменяет данный объект, но никакой другой. Например, y.spam ищется в классе через наследование, но присваивание x. spam присоединяет имя к самому объекту x.

Если только класс с помощью метода перегрузки операции **__setattr__** не переопределяет операцию присваивания атрибутам, чтобы она делала что-то уникальное, или не использует расширенные инструменты для работы с атрибутами, такие как свойства и дескрипторы.

# Оператор class

Класс **MixedNames** содержит два оператора **def**, которые привязывают атрибуты класса к функциям методов. Он также содержит оператор присваивания =; поскольку этот оператор присваивает значение имени data внутри **class**, оно существует в локальной области видимости класса и становится атрибутом объекта класса. Как и все атрибуты класса, data наследуется и разделяется всеми экземплярами класса, которые не имеют собственных атрибутов data. Когда мы создаем экземпляры класса **MixedNames**, имя data присоединяется к ним путем присваивания self.data в методе конструктора.

```
class MixedNames:   # Определить класс
    data = 'spam'   # Присвоить значение атрибуту класса
    def __init__ (self, value): # Присвоить значение имени метода
        self.data = value # Присвоить значение атрибуту экземпляра
    def display(self):
        print(self.data, MixedNames.data) # Атрибут экземпляра, атрибут класса
x = MixedNames(1)
y = MixedNames(2)
x.display()
y.display()
```



```
1 spam
2 spam
```

# Оператор class

Совокупный результат в том, что атрибут data присутствует в двух местах: в объектах экземпляров (создан присваиванием **self. data** в**__ init__** ) и в классе, от которого они наследуют имена (создан присваиванием data в class). Метод **display** класса выводит обе версии, сначала версию, уточненную посредством экземпляра **self**, и затем версию класса. За счет применения таких методик для сохранения атрибутов в разных объектах мы определяем их область видимости. Когда имена присоединены к классам, они разделяются; в экземплярах имена фиксируют данные для каждого экземпляра, не разделяя поведение или данные. Несмотря на то что поиск в иерархии наследования находит для нас имена, мы всегда можем извлечь атрибут откуда угодно из дерева, получая доступ к желаемому объекту напрямую.

В предыдущем примере указание **x.data** или **self.data** возвратит имя экземпляра, которое в нормальной ситуации скрывает такое же имя в классе; тем не менее, **MixedNames.data** явно захватывает версию имени класса.

# Методы

Методы являются просто объектами функций, которые создаются операторами **def**, вложенными в тело оператора **class**. С абстрактной точки зрения методы предоставляют поведение для наследования объектами экземпляров. С точки зрения программирования методы работают в точности таким же образом, как и простые функции, но с одним критически важным исключением: первый аргумент метода всегда получает объект экземпляра, который представляет собой подразумеваемый объект вызова метода. Другими словами, Python автоматически отображает вызовы методов экземпляра на функции методов класса, как показано далее.

Вызовы методов, выполненные через экземпляр:
***экземпляр. метод (аргументы. . .)***
автоматически транслируются в вызовы функций методов класса в следующей форме:
***класс.метод(экземпляр, аргументы...)***
где Python определяет класс, находя имя метода с использованием процедуры поиска в иерархии наследования. На самом деле в Python допустимы обе формы вызова.

Помимо нормального наследования имен атрибутов методов специальный первый аргумент является единственной реальной магией, стоящей за вызовами методов.
В методе класса первый аргумент по соглашению называется **self** (формально важна только его позиция, но не имя). Аргумент **self** снабжает метод привязкой к экземпляру, на котором производился вызов - из-за того, что классы генерируют много объектов экземпляров, им необходимо применять этот аргумент для управления данными, которые варьируются от экземпляра к экземпляру.

# Методы

Методы являются просто объектами функций, которые создаются операторами **def**, вложенными в тело оператора **class**. С абстрактной точки зрения методы предоставляют поведение для наследования объектами экземпляров. С точки зрения программирования методы работают в точности таким же образом, как и простые функции, но с одним критически важным исключением: первый аргумент метода всегда получает объект экземпляра, который представляет собой подразумеваемый объект вызова метода. Другими словами, Python автоматически отображает вызовы методов экземпляра на функции методов класса, как показано далее.

Вызовы методов, выполненные через экземпляр:
***экземпляр. метод (аргументы. . .)***
автоматически транслируются в вызовы функций методов класса в следующей форме:
***класс.метод(экземпляр, аргументы...)***
где Python определяет класс, находя имя метода с использованием процедуры поиска в иерархии наследования. На самом деле в Python допустимы обе формы вызова.

Помимо нормального наследования имен атрибутов методов специальный первый аргумент является единственной реальной магией, стоящей за вызовами методов.
В методе класса первый аргумент по соглашению называется **self** (формально важна только его позиция, но не имя). Аргумент **self** снабжает метод привязкой к экземпляру, на котором производился вызов - из-за того, что классы генерируют много объектов экземпляров, им необходимо применять этот аргумент для управления данными, которые варьируются от экземпляра к экземпляру.

# Методы

Методы являются просто объектами функций, которые создаются операторами **def**, вложенными в тело оператора **class**. С абстрактной точки зрения методы предоставляют поведение для наследования объектами экземпляров. С точки зрения программирования методы работают в точности таким же образом, как и простые функции, но с одним критически важным исключением: первый аргумент метода всегда получает объект экземпляра, который представляет собой подразумеваемый объект вызова метода. Другими словами, Python автоматически отображает вызовы методов экземпляра на функции методов класса, как показано далее.

```
class NextClass:                # Определить класс
    def printer(self, text):    # Определить метод
        self.message = text     # Изменить экземпляр
        print(self.message)     # Получить доступ к экземпляру

x = NextClass()                 # Создать экземпляр
x.printer('instance call')      # Вызвать его метод
print(x.message)                # Экземпляр изменился
```

```
instance call
instance call
```

Когда мы вызываем метод, подобным образом уточняя экземпляр, метод **printer** сначала ищется в иерархии наследования и затем его аргументу **self** автоматически присваивается объект экземпляра (**x**); аргумент **text** получает строку, переданную при вызове (***'instance call'***). Обратите внимание, что поскольку Python автоматически передает первый аргумент **self**, то нам фактически понадобится передать один аргумент. Внутри **printer** имя **self** применяется для доступа или установки данных экземпляра, т.к. оно ссылается на экземпляр, который в текущий момент обрабатывается.

# Методы

Методы могут вызываться одним из двух способов - через экземпляр или через сам класс. Например, мы также можем вызывать **printer**, указывая имя класса, при условии явной передачи экземпляра в аргументе **self**

```
class NextClass:                        # Определить класс
    def printer(self, text):            # Определить метод
        self.message = text             # Изменить экземпляр
        print(self.message)             # Получить доступ к экземпляру

x = NextClass()                         # Создать экземпляр
NextClass.printer(x, 'class call')  # Прямой вызов через класс
```

```
class call
```

# Типы методоы

Одни методы являются частью самого класса, другие - частью объектов, созданных из этого класса, а какие-то не являются ни тем ни другим :

- Если перед методом нет декоратора, то это **метод объекта** и его первым аргументом должен быть **self**, который ссылается на объект
- Если у метода есть декоратор **@classmethod**, то это **метод класса** и его первым аргументом должен быть **cls** (имя может быть любым, кроме зарезервированного слова class), который ссылается на класс
- Если у метода есть декоратор **@staticmethod**, то это **статический метод** и его первым аргументом будет не объект и не класс

Когда вы видите начальный аргумент self в методах внутри определения класса, вы имеете дело с **методом объекта**: такие методы вы обычно пишете, когда создаете собственный класс.

# Типы методоы

**Метод класса** **влияет на весь класс целиком.** Любое изменение, которое происходит с классом, влияет на все его объекты. Внутри определения класса декоратор **@classmethod** показывает, что следующая функция является методом класса Первым параметром метода также является сам класс. По традиции этот параметр называется **cls**, поскольку слово **class** зарезервировано и не может быть использовано в данной ситуации.

```
class A():
    count = 0
    def __init__(self):
        A.count += 1
    def exclaim(self):
        print("I'm an A!")
    @classmethod
    def kids(cls):
        print("A has", cls.count, "little objects.")

easy_a = A()
breezy_a = A()
wheezy_a = A()
A.kids()
```

```
A has 3 little objects.
```

Обратите внимание на то, что мы вызвали метод **A.count** (атрибут класса) вместо **self.count** (который является атрибутом объекта). В методе **kids()** мы использовали вызов **cls.count**, но с тем же успехом могли бы применить вызов A.count

# Типы методы

Третий тип методов не влияет ни на классы, ни на объекты: он находится внутри класса только для удобства, чтобы не располагаться где-то отдельно. Это статический метод, которому предшествует декоратор **@staticmethod**, не имеющий в качестве начального параметра ни self, ни класс class.

```
class CoyoteWeapon():
    @staticmethod
    def commercial():
        print('This CoyoteWeapon has been brought to you by Acme')

CoyoteWeapon.commercial()
```

```
This CoyoteWeapon has been brought to you by Acme
```

***Обратите внимание на то, что нам не нужно создавать объект класса CoyoteWeapon, чтобы получить доступ к этому методу.***

# Магические методы

Имена этих методов начинаются и заканчиваются двойными подчеркиваниями (__). Почему? Как правило, такие имена не выбирают в качестве имен для переменных. Вы уже видели один такой метод: **__init__()** инициализирует только что созданный объект с помощью описания его класса и любых аргументов, которые передаются в этот метод.

| | |
|---|---|
| __eq__(*self*, *other*) | *self* == *other* |
| __ne__(*self*, *other*) | *self* != *other* |
| __lt__(*self*, *other*) | *self* < *other* |
| __gt__(*self*, *other*) | *self* > *other* |
| __le__(*self*, *other*) | *self* <= *other* |
| __ge__(*self*, *other*) | *self* >= *other* |

| | |
|---|---|
| __add__(*self*, *other*) | *self* + *other* |
| __sub__(*self*, *other*) | *self* – *other* |
| __mul__(*self*, *other*) | *self* * *other* |
| __floordiv__(*self*, *other*) | *self* // *other* |
| __truediv__(*self*, *other*) | *self* / *other* |
| __mod__(*self*, *other*) | *self* % *other* |
| __pow__(*self*, *other*) | *self* ** *other* |

# Магические методы

Чтобы узнать о других специальных методах, обратитесь к документации Python (http://bit.ly/pydocs-smn).

| | |
|---|---|
| __str__(self) | str(self) |
| __repr__(self) | repr(self) |
| __len__(self) | len(self) |

# Вызов конструкторов суперклассов

Методы обычно вызываются через экземпляры. Однако вызовы методов через класс обнаруживаются в разнообразных специальных ролях. Один распространенный сценарий касается метода конструктора. Подобно всем атрибутам метод **__ init__** ищется в иерархии наследования. Другими словами, на стадии конструирования Python ищет и вызывает только один метод **__init__** . Если конструкторам подклассов нужна гарантия того, что также выполняется логика стадии конструирования суперкласса, тогда в общем случае они обязаны явно вызывать метод **__init__** суперкласса через класс. Естественно, вы должны вызывать конструктор суперкласса подобным образом, только если действительно хотите, чтобы он выполнился - без такого вызова подкласс заместит его полностью.

```
class Super:
    def __init__(self, x):
        pass

class Sub(Super):
    def __init__ (self, x, y) :
        Super.__init__ (self, x)  # Выполнить метод__ init__ суперкласса
        pass                      # Выполнить специальные действия инициализации
```

# Наследование

Наследование в Python происходит, когда объект уточняется, и оно инициирует поиск в дереве определения атрибутов - в одном и большем числе пространств имен. Каждый раз, когда вы используете выражение вида ***объект.атрибут***, где объект представляет собой объект экземпляра или класса, Python осуществляет поиск в дереве пространств имен снизу вверх, начиная с объекта, с целью нахождения первого атрибута, который удастся обнаружить. Сюда входят ссылки на атрибуты **self** в ваших методах. Поскольку определения, расположенные ниже в дереве, переопределяют те, что находятся выше, наследование формирует основу специализации.

В целом:

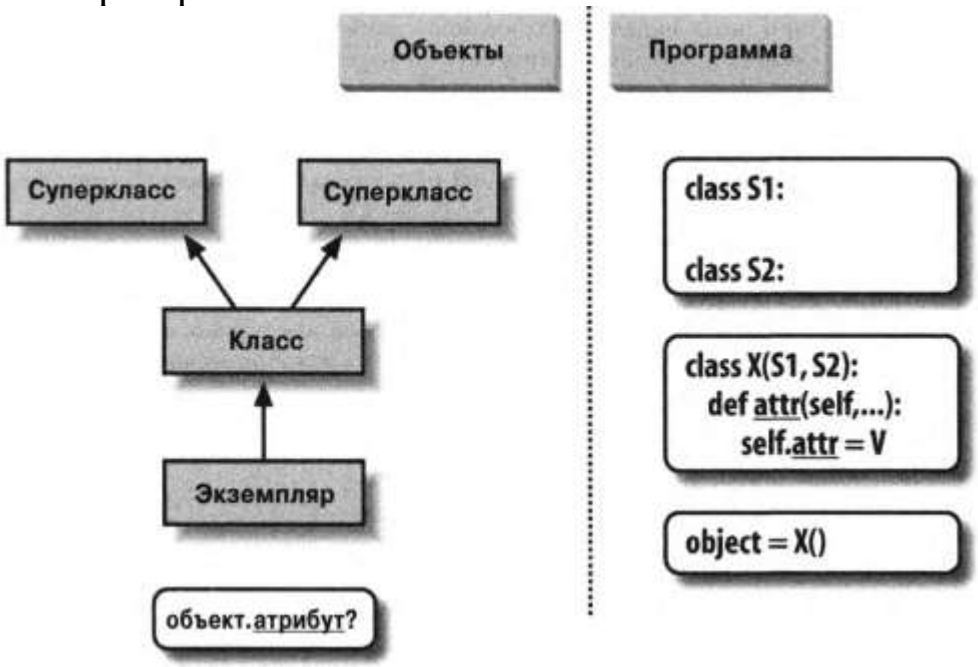
• атрибуты экземпляров генерируются операторами присваивания значений атрибутам **self** в методах;

• атрибуты классов создаются операторами (присваивания) в операторах **class**;

• ссылки на суперклассы образуются на основе списков классов внутри круглых скобок в заголовке оператора **class**.

# Наследование

Код программы создает дерево объектов в памяти, где будет происходить поиск со стороны наследования атрибутов. Обращение к классу создает новый экземпляр, который запоминает свой класс, выполнение оператора **class** создает новый класс, а суперклассы перечисляются внутри круглых скобок в заголовке оператора **class**. Каждая ссылка на атрибут запускает новую процедуру восходящего поиска в дереве - даже ссылки на атрибуты **self** внутри методов класса. Конечным результатом оказывается дерево пространств имен атрибутов, ведущее от экземпляра к классу, из которого он был создан, и ко всем суперклассам, перечисленным в заголовке оператора **class**.




Всякий раз, когда вы применяете уточнение для извлечения имени атрибута из объекта экземпляра, в этом дереве инициируется восходящий поиск в направлении от экземпляров к суперклассам.

# Специализации унаследованных методов

Из-за того, что процесс наследования ищет сначала имена в подклассах до проверки суперклассов, подклассы способны замещать стандартное поведение за счет переопределения атрибутов своих суперклассов. Идея переопределения унаследованных имен положена в основу многообразия методик специализации. Например, подклассы могли бы полностью замещать унаследованные атрибуты, предоставлять атрибуты, которые ожидает обнаружить суперкласс, и расширять методы суперклассов, вызывая их в переопределяемых методах.

```
class Super:
    def method(self):
        print('in Super.method')
class Sub(Super):
    def method(self):                    # Переопределить метод
        print ('starting Sub.method') # Добавить здесь нужные действия
        Super.method (self) # Запустить стандартное действие
        print('ending Sub.method')
x = Super() # Создать экземпляр Super
x.method() # Выполняется Super.method
y = Sub()  # Создать экземпляр Sub
y.method() # Выполняется Sub.method,вызывается Super.method
```

```
in Super.method
starting Sub.method
in Super.method
ending Sub.method
```

Вся суть здесь кроется в прямом вызове метода суперкласса. Класс **Sub** замещает функцию **method** суперкласса **Super** собственной специализированной версией, но внутри самого замещения **Sub** обращается к версии, экспортируемой **Super**, чтобы обеспечить выполнение стандартного поведения.

# Методики связывания классов

Ниже определено множество классов, которые иллюстрируют разнообразные методики связывания:

**Super.** Определяет функцию **method** и метод **delegate**, который ожидает наличие в подклассе метода **action**.

**Inheritor.** Не предоставляет никаких новых имен, поэтому получает все, что определено в **Super**.

**Replacer**. Переопределяет **method** из **Super** посредством собственной версии.

**Extender**. Настраивает **method** из **Super**, переопределяя и обеспечивая выполнение стандартного поведения.

**Provider**. Реализует метод **action**, ожидаемый методом **delegate** из **Super**

# Методики связывания классов

```
class Super:
    def method(self):                # Стандартное поведение
        print('in Super.method')
    def delegate(self):              # Ожидается определение метода
        self.action()

class Inheritor(Super):              # Буквальное наследование метода
    pass
class Replacer(Super) :              # Полное замещение метода
    def method(self):
        print('in Replacer.method')
class Extender(Super):               # Расширение поведения метода
    def method(self):
        print('starting Extender.method') # начало Extender.method
        Super.method(self)
        print('ending Extender.method')   # конец Extender.method
class Provider(Super):                    # Заполнение обязательного метода
    def action(self):
        print('in Provider.action')
```

# Методики связывания классов

```
if __name__ == '__main__' :
    for klass in (Inheritor, Provider, Extender):
        print('\n' + klass.__name__ + '...')
        klass().method()
    print('\nProvider...')
    x = Provider()
    x.delegate()
```

```
Inheritor...
in Super.method

Provider...
in Super.method

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider...
in Provider.action
```

Код самотестирования создает внутри цикла for экземпляры трех разных классов. Поскольку классы являются объектами, вы можете сохранять их в кортеже и создавать экземпляры обобщенным образом без дополнительного синтаксиса. Подобно модулям классы также имеют специальный атрибут__ name__ ; он заранее устанавливается в строку, содержащую имя из заголовка класса.

# Абстрактные суперклассы

Когда мы вызываем метод **delegate** через экземпляр Provider, происходят два независимых поиска со стороны процедуры наследования.

1. В начальном вызове **x.delegate** интерпретатор Python находит метод **delegate** в **Super**, выполняя поиск в экземпляре **Provider** и выше. Экземпляр **x** передается в аргументе **self** метода как обычно.
2. Внутри метода **Super.delegate** метод **self.action** инициирует новый независимый поиск в **self** и выше. Из-за того, что **self** ссылается на экземпляр Provider, метод **action** обнаруживается в подклассе **Provider**.

Такая кодовая структура вида “заполнение пропусков” типична для объектно-ориентированных фреймворков. В более реалистичном контексте заполняемый подобным образом метод мог бы обрабатывать какое-то событие в графическом пользовательском интерфейсе, снабжать данными для визуализации как часть веб-страницы, анализировать текст дескриптора в XML-файле и т.д. - ваш подкласс предоставляет специфические действия, но оставшуюся часть всей работы выполняет фреймворк. По крайней мере, в свете метода **delegate** суперкласс в этом примере является тем, что иногда называют абстрактным суперклассом - классом, который ожидает от своих подклассов предоставления частей своего поведения. Если ожидаемый метод в подклассе не определен, тогда после неудавшегося поиска при наследовании Python генерирует исключение неопределенного имени.

# Абстрактные суперклассы

Временами создатели классов делают такие требования к подклассам более очевидными с помощью операторов ***assert*** или путем генерации встроенного исключения ***NotImplementedError*** посредством операторов ***raise***. В следующей части книги мы подробнее исследуем операторы, способные генерировать исключения; для краткого предварительного ознакомления ниже показана схема **assert** в действии.

```
class Super:
    def delegate(self):
        self.action()
    def action(self):
        assert False, 'action must be defined!' # Если вызвана эта версия
x = Super()
x.delegate()
```

```
AssertionError: action must be defined!
```

# Абстрактные суперклассы

Абстрактные суперклассы, которые требуют заполнения методов подклассами, также могут быть реализованы посредством специального синтаксиса классов. Способ написания кода слегка варьируется в зависимости от версии. В Python 3.X мы применяем ключевой аргумент в заголовке оператора **class** вместе со специальным декораторным синтаксисом **@** и используем встроенный модуль abc, с помощью которого мы можем предотвратить дочерние классы от того экземпляра, когда они не могут переопределить абстрактные методы класса родителей и предков. Абстрактные базовые классы (ABC) определяют, какие производные классы реализуют конкретные методы из базового класса. Вспомогательный класс, метаклассом которого является ABCMeta. С помощью этого класса можно создать абстрактный базовый класс, просто производя его от ABC

```
from abc import ABCMeta, abstractmethod
class Super(metaclass=ABCMeta):
    @abstractmethod
    def method(self, . . .) :
        pass
```

При такой реализации создавать экземпляры класса с абстрактным методом нельзя (т.е. мы не можем создать экземпляр, обратившись к нему), пока в подклассах не будут определены все его абстрактные методы. Хотя такой подход требует написания большего объема кода и добавочных знаний, его потенциальное преимущество в том, что сообщения об ошибках, связанных с недостающими методами, выдаются при попытке создания экземпляра класса, а не позже, когда мы пытаемся вызвать отсутствующий метод. Это средство можно применять и для определения ожидаемого интерфейса, автоматически проверяемого в клиентских классах.

# Пространства имен

В справочных целях ниже приведена краткая сводка всех правил, применяемых для распознавания имен. Первое, что следует за помнить - уточненные и неуточненные имена трактуются по-разному, а некоторые области видимости предназначены для инициализации пространств имен объектов:

• неуточненные имена (например, X) имеют дело с областями видимости;

• уточненные имена (скажем, object.X) используют пространства имен объектов;

• определенные области видимости инициализируют пространства имен объектов (для модулей и классов).

Иногда эти концепции взаимодействуют - например, в object.X имя object ищется по областям видимости, после чего в результирующих объектах производится поиск имени X.

**Простые имена**

***Присваивание (X = значение)***

По умолчанию делает имя локальным: создает либо изменяет имя X в текущей локальной области видимости, если только оно не объявлено как global (или nonlocal в Python 3.X).

***Ссылка (X)***

Ищет имя X в текущей локальной области видимости, затем в любых объемлющих функциях, далее в текущей глобальной области видимости, затем во встроенной области видимости согласно правилу LEGB. Поиск во включающих классах не выполняется: взамен имена классов извлекаются как атрибуты объектов.

# Пространства имен

**В случае объектов классов и экземпляров правила ссылки дополняются с целью включения процедуры поиска при наследовании.**

***Присваивание (object.X = значение)***
Создает или модифицирует имя атрибута X в пространстве имен уточняемого объекта ***object*** и больше нигде. Подъем по дереву наследования происходит только при ссылке на атрибут, но не в случае присваивания значения атрибуту.

***Ссылка (object.X)***
Для объектов, основанных на классах, ищет имя атрибута X в object и затем во всех доступных классах выше, применяя процедуру поиска при наследовании. Для объектов, не основанных на классах, таких как модули, извлекает X из object напрямую.

Ссылочное наследование может обладать более широкими возможностями, чем здесь предполагается, когда развернуты метаклассы, а классы, которые задействуют инструменты управления атрибутами, такие как свойства, дескрипторы и **__ setattr__** , могут перехватывать и направлять присваивания значений атрибутам произвольным образом.

# Пространства имен

```
X = 11                  # Глобальное имя/атрибут модуля (X)
def f () :
    print(X)            # Доступ к глобальному имени X (11)

def g() :               # Локальная переменная в функции (X, скрывает X в модуле)
    X = 22
    print(X)

class C:
    X = 33              # Атрибут класса (C.X)
    def m(self) :
        X = 44          # Локальная переменная в методе (X)
        self.X = 55     # Атрибут экземпляра (экземпляр.X)

if __name__ == '__main__':
    print(X)                #11: имя из модуля (оно же manynames. X за пределами файла)
    f()                     #11: глобальное имя
    g()                     #22: локальное имя
    print(X)                #11: имя из модуля не изменилось

    obj = C()               # Создать экземпляр
    print(obj.X)            # 33: имя из класса, унаследованное экземпляром

    obj.m()                 # Присоединения имени атрибута X к экземпляру
    print(obj.X)            # 55: имя из экземпляра
    print(C.X)              # 33: имя из класса (оно же obj.X, если X в экземпляре отсутствует)

#print(C.m.X)           # НЕУДАЧА: видимо только в методе
#print(g.X)             # НЕУДАЧА: видимо только в функции
```

```
11
11
22
11
33
55
33
```

# Пространства имен

Некоторые имена, определенные в файле, видимы также за пределами файла другим модулям, но не забывайте, что мы обязаны всегда импортировать файл, прежде чем сможем иметь доступ к его именам - в конце концов, как раз изоляция имен является сущностью модулей:

```
import manynames
X = 66
print(X)     # 66: здесь глобальное имя
print(manynames.X)  # 11 после импортирования глобального имени
manynames.f()       # 11 имя X из модуля tynames
manynames.g()       # 22 локальное имя из функции в другом файле
print(manynames.C.X) # 33 атрибут класса в другом модуле
I = manynames.C()
print(I.X)            # 33 все еще имя из класса
I.m()
print(I.X)            #55 а теперь из экземпляра
```

Обратите внимание, что вызов **manynames.f ()** выводит X из manynames, не имя X, присвоенное в данном файле - области видимости всегда определяются местоположением присваиваний в исходном коде (т.е. лексически) и никогда не зависят от того, что и куда импортируется. Кроме того, имейте в виду, что собственное имя X экземп ляра не создается до тех пор, пока мы не вызовем **I.m()** - подобно всем переменным атрибуты появляются, когда им присваиваются значения, не ранее. Обычно мы созда ем атрибуты экземпляров путем присваивания им значений в методах конструкторов **__ init__** классов, но это не единственный вариант.

# Вложенные классы

Несмотря на то что классы обычно определяются на верхнем уровне модуля, иногда они являются вложенными в генерирующие их функции. Правило LEGB применяется к верхнему уровню самого класса и к верхнему уровню вложенных в него функций методов. Оба формируют уровень L этого правила - они представляют собой нормальные локальные области видимости с доступом к их именам, именам в любых объемлющих функциях, глобальным именам во включающем модуле и встроенной области видимости. Как и модули, после выполнения оператора class локальная область видимости класса превращается в пространство имен атрибутов.

Хотя классы имеют доступ к областям видимости объемлющих функций, они не действуют в качестве объемлющих областей видимости для кода, вложенного внутрь класса: Python ищет имена, на которые произведена ссылка, в объемлющих функциях, но никогда не выполняет их поиск в объемлющих классах. То есть класс является локальной областью видимости и имеет доступ к объемлющим локальным областям видимости, но он не служит объемлющей локальной областью видимости для дальнейшего вложенного кода. Из-за того, что процесс поиска имен, используемых в функциях методов, пропускает включающий класс, атрибуты класса должны извлекаться как атрибуты объекта с применением наследования.

# Вложенные классы

Переопределение X создает локальные имена, которые скрывают имена из объемлющих областей видимости, в точности как для простых вложенных функций; уровень включающего класса вовсе не изменяет это правило и на самом деле не имеет к нему отношения

```
X = 1
def nester():
    X = 2
    print(X)                    # Локальное имя: 2
    class C:
        print(X)                # В объемлющем def (nester) : 2
        def method1(self):
            print(X)            # В объемлющем def (nester) : 2
        def method2(self):
            X = 3               # Скрывает имя из объемлющего def (nester)
            print(X)            # Локальное имя 3
    I = C()
    I.method1()
    I.method2()
print(X)                    # Глобальное имя: 1
nester()                    # 2,2,2, 3
```

# Вложенные классы

Если мы заново присваиваем значение тому же самому имени несколько раз по пути: присваивания в локальных областях видимости функций и классов скрывают глобальные имена или совпадающие локальные имена из объемлющих функций независимо от вложенности.

```
X = 1
def nester():
    X = 2
    print(X)                # Локальное имя: 2
    class C:
        X = 3               # Локальное имя из класса скрывает имя из nester
        print(X)            # Локальное имя 3
        def method1(self):
            print(X)        # В объемлющем def (nester) : 2 (не 3 из класса !!!!)
            print(self.X)   # Унаследованное локальное имя класса 3
        def method2(self):
            X = 4           # Скрывает имя из объемлющего def (nester), а не из класса
            print(X)        # Локальное имя 4
            self.X = 5      # Скрывает имя из класса
            print(self.X)   # Находится в экземпляре 5
    I = C()
    I.method1()
    I.method2()
print(X)                  # Глобальное имя: 1
nester()                  # 2,3,2,3,4,5
```

# Вложенные классы

Самое главное, правила поиска для простых имен вроде X никогда не ищут во включающих операторах class - только в операторах def, модулях и встроенной об ласти видимости (правило называется LEGB, а не CLEGB!). Скажем, в method1 имя X находится в def за пределами включающего класса, который имеет то же самое имя в своей локальной области видимости. Чтобы получить имена, присвоенные в классе (например, методы), мы обязаны извлекать их как атрибуты объекта класса или экземпляра через self .X в данном случае.

# Словари пространства имен

Пространства имен модулей имеют конкретную реализацию в виде словарей, доступную через встроенный атрибут **__ dict__.**

Объекты классов и экземпляров в Python по большей части являются всего лишь словарями со связями между ними. Доступ к таким словарям, равно как к их связям, обеспечивается для участия в расширенных ролях (скажем, для создания инструментов).

Первым делом определим суперкласс и подкласс с методами, которые будут сохранять данные в своих экземплярах.

```
class Super:
    def hello(self):
        self.data1 = 'spam'
class Sub(Super):
    def hola(self):
        self.data2 = 'eggs'
```

# Словари пространства имен

Когда мы создаем экземпляр подкласса, он начинает свое существование с пустого словаря пространства имен, но имеет связи с классом для обеспечения работы поиска при наследовании. В действительности дерево наследования явно доступно в специ альных атрибутах, которые можно исследовать. Экземпляры располагают атрибутом **__ class__** , связывающим их с классом, а классы имеют атрибут **__ bases__** , который представляет собой кортеж, содержащий связи с находящимися выше в дереве супер классами.

```
class Super:
    def hello(self):
        self.data1 = 'spam'
class Sub(Super):
    def hola(self):
        self.data2 = 'eggs'

X = Sub()
print(X.__dict__)       # Словарь пространства имен экземпляра
print(X.__class__)      # Класс экземпляра
print(Sub.__bases__)    # Определяем суперкласс класса
print(Super.__bases__)  # Пустота
```

```
{}
<class '__main__.Sub'>
(<class '__main__.Super'>,)
(<class 'object'>,)
```

# Словари пространства имен

Когда классы выполняют присваивание атрибутам **self**, они заполняют объекты экземпляров - т.е. атрибуты оказываются в словарях пространств имен экземпляров, а не классов. Пространство имен объекта экземпляра записывает данные, которые могут варьироваться от экземпляра к экземпляру, причем **self** является привязкой к этому пространству имен.

```
Y = Sub()
X.hello()
print(X.__dict__)
X.hola()
print(X.__dict__)
print(list(Sub.__dict__.keys()))
print(list(Super.__dict__.keys()))
print(Y.__dict__)
```

```
{'data1': 'spam'}
{'data1': 'spam', 'data2': 'eggs'}
['__module__', 'hola', '__doc__']
['__module__', 'hello', '__dict__', '__weakref__', '__doc__']
{}
```

Y, второй экземпляр, созданный в начале сеанса, в конце по-прежнему имеет пустой словарь пространства имен, хотя словарь экземпляра X заполнялся присваиваниями в методах. Опять-таки каждый экземпляр располагает независимым словарем пространства имен, пустым в самом начале и способным хранить атрибуты, которые полностью отличаются от атрибутов, записанных в словарях пространств имен других экземпляров того же самого класса.

# Словари пространства имен

Поскольку атрибуты в действительности представляют собой ключи словаря внутри Python, на самом деле существуют два способа извлечения и присваивания им значений - уточнение либо индексирование по ключу:

```
print(X.data1, X.__dict__['data1'])
X.data3 = 'toast'
print(X.__dict__)
X.__dict__['data3'] = 'ham'
print(X.data3)
```

```
spam spam
{'data1': 'spam', 'data2': 'eggs', 'data3': 'toast'}
ham
```

Однако такая эквивалентность применима только к атрибутам, фактически при соединенным к экземпляру. Так как уточнение при извлечении атрибутов тоже выполняет поиск в дереве наследования, оно может получать доступ к унаследованным атрибутам, что невозможно сделать через индексирование в словаре пространства имен. Скажем, к унаследованному атрибуту X.hello нельзя обратиться посредством X.__ dict__ [ 'hello' ].

# Связи между пространствами имен: инструмент подъема по дереву

Специальные атрибуты экземпляров и классов **__class__** и **__bases__** позволяют инспектировать иерархии наследования внутри написанного вами кода.
Реализуем подъем по деревьям наследования с применением связей между пространствами имен и отображением находящихся выше суперклассов с отступом согласно высоте.

```
def classtree(cls, indent):
    print('.' * indent + cls.__name__ )      # Вывести здесь имя класса
    for supercls in cls.__bases__:           # Вызвать рекурсивно для всех суперклассов
        classtree(supercls, indent+3)        # может посетить суперкласс более одного раза


def instancetree(inst):
    print('Tree of %s' % inst)               # Показать экземпляр
    classtree(inst.__class__, 3)             # Подняться к его классу

def selftest():
    class A: pass
    class B(A): pass
    class C(A): pass
    class D(B, C) : pass
    class E: pass
    class F(D,E) : pass
    instancetree(B())
    instancetree(F())
if __name__ == '__main__':
    selftest ()
```

# Связи между пространствами имен: инструмент подъема по дереву

```
def classtree(cls, indent):
    print('.' * indent + cls.__name__ )      # Вывести здесь имя класса
    for supercls in cls.__bases__:           # Вызвать рекурсивно для всех суперклассов
        classtree(supercls, indent+3)        # может посетить суперкласс более одного раза


def instancetree(inst):
    print('Tree of %s' % inst)               # Показать экземпляр
    classtree(inst.__class__, 3)             # Подняться к его классу

def selftest():
    class A: pass
    class B(A): pass
    class C(A): pass
    class D(B, C) : pass
    class E: pass
    class F(D,E) : pass
    instancetree(B())
    instancetree(F())
if __name__ == '__main__':
    selftest ()
```

Функция classtree в этом сценарии рекурсивна - она выводит имя класса с использованием **__ name__** и затем поднимается к его суперклассам, вызывая саму себя. Такая реализация позволяет функции обходить деревья классов произвольной формы; рекурсия поднимается доверху и останавливается на корневых суперклассах, которые имеют пустые атрибуты **__ bases__** . Когда применяется рекурсия, каждый активный уровень функции получает собственную копию локальной области видимости; здесь это означает, что аргументы **cls** и **indent** будут разными на каждом уровне **classtree**.

# Связи между пространствами имен: инструмент подъема по дереву

```
Tree of <__main__.selftest.<locals>.B object at 0x00C3F6A0>
...B
......A
.........object
Tree of <__main__.selftest.<locals>.F object at 0x00C3F6A0>
...F
......D
.........B
............A
..............object
.........C
............A
..............object
......E
.........object
```

# Связи между пространствами имен: инструмент подъема по дереву

```
Tree of <__main__.selftest.<locals>.B object at 0x00C3F6A0>
...B
......A
.........object
Tree of <__main__.selftest.<locals>.F object at 0x00C3F6A0>
...F
......D
.........B
............A
..............object
.........C
............A
..............object
......E
.........object
```

# Связи между пространствами имен: инструмент подъема по дереву

```
Tree of <__main__.selftest.<locals>.B object at 0x00C3F6A0>
...B
......A
.........object
Tree of <__main__.selftest.<locals>.F object at 0x00C3F6A0>
...F
......D
.........B
............A
..............object
.........C
............A
..............object
......E
.........object
```

# Документация

Строки документации представляют собой строковые литералы, которые отображаются в верхней части разнообразных структур и автоматически сохраняются Python в атрибутах **__ doc__** соответствующих объектов. Они работают для файлов модулей, операторов **def** определения функций, а также классов и методов. Все они могут быть блоками в утроенных кавычках или более простыми однострочными литералами. Основное преимущество строк документации заключается в том, что они не исчезают во время выполнения. Соответственно, если для объекта была предусмотрена строка документации, тогда вы можете дополнить объект атрибутом **__ doc__** и извлечь его документацию.

```
" I am: docstr.__doc__ "
def func(args):
    "I am: docstr. func.__doc__"
    pass

class spam:
    "I am: spam.__doc__ or docstr. spam.__doc__ or self.__doc__"
    def method(self):
        "I am: spam.method.__doc__ or self.method.__doc__"
        print(self.__doc__)
        print(self.method.__doc__ )
```

```
print(func.__doc__)
print(spam.__doc__)
print(spam.method.__doc__)
X = spam()
X.method()
```

```
I am: docstr. func.__doc__
I am: spam.__doc__ or docstr. spam.__doc__ or self.__doc__
I am: spam.method.__doc__ or self.method.__doc__
I am: spam.__doc__ or docstr. spam.__doc__ or self.__doc__
I am: spam.method.__doc__ or self.method.__doc__
```

# Документация

Строки документации представляют собой строковые литералы, которые отображаются в верхней части разнообразных структур и автоматически сохраняются Python в атрибутах **__ doc__** соответствующих объектов. Они работают для файлов модулей, операторов **def** определения функций, а также классов и методов. Все они могут быть блоками в утроенных кавычках или более простыми однострочными литералами. Основное преимущество строк документации заключается в том, что они не исчезают во время выполнения. Соответственно, если для объекта была предусмотрена строка документации, тогда вы можете дополнить объект атрибутом **__ doc__** и извлечь его документацию.

```
" I am: docstr.__doc__ "
def func(args):
    "I am: docstr. func.__doc__"
    pass

class spam:
    "I am: spam.__doc__ or docstr. spam.__doc__ or self.__doc__"
    def method(self):
        "I am: spam.method.__doc__ or self.method.__doc__"
        print(self.__doc__)
        print(self.method.__doc__ )
```

```
print(func.__doc__)
print(spam.__doc__)
print(spam.method.__doc__)
X = spam()
X.method()
```

```
I am: docstr. func.__doc__
I am: spam.__doc__ or docstr. spam.__doc__ or self.__doc__
I am: spam.method.__doc__ or self.method.__doc__
I am: spam.__doc__ or docstr. spam.__doc__ or self.__doc__
I am: spam.method.__doc__ or self.method.__doc__
```

Эмпирическое правило “рекомендуемой методики” в Python предполагает применение строк документации для документирования функциональности (что объекты делают) и комментариев # для документирования на микро-уровнях (как работают загадочные порции кода).

# Классы или модули

## классы

- реализуют новые полнофункциональные объекты;
- создаются посредством операторов class;
- используются путем обращения к ним;
- всегда находятся внутри модуля.

## модули

- реализуют пакеты данных/логики;
- создаются с помощью файлов с кодом на Python или расширений на других язы
- используются путем импортирования;
- формируют верхний уровень структуры программы на Python.

Классы также поддерживают дополнительные возможности, отсутствующие у модулей, такие как перегрузка операций, создание множества экземпляров и наследование.

# Доступ к атрибутам. Геттеры и сеттеры

В Python атрибуты объектов и методы обычно являются общедоступными, поэтому предполагается, что тот, кто их использует, не станет этим злоупотреблять (это иногда называют «установкой на взрослых людей»). Отдельные объектно-ориентированные языки поддерживают закрытые атрибуты объектов, к которым нельзя получить доступ напрямую. Программистам часто приходится писать **геттеры** и **сеттеры**, чтобы прочитать и записать значения таких атрибутов. В Python нет закрытых атрибутов, но вы можете написать геттеры и сеттеры для атрибутов с обфусцированными именами, чтобы создать нечто подобное. Далее в примере мы определим класс **Duck**, имеющий один атрибут экземпляра **hidden_name**. Мы не хотим, чтобы кто-то мог обратиться к атрибуту напрямую, по этому определим два метода: геттер (**get_name()**) и сеттер (**set_name()**). К каждому из них обращается свойство с именем **name**.

*Обфускация - преднамеренное сокрытие программного кода путём его запутывания с сохранением работоспособности продукта.*

*Процедура выполняется вручную (долго, сложно привести в исходный вид — то есть «деобфусцировать») или автоматически (быстро, выполняется специальными программами «обфускаторами» с функцией «деобфускации»). Задачу выполняет программист с той целью, чтобы никакой другой программист не смог прочитать программный код и расшифровать алгоритмы обфускатора.*

# Доступ к атрибутам. Геттеры и сеттеры

```
class Duck():
    def __init__(self, input_name):
        self.hidden_name = input_name
    def get_name(self):
        print('inside the getter')
        return self.hidden_name
    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name
```

```
don = Duck('Donald')
print(don.get_name())
don.set_name('Donna')
print(don.get_name())
```

```
inside the getter
Donald
inside the setter
inside the getter
Donna
```

# Доступ к атрибутам. Геттеры и сеттеры

Более питонским решением для закрытия атрибутов является использование свойств. Существует два способа сделать это  Первый - добавить конструкцию
name = property(get_name, set_name)
последней строкой в предыдущее определение класса Duck

```
class Duck():
    def __init__(self, input_name):
        self.hidden_name = input_name
    def get_name(self):
        print('inside the getter')
        return self.hidden_name
    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name
    name = property(get_name, set_name)

don = Duck('Donald')
print(don.get_name())
don.set_name('Donna')
print(don.get_name())
```

Метод property() обеспечивает интерфейс для атрибутов экземпляра класса. Он инкапсулирует атрибуты экземпляров и предоставляет свойства. Метод property() принимает на вход методы get, set и delete, и возвращает объекты класса property.

# Доступ к атрибутам. Геттеры и сеттеры

Старый подход с геттерами и сеттерами тоже сработает. Но теперь вы также можете использовать свойство name, чтобы получить и установить скрытое имя:

```
class Duck():
    def __init__(self, input_name):
        self.hidden_name = input_name
    def get_name(self):
        print('inside the getter')
        return self.hidden_name
    def set_name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name
    name = property(get_name, set_name)

don = Duck('Donald')
print(don.name)
don.name = 'Donna'
print(don.name)
```

```
inside the getter
Donald
inside the setter
inside the getter
Donna
```

# Доступ к атрибутам. Геттеры и сеттеры

Во втором методе добавляем декораторы и заменяете имена методов get_name и set_name на name

- **@property** – размещается перед геттером;
- **@name.setter** – размещается перед сеттером.

```
class Duck():
    def __init__(self, input_name):
        self.hidden_name = input_name

    @property
    def name(self):
        print('inside the getter')
        return self.hidden_name
    @name.setter
    def name(self, input_name):
        print('inside the setter')
        self.hidden_name = input_name
```

```
fowl = Duck('Howard')
print(fowl.name)
fowl.name = 'Donald'
print(fowl.name)
```

```
inside the getter
Howard
inside the setter
inside the getter
Donald
```

# Доступ к атрибутам. Геттеры и сеттеры

Python предлагает соглашения по именованию для атрибутов, которые не должны быть видимы за пределами определения их классов: имена начинаются с двух нижних подчеркиваний (__).

```
class Duck():
    def __init__(self, input_name):
        self.__name = input_name

    @property
    def name(self):
        print('inside the getter')
        return self.__name
    @name.setter
    def name(self, input_name):
        print('inside the setter')
        self.__name = input_name


fowl = Duck('Howard')
print(fowl.name)
fowl.name = 'Donald'
print(fowl.name)
```

```
inside the getter
Howard
inside the setter
inside the getter
Donald
```

# Доступ к атрибутам. Геттеры и сеттеры

При этом вы не можете получить доступ к атрибуту **__name:**

```
print(fowl.__name)
```

```
AttributeError: 'Duck' object has no attribute '__name'
```

Это соглашение по именованию не делает атрибут полностью закрытым:
Python искажает имя, чтобы внешний код не наткнулся на этот атрибут.

*Но если вам интересно и вы обещаете никому не рассказывать, покажем вам, как будет выглядеть атрибут:*

```
print(fowl._Duck__name)
```

```
Donald
```

*Обратите внимание на то, что на экране не появилась надпись inside the getter. Хотя эта защита неидеальна, искажение имен препятствует случайному или преднамеренному доступу к атрибуту.*

# Классы данных

Многие люди создают объекты для хранения данных (с помощью атрибутов объектов), а не для задания определенного поведения (с помощью методов). Вы уже видели, как именованные кортежи могут стать альтернативным хранилищем данных В Python 3.7 появились классы данных.

```
from dataclasses import dataclass
@dataclass
class AnimalClass:
    name: str
    habitat: str
    teeth: int = 0

snowman = AnimalClass('yeti', 'Himalayas', 46)
duck = AnimalClass(habitat='lake', name='duck')
print(snowman)
print(duck)
```

```
AnimalClass(name='yeti', habitat='Himalayas', teeth=46)
AnimalClass(name='duck', habitat='lake', teeth=0)
```

# Резюме

1 Абстрактный суперкласс - это класс, который вызывает метод, но не наследует и не определяет его; он ожидает заполнения метода подклассом. Абстрактные суперклассы часто используются в качестве способа обобщения классов, когда поведение не может быть спрогнозировано до написания кода более специфического подкласса. Объектно-ориентированные фреймворки также применяют их как способ направления на определяемые клиентом настраиваемые операции.

2. Когда простой оператор присваивания (X = Y) появляется на верхнем уровне оператора class, он присоединяет к классу атрибут данных (Класс.X). Подобно всем атрибутам класса он будет разделяться всеми экземплярами; тем не менее, атрибуты данных не являются вызываемыми функциями методов.

3. В классе должен вручную вызываться метод **__ init__** суперкласса, если в нем определяется собственный конструктор**__ init__** и нужно, чтобы код конструктора суперкласса по-прежнему выполнялся. Сам интерпретатор Python автоматически запускает только один конструктор - самый нижний в дереве. Конструктор суперкласса обычно вызывается через имя класса с передачей экземпляра **self** вручную: **Суперкласс.__ init__ (self, . . .)**.

4. Чтобы вместо замещения дополнить унаследованный метод, его понадобится повторно определить в подклассе, но внутри этой новой версии вручную вызвать версию метода из суперкласса с передачей ей экземпляра **self**: **Суперкласс.метод(self, ...)**.

5. Класс представляет собой локальную область видимости и имеет доступ к объемлющим локальным областям видимости, но он не служит в качестве локальной области видимости для добавочного вложенного кода. Подобно модулям после выполнения оператора class локальная область видимости класса превращается в пространство имен атрибутов.

# Основы ООП

## Лекция 6. Перегрузка операций

# Основы перегрузки

Вообще говоря, «перегрузка операций» просто означает перехват встроенных операций в методах класса - Python автоматически вызывает ваши методы, когда экземпляры класса обнаруживаются во встроенных операциях, и возвращаемое значение
вашего метода становится результатом соответствующей операции. Ниже приведен
обзор ключевых идей, лежащих в основе перегрузки.

• Перегрузка операций позволяет классам перехватывать нормальные операции
Python.

• Классы могут перегружать все операции выражений Python.

• Классы также могут перегружать встроенные операции, такие как вывод, вызовы функций, доступ к атрибутам и т.д.

• Перегрузка делает экземпляры классов более похожими на встроенные типы.

• Перегрузка реализуется за счет предоставления особым образом именованных
методов в классе.

Другими словами, когда в классе предоставляются особым образом именованные методы, тогда Python автоматически вызывает их в случае появления экземпляров данного класса в ассоциированных с ними выражениях. Ваш класс снабжает создаваемые из него объекты экземпляров поведением соответствующей операции. В случае использования такие методы позволяют классам эмулировать интерфейсы встроенных объектов и потому выглядеть более согласованными.

# Конструкторы и выражения:__ init__ и___sub

Рассмотрим метод для перехвата создания экземпляра **(__ init__ )**, а также метод для отлавливания выражений вычитания **(__ sub__ )**. Специальные методы подобного рода являются привязками, которые дают возможность соединяться со встроенными операциями.

```
class Number:
    def __init__ (self, start):          # Для Number(start)
        self.data = start
    def __sub__ (self, other) :          # Для экземпляр - other
        return Number(self.data - other) # Результатом будет новый экземпляр

X = Number(5) # Number. init (X, 5)
Y = X - 2 # Number. sub (X, 2)
print(Y.data) # Y является новым экземпляром Number
```

3

Реализованный в коде метод конструктора **__init__** является наиболее употребительным методом перегрузки операций в Python; он присутствует в большинстве классов и применяется для инициализации вновь созданного объекта экземпляра с использованием любых аргументов, указываемых после имени класса. Метод **__sub__** исполняет роль бинарной операции аналогично методу **__add__**, перехватывая выражения вычитания и возвращая в качестве своего результата новый экземпляр класса (попутно выполняя **__init__** ). Формально создание экземпляра сначала запускает метод **__new__** , который создает и возвращает новый объект экземпляра, передаваемый затем в метод__ init__ для инициализации. Тем не менее, поскольку метод **__new__** имеет встроенную реализацию и переопределяется лишь в крайне ограниченных ситуациях, почти все классы Python инициализируются за счет определения метода **__ init__** .

# Индексирование и нарезание: __ getitem__ и __ setitem__

Если метод **__getitem__** определен в классе (или унаследован им), автоматически вызывается для операций индексирования экземпляров. В случае появления экземпляра X в выражении индексирования вроде X [i] интерпретатор Python вызывает метод **__ getitem__** , унаследованный экземпляром, с передачей X в первом аргументе и индекса, указанного в квадратных скобках, во втором.

```
class Indexer:
    def __getitem__(self, index):
        return index ** 2


X = Indexer()
print(X[2])    # Для X[i] вызывается X.__getitem__ (i)
```

Интересно отметить, что в дополнение к индексированию метод **__getitem__** также вызывается для выражений срезов. Однако на самом деле границы нарезания упаковываются в объект среза и передаются реализации индексирования списка. В действительности вы всегда можете передавать объект среза вручную - синтаксис среза по большей части является синтаксическим сахаром для индексирования объекта среза.

```
L = [2, 3, 5, 7, 9]
print(L[slice(1, None)])
```

```
[3, 5, 7, 9]
```

# Индексирование и нарезание: __ getitem__ и __ setitem__

В классах с методом **__getitem__** это важно - в Python 3.x данный метод будет вызываться для базового индексирования (с индексом) и нарезания (с объектом среза).

```
class Indexer:
    data = [5, 6, 7, 8, 9]
    def __getitem__ (self, index) : # Вызывается для индексирования или нарезания
        print('getitem:', index)
        return self.data[index] # Выполняется индексирование или нарезание

X = Indexer()
print(X[:3])
```

```
getitem: slice(None, 3, None)
[5, 6, 7]
```

Там, где необходимо, метод **__getitem__** может проверять тип своего аргумента и извлекать границы объекта среза - объекты срезов имеют атрибуты **start**, **stop** и **step**, любой из которых можно опустить, указав **None**.

```
class Indexer:
    def __getitem__(self, index):
        if isinstance(index, int): # Проверка режима использования
            print('indexing', index)
        else:
            print('slicing', index.start, index.stop, index.step)

X = Indexer()
X[1:99:2]
```

```
slicing 1 99 2
```

# Индексирование и нарезание: __ getitem__ и __ setitem__

В качестве связанного замечания: при перехвате индексирования в Python 3.x не применяйте (вероятно, неудачно названный) метод **__index__** - он возвращает целочисленное значение для экземпляра и используется встроенными функциями, которые выполняют преобразование в строки цифр.

```
class C:
    def __index__(self):
        return 255

x = C()
print(hex(x))
```

```
0xff
```

Оператор **for** работает путем многократного индексирования последовательности от нуля до более высоких индексов, пока не обнаружится исключение выхода за границы **IndexError**. По этой причине **__getitem__** также оказывается одним из способов перегрузки итерации в Python - если он определен, тогда циклы for на каждом проходе вызывают метод **__getitem__** класса с последовательно увеличивающимися смещениями.
На самом деле это случай “реализовав одну возможность, получаем бесплатно целый набор”. Любой класс, поддерживающий циклы for, автоматически поддерживает
все итерационные контексты в Python. Например, проверка членства in, списковые включения, встроенная функция **map**, присваивания списков и кортежей, а также конструкторы типов будут автоматически вызывать метод **__getitem__.**

# Индексирование и нарезание: __ getitem__ и __ setitem__

```
class StepperIndex:
    def __getitem__(self, i):
        return self.data[i]

X = StepperIndex()
X.data = "Spam"

print(X[1])
for item in X:
    print(item, end=' ')

print()
print ('p' in X)

print([c for c in X])
```

```
p
S p a m
True
['S', 'p', 'a', 'm']
```

# Итерируемые объекты: __ iter__ и ___next__

Несмотря на то что описанный в предыдущем разделе подход с методом **__ getitem__** работает, в действительности он является просто запасным вариантом для итерации. В настоящее время все итерационные контексты языка Python перед **__ getitem__** будут сначала пытаться вызвать метод **___iter__**. То есть для многократного индексирования объекта они выбирают протокол итерации. Формально итерационные контексты работают путем передачи итерируемого объекта встроенной функции **iter** для вызова метода **__ iter__** , который должен возвратить итерируемый объект. Когда этот метод **__ next__** объекта итератора предоставлен, Python будет многократно вызывать его для выпуска элементов до тех пор, пока не сгенерируется исключение **Stopiteration**. В качестве удобства для выполнения итерации вручную доступна также встроенная функция next - вызов next (I) представляет собой то же самое, что и **I.__ next__ ()**.

# Итерируемые объекты: __ iter__ и ___next__

```
class Squares:
    def __init__ (self, start, stop): # Сохранить состояние при создании
        self.value = start - 1
        self.stop = stop
    def __iter__ (self) : # Получить объект итератора при вызове iter
        return self
    def __next__ (self) : # Возвратить квадрат на каждой итерации
        if self.value == self.stop: # Также вызывается встроенной функцией next
            raise StopIteration
        self.value += 1
        return self.value ** 2

for i in Squares(1, 5) : # for вызывает встроенную функцию iter,
    # которая вызывает__iter__
    print(i, end=' ') # Каждая итерация вызывает__next__
```

```
1 4 9 16 25
```

Здесь объект итератора, возвращаемый **__ iter__** , представляет собой просто экземпляр **self**, потому что метод**__ next__** является частью самого класса **Squares**. В более сложных сценариях объект итератора может быть определен как отдельный класс и объект с собственной информацией о состоянии для поддержки множества активных итераций по тем же самым данным.

# Итерируемые объекты: __ iter__ и ___next__

Эквивалентная реализация такого итерируемого объекта посредством **__ getitem__** может оказаться менее естественной, поскольку цикл **for** проходил бы тогда через все смещения с нуля и выше; передаваемые смещения были бы лишь косвенно связанными с диапазоном выпускаемых значений (0. .N) пришлось бы отображать на start. .stop). Из-за того, что объекты**__ iter__** предохраняют явно управляемое поведение между вызовами next, они могут быть более универсальными, чем **__ getitem__** . С другой стороны, итерируемые объекты, основанные на__ iter__ , временами могут быть более сложными и менее функциональными по сравнению с такими объектами, основанными на **__ getitem__** . Они на самом деле предназначены для итерации, а не для произвольного индексирования - в них вообще не перегружается выражение индексирования, хотя их элементы можно собрать в последовательность вроде списка и сделать доступными другие операции. Схема с **__iter___**также реализована для всех остальных итерационных контекстов, которые мы видели в действии с методом **__ getitem__** - проверка членства, конструкторы типов, присваивание последовательностей и т.д. Тем не менее, в отличие от предыдущего примера с **__ getitem__** мы также должны знать, что метод **__iter__** класса может быть предназначен только для единственного обхода, а не для множества. Классы явным образом выбирают поведение просмотра в своем коде. Скажем, поскольку текущий метод **__ iter__** класса Squares всегда возвращает self с только одной копией состояния итерации, он обеспечивает одноразовую итерацию; после итерации экземпляр данного класса становится пустым. Повторный вызов **__ iter__** на том же самом экземпляре снова возвращает **self** независимо от состояния, в котором он был оставлен. Как правило, для каждой новой итерации потребуется создавать новый итерируемый объект

# Альтернативная реализация: __ iter__ плюс yield

В ряде приложений имеется возможность свести к минимуму требования к коду итерируемых объектов, определяемых пользователем, за счет комбинирования исследованного метода **__iter__** и оператора генераторных функций **yield**. Поскольку генераторные функции автоматически сохраняют состояние локальных переменных и создают обязательные методы итераторов, они хорошо подходят для этой роли и дополняют предохранение состояния и другие полезные вещи, получаемые от классов.

Вспомните, что любая функция, которая содержит оператор **yield**, превращается в генераторную функцию. При вызове она возвращает новый генераторный объект с автоматическим предохранением локальной области видимости и позиции в коде, автоматически созданным методом **__ iter__** , просто возвращающим сам объект, и автоматически созданным методом **__next__**, запускающим функцию или возобновляющим ее выполнение с места, которое она оставила в последний раз.

Даже если генераторная функция с оператором **yield** оказывается методом по имени **__ iter___** : всякий раз, когда такой метод вызывается инструментом итерационного контекста, он будет возвращать новый генераторный объект с необходимым методом **__ next__.** В качестве дополнительного бонуса генераторные функции, реализованные как методы в классах, имеют доступ к сохраненному состоянию в атрибутах экземпляров и в переменных локальной области видимости.

# Альтернативная реализация: __ iter__ плюс yield

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

    def __iter__(self):
        for value in range(self.start, self.stop + 1):
            yield value ** 2

for i in Squares(1, 5):
    print (i, end=' ')
```

```
1 4 9 16 25
```

Реализация генератора как **__ iter__** устраняет посредника в коде, хотя обе схемы в итоге создают новый генераторный объект для каждой итерации:

• при наличии **__ iter__** итерация запускает метод **___iter__** , который возвращает новый генераторный объект с методом__ next__ ;

• при отсутствии **__ iter__** в коде производится вызов для создания генераторного объекта, метод **__ iter___** которого возвращает сам объект.

# Членство: __ contains__ ,__ iter__ и___getitem

На самом деле история об итерациях даже обширнее, чем было показано до сих пор. Перегрузка операций часто разделяется на уровни, классы могут предоставлять специфические методы или более универсальные альтернативы, применяемые в качестве запасных вариантов. В области итераций классы могут реализовывать операцию проверки членства in в виде итерации с применением методов **__iter__** или **___getitem__** . Тем не менее, для поддержки более специфической операции проверки членства в классах может быть предусмотрен метод **__contains__** - когда он присутствует, ему отдается предпочтение перед методом **__iter__** , который предпочтительнее **__getitem__** . Метод **__contains__** должен определять членство для отображений как применение к ключам (и может использовать быстрый просмотр), а для последовательностей как поиск.

# Членство:
# __ contains__ ,__ iter__ и___getitem

```
class Iters:
    def __init__ (self, value):
        self.data = value
    def __getitem__ (self, i) : # Запасной вариант для итерации
        print (f'get [{i}] : ', end='') # Также для индексирования, нарезания
        return self.data[i]
    def __iter__ (self) : # Предпочтительнее для итерации
        print ('iter=> ', end=' ') # Допускает только один активный итератор
        self.ix = 0
        return self
    def __next__ (self):
        print('next:', end='')
        if self.ix == len(self.data) :
            raise StopIteration
        item = self.data[self.ix]
        self.ix += 1
        return item
    def __contains__ (self, x) :      # Предпочтительнее для операции in
        print('contains : ', end='')
        return x in self.data
```

# Членство: __ contains__ ,__ iter__ и___getitem

```python
if __name__ == '__main__' :
    X = Iters( [1, 2, 3, 4, 5])       # Создать экземпляр
    print (3 in X)                    # Членство
    for i in X:                       # Циклы for
        print(i, end=' | ' )

print()
print([i ** 2 for i in X])
print( list(map(bin, X)))
I = iter(X)
while True:
    try:
        print(next(I), end=' @')
    except StopIteration:
        break
```

```
contains : True
iter=>  next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=>  next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=>  next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=>  next:1 @next:2 @next:3 @next:4 @next:5 @next:
```

# Членство: __ contains__ ,__ iter__ и___getitem

Однако посмотрите, что произойдет в выводе кода, если мы закомментируем метод __ **contains**__ - теперь операция проверки членства направляется универсальному методу__ **iter**__ :

```
iter=>  next:next:next:True
iter=>  next:1 | next:2 | next:3 | next:4 | next:5 | next:
iter=>  next:next:next:next:next:next:[1, 4, 9, 16, 25]
iter=>  next:next:next:next:next:next:['0b1', '0b10', '0b11', '0b100', '0b101']
iter=>  next:1 @next:2 @next:3 @next:4 @next:5 @next:
```

Наконец, вот вывод в ситуации, когда закомментирован код методов __ **contains**__ и __ **iter**__ - для операции проверки членства и других итерационных контекстов вызывается запасной вариант индексирования __ **getitem**__ с последовательно растущими индексами, пока он не сгенерирует исключение **IndexError**:

```
get [0] : get [1] : get [2] : True
get [0] : 1 | get [1] : 2 | get [2] : 3 | get [3] : 4 | get [4] : 5 | get [5] :
get [0] : get [1] : get [2] : get [3] : get [4] : get [5] : [1, 4, 9, 16, 25]
get [0] : get [1] : get [2] : get [3] : get [4] : get [5] : ['0b1', '0b10', '0b11', '0b100', '0b101']
get [0] : 1 @get [1] : 2 @get [2] : 3 @get [3] : 4 @get [4] : 5 @get [5] :
```

# Ссылка на атрибуты

Метод **__ getattr__** перехватывает ссылки на атрибуты. Он вызывается с именем атрибута в виде строки всякий раз, когда вы пытаетесь уточнить экземпляр с помощью неопределенного (несуществующего) имени атрибута. Метод **__ getattr__** не вызывается, если Python может найти атрибут с применением процедуры поиска в дереве наследования. Из-за своего поведения метод **__ getattr__** удобен в качестве привязки, обеспечивающей реагирование на запросы атрибутов в обобщенной манере. Он обычно используется для делегирования вызовов внедренным (или “вложенным”) объектам из промежуточного объекта контроллера. Метод **__getattr__** также может применяться для адаптации классов к интерфейсу или последующего добавления средств доступа к атрибутам - логики в методе, которая проверяет достоверность или вычисляет значение атрибута после того, как он уже используется через простую точечную запись. Базовый механизм, лежащий в основе достижения указанных целей, прямолинеен. В приведенном ниже классе перехватываются ссылки на атрибуты, динамически вычисляется значение для одного атрибута и генерируется ошибка для остальных неподдерживаемых атрибутов с помощью оператора **raise**.

# Ссылка на атрибуты

Здесь класс **Empty** и его экземпляр **X** не имеют собственных реальных атрибутов, поэтому доступ к **X.age** направляется методу **__getattr__** ; **self** присваивается экземпляр (X), а attrname - строка с именем неопределенного атрибута **(' age ').** Класс делает **age** похожим на реальный атрибут, возвращая действительное значение в качестве результата выражения уточнения X. аде (40). В сущности, **age** становится динамически вычисляемым атрибутом - его значение формируется выполняющимся кодом, а не извлечением какого-то объекта. Для атрибутов, которые класс не знает, как обрабатывать, метод **__getattr__** генерирует встроенное исключение **AttributeError**, сообщая Python о том, что они являются по-настоящему неопределенными именами; обращение к **X.name** инициирует ошибку.

```
class Empty:
    def __getattr__(self, attrname) : # Вызывается для неопределенного атрибута в self
        if attrname == 'age':
            return 40
        else:
            raise AttributeError(attrname)
```

```
X = Empty()
print(X.age)
print(X.name)
```

```
40
AttributeError: name
```

# Присваивание и удаление атрибутов

В рамках той же области метод **__setattr__** перехватывает все присваивания значений атрибутам. Если данный метод определен или унаследован, тогда выражение ***self.атрибут = значение*** становится ***self.__ setattr__ ('атрибут', значение).***
Подобно**__ getattr__** это позволяет классу перехватывать изменения атрибутов и выполнять желаемые проверки достоверности или преобразования. Тем не менее, использовать метод**__ setattr__** несколько сложнее, т.к. присваивание значения любому атрибуту в **self** внутри **__ setattr__** снова вызывает **__setattr__** , что потенциально может стать причиной бесконечного цикла рекурсии (и довольно быстро привести к исключению, связанному с переполнением стека). На самом деле сказанное применимо ко всем присваиваниям атрибутов в **self**, где угодно в классе - все они направляются методу **__setattr__** , даже присваивания, находящиеся в других методах, и присваивания именам, отличным от тех, которые могут запускать метод **__setattr__** в первую очередь. Не забывайте, что метод **___setattr__** перехватывает **ВСЕ** присваивания значений атрибутам.

# Присваивание и удаление атрибутов

Если вы хотите использовать метод **__ setattr__** , то можете избежать циклов, реализуя присваивания значений атрибутам экземпляра в виде присваиваний ключам словаря атрибутов. То есть применяйте **self.__ dict__ [ 'name' ] = x**, но не **self.name = x**; из-за того, что вы не выполняете присваивание самому **__ dict__** .

```
class Accesscontrol:
    def __setattr__(self, attr, value):
        if attr == 'age' :
            self.__dict__[attr] = value + 10
        else:
            raise AttributeError(attr + ' not allowed')

X = Accesscontrol()
X.age = 40 # Вызывается__setattr__
print(X.age)

X.name = 'Bob'
```

50

AttributeError: name not allowed

# Присваивание и удаление атрибутов

В Python существуют и другие способы управления доступом к атрибутам.

• Метод **__getattribute__** перехватывает операции извлечения всех атрибутов, не только тех, которые не определены, но при его применении вы должны соблюдать большую осторожность, чем с **__ getattr__** , чтобы избегать зацикливания.

• Встроенная функция **property** позволяет ассоциировать методы с операциями извлечения и установки для специфического атрибута класса.

• Дескрипторы предоставляют протокол для ассоциирования методов **__get__** и **__ set__** класса с операциями доступа к специфическому атрибуту класса.

• Атрибуты слотов объявляются в классах, но создают неявное хранилище в каждом экземпляре.

# Строковое представление: __repr__ и __str__

В приведенном ниже коде используются конструктор **__init__** и метод перегрузки **__ add__** , с которыми мы встречались ранее. Стандартное отображение объектов экземпляров для класса вроде этого не приносит особой пользы и не может считаться эстетически привлекательным.

```
class adder:
    def __init__(self, value=0):
        self.data = value           # Инициализировать данные
    def __add__(self, other):
        self.data += other          # Добавить other на месте
X = adder () # Стандартные отображения
print(X)
```

```
<__main__.adder object at 0x00702DA8>
```

```
class addrepr(adder):      # Унаследовать__init__ , __ add__
    def __repr__(self):    # Добавить строковое представление
        return f'addrepr({self.data})' # Преобразовать в строку

X = addrepr(2)
print(X)
```

```
addrepr(2)
```

# Строковое представление: __repr__ и __str__

Python предлагает два метода отображения, призванные поддерживать отличающееся отображение для разной аудитории.

• Метод **__ str__** сначала опробуется для операции print и встроенной функции **str** (внутренний эквивалент которой запускает операция **print**). В общем случае он должен возвратить отображение, дружественное к пользователю.

• Метод **__ repr__** используется во всех остальных контекстах: для эхо-вывода в интерактивной подсказке, функции герг и вложенных появлений, а также **print** и **str**, если метод **__ str__** отсутствует. В общем случае он должен возвратить строку как в коде, которую можно было бы применять для воссоздания объекта, или детальное отображение для разработчиков.

То есть **__ repr__** используется везде, исключая print и str, когда метод **___str__** определен. Это означает, что вы можете реализовать метод **__ repr__** для определения единственного формата отображения, применяемого повсюду, и метод **__ str__** либо для поддержки единственно **print** и **str**, либо чтобы предоставить для них альтернативное отображение.

Несмотря на простоту использования, во-первых, имейте в виду, что **__ str__** и **___repr__** обязаны возвращать строки', другие результирующие типы не преобразуются и вызывают ошибки, так что необходимо обеспечить их обработку инструментом преобразования в строку. Во-вторых, в зависимости от логики преобразования в строки дружественное к пользователю отображение **__ str__** может применяться, только когда объекты находятся на верхнем уровне операции print; объекты, вложенные внутрь более крупных объектов, могут по-прежнему выводиться посредством **__repr__** либо их стандартных методов.

# Использование с правой стороны и на месте: __ radd и__ iadd

Следующая группа методов перегрузки расширяет функциональность методов бинарных операций, таких как__ **add**__ и___**sub**__ (вызываемых для + и -), которые мы уже видели. Как упоминалось ранее, одна из причин существования настолько большого количества методов перегрузки операций связана с тем, что они встречаются во многих разновидностях - для каждого бинарного выражения мы можем реализовать варианты с левой стороны, с правой стороны и на месте. Хотя также применяются стандартные реализации, когда не написан код для всех трех вариантов, именно роли ваших объектов диктуют, код скольких вариантов потребуется предоставить.

Реализованные до настоящего времени методы__ **add**__ формально не поддерживают использование объектов экземпляров с правой стороны операции +:

```
class Adder:
    def __init__(self, value=0):
        self.data = value
    def __add__ (self, other):
        return self.data + other

x = Adder (5)
print(x + 2)
print(2 + x)
```

```
7
TypeError: unsupported operand type(s) for +: 'int' and 'Adder'
```

# Использование с правой стороны и на месте: __ radd и__ iadd

Чтобы реализовать более универсальные выражения и тем самым поддерживать коммутативные операции, необходимо также написать код метода**__ radd__** .
Интерпретатор Python вызывает**__ radd__** , только когда объектом с правой стороны операции + является экземпляр вашего класса, но объект с левой стороны не относится к экземплярам вашего класса.

```
class Commuterl:
    def __init__ (self, val):
        self.val = val
    def __add__ (self, other):
        print('add', self.val, other)
        return self.val + other
    def __radd__ (self, other):
        print('radd', self.val, other)
        return other + self.val
x = Commuterl(88)
y = Commuterl(99)

print(x + 1)
print(1 + y)
```

```
add 88 1
89
radd 99 1
100
```

Обратите внимание на реверсирование порядка в**__ radd__** : на самом деле self находится с правой стороны операции + , a other - с левой стороны. Также учтите, что x и y здесь являются экземплярами того же самого класса; когда в выражении смешиваются экземпляры разных классов, Python отдает предпочтение классу экземпляра с левой стороны. При сложении двух экземпляров Python выполняет метод**__ add__** , который в свою очередь запускает**__ radd__** , упрощая левый операнд.

# Использование с правой стороны и на месте: __ radd и__ iadd

Чтобы реализовать дополненное сложение на месте (+=), понадобится написать код либо **__iadd__** , либо**__add__** . Второй метод используется, если отсутствует первый. На самом деле по указанной причине классы Commuter из предыдущего раздела уже поддерживают операцию +=: интерпретатор Python выполняет**__ add__** и присваивает результат. Однако метод **__iadd__** позволяет более эффективно реализовывать изменения на месте, где это применимо.

```
class Number:
    def __init__ (self, val):
        self.val = val
    def __iadd__(self, other) : # Явный метод__iadd__ : x t= y
        self.val += other       # Обычно возвращает self
        return self

x = Number (5)
x += 1
x += 1
print(x.val)
```

7

Для изменяемых объектов метод **__iadd__** часто можно специализировать для выполнения более быстрых изменений на месте:

```
y = Number([4]) # Изменение на месте быстрее, чем +
y += [2]
y += [3]
print(y.val)
```

[4, 2, 3]

# Выражения вызовов:__ call__

Перейдем к нашему следующему методу перегрузки: метод **__call__** вызывается при вызове вашего экземпляра. Нет, это вовсе не циклическое определение - если метод **__call__** определен, то Python выполняет его для выражений вызова функций, применяемых к вашему экземпляру, с передачей ему любых позиционных или ключевых аргументов, которые были отправлены. Это позволят экземплярам соответствовать API-интерфейсу на основе функций. Если кратко, метод **__call__** поддерживает все режимы передачи аргументов.

```
class Callee:
    def __call__(self, *pargs, **kargs) : # Перехватывает вызовы экземпляра
            print('Called: ' , pargs, kargs) # Принимает произвольные аргументы
C = Callee()
C(1, 2, 3)
C(1, 2, 3, x = 4, y = 5)
```

```
Called:  (1, 2, 3) {}
Called:  (1, 2, 3) {'x': 4, 'y': 5}
```

Метод **__ call__** может стать более полезным при взаимодействии с API интерфейсами (т.е. библиотеками), ожидающими функций - он позволяет реализовывать объекты, которые соответствуют ожидаемому интерфейсу вызова функций, но также предохраняют информацию о состоянии и другие активы классов, такие как наследование. Фактически**__ call__** можно считать третьим по частоте использования методом перегрузки операций после конструктора**__ init__** и альтернативных форматов отображения**__ str__** и**___repr__** .

# Деструктор __del__

Метод деструктора **__del__** , запускается автоматически при возвращении пространства памяти, занимаемого экземпляром (т.е. во время “сборки мусора”) .

```
class Life:
    def __init__(self, name='unknown'):
        print('Hello ' + name)
        self.name = name
    def live(self):
        print(self.name)
    def __del__ (self):
        print('Goodbye ' + self.name)

brian = Life('Brian')
brian.live()
brian = 'loretta'
```

```
Hello Brian
Brian
Goodbye Brian
```

Когда объекту brian присваивается строка, ссылка на экземпляр Life утрачивается и потому запускается его метод деструктора. Это работает и может быть полезным для реализации действий по очистке, таких как закрытие подключения к серверу.
Однако деструкторы не настолько часто используется в Python

# Деструктор __del__

• **Необходимость.** С одной стороны, деструкторы не настолько полезны в Python, как в ряде других языков ООП. Поскольку Python автоматически возвращает все пространство памяти, занимаемое экземпляром, когда экземпляр уничтожается, деструкторы не требуются для управления памятью. В текущей реализации CPython также не нужно закрывать в деструкторах файловые объекты, удерживаемые экземпляром, потому что при уничтожении экземпляра они автоматически закрываются. Тем не менее, в главе 9 первого тома упоминалось, что иногда по-прежнему лучше в любом случае запускать методы закрытия файлов, т.к. поведение автоматического закрытия может варьироваться в альтернативных реализациях Python (скажем, в Jython).

• **Предсказуемость.** С другой стороны, не всегда легко спрогнозировать, когда экземпляр будет уничтожен. В ряде случаев внутри системных таблиц могут присутствовать долго существующие ссылки на ваши объекты, которые препятствуют выполнению деструкторов, когда программа ожидает их запуска. Вдобавок Python вовсе не гарантирует, что методы деструкторов будут вызваны для объектов, которые все еще существуют при завершении работы интерпретатора.

# Деструктор __del__

• **Исключения.** На самом деле метод **__ del__** может оказаться сложным в применении даже по более тонким причинам. Например, возникающие в нем исключения просто выводят предупреждающее сообщение в **sys. stderr** (стандартный поток ошибок), а не генерируют событие исключения, из-за непредсказуемого контекста, в котором метод **__ del__** выполняется сборщиком мусора - не всегда возможно знать, куда такое исключение должно быть доставлено.

• **Циклы.** Кроме того, циклические (круговые) ссылки между объектами могут препятствовать запуску сборки мусора, когда она ожидается. По умолчанию включенный необязательный обнаружитель циклов способен со временем автоматически собирать объекты подобного рода, но только если они не имеют методов **__del__** . Поскольку это относительно малоизвестно, здесь мы проигнорируем дальнейшие детали; за дополнительной информацией обращайтесь к описанию метода**__ del__** и модуля сборщика мусора дс в стандартных руководствах по Python.

***Из-за таких недостатков часто лучше реализовывать действия завершения в явно вызываемом методе (скажем, shutdown).***

# Резюме

1. Классы могут поддерживать итерацию путем определения (или наследования) метода__ **getitem**__ или___**iter**__ . Во всех итерационных контекстах Python сначала пытается применить метод__ **iter**__ , возвращающий объект, который поддерживает протокол итерации с помощью метода__ **next**__ : если поиск в иерархии наследования не привел к нахождению метода__ **iter**__ , тогда Python прибегает к методу индексирования__ **getitem**__ , многократно вызывая его с последовательно увеличивающимися индексами. В случае использования оператора **yield** метод__ **next**__ может быть создан автоматически.

**2.** Методы__ **str**__ и___ **repr**__ реализуют отображения объектов при выводе. Первый вызывается встроенными функциями **print** и **str**; второй вызывается **print** и **str**, если отсутствует__ **str**__ , и всегда вызывается встроенной функцией **repr**, при эхо-выводе в интерактивной подсказке и для вложенных появлений. То есть метод__ **repr**__ применяется везде, исключая **print** и **str,** когда определен метод **__str__** Метод__ **str**__ обычно используется для отображений, дружественных к пользователю, а **__repr__** предоставляет для объекта дополнительные детали или форму как в коде.

**3.** Нарезание перехватывается методом индексирования **__getitem__** : он вызывается с объектом среза, а не с простым целочисленным индексом, и при необходимости объекты срезов можно передавать или ожидать.

# Резюме

4. Сложение на месте сначала пытается использовать метод__ **iadd__** и затем __ **add__** с присваиванием. Та же схема применяется для всех бинарных операций. Для правостороннего сложения также доступен метод__ **radd__** .

5. Когда класс естественным образом согласуется с интерфейсами встроенного типа или должен их эмулировать. Например, коллекции могут имитировать интерфейсы последовательностей или отображений, а вызываемые объекты могут быть реализованы для использования с API-интерфейсом, который ожидает функцию. Однако в целом вы не должны реализовывать операции выражений, если они естественно и логически не подходят для ваших объектов – взамен применяйте нормально именованные методы.

# Основы ООП

## Лекция 7. Проектирование

# Полиморфизм означает интерфейсы, а не сигнатуры вызовов

ООП определяют полиморфизм как подразумевающий перегрузку функций на основе сигнатур типов их аргументов - количестве и/или типах переданных аргументов. Ввиду отсутствия объявлений типов в Python такая концепция фактически неприменима; как будет показано, полиморфизм в Python базируется на интерфейсах объектов, а не на типах.

Если вы тоскуете по тем дням, когда программировали на C++, то можете попробовать перегрузить методы посредством списков их аргументов:

```
class C:
    def meth(self, x):
        ...
    def meth(self, x, y, z):
        ...
```

Код запустится, но из-за того, что def просто присваивает объект имени в области видимости класса, останется лишь одно определение функции метода - последнее. Иными словами, ситуация такая же, как если бы вы ввели X = 1 и затем X = 2; имя X получило бы значение 2. Следовательно, может существовать только одно определение имени метода.

В случае реальной необходимости всегда можно написать код выбора на основе типов, используя идеи проверки типов

```
class C:
    def meth(self, *args):
        if len (args) == 1: # Ветвление по количеству аргументов
            ...
        elif type(args[0]) == int: # Ветвление по типам аргументов # (или isinstance ())
```

# Полиморфизм означает интерфейсы, а не сигнатуры вызовов

**Но это не стиль Python**

Вы обязаны писать код в расчете только на интерфейс объекта, но не на специфический тип данных. В итоге код будет полезен для более широкой категории типов и приложений, как сейчас, так и в будущем

```
class C:
    def meth(self, x):
        x.operation() # Предположим, что x делает что-то правильное
```

Также в целом считается лучше применять отличающиеся имена методов для отдельных операций, а не полагаться на сигнатуры вызовов (не имеет значения, на каком языке вы пишете код). Хотя объектная модель Python прямолинейна, основное мастерство в ООП проявляется в способе объединения классов для достижения целей программы.

# ООП и наследование: отношение «является»

С точки зрения программиста наследование вводится уточнениями атрибутов, которые инициируют поиск имен в экземплярах, в их классах и затем в любых суперклассах. С точки зрения проектировщика наследование является способом указания членства в наборе: класс определяет набор свойств, которые могут наследоваться и настраиваться более специфическими наборами (т.е. подклассами).

В целях иллюстрации давайте создадим робота по приготовлению пицц.

```
class Employee:
    def __init__ (self, name, salary=0):
        self.name = name
        self.salary = salary
    def giveRaise(self, percent):
        self.salary = self.salary + (self.salary * percent)
    def work(self):
        print(self.name, "does stuff") # делает что-то
    def __repr__ (self) :
        return "f<Employee: name={self.name}, salary={self.salary}>"
```

Наиболее универсальный класс, Employee, предоставляет общее поведение, такое как повышение зарплат (giveRaise) и вывод (__repr__ ). Есть два вида сотрудников и потому два подкласса Employee - Chef (шеф-повар) и Server (официант). В обоих подклассах переопределяется метод work, чтобы выводить более специфические сообщения. Наконец, наш робот по приготовлению пиццы моделируется даже более специфическим классом - PizzaRobot представляет собой разновидность класса Chef, который является видом Employee. В терминах ООП мы называем такие отношения связями “является”: робот является шеф-поваром, который является сотрудником.

# ООП и наследование: отношение «является»

```
class Chef(Employee):
    def __init__ (self, name):
        Employee.__init__(self, name, 50000)
    def work(self):
        print(self.name, "makes food") # готовит еду

class Server(Employee):
    def __init__ (self, name):
        Employee.__init__ (self, name, 40000)
    def work(self):
        print(self.name, "interfaces with customer") # взаимодействует
                                                     # с клиентом
class PizzaRobot(Chef):
    def __init__(self, name):
        Chef.__init__ (self, name)
    def work(self):
        print(self.name, "makes pizza") # готовит пиццу
```

# ООП и наследование: отношение «является»

```
if __name__ == "__main__" :
    bob = PizzaRobot ('bob' ) # Создать робота по имени bob
    print (bob) # Выполняется унаследованный метод__ repr__
    bob.work () # Выполняется действие, специфичное для типа
    bob.giveRaise (0.20) # Повысить зарплату роботу bob на 20%
    print(bob)
    print()
    for klass in Employee, Chef, Server, PizzaRobot:
        obj = klass(klass.__name__ )
        obj.work()
```

```
<Employee: name=bob, salary=50000>
bob makes pizza
<Employee: name=bob, salary=60000.0>

Employee does stuff
Chef makes food
Server interfaces with customer
PizzaRobot makes pizza
```

Когда мы запускаем код самотестирования, включенный в этот модуль, создается робот для приготовления пиццы по имени bob, который наследует имена от трех классов: PizzaRobot, Chef и Employee. Например, при выводе bob выполняется метод Employee.__ repr__ , а при повышении заработной платы bob вызывается метод Employee.giveRaise, потому что его находит процедура поиска в иерархии наследования.

# ООП и композиция: отношение «имеет»

С точки зрения проектировщика композиция является еще одним способом представления отношений в предметной области. Но вместо членства в наборе композиция имеет дело с компонентами - частями целого. Композиция также отражает взаимосвязи между частями, называемые отношениями «имеет». В некоторых книгах по объектно-ориентированному проектированию на композицию ссылаются как на агрегирование или проводят различие между этими двумя терминами, используя агрегирование для описания более слабой зависимости между контейнером и его содержимым. Здесь под «композицией» понимается просто совокупность внедренных объектов. Составной класс обычно предоставляет собственный интерфейс и реализует его, направляя выполнение действий внедренным объектам.

После реализации классов сотрудников давайте поместим их в пиццерию и предоставим работу. Наша пиццерия является составным объектом: в ней есть духовой шкаф, а также сотрудники вроде официантов и шеф-поваров. Когда клиент входит и размещает заказ, компоненты пиццерии приступают к действиям - официант принимает заказ, шеф-повар готовит пиццу и т.д.

# ООП и композиция: отношение «имеет»

```
from employees import PizzaRobot, Server
class Customer:
    def __init__ (self, name):
        self.name = name
    def order(self, server):
        print(self.name, "orders from", server)
    def pay(self, server):
        print(self.name,"pays for item to", server) # плата за единицу

class Oven:
    def bake(self):             # духовой шкаф выпекает
        print("oven bakes")

class PizzaShop:
    def __init__ (self):
        self.server = Server('Pat') # внедрить другие объекты
        self.chef = PizzaRobot('Bob') # Робот по имени Bob
        self.oven = Oven()
    def order(self, name):
        customer = Customer(name)   # Активизировать другие объекты
        customer.order(self.server) # Заказы клиента, принятые официантом
        self.chef.work()
        self.oven.bake()
        customer.pay(self.server)
```

# ООП и композиция: отношение «имеет»

```
if __name__ == "__main__":
    scene = PizzaShop ()   # Создать составной объект
    scene.order('Homer')  # Эмулировать заказ клиента Homer
    print('...')
    scene.order('Shaggy') # Эмулировать заказ клиента Shaggy
```

```
Homer orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Homer pays for item to <Employee: name=Pat, salary=40000>
...
Shaggy orders from <Employee: name=Pat, salary=40000>
Bob makes pizza
oven bakes
Shaggy pays for item to <Employee: name=Pat, salary=40000>
```

# ООП и композиция: отношение «имеет»

Класс PizzaShop является контейнером и контроллером; его конструктор создает и внедряет экземпляры классов сотрудников, код которых написан в предыдущем разделе, а также экземпляр определенного в текущем разделе класса Oven. Когда в коде самотестирования модуля вызывается метод order класса PizzaShop, внедренным объектам предлагается выполнить их действия по очереди. Обратите внимание, что мы создаем новый объект Customer для каждого заказа и передаем внедренный объект Server методам Customer; клиенты приходят и уходят, но официант представляет собой часть составного объекта пиццерии. Также обратите внимание, что сотрудники по-прежнему участвуют в отношении наследования; композиция и наследование - работающие совместно инструменты.

# ООП и делегирование: промежуточные объекты-оболочки

Наряду с наследованием и композицией программисты, занимающиеся ООП, часто говорят о делегировании, что обычно подразумевает объекты контроллеров с внедренными другими объектами, которым они передают запросы операций. Контроллеры могут заниматься административными действиями, такими как ведение журналов либо проверка достоверности доступа, добавляя дополнительные шаги к компонентам интерфейса или отслеживая активные экземпляры. В известном смысле делегирование является особой формой композиции с единственным внедренным объектом, управляемым классом оболочки (иногда называемого промежуточным классом), который предохраняет большую часть или весь интерфейс внедренного объекта. Понятие промежуточных классов временами применяется и к другим механизмам, таким как вызовы функций; при делегировании нас интересуют промежуточные классы для всех линий поведения объекта, включая вызовы методов и прочие операции.

В Python она часто реализуется с помощью метода **__getattr__**. Поскольку этот метод перегрузки операции перехватывает доступ к несуществующим атрибутам, класс оболочки может использовать **__getattr__** для маршрутизации произвольного доступа к внутреннему объекту. Из-за того, что метод **__getattr__** дает возможность маршрутизировать запросы атрибутов обобщенным образом, класс оболочки предохраняет интерфейс внутреннего объекта и сам может добавлять дополнительные операции.

# Псевдозакрытые атрибуты классов

Методы, определенные внутри класса универсального инструмента, могут быть модифицированы подклассами, если к ним открыт доступ, однако, наряду с поддержкой настройки и прямых вызовов методов они также открыты для случайных замещений. Сокрытие данных является соглашением, а клиенты могут извлекать либо изменять атрибуты в любом классе или экземпляре, на который они имеют ссылку.

На сегодняшний день в Python поддерживается понятие “корректировки” (т.е. расширения) имен для локализации некоторых имен в классах. Скорректированные имена иногда неправильно называют “закрытыми атрибутами”, но в действительности они представляют собой всего лишь способ локализации имени в классе, который его создал - корректировка имен не предотвращает доступ к ним из кода за пределами класса. Данное средство предназначено главным образом для того, чтобы избежать конфликтов между пространствами имен в экземплярах, а не для ограничения доступа к именам в целом; следовательно, скорректированные имена лучше называть “псевдозакрытыми”, чем “закрытыми”.

# Псевдозакрытые атрибуты классов

Псевдозакрытые имена являются продвинутой и совершенно необязательной возможностью; они вряд ли будут крайне полезными, если не создавать универсальные инструменты или более крупные иерархии классов для применения в проектах, где принимает участие много программистов. На самом деле они не всегда используются, даже когда должны - чаще всего программисты на Python записывают внутренние имена с одиночным символом подчеркивания (наподобие _X). Это просто неформальное соглашение, которое уведомляет о том, что имя, как правило, не должно изменяться (для самого Python оно ничего не значит).

# Обзор корректировки имен

Любые имена внутри оператора class, которые начинаются с двух символов подчеркивания, но не заканчиваются ими, автоматически расширяются, чтобы содержать впереди имя включающего класса. Например, имя вроде **__X** внутри класса **Spam** автоматически изменяется на **_Spam__X**: исходное имя снабжается префиксом в виде одиночного символа подчеркивания и именем включающего класса. Из-за того, что модифицированное имя содержит имя включающего класса, оно обычно уникально и не конфликтует с похожими именами, созданными другими классами в иерархии.

Корректировка имен происходит только для имен, появляющихся внутри кода оператора **class**, и затем только для имен, которые начинаются с двух символов подчеркивания. Тем не менее, она работает для каждого имени, предваренного двумя символами подчеркивания - атрибутов классов (в том числе имен методов) и атрибутов экземпляров, присвоенных в **self**. Скажем, в классе Spam метод по имени **__meth** после корректировки превращается в **_Spam__ meth**, а ссылка на атрибут экземпляра **self.__ X** видоизменяется на **self.Spam__ X**.

# Псевдозакрытые атрибуты классов

Псевдозакрытые имена являются продвинутой и совершенно необязательной возможностью; они вряд ли будут крайне полезными, если не создавать универсальные инструменты или более крупные иерархии классов для применения в проектах, где принимает участие много программистов. На самом деле они не всегда используются, даже когда должны - чаще всего программисты на Python записывают внутренние имена с одиночным символом подчеркивания (наподобие _X). Это просто неформальное соглашение, которое уведомляет о том, что имя, как правило, не должно изменяться (для самого Python оно ничего не значит).

Несмотря на корректировку, до тех пор, пока в классе везде, где есть ссылка на имя, применяется версия с двумя символами подчеркивания, все ссылки по-прежнему будут работать. Однако поскольку добавлять атрибуты к экземпляру могут сразу несколько классов, корректировка имен помогает избежать конфликтов - но чтобы увидеть, почему, нам необходимо перейти к конкретному примеру.

# Псевдозакрытые атрибуты классов

Одна из основных проблем, которые призваны смягчить псевдозакрытые атрибуты, связана со способом хранения атрибутов экземпляров. В Python все атрибуты экземпляров оказываются в единственном объекте экземпляра в нижней части дерева классов и разделяются всеми функциями методов уровня класса, которым передается экземпляр. Такая модель отличается от модели, принятой в C++, где каждый класс получает собственное пространство для определяемых им данных-членов.

Внутри метода класса в Python всякий раз, когда метод присваивает значение какому-то атрибуту **self** (например, ***self.атрибут = значение***), он изменяет или создает атрибут в экземпляре (вспомните, что поиск в иерархии наследования происходит только при ссылке, не в случае присваивания). Поскольку это справедливо, даже если множество классов в иерархии присваивают значение тому же самому атрибуту, возможны конфликты.

# Псевдозакрытые атрибуты классов

Скажем, пусть при реализации класса программист предполагает, что класс владеет атрибутом по имени X в экземпляре. В методах данного класса указанное имя устанавливается и позже извлекается.

Далее предположим, что еще один программист, работающий изолированно, делает то же самое допущение в другом классе.

Сами по себе оба класса функционируют. Проблема возникнет, если два класса когда-нибудь смешаются в одном дереве классов:

```
class C1:
    def meth1(self): self.X = 88  # Я предполагаю, что атрибут X - мой
    def meth2(self): print(self.X)

class C2:
    def metha(self): self.X =99      # Я тоже
    def methb(self): print(self.X)

class C3(C1, C2):
    pass

I = C3() # В I есть только один атрибут X!
```

Теперь значение, которое каждый класс получает для выражения **self.X**, будет зависеть о того, какой класс выполнял присваивание последним. Из-за того, что все присваивания **self.X** относятся к тому же самому одиночному экземпляру, существует только один атрибут X - I. X - вне зависимости от того, сколько классов применяют такое имя атрибута.

# Псевдозакрытые атрибуты классов

Чтобы гарантировать принадлежность атрибута классу, который его использует, необходимо снабдить имя префиксом в виде двух символов подчеркивания везде, где оно применяется в классе

```
class C1:
    def meth1(self): self.__X = 88  # Теперь я имею свой атрибут X
    def meth2(self): print(self.__X) # Становится _C1__X в I

class C2:
    def metha(self): self.__X = 99    # Я тоже
    def methb(self) : print(self.__X) # Становится _C2__X в I

class C3(C1, C2): pass

I = C3() # В I есть два имени X

I.meth1()
I.metha()
print(I.__dict__ )
I.meth2()
I.methb()
```

```
{'_C1__X': 88, '_C2__X': 99}
88
99
```

Показанный трюк позволяет избежать потенциальных конфликтов имен в экземпляре, но учтите, что он не означает подлинную защиту. Зная имя включающего класса, вы по-прежнему можете получать доступ к любому из двух атрибутов везде, где имеется ссылка на экземпляр, с использованием полностью развернутого имени.

# Методы являются объектами: связанные или несвязанные методы

**Объекты несвязанных методов (класса): без self**

Доступ к функциональному атрибуту класса путем уточнения с помощью класса возвращает объект несвязанного метода. Чтобы вызвать метод, потребуется в первом аргументе явно указать объект экземпляра. В Python 3.X несвязанный метод представляет собой то же самое, что и простая функция, и может вызываться через имя класса; ***в Python 2.X он является отдельным типом и не может быть вызван без указания экземпляра***.

**Объекты связанных методов (экземпляра): пары self + функция**

Доступ к функциональному атрибуту класса путем уточнения с помощью экземпляра возвращает объект связанного метода. В объект связанного метода Python автоматически помещает экземпляр и функцию, так что для вызова метода передавать экземпляр не нужно.

При вызове объекта связанного метода Python предоставляет экземпляр автоматически - экземпляр, применяемый для создания объекта связанного метода. Таким образом, объекты связанных методов обычно взаимозаменяемы с объектами простых функций, что делает их особенно удобными для интерфейсов, изначально ориентированных на функции.

# Методы являются объектами: связанные или несвязанные методы

Предположим, что мы определили следующий класс:

```
class Spam:
    def doit(self, message):
        print(message)
```

Теперь в нормальной операции мы создаем экземпляр и вызываем его метод, выводящий переданный аргумент:

```
object1 = Spam()
object1.doit('hello world')
```

Тем не менее, в действительности прямо перед круглыми скобками вызова метода генерируется объект связанного метода. Фактически мы можем извлечь связанный метод, не вызывая его. Подобно всем выражениям результатом выражения ***объект.имя*** будет объект. Ниже такое выражение возвращает объект связанного метода с упакованным экземпляром (objectl) и функцией метода (Spam.doit). Мы можем присвоить эту пару связанного метода другому имени и затем вызвать его.

```
object1 = Spam()
x = object1.doit # Объект связанного метода: экземпляр + функция
x('hello world') # Тот же эффект, что и objectl.doit )
```

# Методы являются объектами: связанные или несвязанные методы

С другой стороны, если для того, чтобы добраться до doit, мы указываем класс, тогда получаем обратно объект несвязанного метода, который является просто ссылкой на объект функции. Для вызова метода такого типа мы обязаны передать экземпляр как крайний слева аргумент - иначе он отсутствует в выражении, а метод его ожидает

```
object1 = Spam()
t = Spam.doit          # Объект несвязанного метода (функция в Python 3.X)
t(object1, 'howdy')    # Передача экземпляра (если метод его ожидает в Python 3.X)
```

Более того, то же самое правило применяется внутри метода класса, если мы ссылаемся на атрибуты self, которые относятся к функциям в классе. Выражение ***self.метод*** представляет собой объект связанного метода, потому что self - объект экземпляра

```
class Eggs:
    def m1(self, n):
        print(n)
    def m2(self):
        x = self.m1    # Еще один объект связанного метода
        x(42)          # Выглядит подобно простой функции

Eggs().m2()
```

# Методы являются объектами: связанные или несвязанные методы

В Python 3.X из языка было исключено понятие несвязанных методов. То, что здесь мы описывали как несвязанный метод, в Python 3.X трактуется как простая функция. В большинстве случаев вашему коду это безразлично; оба способа предусматривают передачу экземпляра в первом аргументе при вызове метода через экземпляр.

Кроме того, в Python 3.X допускается вызывать метод без экземпляра при условии, что метод его не ожидает, и метод вызывается только через класс и никогда через экземпляр. То есть Python 3.X будет передавать экземпляр методам только для вызовов через экземпляр. При вызове через класс передавать экземпляр вручную понадобится только в случае, если метод его ожидает:

```
class Selfless:
    def __init__(self, data) :
        self.data = data
    def selfless(arg1, arg2): # Простая функция в Python 3.X
        return arg1 + arg2
    def normal(self, arg1, arg2): # При вызове ожидается экземпляр
        return self .data + arg1 + arg2

X = Selfless(2)  # Экземпляр передается self автоматически: 2+ (3+4)
print(X.normal(3, 4))
```

```
9
```

# Методы являются объектами: связанные или несвязанные методы

```
X = Selfless(2)  # Экземпляр передается self автоматически: 2+ (3+4)
print(Selfless.normal(X, 3, 4)) # Метод ожидает self: передать вручную
print(Selfless.selfless(3, 4))# Без передачи экземпляра : в Python 3.X
                              # работает, в Python 2.X - нет!
```

Последний тест в Python 2.X потерпит неудачу, потому что несвязанные методы по умолчанию требуют передачи экземпляра; в Python 3.X он работает из-за того, что такие методы трактуются как простые функции, не нуждающиеся в экземпляре. Несмотря на то что в Python 3.X перестают отлавливаться некоторые потенциальные ошибки (что, если программист забыл передать экземпляр ?), появляется возможность использовать методы класса как простые функции до тех пор, пока им не передается аргумент экземпляра self и они не рассчитывают на него.

# Методы являются объектами: связанные или несвязанные методы

Тем не менее, следующие два вызова по-прежнему терпят неудачу в Python 3.X и 2.X. Первый (вызов через экземпляр) автоматически передает экземпляр методу, который его не ожидает, в то время как второй (вызов через класс) не передает экземпляр методу, который его ожидает

```
X.selfless(3,4)
```

```
X.selfless(3,4)
TypeError: selfless() takes 2 positional arguments but 3 were given
```

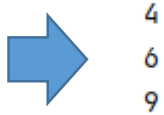
```
Selfless.normal(3, 4)
```

```
Selfless.normal(3, 4)
TypeError: normal() missing 1 required positional argument: 'arg2'
```

# Методы являются объектами: связанные или несвязанные методы

Связанные методы могут обрабатываться как обобщенные объекты подобно простым функциям - их можно произвольно передавать в рамках программы. Кроме того, поскольку связанные методы объединяют функцию и экземпляр в единый пакет, они могут трактоваться как любой другой вызываемый объект и не требовать специального синтаксиса при вызове.

```
class Number:
    def __init__(self, base):
        self.base = base
    def double (self):
        return self.base * 2
    def triple(self):
        return self.base * 3

x = Number(2)
y = Number(3)
z = Number(4)
print(x.double())
acts = [x.double, y.double, y.triple, z.double] # Список связанных методов
for act in acts: # Вызовы откладываются
    print(act())  # Вызов, как если бы это были функции
```

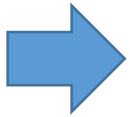


```
4
4
6
9
8
```

# Методы являются объектами: связанные или несвязанные методы

На самом деле связанные методы являются всего лишь одним из нескольких типов вызываемых объектов в Python. Как демонстрируется ниже, простые функции, определенные посредством **def** или **lambda**, экземпляры, которые наследуют **__ call__** , и связанные методы экземпляров могут обрабатываться и вызываться одинаково:

```
def square(arg):
    return arg ** 2 # Простые функции (def или lambda)

class Sum:
    def __init__(self, val) : # Вызываемые экземпляры
        self.val = val

    def __call__(self, arg):
        return self.val + arg

class Product:
    def __init__(self, val):  # Связанные методы
        self.val = val

    def method(self, arg):
        return self.val * arg

sobject = Sum(2)
pobject = Product(3)
actions = [square, sobject, pobject.method]
for act in actions: # Все три вызываются одинаково
    print(act(5)) # Вызов любого вызываемого объекта с одним аргументом
```



```
25
7
15
```

# Классы являются объектами: обобщенные фабрики объектов

Временами проектные решения на основе классов требуют создания объектов в ответ на условия, которые невозможно предсказать на стадии написания кода программы, Паттерн проектирования “Фабрика” делает возможным такой отложенный подход. Во многом благодаря гибкости Python фабрики могут принимать многочисленные формы, ряд которых вообще не кажутся особыми.

Из-за того, что классы также являются объектами “первого класса”, их легко передавать в рамках программы, сохранять в структурах данных и т.д. Вы также можете передавать классы функциями, которые генерируют произвольные виды объектов; в кругах объектно-ориентированного проектирования такие функции иногда называются фабриками. Фабрики могут оказаться крупным делом в строго типизированных языках вроде C++, но довольно просты в реализации на Python.

В коде мы определяем функцию генератора объектов по имени factory. Она ожидает передачи объекта класса (подойдет любой класс) наряду с одним и более аргументов для конструктора класса. Для вызова функции и возвращения экземпляра применяется специальный синтаксис с переменным количеством аргументов.

В оставшемся коде примера просто определяются два класса и генерируются экземпляры обоих классов за счет их передачи функции factory. И это единственная фабричная функция, которую вам когда-либо придется писать на Python; она работает для любого класса и любых аргументов конструкторов.

# Классы являются объектами: обобщенные фабрики объектов

```
def factory(aClass, *pargs, **kargs) : # Кортеж или словарь с переменным
                                       # количеством аргументов
    return aClass(*pargs, **kargs) # Вызывает aClass (или apply в Python 2.X)

class Spam:
    def doit(self, message):
        print(message)

class Person:
    def __init__(self, name, job=None):
        self.name = name
        self.job = job

objectl = factory(Spam) # Создать объект Spam
object2 = factory (Person, "Arthur", "King") # Создать объект Person
object3 = factory (Person, name= "Brian") # Тоже самое, с ключевым
                                          # аргументом и стандартным значением
print(objectl.doit(99))
print(object2.name, object2.job)
print(object3.name, object3.job)
```

```
99
Arthur King
Brian None
```

Трудно показать приложения паттерна проектирования “Фабрика” без приведения кода большего объема, чем для этого есть место. Тем не менее, в целом такая фабрика способна сделать возможной изоляцию кода от деталей динамически конфигурируемого создания объектов.

# Множественное наследование: "подмешиваемые" классы

Многие проектные решения, основанные на классах, требуют объединения разрозненных наборов методов. Как уже было показано, в строке заголовка оператора **class** в круглых скобках можно указывать более одного суперкласса. Поступая так, мы задействуем множественное наследование — класс и его экземпляры наследуют имена из всех перечисленных суперклассов. Для нахождения атрибута процедура поиска в иерархии наследования Python обходит все суперклассы, указанные в заголовке оператора class, слева направо до тех пор, пока не обнаружит соответствие. Формально из-за того, что любой из суперклассов может иметь собственные суперклассы, такой поиск может быть чуть сложнее для более крупных деревьев классов.

• В классических классах (стандарт вплоть до Python 3.0) поиск атрибутов во всех случаях продолжается сначала в глубину на всем пути к вершине дерева наследования и затем слева направо. Такой порядок обычно называется DFLR (DepthFirst, Left-to-Right - сначала в глубину, слева направо).

• В классах нового стиля (необязательные в Python 2.X и стандарт в Python 3.X) поиск атрибутов обычно происходит, как было ранее, но в ромбовидных схемах продолжается по уровням дерева до перехода вверх, т.е. больше в манере сначала в ширину. Такой порядок обычно называется MRO нового стиля (Method Resolution Order - порядок распознавания методов), хотя он применяется для всех атрибутов, а не только для методов.

# Множественное наследование: "подмешиваемые" классы

Многие проектные решения, основанные на классах, требуют объединения разрозненных наборов методов. Как уже было показано, в строке заголовка оператора **class** в круглых скобках можно указывать более одного суперкласса. Поступая так, мы задействуем множественное наследование — класс и его экземпляры наследуют имена из всех перечисленных суперклассов. Для нахождения атрибута процедура поиска в иерархии наследования Python обходит все суперклассы, указанные в заголовке оператора class, слева направо до тех пор, пока не обнаружит соответствие. Формально из-за того, что любой из суперклассов может иметь собственные суперклассы, такой поиск может быть чуть сложнее для более крупных деревьев классов.

• В классических классах (стандарт вплоть до Python 3.0) поиск атрибутов во всех случаях продолжается сначала в глубину на всем пути к вершине дерева наследования и затем слева направо. Такой порядок обычно называется DFLR (DepthFirst, Left-to-Right - сначала в глубину, слева направо).

• В классах нового стиля (необязательные в Python 2.X и стандарт в Python 3.X) поиск атрибутов обычно происходит, как было ранее, но в ромбовидных схемах продолжается по уровням дерева до перехода вверх, т.е. больше в манере сначала в ширину. Такой порядок обычно называется MRO нового стиля (Method Resolution Order - порядок распознавания методов), хотя он применяется для всех атрибутов, а не только для методов.

# Множественное наследование: "подмешиваемые" классы

Несмотря на то что без кода следующей главы ромбовидные схемы понять трудно (и создавать их самостоятельно доведется редко), они образуются, когда множество классов в дереве используют общий суперкласс; порядок поиска нового стиля спроектирован так, чтобы посещать разделяемый суперкласс только один раз и после всех его подклассов. Тем не менее, в любой из двух моделей, когда класс имеет множество суперклассов, поиск в них производится слева направо согласно порядку указания суперклассов в строках заголовка оператора class.

В целом множественное наследование хорошо подходит для моделирования объектов, которые принадлежат более чем одному набору. Скажем, человек может быть инженером, писателем, музыкантом и т.д., а потому наследовать свойства от всех наборов подобного рода. Благодаря множественному наследованию объекты получают объединение всех линий поведения из всех своих суперклассов.

Хотя множественное наследование — полезный паттерн проектирования, его главный недостаток в том, что оно может привести к конфликту, когда то же самое имя метода (или другого атрибута) определено сразу в нескольких суперклассах.

# Множественное наследование: "подмешиваемые" классы

Возникший конфликт разрешается либо автоматически за счет порядка поиска в иерархии наследования, либо вручную в коде.

• **Стандартный способ.**

По умолчанию процедура поиска в иерархии наследования выбирает первое найденное вхождение атрибута, когда на атрибут производится ссылка обычным образом, например, self.method (). В таком режиме Python выбирает самый нижний и крайний слева атрибут в классических классах и при неромбовидных схемах во всех классах; в классах нового стиля при ромбовидных схемах может быть выбран вариант справа или выше.

• **Явный способ**.

В некоторых моделях на основе классов иногда необходимо выбирать атрибут явно, ссылаясь на него через имя класса, скажем, superclass.method(self). Ваш код разрешает конфликт и переопределяет стандартный способ поиска - чтобы выбрать вариант справа или выше принятого по умолчанию при поиске в иерархии наследования.

Проблема возникает, только когда то же самое имя появляется во множестве суперклассов, и вы не хотите использовать первое унаследованное.

# Множественное наследование: "подмешиваемые" классы

Множественное наследование чаще всего используется для “подмешивания” универсальных методов из суперклассов. Такие суперклассы обычно называются подмешиваемыми классами - они предоставляют методы, которые добавляются к прикладным классам через наследование. В некотором смысле подмешиваемые классы похожи на модули: они предлагают пакеты методов для применения в своих клиентских подклассах. Тем не менее, в отличие от простых функций методы в подмешиваемых классах также могут принимать участие в иерархиях наследования и иметь доступ к экземпляру self для использования информации о состоянии и других методов в своих деревьях.

# MRO, слоты, графические пользовательские интерфейсы

***Общие идеи: графические пользовательские интерфейсы, внутренние имена***

Группирование имен с двумя символами подчеркивания, как делалось ранее, может способствовать сокращению размера древовидного отображения, хотя некоторые имена вроде **__ init__** определяются пользователем и заслуживают специального обращения. Схематическое изображение дерева в графическом пользовательском интерфейсе тоже может считаться естественным следующим шагом - комплект инструментов tkinter, задействованный в примерах из предыдущего раздела, поставляется вместе с Python и предлагает базовую, но легкую поддержку, а есть альтернативы с более широкими возможностями, хотя они сложнее.

# Основы ООП

## Лекция 8. Расширенные возможности классов

# Модель классов "нового стиля"

• В Python 3.X все классы являются тем, что прежде называлось “новым стилем”, унаследованы они явно от object или нет. Указывать суперкласс object не обязательно, и он подразумевается.

• В Python 2.X классы должны явно наследоваться от object (или другого встроенного типа), чтобы считаться “новым стилем” и получить все линии поведения нового стиля. Без такого наследования классы будут “классическими”.

Поскольку в Python 3.X все классы автоматически становятся классами нового стиля, в этой линейке возможности классов нового стиля считаются просто нормальными функциональными средствами классов.

Классы нового стиля обладают глубокими отличиями, которые оказывают широкое влияние на программы, особенно когда код задействует добавленные в них расширенные возможности. На самом деле, по крайней мере, с точки зрения их поддержки ООП, эти изменения на ряде уровней превращают Python в совершенно особый язык. Они обязательны в линейке Python 3.X, необязательны в линейке Python 2.X, но только если игнорируются каждым программистом, и в данной области заимствуют намного большее из других языков (и часто обладают сравнимой сложностью).

Классы нового стиля частично являются результатом попытки объединить понятие класса с понятием типа во времена существования версии Python 2.2, хотя для многих они оставались незамеченными, пока не стали необходимым знанием в Python 3.X.

Вам нужно самостоятельно оценить успех такого объединения, но как мы выясним, в модели все еще присутствуют различия - теперь между классом и метаклассом — и один из побочных эффектов заключается в том, что нормальные классы оказываются более мощными, но также и значительно более сложными.

# Изменения в классах "нового стиля"

**Извлечение атрибутов для встроенных операций: экземпляр пропускается**

Обобщенные методы перехвата извлечения атрибутов__ getattr__ и __getattribute__ по-прежнему выполняются для атрибутов, к которым производится доступ по явному имени, но больше не запускаются для атрибутов, неявно извлекаемых встроенными операциями. Они не вызываются для имен методов перегрузки операций__ X__ только во встроенных контекстах - поиск таких имен начинается в классах, не в экземплярах. Это нарушает работу или усложняет объекты, которые служат в качестве промежуточных для интерфейса другого объекта, если внутренние объекты реализуют перегрузку операций. Методы подобного рода должны быть переопределены из-за отличающегося координирования встроенных операций в классах нового стиля.

***Классы и типы объединены: проверка типа***

Классы теперь являются типами, а типы - классами. На самом деле по существу они представляют собой синонимы, хотя метаклассы, которые теперь относятся к категории типов, все еще кое в чем отличаются от нормальных классов. Встроенная функция **type(I)** возвращает класс, из которого создан экземпляр, а не обобщенный тип экземпляра, и обычно дает такой же разультат, как **I.__class__** . Более того, классы являются экземплярами класса **type**, и можно реализовывать подклассы **type** для настройки создания классов с помощью метаклассов, записываемых посредством операторов **class**. Это может повлиять на код, который проверяет типы или по-другому полагается на предыдущую модель классов.

# Изменения в классах "нового стиля"

**Автоматический корневой класс object: стандартные методы**

Все классы нового стиля (отсюда и типы) наследуются от object, который содержит небольшой набор стандартных методов перегрузки операций (скажем, __ repr__ ). В Python 3.X класс object автоматически добавляется выше определяемых пользователем корневых (т.е. самых верхних) классов в дереве и не нуждается в явном указании в качестве суперкласса. Это может повлиять на код, который допускает отсутствие стандартных методов и корневых классов.

***Порядок поиска в иерархии наследования: MRO и ромбы***

Ромбовидные схемы множественного наследования имеют слегка отличающийся порядок поиска - грубо говоря, в ромбах поиск производится раньше и более в стиле сначала в ширину, чем сначала в глубину. Такой порядок поиска атрибутов, известный как **MRO**, можно отследить с помощью нового атрибута **__mro__** , доступного в классах нового стиля. Новый порядок поиска в основном применяется только к деревьям классов с ромбами, хотя сам подразумеваемый корень **object** новой модели образует ромб во всех деревьях множественного наследования. Код, который полагается на предыдущий порядок, не будет работать таким же образом..

# Изменения в классах "нового стиля"

**Алгоритм наследования:**

Алгоритм, используемый при наследовании в классах нового стиля, существенно сложнее, чем модель “сначала в глубину” классических классов, и включает особые случаи для дескрипторов, метаклассов и встроенных операций.

***Новые расширенные инструменты: влияние на код***

Классы нового стиля обладают новыми механизмами, в том числе слотами, свойствами, дескрипторами, встроенной функцией **super** и методом **__getattribute__** . Большинство из них ориентированы на решение весьма специфических задач построения инструментов. Тем не менее, их применение также способно повлиять или нарушить работу существующего кода; например, слоты иногда вообще препятствуют созданию словарей пространств имен экземпляров, а обобщенные обработчики атрибутов могут требовать написания другого кода.

# Почему изменился поиск ?

Первое логическое обоснование подкрепляется частотой применения встроенных операций. Скажем, если каждая операция + требует выполнения дополнительных шагов для экземпляра, то она может уменьшить быстродействие программ - особенно с учетом множества расширений на уровне атрибутов в модели нового стиля.

Второе логическое отражает сложную проблему, привнесенную моделью **метаклассов**. Так как классы теперь являются экземплярами метаклассов и поскольку в метаклассах могут определяться методы встроенных операций для обработки классов,генерируемых метаклассами, то вызов метода, запускаемый для класса, обязан пропустить сам класс и произвести поиск на один уровень выше, чтобы подобрать метод, который обработает этот класс, а не выбрать собственную версию метода, принадлежащую классу. Собственная версия привела бы к вызову несвязанного метода, потому что собственный метод класса обрабатывает экземпляры более низкого уровня.

# Почему изменился поиск ?

В результате, поскольку классы сами по себе являются и типами, и экземплярами, при поиске методов встроенных операций все экземпляры пропускаются. Такой прием применяется к нормальным экземплярам предположительно ради единообразия и согласованности, но для невстроенных имен, а также прямых и явных обращений к встроенным именам по-прежнему осуществляется проверка экземпляра. Несмотря на то что вероятно это последствие, обусловленное введением модели классов нового стиля, для ряда программистов оно может выглядеть как решение, принятое в пользу менее естественного и более неясного принципа, нежели широко применяемый ранее. Его роль в качестве пути оптимизации кажется в большей степени оправданной, но также не без оговорок.

В частности, изменение поиска оказывает потенциально обширное влияние на классы, основанные на делегировании, которые часто называют ***классами-посредниками***, когда внедренные объекты реализуют перегрузку операций. В классах нового стиля такой класс-посредник обычно должен переопределять любые имена подобного рода для перехвата и делегирования, либо вручную, либо посредством инструментов. Конечным результатом становится либо значительное усложнение, либо полный отказ от целой категории программ.

# Требования к коду классов-посредников

При делегировании встроенные операции вроде выражений больше не работают таким же образом, как эквивалентные им традиционные прямые вызовы. И наоборот, прямые обращения к именам встроенных методов по-прежнему работают, но эквивалентные выражения - нет, потому что вызовы через тип терпят неудачу для имен не на уровне класса и выше. Другими словами, это различие возникает только во встроенных операциях, явные извлечения выполняются корректно.

При написании кода посредника объекта, к интерфейсу которого частично могут обращаться встроенные операции, классы нового стиля требуют метода ***__getattr__*** для нормальных имен, а также переопределений методов для всех имен, к которым имеют доступ встроенные операции - кодируются они вручную, получаются из суперклассов или генерируются инструментами. Когда переопределения включаются подобным образом, вызовы через экземпляры и через типы эквивалентны встроенным операциям, хотя переопределенные имена больше не направляются обобщенному обработчику неопределенных имен ***__getattr__*** даже для явных обращений к именам.

# Изменения модели типов

**Классы являются типами**

Объект ***type*** генерирует классы как свои экземпляры, а классы генерируют экземпляры самих себя. Оба считаются типами, потому что они генерируют экземпляры. На самом деле не существует реальной разницы между встроенными типами, такими как списки и строки, и определяемыми пользователем типами, реализованными в виде классов. Мы можем создавать подклассы встроенных типов, подкласс встроенного типа наподобие ***list*** квалифицируется как класс нового стиля и становится новым типом, определяемым пользователем.

**Типы являются классами**

Новые типы, генерирующие классы, могут быть реализованы на Python как метаклассы, - подклассы определяемого пользователем типа, которые записываются посредством нормальных операторов class и управляют созданием классов, являющихся их экземплярами. Метаклассы являются одновременно классами и типами, хотя они отличаются вполне достаточно, чтобы поддерживать разумный аргумент в пользу того, что предшествующее разветвление “тип/класс” превратилось в “метакласс/класс”, возможно ценой добавочной сложности в нормальных классах.

# Изменения модели типов

Еще одно последствие изменения типов в модели классов нового стиля состоит в том, что поскольку все классы являются производными (унаследованными) от класса ***object***, либо неявно, либо явно, и по причине того, что теперь все типы - это классы, каждый объект оказывается производным от встроенного класса ***object***, будь то напрямую или через суперкласс.

```
class C:
    pass

X = C()
print(type(X), type(C))
```

```
<class '__main__.C'> <class 'type'>
```

Как и ранее, типом экземпляра класса будет класс, из которого он был создан, а типом класса - класс **type**, потому что классы и типы объединены. Тем не менее, также верно и то, что экземпляр и класс унаследованы от встроенного класса и типа **object**, т.е. неявного или явного суперкласса каждого класса.

```
print(isinstance(X, object))
print(isinstance(C, object))
```

```
True
True
```

# Изменение ромбовидного наследования

Финальное изменение в модели классов нового стиля также является одним из самых заметных: слегка отличающийся порядок поиска для так называемых деревьев множественного наследования с ромбовидными схемами. Наличие в таких деревьях более чем одного суперкласса приводит к тому же самому расположенному выше суперклассу (их название происходит от ромбовидной формы дерева, если вы его нарисуете — квадрат, опирающийся на один из его углов).

**Для классических классов (стандарт в Python 2.X): DFLR**
Путь поиска при наследовании проходит строго сначала в глубину и затем слева направо - Python поднимается все время к верху, придерживаясь левой стороны дерева, и только потом останавливается и начинает просмотр дальше вправо. Такой порядок поиска известен как DFLR (Depth-First, Left-to-Right - сначала в глубину, слева направо).

**Для классов нового стиля (необязательные в Python 2.X и автоматические в Python 3.X): MRO**
Путь поиска при наследовании в ромбовидных схемах выполняется больше в манере сначала в ширину - Python сначала ищет в любых суперклассах справа от только что просмотренного и только потом поднимается к общему суперклассу вверху. Другими словами, поиск проходит по уровням, прежде чем двигаться вверх. Такой порядок поиска называется MRO нового стиля (Method Resolution Order - порядок распознавания методов), а часто ради краткости — просто MRO, когда используется для противопоставления с порядком DFLR.

# Изменение ромбовидного наследования

Алгоритм MRO нового стиля немного сложнее,, но именно столько нужно знать многим программистам. Тем не менее, важно отметить, что он обладает не только важными преимуществами для кода с классами нового стиля, но и потенциалом нарушения работы существующего кода с классическими классами. Например, алгоритм MRO нового стиля дает возможность нижним суперклассам перегружать атрибуты верхних суперклассов, не обращая внимания на разновидности деревьев множественного наследования, в которых они смешаны. Кроме того правило поиска нового стиля позволяет избежать посещения того же самого суперкласса более одного раза, когда он доступен из множества подклассов. Вероятно алгоритм MRO лучше DFLR, но он применяется к небольшому подмножеству пользовательского кода на Python; однако, как мы увидим, модель классов нового стиля сама по себе делает ромбы гораздо более распространенными, а MRO более важным.

# Алгоритм MRO

1. Построение списка всех классов, от которых унаследован экземпляр, с использованием правила поиска DFLR классических классов и многократное включение класса, если он посещается более одного раза.
2. Просмотр построенного списка на предмет дубликатов с удалением всех вхождений кроме последнего

Результирующий список MRO для заданного класса включает этот класс, его суперклассы и все более высокие суперклассы вплоть до корневого класса **object**, расположенного на верхушке дерева. Он упорядочен так, что каждый класс находится перед
своими родителями, а множество родителей сохраняют порядок, в котором они следуют внутри кортежа суперклассов**__ bases__** .
Тем не менее, важно отметить, что поскольку общие родители в ромбах появляются
только в позициях, в которых они посещались последний раз, поиск в нижних классах
выполняется первым, когда позже список MRO применяется при наследовании атрибутов. Кроме того, каждый класс включается и потому посещается только один раз
независимо от того, сколько классов к нему ведет.

# Алгоритм MRO

Атрибут **mro** представляет собой кортеж, который дает линейный порядок поиска, используемый Python при просмотре атрибутов в суперклассах. В действительности этот атрибут является порядком наследования в классах нового стиля и часто оказывается единственной деталью, относящей к MRO, знание которой вполне достаточно для многих пользователей Python.

```
class A:
    pass
class B(A):
    pass # Ромбы: для классов нового стиля порядок отличается
class C(A):
    pass # Поиск сначала в ширину на нижних уровнях
class D(B, C):
    pass

print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

# Алгоритм MRO

Однако для неромбовидных схем поиск ведет себя так, как было всегда (хотя и с дополнительным корнем object) - вверх и затем вправо (порядок поиска DFLR, применяемый классическими классами в Python 2.X):

```
class A:
    pass
class B(A):
    pass # Ромбы: для классов нового стиля порядок отличается
class C:
    pass # Поиск сначала в ширину на нижних уровнях
class D(B, C):
    pass

print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>]
```

Формально подразумеваемый суперкласс **object** всегда образует ромб в дереве множественного наследования, даже когда ваши классы этого не делают - поиск в ваших классах выполняется, как и ранее, но **MRO** нового стиля гарантирует, что **object** посещается последним, так что ваши классы могут переопределять его стандартные методы.

# Расширения: слоты

Присваивая последовательность строковых имен атрибутов специальному атрибуту **__slots__** класса, мы можем позволить классу нового стиля ограничивать набор допустимых атрибутов, которые будут иметь экземпляры этого класса, а также оптимизировать потребление памяти и возможно скорость работы программы. Однако, как мы увидим, слоты должны использоваться только в приложениях, в которых добавочная сложность очевидно оправдана. Они усложнят ваш код, способны усложнить или нарушить работу кода, с которым вы можете иметь дело, и требовать эффективного универсального ввода в действие.

Для применения слотов нужно присвоить последовательность строковых имен специальной переменной и атрибуту **__ slots__** на верхнем уровне оператора **class**: только именам в списке **__ slots__** можно присваивать значения как атрибутам экземпляров. Тем не менее, подобно всем именам в Python именам атрибутов экземпляров по-прежнему должны присваиваться значения, прежде чем на них можно будет ссылаться, несмотря на то, что они перечислены в **__ slots__**

```
class limiter (object) :
    __slots__ = ['age', 'name', 'job']
x = limiter()
print(x.age)
```

```
AttributeError: age
```

```
x.age = 40
print(x.age)
```

```
40
```

# Расширения: слоты

Средство слотов задумывалось как способ отлавливания опечаток вроде показанного выше (обнаруживается присваивание недопустимым именам атрибутов, которые отсутствуют в **__ slots__** ), а также как механизм оптимизации. Размещение словаря пространств имен для каждого объекта экземпляра может оказаться дорогостоящим в плане памяти, если создается много экземпляров, а обязательных атрибутов лишь несколько. Чтобы сберечь пространство, Python не размещает словарь в каждом экземпляре, а резервирует для каждого экземпляра лишь пространство под хранение значения каждого атрибута слота и унаследованных атрибутов из общего класса для управления доступом к слотам. Такой прием способен дополнительно ускорить выполнение, хотя данное преимущество менее очевидно и может варьироваться в зависимости от программы, платформы и версии Python. Слоты также являются своего рода крупным расхождением с основной динамической природой Python, которая требует, чтобы любое имя можно было создавать присваиванием. Фактически слоты имитируют язык C++ для эффективности за счет гибкости и даже обладают потенциалом нарушать работу ряда программ.

```
x.ape = 1000
```

```
AttributeError: 'limiter' object has no attribute 'ape'
```

# Расширения: слоты

На самом деле одни экземпляры со слотами могут вообще не содержать словаря пространств имен атрибутов **__ dict__** , а другие будут иметь атрибуты данных, которые такой словарь не включает. Для ясности: это крупная несовместимость с традиционной моделью классов — такая, которая может усложнить любой код, получающий доступ к атрибутам обобщенным образом, и даже привести к полному отказу ряда программ.
Если есть вероятность применения слотов, тогда программам, которые составляют списки или обращаются к атрибутам экземпляра по строковым именам, может потребоваться использовать более нейтральные к хранилищу интерфейсы, нежели **__ dict__** . Поскольку данные экземпляра могут включать имена уровня классов, такие как слоты - либо в дополнение, либо вместо хранилища словаря пространств имен, - для полноты может возникнуть необходимость опрашивать оба источника атрибутов.

```
class C:
    __slots__ = ['a', 'b']
X = C()
X.a = 1
print(X.a)
print(X.__dict__)
```

```
AttributeError: 'C' object has no attribute '__dict__'
1
```

Однако мы все еще можем извлекать и устанавливать атрибуты, основанные на слотах, по строковому имени с применением нейтральных к хранилищу инструментов, таких как **getattr**, **setattr** (просматривают за пределами **__dict__** экземпляра и потому охватывают имена уровня класса вроде слотов) и **dir** (собирает все унаследованные имена повсюду в дереве)

# Расширения: слоты

Имейте в виду, что без словаря пространств имен атрибутов выполнять присваивание в экземплярах новым именам, отсутствующим в списке **__slots__** , невозможно:

```
class D:
    __slots__ = ['a', 'b']
    def __init__(self):
        self.d =4
X = D()
```

```
AttributeError: 'D' object has no attribute 'd'
```

Тем не менее, мы по-прежнему может разместить добавочные атрибуты, явно включая __ dict__ в **__slots__** , чтобы создать также и словарь пространств имен атрибутов:

```
class D:
    __slots__= [ 'a', 'b', '__dict__']
    c = 3
    def __init__(self):
        self.d = 4

X = D()
print(X.c, X.d)
```

```
3 4
```

# Расширения: слоты

Из-за того, что имена становятся атрибутами уровня класса, экземпляры обзаводятся объединением всех слотовых имен везде в дереве по нормальному правилу наследования

```
class E:
    __slots__ = ['c', 'd'] # Суперкласс имеет слоты
class D(E) :
    __slots__ = ['a', '__dict__'] # Но его подкласс тоже

X = D()
X.a, X.b, X.c = 1, 2, 3
print(X.a, X.b, X.c)
print(X.__dir__())
```

```
1 2 3
['b', '__module__', '__slots__', 'a', '__dict__', '__doc__', 'c', 'd',
```

# Расширения: слоты

Инструменты, пытающиеся обобщенным образом строить списки атрибутов данных экземпляров, обязаны учитывать слоты и возможно другие “виртуальные” атрибуты экземпляров, подобные свойствам и дескрипторам — имена, которые тоже располагаются в классах, но могут предоставлять значения атрибутов для экземпляров по запросу. Слоты ориентированы на данные, но являются типичным представителем этой более широкой категории. Такие атрибуты требуют инклюзивных подходов, специальной обработки или общего игнорирования — последний вариант становится неприемлемым, как только любой программист начнет применять слоты в своем прикладном коде. По правде говоря, атрибуты экземпляров уровня классов, такие как слоты, неизбежно влекут за собой переопределение понятия данных экземпляра — как локально хранящихся атрибутов, объединения всех унаследованных атрибутов или какого-то их подмножества. Например, некоторые программы могут относить слотовые имена к атрибутам классов, а не экземпляров; в конце концов, эти атрибуты не находятся в словарях пространств имен экземпляров.

# Правила использования слотов

***Слоты в подклассах бессмысленны, когда они отсутствуют в суперклассах.*** Если подкласс унаследован от суперкласса без **__slots__** , то атрибут **__dict__** экземпляра, созданный для суперкласса, будет всегда доступен, делая атрибут **__slots__** в подклассе по существу бессмысленным. Подкласс по-прежнему управляет своими слотами, но никак не вычисляет их значения и не избегает словаря - главной причины применения слотов.

***Слоты в суперклассах бессмысленны, когда они отсутствуют в подклассах.*** Аналогично, поскольку объявление **__slots__** ограничено классом, в котором оно появляется, подклассы будут создавать **__dict__** экземпляра, если в них не определен атрибут **__slots__**, делая **__ slots__** в суперклассе в сущности бессмысленным.

***Переопределение делает бессмысленными слоты суперкласса.*** Если класс определяет такое же слотовое имя, как в суперклассе, то согласно нормальному наследованию его переопределение скрывает слот из суперкласса. Вы можете получить доступ к версии имени, которая определена в слоте суперкласса, только путем извлечения его дескриптора напрямую из суперкласса.

***Слоты препятствуют определению стандартных имен.*** Поскольку слоты реализованы в виде дескрипторов уровня класса (вместе с пространством для каждого экземпляра), вы не можете использовать атрибуты класса с такими же именами для предоставления стандартных имен, как могли бы делать для нормальных атрибутов экземпляра: присваивание значения тому же самому имени в классе переопределяет дескриптор слота.

# Правила использования слотов

***Слоты и__ dict__ .***

Как было показано ранее, список __ slots__ предотвращает существование __ dict__ экземпляра и присваивание значений именам, отсутствующим в списке, если __dict__ явно не включен в список.

**Слоты главным образом оптимизируют потребление памяти, их влияние на скорость работы менее ярко выражено.**

# Основы свойств

Свойство представляет собой тип объекта, присвоенного имени атрибута класса. Для создания свойства необходимо вызвать встроенную функцию ***property*** и передать ей три метода доступа (обработчики операций получения, установки и удаления), а также необязательную строку документации для свойства. Если любой из аргументов опущен или для него передается None, то связанная с ним операция не поддерживается. Результирующий объект свойства обычно присваивается имени на верхнем уровне оператора ***class*** (например, ***имя=property()***), и для автоматизации этого шага доступен специальный синтаксис 0, с которым мы встретимся позже. При таком присваивании последующие обращения к имени свойства класса как к атрибуту объекта (скажем, объект, имя) автоматически направляются одному из методов доступа, которые передавались вызову ***property***.

Для ряда кодовых задач свойства могут оказаться менее сложными и более быстрыми при выполнении, чем традиционные методики. Скажем, когда мы добавляем поддержку присваивания значений атрибутам, то свойства становятся более привлекательными - они требуют меньшего объема кода, а для операций присваивания значений атрибутам, которые нежелательно вычислять динамически, не возникают избыточные вызовы методов

# Основы свойств

```
class properties () : # Для методов установки в Python 2.X нужен object
    def getage(self):
        return 40
    def setage(self, value):
        print (f'set age: {value}')
        self._age = value
    age = property(getage, setage, None, None)

x = properties()
print(x.age) # Запускается getage
x.age = 42 # Запускается setage
print(x._age) # Нормальное извлечение: getage не вызывается
x.job = 'trainer' # Нормальное присваивание: setage не вызывается
print(x.job) # Нормальное извлечение: getage не вызывается
```

свойства также возможно реализовывать с использованием синтаксиса декораторов функций в виде символа @, который описан позже в главе; он является эквивалентом и автоматической альтернативой ручному присваиванию на уровне объявления класса

```
40
set age: 42
42
trainer
```

```
class properties():
    @property # Реализация свойств с помощью декораторов
    def age(self):
        ...
    @age.setter
    def age(self, value):
        ...
```

# Статические методы в Python 2.X и 3.X

Концепция статических методов одинакова в линейках Python 2.X и 3.X, но требования к их реализации в Python 3.X получили некоторое развитие.

- Обе линейки Python 2.X и 3.X производят связанные методы, когда метод извлекается через экземпляр.
- В Python 2.X извлечение метода из класса производит несвязанный метод, который не может быть вызван без передачи экземпляра вручную.
- В Python 3.X извлечение метода из класса производит простую функцию, которая может быть вызвана нормально в отсутствие какого-либо экземпляра.

Методы классов в Python 2.X всегда требуют передачи экземпляра независимо от того, вызываются они через экземпляр или через класс. Напротив, в Python 3.X мы обязаны передавать экземпляр методу, только если он его ожидает - методы, не включающие аргумент экземпляра, могут вызываться через класс без передачи экземпляра. То есть Python 3.X разрешает присутствие в классе простых функций при условии, что они не ожидают аргумента экземпляра, и он им не передается. Ниже описан совокупный эффект.

- В Python 2.X мы должны всегда объявлять метод как статический, чтобы вызывать его без экземпляра через класс или через экземпляр.
- В Python 3.X нам не нужно объявлять такие методы как статические, если они будут вызываться только через класс, но мы обязаны делать это, чтобы вызывать их через экземпляр.

# Статические методы в Python 2.X и 3.X

Класс содержит счетчик, хранящийся в виде атрибута класса, конструктор, увеличивающий счетчик на единицу каждый раз, когда создается новый экземпляр, и метод, отображающий значение счетчика. Не забывайте, что атрибуты класса разделяются всеми экземплярами.

```
class Spam:
    numinstances = 0
    def __init__ (self):
        Spam.numinstances = Spam.numinstances + 1
    def printNumInstances():
        print(f"Number of instances created: {Spam.numinstances}" )

a = Spam()
b = Spam()
print(Spam.numinstances)
```

2

Если у вас есть возможность использовать Python 3.X и вызывать методы без аргумента self только через классы, тогда вы уже имеете в своем распоряжении средство статических методов.

# Методы в Python 2.X и 3.X

Формально язык Python теперь поддерживает три вида методов, относящихся к классам, с отличающимися протоколами аргументов:

- методы экземпляров, которым передается объект экземпляра self (стандарт);
- статические методы, которым не передается какой-то дополнительный объект (через staticmethod);
- методы классов, которым передается объект класса (через classmethod, что присуще метаклассам).

На самом деле из-за того, что методы классов всегда получают самый нижний класс в дереве наследования:

- статические методы и явные имена классов могут оказаться лучшим решением при обработке данных, локальных для класса;
- методы классов могут лучше подойти для обработки данных, отличающихся для каждого класса в иерархии

# Методы в Python 2.X и 3.X

Реализуем эквивалентный статический метод для подсчета экземпляров - он помечается как особый, так что ему автоматически экземпляр не передается:

```
class Spam:
    numinstances = 0 # Использование статического метода для данных класса
    def __init__ (self):
        Spam.numinstances += 1
    def printNumInstances():
        print(f'Number of instances: {Spam.numinstances}')

    printNumInstances = staticmethod(printNumInstances)

a = Spam()
b = Spam()
c = Spam()
Spam.printNumInstances() # Вызывается как простая функция
a.printNumInstances() # Аргумент экземпляра не передается
```

```
Number of instances: 3
Number of instances: 3
```

# Методы в Python 2.X и 3.X

Интересно отметить, что метод класса способен здесь делать похожую работу - показанная ниже версия класса обладает тем же самым поведением, что и рассмотренная ранее версия со статическим методом, но в ней применяется метод класса, который получает класс экземпляра в своем первом аргументе. Вместо жесткого кодирования имени класса метод класса обобщенным образом использует автоматически передаваемый объект класса:

```
class Spam:
    numinstances = 0 # Использование статического метода для данных класса
    def __init__ (self):
        Spam.numinstances += 1
    def printNumInstances(cls):
        print(f'Number of instances: {Spam.numinstances}')

    printNumInstances = classmethod(printNumInstances)

a = Spam()
b = Spam()
Spam.printNumInstances() # В первом аргументе передается класс
a.printNumInstances() # Также в первом аргументе передается класс
```

```
Number of instances: 2
Number of instances: 2
```

Статические методы и методы классов исполняют дополнительные расширенные роли, которые в главе не рассматриваются; ищите описание добавочных сценариев на других ресурсах. Тем не менее, в последних версиях Python обозначение статических методов и методов классов стало даже еще проще с появлением синтаксиса декорирования функций - способа применения одной функции к другой.

# Основы ООП

## Лекция 9. Знакомство Django. Структура проекта. Представление и маршрутизация.

# Общие представления о Django

***Django*** - это веб-фреймворк, написанный на Python и для Python, позволяющий быстро создавать безопасные и удобно поддерживаемые веб-сайты. Django бесплатный и с открытым исходным кодом, имеет активное сообщество, отличную документацию и множество вариантов как бесплатной, так и платной поддержки.

**Отличительные особенности:**

- ***Полнокомплектный инструмент.*** Django следует философии «Все в одном флаконе» и предоставляет почти все необходимое, что разработчикам может понадобиться при разработке своих проектов. Поскольку все, что нужно, является частью единого продукта, все модули безупречно работают вместе в соответствии с последовательными принципами проектирования. Для него также имеется подробная документации.

- ***Универсальный инструмент.*** Django может быть использован для создания практически любых типов веб-сайтов - от систем управления контентом до социальных сетей и новостных порталов. Он может работать с любой клиентской средой и способен доставлять контент практически в любом формате (HTML, RSS-каналы, JSON, XML и т. д.). На Django может быть реализована практически любая функциональность, которая может понадобиться конечному пользователю, он работает с различными популярными базами данных, при необходимости его базовые модули могут быть дополнены сторонними компонентами.

# Общие представления о Django

- ***Инструмент для разработки безопасных приложений.*** Django помогает разработчикам избежать многих распространенных ошибок безопасности, предоставляя фреймворк с автоматической защитой сайта. Так, в Django реализован безопасный способ управления учетными записями пользователей и паролями, что позволяет избежать распространенных ошибок - таких, например, как размещение информации о сеансе в файлах cookie, где она уязвима. В Django файлы cookie содержат только ключ, а содержательная информация хранится отдельно в базе данных. Все пароли хранятся только в зашифрованном или хэшированном виде. Как известно, хэшированный пароль - это пароль фикси , рованной длины, созданный путем его обработки через криптографическую хэш-функцию. Django может проверить правильность введенного пароля, пропустив его через хэш-функцию и сравнив вывод с сохраненным значением хэша. Благодаря «одностороннему» характеру функции, даже если сохраненное хэшзначение скомпрометировано, злоумышленнику будет сложно определить исходный пароль.

- ***Масштабируемые приложения.*** Django использует компонентную архитектуру, при которой каждая часть создаваемого приложения независима от других частей и, следовательно, может быть заменена либо изменена. Четкое разделение между частями означает, что Django может масштабироваться при увеличении трафика путем добавления оборудования на любом уровне: серверы кэширования, серверы баз данных или серверы приложений. Одни из самых загруженных сайтов - Instagram - успешно масштабирован на Django.

# Общие представления о Django

- ***Разработанные приложения удобны в сопровождении.*** Код Django написан на основе принципов и шаблонов проектирования, которые поощряют создание поддерживаемого и повторно используемого кода. В частности, в нем задействован принцип DRY (Don't Repeat Yourself, не повторяйся), что позволяет избежать ненужного дублирования и сокращает объем кода. Django также способствует группированию связанных функциональных возможностей в повторно используемые приложения и группирует связанный программный код в модули в соответствии с концепцией ***Model-View-Controller (MVC)***. ***Model-View-Controller***, или MVC (можно перевести как «Модель-Представление-Контроллер», или «Модель-Вид-Контроллер» ), - это схема разделения приложения на три отдельных компонента: модель (описывает структуру данных), представление (отвечает за отображение данных) и контроллер (интерпретирует действия пользователя). Таким образом, в разработанном приложении модификация каждого компонента может осуществляться независимо друг от друга.

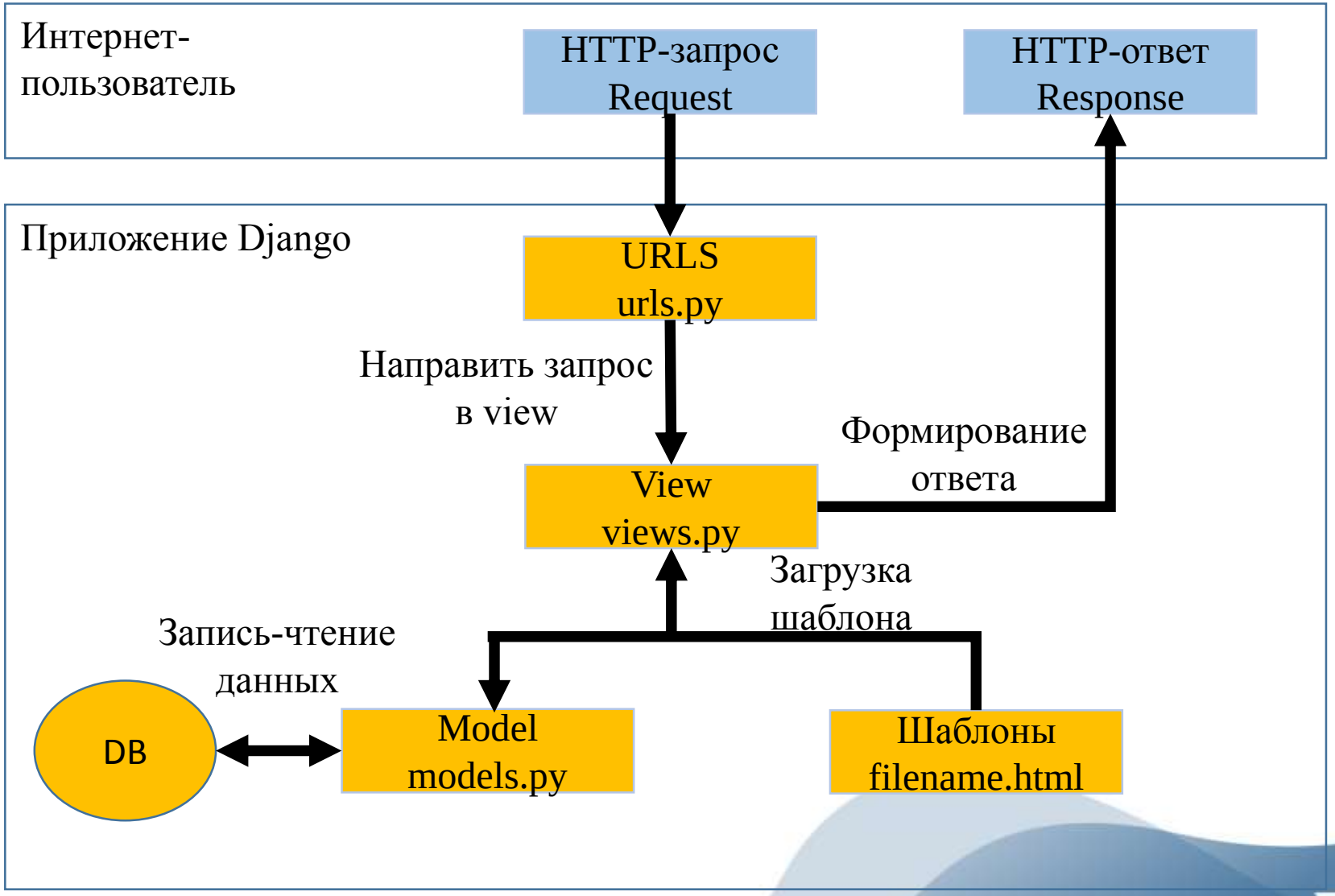
- ***Разработанные приложения являются кросс-платформенными.*** Django написан на Python, который работает на многих платформах. Это означает, что вы не привязаны к какой-либо конкретной серверной платформе и можете запускать приложения на многих версиях Linux, Windows и macOS. Кроме того, Django хорошо поддерживается многими веб-хостингами, которые часто предоставляют определенную инфраструктуру и документацию для размещения сайтов Django.

# Структура приложений

В типовом информационном сайте веб-приложение ожидает HTTP-запросы от веббраузера пользователя. Получив запрос, приложение обрабатывает его и выполняет запрограммированные действия. Затем приложение возвращает ответ веб-браузеру пользователя. Возвращаемая страница может быть либо статической, либо динамической. В последнем случае некоторые данные вставляются в HTML-шаблон.

Веб-приложение, написанное на Django, разбито на четыре базовых блока, которые содержатся в отдельных файлах, не зависимых друг от друга, но при этом и работающих в связке. Фреймворк Django реализует архитектуру ***Model-View-Template***, или, сокращенно, ***MVT***, которая по факту является модификацией распространенной в веб-программировании архитектуры ***MVC*** (***Model-View-Controlle***r).

# Структура приложений



# Структура приложений

Базовые блоки веб-приложения :

**Диспетчер URL-адресов (URLs).** Хотя можно обрабатывать запросы с каждого URL-адреса с помощью одной функции, гораздо удобнее писать отдельную функцию для обработки каждого ресурса. URL-mapper используется для перенаправления HTTP-запросов в соответствующее представление (View) на основе URL-адреса запроса. URL-mapper также может извлекать данные из URL-адреса в соответствии с заданным шаблоном и передавать их в соответствующую функцию в виде аргументов.

**Модели (Models).** Модели представляют собой объекты Python, которые определяют структуру данных приложения и предоставляют механизмы для управления данными (добавления, изменения, удаления) и выполнения запросов к базе данных.

**Шаблоны (Templates).** Template (шаблон)- это текстовый файл, определяющий структуру или разметку страницы (например, HTML-страницы), с полями, которые используются для подстановки актуального содержимого из базы данных или параметров, вводимых пользователем.

# Структура приложений

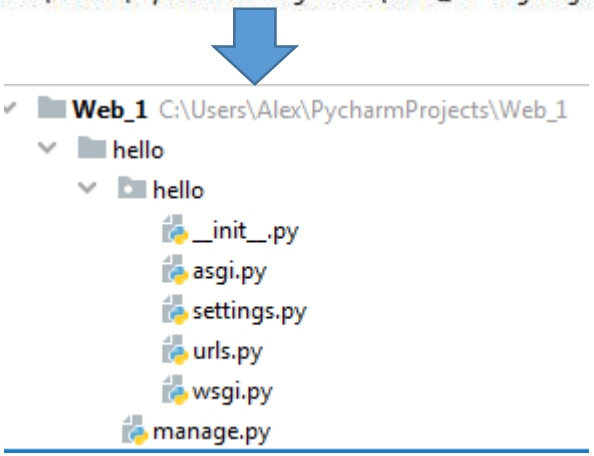
Базовые блоки веб-приложения :

**Представление (View).** Центральная роль в этой архитектуре принадлежит представлению (view). Когда к приложению Django приходит запрос от удаленного пользователя (HTTP-запрос), то URL-диспетчер определяет, с каким ресурсом нужно сопоставить этот запрос, и передает его выбранному ресурсу. Ресурсом в этом случае является представление (view), которое, получив запрос, определенным образом обрабатывает его. В процессе обработки запроса представление (view) может обращаться через модели (model) к базе данных, получать из нее данные или, наоборот, сохранять данные в нее. 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 ).

# Базовый проект

При установке Django в папке виртуальной среды автоматически устанавливается скрипт django-admin.py, а в Windows - также исполняемый файл django-admin.exe. Веб-приложение, или проект Django, состоит из отдельных приложений. Полноценное веб-приложение они образуют в совокупности. Каждое приложение представляет какую-то определенную функциональность или группу функций. Один проект может включать множество приложений. Это позволяет выделить группу задач в отдельный модуль и разрабатывать их относительно независимо от других. Кроме того, мы можем переносить приложение из одного проекта в другой, независимо от функциональности проекта. Теперь можно войти в окно терминала PyCharm и создать новый проект Django с именем hello.

```
Terminal:  Local
Windows PowerShell
(C) Корпорация Майкрософт, 2009. Все права защищены.

PS C:\Users\Alex\PycharmProjects\Web_1> django-admin startproject hello
```



# Базовый проект

***папка Web_1/*** - корневой каталог (папка) нашего проекта PyChann. Он представляет собой контейнер, в котором будут создаваться наши приложения и файлы для управления проектом. Название папки ничего не значит для Django, и его можно поменять на свое усмотрение;

***папка Web_1/ hello*** - это внешняя папка нашего веб-проекта;

***файл manage.py*** - инструмент управления из командой строки, при помощи которого можно управлять проектом различными путями. В частности, при помощи этого инструмента вы можете запустить на исполнение свой проект на сервере Python;

***папка Web_1/hello/hello/-*** внутренняя папка проекта. Она содержит текущий и единственный на текущий момент пакет (или, вернее, проект Python) в нашем проекте. Имя этого пакета в дальнейшем будет использовано для импорта внутренней структуры кода;

***файл hello/__init__.py*** - этот пустой файл предназначен для указания и регистрации пакетов. Наличие его в каталоге Web_1/ указывает интерпретатору Python, что текущий каталог будет рассматриваться в качестве пакета;

***файл hello/asgi.py*** - это точка входа для ASGI-совместимых веб-серверов, обслуживающих ваш проект (он потребуется при развертывании приложения на публичном сайте);

***файл hello/settings.py*** - это файл установок и конфигурации текущего проекта Django;

***файл hello/urls.py*** - это URL-декларации текущего проекта Django, или, иначе говоря, это «таблица контента» Django-nроекта;

***файл hello/wsgi.py*** - это точки входа для WSGI-совместимого веб-сервера (он потребуется при развертывании приложения на публичном сайте).

# Базовый проект

***ASGI (англ. Asynchronous Server Gateway Interface)*** - клиент-серверный протокол взаимодействия веб-сервера и приложения, дальнейшее развитие технологии WSGI. По сравнению с WSGI предоставляет стандарт как для асинхронных, так и для синхронных приложений, с реализацией обратной совместимости WSGI и несколькими серверами и платформами приложений.

***WSGI (англ. Web Server Gateway Interface)*** - стандарт взаимодействия между Python-программой, выполняющейся на стороне сервера, и самим веб-сервером

# Базовый проект

Пока в проекте Django еще нет ни одного приложения. Несмотря на это, мы уже можем запустить проект на выполнение. Для этого нужно сначала войти в папку hello, для чего в окне терминала PyCharm выполнить следующую команду:

***cd hello***

Теперь, находясь в этой папке, можно запустить наш проект на выполнение. Наберите в окне терминала команду запуска локального сервера:

***python manage.py runserver***

```
Django version 3.1.7, using settings 'hello.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

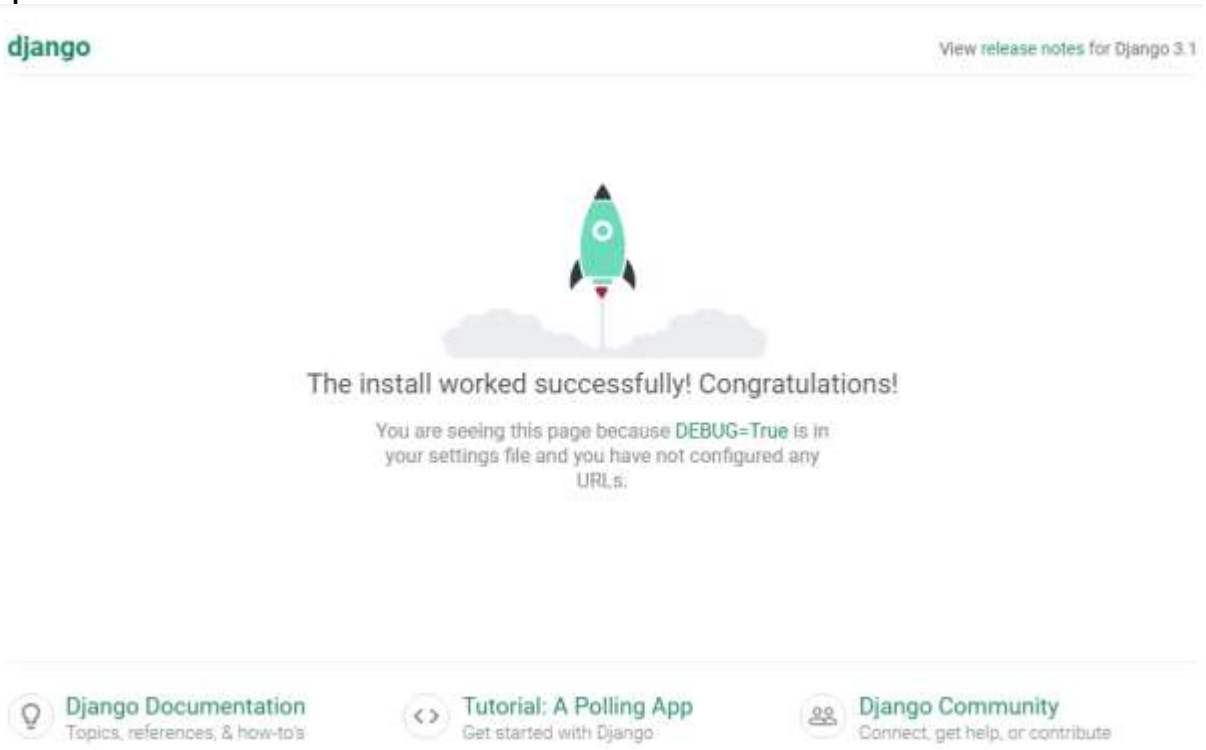
Если после выполнения этой команды вы получили сообщение об ошибке:

*UnicodeDecodeError: 'utf-8' codec can't decode byte Dxcf in position 5:*
*invalid continuation byte*

то это означает, что в имени вашего компьютера присутствуют символы из русского алфавита - например, Vit-ПК. В таком случае нужно войти в свойства компьютера и поменять его имя (убрать символы ПК).

# Базовый проект

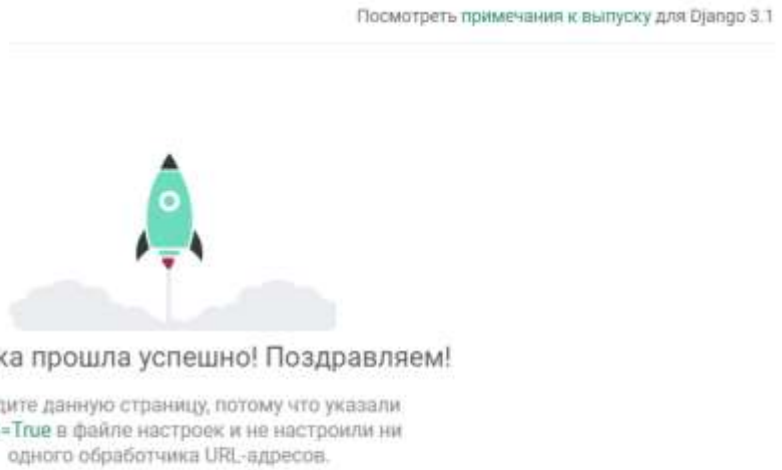
Если локальный сервер запустился без ошибок, щелкните левой кнопкой мыши на его ссылке: http://127.0.0.1:8000/



# Базовый проект

Остановите локальный веб-сервер нажатием комбинации клавиш <***Ctrl***>+<***C***>. Откройте файл ***settings.py*** и найдите фрагмент кода с установкой языка

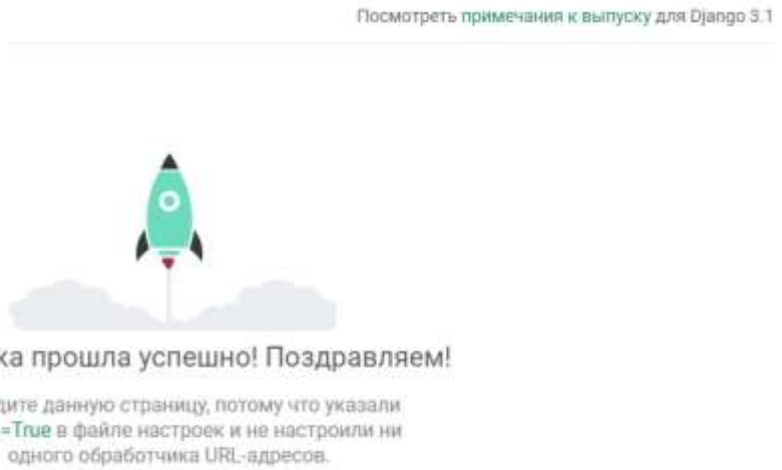
```
# Internationalization
# https://docs.djangoproject.com/en/3.1/topics/i18n/

LANGUAGE_CODE = 'ru-ru'
TIME_ZONE = 'UTC'
USE_I18N = True
USE_L10N = True
USE_TZ = True
```



# Базовый проект

Остановите локальный веб-сервер нажатием комбинации клавиш <***Ctrl***>+<***C***>. Откройте файл ***settings.py*** и найдите фрагмент кода с установкой языка

```
# Internationalization
# https://docs.djangoproject.com/en/3.1/topics/i18n/

LANGUAGE_CODE = 'ru-ru'
TIME_ZONE = 'UTC'
USE_I18N = True
USE_L10N = True
USE_TZ = True
```



# Базовый проект

***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 «клиент-сервер». 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 **settings.py**

```
# Database
# https://docs.djangoproject.com/en/3.1/ref/settings/#databases

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)

# Базовый проект

При создании проекта он уже содержит несколько приложений по умолчанию:

- ❑ *django.contrib.admin;*
- ❑ *django.contrib.auth;*
- ❑ *django.contrib.contenttypes;*
- ❑ *django.contrib.sessions;*
- ❑ *django.contrib.messages;*
- ❑ *django.contrib.staticfiles*

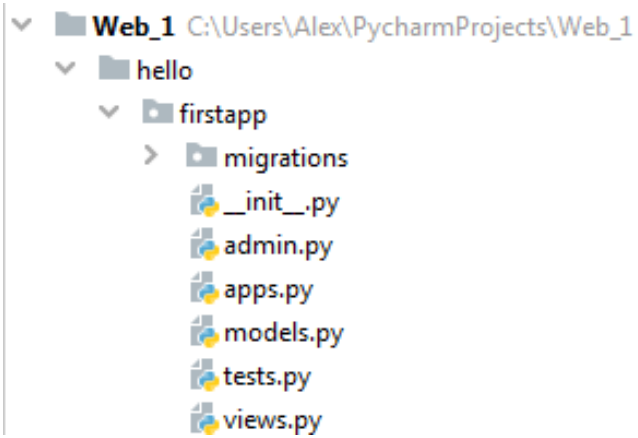
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Для создания нового приложения в окне терминала PyCharm выполните следующую команду

```
PS C:\Users\Alex\PycharmProjects\Web_1\hello> python manage.py startapp firstapp
```

Имя приложения может быть любым - оно указывается после команды ***startapp***.

В результате работы этой команды в проекте Django будет создано приложение ***firstapp***, а в проекте появится новая папка, в которой будут хранить все файлы созданного приложения



# Базовый проект

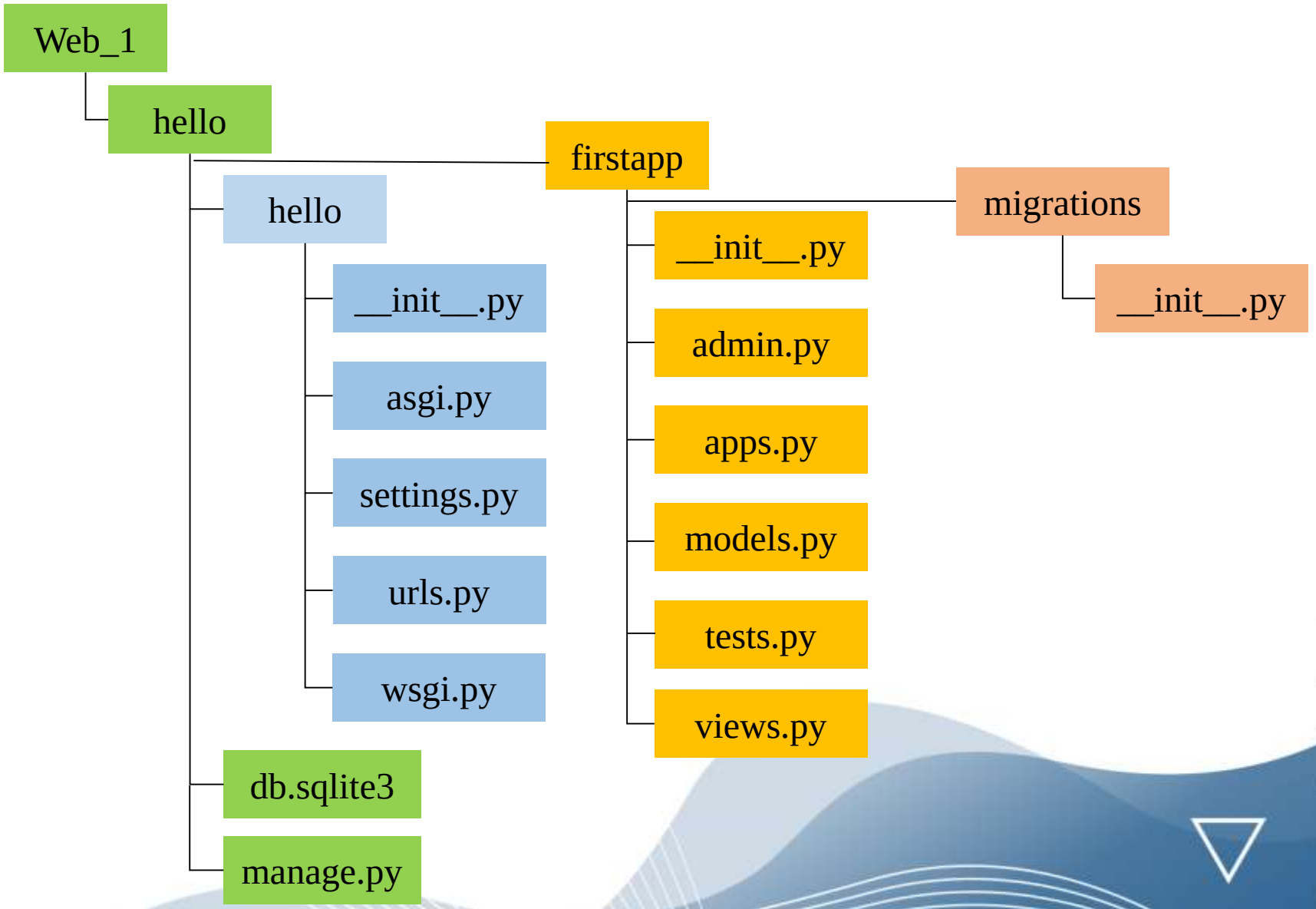
Рассмотрим структуру папок и файлов приложения firstapp:

- ***папка migrations*** - хранит информацию, позволяющую сопоставить базу данных со структурой данных, описанных в модели;
- ***файл _init_.py*** - указывает интерпретатору Python, что текущий каталог будет рассматриваться в качестве пакета;
- ***файл admin.py*** - предназначен для административных функций. В частности, здесь производится регистрация моделей, которые используются в интерфейсе администратора;
- ***файл apps.py*** - определяет конфигурацию приложения;
- ***файл models.py*** - хранит определение моделей, которые описывают используемые в приложении данные;
- ***файл tests.py*** - хранит тесты приложения;
- ***файл views.py*** - определяет функции, которые получают запросы пользователей, обрабатывают их и возвращают ответ

Приложение никак не задействовано, его надо зарегистрировать в проекте Django. Для этого нужно открыть файл ***settings.py*** и добавить в конец массива с приложениями проекта **INSTALLED _ APPS** наше приложение ***firstapp.***

# Базовый проект

Полная структура файлов и папок веб-приложений с проектом ***hello*** и приложением ***firstapp***



# Базовый проект

В проекте может быть несколько приложений, и каждое из них нужно добавлять аналогичным образом. Теперь определим какие-нибудь простейшие действия, которые будет выполнять это приложение, - например, отправлять в ответ пользователю приветствие Hello World.

Для этого перейдем в проекте приложения firstapp к файлу ***views.py***, который по умолчанию должен выглядеть так

```
from django.shortcuts import render

# Create your views here.
```

Импортируем класс ***HttpResponse*** из стандартного пакета ***django.http***. Затем определяем функцию ***index( )*** , которая в качестве параметра получает объект запроса пользователя ***request***. Класс ***HttpResponse*** предназначен для создания ответа который отправляется пользователю. И с помощью кода ***return HttpResponse*** мы отправляем пользователю строку: *"Hello World ! Это мой первый проект на Django ! “*

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def index(request):
    return HttpResponse("Hello World! Это мой первый проект на Django!")
```

# Базовый проект

В проекте Django файл ***urls.py***, позволяет сопоставить маршруты с представлениями, обрабатывающими запрос по этим маршрутам. По умолчанию файл ***urls.py*** выглядит так:

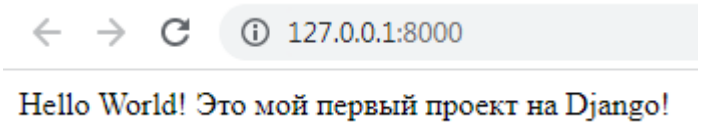
```
from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

В первой строке из модуля ***django.contrib*** импортируется класс ***AdminSite***, который предоставляет возможности работы с интерфейсом администратора сайта. Во второй строке из модуля ***django.urls*** импортируется функция ***path***. Эта функция задает возможность сопоставлять запросы пользователя (определенные маршруты) с функцией их обработки. Так, в нашем случае маршрут ***'admin/'*** будет обрабатываться методом ***admin.site.urls***. То есть если пользователь запросит показать страницу администратора сайта (***'admin/'*** ), то будет вызван метод ***admin.site.urls***, который вернет пользователю HTML-страницу администратора. Но ранее мы определили в файле ***views.py*** функцию ***index***, которая возвращает пользователю текстовую строку. Поэтому изменим файл ***urls.py***,

# Базовый проект

Чтобы использовать функцию ***views. index***, здесь мы вначале импортируем модуль views. Затем сопоставляем маршрут ('') - это, по сути, запрос пользователя к корневой странице сайта, с функцией ***views.index***, а также дополнительно задаем имя для этого маршрута (***name='home'***). То есть если пользователь запросит показать главную (корневую) страницу сайта (' '), то будет вызвана функция ***index*** из файла ***views.py***. А в этой функции мы указали, что пользователю нужно вернуть HTML-страницу с единственным сообщением: "Hello World! Это мой первый проект на Django ! ". По сути, маршрут с именем 'home' будет сопоставляться с запросом к корню приложения.

```
from django.contrib import admin
from django.urls import path
from firstapp import views
urlpatterns = [
    path('', views.index, name = 'home'),
    path('admin/', admin.site.urls),
]
```



# Представления и маршрутизация

Центральным моментом любого веб-приложения является обработка запроса, который отправляет пользователь. В Django за обработку запроса отвечают ***представления (views)***. По сути, в представлениях реализованы функции обработки, которые принимают данные запроса пользователя в виде объекта ***request*** и генерируют некий ***ответ (response)***, который затем отправляется пользователю в виде HTML страницы. По умолчанию представления размещаются в приложении в файле ***views.py***.

При создании нового проекта файл views.py имел следующее содержимое:

```
from django.shortcuts import render
```

Этот код пока никак не обрабатывает запросы - он только импортирует функцию render ( оказывать, предоставлять), которая может использоваться для обработки входящих запросов.

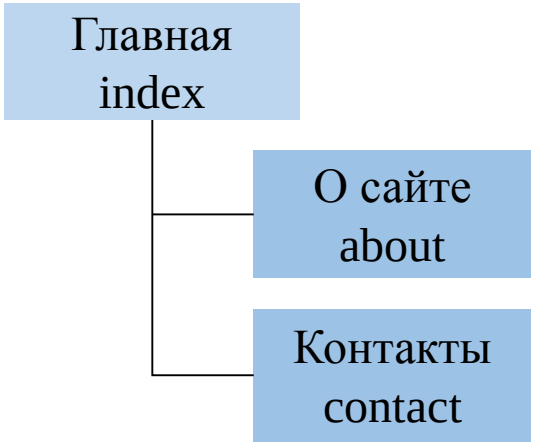
Генерировать результат обработки запроса (ответ пользователю) можно различными способами. Одним из таких способов является использование класса ***HttpResponse*** (ответ HTTP) из пакета ***django.http***, который позволяет отправить текстовое содержимое.

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def index(request):
    return HttpResponse("<h2>Главная<h2>")

def about(request):
    return HttpResponse("<h2>О сайте<h2>")

def contact(request):
    return HttpResponse("<h2>Контакты<h2>")
```

Этим кодом запрограммированы три функции для выдачи заголовков трех страниц сайта: ***Главная (index), О сайте (about), Контакты (contact).***

# Представления и маршрутизация

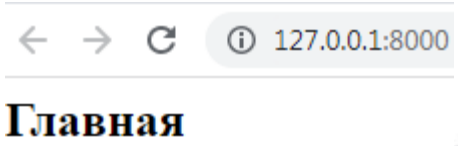
В нашем случае здесь определены три функции, которые будут обрабатывать запросы пользователя. Каждая функция принимает в качестве параметра объект ***request (запрос)***. Для генерации ответа в конструкторе объекта ***HttpResponse (ответ HTTP)*** имеется некоторая строка. Этим ответом может быть и строка кода ***HTML***. Чтобы эти функции сопоставлялись с запросами пользователя, надо в файле ***urls.py*** определить для них в проекте маршруты



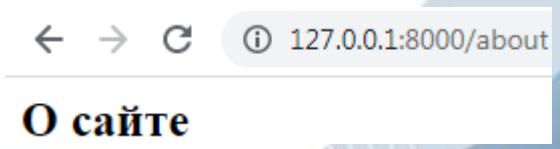

```python
from django.urls import path
from firstapp import views
urlpatterns = [
    path('', views.index),
    path('about', views.about),
    path('contact', views.contact),
]
```

Переменная ***urlpatterns*** (шаблон URL) определяет набор сопоставлений запросов пользователя с функциями обработки этих запросов. В нашем случае, например, запрос пользователя к корню веб-сайта (' ' ) будет обрабатываться функцией ***index***, запрос по адресу ***'about' - функцией about***, а запрос ***'contact'*** - функцией contact.

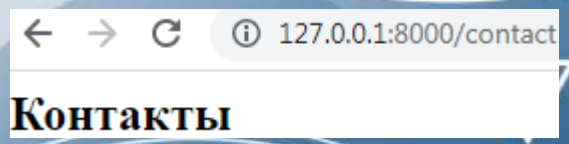
http://127.0.0.1:8000/



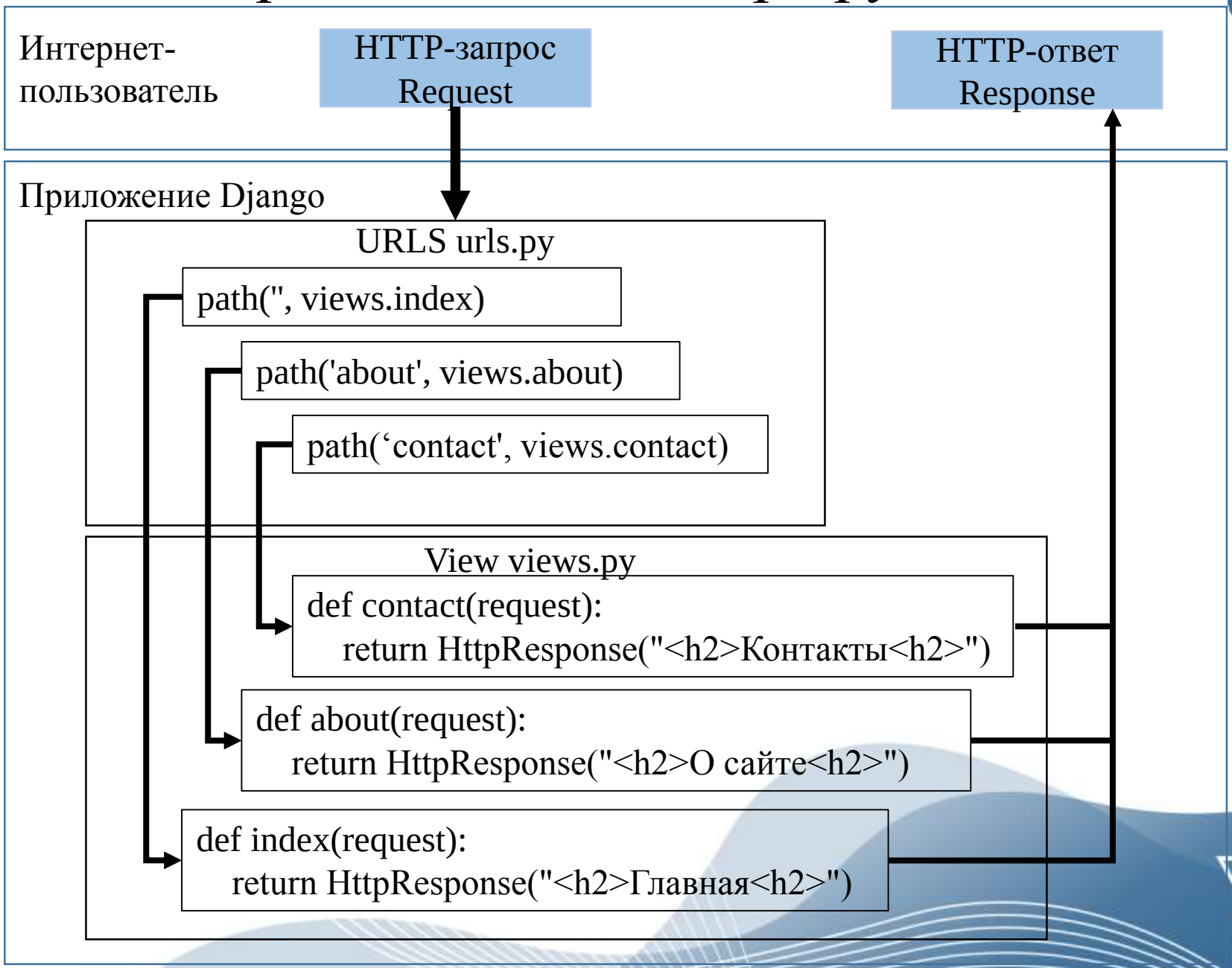

http://127.0.0.1:8000/about




http://127.0.0.1:8000/contact



# Представления и маршрутизация

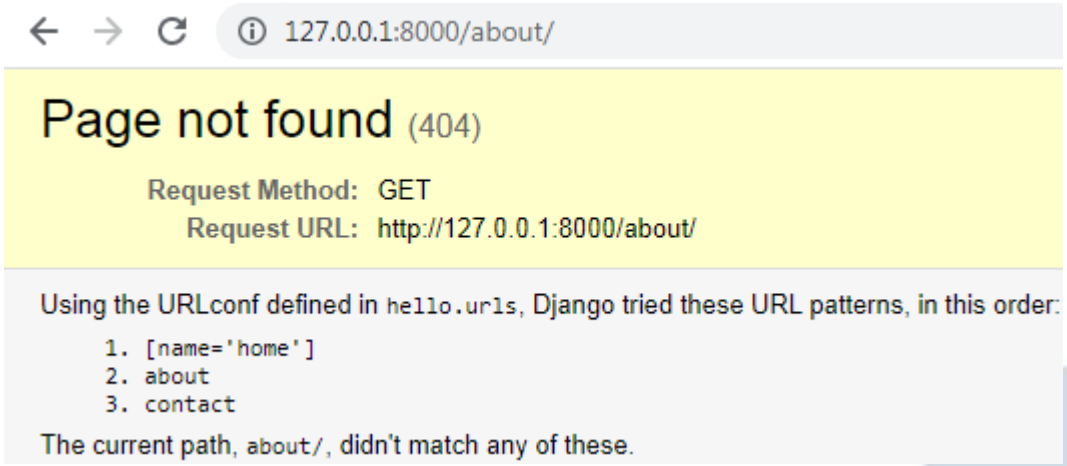


# Представления и маршрутизация

Функции в файле ***views.py*** сопоставляются с адресами URL с помощью функции **path ()** в файле ***urls.py.*** Здесь мы воспользовались функцией ***path()***, расположенной в пакете ***django.urls*** и принимающей два параметра: ***запрошенный адрес URL*** и ***функцию***, которая обрабатывает запрос по этому адресу. Дополнительно через третий параметр можно указать имя маршрута

```
path('', views.index, name='home')
```

Однако функция path имеет серьезное ограничение- запрошенный путь должен абсолютно точно соответствовать указанному в маршруте адресу URL. Так, в приведенном ранее примере, функция ***views.about*** сможет корректно обработать запрос только в том случае, когда адрес будет в точности соответствовать значению. Например, стоит добавить к этому значению слеш ***(' about/' ),*** и Django уже не сможет сопоставить путь с этим запросом

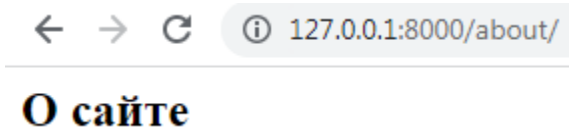


# Представления и маршрутизация

В качестве альтернативы для определения маршрутов мы можем использовать функцию ***re_path(),***также расположенную в пакете ***django.urls***. Ее преимущество состоит в том, что она позволяет задать адреса URL с помощью регулярных выражений. В качестве примера изменим содержание файла ***urls.py*** следующим образом

```
from django.urls import path
from django.urls import re_path
from firstapp import views
urlpatterns = [
    path('', views.index, name='home'),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
]
```

Однако он необязательно в точности должен соответствовать строке ***'about '***, как это было в случае с функцией ***path***. Мы можем обратиться по любому адресу - главное, чтобы он начинался с ***about***, и тогда подобный запрос будет корректно обрабатываться функцией ***views.about***. Если мы теперь укажем слеш в конце ***('about/')***, то Django корректно сопоставит путь с этим запросом, и мы в ответ получим запрошенную страницу.



# Представления и маршрутизация

Когда к приложению приходит запрос от пользователя, система проверяет соответствие запроса маршрутам по мере их следования в модуле ***urls.py***: вначале сравнивается первый маршрут, если он не подходит, то сравнивается второй и т. д. Поэтому более общие маршруты должны определяться в последнюю очередь, а более конкретные маршруты - располагаться в начале этого модуля. Рассмотрим следующий пример следования маршрутов в переменной urlpatterns

```
urlpatterns = [
    re_path(r'^about/contact/', views.contact),
    re_path(r'^about', views.about),
    path('',views.index),
]
```

Здесь адрес ‘^about/contact/’ представляет собой более конкретный маршрут по сравнению с ‘^about’ . Поэтому он определяется в первую очередь.

# Представления и маршрутизация

Некоторые базовые элементы регулярных выражений, которые можно использовать для определения адресов URL

- ^ - начало адреса;
- $ -конец адреса;
- + -1 и более символов;
- ? -0 или 1 символ;
- { n } - n символов;
- {n, m} -от n до m символов;
- . -любой символ;
- \d+ -одна или несколько цифр;
- \D+ - одна или несколько НЕ цифр;
- \w+ -один или несколько буквенных символов.

| Адрес | Запрос |
|---|---|
| r'^$' | **http://127.0.0.1/** (корень сайта) |
| r'^about' | **http://127.0.0.1/about/** или<br>**http://127.0.0.1/about/contact** |
| r'^about\contact' | **http://127.0.0.1/about/contact** |
| r'^products/\d+/' | **http://127.0.0.1/products/23/** или<br>**http://127.0.0.1/products/6459/abc**<br>Но не соответствует запросу:<br>**http://127.0.0.1/products/abc/** |

# Представления и маршрутизация

Функции-представления могут принимать параметры, через которые можно передавать различные данные. Такие параметры передаются в адресе URL. Например, в запросе: ***http://localhost/index/3/5/*** последние два сегмента ***(3/5/)*** могут представлять параметры URL, которые можно связать с параметрами функции-представления через систему маршрутизации. определим в этом приложении в файле views.py следующие дополнительные функции

```
def products(request, productid):
    output = f"<h2>Продукт № {productid}</h2>"
    return HttpResponse(output)

def users(request, id, name):
    output = f"<h2>Пользователь</h2> <h3>id: {id} Имя:{name}</h3>"
    return HttpResponse(output)
```

Теперь изменим файл ***urls.py***, чтобы он мог сопоставить эти функции с запросами пользователя

```
from django.urls import path
from django.urls import re_path
from firstapp import views
urlpatterns = [
    re_path(r'^products/(?P<productid>\d+)/', views.products),
    re_path(r'^users/(?P<id>\d+)/(?P<name>\D+)/', views.users),
    path('', views.index)
]
```

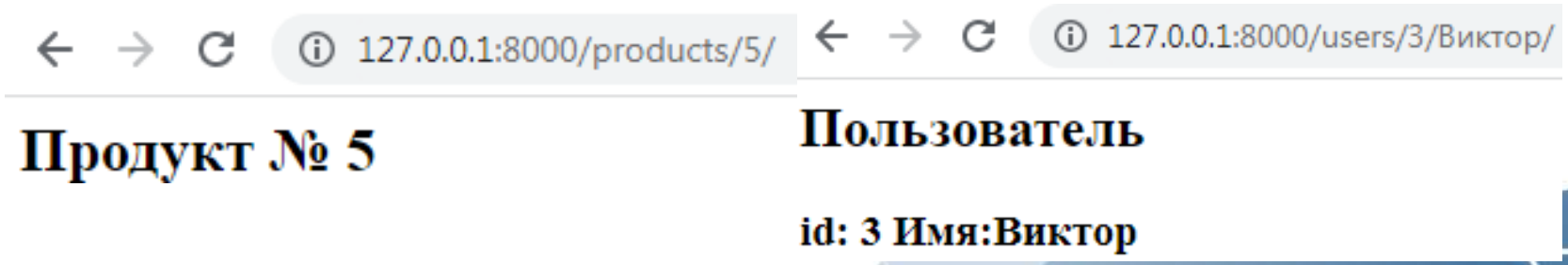
# Представления и маршрутизация

Для представления параметра в шаблоне адреса используется выражение ?P<>.
Общее определение параметра соответствует формату:

***(?P<имя_параметра>регулярное_выражение)***

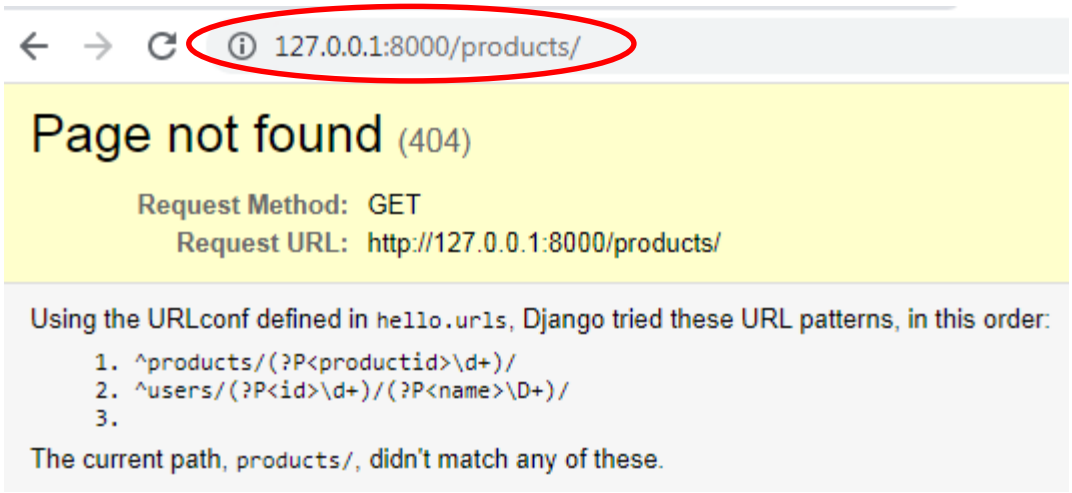
В угловых скобках помещается название параметра. После закрывающей угловой скобки следует регулярное выражение, которому должно соответствовать значение параметра. Например, фрагмент: ***?P<productid>\d+*** определяет, что параметр называется ***productid***, и он должен соответствовать регулярному выражению ***\d+***, т.е. представлять последовательность цифр.

Во втором шаблоне адреса**: (?P<id>\d+)/(?P<name>\D+)** определяются два параметра:
***id и name.*** При этом параметр ***id*** должен представлять число, а параметр ***name*** - состоять только из буквенных символов. Количество и название параметров в шаблонах адресов URL соответствуют количеству и названиям параметров соответствующих функций, которые обрабатывают запросы по этим адресам.



# Представления и маршрутизация

Однако если мы в запросе не передадим значение для параметра или передадим значение, которое не соответствует регулярному выражению, то система не сможет найти ресурс для обработки такого запроса и выдаст сообщение об ошибке. Например если пользователь запросит информацию о продукте, но при этом не укажет идентификационный номер продукта, то система выдаст в ответ следующее сообщение об ошибке



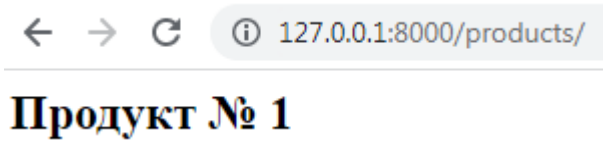
# Представления и маршрутизация

Для предотвращения ошибок такого рода можно в функции products, расположенной в файле ***views.py***, определить значение параметра по умолчанию

```
def products(request, productid=1):
    output = f"<h2>Продукт № {productid}</h2>"
    return HttpResponse(output)
```

То есть если в функцию не было передано значение для параметра productid, то он получает значение по умолчанию 1. В этом случае в файле urls.py надо дополнительно определить еще один маршрут:

```
from django.contrib import admin
from django.urls import path
from django.urls import re_path
from firstapp import views
urlpatterns = [
    path(' ', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    re_path(r'^products/$', views.products),  # маршрут по умолчанию
    re_path(r'^products/(?P<productid>\d+)/', views.products),
    re_path(r'^users/(?P<id>\d+)/(?P<name>\D+)/', views.users),
]
```

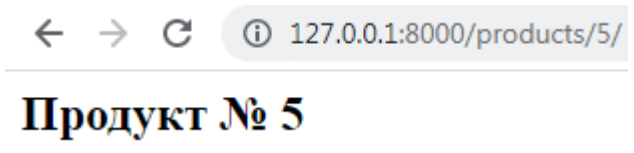



Теперь если в запросе пользователя не будет указан id продукта, то система вернет ему не страницу с ошибкой, а страницу с номером продукта, который в функции products был задан по умолчанию: productid=l

# Представления и маршрутизация

Функция ***path()*** работает здесь аналогично функции ***re_path***. То есть в зависимости от предпочтений программиста можно использовать любую из функций. Параметры функции ***path ()*** заключаются в угловые скобки в формате: ***<спецификатор:название_параметра>***
В примере в маршруте обращения к странице с продуктами параметр ***productid*** имеет спецификатор ***int*** (целое число).

```
from django.urls import path
from django.urls import re_path
from firstapp import views
urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    path('products/<int:productid>/', views.products),
    path('users/<int:id>/<name>/', views.users),
]
```




По умолчанию Django предоставляет следующие спецификаторы параметров функции:

- **str** - соответствует любой строке за исключением символа (/). Если спецификатор не указан, то используется по умолчанию;
- **int** -соответствует любому положительному целому числу;
- **slug**- соответствует последовательности буквенных символов ASCII, цифр, дефиса и символа подчеркивания, например: building-your-1st-django-site;
- **uuid**- соответствует идентификатору UUID, например: 075194d3-6885-417e-a8a8-6c93le272fOO;
- **path**- соответствует любой строке, которая также может включать символ(/), в отличие от спецификатора str

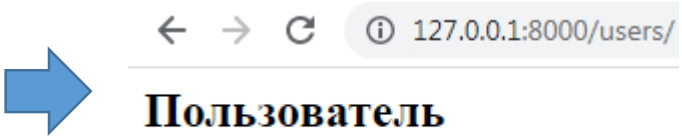
# Представления и маршрутизация

Для примера зададим для функций в файле ***views.py*** значения параметров по умолчанию для тех страниц сайта, которые выдают информацию о продуктах и пользователях. Мы уже ранее задавали для продуктов значение по умолчанию: ***productid=1.*** Теперь зададим для пользователя значения по умолчанию идентификатора пользователя (***id=1***) и имени пользователя (***name="Максим"***).

```
def users(request, id=1, name='Максим'):
    output = f"<h2>Пользователь</h2> <h3>id: {id} Имя:{name}</h3>"
    return HttpResponse(output)
```

После этого для функций products и users в файле urls.py надо определить по два маршрута

```
from django.contrib import admin
from django.urls import path
from django.urls import re_path
from firstapp import views
urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    path('products/', views.products),  # маршрут по умолчанию
    path('products/<int:productid>/', views.products),
    path('users/', views.users),  # маршрут по умолчанию
    path('users/<int:id>/<str:name>/', views.users),
]
```



# Представления и маршрутизация

Следует четко различать параметры, которые передаются через интернет-адрес (URL), и параметры, которые передаются через строку запроса. Например, в запросе:
***http://localhost/index/3/Виктор/***
два последних сегмента: ***3/Виктор/*** представляют собой параметры URL. А в запросе:
***http://localhost/index?id=3&name= Виктор***
те же самые значения 3 и Виктор представляют собой параметры строки запроса.

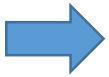
Параметры строки запроса указываются после символа вопросительного знака (?).
Каждый такой параметр представляет собой пару «ключ-значение». Например, в параметре ***id=3: id*** - это ключ параметра, а ***3*** - его значение. Параметры в строке запроса отделяются друг от друга знаком амперсанда (&). Для получения параметров из строки запроса применяется метод ***request.GET.get ().***

```
def products(request, productid):
    category = request.GET.get("cat", "")
    output = f"<h2>Продукт № {productid} Категория {category}</h2>"
    return HttpResponse(output)

def users(request, id, name):
    id = request.GET.get("id", 1)
    name = request.GET.get("name", "Максим")
    output = f"<h2>Пользователь</h2> <h3>id: {id} Имя:{name}</h3>"
    return HttpResponse(output)
```

# Представления и маршрутизация

Функция ***products*** принимает обычный параметр productid (идентификатор продукта), который будет передаваться через интернет-адрес (URL). И также из строки запроса извлекается значение параметра ***cat*** (категория продукта) - ***request.GET.get("cat", "").*** Здесь первый аргумент функции - это название параметра строки запроса, значение которого надо извлечь, а второй аргумент - значение по умолчанию (на случай, если в строке запроса не оказалось подобного параметра). В функции ***users*** из строки запроса извлекаются значения параметров ***id*** и ***name***. При этом заданы следующие значения параметров по умолчанию: ***id=1, name= "Максим".***

```
from django.urls import path
from django.urls import re_path
from firstapp import views
urlpatterns = [
    path('', views.index),
    re_path(r'^about', views.about),
    re_path(r'^contact', views.contact),
    path('products/<int:productid>/', views.products),
    path('users/', views.users),
]
```




При обращении к приложению по адресу:
***http://127 .0.0.1 :8000/products/3/?cat=Телефоны***
число 3 будет представлять параметр URL, присваиваемый параметру productid, а значение cat=Телефоны - представлять параметр cat строки запроса

# Представления и маршрутизация

При перемещении документа с одного адреса на другой мы можем воспользоваться механизмом переадресации, чтобы указать пользователям и поисковику, что документ теперь доступен по новому адресу.
Переадресация бывает временная и постоянная. При временной переадресации мы указываем, что документ временно перемещен на новый адрес. В этом случае в ответ отправляется статусный код ***302***. При постоянной переадресации мы уведомляем систему, что документ теперь постоянно будет доступен по новому адресу.

Для создания временной переадресации применяется класс:
***HttpResponseRedirect***
Для создания постоянной переадресации применяется класс:
***HttpResponsePermanentRedirect***
Оба класса расположены в пакете ***django.http***.

```
from django.http import HttpResponse, HttpResponseRedirect, HttpResponsePermanentRedirect
def index(request):
    return HttpResponse("Index")
def about(request):
    return HttpResponse("About")
def contact(request):
    return HttpResponseRedirect("/about")
def details(request):
    return HttpResponsePermanentRedirect("/")
```

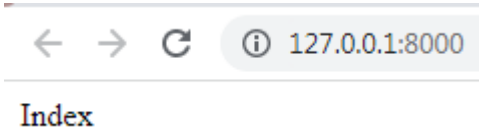
# Представления и маршрутизация

При обращении к функции ***contact*** она станет перенаправлять пользователя по пути "about", который будет обрабатываться функцией ***about***.

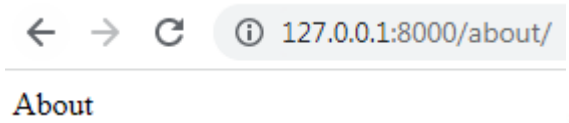
А функция ***details*** станет использовать постоянную переадресацmo и перенаправлять пользователя на «корень» (главную страницу) веб-приложения.

```
from django.urls import path
from firstapp import views
urlpatterns = [
    path('', views.index),
    path(r'about/', views.about),
    path(r'contact/', views.contact),
    path('details/', views.details),
]
```

По умолчанию будет загружена главная страница ***Index***




Теперь в адресной строке браузера наберем адрес страницы ***contact***:
***bttp://127.0.0.1:8000/contact***
При этом, поскольку мы задали переадресацию, вместо страницы ***contact*** будет загружена страница ***About***




Теперь в адресной строке браузера наберем адрес страницы ***details:***
***http:/1127.0.0.1:8000/details***
При этом, поскольку мы задали переадресацию, вместо страницы ***details*** будет загружена главная страница сайта ***Index***

# Представления и маршрутизация

В пакете django.http есть ряд классов, которые позволяют отправлять пользователю определенный статусный код

| Статусный код | Класс |
|---|---|
| 304 (Not Modified) | HttpResponseNotModified |
| 400 (Bad Request) | HttpResponseBadRequest |
| 403 (Forbidden) | HttpResponseForbidden |
| 404 (Not Found) | HttpResponseNotFound |
| 405 (Method Not Allowed) | HttpResponseNotAllowed |
| 410 (Gone) | HttpResponseGone |
| 500 (Internal Server Error) | HttpResponseServerError |

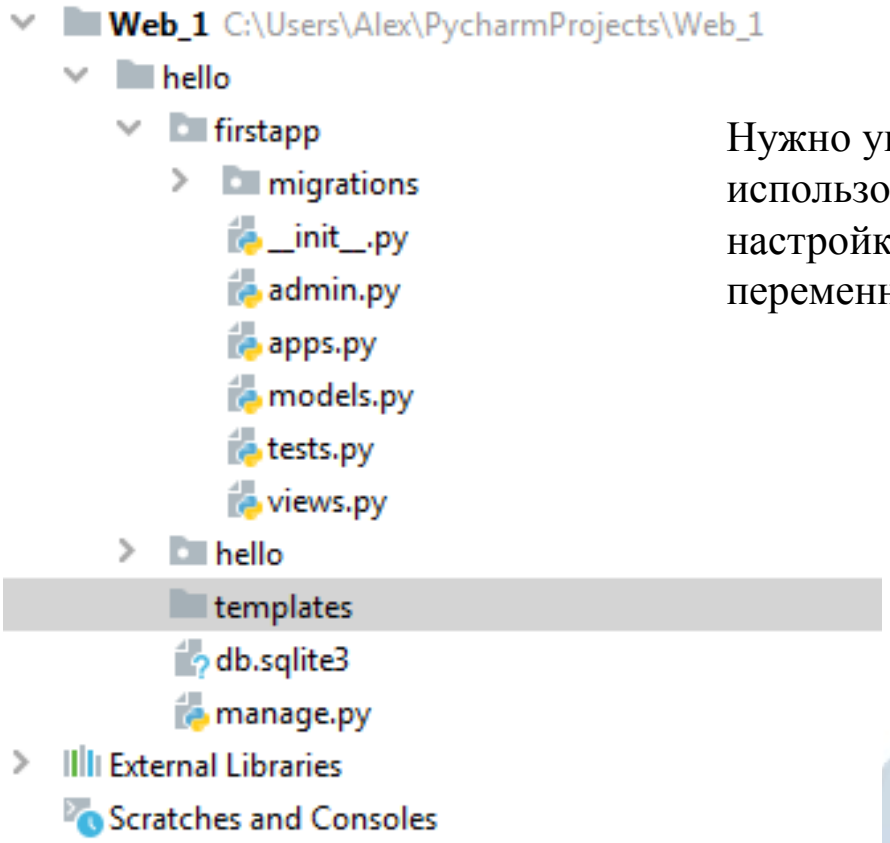
```
from django.http import *
def m304(request):
    return HttpResponseNotModified()
def m400(request):
    return HttpResponseBadRequest("<h2>Bad Request</h2>")
def m403(request):
    return HttpResponseForbidden ( "<h2>Forbidden</h2>")
def m404(request):
    return HttpResponseNotFound("<h2>Not Found</h2>")
def m405(request):
    return HttpResponseNotAllowed("<h2>Method is not allowed</h2>")
def m410(request):
    return HttpResponseGone("<h2>Content is no longer here</h2>")
def m500(request):
    return HttpResponseServerError("<h2>Something is wrong</h2>")
```

# Основы ООП

## Лекция 10. Шаблоны

# Создание шаблонов

***Шаблоны (templates)*** отвечают за формирование внешнего вида приложения. Они предоставляют специальный синтаксис, который позволяет внедрять данные в код HTML.
Для того чтобы добавить шаблоны в проект создадим новый каталог с именем ***templates*** Имя папки с шаблонами может быть любым, но, как правило, для лучшего восприятия структуры проекта этой папке все же лучше присвоить значащее имя ***templates***.




Нужно указать, что каталог templates будет использоваться в качестве хранилища шаблонов настройка шаблонов производится с помощью переменной **TEMPLATES** в файле ***settings.py***.

# Создание шаблонов

Параметр ***DIRS*** задает набор каталогов, которые хранят шаблоны. Но по умолчанию он пуст..

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

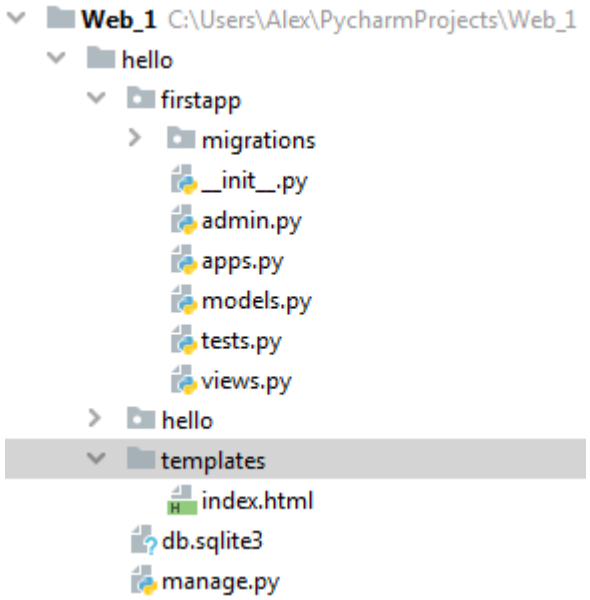
# Создание шаблонов

Изменим этот фрагмент кода следующим образом.

```
import os
from pathlib import Path
.....

TEMPLATE_DIR = os.path.join(BASE_DIR, "templates")

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR,],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

# Создание шаблонов

Создадим в папке templates новый файл index.html. Для этого в окне программной оболочки PyCharm щелкните правой кнопкой мыши на имени проекта Web_l и из появившегося меню выполните команду: ***template - File - HTML File***




```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>

</body>
</html>
```

# Создание шаблонов

Отредактируем файл index.html, это обычная веб-страница, содержащая код HTML. Теперь используем эту страницу для отправки ответа пользователю. Для этого перейдем в приложении ***firstapp*** к файлу ***views.py***, который определяет функции для обработки запроса пользователя

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Вы на главной странице Django</h1>
    <h2>templates/index.html</h2>
</body>
</html>
```

***views.py***

```python
from django.http import *
from django.shortcuts import render


def index(request):
    return render(request, "index.html")


def about(request):
    return HttpResponse("About")
```

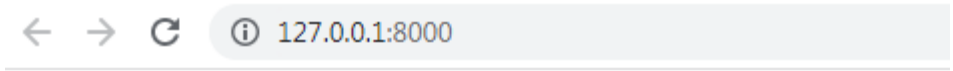
Первым шагом из модуля ***django.shortcuts*** импортируется функцию ***render*** (предоставлять). Вторым шагом изменяется функция ***def index (request)***. Теперь функция ***index(request)*** вызывает функцию render, которой передаются объект запроса пользователя request и файл шаблона ***index.html***, который находится в папке ***templates***.

# Создание шаблонов

Проверим, нужно ли вносить изменения в файл ***urls.py*** в главном проекте hello. Там должно быть прописано сопоставление функции ***index*** с запросом пользователя к «корню» веб-приложения.

```
from django.urls import path
from firstapp import views
urlpatterns = [
    path('', views.index),
    path('about/', views.about),
]
```

После всех этих изменений запустим локальный сервер разработки командой:
***python manage.py runserver***
и перейдем в браузере по адресу: bttp://127.0.0.1:8000/- браузер нам отобразит главную страницу сайта



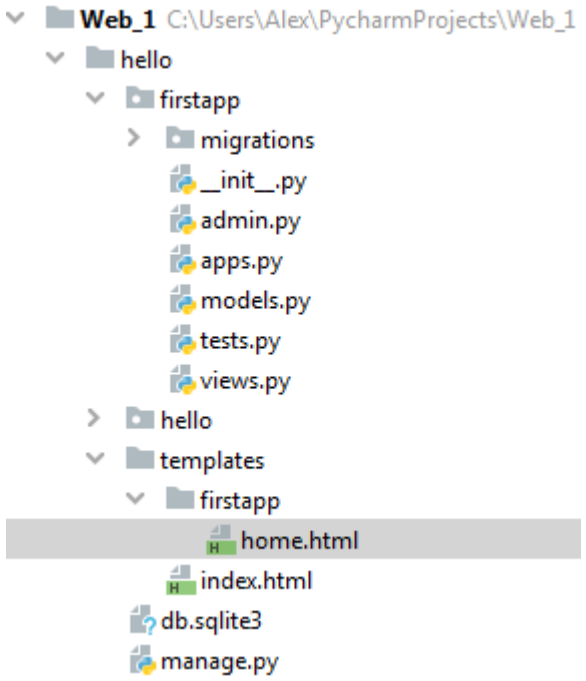

**Вы на главной странице Django**

**templates/index.html**

# Создание шаблонов

Пока он пустой, но это не беда. Теперь мы можем вносить в этот шаблон изменения, менять его дизайн, вставлять в него данные и возвращать пользователю динамически генерируемые веб-страницы.

В проекте Django нередко бывает несколько приложений. И каждое из этих приложений может иметь свой набор шаблонов. Чтобы разграничить шаблоны для отдельных проектов, можно создавать для шаблонов каждого приложения отдельный каталог. Например, в нашем случае у нас одно приложение - ***firstapp***. Создадим для него в папке ***templates*** каталог ***firstapp*** (по имени приложения). И в этом каталоге определим файл ***home.html***

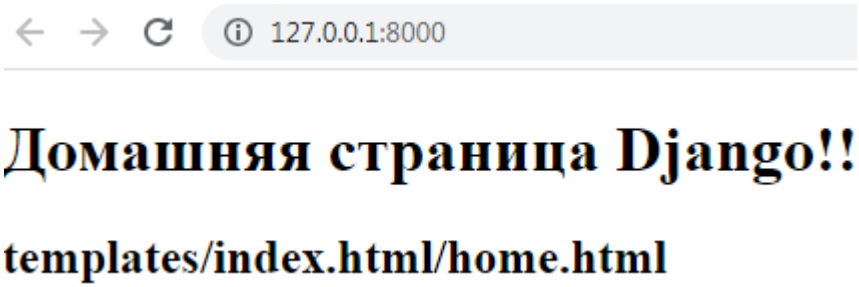



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Домашняя страница Django!!</h1>
    <h2>templates/index.html/home.html</h2>
</body>
</html>
```

# Создание шаблонов

Теперь изменим файл ***views.py*** нашего приложения, указав в нем новый путь к странице сайта, которую нужно выдать пользователю при его обращении к домашней странице сайта - ***home.html***

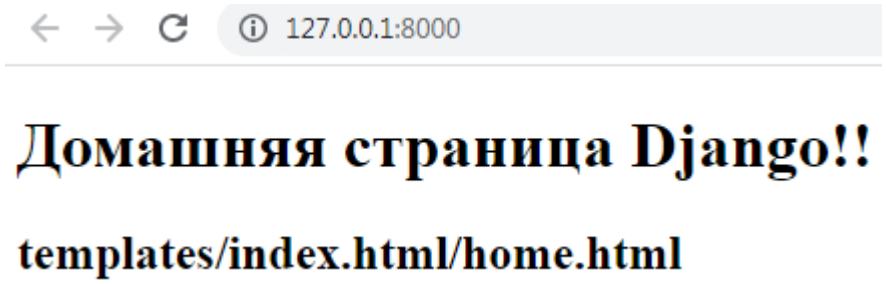
```python
from django.http import *
from django.shortcuts import render


def index(request):
    return render(request, "firstapp/home.html")


def about(request):
    return HttpResponse("About")
```



# Создание шаблонов

Ранее для загрузки (вызова) шаблона применялась функция ***render()***, что является наиболее распространенным вариантом. Однако мы также можем использовать класс ***TemplateResponse*** (шаблонный ответ). Функция ***def index (request)*** при использовании класса ***TemplateResponse*** будет выглядеть следующим образом:

```
from django.template.response import TemplateResponse

def index(request):
    return TemplateResponse(request, "firstapp/home.html")
```



# Передача данных в шаблоны

Одним из преимуществ шаблонов является то, что мы можем передать в них пользователю различные данные, которые будут динамически подгружены из базы данных через представления (***views***). Для вывода данных в шаблоне могут использоваться различные способы. Вывод самых простых данных может осуществляться с помощью двойной пары фигурных скобок:

***{{название_объекта}}***

Вернемся к нашему проекту ***hello***, содержащему приложение ***firstapp***. Добавим в папку ***templates\firstapp*** еще один шаблон страницы - ***index_app1.html***

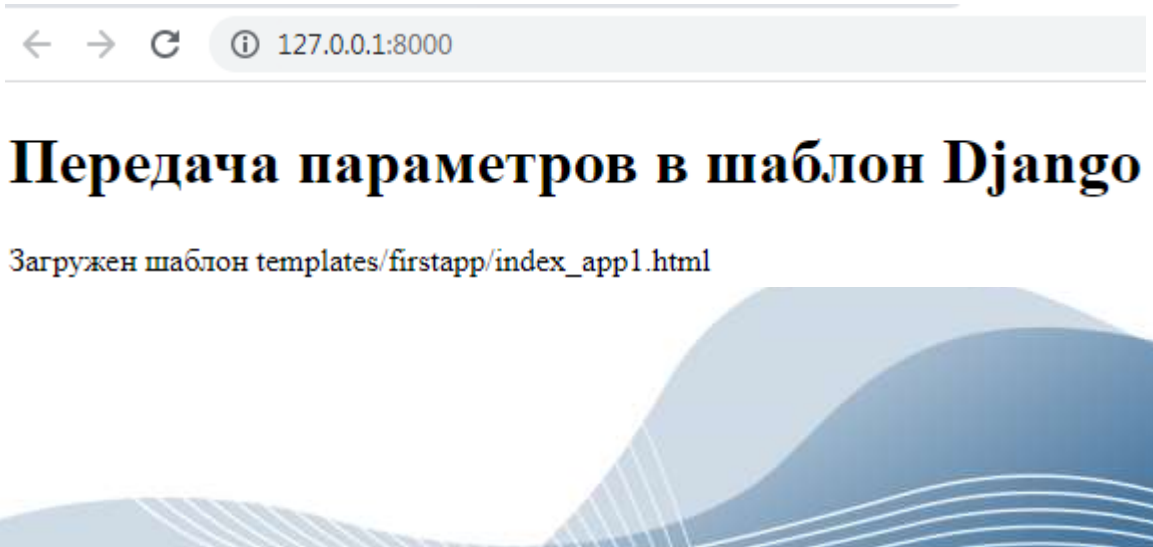
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django!</title>
</head>
<body>
    <h1>{{header}}</h1>
    <p>{{message}}</p>
</body>
</html>
```

***Введены две новые переменные: header и message. Эти переменные и будут получать значения из представления (view).***

# Передача данных в шаблоны

Чтобы из функции-представления передать данные в шаблон применяется еще один (третий) параметр в функции ***render***, который называется контекст ***(context)***.

```python
from django.http import *
from django.shortcuts import render


def index(request):
    # return render(request, "firstapp/home.html")
    data = {"header": "Передача параметров в шаблон Django",
            "message": "Загружен шаблон templates/firstapp/index_app1.html"}
    return render(request, "firstapp/index_app1.html", context=data)
```



# Передача данных в шаблоны

Рассмотрим теперь передачу пользователю через шаблон более сложных данных. Дпя этого изменим функцию ***def index ()*** в представлении (в файле ***views.py***)

```
def index(request):
    header = "Персональные данные"
    langs = ["Английский", "Немецкий", "Испанский"]
    user = {"name": "Максим,", "age": 30}
    addr = ("Виноградная", 23, 45)
    data = {"header": header, "langs": langs, "user": user, "address": addr}
    return render(request, "index.html", context=data)
```

Здесь мы создали несколько переменных: ***header, langs, user и addr***, затем все эти переменные обернули в словарь ***data***. Затем в функции ***render ()*** передали этот словарь третьему параметру ***- context***.

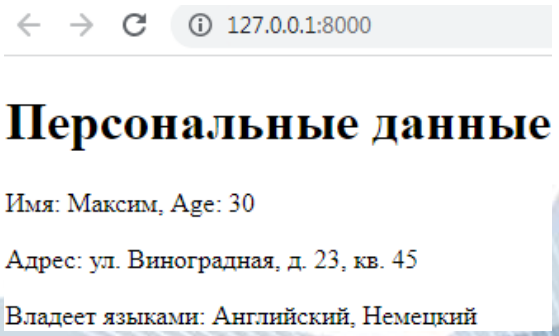
Теперь нам нужно изменить сам шаблон ***templates\index.html***, чтобы он смог принять новые данные

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Передача сложных данных</title>
</head>
<body>
    <h1>{{header}}</h1>
    <p>Имя: {{user.name}} Age: {{user.age}}</p>
    <p>Адрес: ул. {{address.0}}, д. {{address.1}}, кв. {{address.2}}</p>
    <p>Владеет языками: {{langs.0}}, {{langs.1}}</p>
</body>
</html>
```

# Передача данных в шаблоны

Поскольку объекты ***langs*** и ***address*** представляют, соответственно, список и кортеж, то мы можем обратиться к· их элементам через индексы, как мы это делали с ними в любом программном коде на Python. Например, первый элемент кортежа адреса (address) мы можем получить следующим образом: ***address.0***. Соответственно, к элементам массива, содержащего языки (***langs***) можно обращать по номеру элемента в массиве: ***langs.0***, ***langs.l***. Поскольку объект с информацией о пользователе (***user***) представляет словарь, то аналогичным образом мы можем обратиться к его элементам по ключам словаря ***name*** и ***age*** следующим образом: ***user.name, user.age***. Если в функции ***def index ()*** будет применяться класс ***TemplateResponse***, то в его конструктор также через третий параметр можно передать данные для шаблона.

```
from django.template.response import TemplateResponse
def index(request):
    header = "Персональные данные"
    langs = ["Английский", "Немецкий", "Испанский"]
    user = {"name": "Максим,", "age": 30}
    addr = ("Виноградная", 23, 45)
    data = {"header": header, "langs": langs, "user": user, "address": addr}
    return TemplateResponse(request, "index.html", data)
```



# Статические файлы

***Статическими файлами*** называются все файлы каскадных таблиц стилей (***Cascading Style Sheets, CSS***), изображений, а также скриптов ***javascript,*** - т. е. файлы, которые не изменяются динамически и содержание которых не зависит от контекста запроса и одинаково для всех пользователей. Можно воспринимать их как своего рода «макияж» для веб-страниц.

Для придания привлекательности информации, выводимой на HTML-страницах, используются различные стили форматирования текстов, оформленные в виде каскадных таблиц стилей (CSS) с помощью их специального языка. Такой подход обеспечивает возможность прикреплять стиль (тип шрифта, его размер цвет и пр.) к структурированным документам. Обычно CSS-стили используются для создания и изменения стиля элементов веб-страниц и пользовательских интерфейсов, написанных на языках HTML, но также могут быть применены к любому виду XML-документа. Отделяя стиль представления документов от содержимого документов, CSS упрощает создание веб-страниц и обслуживание сайтов.

# Статические файлы

Объявление стиля состоит из двух частей: ***селектора и объявления***. В HTML имена элементов нечувствительны к регистру, поэтому в селекторе значение ***hl*** работает так же, как и ***H1***. Объявление состоит из двух частей: имени свойства (например, ***color***) и значения свойства (например, ***grey***). ***Селектор*** сообщает браузеру, какой именно элемент форматировать, а в ***блоке объявления*** (код в фигурных скобках) указываются форматирующие команды - свойства и их значения


***Стили могут быть следующих видов:***

- ***встроенные;***
- ***внутренние;***
- ***внешняя таблица стилей.***

Внутренние стили имеют приоритет над внешними таблицами стилей, но уступают встроенным стилям, заданным через атрибут ***style***. При использовании встроенных стилей CSS-код располагается в HTML-файле, непосредственно внутри тега элемента с помощью атрибута ***style***

```
<p style="font-weight: bold; color: red;">Этот текст будет красного цвета!</p>
```

# Статические файлы

Изменим стиль выводимого текста: первую строку выведем красным цветом, а вторую - синим. Для этого изменим текст файла home.html следующим образом

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <!-- Это встроенный стиль -->
    <font style="color: red">
    <h1>Домашняя страница Django!</h1>
    <font style="color: blue; font-size: 12px">
    <h2>templates/firstapp/home.html</h2>
</body>
</html>
```

**Домашняя страница Django!**

**templates/firstapp/home.html**

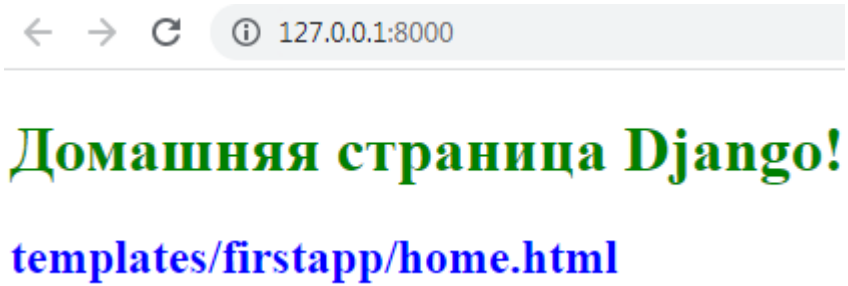
Использовать встроенные стили достаточно просто, но не совсем удобно. Если через какое-то время потребуется изменить цвет всех заголовков на всех страницах, то нужно будет менять код на каждой из страниц

# Статические файлы

Внутренние стили отличаются от встроенных стилей тем, что они встраиваются в раздел **<head> … </head>** HTML-документа.

В теге **<head>** … **</head>** для текста в теге **h1** задали зеленый цвет, но при этом в теге **<body>** ... **</ body >** оставили в стиле для этого текста красный цвет

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- Это внутренний стиль -->
    <style>h1{color: green;}</style>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <!-- Это встроенный стиль -->
    <font style="color: red; font-size: 12px">
    <h1>Домашняя страница Django!</h1>
    <font style="color: blue; font-size: 12px">
    <h2>templates/firstapp/home.html</h2>
</body>
</html>
```

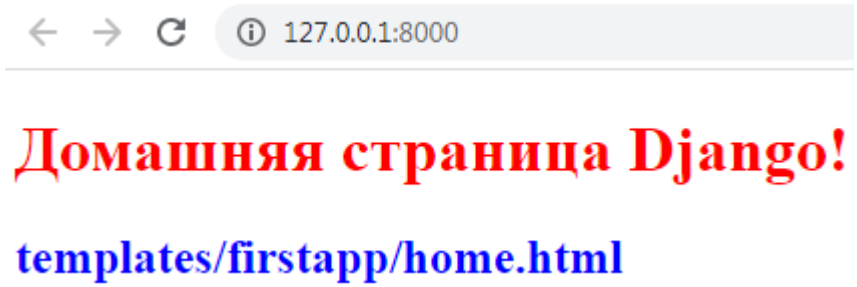



Несмотря на то что перед тегом ***hl*** во встроенном стиле задан красный цвет текста, выведен он в зеленом. То есть здесь задействован тот стиль (внутренний), который был указан в заголовке страницы.

# Статические файлы

Здесь в заголовке страницы для тега ***h1*** задан зеленый цвет текста (внутренний стиль), а в теле страницы для тега ***h1*** - красный (встроенный стиль). Поскольку встроенный стиль имеет приоритет над внутренним стилем, то текст заголовка должен быть выведен красным цветом. Как можно видеть (при выводе цветного изображения на экран компьютера), заголовок страницы имеет красный цвет, что подтверждает приоритет встроенного стиля над внутренним.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <!-- Это внутренний стиль -->
    <style>h1{color: green;}</style>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <!-- Это встроенный стиль-->
    <style>h1{color: red;}</style>
    <h1>Домашняя страница Django!</h1>
    <font style="color: blue; font-size: 12px">
    <h2>templates/firstapp/home.html</h2>
</body>
</html>
```

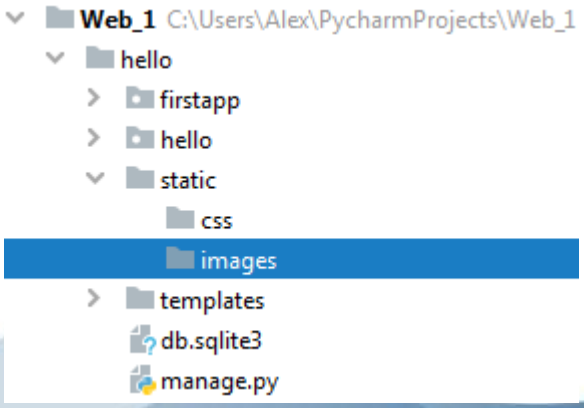


# Статические файлы

Внешняя таблица стилей представляет собой текстовый файл с расширением **css**. В этом файле находится набор CSS-стилей для различных элементов HTML-страниц.
Файл каскадных страниц стилей создается в редакторе кода, так же как и HTML-страница. Однако внутри файла содержатся только стили, без HTML-разметки. Внешняя таблица стилей подключается к веб-странице с помощью тега **<link>**, расположенного внутри раздела **<head>** ... **</head>**

```
<head>
<link rel="stylesheet" href="css/style.css">
</head>
```

В веб-приложениях, как правило, присутствуют различные статические файлы – это изображения, файлы каскадных таблиц стилей (CSS), скрипты javascript и т. п. Рассмотрим, как мы можем использовать подобные файлы в веб-приложениях. Добавим в корневую папку проекта новую вложенную папку ***static***. А чтобы не смешивать в одной папке различные типы файлов, создадим для каждого типа файлов отдельную папку. В частности, создадим в папке static папку images - для изображений, и папку css - для таблиц стилей



# Статические файлы

Создадим в папке для таблиц стилей ***css*** файл ***styles.css***. Для этого в окне программной оболочки PyCharm щелкните правой кнопкой мыши на папке css и из появившегося меню выполните команду New – File.

Теперь используем этот файл в шаблоне. Для этого в начале файла шаблона необходимо определить инструкцию (для Django 3):

***{% load static %}***

Для определения пути к статическим файлам используются выражения такого типа:

***{% static "путь к файлу внутри папки static" %}***

Возьмем шаблон ***home.html*** из папки ***templates\firstapp*** и изменим его следующим образом:

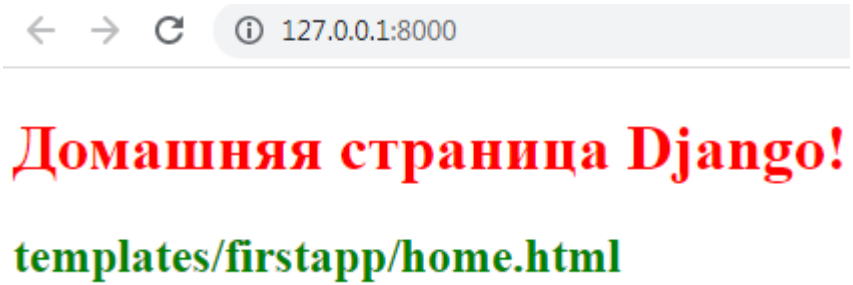
```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
    <link href="{% static 'css/styles.css' %}" rel="stylesheet">
</head>
<body>
    <h1>Домашняя страница Django!</h1>
    <h2>templates/firstapp/home.html</h2>
</body>
</html>
```

# Статические файлы

Чтобы файлы из папки ***static*** могли использоваться в приложениях, надо указать путь к этой папке в файле ***settings.py***, добавив в конец файла следующий код

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [os.path.join(BASE_DIR, "static"), ]
```

После этих изменений обратимся к странице ***home.html*** и получим следующий результат




Как можно здесь видеть (при выводе цветного изображения на экран компьютера), в соответствии со стилями, указанными в файле ***styles.css***, текст тега ***h1*** выделен красным цветом, а текст тега ***h2*** - зеленым. При этом в самом HTML-файле стили для выделения текста каким-либо цветом не указаны, а лишь сделаны ссылки на файл со стилями. Такой подход очень удобен тем, что можно оперативно менять и настраивать стили на десятках станиц сайта, внося изменения в код всего одного файла.

# Статические файлы

Изображения тоже являются статическими файлами. Для хранения изображений мы ранее создали папку ***images***. Разместим в этой папке файл с любым изображением и дадим ему имя ***image1.jpg***. Теперь выведем его на нашей странице ***home.html***, для чего модифицируем код страницы следующим образом.

```
<!DOCTYPE html>
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
    <link href="{% static 'css/styles.css' %}" rel="stylesheet">
</head>
<body>
    <h1>Домашняя страница Django!</h1>
    <h2>templates/firstapp/home.html</h2>
    <img src="{% static 'images/image1.jpg' %}">
</body>
</html>
```

# Статические файлы

Как можно видеть, изображение из папки ***static\images\image1.jpg*** успешно выведено на HTML-странице. Поскольку для изображений в файле slyles.css не было указано стиля вывода, то оно отображается на странице в том виде, в каком было сохранено в файле ***image1.jpg***. Однако для выводимых на HTML-страницах рисунков в файле ***styles.css*** тоже можно указывать стиль отображения. Внесем в файл ***styles.css*** следующие изменения

```
/* static/css/styles.css */
body h1 {color: red;}
body h2 {color: green;}
img{width:250px;}
```

Мы указали, что ширина изображения должна быть 250 пикселов. После такого изменения страница home.html примет вид

**Домашняя страница Django!**

**templates/firstapp/home.html**

# Вызов шаблона

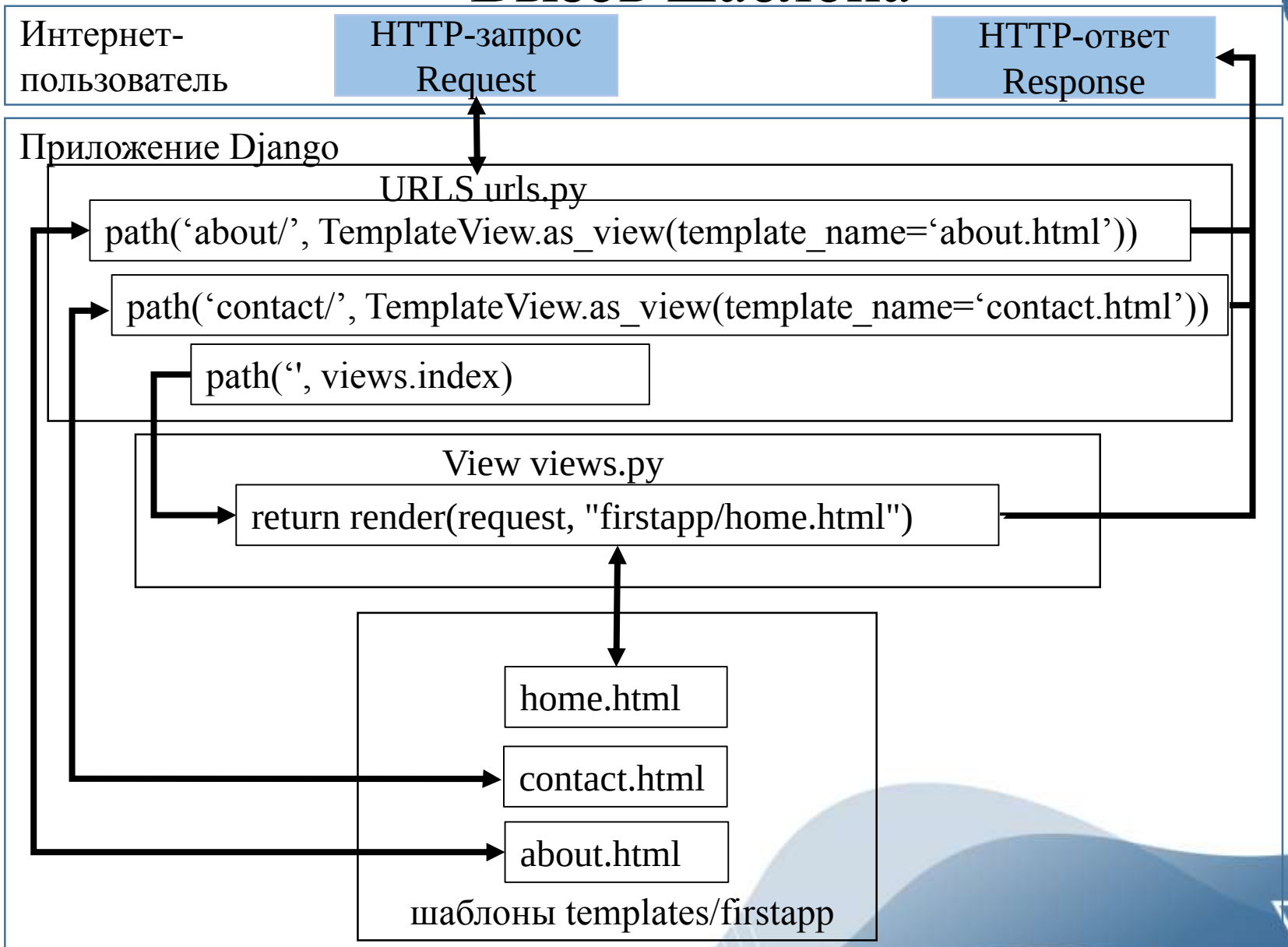
В примерах выше, когда приходил запрос от браузера пользователя, система маршрутизации выбирала нужное представление (***view***), и уже оно вызывало шаблон для генерации ответа. Кроме того, при необходимости, представление могло обратиться к БД и вставить в шаблон некоторые данные из нее. Однако, если нужно просто возвратить пользователю содержимое шаблона, то для этого необязательно определять функцию в представлении и обращаться к ней. Можно воспользоваться встроенным классом ***TemplateView*** и вызвать нужный шаблон, минуя обращение к представлению.

На схеме представлен вариант приложения, в котором шаблон главной страницы сайта home.html вызывается функцией из представления ***view***, а шаблоны страниц ***about*** и ***contact*** вызываются с использованием класса ***TemplateView***.

Проверим, как это работает на простых примерах, для чего определим несколько простейших шаблонов в папке ***hello\templates\firstapp***. Пусть это будет файл-шаблон ***about.html*** со следующим кодом

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Сведения о компании</h1>
    <h2>"Интеллектуальные программные системы"</h2>
    <h3>templates/firstapp/about.html</h3>
</body>
</html>
```

# Вызов шаблона



# Вызов шаблона

В этой же папке создадим файл-шаблон ***contact.html*** со следующим кодом

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Контактные данные компании</h1>
    <h2>"Интеллектуальные программные системы"</h2>
    <h3>Адрес: г. Москва, ул. Короленко, д. 24</h3>
    <h4>templates/firstapp/contact.html</h4>
</body>
</html>
```

Осталось внести в файл ***urls.py*** следующие изменения

```
from django.urls import path
from firstapp import views
from django.views.generic import TemplateView
urlpatterns = [
    path('', views.index),
    path ( 'about/', TemplateView.as_view (template_name="firstapp/about.html")),
    path('contact/',TemplateView.as_view(template_name="firstapp/contact.html")),
]
```

# Вызов шаблона

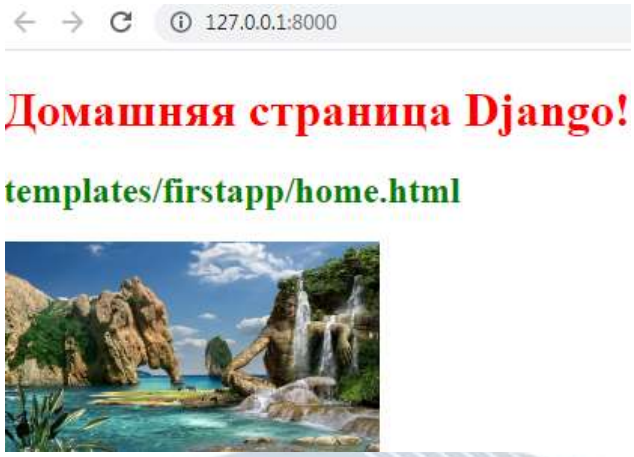
В рассматриваемом программном коде сначала для вызова главной страницы сайта делается обращение к функции ***index()*** в представлении ***view***, и уже функция ***index ()*** вызывает главную страницу приложения ***firstapp/home.html***

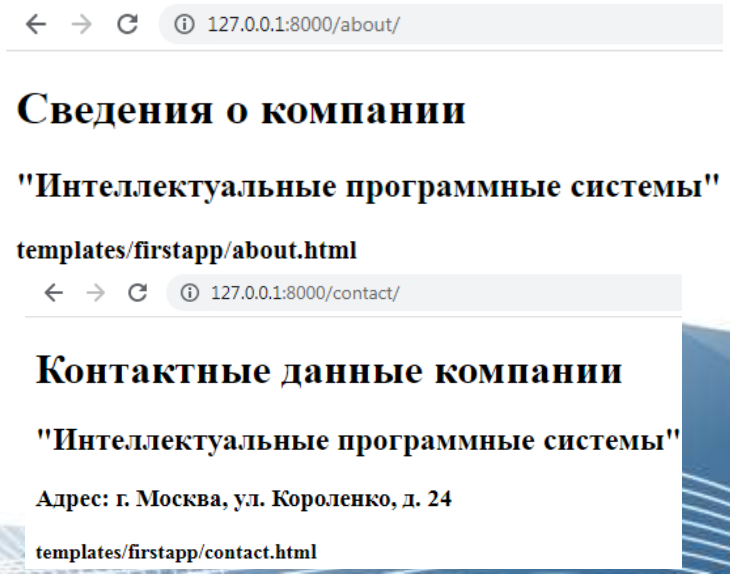
```
def index(request):
    return render(request, "firstapp/home.html")
```

Затем в двух строках программного кода в ***urls.py*** для вызова страниц ***firstapp\about.html*** и ***firstapp\contact.html*** используется класс ***TemplateView:***

```
path ( 'about/', TemplateView.as_view (template_name="firstapp/about.html")),
path('contact/',TemplateView.as_view(template_name="firstapp/contact.html")),
```

По своей сути класс ***TemplateView*** предоставляет функциональность представления. С помощью метода ***as_view()*** через параметр ***template_name*** задается путь к шаблону, который и будет использоваться в качестве ответа.



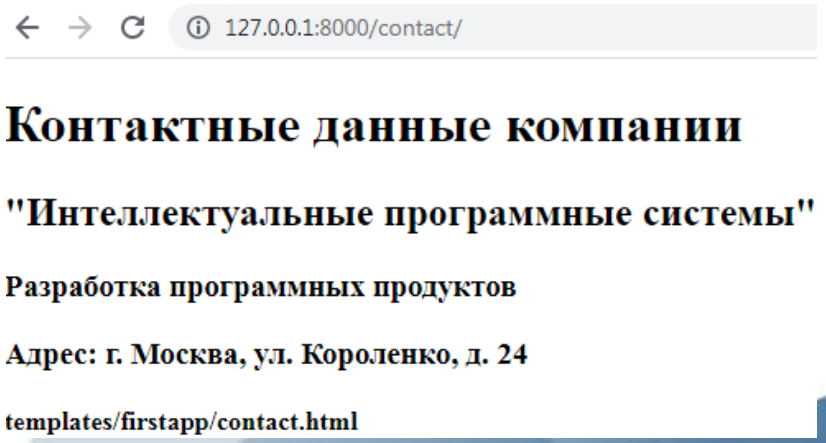
# Вызов шаблона

С помощью параметра ***extra_context*** в метод ***as_view*** можно передать данные для их отображения в шаблоне. Данные должны представлять собой словарь. В качестве примера передадим в шаблон ***contact.html*** виды работ, которые выполняет компания. Для этого используем переменную ***work***.

```
urlpatterns = [
    path('', views.index),
    path ('about/', TemplateView.as_view (template_name="firstapp/about.html")),
    path('contact/',TemplateView.as_view(template_name="firstapp/contact.html",
                                         extra_context={"work":"Разработка программных продуктов"})),
]
```

Здесь в шаблон ***contact.html*** передается объект work, который представляет строку со следующей информацией: "Разработка программных продуктов". И теперь мы можем использовать этот объект в шаблоне ***contact.html***

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Привет Django</title>
</head>
<body>
    <h1>Контактные данные компании</h1>
    <h2>"Интеллектуальные программные системы"</h2>
    <h3>{{work}}</h3>
    <h3>Адрес: г. Москва, ул. Короленко, д. 24</h3>
    <h4>templates/firstapp/contact.html</h4>
</body>
</html>
```



# . Конфигурация шаблонов HTML-страниц

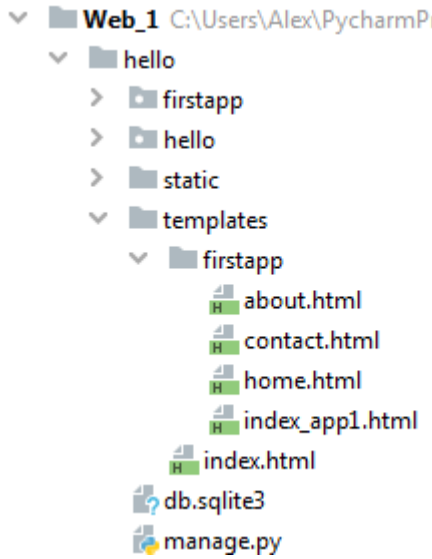
За конфигурацию шаблонов проекта отвечает переменная ***TEMPLATES***, расположенная в файле ***settings.py***

```
TEMPLATE_DIR = os.path.join(BASE_DIR, "templates")

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [TEMPLATE_DIR,],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```

***BACKEND*** - указывает, что надо использовать шаблоны Django;
***DIRS*** - указывает на каталоги (папки) в проекте, которые будут содержать шаблоны;
***APP_DIRS*** - при значении true указывает, что поиск шаблонов будет производиться не только в каталогах (папках), указанных в параметре DIRS, но и в их подкаталогах (подпапках). Если такое поведение недопустимо, то можно установить значение False;
***OPTIONS*** - указывает, какие обработчики (процессоры) будут использоваться при обработке шаблонов.

# . Конфигурация шаблонов HTML-страниц

Как правило, шаблоны помещаются в общую папку в проекте либо в ее подпапки. Например, вы можете определить в проекте общую папку ***templates***. Однако если в проекте имеется несколько приложений, которые должны использовать какие-то свои шаблоны, то, чтобы избежать проблем с именованием, можно создать для каждого приложения свою подпапку, в которую помещаются шаблоны для конкретного приложения. Например, в проекте ***hello*** у нас создано приложение ***firstapp***. В этом случае в папке ***templates*** можно создать подпапку ***firstapp*** и уже в ней хранить все шаблоны, которые относятся к приложению ***firstapp.***




В этом случае берется определенная в самом начале файла settings.py переменная BASE_DIR

```
BASE_DIR = Path(__file__).resolve().parent.parent
```

# . Базовый шаблон

Хорошим тоном программирования считается формирование единообразного стиля сайта, когда веб-страницы имеют одни и те же структурные элементы: меню, шапку сайта (***header***), подвал (***footer***), боковую панель (***sidebar***) и т. д. Конечно, можно сконфигурировать каждый шаблон по отдельности. Однако если возникнет необходимость изменить какой-то блок - например, добавить в общее меню еще один пункт, то придется менять все шаблоны, которых может быть довольно много. И в этом случае более рационально многократно использовать один базовый шаблон, который определяет все основные блоки страниц сайта. Попросту говоря, нужно создать некий базовый шаблон. Тогда все остальные шаблоны будут иметь одинаковую базовую структуру и одни и те же блоки, при этом для отдельных блоков можно формировать различное содержимое. В качестве примера создадим базовый шаблон, который назовем ***base.html***

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>{% block title %}Заголовок{% endblock title %}</title>
</head>
<body>
    <h1>{% block header %}{% endblock header %}</h1>
    <div>{% block content%}{% endblock content %}</div>
    <div>Подвал страницы</div>
</body>
</html>
```

# . Базовый шаблон

Здесь с помощью элементов: ***{% block название_блока %} {% endblock название_блока %}*** определяются отдельные блоки шаблонов.
При этом для каждого блока определяется открывающий элемент:
***{ % block название блока %}***
и закрывающий элемент:
***{% endblock название_блока %}***
Например, блок title имеет такую структуру:
***{% block title %}Заголовок{% endblock title %}***

Когда другие шаблоны будут применять этот базовый шаблон, то они могут определить для блока title какое-то свое содержимое. Подобным же образом здесь определены блоки header и content. Для каждого блока можно определить содержимое по умолчанию. Так, для блока title это строка заголовок. И если другие шаблоны, которые станут использовать этот шаблон, не определят содержимое для блока title, то этот блок будет использовать строку заголовок. Впрочем, содержимое по умолчанию для блоков определять не обязательно. Самих же блоков при необходимости можно определить сколько угодно. Как можно видеть, в базовом шаблоне также определен подвал страницы (footer). Поскольку мы хотим сделать подвал общим для всех страниц, то мы его не определяем как отдельный блок, а задаем значение этой части страницы в виде некой константы.

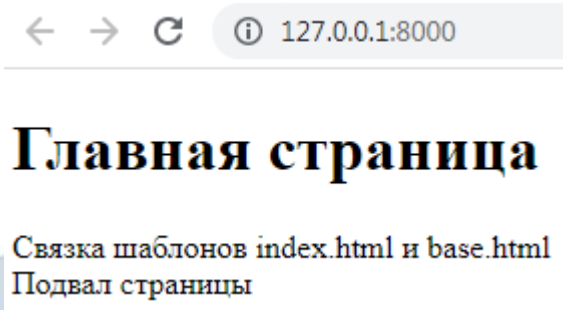
# . Базовый шаблон

Теперь применим наш базовый шаблон. Создадим в каталоге ***templates\firstapp*** новый файл-шаблон главной страницы сайта с именем ***index.html*** и напишем в нем следующий код

```
{% extends "firstapp/base.html" %}
{% block title %}Index{% endblock title %}
{% block header %}Главная страница{% endblock header %}
{% block content %}Связка шаблонов index.html и base.html{% endblock content %}
```

Здесь с помощью выражения ***{% extends "firstapp/base.html" %}*** мы определили, что эта страница будет формироваться (расширяться) на основе базового шаблона с именем ***base.html***, который находится в каталоге шаблонов приложения ***firstapp*** (по пути: ***firstapp\base.html***). Затем задали содержимое для блоков ***title, header и content***. Эти значения и будут переданы в базовый шаблон ***(base.html***). Стоит отметить, что не обязательно указывать содержимое для всех блоков базового шаблона.

```
def index(request):
    return render(request, "firstapp/index.html")
```

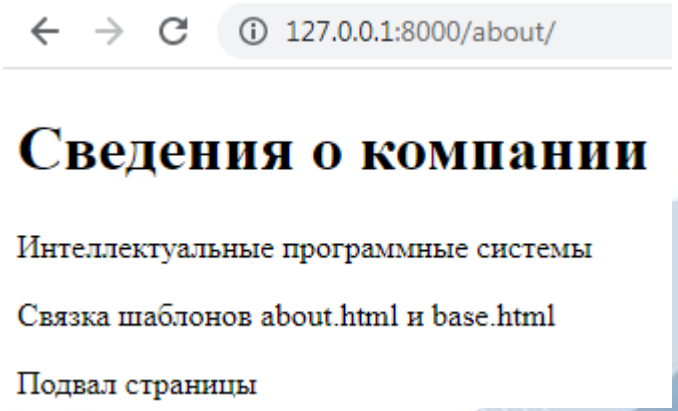


# . Базовый шаблон

Используем этот же базовый шаблон base.html для формирования другой страницы сайта - ***about.html***.

```
{% extends "firstapp/base.html" %}
{% block title %}about{% endblock title %}
{% block header %}Сведения о компании{% endblock header %}
{% block content %}
<p>Интеллектуальные программные системы</p>
<p>Связка шаблонов about.html и base.html</p>
{% endblock content %}
```

Здесь с помощью выражения ***{% extends "firstapp/base.html" %}*** определено, что эта страница будет формироваться (расширяться) на основе базового шаблона с именем ***base.html***, который находится в каталоге шаблонов приложения ***firstapp*** (по пути: ***firstapp\base.html***). Затем задали содержимое для блоков ***title, header и content***. Эти значения и будут переданы в базовый шаблон (***base.html***).



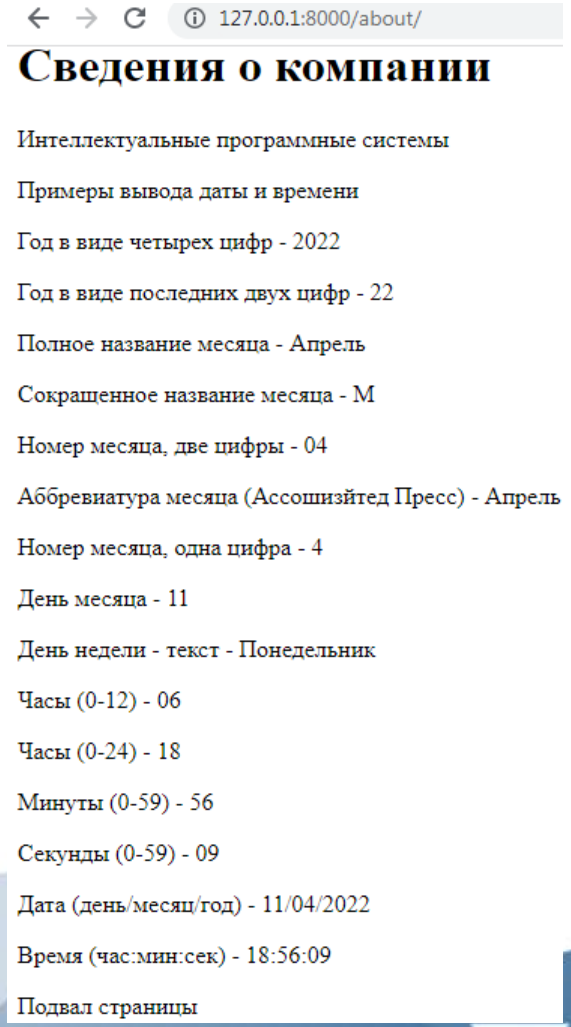
# . Специальные теги в шаблонах

В Django предоставлена возможность использовать в шаблонах ряд специальных тегов, которые упрощают вывод некоторых данных.
Для вывода дат в Django используется следующий тег:
***{% now "формат_данных" %}***
Тег ***now*** позволяет вывести системное время. В качестве параметра тегу ***now*** передается формат данных, который указывает, как форматировать время и дату. Для форматирования времени и дат используются следующие символы
**Y** Год в виде четырех цифр (2021)
**y** Год в виде последних двух цифр (21)
**F** Полное название месяца (Июль)
**M** Сокращенное название месяца - 3 символа (Июл)
**m** Номер месяца - две цифры (07)
**N** .Аббревиатура месяца в стиле Ассошиэйтед Пресс
**n** Номер месяца -одна цифра (7)
**j** День месяца (1-31)
**1** День недели - текст (среда)
**h** Часы (0-12)-9:15
**H** Часы (0-24)-21:15
**i** Минуты (0-59)
**s** Секунды (0-59)

# . Специальные теги в шаблонах

Изменим следующим образом код шаблона страницы ***about.html***

```
{% extends "firstapp/base.html" %}
{% block title %}about{% endblock title %}
{% block header %}Сведения о компании{% endblock header %}
{% block content %}
<p>Интеллектуальные программные системы</p>
<p>Примеры вывода даты и времени</p>
<p>Год в виде четырех цифр - {% now "Y" %}</p>
<p>Год в виде последних двух цифр - {% now "y" %}</p>
<p>Полное название месяца - {% now "F" %}</p>
<p>Сокращенное название месяца - {% now "M" %}</p>
<p>Номер месяца, две цифры - {% now "m" %}</p>
<p>Аббревиатура месяца (Ассошизйтед Пресс) - {% now "N" %}</p>
<p>Номер месяца, одна цифра - {% now "n" %}</p>
<p>День месяца - {% now "j" %}</p>
<p> День недели - текст - {% now "1" %}</p>
<p>Часы (0-12) - {% now "h" %}</p>
<p>Часы (0-24) - {% now "H" %}</p>
<p>Минуты (0-59) - {% now "i" %}</p>
<p>Секунды (0-59) - {% now "s" %}</p>
<p>Дата (день/месяц/год) - {% now "j/m/Y" %}</p>
<p>Время (час:мин:сек) - {% now "H:i:s" %}</p>
{% endblock content %}
```



# . Специальные теги в шаблонах

Тег для вывода информации в зависимости от соблюдения какого-либо условия выглядит следующим образом:

***{% if %} {% endif %}***

В качестве параметра тегу ***if*** передается выражение, которое должно возвращать ***True*** или ***False.***

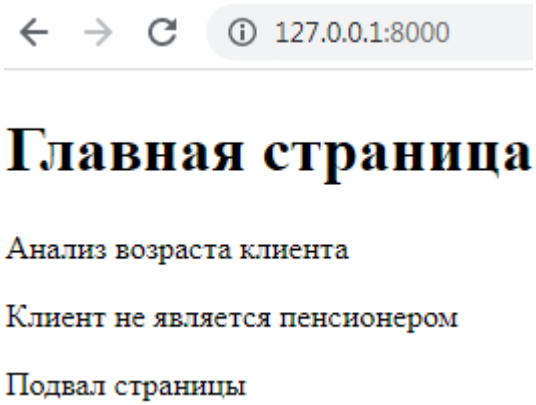
Предположим, что из представления ***view*** в шаблон передаются некоторые значения - например, возраст клиента (***age***). Изменим следующим образом текст функции ***def index ()*** в файле ***view.py***

```
from django.http import *
from django.shortcuts import render
def index(request):
    data = {"age": 50}
    return render(request, "firstapp/index.html", context=data)
```

# . Специальные теги в шаблонах

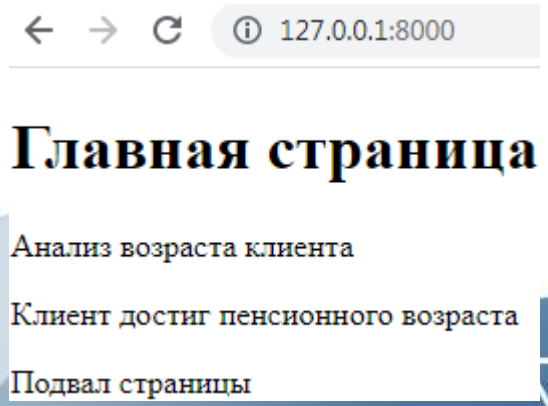
В шаблоне страницы ***firstapp\index.html*** в зависимости от значения переменной ***age*** мы можем выводить разную информацию. Изменим следующим образом текст в файле ***firstapp\index.html***

```
{% extends "firstapp/base.html" %}
{% block title %}Index{% endblock title %}
{% block header %}Главная страница{% endblock header %}
{% block content %}
<p>Анализ возраста клиента</p>
{% if age > 65 %)
    <p>Клиент достиг пенсионного возраста</p>
{% else %)
    <p>Клиент не является пенсионером</p>
{% endif % )
{% endblock content %}
```




Теперь изменим следующим образом текст функции ***def index ()*** в файле ***view.py*** - укажем возраст клиента 66 лет:

***data = {"age" : 66}***



# . Специальные теги в шаблонах

Тег **for** позволяет создавать циклы. Этот тег принимает в качестве параметра некоторую коллекцию и пробегается по этой коллекции, обрабатывая каждый ее элемент. Тег имеет следующую структуру:

***{% for "Индекс элемента" in "Коллекция элементов" %}***

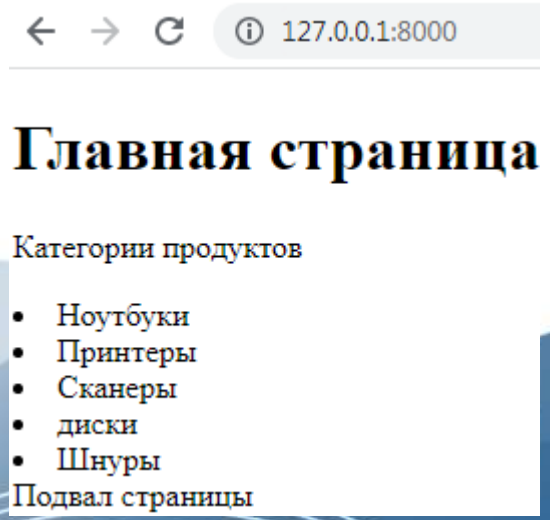
***{% endfor %}***

Предположим, что из представления **view** в шаблон передается массив значений - например, категории товаров (***cat***). Изменим следующим образом текст функции ***def index ()*** в файле ***view.py***

```
from django.http import *
from django.shortcuts import render
def index(request):
    cat = ["Ноутбуки", "Принтеры", "Сканеры", "диски", "Шнуры"]
    return render(request, "firstapp/index.html", context={"cat": cat})
```

Чтобы в шаблоне страницы ***firstapp\index.html*** в цикле выводить информацию из списка cat, изменим следующим образом код в файле ***firstapp\index.html***

```
{% extends "firstapp/base.html" %}
{% block title %}Index{% endblock title %}
{% block header %}Главная страница{% endblock header %}
{% block content %}
<p>Категории продуктов</p>
{% for i in cat %}
    <li>{{ i }}</li>
{% endfor %}
{% endblock content %}
```



# . Специальные теги в шаблонах

Вполне возможно, что массив, переданный из представления ***view*** в шаблон, окажется пустым. На этот случай мы можем использовать дополнительный тег:

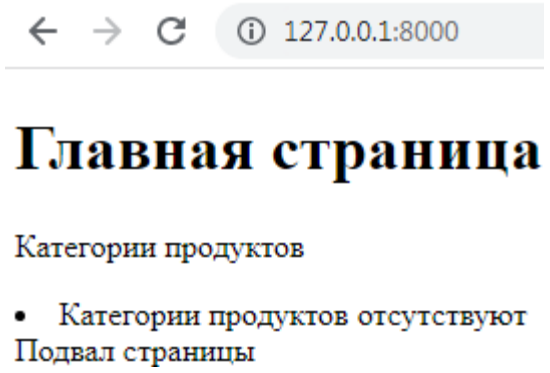
***{% empty %}***

Для этого блок ***content*** на странице ***firstapp\index.html*** можно изменить следующим образом

```
{% extends "firstapp/base.html" %}
{% block title %}Index{% endblock title %}
{% block header %}Главная страница{% endblock header %}
{% block content %}
<p>Категории продуктов</p>
{% for i in cat %}
    <li>{{ i }}</li>
{% empty %}
    <li>Категории продуктов отсутствуют</li>
{% endfor %}
{% endblock content %}
```

Теперь в файле ***view.py*** сделаем список «пустым»:

cat = []



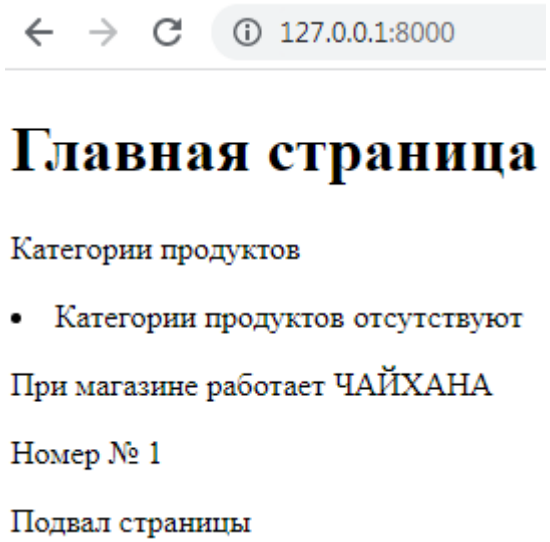
# . Специальные теги в шаблонах

Если требуется определить переменную и использовать ее внутри шаблона, то для этого можно использовать тег:

***{% with %}***

Изменим следующим образом код в файле ***firstapp\index.html***

```
{% extends "firstapp/base.html" %}
{% block title %}Index{% endblock title %}
{% block header %}Главная страница{% endblock header %}
{% block content %}
<p>Категории продуктов</p>
{% for i in cat %}
    <li>{{ i }}</li>
{% empty %}
    <li>Категории продуктов отсутствуют</li>
{% endfor %}
{% with name="ЧАЙХАНА" num=1 %}
    <div>
        <p>При магазине работает {{ name }}</p>
        <p>Номер № { { nom } }</p>
    </div>
{% endwith %}

{% endblock content %}
```




Здесь внутри шаблона мы определили и вывели значения двух переменных: name и num.

# Основы ООП

## Лекция 11. Формы.

# Формы

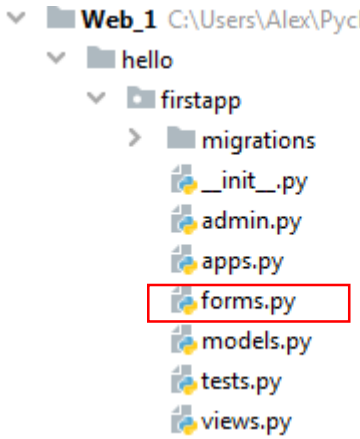
***Форма HTML-*** это группа из одного или нескольких полей (виджетов) на вебстранице, которая используется для получения информации от пользователей для последующей отправки на сервер и представляет собой таким образом гибкий механизм сбора пользовательских данных. Формы несут целый набор виджетов для ввода различных типов данных: текстовые поля, флажки, переключатели, установщики дат и пр. Формы являются относительно безопасным способом взаимодействия пользовательского клиента и сервера, поскольку позволяют отправлять данные в ***POST-запросах***, применяя защиту от межсайтовой подделки запроса ***(Cross Site Request Forgery, CSRF)***.

Django предоставляет различные возможности по работе с формами. Можно определить функциональность форм в одном месте и затем использовать их многократно в разных местах. При этом упрощается проверка корректности данных, связывание форм с моделями данных и многое другое.

Каждая форма определяется в виде отдельного класса, который расширяет класс ***forms.Form***. Классы размещаются внутри проекта, где они используются. Нередко они помещаются в отдельный файл, который называется, к примеру, ***forms.py***. Однако формы могут размещаться и внутри уже имеющихся в приложении файлов - например, в ***views.py*** или ***models.py***.

# Формы

Создадим в приложении ***firstapp*** проекта ***hello*** новый файл ***forms.py.*** Здесь класс формы называется ***userForm***. В нем определены два поля. Поле name (имя) имеет тип ***forms.CharField*** и будет генерировать текстовое поле: ***input type="text"***. Поле age (возраст) имеет тип ***forms.IntegerField*** и будет генерировать числовое поле: ***input type="number"***. То есть первое поле предназначено для ввода текста, а второе - для ввода чисел.




```
from django import forms


class UserForm(forms.Form):
    name = forms.CharField()
    age = forms.IntegerField()
```

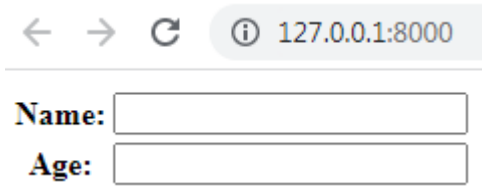
В файле ***views.py*** определим следующий код для функции ***index ()***

```
from django.shortcuts import render
from .forms import UserForm


def index(request):
    userform = UserForm()
    return render(request, "firstapp/index.html", {"form": userform})
```

# Формы

Теперь изменим шаблон ***index.html*** следующим образом.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
    <body>
        <table>
            {{form}}
        </table>
</body>
</html>
```

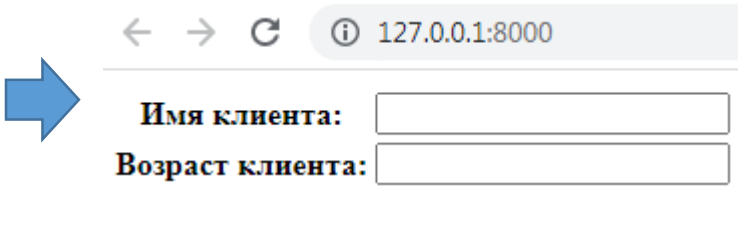



В обрамленные рамками поля пользователь может вводить свои данные. Следует заметить, что в модуле ***forms.py*** мы задали только идентификаторы полей name и age и не задавали метки полей (label), которые будут выводиться на экран. В Django эти метки генерируются автоматически. В частности, как можно видеть, для наших двух полей были сгенерированы метки: Name и Age. По умолчанию в Django в качестве метки берется имя поля, и его первый символ меняется на заглавную букву. С одной стороны, такой прием избавляет программиста от дополнительного задания значения для метки поля, но это не всегда удобно. Часто требуется задавать более осмысленные и достаточно длинные значения для меток полей и на языке, отличном от английского.

# Формы

Однако если мы в PyCharm попытаемся задать имя поля кириллицей на русском языке, то получим предупреждение об ошибке. Впрочем, несмотря на это предупреждение, программа будет выполняться, поскольку Django допускает использование в именах полей символов, отличных от ASCII.

```
from django import forms


class UserForm(forms.Form):
    имя_клиента = forms.CharField()
    возраст_клиента = forms.IntegerField()
```



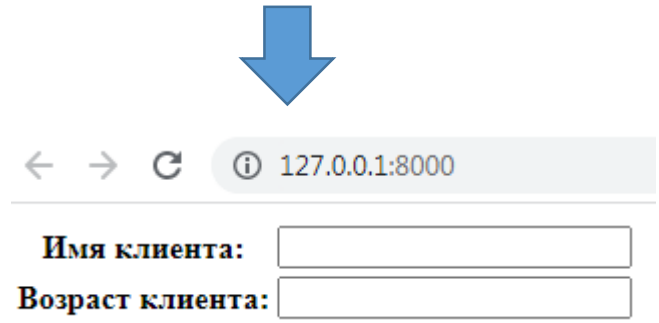

Обратите внимание: если имя поля в Django состоит из нескольких слов, то их нельзя разделять пробелами, а нужно использовать символ нижнего подчеркивания - например, имя_клиента, возраст_клиента и т. п. Django не только меняет в кириллической записи первый символ на заглавную букву, но и еще автоматически заменяет символ нижнего подчеркивания на пробел.

Однако желательно избегать использования кириллицы и других символов, отличных от ASCII, в определении имен полей. Это впоследствии может привести к некорректному отображению информации, когда приложение будет развернуто на внешнем веб-ресурсе.

# Формы

В Django имеется другая возможность задания метки для поля - с помощью параметра ***label***.

```python
from django import forms


class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
    age = forms.IntegerField(label="Возраст клиента")
```



# Использование POST запросов

Для создания формы здесь использован стандартный элемент HTML <forrn>. В начале формы помещен встроенный тег Django ***{% csrf_token %}***, который позволяет защитить приложение от ***CSRF-атак***, добавляя в форму ***csrf***-токен в виде скрытого поля.

*CSRF (Cross-Site Request Forgery, также XSRF) - опаснейшая атака, которая приводит к тому, что хакер может выполнить на неподготовленном сайте массу различных действий от имени других зарегистрированных посетителей.*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
    <table>
        {{form}}
    </table>
    <input type="submit" value="Отправить" >
    </form>
</body>
</html>
```

В нижней части формы помещена кнопка для отправки содержимого этой формы на сервер.

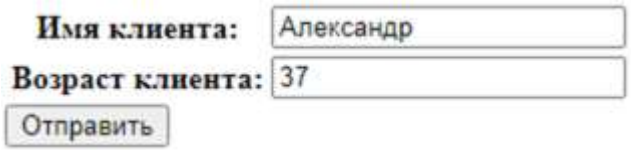
**Токен** -цифровой сертификат безопасность.

# Использование POST запросов

Далее в представлении (в файле ***views.py***) определим следующий код для функции ***index()***
Представление обрабатывает сразу два типа запросов: ***GET*** и ***POST***. Для определения типа запроса делается проверка значения ***request.method*** в структуре ***if ... else***.

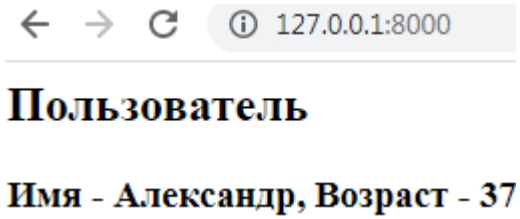
```
from django.http import *
from django.shortcuts import render
from .forms import UserForm


def index(request):
    if request.method == "POST":
        name = request.POST.get("name") # получить значение поля Имя
        age = request.POST.get("age")  # получить значение поля Возраст
        output = f"<h2>Пользователь</h2><h3>Имя - {name}, Возраст - {age}</h3>"
        return HttpResponse(output)
    else:
        userform = UserForm()
        return render(request, "firstapp/index.html", {"form": userform})
```

Если запрос представляет тип ***GET*** (ветка ***else***), то мы просто формируем пользовательскую форму ***userform*** и отправляем ее для ввода данных в шаблон ***index.html***. Таким образом, при первом обращении к приложению мы вначале увидим сформированную нами форму ввода.



# Использование POST запросов

Если запрос будет иметь типа POST (***request.method "POST"***), то это данные формы, отправляемые по нажатию кнопки ***Отправить***. В этом случае отправляемые из формы данные присваиваются переменным ***name*** и ***age***, а их значения – переменной ***output***. После этого значения из ***output*** отправляются через объект ***HttpResponse***. В нашем случае ответ отправляется пользователю на ту же HTML-страницу.




Здесь реализован простейший вариант, когда все действия совершаются в пределах одной страницы. В реально работающих приложениях делается либо переадресация (данные отправляются на другую страницу), или они записываются в базу данных.

# Использование полей в формах Django

В формах Django используются собственные классы для создания полей, через которые пользователь может вводить в приложение различные признаки и данные. Для демонстрации примеров применения различных полей воспользуемся файлом-шаблоном для главной страницы ***firstapp\index.html***.

В наших примерах мы задействуем файл представления ***views.py***, в котором сформируем следующий код для функции ***index ().*** В этой функции мы активируем нашу форму ***userform*** и вызываем главную страницу приложения ***firstapp\index.html***, в которой и будет отображаться эта форма.

```
from django.shortcuts import render
from .forms import UserForm


def index(request):
    userform = UserForm()
    return render(request, "firstapp/index.html", {"form": userform})
```

Все изучаемые нами поля мы будем размещать в коде файла ***forms.py***

```
from django import forms


class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
    age = forms.IntegerField(label="Возраст клиента")
```

# Типы полей

Познакомимся со списком наиболее употребляемых полей, которые используются в формах Django, и рядом свойств, которые являются общими для всех типов полей.

| Тип поля | Описание поля |
|---|---|
| forms.BooleanField | Создает поле checkbox |
| forms.NullBooleanField | Создает поле выбора (не выбрано, да, нет) |
| forms.CharField | Предназначено для ввода текста |
| forms.EmailField | Предназначено для ввода адреса электронной почты |
| forms.GenericIPAddressField | Предназначено для ввода IP-адрес |
| forms. RegexField ( regex="рег_ выр") | Предназначено для ввода текста, который должен соответствовать определенному регулярному выражению |
| forms.SlugField() | Предназначено для ввода текста, который условно называется «slug» - это последовательность символов в нижнем регистре, чисел, дефисов и знаков подчеркивания |
| forms.URLField () | Предназначено для ввода ссылок |
| forms.UUIDField () | Предназначено для ввода UUID (универсального уникального идентификатора) |

# Типы полей

| Тип поля | Описание поля |
|---|---|
| forms.ComboField (fields= [fieldl, field2, .. ]) | Аналогично обычному текстовому полю, однако, требует, чтобы вводимый текст соответствовал требованиям тех полей, которые передаются через параметр fields |
| forms.MultiValueField (fields= [fieldl, field2, .. ]) | Предназначено для создания сложных компоновок, (fields=[fieldl, field2, .. ]) состоящих из нескольких полей |
| forms.FilePathField (path="каталог файлов") | Создает список select, который содержит все папки |
| forms.FileField () | Предназначено для выбора файла |
| forms.ImageField () | Предназначено для выбора файла, но при этом добавляет ряд дополнительных возможностей |
| forms.DateField () | Предназначено для установки даты. В создаваемое поле вводится текст, который может быть сконвертирован в дату-например: 2021-12-25 или 11/25/21 |
| forms.TimeField () | Предназначено для ввода времени - например: 14:30:59 или 14: 30 |

# Типы полей

| Тип поля | Описание поля |
|---|---|
| forms. DateTimeField () | Предназначено для ввода даты и времени, например, 2021-12-25 14:30:59 или 11/25/21 14:30 |
| forms. DurationField () | Предназначено для ввода временного промежутка. Вводимый текст должен соответствовать формату ("DD HH:MM:SS", например, 2 1:10:20 (2 дня 1 час 10 минут 20 секунд) |
| forms.SplitDateTimeField() | Создает два текстовых поля для ввода соответственно даты и времени |
| forms.IntegerField() | Предназначено для ввода целых чисел |
| forms.DecimalField() | Предназначено для ввода чисел с дробной частью |
| forms.FloatField() | Предназначено для ввода чисел с плавающей точкой |
| forms.ChoiceField (choises=кортеж_ кортежей) | Генерирует список select, каждый из его элементов (choises=кортеж_кортежей) формируется на основе отдельного кортежа. Например, =(1, "English"), (2, "German") , (3, "French") |

# Типы полей

| Тип поля | Описание поля |
|---|---|
| forms.TypeChoiceField (choises=кортеж_ кортежей, coerce=функция_преобразования, empty_value=None) | Также генерирует список select на основе кортежа. Однако дополнительно принимает функцию преобразования, которая преобразует каждый элемент. И также принимает параметр empty_value, который указывает на значение по умолчанию. |
| forms.MultipleChoiceField (choises=кортеж_кортежей) | Также генерирует список select на основе кортежа, как и forms. ChoiceField., добавляя к создаваемому полю атрибут multiple="multiple". То есть список поддерживает множественный выбор |
| forms.TypedMultiple ChoiceField | Аналог forms.TypeChoiceField для списка с множественным выбором |

# Типы полей

Каждое поле, когда оно выводится на HTML-страницу, имеет типичный для него внешний вид. Например, поле для ввода текста имеет обрамление в виде рамки. За внешний вид полей отвечает ***виджет*** - представление элемента ввода на HTML-странице. ***Виджеты*** не следует путать с полями формы. Поля формы имеют дело с логикой проверки ввода и используются непосредственно в шаблонах. ***Виджеты*** же имеют дело с отображением элементов ввода HTML-формы на веб-странице и извлечением необработанных отправленных данных. Однако ***виджеты*** должны быть назначены полям формы.

Всякий раз, когда вы указываете поле в форме, Django задействует виджет по умолчанию, соответствующий тому типу данных, которые должны отображаться. Ранее рассмотренные поля при генерации разметки использовали определенные виджеты из пакета ***forms.widgets***. Например, класс ***CharField*** - виджет ***forms.widgets.Textinput***, а класс ***ChoiceField*** - виджет ***forms.widgets.Select.***

Однако есть ряд виджетов, которые по умолчанию не используются полями форм, но тем не менее мы можем их применять:

***Passwordinput*** - генерирует поле для ввода пароля: <input type="password" >;
***Hiddeninput*** - генерирует скрытое поле: <input type="hidden" >;
***MultipleHiddeninput*** - генерирует набор скрытых полей;
***textArea*** - генерирует многострочное текстовое поле: <textarea></textarea>;-
***RadioSelect*** - генерирует список переключателей (радиокнопок): <input type="radio" >;
***CheckboxSelectMultiple*** - генерирует список флажков: <input type="checkbox" >;
***Timeinput***-генерирует поле для ввода _времени (например, 12:41 или 12:41:32);
***SelectDateWidge*** - генерирует три поля select для выбора дня, месяца и года;
***SplitHiddenDateTimeWidget*** - использует скрытое поле для хранения даты и времени;
***Fileinput***- генерирует поле для выбора файла.

# Типы полей

Каждое поле, которое размещено на форме, ориентируется на собственную логику проверки введенных данных, а также принимает несколько других аргументов, которые можно назначить при инициализации поля.

Каждый экземпляр класса ***Field*** имеет метод ***clean()***, который принимает единственный аргумент и вызывает исключение ***django.forms.ValidationError*** в случае ошибки или возвращает чистое значение:

***Field.clean(value)***

```
from django import forms
f = forms.EmailField()
f.clean('info@example.com')
f.clean('Ошибка в написании email адреса')
```

Некоторые классы ***Field*** принимают дополнительные аргументы. Приведенные далее аргументы принимаются всеми полями:

- ***required*** - указывает на необходимость обязательного заполнения поля (по умолчанию: ***required=True***). Для того чтобы сделать поле необязательным для заполнения, нужно указать: ***required=False***;
- ***label*** - позволяет определить видимую пользователем метку для поля (например, если имя поля name и на экране нужно показать Ваше имя, то это можно сделать следующим образом: ***name = forms.CharField ( label=' ваше имя' ) ;***

# Типы полей

- ***label_suffix*** - позволяет переопределить атрибут формы ***label_suffix*** для каждого поля.

```
class ContactForm(forms.Form):
    age = forms.IntegerField()
    nationality = forms.CharField()
    captcha_answer = forms.IntegerField(label='2 + 2', label_suffix=' =')
f = ContactForm(label_suffix='?')
```

- ***initial*** - позволяет определять начальное значение для поля при его отображении на незаполненной форме.

```
class CommentForm(forms.Form) :
    name = forms.CharField(initial=' Ваше имя' )
    url = forms.URLField(initial='http://')
    comment = forms.CharField()
f = CommentForm(auto_id=False)
```

- ***widget*** - позволяет указать класс Widget, который следует использовать при отображении поля вместо того, который задан по умолчанию;

- ***validators*** - позволяет указать список функций, осуществляющих проверку поля;

- ***localize*** - включает локализацию для данных формы как на входе, так и на выходе;

- ***disabled*** - этот аргумент принимает булево значение. При значении ***True*** - выключает поля формы, используя HTML-атрибут ***disabled***. Даже если пользователь отправит данные этого поля, они будут проигнорированы и использовано значение из начальных данных, которые были переданы в форму.

# Типы полей

- ***help_text*** - позволяет указать описание данных, которые нужно внести в поле. Если вы укажете ***help_text***, он будет показан около поля при отображении формы с помощью вспомогательных методов

```
class HelpTextContactForm(forms.Form):
    object = forms.CharField(max_length=100, help_text='Не более 100 символов')
    message = forms.CharField()
    sender = forms.EmailField(help_text='email адрес')
    cc_myself = forms.BooleanField(required=False)
f = HelpTextContactForm(auto_id=False)
```

- ***error_messages*** - позволяет изменить стандартные сообщения об ошибках, которые выдает поле. Создайте словарь с ключами тех сообщений, которые вы желаете изменить. Если не задать значение этому аргументу

```
from django import forms
generic = forms.CharField()
generic.clean(' ')
```

то будет выдано стандартное сообщение об ошибке: ***This field is required***.
Если этот аргумент задать явно, например:

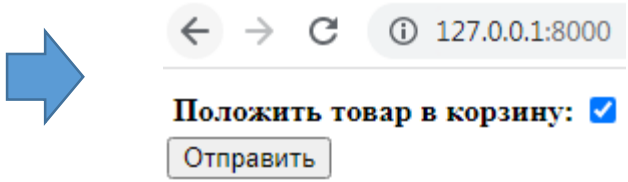
```
name = forms.CharField(error_messages= {'required':'Ошибка, не введено имя'})
name.clean('')
```

то будет выдано заданное сообщение об ошибке: ***Ошибка, не введено имя***

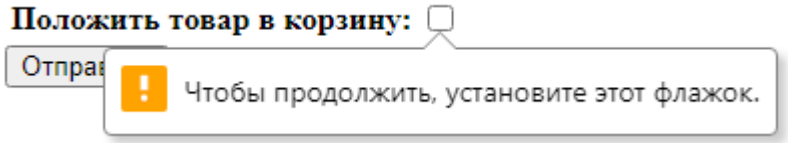
# Типы полей

Метод ***forms.BooleanField*** создает поле <***input type="checkbox"***>. Возвращает значение Boolean: ***True*** - если флажок отмечен и ***False*** - если флажок не отмечен. При инициализации этого поля по умолчанию устанавливается виджет ***Checkboxinput*** и возвращаемое значение ***False***, при этом флажок в ***checkbox*** отсутствует.

```
from django import forms
class UserForm(forms.Form):
    basket = forms.BooleanField(label="Положить товар в корзину")
```




Если клиент не захочет положить товар в корзину, оставив это поле пустым, и нажмет кнопку Отправить, то в ответ получит следующее сообщение




Логика работы приложения нарушена: клиент не хочет приобретать некий товар, но приложение вынуждает его установить флажок в поле выбора. Это происходит потому, что для всех полей по умолчанию установлен признак обязательности внесения значений в поле (***required=True***), а при инициализации поля оно незаполнено (в нем нет флажка). Чтобы пользователь имел возможность оставить это поле пустым, нужно при инициализации поля отключить требование обязательности его заполнения, т. е. свойству ***required*** присвоить значение ***False***.

```
from django import forms
class UserForm(forms.Form):
    basket = forms.BooleanField(label="Положить товар в корзину", required=False)
```

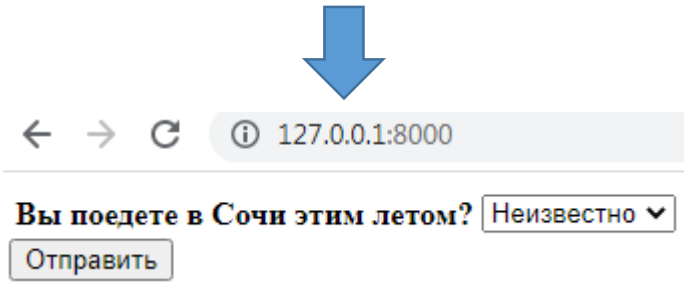
# Типы полей

Метод ***forms.NullBooleanField*** создает на HTML-странице следующую разметку.

```
<select>
<option value="1" selected="selected">Неизвестно</option>
<option value="2">Да</option>
<option value="3">Нет</option>
</select>
```

Внесем изменение в модуль ***forms.py*** и создадим там это поле с помощью следующего кода

```
class UserForm(forms.Form):
    vyb = forms.NullBooleanField(label="Вы поедете в Сочи этим летом?")
```

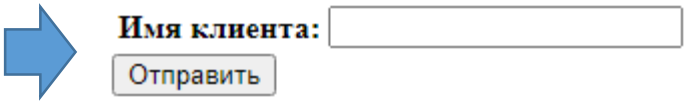



Поле ***NullBooleanField*** по умолчанию использует виджет ***NullBooleanSelect*** и имеет пустое значение ***None***. В зависимости от действия пользователя оно может вернуть три значения: ***None (Неизвестно)***, ***True (Да)*** или ***False (Нет).*** При этом оно никогда и ничего не проверяет (т. е. не вызывает метод ***ValidationError***). Если пользователь нажмет на стрелочку в этом поле, то ему будут предложены три варианта выбора

# Типы полей

Метод ***forms.CharField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Эго поле служит для ввода текста. По умолчанию здесь используется виджет ***textinput***, начальное значение которого - пустая строка (значение ***None***) или текст, который был задан в свойстве поля ***empty_value***.

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
```



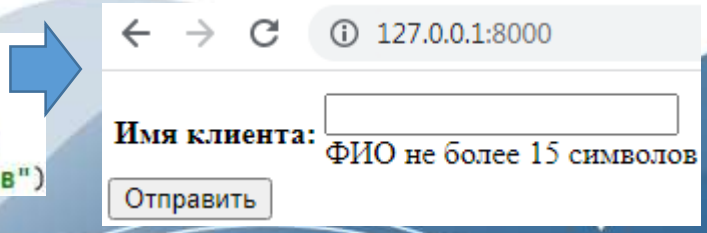

Не забываем, что в таком варианте по умолчанию это поле будет обязательным для заполнения. Если пользователь оставит его пустым, то при нажатии на кнопку ***Отправить*** ему будет выдано напоминание, и программа не даст продвинуться вперед до тех пор, пока поле не будет заполнено.

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента", required=False)
```

Количество символов, которые пользователь может ввести в поле ***CharField***, можно задать с помощью следующих аргументов:

- ***max _ length*** - максимальное количество символов в поле;
- ***min _ length*** - минимальное количество символов в поле.

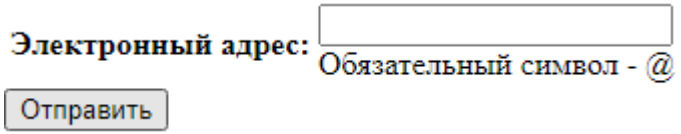
```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента", max_length=15,
                           help_text="ФИО не более 15 символов")
```



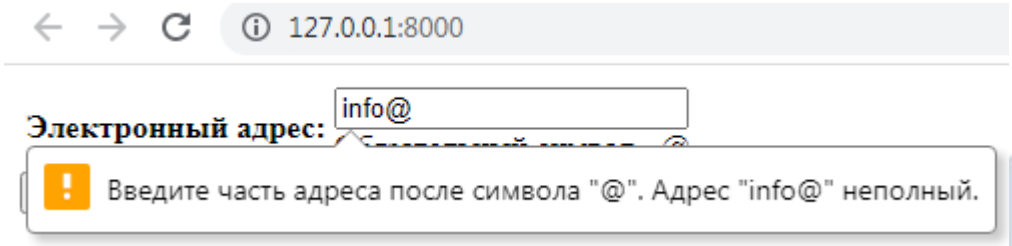
# Типы полей

Метод ***forms. EmailField*** создает на HTML-странице следующую разметку:
***<input type="email">***
Это поле служит для ввода электронного адреса.

```
from django import forms
class UserForm(forms.Form):
    email = forms.EmailField(label="Электронный адрес",
                             help_text="Обязательный символ - @")
```



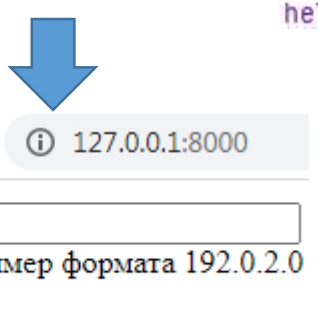

Поле ***EmailField*** по умолчанию использует виджет ***Emailinput*** с пустым значением ***None*** и метод ***EmailValidator*** с умеренно сложным регулярным выражением, проверяющий, что введенное в поле значение является действительным адресом электронной почты. Поле принимает два необязательных аргумента для проверки количества введенных символов: ***max_length*** - максимальная длина строки и ***min_length*** - минимальная длина строки. Они гарантируют, что длина введенной строки не превышает или равна заданной в этих аргументах длине строки. Если пользователь ошибся и ввел некорректный электронный адрес, то при нажатии на кнопку ***Отправить*** ему будет выдано напоминание, и программа не даст продвинуться вперед до тех пор, пока поле не будет заполнено правильно.



# Типы полей

Метод ***forms.GenericIPAddressField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Это поле служит для ввода IP-адреса.

```
from django import forms
class UserForm(forms.Form):
    ip_adres = forms.GenericIPAddressField(label="IP адрес",
                                           help_text="Пример формата 192.0.2.0")
```

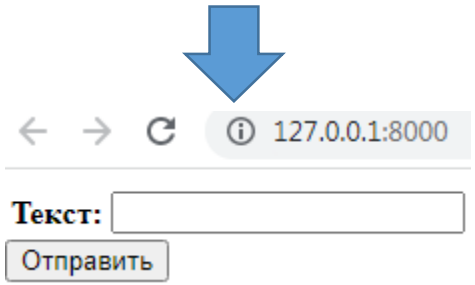



Поле *GenericIPAddressField* по умолчанию использует виджет *Textinput* с пустым значением *None*, при этом проверяется, что введенное значение является действительным IP-адресом.

# Типы полей

Метод ***forms.RegexField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Это поле предназначено для ввода текста, который должен соответствовать определенному регулярному выражению.

```
from django import forms
class UserForm(forms.Form):
    reg_text = forms.RegexField(label="Текст", regex="^[0-9][A-F][0-9] $")
```

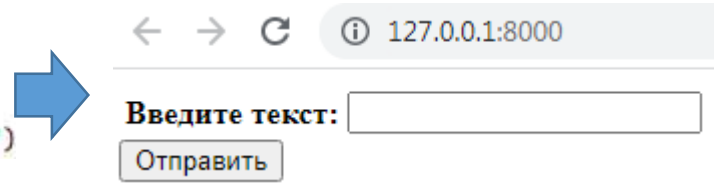



Поле ***RegexField*** по умолчанию использует виджет ***TextInput*** с пустым значением ***None*** и метод ***RegexValidator***, проверяющий, что введенное в поле значение соответствует определенному регулярному выражению, которое указывается в виде строки или скомпилированного объекта регулярного выражения. Поле также принимает параметры: ***max_length*** - максимальная длина строки и ***min_length*** - минимальная длина строки, которые работают так же, как и для поля ***CharField***.

# Типы полей

Метод ***forms.SlugField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Это поле предназначено для ввода текста, который условно называется «slug» и представляет собой последовательность символов в нижнем регистре, чисел, дефисов и знаков подчеркивания.

```
from django import forms
class UserForm(forms.Form):
    slug_text = forms.SlugField(label="Введите текст")
```

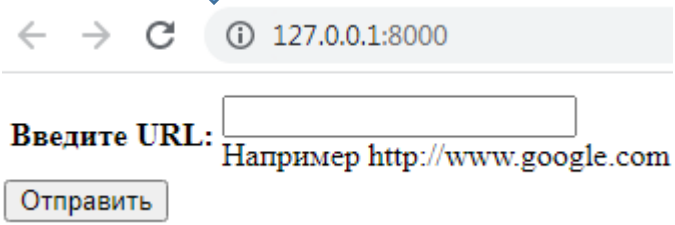



Поле ***SlugField*** по умолчанию использует виджет ***TextInput*** с пустым значением ***None*** и методы ***validate_slug*** или ***validate_unicode_slug***, проверяющие, что введенное значение содержит только буквы, цифры, подчеркивания и дефисы. Поле также принимает необязательный параметр ***allow_unicode*** - логическое указание для поля принимать символы ***Unicode*** в дополнение к символам ASCII. Значение по умолчанию - ***False***.

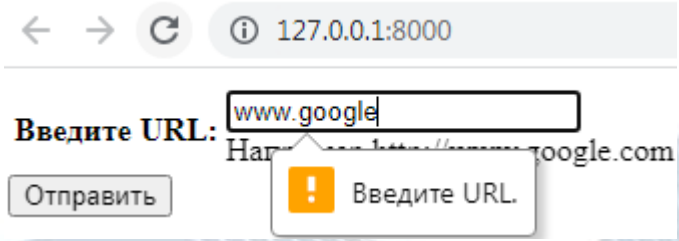
# Типы полей

Метод ***forms.URLField*** создает на HTML-странице следующую разметку:
***<input type="url">***
Это поле предназначено для ввода универсального указателя ресурса (URL) например, такого: ***http://www.google.com***.

```
from django import forms
class UserForm(forms.Form):
    url_text = forms.URLField(label="Введите URL",
                              help_text="Например http://www.google.com")
```




Поле ***URLField*** по умолчанию использует виджет ***URLInput*** с пустым значением None и метод ***URLValidator***, проверяющий, что введенное значение является действительным URL. Поле также принимает параметры: ***max _ length*** - максимальная длина строки и ***min _ length*** - минимальная длина строки, которые работают так же, как и для поля ***CharField***. Если пользователь при вводе URL допустит ошибку в формате этого типа данных, ему будет выдано соответствующее предупреждение.



# Типы полей

Метод ***forms.UUIDField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Это поле предназначено для ввода универсального уникального идентификатора ***UUID*** (например: l 23e4567-e89b-l 2d3-a456-426655440000).

```
from django import forms
class UserForm(forms.Form):
    uuid_text = forms.UUIDField(label="Введите UUID",
                                help_text="Формат xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx")
```

**Введите UUID:**
Формат xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
Отправить

Поле ***UUIDField*** по умолчанию использует виджет ***TextInput*** с пустым значением ***None***. Поле будет принимать любой формат строки, принятый в качестве ***HEX*** аргумента для UUID-конструктора.

# Типы полей

Метод ***forms.ComboField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Это поле имеет структуру ***forms.ComboField (fields=[ fieldl, field2, .. ])*** и представляет собой аналог обычного текстового поля за тем исключением, что требует, чтобы вводимый текст соответствовал требованиям тех полей, которые передаются через параметр ***fields***.

```
from django import forms
class UserForm(forms.Form):
    combo_text = forms.ComboField(label="Введите URL",
                                  fields=[forms.URLField(),
                                  forms.CharField(max_length=20)])
```


Для поля ***ComboField*** делается проверка введенного значения на соответствие каждому из полей, указанных в качестве аргумента при его создании. Оно принимает один дополнительный обязательный аргумент: ***fields***. Это список полей, которые следует использовать для проверки значения, введенного в поле (в порядке, в котором они предоставляются). В нашем примере вводимые пользователем данные будут проверяться на соответствие формату поля для URL, чтобы количество символов не превысило заданной величины (в нашем случае-20 символов)

# Типы полей

Метод ***forms.FilePathField*** создает на HTML-странице следующую разметку:

```
<select>
<option value="folder/filel">fqlder/filel</option>
<option value="folder/file2">folder/file2</option>
<option value="folder/file3">folder/file3</option>
// ..... .
</select>
```

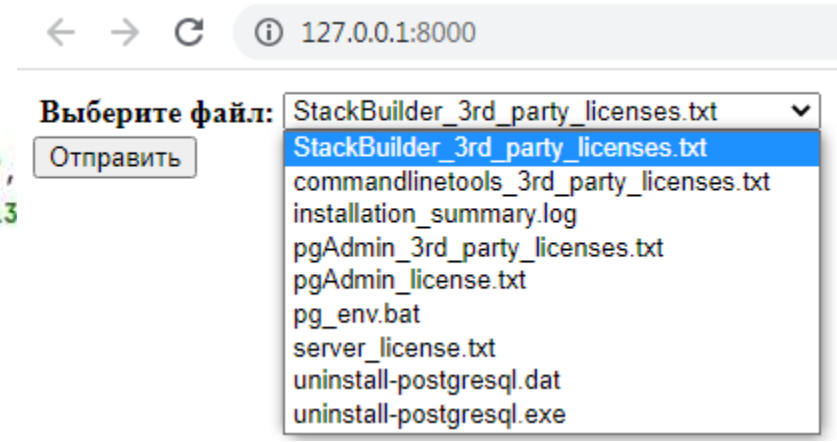
Поле ***FilePathField*** по умолчанию использует виджет ***Select*** с пустым значением ***None*** и проверяет, существует ли выбранный вариант в списке вариантов. Ключи сообщений об ошибках: ***required, invalid_choice***. Поле позволяет делать выбор из файлов внутри определенного каталога. При этом ему требуются пять дополнительных аргументов:

- ***path*** -абсолютный путь к каталогу-(папке), содержимое которого вы хотите отобразить в списке. Это обязательный параметр, и такой каталог должен существовать;
- ***recursive*** - разрешает или запрещает доступ к подкаталогам. Если этот параметр имеет значение False (по умолчанию), то будут предложены к выбору файлы только в указанном каталоге. Если параметр будет иметь значение True, то к выбору будут предложены все вложенные каталоги;
- ***match*** - этот параметр представляет шаблон регулярного выражения (будут отображаться только файлы с именами, совпадающими с этим шаблоном). Регулярное выражение применяется к названию файла, а не к полному пути. Например, выражение ***"foo. *\. txt$"*** покажет файл ***foo23.txt***, но отфильтрует файлы ***bar.tx***t или ***foo23.gif***;
- ***allow_files*** -указывает, следует ли включать файлы в указанном каталоге (по умолчанию ***True***), при этом параметр ***allow_folders*** должен иметь значение ***True***;
- ***allow_folders*** -указывает, следует ли включать подкаталоги в указанном каталоге (по умолчанию ***False***), при этом параметр ***allow_files*** должен иметь значение ***True.***

# Типы полей

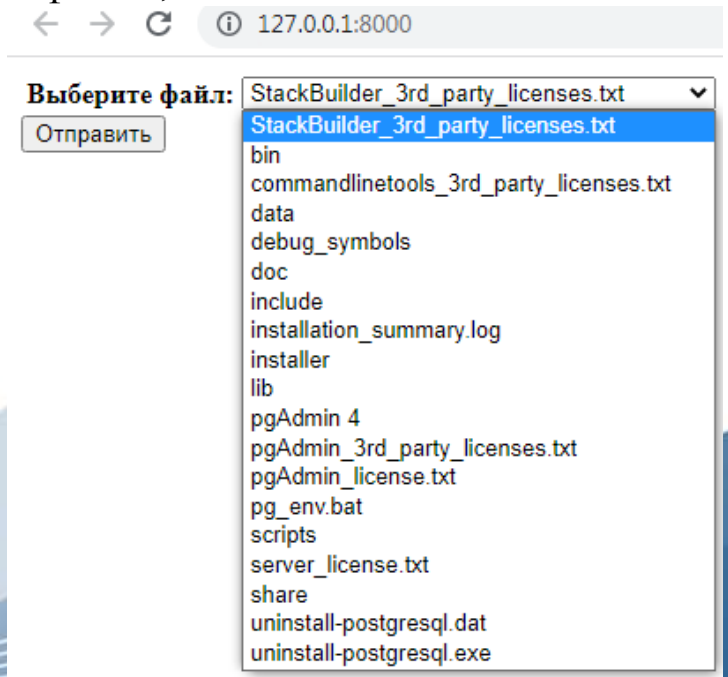
Внесем изменение в модуль ***forms.py*** и создадим там такое поле с помощью следующего кода

```
from django import forms
class UserForm(forms.Form):
    file_path = forms.FilePathField(label="Выберите файл",
                                    path="C:/PostgreSQL/13
```



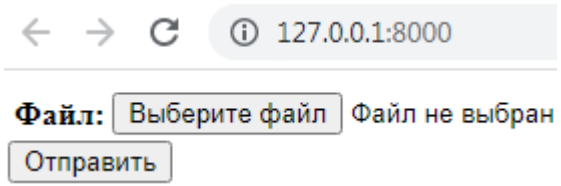

Внесем еще одно изменение в модуль ***forms.py***, с помощью которого реализуем возможность отображать в поле ***FilePathField*** не только файлы, но и вложенные каталоги

```
from django import forms
class UserForm(forms.Form):
    file_path = forms.FilePathField(label="Выберите файл",
                                    path="C:/PostgreSQL/13/",
                                    allow_files="True",
                                    allow_folders="True")
```

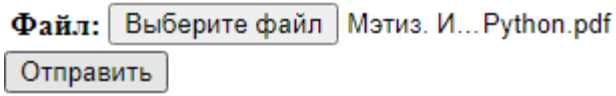


# Типы полей

Метод ***forms.FileField*** создает на HTML-странице следующую разметку:
***<input type="file">***
Поле ***FileField*** предназначено для выбора и загрузки файлов и по умолчанию использует виджет ***ClearableFileInput*** с пустым значением ***None***. Поле формирует объект ***UploadedFile***, который упаковывает содержимое файла и имя файла в один объект. Поле принимает два необязательных аргумента для проверки длины вводимой строки: ***max_length*** (гарантирует, что имя файла не превысит максимальную заданную длину) и ***allow_empty_file*** (гарантирует, что проверка пройдет успешно, даже если содержимое файла пустое).




Поле ***FileField*** представлено в виде кнопки с надписью Выберите файл и сообщением Файл не выбрав. Если теперь нажать на кнопку Выберите файл, то откроется новое окно, в котором можно перемещаться по любым папкам компьютера и просматривать и выбирать любые файлы

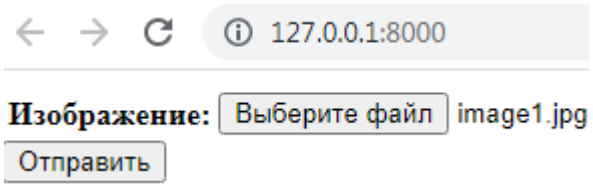


# Типы полей

Метод ***forms.ImageField*** создает на HTML-странице следующую разметку:
***<input type="file">***
Поле ***ImageField*** предназначено для выбора и загрузки файлов, представляющих собой изображения, и по умолчанию использует виджет ***ClearbleFileInput*** с пустым значением ***None***. Поле формирует объект ***UploadedFile***, который упаковывает содержимое файла и имя файла в один объект.

```
from django import forms
class UserForm(forms.Form):
    file = forms.ImageField(label="Изображение")
```

Поле ***ImageField*** представлено в виде кнопки с надписью ***Выберите файл*** и сообщением ***Файл не выбран***. Если теперь нажать на кнопку Выберите файл, то откроется новое окно, в котором можно перемещаться по любым папкам компьютера и просматривать и выбирать любые файлы.



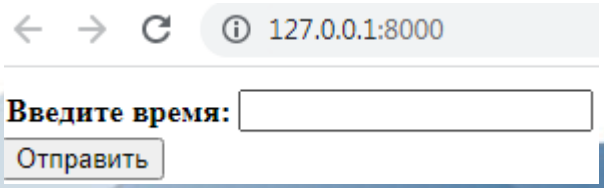
# Типы полей

Метод ***forms.DateField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Поле ***DateField*** служит для ввода дат (например, ***2021-12-25*** или ***25/12/2021***) и по умолчанию использует виджет ***DateInput*** с пустым значением ***None***.
При этом делается проверка, является ли введенное значение либо строкой ***datetime.date***, либо форматированной в определенном формате датой. Поле принимает один необязательный аргумент ***input _ formats***.

```
from django import forms
class UserForm(forms.Form):
    date = forms.DateField(label="Введите дату")
```


Метод ***forms.TimeField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Поле ***TimeField*** служит для ввода значений времени (например, ***14:30:59*** или ***14:30***)
и по умолчанию использует виджет ***TimeInput*** с пустым значением ***None***. При этом делается проверка, является ли введенное значение либо строкой ***datetime.time***, либо форматированным в определенном формате значением времени. Поле принимает один необязательный аргумент ***input_formats***.

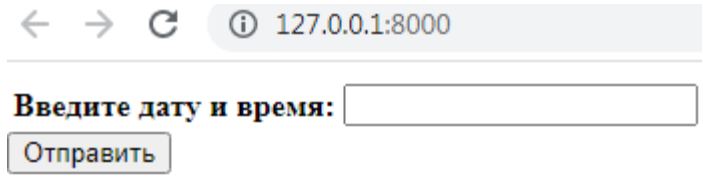
```
from django import forms
class UserForm(forms.Form):
    time = forms.DateField(label="Введите время")
```



# Типы полей

Метод ***forms.DateTimeField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Поле ***DateTimeField*** служит для ввода даты и времени (например, 2021-12-25 14:30:59 или 25/12/2021 14:30) и по умолчанию использует виджет ***DateTimeInput*** с пустым значением ***None***. При этом делается проверка, является ли введенное значение либо строкой ***datetime.datetime, datetime.date***, либо форматированными в определенном формате значениями даты и времени. Поле принимает один необязательный аргумент ***input_formats***.

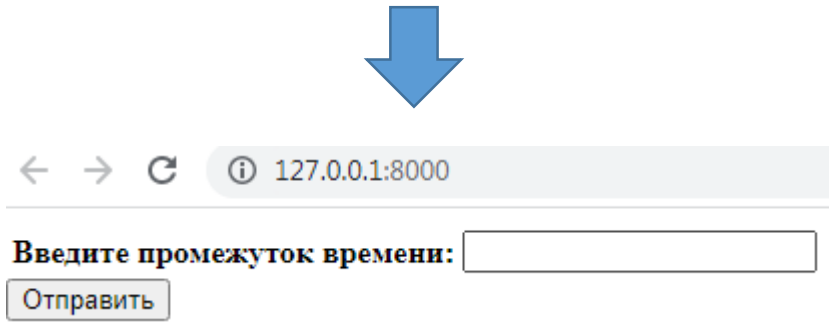
```
from django import forms
class UserForm(forms.Form):
    date_time = forms.DateTimeField(label="Введите дату и время")
```



# Типы полей

Метод ***forms.DurationField*** создает на HTML-странице следующую разметку:
***<input type="text">***
Поле ***DurationField*** предназначено для ввода временного промежутка. Вводимый текст должен соответствовать формату "***DD HH:MM:SS***" -например, 2 1:10:20 (2 дня 1 час 10 минут 20 секунд). По умолчанию поле использует виджет ***TextInput*** с пустым значением ***None***. При этом делается проверка, является ли введенное значение строкой, которую можно преобразовать в ***timedelta***. Поле принимает два аргумента для проверки длины вводимой строки: ***datetime.timedelta.min*** и ***datetime.timedelta.max***. Значение должно быть заключено между этими величинами.

```
from django import forms
class UserForm(forms.Form):
    time_delta = forms.DurationField(label="Введите промежуток времени")
```



# Типы полей

Метод ***forms.SplitDateTimeField*** создает на HTML-странице следующую разметку:

***<input type="text" name="_0" >***

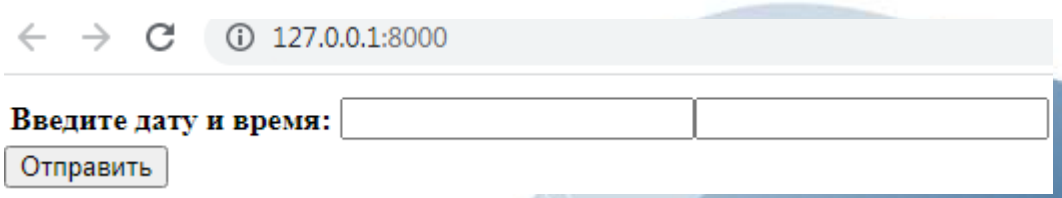
***<input type="text" name="_1" >***

Поле ***SplitDateTimeField*** предназначено для ввода даты и времени в два раздельных текстовых поля и по умолчанию использует виджет ***SplitDateTimeWidget*** с пустым значением ***None***. В первое поле вводится дата, во второе поле - время. При этом делается проверка, является ли введенное значение либо строкой ***datetime.datetime***, либо форматированными в определенном формате значениями даты и времени. Поле принимает два необязательных аргумента:

***input_date_formats*** - список форматов, используемых для попытки преобразования строки в допустимый объект ***datetime.date***. Если аргумент ***input_date_formats*** не указан, используются форматы ввода по умолчанию для поля ***DateField***;

***input_time_ formats*** - список форматов, используемых для попытки преобразования строки в допустимый объект ***datetime.time***. Если аргумент ***input_time_formats*** не указан, используются форматы ввода по умолчанию для поля ***TimeField***.

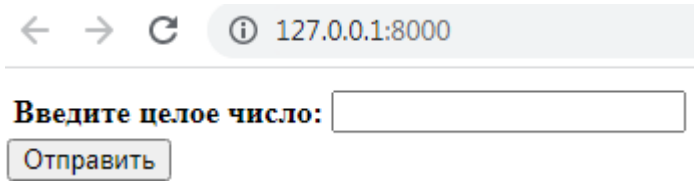
```
from django import forms
class UserForm(forms.Form):
    date_time = forms.SplitDateTimeField(label ="Введите дату и время")
```



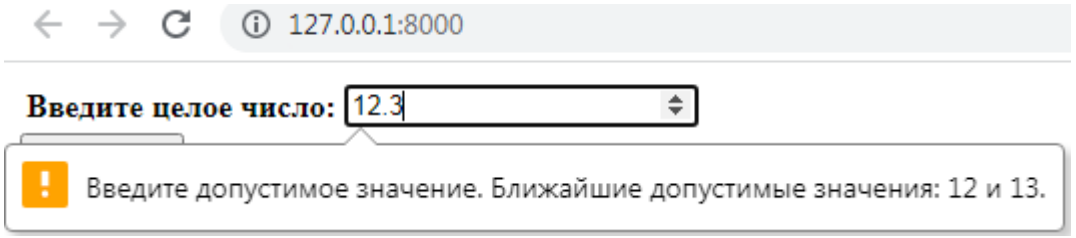
# Типы полей

Метод ***forms.IntegerField*** создает на HTML-странице следующую разметку:
***<input type="number">***
Поле ***IntegerField*** служит для ввода целых чисел и по умолчанию использует виджет ***NumberInput*** с пустым значением ***None***. При этом делается проверка, является ли вводимое число целым. Поле принимает два необязательных аргумента для проверки максимального (***max_value***) и минимального (***min_value***) значения вводимого числа и использует методы ***MaxValueValidator*** и ***MinValueValidator***, если эти ограничения заданы.

```
from django import forms
class UserForm(forms.Form):
    num = forms.IntegerField(label="Введите целое число")
```



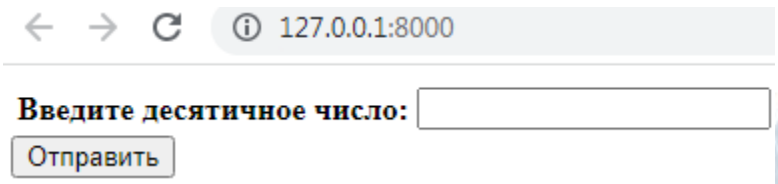

Если в поле ***IntegerField*** попытаться ввести число с дробной частью, то будет выдано предупреждение об ошибке ввода



# Типы полей

Метод ***forms.DecimalField*** создает на HTML-странице следующую разметку:
***<input type="number">***
Поле служит ***DecimalField*** для ввода десятичных чисел и по умолчанию использует виджет ***NumberInput*** с пустым значением ***None***. При этом делается проверка, является ли вводимое число десятичным.
Поле принимает четыре необязательных аргумента:
***max _value*** - максимальное значение вводимого числа;
***min_value*** - минимальное значение вводимого числа;
***max_whole_digits*** -максимальное количество цифр (до десятичной запятой плюс после десятичной запятой, с разделенными начальными нулями), допустимое в значении;
***decimal _places*** -максимально допустимое количество знаков после запятой.
Сообщения об ошибках, выдаваемые при выходе вводимого значения за пределы ***max_value*** и ***min_value***, могут содержать тег ***%(limit_value)s***, замещаемый соответствующим пределом. Аналогичным образом сообщениях об ошибках при выходе: вводимых значений за пределы, определяемые параметрами ***max_digits, max_decimal _places*** и ***max_whole_digits***, могут содержать тег ***% (max) s***:

```
from django import forms
class UserForm(forms.Form):
    num = forms.DecimalField(label="Введите десятичное число")
```

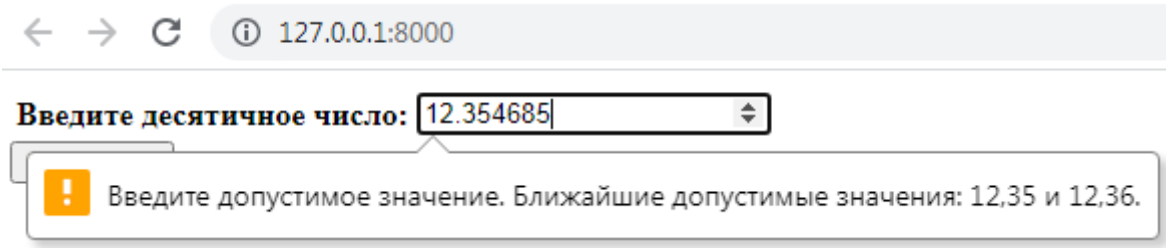


# Типы полей

В поле ***DecimalField*** невозможен ввод никаких символов, кроме чисел и разделителя дробной части, а также значений, выходящих за пределы заданных необязательных аргументов. Чтобы показать это, изменим программный код и укажем, что дробная часть числа ограничена двумя знаками

```
from django import forms
class UserForm(forms.Form):
    num = forms.DecimalField(label="Введите десятичное число",
                             decimal_places=2)
```

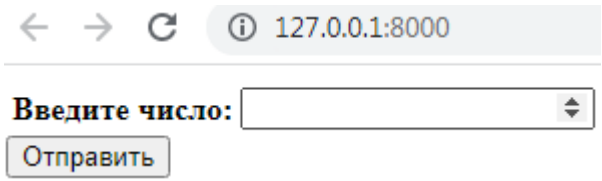
Если теперь в поле ***DecimalField*** попытаться ввести число с более длинной дробной частью, то будет выдано предупреждение об ошибке ввода



# Типы полей

Метод ***forms.FloatField*** создает на HTML-странице следующую разметку:
***<input type="number">***
Поле FloatField служит для ввода чисел с плавающей точкой и по умолчанию использует виджет ***NumberInput*** с пустым значением ***None***. При этом делается проверка, является ли вводимое число числом с плавающей точкой. Поле принимает два необязательных аргумента для проверки максимального (***max_value***) и минимального (***min_value***) значения вводимого числа, контролирующие диапазон значений, разрешенных в поле.

```
from django import forms
class UserForm(forms.Form):
    num = forms.FloatField(label="Введите число")
```



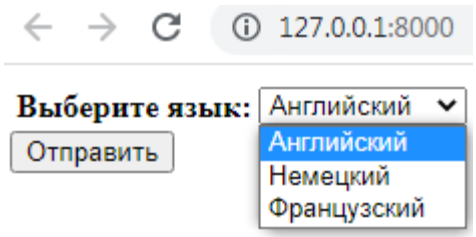
# Типы полей

Метод ***forms.ChoiceField*** создает на HTML-странице следующую разметку.

```
<select>
<option value="l">Date l</option>
<option value="2"> Date 2</option>
<option value="3"> Date 3</option>
</select>
```

Поле ***ChoiceField*** служит для выбора данных из списка и по умолчанию использует виджет ***Select*** с пустым значением ***None.*** Поле имеет структуру ***forms.ChoiceField (choises=кортеж_кортежей***) и генерирует список ***Select***, каждый из элементов которого формируется на основе отдельного кортежа. Поле ***ChoiceField*** принимает один аргумент - ***choices***. Это либо итерация из двух кортежей, используемая в качестве элемента выбора для этого поля, либо вызываемая функция, которая возвращает такую итерацию.

```
from django import forms
class UserForm(forms.Form):
    ling = forms.ChoiceField(label="Выберите язык",
                             choices=((1, "Английский"),
                                      (2, "Немецкий"),
                                      (3, "Французский")))
```



# Типы полей

Метод ***forms.TypedChoiceField*** создает на HTML-странице следующую разметку.

```
<select>
<option value="1">Date 1</option>
<option value="2"> Date 2</option>
<option value="3"> Date 3</option>
</select>
```

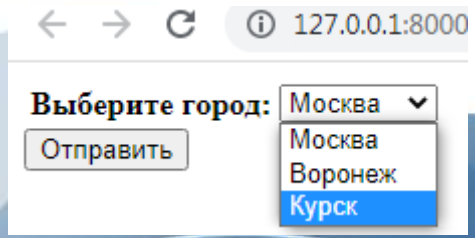
Поле ***TypedChoiceField*** служит для выбора данных из списка. Оно аналогично полю ***ChoiceField*** и по умолчанию также использует виджет ***Select*** с пустым значением ***None***. Поле имеет следующую структуру;

***forms.TypedChoiceField(choises=кортеж:_кортежей,***
***coerce=функция_преобразования,***
***empty_value=None)***

Поле генерирует список дополнительно принимает еще два аргумента:

- ***coerce*** - функцию преобразования, которая принимает один аргумент и возвращает его приведенное значение;
- ***empty_value*** - значение, используемое для представления «пусто». По умолчанию используется пустая строка.

```
from django import forms
class UserForm(forms.Form):
    city = forms.TypedChoiceField(label="Выберите город",
                                  empty_value=None,
                                  choices=((1, "Москва"),
                                           (2, "Воронеж"),
                                           (3, "Курск")))
```



# Типы полей

Поле ***TypedMultipleChoiceField*** имеет следующую структуру:

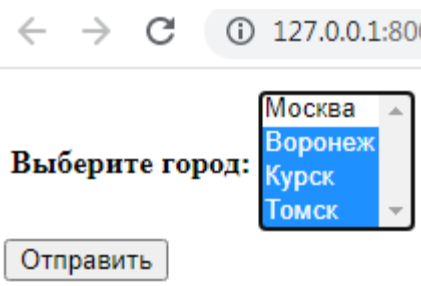
***forms.TypedMultipleChoiceField(choises=кортеж_кортежей,***
***coerce=функция_преобразования,***
***empty_value=None)***

Поле генерирует список ***Select*** на основе кортежа, как и поле ***forms.TypedChoiceField***, добавляя к создаваемому полю атрибут ***multiple="multiple"*** и обеспечивая тем самым поддержку ***множественного выбора***.

По умолчанию поле ***TypedMultipleChoiceField*** использует виджет ***SelectMultiple*** с пустым значением [], а также принимает два дополнительных аргумента:

- ***coerce*** - функцию преобразования, которая принимает один аргумент и возвращает его приведенное значение;
- ***empty_value*** - значение, используемое для представления «пусто». По умолчанию используется пустая строка.

```
from django import forms
class UserForm(forms.Form):
    city = forms.TypedMultipleChoiceField(label="Выберите город",
                                          empty_value=None,
                                          choices=((1, "Москва"),
                                                   (2, "Воронеж"),
                                                   (3, "Курск"),
                                                   (4, "Томск")))
```




В поле одновременно выводится весь список, и пользователь имеет возможность выбрать из него несколько элементов. Для этого необходимо поочередно щелкнуть мышью на нужных элементах списка, удерживая при этом нажатой клавишу <Ctrl>, - выбранные пользователем элементы будут выделены цветом

# Настройка формы и ее полей

Параметр ***widget*** позволяет задать объект, который будет использоваться для генерации HTML-разметки и тем самым определять внешний вид поля на веб-странице. Если пользователь явно не указывает виджет, то Django применит тот виджет, который задан по умолчанию. Однако пользователь может задать для поля свой виджет, заменив им тот, который используется по умолчанию.

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
    age = forms.IntegerField(label="Возраст")
    comment = forms.CharField(label="Комментарий")
```

Имя:
Возраст:
Комментарий:
Отправить

Поскольку мы явно не указали виджет для этих полей, то по умолчанию для них использовался виджет TextInput. Для ввода имени и возраста этот виджет вполне подходит, а вот для комментария его использовать не совсем удобно, поскольку желательно иметь для ввода текста более широкое поле. Назначим для поля ввода комментариев другой виджет – ***Textarea.***

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
    age = forms.IntegerField(label="Возраст")
    comment = forms.CharField(label="Комментарий",
                              widget=forms.Textarea)
```

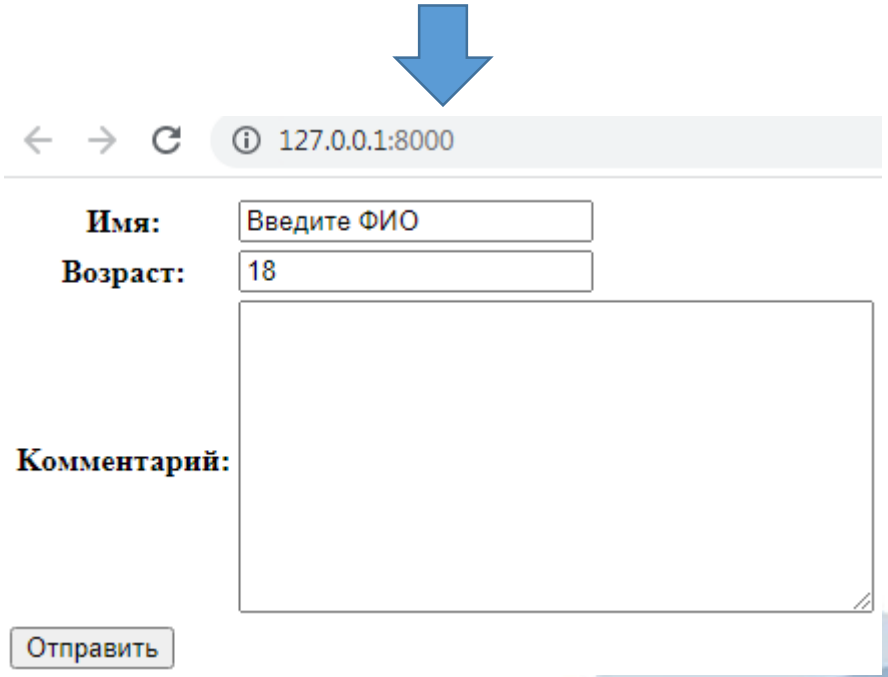
127.0.0.1:8000

Имя:
Возраст:
Комментарий:
Отправить

# Настройка формы и ее полей

Когда в программе создается новое поле, то оно, как правило, имеет пустое значение. Однако его можно сделать не пустым, воспользовавшись свойством ***initial***

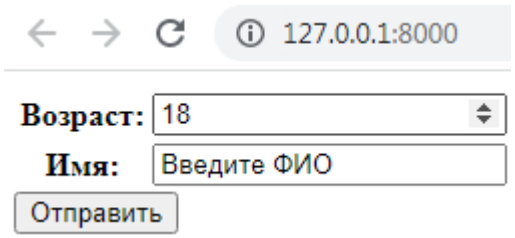
```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя", initial="Введите ФИО")
    age = forms.IntegerField(label="Возраст", initial=18)
    comment = forms.CharField(label="Комментарий",
                              widget=forms.Textarea)
```



# Настройка формы и ее полей

По умолчанию все поля на форме идут в той последовательности, в которой они описаны в модуле инициализации формы. Однако в процессе работы над проектом разработчик может многократно добавлять или удалять поля. Чтобы иметь возможность гибко менять порядок следования полей на форме, можно использовать свойство формы ***field_order***. Эго свойство позволяет переопределить порядок следования полей как в классе формы,.

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя", initial="Введите ФИО")
    age = forms.IntegerField(label="Возраст", initial=18)
    field_order = ["age", "name"]
```

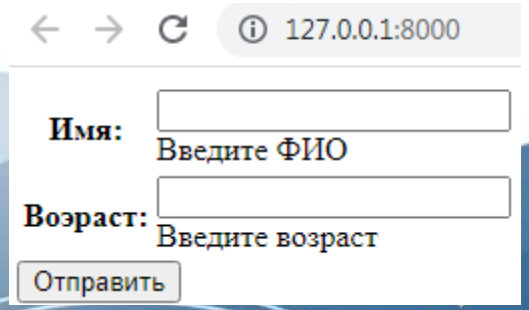



так и при определении объекта формы в представлении ***view***

```
def index(request):
    userform = UserForm(field_order=["age", "name"])
    return render(request, "firstapp/index.html", {"form": userform})
```

Для того чтобы предоставить пользователю дополнительную информацию о том, какие данные следует вводить в то или иное поле, можно использовать свойство поля ***help_text***

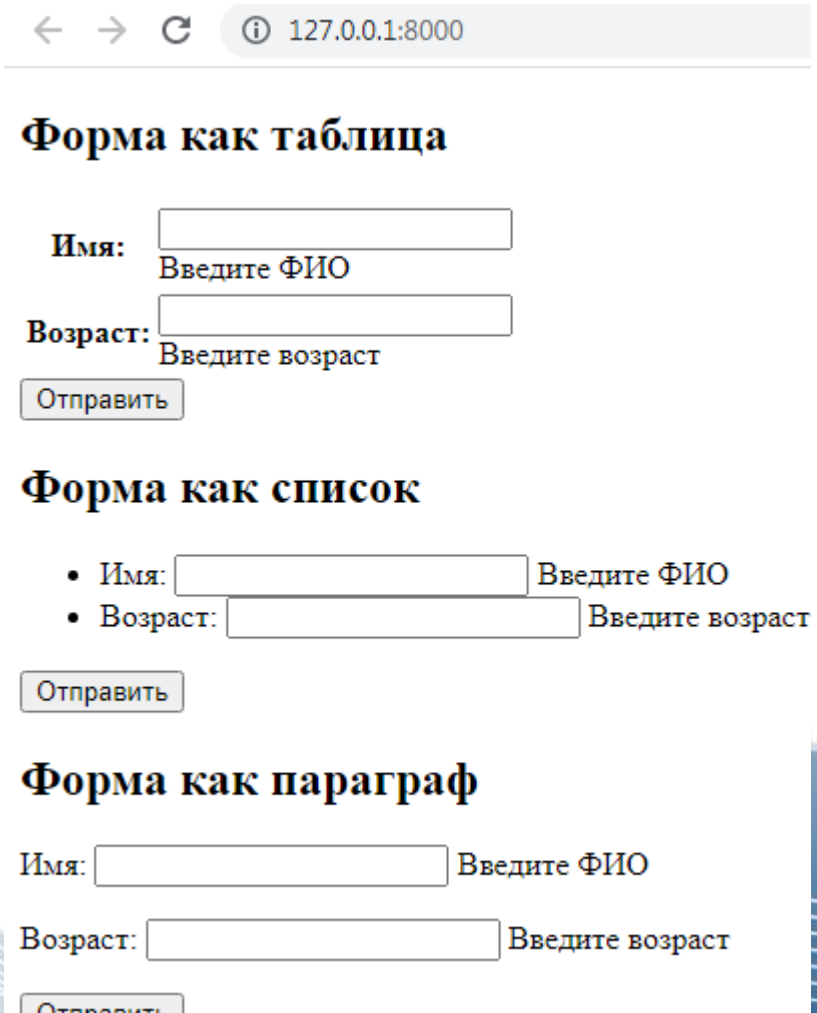
```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя", help_text="Введите ФИО")
    age = forms.IntegerField(label="Возраст", help_text="Введите возраст")
```



# Настройки вида формы

Общее отображение полей на форме можно настроить с помощью специальных методов:

- ***as_table ( )*** - отображение полей в виде таблицы;
- ***as_ul ( )*** - отображение полей в виде списка;
- ***as_P ( )*** - отображение каждого поля формы в отдельном параграфе (абзаце).



# Настройки вида формы

Общее отображение полей на форме можно настроить с помощью специальных методов:

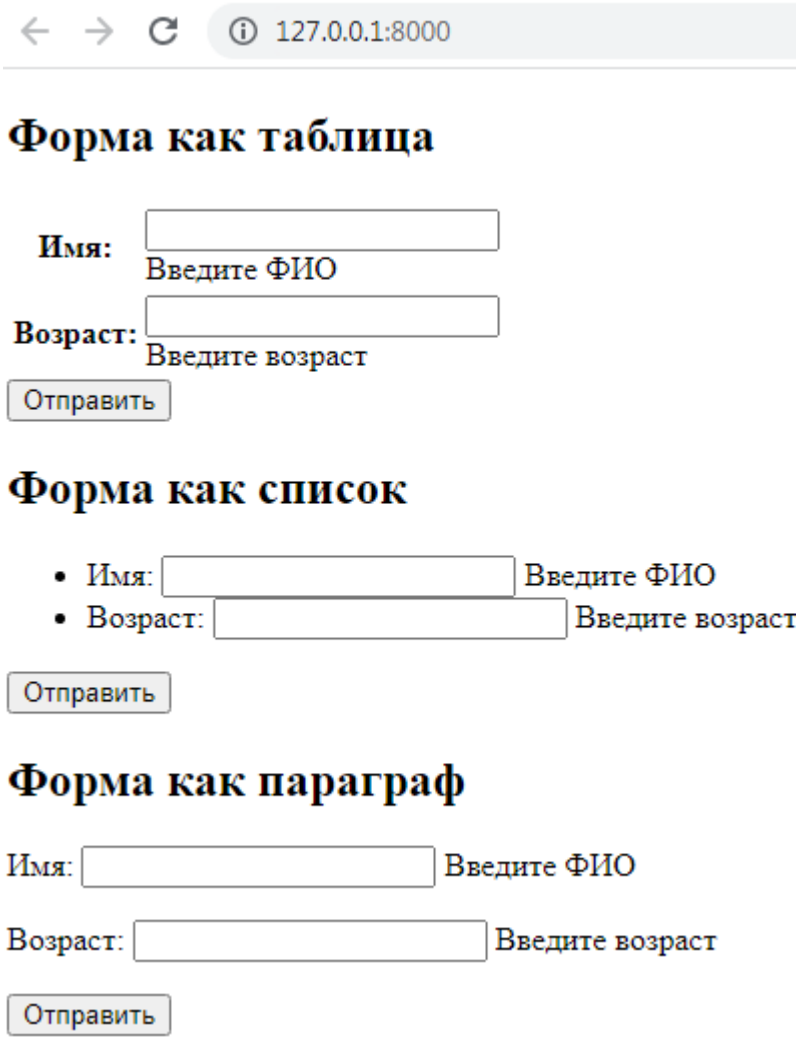
- ***as_table ( )*** - отображение полей в виде таблицы;
- ***as_ul ( )*** - отображение полей в виде списка;
- ***as_P ( )*** - отображение каждого поля формы в отдельном параграфе (абзаце).

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
<h2>Форма как таблица</h2>
<form method="POST">
    {% csrf_token %}
    <table>
        {{ form.as_table }}
    </table>
    <input type="submit" value="Отправить" >
</form>
```

```
<h2>Форма как список</h2>
<form method="POST">
    {% csrf_token %}
    <ul>
        {{ form.as_ul }}
    </ul>
    <input type="submit" value="Отправить" >
</form>

<h2>Форма как параграф</h2>
<form method="POST">
    {% csrf_token %}
    <div>
        {{ form.as_p }}
    </div>
    <input type="submit" value="Отправить" >
</form>

</body>
</html>
```

# Настройки вида формы

Текстовые поля на странице index.html представлены в трех разных видах




В первом случае поля формы представлены в виде таблицы. При этом идентификаторы полей выровнены по центру, а правая и левая граница каждого поля – между собой.

Во втором случае поля формы представлены в виде списка. При этом слева от идентификатора поля присутствует признак элемента списка (маркер).

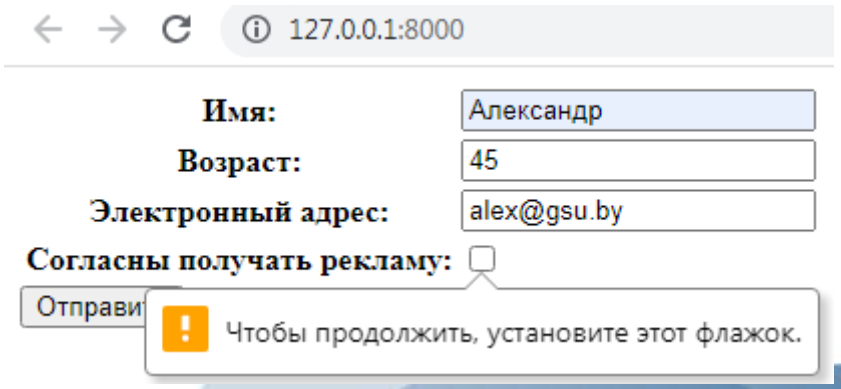
В третьем случае поля формы представлены в виде параграфа (абзаца). При этом идентификаторы полей и сами поля выровнены по левому краю.

# Валидация данных

Теоретически пользователь может ввести в форму и отправить какие угодно данные. Однако не все данные бывают уместными или корректными. Например, в поле для возраста пользователь может ввести отрицательное или четырехзначное число, что вряд ли может считаться корректным возрастом. В Django для проверки корректности вводимых данных используется механизм ***валидации***. Основными элементами валидации являются правила, которые определяют параметры корректности вводимых данных. Например, для всех полей по умолчанию устанавливается обязательность ввода значения, а при генерации HTML-кода для поля ввода задается атрибут ***required*** (обязательное).

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
    <table>
        {{form.as_table}}
    </table>
    <input type="submit" value="Отправить" >
    </form>
</body>
</html>
```

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
    age = forms.IntegerField(label="Возраст")
    email = forms.EmailField(label="Электронный адрес")
    reklama = forms.BooleanField(label="Согласны получать рекламу")
```




Если одно из полей в форме окажется незаполненным, то при нажатии на кнопку ***Отправить*** пользователь получит предупреждение об ошибке

# Валидация данных

Для того чтобы дать пользователю возможность оставить какое-либо поле пустым, необходимо явно отключить для него атрибут ***required***. В нашем случае для этого поле reklama нужно инициировать с помощью следующего программного кода:

```
reklama = forms.BooleanField(label="Согласны получать рекламу",
                             required=False)
```

Для полей, которые требуют ввода текста, - например, ***CharField, EmailField*** и подобных, иногда требуется ограничивать количество вводимых символов. Это можно сделать с помощью параметров: ***max_length*** - максимальное количество символов и ***min_length*** - минимальное количество символов. По аналогии для числовых полей *IntegerField*, ***DecimalField*** и ***FloatField*** можно устанавливать параметры: ***max_value*** и ***min_value***, которые задают соответственно максимально допустимое и минимально допустимое значения. Для поля ***DecimalField*** дополнительно можно задать еще один параметр - ***decimal_places***, который определяет максимальное количество знаков после запятой.

Рассмотренные здесь атрибуты позволяют делать проверку введенных значений на, стороне клиента. Однако возможны варианты, когда пользователи все же смогут отправить форму с заведомо некорректными данными - например, воспользовавшись инструментами для разработчиков. Чтобы предупредить такое развитие событий, можно подправить исходный код формы, добавив к ней атрибут ***novalidate***; который отключает в веб-браузере проверку данных на стороне клиента, и предусмотреть проверку корректности данных на стороне сервера. Для этого у формы вызывается метод ***is_valid()***, который возвращает True, если данные корректны и ***False*** - если данные некорректны.

# Валидация данных

Рассмотрим пример проверки корректности данных на стороне сервера

```python
from django.http import *
from django.shortcuts import render
from .forms import UserForm


def index(request):
    if request.method == "POST":
        userform = UserForm(request.POST)
        if userform.is_valid():
            name = userform.cleaned_data["name"]
            return HttpResponse(f"<h2>Имя введено корррректно - {name} </h2>")
        else:
            return HttpResponse("Ошибка ввода данных")
    else:
        userform = UserForm()
        return render(request, "firstapp/index.html", {"form": userform})
```

# Валидация данных

Если приходит POST-запрос, то вначале происходит заполнение формы пришедшими данными

```
userform = UserForm(request.POST)
```

Потом с помощью метода ***is _ valid ()*** делается проверка их корректности. Если данные введены корректно, то через объект ***cleaned_data*** в переменную ***name*** заносим введенное пользователем значение и формируем ответную страницу с сообщением, что данные корректны.

```
name = userform.cleaned_data["name"]
return HttpResponse(f"<h2>Имя введено коррректно - {name} </h2>")
```

Если данные введены не корректно, то формируем ответную страницу с сообщением об ошибке:

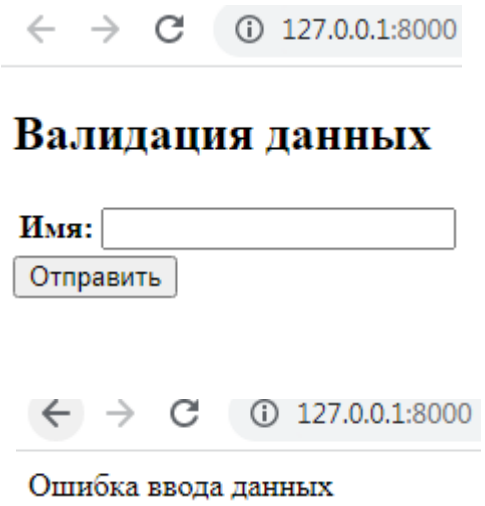
```
userform = UserForm()
return render(request, "firstapp/index.html", {"form": userform})
```

# Валидация данных

Для тестирования нашей формы нужно отключить проверку корректности данных на стороне клиента. Для этого откроем шаблон главной страницы ***firstapp\index.html*** и установим в нем атрибут ***novalidate***

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
<h2>Валидация данных</h2>
    <form method="POST"  novalidate>
        {% csrf_token %}
    <table>
        {{form.as_table}}
    </table>
    <input type="submit" value="Отправить" >
    </form>
</body>
</html>
```

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя")
```




Оставим поле с именем клиента пустым и нажмем кнопку ***Отправить***. Поскольку мы с помощью атрибут ***novalidat***e искусственно отключили возможность проверки данных на стороне клиента, то пустое поле с именем отправится на сторону сервера, где оно будет обработано с помощью метода ***userform. is _ valid ()*** . Поскольку поле оказалось незаполненным, то сервер вернет пользователю страницу с сообщением об ошибке

# Детальная настройка полей формы

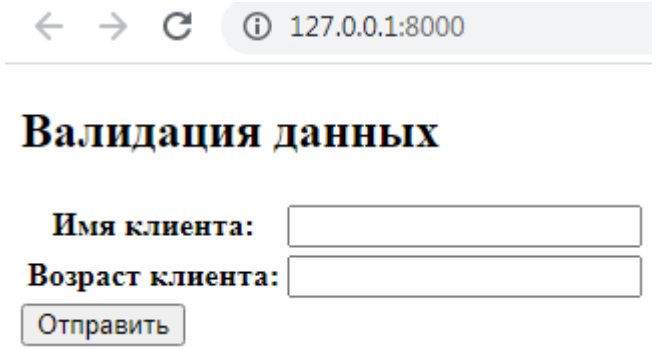
В Django имеется возможность коренным образом изменить всю композицию создаваемых полей. В частности, в шаблоне компонента мы можем обратиться к каждому отдельному полю формы через название формы: ***form.название_поля***. По названию поля мы можем непосредственно получить генерируемый им HTML-элемент без внешних надписей и какого-то дополнительного кода. Кроме того, каждое поле имеет ряд ассоциированных с ним значений:

- ***form.название_поля.name*** - возвращает название поля;
- ***form. название_ поля.value*** - возвращает значение поля, которое ему было передано по умолчанию;
- ***form. название_ поля.label*** - возвращает текст метки, которая генерируется рядом с полем;
- ***form.название_поля.id_for_label*** - возвращает id для поля, которое по умолчанию создается по схеме id _имя_ поля;
- ***form.название_поля.auto_id*** - возвращает id для поля, которое по умолчанию создается по схеме id _имя_ поля***;***
- ***form.название_поля.label_tag***- возвращает элемент label, который представляет метку рядом с полем;
- ***form.название_поля.help_text*** - возвращает текст подсказки, ассоциированной с полем;
- ***form.название_поля.errors*** - возвращает ошибки валидации, связанные с полем;
- ***form. название_ поля. css_classes*** - возвращает CSS-классы поля***;***
- ***form.название_поля.as_hidden*** - генерирует для поля разметку в виде скрытого поля <input type="hidden">;
- ***form. название_ поля. is _ hidden*** - возвращает True или False в зависимости от того, является ли поле скрытым;

# Детальная настройка полей формы

- ***form.название_поля.as_text*** - генерирует для поля разметку в виде текстового поля <input type="text">;
- ***form.название_поля.as_textarea*** - генерирует для поля разметку в виде <textarea> </textarea>;
- ***form.название_поля.as_widget*** - возвращает виджет Django, который ассоциирован с полем.

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента")
    age = forms.IntegerField(label="Возраст клиента")
```



# Детальная настройка полей формы

В этом шаблоне форма представляет собой набор полей, расположенных внутри цикла ***for***. С помощью выражения ***{% for field in form %}*** в цикле делается проход по всем полям формы, при этом есть возможность управлять отображением как самих полей, так и связанных с ними атрибутов: ошибок, текста подсказки, меток и т. д.

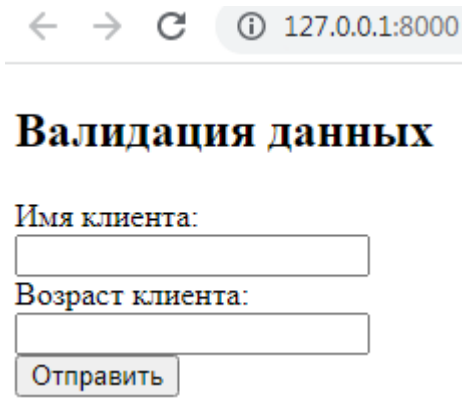
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
</head>
<body>
<h2>Валидация данных</h2>
    <form method="POST"  novalidate>
        {% csrf_token %}
    <div class="form-group">
        {% for field in form %}
            <div>{{field.label_tag}}</div>
            <div>{{field}}</div>
            <div class="error">{{field.errors}}</div>
        {% endfor %}
    </div>
    <input type="submit" value="Отправить" >
    </form>
</body>
</html>
```

Здесь мы сгруппировали отображение следующих значений полей формы:

***field. label _ tag*** - элемент label, который представляет метку рядом с полем;

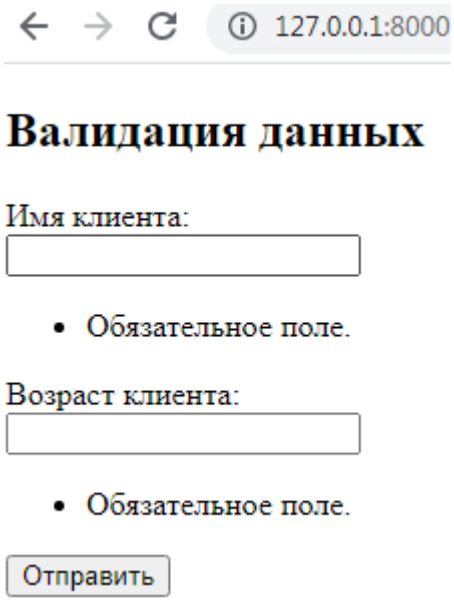
***field*** - само поле;

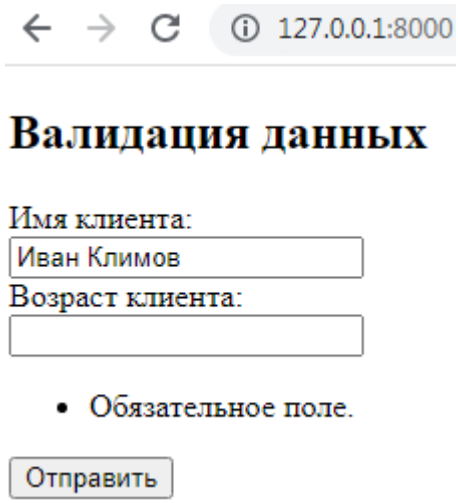
***field.errors*** - ошибки валидации, связанные с полем.



# Детальная настройка полей формы

Если мы теперь при незаполненных полях нажмем кнопку ***Отправить***, то возле каждого поля получим сообщение о наличии ошибки.




Одно поле может содержать несколько ошибок. В этом случае можно использовать тег for для их последовательного вывода:

***{% for error in field.errors %)***
*** {{error}} ***
***{% endfor %}***

# Присвоение стилей полям формы

Поля формы имеют определенные стили по умолчанию. Если же мы хотим применить к ним какие-то собственные стили и классы, то нам надо использовать ряд заложенных в Django механизмов. Прежде всего, имеется возможность вручную выводить каждое поле и определять правила присвоения стилей ему или окружающим его блокам.

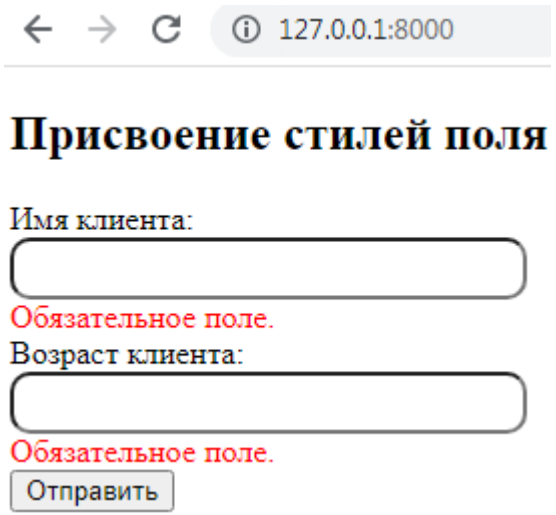
```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента", min_length = 3)
    age = forms.IntegerField(label="Возраст клиента", min_value=1, max_value=100)
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
    <style>
    .alert{color:red}
    .form-group{margin: 10px 0;}
    .form-group input{width:250px;height: 25px;border-radius:10px;}
</style>
</head>
```

# Присвоение стилей полям формы

Здесь в заголовке ***head*** задан стиль ***alert {color: red}*** (красный цвет для отображения ошибок) и стили вывода полей, которые объединены в группы ( отступы, ширина и высота рамки, радиус округления углов). Если после этих изменений запустить наше приложение, то будет видно, что рамки имеют округлую форму, а сообщение об ошибках окрашиваются красным цветом.

```
<body>
<h2>Присвоение стилей поля</h2>
    <form method="POST"  novalidate>
        {% csrf_token %}
    <div class="form-group">
        {% for field in form %}
            <div>{{field.label_tag}}</div>
            <div>{{field}}</div>
            {% if field.errors %}
                {% for error in field.errors %}
                    <div class="alert alert-danger">
                        {{error}}
                    </div>
                {% endfor %}
            {% endif %}
        {% endfor %}
    </div>
    <input type="submit" value="Отправить" >
    </form>
</body>
</html>
```

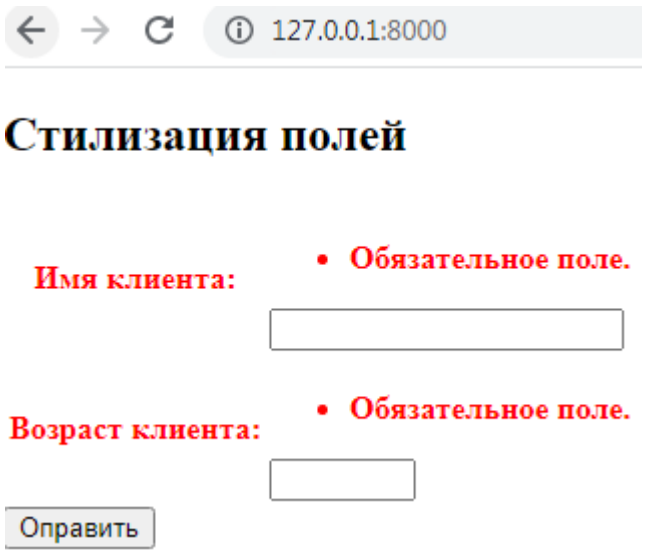


# Присвоение стилей полям формы

В Django имеется еще один механизм для присвоения стилей формам - через использование свойств формы: ***required_css_class*** и ***error_css_class***. Эти свойства применяют CSS-класс к метке, создаваемой для поля формы, и к блоку ассоциированных с ним ошибок. Рассмотрим применение этого механизма на следующем примере.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
    <style>
    .field{font-weight:bold;}
    .error{color:red;}
    </style>
</head>

<body class="container">
<h2>Стилизация полей</h2>
    <form method="POST"  novalidate>
        {% csrf_token %}
    <table>
        {{form}}
    </table>
    <input type="submit" value="Оправить">
    </form>
</body>
</html>
```

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента", min_length=3)
    age = forms.IntegerField(label="Возраст клиента", min_value=1, max_value=100)
    required_css_class = "field"
    error_css_class = "error"
```



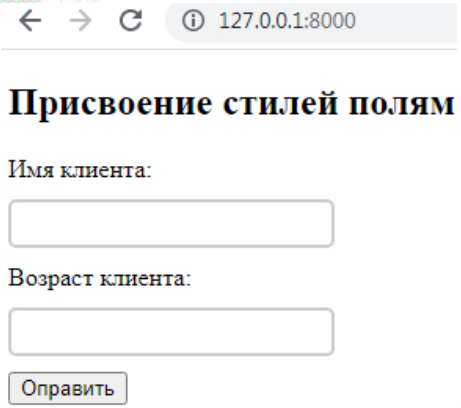
# Присвоение стилей полям формы

В Django имеется и третий механизм присвоения с:гилей формам - через установку стилей в виджетах. В этом коде через параметр виджетов ***attrs*** задаются атрибуты того HTML-элемента, который будет генерироваться. В частности, здесь для обоих полей устанавливается атрибут ***class***, который представляет класс ***myfield***.

```
from django import forms
class UserForm(forms.Form):
    name = forms.CharField(label="Имя клиента",
                           widget=forms.TextInput(attrs={"class": "myfield"}))
    age = forms.IntegerField(label="Возраст клиента",
                             widget=forms.NumberInput(attrs={"class": "myfield"}))
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Форма Django!</title>
    <style>
    .myfield{
            border: 2px solid #ccc;
            border-radius: 5px;
            height: 25px;
            width: 200px;
            margin: 10px 10px 10px 0;
            }
    </style>
</head>
```

```
<body class="container">
<h2>Присвоение стилей полям</h2>
    <form method="POST"  novalidate>
        {% csrf_token %}
    <div>
        {% for field in form %}
        <div class="row">
            {{field.label_tag}}
        <div class="col-md-10">{{field}}</div>
    </div>
        {% endfor %}
    </div>
    <input type="submit" value="Оправить">
    </form>
</body>
</html>
```



# Основы ООП

## Лекция 13. Модели данных

# Модели

Практически любое веб-приложение, так или иначе, работает с базой данных. При этом с системой управления базами данных (СУБД) приложение общается каким-нибудь универсальным способом - например, посредством языка SQL. Однако, программисту чаще всего хочется иметь некую абстракцию, позволяющую большую часть времени работать с привычными сущностями языка. Такой абстракцией является ***Object-Relational Mapping (ORM)*** - отображение сущностей предметной области и их взаимосвязей в объекты, удобные для использования программистом.

Имеются разные подходы к тому, как нужно изолировать пользователя от конкретного хранилища данных. Есть такие, которые полностью скрывают всю работу с БД, - вы пользуетесь объектами, изменяете их состояние, а ***ORM*** неявно синхронизирует состояние объектов и сущностей в хранилище. Другие ORM всего лишь оборачивают сущности БД в структуры языка, но все запросы нужно писать вручную. Это два разных полюса, каждый со своими плюсами и минусами. Авторы Django решили остаться где-то посередине.

# Модели

Используя ***Django ORM***, вы работаете с объектами и выполняете вручную их загрузку и сохранение, однако используете для этого привычные средства языка - вызовы методов и свойств. При этом Django же берет на себя обеспечение правильной работы вашего приложения с конкретными хранилищами данных. Эта изоляция от конкретного хранилища позволяет использовать разные БД в разных условиях: при разработке и тестировании задействовать легковесные СУБД, а при развертывании сайтов в реальных условиях применять более мощные и производительные базы данных

Термин ***«модель»*** часто используется в качестве замены словосочетания ***«Django ORM»***. ***«Модель»*** говорит нам о том, что наша предметная область смоделирована с помощью средств фреймворка. При этом отдельные сущности тоже называются моделями – модели поменьше собираются в большую «***Модель»*** всей предметной области. Связь между моделями и таблицами в БД максимально прямая: одна таблица - одна модель. В этом плане Django ORM не отходит далеко от схемы БД - вы всегда имеете представление о том, как фактически описаны ваши данные с точки зрения СУБД. Что очень полезно, когда нужно что-либо где-то оптимизировать.

# Создание моделей

Модели в Django описывают структуру используемых в программе данных. Эти данные хранятся в базах данных, и с помощью моделей как раз осуществляется взаимодействие с такими базами.

По умолчанию Django в качестве базы данных задействует ***SQLite***. Она очень проста в использовании и не требует запущенного сервера. Все файлы базы данных могут легко переноситься с одного компьютера на другой. Однако при необходимости мы можем использовать в Django большинство других распространенных СУБД.

Для работы с базами данных в проекте Django в файле settings.py определен параметр DATABASES

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

Конфигурация используемой базы данных в таком случае складывается из двух параметров. Параметр ***ENGINE*** указывает на применяемый для доступа к БД движок. В нашем случае это встроенный пакет ***django.db.backends.sqlite3***. Второй параметр - ***NAME*** указывает на имя и путь к базе данных. После первого запуска проекта в нем по умолчанию будет создан файл ***db.sqlite3***, который, собственно, и будет служить в качестве базы данных.

# Создание моделей

Чтобы использовать другие системы управления базами данных, 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 |

При создании приложения по умолчанию в его каталог добавляется файл models.py, который применяется для определения и описания моделей. Модель представляет собой класс, унаследованный от ***django.db.models.Model***.

# Создание моделей

Рассмотрим процесс создания модели на простом примере и создадим модель, описывающую клиента, имеющего две характеристики: ***имя (name)*** и ***возраст (age)***. Изменим файл ***models.py***

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
```

Определена простейшая модель с именем ***Person***, представляющая характеристики человека (клиента системы). В модели определены два поля для хранения имени и возраста. Поле name представляет тип ***CharField*** - текстовое поле, которое хранит последовательность символов. В нашем случае – имя клиента. Для ***CharField*** указан параметр ***max_length***, задающий максимальную длину хранящейся строки. Поле ***age*** представляет тип ***IntegerField*** - числовое поле, которое хранит целые числа, предназначено для хранения возраста человека.

# Создание моделей

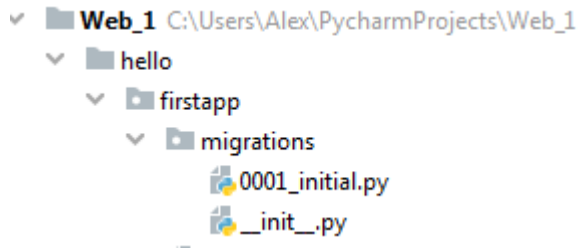
Каждая модель сопоставляется с определенной таблицей в базе данных. Однако пока у нас нет в БД ни одной таблицы, которая бы хранила объекты модели ***Person***. На следующем шаге нам надо создать такую таблицу. В Django это делается с помощью миграции - процесса внесения изменений в базу данных в соответствии с определениями в модели.

***Миграция формируется в два шага.***

На первом шаге необходимо создать миграцию (файл с параметрами миграции) с помощью команды:

***python manage.py makemigrations firstapp***

```
PS C:\Users\Alex\PycharmProjects\Web_1\hello> python manage.py makemigrations firstapp
Migrations for 'firstapp':
  firstapp\migrations\0001_initial.py
    - Create model Person
```




После выполнения этой команды создан файл с миграцией ***firstapp\migrations\0001_initial.py*** и получено сообщение: ***Create model Person*** (создана модель ***Person***)

# Создание моделей

Новый файл ***0001_initial.py***, расположенный в папке ***migrations*** нашего приложения ***firstapp***, будет иметь примерно следующее содержимое

```
from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Person',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=20)),
                ('age', models.IntegerField()),
            ],
        ),
    ]
```

# Создание моделей

На втором шаге миграции необходимо на основе описания полей в модели данных создать соответствующую таблицу в БД. Это делается с помощью команды:

***python manage.py migrate***

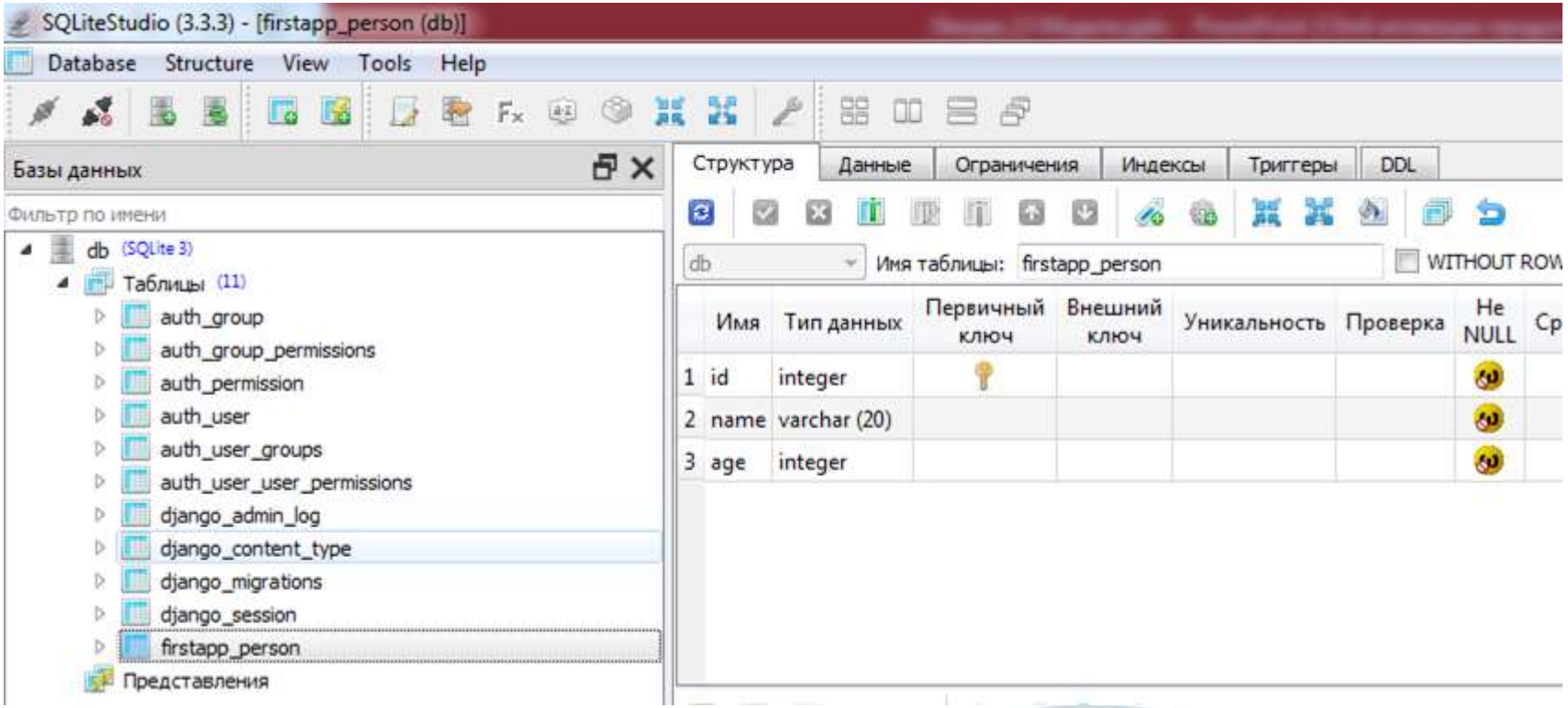
```
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, firstapp, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
```

Теперь, если мы откроем присутствующую в проекте базу данных db.sqlite3 с помощью программы для просмотра БД (например, SQLiteStudio), то увидим, что она содержит ряд таблиц. В основном это все служебные таблицы. В Django имена таблиц формируются автоматически, при этом имя таблицы состоит из двух частей: из имени приложения и имени модели. В нашем случае таблица будет иметь имя ***firstapp _person***.

# Создание моделей

Для просмотра БД откройте SQLiteStudio и добавьте базу из вашего проекта dbsqlite3 файл расположен в папке с ваши проектом

**Database → Add a database**



# Типы полей

В Django для определения моделей данных можно использовать следующие типы полей:

- ***BinaryField ()*** - хранит бинарные данные;
- ***BooleanField ()*** - хранит значение True или False (0 или 1 );
- ***NullBooleanField ()*** - хранит значение True, False или Null;
- ***DateField ()*** - хранит дату;
- ***TimeField ()*** - хранит время;
- ***DateTimeField ()*** - хранит дату и время;
- ***DurationField ()*** - хранит период времени (интервал времени);
- ***AutoField ()*** - хранит целочисленное значение, которое автоматически инкрементируется (увеличивается на 1 ). Обычно применяется для первичных ключей;
- ***BigintegerField ()*** - представляет целое число (значение типа Number), которое
- укладывается в диапазон от - 9223372036854775808 до 9223372036854775807 (в зависимости от выбранной СУБД диапазон может немного отличаться);
- ***DecimalField (decimal_places=X, max_digits=Y)*** - представляет значение числа с дробной частью (типа Number), которое имеет максимум ***X*** разрядов и ***Y*** знаков после запятой;
- ***FloatField ()*** - хранит значение типа Number, которое представляет число с плавающей точкой;
- ***IntegerField ()*** - хранит значение типа Number, которое представляет целочисленное значение;
- ***PositiveIntegerField ()*** - хранит значение типа Number, которое представляет положительное целочисленное значение (от 0 до 2147483647);
- ***PositiveSmallIntegerField()*** - хранит значение типа NшnЬer, которое представляет небольшое положительное целочисленное значение (от 0 до 32767);

# Типы полей

- ***SmallIntegerField ()*** - хранит значение типа NumЬer, которое представляет небольшое целочисленное значение (от -32768 до 32767);
- ***CharField (max _ length=N)*** - хранит строку длиной не более N символов;
- ***TextField ()*** - хранит строку неопределенной длины;
- ***EmailField ()*** - хранит строку, которая представляет email-адрес (значение автоматически проверяется встроенным валидатором EmailValidator);
- ***FileField ()*** - хранит строку, которая представляет имя файла;
- ***FilePathField ()*** - хранит строку, которая представляет путь к файлу длиной в 100 символов;
- ***ImageField ()*** - хранит строку, которая представляет данные об изображении;
- ***GenericIPAddressField ()*** - хранит строку, которая представляет IP-адрес в формате IP4v или IP6v;
- ***SlugField()*** - хранит строку, которая может содержать только буквы в нижнем регистре, цифры, дефис и знак подчеркивания;
- ***URLField ()*** - хранит строку, которая представляет валидный URL-адрес;
- ***UUIDField ()*** - хранит строку, которая представляет UUID-идентификатор.

| Тип | SQLite | MySQL | PostgreSQL | Oracle |
|---|---|---|---|---|
| BinaryField() | BLOB NOT NULL | longblob NOT NULL | bytea NOT NULL | BLOB NULL |
| BooleanField() | bool NOT NULL | bool NOT NULL | boolean NOT NULL | NUMBER(1) NOT NULL CHECK("Значение" IN(0,1)) |
| NullBooleanField() | bool NULL | bool NULL | boolean NULL | NUMBER(1) NOT NULL CHECK(("Значение" IN(0,1)) OR ("Значение" IS NULL)) |
| DateField() | date NULL | date NULL | date NULL | DATE NOT NULL |

# Манипуляция с данными в Django

Аббревиатура ***CRUD*** обозначает четыре основные операции, которые используются при работе с базами данных. Этот термин представляет собой сочетание первых букв английских слов: создание (***Create***), чтение (***Read***), модификация (***Update***) и удаление (***Delete***). Это, по своей сути, стандартная классификация функций для манипуляций с данными. В SQL этим функциям соответствуют операторы ***Insert*** (создание записей), ***Select*** (чтение записей), ***Update*** (редактирование записей) и ***Delete*** (удаление записей). В Django при создании моделей данных они наследуют свое поведение от класса django.db.models.Model, который предоставляет базовые операции с данными (добавление, чтение, обновление и удаление).

В Django для добавления данных в БД можно использовать два метода: ***create()(создать)*** и ***save ( )(сохранить)***. Для добавления данных методом ***create()*** для нашей модели можно использовать следующий код:

```
igor = Person.objects.create(name="Игорь", age=23)
```

Однако, в своей сути, метод ***create()*** использует другой метод - ***save()***, и его мы также можем использовать самостоятельно для добавления объекта в БД

```
igor = Person(name="Игорь", age=23)
igor.save()
```

После успешного добавления данных в БД можно аналогично получить идентификатор добавленной записи - через ***igor.id*** и сами добавленные значения – через ***igor.name*** и ***igor.age***.

# Манипуляция с данными в Django

В Django получить значение данных из БД можно несколькими методами:
***get ( )*** - для одного объекта;
***get _ or _ create ()*** - для .одного объекта с добавлением его в БД;
***all ()*** - для всех объектов;
***filter ()*** - для группы объектов по фильтру;
***exclude ()*** - для группы объектов с исключением некоторых;
***in _bulk ()*** - для группы объектов в виде словаря.

Методы ***all()***, ***filter ()*** и ***exclude()*** возвращают объект ***QuerySet***. Это, по сути, некое промежуточное хранилище, в котором содержится информация, полученная из БД. Объект ***QuerySet*** может быть создан, отфильтрован и затем использован фактически без выполнения запросов к базе данных. База данных не будет затронута, пока вы не инициируете новое выполнение ***QuerySet***

# Манипуляция с данными в Django

Метод ***get ()*** возвращает один объект - т. е. одну запись из БД по определенному условию, которое передается в качестве параметра.

```
klient1 = Person.objects.get(name="Виктор")
klient2 = Person.objects.get(age=25)
klient3 = Person.objects.get(name="Василий", age=23)
```

В этом случае в переменную ***klient1*** будет считан объект, у которого поле ***name="Виктор"***, в переменную ***klient2*** - объект, у которого поле age=25. Соответственно, в переменную ***klient3*** будет считан объект, у которого поле ***name="Василий"*** и поле ***age=23***.

При использовании этого метода надо учитывать, что он предназначен для выборки таких объектов, которые имеются в базе данных в единственном числе. Если в таблице не окажется подобного объекта, то будет выдана ошибка имя_модели ***DoesNotExist***. Если же в таблице присутствуют несколько объектов, которые соответствуют указанному условию, то будет сгенерировано исключение ***MultipleObjectsReturned***. ***Поэтому следует применять этот метод с достаточной осторожностью.***

# Манипуляция с данными в Django

Метод ***get_or_create()*** получает объект из БД, а если его там нет, то он будет добавлен в БД как новый объект. Рассмотрим следующий код:

```
bob, created = Person.objects.get_or_create(name="Bob", age=24)
print(bob.name)
print(bob.age)
```

Этот код вернет добавленный в БД объект (в нашем случае - переменную ***bob***) и булево значение (***created***), которое будет иметь значение ***True***, если добавление прошло успешно.

Если необходимо получить все имеющиеся объекты из базы данных, то применяется метод ***all ()***

```
people = Person.objects.all()
```

Если требуется получить все объекты, которые соответствуют определенному критерию, то применяется метод ***filter ( )***, который в качестве параметра принимает критерий выборки.

```
people = Person.objects.filter(age=23)
people2 = Person.objects.filter(name="Tom", age=23)
```

Здесь в ***people*** будут помещены все объекты из БД, для которых ***age=23***, а в people2 - все объекты с параметрами ***name="Tom"*** и ***age=23***

# Манипуляция с данными в Django

Метод ***exclude()*** позволяет выбрать из БД все записи, за исключением тех, которые соответствуют переданному в качестве параметра критерию.

```
people = Person.objects.exclude(age=23)
```

Здесь в ***people*** будут помещены все объекты из БД, за исключением тех, у которых ***age=23***. Можно комбинировать оба эти метода:

```
people = Person.objects.filter(name="Tom").exclude(age=23)
```

Здесь в ***people*** будут помещены все объекты из БД с именем ***name="Tom"***, за исключением тех, у которых ***age=23***.

Метод ***in_bulk()*** является наиболее эффективным способом для чтения большого количества записей. Он возвращает словарь, т. е. объект ***dict***, тогда как методы ***all()***, ***filter()*** и ***exclude()*** возвращают объект ***QuerySet***. Все объекты из БД можно получить с помощью следующего кода:

```
people = Person.objects.in_bulk()
```

Для получения доступа к одной из выбранных из БД записей нужно использовать идентификатор записи в словаре:

```
people = Person.objects.in_bulk()
for id in people:
    print(people[id].name)
    print(people[id].age)
```

С помощью метода ***in_bulk()*** можно получить и часть объектов из БД. Например, в следующем программном коде из БД будут получены только те объекты, у которых ключевые значения полей равны 1 и 3

```
people2 = Person.objects.in_bulk([1,3])
for id in people2:
    print(people2[id].name)
    print(people2[id].age)
```

# Манипуляция с данными в Django

Для обновления объекта в БД применяется метод ***save ()***. При этом ***Django*** полностью обновляет объект и все его свойства, даже если мы их не изменяли.

```
nic = Person.objects.get(id=2)
nic.name = "Николай Петров"
nic.save()
```

Когда нужно обновить только определенные поля, следует использовать параметр ***update_fields***.

```
nic = Person.objects.get(id=2)
nic.name = "Николай Петров"
nic.save(update_fields= ["name"])
```

Такой подход позволяет повысить скорость работы приложения, особенно в тех случаях, когда требуется обновить большой массив информации. Другой способ обновления объектов в БД предоставляет метод ***update()*** в сочетании с методом ***filter()***, которые вместе выполняют один запрос к базе данных. Предположим, что нам нужно обновить имя клиента в записи таблицы БД с ***id=2***. Это можно сделать с помощью следующего кода.

```
Person.objects.filter(id=2).update(name="Михаил")
```

При таком подходе не нужно предварительно получать из БД обновляемый объект, что обеспечивает увеличение скорости взаимодействия приложения с БД.

Иногда возникает необходимость изменить значение столбца в БД на основании уже имеющегося значения. В этом случае мы можем использовать ***функцию F( ) :***

```
from django.db.models import F
Person.objects.all(id=2) .update(age = F("age") + 1)
```

# Манипуляция с данными в Django

Когда необходимо обновить все записи в столбце таблицы БД вне зависимости от условия, то надо комбинировать метод ***update ()*** с методом ***all ()*** без параметров.

```
from django.db.models import F
Person.objects.all().update(name="Михаил")
Person.objects.all().update(age = F("age") + 1)
```

Здесь всем клиентам будет присвоено значение Михаил, и возраст всех клиентов будет увеличен на единицу. Для обновления записей в БД есть еще один метод - ***update _or _ create***.
Если запись существует, то этот метод ее обновит, а если записи нет, то добавит ее в таблиц

```
values_for_update={"name": "Михаил", "age": 31}
bob, created = Person.objects.update_or_create(id=2, defaults = values_for_update)
```

Метод ***update_or_create()*** здесь принимает два параметра. Первый параметр представляет критерий выборки объектов, которые должны обновляться. Второй параметр предоставляет объект со значениями, которые будут переданы записям, соответствующим критерию из первого параметра. Если соответствующих критерию записей обнаружено не будет, в таблицу добавится новый объект, а переменной ***created*** присвоено значение ***True***. В приведенном примере критерием для обновления записи является идентификатор записи ***id=2***. А поля, которые будут обновлены: ***"name": "Михаил" и "age": 31***.

# Манипуляция с данными в Django

Для удаления информации из БД используется метод delete (). Удаление единственной записи из таблицы можно выполнить с помощью ее id. Например:

```
person = Person.objects.get(id=2)
person.delete()
```

Здесь в первой строке в элемент person загружена информация из строки таблицы базы данных с ***id=2***, а во второй строке вызван метод, который удалил из таблицы БД эту строку. Если не требуется получение отдельного объекта из базы данных, тогда можно удалить объект одной строкой программного кода с помощью комбинации методов ***filter ()*** и ***delete():***

```
Person.objects.filter(id=4).delete()
```

Этой командой будет удалена строка с ***id=4*** непосредственно в базе данных, без предварительной загрузки ее содержимого в приложение.

В Django с помощью свойства ***query*** можно получить и посмотреть текст выполняемого SQL-запроса.

```
people = Person.objects.filter(name="Tom").exclude(age=34)
print(people.query)
```

```
SELECT "firstapp_person"."id", "firstapp_person"."name", "firstapp_person"."age"
FROM "firstapp_person"
WHERE ("firstapp_person"."name" = Tom
       AND NOT ("firstapp_person"."age" = 34))
```

# Пример работы с объектами модели данных

Рассмотрим работу с объектами модели данных на простом примере. Для этого воспользуемся моделью данных Person

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
    objects = models.Manager()
```

В Django за взаимодействие с БД отвечает представление (***view***). Так что на следующем шаге мы в файле ***views.py*** реализуем две функции: для получения сведений из БД и для сохранения данных, введенных пользователем

```
from django.shortcuts import render
from django.http import HttpResponseRedirect
from .models import Person


def index(request):
    people = Person.objects.all()
    return render(request, "index.html", {"people": people})


def create(request):
    if request.method == "POST":
        klient = Person()
        klient.name = request.POST.get("name")
        klient.age = request.POST.get("age")
        klient. save ()
    return HttpResponseRedirect("/")
```

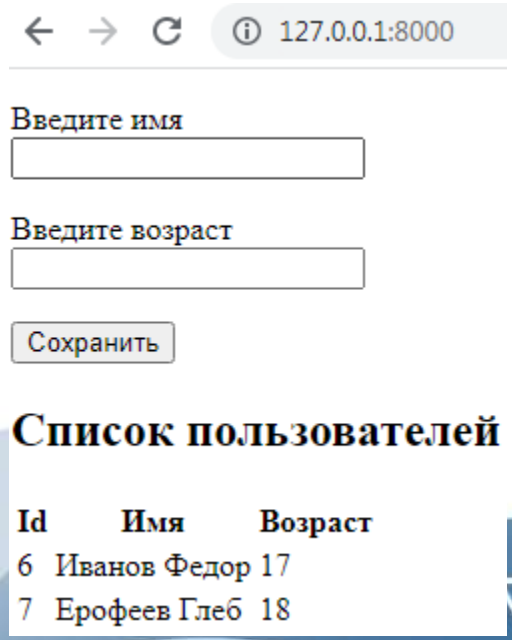
Здесь нами созданы две функции: ***index ()*** и ***create ()***. В функции ***index ()*** мы получаем все данные в объект ***people*** с помощью метода ***Person.objects.all ()*** и затем передаем их в шаблон index.html. В функции ***create ()*** мы получаем из запроса типа POST данные, которые пользователь ввел в форму, затем сохраняем их в БД с помощью метода ***save ( )*** и выполняем через функцию ***index( )*** их переадресацию на главную страницу веб-сайта - т. е. на ***index.html***.

# Пример работы с объектами модели данных

В папке ***templates*** определим шаблон главной страницы ***index.html***

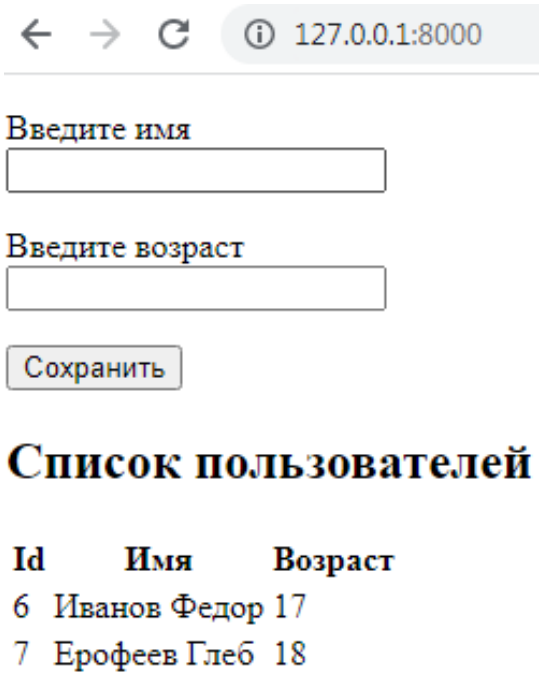
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Модели в Django</title>
</head>


<body class="container">
    <form method="POST" action="create/">
        {% csrf_token %}
        <p>
            <label>Введите имя</label><br>
            <input type="text" name="name" />
        </p>
        <p>
            <label>Введите возраст</label><br>
            <input type="number" name="age" />
        </p>
        <input type="submit" value="Сохранить" >
```

```
</form>
{% if people.count > 0 %}
<h2>Список пользователей</h2>
<table>
    <tr><th>Id</th><th>Имя</th><th>Возраст</th></tr>
    {% for person in people %}
    <tr><td>{{ person.id }}</td>
        <td>{{ person.name }}</td>
            <td>{{ person.age }}</td></tr>
    {% endfor %}
</table>
{% endif %}
</body>
</html>
```



# Пример работы с объектами модели данных

Когда же пользователь заведет данные нескольких клиентов (пользователей), информация о них будет считана из БД и отображена на главной странице




```
SELECT id,
       name,
       age
  FROM firstapp_person;
```

Табличный вид | Форма

| | id | name | age |
|---|---|---|---|
| 1 | 6 | Иванов Федор | 17 |
| 2 | 7 | Ерофеев Глеб | 18 |

# Пример работы с объектами модели данных

Рассмотрим пример с редактированием и удалением объектов модели. Для начала добавим в файл ***views.py*** функции, которые будут выполнять редактирование и удаление информации из БД

```
from django.shortcuts import render
from django.http import HttpResponseRedirect, HttpResponseNotFound
from .models import Person


def index(request):
    people = Person.objects.all()
    return render(request, "index.html", {"people": people})


def create(request):
    if request.method == "POST":
        klient = Person()
        klient.name = request.POST.get("name")
        klient.age = request.POST.get("age")
        klient. save ()
    return HttpResponseRedirect("/")
```

# Пример работы с объектами модели данных

Первые две функции: ***index ()*** и ***create ()*** - здесь остаются без изменений.

Функция ***edit()*** выполняет редактирование объекта. Она в качестве параметра принимает идентификатор объекта id из базы данных. В начале по этому идентификатору мы пытаемся найти объект в БД с помощью метода ***Person.objects.get ( id=id)***. Поскольку в случае отсутствия объекта мы можем столкнуться с исключением ***Person.DoesNotExist*** (объект не найден), то нужно обработать это исключение. И если объект найден не будет, пользователю вернется сообщение об ошибке 404- через вызов метода ***return HttpResponseNotFound ().***

```
def edit(request, id):
    try:
        person = Person.objects.get(id=id)
        if request.method == "POST":
            person.name = request.POST.get("name")
            person.age = request.POST.get("age")
            person. save()
            return HttpResponseRedirect("/")
        else:
            return render(request, "edit.html", {"person": person})
    except Person.DoesNotExist:
        return HttpResponseNotFound("<h2>Клиент не найден</h2>")

def delete(request, id):
    try:
        person = Person.objects.get(id=id)
        person.delete()
        return HttpResponseRedirect("/")
    except Person.DoesNotExist:
        return HttpResponseNotFound("<h2>Клиент не найден</h2>")
```

Когда объект найден, обработка будет делиться на две ветви. Если поступит запрос POST, т. е. пользователь отправил новые (измененные) данные для объекта, мы сохраняем эти данные в БД и выполняем переадресацию на главную страницу вебсайта. Если поступит запрос GET, отображаем пользователю страницу ***edit.html*** с формой для редактирования объекта.

Функция ***delete ()*** аналогичным образом находит объект и выполняет его удаление.

# Пример работы с объектами модели данных

Теперь нужно создать HTML-страницу, в которой пользователь может редактировать данные.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Модели в Django</title>
</head>
<body>
    <form method="POST">
        {% csrf_token %}
        <p>
          <label>Введите имя</label><br>
          <input type="text" name="name" value="{{person.name}}" />
        </p>
        <p>
          <label>Введите возраст</label><br>
          <input type="number" name="age" value="{{person.age}}" />
        </p>
        <input type="submit" value="Сохранить" >
    </form>
</body>
</html>
```

# Пример работы с объектами модели данных

Чтобы обеспечить редактирование и удаление объектов непосредственно на главной странице, изменим шаблон ***index.html***

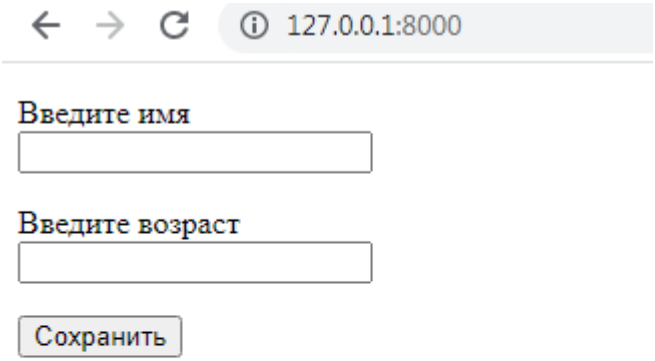
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Модели в Django</title>
</head>


<body class="container">
    <form method="POST" action="create/">
        {% csrf_token %}
        <p>
            <label>Введите имя</label><br>
            <input type="text" name="name" />
        </p>
        <p>
            <label>Введите возраст</label><br>
            <input type="number" name="age" />
        </p>
        <input type="submit" value="Сохранить" >
</form>
```

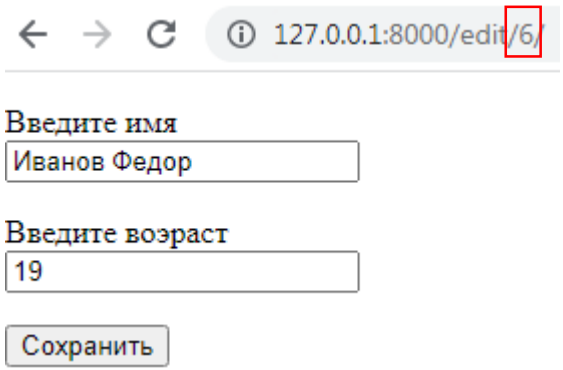
```
{% if people.count > 0 %}
<h2>Список пользователей</h2>
<table>
    <tr>
        <th>Id</th><th>Имя</th><th>Возраст</th><th></th>
    </tr>
    {% for person in people %}
    <tr>
        <td>{{ person.id }}</td>
        <td>{{ person.name }}</td>
        <td>{{ person.age }}</td>
        <td><a href="edit/{{person.id}}">Изменить</a>
            <a href="delete/{{person.id}}">Удалить</a></td>
    </tr>

    {% endfor %}
</table>
{% endif %}
</body>
</html>
```

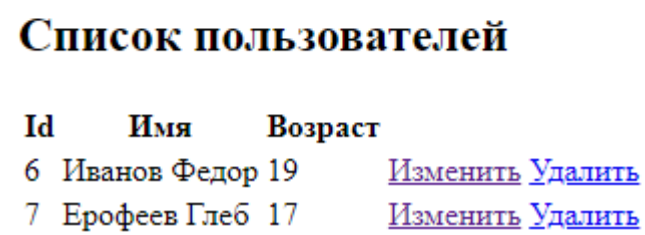
# Пример работы с объектами модели данных

И наконец, в файле ***urls.py*** свяжем маршруты с представлениями

```
from django.urls import path
from firstapp import views
urlpatterns = [path('', views.index),
               path('create/', views.create),
               path('edit/<int:id>/', views.edit),
               path('delete/<int:id>/', views.delete),
]
```

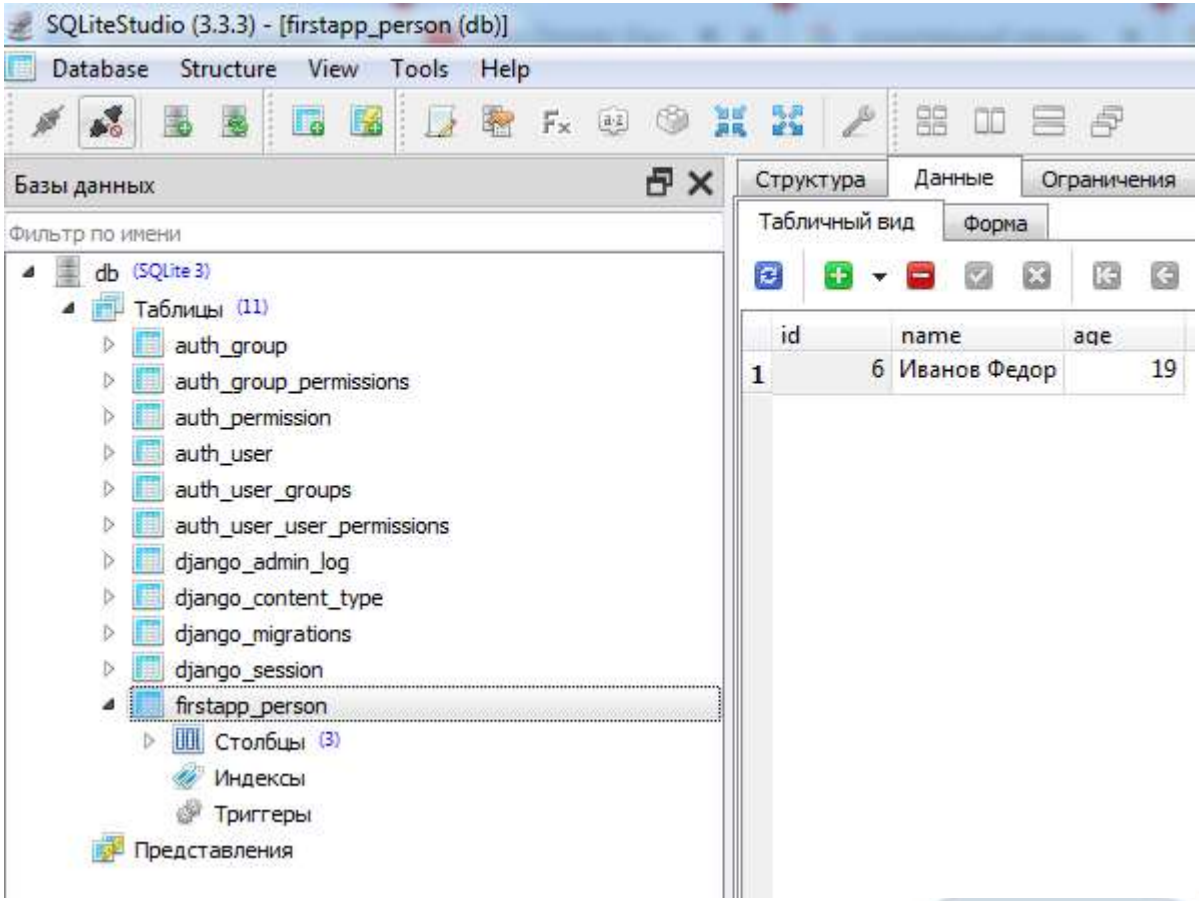


# Пример работы с объектами модели данных

Проверка изменений в БД



# Связи

Большинство таблиц в базе данных имеют связи между собой. Django позволяет определить три наиболее употребительных типа отношений: ***«один-ко-многим», «многие-ко-многим» и «один-к-одному»***

Рассмотрим организацию связи между таблицами БД «один-ко-многим», при которой одна главная сущность может быть связаны с несколькими зависимыми сущностями. Приведем несколько примеров таких связей:

- одна компания, которая выпускает множество видов товаров;
- один автомобиль, который состоит из множества составных частей;
- одна гостиница, которая имеет множество комнат с разными характеристиками;
- одна книга, у которой несколько авторов;
- один город, который имеет множество улиц;
- одна улица, которая имеет множество домов, и т. п

Покажем, как можно связать две таблицы в БД через связанные модели на примере «одна компания- множество товаров»

# Один ко многим

В этом примере модель Company представляет собой производители продукции и является главной моделью (главной таблицей в БД), а модель Product представляет собой различные товары, производимые этой компанией, и является зависимой моделью (зависимой таблицей в Бд). Конструктор типа ***models***. ***ForeignKey*** в классе ***Product*** настраивает связь с главной сущностью. Здесь первый параметр указывает, с какой моделью будет создаваться связь, - в нашем случае это модель ***Company***. Второй параметр (***on_delete***) задает опцию удаления объекта текущей модели при удалении связанного объекта главной модели. В частности, в приведенном коде задано каскадное удаление (***models.CASCADE***). То есть если из БД будет удалена компания, то автоматически из БД будет удалена и вся продукция, которая выпускается этой компанией. Всего для параметра ***on _ delete*** могут использовать следующие значения:

***models.CASCADE*** - автоматически удаляет строку ( строки) из зависимой таблицы, если удаляется связанная строка из главной таблицы;

***models. PROTECT*** - блокирует удаление строки из главной таблицы, если с ней связаны какие-либо строки в зависимой таблице;

***models. SET _ NULL*** - устанавливает значение NULL при удалении связанной строки из главной таблицы;

***models.SET_DEFAULT*** - устанавливает значение по умолчанию для внешнего ключа в зависимой таблице (в таком случае для этого столбца должно быть задано значение по умолчанию);

***models.DO_NOTHING*** - при удалении связанной строки из главной таблицы не производится никаких действий в зависимой таблице.

# Один ко многим

Внесем изменения в ***models.py***.

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=20)
    age = models.IntegerField()
    objects = models.Manager()
    DoesNotExist = models.Manager


class Company(models.Model):
    name = models.CharField(max_length=30)

class Product(models.Model):
    company = models.ForeignKey(Company, on_delete=models.CASCADE)
    name = models.CharField(max_length=30)
    price = models.IntegerField()
```

И выполним миграцию с использованием команды

***python manage.py makemigrations***

В результате выполнения этой команды на основе моделей ***Company*** и ***Product*** в каталоге migrations автоматически будет создан новый файл ***0002_company_product.py***

Затем внесем изменения в саму БД.

***python manage.py migrate***

В результате миграции на основе моделей company и Product в базе данных SQLite автоматически будут созданы таблицы: firstapp _ company и firstapp _product

# Один ко многим

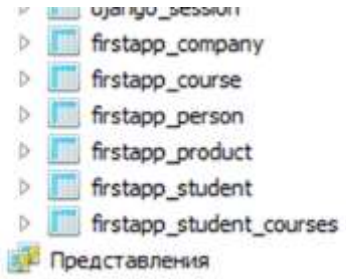
файл ***0002_company_product.py***

```
class Migration(migrations.Migration):

    dependencies = [
        ('firstapp', '0001_initial'),
    ]

    operations = [
        migrations.CreateModel(
            name='Company',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=30)),
            ],
        ),
        migrations.CreateModel(
            name='Product',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=30)),
                ('price', models.IntegerField()),
                ('company', models.ForeignKey(on_delete=django.db.models.deletion.CASCADE, to='firstapp.company')),
            ],
```

# Многие ко многим

Связь ***«многие-ко-многим»*** - это связь, при которой множественным записям из одной таблицы (A) могут соответствовать множественные записи из другой таблицы (B). Примером такой связи может служить учебное заведение, где преподаватели обучают учащихся. В большинстве учебных заведений (школа, университет) каждый преподаватель обучает многих учащихся, а каждый учащийся может обучаться у нескольких преподавателей. Еще один пример- это книги и авторы книг. У одной книги может быть несколько авторов, в то же время у одного автора может быть несколько книг. Связь ***«многие-ко-многим»*** создается с помощью трех таблиц: две их них (A и B) - «источники» и одна таблица- соединительная. Первичный ключ соединительной таблицы (A-B)- составной. Она состоит из двух полей: двух внешних ключей, которые ссылаются на первичные ключи таблиц A и B. Все первичные ключи должны быть уникальными. Это подразумевает и то, что комбинация полей A и B должна быть уникальной в таблице A-B. Для создания отношения «многие-ко-многим» применяется тип связи ManyToManyField.

```python
class Course(models.Model):
    name = models.CharField(max_length=30)
class Student(models.Model):
    name = models.CharField(max_length=30)
    courses = models.ManyToManyField(Course)
```



# Многие ко многим

В результате выполнения этой команды на основе моделей Course и Student в каталоге migrations автоматически будет создан новый файл 0003_course_student.py

```
class Migration(migrations.Migration):

    dependencies = [
        ('firstapp', '0002_company_product'),
    ]

    operations = [
        migrations.CreateModel(
            name='Course',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=30)),
            ],
        ),
        migrations.CreateModel(
            name='Student',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=30)),
                ('courses', models.ManyToManyField(to='firstapp.Course')),
            ],
        ),
```

# Один к одному

При организации связи «один-к-одному» каждая запись из таблицы A может быть ассоциирована только с одной записью таблицы B. Связь «один-к-одному» легко моделируется в одной таблице. Записи такой таблицы содержат данные, которые находятся в связи «один-к-одному» с первичным ключом. В редких случаях связь «один-к-одному» моделируется с использованием двух таблиц. Такой вариант иногда необходим, чтобы преодолеть ограничения СУБД, или с целью увеличения производительности (производится, например, вынесение ключевого поля в отдельную таблицу для ускорения поиска по другой таблице). Или вы сами захотите разнести две сущности, имеющие связь «один-к-одному», по разным таблицам. Например, всю базовую информацию о пользователе (имя, возраст, электронный адрес и пр.) выделить в одну модель, а его учетные данные (логин, пароль, время последнего входа в систему, количество неудачных входов и т. п.) - в другую. ***Но обычно наличие двух таблиц в связи «один-к-одному» считается плохой практикой.***

```
class User(models.Model):
    name = models.CharField(max_length=20)
class Account(models.Model):
    login = models.CharField(max_length=20)
    password = models.CharField(max_length=20)
    user = models.OneToOneField(User,
                                on_delete = models.CASCADE,
                                primary_key = True)
```

# Один к одному

Здесь мы создали модель пользователя (user) с одним полем name и модель учетных данных пользователя (Account) с двумя полями: login и password. Для создания отношения ***«один-к-одному»*** был применен конструктор типа models. ***OneToOneField ()***. Его первый параметр указывает, с какой моделью будет ассоциирована эта сущность (в нашем случае ассоциация с моделью user). Второй его параметр ***(on_delete=models.CASCADE)*** говорщ, что данные текущей модели (Account) будут удаляться в случае удаления связанного объекта главной модели (user). Третий параметр (***primary_key=True***) указывает, что внешний ключ (через который идет связь с главной моделью) одновременно будет выступать и в роли первичного ключа. И соответственно, создавать отдельное поле для первичного ключа. .

# Один к одному

```python
class Migration(migrations.Migration):

    dependencies = [
        ('firstapp', '0003_course_student'),
    ]

    operations = [
        migrations.CreateModel(
            name='User',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('name', models.CharField(max_length=20)),
            ],
        ),
        migrations.CreateModel(
            name='Account',
            fields=[
                ('login', models.CharField(max_length=20)),
                ('password', models.CharField(max_length=20)),
                ('user', models.OneToOneField(on_delete=django.db.models.deletion.CASCADE, primary_key=True, serializ
            ],
        ),
```

# Лабораторные работы

Последовательность шагов при создании графического приложения имеет свои особенности. Программа должна выполнять свое основное назначение, быть удобной для пользователя, реагировать на его действия. Мы не будем вдаваться в подробности разработки, а рассмотрим какие этапы приблизительно нужно пройти при программировании, чтобы получить программу с GUI:

- Импорт библиотеки
- Создание главного окна
- Создание виджет
- Установка их свойств
- Определение событий
- Определение обработчиков событий
- Расположение виджет на главном окне
- Отображение главного окна

1. Импорт модуля tkinter

Как и любой модуль, tkinter в Python можно импортировать двумя способами: командами import tkinter или from tkinter import *. В дальнейшем мы будем пользоваться только вторым способом, т. к. это позволит не указывать каждый раз имя модуля при обращении к объектам, которые в нем содержатся. Следует обратить внимание, что в версии Python 3 имя модуля пишется со строчной буквы (tkinter), хотя в более ранних версиях использовалась прописная (Tkinter). Итак, первая строчка программы должна выглядеть так:

```
from tkinter import *
```

2. Создание главного окна

В современных операционных системах любое пользовательское приложение заключено в окно, которое можно назвать главным, т.к. в нем располагаются все остальные виджеты. Объект окна верхнего уровня создается при обращении к классу Tk модуля tkinter. Переменную связанную с объектом-окном принято называть root (хотя понятно, что можно назвать как угодно, но так уж принято). Вторая строчка кода:

```
root = Tk()
```

3. Создание виджет

Допустим в окне будет располагаться всего одна кнопка. Кнопка создается при обращении к классу Button модуля tkinter. Объект кнопка связывается с какой-нибудь переменной. У класса Button (как и всех остальных классов, за исключением Tk) есть обязательный параметр — объект, которому кнопка принадлежит (кнопка не может "быть ничейной"). Пока у нас есть единственное окно (root), оно и будет аргументом, передаваемым в класс при создании объекта-кнопки:

```
but = Button(root)
```

4. Установка свойств виджет

У кнопки много свойств: размер, цвет фона и надписи и др. Мы рассмотрим их далее. Например, текст надписи (text):

```
but["text"] = "Печать"
```

5-6. Определение событий и их обработчиков

Многообразие событий и способов их обработки будет рассмотрено далее. Здесь же просто коснемся данного вопроса в связи с потребностью.

Что же будет делать кнопка и в какой момент она это будет делать? Предположим, что задача кнопки вывести какое-нибудь сообщение в поток вывода, используя функцию print. Делать она это будет при нажатии на нее левой кнопкой мыши.

Действия (алгоритм), которые происходят при том или ином событии, могут быть достаточно сложным. Поэтому часто их оформляют в виде функции, а затем вызывают,

когда они понадобятся. Пусть у нас печать на экран будет оформлена в виде функции printer:

```
def printer(event):
print ("Каквсегдаочередной 'Hello World!'")
```

Не забывайте, что функцию желательно (почти обязательно) размещать в начале кода. Параметр event – это какое-либо событие.

Событие нажатия левой кнопкой мыши выглядит так: <Button-1>. Требуется связать это событие с обработчиком (функцией printer). Для связи предназначен метод bind. Синтаксис связывания события с обработчиком выглядит так:

```
but.bind("<Button-1>",printer)
```

7. Размещениевиджет

В любом приложении виджеты не разбросаны по окну как попало, а хорошо организованы, интерфейс продуман до мелочей и обычно подчинен определенным стандартам. До стандартов нам далеко, нужно просто кнопку как-то отобразить в окне. Самый простой способ — это использование метода pack.

```
but.pack()
```

Если не вставить эту строчку кода, то кнопка в окне так и не появится, хотя она есть в программе.

8. Отображение главного окна

Ну и наконец, главное окно тоже не появится, пока не будет вызван специальный метод mainloop:

```
root.mainloop()
```

Данная строчка кода должна быть всегда в конце скрипта!

В итоге, код программы может выглядеть таким образом:

```
from tkinter import *
def printer(event):
print ("Как всегда очередной 'Hello World!'")
root = Tk()
but = Button(root)
but["text"] = "Печать"
but.bind("<Button-1>",printer)

but.pack()
root.mainloop()
```

При программировании графического интерфейса пользователя более эффективным оказывается объектно-ориентированный подход. Поэтому многие «вещи» оформляются в виде классов. В нашем примере также можно использовать класс:

```
from tkinter import *
 class But_print:
    def __init__(self):
        self.but = Button(root)
        self.but["text"] = "Печать"
        self.but.bind("<Button-1>",self.printer)
        self.but.pack()
    def printer(self,event):
print ("Как всегда очередной 'Hello World!'")

root = Tk()
obj = But_print()
```

root.mainloop()

В практической части работы будут подробно рассмотены различные компоненты и функции данного модуля.

**Оборудование и материалы.**

Персональный компьютер, среда разработки Python.

**Указания по технике безопасности:**

Соответствуют технике безопасности по работе с компьютерной техникой.

## Задания

Гораздо более серьёзными возможностями, чем модуль turtle, обладает модуль Tkinter. Основное предназначение этого модуля — создание графических интерфейсов (GUI — Graphical User Interface) для программ на Python. Но благодаря наличию элемента графического интерфейса (или, как говорят, "виджета") canvas ("холст") Tkinter можно применять для рисования на основании координат, рассчитанных по формулам, используя доступные элементы векторной графики — кривые, дуги, эллипсы, прямоугольники и пр.

Рассмотрим задачу построения графика некоторой функции по вычисляемым точкам с помощью Tkinter.

Поскольку Tkinter позволяет работать с элементами GUI, создадим окно заданного размера, установим для него заголовок и цвет фона "холста", а также снабдим окно программной "кнопкой". На "холсте" определим систему координат и нарисуем "косинусоиду".

```
importtkinter
importmath
#
tk=tkinter.Tk()
tk.title("Sample")
#
button=tkinter.Button(tk)
button["text"]="Закрыть"
button["command"]=tk.quit
button.pack()
#
canvas=tkinter.Canvas(tk)
canvas["height"]=360
canvas["width"]=480
canvas["background"]="#eeeeff"
canvas["borderwidth"]=2
canvas.pack()
#
canvas.create_text(20,10,text="20, 10")
canvas.create_text(460,350,text="460, 350")
#
points=[]
ay=150
y0=150
x0=50
x1=470
dx=10
#
for n in range(x0,x1,dx):
```

```
        y=y0-ay*math.cos(n*dx)
        pp=(n, y)
        points.append(pp)
#
canvas.create_line(points, fill="blue", smooth=1)
#
y_axe=[]
yy=(x0,0)
y_axe.append(yy)
yy=(x0, y0+ay)
y_axe.append(yy)
canvas.create_line(y_axe, fill="black", width=2)
#
x_axe=[]
xx=(x0,y0)
x_axe.append(xx)
xx=(x1,y0)
x_axe.append(xx)
canvas.create_line(x_axe, fill="black", width=2)
#
tk.mainloop()
```

Посмотрим на результат (рис. 15.1), и разберём текст примера.

Итак, первые две строки программы понятны —подключаются библиотеки. Поскольку в примере должны использоваться тригонометрические функции, необходимо подключить модуль math.

Затем создаётся так называемое "корневое" окно — говоря научным языком, "экземпляр интерфейсного объекта Tk", который представляет собой просто окно без содержимого. Для этого окна устанавливается значение свойства title (создаётся заголовок).

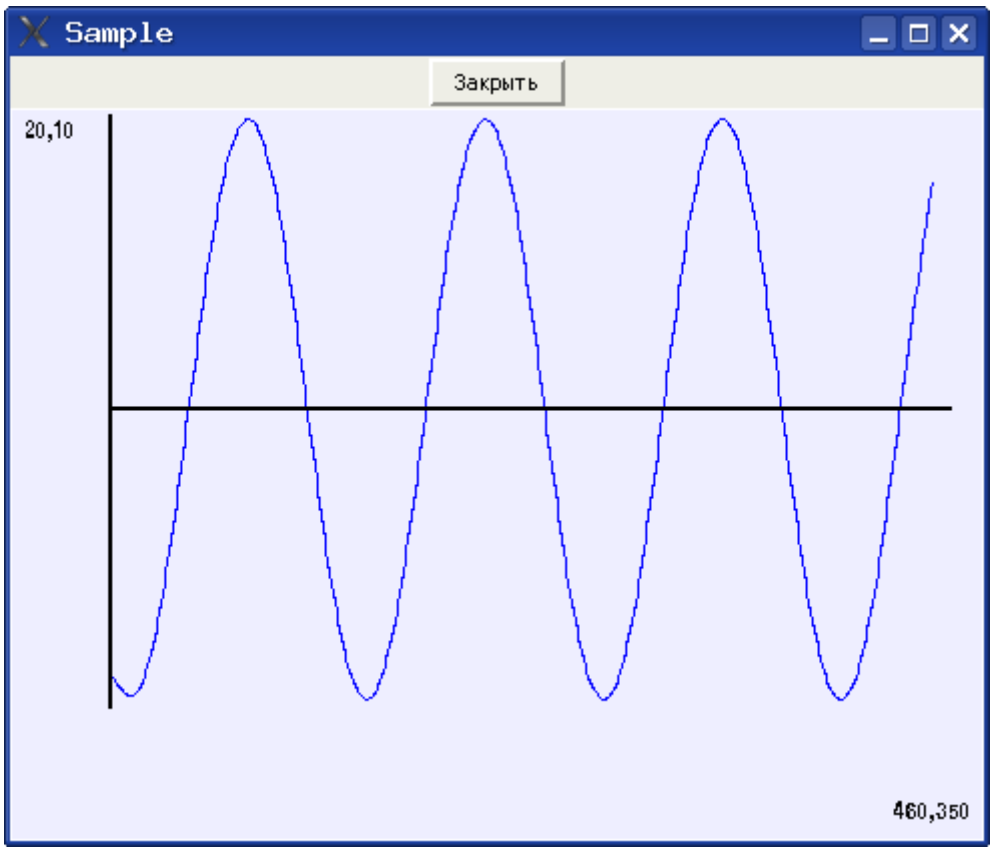



**Рис. 15.1.** Окно Tkinter с кнопкой и графиком

Далее начинается заполнение окна интерфейсными объектами ("виджетами" — widgets). В данном примере используется два объекта — кнопка и "холст". Для размещения объекта в окне используется метод pack(), а порядок объектов определяется порядком выполнения этой функции.

Кнопка создаётся как экземпляр объекта Button модуля Tkinter, связанный с "корневым" окном. Для кнопки можно установить текст надписи (свойство text) и связать кнопку с выполнением какой-либо команды (функции или процедуры), установив значение свойства command.

В приведённом примере кнопка связана с командой закрытия окна и прекращения работы интерпретатора, однако ничто не мешает также закрывать окно нашего "приложения" обычным образом — с помощью стандартной кнопки закрытия окна в верхнем правом углу.

После создания и размещения кнопки создаётся холст. Для элемента (объекта) canvas указываются высота, ширина, цвет фона и отступ от границ окна (таким образом, размеры окна получаются несколько больше, чем размеры объекта canvas). Размеры окна автоматически подстраиваются так, чтобы обеспечить размещение всех объектов (элементов интерфейса).

Прежде чем приступать к рисованию, исследуем систему координат. Поскольку размеры окна уже нами заданы, полезно определить, где находится точка с координатами (0, 0). Как видно из попыток вывести значения координат с помощью метода canvas.create_text(), начало координат находится в верхнем левом углу холста.

Теперь, определившись с координатами, можно выбрать масштабные коэффициенты и сдвиги и сформировать координаты точек для рисования кривой.

При использовании метода canvas. create_line() в качестве координат требуется список пар точек (кортежей) (x,y). Этот список формируется в цикле с шагом dx.

Для линии графика устанавливаются цвет и режим сглаживания. Сглаживание обеспечивает некоторую "плавность" кривой. Если его убрать, линия будет состоять из отрезков прямых. Кроме того, для линий можно устанавливать толщину, как это показано на примере осей координат.

***Моделирование математических функций***

Пусть требуется построить график функции, выбираемой из заданного списка. Здесь потребуется уже использование дополнительных интерфейсных элементов модуля Tkinter, а также создание собственных (пользовательских) процедур или функций для облегчения понимания кода.

Результат решения задачи (вариант внешнего вида "приложения") показан на рис. 3.5.

Для выбора вида математической функции используется раскрывающийся список, после выбора вида функции и нажатия на кнопку "Нарисовать" на "холсте" схематически изображается график этой функции. Кнопка "Закрыть" закрывает наше "приложение". Теперь посмотрим на код.

```
# -*- coding: utf-8 -*-
 import Tkinter
 import math
#
# Пользовательские процедуры
 def plot_x_axe(x0, y0, x1):
  x_axe=[]
  xx=(x0, y0)
  x_axe.append(xx)
  xx=(x1, y0)
  x_axe.append(xx)
  canvas.create_line(x_axe, fill="black", width=2)
#
 def plot_y_axe(x0, y0, y1):
  y_axe=[]
  yy=(x0, y1)
  y_axe.append(yy)
  yy=(x0, y0)
  y_axe.append(yy)
  canvas.create_line(y_axe, fill="black", width=2)
#
 def plot_func0(x0, x1, dx, y0, y1):
  x0i=int(x0)
  x1i=int(x1)
  y0i=int(y0)
  y1i=int(y1)
  a=y1
  b=(y0-y1) / (x1-x0)
  points=[]
          for x in range(x0i, x1i, dx):
             y=int(a+b*x)
             pp=(x, y)
             points.append(pp)
             #
             canvas.create_line(points, fill="blue", smooth=1)
             plot_y_axe(x0i, y0i, y1i)
             plot_x_axe(x0i, y0i, x1i)
             #
 def plot_func1(x0, x1, dx, y0, y1):
```

```
    x0i=int(x0)
    x1i=int(x1)
    y0i=int(y0)
    y1i=int(y1)
    a=y0
    b=y0-y1
    points=[]
            for x in range(x0i, x1i, dx):
                y=int(a-y1i*b/ x)
                pp=(x, y)
                points.append(pp)
    #
    canvas.create_line(points, fill="blue", smooth=1)
    plot_y_axe(x0i, y0i, y1i)
    plot_x_axe(x0i, y0i, x1i)
#
  def plot_ func2(x0, x1, dx, y0, y1):
    x0i=int(x0)
    x1i=int(x1)
    y0i=int(y0)
    y1i=int(y1)
    a=(y0-y1)/(15*x1)
    b=1+((y0-y1)/(x1-x0))
    points=[]
            for x in range(x0i, x1i, dx):
                y=y0i-int(a*(x-x0i)**b)
                pp=(x, y)
                points.append(pp)
    #
    canvas.create_line(points, fill="blue", smooth=1)
    plot_y_axe(x0i, y0i, y1i)
    plot_x_axe(x0i, y0i, x1i)
#
  def plot_ func3(x0, x1, dx, y0, y1):
    x0i=int(x0)
    x1i=int(x1)
    y0i=int(y0)
    y1i=int(y1)
    ay=150
    y0i=150
    points=[]
            for x in range(x0i, x1i, dx):
                y=y0i-ay*math.cos(x*dx)
                pp=(x, y)
                points.append(pp)
    #
    canvas.create_line(points, fill="blue", smooth=1)
    plot_y_axe(x0i, 0, y0i+ay)
    plot_x_axe(x0i, y0i, x1i)
#
  def DrawGraph():
    fn=func.get()
    f=fn[0]
    x0=50.0
    y0=250.0
    x1=450.0
    y1=50.0
```

```
   dx=10
#
            if f=="0":
               canvas.delete("all")
               plot_ func0(x0, x1, dx, y0, y1)
            elif f=="1":
               canvas.delete("all")
               plot_ func1(x0, x1, dx, y0, y1)
            elif f=="2":
               canvas.delete("all")
               plot_ func2(x0, x1, dx, y0, y1)
            else:
               canvas.delete("all")
               plot_ func3(x0, x1, dx, y0, y1)
#
# Основная часть
tk=Tkinter.Tk()
tk.title(" Sample ")
# Верхняя часть окна со списком и кнопками
menuframe=Tkinter.Frame(tk)
menuframe.pack({"side":"top", "fill":"x"})
# Надписьдлясписка
lbl=Tkinter.Label(menuframe)
lbl["text"]="Выбор:"
lbl.pack({"side":"left"})
# Инициализация и формирование списка
func=Tkinter.StrigVar(tk)
func.set('0 y=Ax+B')
#
fspis=Tkinter.OptionMenu(menuframe, func,
   '0 y=Ax+B',
   '1 y=A+B/ x',
   '2 y=Ax^B',
   '3 y=A*cos(Bx)')
fspis.pack({"side":"left"})
# Кнопка управления рисованием
btnOk=Tkinter.Button(menuframe)
btnOk["text"]="Нарисовать"
btnOk["command"]=DrawGraph
btnOk.pack({"side":"left"})
# Кнопказакрытияприложения
button=Tkinter.Button(menuframe)
button["text"]="Закрыть"
button["command"]=tk.quit
button.pack({"side":"right"})
# Областьрисования (холст)
canvas=Tkinter.Canvas(tk)
canvas["height"]=360
canvas["width"]=480
canvas["background"]="#eeeeff"
canvas["borderwidth"]=2
canvas.pack({"side":"bottom"})
tk.mainloop()
```

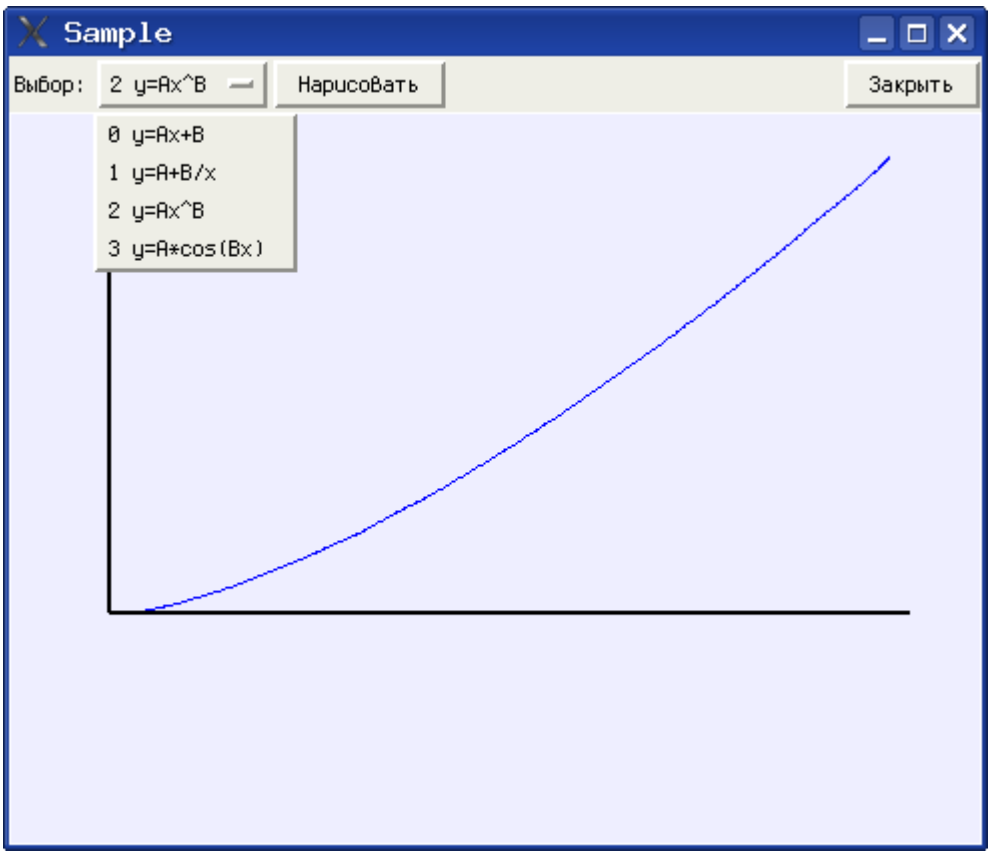



**Рис. 15.2.** Построение графика для функции, выбираемой из списка

Основная часть программы (интерфейсная) начинается с момента создания корневого окна (инструкция tk=Tkinter.Tk()). В этом окне располагаются два интерфейсных элемента — рамка (Frame) и холст (canvas). Рамка является "контейнером" для остальных интерфейсных элементов — текстовой надписи (метки — Label), раскрывающегося списка вариантов (OptionMenu) и двух кнопок. Как видно, кнопка закрытия стала объектом рамки, а не корневого окна, но по-прежнему закрывает всё окно.

Для получения нужного расположения элементов метод pack() используется с указанием, как именно размещать элементы интерфейса (к какой стороне элемента-контейнера их нужно "прижимать").

Есть некоторые тонкости в создании раскрывающегося списка. Для успешного выполнения этой операции нужно предварительно сформировать строку (а точнее, объект Tkinter.StringVar()) и определить для этого объекта значение по умолчанию (это значение будет показано в только что запущенном приложении). Затем при определении объекта OptionMenu() список значений дополняется. При выборе элемента списка изменяется значение именно этой строки и для дальнейшей работы нужно его анализировать, что и делается в процедуре DrawGraph().

"Вычислительная" часть, а именно, все процедуры и функции, обеспечивающие вычисления координат точек и рисование линий, вынесена в начало текста программы.

Определение каждой пользовательской подпрограммы обеспечивается составным оператором def. Поскольку эти подпрограммы занимаются только рисованием, они не возвращают никаких значений (т.е. результаты выполнения этих подпрограмм не присваиваются никаким переменным).

Собственно подпрограмма рисования графика DrawGraph() вызывается при нажатии кнопки "Нарисовать", и имя этой подпрограммы является командой, которая сопоставляется кнопке.

Эта подпрограмма берёт значение из списка (метод get()), выбирает первый символ получившейся строки и в зависимости от этого символа вызывает другие подпрограммы для построения конкретных графиков с установленными масштабными коэффициентами.

Перед рисованием следующего графика математической функции холст очищается командой canvas.delete("all").

Для построения графика каждой функции вычисляются собственные масштабные коэффициенты, поэтому их вычисление включено в код соответствующей подпрограммы. Кроме того, для графика нужны целые значения координат, поэтому в каждой подпрограмме выполняются соответствующие преобразования с помощью функции int().

Для каждого графика требуется нарисовать оси, и действия по рисованию осей также вынесены в отдельные подпрограммы.

Таким образом, оказывается, что программу нужно читать "с конца", и писать тоже.

## Лабораторная работа №4 Конструкторы и деструкторы, работа с атрибутами

1. Создайте файл **lab_04_01.py**:

```
import time

class Ticket:

    def __init__(self, date, name,deadline):
        self.createDate = date
        self.owner = name
        self.deadline = deadline

    def __del__(self):
        print("Delete                                ticket:
",time.asctime(self.createDate))

    def display(self):
        print("Ticket:")
        print("                                createDate:
",time.asctime(self.createDate))
        print(" owner: ",self.owner)
        print("                                  deadline:
",time.asctime(self.deadline))

# создание объект класса
ticket1 = Ticket(time.localtime(),"Ivan Ivanov", \
                    time.strptime("17.12.2017",
"%d.%m.%Y"))
# вызов метода
ticket1.display()
# получение значения атрибута
print("Owner: ", ticket1.owner)
print("Owner(getattr): ", getattr(ticket1,"owner"))
# проверка наличия атрибута
print("hasattr: ", hasattr(ticket1,"owner"))
setattr(ticket1,"owner","Alexei Petrov") # установка
значения атрибута
print("Owner(setattr): ", ticket1.owner)
#   delattr(ticket1,"owner")   #   удаление   значения
атрибута
# print("delattr: ", ticket1.owner)
```

```
# del ticket1 # удаление объекта
# print(ticket1)
```

Удостоверьтесь в работоспособности программы, запустив ее через терминал. Ознакомьтесь с выведенной информацией. Пример результата выполнения программы приведен ниже:

```
Ticket:
 createDate:  Thu Mar  9 14:53:55 2017
 owner:  Ivan Ivanov
 deadline:  Sun Dec 17 00:00:00 2017
Owner:  Ivan Ivanov
Owner(getattr):  Ivan Ivanov
hasattr:  True
Owner(setattr):  Alexei Petrov
Delete ticket:  Thu Mar  9 14:53:55 2017
```

**Классы и объекты классов**

Python является объектно-ориентированным языком программирования. В объектно-ориентированном программировании основными понятиями являются «класс» и «объект» класса. Класс представляет собой категорию, к которой можно отнести множество объектов, обладающих характерными признаками и выполняющих одинаковые действия. Объявление класса происходит с использованием ключевого слова **`class`** и указанием имени класса (имена классов начинаются с большой буквы). Класс может содержать атрибуты и методы (функции, описывающие действия его объектов). В языке Python объявление класса происходит следующим образом:

```
class Ticket:

    def __init__(self, date, name, deadline):
        pass

    def __del__(self):
        pass

    def display(self):
        pass
```

В качестве первого аргумента методов выступает ссылка на текущий объект класса **self** (аналогичный **this** в большинстве объектно-ориентированных языках программирования). При вызове методов в передаваемых аргументах объект класса не указывается, например:

```
t = Ticket(date, name, deadline)
```

где **t** – объект класса **Ticket**.

Существует два базовых метода, предусмотренных для всех создаваемых классов:

- **__init__** – конструктор класса.
- **__del__** – деструктор класса

Атрибуты класса задаются внутри его методов. Атрибуты, заданные вне методов, являются общими для всех объектов класса и могут изменять свои значения в случае их редактирования.

Вызов методов класса осуществляется с указанием имени метода через точку после имени объекта, от которого вызывается метод. Также существуют методы, предоставляющие возможности работы с атрибутами классов:

- **hasattr(obj,attr)** – проверка наличия атрибута **attr** у объекта **obj**
- **setattr(obj,attr,value)** – установка значения **value** атрибута **attr** у объекта **obj**
- **getattr(obj,attr)** – получение значения атрибута **attr** у объекта **obj**
- **delattr(obj,attr)** – удаление атрибута **attr** у объекта **obj**

Методы класса объявляются как функции внутри класса.

**Работа с подключаемой библиотекой**

Для подключения внешних библиотек в языке программирования Python используется ключевое слово **import**. Подключение самостоятельной библиотеки или модуля происходит следующим образом:

```
import name1, name2, …
```

где **name1, name2,** … – имена подключаемых библиотек/модулей.

Для подключения библиотеки из какого-либо пакета используется следующий формат записи:

**from package import name1, name2,** …

где **package** – имя пакета, **name1, name2,** … – имена подключаемых библиотек/модулей.

Для сокращения названий подключаемых библиотек можно указывать собственные названия после ключевого слова **as**:

```
from package import name1 as N
```

где **N** – собственное название подключаемой библиотеки.

Для работы с датами и временем используется библиотека **time**. Основными функциями данной библиотеки являются:

- **time()** – получение времени в системе Unix-время
- **localtime()** – получение текущего времени машины в виде кортежа
- **asctime(timeT)** – автоматическое приведение времени, представленного кортежем (**timeT**), к формату:
  **Thu Mar  9 14:53:55 2017**
- **strptime(str, format)** – создание кортежа с данными времени по строке **str**, представленной в формате **format**
- **strftime(timeT, format)** – представление кортежа с данными времени **timeT** в виде строки в формате **format**

Со списком форматов для функций **strptime()** и **strftime()**, а также списком остальных функций библиотеки можно ознакомиться по ссылке:
**https://docs.python.org/3/library/time.html**

2. Модифицируйте код программы **lab_04_01.py**. Раскомментируйте строки:
'**# delattr(ticket1,"owner") # удаление значения атрибута**
**# print("delattr: ", ticket1.owner)**'.
Объясните поведение программы. Исправьте программу с помощью стандартных функций так, чтобы ошибка не возникала.

3. Модифицируйте код программы **lab_04_01.py**. Раскомментируйте строки:
'**# del ticket1 # удаление объекта**
**# print(ticket1)**'.
Объясните поведение программы.

4. Дополните код программы **lab_04_01.py**. Получите текущее время сервера (или компьютера) и выведите его на экран в аналогичном формате: **9 Mar 2017 14:53:55**.

5. Дополните код программы **lab_04_01.py**. Создайте объект типа **time** по следующей строке с использованием соответствующей функции библиотеки **time**: **17.07.2017 10:53:00**.

6. Создайте файл **lab_04_02.py**:

```
class Worker:
    'doc class Worker'
    count = 0

    def __init__(self,name,surname):
        self.name = name
        self.surname = surname
        Worker.count += 1

    def display(self):
        print("Worker:")
        print("{} {}".format(self.name,self.surname))

w1 = Worker("Ivan", "Ivanov")
print("w1.count: ", w1.count)
w2 = Worker("Alexei", "Petrov")
print("w2.count: ", w2.count)
print("w1.count: ", w1.count)
print("Worker.count: {0} \n".format(Worker.count))
print("Worker.__name__: ", Worker.__name__)
print("Worker.__dict__: ", Worker.__dict__)
print("Worker.__doc__: ", Worker.__doc__)
print("Worker.__bases__: ", Worker.__bases__)
```

Удостоверьтесь в работоспособности программы, запустив ее через терминал. Ознакомьтесь с выведенной информацией. Результат выполнения программы приведен ниже:

```
w1.count:  1
w2.count:  2
w1.count:  2
Worker.count: 2

Worker.__name__:  Worker
Worker.__dict__:   {'count': 2, 'display': <function
Worker.display   at   0x7fbc0afb5a60>,   '__module__':
'builtins',   '__dict__':   <attribute   '__dict__'   of
'Worker'      objects>,      '__init__':      <function
Worker.__init__   at   0x7fbc0afb5ae8>,   '__weakref__':
<attribute   '__weakref__'   of   'Worker'   objects>,
'__doc__': 'doc class Worker'}
Worker.__doc__:  doc class Worker
Worker.__bases__:  (<class 'object'>,)
```

**Общие атрибуты класса**

Общие атрибуты класса доступны всем объектам данного класса. Значения, установленные для общих атрибутов класса, могут быть впоследствии отредактированы, что повлечет за собой их изменения во всех объектах класса. Общие атрибуты задаются вне методов класса:

```
class Worker:
    count = 0
```

Доступ к общим атрибутам класса может быть получен как от объектов класса, так и из самого класса, например:

```
w1.count
Worker.count
```

Описание класса (документация) добавляется в виде строки сразу после начального объявления класса:

```
class Worker:
    'doc class Worker'
    ...
```

Для получения информации о классе используются следующие базовые атрибуты:

- **__name__** – получение имени класса в виде строки
- **__dict__** – получение сведений о всех доступных методах и общих атрибутах класса
- **__doc__** – получение строки с описанием (документацией) класса
- **__bases__** – классы, от которых был унаследован данный класс. По умолчанию все классы наследуются от общего класса, представляющего объекты (**object**)

7. Дополните код программы **lab_04_02.py**. Создайте класс **Animal**, обозначив для него в конструкторе атрибуты **name**, **age** и общий атрибут **id**, который будет увеличиваться на единицу при создании каждого нового объекта класса. Определите также метод **display()**, осуществляющий вывод на экран информации по объекту класса в следующем формате, подставив нужные значения:

**Animal *id*:**
**Name: *name***
**Age: *age***

Создайте три объекта класса **Animal**, заполнив их произвольными значениями **name** и **age**. Осуществите вывод информации об объектах на экран с помощью функции **display()**.

8. Дополните код программы **lab_04_02.py**. По выведенной на прошлом шаге информации об объектах объясните поведение атрибута **id**.

9. Модифицируйте код программы **lab_04_02.py**. Измените название атрибута **id** на **count**. Модифицируйте программу так, чтобы, определив атрибут **id**, он являлся уникальным для каждого объекта, изменяясь на единицу с увеличением количества объектов класса.

10. Создайте файл **lab_04_03.py**:

```
class Geometric:
    def calculateArea(self):
        print("Calculating area")

class Square(Geometric):
    def __init__(self,a):
        self.side = a
```

```
    def _perimeter(self):
        print("Perimeter of Square {}:
{}\n".format(self.side, self.side*4))
    def calculateArea(self):
        print("Area of Square {}:
{}\n".format(self.side, pow(self.side,2)))

geom = Geometric()
geom.calculateArea()
sq = Square(5)
sq.calculateArea()
sq._perimeter()

print("Check subclass: ",
issubclass(Square,Geometric))
print("Check instance sq->Square: ",
isinstance(sq,Square))
print("Check instance sq->Geometric: ",
isinstance(sq,Geometric))
print("Check instance sq->dict: ",
isinstance(sq,dict))

print("Geometric.__bases__: ", Geometric.__bases__)
print("Square.__bases__: ", Square.__bases__)
```

Удостоверьтесь в работоспособности программы, запустив ее через терминал. Ознакомьтесь с выведенной информацией. Результат выполнения программы приведен ниже:

```
Calculating area
Area of Square 5: 25

Perimeter of Square 5: 20

Check subclass:  True
Check instance sq->Square:  True
Check instance sq->Geometric:  True
Check instance sq->dict:  False
Geometric.__bases__:  (<class 'object'>,)
Square.__bases__:  (<class 'Geometric'>,)
```

**Наследование**

Наследование между классами в языке программирования Python реализуется следующим образом:

```
class Ancestor:
    def func(self):
        pass

class Descendant(Ancestor):
    def func(self):
        pass
```

Имя родительского класса (**Ancestor**) указывается в скобках при объявлении дочернего (**Descendant(Ancestor)**).

Для показания приватности метода в начале его названия добавляется символ подчеркивания "**_**". Однако, следует отметить, что в Python не предусмотрено полностью приватных атрибутов и методов: доступ к ним в любом случае может быть получен из основного кода программы. Для того, чтобы метод/атрибут казался более закрытым, допускается добавление двух символов подчеркивания перед названием метода "**__**", но в этом случае доступ к методу/атрибуту может быть получен с использованием имени класса "**_Class__method**".

Переопределение методов аналогично многим объектно-ориентированным языкам программирования осуществляется путем указания функции родительского класса с тем же именем и аргументами внутри дочернего класса.

Для наследования атрибутов родительского класса, указанных в конструкторе родительского класса, необходимо внутри конструктора дочернего класса первоначально вызвать конструктор родительского класса, например:

```
class Ancestor:
    def __init__(self,param):
        self.param = param
    def func(self):
        pass

class Descendant(Ancestor):
    def __init__(self, param, number):
```

```
        Ancestor.__init__(self, param)
        self.number = number
    def func(self):
        pass
```

Здесь:
**Ancestor.__init__(self, param)** – строка вызова конструктора родительского класса.
Также может быть использована следующая запись:
**super(Descendant, self).__init__(param)**

Для проверки наследования класса (**Descendant**) от известного родительского класса (**Ancestor**) используется функция **issubclass(Descendant, Ancestor)**. Для проверки принадлежности объекта (**obj**) какому-либо классу (**Class**) используется функция **isinstance(obj,Class)**. Также для проверки наследования классов может быть использован атрибут **__bases__**.

11. Дополните код программы **lab_04_03.py**. Создайте класс **Circle**, унаследованный от класса **Geometric**, для которого определите атрибут **radius**. Сделайте атрибут **radius** приватным (с использованием двойного подчеркивания). Переопределите унаследованный метод **calculateArea()**, рассчитав площадь окружности (значение числа Пи доступно в библиотеке **math**).

12. Создайте файл **lab_04_04.py**. Создайте класс **Encoder**, определив для него методы **encode()** и **decode()**, аргументом которых является строка, а выходными данными – закодированная и декодированная строка соответственно. Создайте классы **HuffmanEncoder** и **LZEncoder**, унаследованные от класса **Encoder**, определив для них атрибут **compressionCoef** в конструкторах классов. Для созданных классов **HuffmanEncoder** и **LZEncoder** переопределите методы **encode()** и **decode()**, реализующие кодирование и декодирование поданных в качестве аргументов строк, используя методы Хаффмана и Лемпеля-Зива соответственно. Определите приватный метод **setCompressionCoef()** (с использованием двойного подчеркивания), осуществляющий расчет коэффициентов сжатия для методов Хаффмана и Лемпеля-Зива в соответствующих классах. Метод **setCompressionCoef()** должен вызываться при работе внутри класса в методе **encode()**. Определите общедоступный метод

**getCompressionCoef()**, позволяющий получить значение коэффициента сжатия. Осуществите проверку методов классов на следующей строке:

```
Python is a widely used high-level programming language for
general-purpose programming, created by Guido van Rossum and
first released in 1991.
```

13. Создайте файл **lab_04_05.py**. Создайте класс **Person**, определив для него в конструкторе атрибуты **firstname**, **lastname** и **age**, а также метод **display()**, выводящий информацию об объекте класса на экран. Создайте класс **Student**, унаследованный от **Person**, определив для него дополнительные атрибуты **studentID**, являющийся уникальным номером для каждого объекта класса, и **recordBook**, содержащий информацию о количестве пятерок, четверок, троек и двоек студента в виде списка. В классе **Student** дополните метод **display()** выводом значений атрибутов **studentID** и **recordBook** на экран. Создайте три объекта класса **Student** для проверки работы методов и выведите информацию о них на экран.

14. Дополните код программы **lab_04_05.py**. Создайте класс **Professor**, унаследованный от **Person**, определив для него дополнительные атрибуты **professorID**, являющийся уникальным номером для каждого объекта класса, и **degree**, содержащий информацию о научной степени в виде строки. В классе **Professor** дополните метод **display()** выводом значений атрибутов **professorID** и **degree** на экран. Создайте три объекта класса **Professor** для проверки работы методов и выведите информацию о них на экран.

15. Создайте файл **lab_04_06.py**. Создайте класс **HammingEncoder**, конструктор которого принимает на вход количество информационных разрядов **dataBits**, на основе которых рассчитывает значение количества контрольных разрядов – **controlBits**. **dataBits** и **controlBits** являются атрибутами класса. Реализуйте метод **encode(str)**, служащий для кодирования строки двоичных символов **str** с использованием кода Хэмминга с установленными параметрами **dataBits** и **controlBits**. Реализуйте метод **decode(str)**, служащий для определения кода ошибки закодированной строки **str**.

16. Создайте файл **lab_04_07.py**. Создайте следующие классы и реализуйте в них указанные методы:

- Класс **Row**
  Атрибуты:
  - **id** – идентификатор строки.
  - **collection** – список значений переменных для текущего значения функции.
  - **value** – значение функции.

  Методы:
  - **__init__(collection, value)** – конструктор класса, принимающий на вход значения **collection** и **value**.
- Класс **Table**
  Атрибуты:
  - **rows** – список объектов класса **Row**.
  - **rowsNum** – количество строк таблицы.

  Методы:
  - **__init__(rowsNum)** – конструктор класса, принимающий на вход значение **rowsNum**.
  - **addRow(row)** – добавление строки (объект **row** класса **Row**) в список. Если в списке уже находится строка с таким же идентификатором, должна выдаваться ошибка.
  - **setRow(row)** – изменение строки (объект **row** класса **Row**). Если в списке нет строки с таким же идентификатором, должна выдаваться ошибка.
  - **getRow(rowId)** – получение строки с идентификатором **rowId**. Возвращается объект класса **Row**.
  - **display()** – вывод таблицы на экран в следующем формате:

```
id  x1  x2   f(x1,x2)
1   0   0  |    1
2   0   1  |    0
3   1   0  |    0
4   1   1  |    1
```

- Класс **LogicFunction**
  Атрибуты:
  - **variablesNum** – количество переменных функции
  - **table** – объект класса **Table**. Таблица истинности логической функции.

  Методы:
  - **__init__(variablesNum,table)** – конструктор класса, принимающий на вход значения **variablesNum** и **table**.

- **`getExpression()`** – вычисляет и возвращает минимальную формулу логической функции.
- **`getTable()`** – получение значения **`table`**.
- **`printTable()`** – вывод значения **`table`** на экран.

# Проект на Django

- Реализовать регистрацию учетных пользователей на сайте (подтверждение через email отдельный +)
- Реализовать распределение ролей пользователей (user, admin)
- Реализовать возможность для пользователей оставлять комментарии, добавлять аватарки
- Для admin реализовать редактирование и удаление комментариев пользователя, блокировку пользователя
- Реализация не менее 4 шаблонов, использовать CSS
- Продумать общий стиль ресурса, за дизайн отдельный +
- Захостить сайт, например, на Heroku, за дополнительный +

# Тестовые задания

# Базовые понятия и определения

**1. Процесс разделения задачи на более мелкие фрагменты, которые бы решали конкретные подзадачи называют:**

- декомпозиция
- дефрагментация
- декорирование
- диверсификация
- деинкапсуляция

**2. Основой объектно-информационной модели являются:**

- объекты
- модели
- информация
- реляционные таблицы
- нереляционные таблицы

**3. Основой объектно-информационной модели являются:**

- группы
- классы
- атрибуты
- сущности
- таблицы

**4. Структурное отношение, показывающее, что объекты одного типа логически связаны с объектами другого типа называют:**

- связывание
- ассоциация
- инкапсуляция
- имплементация
- нет правильного ответа

**5. Отношение типа «является частью» («is-part-of»), когда объект-целое состоит из нескольких объектов-частей называют:**

- декомпозиция
- агрегирование
- объединение
- ассоциация
- имплементация

**6. Как называют специальные методы для создания объектов класса:**

- инициализаторы
- конструкторы
- активаторы

- сеттеры
- геттеры

**7. Как называют специальный метод класса, занимающийся уничтожением объектов:**

- коллектор
- деструктор
- деактиватор
- деинициализатор
- нет правильного ответа

**8. Как называют сокрытие отдельных деталей внутреннего устройства класса от внешних по отношению к нему объектов или пользователей:**

- полиморфизм
- инкапсуляция
- композиция
- наследование
- композиция

**9. Наиболее «строгий» модификатор доступа это:**

- private
- protected
- public
- local
- global

**10. Механизм, который позволяет создавать новые классы на основе существующих, используя их структурные и поведенческие характеристики:**

- абстракция
- инкапсуляция
- полиморфизм
- наследование
- агрегирование

**11. Способность использовать общий интерфейс для нескольких типов данных называют:**

- абстракция
- инкапсуляция
- полиморфизм
- наследование
- интроспекция

**12. Способность объекта во время выполнения получить тип, доступные атрибуты и методы, а также другую информацию, необходимую для выполнения дополнительных операций с объектом называют:**

- нет правильного ответа

- агрегирование
- полиморфизм
- интроспекция
- инкапсуляция

**13. В какой момент начинается исполнение конструктора класса:**

- при создании каждого экземпляра класса
- при каждом доступе к атрибутам класса
- при создании первого экземпляра класса
- при создании объекта класса в памяти
- при первом доступе к атрибутам класса

**14. При помощи чего можно получить список методов и атрибутов объекта x ?:**

- root(x)
- dir(x)
- dict(x)
- info(x)
- type(x)

**15. Какой модуль способен преобразовывать практически произвольный объект Python из памяти в строку байтов, которую позже можно использовать для воссоздания первоначального объекта в памяти ?:**

- нет правильного ответа
- frame
- byte
- shelve
- pickle

**16. Процесс превращения исходного кода в нечитаемый для человека с сохранением его функциональности называется:**

- имплементация
- интроспекция
- рефакторинг
- обфуркация
- обфускация

**17. Полиморфизм в Python базируется на:**

- интерфейсах
- декораторах
- сигнатурах вызова
- переприсваивании атрибутов
- нет правильного ответа

**18. Совокупность внедренных объектов, отражающих взаимосвязь между частями называют:**

- композиция

- агрегирование
- фабрика объектов
- контейнер
- инсталляция

**19. Служебная функция, которая проверяет, принадлежит ли объект определенному типу или нет это:**

- isinstance
- is
- isinstanceof
- type
- typeof

**20. Положительный результат выполнения проверки на совместимость типов оператором isinstance означает, что:**

- переменная либо относится к указанному классу, либо относится к классу, являющемуся потомком указанного
- переменная относится к указанному классу
- переменная относится к классу, потомком которого является указанный
- в Python нет такого оператора
- нет правильного ответа

**21. Делегирование обычно в Python подразумевает:**

- композицию
- агрегирование
- фабрику объектов
- контейнеры
- контроллеры

**22. Какой алгоритм поиска атрибутов являлся стандартом до Python 3.x ?**

- LRDF
- DFLR
- MRO
- DFMRO
- MLO

**23. Какое ключевое слово используется для вызова методов родительского класса?**

- super
- parent
- upper
- sub
- obj

**24. Что относится к основным принципам ООП ?**

- Инкапсуляция, полиморфизм, делегирование, абстракция

- Инкапсуляция, полиморфизм, наследование, абстракция
- Полиморфизм, разделение интерфейса, наследование, абстракция
- Инкапсуляция, наследование, абстракция, открытость/закрытость
- Полиморфизм, разделение интерфейса, делегирование, наследование

**25. Какой параметр обязательно принимает в себя метод экземпляра?**

- тип объекта
- атрибут класса
- таких аргументов нет
- сам экземпляр объекта
- название класса

**26. Метакласс - это:**

- абстрактный класс
- класс, объекты которого являются объектами других классов
- класс, объекты которого являются классами
- класс, объекты которого являются функциями
- верхний класс в иерархии классов

**27. Метаклассами являются следующие встроенные классы:**

- object
- metaclass
- type
- в языке Python нет встроенных метаклассов
- нет правильного ответа

**28. Методы, определенные в метаклассе содержатся в пространстве имен:**

- метакласса
- класса и экземпляра класса
- метакласса и класса
- метакласса, класса и экземпляра класса
- в метаклассе нельзя определять методы

**29. Метакласс класса C можно определить по:**

- атрибуту класса C.__metaclass__
- вызвав функцию type(C())
- вызвав функцию type(C)
- вызвав функцию class(C)
- нет правильного ответа

**30. Какой стандарт PEP описывает соглашение о том как писать код на Python ?**

- PEP1
- PEP8
- PEP12
- PEP36

- PEP6

## Атрибуты

**31. Результат выполнения приведенного ниже кода**

```
class Q:
    attr = 0

    def __init__(self):
        attr = 1

print(Q.attr, Q().attr):
```

- 1 1
- 1 0
- 0 1
- 0 0
- Error

**32. Результат выполнения приведенного ниже кода**

```
class Pet:
  def __init__(self):
    self.name = "Джек"
    print(self.name, end=' ')
  def say(self):
    print("Привет!")

class Dog(Pet):
  def say(self, name):
    print(self.name, end=' ')

p = Dog()
p.say("Белка")
p.say("Каштанка")
```

:

- Error
- Белка Каштанка
- Джек Джек
- Джек Джек Джек
- Джек Белка Каштанкаr

**33. Результат выполнения приведенного ниже кода**

```
class Human:
    name = "Vasya"

man = Human()
Human.name = "Vlad"
man.name = "Petr"
setattr(Human, "name", "Ivan")
print(Human.name)
```

:

- Vasya
- Petr
- Vlad
- Ivan
- None

**34. Результат выполнения приведенного ниже кода**

```
class Foo:
    value = 0
    def __init(self):
        self.value += 1

sample = Foo()
print(sample.value)
```

:

- 2
- Error
- None
- 0
- 1

**35. Результат выполнения приведенного ниже кода**

```
class Foo:
    value = 0
    def __init(self):
        self.value += 1

sample = Foo()
print(sample.value)
```

:

- 2
- Error
- None
- 0
- 1

**36. Результат выполнения приведенного ниже кода**

```
class A:
    val = 1

    def foo(self):
        A.val += 2

    def bar(self):
        self.val += 1


a = A()
b = A()

a.bar()
a.foo()

print(a.val, b.val)
```

:

- 1 1
- 3 1
- 3 2
- 2 3
- 2 1

**37. Результат выполнения приведенного ниже кода**

```
class Test:
    test = None
print(Test.test)
```

:

- None
- False
- Error
- 0
- test

**38. Результат выполнения приведенного ниже кода**

```
class A:
    val = 1

    def foo(self):
        A.val += 2

    def bar(self):
        self.val += 1


a = A()
b = A()

a.bar()
a.foo()

c = A()

print(c.val)
```

:

- 1
- 0
- 2
- 3
- 4

**39. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __str__(self):
        return str(self.b)

f = Foo()
f.b = 4
print(f)
```

:

- b
- 0
- None
- 4
- Error

**40. Какие цифры в столбик выведет приведенный ниже код**

```
class A:
    def __init__(self, val=0):
        self.val = val

    def add(self, x):
        self.val += x

    def print_val(self):
        print(self.val)


a = A()
b = A(2)

a.add(2)
b.add(2)

a.print_val()
b.print_val()
```

:

- 0 4
- 2 2
- 4 2
- 2 4
- 0 2

**41. Результат выполнения приведенного ниже кода**

```
class A:
    def __init__(self, id):
        id = 555
        self.id = 222
        self.id = id

print(A(111).id)
```

:

- 111
- 555
- 222
- None
- нет правильного ответа

**42. Результат выполнения приведенного ниже кода**

```
class A:
    val = 5
    def __init__(self, val=0):
        self.val = val


class B(A):
    val = 2


B.val = 4
a = B()
setattr(B, "val", 3)
print(a.val)
```

:

- 4
- 5
- 0
- 3
- 2

**43. Результат выполнения приведенного ниже кода**

```
class Foo:
    n = 5
    def __init__(self, n):
        self.n = n // 2
foo = Foo(7)
print(foo.n)
```

:

- 7
- 5
- 3
- 4
- 2

**44. Результат выполнения приведенного ниже кода**

```
class Foo:
    x = 1
    def __init__(self):
        self.y = 2
foo = Foo()
foo.z = 3
print(len(foo.__dict__))
```

:

- 1
- 2
- 3

- 0
- None

**45. Результат выполнения приведенного ниже кода**

```
class Foo:
    x = 1
    def __init__(self):
        self.x = 2

print(Foo().x, Foo.x)
```

:

- 1 1
- 2 1
- 2 2
- 1 2
- Error

**46. Результат выполнения приведенного ниже кода**

```
class Foo:
    x = 1
    def __init__(self):
        self.y = 2

a = Foo()
Foo.x = 3
Foo.y = 4
print(a.x, a.y)
```

:

- 1 2
- 3 2
- 3 4
- 2 3
- Error

**47. Результат выполнения приведенного ниже кода**

```
class Foo:
    x = 2

a = Foo()
a.x = 1
type(a).x = 0
print(Foo.x)
```

:

- 1
- 0
- 2
- None
- Error

## «Магические» методы, перегрузка методов, полиморфизм

**48. Выберите «магический» метод являющийся конструктором:**

- __add__
- __repr__
- __init__
- __str__
- __self__

**49. Какой из методов отвечает за вывод и отображение объектов:**

- __print__
- __view__
- __init__
- __str__
- __self__

**50. Результат выполнения приведенного ниже кода**

```
class A:
    def __init__(x, y, z):
        print(y)

A(5,3)
```

:

- None
- 5 3
- 0
- 3
- 5

**51. Результат выполнения приведенного ниже кода**

```
class Complex:
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __repr__(self):
        return f'Complex({self.real}, {self.imag})'

    def __str__(self):
        return f'{self.real} + {self.imag}i'

t = Complex(10, 20)

print(t)
```

:

- 20 + 10i
- 10 + 20i

- Complex(10, 20)
- [Complex(10, 20), 1, 2]
- нет правильного ответа

**52. Что является эквивалентом «магического» метода __eq__(self, other) ?:**

- нет правильного ответа
- self = other
- self <> other
- self != other
- self == other

**53. Какой "магический" метод следует переопределить, чтобы по экземпляру класса можно было пройтись в цикле for ?:**

- __yield__
- __next__
- __cycle__
- __for__
- __iter__

**54. Результат выполнения приведенного ниже кода**

```
""" Функция для суммирования """
def sum_numbers(a, b): # объявляем функцию
    """Сложение двух чисел"""
    return a + b
print(sum_numbers.__doc__)
```

:

- Сложение двух чисел
- объявляем функцию
- Функция для суммирования
- Error
- None

**55. Результат выполнения приведенного ниже кода**

```
class Foo:
    obj=0
    def __add__(self, x):
        return self.obj+2*x
o = Foo()
o += 1
print(o)
```

:

- 2
- 3
- 0
- 1
- Error

**56. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __new__(cls,*dt,**mp):
        print ('new', end=' ')
    def __init__(self):
        print ('init', end=' ')
o = Foo()
```

:

- new
- init
- new init
- init new
- Error

**57. Результат выполнения приведенного ниже кода**

```
class Foo:
    obj=0
    def __new__(cls,*dt,**mp):
        print (cls, end=' ')
        return object.__new__(cls,*dt,**mp).obj
    def __init__(self):
        self.obj+=2
        print (self, end=' ')
    def __str__(self, x):
        return obj
o = Foo()
print (o, end=' ')
```

:

- <class '__main__.Foo'> 0
- <class '__main__.Foo'> 2 2
- 0 2 2
- 2 2
- Error

**58. Результат выполнения приведенного ниже кода**

```
class Foo:
    obj=0
    def __new__(cls,*dt,**mp):
        print ('1')
        return object.__new__(cls,*dt,**mp).obj
    def __init__(self):
        print ('2')
o = Foo()
print (type(o))
```

:

- 1 2 <class '__main__.Foo'>
- 2 <class '__main__.Foo'>
- 1 <class 'int'>
- 2 1 <class 'int'>

- Error

**59. Результат выполнения приведенного ниже кода**

```
class Foo:
    obj=2
    def __init__(self):
        self.obj=self+3
    def __add__(self, x):
        return self.obj+2*x
x = Foo()
print(x.obj)
```

:

- 2
- 5
- 8
- 11
- 3

**60. Если в классе определены два метода с одинаковыми именами и разными списками параметров, то ?**

- при выполнении скрипта будет сгенерирована ошибка
- будет сгенерировано предупреждение, второе определение заменит первое
- не будет сгенерировано ни предупреждения, ни ошибки; второе определение заменит первое
- не будет сгенерировано ни предупреждения, ни ошибки; вызов того или иного метода будет зависеть от типа и количества указанных при вызове параметров
- будет сгенерировано предупреждение; вызов того или иного метода будет зависеть от типа и количества указанных при вызове параметров

**61. Результат выполнения приведенного ниже кода**

```
class Foo:
    value = 12
    def __truediv__(self, x):
        return self.value/x*2
class Bar(Foo):
    pass
x = Bar()
x = x/3
x = x/2
print(x)
```

:

- 2.0
- 3.0
- 4.0
- 1.0
- Error

**62. Результат выполнения приведенного ниже кода**

```
class Foo:
    __value__ = 0
    def __add__(self, x):
        y = Foo()
        y.__value__=self.__value__+x*2
        return y
    def __repr__(self):
        return str(self.__value__)
class Bar(Foo):
    def __repr__(self):
        return str(self.__value__+1)
res = Bar()
res = res + 3
print(res)
```

:

- 7
- 6
- 1
- 3
- 0

**63. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, value): self.__val__=value
    def __sub__ (self, x): return Foo(self.__val__-x)
    def __add__ (self, x): return Foo(self.__val__+x)
class Bar(Foo):
    def __sub__ (self, x): return Bar(self.__val__-x*2)
    def __mul__ (self, x): return Bar(self.__val__*x*2)
class Baz (Bar):
    def __mul__ (self, x): return Baz(self.__val__*x)
res = Baz(1)
res*=2
res-=4
res+=3
print(res.__val__)
```

:

- -1
- -3
- 0
- 2
- 1

**64. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, value): self.__val__=value
    def __sub__ (self, x): return Foo(self.__val__-x)
    def __add__ (self, x): return Foo(self.__val__+x)
class Bar(Foo):
    def __sub__ (self, x): return Bar(self.__val__-x*3)
    def __mul__ (self, x): return Bar(self.__val__*x*3)
class Baz (Bar):
    def __sub__ (self, x): return Baz(self.__val__-x*2)
x = Baz(2)
x*=2
x-=1
x+=3
print(x.__val__)
```

:

- 13
- 12
- 9
- 10
- 6

**65. В языке Python метод является абстрактным, если:**

- перед его определением стоит ключевое слово abstract
- его имя начинается с двойного подчеркивания
- если он помечен как @abstractmethod
- в языке Python не существует встроенной программной реализации абстрактных методов
- о его имя начинается с одного подчеркивания

**66. Результат выполнения приведенного ниже кода**

```
class Foo:
    def meth (self, x=1):
        return x*2
    def meth (self, *x):
        s=0
        for i in x: s+=i
        return s
x = Foo()
print(o.meth(-1)+o.meth(2))
```

:

- 2
- 1
- 4
- 0
- 3

**67. Результат выполнения приведенного ниже кода**

```
class Foo:
    def meth(self, x=1):
        return x*2
    def meth(self, x, *y):
        s=x
        for i in y: s+=i*2
        return s
x = Foo()
print(x.meth(3)+x.meth(4, 5));
```

- 14
- 17
- 20
- 11
- 9

**68. Результат выполнения приведенного ниже кода**

```
class Foo:
    def meth(self, x):
        return x*2
    def meth(self, x, y=-2):
        return x+y
x=Foo()
print (x.meth(3)+x.meth(4))
```

- 3
- 8
- 9
- 14
- 0

**69. Результат выполнения приведенного ниже кода**

```
class Foo:
    def meth(self, x):
        return x*2
    def meth (self, x, y=3):
        return (x+y)*3
x=Foo()
print(x.meth(1)+x.meth(1, 2));
```

- 4
- 21
- 11
- 14
- Error

**70. Результат выполнения приведенного ниже кода**

```
class Foo:
    def meth(self, x=0):
        return x*2
    def meth(self, x, y=2):
        return (x+y)/2
x=Foo()
print(x.meth()+x.meth(2))
```

- 0
- 2
- 1
- 3
- Error

**71. Результат выполнения приведенного ниже кода**

```
class Foo:
    def meth(self, x=0):
        return x*2
    def meth(self, *x):
        s=0
        for i in x: s+=i
        return s
x=Foo()
print(x.meth(-1)+x.meth(2,3))
```

- Error
- 2
- 1
- 3
- 4

**72. Результат выполнения приведенного ниже кода**

```
class Foo(str):
    def __init__(self, x):
        self.__val__=x
    def __add__(self, val):
        return Foo(int(self.__val__)+int(val.__val__))
    def __str__(self):
        return str(self.__val__)
print((Foo('1')+Foo('2'))*2)
```

- Error
- 24
- 6
- 1212
- 33

**73. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    def __init__(self, x):
        self.__val__=x
    def __add__(self, val):
        return Foo(str(self.__val__)+str(val.__val__))
    def __str__(self):
        return str(self.__val__)
print((Foo('1')+Foo('2'))*2)
```

:

- Error
- 24
- 6
- 1212
- 33

**74. Результат выполнения приведенного ниже кода**

```
class Foo(float):
    def __init__(self, x):
        self.__val__=x
    def __add__(self, val):
        return Foo(int(self.__val__)+int(val.__val__))
    def __str__(self):
        return str(self.__val__)
print((Foo('1')+Foo('2'))*2)
```

:

- 6
- 6.0
- 24
- 24.0
- Error

**75. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, x):
        self.__val__=x
    def __add__(self, val):
        return Foo(self.__val__+val.__val__)
    def __str__(self):
        return str(self.__val__)
print(Foo(1)+Foo(2), Foo('1')+Foo('2'))
```

:

- 3 3
- 3 12
- 12 3
- 12 12
- Error

**76. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    def __init__(self, x):
        self.__val__=x
    def __add__(self, val):
        return Foo(val.__val__.__class__(self.__val__)+val.__val__)
    def __str__(self):
        return str(self.__val__)
print (Foo(1)+Foo('2'), Foo('1')+Foo(2))
```

:

- 3 3
- 12 3
- 12 12
- 3 12
- Error

**77. Результат выполнения приведенного ниже кода**

```
class str:
    def __new__ (self, lst):
        res=0
        for i in lst: res+=1
        return res
print(str(range(5, 8)))
```

:

- 123
- 3
- 567
- [1, 2, 3]
- [5, 6, 7]

**78. Результат выполнения приведенного ниже кода**

```
def add(x, y):
    return x+y
class add:
    def __call__(self, x, y):
        return (x+y)*2
f = add()
print(f(1, 2))
```

:

- 3
- 6
- None
- 12
- Error

**79. Результат выполнения приведенного ниже кода**

```
def add(x, y):
    return x+y
class add:
    def __init__(self, x, y):
        return (x+y)*2
print(add(1, 2))
```

:

- 3
- 6
- None
- 12
- Error

**80. Результат выполнения приведенного ниже кода**

```
class len:
    def __call__(self, lst):
        res=0
        for i in lst: res+=1
        return res
print(len(range(5, 8)))
```

:

- 3
- 4
- None
- 1
- Error

**81. Результат выполнения приведенного ниже кода**

```
class len:
    def __call__ (self, lst):
        res=0
        for i in lst: res+=1
        return res
foo = len()
print(foo(range(8)))
```

:

- None
- 7
- 8
- 0
- Error

**82. Результат выполнения приведенного ниже кода**

```
class str:
    def __init__(self, lst):
        res=0
        for i in lst: res+=1
        return res
print(str(range(0, 4)))
```

:

- None
- [0, 1, 2, 3]
- 0123
- 01234
- Error

**83. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, v):
        self.__val__ = v
    def __add__(self, y):
        return Foo(self.__val__ + y - 1)
    def __repr__ (self):
        return str(self.__val__)
x = Foo(1)
print(x+2, 2+x)
```

:

- 2 2
- 2 3
- 3 2
- 3 3
- Error

**84. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, v):
        self.__val__ = v
    def __isub__(self, y):
        return self.__val__ - y + 1
    def __sub__(self, y):
        return self.__val__ - y  + 1
    def __repr__ (self):
        return str(self.__val__)
x = Foo(3)
x-=2
print(x, x-1)
```

:

- 1 0
- 1 1
- 2 1
- 2 2

- Error

**85. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, v):
        self.__val__ = v
    def __rsub__(self, y):
        return self.__val__ - y + 1
    def __isub__(self, y):
        return Foo(self.__val__ - y + 1)
    def __repr__ (self):
        return str(self.__val__)
x = Foo(5)
x-=3
print(x, x-1)
```

:

- 2 1
- 3 2
- Error
- 2 2
- 3 3

**86. Какие классы будут указаны в результате выполнения приведенного ниже кода**

```
class Foo(int):
    def __init__(self, v):
        self.__val__ = v
    def __mul__(self, y):
        self.__val__ *= y
        return self
x = Foo(2)
print(type(x*5), type(5*x))
```

:

- Foo Foo
- int int
- Foo int
- int Foo
- Error

**87. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    def __init__(self, v):
        self.__val__ = v
    def __imul__(self, y):
        self.__val__ *= y
        return self
    def __rmul__(self, y):
        self.__val__ *= y
        return self
    def __repr__ (self):
        return str(self.__val__)
x = Foo(2)
print(x*3, 3*x)
```

:

- 6 2
- 2 6
- 6 6
- 2 2
- Error

**88. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    def __init__(self, v):
        self.__val__ = v
    def __add__(self, y):
        return Foo(self.__val__ + y - 1)
    def __repr__ (self):
        return str(self.__val__)
x = Foo(1)
print(x+4, 4+x)
```

:

- 5 5
- 5 4
- 4 5
- 4 4
- Error

**89. Результат выполнения приведенного ниже кода**

```
class Vector:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __add__(self, other):
    return Vector(self.x +other.x, self.y +other.y)
  def __str__(self):
      return f"({self.x}, {self.y})"

print(Vector(1,2)+Vector(1,2))
```

:

- (3, 3)
- 2 4
- (2, 4)
- 3 3
- Error

**90. Результат выполнения приведенного ниже кода**

```
class Foo:
    x = 10
    def print(self):
        print(self.x)

f = Foo()
Foo.print(f)
```

:

- None
- 0
- 10
- 1
- Error

**91. Результат выполнения приведенного ниже кода**

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

    """Doc"""

print(Foo.__doc__)
```

:

- Doc
- Foo.__doc__
- None
- x
- Error

**92. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, seq):
        self.seq = seq

    def __len__(self):
        return len(self.seq)

f = Foo([])
print(bool(f))
```

:

- 0
- True
- False
- None
- Error

**93. Результат выполнения приведенного ниже кода**

```
class MySuperClass:
    def __init__(self):
        self.i = 0
    def __call__(self):
        self.i += 1
        return self

print(MySuperClass()()()().i)
```

:

- 4 Error
- 3 Error
- 3
- 4
- Error

**94. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __new__(cls, *args, **kwargs):
        print(1, end=' ')
        return None
    def __init__(self):
        print(2, end=' ')

a = Foo()
print(3 if a is None else 4)
```

:

- 2 4
- 1 2 3
- 1 3
- 2 3

- 1 4

**95. Результат выполнения приведенного ниже кода**

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

    def __add__(self, other):
        return Vec(self.x + other.x , self.y + other.y)

a = Vec(2, 1)
b = Vec(3, 0)

c = a + b
print(c.x + c.y)
```

:

- 5
- 3
- 6
- 1
- Error

## Наследование

**96. Результат выполнения приведенного ниже кода**

```
class Base:
    def __init__(self):
        self.val = 0

    def add_one(self):
        self.val += 1


class Derived(Base):
    def add_one(self):
        self.val += 10


a = Derived()
b = Base()
a.add_one()
b.add_one()
print(a.val, b.val)
```

:

- 0 0
- 10 0

- 1 0
- 0 1
- 10 1

**97. Результат выполнения приведенного ниже кода**

```
class A:
    counter = 10
    def __init__(self):
        self.counter =  self.counter // 5

class B(A):
    counter = 15
    def init(self):
        self.counter += 30

beta = B()
print(beta.counter)
```

:

- 2
- 40
- 45
- 30
- 3

**98. Результат выполнения приведенного ниже кода**

```
class A:
    def __init__(self):
        self.attr = 1

class B(A):
    def __init__(self):
        super().__init__()

class C(A):
    def __init__(self):
        self.attr = 2

class D(B, C):
    def __init__(self):
        super().__init__()


d = D()
print(d.attr)
```

:

- 2
- 0
- None

- 1
- 3

**99. Результат выполнения приведенного ниже кода**

```
class A:
   def foo(self):
      print("A")

class B(A):
   pass

class C(A):
   def foo(self):
      print("C")

class D:
   def foo(self):
      print("D")

class E(B, C, D):
   pass

E().foo()
```

:

- C
- A
- None
- D
- Error

**100. Какая последовательность по иерархии поиска будет корректным порядком разрешения методов для класса E?**

```
class A:
    pass

class B(A):
    pass

class C:
    pass

class D(C):
    pass

class E(B, C, D):
    pass
```

:

- E, B, A, C, D, object

- E, B, C, D, A, object
- E, B, D, C, A, object
- E, B, A, D, C, object
- нет правильного ответа

**101. Результат выполнения приведенного ниже кода**

```
class A:
    val = 0
class B:
    val = 1
class C(A):
    val = 3
class D(C, A, B):
    def __str__(self):
        return str(self.val)

print(D())
```

**:**

- 0
- 3
- None
- 1
- Error

**102. С помощью чего можно отследить порядок поиска атрибутов в Python 3.x ?**

- __mro__
- __find__
- __dir__
- __search__
- __dict__

**103. Как будет выглядеть поиск по иерархии для класса D**

```
class A:
    pass
class B(A):
    pass
class C:
    pass
class D(A, C):
    pass
```

**?**

- [<class '__main__.D'>, <class '__main__.A'>, <class '__main__.C'>, <class 'object'>]
- [<class '__main__.D'>, <class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>]
- [<class '__main__.D'>, <class '__main__.A'>, <class '__main__.C'>]
- [<class '__main__.D'>, <class '__main__.A'>, <class '__main__.C'>, <class '__main__.B'>]
- [<class '__main__.D'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]

- [<class '__main__.D'>, <class '__main__.A'>, <class 'object'>]

**104. Результат выполнения приведенного ниже кода**

```
class A:
    def a(self):
        return 1


class B(A):
    def b(self):
        return 2
b = B()
print(isinstance(b, A))
```

:

- True
- 0
- 1
- False
- None

**105. Результат выполнения приведенного ниже кода**

```
class Test:
    def print_text(self):
        print("Это родительский класс Test")
class Test2(Test):
    def print_text(self):
        print("Это дочерний класс Test2")
test2 = Test2()
test2.print_text()
```

:

- Это родительский класс Test
- Это дочерний класс Test2
- Error
- Это родительский класс Test Это дочерний класс Test2
- None

**106. При наследовании в языке Python:**

- подклассы наследуют от класса object
- подклассы наследуют все методы суперкласса, кроме специально отмеченных
- подклассы наследуют только специально отмеченные методы суперкласса
- методы в языке Python не наследуются
- подклассы наследуют все методы суперкласса

**107. При наследовании в языке Python порядок разрешения методов для "классических" классов (т.е. классов, не являющихся наследниками класса object) определяется следующим образом:**

- нет правильного ответа
- рассматривается последний суперкласс, если метод не найден, то рассматривается предпоследний суперкласс и т.д., если во всех суперклассах метод не найден, то рассматривается суперклассы последнего суперкласса и т.д.

- рассматривается последний суперкласс и далее его суперклассы, если метод не найден, то рассматривается предпоследний суперкласс
- рассматривается первый суперкласс, если метод не найден, то рассматривается второй суперкласс и т.д., если во всех суперклассах метод не найден, то рассматривается суперклассы первого суперкласса и т.д.
- рассматривается первый суперкласс и далее его суперклассы, если метод не найден, то рассматривается второй суперкласс

**108. При наследовании в языке Python:**

- подклассы наследуют все атрибуты суперкласса, кроме специально помеченных
- подклассы наследуют все атрибуты суперкласса, кроме приватных
- подклассы не наследуют никакие атрибуты суперкласса, кроме специально помеченных
- нет правильного ответа
- подклассы наследуют все атрибуты суперкласса

**109. Результат выполнения приведенного ниже кода**

```
class Foo:
    __val__=0
    def m1 (self): self.__val__+=1
    def m2 (self): self.__val__+=2
class Bar(Foo):
    __val__=1
    def m2 (self): self.__val__-=1
    def m3 (self): self.__val__-=2
class Baz (Bar):
    __val__=2
    def m1 (self): self.__val__+=5
x = Baz()
x.m1()
x.m2()
x.m3()
print(x.__val__)
```

:

- 4
- 5
- 3
- 6
- 2

**110.** **Результат выполнения приведенного ниже кода**

```
class Foo:
    __val__=[0]
    def m1 (self): self.__val__+=[1]
    def m2 (self): self.__val__+=[2]
class Bar (Foo):
    __val__=[-1]
    def m1 (self): self.__val__+=[3]
    def m3 (self): self.__val__+=[4]
class Baz (Bar):
    __val__=[-2]
    def m2 (self): self.__val__+=[5]
x = Baz()
x.m1()
x.m2()
x.m3()
print(x.__val__)
```

:

- [0, 1, 2, 4]
- [-2, 3, 5, 4]
- [-2, 1, 2, 4]
- [-1, 3, 2, 4]
- [-1, 1, 5, 4

**111.** **Результат выполнения приведенного ниже кода**

```
class Foo:
    def method1(self): print('1')
    def method2(self): print('2')
class Bar(Foo):
    def method2(self): print('2+')
    def method3(self): print('3+')
class Baz(Bar):
    def method3(self): print('3++')
x = Baz()
x.method1()
x.method2()
x.method3()
```

:

- 1 2 3+
- 1 2+ 3++
- 1 2+ 3+
- 1 2 3++
- Error

**112.** **Результат выполнения приведенного ниже кода**

```
class Foo:
    def method1(self):
        return 1
class Bar(Foo):
    def method1(self):
        return 3
class Baz(Bar):
    def method1(self):
        return 2
class Lo(Baz, Bar):
    __x__ = 4
x = Lo()
print(x.method1())
```

:

- 1
- 2
- 3
- Error
- 4

**113.** **Результат выполнения приведенного ниже кода**

```
class Foo:
    def method1 (self):
        return 1
class Bar(Foo):
    pass
class Baz (Foo):
    def method1 (self):
        return 2
class Lo (Bar, Baz):
    x = 3
x = Lo()
print(x.method1())
```

:

- 3
- 2
- 1
- Error
- None

**114.** **Результат выполнения приведенного ниже кода**

```
class Foo:
    def method1(self):
        return 1
class Bar(Foo):
    pass
class Baz(Bar):
    def method1(self):
        return 2
class Lo(Bar, Baz):
        x = 3
x = Lo()
print(x.method1())
```

:

- 3
- 2
- 1
- Error
- None

**115.** **Результат выполнения приведенного ниже кода**

```
x = object()
x.a = 2
print(x.__dict__)
```

:

- {2}
- {'a':2}
- 2
- Error
- None

**116.** **Результат выполнения приведенного ниже кода**

```
class A:
    x = 1
class B(A):
    pass
class C(A):
    pass

A.x = 3
B.x = 2
print(B.x, C.x)
```

:

- 1 3
- 3 1
- 2 1
- 2 3
- 2 None

## Пространство имен, область видимости

**117.** **Какие числа будут результатом выполнения приведенного ниже кода**

```
x, y = 1, 2

def foo():
    global y
    if y == 2:
        x = 2
        y = 1

foo()
print(x)
if y == 1:
    x = 3
print(x)
```

:

- 1 3
- 1 1
- 2 3
- 2 2
- 4 3

**118.** **В какой момент создается локальное пространство имен для функции?:**

- нет правильного ответа
- В момент объявления функции
- После выполнения строки с return
- В момент создания объекта функции
- В момент вызова функции

**119.** **Результат выполнения приведенного ниже кода**

```
lst = [1]
class Foo:
    lst += [2]
ob1 = Foo()
ob2 = Foo()
print(ob2.lst)
```

:

- [1]
- [1, 2]
- [1, 2, 1, 2]
- [1, 2, 2]
- 3

**120.** **Результат выполнения приведенного ниже кода**

```
tpl=1,2
class Foo:
    tpl+=(3,)
ob = Foo()
print(tpl, ob.tpl)
```

- (1, 2, 3)
- (1, 2) (1, 2, 3)
- (1, 2)
- (1, 2, 1, 2, 3)
- 3

**121.** **Результат выполнения приведенного ниже кода**

```
lst=[0]
class Foo:
    lst=lst + [1]
x = Foo()
print(lst, x.lst)
```

- [0] [0]
- [0] [0, 1]
- [0, 1] [0]
- [0, 1] [0, 1]
- Error

**122.** **Результат выполнения приведенного ниже кода**

```
class Foo:
    def value(self, x):
        return x.str
    def value(self, x):
        pass
    def value(self, x):
        return x
ob = Foo()
print(type(ob.value(0)))
```

- <class 'str'>
- <class 'int'>
- <class 'NoneType'>
- <class 'Foo'>
- Error

**123. Результат выполнения приведенного ниже кода**

```
class Foo:
    obj=0
    def obj(self):
        return 'x'
ob = Foo()
print(type(ob.obj), type(ob.obj()))
```

:

- <class 'int'> <class 'int'>
- <class 'method'> <class 'str'>
- <class 'int'> <class 'NoneType'>
- <class 'NoneType'> <class 'NoneType'>
- <class 'method'> <class 'int'>

**124. Результат выполнения приведенного ниже кода**

```
Path = str
def f(path: Path):
    return type(path)

print(f('x'))
```

:

- <class 'Path'>
- <class 'str'>
- <class 'NoneType'>
- None
- Error

**125. Результат выполнения приведенного ниже кода**

```
lst=[]
class Foo:
    obj = 0
    lst.append(0)
ob1 = Foo()
ob2 = Foo()
print(lst)
```

:

- [0] [0]
- [0]
- [0, 0, 0]
- []
- Error

**126.** **Результат выполнения приведенного ниже кода**

```
lst=[1]
class Foo:
    lst.append(2)
    print(lst, end=' ')
ob1 = Foo()
ob2 = Foo()
```

:

- [1, 2, 2]
- [1, 2]
- [1]
- [1, 2] [1, 2]
- Error

**127.** **Результат выполнения приведенного ниже кода**

```
foo = [1]
class Foo:
    bar = foo
    bar += [1]
ob = Foo()
print(foo)
```

:

- [1] [1]
- [1, 1]
- [1]
- [2]
- Error

**128.** **Результат выполнения приведенного ниже кода**

```
class Foo:
    def obj(self):
        return [0]
    def __init__(self,*dt,**mp):
        self.obj=lambda i : 0
    def __new__(cls,*dt,**mp):
        cls.obj=lambda i : "0"
        return object.__new__(cls,*dt,**mp)
ob = Foo()
print(type(ob.obj(0)))
```

:

- <class 'list'>
- <class 'int'>
- <class 'str'>
- <class 'Foo'>
- Error

**129. Результат выполнения приведенного ниже кода**

```
class Foo:
    def value(self, x):
        return [x]
    def __init__(self):
        self.value = lambda i: {'0':i}
    def value(self, x):
        return (x,)
x = Foo()
print (x.value(0))
```

:

- {0}
- {'0':0}
- (0, )
- 0
- [0]

**130. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __new__(cls,*dt,**mp):
        cls.obj=lambda i : str(i)
        return object.__new__(cls,*dt,**mp)
    def obj(self, x):
        pass
a = Foo()
print(type(a.obj()))
```

:

- <class 'str'>
- <class 'int'>
- <class 'Foo'>
- <class 'function'>
- Error

**131. Результат выполнения приведенного ниже кода**

```
class Foo:
    def method(self):
        return '012'
    val = [method]
x = Foo()
y = x.val
del x
print(y())
```

:

- Error
- None
- method
- [method]
- 012

**132. Результат выполнения приведенного ниже кода**

```
class Foo:
    val = (1,2,3)
x = {'0':Foo()}
y = x['0'].val
del x
print(y)
```

:

- (1, 2, 3)
- x[‘0’].val
- ‘0’
- 1
- Error

**133. Результат выполнения приведенного ниже кода**

```
class Foo:
    def method():
        return '012'
    val = [method]
x = Foo()
y = x.val[0]
del x
print(y())
```

:

- 012
- x.val[0]
- None
- method
- Error

**134. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __repr__(self):
        return 'class'
x = [Foo()]
y = x[0]
del x
print(y)
```

:

- пустая строка
- c
- None
- class
- Error

**135. Результат выполнения приведенного ниже кода**

```
class Foo:
    val = [1,2,3]
x = Foo()
y = x.val[0]
del x
print(y)
```

:

- пустая строка
- [1,2,3]
- None
- 1
- Error

**136. Результат выполнения приведенного ниже кода**

```
class Foo:
    def method(self):
        return 'call method'
x = (Foo().method,)
y = x[0]
del x
print(y())
```

:

- c
- call
- None
- call method
- Error

**137. Методы класса содержатся в пространстве имен:**

- класса
- класса и экземпляров класса
- класса, его метакласса и экземплярах класса
- класса и метакласса
- в глобальном пространстве имен

**138. Результат выполнения приведенного ниже кода**

```
def func(*args):
    args.append(3)
    return args
print(*func(1,2))
```

:

- (1, 2, 3)
- [1,2,3]
- 1 2 3
- 1 2
- Error

**139. Результат выполнения приведенного ниже кода**

```
def func(x,/,y):
    print(x+y)

func(1,2)
```

:

- 2
- 3
- 1
- 0.5
- 0

**140. Результат выполнения приведенного ниже кода**

```
def func(a: int, b: int)->int:
    return a+b
print(func(1.0, 2.5))
```

:

- 3
- 3.5
- 4
- None
- Error

**141. Результат выполнения приведенного ниже кода**

```
x = 1
def foo():
    x = 2
    def bar():
        nonlocal x
        x += 1
        return x
    return bar

print(foo()())
```

:

- 1
- 3
- 2
- None
- Error

**142. Результат выполнения приведенного ниже кода**

```
def a(x):
    def b():
        print(x, end=' ')
        return b
    x = 1
    return b

a(4)()()
```

:

- 4
- 1 1
- 4 4
- 1
- Error

**143. Результат выполнения приведенного ниже кода**

```
def f(a, x):
    return a.x + x

class Foo:
    def f(self, x):
        self.x = x
        return x + 2

res = Foo()
print(f(res, res.f(2)))
```

:

- 4
- 6
- 2
- 8
- Error

## Инкапсуляция

**144. Какой вариант из перечисленных гарантирует принадлежность атрибута X конкретному классу и позволяет избежать конфликта имен ?**

- self._X = 99
- self.__X = 99
- __X = 99
- _X = 99
- __self._X = 99

**145. Присваивая последовательность строковых имен атрибутов специальному атрибуту __slots__ класса мы можем:**

- нет правильного ответа

- ограничить набор допустимых атрибутов атрибутами указанными в последовательности
- расширить набор допустимых атрибутов
- исключить перечисленные в последовательности атрибуты
- удалить атрибуты с указанными в последовательности именами

**146. К каким атрибутам экземпляра X можно обратиться в коде**

```
class A:
    c = 5
    __slots__ = ['a', 'b']
    def __init__(self):
        self.a = 3

X = A()
```

**?**

- X.a, X.c
- X.a, X.b
- X.c
- X.b, X.c
- X.a

**147. Какой из перечисленных вариантов является правильным объявлением private поля ?**

- private field = 0
- field = 0
- _field = 0
- __field = 0
- _field_ = 0

**148. Результат выполнения приведенного ниже кода**

```
class Test:
    __test = 0
print(Test.__test)
```

- None
- Error
- 0
- Test.__test
- Test.0

**149. Одиночное подчеркивание в начале имени атрибута класса указывает на:**

- то, что атрибут является свойством
- то, что он приватный
- то, что он приватный и доступ к нему не может быть получен через instance._attribute
- то, что атрибут является атрибутом класса, т.е. к нему можно получить доступ без инстанциирования класса
- в случае атрибутов класса одиночное подчеркивание ничего не означает

**150. Прямой доступ к атрибуту класса нельзя получить, если:**

- перед определением атрибутом стоит идентификатор private
- если имя атрибута начинается с подчеркивания и кончается на подчеркивание
- если имя атрибута начинается с двойного подчеркивания и кончается на двойное подчеркивание
- если имя атрибута начинается с одного подчеркивания и кончается на одно подчеркивание
- в языке Python можно получить прямой доступ к любому атрибуту

**151. В языке Python встроенный метод property() используется для:**

- получения информации об объекте, метод которого вызывается
- получения информации обо всех свойствах объекта, метод которого вызывается
- реализации доступа к определенному атрибуту класса как к свойству
- реализации доступа к любым атрибутам класса как к свойствам
- нет правильного ответа

**152. В языке Python инкапсуляция достигается:**

- путем введения градаций доступности данных и методов класса, обязательных к использованию
- путем четкого разделения данных и методов класса на закрытые и открытые средствами языка
- путем соглашения между программистами об условном обозначении закрытых и открытых данных и полей
- никак не достигается
- при помощи модификаторов доступа private, public, protected

**153. Результат выполнения приведенного ниже кода**

```
class Foo:
    def method (self):
        print('1', end=' ')
    def _method_ (self):
        print('2', end=' ')
    def __method__ (self):
        print('3', end=' ')
o=Foo()
o.method()
o._method_()
o.__method__()
```

:

- 1 2 и сообщение об ошибке
- 1 2 3
- 1 и сообщение об ошибке
- сообщение об ошибке
- 1 2 3 и сообщение об ошибке

**154. Результат выполнения приведенного ниже кода**

```
class Foo:
    x=1
    __x=2
    x__=3
print(Foo.x)
print(Foo.__x)
print(Foo.x__)
```

:

- 1 2 и сообщение об ошибке
- 1 2 3
- 1 и сообщение об ошибке
- сообщение об ошибке
- 1 2 3 и сообщение об ошибке

**155. Результат выполнения приведенного ниже кода**

```
class Foo:
    def method(self):
        print('1', end=' ' )
    def __method(self):
        print('2', end=' ' )
    def method__(self):
        print('3', end=' ' )
x = Foo()
x.method()
x.method__()
x.__method()
```

:

- 1 3 и сообщение об ошибке
- 1 2 3
- 1 и сообщение об ошибке
- сообщение об ошибке
- 1 2 3 и сообщение об ошибке

**156. Результат выполнения приведенного ниже кода**

```
class Foo:
    x=1
    _x=2
    __x=3
print(Foo.x+Foo._x, end=" ")
print(Foo._x+Foo._Foo__x)
```

:

- 3 5
- 4 5
- 3 и сообщение об ошибке
- 4 и сообщение об ошибке
- 5

**157. Результат выполнения приведенного ниже кода**

```
class Foo:
    x=1
    _x_=2
    __x__=3
print(Foo.x, Foo._x_, Foo.__x__)
```

:

- 1 2 3
- 1 2
- 1, 2 и сообщение об ошибке
- 1 и сообщение об ошибке
- сообщение об ошибке

**158. Как получить __value атрибут у объекта foo**

```
class Foo:
    def __init__(self):
        self.__value = 42

foo = Foo()
```

?

- foo._Foo__value
- foo.value
- foo.__value
- нет правильного ответа
- foo.Foo__value

**159. Результат выполнения приведенного ниже кода**

```
class User:
    def __init__(self, name):
        self.__name = name
    def __str__(self):
        return str(self.__name)
user = User('Oleg')
user.__name = 'Ivan'
print(user)
```

:

- Oleg
- Ivan
- None
- Error
- name

**160. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self):
        self._x = 1
        self.__y = 2

    def display(self):
        print(self._x, self.__y)

class Test(Foo):
    def __init__(self):
        super().__init__()
        self._x = 3
        self.__y = 4

Test().display()
```

:

- 3 2
- 1 2
- 1 4
- 3 4
- Error

## Конструкторы и деструкторы

**161. Как создать конструктор класса A?**

- A(параметры конструктора)
- def __init__(параметры конструктора)
- def __A__(параметры конструктора)
- def init(параметры конструктора)
- def __init__(A, параметры конструктора)

**162. Как много конструкторов в классе может иметь Python?**

- 0
- бесконечно много
- 2
- 1
- 3

**163. Если в классе определен деструктор с двумя и более параметрами, то?**

- будет сгенерирована ошибка, т.к. деструктор не может иметь более одного параметра
- будет сгенерировано предупреждение, и такой деструктор должен вызываться только явно
- не будет сгенерировано ни предупреждения, ни ошибки; при неявном вызове деструктора значение параметра будет равно None
- предупреждения не будет сгенерировано, но такой деструктор должен вызываться только явно

- нет правильного ответа

**164. Деструктор класса задается методом с именем:**

- __des__
- __destructor__
- __del__
- __destr__
- __delete__

**165. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self):
        print ('constructor', end=' ')
    self.__del__(self)
    def __del__(self):
        print ('destructor', end=' ')
obj = Foo()
```

:

- Error
- constructor
- constructor destructor
- destructor
- destructor constructor

**166. Результат выполнения приведенного ниже кода**

```
x=0
class Foo:
    count=x
    def __init__(self):
        self.count+=1
    def __del(self):
        self.count+=1
obj = Foo()
print (obj.count)
```

:

- 1
- 0
- 2
- None
- Error

**167. Результат выполнения приведенного ниже кода**

```
class Foo:
    def foo(self):
        print('foo')
        del self
    def __del__(self):
        print('del')
obj = Foo()
obj.foo()
```

:

- foo del
- foo
- del
- None
- Error

**168. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self):
        print ('construct')
        del self
obj = Foo()
if obj: print ('exist')
```

:

- construct exist
- None
- construct
- exist
- Error

**169. Результат выполнения приведенного ниже кода**

```
x=0
class Foo:
    x = 1

foo = Foo()
foo.x = 2
del foo.x

print(foo.x)
```

:

- 0
- 1
- 2
- None
- Error

## Геттеры и сеттеры

**170. Встроенный метод __setattr__ вызывается:**

- автоматически, при попытке присвоить значение атрибута через instance.attribut
- автоматически, при попытке присвоить значение атрибута через instance.attribute, если не найден атрибут, к которому идет обращение
- автоматически,при попытке присвоить значение атрибута через instance.attribute или instance._class__attribute, если не найден атрибут, к которому идет обращение
- автоматически, при попытке присвоить значение атрибута через instance.attribute или instance._class__attribute
- только явно

**171. Результат выполнения приведенного ниже кода**

```
class Foo:
    __value__=0
    def __getattr__(self, name):
        return (name+'_'+self.__value__.__str__())
    def __setattr__(self, name, value):
        object.__setattr__(self, '__value__', value)
x = Foo()
x.a = 1
print(x.a, x.b)
```

:

- a_1 b_1
- a_1 b_0
- a_0 b_0
- 1 0
- a_1 0

**172. Результат выполнения приведенного ниже кода**

```
class Foo:
    __value__=0
    def __getattr__(self, name):
        print(name+'_'+self.__value__.__str__(), end = ' ')
    def __setattr__(self, name, value):
        object.__setattr__(self, '__value__', value)
x = Foo()
x.a = 1
print(x.a, x.b)
```

:

- a_1 b_1
- a_1 b_1 None None
- a_0 b_0
- a_1 None
- 1 None

**173. Результат выполнения приведенного ниже кода**

```
class Foo:
    __value__={}
    def __getattr__(self, name):
        if name in self.__value__:
            return self.__value__[name]
        else:
            return (name+'_atr')
    def __setattr__(self, name, value):
        self.__value__[name]=value
o = Foo()
o.a=12
print(o.a, o.b)
```

:

- a_atr None
- 12 b_atr
- 12 None
- 12 12
- None

**174. Результат выполнения приведенного ниже кода**

```
class Foo:
    __value__={}
    def __getattr__(self, name):
        if self.__value__.get(name, 'Not found'):
            return name
    def __setattr__(self, name, value):
        object.__setattr__(self, name, value)
x = Foo()
x.a = 8
print(x.a, x.b)
```

:

- 8 'Not found'
- 8 b
- a b
- a 'Not found'
- Error

**175. Результат выполнения приведенного ниже кода**

```
class Foo:
    a=1
    def __getattr__(self, name):
        return (str(self.a)+name)
class Bar(Foo):
    a=0
x = Bar()
x.a=2
print (x.a, x.b)
```

:

- 2
- 4
- 3
- a1
- 5

**176. Результат выполнения приведенного ниже кода**

```
class Foo:
    a=1
    def __getattr__(self, name):
        return (str(self.a)+name)
class Bar(Foo):
    a=0
x = Bar()
x.a=2
print (x.a, x.b)
```

:

- 2 None
- 2 2b
- 2 0
- 2 2
- 2a 2b

**177. Результат выполнения приведенного ниже кода**

```
class Foo:
    a=1
    def __setattr__(self, name, value):
        object.__setattr__(self, name, value*2)
    def __getattr__(self, name):
        return self.a + 1
class Bar (Foo):
        a=0
x = Bar()
x.a=-1
print(x.a)
```

:

- 0
- 2
- 1
- -1
- None

**178. Результат выполнения приведенного ниже кода**

```
class Foo:
    __value__={}
    def __getattr__(self, name):
        if self.__value__.has_key(name):
            return self.__value__[name]
        else:
            return (name+'_atr')
    def __setattr__(self, name, value):
        self.__value__[name]=value+1
class Bar(Foo):
    a=0
x = Bar()
x.a=5
print(x.a)
```

:

- 1
- 0
- 5
- 6
- Error

**179. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __setattr__(self, name, value):
        object.__setattr__(self, name, value+1)
class Bar(Foo):
    a=0
x=Bar()
x.a=2
print(x.a)
```

:

- 1
- 3
- 2
- 0
- None

**180. Результат выполнения приведенного ниже кода**

```
class Foo:
    __value__=1
    def __getattr__(self, name):
        return (str(self.__value__)+name)
class Bar(Foo):
    a=0
x = Bar()
x.a=3
print(x.a)
```

:

- 1a
- 3
- 3a
- 0a
- 0

**181. Среди приведенных ниже фрагментов укажите вариант кода, при подстановке которого вместо знаков подчеркивания результатом выполнения скрипта станет 7**

```
def foo(f):
    class X:
        pass
    ___________
    return X
def method(lst):
    return len(lst)
Cs = foo(method)
print(Cs.method(range(7)))
```

- setattr(X, 'method', f)
- setattr(X, f)
- X.__dict__['method']=classmethod(f)
- X.method=classmethod(f)
- setattr(X, 'method', classmethod(lambda i: len(i)))

**182. Среди приведенных ниже фрагментов укажите вариант кода, при подстановке которого вместо знаков подчеркивания результатом выполнения скрипта станет 3**

```
def foo(f):
    class X:
        pass
    return X
def method(lst):
    return len(lst)
Cs = foo(method)

___________________
print(Cs.method(range(3)))
```

- setattr(Cs, 'method', classmethod(lambda i, x: method(x)))
- setattr(Cs, 'method', classmethod(method))
- Cs.__dict__['method']=lambda i, x: method(x)
- Cs.__dict__['method']=classmethod(method)
- нет правильного ответа

**183. Среди приведенных ниже фрагментов укажите вариант кода, при подстановке которого вместо знаков подчеркивания результатом выполнения скрипта станет**

**hello**

```
def foo(f):
    class X:
        pass
    return X
def method():
    return 'hello'
Cs = foo(method)
o = Cs()

___________
print(o.method())
```

:

- setattr(Cs, 'method', method)
- setattr(o, 'method', method)
- setattr(o, 'method', lambda i: 'hello')
- setattr(Cs, method.__name__, method)
- нет правильного ответа

**184. Среди приведенных ниже фрагментов укажите вариант кода, при подстановке которого вместо знаков подчеркивания результатом выполнения скрипта станет 3**

```
def foo(f):
    class X:
        pass

    ___________
    return X
def method(x, y):
    return x+y
Cs = foo(method)
o = Cs()
print(o.method(1, 2))
```

:

- setattr(X, method.__name__, f)
- setattr(X, f.__name__, lambda i, x, y : f(x,y))
- X.__dict__["method"] = f
- X.__dict__[f.__name__] = lambda i, x, y : f(x,y)
- X.method = f

**185. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __getattr__(self, name):
        return 0
print(Foo().x)
```

:

- x
- 0
- name
- None
- Error

**186. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self):
        self.x = 1
    def __getattribute__(self, item):
        return 2
    def __getattr__(self, item):
        return 3

print(Foo().x)
```

:

- 1
- 2
- 3
- None
- Error

## Исключения

**187. Какое зарезервированное слово не используется при построении полной формы исключения:**

- используются все перечисленные
- try
- else
- finally
- except

**188. Результат выполнения приведенного ниже кода**

```
class Foo:
    val = 0
    def fun(self):
        try:
            self.val[0]=0
            return self.val
        except TypeError:
            return self.val
x=Foo()
x.val=(1,2,3)
print(x.fun(), end=' ')
x.val='123'
print(x.fun(), end=' ')
```

:

- (0,2,3) 023
- (1,2,3) 123
- (0,2,3) 123
- (1,2,3) 023

- Error

**189. Результат выполнения приведенного ниже кода**

```
class Foo:
    val = 0
    def fun(self):
        try:
            self.val[0]+=self.val[0]
            return self.val
        except TypeError:
            return self.val
x=Foo()
x.val=12
print(x.fun(), end=' ')
x.val='12'
print(x.fun(), end=' ')
```

:

- 12 112
- 12 12
- 112 112
- 112 12
- Error

**190. Результат выполнения приведенного ниже кода**

```
class Foo:
    val = 0
    def fun(self):
        try:
            return self.val[-1]
        except TypeError:
            return self.val
x=Foo()
x.val='Hello'
print(x.fun(), end=' ')
x.val=15
print(x.fun(), end=' ')
```

:

- o 5
- H 15
- o 15
- o 1
- Error

**191. Результат выполнения приведенного ниже кода**

```
class Foo:
    val = 0
    def fun(self):
        try:
            return self.val[-1]
        except TypeError:
            return self.val
x=Foo()
x.val=(1,2,3,4)
print(x.fun(), end=' ')
x.val={0:1, 1:2, 2:3, 3:-1}
print(x.fun(), end=' ')
```

:

- 4 4
- 4 {0:1, 1:2, 2:3, 3:-1}
- 4 и KeyError
- (1,2,3,4) {0:1, 1:2, 2:3, 3:-1}
- Error

**192. Результат выполнения приведенного ниже кода**

```
class Foo:
    val = 0
    def fun(self):
        try:
            return self.val[-1]
        except TypeError:
            return self.val
x=Foo()
x.val=[1,2,3,4]
print(x.fun(), end=' ')
x.val='1234'
print(x.fun(), end=' ')
```

:

- 4 4
- 1 1
- 4 1234
- [1,2,3,4] 4
- Error

**193. Результат выполнения приведенного ниже кода**

```
try:
    print(36**0.5, end='')
except:
    print(0, end='')
finally:
    print(1, end='')
```

:

- 6.01

- 6.0
- 601
- 61
- Error

**194. Результат выполнения приведенного ниже кода**

```
import sys

try:
    sys.exit()
    print("1", end=' ')
finally:
    print("2", end=' ');
```

- 2
- 1
- 1 2
- 2 1
- Error

## Декораторы

**195. Результат выполнения приведенного ниже кода**

```
def fun1(f):
    print(f(1), end=' ')
@fun1
@fun1
def m(x):
    return x+2
```

- 1 3
- None
- 3 и сообщение об ошибке
- 3 5
- 5 и сообщение об ошибке

**196. Какой из типов методов не может иметь декоратор ?**

- метод класса
- декоратор может быть у всех перечисленных типов методов
- метод объекта
- декоратора нет у всех перечисленных типов методов
- статический метод

**197. Какой декоратор необходимо использовать, когда нужно вернуть объект, независимо от вызываемого дочернего класса ?**

- @property

- @classmethod
- @staticmethod
- @parentmethod
- @dynamicmethod

**198. Результат выполнения приведенного ниже кода**

```
class Alpha:

    @staticmethod
    def count(self, num_1, num_2):
        if num_1 > num_2:
            return num_2 * num_1
        else:
            return num_1 + num_2

a = Alpha()
print(a.count(-3, -5))
```

:

- Error
- 15
- -8
- -15
- None

**199. Результат выполнения приведенного ниже кода**

```
def fun1(f):
    return lambda i: f(i)+2
def fun2(f):
    print(f(3), end=' ')
@fun1
@fun2
def m(x):
    return x+2
```

:

- 5
- 7
- 9
- Error
- None

**200. Результат выполнения приведенного ниже кода**

```
def fun1(f):
    print(f(3), end=' ')
def fun2(f):
    return lambda i: f(i)+2
@fun1
@fun2
def m(x):
    return x+2
```

:

- 5
- 7
- 9
- Error
- None

**201. Результат выполнения приведенного ниже кода**

```
def fun1(f):
    print(f(1), end=' ')
@fun1
def m(x):
    return x+1
@fun1
def m2(x):
    return x+2
```

:

- None
- 2 3
- 1 1
- 1 2
- 4

**202. Результат выполнения приведенного ниже кода**

```
def fun1(f):
    print(f(1), end=' ')
def fun2(f):
    print(f(2), end=' ')
@fun2
def m(x):
    return x+1
@fun1
def m(x):
    return x+2
```

:

- 4 2
- 3 3
- 2 4
- 1 2
- 2 1

**203. Результат выполнения приведенного ниже кода**

```
def fun1(f):
    print(f(1), end=' ')
@fun1
def m(x):
    return (x,)
@fun1
def m2(x):
    return [x]
```

:

- None
- (1,) [1]
- 1 1
- (1) [1]
- [1] (1,)

**204. Результат выполнения приведенного ниже кода**

```
def execute(class_):
    def wrapper():
        return class_

    return wrapper

@execute
class A:
    pass

print(type(A()))
```

:

- <class'__main__.A'>
- <class 'type'>
- <class'__main__.class_'>
- <class 'A'>
- Error

**205. Результат выполнения приведенного ниже кода**

```
def decorator(func):
    def wrapper(x):
        return func(x)*func(x)
    return wrapper

@decorator
def add_2(x):
    return x + 2

print(add_2.__name__)
```

:

- add_2

- wrapper
- decorator
- None
- Error

**206. Результат выполнения приведенного ниже кода**

```
def squared(func):
    return lambda x: func(x)*func(x)

@squared
def mult_2(a):
    return a * 2

print(mult_2(3))
```

:

- 18
- 36
- 6
- None
- Error

**207. Результат выполнения приведенного ниже кода**

```
def squared(func):
    return lambda x: func(x)*func(x)

@squared
def add_2(a, b):
    return a + b

print(add_2(1, 4))
```

:

- 5
- 25
- 4
- 16
- Error

## Итераторы

**208. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, lst):
        self.lst = lst
    def __iter__(self):
        return self
    def __next__(self):
        raise StopIteration
for i in Foo(range(4)):
     print (i+1, end=' ')
```

:

- пустая строка
- 1 2 3 4
- 0 1 2 3
- 4 3 2 1
- Error

**209. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, lst):
        self.lst = lst
    def __iter__(self):
        return self
    def __next__(self):
        if self.lst:
            return self.lst.pop()
        else:
            raise StopIteration
for i in Foo(list(range(4))):
        print (i+1, end=' ')
```

:

- пустая строка
- 1 2 3 4
- 4
- 4 3 2 1
- Error

**210. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, lst):
        self.lst = lst
    def __iter__(self):
        return self
    def __next__(self):
        if self.lst:
            return self.lst[-1]
        else:
            raise StopIteration
for i in Foo('Hello'):
    print (i, end=' ')
```

:

- None
- H e l l o
- o l l e H
- скрипт будет бесконечно долго
- Error

**211. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, lst):
        self.lst = lst
    def __iter__(self):
        return self
    def __next__(self):
        return self.lst.pop()
for i in Foo(list(range(4))):
    print(i+1, end=' ')
```

:

- пустая строка
- 1 2 3 4
- 4 3 2 1 и Error
- 1 2 3 4 и Error
- 4 3 2 1

**212. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, lst):
        self.lst = lst
    def __iter__(self):
        return self
    def __next__(self):
        if self.lst:
            return self.lst.pop()
        else:
            raise StopIteration
for i in Foo(list(range(4))):
    print(i, end=' ')
```

:

- 0 1 2 3
- None
- 3 2 1 0
- Error
- 4 3 2 1

**213. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self, lst):
        self.lst = lst
    def __iter__(self):
        return self
    def __next__(self):
        if self.lst:
            return self.lst.pop(1)
        else:
            raise StopIteration
for i in Foo(list(range(4))):
    print(i + 1, end=' ')
```

:

- 2 3 4 1
- 1 2 3 4
- 2 3 4 и Error
- Error
- 4 3 2 1

## Модули и пакеты

**214. Чем отличаются в языке Python обычные модули от модулей расширения?**

- тем, что модули расширения могут состоять из нескольких файлов
- языком программирования, на котором они написаны
- способом подключения к главному модулю
- ничем не отличаются

- в Python нет понятия модули расширения

**215. В языке Python при импортировании пакета, если файл __init__.py отсутствует, то:**

- будет выдана ошибка, пакет импортирован не будет
- будет выдано предупреждение и пакет будет импортирован
- ошибки выдано не будет и пакет импортирован не будет
- ошибки выдано не будет, пакет будет импортирован, но не будут импортированы его модули
- нет правильного ответа

**216. Чем отличаются файлы *.py от файлов *.pyc ?**

- *.pyc - это откомпилированные файлы *.py
- ничем
- языком программирования, на котором они написаны
- *.pyc не могут быть подключены в качестве модулей, в отличие от файлов *.py
- *.pyc не имеют отношения к языку Python

**217. В языке Python модулем называют:**

- файл, содержащий определения и другие инструкции на каком-либо языке программирования
- набор классов, связанных наследованием и агрегацией
- класс, определенный специальным способом
- 1 файл, содержащий определения и другие инструкции только на языке Python
- набор файлов, содержащих код на языке Python

**218. В языке Python пакетом называют:**

- набор модулей, структурированных определенным образом
- набор модулей
- класс, определенный специальным способом
- файл, содержащий определения и другие инструкции на каком-либо языке программирования
- набор классов, связанных наследованием и агрегацией

**219. В языке Python при импортировании модуля с помощью инструкции import M:**

- в текущее пространство имен добавляется только имя модуля
- в текущее пространство имен добавляется имена имя модуля и всех его переменных, не начинающихся с подчеркивания
- в текущее пространство имен добавляется имена всех его переменных, не начинающихся с подчеркивания. Имя модуля не добавляется
- в текущее пространство имен добавляется имена всех его переменных, но имя модуля не добавляется
- в текущее пространство имен добавляется имя модуля и всех его переменных

**220. При импортировании модуля если в пространстве имен уже существует имя импортированного модуля, то:**

- модуль импортируется и заменяет своим именем уже существующее
- модуль импортируется только в случае, если определенное имя не является именем модуля

- будет выдана ошибка
- ошибки выдано не будет, но модуль не импортируется
- нет правильного ответа

**221. Встроенный атрибут объекта модуля __dict__:**

- является атрибутом только для чтения
- позволяет менять свои элементы по отдельности, но присваивание всего словаря не допустимо
- поддерживает операцию присваивания, но не позволяет изменять свои элементы по отдельности
- поддерживает как присваивание всего списка, так и изменение элементов по отдельности
- поддерживает как удаления элементов списка, также можно удалить сам атрибут

**222. В языке Python при импортировании модуля с помощью инструкции from .. import *:**

- в текущее пространство имен добавляется имя модуля и всех его переменных
- в текущее пространство имен добавляется имена всех его переменных, не начинающихся с подчеркивания. Имя модуля не добавляется
- в текущее пространство имен добавляется имена всех его переменных, но имя модуля не добавляется
- в текущее пространство имен добавляется имена имя модуля и всех его переменных, не начинающихся с подчеркивания
- в текущее пространство имен добавляется только имя модуля

**223. В языке Python встроенная функция dir() без аргументов используется для:**

- получения списка всех имен, доступных в текущей области видимости
- получения всех имен, доступных в текущем модуле
- получения имен всех функций, доступных в текущей области видимости
- получения всех имен модулей, доступных к подключению на текущий момент
- нет правильного ответа

**224. Встроенный атрибут объекта модуля __dict__ содержит:**

- таблицу имен всех доступных словарей в модуле
- таблицу имен всех доступных классов и функций в модуле
- таблицу всех имен, определенных или переопределенных в модуле
- таблицу всех имен модуля, в том числе и встроенных
- нет правильного ответа

**225. Встроенный атрибут объекта модуля __name__:**

- содержит имя модуля и является атрибутом только для чтения
- содержит имя файла, в котором содержится модуль и является атрибутом только для чтения
- содержит имя файла, в котором содержится модуль и может быть изменен
- содержит имя модуля и может быть изменен
- нет правильного ответа

**226. Файл foo.py содержит следующий код:**

```
def f(x, y):
    return x+y
```

**Скрипт a.py содержит следующий код**

```
_1_
print(_2_)
```

**Какие фрагменты кода нужно подставить вместо _1_ и _2_ чтобы в результате выполнения скрипта на выходе получилось 5 ?**

- _1_: import foo _2_: f(2, 3)
- _1_ import foo as mod _2_: foo.f(1, 4)
- _1_: from foo import * _2_: foo.f(0, 5)
- _1_: from foo import * as fun _2_: fun(5, 0)
- _1_: from foo import f as fun _2_: fun(3, 2)

**227. Файл mod.py содержит следующий код:**

```
def fun():
    return (1, 2, 3)
```

**Скрипт a.py содержит следующий код**

```
_1_
print(_2_)
```

**Какие фрагменты кода нужно подставить вместо _1_ и _2_ чтобы в результате выполнения скрипта на выходе получилось (1, 2, 3) ?**

- _1_: import mod _2_: mod.fun()
- _1_: import mod as fun _2_: mod.fun()
- _1_: from mod import * as fun _2_: fun()
- _1_: from mod import * _2_: mod.fun()
- _1_: from mod import fun as fun _2_: mod.fun()

**228. Файл module.py содержит следующий код:**

```
def function(x, y):
    return x*y
```

**Скрипт a.py содержит следующий код**

```
_1_
print(_2_)
```

**Какие фрагменты кода нужно подставить вместо _1_ и _2_ чтобы в результате выполнения скрипта на выходе получилось 12 ?**

- _1_: import module as module _2_: module.function(3, 4)
- _1_: from module import * as fun _2_: fun(6, 2)
- _1_: from module import function as fun _2_: module.fun(4, 3)
- _1_: import module _2_: function(2, 6)
- нет правильного ответа

**229. Файл foo.py содержит следующий код:**

```
def f(x):
    return x**2
```

**Скрипт a.py содержит следующий код**

```
_1_
print(_2_)
```

**Какие фрагменты кода нужно подставить вместо _1_ и _2_ чтобы в результате выполнения скрипта на выходе получилось 9 ?**

- _1_: import foo _2_: f(3)
- _1_: import foo _2_: foo.f(x)
- _1_: from foo import * _2_: foo.f(3)
- _1_: from foo import * _2_: f(3)
- нет правильного ответа

**230. Файл mod.py содержит следующий код:**

```
def fun():
    return 'Hello'
```

**Скрипт a.py содержит следующий код**

```
_1_
print(_2_)
```

**Какие фрагменты кода нужно подставить вместо _1_ и _2_ чтобы в результате выполнения скрипта на выходе получилась строка ‘Hello’ ?**

- _1_: from mod import * _2_: fun()
- _1_: import mod _2_: fun()
- _1_: import mod.py _2_: fun()
- _1_: from mod.py import * _2_: fun()
- _1_: from mod.py import * as fun _2_: fun()

**231. Файл module.py содержит следующий код:**

```
def function(x):
    return [x]
```

**Скрипт a.py содержит следующий код**

```
_1_
print(_2_)
```

**Какие фрагменты кода нужно подставить вместо _1_ и _2_ чтобы в результате выполнения скрипта на выходе получилась строка [1] ?**

- _1_: import module _2_: module.function(1)
- _1_: import module.py _2_: module.function(1)
- _1_: from module import * _2_: module.function(1)
- _1_: from module.py import * _2_: module.function(1)
- _1_: from module import function _2_: module.function(1)

**232. Результат выполнения приведенного ниже кода**

```
# foo.py
print(__name__)

# main.py
import foo
```

:

- __name__
- foo

- __main__
- Error
- None

## Статические методы и методы класса

**233. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    @classmethod
    def method(cls, val=3):
        if cls is Foo:
            return val+2
        else:
            return val+3
    @staticmethod
    def method1(cls, val=3):
        if cls is Foo:
            return val+2
        else:
            return val+3
print (Foo.method(1), Foo.method1(1));
```

- 4 4
- 3 4
- 3 6
- 4 3
- 6 3

**234. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    @classmethod
    def method(cls, val=3):
        if isinstance (cls, Foo):
            return val+2
        else:
            return val+3
    @staticmethod
    def method1(cls, val=3):
        if isinstance (cls, Foo):
            return val+2
        else:
            return val+3
f=Foo()
print (f.method(1), f.method1(1)) ;
```

- 6 4
- 3 6

- 4 6
- 3 3
- 6 3

**235. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self):
        self.__val__=2
    def method(cls, val):
        if isinstance(cls, Foo):
            return val+cls.__val__
        else:
            return val+1
    method = classmethod(method)
f=Foo()
print (Foo.method(1), f.method(1))
```

:

- 1 1
- 1 2
- 2 1
- 2 2
- Error

**236. Результат выполнения приведенного ниже кода**

```
class Foo:
    def __init__(self):
        self.__val__=2
    def method(cls, val):
        if isinstance(cls, Foo):
            return val+cls.__val__
        else:
            return val+1
    method = classmethod(method)
f=Foo()
print (Foo.method(1), f.method(1))
```

:

- 1 1
- 1 2
- 2 1
- 2 2
- Error

**237. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    @staticmethod
    def method(cls, val=3):
        if cls == Foo:
            return val+2
        else:
            return val+1
f=Foo()
print(Foo.method(1), f.method(1))
```

:

- 2 3
- 3 3
- 3 2
- 4 3
- 4 4

**238. Результат выполнения приведенного ниже кода**

```
class Foo(int):
    @staticmethod
    def method(cls, val=3):
        if cls == Foo:
            return val+2
        else:
            return val+1
    method = classmethod(method)
f=Foo()
print(Foo.method(1), f.method(1))
```

:

- 2 3
- 3 4
- 3 2
- 4 3
- Error

**239. Результат выполнения приведенного ниже кода**

```
class Foo:
    @classmethod
    def create(cls):
        print(cls.__name__)

class Bar(Foo):
    pass

Bar.create()
```

:

- Bar.__name__
- Error
- Foo.__name__

- Foo
- Bar

**240. Результат выполнения приведенного ниже кода**

```
class Bar:
    @staticmethod
    def quiz(*args):
        return sum(args)

print(Bar.quiz(1, 2), Bar().quiz(1,2));
```

- None 3
- 3 Error
- 1 2
- 3 None
- 3 3

## Основы фреймворка Django

**241. На каком языке программирования написан Django ?**

- Python
- C++
- Java
- JS
- C#

**242. Какой тип архитектуры реализует Django ?**

- Model-View-Template
- Model-View-Controller
- Url-View-Model
- Template-Model-Controller
- Template-Model-View

**243. На сколько базовых блоков разбито любое приложение на Django ?**

- 4
- 3
- 5
- 6
- 2

**244. Какой из приведенных ниже блоков Django отвечает за структуру данных приложения и предоставляет механизмы для управления данными ?**

- Модели
- Формы
- Шаблоны
- Диспетчер адресов

- Представление

**245. Какой из блоков определяет, с каким ресурсом нужно сопоставить запрос, и затем передает его выбранному ресурсу ?**

- Модели
- Формы
- Шаблоны
- Диспетчер адресов
- Представление

**246. Какая команда из перечисленных ниже позволит создать свой проект на Django через командную строку ?**

- django runproject название_проекта
- django startproject название_проекта
- django-admin runproject название_проектааблоны
- python startproject название_проекта
- django-admin startproject название_проекта

**247. Какой из обязательных файлов проекта на Django отвечает за указание и регистрацию пакетов ?**

- manage.py
- __init__.py
- settings.py
- urls.py
- asgi.py

**248. Какая база данных используется в Django по умолчанию ?**

- SQLite
- PostgreSQL
- Oracle
- MariaDB
- MySQL

**249. Какой командой запускается локальный сервер ?**

- manage.py runserver
- django manage.py startserver
- django manage.py runserver
- python manage.py startserver
- python manage.py runserver

**250. В каком файле из перечисленных по умолчанию указывается используемая в проекте база данных ?**

- settings.py
- manage.py
- views.py
- models.py

- __init__.py

**251. Как называется встроенный класс Django предназначенный для генерирования результата обработки запроса ?**

- HttpResponse
- render
- HttpAnswer
- HttpRedirect
- request

**252. Какая функция из перечисленных задает возможность сопоставлять запросы пользователя (определенные маршруты) с функцией их обработки ?**

- path
- http_path
- href_path
- url_path
- source_path

**253. Выберите метод, невозвращающий объект типа QuerySet ?**

- все перечисленные возвращают объект типа QuerySet
- all()
- filter()
- exclude()
- in_bulk()

**254. Какой символ или несколько символов перед строкой соответствует регулярному выражению ?**

- r
- reg
- rex
- f
- regular

**255. Каким символом отделяются друг от друга параметры в строке запроса ?**

- &
- @
- $
- ?
- =

**256. В каком из стандартных файлов проекта Django находится переменная TEMPLATES, отвечающая за настройку шаблонов ?**

- urls.py
- settings.py
- views.py
- models.py

- forms.py

**257. Какая функция из модуля django.shortcuts в представлении принимает объект запроса пользователя и файл шаблона в качестве параметров для последующего отображения ?**

- request
- render
- template
- httpresponse
- нет правильного ответа

**258. Как называется именованный параметр функции, позволяющий передать данные в шаблон ?**

- data
- context
- source
- text
- url

**259. В блоке h1 {font-family: Arial; color: grey;} что собой представляет h1 ?**

- селектор
- описание
- свойство
- объявление
- нет правильного ответа

**260. Выберите несуществующий структурный элемент веб-страницы ?**

- floor
- header
- footer
- sidebar
- все существуют

**261. При помощи каких символов определяются блоки шаблонов ?**

- {% %}
- {{ }}
- <> </>
- {& &}
- {? ?}

**262. Как называется группа из одного или нескольких полей (виджетов) на веб-странице, которая используется для получения информации от пользователей для последующей отправки на сервер ?**

- форма
- шаблон
- представление

- регулярное выражение
- нет правильного ответа

**263. Каждая форма определяется в виде отдельного класса. От какого встроенного класса Django эти классы наследуется ?**

- forms.Form
- templates.Template
- forms.Template
- templates.Form
- templates.View

**264. При помощи какого параметра можно задать метку для поля ?**

- label
- mark
- tag
- marker
- point

**265. Что такое токен ?**

- цифровой сертификат безопасности
- хэш-функция
- тип виджета
- стандартное поле веб-приложения
- тип формы

**266. Выберите несуществующий в Django тип поля:**

- BooleanField
- RegexField
- UUIDField
- FilePathField
- CatalogField

**267. Какой дополнительный аргумент классов типа Field указывает на необходимость обязательного заполнения поля (по умолчанию True) ?**

- widget
- disabled
- validators
- initial
- required

**268. Какой дополнительный аргумент классов типа Field позволяет указать максимально допустимое количество знаков после запятой ?**

- max_whole_digits
- limit_values
- max_digits
- min_digits

- decimal _places

**269. Какой дополнительный аргумент классов типа Field позволяет задать значение поля по умолчанию ?**

- coerce
- empty_value
- initial
- context
- help_text

**270. С помощью какого метода осуществляется проверка корректности ввода данных на стороне сервера ?**

- check()
- validate()
- is_valid()
- is_correct()
- нет правильного ответа

**271. С системой управления базами данных (СУБД) приложение общается каким-нибудь универсальным способом - например, посредством языка SQL. Однако, программисту чаще всего хочется иметь некую абстракцию, позволяющую большую часть времени работать с привычными сущностями языка программирования. Как называется такая абстракция в Django ?**

- LRD
- Postgre
- ORM
- MVC
- CSRF

**272. Как называется класс от которого наследуют все модели в Django ?**

- models.Model
- forms.Form
- templates.Model
- forms.Model
- models.View

**273. Какой командой создается файл с параметрами миграции ?**

- python manage.py makemigrations
- python manage.py migrate
- python models.py migrate
- python models.py makemigrations
- python urls.py makemigrations

**274. Какая аббревиатура обозначает четыре основные операции, которые используются при работе с базами данных ?**

- JIRA
- CRUD

- ACID
- DRYS
- ORMD

**275. Какой оператор языка структурированных запросов соответствует чтению записей ?**

- Insert
- Select
- Read
- Update
- Drop