Вопросы Junior

Общая теория

Что такое DRY?

В программировании термин "DRY" относится к принципу проектирования и разработки программного кода, который означает "Don't Repeat Yourself" (не повторяйся). Этот принцип подразумевает, что каждая часть системы или функциональность должна быть реализована и определена в одном месте, чтобы избежать дублирования кода.

Идея "dry" состоит в том, что когда код дублируется в нескольких местах, это может привести к следующим проблемам:

- Усложнение поддержки: Изменения, вносимые в одном месте, могут быть забыты или неправильно применены в других местах, что приводит к ошибкам и багам.
- Увеличение объема кода: Дублирование кода приводит к ненужному расходованию места и усложнению структуры программы.
- Ухудшение читаемости: Когда логика повторяется в разных частях кода, это затрудняет его чтение и понимание.

При следовании принципу "DRY", программисты стараются выносить общую функциональность или логику в отдельные функции, классы, или модули, которые затем можно использовать многократно в разных частях программы. Это упрощает поддержку кода, уменьшает его размер и улучшает его читаемость.

Принцип "DRY" тесно связан с другими принципами проектирования, такими как "Single Responsibility Principle" (принцип единственной ответственности) и "Separation of Concerns" (разделение забот). Все эти принципы стремятся к созданию более чистого, модульного и легко поддерживаемого программного кода.

Что такое KISS?

"KISS" - это аббревиатура, которая в контексте программирования и проектирования обозначает принцип "Keep It Simple, Stupid" (Делай просто, глупец). Это один из ключевых принципов разработки программного обеспечения и означает, что решения должны быть максимально простыми, понятными и минималистичными.

Главная идея принципа KISS заключается в том, что сложные системы имеют больше шансов на ошибки, сложнее поддерживать и труднее понимать. Простота кода и архитектуры упрощает разработку, отладку и сопровождение программы.

Важные аспекты принципа KISS:

- Простота: Старайтесь использовать наиболее простые и прямолинейные решения для решения задачи. Избегайте излишней сложности и избыточных деталей.
- Читаемость: Код должен быть легко читаемым и понятным для других разработчиков. Используйте понятные и описательные имена переменных, функций и классов.
- Минимализм: Избегайте добавления избыточных функций или компонентов, если они не требуются. Сосредотачивайтесь только на необходимой функциональности.
- Понимание: Старайтесь, чтобы ваш код был понятен не только вам, но и другим разработчикам, которые будут работать с этим кодом в будущем.

Что такое YAGNI?

"YAGNI" - это аббревиатура, которая относится к принципу программирования и проектирования и означает "You Aren't Gonna Need It" (Тебе это не понадобится). Этот принцип предупреждает разработчиков от добавления функциональности, которая в данный момент не требуется, но может быть полезна в будущем.

Основные идеи принципа YAGNI:

- Простота: По аналогии с принципом KISS, стоит избегать добавления сложных и избыточных функций, если они не являются неотъемлемой частью текущей задачи.
- Сосредоточьтесь на текущей задаче: При разработке новой функциональности или модуля следует сосредотачиваться на решении текущих задач и требований, а не предполагать возможные потребности в будущем.
- Избегайте переусложнения: Добавление функциональности, которая может быть полезной в далеком будущем, может привести к переусложнению кода и усложнить его поддержку.
- Решайте проблемы, когда они возникают: Если какая-то функциональность потребуется в будущем, лучше добавить ее в код, когда это действительно станет необходимым, а не заблаговременно.

Принцип YAGNI помогает создавать более простое и эффективное программное обеспечение. Избегая лишних функций и компонентов, разработчики сокращают объем работы, снижают возможные ошибки и облегчают поддержку кода. Однако важно найти баланс, чтобы не отказываться от функциональности, которая действительно может потребоваться в скором времени или предусмотреть планы на будущее при разработке архитектуры.

Что такое GRASP?

"GRASP" - это аббревиатура, стоящая за "General Responsibility Assignment Software Patterns" (Паттерны общего назначения для назначения ответственности в программном обеспечении). GRASP - это набор принципов и шаблонов проектирования, разработанных для помощи разработчикам в определении структуры объектно-ориентированных систем.

Цель GRASP состоит в том, чтобы помочь в архитектуре и проектировании объектно-ориентированных систем, чтобы получить хорошо структурированный и гибкий код с четко определенными отношениями между объектами.

- Expert (Эксперт): Принцип, по которому ответственность должна быть назначена классу, который имеет наибольшие знания и информацию для выполнения определенной операции.
- Creator (Создатель): Принцип, определяющий, что класс должен создавать экземпляры другого класса, если он содержит или агрегирует этот класс.
- Low Coupling (Низкая связность): Принцип, ставящий целью минимизацию зависимостей между классами, чтобы изменения в одном классе не приводили к изменениям во многих других классах.
- High Cohesion (Высокая связность): Принцип, при котором класс должен быть спроектирован так, чтобы его методы тесно связаны с его состоянием и выполняли одну логическую задачу.
- Controller (Контроллер): Принцип, определяющий, что класс, ответственный за обработку пользовательского ввода и координацию работы системы, должен быть отделен от других классов.
- Polymorphism (Полиморфизм): Принцип, по которому общее поведение может быть представлено в виде общего интерфейса или абстрактного класса, позволяя объектам различных классов иметь разное поведение при реализации этого интерфейса.
- Indirection (Индиректность): Принцип, который говорит о том, что промежуточные объекты или контроллеры могут использоваться для уменьшения прямых связей между объектами.

Что такое SOLID?

"SOLID" - это аббревиатура, представляющая собой первые буквы пяти основных принципов объектно-ориентированного программирования и проектирования. Эти принципы помогают разработчикам создавать гибкие, поддерживаемые и масштабируемые программные системы. SOLID был представлен Робертом Мартином и Майклом Фезерсом как мнемоническое устройство для запоминания этих принципов.

- Принцип единственной ответственности (Single Responsibility Principle - SRP): Каждый класс должен иметь только одну причину для изменения. Это означает, что класс должен быть ответственен только за одну четко определенную функциональность или задачу.
- Принцип открытости/закрытости (Open/Closed Principle - OCP): Сущности программы должны быть открыты для расширения, но закрыты для изменения. Это можно достичь путем использования абстракций, интерфейсов и наследования, чтобы изменять поведение программы без изменения исходного кода.
- Принцип подстановки Барбары Лисков (Liskov Substitution Principle - LSP): Объекты базовых классов должны быть заменяемыми своими подклассами без нарушения корректности программы. То есть, код, использующий базовый класс, должен корректно работать с любым его подклассом, не зная об этом.
- Принцип разделения интерфейса (Interface Segregation Principle - ISP): Много специализированных интерфейсов лучше, чем один универсальный. Клиенты не должны зависеть от интерфейсов, которые они не используют, и классы должны предоставлять только те методы, которые требуются клиентам.
- Принцип инверсии зависимостей (Dependency Inversion Principle - DIP): Зависимости должны строиться на абстракциях, а не на конкретных реализациях. Высокоуровневые модули не должны зависеть от низкоуровневых модулей, а оба типа модулей должны зависеть от абстракций.

Что такое REST?

REST (Representational State Transfer) - это архитектурный стиль для проектирования сетевых приложений, который используется для создания веб-сервисов. REST основывается на простоте, масштабируемости, легкости использования и независимости от состояния.

Основные принципы REST:

- Ресурсы (Resources): В REST все данные представлены в виде ресурсов, которые могут быть доступны через уникальные идентификаторы (URI). Ресурсы могут быть объектами, коллекциями, услугами и т.д.
- Методы HTTP (HTTP Methods): REST использует основные методы HTTP для работы с ресурсами. Наиболее распространенные методы это GET (получение данных), POST (создание нового ресурса), PUT (обновление ресурса), DELETE (удаление ресурса).
- Представление ресурсов: Ресурсы могут иметь различные представления, такие как XML, JSON, HTML и т.д. Клиент может выбрать формат, который лучше всего подходит для его нужд.
- Без состояния (Stateless): Каждый запрос к серверу должен содержать всю необходимую информацию для обработки этого запроса, и сервер не должен хранить информацию о предыдущих запросах клиента. Это делает систему более масштабируемой и уменьшает нагрузку на сервер.
- Унифицированный интерфейс (Uniform Interface): REST использует унифицированный интерфейс для взаимодействия с ресурсами, что упрощает и облегчает использование API.

RESTful API - это веб-сервис, который следует архитектурному стилю REST. RESTful API обеспечивает доступ к ресурсам через стандартные HTTP-методы и обычно использует форматы данных, такие как JSON или XML, для обмена информацией с клиентами.

REST стал очень популярным подходом для создания веб-сервисов, так как он обладает простотой, надежностью и хорошей масштабируемостью. Множество современных веб-приложений и мобильных приложений используют RESTful API для взаимодействия с сервером и получения данных.

Что такое CORS?

CORS (Cross-Origin Resource Sharing) - это механизм безопасности, используемый в веб-разработке для контроля доступа к ресурсам на веб-странице, размещенным на другом домене (origin). "Origin" - это комбинация протокола (например, http или https), домена и порта веб-страницы.

Браузеры применяют политику CORS для защиты веб-приложений от несанкционированных запросов к ресурсам на других доменах, что называется "Cross-Origin Request" (CORS запрос). По умолчанию, из-за политики безопасности браузера, JavaScript-код, выполняемый на веб-странице, имеет доступ только к ресурсам, находящимся на том же домене, с которого была загружена страница.

Однако с помощью заголовков CORS на сервере, можно разрешить браузеру выполнять CORS запросы и получать доступ к ресурсам на других доменах. Это делает возможным безопасное взаимодействие между веб-приложениями, размещенными на разных доменах.

Когда браузер делает запрос на другой домен, он сначала отправляет предварительный запрос (preflight request) методом OPTIONS, чтобы проверить, разрешен ли запрос. Сервер должен ответить с заголовками CORS, указывающими, разрешено ли выполнять запросы с данного домена.

CORS является важным аспектом безопасности веб-приложений и позволяет предотвратить различные типы атак, таких как CSRF (Cross-Site Request Forgery) и XSS (Cross-Site Scripting).

Что такое HTTP?

HTTP (HyperText Transfer Protocol) - это протокол передачи данных в сети, который определяет правила и формат взаимодействия между клиентом (например, веб-браузером) и сервером (например, веб-сайтом) во время обмена информацией.

Основное назначение HTTP - обеспечить передачу данных и запросов между клиентом и сервером в виде гипертекста, который представляет собой текст с гиперссылками (URL) и другими элементами для создания веб-страниц и ресурсов. HTTP обеспечивает простой и расширяемый способ обмена данными в сети, что делает возможным функционирование всемирной паутины (World Wide Web).

Основные характеристики HTTP:

- Stateless (без сохранения состояния): Каждый запрос от клиента к серверу считается отдельным запросом, не зависящим от предыдущих запросов. Сервер не хранит информацию о состоянии клиента между запросами.
- Client-Server (клиент-серверная архитектура): Клиент и сервер взаимодействуют друг с другом, обмениваясь запросами и ответами. Клиент и сервер могут быть на различных физических машинах.
- Протокол запрос-ответ: Клиент отправляет HTTP-запрос на сервер, а сервер отвечает на него с помощью HTTP-ответа, содержащего статус операции и необходимые данные.
- Без сохранения состояния (stateless): Каждый запрос от клиента к серверу рассматривается в изоляции и не зависит от предыдущих запросов. Сервер не хранит информацию о состоянии клиента между запросами.
- Сессии и куки: Для поддержания состояния между запросами, HTTP использует механизмы сессий и куки, которые позволяют сохранять данные на стороне клиента и передавать их между запросами.

Какие методы HTTP вам известны?

- GET: Используется для получения данных с сервера. Когда клиент отправляет GET-запрос, сервер возвращает запрошенные данные в теле ответа. Этот метод не должен иметь побочных эффектов и не изменяет данные на сервере.
- POST: Используется для отправки данных на сервер для обработки. POST-запрос может содержать данные в теле запроса, которые могут быть сохранены или обработаны на сервере. Часто используется для создания новых ресурсов на сервере.
- PUT: Используется для обновления данных на сервере. PUT-запрос содержит данные, которые заменяют существующий ресурс на сервере или создают новый, если такой ресурс отсутствует.
- DELETE: Используется для удаления данных на сервере. DELETE-запрос указывает на ресурс, который должен быть удален.
- PATCH: Используется для частичного обновления данных на сервере. Похож на PUT, но обновляет только определенные поля ресурса, не затрагивая остальные.
- HEAD: Аналогичен GET, но возвращает только заголовки ответа без тела ответа. Используется, когда клиенту нужно получить метаданные ресурса без его фактического содержимого.
- OPTIONS: Используется для определения возможностей сервера и параметров соединения для указанного ресурса.
- CONNECT: Используется для установки сетевого соединения с ресурсом через прокси.
- TRACE: Возвращает диагностические данные, которые позволяют клиенту узнать, как прокси или сервер обрабатывают запрос.

Идемпотентные и не идемпотентные методы HTTP?

Методы запроса могут быть разделены на идемпотентные и не идемпотентные в зависимости от того, как повторные запросы с одинаковыми параметрами и данными влияют на состояние сервера. Идемпотентные методы не должны изменять состояние сервера при многократном повторении одного и того же запроса, в то время как не идемпотентные методы могут менять состояние сервера при каждом их повторении.

Идемпотентные методы HTTP:

- GET: Получение информации или ресурсов с сервера. Повторные запросы на чтение данных не должны менять состояние сервера.
- HEAD: То же самое, что и GET, но без тела ответа. Также является идемпотентным.
- PUT: Загрузка данных на сервер по определенному URL. Повторные PUT-запросы с теми же данными должны заменять или обновлять ресурс, но не создавать новый.
- DELETE: Удаление ресурса с сервера. Повторные запросы на удаление уже удаленного ресурса не должны вызывать ошибку или изменять состояние сервера.

Не идемпотентные методы HTTP:

- POST: Создание нового ресурса на сервере. Повторные POST-запросы с теми же данными могут привести к созданию дубликатов или изменению состояния сервера каждый раз.
- PATCH: Частичное обновление ресурса. Повторные PATCH-запросы могут менять состояние ресурса каждый раз.
- PUT: По соглашению PUT может быть рассмотрен как не идемпотентный, если сервер выполняет какую-либо дополнительную обработку при каждом PUT-запросе.

Что такое MVC?

MVC - это аббревиатура, которая означает "Model-View-Controller" (Модель-Представление-Контроллер). Это шаблон проектирования (паттерн), используемый в разработке программного обеспечения, включая веб-приложения.

В архитектуре MVC компоненты разделены на три основных уровня:

- Модель (Model): Отвечает за управление данными и бизнес-логикой приложения. Модель представляет собой структуру данных, обрабатывает операции с этими данными, а также обеспечивает доступ к хранилищу данных, такому как база данных.
- Представление (View): Представляет пользовательский интерфейс. Он отвечает за отображение данных из Модели пользователю и обработку взаимодействия пользователя с интерфейсом. Представление получает данные из Модели и отображает их пользователю в удобном виде.
- Контроллер (Controller): Действует в качестве посредника между Моделью и Представлением. Контроллер обрабатывает пользовательский ввод и взаимодействие с интерфейсом, затем обновляет Модель в соответствии с этим вводом и передает обновленные данные Представлению для отображения.

Расскажите про принципы ООП?

ООП означает "Объектно-ориентированное программирование". Это парадигма программирования, которая ставит в центр разработки программ объекты и их взаимодействие. В ООП, программа рассматривается как набор взаимодействующих между собой объектов, каждый из которых представляет реальный объект или абстракцию.

Основные принципы ООП включают:

- Инкапсуляция: Объекты могут скрывать свою внутреннюю реализацию и предоставлять интерфейс для взаимодействия с внешним миром. Это позволяет изменять внутреннюю реализацию объекта без влияния на внешний код.
- Наследование: Один класс может наследовать свойства и методы от другого класса. Наследование позволяет создавать иерархию классов, где дочерние классы получают функциональность родительских классов.
- Полиморфизм: Это позволяет объектам различных классов иметь одинаковый интерфейс, но различную реализацию. Полиморфизм позволяет использовать объекты разных типов с одним и тем же интерфейсом.
- Абстракция: Это позволяет создавать абстрактные классы и интерфейсы, которые определяют общий набор функциональности, но не содержат конкретной реализации. Абстракция помогает создавать модели и обобщенные концепции, упрощая сложные системы.

Что такое композиция?

Композиция - это концепция в объектно-ориентированном программировании, которая позволяет создавать более сложные объекты путем объединения или составления объектов более простых классов. В композиции один класс содержит объекты других классов в качестве своих частей и использует их для реализации своей функциональности.

Ключевые особенности композиции:

- Отношение часть-целое: Композиция устанавливает отношение "часть-целое" между классами. Один класс (целое) содержит экземпляры других классов (части).
- Повторное использование кода: Композиция позволяет повторно использовать функциональность уже существующих классов, просто комбинируя их в новом классе.
- Легкая модификация: Композиция обеспечивает гибкость при изменении функциональности объекта. Вы можете изменить или заменить компоненты, не затрагивая другие части кода.

Что такое кэширование?

Кэширование - это техника, используемая в программировании и информационных технологиях, которая заключается в сохранении временных копий данных, результатов вычислений или ресурсов в более быстром доступе, чтобы улучшить производительность и сократить нагрузку на систему.

Когда данные запрашиваются или вычисляются, результат сохраняется в кэше. При последующих запросах к тому же ресурсу, система сначала проверяет наличие данных в кэше. Если данные найдены в кэше, они возвращаются без необходимости повторного вычисления или обращения к источнику данных. Это позволяет значительно ускорить доступ к данным и уменьшить нагрузку на серверы или другие системы.

Основные преимущества кэширования:

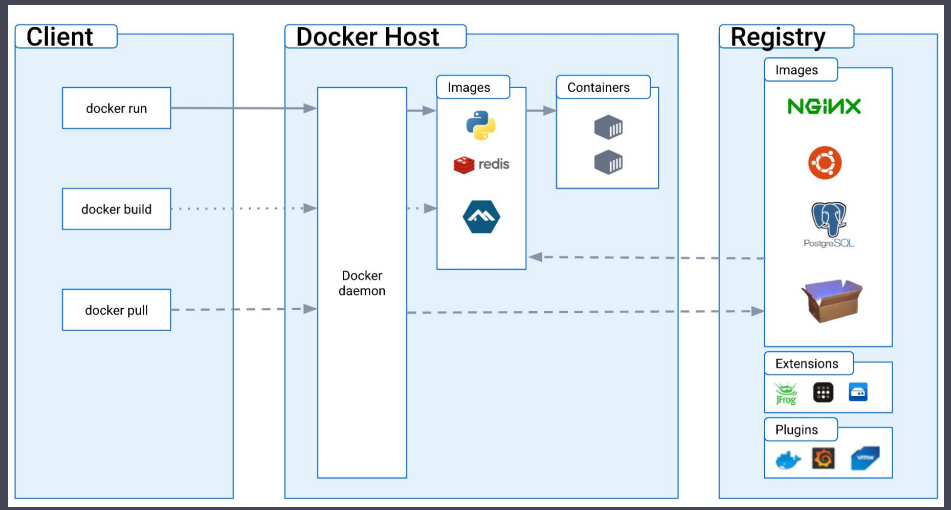
- Улучшение производительности: Кэширование позволяет быстрее получать данные, так как они уже хранятся в более быстром доступе, вместо того чтобы выполнять дорогостоящие операции загрузки или вычисления.
- Сокращение нагрузки на систему: Когда данные уже находятся в кэше, система может избежать дополнительных запросов к базе данных или другим источникам данных, что уменьшает нагрузку на серверы и сеть.
- Экономия времени и ресурсов: Кэширование позволяет избежать повторных вычислений и запросов, что экономит время и ресурсы компьютерной системы.

Однако кэширование также имеет свои ограничения и недостатки:

- Устаревание данных: Кэшированные данные могут устареть, если исходные данные были изменены, и это может привести к некорректным результатам.
- Занимаемое место: Кэш может занимать дополнительное место в памяти или на диске, что может быть проблемой, особенно при ограниченных ресурсах.
- Согласованность данных: Если данные обновляются в источнике, необходимо правильно обрабатывать синхронизацию и инвалидацию кэша, чтобы предотвратить предоставление устаревших данных.

Что такое Docker?

Docker - это платформа для контейнеризации приложений, которая позволяет упаковать приложение и все его зависимости в стандартизированный контейнер. Контейнер представляет собой самодостаточную среду, которая включает в себя код, исполняемые файлы, библиотеки, настройки и все необходимое для работы приложения. Это облегчает развертывание и запуск приложения в различных средах без необходимости устанавливать и настраивать все его зависимости отдельно.



Что такое Dockerfile?

Dockerfile - это текстовый файл, который используется для определения инструкций и конфигурации, необходимых для создания Docker образа. Dockerfile является основным компонентом процесса создания контейнеров в Docker.

Dockerfile содержит набор команд, которые описывают, как собрать образ контейнера. Когда вы создаете Docker образ, Dockerfile указывает, какие зависимости и настройки нужно включить в образ, чтобы приложение успешно выполнялось в контейнере.

```
# Используем базовый образ Python
FROM python:3.9

# Устанавливаем рабочую директорию в контейнере
WORKDIR /app

# Копируем зависимости из текущего каталога в контейнер
COPY requirements.txt .

# Устанавливаем зависимости через pip
RUN pip install --no-cache-dir -r requirements.txt

# Копируем все файлы приложения из текущего каталога в контейнер
COPY . .

# Определяем команду, которая будет запущена при старте контейнера
CMD ["python", "app.py"]
```

Что такое Docker Compose?

Docker Compose - это инструмент для определения и управления многоконтейнерными приложениями в Docker. Он позволяет определить структуру и зависимости между контейнерами в едином файле конфигурации, называемом "docker-compose.yml".

С помощью Docker Compose можно определить несколько сервисов, каждый из которых представляет собой контейнер Docker, и указать их свойства, такие как используемый Docker образ, порты, переменные окружения, сетевые настройки и другие параметры. Docker Compose позволяет запускать и останавливать все сервисы одновременно с помощью простых команд.

```yaml
version: '3'

services:
  web:
    image: nginx:latest
    ports:
      - "80:80"

  db:
    image: mysql:latest
    environment:
      MYSQL_ROOT_PASSWORD: mysecretpassword
      MYSQL_DATABASE: mydatabase
```

Что такое Kubernetes?

Kubernetes (K8s) - это открытая система для автоматизации развертывания, масштабирования и управления контейнеризированными приложениями. Он был разработан компанией Google и предоставлен в 2014 году как проект с открытым исходным кодом, в настоящее время под управлением Cloud Native Computing Foundation (CNCF).

Основной целью Kubernetes является упрощение управления контейнеризированными приложениями в динамичных, распределенных средах. Kubernetes предоставляет средства для автоматизации развертывания, масштабирования и управления приложениями в контейнерах. Он позволяет разработчикам сосредотачиваться на логике приложения, не вникая в детали управления инфраструктурой.

Основные компоненты Kubernetes включают:

- Под (Pod): Наименьшая и базовая единица развертывания в Kubernetes. Под представляет собой один или несколько контейнеров, которые разделяют сеть и хранение.
- Репликация (Replication Controller/ReplicaSet): Отвечает за поддержание заданного количества экземпляров Подов. Если один или несколько Подов выходят из строя, Репликация запускает новые экземпляры, чтобы обеспечить желаемое количество.
- Сервис (Service): Объединяет группу Подов и предоставляет стабильную IP-адресацию и DNS-имя для доступа к этим Подам.
- Ресурс (Resource): Определяет ограничения и требования ресурсов для контейнеров, такие как использование CPU и памяти.
- Развертывание (Deployment): Предоставляет декларативный способ обновления Подов и Репликаций во времени выполнения.
- Конфигмап (ConfigMap) и Секрет (Secret): Позволяют внедрять конфигурацию и секретные данные в контейнеры без изменения контейнера.

Kubernetes предоставляет масштабируемый, отказоустойчивый и самоуправляемый механизм управления контейнеризированными приложениями, который позволяет разрабатывать и запускать приложения в различных облачных и локальных средах, таких как AWS, Google Cloud, Microsoft Azure и другие. Он стал широко используемой платформой для разработки современных, облачных и микросервисных приложений

В чем разница между образом Docker и контейнером Docker?

Образ Docker (Docker Image):
Образ Docker представляет собой статическую и неизменяемую сущность, которая содержит все необходимое для запуска приложения, включая код, зависимости, библиотеки, настройки и другие компоненты. Образ является шаблоном или "инструкцией" для создания контейнера.
Образы создаются из Dockerfile - текстового файла, который определяет инструкции для сборки образа. Когда Dockerfile обрабатывается командой `docker build`, Docker создает образ на основе указанных в нем инструкций.
Образы являются неизменяемыми, что означает, что после того, как образ создан, он не может быть изменен. Если нужно внести изменения, необходимо создать новый образ с обновленными инструкциями в Dockerfile.

Контейнер Docker (Docker Container):
Контейнер Docker - это экземпляр образа. Когда образ запускается с помощью команды `docker run`, Docker создает контейнер, который представляет собой работающий экземпляр образа.
Контейнер является "живой" сущностью, в которой приложение может выполняться. Контейнер запускается в изолированной среде с собственным пространством имен процессов, сетевыми интерфейсами и файловой системой.
Контейнеры могут быть запущены, остановлены, удалены и повторно запущены, и они предоставляют изолированный и повторяемый способ запуска приложений в контролируемой среде.

Для чего нужен GIT?

Git играет ключевую роль в управлении версиями программного обеспечения и обеспечивает совместную разработку проектов. Вот основные причины, почему Git является важным инструментом для разработчиков:

- Управление версиями: Git отслеживает все изменения в коде и файловой структуре проекта. Это позволяет разработчикам легко переключаться между различными версиями проекта, возвращаться к предыдущим состояниям кода, а также анализировать историю изменений.
- Работа в команде: Git обеспечивает возможность совместной разработки. Каждый разработчик может работать над своей версией кода в отдельной ветке (branch) и затем объединить свои изменения с основной веткой (master) через процесс слияния (merge).
- Ветвление и слияние: Ветвление позволяет создавать отдельные ветки для разработки новых функций или исправления ошибок, не затрагивая основную кодовую базу. После тестирования и проверки изменений можно объединить их с основным проектом.
- Резервное копирование и восстановление: Git обеспечивает надежное хранение истории изменений проекта. Это позволяет легко восстанавливать прошлые версии кода в случае возникновения проблем или ошибок.
- Устранение конфликтов: В многопользовательской среде может возникнуть ситуация, когда разные разработчики вносят изменения в одну и ту же часть кода. Git помогает управлять такими конфликтами и разрешать их в процессе слияния.
- Публикация и совместное использование: Git предоставляет возможность публиковать и совместно использовать код, делая его доступным для других разработчиков и участников проекта.
- Работа в автономном режиме: Git распределенная система, поэтому каждый разработчик имеет локальную копию всего репозитория. Это позволяет работать над проектом в автономном режиме, даже если нет подключения к сети или центральному серверу.

Git является одним из наиболее популярных и широко используемых инструментов в сфере разработки программного обеспечения, и его преимущества сделали его незаменимым инструментом для множества разработчиков и проектов.

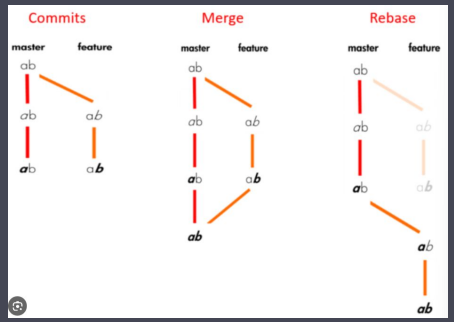
В чем разница между git merge и git rebase?

`git merge`:

- При использовании `git merge` создается новый коммит, который объединяет изменения из исходной и целевой ветки. Этот коммит имеет двух родительских коммитов, представляющих историю обеих веток.
- История коммитов остается прежней, со всеми коммитами исходной ветки, а также с новым коммитом слияния.
- Слияние легко понять и показывает явно, когда и где произошла интеграция.

`git rebase`:

- При использовании `git rebase` ветка перемещается на новую базовую точку. Изменения, сделанные в исходной ветке, воспроизводятся поверх целевой ветки, создавая новые коммиты.
- Это приводит к линейной истории коммитов, как если бы изменения были разработаны последовательно на целевой ветке без каких-либо коммитов слияния.
- `git rebase` помогает поддерживать чистую и линейную историю коммитов, которая может быть проще для чтения и навигации.



Что такое commit?

В Git, "commit" (коммит) - это основной строительный блок и фиксированный набор изменений в репозитории. Когда вы делаете коммит в Git, вы сохраняете текущее состояние файлов и их структуры в вашем проекте.

Каждый коммит содержит следующую информацию:

- Снимок файлов: Коммит включает набор изменений (diff) всех файлов, которые вы добавили, изменили или удалили с момента предыдущего коммита.
- Уникальный идентификатор: Каждый коммит имеет уникальный SHA-1 хеш (хеш-код), который служит идентификатором коммита и позволяет отличать один коммит от другого.
- Автор и коммиттер: Коммит содержит информацию об авторе, т.е. том, кто создал изменения, и коммиттере, т.е. том, кто сделал коммит в репозиторий.
- Дата и время: Зафиксированное время и дата создания коммита.
- Комментарий: Коммит сопровождается комментарием (commit message), где вы можете описать, что было сделано в данном коммите и зачем.

Коммиты играют важную роль в истории проекта, так как они сохраняют все изменения, которые сделаны во время разработки. Поэтому можно легко вернуться к предыдущим версиям кода, отслеживать изменения, решать проблемы и совместно работать над проектом в команде. Каждый коммит действует как моментальный снимок проекта в определенный момент времени и представляет собой важный элемент системы управления версиями, которая делает Git таким мощным инструментом для разработчиков.

Что такое Git Flow?

Git Flow - это набор стратегий и методологий работы с Git, которые предоставляют структурированный подход к организации рабочего процесса и управлению версиями программного обеспечения. Эта модель разработки была предложена Винсентом Дриессеном и стала популярной в сообществе разработчиков благодаря своей логичности и простоте.

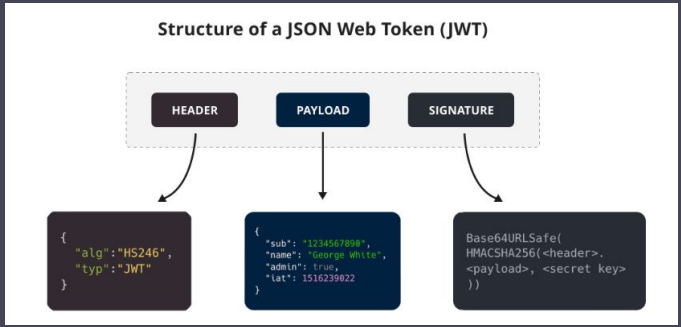
- `master`: В этой ветке находятся стабильные и готовые к выпуску версии программного обеспечения. Код в `master` всегда должен быть рабочим и проходить все необходимые проверки.
- `develop`: Это ветка разработки, где объединяются все изменения от различных фич-веток. Код в `develop` может быть временно нестабильным, но должен быть работоспособным.
- `feature`: Это ветки, которые создаются для разработки новых функций. Они отходят от `develop` и объединяются обратно в `develop`, когда функциональность готова.
- `release`: Ветки `release` создаются для подготовки новой версии программного обеспечения. В них выполняется тестирование, исправление ошибок и подготовка к выпуску. Затем они объединяются как в `master`, так и в `develop`.
- `hotfix`: Если в производственной версии обнаруживается критическая ошибка, создается ветка `hotfix` для исправления проблемы. После исправления она объединяется как в `master`, так и в `develop`.
- Новые версии программного обеспечения выпускаются путем слияния ветки `release` в `master`.
- Затем слияние `release` в `develop` обновляет последнюю разработку для будущих версий.

Что такое JWT?

JWT расшифровывается как "JSON Web Token" (токен веба на основе JSON). Это открытый стандарт (RFC 7519), который определяет компактный и самодостаточный способ представления информации между двумя сторонами в формате JSON. JWT часто используется для передачи данных аутентификации и авторизации между клиентом и сервером в безопасной и компактной форме.

JWT состоит из трех частей:

- Заголовок (Header): Содержит информацию о типе токена (часто это JWT) и алгоритме шифрования, используемом для создания подписи. Обычно выглядит так:
- Полезная нагрузка (Payload): Это место, где хранится сама информация токена. В JWT эта информация представляется в виде JSON и может включать любые данные, которые нужно передать между клиентом и сервером. Такие данные могут быть, например, идентификатор пользователя, срок действия токена, разрешения и прочее.
- Подпись (Signature): Подпись используется для проверки подлинности токена и обеспечивает его целостность. Он создается путем объединения заголовка и полезной нагрузки, а затем подписывается секретным ключом, который хранится на сервере. Таким образом, сервер может проверить, не были ли данные токена изменены в процессе передачи.



Что происходит когда ты отправляешь запрос с браузера?

- Ввод URL: Вы вводите URL (Uniform Resource Locator) в адресной строке браузера. URL представляет собой адрес веб-ресурса, который вы хотите посетить, например, "https://www.example.com".
- DNS запрос: Когда вы нажимаете Enter, браузер отправляет DNS (Domain Name System) запрос на DNS-сервер, чтобы разрешить доменное имя в IP-адрес. Это необходимо для того, чтобы определить, какой сервер хостит запрашиваемый веб-ресурс.
- Установка TCP соединения: После получения IP-адреса браузер устанавливает TCP (Transmission Control Protocol) соединение с сервером, который хостит веб-ресурс. TCP - это протокол передачи данных, который обеспечивает надежное и упорядоченное обмен сообщениями между устройствами через сеть.
- Отправка HTTP запроса: После установки TCP соединения браузер отправляет HTTP (Hypertext Transfer Protocol) запрос на сервер. Этот запрос содержит информацию о том, какой ресурс вы запрашиваете (например, конкретный путь на сервере, параметры запроса и т. д.).
- Обработка запроса на сервере: Сервер получает HTTP запрос и обрабатывает его. В зависимости от запрошенного ресурса и действий пользователя сервер может обратиться к базе данных, выполнить некоторые вычисления или просто вернуть запрошенные данные.
- Отправка HTTP ответа: После обработки запроса сервер формирует HTTP ответ, который содержит запрошенные данные или информацию об успешном выполнении операции. Этот ответ также отправляется через TCP соединение обратно в ваш браузер.
- Получение и отображение контента: Браузер получает HTTP ответ от сервера и анализирует его. Если это веб-страница, браузер обрабатывает HTML, CSS и JavaScript код, чтобы отобразить контент на экране. Если это другой тип файла, браузер может попробовать открыть его или предложить вам сохранить файл на компьютере.
- Закрытие TCP соединения: После завершения обработки HTTP ответа браузер закрывает TCP соединение с сервером.

Что такое TDD?

TDD (Test-Driven Development) – это методология разработки программного обеспечения, которая акцентирует внимание на написании тестов перед написанием кода функциональности. В TDD процесс разработки состоит из трех этапов: написания теста, написания кода и рефакторинга.

- Написание теста (Red): На первом этапе разработчик пишет автоматизированный тест, который должен проверить определенное поведение или функциональность системы. На этом этапе тест обычно не проходит и не выполняется, так как функциональность, которую он проверяет, еще не реализована.
- Написание кода (Green): На втором этапе разработчик начинает писать минимальное количество кода, необходимое для того, чтобы пройти тест, то есть сделать его успешным (зеленым). Главная цель – сделать код работающим, чтобы удовлетворить требования, определенные в тесте.
- Рефакторинг (Refactor): После успешного прохождения теста разработчик переходит к этапу рефакторинга. На этом этапе код улучшается и оптимизируется без изменения его функциональности. Цель состоит в том, чтобы улучшить структуру кода, сделать его более читаемым, поддерживаемым и эффективным, но без нарушения существующего поведения.

Процесс затем повторяется для каждой новой функциональности или изменения, которые необходимо внести в систему. По сути, TDD способствует улучшению качества кода, его надежности и уменьшению количества ошибок, так как каждый блок функциональности обязательно покрывается соответствующими тестами.

TDD также способствует улучшению документации проекта, так как тесты являются своеобразной спецификацией того, что должна делать система. Таким образом, другие разработчики могут понять предназначение каждой части кода и изменения, вносимые в него.

Что делает хороший код хорошим?

- Читаемость: Хороший код легко читается и понимается другими разработчиками. Он использует понятные и описательные имена переменных, функций и классов, а также имеет хорошую структуру и форматирование.
- Понятность: Хороший код ясно выражает свою цель и намерения. Он должен быть простым, без излишних усложнений, и каждая его часть должна иметь свою определенную задачу.
- Эффективность: Хороший код оптимизирован и эффективен в использовании ресурсов. Он не содержит лишних вычислений или операций, и обладает хорошей производительностью.
- Масштабируемость: Хороший код спроектирован с учетом будущего расширения и масштабирования. Он легко адаптируется к изменениям требований и росту проекта.
- Поддерживаемость: Хороший код удобен для поддержки и обновлений. Он разделен на модули, имеет хорошую документацию, комментарии и следует принципам хорошего программирования.
- Надежность: Хороший код минимизирует возможность ошибок и багов. Он проверяет входные данные на корректность, обрабатывает ошибки и предотвращает сбои.
- Тестируемость: Хороший код легко поддается автоматизированному тестированию. Он имеет хорошо структурированные модули и четкие интерфейсы, что упрощает написание тестов.
- Безопасность: Хороший код обеспечивает безопасность данных и защиту от возможных атак, таких как инъекции и переполнение буфера.
- Правильное использование ресурсов: Хороший код управляет ресурсами (памятью, файлами, сетью) должным образом и предотвращает утечки ресурсов.
- Соблюдение стандартов: Хороший код следует установленным стандартам кодирования и принятым правилам стиля в рамках проекта или команды разработчиков.

Вопросы Junior

Python теория

Типы данных Python

1. *None* (неопределенное значение переменной)
2. Логические переменные (*Boolean Type*)
3. Числа (*Numeric Type*)
    a. *int* – целое число
    b. *float* – число с плавающей точкой
    c. *complex* – комплексное число
4. Списки (*Sequence Type*)
    a. *list* – список
    b. *tuple* – кортеж
    c. *range* – диапазон
5. Строки (*Text Sequence Type*)
    a. str
6. Бинарные списки (*Binary Sequence Types*)
    a. *bytes* – байты
    b. *bytearray* – массивы байт
    c. *memoryview* – специальные объекты для доступа к внутренним данным объекта через protocol buffer
7. Множества (*Set Types*)
    a. *set* – множество
    b. *frozenset* – неизменяемое множество
8. Словари (*Mapping Types*)
    a. *dict* – словарь

Генератор и итератор

Итератор является более общей концепцией, чем генератор, и представляет собой любой объект, класс которого имеет методы __next__ и __iter__. Генератор – это итератор, который обычно создается путем вызова функции, содержащей не менее одного оператора yield. Это ключевое слово действует аналогично return, но возвращает объект-генератор.

Между ними существуют тонкие различия:

- Для генератора мы написали функцию, а для итератора можно использовать встроенные функции iter() и next().
- Для генератора используется ключевое слово yield для выдачи по одному объекту за раз.
- В генераторе может быть сколько угодно операторов yield.
- Генератор сохраняет текущее состояние локальных переменных (local variables) каждый раз, когда yield приостанавливает цикл (loop). Итератор не использует локальные переменные, он работает только с итерируемым объектом (iterable).
- Итератор можно использовать с помощью класса, а генератор — нет.
- Генераторы работают быстро, компактно и проще.
- Итераторы экономнее потребляют память.

Изменяемые и неизменяемые типы данных

Существуют изменяемые и неизменяемые типы.

К **неизменяемым** (*immutable*) типам относятся: целые числа (*int*), числа с плавающей точкой (*float*), комплексные числа (*complex*), логические переменные (*bool*), кортежи (*tuple*), строки (*str*) и неизменяемые множества (*frozen set*).

К **изменяемым** (*mutable*) типам относятся: списки (*list*), множества (*set*), словари (*dict*).

Как уже было сказано ранее, при создании переменной, вначале создается объект, который имеет уникальный идентификатор, тип и значение, после этого переменная может ссылаться на созданный объект.

Что быстрее dict, list, set, tuple?

Средняя временная сложность поиска в множествах и словарях соответствует O(1), в случае последовательностей O(n). Кортежи – это неизменяемый тип, поэтому они могут давать выигрыш в скорости перед списками.

Что такое исключения (exceptions) в Python и как их обрабатывать?

Исключения (exceptions) представляют собой ошибки, которые возникают во время выполнения программы и могут привести к прерыванию её нормального выполнения. Например, исключение может возникнуть при делении на ноль, доступе к несуществующему индексу списка или открытии файла, который не существует.

Когда исключение возникает, интерпретатор Python генерирует объект-исключение, который содержит информацию об ошибке, такую как тип ошибки и сообщение об ошибке. После генерации исключения программа останавливается, если исключение не обработано.

Для обработки исключений в Python используется конструкция `try-except`. Она позволяет предотвратить аварийное завершение программы и предоставляет возможность выполнить альтернативные действия в случае возникновения исключения

```
try:
    x = int(input("Введите число: "))
    result = 10 / x
except ZeroDivisionError as e:
    print("Ошибка деления на ноль:", e)
except ValueError as e:
    print("Ошибка преобразования строки в число:", e)
else:
    print("Результат:", result)
```

Какие различия между списками (lists) и кортежами (tuples) в Python, и в каких случаях лучше использовать каждый из них?

Списки (lists) и кортежи (tuples) являются двумя различными типами структур данных, которые предназначены для хранения упорядоченных коллекций элементов. Несмотря на некоторое сходство, у них есть важные различия, которые определяют их применение в различных сценариях.

Мутабельность:

- Списки: Являются изменяемыми (mutable), что означает, что после создания их можно модифицировать, добавлять, удалять элементы, а также изменять значения элементов.
- Кортежи: Являются неизменяемыми (immutable), их элементы не могут быть изменены после создания кортежа. После объявления кортежа, его элементы остаются неизменными.

Синтаксис:

- Списки: Объявляются с использованием квадратных скобок `[]`.
- Кортежи: Объявляются с использованием круглых скобок `()`.

Что такое условные операторы (if-elif-else) в Python и как они работают?

Условные операторы (if-elif-else) представляют собой механизм для выполнения различных блоков кода в зависимости от выполнения определенных условий. Они позволяют программе принимать решения и выполнять различные действия на основе значений переменных или результатов сравнений.

Когда Python встречает условный оператор, он проверяет условие в первом `if`. Если условие истинно, выполняется соответствующий блок кода и программа выходит из условного оператора. Если условие ложно, Python переходит к следующему блоку `elif` и проверяет его условие. Если условие истинно, выполняется соответствующий блок кода и программа выходит из условного оператора. Если ни одно из условий в `if` или `elif` не является истинным, выполняется блок кода в `else`.

```
x = 10

if x > 0:
    print("Число положительное")
elif x < 0:
    print("Число отрицательное")
else:
    print("Число равно нулю")
```

Что такое циклы (loops) в Python, какие типы циклов существуют, и для чего они используются?

Циклы (loops) представляют собой механизмы, которые позволяют выполнять определенный блок кода несколько раз. Они позволяют автоматизировать повторяющиеся задачи и обрабатывать коллекции элементов, такие как списки или строки.

Цикл `for`: Цикл `for` предназначен для обхода элементов в итерируемом объекте, таком как список, кортеж, строка или словарь. Он выполняет заданный блок кода для каждого элемента в итерируемом объекте, пока не пройдет через все элементы или не будет выполнено условие прерывания

Цикл `while`: Цикл `while` выполняет блок кода, пока условие истинно. Он продолжает выполнение, пока условие не станет ложным. Цикл `while` особенно полезен, когда количество итераций не известно заранее

Что такое модули (modules) в Python и какие преимущества они предоставляют для организации кода?

Модули (modules) - это файлы, содержащие определения функций, классов и переменных, которые можно использовать в других программах. Модули позволяют разделять код на логические блоки, упрощают организацию и структурирование кода, а также способствуют повторному использованию кода.

Модульность: Модули позволяют разделять код на небольшие, независимые блоки, что делает программу более понятной и облегчает её сопровождение и обновление.

Повторное использование: Один и тот же модуль можно использовать в разных программах или частях программы, что снижает дублирование кода и упрощает разработку.

Импорт: Модули могут быть импортированы в другие программы с помощью ключевого слова `import`. Это позволяет использовать функции, классы и переменные из другого модуля, необходимые для выполнения определенной задачи.

Что такое методы (methods) классов в Python и как они отличаются от обычных функций?

Методы (methods) классов - это функции, которые определены внутри классов и предназначены для работы с объектами этого класса. Они позволяют определить поведение объектов и обеспечивают доступ к данным объектов, а также манипулирование этими данными.

Синтаксис: Методы определяются внутри классов и имеют доступ к атрибутам и методам класса через ключевое слово `self`. Обычные функции не связаны с классами и не имеют доступа к атрибутам классов.

Связь с объектами: Методы классов являются частью определения класса и работают с экземплярами (объектами) этого класса. При вызове метода у объекта, он получает доступ к его атрибутам и может модифицировать их значения.

```python
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def get_area(self):
        """Метод для вычисления площади круга."""
        return 3.14 * self.radius**2

    def get_circumference(self):
        """Метод для вычисления длины окружности."""
        return 2 * 3.14 * self.radius

# Создаем объект класса Circle
my_circle = Circle(5)

# Вызываем методы объекта
area = my_circle.get_area()
circumference = my_circle.get_circumference()

print("Площадь круга:", area)
print("Длина окружности:", circumference)
```

Что такое словари (dictionaries) в Python и каким образом они устроены? Какие преимущества они предоставляют при работе с данными?

Словари (dictionaries) - это структуры данных, представляющие собой неупорядоченные коллекции элементов, состоящих из пар ключ-значение. Каждый элемент в словаре состоит из уникального ключа и связанного с ним значения. Словари позволяют быстро находить и получать доступ к данным по ключу, что делает их очень эффективными для работы с большим объемом информации.

Ключи в словаре должны быть уникальными и неизменяемыми объектами, такими как строки, числа или кортежи. Обычно, ключи используются для обозначения свойств или идентификации данных.

Значения в словаре могут быть любого типа данных, включая числа, строки, списки, другие словари и т. д. Каждому ключу соответствует одно значение.

Быстрый доступ: Словари обеспечивают быстрый доступ к данным по ключу. При поиске значения по ключу, Python использует хэш-таблицы, что позволяет получить доступ к элементам словаря за константное время, даже при большом количестве элементов.

Что такое рекурсия в Python и как она работает? В чём заключаются преимущества и ограничения использования рекурсии?

Рекурсия - это техника, при которой функция вызывает саму себя для решения задачи. Когда функция вызывает саму себя, она создает новый экземпляр функции, который работает независимо от оригинального вызова, и таким образом, решает более простую версию задачи. Процесс повторяется до тех пор, пока не будет достигнуто базовое (терминальное) условие, которое указывает на завершение рекурсии.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Что такое PEP 8 и какую роль он играет в разработке на Python?

PEP 8 (Python Enhancement Proposal 8) - это руководство по стилю кодирования для языка Python. Он определяет рекомендации и правила, которые помогают разработчикам писать читаемый, согласованный и понятный код. PEP 8 разработана для обеспечения единого стиля программирования в сообществе Python.

PEP 8 включает в себя рекомендации по оформлению кода, именованию переменных, использованию отступов, длине строк, комментированию кода и многому другому. Эти рекомендации помогают сделать код более понятным, удобочитаемым и легким для сопровождения.

Улучшение читаемости: Согласованный стиль кодирования, определенный в PEP 8, облегчает понимание кода другими разработчиками и повышает читаемость программы.

Согласованность в сообществе: PEP 8 обеспечивает единство стиля программирования в сообществе Python. Согласованный код становится более понятным и удобным для совместной работы.

Удобство сопровождения: Соблюдение рекомендаций PEP 8 делает код более предсказуемым и легким для сопровождения. Разработчики, работающие с кодом, быстро адаптируются к единому стилю и могут легко идентифицировать ошибки или проблемы.

Что такое виртуальное окружение (virtual environment) в Python и зачем его использовать? Как создать и активировать виртуальное окружение?

Виртуальное окружение (virtual environment) - это изолированное пространство, в котором можно устанавливать пакеты и зависимости для проекта независимо от системной установки Python. Использование виртуальных окружений позволяет избежать конфликтов между версиями пакетов разных проектов и обеспечивает портабельность кода.

Изоляция: Виртуальные окружения позволяют изолировать зависимости проекта от глобальной установки Python и других проектов, предотвращая конфликты между различными версиями пакетов.

Чистота: Виртуальное окружение позволяет создавать "чистые" и незагрязненные среды для разных проектов, что способствует более чистому и понятному коду.

Портабельность: Виртуальное окружение позволяет легко переносить проекты между разными системами без необходимости переустановки пакетов.

Что такое list comprehension (генератор списка) в Python и какие преимущества они предоставляют для работы с данными?

List comprehension (генератор списка) - это компактный способ создания нового списка путем применения выражения к каждому элементу другого итерируемого объекта, такого как список, кортеж или строка. Он предоставляет удобный и читаемый синтаксис для быстрого формирования списков на основе существующих данных.

Краткость и удобочитаемость: List comprehension позволяет создавать новые списки с минимальным количеством кода, что делает его более компактным и легким для чтения и понимания.

Высокая производительность: List comprehension обычно работает быстрее, чем обычные циклы `for`, так как он использует внутренние оптимизации языка Python.

Возможность фильтрации данных: List comprehension позволяет применять условия для отбора элементов из исходного итерируемого объекта, что делает его мощным инструментом для фильтрации данных.

```
numbers = [x for x in range(1, 11)]
print(numbers)
# Вывод: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Что такое дандер методы?

"дандер" методы (или магические методы) - это специальные методы, имена которых начинаются и заканчиваются двойными подчеркиваниями, например, `__init__`, `__str__`, `__add__` и т. д. Эти методы предоставляют специальное поведение для классов и позволяют переопределять стандартные операции, такие как инициализация объектов, представление в виде строки, арифметические операции и другие.

Дандер методы автоматически вызываются интерпретатором Python при выполнении определенных операций. Например, когда вы создаете объект, вызывается метод `__init__`, который позволяет инициализировать его атрибуты. Когда вы вызываете функцию `print()` для объекта, вызывается метод `__str__`, чтобы получить строковое представление объекта.

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

    def __str__(self):
        return f"Имя: {self.name}, Возраст: {self.age}"

person = Person("Алексей", 30)
print(person)  # Вывод: Имя: Алексей, Возраст: 30
```

Разница между __new__ и __init__?

Есть два специальных метода класса: `__new__` и `__init__`. Они оба связаны с процессом создания объектов, но выполняют разные задачи и вызываются в разные моменты жизненного цикла объекта.

`__new__` является статическим методом класса (не привязан к экземпляру), который вызывается для создания нового экземпляра класса. Он отвечает за выделение памяти и создание объекта до его инициализации (инициализация выполняется методом `__init__`). Возвращает новый экземпляр класса. Первый аргумент метода - сам класс (`cls`), остальные аргументы могут быть использованы для передачи в метод `__init__`.

`__init__` является методом экземпляра класса (привязан к объекту), который вызывается после создания объекта (после метода `__new__`) и используется для инициализации атрибутов объекта.
Он не возвращает ничего, а просто инициализирует значения атрибутов объекта.
В `__init__` обычно производятся все необходимые действия для настройки объекта перед его использованием.

```python
class MyClass:
    def __new__(cls, *args, **kwargs):
        print("Метод __new__ вызван")
        instance = super().__new__(cls)
        return instance

    def __init__(self, x):
        print("Метод __init__ вызван")
        self.x = x
```

Какая разница между одинарным (_) и двойным (__) подчеркиванием?

Разница между одинарным и двойным подчеркиванием касается их семантики и поведения в контексте именования и доступа к атрибутам и методам классов.

Одиночное подчеркивание: Имена, начинающиеся с одиночного подчеркивания _, считаются конвенцией именования, которая дает понять программисту, что переменная или метод являются "приватными" или предназначены для внутреннего использования внутри класса или модуля. Одиночное подчеркивание само по себе не влияет на поведение объектов или переменных, но оно служит предупреждением для других разработчиков, чтобы не обращаться к этим именам извне класса или модуля.

Двойное подчеркивание: Имена, начинающиеся с двойного подчеркивания __, имеют специальное поведение и подвержены механизму "манглинга" (name mangling). Механизм манглинга изменяет имя переменной, добавляя перед ним префикс `_ИмяКласса`, чтобы предотвратить случайное перекрытие имен между подклассами. Двойное подчеркивание позволяет создавать "защищенные" атрибуты, которые не будут доступны извне класса или его подклассов.

Чем отличаются и как используются декораторы @classmethod и @staticmethod?

Декораторы `@classmethod` и `@staticmethod` предоставляют специальные методы класса, которые имеют различные характеристики и используются в разных сценариях.

`@classmethod`:

- Декоратор `@classmethod` применяется к методу класса и обозначает, что этот метод является методом класса, а не методом экземпляра.
- Метод класса принимает первый аргумент `cls`, который ссылается на сам класс, а не на экземпляр класса. Обычно этот аргумент именуется `cls` по соглашению, но имя может быть любым.
- Методы класса часто используются для создания альтернативных конструкторов или для доступа к общим атрибутам и методам класса, которые могут быть общими для всех экземпляров.

`@staticmethod`:

- Декоратор `@staticmethod` применяется к методу класса и обозначает, что этот метод является статическим методом.
- Статические методы не требуют доступа к экземпляру класса или к его атрибутам. Они могут быть вызваны непосредственно через класс без создания экземпляра класса.
- Статические методы часто используются для группировки функциональности, которая связана с классом, но не требует доступа к его атрибутам или состоянию.

Что такое property?

`property` - это встроенный декоратор, который позволяет превратить метод класса в атрибут, при этом обеспечивая контроль над доступом, чтением и записью значения этого атрибута. Это позволяет создавать атрибуты с "геттерами", "сеттерами" и "делятерами", которые могут быть использованы для управления значением и поведением объекта.

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        print("Чтение радиуса")
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Радиус не может быть отрицательным")
        print("Запись радиуса")
        self._radius = value

circle = Circle(5)
print(circle.radius)  # Вывод: Чтение радиуса \n 5
circle.radius = 10  # Вывод: Запись радиуса
print(circle.radius)  # Вывод: Чтение радиуса \n 10
```

Что такое enumerate?

`enumerate()` - это встроенная функция, которая используется для перебора элементов последовательности (например, списков, кортежей, строк и т.д.) и получения пары (индекс, элемент) для каждого элемента. Таким образом, она предоставляет удобный способ получить итератор с парами индексов и значений элементов последовательности.

```
fruits = ['apple', 'banana', 'orange']

for index, fruit in enumerate(fruits):
    print(f"Index: {index}, Fruit: {fruit}")
```

В чем разница между copy или deepcopy?

`copy()`: Эта функция используется для создания поверхностной копии (shallow copy) объекта. Поверхностная копия создает новый объект, но копирует только ссылки на элементы из оригинального объекта, а не копирует сами элементы. Если элементы являются изменяемыми (например, списки или словари), изменения в копии будут отражаться в оригинале и наоборот.

`deepcopy()`: В отличие от поверхностной копии, эта функция создает глубокую копию (deep copy) объекта. Глубокая копия создает новый объект и рекурсивно копирует все элементы во всех вложенных структурах данных (например, во вложенных списках или словарях). Таким образом, изменения в глубокой копии не влияют на оригинал, и наоборот.

Python - интерпретируемый язык или компилируемый?

Python является интерпретируемым языком программирования. Это означает, что код на Python выполняется путем интерпретации пошагово средой исполнения (интерпретатором) вместо того, чтобы быть предварительно скомпилированным в машинный код, как это происходит в компилируемых языках.

Когда вы запускаете программу на Python, интерпретатор читает и анализирует код по одной строке или блоку кода за раз и немедленно выполняет его. Это означает, что Python код может быть исполнен на лету без необходимости явной компиляции в машинный код. Интерпретируемый подход делает язык Python очень гибким и удобным для разработки и тестирования, так как изменения в коде могут быть видны сразу же после их внесения.

Однако интерпретация может привести к некоторому уменьшению производительности по сравнению с полностью компилируемыми языками, так как код на Python обрабатывается в режиме реального времени интерпретатором. Тем не менее, большинство задач в разработке программного обеспечения на Python не требуют высокой производительности, и простота и быстрота разработки делают язык популярным выбором для множества приложений.

Какие есть виды импорта?

Импорт модуля целиком:

```
import module_name
```

Импорт модуля с псевдонимом:

```
import module_name as alias
```

Импорт конкретных элементов из модуля:

```
from module_name import function_name, class_name, variable_name
```

Импорт всех элементов из модуля:

```
from module_name import *
```

Динамический импорт:

```
module_name = __import__('module_name')
```

```
import importlib
module_name = importlib.import_module('module_name')
```

Что такое область видимости переменных?

Область видимости переменных - это часть программы, где определенная переменная имеет смысл и доступна для использования. Каждая переменная в Python имеет свою область видимости, что означает, что она может быть использована только в определенной части кода, а не везде в программе.

Глобальная область видимости (Global scope): Переменные, которые определены вне всех функций или классов, имеют глобальную область видимости. Это означает, что такие переменные доступны в любой части программы после их определения и до конца выполнения программы. Глобальные переменные можно использовать внутри функций или классов, но для изменения их значения внутри функции или класса необходимо использовать ключевое слово `global`.

Локальная область видимости (Local scope): Переменные, определенные внутри функций или классов, имеют локальную область видимости. Это означает, что такие переменные существуют только внутри тела функции или класса и не видны за его пределами. Локальные переменные могут быть использованы только внутри функции или класса, где они были объявлены.

Как можно преобразовать строку (string) в нижний регистр (lowercase)?

В Python можно преобразовать строку в нижний регистр с помощью метода `lower()` для строк. Этот метод возвращает новую строку, в которой все символы исходной строки приведены к нижнему регистру.

```
my_string = "Hello, World!"
lowercase_string = my_string.lower()
print(lowercase_string)
```

Для преобразования в верхний регистр (uppercase) используется метод upper(). Еще есть методы isupper() (все символы в верхнем регистре) и islower() (все символы в нижнем регистре), которые проверяют регистр всех символов имени. Еще есть метод istitle(), который проверяет строку на стиль заголовка (все слова должны начинаться с символа в верхнем регистре)

Для чего нужен pass (pass statement) в питоне? Зачем нужны break и continue?

`pass`, `break` и `continue` - это ключевые слова, которые позволяют управлять потоком выполнения программы в различных ситуациях.

`pass` - это пустой оператор, который ничего не делает. Он используется там, где синтаксически необходим оператор, но ничего делать не требуется, например, в теле функции или цикла. `pass` обеспечивает синтаксическую корректность программы, когда вам нужно объявить блок кода, но его содержание пока не определено или не требуется.

`break` используется внутри циклов (например, `for` или `while`) для прерывания выполнения цикла, когда выполняется определенное условие. Как только `break` достигается внутри цикла, выполнение выходит из цикла, и управление передается следующей инструкции после цикла.

`continue` также используется внутри циклов и предназначен для перехода к следующей итерации цикла, минуя оставшуюся часть текущей итерации. Когда `continue` достигается внутри цикла, оставшийся код в текущей итерации не выполняется, и управление передается следующей итерации.

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        continue  # Пропуск вывода числа 3
    print(num)
```

```
def some_function():
    pass  # Пока функция пустая, но она не вызовет ошибку
```

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        break  # Выход из цикла, когда num равно 3
    print(num)
```

Что такое класс?

В программировании, класс - это шаблон или абстрактный тип данных, который определяет состояние и поведение объектов, созданных на его основе. Классы являются основной концепцией объектно-ориентированного программирования (ООП) и позволяют объединить данные и методы (функции) для работы с этими данными в единый объект.

Классы могут содержать переменные класса (атрибуты), которые хранят данные, и методы класса (функции), которые определяют поведение объектов, созданных на основе класса. Когда класс определен, он становится типом данных, и объекты этого класса называются экземплярами класса или просто объектами.

```
class MyClass:
    # Переменные класса (атрибуты)
    attribute1 = "Value 1"
    attribute2 = 42

    # Методы класса (функции)
    def method1(self):
        print("This is method 1")

    def method2(self, parameter):
        print("This is method 2 with parameter:", parameter)
```

Что такое функция?

Функция - это блок кода, который выполняет определенную задачу или набор задач. Функции используются для группировки повторяющегося кода, чтобы улучшить читаемость, обеспечить повторное использование и упростить структуру программы.

Функции в Python имеют имя, список параметров (необязательно) и блок кода, который выполняется, когда функция вызывается. Определение функции начинается с ключевого слова `def`, за которым следует имя функции, а затем список параметров в круглых скобках. Тело функции должно быть с отступом и выполняет определенные операции.

```
def greet(name):
    print("Hello, " + name + "!")

# Вызов функции
greet("Alice")
```

Что такое слайсинг в пайтон?

Слайсинг (slicing) в Python - это механизм извлечения подпоследовательности элементов из последовательности, такой как строка, список, кортеж и другие итерируемые объекты. Слайсинг позволяет получить часть последовательности, указав начальный индекс, конечный индекс и шаг (step).

```
sequence[start:stop:step]
```

`start` (опционально): индекс начала слайсинга. Если не указан, используется начало последовательности

`stop`: индекс конца слайсинга. Подпоследовательность будет содержать элементы до этого индекса

`step` (опционально): шаг, с которым нужно извлекать элементы. Если не указан, используется шаг равный 1

Что такое декоратор?

В Python декоратор (decorator) - это функция, которая позволяет изменить поведение другой функции или метода без изменения его собственного кода. Декораторы позволяют добавлять функциональность к существующим функциям, делая код более модульным и переиспользуемым.

Декораторы реализуются с использованием функций высшего порядка (higher-order functions), то есть функций, которые принимают другую функцию в качестве аргумента или возвращают другую функцию в качестве результата. Они применяются с помощью символа `@` перед определением функции.

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Для чего нужны комментарии в коде?

Объяснение кода: Комментарии помогают объяснить, что делает определенный кусок кода, особенно если код содержит сложные алгоритмы или не очевидную логику.

Документирование функций и классов: Комментарии могут быть использованы для описания назначения функций и классов, их параметров, возвращаемых значений и т.д. Это облегчает понимание использования функций и классов другими разработчиками.

Отладка: Комментарии могут использоваться для временного отключения определенных частей кода во время отладки.

Пометки для будущего изменения: Комментарии могут содержать пометки или напоминания для будущих изменений или улучшений в коде.

В Python комментарии начинаются с символа `#` и продолжаются до конца строки. Комментарии могут быть однострочными или многострочными.

Для чего нужны docstring?

Docstring (документирующая строка) в Python - это специальный тип комментариев, который используется для документирования функций, классов и модулей. Docstring предоставляет документацию, описывающую назначение, параметры, возвращаемые значения и примеры использования функций или классов, что делает код более понятным и помогает другим разработчикам (и себе в будущем) легко понимать и использовать ваш код.

Читаемость и понимание: Docstring помогает другим разработчикам и вам самим легче понять назначение и функциональность функций и классов, особенно когда код становится сложным или содержит большое количество функций.

Документация кода: Docstring является формой документации кода внутри самого кода. Это упрощает поддержку кода, так как разработчикам не нужно искать внешнюю документацию отдельно.

Автодокументирование: Некоторые инструменты, такие как `help()` или автодокументирующиеся инструменты, могут извлекать информацию из docstring и предоставлять ее в формате справки.

```
def add_numbers(a, b):
    """
    Функция для вычисления суммы двух чисел.

    Параметры:
        a (число): Первое слагаемое.
        b (число): Второе слагаемое.

    Возвращает:
        int: Сумма двух чисел.

    Пример использования:
        >>> add_numbers(3, 5)
        8
    """
    return a + b
```

Можно ли использовать несколько декораторов для одной функции?

Да, в Python можно использовать несколько декораторов для одной функции. Это называется "стеком декораторов" (decorators stacking). Когда функция декорируется несколькими декораторами, они применяются к функции в порядке, в котором они указаны сверху вниз.

```
@decorator1
@decorator2
@decorator3
def my_function():
    # Тело функции
    pass
```

Можно ли создать декоратор из класса?

Да, в Python можно создать декоратор из класса. Декораторы, определенные как классы, предоставляют другой способ создания декораторов с использованием объектно-ориентированного подхода. Для того чтобы класс мог быть использован в качестве декоратора, он должен определить методы `__init__()` и `__call__()`.

Метод `__init__()` инициализирует объект класса и позволяет передавать аргументы декоратору (если необходимо).

Метод `__call__()` делает объект класса вызываемым. Этот метод будет вызываться, когда декорируемая функция будет вызываться.

Какие есть основные популярные пакеты?

`requests`: Пакет `requests` используется для работы с HTTP-запросами. Он обеспечивает удобный интерфейс для отправки HTTP-запросов на удаленные серверы и обработки ответов.

`pytest`: `pytest` - это фреймворк для тестирования в Python. Он предоставляет простой синтаксис для написания тестов и автоматического обнаружения и выполнения тестовых функций.

`numpy`: `numpy` - это библиотека для работы с многомерными массивами и математическими функциями. Он предоставляет эффективные инструменты для работы с числовыми данными и вычислений.

`django` и `Flask`: `django` и `Flask` - это популярные фреймворки для веб-разработки на Python. Они облегчают создание веб-приложений и веб-сервисов.

`tensorflow` и `pytorch`: `tensorflow` и `pytorch` - это библиотеки для глубокого обучения. Они предоставляют множество инструментов и функций для создания, обучения и развертывания нейронных сетей.

Что такое lambda-функции?

Lambda-функции (анонимные функции) - это специальный тип функций в Python, которые могут быть определены в одной строке кода без необходимости использования ключевого слова `def`. Они используются для создания простых функций с одним выражением, которые могут быть переданы как аргументы другим функциям или использованы для выполнения простых операций.

```python
# Обычная функция для сложения двух чисел
def add(a, b):
    return a + b

result = add(3, 5)
print(result)  # Вывод: 8

# Та же функция, но в виде lambda-функции
add_lambda = lambda a, b: a + b
result_lambda = add_lambda(3, 5)
print(result_lambda)  # Вывод: 8
```

Что означает *args, **kwargs и как они используются?

`*args` и `**kwargs` - это специальные параметры функций в Python, которые позволяют передавать переменное количество аргументов в функцию. Они часто используются, когда вы не знаете точное количество аргументов, которые будут переданы функции.`*args`:

`*args` используется для передачи не именованных аргументов или аргументов, которые не имеют ключевого слова, в функцию. При использовании `*args` аргументы передаются в виде кортежа. Название `args` не обязательно. Это просто договоренность, что символ `*` перед именем аргумента используется для сбора не именованных аргументов.

`**kwargs` используется для передачи именованных аргументов (ключ-значение) в функцию. При использовании `**kwargs` аргументы передаются в виде словаря, где ключи - это имена аргументов, а значения - соответствующие значения.

Что знаете из модуля collections, какими еще built-in модулями пользовались?

Из модуля `collections` в Python, я знаю несколько полезных структур данных и инструментов:

- `namedtuple`: `namedtuple` - это фабрика для создания именованных кортежей (named tuples). Именованные кортежи представляют собой неизменяемые кортежи, у которых элементы доступны по именам, что делает их более читаемыми и понятными.
- `defaultdict`: `defaultdict` - это подкласс словаря (dict), который предоставляет значение по умолчанию для отсутствующих ключей. Это удобно, когда вы работаете с словарями и не хотите проверять наличие ключей перед доступом к значениям.
- `Counter`: `Counter` - это словарь, предназначенный для подсчета элементов в последовательности. Он предоставляет удобные методы для подсчета повторяющихся элементов и получения наиболее часто встречающихся элементов.
- `deque`: `deque` - это двусторонняя очередь (double-ended queue), которая предоставляет эффективные операции добавления и удаления элементов как с начала, так и с конца очереди.

Кроме модуля `collections`, есть еще множество встроенных модулей в Python, которые предоставляют различные функции и возможности. Некоторые из них:

- `os`: Модуль `os` предоставляет функции для работы с операционной системой, такие как создание/удаление директорий, работа с файлами, получение информации о файловой системе и др.
- `datetime`: Модуль `datetime` предоставляет классы для работы с датами и временем, а также функции для форматирования и разбора дат и времени.
- `math`: Модуль `math` предоставляет математические функции и константы для выполнения различных вычислений.
- `random`: Модуль `random` предоставляет функции для генерации случайных чисел и элементов.
- `json`: Модуль `json` предоставляет функции для работы с данными в формате JSON (JavaScript Object Notation).
- `re`: Модуль `re` предоставляет поддержку регулярных выражений для поиска и обработки текста.

Как Python работает с HTTP-сервером?

Python имеет различные способы работы с HTTP-серверами, но одним из наиболее популярных и широко используемых способов является использование стандартной библиотеки `http.server`. Данная библиотека предоставляет простые классы для создания HTTP-серверов для обслуживания статических файлов и обработки HTTP-запросов.

```python
import http.server
import socketserver

# Задаем IP-адрес и порт сервера
IP_ADDRESS = "127.0.0.1"
PORT = 8000

# Создаем обработчик запросов, наследуясь от класса http.server.BaseHTTPReques
class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header("Content-type", "text/html")
        self.end_headers()
        self.wfile.write(b"Hello, world!")

# Запускаем HTTP-сервер на указанном адресе и порте с заданным обработчиком
with socketserver.TCPServer((IP_ADDRESS, PORT), MyHandler) as server:
    print(f"Serving on http://{IP_ADDRESS}:{PORT}")
    server.serve_forever()
```

Что такое контекстные менеджеры?

Контекстные менеджеры (Context Managers) - это объекты в Python, которые позволяют определить и управлять контекстом выполнения блока кода с помощью ключевого слова `with`. Контекстные менеджеры очень полезны для автоматического управления ресурсами, такими как файлы, сетевые соединения, базы данных и т.д., гарантируя, что ресурсы будут правильно открыты и закрыты, даже если происходит исключение.

Для создания контекстного менеджера в Python, объект должен реализовать два метода: `__enter__()` и `__exit__()`. Эти методы определяют, что должно произойти при входе в контекст (начало блока `with`) и при выходе из контекста (окончание блока `with`).

```python
class MyFileManager:
    def __init__(self, filename):
        self.filename = filename

    def __enter__(self):
        self.file = open(self.filename, 'r')
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

# Использование контекстного менеджера
with MyFileManager('example.txt') as file:
    content = file.read()
    print(content)
# Выйдя из блока 'with', файл автоматически закроется
```

Разница между is и ==?

В Python операторы `is` и `==` выполняют сравнение объектов, но они имеют разное поведение, потому что сравнивают разные аспекты объектов.

`is` используется для проверки, являются ли две переменные одним и тем же объектом в памяти. Если две переменные ссылаются на один и тот же объект, то оператор `is` вернет `True`, иначе вернет `False`. `is` проверяет идентичность объектов, то есть они указывают на один и тот же участок памяти.

`==` используется для сравнения значений объектов, то есть проверки на равенство значений. Если две переменные содержат одинаковые значения, то оператор `==` вернет `True`, иначе вернет `False`. `==` сравнивает значения объектов, независимо от того, являются ли они одним и тем же объектом в памяти.

Как работает хеширование в Python?

В Python, хеширование - это процесс преобразования данных в хеш-значение с помощью хеш-функции. Хеш-значение - это уникальная строка фиксированного размера, которая представляет входные данные. Хеширование является важным аспектом многих алгоритмов и структур данных в Python и используется в различных областях программирования.

В Python для хеширования используется встроенная функция `hash()`. Эта функция принимает один аргумент (хешируемый объект) и возвращает целочисленное значение хеша.

```
# Хеширование чисел
print(hash(42))  # Вывод: 42
print(hash(3.14))  # Вывод: 1152921504606846985

# Хеширование строк
print(hash("hello"))  # Вывод: -4462666988985169410
print(hash("world"))  # Вывод: 1459122677275263495

# Хеширование кортежей
print(hash((1, 2, 3)))  # Вывод: 2528502973977326415
```

Как отформатировать строку Python?

Используя оператор %: Это старый способ форматирования строк, но все еще поддерживается в Python. Вы можете использовать оператор `%` для вставки значений в строку.

```
name = "John"
age = 30
formatted_string = "Меня зовут %s и мне %d лет" % (name, age)
print(formatted_string)
```

С использованием метода `format()`: Метод `format()` позволяет вставлять значения в строку, используя фигурные скобки вместо оператора `%`. Пример:

```
name = "John"
age = 30
formatted_string = "Меня зовут {} и мне {} лет".format(name, age)
print(formatted_string)
```

Используя f-строки (f-strings): С f-строками в Python 3.6+ появился удобный способ форматирования строк. Вы можете вставлять значения прямо в строку, обрамляя переменные в фигурные скобки и предварив строку префиксом `f`. Пример:

```
name = "John"
age = 30
formatted_string = f"Меня зовут {name} и мне {age} лет"
print(formatted_string)
```

Что такое map()?

`map()` применяет указанную функцию к каждому элементу последовательности (например, списку) и возвращает итератор с результатами преобразования. Синтаксис: `map(function, iterable)`

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

Что такое filter()?

`filter()` используется для отбора элементов из последовательности на основе условия, заданного в виде функции. Он возвращает итератор, содержащий только те элементы, для которых условие истинно. Синтаксис: `filter(function, iterable)`

Что такое reduce()?

`reduce()` применяет указанную функцию к элементам последовательности слева направо с аккумулированием результатов. Он возвращает одно значение, а не итератор. Обратите внимание, что с Python 3 `reduce()` был перенесен из встроенных функций в модуль `functools`. Чтобы использовать `reduce`, нужно импортировать его: `from functools import reduce`. Синтаксис: `reduce(function, iterable[, initializer])`

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers)  # Output: 15 (1 + 2 + 3 + 4 + 5)
```

Что такое id()?

Функция `id()` используется для получения уникального идентификатора (ID) объекта. Этот идентификатор представляет собой целочисленное значение, которое гарантированно уникально для каждого объекта во время его существования. При этом, не обязательно, чтобы идентификаторы были последовательными или увеличивались с каждым созданием нового объекта.

```
a = [1, 2, 3]
b = a

print(id(a))  # Пример вывода: 2400321179328
print(id(b))  # Пример вывода: 2400321179328

# Проверяем, являются ли a и b одним и тем же объектом
print(a is b)  # Output: True
```

Для чего нужна функция dir()?

Функция `dir()` в Python используется для получения списка имен (атрибутов) в указанном пространстве имен (объекте). Она позволяет получить список всех методов и атрибутов, которые доступны в объекте, включая встроенные методы и атрибуты, а также те, которые были определены пользователем.

```
x = [1, 2, 3]
print(dir(x))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
```

Что такое абстрактный класс?

Абстрактный класс - это класс в объектно-ориентированном программировании, который не предназначен для создания объектов напрямую. Он служит в качестве шаблона или базового класса для других классов и определяет общие характеристики и поведение для группы связанных классов, которые будут его подклассами.

От абстрактных классов нельзя создать экземпляр, так как они содержат абстрактные методы, которые не имеют реализации в самом абстрактном классе. Абстрактные методы предназначены для переопределения в подклассах, и каждый подкласс должен обязательно предоставить свою реализацию абстрактных методов.

В языке программирования Python абстрактные классы реализуются с помощью модуля `abc` (Abstract Base Classes). Этот модуль предоставляет декораторы и классы для определения абстрактных классов и методов.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass
```

```python
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius**2

    def perimeter(self):
        return 2 * 3.14 * self.radius

# Нельзя создать объект от абстрактного класса Shape
# shape = Shape()  # Вызовет ошибку TypeError

circle = Circle(5)
print("Площадь круга:", circle.area())
print("Периметр круга:", circle.perimeter())
```

Что такое геттеры, сеттеры?

Геттеры и сеттеры - это методы в объектно-ориентированном программировании, которые позволяют управлять доступом к свойствам (атрибутам) объекта. Они обеспечивают инкапсуляцию данных и позволяют установить ограничения на доступ к ним.

Геттеры (Getter methods): Геттер - это метод, который используется для получения значения определенного свойства объекта. Он предоставляет доступ к приватным атрибутам объекта, не позволяя напрямую обращаться к ним извне класса. Геттеры обычно именуются с префиксом get_, за которым следует имя свойства, которое он возвращает.

Сеттеры (Setter methods): Сеттер - это метод, который используется для установки значения определенного свойства объекта. Он позволяет контролировать и валидировать присваиваемые значения. Сеттеры обычно именуются с префиксом set_, за которым следует имя свойства, которое он устанавливает.

```
class Person:
    def __init__(self, name):
        self._name = name  # Приватный атрибут

    def get_name(self):
        return self._name

person = Person("Alice")
print(person.get_name())  # Output: "Alice"
```

```
class Person:
    def __init__(self, name):
        self._name = name  # Приватный атрибут

    def get_name(self):
        return self._name

    def set_name(self, new_name):
        if len(new_name) > 0:
            self._name = new_name

person = Person("Alice")
print(person.get_name())  # Output: "Alice"

person.set_name("Bob")
print(person.get_name())  # Output: "Bob"

person.set_name("")  # Не изменится, так как длина имени равна 0
print(person.get_name())  # Output: "Bob"
```

Как расшифровать LEGB?

LEGB - это акроним, который используется для объяснения правил разрешения области видимости (поиска имен) в Python. Когда в коде встречается имя переменной, интерпретатор Python ищет это имя в различных областях видимости, определяемых четырьмя уровнями LEGB:

- L - Local (локальная область видимости): Это область видимости, которая охватывает текущую функцию или блок кода. Имена, определенные внутри этой функции или блока, считаются локальными и могут быть использованы только внутри него. Когда функция завершает свою работу или блок кода завершается, локальные переменные уничтожаются.
- E - Enclosing (область видимости замыкания): Это область видимости, которая охватывает вложенные функции (функции, определенные внутри других функций). Если внутри вложенной функции используется имя, которое не определено в ее локальной области видимости, интерпретатор будет искать это имя в области видимости замыкания, то есть в области видимости внешней функции.
- G - Global (глобальная область видимости): Это область видимости, которая охватывает весь модуль (файл). Имена, определенные на верхнем уровне файла, считаются глобальными и могут быть использованы в любой функции или блоке кода в этом модуле. Имена, объявленные на верхнем уровне файла, также доступны внутри функций, если они не были переопределены локально.
- B - Built-in (встроенная область видимости): Это область видимости, которая содержит имена встроенных функций, модулей и исключений Python, таких как `print()`, `len()`, `sum()`, и т.д. Эти имена доступны в любой области видимости без необходимости импорта.

Когда интерпретатор Python ищет имя переменной, он следует порядку LEGB: сначала ищет в локальной области видимости, затем в области видимости замыкания, далее в глобальной области видимости и, наконец, встроенной области видимости. Если интерпретатор находит имя в одной из этих областей, он использует найденное значение, иначе возникает ошибка "NameError".

Что такое next()?

`next()` - это встроенная функция, которая используется для получения следующего элемента из итерируемого объекта. Итерируемый объект - это объект, который поддерживает итерацию или последовательный доступ к его элементам, такой как список, кортеж, строка, словарь и т.д.

```
my_list = [1, 2, 3, 4]
my_iter = iter(my_list)

print(next(my_iter))  # Output: 1
print(next(my_iter))  # Output: 2
print(next(my_iter))  # Output: 3
print(next(my_iter))  # Output: 4

# Итератор исчерпан, вызовет ошибку StopIteration
# print(next(my_iter))
```

Как прочитать файл в Python?

Чтение файла в Python можно выполнить с помощью встроенной функции `open()`. Эта функция открывает файл и возвращает файловый объект, который позволяет вам работать с содержимым файла.

```
# Открываем файл для чтения (по умолчанию режим 'r')
with open('example.txt', 'r') as file:
    content = file.read()

print(content)
```

Как работать с json в Python?

Работа с JSON (JavaScript Object Notation) в Python очень проста, так как язык имеет встроенную поддержку для этого формата данных. JSON - это текстовый формат обмена данными, который представляет объекты и массивы в виде строки. В Python для работы с JSON используется модуль `json`.

Преобразование Python объектов в JSON: Вы можете преобразовать Python объекты (списки, словари, строки и т.д.) в JSON с помощью функции `json.dumps()`

Преобразование JSON в Python объекты: Если у вас есть JSON-строка, вы можете преобразовать ее в Python объекты с помощью функции `json.loads()`

Запись JSON в файл: Вы можете записать JSON-данные в файл с помощью функции `json.dump()`

Чтение JSON из файла: Вы можете прочитать JSON-данные из файла с помощью функции `json.load()`

```python
data = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

with open('data.json', 'w') as json_file:
    json.dump(data, json_file)
```

Сколько может быть родителей и наследников у класса?

В языке программирования Python класс может иметь любое количество родительских классов (базовых классов) и любое количество наследников (подклассов). Python поддерживает множественное наследование, что означает, что класс может наследоваться от нескольких других классов одновременно. То есть пока не закончится память.

Чем файл .pyc отличается от .py?

Файлы с расширением `.pyc` и `.py` являются файлами программ на языке программирования Python, но они имеют несколько различий:

Расширение:

- `.py`: Это расширение файлов исходного кода Python. В таких файлах содержится читаемый текст кода на языке Python.
- `.pyc`: Это расширение файлов скомпилированного (байт-кода) кода Python. Файлы `.pyc` содержат скомпилированный байт-код Python, который может выполняться интерпретатором Python. Они создаются автоматически, когда Python выполняет файлы `.py`.

Содержание:

- Файлы `.py` содержат читаемый исходный код на языке Python, который можно редактировать и изменять в текстовом редакторе.
- Файлы `.pyc` содержат скомпилированный байт-код Python, который представляет собой набор инструкций, понимаемых интерпретатором Python. Это бинарный формат, не предназначенный для чтения человеком.

Использование:

- Файлы `.py` используются для написания исходного кода программ на языке Python.
- Файлы `.pyc` используются для ускорения загрузки и выполнения программ на Python. Когда файл `.py` выполняется, интерпретатор Python компилирует его в байт-код и сохраняет его в файл `.pyc` в той же директории. При последующих запусках программы интерпретатор сначала проверяет наличие файла `.pyc` и, если он существует и соответствует файлу `.py`, то использует скомпилированный байт-код, что может ускорить процесс запуска программы.

Обратите внимание, что файлы `.pyc` являются дополнительными файлами, которые создаются интерпретатором Python при выполнении программы. Вы всегда можете изменить файл `.py`, и при следующем запуске интерпретатор Python автоматически пересоздаст файл `.pyc` с обновленным байт-кодом.

Как производится debug программы на Python?

Отладка программы на Python производится с помощью различных инструментов и методов, чтобы обнаруживать и исправлять ошибки в коде. Вот некоторые способы и инструменты для проведения отладки программы на Python

Использование `print`: Один из самых простых способов отладки - использование функции `print` для вывода значений переменных, сообщений и промежуточных результатов в консоль. Это позволяет вам видеть, какие значения имеют переменные в разных точках программы и отслеживать выполнение кода.

Использование модуля `pdb`: Модуль `pdb` предоставляет интерактивный отладчик для Python. Вы можете разместить точки останова в коде, чтобы отслеживать выполнение программы, и использовать команды отладчика для исследования состояния программы и переменных.

Использование `logging`: Модуль `logging` позволяет записывать сообщения в журнал, который помогает отслеживать выполнение программы и значения переменных без необходимости вывода в консоль. Это более структурированный подход к отладке, который позволяет управлять уровнем логирования и сохранять данные в файл.

Объясните разницу между функциями str и repr

В Python функции `str()` и `repr()` используются для получения строкового представления объектов, но есть некоторые различия в том, как они возвращают значения

`str()`: Эта функция используется для получения "неформального" или "читаемого" представления объекта в виде строки. Она предназначена для использования в основном в контексте вывода для пользователей или журналирования. Результат `str()` может быть представлен без кавычек и должен быть понятным для человека.

`repr()`: Эта функция используется для получения "формального" представления объекта в виде строки. Ее цель - представить объект так, чтобы его можно было точно воссоздать с помощью Python-кода. Результат `repr()` заключается в кавычки и может содержать дополнительную информацию, такую как тип объектая.

Что такое байт код?

Байт-код (bytecode) - это промежуточное представление программы, которое является результатом компиляции исходного кода высокоуровневого языка программирования (например, Python) в более низкоуровневое, но всё ещё понятное компьютеру представление.

В контексте Python байт-код - это набор инструкций, предназначенных для виртуальной машины Python (Python Virtual Machine - PVM). При выполнении программы Python интерпретатор читает и исполняет байт-код по одной инструкции за раз. Таким образом, Python является интерпретируемым языком программирования.

Процесс работы с байт-кодом выглядит следующим образом:

Исходный код на Python (.py) компилируется в байт-код (.pyc) с помощью компилятора Python. Виртуальная машина Python (PVM) исполняет байт-код и выполняет операции, указанные в инструкциях байт-кода.

Использование байт-кода обеспечивает переносимость программы между различными платформами. Исходный код компилируется в байт-код один раз, и затем этот байт-код может быть исполнен на любой платформе, на которой установлен интерпретатор Python. Это позволяет программам на Python быть кросс-платформенными без необходимости перекомпиляции исходного кода для каждой платформы.

Байт-код в Python также обеспечивает более быструю загрузку программы, так как интерпретатор может просто выполнять предварительно скомпилированный байт-код, что ускоряет стартовое время выполнения программы.

Типизация в Python

В Python существует динамическая типизация, что означает, что переменные могут автоматически изменять свой тип данных во время выполнения программы. Python определяет тип переменной на основе значения, которое она содержит.

Например, вы можете присвоить переменной целочисленное значение, а затем переопределить ее как строку без необходимости явно указывать тип данных

```
# Динамическая типизация в Python

# Целочисленная переменная
num = 10
print(type(num))  # Выведет <class 'int'>

# Переопределение переменной в строку
num = "Hello, Python!"
print(type(num))  # Выведет <class 'str'>
```

Что такое GIL?

В Python GIL (Global Interpreter Lock) - это механизм, используемый для обеспечения потокобезопасности в интерпретаторе Python. Это особенность, присутствующая в стандартной реализации интерпретатора CPython.

GIL представляет собой мьютекс (mutex), который действует как блокировка на уровне интерпретатора. Он позволяет только одному потоку выполнять байткод (инструкции Python) в любой момент времени. Это означает, что в многопоточных программах на Python, даже если у вас есть несколько потоков, на практике они выполняют свой код последовательно, а не параллельно на нескольких ядрах процессора.

Идея GIL возникла как механизм упрощения управления памятью и избежания проблем с многопоточным доступом к общим объектам и структурам данных. Благодаря GIL, интерпретатор может избежать таких проблем, как состояния гонки (race conditions) и взаимоблокировки (deadlocks).

Однако GIL также имеет свои ограничения. Поскольку только один поток может активно выполнять Python-код в определенный момент времени, многопоточные приложения, которые испытывают интенсивную вычислительную нагрузку (CPU-bound), могут не получать существенного выигрыша в производительности от использования многопоточности. Вместо этого GIL может стать узким местом в таких приложениях.

Однако важно отметить, что GIL касается только многопоточности внутри интерпретатора Python. Если ваше приложение выполняет многопроцессорные операции (процессы, а не потоки) или использует сторонние библиотеки, которые выполняют нативный код (например, написанный на C или C++), то эти операции могут эффективно использовать многопроцессорные ресурсы вашей системы.

Также стоит отметить, что существуют альтернативные реализации Python, такие как Jython, IronPython и Pypy, которые имеют разные подходы к управлению потоками и обходу ограничений GIL.

Процессы и потоки в Python

Процессы: Процессы - это независимые исполняемые единицы, каждый из которых имеет свою собственную память и исполнение. Процессы обладают собственным пространством памяти и не могут напрямую обмениваться данными друг с другом. Каждый процесс запускается в своем собственном интерпретаторе Python. Это означает, что процессы могут использовать многопроцессорное окружение, чтобы выполнять задачи параллельно на разных ядрах процессора. Для создания процессов в Python можно использовать модуль `multiprocessing`. Он предоставляет класс `Process`, который позволяет создавать и управлять процессами.

```python
import multiprocessing

def worker_function():
    print("Worker is doing some task.")

if __name__ == "__main__":
    process = multiprocessing.Process(target=worker_function)
    process.start()
    process.join()
    print("Main process continues...")
```

Потоки (threads) - это более легковесные исполняемые единицы, чем процессы, и они существуют внутри процесса. Все потоки в пределах одного процесса используют общую память. Потоки обычно используются для задач с высокой степенью ввода-вывода (I/O-bound), таких как чтение и запись файлов или сетевая коммуникация, когда задачи блокируются на ввод-выводе и основной процесс может продолжать работу. Для работы с потоками в Python есть встроенный модуль `threading`. Он предоставляет класс `Thread`, который позволяет создавать и управлять потоками.

```python
import threading

def worker_function():
    print("Worker is doing some task.")

if __name__ == "__main__":
    thread = threading.Thread(target=worker_function)
    thread.start()
    thread.join()
    print("Main thread continues...")
```

Что такое PIP, и за что он отвечает?

PIP (Python Package Index) - это система управления пакетами для языка программирования Python. Он предоставляет командную строку интерфейса, который позволяет устанавливать, обновлять и удалять сторонние пакеты Python из огромного репозитория пакетов, известного как Python Package Index.

Установка пакетов: PIP позволяет устанавливать сторонние пакеты Python в вашем локальном окружении. Когда вы хотите использовать сторонний модуль или библиотеку, вы можете просто выполнить команду `pip install <имя_пакета>` в командной строке, и PIP загрузит и установит пакет из Python Package Index или из другого указанного источника.

Обновление пакетов: PIP также позволяет обновлять установленные пакеты до последних версий. Выполнив команду `pip install --upgrade <имя_пакета>` , PIP проверит Python Package Index на наличие более новой версии указанного пакета и, если найдет, обновит его.

Удаление пакетов: Если вы больше не нуждаетесь в определенном пакете, PIP позволяет легко удалить его из вашего окружения. Для этого достаточно выполнить команду `pip uninstall <имя_пакета>` .

Поиск пакетов: PIP предоставляет возможность искать пакеты в Python Package Index по ключевым словам или именам, чтобы найти нужные библиотеки или инструменты для вашего проекта.

Каковы основные преимущества Python перед другими языками программирования?

- Простота и читаемость кода: Python имеет простой и чистый синтаксис, который делает код более читаемым и понятным. Он призван быть единообразным и минималистичным, что помогает разработчикам легко писать и поддерживать код.
- Богатая стандартная библиотека: Python поставляется с обширной стандартной библиотекой, которая включает множество модулей и инструментов для различных задач, таких как работа с файлами, сетью, базами данных, регулярными выражениями, обработкой данных и многое другое. Это делает Python очень мощным и удобным для разработки приложений различных типов.
- Кросс-платформенность: Python доступен для различных операционных систем, таких как Windows, macOS, Linux и другие, что позволяет выполнять программы на разных платформах без изменений в исходном коде.
- Обширное сообщество и поддержка: Python имеет огромное сообщество разработчиков, которые активно сотрудничают, делятся опытом и создают библиотеки и фреймворки для разных областей. Это обеспечивает хорошую поддержку и быстрое решение проблем, а также обновления и улучшения.
- Высокая производительность: Python обладает отличной производительностью для задач, связанных с обработкой текстов, сетевой коммуникацией, веб-разработкой, а также задачами с большим объемом данных. Благодаря оптимизациям и JIT-компиляции, Python становится все более эффективным и конкурентоспособным.
- Расширяемость: Python легко интегрируется с другими языками программирования, такими как C, C++, Java, что позволяет использовать библиотеки и функциональность из других языков и расширять возможности Python.
- Многоцелевой язык: Python подходит для разработки различных типов приложений, включая веб-приложения, научные вычисления, искусственный интеллект, анализ данных, автоматизацию задач, сценарии, игры и многое другое.
- Простая интеграция с другими технологиями: Python хорошо интегрируется с различными базами данных, веб-серверами, API и другими технологиями, что облегчает разработку сложных и комплексных приложений.
- Широкое применение: Python активно используется в таких областях, как веб-разработка (Django, Flask), научные и инженерные расчеты (NumPy, SciPy), анализ данных (Pandas), искусственный интеллект и машинное обучение (TensorFlow, PyTorch), автоматизация и системное администрирование, разработка игр и т.д.

Как передаются аргументы в Python?

Аргументы передаются по ссылке, но также нужно учитывать, что это может проявляться по-разному в зависимости от типа данных аргумента. Это важное понятие, которое может повлиять на то, какие изменения будут внесены в переменные при их передаче в функции.

Когда вы передаете аргумент в функцию в Python, сам аргумент не копируется. Вместо этого передается ссылка на объект в памяти, где хранится значение этого аргумента. Это означает, что функция может изменить содержимое этого объекта, и эти изменения будут отражены в оригинальной переменной, которую вы передали в функцию.

Однако, если аргумент является неизменяемым типом данных, таким как целые числа, строки или кортежи, то функция создаст локальную копию этого значения, и изменения, внесенные внутри функции, не повлияют на оригинальную переменную.

Вопросы Junior

Databases

Что такое база данных?

База данных (БД) - это структурированное собрание данных, организованных и хранимых в компьютерной системе. Она представляет собой совокупность информации, которая позволяет хранить, управлять, обновлять и извлекать данные для различных приложений и задач.

Основные характеристики базы данных включают:

- Структурированность: Данные организованы в определенной структуре, что облегчает доступ и обработку.
- Централизованность: Базы данных обычно хранятся на центральном сервере, к которому имеют доступ пользователи и приложения.
- Многопользовательский доступ: Базы данных поддерживают одновременный доступ нескольких пользователей, что позволяет им одновременно работать с данными.
- Долгосрочное хранение: Данные в базе данных сохраняются долгое время и не теряются после выключения системы или завершения программы.
- Эффективные методы обработки: Базы данных обеспечивают эффективные методы поиска, сортировки и фильтрации данных, что делает их полезными для различных операций и запросов.
- Безопасность: Базы данных могут предоставлять различные уровни доступа и права для защиты данных от несанкционированного доступа.

Базы данных используются в различных областях, включая бизнес, науку, образование, здравоохранение и другие. Примеры систем управления базами данных (СУБД) включают MySQL, PostgreSQL, Oracle, Microsoft SQL Server, MongoDB и многие другие. Каждая СУБД имеет свои особенности и подходы к управлению данными, в зависимости от требований и задач пользователей.

Какие базы данных вы знаете?

Реляционные базы данных (SQL):

- MySQL
- PostgreSQL
- Oracle Database
- Microsoft SQL Server
- SQLite
- IBM Db2
- MariaDB

Нереляционные базы данных (NoSQL):

- MongoDB
- Cassandra
- Redis
- Couchbase
- Neo4j
- Amazon DynamoDB
- Google Cloud Firestore
- Apache HBase

Что такое СУБД?

СУБД - это аббревиатура, которая расшифровывается как "Система Управления Базами Данных". Она представляет собой программное обеспечение, которое обеспечивает управление и работу с базами данных. СУБД предоставляет интерфейс между приложениями и физическим хранением данных, позволяя пользователям удобно хранить, извлекать, изменять и управлять данными.

- Создание баз данных: СУБД предоставляют средства для создания и определения структуры баз данных, включая таблицы, поля, ограничения и связи.
- Хранение данных: СУБД управляют физическим хранением данных на диске или в памяти компьютера, обеспечивая эффективное использование ресурсов.
- Извлечение данных: Пользователи и приложения могут выполнять запросы к базам данных для получения нужных данных с помощью языка запросов, такого как SQL (Structured Query Language).
- Обновление данных: СУБД позволяют вставлять, изменять и удалять данные в базе данных, поддерживая целостность и безопасность данных.
- Управление транзакциями: СУБД обеспечивают контроль за транзакциями, что позволяет гарантировать атомарность, согласованность, изолированность и устойчивость данных (свойства, обозначаемые как ACID).
- Безопасность: СУБД предоставляют средства для установки прав доступа и защиты данных от несанкционированного доступа.
- Резервное копирование и восстановление: СУБД обеспечивают механизмы резервного копирования данных и восстановления после сбоев или ошибок.

В чем разница между реляционной базой данных и нереляционной?

- Реляционные базы данных: Основаны на реляционной модели данных, где данные представлены в виде таблиц с рядами и столбцами. Каждая таблица имеет имя и состоит из набора атрибутов (столбцов), которые определяют типы данных, а каждая строка таблицы представляет отдельную запись данных.
- Нереляционные базы данных: Не следуют реляционной модели данных и могут использовать различные модели данных, такие как документы, столбцы, ключ-значение или графы. Вместо таблиц, данные хранятся в более свободной форме, позволяющей гибко представлять сложные структуры и связи.
- Реляционные базы данных: Имеют строгую предопределенную схему данных, которая определяет структуру таблиц и типы данных для каждого столбца. Изменение схемы данных может быть сложным и требует перестройки таблиц и переноса данных.
- Нереляционные базы данных: Обычно имеют динамическую схему, что означает, что каждая запись (документ, элемент) может иметь свою собственную структуру данных. Это делает их более гибкими для изменения структуры данных без необходимости изменения всей базы данных.
- Реляционные базы данных: Используют язык структурированных запросов (SQL) для выполнения операций с данными, таких как SELECT, INSERT, UPDATE и DELETE.
- Нереляционные базы данных: Каждый тип NoSQL может использовать собственный язык запросов или API для работы с данными. Например, MongoDB использует BSON-подобные запросы, а Redis предоставляет свой собственный набор команд.
- Реляционные базы данных: Обычно масштабируются вертикально, добавляя более мощное оборудование. Это может быть ограничивающим фактором в случае высоких нагрузок и больших объемов данных.
- Нереляционные базы данных: Часто могут масштабироваться горизонтально, добавляя новые узлы или серверы, что делает их более масштабируемыми и способными обрабатывать большие объемы данных и высокие нагрузки.

Что такое ACID?

ACID - это акроним, который представляет собой совокупность свойств, обозначающих надежность транзакций в системах управления базами данных (СУБД). Каждая буква в слове "ACID" представляет одно из основных свойств транзакций, которые гарантируют корректное выполнение операций с данными.

- Атомарность (Atomicity): Атомарность означает, что транзакция считается единой и неделимой операцией. Либо все ее части успешно выполняются, либо ни одна из них не выполняется. Если даже одна операция внутри транзакции не может быть завершена успешно, все изменения, внесенные другими операциями, отменяются, и состояние базы данных возвращается к исходному.
- Согласованность (Consistency): Согласованность означает, что транзакция приводит базу данных из одного согласованного состояния в другое согласованное состояние. То есть все правила, ограничения и связи данных, заданные в схеме базы данных, должны быть соблюдены после успешного завершения транзакции.
- Изолированность (Isolation): Изолированность обеспечивает, что выполнение одной транзакции не будет влиять на выполнение других транзакций. Каждая транзакция должна быть выполнена как будто она выполняется в изоляции от других транзакций, чтобы предотвратить нежелательные взаимодействия и обеспечить предсказуемость работы с данными.
- Устойчивость (Durability): Устойчивость гарантирует, что после успешного завершения транзакции изменения, внесенные в базу данных, остаются постоянными и не будут потеряны при возникновении сбоев или ситуаций, прерывающих работу системы. Это обеспечивается сохранением изменений на долгосрочное хранение, например, на диске или других устойчивых носителях данных.

В чем разница между PostgreSQL и MongoDB?

Модель данных:

- PostgreSQL: Это реляционная база данных, основанная на реляционной модели данных. Данные организованы в виде таблиц с рядами и столбцами. Таблицы могут быть связаны друг с другом через внешние ключи, обеспечивая строгую согласованность данных и поддержку сложных запросов с использованием SQL.
- MongoDB: Это нереляционная база данных, использующая модель данных "документов". Данные хранятся в виде документов, которые могут быть вложены друг в друга. MongoDB не требует строгой схемы данных, что позволяет гибко представлять сложные структуры и связи между данными.

Язык запросов:

- PostgreSQL: Использует SQL (Structured Query Language) для выполнения операций с данными. SQL предоставляет мощные возможности для работы с реляционными данными, такими как объединение таблиц, группировка, сортировка и подзапросы.
- MongoDB: Использует MongoDB Query Language, который предоставляет более гибкие запросы, ориентированные на структуру документов. Он поддерживает поиск по значениям полей, вложенные запросы и операции агрегации.

Схема данных:

- PostgreSQL: Имеет строгую предопределенную схему данных, которая определяет структуру таблиц и типы данных для каждого столбца. Изменение схемы данных может быть сложным и требует перестройки таблиц и переноса данных.
- MongoDB: Имеет динамическую схему, что означает, что каждый документ может иметь свою собственную структуру данных без строгих ограничений. Это обеспечивает гибкость при работе с данными различных форматов.

Масштабируемость:

- PostgreSQL: Обычно масштабируется вертикально, добавляя более мощное оборудование. Масштабируемость ограничивается ресурсами одного сервера.
- MongoDB: Часто масштабируется горизонтально, добавляя новые узлы или серверы. MongoDB разработана для обработки больших объемов данных и высоких нагрузок.

Уровни изоляции транзакций

Уровни изоляции транзакций определяют, насколько одна транзакция изолирована от других транзакций, работающих с данными в той же базе данных. Они контролируют видимость изменений, внесенных одной транзакцией, для других транзакций. В стандарте SQL определено четыре уровня изоляции:

READ UNCOMMITTED (Чтение незафиксированных данных):

- Этот уровень позволяет транзакциям читать данные, которые были изменены другими транзакциями, но еще не были зафиксированы (подтверждены). Таким образом, это самый низкий уровень изоляции и предоставляет минимальную защиту от конфликтов параллельных транзакций.

READ COMMITTED (Чтение зафиксированных данных):

- Этот уровень позволяет транзакциям читать только те данные, которые были зафиксированы (подтверждены) другими транзакциями. Это предотвращает чтение "грязных" данных, но может привести к так называемому "фантомному чтению", когда одна транзакция видит изменения, внесенные другой транзакцией, после того как она начала свою работу.

REPEATABLE READ (Повторяемое чтение):

- На этом уровне транзакция видит только те данные, которые были зафиксированы до начала самой транзакции. Это предотвращает как "грязное чтение", так и "фантомное чтение", но может привести к проблеме "двойного чтения", когда одна и та же запись читается дважды и результаты различны.

SERIALIZABLE (Сериализуемость):

- Этот уровень обеспечивает максимальную изоляцию. Он гарантирует, что транзакции выполняются так, как будто они выполняются последовательно, без параллельных операций. Это предотвращает все типы аномалий чтения и обеспечивает наивысшую степень надежности, но также может привести к замедлению системы из-за блокировок.

Что такое индексы?

Индексы (индексирование) - это структуры данных в базах данных, созданные для ускорения операций поиска и извлечения данных. Они представляют собой дополнительные структуры, которые содержат информацию о значениях определенных столбцов таблицы и их расположении в базе данных.

Индексы позволяют системе управления базами данных (СУБД) быстрее находить и обрабатывать данные, поскольку они создают отдельный путь к данным, оптимизированный для выполнения запросов. Вместо просмотра всех строк таблицы, СУБД может использовать индекс для поиска конкретных значений и точного расположения соответствующих данных.

Преимущества использования индексов:

- Ускорение поиска данных: Индексы позволяют более эффективно выполнять операции поиска и фильтрации данных.
- Улучшение производительности: Использование индексов может существенно ускорить выполнение запросов, особенно в больших таблицах с большим объемом данных.
- Оптимизация сортировки: Индексы также помогают ускорить сортировку данных, что полезно при выполнении запросов с оператором ORDER BY.
- Улучшение уникальности: Индексы могут использоваться для обеспечения уникальности значений в столбце, что предотвращает дубликаты данных.

Однако следует учитывать, что использование индексов также имеет свои недостатки:

- Затраты на хранение: Индексы занимают дополнительное место на диске или в памяти, что может увеличить объем занимаемой базой данных.
- Обслуживание: Индексы требуют обновления при вставке, обновлении или удалении данных, что может повлиять на производительность при выполнении большого количества таких операций.
- Следование балансу: Не следует создавать слишком много индексов, так как это может привести к ухудшению производительности при выполнении операций изменения данных.

Что такое ForeignKey?

Внешний ключ (Foreign Key) - это столбец или набор столбцов в таблице, который устанавливает связь между данными в этой таблице и данными в другой таблице. Он обеспечивает целостность данных и позволяет связывать записи из одной таблицы с записями из другой таблицы.

- Сохранение ссылочной целостности: Внешние ключи помогают обеспечить ссылочную целостность данных, что означает, что ссылки на родительские записи всегда остаются корректными. Если родительская запись удаляется или изменяется, соответствующие дочерние записи также могут быть автоматически удалены или обновлены.
- Ограничения на вставку и обновление: Внешние ключи могут устанавливать ограничения на вставку или обновление данных в дочерней таблице, чтобы убедиться, что значения внешних ключей существуют в родительской таблице.
- Улучшение целостности данных: Использование внешних ключей помогает предотвратить появление некорректных данных в дочерних таблицах, так как они связаны только с допустимыми значениями из родительской таблицы.

Что такое JOIN?

В SQL, оператор JOIN используется для объединения данных из двух или более таблиц на основе определенных условий. Это позволяет комбинировать информацию из различных таблиц в один результат. Всего существует несколько типов JOIN-операторов

```sql
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

Вопросы Junior

Django теория

Что такое Django?

Django - это бесплатный и открытый веб-фреймворк на языке Python, который предоставляет удобные средства для разработки веб-приложений. Он был создан с целью упростить и ускорить процесс разработки, обеспечивая готовые решения для общих задач и способствуя удобству и эффективности разработчиков

Что такое MVC?

MVC (Model-View-Controller): MVC - это архитектурный паттерн, который разделяет веб-приложение на три основных компонента:

- Модель (Model): Этот компонент отвечает за обработку данных и бизнес-логику приложения. Он представляет собой представление данных, с которыми работает приложение, и обеспечивает их доступ и манипуляции.
- Представление (View): Этот компонент отвечает за отображение данных пользователю и интерактивность интерфейса. Он получает информацию из модели и готовит ее для отображения пользователю.
- Контроллер (Controller): Этот компонент управляет взаимодействием между моделью и представлением. Он обрабатывает входящие запросы от пользователя, вызывает соответствующие методы модели и подготавливает данные для отображения в представлении.

Что такое MVT?

MVT (Model-View-Template): MVT - это вариация паттерна MVC, которая часто используется в веб-фреймворке Django. В MVT компоненты имеют схожие функции с компонентами в MVC:

- Модель (Model): Этот компонент аналогичен компоненту модели в MVC. Он обрабатывает данные и бизнес-логику приложения.
- Представление (View): В MVT представление отвечает за обработку входящих запросов и взаимодействие с моделью для получения необходимых данных. Однако, в отличие от классического MVC, представление в MVT не обязательно занимается отображением данных. Вместо этого оно возвращает данные в виде HTTP-ответа или передает их в шаблон для дальнейшего отображения.
- Шаблон (Template): Это компонент, который отвечает за отображение данных пользователю. Он содержит HTML-разметку с динамическими переменными, которые заполняются данными, полученными из представления.

Таким образом, в MVT шаблоны заменяют роль представлений в классическом MVC. Фактически, взаимодействие с пользователем в MVT происходит в два этапа: сначала представление обрабатывает запросы и формирует данные, а затем шаблон отображает эти данные в пользовательском интерфейсе.

Что такое Django сигналы?

Django сигналы (Django signals) - это механизм веб-фреймворка Django, который позволяет приложениям реагировать на определенные события, происходящие внутри самого фреймворка или в базе данных. Сигналы предоставляют способ связи различных компонентов Django, не являясь прямым вызовом функций или методов.

Когда происходит определенное событие, Django отправляет сигнал, который может быть перехвачен другими частями приложения или сторонними пакетами для выполнения дополнительных действий. Это позволяет разделить логику приложения на более отдельные и независимые компоненты, что способствует улучшению модульности и повторному использованию кода.

- Сохранение объекта модели в базе данных.
- Удаление объекта модели из базы данных.
- Создание нового пользователя (получение сигнала от системы аутентификации Django).
- Отправка электронной почты (получение сигнала от модуля отправки электронной почты Django).

```python
from django.db.models.signals import post_save
from django.dispatch import receiver
from myapp.models import MyModel

@receiver(post_save, sender=MyModel)
def my_model_post_save_handler(sender, instance, **kwargs):
    # Ваш код для обработки события сохранения объекта MyModel
    pass
```

Стратегии кэширования в Django?

Кэширование относится к методике сохранения выходных результатов при их первоначальной обработке, чтобы в следующий раз, когда те же результаты будут получены снова, вместо повторной обработки можно было бы использовать уже сохраненные результаты, что приводит к более быстрому доступу, а также к меньшему использованию ресурсов. Django предоставляет нам надежную систему кеширования, которая может хранить динамические веб-страницы, так что эти страницы не нужно повторно оценивать для каждого запроса.

- **Memcached/Redis** - Кэш-сервер на основе памяти - самый быстрый и эффективный
- **FileSystem Caching** - Значения кеша хранятся в виде отдельных файлов в сериализованном порядке.
- **Local-memory Caching** - Это используется Django в качестве стратегии кэширования по умолчанию, если вы ничего не установили.
- **Database Caching** - Данные кеша будут храниться в базе данных и работать очень хорошо, если у вас есть быстрый и хорошо индексируемый сервер БД.

Разница между OneToOneField, ForeignKey, ManyToManyField?

- `OneToOneField` определяет однозначное отношение "один к одному" между двумя моделями. Это означает, что каждая запись в одной модели может быть связана только с одной записью в другой модели, и наоборот. В базе данных это реализуется путем добавления внешнего ключа к связываемой модели.
- `ForeignKey` определяет отношение "один ко многим" между двумя моделями. Это означает, что каждая запись в одной модели может быть связана с несколькими записями в другой модели, но каждая запись в другой модели может быть связана только с одной записью в первой модели. В базе данных это реализуется с использованием внешнего ключа.
- `ManyToManyField` определяет отношение "многие ко многим" между двумя моделями. Это означает, что каждая запись в одной модели может быть связана с несколькими записями в другой модели, и наоборот. В базе данных это реализуется с помощью дополнительной промежуточной таблицы, которая содержит пары связанных записей.

Разница между select_related и prefetch_related?

`select_related` используется для загрузки связанных объектов, которые установлены как `ForeignKey` или `OneToOneField` в текущем запросе. Он делает "жадный" запрос, который выполняет JOIN операцию в базе данных, чтобы получить связанные объекты за один запрос. Это особенно полезно, когда у вас есть много объектов и вы хотите избежать отдельных запросов для каждого связанного объекта.

```python
# Модели
class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    title = models.CharField(max_length=100)

# Запрос с использованием select_related
books = Book.objects.select_related('author').all()
```

`prefetch_related` используется для загрузки связанных объектов, которые установлены как `ManyToManyField` или обратные связи `ForeignKey` (например, связи, определенные через `related_name`). Он делает "жадный" запрос, чтобы получить все связанные объекты, и затем связывает их с основными объектами в памяти Python. Это особенно полезно, когда у вас есть много объектов и вы хотите избежать множества дополнительных запросов для каждого связанного объекта.

```python
class Category(models.Model):
    name = models.CharField(max_length=100)

class Product(models.Model):
    categories = models.ManyToManyField(Category)
    name = models.CharField(max_length=100)

# Запрос с использованием prefetch_related
products = Product.objects.prefetch_related('categories').all()
```

Как реализовано ManyToMany под капотом?

В Django, связи типа "многие ко многим" (ManyToMany) реализуются с использованием таблицы связи (intermediate table) между двумя связанными моделями. Эта промежуточная таблица хранит связи между объектами обеих моделей и позволяет устанавливать отношения "многие ко многим" без необходимости создания дополнительных полей в самих моделях.

Важно заметить, что промежуточная таблица генерируется автоматически и невидима для разработчика. Django обрабатывает создание и управление промежуточной таблицей в фоновом режиме.

Для определения отношения "многие ко многим" между двумя моделями в Django, необходимо использовать поле `ManyToManyField` в одной из моделей и указать другую модель в качестве аргумента этого поля.

Что такое Q объект?

В Django, `Q` - это объект, используемый для создания сложных запросов к базе данных с использованием операторов `OR` и `AND`. Он представляет собой "поисковое выражение" (query expression), которое позволяет объединять несколько условий в одном запросе.

Обычно в Django для выполнения запросов к базе данных используется менеджер моделей (`objects`) и его методы, такие как `filter()` и `exclude()`, которые позволяют выбирать объекты из базы данных, основываясь на заданных условиях.

Однако в некоторых случаях может потребоваться создать запрос с использованием операторов `OR` и `AND` для объединения нескольких условий. Это может быть полезно, например, когда нужно найти объекты, которые удовлетворяют одному условию или другому. Вот где приходит на помощь `Q`.

```python
from django.db.models import Q
from myapp.models import MyModel

# Создаем два объекта Q для двух различных условий
q1 = Q(name__icontains='apple')
q2 = Q(category='fruit')

# Получаем объекты, удовлетворяющие одному из условий (оператор OR)
result1 = MyModel.objects.filter(q1 | q2)

# Получаем объекты, удовлетворяющие обоим условиям (оператор AND)
result2 = MyModel.objects.filter(q1 & q2)
```

Что такое middleware в Django?

Middleware в Django - это механизм, который позволяет обрабатывать запросы и ответы перед тем, как они попадают во view (представление) или после того, как представление вернуло ответ. Middleware представляет собой слой программного обеспечения, который обеспечивает перехват и обработку запросов и ответов на уровне HTTP-запроса.

Когда клиент делает HTTP-запрос к вашему Django-приложению, запрос проходит через цепочку middleware перед тем, как попасть во view, которое обрабатывает этот запрос. После обработки представлением, ответ также проходит через цепочку middleware перед тем, как быть отправлен обратно клиенту.

Middleware может выполнять различные задачи, такие как:

- Аутентификация: Проверка и аутентификация пользователя перед обработкой запроса.
- Авторизация: Проверка прав доступа пользователя к определенным ресурсам или действиям.
- Логирование: Запись информации о запросах и ответах для отладки и мониторинга.
- Обработка исключений: Перехват и обработка исключений, возникающих при обработке запросов.
- Кэширование: Проверка наличия закэшированной версии ответа и, при необходимости, возврат кэшированного ответа.

```python
class SimpleLoggingMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Код, выполняющийся перед обработкой view (представления)
        response = self.get_response(request)
        # Код, выполняющийся после обработки view (представления)
        return response
```

```python
MIDDLEWARE = [
    # Другие middleware...
    'myapp.middleware.SimpleLoggingMiddleware',
]
```

Опишите алгоритм работы CSRF middleware

В Django, CSRF (Cross-Site Request Forgery) middleware - это промежуточное программное обеспечение (middleware), которое помогает защитить приложение от атак CSRF. CSRF - это тип атаки, при которой злоумышленник пытается выполнить нежелательные действия от имени авторизованного пользователя. Middleware в Django предоставляет механизм для проверки и предотвращения таких атак.

Ниже представлен общий алгоритм работы CSRF middleware в Django:

- Генерация токена: При первом запросе на страницу, требующую защиту от CSRF (обычно формы), Django генерирует уникальный CSRF-токен. Этот токен будет использоваться для проверки подлинности запроса в последующих запросах.
- Включение токена в формы: Django автоматически вставляет сгенерированный CSRF-токен в скрытое поле формы или в заголовок запроса (в зависимости от типа запроса). Это делается с помощью тега `{% csrf_token %}` в шаблонах Django или функции `csrfmiddlewaretoken` в HTML-формах.
- Проверка токена: При получении запроса, содержащего данные от пользователя (например, отправленные формы), CSRF middleware автоматически проверяет наличие правильного CSRF-токена. Если токен не найден или не соответствует ожидаемому значению, запрос считается недействительным, и Django возбудит исключение `Forbidden` (ошибка 403).
- Обработка недействительного запроса: При возникновении ошибки CSRF (403 Forbidden), Django может выполнить различные действия в зависимости от конфигурации приложения. Обычно это перенаправление пользователя на страницу с ошибкой или отправка пользователю сообщения об ошибке.
- Опциональная настройка исключений: В некоторых случаях необходимо разрешить определенным запросам проходить без проверки CSRF-токена. Django предоставляет возможность настройки исключений для таких запросов, которые можно указать в настройках приложения.
- Ограничение области действия токена: CSRF-токен обычно связывается с сеансом пользователя или конкретной формой. Это означает, что токен имеет ограниченную область действия и не может быть использован для других действий или сессий.

Миграции в Django

Миграции (migrations) в Django - это механизм, который позволяет автоматически создавать и обновлять структуру базы данных в соответствии с изменениями в моделях Django. Он предоставляет инструмент для управления эволюцией схемы базы данных без необходимости вручную создавать и изменять таблицы и поля.

Когда вы создаете модели в Django (классы Python, которые определяют структуру данных в приложении), миграции помогают перенести эти модели в таблицы базы данных и обеспечить согласованность между вашим кодом и структурой базы данных.

Зачем нужны миграции:

- Автоматическое создание таблиц: Django автоматически создает таблицы в базе данных на основе определенных вами моделей, что упрощает и ускоряет процесс создания базы данных.
- Изменение схемы базы данных: При изменении структуры моделей, например, добавлении нового поля или изменении типа данных, миграции автоматически создают соответствующие изменения в схеме базы данных.
- Перенос между средами: Миграции позволяют переносить схему базы данных между различными средами (например, разработка, тестирование, производство) без необходимости вручную создавать и обновлять базу данных на каждом этапе.
- Обратимость изменений: Django сохраняет информацию о предыдущих миграциях, что позволяет откатывать изменения схемы базы данных на предыдущие состояния.
- Синхронизация моделей и базы данных: Миграции помогают обеспечить соответствие между структурой базы данных и определенными моделями Django, что уменьшает вероятность возникновения ошибок при работе с данными.

Для работы с миграциями, Django использует команду `makemigrations` для создания файлов миграций на основе изменений в моделях и команду `migrate` для применения миграций к базе данных.

Как выглядит структура в Django

- `project_name/` : Это корневая директория вашего проекта Django. Вы можете выбрать имя проекта при его создании.
- `manage.py`: Это скрипт командной строки, который используется для управления проектом Django. С его помощью вы можете запускать сервер разработки, создавать миграции базы данных, создавать суперпользователя и многое другое.
- `project_name/` : Это вторая директория с тем же именем, что и корневая директория. В ней содержатся основные файлы настройки проекта.
- `settings.py`: Этот файл содержит настройки проекта Django, такие как базы данных, статические файлы, шаблоны, middleware и другие параметры.
- `urls.py`: Этот файл определяет основные маршруты URL вашего проекта и связывает их с представлениями (views).
- `asgi.py`: Этот файл содержит точку входа для ASGI-совместимых серверов, которые поддерживают асинхронную обработку запросов.
- `wsgi.py`: Этот файл содержит точку входа для WSGI-совместимых серверов, которые обслуживают ваше приложение в рабочей среде.
- `app1/`, `app2/`, и т.д.: Это директории для ваших приложений Django. Каждое приложение Django должно находиться в отдельной директории. При создании приложения, оно автоматически создастся в этой директории.
- `migrations/` : Это директория, в которой хранятся файлы миграций для базы данных вашего приложения. Миграции используются для автоматического обновления схемы базы данных при изменении моделей в приложении.
- `templates/` : Эта директория содержит шаблоны (HTML-файлы) вашего приложения, которые используются для отображения пользовательского интерфейса.
- `static/` : В этой директории находятся статические файлы, такие как CSS, JavaScript, изображения и другие ресурсы, используемые в вашем приложении.
- `__init__.py` : Это пустой файл, который указывает, что директория является пакетом Python.
- `admin.py`: В этом файле определяются настройки административного интерфейса Django для моделей приложения.
- `models.py`: В этом файле определяются модели данных вашего приложения, которые представляют структуру таблиц в базе данных.
- `views.py`: В этом файле определяются представления, которые обрабатывают запросы и возвращают HTTP-ответы.
- `apps.py`: Этот файл содержит конфигурацию приложения, такую как его имя и другие параметры.

```
project_name/
├── manage.py
├── project_name/
│   ├── settings.py
│   ├── urls.py
│   ├── asgi.py
│   └── wsgi.py
├── app1/
│   ├── migrations/
│   ├── templates/
│   ├── static/
│   ├── __init__.py
│   ├── admin.py
│   ├── models.py
│   ├── views.py
│   └── apps.py
├── app2/
│   ├── migrations/
│   ├── templates/
│   ├── static/
│   ├── __init__.py
│   ├── admin.py
│   ├── models.py
│   ├── views.py
│   └── apps.py
└── ...
```

Преимущества Django

Использование Django предоставляет ряд значительных преимуществ, делающих его популярным фреймворком для разработки веб-приложений:

- Быстрая разработка: Django предоставляет множество инструментов и встроенных функций, что позволяет ускорить процесс разработки. Готовые компоненты, такие как аутентификация, административный интерфейс, URL-маршрутизация и миграции, сокращают время разработки и упрощают создание полноценных приложений.
- Мощная административная панель: Django предоставляет автоматически сгенерированный административный интерфейс, который позволяет управлять данными и администрировать приложение без необходимости создавать пользовательский интерфейс.
- ORM (Object-Relational Mapping): Django предоставляет ORM, который позволяет работать с базами данных через объектно-ориентированный подход. Это упрощает взаимодействие с базой данных и устраняет необходимость писать SQL-запросы вручную.
- Расширяемость: Django имеет большое количество сторонних пакетов и библиотек, которые позволяют расширить его функциональность. Кроме того, фреймворк легко расширяется собственными приложениями и компонентами.
- Безопасность: Django предоставляет встроенные механизмы для обеспечения безопасности, такие как защита от атак CSRF (Cross-Site Request Forgery) и XSS (Cross-Site Scripting). Это уменьшает вероятность возникновения уязвимостей в вашем приложении.
- Автоматические миграции: Механизм миграций в Django позволяет автоматически создавать и обновлять структуру базы данных в соответствии с изменениями в моделях. Это значительно упрощает управление базой данных и обновление схемы.
- Отличное сообщество: Django имеет активное и дружелюбное сообщество разработчиков, которые активно обсуждают, помогают решать проблемы и вносят вклад в развитие фреймворка. Это обеспечивает поддержку и актуальность Django.
- Хорошая документация: Django обладает обширной и хорошо организованной документацией, что облегчает изучение и использование фреймворка даже начинающим разработчикам.
- Поддержка для асинхронности: С версии Django 3.1 добавлена поддержка асинхронных представлений, что позволяет создавать асинхронные приложения и обрабатывать большое количество запросов более эффективно.

Какие команды вы знаете?

- Запуск сервера разработки: `python manage.py runserver`
- Создание нового Django проекта: `django-admin startproject project_name`
- Создание нового Django приложения: `python manage.py startapp app_name`
- Создание миграций базы данных на основе изменений в моделях: `python manage.py makemigrations`
- Применение миграций и обновление базы данных: `python manage.py migrate`
- Создание суперпользователя для доступа к административной панели: `python manage.py createsuperuser`
- Загрузка данных из файла фикстур в базу данных: `python manage.py loaddata fixture_file.json`
- Создание файла фикстур на основе данных из базы данных: `python manage.py dumpdata app_name > fixture_file.json`
- Запуск тестов: `python manage.py test`
- Создание переводных файлов для локализации: `python manage.py makemessages -l lang_code`
- Компиляция переводных файлов: `python manage.py compilemessages`
- Проверка синтаксиса Python кода в проекте: `python manage.py check`
- Очистка устаревших миграций: `python manage.py squashmigrations app_name migration_name`
- Создание административной панели для указанных моделей: `python manage.py createsuperuser`

Аутентификация пользователей в Django?

Аутентификация пользователей в Django - это процесс проверки личности пользователя и предоставления доступа к определенным частям веб-приложения на основе его учетных данных. Django предоставляет надежную и встроенную систему аутентификации, которая упрощает реализацию аутентификации пользователей в веб-приложениях.

Вот как работает аутентификация пользователей в Django:

- Модель пользователя: Django имеет встроенную модель `User`, которая представляет учетные записи пользователей. Эта модель содержит поля для общих атрибутов пользователей, таких как имя пользователя (username), электронная почта (email) и пароль (зашифрованный для безопасности). Вы можете использовать эту модель напрямую или расширить ее, чтобы добавить дополнительные пользовательские поля.
- Регистрация и вход: Пользователи могут зарегистрироваться, предоставив необходимую информацию, такую как имя пользователя, электронную почту и пароль. Для входа они вводят свои учетные данные (имя пользователя/электронную почту и пароль) на странице входа. Django обрабатывает процесс аутентификации за кулисами.
- Аутентификационные бэкенды: Django поддерживает несколько аутентификационных бэкендов, которые отвечают за проверку учетных данных пользователя и проверку его активности или неактивности. Встроенный аутентификационный бэкенд в Django использует поля имя пользователя и пароль, но вы можете настроить кастомный аутентификационный бэкенд для использования электронной почты или других методов аутентификации.
- Сессии и куки: Django использует сессии и куки для отслеживания аутентифицированных пользователей между запросами. После успешной аутентификации Django создает уникальную сессию для пользователя и хранит его идентификатор в куках браузера.
- Декораторы и middleware: Django предоставляет декораторы и middleware для защиты определенных представлений (views) или URL-маршрутов от неавторизованного доступа. Это позволяет вам контролировать доступ к различным частям вашего приложения на основе аутентификации пользователя.

Что такое FBV (Function-Based Views) и CBV (Class-Based Views)?

FBV (Function-Based Views) - Представления на основе функций. В FBV используются обычные функции Python для определения представлений. Когда Django получает запрос, он вызывает соответствующую функцию представления, которая обрабатывает запрос и возвращает HTTP-ответ.

```python
from django.http import HttpResponse

def hello(request):
    return HttpResponse("Hello, World!")
```

CBV (Class-Based Views) - Представления на основе классов. В CBV используются классы Python для определения представлений. Каждый класс-представление определяет методы, которые соответствуют различным HTTP-методам (GET, POST, etc.). Django вызывает соответствующие методы для обработки запросов.

```python
from django.http import HttpResponse
from django.views import View

class HelloView(View):
    def get(self, request):
        return HttpResponse("Hello, World!")
```

Как работает система маршрутизации URL в Django, и какая разница между `path` и `re_path`?

Система маршрутизации URL (URL routing) в Django определяет, как приложение обрабатывает входящие URL-запросы от пользователей и каким образом они соответствуют определенным представлениям (views). В Django URL-маршрутизация основана на использовании модуля `urls`, который связывает URL с определенными представлениями и функциями обработки запросов.

- `path` - это наиболее распространенный способ определения маршрутов в Django. Он использует простой синтаксис для определения URL-маршрутов с обязательным указанием конкретного пути (URL-шаблона) и соответствующего представления (view). В `path` можно использовать переменные частей URL, которые передаются в представление как параметры
- `re_path` - это альтернативный способ определения маршрутов в Django с использованием регулярных выражений (regular expressions). Он позволяет определить более сложные шаблоны URL с помощью регулярных выражений для более гибкой обработки запросов.

Какие встроенные представления (views) есть в Django?

- `View` - это базовое класс-представление, который можно использовать для создания собственных классов-представлений на основе классов. Он предоставляет методы для обработки различных типов запросов, таких как GET, POST, PUT и DELETE.
- `TemplateView`- представление, которое отображает статический HTML-шаблон без какой-либо дополнительной логики. Оно полезно, когда не требуется дополнительная обработка данных и достаточно просто отобразить HTML-страницу.
- `ListView`- представление, которое отображает список объектов определенной модели. Оно автоматически извлекает данные из базы данных и передает их в шаблон для отображения.
- `DetailView`- представление, которое отображает детали определенного объекта модели. Оно получает одиночный объект из базы данных на основе его уникального идентификатора и передает его в шаблон для отображения.
- `FormView`- представление, которое отображает форму и обрабатывает данные, отправленные пользователем. Оно упрощает создание форм и их валидацию.
- `CreateView`- представление, которое обрабатывает создание нового объекта модели на основе данных, отправленных пользователем через форму.
- `UpdateView`- представление, которое обрабатывает обновление существующего объекта модели на основе данных, отправленных пользователем через форму.
- `DeleteView`- представление, которое обрабатывает удаление объекта модели.

В чем разница между Flask и Django?

Flask и Django - это два популярных веб-фреймворка для разработки веб-приложений на языке Python. Они имеют разные философии и подходы, что приводит к некоторым ключевым различиям между ними. Вот основные различия между Flask и Django:

- Размер и сложность: Flask - это минималистичный и легковесный фреймворк. Он предоставляет основные инструменты для создания веб-приложений, но не содержит множество встроенных компонентов, что делает его гибким и легким в освоении. Django, с другой стороны, является полноценным фреймворком с обширным набором инструментов и встроенных функций, что делает его более мощным, но может быть сложнее в изучении для начинающих разработчиков.
- Философия и подход: Flask следует принципу "больше свободы", предоставляя разработчикам больше свободы в выборе инструментов и библиотек, которые они хотят использовать. Django же следует принципу "больше готовых решений", предоставляя множество встроенных компонентов и функций, что упрощает и ускоряет разработку веб-приложений.
- ORM и базы данных: Оба фреймворка имеют встроенные ORM (Object-Relational Mapping) для работы с базами данных. В Django ORM является частью фреймворка и предоставляет более расширенные возможности, в то время как в Flask ORM может быть выбран отдельно в зависимости от предпочтений разработчика.
- Административная панель: Django предоставляет встроенную административную панель, которая автоматически генерирует интерфейс для управления данными в приложении. В Flask такая административная панель не предоставляется встроенно, но может быть добавлена сторонними расширениями.
- Расширяемость: Flask обладает высокой степенью расширяемости, позволяя разработчикам выбирать и добавлять сторонние расширения и библиотеки по мере необходимости. Django также поддерживает расширения, но его обширные встроенные функции могут оказаться более удобными для разработки небольших и средних проектов.

Оба фреймворка имеют свои преимущества и подходят для различных сценариев разработки. Flask хорош для небольших проектов, требующих минимальной настройки и большей свободы выбора инструментов. Django же отлично подходит для разработки крупных проектов с готовыми решениями и хорошо структурированным кодом. Выбор между Flask и Django зависит от требований вашего проекта, уровня опыта разработчиков и предпочтений в разработке.

Что такое Django REST Framework?

Django REST Framework (DRF) - это мощный инструментарий для создания веб-приложений с использованием фреймворка Django и разработки веб-сервисов API (Application Programming Interface). Он предоставляет разработчикам удобные средства для создания RESTful API, которые могут взаимодействовать с клиентскими приложениями.

Основные возможности Django REST Framework включают:

- Сериализация данных: DRF предоставляет мощные средства для преобразования моделей Django и других типов данных в JSON или другие форматы данных, а также для десериализации обратно в объекты Python.
- Поддержка HTTP-методов: DRF обрабатывает различные HTTP-методы (GET, POST, PUT, DELETE и т.д.) для взаимодействия с API ресурсами.
- Аутентификация и авторизация: DRF обеспечивает различные способы аутентификации пользователей и авторизации доступа к API ресурсам, включая токены, базовую аутентификацию, аутентификацию OAuth и многое другое.
- Представления (Views): DRF предоставляет различные типы представлений, такие как представления на основе функций или классов, которые определяют, как обрабатывать запросы и возвращать ответы.
- Роутинг (Routing): DRF облегчает управление маршрутами для ваших API ресурсов, позволяя определить и настраивать URL-шаблоны для различных действий API.
- Пагинация и фильтрация: DRF предоставляет возможности для настройки пагинации данных и фильтрации результатов запросов.
- Обработка ошибок: DRF предоставляет средства для обработки ошибок и возврата соответствующих ответов клиенту.

Какой класс наследуется для создания представлений (views) в Django REST Framework и каковы основные различия между ними?

В Django REST Framework (DRF), для создания представлений (views) используются классы, которые наследуются от различных базовых классов, предоставляемых самим фреймворком. Основные классы представлений в DRF включают:

- `APIView`: Это базовый класс представлений, от которого можно наследовать для создания пользовательских представлений. Он предоставляет базовый функционал для обработки HTTP-запросов, но не содержит предопределенной логики для обработки конкретных методов запросов.
- `APIView` с миксинами (Mixins): DRF предоставляет миксины (Mixins) - это классы, которые содержат некоторую специфическую логику, которую можно добавлять в представления. Классы представлений с миксинами обычно используют сочетание базового `APIView` и одного или нескольких миксинов для предоставления дополнительных функций, таких как аутентификация, авторизация, пагинация, фильтрация и другие.
- `GenericAPIView`: Этот класс наследуется от `APIView` и предоставляет дополнительный функционал для работы с моделями Django и сериализаторами. Он часто используется для создания представлений, связанных с моделями, такими как `ListCreateAPIView` для списка и создания записей или `RetrieveUpdateDestroyAPIView` для получения, обновления и удаления конкретной записи.
- `ViewSet`: Это особый тип представлений, который предоставляет полный набор CRUD (Create, Retrieve, Update, Delete) операций для моделей Django. `ViewSet` обычно используется совместно с `Router`, который позволяет автоматически настраивать URL-шаблоны для представлений.

Какие типы аутентификации поддерживает Django REST Framework, и как настроить аутентификацию для вашего API?

Django REST Framework (DRF) поддерживает различные типы аутентификации, чтобы обеспечить безопасный доступ к вашему API. Вот некоторые из поддерживаемых типов аутентификации:

- `SessionAuthentication`: Этот тип аутентификации использует механизм сессий Django, основанный на cookies. Когда пользователь входит в систему, создается сессия, и сервер хранит информацию о пользователе на стороне сервера. При каждом запросе клиент должен отправить сессионную куку для подтверждения своей аутентификации.
- `TokenAuthentication`: Этот тип аутентификации использует токены для аутентификации пользователей. При успешной аутентификации сервер выдает уникальный токен для пользователя, который затем отправляется с каждым запросом в заголовке Authorization. Токены обычно имеют ограниченное время жизни.
- `BasicAuthentication`: Это базовый тип аутентификации, который предоставляет базовый механизм аутентификации с использованием имени пользователя и пароля. Пользователь должен предоставить учетные данные с каждым запросом в заголовке Authorization.
- `TokenAuthentication`: Этот тип аутентификации аналогичен `TokenAuthentication`, но использует токены безопасности JSON (JWT) для аутентификации. JWT представляет собой JSON-объект, подписанный сервером, и содержит информацию о пользователе и правах.
- `OAuth` и `OAuth2`: DRF поддерживает аутентификацию через стандарты OAuth и OAuth2, которые позволяют пользователям предоставлять доступ к своим данным сторонним приложениям без раскрытия своих учетных данных.

Что такое пагинация в Django REST Framework?

Пагинация в Django REST Framework (DRF) - это механизм, который позволяет разбивать большой объем данных на отдельные страницы, чтобы упростить обработку и передачу данных клиентским приложениям. Когда в вашем API возвращается много результатов, пагинация делает их доступными по частям (страницам), что улучшает производительность и оптимизирует использование ресурсов.

DRF предоставляет несколько классов пагинации, которые можно использовать в представлениях для различных стилей пагинации, включая:

- `PageNumberPagination`: Этот класс позволяет пагинировать результаты по номерам страниц. Клиентское приложение может запросить конкретную страницу или переходить на следующую/предыдущую страницу.
- `LimitOffsetPagination`: Позволяет пагинировать результаты с использованием смещения и лимита. Клиентское приложение указывает смещение (offset) и лимит (limit) для запроса определенных данных.
- `CursorPagination`: Пагинация на основе курсора предоставляет клиенту токен, который используется для получения следующей или предыдущей страницы данных. Этот метод особенно полезен для пагинации больших объемов данных без необходимости смещения.

Настройка пагинации в DRF достаточно проста. Вы можете выбрать подходящий класс пагинации и указать его в настройках DRF, а затем добавить атрибут `pagination_class`в представления, для которых вы хотите применить пагинацию. Это позволяет автоматически разбивать результаты запросов на страницы и предоставлять информацию о доступных страницах в ответе клиентскому приложению.

Как можно ограничить доступ к определенным представлениям или ресурсам в Django REST Framework с помощью авторизации?

Ограничение доступа к определенным представлениям или ресурсам в Django REST Framework (DRF) можно реализовать с помощью различных классов авторизации, предоставляемых DRF. Каждый класс авторизации предоставляет специфические методы для проверки прав доступа пользователя к определенному представлению или ресурсу.

```python
from rest_framework.permissions import IsAuthenticated
from rest_framework.views import APIView

class MyProtectedView(APIView):
    permission_classes = [IsAuthenticated]

    def get(self, request):
        # Ваш код представления
```

Что такое ViewSet?

В Django REST Framework (DRF), `ViewSet` - это особый тип представления (view), который предоставляет полный набор CRUD (Create, Retrieve, Update, Delete) операций для моделей Django. ViewSet упрощает и стандартизирует создание API для ваших моделей, не требуя от вас явно определять методы для каждой операции.

ViewSet объединяет обработку различных HTTP-методов (GET, POST, PUT, DELETE и др.) для определенной модели в одном представлении. Это позволяет сократить код и уменьшить повторение логики для разных операций, связанных с моделью.

ViewSet обычно используется совместно с `Router`, который автоматически настраивает URL-шаблоны для представлений, основанных на их функциональности. Когда вы объявляете ViewSet, Router автоматически определяет соответствующие URL-шаблоны для каждой операции (список, создание, получение, обновление, удаление).

```python
from rest_framework import viewsets
from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyModelViewSet(viewsets.ModelViewSet):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer
```

Что такое Serializer?

В Django REST Framework (DRF) `serializer` - это класс, который определяет, как модели Django и другие данные должны быть преобразованы в форматы, такие как JSON, XML, или другие, и обратно. Сериализаторы позволяют преобразовывать сложные структуры данных, такие как модели Django и связанные объекты, в форматы, удобные для передачи через API.

Основная задача сериализаторов - преобразование объектов моделей Django или других данных в словари Python (или JSON), которые могут быть прочитаны и переданы через API. Сериализаторы также обрабатывают десериализацию, то есть преобразование словарей обратно в объекты моделей или данные.

```python
from rest_framework import serializers
from myapp.models import MyModel

class MyModelSerializer(serializers.ModelSerializer):
    class Meta:
        model = MyModel
        fields = '__all__'
```

Как проходит валидация в Django REST Framework?

В Django REST Framework (DRF) валидация происходит в несколько этапов и обеспечивает проверку данных перед сохранением или обновлением объектов модели Django через API. Это позволяет удостовериться в том, что данные, полученные от клиентов, соответствуют ожидаемому формату и правилам.

Валидация в DRF проходит следующим образом:

- При получении данных от клиента, DRF сначала десериализует эти данные с помощью соответствующего сериализатора. Это преобразует данные из JSON или другого формата в словарь Python.
- После десериализации, DRF запускает валидацию данных, используя метод `is_valid()` на сериализаторе. В этом этапе проверяются встроенные валидаторы, определенные в модели Django (например, `unique`, `max_length`, `blank` и т. д.), а также кастомные валидаторы, которые могут быть определены в сериализаторе.
- Если данные не прошли валидацию, DRF вернет ошибку с информацией о том, какие поля не прошли проверку и какие правила валидации были нарушены. Код ответа будет 400 Bad Request, а в теле ответа будет содержаться информация о недопустимых данных.
- Если данные прошли валидацию успешно, DRF выполнит сохранение или обновление объектов модели Django с использованием данных из десериализованного словаря. В этом этапе также могут применяться дополнительные правила валидации, определенные в методах представлений или переопределенных методах сериализаторов.

Что такое WSGI?

WSGI (Web Server Gateway Interface) - это стандарт интерфейса, который определяет способ взаимодействия между веб-приложениями, написанными на языке Python, и веб-серверами. Он позволяет веб-серверам передавать запросы к приложениям и получать от них ответы, обеспечивая таким образом взаимодействие между серверами и веб-приложениями.

Популярные WSGI-совместимые веб-серверы:

- Gunicorn (Green Unicorn): Это простой и мощный веб-сервер, спроектированный для обработки Python WSGI приложений. Gunicorn широко используется и известен своей стабильностью и производительностью.
- uWSGI: Это веб-сервер и приложение WSGI, который предлагает высокую производительность и широкий набор функций. uWSGI может работать с различными протоколами и поддерживает асинхронные фреймворки.
- mod_wsgi: Это модуль для серверов Apache, который позволяет запускать WSGI приложения внутри сервера. Он обеспечивает простую интеграцию WSGI-приложений в среду Apache.

Что такое Django Channels?

Django Channels - это расширение Django, которое позволяет добавить поддержку WebSocket и асинхронных протоколов в ваше Django приложение. Стандартный вариант Django работает синхронно, что означает, что каждый запрос обрабатывается последовательно, что не подходит для реализации в реальном времени или долгих операций в фоновом режиме.

Django Channels позволяет работать асинхронно, что предоставляет следующие возможности:

- WebSocket поддержка: Django Channels позволяет обрабатывать WebSocket соединения, что позволяет реализовать взаимодействие в режиме реального времени между клиентом и сервером. WebSocket позволяет длительным соединениям между клиентом и сервером, что особенно полезно для чатов, уведомлений, онлайн-игр и других приложений, где данные должны передаваться в режиме реального времени.
- Асинхронное выполнение: Django Channels позволяет выполнять асинхронные задачи, что увеличивает производительность вашего приложения, особенно при обработке долгих операций в фоновом режиме. Асинхронное выполнение позволяет обрабатывать несколько запросов одновременно, что улучшает отзывчивость и производительность вашего приложения.
- Асинхронные вьюхи: Django Channels позволяет создавать асинхронные представления (views), что позволяет обрабатывать запросы асинхронно. Это особенно полезно для обработки WebSocket соединений и асинхронных задач.
- Каналы: Каналы в Django Channels - это механизм для обмена сообщениями между веб-сервером и клиентами в режиме реального времени. Каждое WebSocket соединение имеет свой собственный канал, который позволяет отправлять и получать сообщения между сервером и клиентами.

SHORT VERSION OF CV

ALEXEY BELOV

PYTHON ARCHITECT

LOCATION
Kyiv, Ukraine

EMAIL
alenorze@gmail.com

LINKEDIN
linkedin.com/in/alenorze

GITHUB
github.com/Alenorze

LANGUAGES
English - Advanced
Ukrainian - Native
Russian - Native

EXPERIENCE IN DEVELOPMENT
8 years

Summary: 8 years of experience in backend development, especially with Python, broad knowledge in frontend and clouds.

Skills: Python, Rust, Typescript, Javascript, Go, Flask, FastAPI, Django, Django REST Framework, SQL, PostgresSQL, MongoDB, CI/CD, Docker, Kubernetes, AWS, GCP, Azure, SQLAlchemy, Numpy, OpenCV, MongoDB, WEB3, Solidity, Solana, Linux, Windows, MacOS, AsyncIO, Celery, RabbitMQ, Redis, React.JS, Angular, Vue.js

Work experience:

Customertimes | Python Architect | September 2020 - Present
Team leading, 10 members mixed team. Work as Solution Achitect,
python backend development, microservice and monolithic architecture design.

MWDN | Senior Backend Cloud Developer | March 2020 - September 2020
Development of a global repository of genomes and genomic data and associated metadata,
using AWS infrastructure.

Confidential | Python Team Lead | September 2018 - February 2020
Creation of a control system for non-normative and morally violating statements, obscene language.
Using tensorflow as a basic tool. Leading team of 6 python engineers.

TND | Senior Python Developer | September 2016 - September 2018
Development of various api to simplify interaction with various cryptocurrency exchanges

NEAPOLYS | Middle Python Developer | March 2014 - August 2016
Development of an interface for non-relational databases based on django,
support and visual improvements of selling pages.

FREELANCE | Junior Python Developer | January 2014 - March 2014
Development of various applications based on the Django and Flask frameworks.
Creating ecommerce applications using mvc methodologies.

FREELANCE | Junior Javascipt Developer | September 2013 - January 2014
Creating various html layouts and js scripts.

ALEXEY BELOV

PYTHON ARCHITECT

INFO

EMAIL
alenorze@gmail.com

LOCATION
Kyiv, Ukraine

INSTAGRAM
@alenorze

FACEBOOK
@alenorze

LINKEDIN
linkedin.com/in/alenorze

GITHUB
github.com/Alenorze

TWITTER
@alenorze

TELEGRAM
@alenorze

SKYPE
alenorze

SUMMARY

• 8 years of experience in developing, engineering, testing, designing and implementing of various standalone and client-server architecture-based application software in Python, Rust, Go and Javascript. There is also a lot of experience in machine learning, in particular, computer vision, deep learning, natural language processing, integrating machine learning algorithms.

• Expertise in Object Oriented Concepts, Object Oriented Design (OOD), Object Oriented Analysis(OOA) and Programming concepts in Python and Rust.

• Deep understanding of microservice and monolith architecture.

• Good experience in developing web applications implementing Model View Controller architecture using Flask and Django web application frameworks.

• Good experience in developing using Django(Django REST Framework), Flask, FastAPI to create various API backends.

• Good experience in creation of smart contracts using Solidity.

• Strong experience in creation of NFT projects using Solana and Metaplex.

• Good at writing SQL Queries, Stored procedures, functions, packages, tables, views, triggers using relational databases like PostgreSQL, MySQL , Aurora, Sqlite3 and Good knowledge in using NoSQL database MongoDB, Cassandra, DynamoDB.

• Experience with JavaScript and TypeScript leading Frameworks Team like Angular, React. js, Vue.js, and Node.js.

• Strong experience with Server Side Rendering

• Strong frontend UI, UX development skills using scripting languages like HTML, CSS(SASS, SCSS), JavaScript, JQuery, AJAX, JSON, DOM, JSP.

• Experience in the development and training of various neural networks using Tensorflow and Pytorch.

• Great experience with computer vision and image progressing using OpenCV.

• Extensive experience in working with graphics and tensor Nvidia accelerators, in order to repeatedly increase productivity in the training of neural networks and video processing.

• Development in cloud infrastructure like AWS, GCC, Azure.

• Hands on experience in AWS provisioning and good knowledge of AWS services like EC2, S3, SageMaker, Cognito, Lambda, Certificate Manager, ELB, RDS, Redshift, IAM, Route 53, VPC, Auto scaling, CloudFront, CloudWatch, CloudTrail, CloudFormation, Security Groups, Balance Loader.

• Experience in creating a CI / CD infrastructure using Travis and Jenkins.

• Understanding of security and encryption technologies.

• Knowledge of authentication protocols including OpenID, OIDC, OAuth, SAML, and LDAP.

• Knowledge of Docker containerization and Kubernetes/EKS cluster management for container orchestration.

• Experience with Version Control Systems like GIT, Mercury.

• Experienced in using Caching applications for large scale applications like Memcached, Redis.

• Good Knowledge in implementation of Python best Practices (PEP-8).

• Worked on various operating systems like Windows, Ubuntu, MacOS.

• Testing with PyTest, unittest libs for all Python applications.

• Evaluated potential software products based on new and existing system development and migration requirements.

• Performed regression and system-level testing to verify software quality and function before it was released.

• Superior Troubleshooting and Technical support abilities with Migrations, Network connectivity and Security and Database applications.

• Assessed costs and risks associated with developing new features and product.

• Experienced in requirement gathering, use case development, Business Process Flow, Business Process Modelling.

• Able to analyse a requirement and act accordingly by managing the resources efficiently.

• Involved in Python OOD code for quality, logging, monitoring, and debugging code optimization.

• Practical experience migrating large-scale infrastructure onto public cloud technology platforms such as AWS, Azure, GCP.

• Excellent software engineering fundamentals, including knowledge of algorithms and data structures, UML and other forms of systems design collaboration.

SKILLS

Python		Typescript	
Go		Solidity	
Rust		Solana	
Javascript		SQL	

Languages: Python, Go, Rust, Javascript, Typescript
Frameworks: Django(DRF, DC), Flask, FastAPI, ReactJS, Angular, ReactJS, VueJS, NodeJS, Tensorflow, PyTorch, Pyramid, CherryPy
Cloud Environment: AWS(EC2, S3, RDS, Elastic Beanstalk, Cognito, IAM, Sagemaker, Glacier, ECR, ECS, Batch, Step Functions, Lambda, CloudWatch, CloudFront, ElastiCache), GCP, Azure, DO
Operating System: Linux, Windows, MacOS
Tools: Celery, RabbitMQ, SQLAlchemy, Redis, Docker, Kubernetes, Elasticsearch, Keras, Jira, OpenCV, Numpy, Skimage, Matplotlib, Scipy, Pandas, OpenNN, Google ML Kit, Apache Spark, SpaCy, GraphQL
Databases: PostgreSQL, MongoDB, MySQL, Aurora, DynamoDB, Cassandra
Servers: Nginx, Apache
Web3.0: Solidity, Solana, web3.py, web3.js, Hardhat, Metaplex

LANGUAGES

ENGLISH - Advanced

UKRAINIAN - Native

RUSSIAN - Native

EDUCATION

BACHELOR DEGREE

National University of «Kyiv-Mohyla Academy» / 2017 - 2018

Faculty of Science - Physics

BACHELOR DEGREE

National Aviation University / 2018 - 2022

Faculty of Informatics - Software engineering

ABOUT ME

I am a Python developer with 8 years of experience, living in Kyiv, at 5 years old age I received my first computer. In 2014 I started my development career as Javascript Developer. After a while in 2015 year, I started working freelance, and at the same time, I started learning Python, later, I started working with the Flask and Django frameworks. For two years I worked in various teams and small startups. Then I started working at outsourcing companies as a web developer. In 2018 year I started working in the field of machine learning. In 2020 year, I started some positions as the Python Team Lead. Right now, I have some plans to expand my knowledge in the Python field to the maximum, as well as learn new languages such as Rust, and Go. Currently, 2022 year, started a movement toward the WEB3.0 paradigm.

WORK EXPERIENCE

Customertimes

Python Architect

September 2020 - Present

Work directly as a Python developer. Development of internal solutions in the practice department. Creation of my own initiatives within the company, and the subsequent implementation and sale of them to clients and partners. Multiple participation in pre-sales related to areas of Python and Machine Learning. Development of Python development department. Conducting technical interviews, Code Review, choice of technical stack, building global application architectures. Solution assistance and guidance for Solution Architects. Management of development teams in the Team Lead role. Mentoring newly arrived Junior, Middle, Senior developers.

Creation of a user support system using a microservice architecture based on Python and its frameworks. Designing the application architecture and machine learning model. Creation of dataset based on user requests database. Highload development. Creation of AI Support bot, system on B2B basics.

Creation of a system of automatic training and re-training of models.

Designing and architecting a solution for moving relational data into ElasticSearch for working with big data, enabling rapid reporting, and performing predictive analytics.

Also work with Salesforce and obtaining certificates.

In the role of Solution Architect, creating and conducting various presentations, including architecture, business logic, areas of use, possible client painpoints, timelines.

Development of Solana applications with Rust. Creation and implementation of Solidity smart contracts. Promotion of NFT projects.

MWDN

Senior Back End Cloud Developer

March 2020 - September 2020

Development of a global repository of genomes and genomic data and associated metadata. Which allows various research labs to interact with genomes, exomes and other statistics. For research in the field of medicine and in search of vaccines and medication for HIV, AIDS, cancer, Covid-19, etc.

The project was based entirely on AWS cloud technologies. Designing of serverless architecture. Which involved the pretty woven development of python lambdas connected through API Gateway. Also for more time-consuming operations, ECS and Batch was used to process data with pipelines.

Work with DynamoDB, RDS for storage, and S3 + Glacier for large files, Athena for Big Data. Creation of step functions with over 50 layers.

Choice of technical stack, code review. Formation of technical specifications. Collaboration with Bioinformatics Engineers.

Collaborating with product and customers to search for the most suitable technical solutions and create the features necessary for the existence of the project.

Project — confidential
Python Team Lead

September 2018 - February 2020

Creation of a control system for non-normative and morally violating statements, obscene language. Using tensorflow as a basic tool. Creating a set of convolutional neural networks. Multi-GPU network training. Recognition of images containing adult content. Work with math libraries like Numpy, Scipy, Pandas. Creation and training, feed forward neural networks, convolutional neural networks, deconvolutional networks using Keras and Tensorflow. Working with a GPU array to improve learning performance. Backend development with Flask, Django, CherryPy. Creation of a service for identifying external vulnerabilities associated with a specific domain. Front-end development based on Angular, ReactJS. Creation of algorithms capable of finding similarities in phrases, based on the Levenshtein distance. Attended many day-to-day meetings with developers and users and performed QA testing on the application.

TND
Senior Python Developer

September 2016 - September 2018

Development of various api to simplify interaction with various cryptocurrency exchanges. Create additional layers of protection based on AWS Cognito. Configure interaction with the AWS environment, including RDS, S3, EC2. Telegram bot integration.
Lead a small team to creating a cryptocurrency exchange service. Collaborating with automation engineers to continually improve and grow test automation. Choice of technical stack, code review. Formation of sprints for team members. Formation of technical specifications for each member. Developing games for training managers, working with employees, interacting with the GCP environment. Also the development of frontend based on Angular. Development of asynchronous interfaces using asyncio.

NEAPOLYS
Middle Python Developer

March 2014 - August 2016

Support and completion of databases, including tables with patterns. Development of an interface for non-relational databases based on django, support and visual improvements of selling pages, as well as the introduction of tables for the agro-vendor. Working with raw SQL query, creation of tables. Working with celery and redis. Creation of custom middleware system. Development of frontend based on HTML and JQuery. Maintenance of project. Unit test coverage. Security practics implementation.

FREELANCE
Junior Python Developer

January 2014 - March 2014

Development of various applications based on the Django and Flask frameworks. Creating ecommerce applications using mvc methodologies.

FREELANCE
Junior Javascipt Developer

September 2013 - January 2014

Creating various html layouts and js scripts.