1. Реализовать чат для общения c помощью библиотеки socket на Python. Клиент подключается к серверу и отправляет сообщение, сервер получает это сообщение и отправляет в ответ клиенту тоже самое сообщение. Реализация должна быть асинхронной, то есть несколько клиентов могут подключаться к серверу одновременно. Рассмотреть реализацию с использованием генераторов в Python и библиотеки asyncio.

Для реализации чата с использованием библиотеки socket и асинхронности в Python, вы можете использовать библиотеку asyncio. Вот пример простого чата с использованием asyncio:

import asyncio

class ChatServer:

def \_\_init\_\_(self):

self.clients = []

async def handle\_client(self, reader, writer):

self.clients.append(writer)

print("New client connected")

while True:

data = await reader.read(100)

message = data.decode().strip()

print(f"Received message: {message}")

if message == "exit":

self.clients.remove(writer)

print("Client disconnected")

break

for client in self.clients:

if client != writer:

client.write(data)

await client.drain()

async def run\_server(self):

server = await asyncio.start\_server(

self.handle\_client, '127.0.0.1', 8888)

async with server:

await server.serve\_forever()

chat\_server = ChatServer()

asyncio.run(chat\_server.run\_server())

Этот код создает сервер чата, который принимает подключения от клиентов и передает сообщения между ними. Каждый клиент обрабатывается в отдельном корутине, что позволяет серверу обслуживать несколько клиентов одновременно. Клиенты могут быть реализованы с использованием библиотеки asyncio также, отправляя сообщения серверу и получая ответы.

Данный код реализует простой чат-сервер с использованием библиотеки asyncio в Python. 1. Создается класс ChatServer, который содержит список подключенных клиентов (self.clients). 2. Внутри класса определен метод handle\_client, который асинхронно обрабатывает подключение каждого клиента. Когда новый клиент подключается, его объект writer добавляется в список self.clients. Затем сервер ожидает получения сообщений от клиента. Если приходит сообщение "exit", то клиент удаляется из списка и обработка его соединения завершается. В противном случае сообщение отправляется всем другим клиентам. 3. Метод run\_server запускает сервер на адресе '[127.0.0.1](https://127.0.0.1/)' и порту 8888, используя функцию asyncio.start\_server. Затем сервер остается активным, пока не будет остановлен. 4. Создается объект chat\_server класса ChatServer, и вызывается асинхронная функция asyncio.run(chat\_server.run\_server()), которая запускает сервер. Таким образом, этот код создает асинхронный чат-сервер, который может принимать подключения от нескольких клиентов и обрабатывать сообщения между ними.

import asyncio

async def send\_message(message):

reader, writer = await asyncio.open\_connection('127.0.0.1', 8888)

print(f'Send: {message!r}')

writer.write(message.encode())

await writer.drain()

writer.close()

await writer.wait\_closed()

asyncio.run(send\_message('Hello, World!'))

Это простой пример использования библиотеки asyncio для создания асинхронного чата на Python.

Данный код представляет собой асинхронную функцию send\_message, которая используется для отправки сообщения на сервер. 1. Внутри функции создается соединение с сервером с помощью asyncio.open\_connection('127.0.0.1', 8888). Эта функция возвращает объекты reader и writer, которые можно использовать для обмена данными с сервером. 2. Затем функция выводит сообщение, которое будет отправлено, с помощью print(f'Send: {message!r}'). 3. Далее сообщение кодируется в байтовый формат и отправляется на сервер с помощью writer.write(message.encode()). После этого вызывается await writer.drain(), чтобы убедиться, что все данные были отправлены. 4. После отправки сообщения соединение закрывается с помощью writer.close(), и ожидается завершение закрытия с помощью await writer.wait\_closed(). 5. Наконец, асинхронная функция send\_message запускается с помощью asyncio.run(send\_message('Hello, World!')). Это запускает цикл событий asyncio и вызывает функцию send\_message с сообщением "Hello, World!" для отправки на сервер. Таким образом, данный код позволяет отправлять сообщения на сервер с использованием асинхронной функции и библиотеки asyncio.

1. Реализовать механизм записи данных пользователя в текстовый файл с последующим чтением этих данных. Пользователь должен иметь возможность вводить данные в консоль в формате id last\_name first\_name middle\_name age. После каждого введенного сообщения данные должны быть сохранены в текстовый файл. Если пользователь вводит команду showdata\* , все данные из файла должны быть выведены в консоль в видетаблицы: id last\_name first\_name middle\_name age 1 Ivanov Ivan Ivanovich 27 2 Stepanov Anton Mikhailovich 56 … … … … … n + 1 Fedorov Nikita Stepanovich 35. Если пользователь вводит команду showdata {integernumber}, например, showdata 10, то необходимо вывести первые 10 записей из текстового файла в формате таблицы представленной выше.

Для реализации этой задачи, вы можете использовать стандартные возможности Python для работы с файлами. Вот пример кода, который позволяет пользователю вводить данные в консоль и сохранять их в текстовый файл, а также выводить данные из файла в консоль в соответствии с заданными условиями:

def save\_data\_to\_file(data):

with open('data.txt', 'a') as file:

file.write(data + '\n')

def show\_all\_data():

with open('data.txt', 'r') as file:

for line in file:

print(line.strip())

def show\_data\_by\_limit(limit):

with open('data.txt', 'r') as file:

for \_ in range(limit):

line = file.readline().strip()

if not line:

break

print(line)

def main():

while True:

user\_input = input("Введите данные в формате id last\_name first\_name middle\_name age: ")

if user\_input.lower() == "showdata\*":

show\_all\_data()

elif user\_input.startswith("showdata "):

try:

limit = int(user\_input.split()[1])

show\_data\_by\_limit(limit)

except ValueError:

print("Некорректный формат команды showdata")

else:

save\_data\_to\_file(user\_input)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Этот код позволяет пользователю вводить данные в консоль и сохранять их в файл data.txt. Кроме того, он обрабатывает команды showdata\* и showdata {integernumber} для вывода данных из файла в соответствии с заданными условиями.

Данный код представляет собой простую программу для ввода данных и их сохранения в файл, а также для отображения сохраненных данных в консоли. 1. Функция save\_data\_to\_file(data) открывает файл "data.txt" в режиме добавления ('a') и записывает в него переданные данные с переносом строки. 2. Функция show\_all\_data() открывает файл "data.txt" в режиме чтения ('r') и выводит все строки из файла в консоль, удаляя символы переноса строки с помощью strip(). 3. Функция show\_data\_by\_limit(limit) также открывает файл "data.txt" в режиме чтения ('r') и выводит указанное количество строк из файла (ограниченное параметром limit), также удаляя символы переноса строки с помощью strip(). 4. Функция main() представляет основной цикл программы, который запрашивает у пользователя ввод данных в определенном формате и сохраняет их в файл. Если пользователь вводит команду "showdata\*", программа выводит все сохраненные данные. Если пользователь вводит команду "showdata N", где N - целое число, программа выводит N строк данных из файла. 5. В конце кода проверяется, является ли данный скрипт основным (главным), и если да, то вызывается функция main(). При запуске данной программы будет предложено ввести данные в формате "id last\_name first\_name middle\_name age". После этого программа будет ожидать команды от пользователя. Если пользователь введет "showdata\*", программа выведет все сохраненные данные из файла "data.txt". Если пользователь введет "showdata N", где N - целое число, программа выведет N строк данных из файла.

1. Реализовать консольный калькулятор Необходимо реализовать консольный калькулятор, позволяющий выполнять такие арифметические операции, как: − сложение − вычитание − деление − умножение − возведение в степень Все действия должны быть обернуты в блоки try except и при вводе некорректных данных необходимо выводить в консоль сообщение для пользователя с описанием ошибки.

Для реализации консольного калькулятора в Python, вы можете использовать следующий код:

def add(x, y):

return x + y

def subtract(x, y):

return x - y

def multiply(x, y):

return x \* y

def divide(x, y):

if y == 0:

raise ValueError("Деление на ноль недопустимо")

return x / y

def power(x, y):

return x \*\* y

def main():

while True:

try:

operation = input("Выберите операцию (+, -, \*, /, \*\*) или введите 'exit' для выхода: ")

if operation.lower() == "exit":

break

if operation not in ['+', '-', '\*', '/', '\*\*']:

print("Некорректная операция")

continue

num1 = float(input("Введите первое число: "))

num2 = float(input("Введите второе число: "))

if operation == '+':

result = add(num1, num2)

elif operation == '-':

result = subtract(num1, num2)

elif operation == '\*':

result = multiply(num1, num2)

elif operation == '/':

result = divide(num1, num2)

elif operation == '\*\*':

result = power(num1, num2)

print(f"Результат: {result}")

except ValueError as e:

print(f"Ошибка: {e}")

except ZeroDivisionError as e:

print(f"Ошибка: {e}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Этот код предоставляет простой консольный интерфейс для выполнения арифметических операций. Он обрабатывает ошибки, такие как некорректный ввод чисел и деление на ноль, и выводит соответствующие сообщения об ошибках.

Данный код представляет собой простой калькулятор, который позволяет пользователю выполнять арифметические операции (+, -, \*, /, \*\*) над числами. 1. Функции add, subtract, multiply, divide и power выполняют соответствующие арифметические операции над двумя числами и возвращают результат. 2. Функция main представляет основной цикл программы, который запрашивает у пользователя выбор операции и два числа, над которыми нужно выполнить операцию. Затем программа вызывает соответствующую функцию для выполнения операции и выводит результат. Если пользователь вводит "exit", программа завершает свою работу. 3. В конце кода проверяется, является ли данный скрипт основным (главным), и если да, то вызывается функция main(). При запуске данной программы будет предложено выбрать операцию (+, -, \*, /, \*\*) или ввести "exit" для выхода. Затем пользователю будет предложено ввести два числа. После этого программа выполнит выбранную операцию над этими числами и выведет результат. Если произойдет ошибка (например, деление на ноль), программа сообщит об этом пользователю.

1. Повторить механизм обмена данными между клиентом и сервером с помощью сокетов. (Повторить асинхронную реализацию сокетов)

Для повторения механизма обмена данными между клиентом и сервером с помощью сокетов в Python, вы можете создать простой сервер и клиент с использованием модуля socket. Ниже приведен пример кода для сервера и клиента:

Сервер:

import socket

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_socket.bind(('127.0.0.1', 12345))

server\_socket.listen(1)

print("Ждем подключения клиента...")

client\_socket, client\_address = server\_socket.accept()

print(f"Подключен клиент {client\_address}")

while True:

data = client\_socket.recv(1024)

if not data:

break

print(f"Получено от клиента: {data.decode()}")

client\_socket.sendall(data)

client\_socket.close()

server\_socket.close()

Данный код представляет собой простой TCP-сервер, который принимает соединение от клиента, получает данные от клиента, отправляет их обратно и закрывает соединение. 1. Сначала создается объект сокета server\_socket с использованием socket.socket(socket.AF\_INET, socket.SOCK\_STREAM). Это создает TCP-сокет для общения по сети. 2. Затем вызывается метод bind, чтобы связать сокет с адресом и портом. В данном случае сервер будет слушать подключения на адресе '[127.0.0.1](https://127.0.0.1/)' (localhost) и порту 12345. 3. Метод listen(1) указывает серверу прослушивать входящие подключения. Аргумент 1 означает, что сервер будет принимать только одно подключение одновременно. 4. После этого выводится сообщение "Ждем подключения клиента...". 5. Метод accept() блокирует выполнение программы, пока не будет получено входящее подключение. Когда клиент подключается, метод возвращает кортеж, содержащий новый сокет client\_socket для общения с клиентом и адрес клиента client\_address. 6. Далее запускается бесконечный цикл, в котором сервер получает данные от клиента с помощью метода recv(1024), который читает до 1024 байт данных из сокета. 7. Если данные не получены (то есть data равно пустой строке), цикл прерывается с помощью break. 8. В противном случае данные декодируются из байтового формата в строку с помощью data.decode() и выводятся на экран. 9. Затем сервер отправляет обратно те же данные клиенту с помощью метода sendall(data). 10. После выхода из цикла происходит закрытие сокетов с помощью методов close(). Этот код представляет собой простую реализацию однопоточного TCP-сервера, который может обслуживать только одно подключение одновременно.

Клиент:

import socket

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

client\_socket.connect(('127.0.0.1', 12345))

message = "Привет, сервер!"

client\_socket.sendall(message.encode())

data = client\_socket.recv(1024)

print(f"Получено от сервера: {data.decode()}")

client\_socket.close()

Данный код представляет собой простого TCP-клиента, который подключается к серверу, отправляет сообщение, получает ответ от сервера и закрывает соединение. 1. Создается объект сокета client\_socket с использованием socket.socket(socket.AF\_INET, socket.SOCK\_STREAM). Это создает TCP-сокет для общения по сети. 2. Затем вызывается метод connect(('127.0.0.1', 12345)), чтобы установить соединение с сервером, который находится на адресе '[127.0.0.1](https://127.0.0.1/)' (localhost) и порту 12345. 3. Далее создается строка message, содержащая текст "Привет, сервер!". 4. Метод sendall(message.encode()) отправляет сообщение на сервер, предварительно преобразовав его из строки в байтовый формат с помощью encode(). 5. После этого клиент ждет ответа от сервера с помощью метода recv(1024), который читает до 1024 байт данных из сокета. 6. Полученные данные декодируются из байтового формата в строку с помощью data.decode() и выводятся на экран. 7. Затем клиент закрывает сокет с помощью метода close(). Этот код представляет собой простую реализацию TCP-клиента, который устанавливает соединение с сервером, отправляет сообщение, получает ответ и закрывает соединение.

Повторить асинхронную реализацию сокетов

Асинхронная реализация сокетов в Python с использованием библиотеки asyncio:

import asyncio

import socket

async def handle\_client(reader, writer):

data = await reader.read(100)

message = data.decode()

addr = writer.get\_extra\_info('peername')

print(f"Received {message} from {addr}")

print(f"Send: {message}")

writer.write(data)

await writer.drain()

print("Close the connection")

writer.close()

async def main():

server = await asyncio.start\_server(

handle\_client, '127.0.0.1', 8888)

addr = server.sockets[0].getsockname()

print(f'Serving on {addr}')

async with server:

await server.serve\_forever()

asyncio.run(main())

Объяснение:

1. Импортируем библиотеки asyncio и socket.
2. Создаем функцию **handle\_client**, которая принимает **reader** и **writer** и обрабатывает данные, полученные от клиента. При получении данных она декодирует их, получает информацию об адресе отправителя, выводит сообщение и отправляет те же данные обратно клиенту.
3. Создаем асинхронную функцию **main**, которая запускает сервер на локальном адресе (**127.0.0.1**) и порту (**8888**) с использованием **asyncio.start\_server**.
4. Печатаем информацию о том, на каком адресе запущен сервер.
5. Запускаем сервер с помощью **serve\_forever**.

Теперь давайте посмотрим на вывод этого кода при запуске. Предположим, что клиент отправляет строку "Hello, server!" на сервер. Тогда вывод будет следующим:

Serving on ('127.0.0.1', 8888)

Received Hello, server! from ('127.0.0.1', 50972)

Send: Hello, server!

Close the connection

1. Повторить работу с файловой системой (открытие, закрытие, чтение, запись файла). Техники чтения больших файлов

Для повторения работы с файловой системой в Python, вы можете использовать встроенные функции open(), read(), write() и close() для работы с файлами. Техники чтения больших файлов включают использование цикла для построчного чтения файла или использование метода read() с указанием размера блока данных. Пример открытия, чтения и записи файла:

# Открытие файла для чтения

with open("example.txt", "r") as file:

data = file.read()

print(data)

# Открытие файла для записи

with open("output.txt", "w") as file:

file.write("Пример текста для записи в файл")

BAI

1. Пример чтения файла:

# Открытие файла для чтения

with open('example.txt', 'r') as file:

data = file.read()

print(data)

1. Пример записи в файл:

# Открытие файла для записи

with open('example\_output.txt', 'w') as file:

file.write('Это пример записи в файл.')

1. Чтение больших файлов построчно:

# Чтение больших файлов построчно

file\_path = 'large\_file.txt'

with open(file\_path, 'r') as file:

for line in file:

process\_line(line)

Объяснение:

1. В первом примере мы открываем файл 'example.txt' в режиме чтения (**'r'**), считываем его содержимое с помощью **file.read()** и выводим его на экран.
2. Во втором примере мы открываем файл 'example\_output.txt' в режиме записи (**'w'**), записываем строку "Это пример записи в файл." с помощью **file.write()** и закрываем файл. Если файл с таким именем уже существует, то его содержимое будет затерто.
3. В третьем примере мы открываем большой файл 'large\_file.txt' и читаем его содержимое построчно с помощью цикла **for line in file**. Здесь **process\_line()** представляет функцию для обработки каждой строки файла.

Если файл 'example.txt' содержит текст "Пример содержимого файла.", то вывод в первом примере будет:

Пример содержимого файла.

1. Повторить обработку исключений

Для повторения обработки исключений в Python, вы можете использовать конструкцию try-except для обработки ошибок. Пример обработки исключений:

try:

result = 10 / 0

except ZeroDivisionError as e:

print(f"Ошибка деления на ноль: {e}")

except Exception as e:

print(f"Произошла ошибка: {e}")

Обработка исключений в Python позволяет управлять ошибками, которые могут возникнуть во время выполнения программы. Вот пример кода с обработкой исключений:

# Пример обработки исключений

try:

# Попытка выполнить определенный код

x = 10 / 0 # Это приведет к возникновению исключения ZeroDivisionError

except ZeroDivisionError as e:

# Обработка исключения

print(f"Произошла ошибка: {e}")

else:

# Блок else выполняется, если исключение не возникло

print("Всё прошло успешно!")

finally:

# Блок finally выполняется всегда, вне зависимости от того, возникло исключение или нет

print("Завершение работы")

Объяснение:

1. Мы использовали конструкцию **try-except** для обработки исключения. Код, который может вызвать исключение, помещается в блок **try**.
2. Если исключение произошло внутри блока **try**, выполнение этого блока прекращается, и управление передается блоку **except**, в котором происходит обработка исключения.
3. В данном примере, мы пытаемся выполнить деление на ноль (**10 / 0**), что вызовет исключение **ZeroDivisionError**. Мы его перехватываем и выводим сообщение об ошибке.
4. Блок **else** будет выполнен, если исключение не возникло.
5. Блок **finally** будет выполнен в любом случае, даже если возникло исключение.

Если запустить данный код, то вывод будет следующим:

Произошла ошибка: division by zero

Завершение работы

Таким образом, блок **finally** выполняется независимо от того, возникло исключение или нет.

4. Повторить запуск отдельных процессов и потоков

Для повторения запуска отдельных процессов и потоков в Python, вы можете использовать модули multiprocessing для процессов и threading для потоков. Пример запуска отдельных процессов:

import multiprocessing

def worker():

print("Выполнение в отдельном процессе")

process = multiprocessing.Process(target=worker)

process.start()

process.join()

Пример запуска потоков:

import threading

def worker():

print("Выполнение в отдельном потоке")

thread = threading.Thread(target=worker)

thread.start()

thread.join()

Эти примеры помогут вам повторить основные концепции работы с сокетами, файловой системой, обработкой исключений, а также запуском отдельных процессов и потоков в Python.

Этот код демонстрирует создание и запуск отдельных процессов и потоков в Python с использованием модулей multiprocessing и threading. 1. Пример с использованием multiprocessing: - Создается функция worker, которая просто выводит сообщение "Выполнение в отдельном процессе". - Создается объект process класса multiprocessing.Process, указывая в качестве цели выполнения функцию worker. - Вызывается метод start() для запуска процесса. - Метод join() блокирует основной процесс до тех пор, пока процесс process не завершится. 2. Пример с использованием threading: - Создается функция worker, которая также выводит сообщение "Выполнение в отдельном потоке". - Создается объект thread класса threading.Thread, указывая в качестве цели выполнения функцию worker. - Вызывается метод start() для запуска потока. - Метод join() также блокирует основной поток до тех пор, пока поток thread не завершится. В обоих случаях функции worker выполняются в отдельных процессах или потоках, позволяя основному процессу или потоку продолжать работу параллельно. При запуске каждого из этих примеров будет выведено сообщение "Выполнение в отдельном процессе" или "Выполнение в отдельном потоке", в зависимости от того, какой пример был запущен.