IT CAREER HUB

Python

MySQL и Python

Преподаватель

Портрет

Имя Фамилия

Текущая должность

Количество лет опыта

Какой у Вас опыт - ключевые кейсы

Самые яркие проекты

Дополнительная информация по вашему усмотрению

Корпоративный e-mail

Социальные сети (по желанию)

Важно

- Камера должна быть включена на протяжении всего занятия
- В течение занятия вопросы задавать в чате или когда преподаватель спрашивает, есть ли у Вас вопросы
- Вести себя уважительно и этично по отношению к остальным участникам занятия
- Организационные вопросы по обучению решаются с кураторами, а не на тематических занятиях
- Во время занятия будут интерактивные задания, будьте готовы включить камеру или демонстрацию экрана по просьбе преподавателя

Повторение

- Методы сравнения
- Декоратор functools.total_ordering
- Методы индексации и доступа к элементам
- Арифметические методы
- Магический метод __bool__
- Магический метод __call__

План занятия

- Работа с MySQL из Python
- Подключение к базе данных
- Работа с курсором
- Получение результатов запроса
- Параметризованные запросы
- Именованные параметры
- Обработка ошибок
- Контекстный менеджер

ICH IT CAREER HUB

ОСНОВНОЙ БЛОК

IT CAREER HUB

Работа с MySQL из Python

Большие объёмы данных

Когда программы становятся больше, появляется необходимость хранить и обрабатывать большие объёмы данных — о товарах, пользователях, сообщениях, заказах и т.д.

Использование баз данных позволяет:

- Хранить структурированные данные вне программы
- Быстро искать, обновлять и удалять записи
- Работать с очень большими объёмами данных без загрузки их всех в память
- Обеспечить многопользовательский доступ и защиту данных
- Делать программы более надёжными и масштабируемыми

Базы данных

Для **Python** есть удобные библиотеки для соединения и работы с базами данных.

ICH IT CAREER HUB

Подключение к базе данных

Работа в Python с MySQL

Нужно использовать специальную библиотеку, которая умеет:

- устанавливать соединение с сервером базы данных
- отправлять SQL-запросы и получать результаты
- обрабатывать ошибки при работе с базой

Популярные библиотеки для работы с MySQL

Библиотека	Особенности
`PyMySQL`	Простая, полностью на Python
`mysql-connector-python`	Официальная библиотека от MySQL, не всегда удобно интегрируется
`MySQLdb (mysqlclient)`	Очень быстрая, но сложнее в установке (особенно на Windows)

Установка библиотеки PyMySQL

```
pip install pymysql
```

В этой лекции мы будем использовать PyMySQL, потому что она работает стабильно на всех операционных системах

Создание соединения

Чтобы подключиться к серверу MySQL, нужно:

- Импортировать модуль `pymysql`,
- Вызвать функцию `pymysql.connect()`,
- Передать в неё параметры подключения.

Синтаксис

```
import pymysql                # импорт модуля

connection = pymysql.connect(
    host="localhost",         # адрес сервера базы данных
    user="root",              # имя пользователя
    password="yourpassword", # пароль для указанного пользователя
    database="yourdatabase", # название базы данных (необязательно)
    charset="utf8mb4",
# кодировка соединения (необязательно, указывать для поддержки русского)
)
```

Создание соединения

Особенности:

- Параметры подключения можно передавать **вручную**.
- Либо можно **хранить параметры в словаре** и **распаковать его** с помощью ** при подключении.

Пример

```
import pymysql

config = {'host': 'ich-db.edu.itcareerhub.de',
          'user': 'ich1',
          'password': 'password',
          'database': 'hr',
          }

connection = pymysql.connect(**config)  # распаковка словаря как аргументы
```

Когда удобно использовать словарь

- Когда параметры нужно передавать в разных частях программы.
- Когда подключений будет много — например, к разным базам данных.
- Когда параметры удобно хранить в файлах (`.env`, `JSON`, `YAML` и т.д.).

Проверка соединения

```
if connection.open:
    print("Connection successful!")
```

Закрытие соединения

```
if connection.open:
    print("Connection successful!")
```

Почему важно закрывать соединение

- Открытые соединения занимают ресурсы сервера базы данных.
- Если не закрывать соединения, можно быстро исчерпать лимит подключений.
- Правильная практика: **всегда закрывать соединение**, как только оно больше не нужно.

IT CAREER HUB

Работа с курсором

Работа с курсором

После подключения к базе данных необходимо создать **объект курсора**, через который:

- отправляются SQL-запросы в базу,
- получаются результаты выполнения запросов.

Создание курсора

Синтаксис

```
cursor = connection.cursor()
```

Пояснения

- `cursor()` — метод соединения (`connection`), который создаёт новый курсор для работы.
- Один `cursor` используется для выполнения одного или нескольких запросов.

Закрытие курсора

Синтаксис

```
cursor.close()
```

Пояснения

- После использования курсор нужно **закрыть вручную**, чтобы освободить ресурсы
- После закрытия курсор становится недействительным и для новых запросов нужно создать новый курсор

Выполнение запросов

Пример

```
cursor = connection.cursor()
cursor.execute("SELECT * FROM departments")
```

Пояснения

- После создания курсора можно **выполнять SQL-запросы** с помощью метода `execute()`

Особенности работы с курсором

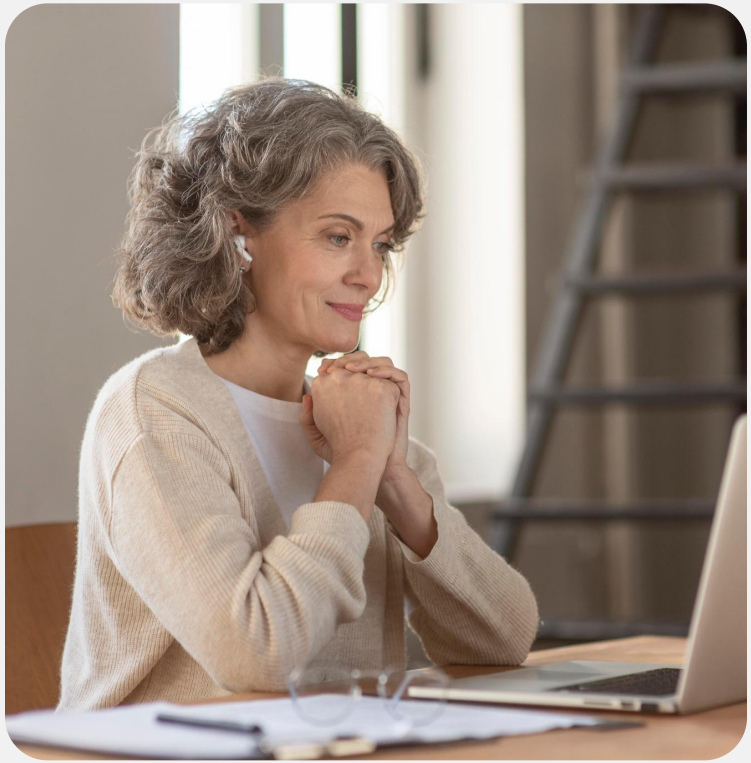
- `cursor.execute(sql)` — отправляет SQL-запрос к базе данных.
- SQL-запрос передаётся в виде строки.
- После выполнения запроса можно получить результат.
- Можно выполнять любой SQL-запрос: `SELECT`, `INSERT`, `UPDATE`, `DELETE` и др.
- После каждого нового запроса старые результаты очищаются (если результат не считать, некоторые библиотеки, например `mysql-connector`, вызовут ошибку при следующем запросе).
- При ошибке в запросе выбрасывается исключение.

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ЗАДАНИЯ

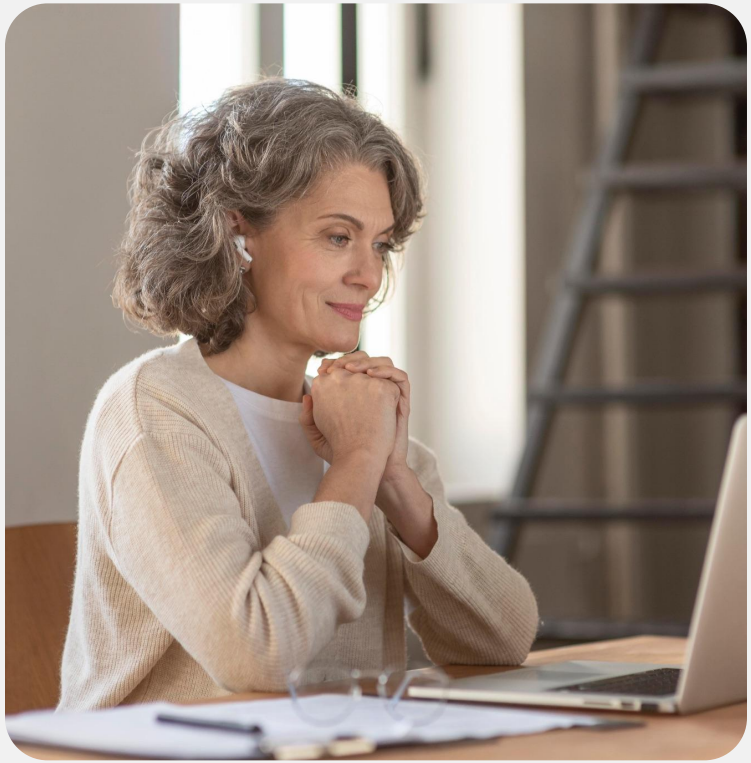


Выполните задание

Сопоставьте понятие с описанием:

1. `pymysql.connect()`
2. `cursor.execute()`
3. `connection.close()`
4. `cursor = connection.cursor()`

a. Закрывает соединение с базой данных
b. Выполняет SQL-запрос
c. Создаёт курсор для работы с базой
d. Создаёт подключение к MySQL



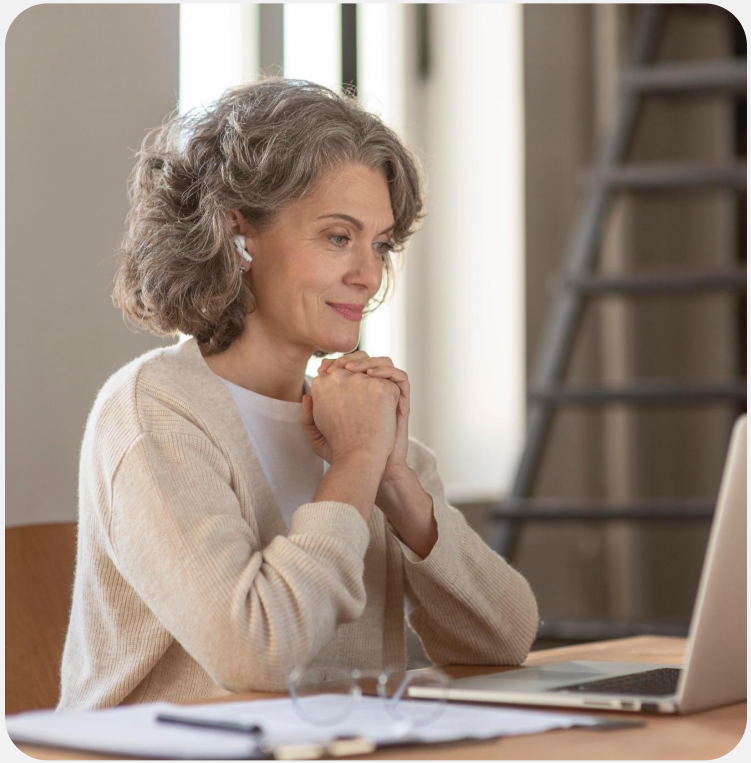
Выполните задание

Сопоставьте понятие с описанием:

1. `pymysql.connect()`
2. `cursor.execute()`
3. `connection.close()`
4. `cursor = connection.cursor()`

a. Закрывает соединение с базой данных
b. Выполняет SQL-запрос
c. Создаёт курсор для работы с базой
d. Создаёт подключение к MySQL

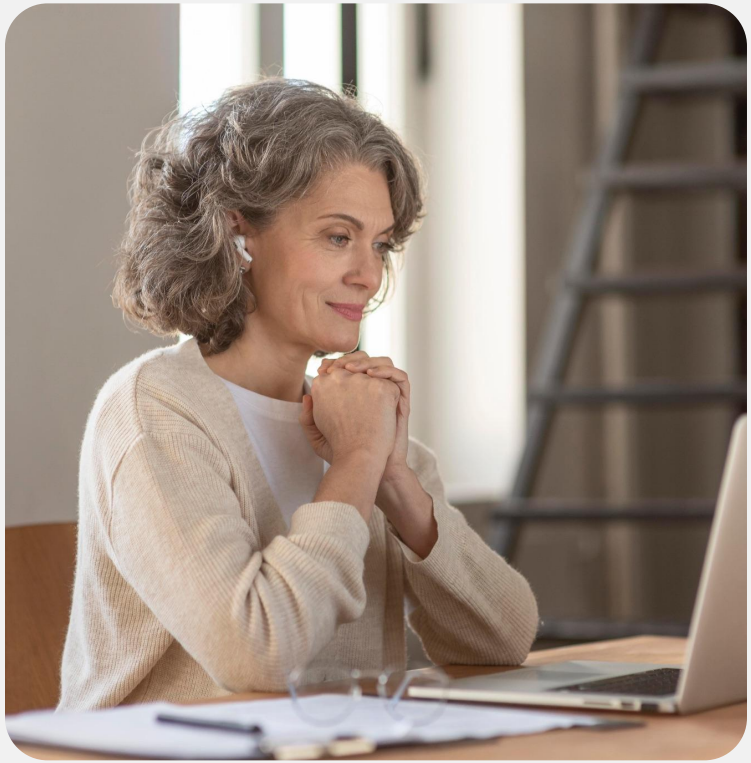
Ответ: 1-d, 2-b, 3-a, 4-c



Выберите верные варианты ответа

Выберите корректные утверждения или несколько:

1. `cursor.execute()` можно вызывать до создания соединения
2. Курсор автоматически закрывается после каждого запроса
3. Один курсор можно использовать для нескольких запросов
4. `pymysql.connect()` возвращает объект курсора
5. `connection.open` проверяет статус соединения



Выберите верные варианты ответа

Выберите корректные утверждения или несколько:

a. `cursor.execute()` можно вызывать до создания соединения
b. Курсор автоматически закрывается после каждого запроса
c. Один курсор можно использовать для нескольких запросов
d. `pymysql.connect()` возвращает объект курсора
e. `connection.open` проверяет статус соединения

IT CAREER HUB

Получение результатов запроса

Важно

После выполнения запроса SELECT данные остаются в курсоре. Чтобы получить их, используются специальные методы.

Методы получения данных

Метод	Что делает
`fetchone()`	Получает **одну следующую строку**
`fetchall()`	Получает **все строки сразу**
`fetchmany(size)`	Получает **ограниченное число строк**

Выполнение запросов

Пример

```
cursor.execute("SELECT * FROM employees")

for row in cursor:
    print(row)
```

Пояснения

- После выполнения запроса SELECT данные остаются в курсоре, пока не будут получены. Для их получения курсор можно использовать **как итератор** в цикле
- Каждая итерация даёт **одну строку результата** (в виде кортежа)
- Все строки читаются **последовательно**, до окончания выборки

Пример

```
cursor.execute("SELECT * FROM departments")

row = cursor.fetchone()   # получить одну строку
print("One row:")
print("\t", row)

print("Five rows:")
rows_5 = cursor.fetchmany(5)   # получить указанное кол-во строк
for r in rows_5:
    print("\t", r)

print("All rows:")
rows = cursor.fetchall()  # получить все строки
for r in rows:
    print("\t", r)
```

Особенности

- После того как строки считаны, повторный вызов `fetchone()` вернёт `None`, а `fetchmany()`, `fetchall()` - пустую коллекцию.
- `fetchmany(size)` позволяет считывать данные порциями, чтобы **не перегружать память** на больших выборках.
- **Данные не загружаются заранее** — они извлекаются **только при вызове методов получения**. Это позволяет обрабатывать **большие запросы без нагрузки на память**.
- **Курсор — это итератор**: каждый вызов чтения сдвигает "указатель", поэтому **повторный вызов `"fetch"` методов будет читать только оставшиеся строки**, а не начинать с начала.

Выполнение запросов

Пример

```
cursor.execute("SELECT * FROM employees")

for row in cursor:
    print(row)
```

Пояснения

- После выполнения SELECT-запроса, курсор можно использовать как итератор в цикле — это удобно и эффективно, особенно для больших выборок
- Каждая итерация даёт **одну строку результата** (в виде кортежа)
- Это **эквивалентно повторным вызовам** `fetchone()`, но удобнее и компактнее
- Все строки читаются **последовательно**, до окончания выборки

IT CAREER HUB

Параметризованные запросы

Важно

При выполнении запросов в базу данных **никогда нельзя подставлять значения напрямую в строку SQL**. Это может привести к **SQL-инъекциям**.

Полезно знать

ICH
IT CAREER HUB

SQL-инъекция

Это тип уязвимости, при котором злоумышленник может вставить произвольный SQL-код в запрос, чтобы получить несанкционированный доступ к данным, изменить или удалить информацию в базе данных

Пример

```
# Прямое включение пользовательского ввода
user_input = "1 OR 1=1"
sql = f"SELECT * FROM employees WHERE employee_id = {user_input}"
cursor.execute(sql)  # выполнит: SELECT * FROM employees WHERE employee_id = 1 OR 1=1
# Получены все записи, вместо одной
for r in cursor:
    print(r)
```

Использование параметризованных запросов

```
cursor.execute(
    "SELECT * FROM table_name WHERE column1 = %s AND column2 > %s",
    (value1, value2)
)
```

- Первый аргумент — SQL-запрос с плейсхолдерами %s
- Второй аргумент — **последовательность значений** (кортеж или список), которые подставляются вместо %s

Пример: несколько значений

```
cursor.execute(
    "SELECT * FROM employees WHERE department_id = %s OR salary > %s",
    (60, 20000)
)
for r in cursor:
    print(r)
```

- Подставляются оба значения в соответствующие %s.
- Порядок значений в последовательности должен совпадать с порядком %s в запросе.

Пример: одно значение

```
cursor.execute(
    "SELECT * FROM employees WHERE department_id = %s",
    (100,)  # Обязательно запятая!
)
for r in cursor:
    print(r)
```

- Если передаётся только одно значение, нужно ставить запятую после него, чтобы создать кортеж.

Преимущества параметризованных запросов

- **Безопасность** — защита от SQL-инъекций
- **Универсальность** — одно и то же выражение можно выполнять с разными данными
- **Автоматическое экранирование** — библиотека сама правильно обрабатывает типы данных (строки, числа и т.д.)
- **Улучшение производительности** — сервер базы данных может кэшировать план выполнения запроса

Именованные параметры

Вместо позиционных %s, в запросах можно использовать **именованные параметры**. Это делает код **более читабельным**, особенно когда передаётся много значений.

Именованные параметры

```
cursor.execute(
    "SELECT * FROM table_name WHERE column1 = %(param1)s AND column2 > %(param2)s",
    {"param1": value1, "param2": value2}
)
```

- В SQL-запросе используются плейсхолдеры вида `%(name)s`
- Вторым аргументом передаётся словарь, где ключи соответствуют плейсхолдерам
- Порядок ключей в словаре не важен — главное, чтобы ключи словаря совпадали с именами плейсхолдеров в запросе

Пример

```
cursor.execute(
    "SELECT * FROM employees WHERE department_id = %(dep_id)s OR salary > %(min_salary)s",
    {"min_salary": 20000, "dep_id": 60}
)
for r in cursor:
    print(r)
```

Преимущества

- Повышается читаемость запроса
- Удобно, когда значения приходят как словарь
- Легче избежать ошибок с порядком аргументов

IT CAREER HUB

Обработка ошибок

Важно

Чтобы программа не завершалась аварийно, ошибки нужно обрабатывать с помощью конструкции

try ... except.

Пример: ошибка при подключении

```
import pymysql

try:
    connection = pymysql.connect(
        host="ich-db.edu.itcareerhub.de",
        user="root",
        password="wrong_password",  # неправильный пароль
        database="test"
    )
    print("Connected!")
except pymysql.MySQLError as e:
    print("Connection error:", e)
```

Пример: ошибка при запросе

```
try:
    with connection.cursor() as cursor:
        cursor.execute("SELECT * FROM non_existing_table")  # таблицы не существует
        result = cursor.fetchall()
except pymysql.MySQLError as e:
    print("Query error:", e)
```

Особенности

- Базовый класс всех исключений: `pymysql.MySQLError`
- Можно также обрабатывать конкретные ошибки, например `pymysql.ProgrammingError`, `pymysql.OperationalError`

ICH IT CAREER HUB

Контекстный менеджер with

Важно

Чтобы упростить работу с соединением и курсором, можно использовать конструкцию `with`, которая **автоматически закрывает** ресурсы даже при ошибках.

Это делает код **чище** и **защищает от утечек соединений**.

Пример

```
with pymysql.connect(
    host='ich-db.edu.itcareerhub.de',
    user='ich1',
    password='password',
    database='hr'
) as connection:  # автоматически закроет connection
    with connection.cursor() as cursor:  # автоматически закроет cursor
        cursor.execute("SELECT * FROM employees")
        for row in cursor:
            print(row)
```

Особенности

- Блок `with` автоматически вызывает `close()` для курсора и/или соединения
- Код становится компактнее и безопаснее
- Даже если внутри блока произойдёт ошибка — ресурсы **корректно закроются**

Пример

1

```
from functools import total_ordering

@total_ordering
class Book:
    def __init__(self, title):
        self.title = title

    def __eq__(self, other):
      if not isinstance(other, Book):
          return NotImplemented
      return self.title == other.title

    def __lt__(self, other):
        return self.title < other.title
```

2

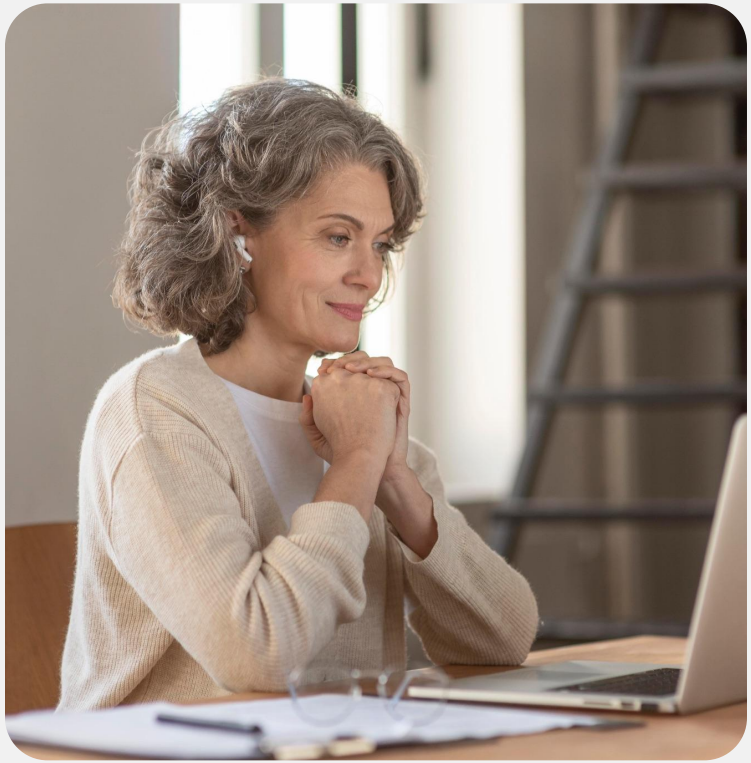
```
b1 = Book("1984")
b2 = Book("Brave New World")

print(b1 < b2)    # < реализовано вручную
print(b1 >= b2)   # >= создано
автоматически
```

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

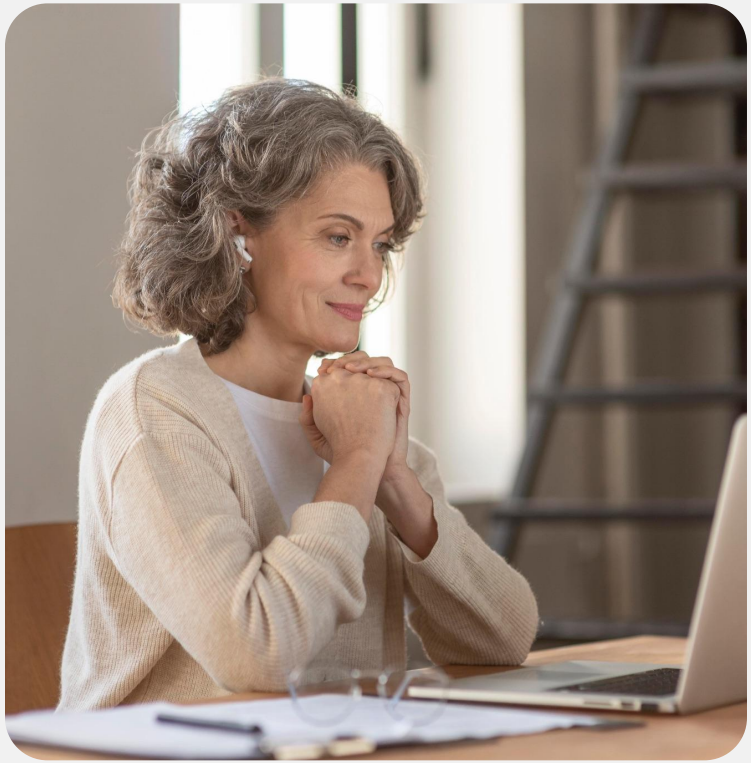
ЗАДАНИЯ



Выберите верные варианты ответа

Зачем использовать параметризованные запросы?

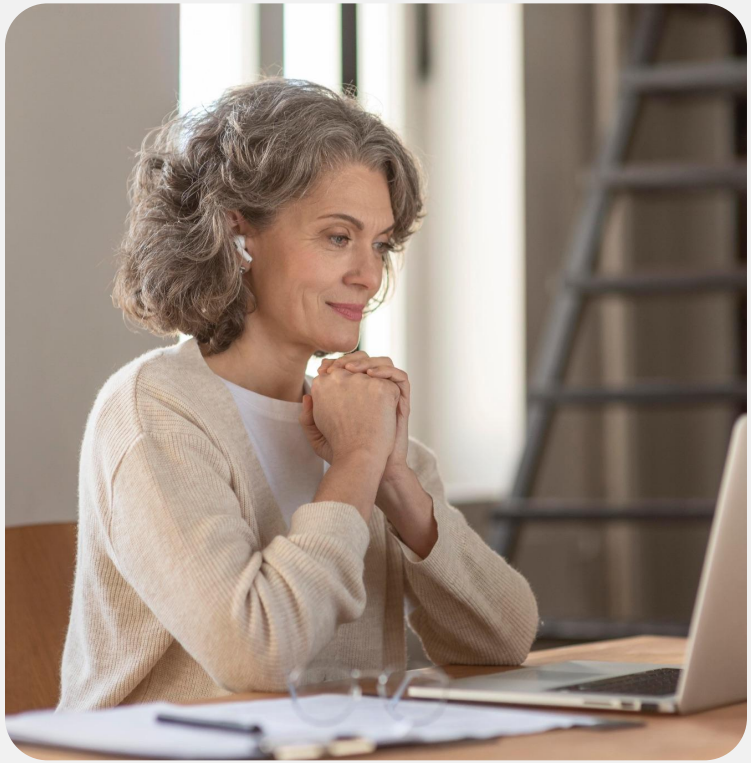
a. Для сокращения кода
b. Для защиты от SQL-инъекций
c. Для правильной подстановки значений и экранирования
d. Для красивого синтаксиса



Выберите верные варианты ответа

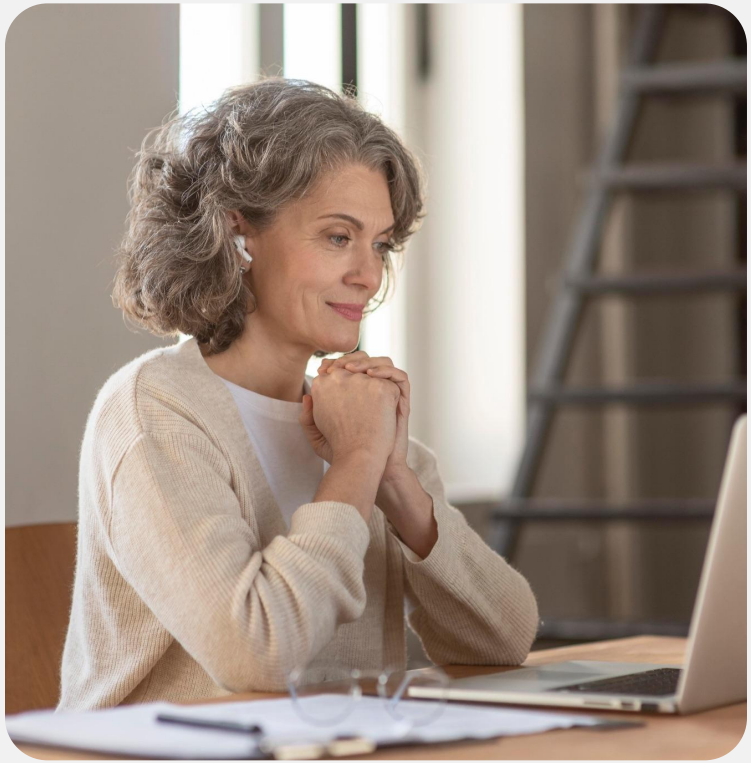
Зачем использовать параметризованные запросы?

a. Для сокращения кода
b. Для защиты от SQL-инъекций
c. Для правильной подстановки значений и экранирования
d. Для красивого синтаксиса



Ответьте на вопрос

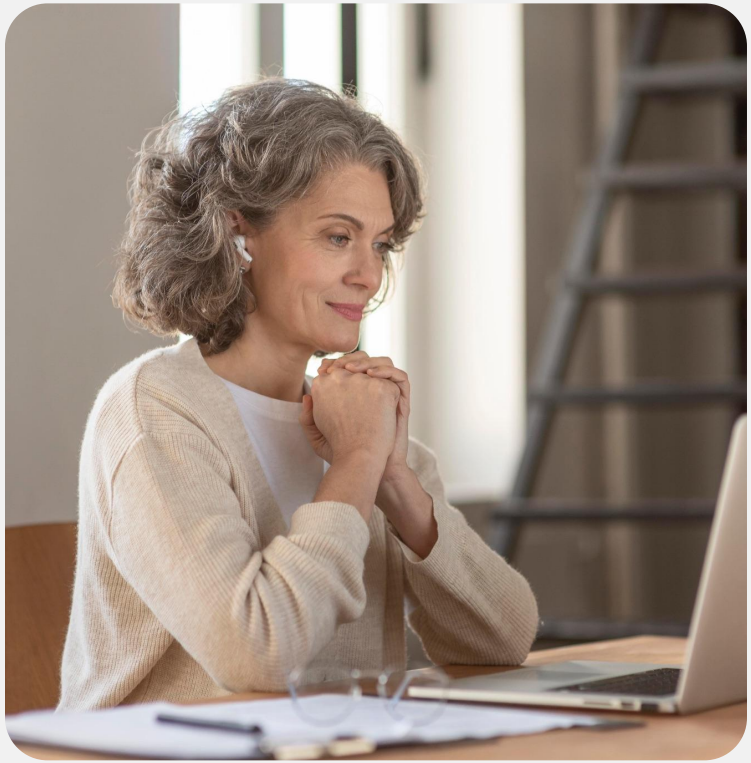
Что произойдёт, если снова вызвать fetchall() после того, как раньше были считаны все строки из запроса?



Ответьте на вопрос

Что произойдёт, если снова вызвать fetchall() после того, как раньше были считаны все строки из запроса?

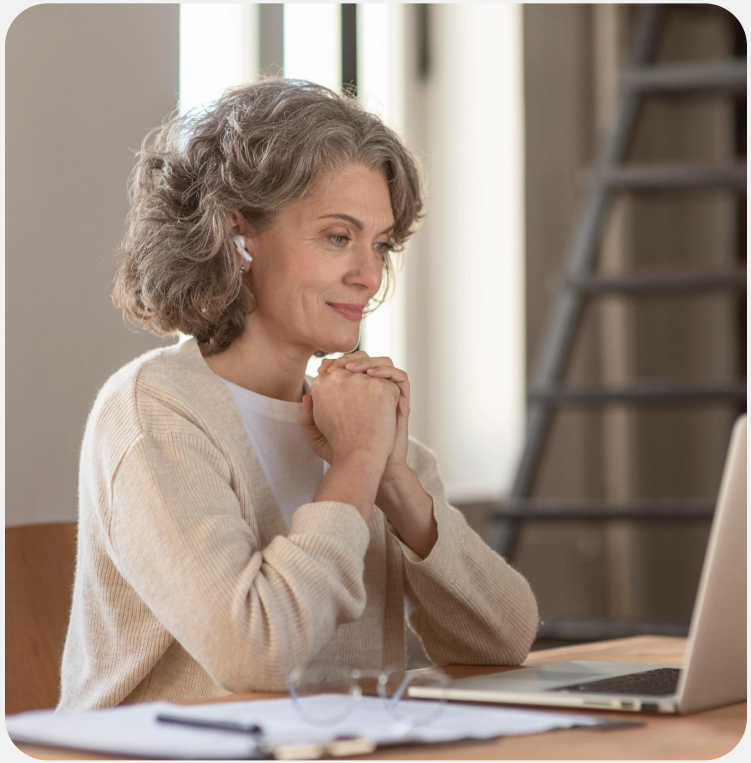
Ответ: Будет возвращена пустая коллекция



Выберите верные варианты ответа

Какие утверждения верны для конструкции с with?

a. Соединение автоматически закрывается после блока `with`
b. Внутри него можно использовать `with connection.cursor()`
c. Он помогает автоматически закрыть соединение даже при ошибке
d. Его можно использовать только для SELECT



Выберите верные варианты ответа

Какие утверждения верны для конструкции с `with`?

a. Соединение автоматически закрывается после блока `with`
b. Внутри него можно использовать `with connection.cursor()`
c. Он помогает автоматически закрыть соединение даже при ошибке
d. Его можно использовать только для SELECT

IT CAREER HUB

ВОПРОСЫ

IT CAREER HUB

ДОМАШНЕЕ ЗАДАНИЕ

Домашнее задание

1. Список всех стран

Используя базу данных `world`, выведи **названия всех стран** из таблицы `country`.
Каждое название должно отображаться с новой строки и иметь номер.

Пример вывода:

```
1. Aruba
2. Afghanistan
3. Angola
...
239. Zimbabwe
```

Домашнее задание

2. Города выбранной страны

Добавьте к предыдущей программе возможность выбора страны. Пользователь должен ввести название страны. Далее выведите **все города этой страны и их численность населения**.

Пример вывода:

```
Введите страну: Germany
Berlin – 3386667
Hamburg – 1704735
Munich [München] – 1194560
...
```

Заключение