

Мини-учебник по основам разработки чат-ботов на Python

1. Переменные (variables)

Переменная — это именованная область памяти, в которую можно сохранять данные (числа, строки и т. д.).

В Python переменные не требуют явного объявления типа. Это язык с динамической типизацией.

Пример

```
# Присваиваем переменным начальные значения
age = 25                # Целое число
price = 19.99           # Число с плавающей точкой
name = "Alice"          # Строка
is_student = True       # Логическая переменная

print(age, price, name, is_student)
```

Особенности

1. Имена переменных должны начинаться с буквы (a–z, A–Z) или символа подчёркивания (_). Название переменной может содержать цифры, но не должно с них начинаться.
2. Python чувствителен к регистру, поэтому `age` и `Age` — это разные переменные.

2. Условия (if, else, elif)

Условия в Python позволяют выполнять код, если истинно какое-то логическое выражение. Ключевые слова: `if`, `elif` (сокращение от *else if*), `else`.

Пример

```
age = 20

if age < 18:
    print("Вы слишком молоды, доступ запрещён.")
elif age < 21:
    print("Ограниченный доступ.")
else:
    print("Доступ разрешён.")
```

- `if age < 18:` — проверяем условие `age < 18`.
- `elif age < 21:` — это дополнительная проверка, если предыдущее условие не выполнилось.
- `else:` — будет выполнено, если все предыдущие условия оказались ложными.

3. Циклы (`for`, `while`)

Цикл `for`

В Python цикл `for` перебирает элементы итерируемого объекта (например, списка или диапазона чисел).

Пример

```
numbers = [10, 20, 30, 40, 50]

for num in numbers:
    print(num)
```

Ещё один пример, с использованием `range()` для генерации последовательности чисел:

```
for i in range(5): # range(5) создаёт числа от 0 до 4
    print("Итерация номер:", i)
```

Цикл `while`

Цикл `while` повторяется, пока некоторое условие остаётся истинным:

```
count = 0

while count < 5:
    print("Счётчик =", count)
    count += 1 # аналог count = count + 1
```

Будьте аккуратны с циклом `while`, чтобы не создавать бесконечные циклы. Всегда нужно следить, чтобы в какой-то момент условие стало ложным внутри цикла.

4. Словари (`dict`)

Словарь (dictionary) в Python — это структура данных, которая хранит пары **ключ: значение**.

Ключ может быть почти любым неизменяемым типом (строка, число), а значение — любым типом данных.

Пример создания словаря

```
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

# Обращение к значению по ключу:
print(person["name"]) # Выведет: Alice

# Добавление новой пары ключ: значение:
person["email"] = "alice@example.com"

# Перебор всех ключей:
for key in person:
    print(key, "->", person[key])
```

Часто используют методы:

- `dict.keys()` — возвращает все ключи словаря;
- `dict.values()` — возвращает все значения;
- `dict.items()` — возвращает список пар (ключ, значение).

5. Основы работы с python-telegram-bot

Установка

Чтобы установить библиотеку, используйте `pip` (командная строка / терминал):

bash

```
pip install python-telegram-bot
```

Простой бот «Эхо» (echo-bot)

Приведём пример простого бота, который будет отвечать пользователю тем же текстом, который тот отправил.



Узнать подробнее про функции в Python можно в [курсе Академии ИИ](#).

```
from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler,
MessageHandler, filters, ContextTypes
import asyncio

# Функция обработчика команды /start
async def start_command(update: Update, context: ContextTypes.
DEFAULT_TYPE):
    await update.message.reply_text("Привет! Я эхо-бот. Напиши
мне что-нибудь, и я повторю.")

# Функция для обработки входящих сообщений и эхо-ответа
async def echo_message(update: Update, context: ContextTypes.
DEFAULT_TYPE):
    text_received = update.message.text
    await update.message.reply_text(text_received)

async def main():
    # Замените YOUR_TOKEN_HERE на свой реальный токен
    app = ApplicationBuilder().token("YOUR_TOKEN_HERE").build()

    # Добавляем обработчик команды /start
    app.add_handler(CommandHandler("start", start_command))

    # Добавляем обработчик всех текстовых сообщений
    app.add_handler(MessageHandler(filters.TEXT & ~filters.
COMMAND, echo_message))

    # Запускаем бота
    await app.run_polling()

if __name__ == "__main__":
    asyncio.run(main())
```

Пояснения по коду

1. `ApplicationBuilder` создаёт экземпляр приложения бота.
2. `CommandHandler("start", start_command)` регистрирует функцию `start_command` как обработчик для команды `/start`.
3. `MessageHandler(filters.TEXT & ~filters.COMMAND, echo_message)` перехватывает все сообщения, которые не являются командами, и вызывает `echo_message`.

4. `await app.run_polling()` запускает «бесконечный» цикл, в котором бот проверяет новые сообщения и отвечает на них.

6. Inline-клавиатуры (InlineKeyboards)

Inline-клавиатура — это набор кнопок, которые встроены в сообщение бота. При нажатии на кнопку Telegram отправляет боту специальное событие — `callback query`, и бот может ответить любым способом (редактировать сообщение, отправить новое и т. д.).

Пример с одной кнопкой

```
from telegram import InlineKeyboardButton,
InlineKeyboardMarkup, Update
from telegram.ext import (
    ApplicationBuilder,
    CommandHandler,
    MessageHandler,
    CallbackQueryHandler,
    ContextTypes,
    filters
)

async def start_command(update: Update, context: ContextTypes.
DEFAULT_TYPE):
    # Создаём кнопку, которая при нажатии отправляет
    callback_data="button_clicked"
    button = InlineKeyboardButton(text="Нажми меня",
    callback_data="button_clicked")

    # Создаём Inline-разметку из этой кнопки
    keyboard = InlineKeyboardMarkup([[button]])

    # Отправляем сообщение с этой клавиатурой
    await update.message.reply_text(
        "Привет! Это пример Inline-кнопки:",
        reply_markup=keyboard

# Обработчик "callback query" — события нажатия
на Inline-кнопку
async def button_handler(update: Update, context: ContextTypes.
DEFAULT_TYPE):
    query = update.callback_query
    await query.answer() # Обязательный вызов для Telegram,
чтобы закрыть запрос
```

```
# Допустим, нам пришла callback_data="button_clicked"
if query.data == "button_clicked":
    await query.message.reply_text("Кнопка нажата!")

async def echo_message(update: Update, context: ContextTypes.
DEFAULT_TYPE):
    text_received = update.message.text
    await update.message.reply_text(f"Эхо: {text_received}")

async def main():
    app = ApplicationBuilder().token("YOUR_TOKEN_HERE").build()

    app.add_handler(CommandHandler("start", start_command))
    # Обработчик нажатия на Inline-кнопку
    app.add_handler(CallbackQueryHandler(button_handler))
    # Обработчик обычных текстовых сообщений

    app.add_handler(MessageHandler(filters.TEXT & ~filters.COMMAND,
echo_message))

    await app.run_polling()

if __name__ == "__main__":
    import asyncio

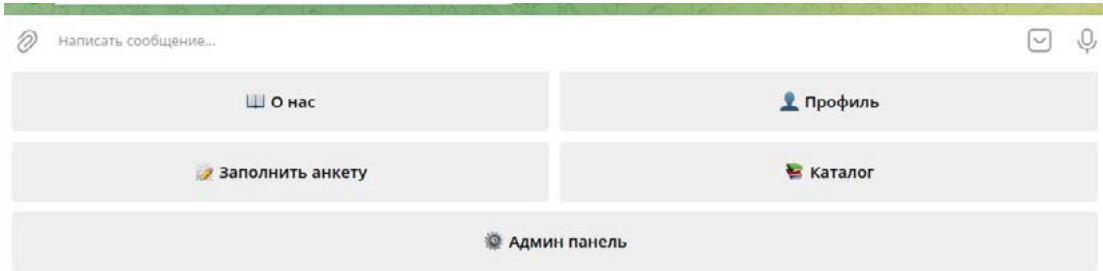
    asyncio.run(main())
```

Пояснения

1. **InlineKeyboardButton** — [класс](#) для создания кнопок.
 - **text** — текст, отображаемый на кнопке.
 - **callback_data** — данные, которые получит бот при нажатии на кнопку. Обычно это строка вида **"button_clicked"** (**кнопка нажата**).
О классах в Python подробнее можно узнать на [курсе Академии ИИ](#).
2. **InlineKeyboardMarkup** — класс, который объединяет кнопки в разметку. В примере используется список списков, что позволяет располагать кнопки в «ряды».
3. **CallbackQueryHandler** — специальный обработчик, который «слушает» события нажатий на Inline-кнопки.
4. **update.callback_query** — объект события нажатия на кнопку.
 - **query.answer()** — важно вызывать, чтобы Telegram знал, что событие обработано, иначе кнопка будет находиться в состоянии ожидания.
 - **query.data** — данные (**callback_data**), которые были указаны при создании кнопки.

7. Reply-клавиатуры (ReplyKeyboards)

Reply-клавиатура — это набор кнопок, который появляется в области ввода текста аналогично клавиатуре телефона. При нажатии на такую кнопку Telegram отправляет текст (или контакт, локацию и т. д.) в чат с ботом.



Пример Reply-клавиатуры

```
from telegram import (
    ReplyKeyboardMarkup,
    KeyboardButton,
    Update
)

async def start_command(update: Update, context: ContextTypes.
    DEFAULT_TYPE):
    # Создаём обычные кнопки
    button1 = KeyboardButton("Привет!")
    button2 = KeyboardButton("Как дела?")

    # Можно создавать несколько рядов кнопок
    keyboard = [
        [button1, button2],

        [KeyboardButton("Отправить контакт", request_contact=True)],

        [KeyboardButton("Отправить локацию", request_location=True)]
    ]

    # Создаём разметку для Reply-клавиатуры
    reply_markup = ReplyKeyboardMarkup(keyboard,
        resize_keyboard=True)

    await update.message.reply_text(
        "Пример Reply-клавиатуры:",
        reply_markup=reply_markup
    )
```

```
async def echo_message(update: Update, context: ContextTypes.  
DEFAULT_TYPE):  
    # Просто повторяем отправленный текст  
    text_received = update.message.text  
  
    await update.message.reply_text(f"Вы ввели: {text_received}")  
  
async def main():  
    app = ApplicationBuilder().token("YOUR_TOKEN_HERE").build()  
  
    app.add_handler(CommandHandler("start", start_command))  
    app.add_handler(MessageHandler(filters.TEXT & ~filters.  
COMMAND, echo_message))  
  
    await app.run_polling()  
  
if __name__ == "__main__":  
    import asyncio  
  
    asyncio.run(main())
```

Пояснения

1. **ReplyKeyboardMarkup** — класс, который объединяет список кнопок (list[list[KeyboardButton]]).
2. **KeyboardButton** — класс кнопки. Можно задавать:
 - **text** — текст кнопки,
 - **request_contact=True** — при нажатии пользователь сразу отправит свой контакт,
 - **request_location=True** — при нажатии пользователь отправит свою геолокацию.
3. Параметр **resize_keyboard=True** заставляет Telegram учесть количество кнопок и сделать клавиатуру более компактной.

Важный момент: для того чтобы пользователь мог отправлять контакт или локацию, нужно, чтобы он дал соответствующие разрешения Telegram. Это обычно работает на мобильных устройствах.

Ниже приведён пример, как можно удалить все кнопки (свести Reply-клавиатуру на нет). Для этого в **python-telegram-bot** используется специальная «разметка» — **ReplyKeyboardRemove**, которая убирает клавиатуру у пользователя.

```
from telegram import ReplyKeyboardRemove
from telegram.ext import CommandHandler, ContextTypes
from telegram import Update

async def remove_keyboard_command(update: Update, context:
ContextTypes.DEFAULT_TYPE):
    # При вызове этой команды удаляем все кнопки
    (Reply-клавиатуру)
    await update.message.reply_text(
        "Клавиатура убрана!",
        reply_markup=ReplyKeyboardRemove()
    )
```

Примерный сценарий

1. По команде `/start` мы показываем две кнопки «Привет!» и «Как дела?».
2. По команде `/remove` (или любой другой) убираем всю клавиатуру, отправив сообщение с `reply_markup=ReplyKeyboardRemove()`.

Пример кода целиком

```
from telegram import (
    ReplyKeyboardMarkup,
    KeyboardButton,
    ReplyKeyboardRemove,
    Update
)

from telegram.ext import (
    ApplicationBuilder,
    CommandHandler,
    ContextTypes
)

# Команда /start: показать клавиатуру с двумя кнопками
async def start_command(update: Update, context: ContextTypes.
DEFAULT_TYPE):
    keyboard = [
        [KeyboardButton("Привет!"),
         KeyboardButton("Как дела?")]
    ]
```

```
reply_markup = ReplyKeyboardMarkup(keyboard, resize_
keyboard=True)

    await update.message.reply_text(
        "Привет! Нажми на кнопку:",
        reply_markup=reply_markup
    )

# Команда /remove: убрать все кнопки
async def remove_keyboard_command(update: Update, context:
ContextTypes.DEFAULT_TYPE):
    await update.message.reply_text(
        "Клавиатура убрана!",
        reply_markup=ReplyKeyboardRemove()
    )

async def main():
    app = ApplicationBuilder().token("YOUR_BOT_TOKEN").build()

    app.add_handler(CommandHandler("start", start_command))

    app.
add_handler(CommandHandler("remove", remove_keyboard_
command))

    await app.run_polling()

if __name__ == "__main__":
    import asyncio

    asyncio.run(main())
```

Теперь:

- Когда пользователь вводит `/start`, бот присылает сообщение с двумя кнопками (Reply-клавиатура).
- Когда пользователь вводит `/remove`, бот отправляет сообщение с `ReplyKeyboardRemove()` и все кнопки пропадают из области ввода.

Таким образом, вы контролируете показ или скрытие Reply-клавиатуры, используя разные команды и нужные варианты `reply_markup`.

8. Работа со стикерами

Отправка стикеров в Telegram-боте выполняется через метод `send_sticker()` (или асинхронный метод `reply_sticker()`, если используются последние версии библиотеки с асинхронными функциями).

Отправляем стикер в ответ на сообщение

Ниже пример того, как бот может реагировать на текст пользователя и отправлять ему стикер.

Для этого необходимо знать `file_id` стикера или иметь ссылку на стикер.

```
from telegram import Update
from telegram.ext import (
    ApplicationBuilder,
    MessageHandler,
    ContextTypes,
    filters
)

# Пример file_id (у каждого стикера он свой)
MY_STICKER_FILE_ID = "CAACAgIAAxkBAAEE..."

async def sticker_reply(update: Update, context: ContextTypes.
    DEFAULT_TYPE):
    # Отправляем стикер в чат

    await
    update.message.reply_sticker(sticker=MY_STICKER_FILE_ID)

async def main():
    app = ApplicationBuilder().token("YOUR_TOKEN_HERE").build()

    # Допустим, хотим на любое текстовое сообщение отправлять
    # стикер:
    app.add_handler(MessageHandler(filters.TEXT & ~filters.
        COMMAND, sticker_reply))

    await app.run_polling()

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Как получить `file_id` стикера?

1. Отправьте боту в личном сообщении любой стикер.
2. Включите у своего бота режим получения «полного» Update. Самый простой способ — это временно сделать так, чтобы ваш бот отвечал на все входящие обновления и печатал их в консоль.
3. В объекте `update.message.sticker` вы найдете поле `file_id`. Запомните его, чтобы использовать для повторной отправки.

Также вы можете использовать дебаг-бота (пример такого бота: [@LeadConverterToolkitBot](#)) или другие подобные инструменты, чтобы быстро получить `file_id`.

9. Работа с SQLite в Python

Что такое SQLite?

SQLite — это встраиваемая реляционная база данных, которая хранится в одном файле (или в оперативной памяти, если так настроить). В отличие от «больших» СУБД (PostgreSQL, MySQL, MS SQL), для SQLite не нужно устанавливать отдельный сервер — всё упаковано в одну библиотеку, и управление базой может вестись напрямую.

Основные преимущества

1. **Лёгкость и простота:** не требует отдельного сервера БД.
2. **Файловое хранение:** всё содержимое базы данных помещается в один файл `.db`.
3. **Отсутствие сложностей с конфигурацией:** достаточно просто подключиться к файлу.
4. **Поддержка стандартного SQL.**

Это делает SQLite удобным выбором для небольших проектов, прототипирования, локального хранения данных или кэширования, а также для обучения SQL.

Подключение из Python

В Python есть встроенный модуль `sqlite3`, который не требует дополнительной установки.

Пример

```
import sqlite3

# Подключаемся к файлу базы данных mydatabase.db
# Если файла не существует, он будет создан автоматически
conn = sqlite3.connect(mydatabase.db)

# Создаём объект cursor для взаимодействия с базой
cursor = conn.cursor()

# Выполнив все операции, не забудьте закрыть соединение
conn.close()
```

Если вы хотите создать базу данных только в оперативной памяти (без файла), можно использовать:

```
sqlite3.connect(":memory:")
```

После остановки программы такая БД будет удалена.

Создание таблицы (CREATE)

SQL-запрос для создания таблицы с тремя столбцами («Номер», «Имя» и «Возраст») выглядит так:

sql

```
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER
);
```

В Python SQL-запрос встраивается методом `cursor.execute(...)`:

```
import sqlite3

conn = sqlite3.connect(mydatabase.db)
cursor = conn.cursor()

# Создаём таблицу users
cursor.execute("""
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        age INTEGER
    );
""")
```

```
# Сохраняем изменения (коммитим транзакцию)
conn.commit()
conn.close()
```

Добавление записей (INSERT)

Чтобы добавить новые данные в таблицу, можно действовать так:

```
import sqlite3

conn = sqlite3.connect("mydatabase.db")
cursor = conn.cursor()

# Пример добавления одной записи
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)",
               ("Alice", 25))

# Пример добавления нескольких записей сразу (executemany)
people = [
    ("Bob", 30),
    ("Charlie", 22),
    ("Diana", 27)
]
cursor.executemany("INSERT INTO users (name, age) VALUES (?, ?)",
                  people)

conn.commit()
conn.close()
```

Важно! Обратите внимание на использование знаков вопроса (?) для передачи параметров. Это позволяет защититься от [SQL-инъекций](#) и следует рекомендованному стилю работы.

Чтение данных (SELECT)

Для получения данных из таблицы используют запросы **SELECT**. Результаты можно итерировать в Python.


```
import sqlite3

conn = sqlite3.connect("mydatabase.db")
cursor = conn.cursor()

cursor.execute("SELECT id, name, age FROM users")
rows = cursor.fetchall() # Забираем все строки результата
в переменную rows

for row in rows:
    print(row)
    # row — это кортеж (id, name, age), например (1, Alice, 25)

conn.close()
```

Обновление данных (UPDATE)

Чтобы изменить существующие записи, выполняем запрос **UPDATE**:

```
import sqlite3

conn = sqlite3.connect("mydatabase.db")
cursor = conn.cursor()

# Увеличим возраст пользователя по имени "Alice"
cursor.execute("UPDATE users SET age = ? WHERE name = ?", (26,
Alice))

conn.commit()
conn.close()
```

Удаление данных (DELETE)

Аналогично, удаляем записи:

```
import sqlite3

conn = sqlite3.connect("mydatabase.db")
cursor = conn.cursor()

# Удаляем все записи, где возраст меньше 23
cursor.execute("DELETE FROM users WHERE age < ?", (23,))

conn.commit()
conn.close()
```

Обработка ошибок (try/except/finally)

При работе с файлами или сетевыми ресурсами (а также с базами данных) часто стоит использовать **try/except/finally**, чтобы гарантированно закрыть соединение, даже если возникнет ошибка:

```
import sqlite3

try:
    conn = sqlite3.connect("mydatabase.db")
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM users")
    rows = cursor.fetchall()

    for row in rows:
        print(row)

except sqlite3.Error as e:
    print("Ошибка при работе с SQLite:", e)

finally:
    if conn:
        conn.close()
```

Использование контекстного менеджера with

Более характерный для Python способ – использовать **with**, который автоматически закроет ресурс:

```
import sqlite3

with sqlite3.connect("mydatabase.db") as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM users")
    rows = cursor.fetchall()
    print(rows)
```

Как только блок **with** завершится, соединение будет закрыто.

Подробнее о контекстном менеджере **with** можно узнать [из документации Python](#).

Дополнительно. Работа с файлом `.env` (хранение конфиденциальных данных и переменных окружения)

Очень часто при разработке приложений, в том числе Telegram-ботов, возникает необходимость хранить конфиденциальные данные: токены, пароли и другие секреты, которые не должны попадать в открытые репозитории (например, на GitHub).

Чтобы не «зашивать» такие секреты прямо в код, используют переменные окружения (*environment variables*) и файлы `.env`.

Что такое файл `.env`?

Файл с расширением `.env` — это обычный текстовый файл, где каждая строка содержит переменную окружения в формате:

`ПЕРЕМЕННАЯ=значение`

Пример файла `.env`

ini

`TOKEN=123456:ABC-DEF1234ghIk1-zyx57W2v1u123ew11`

`DEBUG=True`

`DB_PASSWORD=MySuperSecret`

Обычно файл `.env` не добавляется в систему контроля версий (например, в `.gitignore`), чтобы никто случайно не показал пароли и токены.

Как подключить `.env` в Python?

Для удобной работы с `.env` в Python часто используют библиотеку [python-dotenv](#).

Установка

bash

`pip install python-dotenv`

Пример кода

```
import os
from dotenv import load_dotenv

# Загружаем переменные из файла .env
load_dotenv()
```

```
# Теперь можем получать значения переменных
token = os.getenv("TOKEN")          # Возвращает строку или None,
если переменная не найдена
debug_mode = os.getenv("DEBUG")    # "True" или "False", в зави-
симости от значения в .env
db_password = os.getenv("DB_PASSWORD")

print("Токен бота:", token)
print("Режим отладки:", debug_mode)
print("Пароль от БД:", db_password)
```

Шаги

1. Создайте файл `.env` в корневой папке вашего проекта (рядом с `main.py` или другим вашим основным модулем).
2. Впишите туда нужные секреты, например `TOKEN=ваш_токен_бота`.
3. Убедитесь, что `.env` добавлен в `.gitignore`, чтобы случайно не отправить секреты в общий репозиторий.
4. В вашем коде вызывайте `load_dotenv()`, затем получайте переменные с помощью `os.getenv("ИМЯ_ПЕРЕМЕННОЙ")`.

Применение на практике

Предположим, у вас есть телеграм-бот, который требует токен и пароль к базе данных. Вместо того чтобы писать:

python

```
# Небезопасный способ: зашиваем секреты прямо в код
TOKEN = "123456:ABC-DEF1234..."
DB_PASSWORD = "MySuperSecret"
```

Мы создаём файл `.env`:

ini

```
TOKEN=123456:ABC-DEF1234...
DB_PASSWORD=MySuperSecret
DEBUG=True
```

А в коде пишем:

```
import os
from dotenv import load_dotenv

load_dotenv()

TOKEN = os.getenv("TOKEN")
DB_PASSWORD = os.getenv("DB_PASSWORD")
DEBUG_MODE = os.getenv("DEBUG") == "True"

print("Token: ", TOKEN)
print("Database password: ", DB_PASSWORD)
print("Debug mode: ", DEBUG_MODE)
```

Таким образом:

- Секретная информация не хранится в открытом доступе.
- Разным участникам команды достаточно скопировать свой файл `.env` с корректными данными.
- В продакшене или на сервере вы можете задавать переменные окружения не только через файл, но и напрямую в системе (например, через docker, секреты GitHub Actions и т. д.).

Дополнительно. Хранение данных с помощью `pickle`

`pickle` — это стандартный модуль Python для сериализации и десериализации объектов.

Сериализация — это процесс преобразования объекта Python (например, словаря или списка) в байтовую последовательность (которая может быть записана в файл или передана по сети).

Десериализация — обратный процесс, когда мы считываем байты и восстанавливаем исходный объект.

Почему стоит использовать `pickle`?

1. **Простота.** Вы можете сохранить любой (почти) объект Python в файл буквально одной строкой и потом прочитать его так же легко.
2. **Удобство.** Не нужно создавать схему БД, таблицы, писать SQL-запросы. Вы работаете со структурой данных (например словарём) так, как привыкли, а перед выходом из программы сохраняете её в файл.
3. **Перезапуск.** Данные в файле сохраняются между запусками кода, поэтому можно продолжать работу с того же места.

Пример использования

Допустим, у вас в программе есть словарь `data`, в котором хранится, например, информация о пользователях:

```
import pickle

# Предположим, это наш словарь, который хотим сохранить
data = {
    "user1": {"age": 25, "city": "New York"},
    "user2": {"age": 30, "city": "London"}
}

# Сохраняем (записываем в файл data.pkl)
with open("data.pkl", "wb") as f:
    pickle.dump(data, f)

print("Данные успешно сохранены в data.pkl")
```

Обратите внимание на то, что файл мы открываем с режимом `"wb"` (write binary), так как `pickle` работает с двоичными данными.

Чтобы загрузить данные из файла и восстановить словарь, действуйте так:

```
import pickle

with open("data.pkl", "rb") as f:
    loaded_data = pickle.load(f)

print("Загруженные данные:", loaded_data)
```

Здесь файл открывается в режиме `"rb"` (read binary).

Использование pickle в боте

Упрощённый пример использования `pickle` в контексте Telegram-бота может выглядеть так:

```
import pickle
from telegram import Update
from telegram.ext import ApplicationBuilder, CommandHandler,
ContextTypes

# Файл, где будут храниться данные
PICKLE_FILE = "bot_data.pkl"
```

```
# Глобальный словарь для хранения состояния (например, счётчик
пользователей)
bot_data = {} # Или загрузим его при старте

def load_data():
    global bot_data
    try:
        with open(PICKLE_FILE, "rb") as f:
            bot_data = pickle.load(f)
    except FileNotFoundError:
        bot_data = {}

def save_data():
    with open(PICKLE_FILE, "wb") as f:
        pickle.dump(bot_data, f)

async def start_command(update: Update, context: ContextTypes.
DEFAULT_TYPE):
    # Допустим, считаем количество запусков /start
    bot_data["start_count"] = bot_data.get("start_count", 0)
    + 1
    save_data()

    await update.message.reply_text(
        f"Вы вызвали /start {bot_data['start_count']} раз(a).")

    )

async def main():
    load_data()

    app = ApplicationBuilder().token("YOUR_TOKEN_HERE").build()
    app.add_handler(CommandHandler("start", start_command))

    await app.run_polling()

if __name__ == "__main__":
    import asyncio
    asyncio.run(main())
```

Здесь:

1. При запуске бота мы загружаем данные (`load_data()`) из файла, если он существует.

2. В обработчике команды `/start` изменяем счётчик, сохраняем данные (функция `save_data()`) и сообщаем пользователю, сколько раз он уже вызывал `/start`.
3. Таким образом, после перезапуска бота счётчик сохраняется.

Плюсы и минусы

Плюсы

- Максимально простой способ сохранять структуры данных Python.
- Не требует дополнительных библиотек (модуль `pickle` встроен).
- Для небольших проектов и экспериментов зачастую достаточно.

Минусы

- **Безопасность.** Загружать `pickle`-файлы из недоверенных источников небезопасно, так как во время десериализации могут быть выполнены произвольные команды (уязвимость).
- **Неудобочитаемость.** Файлы `pickle` в двоичном формате, вы не сможете легко просмотреть или поправить их «руками» (в отличие от `JSON` или `CSV`).
- **Сложности при версии кода.** Если структура данных сильно меняется со временем, старые `pickle`-файлы могут быть несовместимы.
- **Объём памяти.** При хранении больших структур Python файл может вырастать, а при загрузке он полностью развернётся в оперативную память.