

L2. Python для бота: команды, текст, кнопки

Углубляем знания Python и создаем интерактивного бота с командами, обработкой текста и удобными кнопками. Опираемся на Hello Bot из первого урока и добавляем новую функциональность для улучшения пользовательского опыта.



Маршрут занятия

01

Python-блок

Работа со строками, списками, словарями, условиями и функциями для обработки пользовательского ввода

03

Интерактивные кнопки

ReplyKeyboard для быстрого доступа и опционально Inline-кнопки для подтверждений

Продолжительность: 90 минут с демонстрацией кода в реальном времени

02

Обработчики команд

Создание команд /about, /echo, /sum с валидацией и обработкой ошибок

04

Запуск и отладка

Логирование, частые ошибки и лучшие практики разработки

Python: Интерпретируемый и Объектно-ориентированный

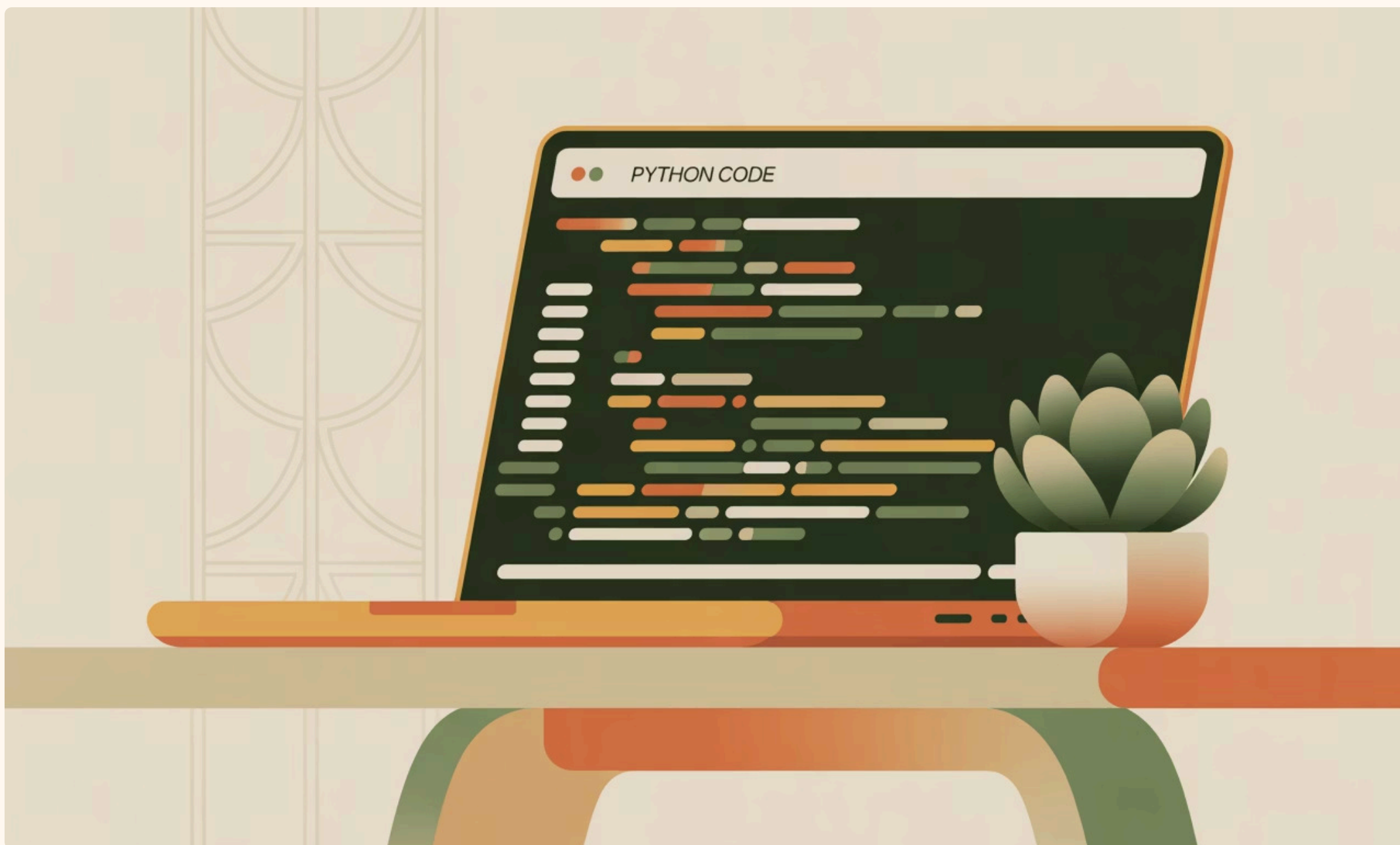
Python является ключевым инструментом в разработке ботов благодаря своей гибкости и мощным парадигмам.

Интерпретируемый язык

Код выполняется построчно без предварительной компиляции. Это ускоряет процесс разработки и упрощает отладку, позволяя быстро тестировать изменения.

Объектно-ориентированное программирование (ООП)

Позволяет организовать код в виде классов и объектов. Это повышает модульность, переиспользуемость и поддерживаемость, что критично для сложных проектов.



Переменные и типы данных

Основные типы

- int - целые числа
- str - строки текста
- bool - логические значения
- type(x) - проверка типа

Приведение типов

- int("42") ✓ работает
- int("3.14") ошибка
- Всегда валидируйте ввод

```
age_str = "18"  
age = int(age_str)  
print(type(age), type(age_str))
```


Обработка ошибок

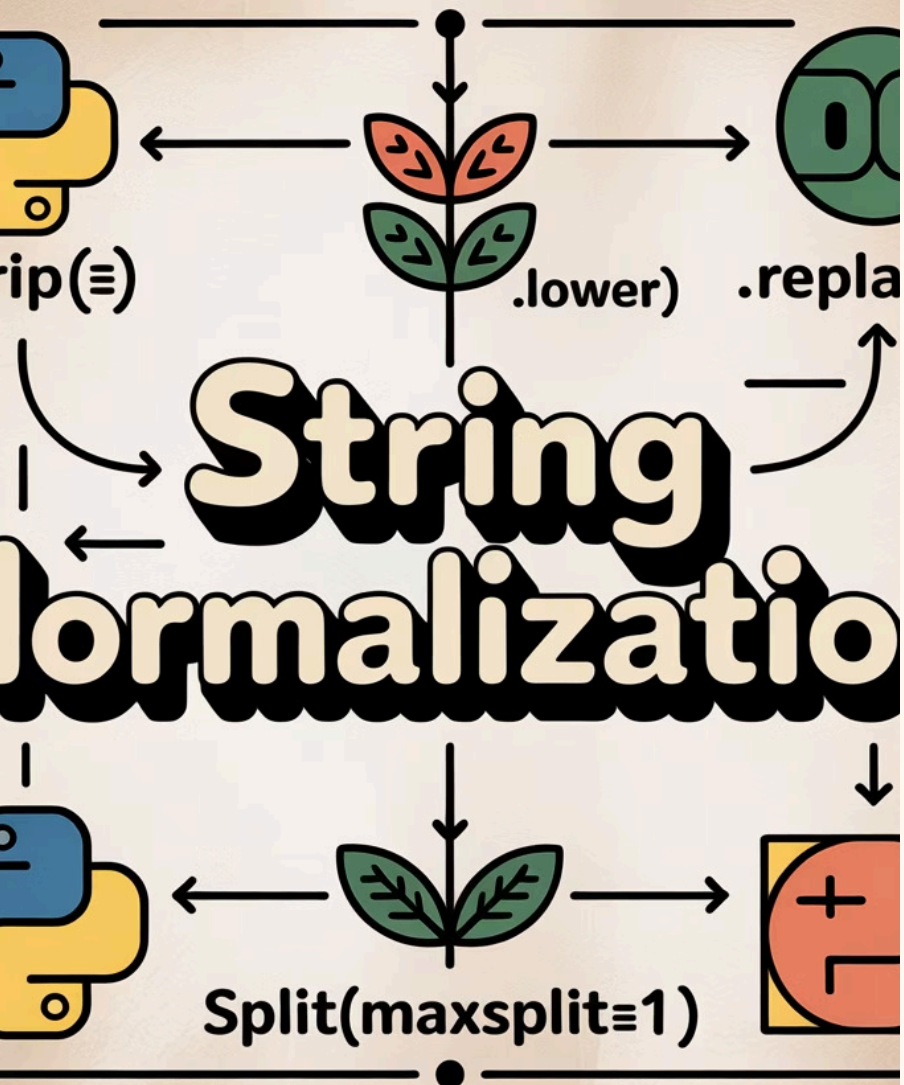
try:

```
    num = int(user_input)
```

except ValueError:

```
    print("Не число!")
```

 Понимание типов критично для обработки аргументов команд бота



Строки: нормализация ПОЛЬЗОВАТЕЛЬСКОГО ВВОДА

Ключевые методы для обработки текста

- `.strip()` - удаление пробелов по краям
- `.lower()` - приведение к нижнему регистру
- `.replace()` - замена символов
- `split(maxsplit=1)` - разделение на команду и аргумент

```
cmd = "/echo Привет, мир"
parts = cmd.split(maxsplit=1)
command = parts[0] # "/echo"
arg = parts[1] if len(parts) > 1 else "" # "Привет, мир"
```

Этот паттерн - основа для всех команд с аргументами

Списки и обход элементов

Создание и работа со списками

- Создание: `list()` или `[]`
- Добавление: `.append()`
- Обход: `for item in list`
- Фильтрация: list comprehension

List Comprehension для обработки ввода

List comprehension предлагает компактный и читаемый способ создания новых списков на основе существующих. Особенно полезен для фильтрации и преобразования пользовательского ввода.

```
# Пример: Извлечение числовых аргументов из пользовательского сообщения
user_message = "закажи 3 пиццы, 2 колы и 1 салат"
words = user_message.split()
```

```
# Используем list comprehension для извлечения только чисел
# (которые могут быть аргументами для команд бота)
quantities = [int(word) for word in words if word.isdigit()]
# quantities будет [3, 2, 1] - бот может использовать их для выполнения заказа
```

```
# Пример: Фильтрация команд, начинающихся с определенного префикса
all_input_tokens = ["/start", "привет", "/help", "как дела?", "/settings user"]
```

```
# Фильтруем только те токены, которые похожи на команды бота
bot_commands = [token for token in all_input_tokens if token.startswith("/")]
# bot_commands будет ["/start", "/help", "/settings user"]
```

Такой подход делает код более лаконичным и эффективным при работе с данными, полученными от пользователя.

Нормализация разделителей

```
text = "2, 3, -5"
# Заменяем запятые на пробелы
tokens = text.replace(",", " ").split()
# ['2', '3', '-5']
```



Списки позволяют гибко обрабатывать пользовательский ввод с различными разделителями

Словари: временное состояние пользователя

Словари в Python – это неупорядоченные коллекции элементов, где каждый элемент хранится как пара "ключ:значение". Они оптимизированы для извлечения значений по ключам, что делает их идеальными для хранения структурированных данных, таких как временные состояния пользователей в ботах. Ключи должны быть уникальными и неизменяемыми (например, строки, числа, кортежи), а значения могут быть любого типа.

Создание словарей

```
# Пустой словарь
user_data = {}

# Словарь с начальными значениями
bot_settings = {
    "language": "ru",
    "notifications": True,
    "theme": "dark"
}

# Создание из пар ключ-значение с помощью dict()
user_profiles = dict(user1="admin", user2="guest")
```

Основные операции со словарями

Добавление и изменение элементов происходит через присваивание значения по ключу. Если ключ уже существует, его значение обновится.

```
# Добавление нового элемента
user_data["user_id_1"] = {"name": "Alice", "balance": 100}

# Изменение существующего элемента
bot_settings["theme"] = "light"

# Если user_id_2 не существует, он будет добавлен
# Если user_id_2 существует, его значение будет обновлено
user_data["user_id_2"] = {"name": "Bob", "balance": 50}
```

Получение значения по ключу. Используйте метод `.get()` для безопасного извлечения, чтобы избежать ошибок `KeyError`, если ключ отсутствует.

```
# Получение значения
current_language = bot_settings["language"] # "ru"

# Безопасное получение значения (если ключа нет, вернет None)
notifications_enabled = bot_settings.get("notifications") # True

# Безопасное получение значения с значением по умолчанию
default_currency = user_data.get("currency", "USD") # "USD" если
ключа нет

# Проверка наличия ключа
if "user_id_1" in user_data:
    print("Пользователь 1 найден")
```



Словари: основные операции и методы

Словари в Python – это неупорядоченные коллекции элементов, где каждый элемент хранится как пара "ключ:значение". Они оптимизированы для извлечения значений по ключам, что делает их идеальными для хранения структурированных данных, таких как временные состояния пользователей в ботах. Ключи должны быть уникальными и неизменяемыми (например, строки, числа, кортежи), а значения могут быть любого типа.

Добавление и изменение элементов происходит через присваивание значения по ключу. Если ключ уже существует, его значение обновится.

Получение значения по ключу. Используйте метод `.get()` для безопасного извлечения, чтобы избежать ошибок `KeyError`, если ключ отсутствует.


Методы словарей: `.keys()`, `.values()`, `.items()`



`.keys()`

Возвращает объект представления, содержащий все ключи словаря.


```
config = {"host": "localhost", "port": 8080}
for key in config.keys():
    print(key)
# Вывод: host, port
```



`.values()`

Возвращает объект представления, содержащий все значения словаря.

```
user_actions = {"Alice": "start", "Bob": "help"}
for value in user_actions.values():
    print(value)
# Вывод: start, help
```



`.items()`

Возвращает объект представления, содержащий пары ключ-значение в виде кортежей.

```
chat_status = {"chat_id_1": "active", "chat_id_2": "idle"}
for chat_id, status in chat_status.items():
    print(f'{chat_id}: {status}')
# Вывод: chat_id_1: active, chat_id_2: idle
```

Применение словарей в ботах для управления состоянием

Хранение состояния пользователя

Словари позволяют отслеживать состояние каждого пользователя между сообщениями, сохраняя контекст диалога.

```
user_state = {} # Глобальный словарь для хранения состояний
# Пример инициализации или обновления состояния:
user_state[user_id] = {
    "mode": "awaiting_sum_input", # Ожидаем ввод суммы
    "data": [], # Временные данные для текущего диалога
    "last_command": "/start"
}
```

Реализация многошаговых сценариев

Используя состояние пользователя, можно реализовать сложные диалоги без необходимости обращения к базе данных на каждом шаге.

```
# Предположим, user_id - идентификатор текущего пользователя
if user_id in user_state:
    current_mode = user_state[user_id]["mode"]
    if current_mode == "awaiting_sum_input":
        # Обработка полученного сообщения как суммы
        # ...
        user_state[user_id]["mode"] = "sum_processed" #
    Обновляем состояние
else:
    # Инициализация состояния для нового пользователя
    user_state[user_id] = {"mode": "start", "data": []}
```

Словари являются мощным инструментом для разработки интерактивных ботов. В следующих уроках изучим `register_next_step_handler` для элегантной работы с состояниями и переходами между шагами диалога.

Условия, циклы и функции для валидации

Создание утилитарных функций

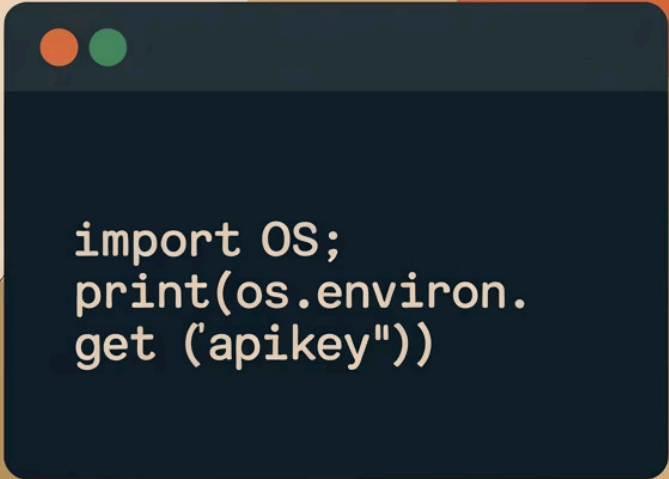
```
def is_int_token(t: str) -> bool:
    """Проверяет, является ли строка целым числом (включая отрицательные)"""
    return t.strip().lstrip("-").isdigit()

def validate_user_input(text: str) -> bool:
    """Проверяет корректность пользовательского ввода"""
    if not text or not text.strip():
        return False
    return True
```

- Выносим проверки в отдельные функции
- Используем типизацию для ясности кода
- Обрабатываем крайние случаи (пустой ввод, отрицательные числа)

Готовим фундамент для команды /sum с надежной валидацией

Каркас TeleBot с защищенной конфигурацией



```
import OS;  
print(os.environ.  
get ('apikey'))
```

```
import os  
from dotenv import load_dotenv  
import telebot  
from telebot import types  
  
load_dotenv()  
TOKEN = os.getenv("TOKEN") or ""  
  
if not TOKEN:  
    raise RuntimeError("Нет TOKEN в .env")  
  
bot = telebot.TeleBot(TOKEN)
```

⊗ Никогда не храните токены в коде! Используйте .env файлы и добавляйте их в .gitignore

Безопасная инициализация - основа профессиональной разработки

Команды /start и /help с меню

Настройка меню команд

```
def setup_bot_commands():  
    cmds = [  
        types.BotCommand("start", "Запуск"),  
        types.BotCommand("help", "Помощь"),  
        types.BotCommand("about", "О боте"),  
        types.BotCommand("sum", "Сумма чисел"),  
        types.BotCommand("echo", "Повторить текст"),  
    ]  
    bot.set_my_commands(cmds)
```

Обработчик команд

```
@bot.message_handler(commands=['start','help'])  
def start_help(m):  
    bot.send_message(  
        m.chat.id,  
        "Привет! Доступно: /about, /sum, /echo"  
    )
```

Меню команд появится в интерфейсе Telegram, упрощая взаимодействие с ботом

Команда /about - представление бота

Простая реализация

```
@bot.message_handler(commands=['about'])
def about(m):
    bot.reply_to(m,
        "Я учебный бот. Команды: "
        "/start, /help, /about, /sum, /echo"
    )
```

Используем `reply_to` для связывания ответа с исходным сообщением

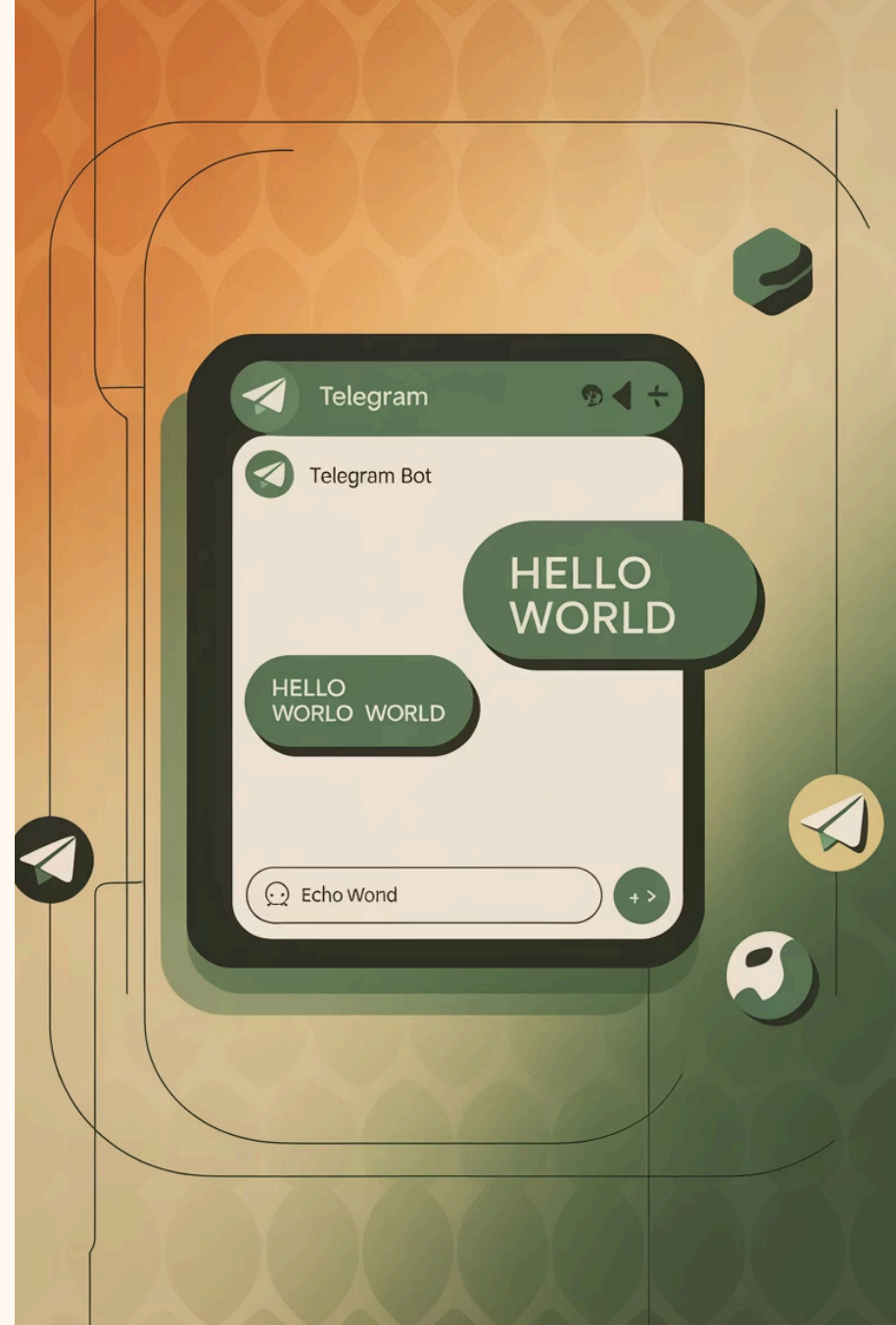
Команда /echo - парсинг аргументов

Надежная обработка команды с аргументами

```
@bot.message_handler(commands=['echo'])
def echo_cmd(m):
    parts = m.text.split(maxsplit=1)

    if len(parts) < 2 or not parts[1].strip():
        bot.reply_to(m, "Пример: /echo Привет, мир")
    else:
        bot.reply_to(m, parts[1])
```

- `maxsplit=1` - разделяем только на команду и аргумент
- Проверяем наличие и непустоту аргумента
- Даем четкую подсказку при некорректном вводе



Команда /sum - устойчивый парсинг чисел

```
def parse_ints_from_text(text: str) -> list[int]:  
    """Извлекает целые числа из текста, игнорируя команды и спецсимволы"""  
    text = text.replace(",", " ") # Нормализуем разделители  
    tokens = [t for t in text.split() if not t.startswith("/")]  
    return [int(t) for t in tokens if t.strip().lstrip("-").isdigit()]  
  
@bot.message_handler(commands=['sum'])  
def cmd_sum(m):  
    nums = parse_ints_from_text(m.text)  
  
    if not nums:  
        bot.reply_to(m,  
            "Нужно написать числа. Пример: /sum 2 3 10 или /sum 2, 3, -5")  
    else:  
        bot.reply_to(m, f"Сумма: {sum(nums)}")
```

✅ Обрабатывает: запятые, пробелы, отрицательные числа, игнорирует команды

Создание ReplyKeyboard

Функция создания клавиатуры

```
def make_main_kb() -> types.ReplyKeyboardMarkup:
    kb = types.ReplyKeyboardMarkup(resize_keyboard=True)
    kb.row("О боте", "Сумма")
    kb.row("/help")
    return kb

@bot.message_handler(commands=['start','help'])
def start_help(m):
    bot.send_message(
        m.chat.id,
        "Привет! Доступно: /about, /sum, /echo\nИли пользуйтесь кнопками ↓",
        reply_markup=make_main_kb()
    )
```

- `resize_keyboard=True` - адаптивный размер кнопок
- `.row()` - добавление ряда кнопок
- `reply_markup` - прикрепление клавиатуры к сообщению

Обработка нажатий кнопок

```
@bot.message_handler(func=lambda m: m.text == "О боте")
```

```
def kb_about(m):
```

```
    about(m) # Переиспользуем существующую функцию
```

```
@bot.message_handler(func=lambda m: m.text == "Сумма")
```

```
def kb_sum(m):
```

```
    bot.send_message(m.chat.id,
```

```
        "Введите числа через пробел или запятую:")
```

```
    bot.register_next_step_handler(m, on_sum_numbers)
```

```
def on_sum_numbers(m):
```

```
    nums = parse_ints_from_text(m.text)
```

```
    if not nums:
```

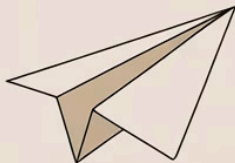
```
        bot.reply_to(m, "Не вижу чисел. Пример: 2 3 10")
```

```
    else:
```

```
        bot.reply_to(m, f"Сумма: {sum(nums)}")
```

`register_next_step_handler` - элегантный способ создания одношаговых сценариев без базы данных

Send a Message



Скрытие клавиатуры

Команда для скрытия

```
@bot.message_handler(commands=['hide'])
def hide_kb(m):
    rm = types.ReplyKeyboardRemove()
    bot.send_message(
        m.chat.id,
        "Спрятал клавиатуру.",
        reply_markup=rm
    )
```

Когда полезно?

- Временная очистка экрана
- Переход к свободному вводу
- Разные режимы взаимодействия
- Адаптация под задачи пользователя

Inline-кнопки для подтверждений

Кнопки под сообщением с callback_data

```
@bot.message_handler(commands=['confirm'])
def confirm_cmd(m):
    kb = types.InlineKeyboardMarkup()
    kb.add(
        types.InlineKeyboardButton("Да", callback_data="confirm:yes"),
        types.InlineKeyboardButton("Нет", callback_data="confirm:no")
    )
    bot.send_message(m.chat.id, "Понравился пример?", reply_markup=kb)

@bot.callback_query_handler(func=lambda c: c.data.startswith("confirm:"))
def on_confirm(c):
    choice = c.data.split(":", 1)[1]
    bot.answer_callback_query(c.id, "Спасибо!")
    bot.edit_message_reply_markup(c.message.chat.id, c.message.message_id, reply_markup=None)

    response = "Отлично!" if choice == "yes" else "Окей, улучшим!"
    bot.send_message(c.message.chat.id, response)
```

Логирование для отладки

Базовая настройка

```
import logging

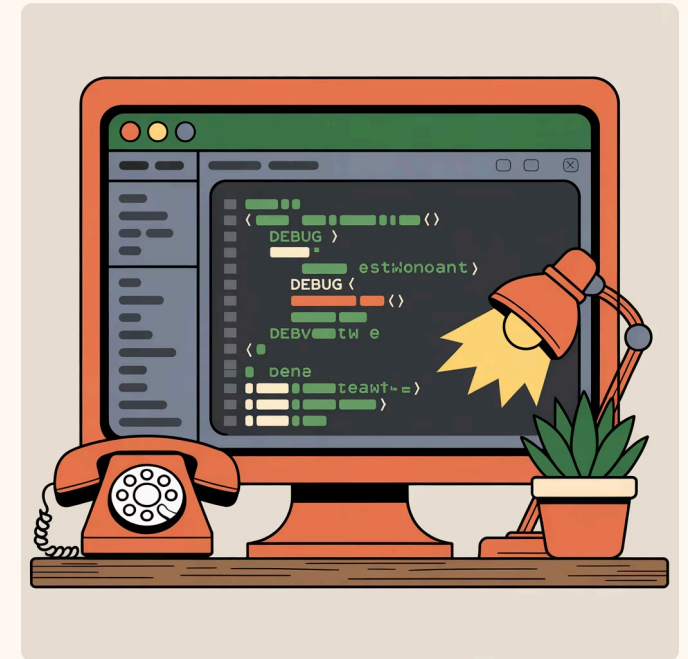
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)
```

Зачем нужны логи?

- Видно, приходят ли сообщения
- Отслеживание ошибок
- Мониторинг активности

Пример в коде

```
@bot.message_handler(commands=['sum'])
def cmd_sum(m):
    logging.info(f'Sum command from {m.from_user.first_name}')
    nums = parse_ints_from_text(m.text)
    logging.info(f'Parsed numbers: {nums}')
    # ... остальная логика
```



Итоговая структура main.py

1

Импорты и конфигурация

```
import os, logging
from dotenv import load_dotenv
import telebot
from telebot import types
```

2

Утилитарные функции

```
def parse_ints_from_text(text):
def make_main_kb():
def setup_bot_commands():
```

3

Обработчики команд

```
@bot.message_handler(commands=['start'])
@bot.message_handler(commands=['echo'])
@bot.message_handler(commands=['sum'])
```

4

Точка входа

```
if __name__ == "__main__":
    setup_bot_commands()
    bot.infinity_polling(skip_pending=True)
```




Запуск и тестирование бота

01

Активация окружения

Убедитесь, что виртуальное окружение
активировано и все зависимости
установлены

02

Запуск бота

```
python main.py
```

Должны увидеть логи о запуске polling

03

Тестирование в Telegram

Откройте чат с ботом → /start → проверьте кнопки → протестируйте /echo и /sum

Частые ошибки и их решения

Бот не отвечает

- Проверьте активацию `venv`
- Убедитесь в наличии интернета
- Проверьте корректность `TOKEN`
- Посмотрите логи в консоли

Токен не подхватился

- Вызван ли `load_dotenv()`?
- Находится ли `.env` в корне?
- Нет ли пробелов в `TOKEN=...`
- Добавлен ли `.env` в `.gitignore`?

Ошибки парсинга

- Проверьте нормализацию запятых
- Обработайте пустые строки
- Валидируйте отрицательные числа
- Игнорируйте лишние символы

Безопасность и Git-практики

Работа с секретами

- Все токены только в .env файлах
- .env обязательно в .gitignore
- Никогда не коммитим токены
- Используем example.env для документации

Качество кода

- Маленькие осмысленные коммиты
- Описательные сообщения коммитов
- Pull Request с описанием изменений
- Код-ревью перед слиянием



⊗ Один случайно загруженный токен может скомпрометировать весь проект!

Мини-квиз для закрепления

Вопрос 1

Что делает `split(maxsplit=1)` и чем это отличается от обычного `split()`?

Вопрос 2

В чем разница между операторами `=` и `==` в Python?

Вопрос 3

Когда выбрать `ReplyKeyboard`, а когда оставить свободный ввод?

Вопрос 4

Что делает параметр `skip_pending=True` в `infinity_polling()`?

Обсудите ответы с коллегами - это поможет лучше понять материал

Мини-квиз для закрепления

Вопрос 1

Что делает `split(maxsplit=1)` и чем это отличается от обычного `split()`?

Ответ: **`split(maxsplit=1)`** делит строку только по первому вхождению разделителя, создавая список из двух частей. Обычный **`split()`** делит строку по всем вхождениям разделителя.

Вопрос 2

В чем разница между операторами `=` и `==` в Python?

Ответ: `=` используется для присваивания значения переменной. `==` используется для сравнения двух значений на равенство.

Вопрос 3

Когда выбрать `ReplyKeyboard`, а когда оставить свободный ввод?

Ответ: **`ReplyKeyboard`** выбирают, когда есть ограниченный набор predetermined вариантов ответа. Свободный ввод оставляют, когда ответ может быть произвольным текстом или числом.

Вопрос 4

Что делает параметр `skip_pending=True` в `infinity_polling()`?

Ответ: **`skip_pending=True`** указывает боту игнорировать все сообщения, отправленные ему, пока он был офлайн. Обработываются только новые сообщения после запуска.

Обсудите ответы с коллегами - это поможет лучше понять материал



Что изучим дальше

Семинары S3-S4

Закрепляем Python и команды, добавляем новую команду в домашнем задании

1

2

3

Проекты

Создание полнофункциональных ботов с персистентным хранением данных

Лекции L3-L4

Работа с базами данных SQLite, сохранение состояния пользователей

 Подготовьте рабочее окружение и убедитесь, что ваш Hello Bot из S1 работает корректно

Вопросы и завершение

Ключевые напоминания

- Код разрабатываем локально
- Токены храним в .env файлах
- Используем long polling для простоты
- Логируем важные события
- Тестируем каждую команду

Демонстрационный репозиторий на GitHub доступен без токенов - можете использовать как шаблон для своих проектов.



Практические советы

- Начинайте с простых команд
- Добавляйте функции постепенно
- Тестируйте каждое изменение
- Читайте документацию TeleBot

Время для ваших вопросов!