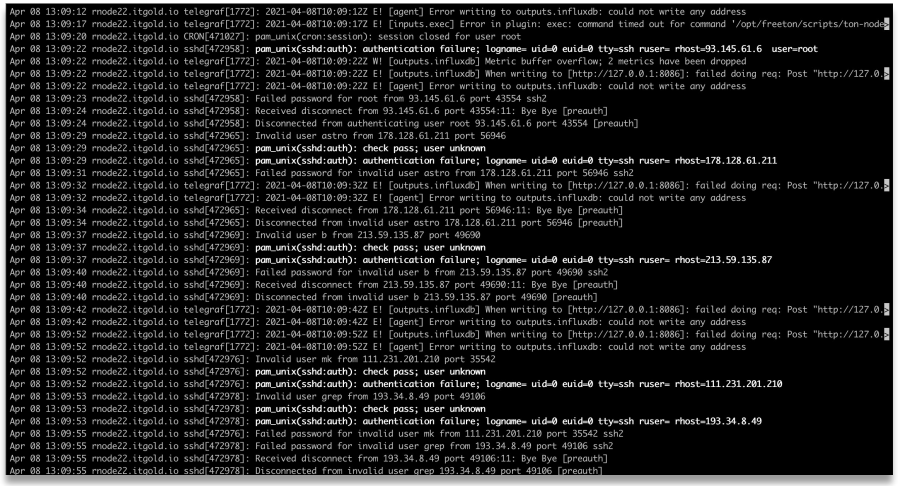
Python-11
Логирование (logging)
Виртуальная память/Адресация данных.
Основы БД. SQLite

Логирование

Логирование в программировании представляет собой процесс записи информации о ходе выполнения программы в файлы или другие выходные устройства. Этот инструмент является важной частью разработки программного обеспечения, поскольку позволяет отслеживать и регистрировать различные события и состояния приложения. Нужно для:

1) Обеспечение прозрачности
Логирование помогает разработчикам следить за ходом выполнения программы, обнаруживать ошибки и недочеты, а также обеспечивать прозрачность процесса работы программы.

2) Обнаружение и исправление ошибок
Запись логов позволяет быстро выявлять и анализировать ошибки в приложении. Четкие и информативные логи существенно упрощают процесс отладки.



Внешний модуль <logging>

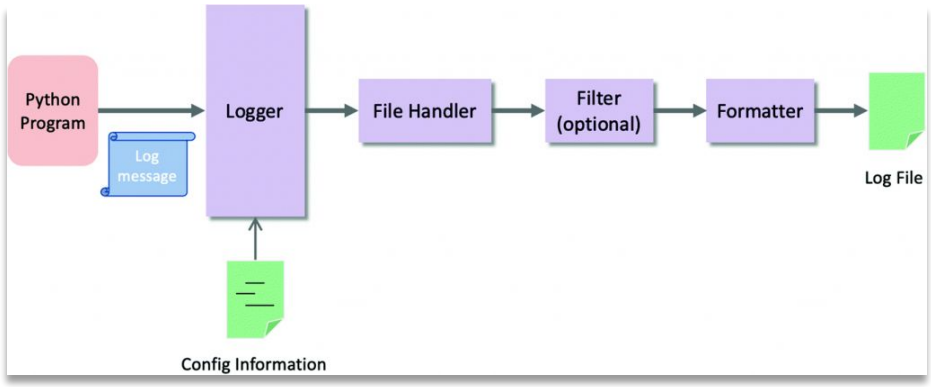
Модуль logging в Python предоставляет гибкий и мощный механизм для регистрации логов. Он включает в себя различные уровни логирования, настраиваемые обработчики и форматирование логов.

Основные компоненты модуля logging:

Logger - объект, представляющий собой точку входа для логирования в приложении.

Handler - отправляет логи в различные места, такие как файлы, консоль или сетевые службы.

Formatter - определяет формат записей лога, включая дату, уровень логирования и текст сообщения.



Уровни логирования

DEBUG - информация для отладки.
INFO - информационные сообщения о ходе выполнения программы.
WARNING - предупреждения о возможных проблемах.
ERROR - сообщения об ошибках, но не критических.
CRITICAL - критические ошибки, приводящие к прекращению работы приложения.

		Detail Included in Logs				
		Debug statements	Informational messages	Warning statements	Error statements	Fatal statements
Logging Level	ALL	Included	Included	Included	Included	Included
	DEBUG	Included	Included	Included	Included	Included
	INFO	Excluded	Included	Included	Included	Included
	WARN	Excluded	Excluded	Included	Included	Included
	ERROR	Excluded	Excluded	Excluded	Included	Included
	FATAL	Excluded	Excluded	Excluded	Excluded	Included
	OFF	Excluded	Excluded	Excluded	Excluded	Excluded

Создание логгера

```
import logging


logger = logging.getLogger('my_logger')
logger.setLevel(logging.DEBUG)
```

Добавление обработчика

```
...
file_handler = logging.FileHandler('app.log')
file_handler.setLevel(logging.INFO)

console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)

logger.addHandler(file_handler)
logger.addHandler(console_handler)
```

Форматирование логов

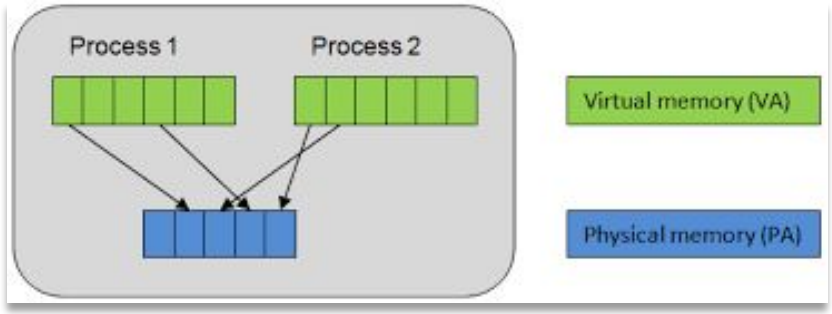
```
...
file_handler= logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
file_handler.setFormatter(formatter)
console_handler.setFormatter(formatter)
```

Запись логов

```
logger.debug('Это сообщение уровня DEBUG')
logger.info('Это информационное сообщение')
logger.warning('Это предупреждение')
logger.error('Это ошибка')
logger.critical('Это критическая ошибка')
```

Виртуальная память

Виртуальная память - это абстрактный идеальный слой памяти, предоставляемый операционной системой, который представляет собой комбинацию физической памяти (RAM/ОЗУ) и внешнего хранилища (например, жесткий диск (HDD), SSD...). Она позволяет программам использовать больше памяти, чем физически доступно в компьютере. *Нужна для:*



1) Эффективное управление памятью
Виртуальная память позволяет операционной системе управлять физической памятью более эффективно. Программы получают впечатление, что у них есть непрерывный блок памяти, хотя на самом деле данные могут храниться в разных областях физической и внешней памяти.

2) Поддержка больших программ
Виртуальная память позволяет запускать программы, которые требуют больше оперативной памяти, чем физически доступно на компьютере.

**VM <VA> стремится записаться -> PM <PA>*

Структура памяти

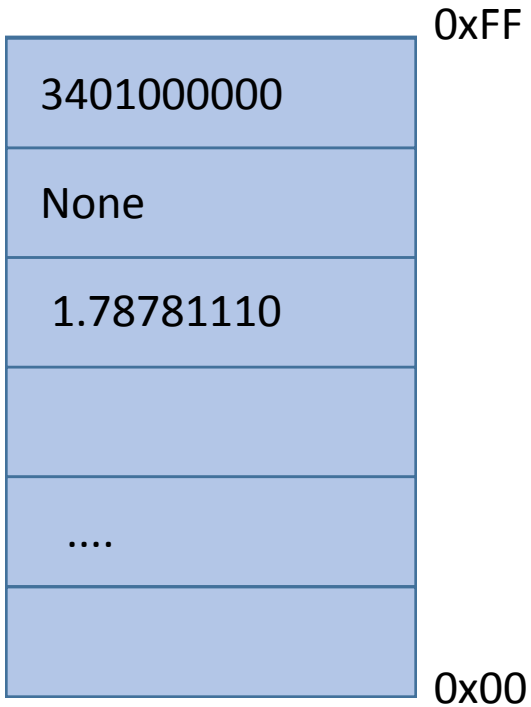
Адрес памяти представляет собой числовое значение, которое идентифицирует конкретное местоположение в памяти компьютерной системы.

Каждая ячейка памяти имеет определенный адрес, а каждый адрес (необязательно) используется для доступа к каким-либо данным (например, объекты, числа, и др данные).

В Python записывается 0xFF как “\0xFF”

Для более детально анализа используют языки ассемблера (ASM, NASM, TASM...) и соответствующие ОС x86-16, x86-32, x86-64. (Intel 8086)

x86-32 -> 2^32 = ...
x86-64 -> 2^64 =...



Язык Ассемблера

Язык Ассемблера - это низкоуровневый язык программирования, который использует регистры, сегменты и инсрукции.

Систему можно разделить на 3 типа:

- x86-16 - использует регистры AH, AL, AX, BX, CX, DX....
- x86-32 - использует регистры (префикс E) EAX, EBX, ECX,...
- x86-64 - использует регистры (префикс R) RAX, RBX, RCX, RDX...

Инструкции/Метки

Инструкций в языке ассемблера очень много, можно взять самые базовые MOV (перемещение данных), CMP (для сравнения данных в регистрах),...

Также есть метки (например метка main:)

```
section .text:
global main
main:
xor rax, rax
ret
```

Анализ кода (из лабы)

*Задача заключается в получении надписи **Access Granted!** без изменения класс Caller и его методов.*

```
class Caller:
    def __init__(self, base_address: any) -> None:
        self.base_address = base_address
        self.value =
self.read_file("/your_path_to_file/memory_addresses.txt
")

    def read_file(self, file_path):
        result_dict = {}
        with open(file_path, 'r') as file:
            data = file.read()
            data = data.split("\n")
            for line in data:
                parts = line.strip().split("-")
                if len(parts) == 2:
                    key, value = parts
                    result_dict[key] = value
        if self.base_address in result_dict:
            print(result_dict[self.base_address])
            return result_dict[self.base_address]
        else:
            return f"No value found for key:
{self.base_address}"
```

```
    @staticmethod
    def cmp_j(rax, rbx) -> str:
        if rbx == -1:
            return "RBX error!"
        if rax > rbx:
            return "Access Denied!"
        else:
            return "Access Granted!"

    @staticmethod
    def __call__access__(call_object, *args, **kwargs)
-> str:
        rax = args[0]
        rax *= -1
        r12 = 0
        rbx = args[len(args) - 1]
        rax *= rbx / 2
        rbx = 0.1
        r12 = int(call_object.value)
        rcx = -1 * 100 * (10 - args[kwargs["arg"]] +
r12) / 80
        rbx *= rcx
        rcx = 0
        return call_object.cmp_j(rax, rbx)
```

*По **__call__access__** - вызываемой функции программного кода видно:*

*- **регистры** (rax, rbx, rcx, r12)*
*- входные данные ***args/**kwargs***

rax принимает первый элемент входа args, после уменьшается на 1.

***r12** равен 0;*

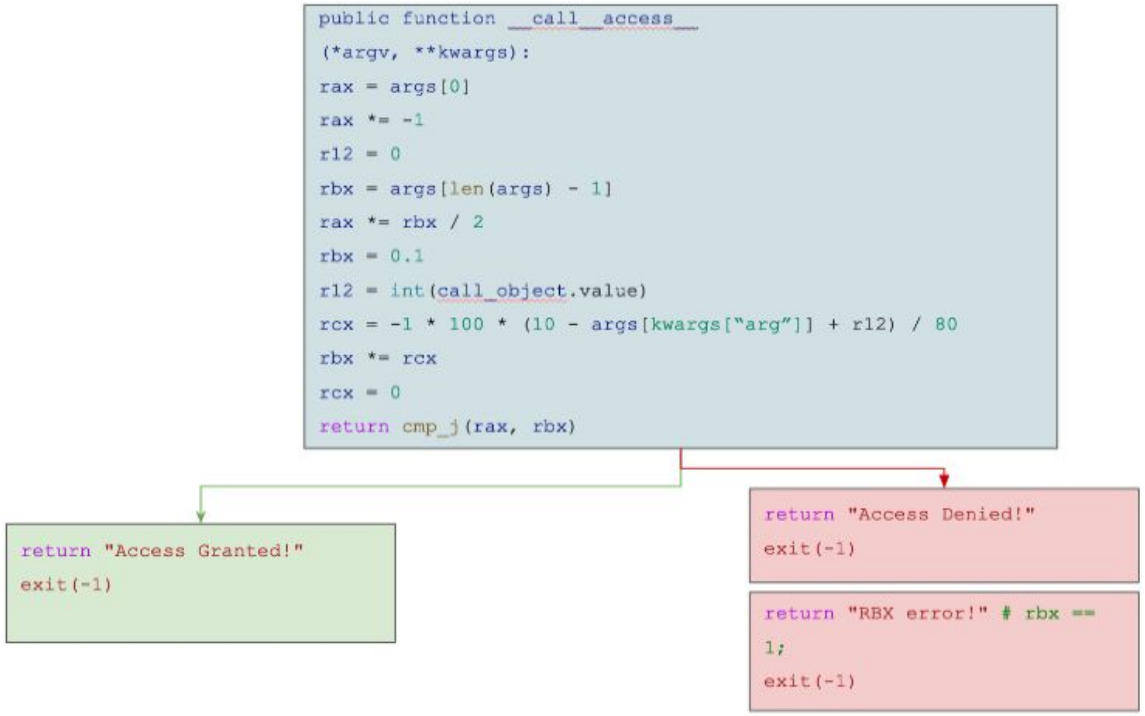
***rbx** принимает последний элемент входа args;*

затем происходят вычисления;

***rcx** вычисляется с приминением данных из kwargs + регистр **r12**;*

*в конце возврат результата работы **CMP_J** (сравнения регистра rax с rbx)*

Блок схема кода (IDA)



Блок схема *(зеленая (нужное) - если* ***rax*** *<=* ***rbx****,*
красная если ***rax*** *>* ***rbx*** *или* ***rbx****= -1):*

cmp_j *<инструкция> для сравнения регистров* ***rax*** *и*
rbx *используется.*

База данных.

База данных (БД) - это специальное место, где хранятся информация и данные. Давайте представим себе большой шкаф, в котором лежат папки с информацией.

Внутри базы данных есть таблицы, как листы в тетради. В таблицах данные организованы в строки и столбцы. **Каждая строка** - это запись о чем-то, а **столбцы** - это категории информации.

Запись - это какая-то информация о чем-то конкретном. Например, если база данных о книгах, то каждая запись может представлять собой информацию о одной книге. **Атрибут** - это какое-то свойство или информация о записи. Например, если у нас есть таблица с информацией о книгах, то атрибутами могут быть "Название книги", "Автор" и "Год выпуска".

Первичный ключ (Primary Key) - это как уникальный шифр или номер каждой записи. Он позволяет легко находить нужную запись в базе данных.

База данных. Понятия (доп)

SQL - это специальный язык, который используется для работы с базами данных (БД). Это как инструкции или команды, которые говорят базе данных, что делать с данными. **СУБД (Система Управления Базами Данных**)- это как директор или управляющий базой данных. Она следит за тем, чтобы данные были хорошо организованы и доступны.

Нормализация - это правила, которые помогают хранить данные аккуратно и в порядке. **Денормализация** - это как сделать данные более доступными и быстрыми, но менее аккуратными.

Введение в SQLite.

SQLite - это легковесная и встроенная СУБД (Система Управления Базами Данных), которая позволяет создавать и управлять базами данных в приложениях Python и других языках программирования.

Установка библиотеки через CMD:

```
pip install sqlite3
```

Импорт библиотеки в код программы:

```
import sqlite3
```

SQLite. Создание первой таблицы.

После импорта sqlite нужно создать соединение с базой данных. Если база данных не существует, она будет автоматически создана. В примере ниже создадим базу данных с именем "mydatabase.db".

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

Теперь можете создать таблицу в базе данных с помощью SQL-запроса. В примере ниже создадим таблицу "users" с двумя полями: "id" и "name".

```
cursor.execute('''CREATE TABLE users
                  (id INT PRIMARY KEY NOT NULL,
                   name TEXT NOT NULL)''')
```

Этот запрос создаст таблицу "users" с двумя столбцами: "id" и "name". Столбец "id" будет содержать целые числа, и он является первичным ключом (PRIMARY KEY), что означает, что каждая запись в таблице будет иметь уникальный идентификатор. Столбец "name" будет содержать текстовые данные.

После создания таблицы, не забудьте сохранить изменения в базе данных и закрыть соединение:

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

SQLite. Добавление.

Для добавления данных в таблицу используется оператор INSERT INTO. Например, допустим, вы хотите добавить нового пользователя в таблицу "users":

```
import sqlite3

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

# Добавление нового пользователя
cursor.execute("INSERT INTO users (id, name) VALUES (1, 'John')")

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

SQLite. Удаление.

Для удаления данных используется оператор DELETE. Например, допустим, вы хотите удалить пользователя с определенным идентификатором из таблицы "users":

```python
import sqlite3

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

# Удаление пользователя с определенным ID
user_id_to_delete = 1
cursor.execute("DELETE FROM users WHERE id = ?", (user_id_to_delete,))

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

? - необязательное поле означает.

SQLite. Обновление данных в таблице.

Для обновления данных используется оператор UPDATE. Например, допустим, вы хотите изменить имя пользователя с определенным идентификатором:

```
import sqlite3

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

# Обновление имени пользователя с определенным ID
new_name = "Jane"
user_id_to_update = 1
cursor.execute("UPDATE users SET name = ? WHERE id = ?", (new_name, user_id_to_update))

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

SQLite. Извлечение данных из таблицы.

Для извлечения данных используется оператор SELECT. Например, допустим, вы хотите получить список всех пользователей:

```
import sqlite3

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

# Извлечение всех пользователей
cursor.execute("SELECT * FROM users")
users = cursor.fetchall()

for user in users:
    print(user)

conn.close()
```

SQLite. Извлечение данных с условием из таблицы.

Чтобы получить данные, удовлетворяющие определенным условиям, используйте оператор SELECT с условием WHERE. Например, чтобы получить список пользователей с именем "John", вы можете сделать следующее:

```
import sqlite3

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

# Извлечение пользователей с именем "John"
name_to_find = "John"
cursor.execute("SELECT * FROM users WHERE name = ?", (name_to_find,))
johns = cursor.fetchall()

for user in johns:
    print(user)

conn.close()
```

SQLite. Операции с множеством записей.

Вы можете добавлять, обновлять и удалять множество записей одновременно, используя операторы INSERT INTO, UPDATE и DELETE. Например, чтобы добавить несколько пользователей сразу:

```
# ...
users_to_insert = [
    (2, 'Alice'),
    (3, 'Bob'),
    (4, 'Eve')
]

cursor.executemany("INSERT INTO users (id, name) VALUES (?, ?)", users_to_insert)
```

*Этот код добавит несколько пользователей в таблицу "users".

Пример. <sqlite>

Предположим мы захотели написать tgbot<ShopCenter>. Нам бы потребовались системы:

- Вход/Регистрация пользователей;
- Система ленты товаров;
- Система оплаты;
- ... др какие-то системы;

для авторизации пользователей как раз можно использовать базу данных в нашем случае <локальную БД> - SQLite.

Первым делом создадим директорию с файлом db/db.db - звучит прикольно!)
Вторым делом создадим основной файл программы main.py + второй файл для работы с методами нашей SQL. <sql.py>
Третим делом создадим tgbot - и получим специальный ТОКЕН.
...

<sql.py> (без реализации)

```python
class SQL:
    def __init__(self, table_name):
        self.table_name = table_name
        pass

    def add_user(self, data):
        pass

    def update_user(self, data):
        pass

    def delete_user(self, user_name):
        pass

    def get_user_info(self):
        pass

    def custom_query(self, query):
        pass

    def fetch_all_records(self):
        pass

    def count_users(self):
        pass

    def search_users(self, keyword):
        pass
```

<sql.py> INIT

```python
import sqlite3


class SQL:
    def __init__(self, table_name, path_db):
        self.table_name = table_name
        self.connection = sqlite3.connect(f'{path_db}')
        self.cursor = self.connection.cursor()
```

доопределим конструктор класса + import sqlite3

<sql.py> ADD_USER

```python
def add_user(self, data):
    columns = ', '.join(data.keys())
    values = ', '.join(['?' for _ in range(len(data))])
    query = f"INSERT INTO {self.table_name} ({columns}) VALUES ({values})"
    self.cursor.execute(query, tuple(data.values()))
    self.connection.commit()
```

доопределим add_user(..)

columns - строка, представляющая имена столбцов в таблице базы данных. Создается путем соединения ключей data (DICT). *Например*, если data является {'user_name': 'John', 'age': 25, 'email': 'john@example.com'}, затем columns было бы 'user_name, age, email'.

values - строка, которая представляет собой заполнители для значений в запросе SQL. Создается путем объединения заполнителей '?' через запятые, а кол-во заполнителей определяется data dictionary. *Пример*, если data {'user_name': 'John', 'age': 25, 'email': 'john@example.com'}, то шаблон таков '?, ?, ?'.

<sql.py> UPDATE_USER

```
def update_user(self, data):
    set_clause = ', '.join([f"{key} = ?" for key in data.keys()])
    query = f"UPDATE {self.table_name} SET {set_clause} WHERE user_name = ?"
    self.cursor.execute(query, tuple(data.values()) + (data['user_name'],))
    self.connection.commit()
```

доопределим update_user(..)

<sql.py> DELETE_USER/GET_USER_INFO

```python
def delete_user(self, user_name):
    query = f"DELETE FROM {self.table_name} WHERE user_name = ?"
    self.cursor.execute(query, (user_name,))
    self.connection.commit()

def get_user_info(self):
    query = f"SELECT * FROM {self.table_name}"
    self.cursor.execute(query)
    return self.cursor.fetchall()
```

доопределим delete_user(..)/get_user_info(self)

<sql.py> COUNT_USERS/SEARCH_USERS

```python
def count_users(self):
    query = f"SELECT COUNT(*) FROM {self.table_name}"
    self.cursor.execute(query)
    return self.cursor.fetchone()[0]

def search_users(self, keyword):
    query = f"SELECT * FROM {self.table_name} WHERE user_name LIKE ?"
    self.cursor.execute(query, (f'%{keyword}%',))
    return self.cursor.fetchall()
```

доопределим count_users(..)/search_users(..)

GET TOKEN/CREATE BOT

BOT LINK: t.me/shopcenter_328u423bot
TOKEN: 6628845347:AAFg6qKw_--OPZLzIL_ZDzSdZBAyoIvUaPk

в **main.py** импортируем библиотеки:

```
import telebot
import os
from sql import SQL

TOKEN = '6628845347:AAFg6qKw_--OPZLzIL_ZDzSdZBAyoIvUaPk'
bot = telebot.TeleBot(TOKEN)
```

<main.py> START

```python
def create_sql_instance():
    return SQL("users")
```

```python
@bot.message_handler(commands=['start'])
def handle_start(message):
    user_id = message.chat.id
    user_data = {'user_id': user_id, 'user_name': message.chat.username, 'first_name': message.chat.first_name,
                 'last_name': message.chat.last_name}
    sql_instance = create_sql_instance()
    if not sql_instance.search_users(keyword=message.chat.username):
        sql_instance.add_user(user_data)
        bot.send_message(user_id, text: "Вы успешно зарегистрировались!")
    markup = telebot.types.ReplyKeyboardMarkup(resize_keyboard=True)
    markup.row(*args: '👚 Лента товаров', 'Профиль')
    bot.send_message(user_id, text: "Добро пожаловать!", reply_markup=markup)
```

<main.py> ITEMS/PROFILE

```python
@bot.message_handler(func=lambda message: message.text == '👜 Лента товаров')
def handle_products_feed(message):
    user_id = message.chat.id
    sql_instance = create_sql_instance()
    # Получаем список товаров из базы данных
    # products = sql_instance.fetch_all_records_from_products_table()
    products = [
        {'product_name': 'Товар1', 'price': 10, 'currency': 'USD'},
        {'product_name': 'Товар2', 'price': 20, 'currency': 'USD'},
        {'product_name': 'Товар3', 'price': 30, 'currency': 'USD'},
        {'product_name': 'Товар4', 'price': 40, 'currency': 'USD'},
        {'product_name': 'Товар5', 'price': 50, 'currency': 'USD'},
    ]
    if not products:
        bot.send_message(user_id, text: "К сожалению, товаров в базе данных нет.")
        return
    # Отправляем первые три товара
    send_products(user_id, products[:3], offset: 0)

@bot.message_handler(func=lambda message: message.text == 'Профиль')
def handle_products_feed(message):
    user_id, user_name = message.chat.id, message.chat.username
    sql_instance = create_sql_instance()
    info = sql_instance.get_user_info(user_name)
    bot.send_message(message.chat.id, text: f"Информация по твоему профилю:\n\n{info}")
```

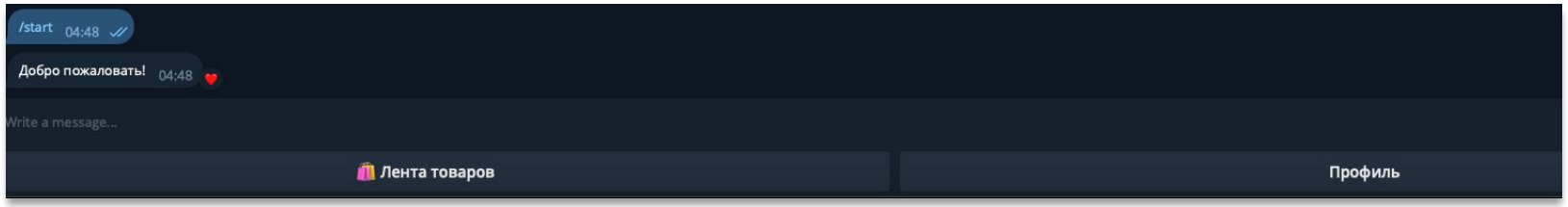
<main.py> SEND_PRODUCT/CALLBACK

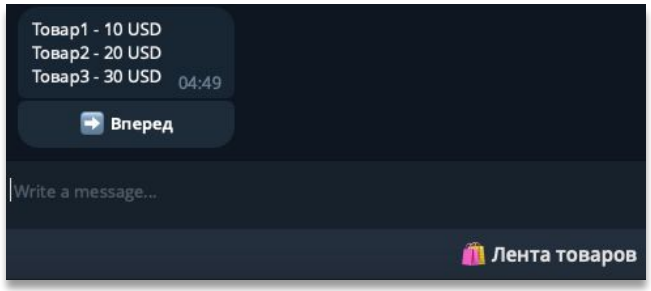
```python
@bot.message_handler(func=lambda message: True)
def handle_unknown(message):
    bot.send_message(message.chat.id, text: "Извините, я не понимаю ваш запрос. Выберите команду из меню.")

3 usages
def send_products(user_id, products, offset):
    product_messages = []
    for product in products:
        product_messages.append(f"{product['product_name']} - {product['price']} {product['currency']}")
    # Формируем сообщение с товарами
    message_text = "\n".join(product_messages)
    markup = telebot.types.InlineKeyboardMarkup()
    if offset > 0:
        markup.add(telebot.types.InlineKeyboardButton(text='⬅️ Назад', callback_data=f'prev_{offset}'))
    else:
        markup.add(telebot.types.InlineKeyboardButton(text='➡️ Вперед', callback_data=f'next_{offset}'))
    bot.send_message(user_id, message_text, reply_markup=markup)
```

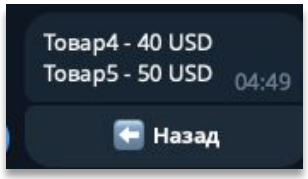
```python
@bot.callback_query_handler(func=lambda call: True)
def callback_handler(call):
    user_id = call.message.chat.id
    sql_instance = create_sql_instance()
    products = [
        {'product_name': 'Товар1', 'price': 10, 'currency': 'USD'},
        {'product_name': 'Товар2', 'price': 20, 'currency': 'USD'},
        {'product_name': 'Товар3', 'price': 30, 'currency': 'USD'},
        {'product_name': 'Товар4', 'price': 40, 'currency': 'USD'},
        {'product_name': 'Товар5', 'price': 50, 'currency': 'USD'},
    ]
    if call.data.startswith('prev_'):
        offset = int(call.data.split('_')[1]) - 3
        send_products(user_id, products[offset:offset + 3], offset)
    elif call.data.startswith('next_'):
        offset = int(call.data.split('_')[1]) + 3
        send_products(user_id, products[offset:offset + 3], offset)
```

```python
if __name__ == "__main__":
    bot.polling(none_stop=True)
```

<working>







FULL CODE: *https://disk.yandex.ru/d/7dQ_Mdms_Sel2Q*