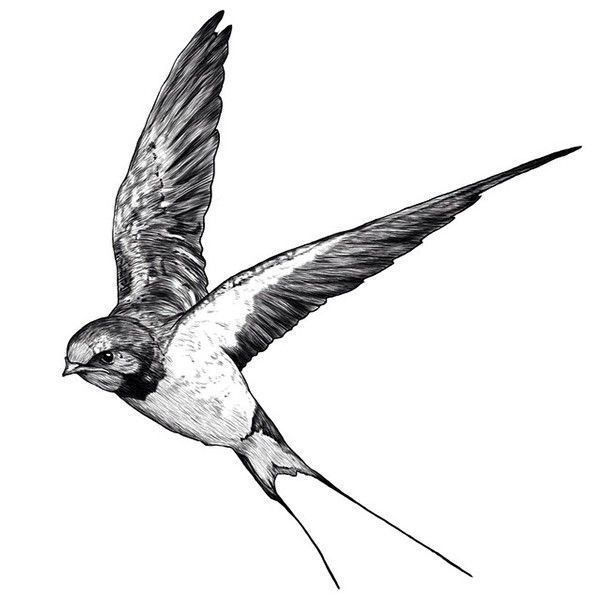
Лабораторный практикум по

дисциплине

Введение в информационные технологии



Павликов А.Е.

Чайка А.Д.

Мкртчян Г.М.

Аршинов Е.А.

Содержание

[**Лабораторная работа №1: Введение в Python и Git**](#_3y0v7nkq3xvy) **2**

[**Лабораторная работа №2: Создание приложения с метео-информацией**](#_l4a2tjhvsc4p) **8**

[**Лабораторная работа №3: Создание базы данных**](#_277rokpg3hpw) **13**

[**Лабораторная работа №4: Создание системы авторизации в веб-приложении**](#_3oodrtejif3j) **17**

[**Лабораторная работа №5: Создание системы регистрации в веб-приложении**](#_3adqbu51wzry) **26**

[**Лабораторная работа №6: Создание оконного приложения-калькулятора**](#_haq8eve8m8lh) **29**

[**Лабораторная работа №7: Создание telegram-бота с расписанием**](#_kydkno159aa6) **33**

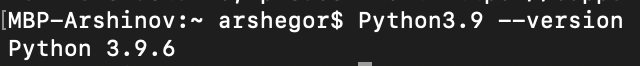
[**Лабораторная работа №8: Создание визуального интерфейса для базы данных**](#_45zo50ubzafz) **39**

### 

### **Лабораторная работа №1: Введение в Python и Git**

1. **Устанавливаем Python**

* Открываем терминал
* Вводим следующие команды:

* **sudo apt update** – обновляем список пакетов, доступных в репозиториях
* **sudo apt install software-properties-common**
* **sudo add-apt-repository ppa:deadsnakes/ppa** – добавляем deadsnakes PPA к списку источников в нашей системе
* **sudo apt install python3.9** – устанавливаем Python 3.9
* **python3.9 --version** – проверяем корректность установки

Если на экране мы видим сообщение, подобное показанному на предыдущем снимке экрана, Python 3.9 установлен в нашей Ubuntu, и мы можем начать его использовать.

1. **Устанавливаем Git**
   * В терминале вводим следующую команду:

* **sudo apt install git** – устанавливаем Git

1. **Устанавливаем Pycharm**
   * В терминале вводим следующую команду:

* **sudo snap install pycharm-community --classic**
* Ищем в приложениях PyCharm и открываем его

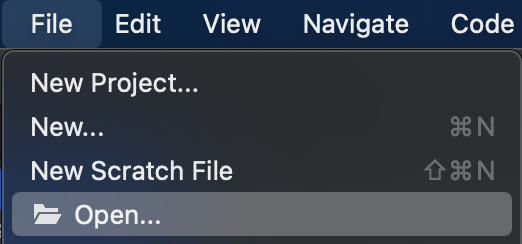
1. **Создаем и настраиваем директорию проекта**
   * В терминале вводим следующие команды:

* **mkdir MyApp** – создаем директорию с именем "MyWebApp"
* **cd MyApp** – переходим в директорию

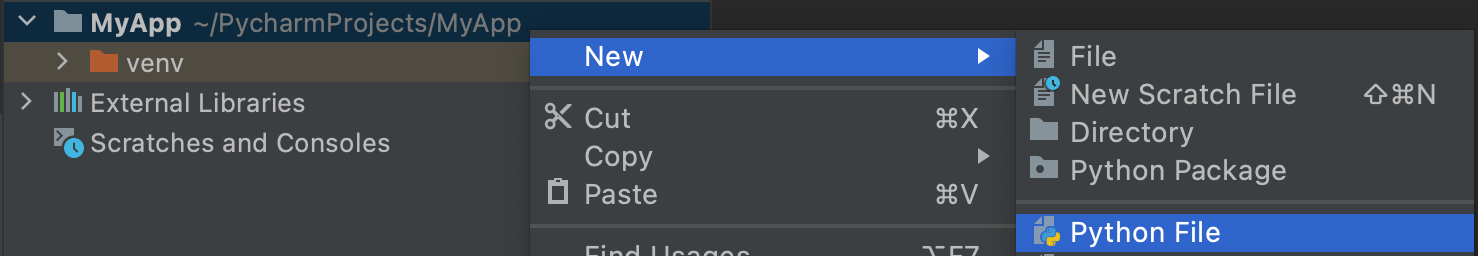
* После создания директории создаем и активируем виртуальное окружение Python
* **python3 -m venv venv** – создаем виртуальное окружение
* **source ./venv/bin/activate** – активируем виртуальное окружение

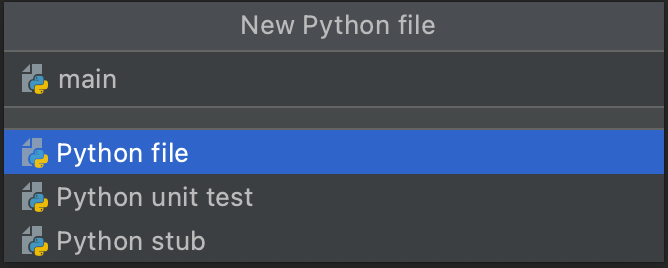
1. **Открываем директорию в PyCharm:**

* Открываем нашу директорию с именем MyApp:



1. **Создаем новый файл с именес main.py в папке MyApp**

****

****

1. **Решим простую задачу**

Задача: В программе содержится некий список с длинами сторон треугольника. Необходимо найти комбинацию из трех длин, совместив которые получается максимальная площадь треугольника.

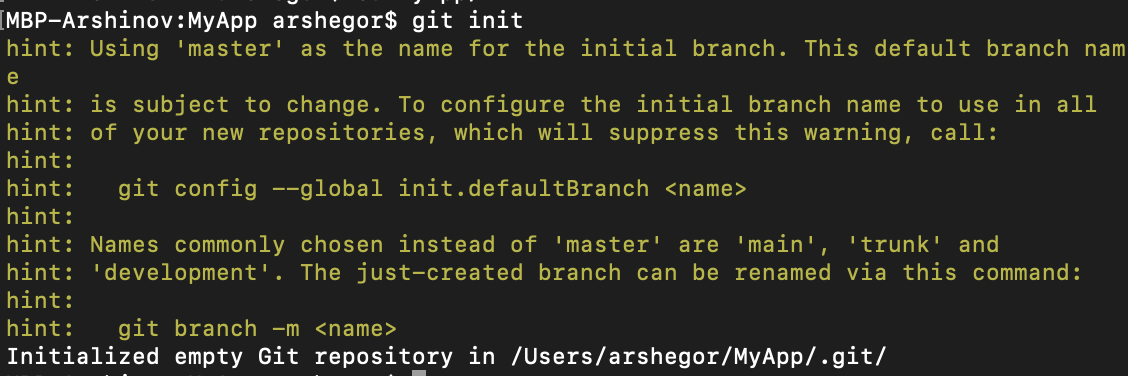
Решение:

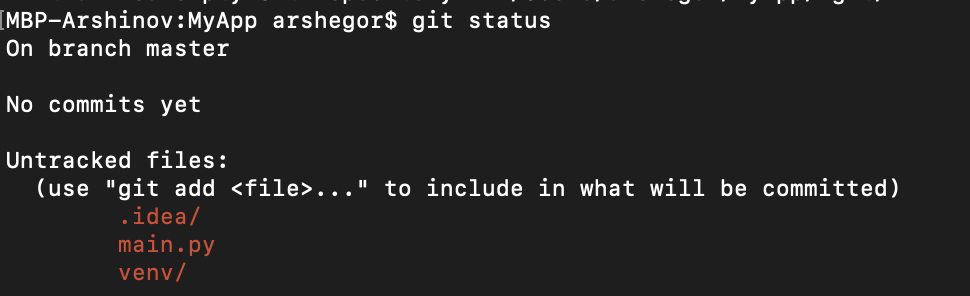
* Копируем следующий код в файл main.py

| sides = [3, 2, 4, 7, 5, 12, 11, 13, 15, 16, 14, 14]   sides = sorted(sides, reverse=True)   smax = 0 for i in range(len(sides)):  for j in range(i + 1, len(sides)):  for k in range(j + 1, len(sides)):  a = sides[i]  b = sides[j]  c = sides[k]  if a + b > c and a + c > b and b + c > a:  p = (a + b + c) / 2  s = (p \* (p - a) \* (p - b) \* (p - c)) \*\* (1 / 2)  if s > smax:  smax = s   print("Максимальная площадь треугольника", smax) |
| --- |

* Запускаем нашу программу

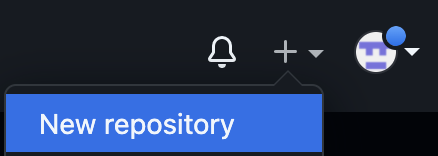
### 

1. **Зафиксируем изменения в нашем проекте в репозитории Git**
   * В терминале, в нашей директории инициализируем пустой репозиторий – вводим команду **git init**

* Проверяем статус нашего репозитория – **git status**
* Добавим в индекс файл main.py – **git add main.py**

Мы работали с только с файлом main.py поэтому папки ./idea и venv/ мы не будем добавлять в индекс.

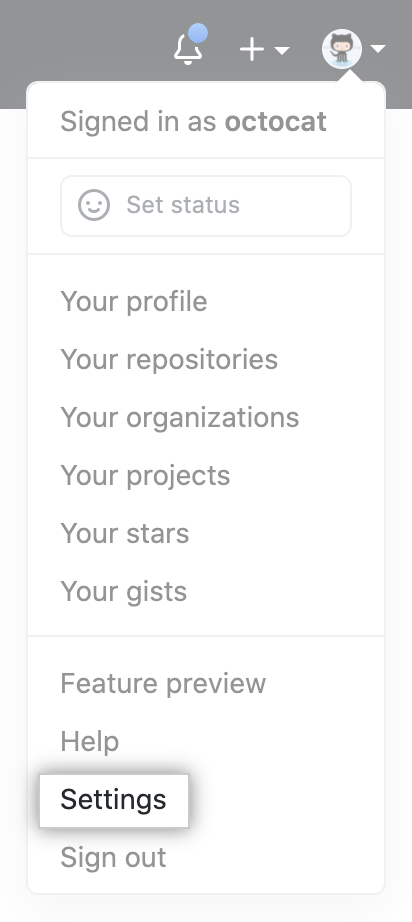
* Зафиксируем изменения - **git commit -m "Find max square of triangles"**

1. **Зафиксируем изменения локального репозитория в удаленном репозитории**
   * Регистрируемся на сайте github.com
   * Создаем новый репозиторий
   * Копируем url-адрес репозитория
   * Добавляем удаленный репозиторий – **git remote add origin <url-адрес репозитория>**

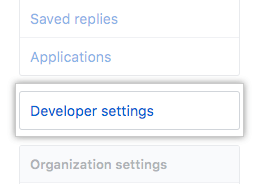
**Если Вас просят ввести токен то выполните следующие шаги:**

Подтвердите свой адрес электронной почты, если он еще не был подтвержден.

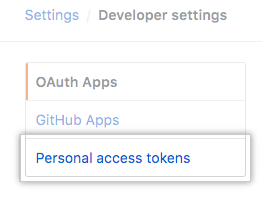
В правом верхнем углу любой страницы щелкните фотографию своего профиля, затем нажмите "Настройки".



На левой боковой панели нажмите Настройки разработчика.



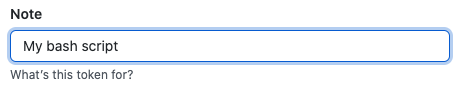
На левой боковой панели выберите Маркеры личного доступа.



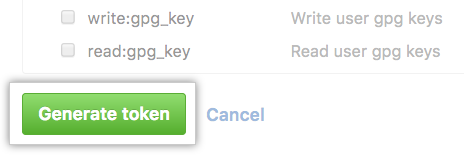
Нажмите кнопку Создать новый токен.

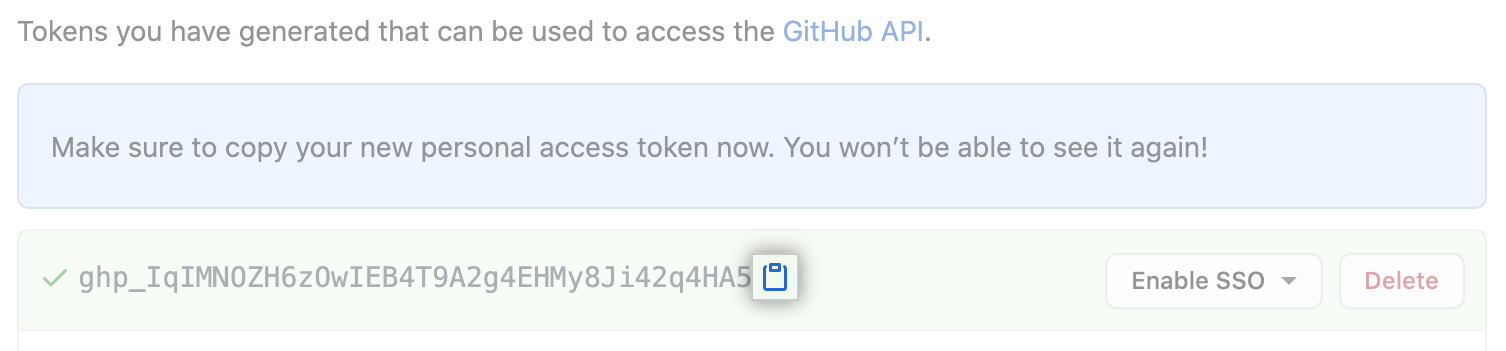


Дайте вашему токену описательное имя.



Нажмите кнопку Создать токен.





* + Фиксируем изменения – **git push origin master**

**Домашнее задание:**

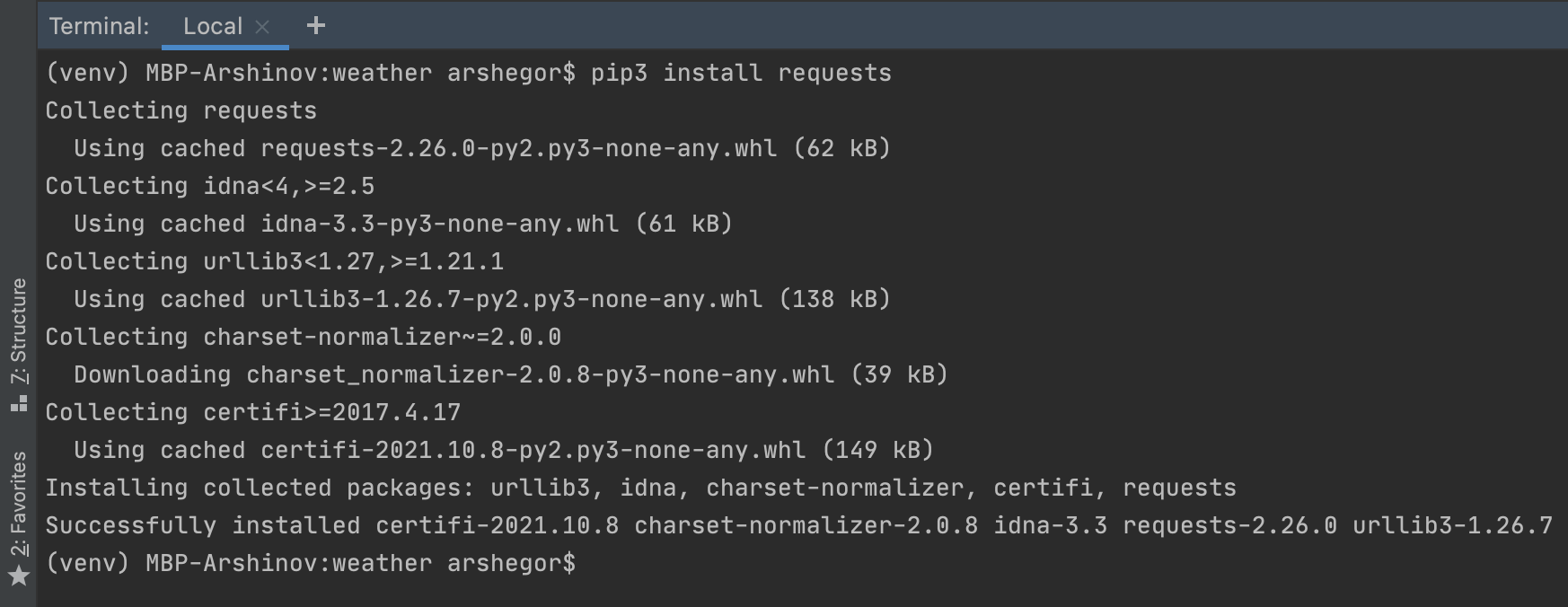
Задача: На вход программе подаются 3 коэффициента квадратного уравнения. Программа должна находить корни квадратного уравнения.

Залить данную задачу на локальный Git и GitHub.

### **Лабораторная работа №2: Создание приложения с метео-информацией**

1. **Создаем новый проект в PyCharm**
2. **Устанавливаем библиотеку requests**
   * В терминале PyCharm вводим следующую команду:

**pip3 install requests**

****

Библиотека requests является стандартным инструментом для составления HTTP-запросов в Python. Простой и аккуратный API значительно облегчает трудоемкий процесс создания запросов.

1. **Регистрируемся на сайте** [**openweathermap.org**](http://openweathermap.org)
2. **В файле main.py импортируем библиотеку requests**

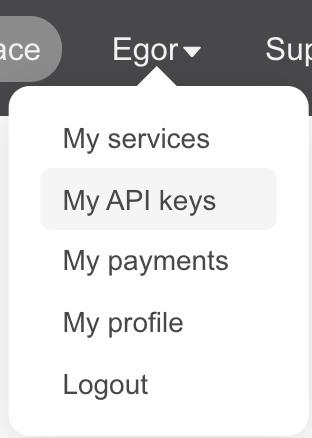
| **import requests** |
| --- |

1. **Далее запоминаем в переменные город, который нас интересует и APPID**

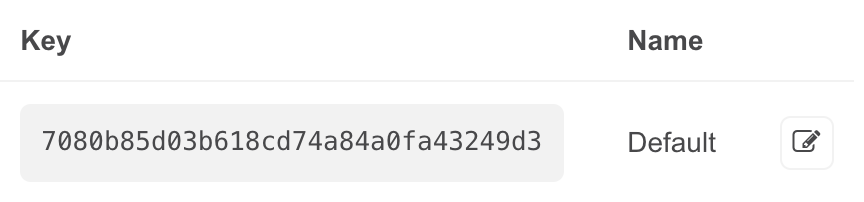
| **city = "Moscow,RU" appid = "Ваш APPID"** |
| --- |

**Получение APPID:**

* Заходим в **My API keys**

****

* Копируем ключ

****

1. **Отправляем запрос на сервис и получаем данные**

| **res = requests.get("http://api.openweathermap.org/data/2.5/weather",  params={'q': s\_city, 'units': 'metric', 'lang': 'ru', 'APPID': appid}) data = res.json()** |
| --- |

GET является одним из самых популярных HTTP методов. Метод GET указывает на то, что происходит попытка извлечь данные из определенного ресурса. Для того, чтобы выполнить запрос GET, используется requests.get().

Параметр q используется для указания города.

Параметр units используется для указания системы измерений (нам подойдет метрическая).

Параметр lang используется для указания языка отображения данных.

Параметр APPID необходимо указать, чтобы сервис не отклонил наш запрос, а принял нас как зарегистрированных пользователей.

JSON построен на двух структурах:

* Набор пар «имя-значение». Они могут быть реализованы как объект, запись, словарь, хеш-таблица, список «ключей-значений» или ассоциативный массив.
* Упорядоченный список значений. Его реализуют в виде массива, вектора, списка или последовательности.

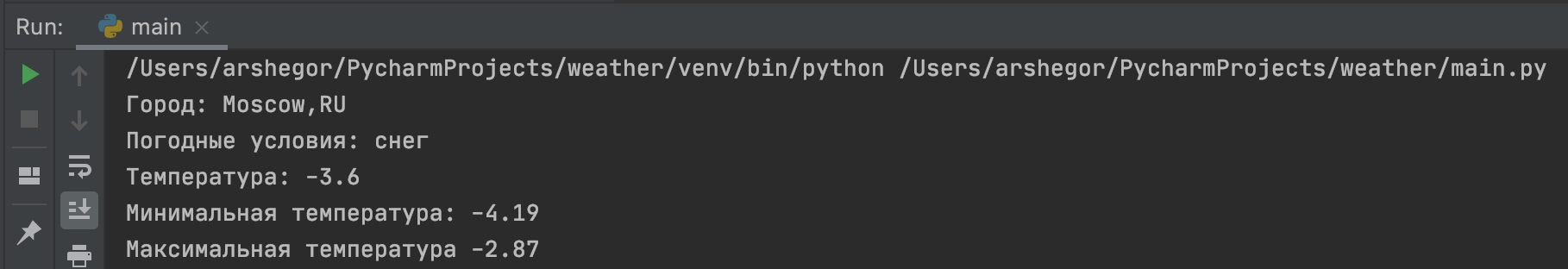
Для сохранения результатов используем переменную data. Так как сервис отдает нам информацию в формате json, нам необходимо использовать метод json() для декодирования информации полученной от сервиса.

1. **Выводим информацию в удобном для восприятия виде**

| **print("Город:", s\_city) print("Погодные условия:", data['weather'][0]['description']) print("Температура:", data['main']['temp']) print("Минимальная температура:", data['main']['temp\_min']) print("Максимальная температура", data['main']['temp\_max'])** |
| --- |

Так как в переменной data хранится словарь json, то данные из него можно брать подобно обычному словарю Python.

1. **Проверяем работу программы**

****

1. **Узнаем прогноз погоды на неделю**

| **res = requests.get("http://api.openweathermap.org/data/2.5/forecast",  params={'q': s\_city, 'units': 'metric', 'lang': 'ru', 'APPID': appid}) data = res.json() print("Прогноз погоды на неделю:") for i in data['list']:  print("Дата <", i['dt\_txt'], "> \r\nТемпература <", '{0:+3.0f}'.format(i['main']['temp']), "> \r\nПогодные условия <", i['weather'][0]['description'], ">")  print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")** |
| --- |

Чтобы узнать прогноз погоды на неделю нам необходимо отправлять запрос не на путь "/weather", а на путь "/forecast"

**Домашнее задание:**

* Изучить документацию на сайте <https://openweathermap.org/api>
* Вывести в текущем и недельном прогнозе скорость ветра и видимость.

### **Лабораторная работа №3: Создание базы данных**

1. Устанавливаем PostgreSQL

Установите пакет Postgres вместе с пакетом -contrib, который содержит дополнительные утилиты и функциональные возможности:

* **sudo apt install postgresql postgresql-contrib**

По умолчанию Postgres использует концепцию «ролей» для выполнения аутентификации и авторизации.

В ходе установки была создана учетную запись пользователя postgres, которая связана с используемой по умолчанию ролью postgres. Существует несколько способов использования этой учетной записи для доступа к Postgres. Один из способов — переход к учетной записи postgres на вашем сервере с помощью следующей команды:

* **sudo -i -u postgres**

Затем вы можете получить доступ к командной строке Postgres с помощью команды:

* **psql**

1. Создание базы данных

Чтобы выполнять базовые действия в СУБД, нужно знать Structured Query Language (SQL).

Для создания базы данных используется команда create database. В приведенном ниже примере создается база данных с именем mtuci\_db.

| **CREATE DATABASE mtuci\_db** |
| --- |

Для подключения к созданной базе данных необходимо выполнить команду

\c mtuci\_db

1. **Создание таблиц**

Таблицы являются объектами, которые содержат все данные в базах данных. В таблицах данные логически организованы в виде строк и столбцов по аналогии с электронной таблицей. Каждая строка представляет собой уникальную запись, а каждый столбец — поле записи. Например, таблица, содержащая данные о сотрудниках компании, может иметь строку для каждого сотрудника и столбцы, представляющие сведения о сотрудниках (например, его идентификационный номер, имя, адрес)

Вы можете создать таблицу, указав её имя и перечислив все имена столбцов и их типы:

CREATE TABLE student\_group (id SERIAL PRIMARY KEY, numb varchar NOT NULL, chair varchar NOT NULL);

SERIAL - целое число, которое увеличивается на 1 при добавлении новой записи

PRIMARY KEY - ограничение первичного ключа означает, что образующий его столбец или группа столбцов может быть уникальным идентификатором строк в таблице. Для этого требуется, чтобы значения были одновременно уникальными и отличными от NULL.

integer - целые числа

varchar(n) - строка ограниченной переменной длины

NOT NULL - ограничение на недопустимость ввода пустого значения

1. **Добавление записей в таблицу**

Для добавления данных применяется команда INSERT. После INSERT INTO идет имя таблицы, затем в скобках указываются все столбцы через запятую, в которые надо добавлять данные. И в конце после слова VALUES в скобках перечисляются добавляемые значения:

| **INSERT INTO student\_group (numb, chair) VALUES ('БВТ2001', 'МКиИТ');** |
| --- |

1. **Выборка**

Для извлечения данных из таблицы используется команда SELECT. Она имеет следующий синтаксис:

| **SELECT список\_столбцов FROM имя\_таблицы;** |
| --- |

Если необходимо вывести все столбцы содержащиеся в таблице, вместо перечисления можно использовать \*.

При необходимости получить записи, соответствующие каким-то условиям, следует использовать оператор WHERE, после которого указывается условие, на основании которого производится фильтрация. Например:

| **SELECT chair FROM student\_group WHERE numb='БВТ2001';** |
| --- |

В PostgreSQL можно применять следующие операции сравнения:

=: сравнение на равенство

<>: сравнение на неравенство

<: меньше чем

>: больше чем

!<: не меньше чем

!>: не больше чем

<=: меньше чем или равно

>=: больше чем или равно

1. **Удаление записи**

Команда DELETE удаляет из указанной таблицы строки, удовлетворяющие условию WHERE. Если предложение WHERE отсутствует, она удаляет из таблицы все строки, в результате будет получена рабочая, но пустая таблица. Например:

| **DELETE FROM student\_group WHERE numb='БВТ2001';** |
| --- |

Данная команда удаляет все записи, в которых numb=’БВТ2001’.

1. **Обновление записи**

UPDATE изменяет значения указанных столбцов во всех строках, удовлетворяющих условию. В предложении SET должны указываться только те столбцы, которые будут изменены; столбцы, не изменяемые явно, сохраняют свои предыдущие значения. Например:

| **UPDATE student\_group SET numb='БИН2005' WHERE chair='СиСС';** |
| --- |

Данная команда устанавливает значение поля numb равным ’БИН2005’ всем записям, в которых поле chair равно ’СиСС’.

1. **Связи между таблицами**

Связь между таблицами устанавливает отношения между значениями в ключевых полях — часто между полями, имеющими одинаковые имена в обеих таблицах. В большинстве случаев с первичным ключом одной таблицы, являющимся уникальным идентификатором каждой записи, связывается внешний ключ другой таблицы.

Внешний ключ устанавливается для столбца из зависимой, подчиненной таблицы, и указывает на один из столбцов из главной таблицы.

Создадим еще одну таблицу, содержащую внешний ключ и свяжем ее с таблицей student\_group:

| **CREATE TABLE student (id SERIAL PRIMARY KEY, full\_name varchar NOT NULL, passport varchar(10) NOT NULL, group\_numb varchar REFERENCES student\_group(numb));** |
| --- |

### 

Домашнее задание:

Создайте следующую базу данных:

1. таблица с информацией о кафедре (id, название, деканат)
2. таблица с информацией о студенческой группе (id, название, кафедра)
3. таблица с информацией о студентах (id, имя, паспортные данные, группа).
4. Между всеми таблицами должны быть связи. Заполнить таблицу кафедра 2 записями, таблицу групп 4 записями (по 2 группы на кафедру) и в таблицу студенты по 5 студентов на группу.

### **Лабораторная работа №4: Создание системы авторизации в веб-приложении**

1. **Создание директории**

* Открываем терминал
* Вводим следующие команды:

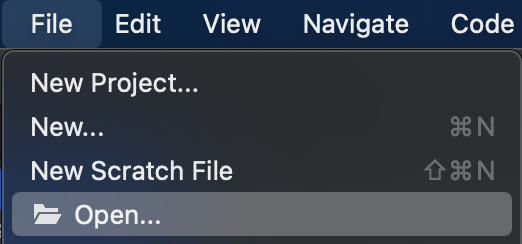
**mkdir MyWebApp** – создаем директорию с именем "MyWebApp"

**cd MyWebApp** – переходим в директорию

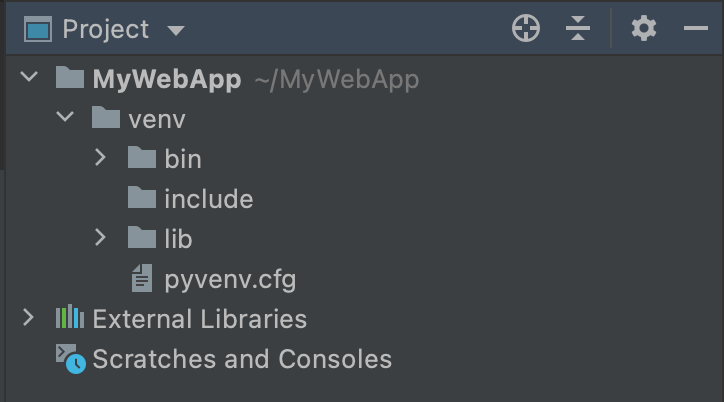
* После создания директории создаем и активируем виртуальную среду Python

1. **Открываем директорию в PyCharm:**

* Открываем нашу директорию:

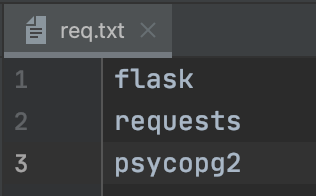


* Структура проекта на данном этапе выглядит следующим образом:



1. **Устанавливаем необходимые инструменты для дальнейшей работы:**

* Создаем файл req.txt
* Заполняем файл req.txt названиями необходимых нам инструментов



* Устанавливаем инструменты в терминале через менеджера пакетов Python pip3:

**pip3 install -r req.txt**

1. **Создаем приложение:**
   * Создаем файл app.py

* Импортируем необходимые инструменты

| import requests from flask import Flask, render\_template, request import psycopg2 |
| --- |

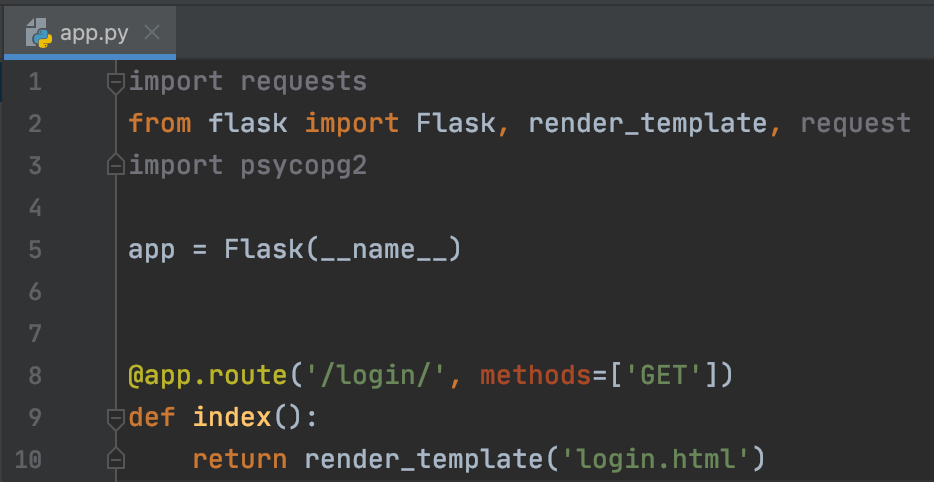
* Создаем приложение

| app = Flask(\_\_name\_\_) |
| --- |

* Создаем первый декоратор

| @app.route('/login/', methods=['GET']) def index():  return render\_template('login.html') |
| --- |

**На данном этапе файл app.py должен выглядеть следующим образом:**

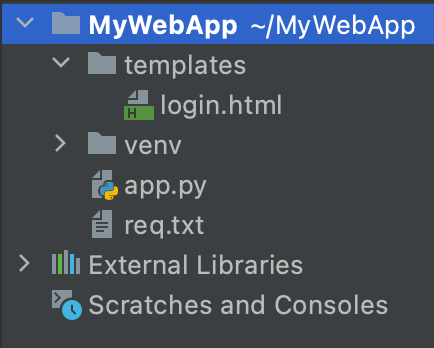


* Создаем директорию templates
* Внутри этой директории создаем файл login.html
* Удаляем содержимое файла login.html и вставляем текст разметки

| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>Login</title> </head> <body>   <form action="" method="post">  <p>  <label for="username">Username</label>  <input type="text" name="username">  </p>  <p>  <label for="password">Password</label>  <input type="password" name="password">  </p>  <p>  <input type="submit">  </p>  </form>   </body> </html> |
| --- |

* Запускаем приложение

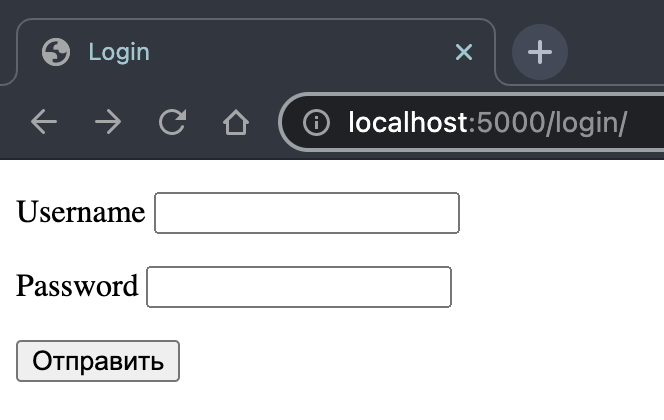
На данном этапе структура проекта выглядит следующим образом:



В терминале в папке MyWebApp запускаем команду

**flask run**

* Переходим по ссылке <http://localhost:5000/login/>



1. **Устанавливаем PostgreSQL**
   * Обновляем список пакетов

* **sudo apt update**
* Устанавливаем пакеты Postgres и contrib
* **sudo apt install postgresql postgresql-contrib**
* Запускаем сервер PostgreSQL
* **sudo -u postgres psql**

1. **Создаем базу данных**
   * Создаем базу данных

| **CREATE DATABASE service\_db;** |
| --- |

* + Подключаемся к базе данных

| **\c service\_db** |
| --- |

* + Создаем схему

| **CREATE SCHEMA service;** |
| --- |

* + Создаем таблицу пользователей

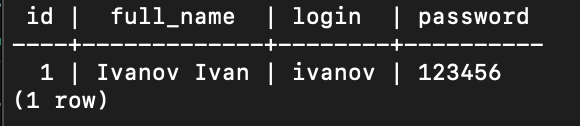
| **CREATE TABLE service.users (id SERIAL NOT NULL, full\_name VARCHAR NOT NULL, login VARCHAR NOT NULL, password VARCHAR NOT NULL);** |
| --- |

* + Заполняем таблицу пользователей

| **INSERT INTO service.users (full\_name, login, password) VALUES ('<Полное имя пользователя>','<логин>', '<пароль>');** |
| --- |

* Проверяем заполнение таблицы

| **SELECT \* FROM service.users;** |
| --- |

****

1. **Модернизируем приложение:**
   * В файл app.py добавляем подключение к базе данных сразу после строки "app = Flask(\_\_name\_\_)"

| conn = psycopg2.connect(database="service\_db",  user="postgres",  password="пароль",  host="localhost",  port="5432") |
| --- |

* Добавляем курсор для обращения к базе данных

| cursor = conn.cursor() |
| --- |

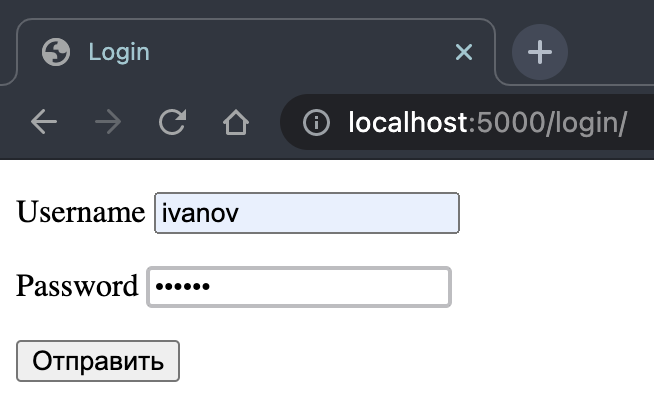
* Создаем еще один декоратор

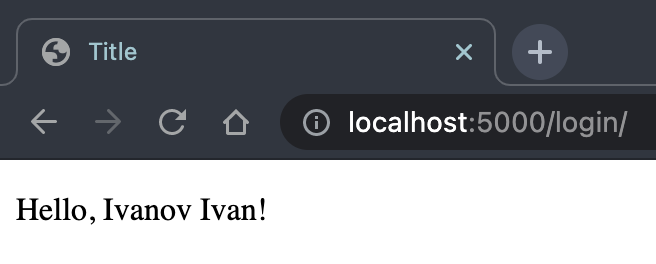
| @app.route('/login/', methods=['POST']) def login():  username = request.form.get('username')  password = request.form.get('password')  cursor.execute("SELECT \* FROM service.users WHERE login=%s AND password=%s", (str(username), str(password)))  records = list(cursor.fetchall())   return render\_template('account.html', full\_name=records[0][1]) |
| --- |

* Создаем файл account.html

| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>Title</title>  <form action="" method="post">  {% if full\_name %}  <p>Hello, {{full\_name}}! </p>  {% endif %}   </p>   </form> </head> <body>  </body> </html> |
| --- |

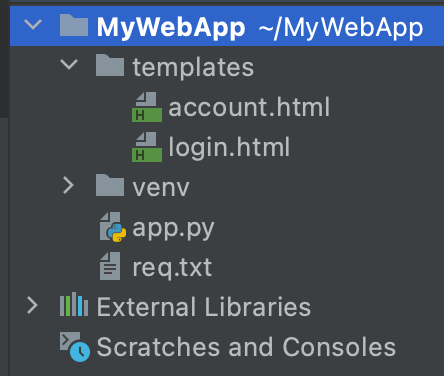
1. **Посмотрим что получилось**



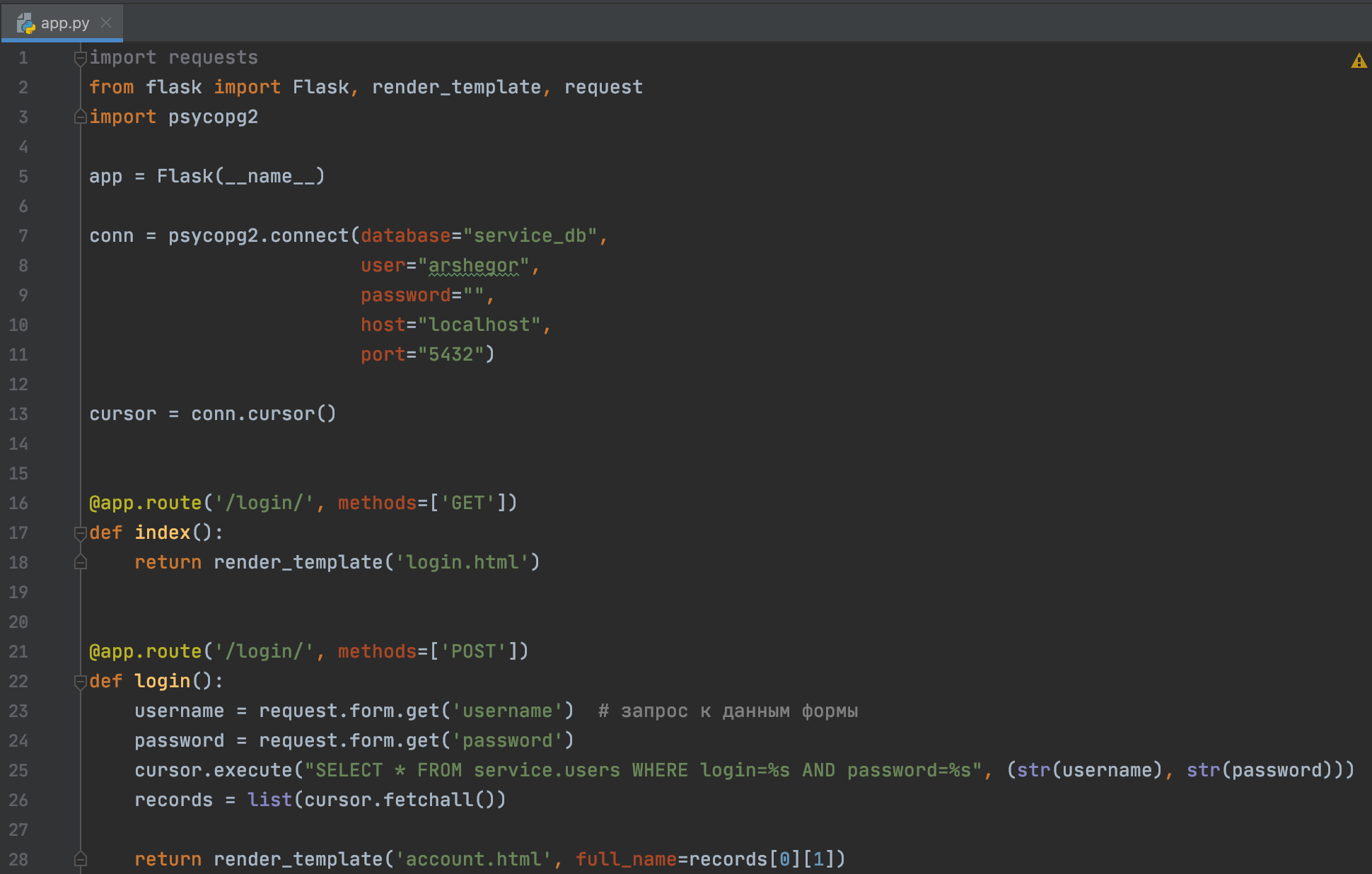


1. **Домашнее задание**
   * Дополнить таблицу users в базе данных до 10 пользователей
   * Сделать обработку исключения на ввод пустого логина и пароля
   * Сделать обработку исключения на отсутствие пользователя в базе данных
   * Вывести на странице аккаунта помимо имени пользователя его логин и пароль

**Структура проекта выглядит следующим образом:**



**Файл app.py выглядит следующим образом:**



### 

### **Лабораторная работа №5: Создание системы регистрации в веб-приложении**

1. **В файле app.py импортируем метод "redirect", отвечающий за перенаправление на другой путь:**

**from flask import Flask, render\_template, request, redirect**

1. **Удалите существующие декораторы из файла app.py, чтобы сократить объем программы:**

| @app.route('/login/', methods=['GET']) def index():  return render\_template('login.html') |
| --- |

1. **Модернизируем декоратор для пути "/login/":**

| @app.route('/login/', methods=['POST', 'GET']) def login():  if request.method == 'POST':  if request.form.get("login"):  username = request.form.get('username') password = request.form.get('password')  cursor.execute("SELECT \* FROM service.users WHERE login=%s AND password=%s", (str(username), str(password)))  records = list(cursor.fetchall())   return render\_template('account.html', full\_name=records[0][1])  elif request.form.get("registration"):  return redirect("/registration/")   return render\_template('login.html') |
| --- |

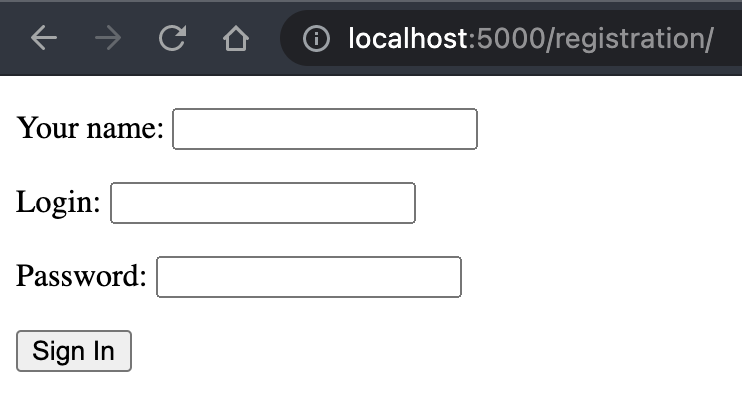
1. **В папке templates изменяем файл login.html, чтобы он выглядел следующим образом.**

| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>Login</title> </head> <body>  <form action="" method="post">  <p>  <label for="username">Username</label>  <input type="text" name="username">  </p>  <p>  <label for="password">Password</label>  <input type="password" name="password">  </p>  <p>  <input type="submit" value="login" name="login">  <input type="submit" value="registration" name="registration">  </p>   </form>  </body> </html> |
| --- |

1. **В папке templates создадим файл registration.html и вставим в него следующий код:**

| <!DOCTYPE html> <html lang="en"> <head>  <meta charset="UTF-8">  <title>Registration</title> </head> <body>  <form action="" method="post">  <p>  <label for="name">Your name:</label>  <input type="text" name="name">  </p>  <p>  <label for="login">Login:</label>  <input type="text" name="login">  </p>  <p>  <label for="password">Password:</label>  <input type="password" name="password">  </p>  <p>  <input type="submit" value="Sign In">  </p>  </form>  </body> </html> |
| --- |

1. **Запускаем приложение, на данном этапе оно должно выглядеть следующим образом:**



1. **Добавляем еще один декоратор, отвечающий за путь "/registration/", в файл app.py:**

| @app.route('/registration/', methods=['POST', 'GET']) def registration():  if request.method == 'POST':  name = request.form.get('name')  login = request.form.get('login')  password = request.form.get('password')   cursor.execute('INSERT INTO service.users (full\_name, login, password) VALUES (%s, %s, %s);',  (str(name), str(login), str(password)))  conn.commit()   return redirect('/login/')   return render\_template('registration.html') |
| --- |

1. **Проверяем таблицу users в базе данных на наличие нового пользователя в системе**

**Домашнее задание:**

Реализовать обработку всех исключений при регистрации

### 

### Лабораторная работа №6: Создание оконного приложения-калькулятора

1. **Создаем новый проект в PyCharm с названием Calculator**
2. **Устанавливаем библиотеку для создания оконных приложений PyQt5**

pip3 install PyQt5

1. **В проекте создаем новый файл с названием calculator.py**
2. **Импортируем необходимые библиотеки для создания приложения**

| import sys from PyQt5.QtWidgets import QApplication, QWidget, QLineEdit, QHBoxLayout, QVBoxLayout, QPushButton |
| --- |

Библиотека sys используется для получения информации об операционной системе.

Библиотека PyQt5 используется для создания оконных приложений.

PyQt5 используется в нашем проекте не полностью для уменьшения объема зависимостей приложения. Поэтому мы импортируем лишь некоторые классы из нее.

QApplication – управляет потоком управления и основными настройками приложения с графическим интерфейсом

QWidget – является базовым классом для всех объектов пользовательского интерфейса

QLineEdit – виджет, который разрешает вводить и редактировать одну строку текста

QHBoxLayout – выстраивает виджеты по горизонтали

QVBoxLayout – выстраивает виджеты по вертикали

QPushButton – кнопка, на которую можно нажимать

1. **Создаем класс Calculator и наследуем его от класса QWidgets**

| class Calculator(QWidget):  def \_\_init\_\_(self):  super(Calculator, self).\_\_init\_\_() |
| --- |

1. **Внутри конструктора создаем оси выравнивания**

| self.vbox = QVBoxLayout(self)  self.hbox\_input = QHBoxLayout()  self.hbox\_first = QHBoxLayout()  self.hbox\_result = QHBoxLayout()   self.vbox.addLayout(self.hbox\_input)  self.vbox.addLayout(self.hbox\_first)  self.vbox.addLayout(self.hbox\_result) |
| --- |

Вертикальная ось будет главной в окне.

К ней привязываем горизонтальные оси выравнивания с помощью функции addLayout()

1. **В конструкторе создаем виджеты и привязываем их к соответствующим осям выравнивания**

| self.input = QLineEdit(self)  self.hbox\_input.addWidget(self.input)   self.b\_1 = QPushButton("1", self)  self.hbox\_first.addWidget(self.b\_1)   self.b\_2 = QPushButton("2", self)  self.hbox\_first.addWidget(self.b\_2)   self.b\_3 = QPushButton("3", self)  self.hbox\_first.addWidget(self.b\_3)   self.b\_plus = QPushButton("+", self)  self.hbox\_first.addWidget(self.b\_plus)   self.b\_result = QPushButton("=", self)  self.hbox\_result.addWidget(self.b\_result) |
| --- |

Привязка виджетов к осям осуществляется с помощью функции addWidget()

1. **Создаем события, отвечающие за реакции на нажатия по кнопкам**

| self.b\_plus.clicked.connect(lambda: self.\_operation("+"))  self.b\_result.clicked.connect(self.\_result)   self.b\_1.clicked.connect(lambda: self.\_button("1"))  self.b\_2.clicked.connect(lambda: self.\_button("2"))  self.b\_3.clicked.connect(lambda: self.\_button("3")) |
| --- |

Функция connect(<имя\_функции/метода>), вызывает функцию/метод с именем указанным в аргументах. В указанную функцию/метод нельзя передавать аргументы. Для решения этой проблемы используем lambda-функции.

1. **Создаем метод класса для обработки кнопок, отвечающих за ввод цифр в линию ввода текста**

| def \_button(self, param):  line = self.input.text()  self.input.setText(line + param) |
| --- |

Уже существующая строка в линии ввода конкатенируется с аргументом param и устанавливается как отображаемый в линии ввода текст.

1. **Создаем метод класса для обработки нажатия на кнопку математической операции**

| def \_operation(self, op):  self.num\_1 = int(self.input.text())  self.op = op  self.input.setText("") |
| --- |

Запоминаем первое введенное число в целочисленном типе данных.

Запоминаем в качестве операции аргумент op.

Очищаем линию ввода.

1. **Создаем метод класса для обработки нажатия на кнопку результата**

| def \_result(self):  self.num\_2 = int(self.input.text())  if self.op == "+":  self.input.setText(str(self.num\_1 + self.num\_2)) |
| --- |

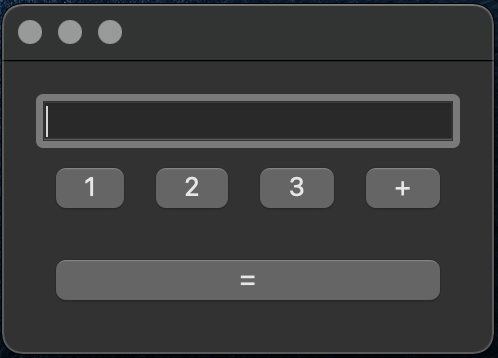
Запоминаем второе введенное число в целочисленном типе данных.

Производим вычисление в зависимости от операции и устанавливаем его в качестве текста в линию ввода.

1. **Запускаем приложение**

| app = QApplication(sys.argv)  win = Calculator() win.show()  sys.exit(app.exec\_()) |
| --- |

Приложение должно выглядеть подобным образом:



1. **Домашнее задание:**
   * Обработать все возможные исключения
   * Добавить кнопку для добавления плавающей точки
   * Добавить кнопки для математических операций вычитания, умножения, деления
   * Создать полноценный калькулятор
   * Создать для этих кнопок методы-обработчики

### **Лабораторная работа №7: Создание telegram-бота с расписанием**

**Техническое задание:**

Создать телеграм-бота с расписанием для Вашей группы.

**Минимальные требования к разрабатываемой системе:**

1. Бот должен иметь ник-нейм формата <номер группы>\_<фамилия разработчика>\_bot.
2. Бот должен иметь имя формата <номер группы>\_<фамилия разработчика>.
3. Манера общения бота – "на Вы".
4. Использование базы данных PostgreSQL.
5. Использование pyTelegramBotAPI.
6. Использование адаптера psycopg2.
7. Формат вывода на каждый день недели:

<День недели>

\_\_\_\_\_\_\_\_\_\_\_\_

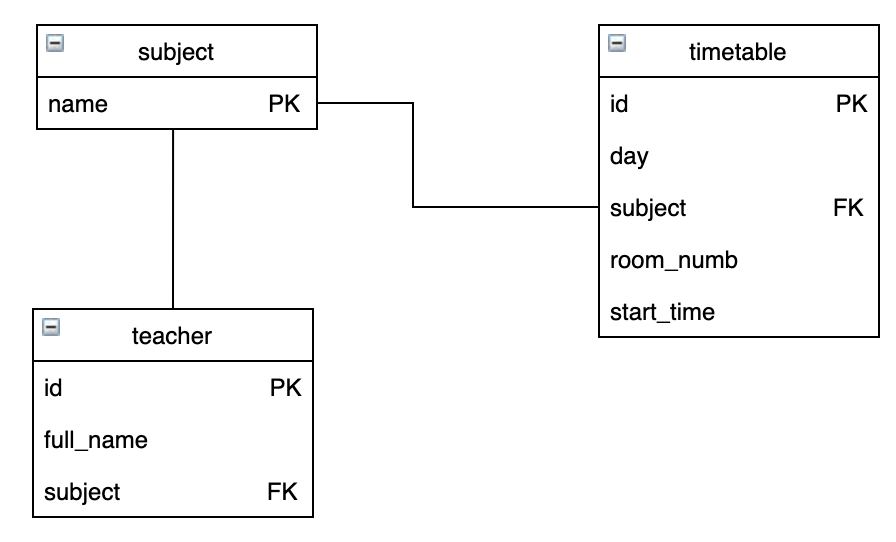
<Предмет> <Кабинет> <Время> <Преподаватель>

…

<Предмет> <Кабинет> <Время> <Преподаватель>

\_\_\_\_\_\_\_\_\_\_\_\_

1. Во время работы бота должны быть отображены следующие графические кнопки:
   1. Понедельник
   2. Вторник
   3. Среда
   4. Четверг
   5. Пятница
   6. Расписание на текущую неделю
   7. Расписание на следующую неделю
2. При нажатии на кнопку с днем недели бот должен выводить информацию из базы данных с расписанием на выбранный день текущей недели.
3. При нажатии на кнопку "Расписание на текущую неделю" бот должен выводить информацию из базы данных с расписанием на всю текущую неделю.
4. При нажатии на кнопку "Расписание на следующую неделю" бот должен выводить информацию из базы данных с расписанием на всю следующую неделю неделю.
5. При использовании команды /week бот должен выводить какая на данный момент неделя – верхняя/нижняя.
6. При использовании команды /mtuci бот должен выводить ссылку на официальный сайт МТУСИ – https://mtuci.ru/
7. При использовании команды /help бот должен выводить краткую информацию о себе, краткую документацию и список команд с их пояснениями.
8. При вводе неизвестной команды или неизвестного боту сообщения, бот должен отправлять пользователю сообщение – "Извините, я Вас не понял".
9. Структура базы данных может дополняться полями в таблицах, но при этом должна иметь следующую структуру:



**Система может быть дополнена Вашим функционалом, но должна соответствовать минимальным требованиям.**

**Создание простого Telegram-бота**

1. **Создаем новый проект с именем Simple-bot**
2. **Загружаем библиотеку для создания Telegram-ботов**

pip3 install pyTelegramBotAPI

1. **Регистрируем бота в Telegram**
   * В поисковой строке ищем бота с именем @BotFather



* + Отправляем боту команду/сообщение – /start
  + Отправляем боту команду/сообщение с именем бота, которого хотим зарегистрировать, в нашем случае <Ваша фамилия>\_<Ваше имя>\_bot
  + Для ознакомления с перечнем команд @BotFather отправляем боту команду/сообщение – /help

1. **В файл main.py импортируем библиотеки для создания back-end части бота**

| import telebot from telebot import types |
| --- |

1. **Из сообщения об успешной регистрации бота копируем токен для управления нашим ботом. В файле main.py оздаем переменную, хранящую в себе токен**

token = "Ваш токен"

1. **Создаем объект бота, к которому мы будем в дальнейшем обращаться**

bot = telebot.TeleBot(token)

В класс TeleBot передаем переменную token, для того, чтобы обращаться именно к вашему боту

1. **Создаем декоратор, отвечающий за команду /start**

| @bot.message\_handler(commands=['start']) def start(message):  keyboard = types.ReplyKeyboardMarkup()  keyboard.row("Хочу", "/help")  bot.send\_message(message.chat.id, 'Привет! Хочешь узнать свежую информацию о МТУСИ?', reply\_markup=keyboard) |
| --- |

Класс ReplyKeyboardMarkup создает пользовательскую клавиатуру с текстовыми кнопками на месте стандартной клавиатуры.

Метод row() заполняет клавиатуру кнопками.

Метод send\_message отправляет пользователю сообщение.

Аргумент message.chat.id используется для того, чтобы бот отправил сообщение тому пользователю, который отправил сообщение, на которое бот в данный момент времени отвечает.

Аргумент reply\_markup=keyboard используется для отправки пользовательской клавиатуры, для ее дальнейшего отображения.

1. **Создаем декоратор отвечающий за команду /help**

| @bot.message\_handler(commands=['help']) def start\_message(message):  bot.send\_message(message.chat.id, 'Я умею...') |
| --- |

В сообщении вы можете указать что умеет бот, включая команды, на которые он умеет реагировать.

1. **Создаем декоратор отвечающий за ответ на сообщение "Хочу"**

| @bot.message\_handler(content\_types=['text']) def answer(message):  if message.text.lower() == "хочу":  bot.send\_message(message.chat.id, 'Тогда тебе сюда - https://mtuci.ru/') |
| --- |

Данный декоратор должен стоять ниже, чем декораторы команд, так как в противном случае декораторы команд обрабатываться не будут, потому что команды в своем роде тоже текстовые сообщения.

В этом декораторе аргумент content\_types=['text'] отвечает за реакцию на текстовый тип контента сообщения.

Для проверки конкретного текста используется условная конструкция с условием message.text.lower() == "<текст>". Причем функция lower() отвечает за перевод текста в нижний регистр для удобства использования, и может применяться не только для библиотеки telebot, но и для любых строковых операций и переменных.

**Домашнее задание:**

* Создать обработку трех любых сообщений.
* Создать обработку трех любых команд.
* Обработать команду /help

### **Лабораторная работа №8: Создание визуального интерфейса для базы данных**

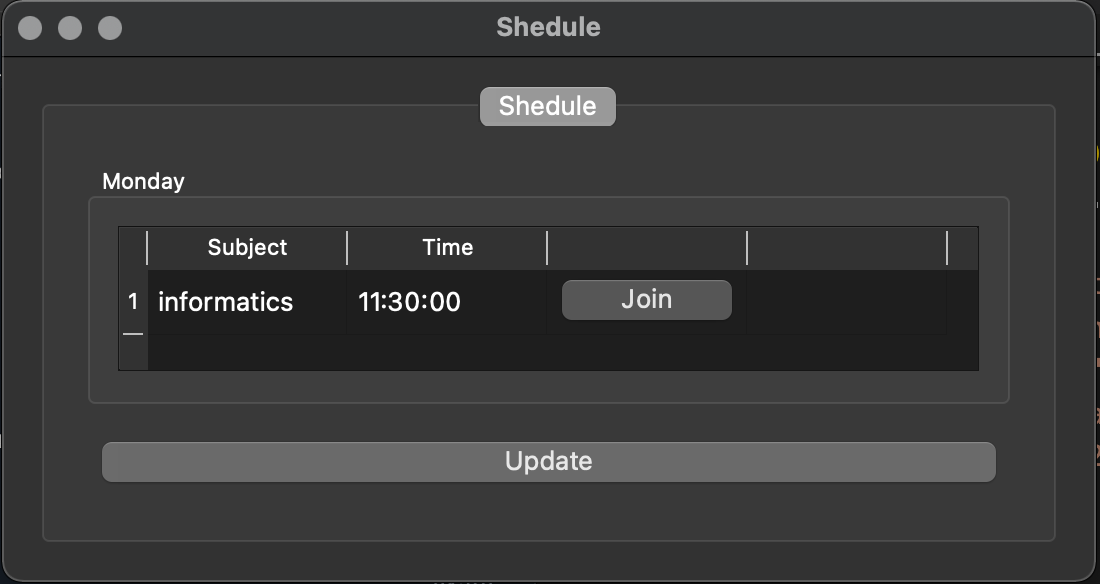
**Техническое задание:**

Создать оконное приложение позволяющее редактировать базу данных с расписанием Вашей группы.

**Минимальные требования к разрабатываемой системе:**

1. Использование библиотеки PyQt5.
2. Использование адаптера psycopg2.
3. Приложение должно иметь при себе функционал позволяющий: просматривать базу данных в удобном для пользователя формате, удалять, добавлять и изменять записи в этой же базе данных.
4. Визуальная часть должна иметь при себе:
   1. Минимум 3 вкладки, в каждой из которых содержится информация из отдельной таблицы в базе данных.
   2. Внутри каждой вкладки информация должна отображаться в виде таблиц.
   3. Внутри каждой вкладки должна отображаться кнопка с обновлением информации.
   4. Внутри каждой таблицы должны отображаться все поля из таблицы в базе данных в виде колонок
   5. Внутри каждой таблицы после каждой строки записи должны быть отображены кнопки изменения и удаления записи
   6. В конце каждой таблицы должна находиться пустая строка с кнопкой для добавления новой записи.
   7. На вкладке с расписанием дни недели должны быть указаны в отдельных таблицах.

**Пример частичной реализации:**

****

1. **Импортируем необходимые библиотеки и адаптеры**

| import psycopg2 import sys  from PyQt5.QtWidgets import (QApplication, QWidget,  QTabWidget, QAbstractScrollArea,  QVBoxLayout, QHBoxLayout,  QTableWidget, QGroupBox,  QTableWidgetItem, QPushButton, QMessageBox) |
| --- |

1. **Создаем класс MainWindow с конструктором**

| class MainWindow(QWidget):  def \_\_init\_\_(self):  super(MainWindow, self).\_\_init\_\_()   self.\_connect\_to\_db()   self.setWindowTitle("Shedule")   self.vbox = QVBoxLayout(self)   self.tabs = QTabWidget(self)  self.vbox.addWidget(self.tabs)s   self.\_create\_shedule\_tab() |
| --- |

Класс QTabWidget создает структуру, которую можно заполнять вкладками.

Вкладки это подстраницы в окне приложения. Аналогом вкладок в оконных приложениях являются вкладки в веб-браузере.

1. **Создаем метод для подключения к базе данных**

| def \_connect\_to\_db(self):  self.conn = psycopg2.connect(database="<название вашей базы данных>",  user="postgres",  password="1234",  host="localhost",  port="5432")   self.cursor = self.conn.cursor() |
| --- |

1. **Создаем метод для отображения вкладки с расписанием**

| def \_create\_shedule\_tab(self):  self.shedule\_tab = QWidget()  self.tabs.addTab(self.shedule\_tab, "Shedule")   self.monday\_gbox = QGroupBox("Monday")   self.svbox = QVBoxLayout()  self.shbox1 = QHBoxLayout()  self.shbox2 = QHBoxLayout()   self.svbox.addLayout(self.shbox1)  self.svbox.addLayout(self.shbox2)   self.shbox1.addWidget(self.monday\_gbox)   self.\_create\_monday\_table()   self.update\_shedule\_button = QPushButton("Update")  self.shbox2.addWidget(self.update\_shedule\_button)  self.update\_shedule\_button.clicked.connect(self.\_update\_shedule)   self.shedule\_tab.setLayout(self.svbox) |
| --- |

Класс QWidget() создает виджет, который будет являться вкладкой в нашем приложении. Данный класс может также использоваться для создания отдельных окон, но в нашем случае будет вкладкой.

self.tabs.addTab(self.shedule\_tab, "Shedule") добавляет в структуру с вкладками новую вкладку с названием "Shedule".

Класс QGroupBox() может группировать виджеты, он предоставляет

рамку, заголовок вверху и может отображать несколько виджетов

внутри. В нашем случае он служит исключительно в декоративных

целях

1. **Создаем метод для отображения таблицы с расписанием на понедельник**

| def \_create\_monday\_table(self):  self.monday\_table = QTableWidget()  self.monday\_table.setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)   self.monday\_table.setColumnCount(4)  self.monday\_table.setHorizontalHeaderLabels(["Subject", "Time", "", ""])   self.\_update\_monday\_table()   self.mvbox = QVBoxLayout()  self.mvbox.addWidget(self.monday\_table)  self.monday\_gbox.setLayout(self.mvbox) |
| --- |

Класс QTableWidget() создает пустую пользовательскую таблицу аналогичную таблицам Excel.

setSizeAdjustPolicy(QAbstractScrollArea.AdjustToContents)

устанавливает возможность изменения размера под размер данных в ячейке.

Метод setColumnCount() задает таблице количество колонок.

Метод setHorizontalHeaderLabels(["Название", "Название"])

задает колонкам подписи.

1. **Создаем метод для обновления таблицы с расписанием на понедельник**

| def \_update\_monday\_table(self):  self.cursor.execute("SELECT \* FROM timetable WHERE day='wednesday'")  records = list(self.cursor.fetchall())   self.monday\_table.setRowCount(len(records) + 1)   for i, r in enumerate(records):  r = list(r)  joinButton = QPushButton("Join")   self.monday\_table.setItem(i, 0,  QTableWidgetItem(str(r[0])))  self.monday\_table.setItem(i, 1,  QTableWidgetItem(str(r[2])))  self.monday\_table.setItem(i, 2,  QTableWidgetItem(str(r[4])))  self.monday\_table.setCellWidget(i, 3, joinButton)   joinButton.clicked.connect(lambda ch, num=i: self.\_change\_day\_from\_table(num))   self.monday\_table.resizeRowsToContents() |
| --- |

Заполнение таблицы происходит в цикле for для того, чтобы динамически обрабатывать изменения в количестве записей.

Метод setRowCount() задает таблице количество строк.

Кнопка joinButton не является отдельным свойством класса MainWindow, так как нам не нужно ее "запоминать". Далее интерпретатор запоминает ее с помощью функции-обработчика clicked.connect().

Метод setItem(<Номер строки>, <Номер колонки>, <Строка с данными>) записывает в ячейку с определенным адресом строковые данные.

Метод setCellWidget(<Номер строки>, <Номер колонки>, <Виджет>)

помещает в ячейку с определенным адресом виджет.

В нашем случае это кнопка "Join".

Метод resizeRowsToContents() автоматически адаптирует размеры ячеек таблицы под размер данных внутри этой ячейки. Это необходимо использовать для экономии визуального пространства.

1. **Создаем метод изменяющий запись в базе данных по нажатию на кнопку "Join"**

| def \_change\_day\_from\_table(self, rowNum, day):  row = list()  for i in range(self.monday\_table.columnCount()):  try: row.append(self.monday\_table.item(rowNum, i).text())  except: row.append(None)   try: self.cursor.execute("UPDATE SQL запрос на изменение одной строки в базе данных", (row[0],)) self.conn.commit()  except: QMessageBox.about(self, "Error", "Enter all fields") |
| --- |

Метод columnCount() возвращает количество колонок таблицы.

Конструкция item(<Номер строки>, <Номер столбца>).text() возвращает текст, записанный в определенной ячейке.

1. **Создаем метод обновляющий все таблицы на вкладке**

| def \_update\_shedule(self):  self.\_update\_monday\_table()  ...  Ваши методы обновления таблиц |
| --- |

1. **"Запускаем" наше приложение**

| app = QApplication(sys.argv) win = MainWindow() win.show() sys.exit(app.exec\_()) |
| --- |

**Система может быть дополнена Вашим функционалом, но должна соответствовать минимальным требованиям.**

### 

### 

### 