

Монитор здоровья: настольное приложение на Python и Tkinter для ведения личных медицинских показателей

Ран Максим Александрович

Приамурский государственный университет имени Шолом-Алейхема

Студент

Аннотация

В работе рассматривается разработка настольного приложения «Монитор здоровья», реализованного на языке Python с использованием библиотеки Tkinter. Приложение предназначено для ведения личных медицинских показателей (артериальное давление, уровень сахара в крови, масса тела), хранения истории измерений и визуализации динамики на графиках. Хранение данных реализовано на основе встроенной СУБД SQLite с поддержкой авторизации пользователей и привязкой записей к конкретному аккаунту. Интерфейс включает форму входа/регистрации, главный экран с блоками ввода новых измерений, таблицу истории на базе Treeview и область графиков, построенных с использованием библиотеки Matplotlib. Для удобного выбора даты используется всплывающий календарь tkcalendar.Calendar, а опасные значения показателей автоматически выделяются и сопровождаются предупреждениями. Показано, что связка Python + Tkinter + SQLite позволяет создать удобное и наглядное настольное приложение для мониторинга состояния здоровья.

Ключевые слова: Python, Tkinter, мониторинг здоровья, артериальное давление, уровень сахара, масса тела, SQLite, Matplotlib, tkcalendar.

Health Monitor: a desktop application in Python and Tkinter for personal health tracking

Ran Maksim Aleksandrovich

Sholom-Aleichem Priamursky State University

Student

Abstract

This paper presents the development of a desktop application called “Health Monitor” implemented in Python using the Tkinter library. The application is designed for recording personal health indicators (blood pressure, blood sugar level, body weight), storing measurement history and visualizing trends using charts. Data storage is based on the embedded SQLite database with user authentication and per-user data isolation. The user interface includes a login/registration screen and a main screen with sections for entering new measurements, a history table built on Treeview, and a chart area rendered using Matplotlib. A pop-up calendar component (tkcalendar.Calendar) is employed for convenient date selection, while out-of-range

values are automatically highlighted and accompanied by warning messages. The study demonstrates that the combination of Python, Tkinter and SQLite allows building a convenient and informative desktop application for personal health monitoring.

Keywords: Python, Tkinter, health monitoring, blood pressure, blood sugar, body weight, SQLite, Matplotlib, tkcalendar.

1 Введение

1.1 Актуальность

Своевременный контроль жизненно важных показателей организма — артериального давления, уровня сахара в крови, массы тела — является важным фактором профилактики и раннего выявления хронических заболеваний. Многие пациенты используют домашние тонометры, глюкометры и напольные весы, однако результаты измерений часто остаются на отдельных бумажных листах или в памяти устройства, что затрудняет анализ состояния в динамике.

При этом далеко не всем пользователям удобны сложные медицинские информационные системы или мобильные приложения, требующие регистрации в облачных сервисах. Для части пользователей более привлекательным вариантом остаётся простое настольное приложение, устанавливаемое на персональный компьютер и не требующее постоянного доступа к сети.

В этой связи актуальной является разработка компактного настольного приложения, позволяющего в удобной форме вводить, хранить и анализировать личные измерения здоровья, а также визуализировать динамику показателей наглядными графиками.

1.2 Обзор исследований

Для более глубокого знакомства с языком программирования Python и его использованием при разработке графических интерфейсов целесообразно обратиться к следующим источникам. В.А. Машков и В.И. Литвиненко анализируют применение Python для решения задач самодиагностики на системном уровне [1]. Г.Д. Бухарова и П.С. Комельских акцентируют внимание на значимости включения Python в образовательный процесс [2]. Г.С. Сейдаметов раскрывает особенности использования Python в подготовке инженеров-программистов [3], а Э.А. Усеинов демонстрирует практику применения объектно-ориентированного подхода в рамках изучения данного языка [4]. Разработка приложений с графическим пользовательским интерфейсом (GUI) на Python с использованием библиотеки Tkinter также получила широкое освещение в научных работах. В частности, А.С. Матвеева в своей статье рассматривает применение паттерна «Команда» для построения модульных и структурированных приложений на Tkinter, что способствует повышению гибкости, расширяемости, надежности и удобства использования программных систем [5].

1.3 Цель исследования

Целью работы является разработка настольного приложения «Монитор здоровья» на языке Python с использованием библиотеки Tkinter и СУБД SQLite, обеспечивающего:

- регистрацию и аутентификацию пользователей;
- внесение измерений основных показателей здоровья (давление, сахар, вес);
- хранение истории измерений с выделением опасных значений;
- визуализацию динамики показателей на графиках;
- удобный выбор даты при работе с формой ввода.

2 Материалы и методы

Для реализации приложения использовался следующий набор технологий и библиотек:

- Python — язык программирования общего назначения для реализации логики приложения и взаимодействия с базой данных;
- Tkinter — стандартная библиотека Python для создания графического интерфейса (окна, формы, таблицы, кнопки, поля ввода);
- SQLite — встраиваемая реляционная СУБД для локального хранения данных пользователей и их измерений;
- tkcalendar (Calendar) — виджет календаря для удобного выбора даты во всплывающем окне;
- Matplotlib — библиотека визуализации данных, используемая для построения графиков динамики артериального давления, уровня сахара и массы тела;
- ttk.Treeview — элемент интерфейса Tkinter для отображения табличных данных (история измерений) с поддержкой выделения строк и применения стилей.

Такой стек позволяет создать полностью автономное настольное приложение, не требующее внешних серверов и обеспечивающее все операции локально на стороне пользователя.

3 Результаты и обсуждения

Python-часть — модели и работа с базой данных

Точка входа main.py.

Файл main.py содержит минимальный, но важный код запуска приложения. В нём создаётся корневое окно Tk, инициализируется объект HealthMonitorApp из модуля gui.py и запускается главный цикл обработки событий (рис.1).

```
import tkinter as tk
from gui import HealthMonitorApp

if __name__ == "__main__":
    root = tk.Tk()
    app = HealthMonitorApp(root)
    root.mainloop()
```

Рисунок 1. Код main.py

Таким образом, вся логика приложения вынесена в отдельные модули, а main.py выступает только точкой входа.

Модуль models.py содержит два класса, описывающих предметную область приложения: пользователя (рис.2) и измерение показателей здоровья (рис.3).

```
class User:
    def __init__(self, username, password_hash, id=None):
        self.id = id
        self.username = username
        self.password_hash = password_hash

    @staticmethod
    def hash_password(password):
        return hashlib.sha256(password.encode()).hexdigest()

    def check_password(self, password):
        return self.password_hash == self.hash_password(password)
```

Рисунок 2. Класс User

Объект User хранит:

- id — идентификатор пользователя в базе данных (может быть None до записи в БД);
- username — имя пользователя (логин);
- password_hash — хешированный пароль.

Для работы с паролями использован алгоритм SHA-256 из стандартного модуля hashlib. Метод check_password() сравнивает сохранённый хеш со значением, вычисленным для введённого пароля. Таким образом, пароли никогда не хранятся в открытом виде, а проверка выполняется через сравнение хешей.

```
class HealthMeasurement:
    def __init__(self, date, systolic_bp=None, diastolic_bp=None, blood_sugar=None, weight=None, notes=""):
        self.date = date
        self.systolic_bp = systolic_bp # Систолическое давление
        self.diastolic_bp = diastolic_bp # Диастолическое давление
        self.blood_sugar = blood_sugar # Уровень сахара
        self.weight = weight # Вес
        self.notes = notes

    def is_dangerous(self):
        # Логика определения опасных значений
        dangerous = False
        if self.systolic_bp and (self.systolic_bp > 140 or self.systolic_bp < 90):
            dangerous = True
        if self.diastolic_bp and (self.diastolic_bp > 90 or self.diastolic_bp < 60):
            dangerous = True
        if self.blood_sugar and (self.blood_sugar > 7.0 or self.blood_sugar < 3.9):
            dangerous = True
        if self.weight and (self.weight > 100 or self.weight < 40): # Примерные границы
            dangerous = True
        return dangerous

    def to_dict(self):
        return {
            'date': self.date.isoformat(),
            'systolic_bp': self.systolic_bp,
            'diastolic_bp': self.diastolic_bp,
            'blood_sugar': self.blood_sugar,
            'weight': self.weight,
            'notes': self.notes
        }
```

Рисунок 3. Класс HealthMeasurement

Объект HealthMeasurement описывает одну запись измерений:

- date — дата измерения (объект datetime);
- systolic_bp — систолическое давление;
- diastolic_bp — диастолическое давление;
- blood_sugar — уровень сахара в крови (ммоль/л);
- weight — масса тела (кг);
- notes — текстовая заметка пользователя.

Ключевым методом является is_dangerous(), определяющий, выходят ли показатели за условно безопасные пределы. Пороговые значения (давление, сахар, вес) заданы ориентировочно и могут в дальнейшем настраиваться. Если хотя бы одно измерение выходит за этот диапазон, метод возвращает True. Именно это свойство дальше используется для визуального выделения «опасных» строк в таблице и вывода предупредительных сообщений. Для сериализации данных и возможного обмена в формате словаря реализованы методы:

- to_dict() — перевод объекта в словарь, при этом date сохраняется в ISO-формате (date.isoformat());
- from_dict(cls, data) — обратное восстановление объекта HealthMeasurement из словаря, где дата восстанавливается через datetime.fromisoformat.

Модуль database.py реализует класс HealthDatabase, инкапсулирующий все операции с базой данных SQLite (рис.4).

```
class HealthDatabase:
    def __init__(self, db_name='health_monitor.db'):
        self.db_name = db_name
        self.create_tables()
```

Рисунок 4. Инициализация и создание таблиц

При создании экземпляра класса автоматически вызывается метод `create_tables()`, который обеспечивает наличие необходимых таблиц (рис.5).

```
def create_tables(self):
    conn = sqlite3.connect(self.db_name)
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT UNIQUE NOT NULL,
            password_hash TEXT NOT NULL
        )
    ''')
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS measurements (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            date TEXT NOT NULL,
            systolic_bp REAL,
            diastolic_bp REAL,
            blood_sugar REAL,
            weight REAL,
            notes TEXT,
            FOREIGN KEY (user_id) REFERENCES users (id)
        )
    ''')
    conn.commit()
    conn.close()
```

Рисунок 5. Создание таблиц

Создаются:

- таблица `users` с уникальным `username` и хешем пароля;
- таблица `measurements` с полями для всех измеряемых показателей и внешним ключом `user_id`, ссылающимся на таблицу `users`. Дата хранится как строка (TEXT) в ISO-формате, что упрощает сортировку и фильтрацию по датам.

Операции с пользователями

Метод `add_user()` добавляет нового пользователя (рис.6).

```
def add_user(self, user):
    conn = sqlite3.connect(self.db_name)
    cursor = conn.cursor()
    try:
        cursor.execute('''
            INSERT INTO users (username, password_hash) VALUES (?, ?)
        ''', (user.username, user.password_hash))
        conn.commit()
        return cursor.lastrowid
    except sqlite3.IntegrityError:
        return None
    finally:
        conn.close()
```

Рисунок 6. Метод добавления пользователя

При нарушении уникальности логина выбрасывается `IntegrityError`, и метод возвращает `None`. Это далее трактуется как попытка регистрации уже существующего пользователя.

Метод `get_user(username)` выполняет поиск пользователя по имени (рис.7).

```
def get_user(self, username):
    conn = sqlite3.connect(self.db_name)
    cursor = conn.cursor()
    cursor.execute('SELECT id, username, password_hash FROM users WHERE username = ?', (username,))
    row = cursor.fetchone()
    conn.close()
    if row:
        return User(row[1], row[2], row[0])
    return None
```

Рисунок 7. Получение пользователя

Если пользователь найден, создаётся объект `User` с полями, полученными из БД; в противном случае возвращается `None`.

Операции с измерениями

Добавление нового измерения (рис.8).

```
def add_measurement(self, measurement, user_id):
    conn = sqlite3.connect(self.db_name)
    cursor = conn.cursor()
    cursor.execute('''
        INSERT INTO measurements (user_id, date, systolic_bp, diastolic_bp, blood_sugar, weight, notes)
        VALUES (?, ?, ?, ?, ?, ?, ?)
    ''', (user_id, measurement.date.isoformat(), measurement.systolic_bp, measurement.diastolic_bp,
        measurement.blood_sugar, measurement.weight, measurement.notes))
    conn.commit()
    conn.close()
```

Рисунок 8. Добавление нового измерения

Здесь дата сохраняется как ISO-строка, остальные поля переносятся в соответствующие столбцы.

Получение всех измерений конкретного пользователя (рис.9).

```
def get_all_measurements(self, user_id):
    conn = sqlite3.connect(self.db_name)
    cursor = conn.cursor()
    cursor.execute('SELECT * FROM measurements WHERE user_id = ? ORDER BY date DESC', (user_id,))
    rows = cursor.fetchall()
    conn.close()
    measurements = []
    for row in rows:
        date = datetime.fromisoformat(row[2])
        measurement = HealthMeasurement(date, row[3], row[4], row[5], row[6], row[7])
        measurements.append(measurement)
    return measurements
```

Это позволяет централизованно менять тексты интерфейса и поддерживать единый стиль локализации.

Компонент ModernDatePicker (рис.11.1-11.2)

Класс ModernDatePicker(ttk.Frame) реализует компактный виджет выбора даты:

- В конструкторе (`__init__`) задаются:
 - `selected_date` — текущая выбранная дата (по умолчанию — сегодняшняя);
 - `entry_var` — строковая переменная для отображения даты;
 - поле Entry (только для чтения), в котором показывается выбранная дата;
 - кнопка «Выбрать», открывающая календарь.
- Метод `open_calendar()` создаёт всплывающее окно Toplevel с заголовком `DATE_PICKER_TITLE`. Окно оформлено в тёмной цветовой гамме и позиционируется рядом с полем ввода. Внутри размещается виджет Calendar из библиотеки `tkcalendar` с настройкой цветовых параметров (фон, выделение дней, цвета выходных) и без отображения номеров недель (`showweeknumbers=False`). Ниже располагается кнопка «Применить дату».
- Метод `_apply_date()` считывает выбранную дату из Calendar, обновляет `selected_date` и `entry_var`, при наличии колбэка `on_change` вызывает его, и закрывает всплывающее окно.
- Методы `get_date()` и `set_date(value)` используются для получения и установки даты извне, что упрощает взаимодействие с этим компонентом в других частях интерфейса.

Таким образом, ModernDatePicker обеспечивает современный и удобный выбор дат без ручного ввода.

```
class ModernDatePicker(ttk.Frame):
    """Compact date picker with a styled calendar popover."""

    def __init__(self, parent, initial_date=None, on_change=None):
        super().__init__(parent)
        self.on_change = on_change
        self.selected_date = (initial_date or datetime.now().date())
        self.entry_var = tk.StringVar(value=self._format_date(self.selected_date))
        self.popup = None

        self.columnconfigure(0, weight=1)

        self.entry = ttk.Entry(self, textvariable=self.entry_var, state="readonly", width=18, font=("Segoe UI", 11))
        self.entry.grid(row=0, column=0, sticky="ew")

        self.trigger_btn = ttk.Button(self, text=BTN_DATE_OPEN, command=self.open_calendar, width=10, style="Accent.TButton")
        self.trigger_btn.grid(row=0, column=1, padx=(8, 0))

    def _format_date(self, value):
        return value.strftime("%Y-%m-%d")

    def open_calendar(self):
        if self.popup and tk.Toplevel.winfo_exists(self.popup):
            self.popup.destroy()

        self.popup = tk.Toplevel(self)
        self.popup.title(DATE_PICKER_TITLE)
        self.popup.configure(bg="#f1f3a")
        self.popup.resizable(False, False)
        self.popup.transient(self.winfo_toplevel())
        self.popup.grab_set()

        x = self.winfo_rootx()
        y = self.winfo_rooty() + self.winfo_height() + 4
        self.popup.geometry(f"340x360+{x}+{y}")
```

Рисунок 11.1. Компонент ModernDatePicker

```

self.calendar = Calendar(
    self.popup,
    selectmode="day",
    date_pattern="yyyy-mm-dd",
    background="#0f1f3a",
    disabledbackground="#e7ecf5",
    bordercolor="#0f1f3a",
    headersbackground="#0f1f3a",
    normalbackground="#f6f9ff",
    weekendbackground="#eef3ff",
    othermonthbackground="#edf2ff",
    othermonthwebackground="#e3ebff",
    selectbackground="#1f6feb",
    selectforeground="ffffff",
    normalforeground="#0f1f3a",
    weekendforeground="#1f6feb",
    headersforeground="ffffff",
    showweeknumbers=False,
)
self.calendar.pack(fill="both", expand=True, padx=12, pady=(12, 6))
self.calendar.selection_set(self.selected_date)

ttk.Button(
    self.popup,
    text=BTN_DATE_APPLY,
    command=self._apply_date,
    style="Accent.TButton",
).pack(fill="x", padx=12, pady=(0, 12))

def _apply_date(self):
    self.selected_date = self.calendar.selection_get()
    self.entry_var.set(self._format_date(self.selected_date))
    if self.on_change:
        self.on_change(self.selected_date)
    if self.popup and tk.Toplevel.winfo_exists(self.popup):
        self.popup.destroy()

def get_date(self):
    return self.selected_date

def set_date(self, value):
    self.selected_date = value
    self.entry_var.set(self._format_date(self.selected_date))

```

Рисунок 11.2 Компонент ModerDatePicker

Класс HealthMonitorApp — основное окно и логика работы
Инициализация окна и стилей

В конструкторе `__init__` (рис.12):

- настраивается главное окно:
 - заголовок (APP_TITLE);
 - размеры (1180x780, минимальный размер 980x680);
 - фон и возможность изменения размеров;
- создаётся экземпляр HealthDatabase и переменная current_user_id для хранения идентификатора текущего пользователя;
- инициализируется стиль `tk.Style()` с темой `clam`, после чего вызывается `_setup_styles()`.

```

class HealthMonitorApp:
    def __init__(self, root):
        self.root = root
        self.root.title(APP_TITLE)
        self.root.geometry("1180x780")
        self.root.minsize(980, 680)
        self.root.configure(bg="#e9eff7")
        self.root.resizable(True, True)

        self.db = HealthDatabase()
        self.current_user_id = None

        self.style = ttk.Style()
        self.style.theme_use("clam")
        self._setup_styles()

        self.main_frame = ttk.Frame(self.root, padding=0)
        self.main_frame.pack(fill="both", expand=True)

        self.canvas = tk.Canvas(self.main_frame, highlightthickness=0, bg="#e9eff7")
        self.v_scroll = ttk.Scrollbar(self.main_frame, orient="vertical", command=self.canvas.yview)
        self.canvas.configure(yscrollcommand=self.v_scroll.set)

        self.canvas.pack(side="left", fill="both", expand=True)
        self.v_scroll.pack(side="right", fill="y")

        self.canvas.bind("<Configure>", self._on_canvas_resize)
        self.canvas.bind_all("<MouseWheel>", self._on_mousewheel)

        self.draw_gradient()

        self.show_login_screen()

```

Рисунок 12. Конструктор __init__

Метод _setup_styles() задаёт цветовую схему приложения (рис.13):

- общие цвета фона, текста и акцентный цвет;
- стили для:
 - рамок и карточек (TFrame, Card.TFrame, Section.TLabelframe);
 - подписей (TLabel, Bold.TLabel);
 - кнопок (обычных TButton и акцентных Accent.TButton с изменением цвета при наведении);
 - таблицы Treeview и её заголовков (шрифты, высота строк, цвет выбранной строки).

```

def _setup_styles(self):
    accent = "#1f6feb"
    bg = "#e9eff7"
    surface = "#f8fbff"
    text = "#0f1f3a"

    self.style.configure("TFrame", background=bg)
    self.style.configure("Card.TFrame", background=surface, relief="flat")
    self.style.configure("TLabel", background=bg, foreground=text, font=("Segoe UI", 11))
    self.style.configure("Bold.TLabel", background=bg, foreground=text, font=("Segoe UI Semibold", 12))
    self.style.configure("Section.TLabelframe", background=surface, relief="flat", padding=16, borderwidth=0)
    self.style.configure("Section.TLabelframe.Label", background=surface, foreground=text, font=("Segoe UI Semibold", 12))
    self.style.configure("TEntry", font=("Segoe UI", 11))
    self.style.configure("Accent.TButton", font=("Segoe UI Semibold", 11), foreground="ffffff", background=accent, padding=(14, 8))
    self.style.map("Accent.TButton", background=[("active", "#1559c2")])
    self.style.configure("TButton", font=("Segoe UI", 10), padding=(12, 6), background=surface)
    self.style.configure("Treeview", font=("Segoe UI", 10), rowheight=26, borderwidth=0, relief="flat")
    self.style.configure("Treeview.Heading", font=("Segoe UI Semibold", 10), background=surface, foreground=text)
    self.style.map("Treeview", background=[("selected", "#d9e7ff")])

```

Рисунок 13. Метод для задания стилей

Canvas, прокрутка и градиентный фон

Далее создаётся основной контейнер `main_frame`, внутри которого размещается (рис.14):

- виджет `Canvas` для рисования фона и размещения всех остальных элементов;
- вертикальная полоса прокрутки `Scrollbar`, связанная с `Canvas` через `yscrollcommand`.

```
def _on_canvas_resize(self, event):
    self.draw_gradient()
    if hasattr(self, "login_window"):
        self.canvas.coords(self.login_window, event.width / 2, event.height / 2)
    if hasattr(self, "main_container_window"):
        self.canvas.coords(self.main_container_window, event.width / 2, 28)
        self.canvas.itemconfigure(self.main_container_window, width=event.width - 80)
    self.canvas.configure(scrollregion=self.canvas.bbox("all"))

def _on_mousewheel(self, event):
    # Windows scroll: event.delta in multiples of 120
    self.canvas.yview_scroll(int(-1 * (event.delta / 120)), "units")
```

Рисунок 14. Обработчики событий `Canvas` и прокрутки

Обработчики событий:

- `_on_canvas_resize()` — вызывается при изменении размеров окна. В нём перестраивается градиентный фон (`draw_gradient()`), переопределяются координаты окна логина или основного контейнера и обновляется область прокрутки (`scrollregion`).
- `_on_mousewheel()` — обеспечивает прокрутку содержимого колёсиком мыши.

Метод `draw_gradient()` формирует фон (рис.15):

- очищает предыдущие элементы с тегом `"gradient"`;
- рисует вертикальные линии с плавным изменением цвета (эффект градиента);
- поверх добавляет несколько полупрозрачных окружностей случайного положения и радиуса, создающих «облачный» фон;
- отправляет фон на задний план, чтобы он не перекрывал остальные элементы.

```

def draw_gradient(self):
    self.canvas.delete("gradient")
    width = max(self.canvas.winfo_width(), 1180)
    height = max(self.canvas.winfo_height(), 780)

    for i in range(height):
        color = "#%02x%02x%02x" % (233 - int(i * 0.05), 239 - int(i * 0.05), 247 - int(i * 0.03))
        self.canvas.create_line(0, i, width, i, fill=color, tags="gradient")

    for _ in range(18):
        x = random.randint(60, width - 60)
        y = random.randint(60, height - 60)
        r = random.randint(24, 46)
        self.canvas.create_oval(x - r, y - r, x + r, y + r, fill="#f5fbff", outline="#b8ccf2", width=1, tags="gradient")

    self.canvas.tag_lower("gradient")

```

Рисунок 15. Метод draw_gradient

Метод `_clear_content()` удаляет элементы с тегом "content" и сбрасывает ссылки на окна логина и главного контейнера, что позволяет переключаться между экранами (рис.16).

```

def _clear_content(self):
    self.canvas.delete("content")
    if hasattr(self, "login_window"):
        del self.login_window
    if hasattr(self, "main_container_window"):
        del self.main_container_window

```

Рисунок 16. Метод очистки элементов

Экран логина

Метод `show_login_screen()` (рис.17):

- очищает содержимое, перерисовывает фон;
- создаёт карточку (Card.TFrame) по центру Canvas с помощью `create_window`;
- размещает на карточке:
 - заголовок с названием приложения;
 - метки и поля ввода для логина и пароля;
 - две кнопки: «Войти» (`login()`) и «Зарегистрироваться» (`register()`).

```

def show_login_screen(self):
    self._clear_content()
    self.draw_gradient()

    card = ttk.Frame(self.canvas, style="Card.TFrame", padding=28)
    self.canvas.update_idletasks()
    self.login_window = self.canvas.create_window(self.canvas.winfo_width() / 2, self.canvas.winfo_height() / 2, window=card, anchor="center", tags=("content",))
    self.refresh_scrollregion()

    ttk.Label(card, text=APP_TITLE, font=("Segoe UI Semibold", 16), background="#f8fbff").grid(row=0, column=0, colspan=2, pady=(0, 12))
    ttk.Label(card, text=LOGIN_LABEL, style="Bold.TLabel", background="#f8fbff").grid(row=1, column=0, sticky="w", pady=(4, 4))
    self.username_entry = ttk.Entry(card, width=26)
    self.username_entry.grid(row=1, column=1, pady=(4, 4))

    ttk.Label(card, text=PASSWORD_LABEL, style="Bold.TLabel", background="#f8fbff").grid(row=2, column=0, sticky="w", pady=(4, 10))
    self.password_entry = ttk.Entry(card, show="*", width=26)
    self.password_entry.grid(row=2, column=1, pady=(4, 10))

    ttk.Button(card, text=BTN_LOGIN, style="Accent.TButton", command=self.login).grid(row=3, column=0, pady=(6, 0), sticky="ew", colspan=2)
    ttk.Button(card, text=BTN_REGISTER, command=self.register).grid(row=4, column=0, pady=(8, 0), sticky="ew", colspan=2)

```

Рисунок 17. Метод показа экрана входа

Метод login() (рис.18):

- считывает значения полей, запрашивает пользователя через `self.db.get_user(username)`;
- при наличии пользователя проверяет пароль методом `check_password`;
- при успехе устанавливает `current_user_id` и вызывает `show_main_screen()`, иначе показывает сообщение об ошибке (`ERR_LOGIN`).

```
def login(self):
    username = self.username_entry.get()
    password = self.password_entry.get()
    user = self.db.get_user(username)
    if user and user.check_password(password):
        self.current_user_id = user.id
        self.show_main_screen()
    else:
        messagebox.showerror(MSG_ERROR, ERR_LOGIN)
```

Рисунок 18. Метод авторизации

Метод register() (рис.19):

- при наличии логина и пароля создаёт объект `User` с хешированным паролем;
- пытается добавить пользователя через `db.add_user(user)`;
- при успехе сообщает о создании аккаунта (`INFO_CREATED`), при конфликте логина — об ошибке (`ERR_EXISTS`);
- при незаполненных полях сообщает `ERR_EMPTY`.

```
def register(self):
    username = self.username_entry.get()
    password = self.password_entry.get()
    if username and password:
        user = User(username, User.hash_password(password))
        user_id = self.db.add_user(user)
        if user_id:
            messagebox.showinfo(MSG_READY, INFO_CREATED)
        else:
            messagebox.showerror(MSG_ERROR, ERR_EXISTS)
    else:
        messagebox.showerror(MSG_ERROR, ERR_EMPTY)
```

Рисунок 19. Метод регистрации

Метод `logout()` сбрасывает `current_user_id`, очищает содержимое и возвращает интерфейс к экрану входа (рис.20).

```
def logout(self):
    """Return to login screen and drop current user context."""
    self.current_user_id = None
    self._clear_content()
    self.show_login_screen()
```

Рисунок 20. Метод выхода

Главный экран мониторинга

Метод `show_main_screen()` (рис.21):

- очищает предыдущий экран и перерисовывает фон;
- создаёт основной контейнер (`Card.TFrame`), который размещается в верхней части `Canvas`;
- внутри контейнера формирует три блока:
 - шапку (заголовок + кнопка выхода);
 - секцию ввода новых измерений;
 - секцию истории и блок графиков.

```
def show_main_screen(self):
    self._clear_content()
    self.draw_gradient()

    main_container = ttk.Frame(self.canvas, style="Card.TFrame", padding=20)
    main_container.columnconfigure(0, weight=1)
    main_container.columnconfigure(1, weight=1)
    main_container.rowconfigure(1, weight=1)
    main_container.rowconfigure(2, weight=1)
    self.canvas.update_idletasks()
    self.main_container_window = self.canvas.create_window(self.canvas.winfo_width() / 2, 28, window=main_container, anchor="n", tags=("content",))
    self.canvas.itemconfigure(self.main_container_window, width=self.canvas.winfo_width() - 80)
    self._refresh_scrollregion()

    header_row = ttk.Frame(main_container, style="Card.TFrame")
    header_row.grid(row=0, column=0, columnspan=2, sticky="ew", pady=(0, 12))
    header_row.columnconfigure(0, weight=1)

    header = ttk.Label(header_row, text=TITLE_MEASUREMENTS, style="Bold.TLabel", background="#f8fbff")
    header.grid(row=0, column=0, sticky="w")
    ttk.Button(header_row, text=BTN_LOGOUT, command=self.logout).grid(row=0, column=1, sticky="e")

    self._build_input_block(main_container)
    self._build_list_block(main_container)
    self._build_chart_block(main_container)

    self.load_measurements()
```

Рисунок 21. Метод показа главного экрана

После построения интерфейса вызывается `load_measurements()` для загрузки данных пользователя (рис.22).

```
def load_measurements(self):
    for row in self.tree.get_children():
        self.tree.delete(row)
    measurements = self.db.get_all_measurements(self.current_user_id)
    for m in measurements:
        bp = f"{int(m.systolic_bp)}/{int(m.diastolic_bp)}" if m.systolic_bp and m.diastolic_bp else ""
        tags = ("danger",) if m.is_dangerous() else ()
        self.tree.insert("", "end", values=(m.date.strftime("%Y-%m-%d"), bp, m.blood_sugar or "", m.weight or "", m.notes), tags=tags)
    self.tree.tag_configure("danger", background="#ffeded")
    self._refresh_scrollregion()
```

Рисунок 22. Метод загрузки

Блок ввода новых измерений

Метод `_build_input_block(parent)` создаёт Labelframe «Новая запись» с полями (рис.23):

- дата (через `ModernDatePicker`);
- верхнее и нижнее давление;
- сахар;
- вес;
- заметка.

```
def _build_input_block(self, parent):
    input_frame = ttk.LabelFrame(parent, text=TITLE_NEW_ENTRY, style="Section.TLabelFrame")
    input_frame.grid(row=1, column=0, sticky="nsew", padx=(0, 14), pady=(0, 14))
    input_frame.columnconfigure(1, weight=1)

    ttk.Label(input_frame, text=LABEL_DATE, style="Bold.TLabel", background="#f8fbff").grid(row=0, column=0, sticky="w", pady=4)
    self.date_picker = ModernDatePicker(input_frame, initial_date=datetime.now().date())
    self.date_picker.grid(row=0, column=1, sticky="ew", pady=4)

    ttk.Label(input_frame, text=LABEL_SYS, style="Bold.TLabel", background="#f8fbff").grid(row=1, column=0, sticky="w", pady=4)
    self.systolic_entry = ttk.Entry(input_frame)
    self.systolic_entry.grid(row=1, column=1, sticky="ew", pady=4)

    ttk.Label(input_frame, text=LABEL_DIA, style="Bold.TLabel", background="#f8fbff").grid(row=2, column=0, sticky="w", pady=4)
    self.diastolic_entry = ttk.Entry(input_frame)
    self.diastolic_entry.grid(row=2, column=1, sticky="ew", pady=4)

    ttk.Label(input_frame, text=LABEL_SUGAR, style="Bold.TLabel", background="#f8fbff").grid(row=3, column=0, sticky="w", pady=4)
    self.sugar_entry = ttk.Entry(input_frame)
    self.sugar_entry.grid(row=3, column=1, sticky="ew", pady=4)

    ttk.Label(input_frame, text=LABEL_WEIGHT, style="Bold.TLabel", background="#f8fbff").grid(row=4, column=0, sticky="w", pady=4)
    self.weight_entry = ttk.Entry(input_frame)
    self.weight_entry.grid(row=4, column=1, sticky="ew", pady=4)

    ttk.Label(input_frame, text=LABEL_NOTE, style="Bold.TLabel", background="#f8fbff").grid(row=5, column=0, sticky="w", pady=4)
    self.notes_entry = ttk.Entry(input_frame)
    self.notes_entry.grid(row=5, column=1, sticky="ew", pady=4)

    ttk.Button(input_frame, text=BTN_SAVE, style="Accent.TButton", command=self.add_measurement).grid(row=6, column=0, columnspan=2, pady=(10, 0), sticky="ew")
```

Рисунок 23. Создание формы

Под полями расположена кнопка «Сохранить измерение», привязанная к методу `add_measurement()` (рис.24).

```
def add_measurement(self):
    try:
        date_value = self.date_picker.get_date()
        date = datetime.combine(date_value, datetime.min.time())
        systolic = float(self.systolic_entry.get()) if self.systolic_entry.get() else None
        diastolic = float(self.diastolic_entry.get()) if self.diastolic_entry.get() else None
        sugar = float(self.sugar_entry.get()) if self.sugar_entry.get() else None
        weight = float(self.weight_entry.get()) if self.weight_entry.get() else None
        notes = self.notes_entry.get()

        measurement = HealthMeasurement(date, systolic, diastolic, sugar, weight, notes)
        self.db.add_measurement(measurement, self.current_user_id)
        self.load_measurements()
        self.clear_entries()
        messagebox.showinfo(MSG_READY, INFO_SAVED)
        if measurement.is_dangerous():
            messagebox.showwarning(MSG_WARNING, WARN_LIMITS)
    except ValueError as e:
        messagebox.showerror(MSG_ERROR, f"{ERR_DATA}{e}")
```

Рисунок 24. Добавление записи

Метод `add_measurement()`:

1. Получает дату из `ModernDatePicker` и преобразует её к `datetime` (с временем 00:00).

2. Считывает значения из полей, преобразуя их к типу float при наличии данных, либо оставляя None.
3. Создает объект HealthMeasurement с полученными показателями и заметкой.
4. Вызывает self.db.add_measurement(measurement, self.current_user_id) для записи в БД.
5. Обновляет таблицу (load_measurements()), очищает поля (clear_entries()) и выводит сообщение о сохранении (INFO_SAVED).
6. Если measurement.is_dangerous() возвращает True, показывает предупреждение (WARN_LIMITS).

В случае ошибки преобразования типов (например, введен текст вместо числа) перехватывается ValueError, и пользователю выводится сообщение ERR_DATA с деталями исключения.

Блок истории измерений

Метод _build_list_block(parent) создает Labelframe «История», внутри которого (рис.25):

- виджет Treeview с колонками:
 - дата (GRID_DATE),
 - давление (GRID_BP),
 - сахар (GRID_SUGAR),
 - вес (GRID_WEIGHT),
 - заметка (GRID_NOTE);
- вертикальный скроллбар, связанный с таблицей.

```
def _build_list_block(self, parent):
    list_frame = ttk.Labelframe(parent, text=TITLE_HISTORY, style="Section.TLabelframe")
    list_frame.grid(row=1, column=1, sticky="nsew", pady=(0, 14))
    list_frame.columnconfigure(0, weight=1)
    list_frame.rowconfigure(0, weight=1)

    self.tree = ttk.Treeview(list_frame, columns=("date", "bp", "sugar", "weight", "notes"), show="headings", height=12)
    self.tree.heading("date", text=GRID_DATE)
    self.tree.heading("bp", text=GRID_BP)
    self.tree.heading("sugar", text=GRID_SUGAR)
    self.tree.heading("weight", text=GRID_WEIGHT)
    self.tree.heading("notes", text=GRID_NOTE)
    self.tree.column("date", width=110, anchor="center")
    self.tree.column("bp", width=120, anchor="center")
    self.tree.column("sugar", width=80, anchor="center")
    self.tree.column("weight", width=80, anchor="center")
    self.tree.column("notes", width=200, anchor="w")
    self.tree.grid(row=0, column=0, sticky="nsew")

    scrollbar = ttk.Scrollbar(list_frame, orient="vertical", command=self.tree.yview)
    self.tree.configure(yscrollcommand=scrollbar.set)
    scrollbar.grid(row=0, column=1, sticky="ns")
```

Рисунок 25. Метод создания «Истории»

Метод load_measurements() (рис.22):

- очищает таблицу от старых строк;
- получает список измерений текущего пользователя через db.get_all_measurements;

- для каждого измерения формирует строку:
 - дата (`m.date.strftime("%Y-%m-%d")`);
 - давление в виде SYS/DIA, если оба значения заданы;
 - сахар и вес, либо пустые поля, если значений нет;
 - заметка;
- если `m.is_dangerous()` возвращает `True`, строке присваивается тег `"danger"`.
- далее тег `"danger"` настраивается так, что фон строки становится светло-красным (`#ffeded`), визуально выделяя потенциально опасные измерения.

После заполнения таблицы вызывается `_refresh_scrollregion()`, чтобы учесть изменение размеров содержимого при прокрутке.

Блок графиков

Метод `_build_chart_block(parent)` создаёт `Labelframe` «Графики» (рис.26).
Внутри:

- панель заголовка с кнопкой «Обновить графики» (вызов `plot_charts()`);
- объект `Figure` из `Matplotlib` с заранее заданными размерами, настройкой отступов и фона;
- холст `FigureCanvasTkAgg`, встраивающий фигуру в интерфейс `Tkinter`.

```
def _build_chart_block(self, parent):
    chart_frame = ttk.Labelframe(parent, text=TITLE_CHARTS, style="Section.TLabelframe")
    chart_frame.grid(row=2, column=0, columnspan=2, sticky="nsew")
    chart_frame.columnconfigure(0, weight=1)
    chart_frame.rowconfigure(1, weight=1)

    header_bar = ttk.Frame(chart_frame, style="Card.TFrame")
    header_bar.grid(row=0, column=0, sticky="ew", pady=(0, 6))
    header_bar.columnconfigure(0, weight=1)
    ttk.Button(header_bar, text=BTN_CHART, style="Accent.TButton", command=self.plot_charts).grid(row=0, column=1, sticky="e")

    self.figure = plt.Figure(figsize=(9.2, 4.4), dpi=100, facecolor="#f8fbff")
    self.figure.subplots_adjust(hspace=0.4, wspace=0.3)
    self.chart_canvas = FigureCanvasTkAgg(self.figure, chart_frame)
    self.chart_canvas.get_tk_widget().grid(row=1, column=0, sticky="nsew")
```

Рисунок 26. Создание «Графиков»

Метод `plot_charts()` (рис.27.1-27.2):

1. Получает все измерения пользователя, сортируя их по дате по возрастанию.
2. Если данных нет, выводит сообщение `INFO_ADD_DATA` и завершает работу.
3. Формирует отдельные списки дат и значений:
 - `bp_dates`, `systolic`, `diastolic` — для давления;
 - `sugar_dates`, `sugars` — для уровня сахара;
 - `weight_dates`, `weights` — для массы тела.

4. Очищает фигуру и настраивает общие параметры (subplots_adjust, отступы, поля).
5. Создает локатор дат AutoDateLocator и форматтер DateFormatter("%d %b").
6. Определяет вспомогательную функцию style_axis(ax), которая:
 - включает сетку,
 - задает цвет фона осей,
 - настраивает шрифт и наклон подписей по оси X,
 - слегка ослабляет верхнюю и правую границы,
 - назначает локатор и форматтер дат,
 - задает поля по осям (margins).

```
def plot_charts(self):
    measurements = sorted(self.db.get_all_measurements(self.current_user_id), key=lambda m: m.date)
    if not measurements:
        messagebox.showinfo(MSG_NO_DATA, INFO_ADD_DATA)
        return

    bp_dates, systolic, diastolic = [], [], []
    sugar_dates, sugars = [], []
    weight_dates, weights = [], []

    for m in measurements:
        if m.systolic_bp is not None or m.diastolic_bp is not None:
            bp_dates.append(m.date)
            systolic.append(m.systolic_bp)
            diastolic.append(m.diastolic_bp)
        if m.blood_sugar is not None:
            sugar_dates.append(m.date)
            sugars.append(m.blood_sugar)
        if m.weight is not None:
            weight_dates.append(m.date)
            weights.append(m.weight)

    self.figure.clear()
    self.figure.subplots_adjust(hspace=0.45, wspace=0.35, left=0.08, right=0.98, top=0.92, bottom=0.18)

    date_locator = mdates.AutoDateLocator(minticks=3, maxticks=6)
    date_formatter = mdates.DateFormatter("%d %b")

    def style_axis(ax):
        ax.grid(True, alpha=0.2, linestyle="--")
        ax.set_facecolor("#fcfdff")
        ax.tick_params(axis="x", rotation=25, labelsize=9)
        ax.tick_params(axis="y", labelsize=9)
        ax.spines["top"].set_alpha(0.2)
        ax.spines["right"].set_alpha(0.2)
        ax.xaxis.set_major_locator(date_locator)
        ax.xaxis.set_major_formatter(date_formatter)
        ax.margins(x=0.02, y=0.15)
```

Рисунок 27.1. Метод построения «Графиков»

```

if bp_dates:
    ax1 = self.figure.add_subplot(221)
    if any(s is not None for s in systolic):
        ax1.plot(bp_dates, [v if v is not None else float("nan") for v in systolic], label=PLOT_SYS, color="#d9534f", linewidth=2.2, marker="o", markersize=5)
    if any(d is not None for d in diastolic):
        ax1.plot(bp_dates, [v if v is not None else float("nan") for v in diastolic], label=PLOT_DIA, color="#0275d8", linewidth=2.2, marker="o", markersize=5)
    ax1.set_title(PLOT_BP)
    ax1.legend(framealpha=0.1, fontsize=9)
    style_axis(ax1)

if sugars:
    ax2 = self.figure.add_subplot(222)
    ax2.plot(sugar_dates, sugars, color="#3daa6b", linewidth=2.3, marker="o", markersize=5)
    ax2.fill_between(sugar_dates, sugars, color="#3daa6b", alpha=0.12)
    ax2.set_title(PLOT_SUGAR)
    style_axis(ax2)

if weights:
    ax3 = self.figure.add_subplot(223)
    ax3.plot(weight_dates, weights, color="#6f42c1", linewidth=2.3, marker="o", markersize=5)
    ax3.fill_between(weight_dates, weights, color="#6f42c1", alpha=0.1)
    ax3.set_title(PLOT_WEIGHT)
    style_axis(ax3)

self.chart_canvas.draw()
self._refresh_scrollregion()

```

Рисунок 27.2. Метод построения «Графиков»

Далее по наличию данных строятся соответствующие графики:

- если есть значения давления, создаётся подграфик (например, `ax1 = self.figure.add_subplot(221)`), на котором:
 - систолическое давление рисуется красной линией с маркерами;
 - диастолическое — синей линией;
 - выводится легенда и заголовок «Давление» (`PLOT_BP`);
 - применяются стили через `style_axis(ax1)`.
- если есть данные по сахару, создаётся второй подграфик (`ax2`), на котором:
 - строится зелёная линия с маркерами;
 - область под графиком заливается полупрозрачным цветом;
 - задаётся заголовок `PLOT_SUGAR` и вызывается `style_axis(ax2)`.
- аналогично для массы тела (`ax3`) строится фиолетовая линия с заливкой и заголовком `PLOT_WEIGHT`.

В конце вызывается `self.chart_canvas.draw()` для обновления картинки и `_refresh_scrollregion()` для корректной работы прокрутки.

Все четыре модуля приложения работают совместно, образуя чёткую многослойную структуру:

- `main.py` — инициализирует корневое окно и запускает приложение;
- `models.py` — описывает доменные сущности (пользователь и измерение) и их логику (хеширование пароля, определение опасных значений);
- `database.py` — обеспечивает устойчивое хранение данных в SQLite и преобразование строк таблиц в объекты моделей;
- `gui.py` — реализует весь графический интерфейс, взаимодействие с пользователем, работу с формами, таблицей и графиками.

Внешний вид приложения:

- Экран авторизации/регистрации (рис.28)

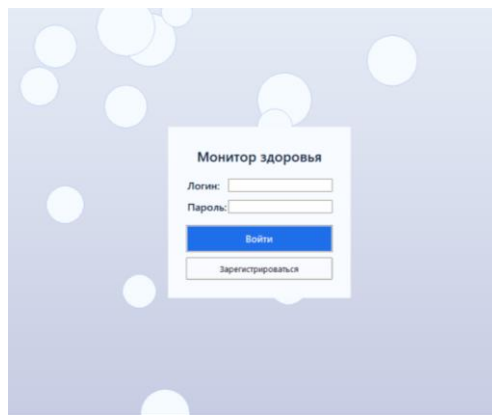


Рисунок 28. Экран авторизации/регистрации

- Главный экран приложения (рис.29)

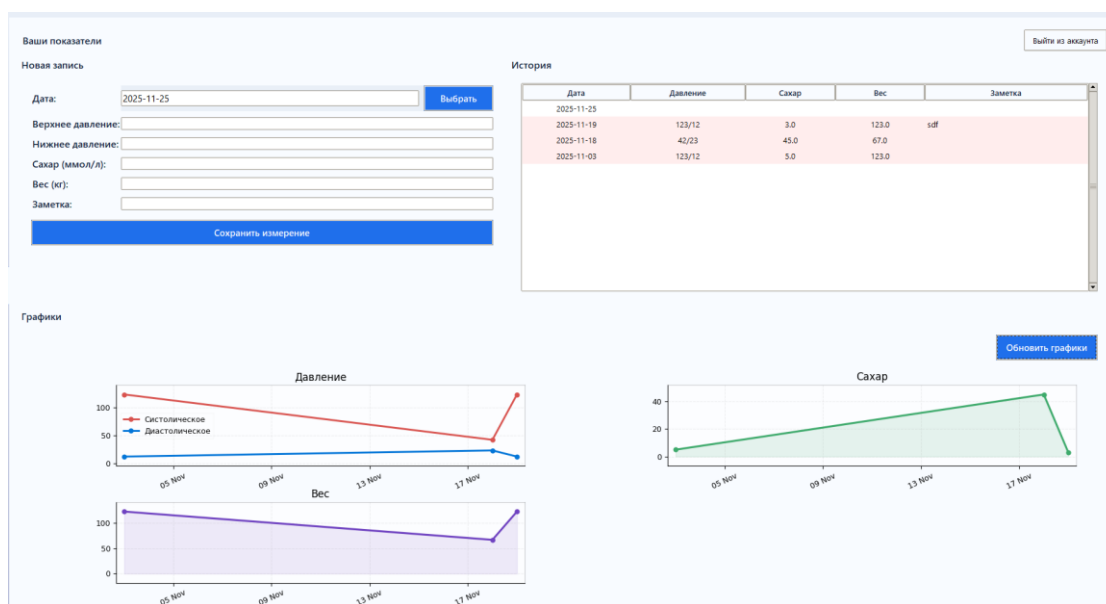


Рисунок 29. Главное окно приложения

4 Выводы

В ходе работы разработано настольное приложение «Монитор здоровья» на языке Python с использованием библиотек Tkinter, tkcalendar и Matplotlib, а также СУБД SQLite для хранения данных. Приложение обеспечивает:

- регистрацию и аутентификацию пользователей с изоляцией их записей;
- ввод и хранение измерений артериального давления, уровня сахара и массы тела;
- подсветку опасных значений и вывод предупреждений пользователю;
- наглядное представление истории измерений в табличном виде;
- визуализацию динамики показателей на графиках с удобной временной шкалой;

- удобный выбор дат через встроенный календарь и комфортную навигацию по интерфейсу с использованием прокрутки.

Полученный прототип может использоваться как простое средство само мониторинга состояния здоровья, а также как учебный пример при изучении разработки настольных приложений на Python с графическим интерфейсом и локальным хранилищем данных.

Библиографический список

1. Машков В.А., Литвиненко В.И. Использование языка программирования python 3 и системы компьютерной алгебры sympy на факультативных занятиях по теории чисел // Электротехнические и компьютерные системы. 2017. С. 48-54
2. Бухарова Г.Д., Комельских П.С. Важность и необходимость внедрения языка программирования python в процесс обучения студентов // Новые информационные технологии в образовании Материалы VII международной научно-практической конференции. 2014. С. 40-42.
3. Сейдаметов Г.С. Особенности использования языка программирования python в подготовке будущих инженеров-программистов // International scientific review. 2019. №. 9(10). С. 50-51
4. Усеинов Э.А. Объектно-ориентированное программирование в рамках дисциплины «язык программирования python» // Ученые записки крымского инженерно-педагогического университета. 2012. №. 34. С. 157-160.
5. Матвеева А. С. Реализация паттерна поведения «Команда» в графическом интерфейсе пользователя на Python // Постулат. 2025. №1.