**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ КЫРГЫЗСКОЙ РЕСПУБЛИКИ**

**БИШКЕКСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. К. КАРАСАЕВА**



**КУРСОВАЯ РАБОТА**

**По дисциплине**: Дисциплина профиля по выбору студента

**Тема:**«Простое приложение по учету расходов с сохранением данных в базе или в файле»

**Выполнила:** Токтогулова Нуржан

студентка 3 курса В-ВА 22/2К

**Научный руководитель:** Жакыпова Н. Н.

**Бишкек 2025**

Оглавление

[Оглавление 2](#_Toc193806027)

[Введение 2](#_Toc193806028)

[1.1 Структура курсовой работы: цель и задачи работы 2](#_Toc193806029)

[2. Теоретическая часть 4](#_Toc193806030)

[2.1. Обзор систем для учета расходов 4](#_Toc193806031)

[2.2. Современные технологии разработки приложений 6](#_Toc193806032)

[2.3. Использование Python и Tkinter для создания графических интерфейсов 8](#_Toc193806033)

[2.4. Принципы объектно-ориентированного программирования 10](#_Toc193806034)

[**2**.5. Использование PostgreSQL для хранения данных и Psycopg2 11](#_Toc193806035)

[**2.6 Сохранение данных в Neon.tech и GitHub** 12](#_Toc193806036)

[Глава 3. Этапы жизненного цикла разработки программы 13](#_Toc193806037)

[3.1. Планирование и анализ требований 13](#_Toc193806038)

[**3.2. Определение требований** 15](#_Toc193806039)

[3.3. Проектирование архитектуры 15](#_Toc193806040)

[3.4. Разработка продукта 16](#_Toc193806041)

[3.5.Тестирование и интеграция продукта 19](#_Toc193806042)

[3.6. Развертывание и обслуживание продукта 21](#_Toc193806043)

[4.Размещение в GitHub 23](#_Toc193806044)

[Заключение 25](#_Toc193806045)

[Использованные источники 27](#_Toc193806046)

[Листинг: 28](#_Toc193806047)

# 

# **Введение**

## **1.1 Структура курсовой работы: цель и задачи работы**

Современное общество сталкивается с необходимостью управления личными финансами. В условиях растущих затрат и неопределенности финансовых потоков, ведение учета расходов становится важной задачей для большинства людей. Ручной учет финансов, использование бумажных записей или простых таблиц могут быть неудобными и подвержены ошибкам, особенно когда объем данных увеличивается. Чтобы облегчить процесс управления личными финансами и сделать его более удобным, необходимо использовать специализированные приложения для учета расходов.

Актуальность темы заключается в том, что в наше время важно иметь под рукой эффективный инструмент для отслеживания своих финансовых операций. Приложение для учета расходов позволяет пользователю быстро и удобно вносить информацию о своих доходах и расходах, а также получать подробные отчеты и анализы, что помогает принимать обоснованные решения по управлению личными финансами.

Цель моей работы — разработать простое приложение для учета расходов, которое позволит пользователю добавлять, редактировать и удалять записи о своих расходах, а также хранить все данные в базе данных или файле. Для достижения этой цели я поставила перед собой следующие задачи:

1. Изучить существующие решения для учета расходов и определить их преимущества и недостатки.
2. Определить, какие функции должны быть в приложении для учета расходов, включая возможность добавления и удаления записей, а также создание отчетов.
3. Разработать структуру приложения и выбрать подходящие технологии для реализации (например, использование базы данных PostgreSQL для хранения данных).
4. Создать основные функции приложения, такие как добавление, редактирование и удаление записей о расходах, а также управление категориями расходов.
5. Протестировать приложение и убедиться в его корректной работе, обеспечив удобный интерфейс для пользователей.

# **2. Теоретическая часть**

## **2.1. Обзор систем для учета расходов**

Системы для учета расходов — это специализированные программные решения, предназначенные для автоматизации процессов управления личными финансами. Они помогают упростить задачи учета доходов и расходов, отслеживания финансовых п

отоков, управления категориями расходов и формирования отчетности. Современные системы учета расходов включают в себя функции записи и анализа данных, а также предоставляют пользователю различные инструменты для контроля и планирования бюджета.

Основные функции систем учета расходов:

1. **Запись и хранение данных:**

-Создание и ведение базы данных расходов (сумма, категория, дата).

-Учет информации о расходах (сумма, категория, дата, комментарии).

-Возможность добавления различных категорий расходов (например, еда, транспорт, жилье и т.д.).

-Поддержка различных форматов данных для удобства хранения и анализа информации.

1. **Поиск и фильтрация данных:**

-Быстрый поиск расходов по различным критериям (сумма, категория, дата).

-Возможность фильтрации результатов по категориям или временным промежуткам.

-Генерация отчетов по расходам за выбранный период (например, месяц, квартал, год).

1. **Управление категориями расходов:**

- Возможность добавления, редактирования и удаления категорий расходов.

- Управление списком категорий для удобства пользователя и улучшения структуры учета.

- Поддержка динамического добавления новых категорий по мере необходимости.

1. **Финансовая отчетность и аналитика:**

- Генерация отчетов о расходах, балансе и доходах.

- Статистика по категориям расходов, анализ динамики расходов по времени.

- Экспорт отчетов в различные форматы (например, PDF, Excel) для дальнейшего анализа.

1. **Управление балансом:**

- Учет текущего баланса пользователя.

- Автоматическое обновление баланса с учетом добавленных расходов.

- Возможность корректировки баланса и учета различных источников дохода и расхода.

## **2.2. Современные технологии разработки приложений**

Для создания систем учета расходов используются современные технологии и языки программирования, которые позволяют разработать эффективные и удобные приложения. Одним из самых популярных языков для разработки таких приложений является Python, благодаря своей простоте и широким возможностям. Для создания графических интерфейсов в Python часто используется библиотека Tkinter, которая предоставляет разработчикам мощные инструменты для создания кроссплатформенных приложений с простыми и удобными интерфейсами.

**Преимущества Python:**

**Простота**: Python имеет простой и понятный синтаксис, что делает его идеальным для начинающих разработчиков и позволяет быстро приступить к созданию приложения без необходимости осваивать сложные концепции.

**Широкие возможности**: Python поддерживает множество библиотек и фреймворков для решения различных задач, включая разработку веб-приложений, обработку данных, создание систем машинного обучения и многое другое. Это позволяет использовать его для разработки многозадачных и многокомпонентных систем.

**Кроссплатформенность**: Программы, написанные на Python, могут работать на различных операционных системах, таких как Windows, macOS и Linux, что делает их универсальными и доступными для широкого круга пользователей.

**Библиотека Tkinter:**

**Простота использования**: Tkinter предоставляет разработчикам удобный набор виджетов (кнопки, текстовые поля, таблицы, метки и т.д.), которые можно использовать для создания интуитивно понятных и функциональных интерфейсов. Это позволяет быстро разрабатывать графические интерфейсы без необходимости погружаться в сложные детали.

**Кроссплатформенность**: Приложения, созданные с использованием Tkinter, могут работать на различных операционных системах, что делает их доступными для пользователей независимо от используемой платформы.

**Интеграция с другими библиотеками**: Tkinter можно использовать в сочетании с другими библиотеками Python, такими как базы данных (например, PostgreSQL), для создания полноценных приложений с базой данных, которые обеспечивают эффективное хранение и обработку информации о расходах.

## **2.3. Использование Python и Tkinter для создания графических интерфейсов**

Tkinter — это стандартная библиотека Python для создания графических интерфейсов. Она предоставляет широкий набор виджетов (кнопки, текстовые поля, таблицы и т.д.), которые позволяют создавать интуитивно понятные и функциональные интерфейсы для пользователей. В данной работе Tkinter используется для создания графического интерфейса приложения для учета расходов, обеспечивая простоту и удобство работы с данным инструментом.

**Основные виджеты Tkinter:**

1. **Кнопки (Button)**: Кнопки используются для выполнения различных действий, таких как добавление или удаление данных, сохранение изменений и т.д. Например, кнопка может быть использована для добавления нового расхода или изменения баланса пользователя.
2. **Текстовые поля (Entry)**: Эти виджеты используются для ввода данных пользователем. Например, текстовое поле может быть использовано для ввода суммы расхода, категории или других данных, которые необходимо сохранить в приложении.
3. **Таблицы (Treeview)**: Этот виджет позволяет отображать данные в виде таблицы. В приложении для учета расходов таблица может быть использована для отображения списка всех расходов с их суммами, категориями и датами. Treeview помогает структурировать информацию и делает ее легко доступной для пользователя.
4. **Метки (Label)**: Метки используются для отображения текста на экране, например, заголовков, подсказок или статуса приложения. В приложении метки могут отображать общий баланс, общую сумму расходов и другие важные данные.

**Преимущества Tkinter:**

* **Простота**: Tkinter — это один из самых простых и доступных инструментов для создания графических интерфейсов. Библиотека интуитивно понятна, и для ее освоения не требуется большого опыта в разработке графических приложений.
* **Кроссплатформенность**: Программы, созданные с использованием Tkinter, могут работать на различных операционных системах, таких как Windows, macOS и Linux, что делает их доступными для большинства пользователей.
* **Интеграция с другими библиотеками**: Tkinter можно использовать в сочетании с другими мощными библиотеками Python для создания более сложных и функциональных приложений. Например, можно интегрировать Tkinter с библиотеками для работы с базами данных (PostgreSQL, SQLite) или для анализа данных (Pandas), что позволяет расширить функциональность приложения и обеспечивать более высокую производительность

## 

## **2.4. Принципы объектно-ориентированного программирования**

Объектно-ориентированное программирование (ООП) основывается на использовании объектов и классов. В данной работе принципы ООП помогают организовать код приложения для учета расходов, например, классы для работы с расходами и общей финансовой информацией.

**Основные принципы ООП:**

1. **Инкапсуляция**:  
   Сокрытие внутренних деталей объекта и предоставление методов для работы с данными. В приложении класс Expense может инкапсулировать информацию о расходах, позволяя работать с ней через интерфейс, не раскрывая внутреннюю реализацию.
2. **Наследование**:  
   Создание новых классов на основе существующих. Например, класс Expense может наследовать общие свойства и методы от базового класса Transaction, что позволяет расширять функциональность без дублирования кода.
3. **Полиморфизм**:  
   Возможность обработки объектов разных классов через общий интерфейс. В приложении это позволяет работать с различными типами операций (например, расходы и доходы) одинаково, используя общие методы для их обработки.

Использование ООП в разработке помогает сделать приложение более структурированным, гибким и удобным для расширения.

## **2**.**5. Использование PostgreSQL для хранения данных и Psycopg2**

Для создания простого приложения по учету расходов эффективным решением является использование реляционной базы данных PostgreSQL. Эта система управления базами данных обеспечивает надежность, высокую производительность и возможность масштабирования, что делает ее подходящей для хранения информации о расходах, таких как суммы, категории, даты и описания транзакций.

Для работы с PostgreSQL в Python используется библиотека **Psycopg2**, которая позволяет устанавливать соединение с базой данных, выполнять SQL-запросы и обрабатывать результаты. Через эту библиотеку можно легко реализовать операции добавления, извлечения и обновления данных, обеспечивая эффективное хранение и обработку информации о расходах.

Основные преимущества использования PostgreSQL для учета расходов включают:

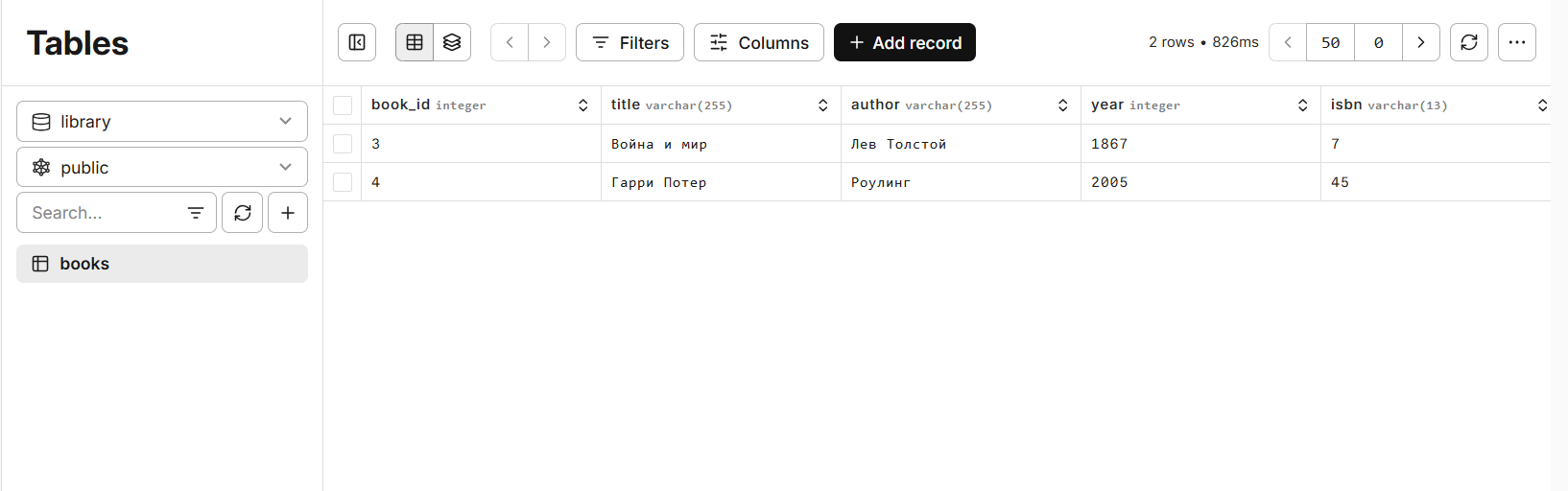
* **Управление транзакциями**, что гарантирует целостность данных.
* **Производительность**, позволяющая работать с большими объемами данных.
* **Масштабируемость**, что дает возможность увеличивать нагрузку с ростом объема данных.
* **Гибкость** в выполнении сложных SQL-запросов для анализа расходов.

PostgreSQL в сочетании с Psycopg2 представляет собой надежное и эффективное решение для хранения и управления данными о расходах.

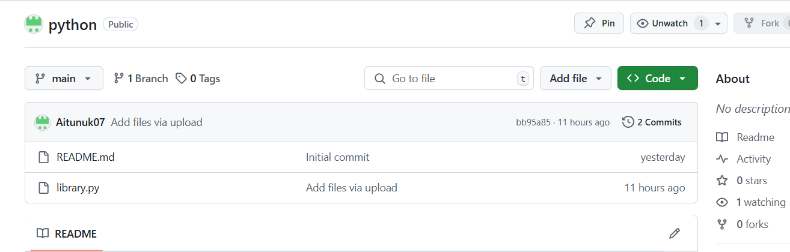
## **2.6 Сохранение данных в Neon.tech и GitHub**

Neon.tech — это облачная платформа баз данных с открытым исходным кодом, которая предоставляет полностью управляемые базы данных PostgreSQL с высокой доступностью и масштабируемостью.

В данном случае использована Neon.tech4 как хранение данных в таблице:



GitHub - это облачный хостинг репозиториев Git, или папок, где Git отслеживает изменения. Это Google Фото, только не для картинок, а для репозиториев. Но GitHub5 не только хранилище файлов с кодом. Эта платформа также предлагает безопасную авторизацию по токенам, инструменты тестирования и анализа кода, сервисы деплоя проектов — GitHub Actions



# 

# **Глава 3. Этапы жизненного цикла разработки программы**

При разработке программного обеспечения для учета расходов с сохранением данных на базе данных или файле, жизненный цикл разработки программного обеспечения включает несколько ключевых этапов, каждый из которых важен для успешного выполнения проекта. Рассмотрим эти этапы более детально в контексте

создания простого приложения для учета расходов.

## **3.1. Планирование и анализ требований**

На первом этапе важно сформулировать цели и задачи приложения для учета расходов, а также собрать информацию о потребностях пользователей. Анализ требований включает определение основных функциональных возможностей приложения, таких как:

1. Добавление, удаление и редактирование записей о расходах.
2. Классификация расходов по категориям (еда, транспорт, развлечения и т. д.).
3. Подсчет общей суммы расходов за определённый период.
4. Сохранение данных в файл или базу данных для последующего анализа.

Задачи этого этапа:

–Определить требования и ограничения, связанные с хранением данных (например, использование базы данных SQLite или простого текстового файла).

–Обсудить с заказчиком (или потенциальными пользователями) ключевые функции, которые должны быть реализованы в приложении.

–Прогнозирование стоимости разработки, определение сроков и необходимых ресурсов.

Результатом этого этапа будет документ, в котором подробно описаны требования к приложению и основные функциональные возможности.

## **3.2. Определение требований**

На этапе определения требований необходимо:

–Составить детализированную спецификацию всех требований к функционалу приложения.

–Разработать пользовательские сценарии, которые помогут понять, как будет использоваться приложение в реальных условиях.

–Обсудить формат хранения данных (например, JSON, CSV или использование базы данных SQLite).

–Разработать схемы интерфейса пользователя и базы данных.

Результатом этого этапа является точная документация, которая включает:

1.Описание всех функций приложения.

2Утвержденный список требований от заказчика.

1. Прототипы пользовательского интерфейса (если это необходимо).

## **3.3. Проектирование архитектуры**

На этапе проектирования архитектуры создается структура приложения и базы данных. Это включает:

–Разработку архитектуры приложения, которая должна быть масштабируемой и удобной для поддержки в будущем.

–Проектирование базы данных или файловой системы для хранения информации о расходах (например, создание таблиц в базе данных для хранения записей о расходах, категориях и пользователях).

–Определение взаимодействия между компонентами системы, таким как фронтенд и бэкенд (если приложение клиент-серверное).

–Выбор технологий для разработки (например, использование языка программирования Python с библиотекой SQLite для работы с базой данных или использование текстовых файлов для хранения данных)

Результатом этапа является архитектурная схема системы, включающая описание структуры приложения и базы данных.

## 

## **3.4. Разработка продукта**

На этом этапе происходит написание кода и реализация функционала. Основные компоненты приложения:

**Класс Expense**:

class Expense:

def \_\_init\_\_(self, expense\_id, amount, category, date):

self.expense\_id = expense\_id

self.amount = amount

self.category = category

self.date = date

**Класс ExpenseTracker**:

class ExpenseTracker:

def \_\_init\_\_(self):

self.connection = get\_db\_connection()

self.cursor = self.connection.cursor() if self.connection else None

self.create\_tables()

def create\_tables(self):

if not self.cursor:

return

try:

self.cursor.execute("""

CREATE TABLE IF NOT EXISTS expenses (

id SERIAL PRIMARY KEY,

amount REAL,

category TEXT,

date TEXT

);

""

)

self.cursor.execute("""

CREATE TABLE IF NOT EXISTS categories (

id SERIAL PRIMARY KEY,

name TEXT UNIQUE

);

""

)

self.cursor.execute("""

CREATE TABLE IF NOT EXISTS finance\_info (

id SERIAL PRIMARY KEY,

balance REAL,

expenses\_total REAL

);

""

)

self.connection.commit()

except Exception as e:

print(f"Ошибка создания таблиц: {e}")

**Графический интерфейс:**

–Основное окно для отображения баланса и списка расходов.

–Форма для добавления новых расходов.

–Окно для редактирования баланса.

–Дерево категорий с возможностью добавления и удаления категорий.

Приложение реализовано на **Tkinter** с использованием **PostgreSQL (Neon)** для хранения данных. Оно позволяет пользователю вести учет финансов, отслеживать баланс, управлять категориями расходов и анализировать свои траты.

## 

## **3.5. Тестирование и интеграция продукта**

Тестирование системы учета расходов является важным этапом разработки, позволяющим убедиться в ее корректной работе, надежности и удобстве использования. В процессе тестирования были проведены несколько видов проверок:

1. **Юнит-тестирование** — тестирование отдельных функций приложения, таких как добавление, редактирование и удаление записей о расходах. Это позволяет выявить ошибки на ранних стадиях разработки и убедиться в правильности работы каждой части кода. Пример юнит-теста для функции добавления расхода:

import unittest

from app import app

class TestExpenseSystem(unittest.TestCase):

def setUp(self):

self.app = app.test\_client()

def test\_index\_page(self):

response = self.app.get('/')

self.assertEqual(response.status\_code, 200)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

1. **Интеграционное тестирование** — проверка взаимодействия между различными компонентами приложения, включая работу с базой данных SQLite и PostgreSQL (Neon). Этот этап гарантирует, что данные корректно сохраняются, извлекаются и обновляются в базе.
2. **Тестирование пользовательского интерфейса** — оценка удобства использования графического интерфейса на Tkinter. Проверяется корректность отображения данных, интуитивность управления и отсутствие проблем при внесении записей о расходах.
3. **Нагрузочное тестирование** — моделирование работы системы при большом объеме данных или одновременном доступе множества пользователей (если система предусматривает сетевое взаимодействие). Оценивается скорость обработки запросов, устойчивость к нагрузкам и производительность базы данных.
4. **Проверка безопасности и стабильности** — анализ возможных уязвимостей, проверка устойчивости приложения к сбоям и некорректным данным. Например, тестируется защита от SQL-инъекций, обработка некорректных или пустых полей ввода.

После проведения всех тестов и исправления найденных ошибок система учета расходов была подготовлена к развертыванию. Убедившись в стабильной работе приложения, разработчик обеспечил его надежность, удобство и соответствие требованиям пользователей

## **3.6. Развертывание и обслуживание продукта**

После завершения разработки и тестирования приложение готово к развертыванию. Этот этап включает:

–Подготовку продукта к установке на устройства пользователей (для локальных приложений это может быть установка на ПК или мобильные устройства).

–Создание документации для пользователей (например, руководство пользователя или описание функционала).

–Развертывание приложения и запуск его в эксплуатацию.

После развертывания необходимо обеспечить постоянную поддержку:

–Регулярное обновление приложения с добавлением новых функций или исправлением выявленных ошибок.

–Обработка отзывов пользователей и внесение улучшений в интерфейс или функционал.

–Осуществление мониторинга работы приложения для выявления возможных сбоев или проблем.

Результатом этого этапа является работающее приложение, которое установлено и доступно пользователю, а также план по дальнейшему обслуживанию и поддержке приложения.

Жизненный цикл разработки простого приложения для учета расходов с сохранением данных на базе или файле состоит из нескольких последовательных этапов, каждый из которых важен для успешной реализации проекта. Правильное планирование и тщательная проработка каждого этапа позволяют создать качественное и удобное в использовании программное обеспечение.

# **4.Размещение в GitHub**

После завершения разработки системы было принято решение разместить проект на GitHub. Это обеспечило удобный доступ к коду, возможность совместной работы и контроля версий. Для этого были выполнены следующие команды:

git init

git add .

git commit -m "Initial commit"

git remote add origin <репозиторий>

git push -u origin main

Инициализация репозитория (git init) позволила превратить локальную папку с проектом в репозиторий Git. Затем команда git add . добавила все файлы проекта в индекс для последующего коммита. После этого выполнен первый коммит (git commit -m "Initial commit"), фиксирующий начальное состояние кода.

Далее репозиторий был привязан к удаленному хранилищу на GitHub с помощью команды git remote add origin <репозиторий>. Это дало возможность публиковать изменения в удаленном репозитории. Завершающим этапом стало выполнение git push -u origin main, что загрузило все файлы на сервер и установило отслеживание основной ветки.

Чтобы избежать публикации временных файлов и конфиденциальных данных, был добавлен файл gitignore. В нем перечислены файлы и директории, которые не должны попадать в репозиторий, например, временные файлы, файлы конфигурации среды разработки и локальные базы данных.

Кроме того, в процессе разработки были внедрены четкие правила именования коммитов. Каждое значимое изменение сопровождалось осмысленным комментарием, позволяющим легко отслеживать историю изменений и быстро находить нужные версии кода. Это значительно упростило работу с проектом и обеспечило удобство при дальнейших доработках.

Таким образом, размещение системы на GitHub не только облегчило процесс разработки и внесения улучшений, но и обеспечило возможность коллективной работы и резервного хранения кода.

# **Заключение**

Целью данной курсовой работы было создание простого приложения для учета расходов, которое позволяет добавлять, редактировать и удалять записи о расходах, а также сохранять данные в базе данных или текстовых файлах для последующего анализа. Для достижения поставленных целей были успешно выполнены задачи, определенные в рамках работы. В процессе разработки были изучены основы программирования на Python, освоена работа с библиотекой Tkinter для создания графического интерфейса, а также интегрирована база данных с использованием библиотеки SQLite для хранения информации о расходах.

В результате проделанной работы были достигнуты следующие основные выводы и результаты:

1. **Освоение технологий разработки:** В ходе работы были изучены и применены современные технологии, такие как Python и Tkinter, что позволило создать удобный и интуитивно понятный графический интерфейс. Библиотека Tkinter обеспечила возможность разработки визуальной части приложения, что сделало его доступным для пользователей с разным уровнем технической подготовки.
2. **Реализация функционала приложения:** Приложение предоставляет возможность добавлять, редактировать и удалять записи о расходах, а также сохранять и извлекать данные из базы данных или текстового файла. Это позволяет пользователям эффективно отслеживать свои расходы и анализировать финансовые потоки за определённый период.
3. **Интеграция базы данных:** Для хранения данных о расходах и их категориях была использована база данных SQLite. Это обеспечивает надежное хранение информации и быстрый доступ к ней, а также позволяет работать с большим объемом данных. В случае необходимости, данные могут быть легко экспортированы или импортированы.
4. **Улучшение пользовательского опыта:** Пользовательский интерфейс был разработан с учетом простоты использования и минимализма, что способствует комфортной работе с приложением. Интерфейс позволяет пользователю легко добавлять и редактировать данные, а также просматривать результаты учета расходов без лишних усилий.
5. **Тестирование и устранение ошибок:** Приложение прошло серию тестирований, в ходе которых были выявлены и устранены все ошибки. Это обеспечило стабильную работу программы, которая функционирует без сбоев, что важно для ее практического использования в реальных условиях.
6. **Практическое применение знаний:** В ходе работы были углублены навыки программирования на Python, а также освоены новые технологии, такие как работа с базами данных SQLite и создание графических интерфейсов. Эти знания позволили не только успешно реализовать поставленную задачу, но и расширить профессиональные компетенции в области разработки ПО.

В данной работе были выполнены все поставленные задачи, и цели – достигнуты. Созданное приложение представляет собой сочетание простоты в использовании и функциональности, что делает его полезным инструментом для учета расходов. Оно готово к использованию и может быть внедрено в повседневную практику пользователей, которые желают эффективно управлять своими финансами, отслеживать расходы и анализировать их на основе собранных данных.

# **Использованные источники**

1. ***Официальная документация по Python*** *-* [*https://www.nic.ru/help/osnovy-yazyka-programmirovaniya-python\_11662.html?utm\_source=google.com&utm\_medium=organic&utm\_campaign=google.com&utm\_referrer=google.com*](https://www.nic.ru/help/osnovy-yazyka-programmirovaniya-python_11662.html?utm_source=google.com&utm_medium=organic&utm_campaign=google.com&utm_referrer=google.com)
2. ***Руководство по PostSQL*** *-* [*https://skillbox.ru/media/code/postgresql-vsye-chto-nuzhno-znat-dlya-bystrogo-starta/*](https://skillbox.ru/media/code/postgresql-vsye-chto-nuzhno-znat-dlya-bystrogo-starta/)
3. ***Psycopg2 для работы с базами данных*** *-* [*https://www.geeksforgeeks.org/postgresql-python-querying-data/?ysclid=m85gmrj3z9318342174*](https://www.geeksforgeeks.org/postgresql-python-querying-data/?ysclid=m85gmrj3z9318342174)
4. ***Neon Tech для хранения данных в таблицах*** *-* [*https://console.neon.tech/app/projects/damp-hill-87096203/branches/br-icy-union-a8yiaxtb*](https://console.neon.tech/app/projects/damp-hill-87096203/branches/br-icy-union-a8yiaxtb)
5. ***Хранение кода на GitHub*** *-* [*https://github.com/nrjana/kursov.git*](https://github.com/nrjana/kursov.git)
6. ***Документация о жизненном цикле разработки программного обеспечения*** *-* [*https://blog.skillfactory.ru/glossary/sdlc/*](https://blog.skillfactory.ru/glossary/sdlc/)
7. ***Документация о этапах жизненного цикла разработки*** *-* [*https://habr.com/ru/articles/801531/*](https://habr.com/ru/articles/801531/)

# 

# **Листинг:**

import os

import psycopg2

import tkinter as tk

from tkinter import messagebox, ttk

from datetime import datetime

# Загрузить переменную окружения для подключения к Neon PostgreSQL

database\_url = os.getenv('postgresql://neondb\_owner:npg\_soDn7mHl2xJf@ep-red-haze-a85ugmii-pooler.eastus2.azure.neon.tech/neondb?sslmode=require')

# Проверка, что строка подключения получена

print(f"Database URL: {database\_url}")

# Connect to the Neon PostgreSQL database

conn = psycopg2.connect("dbname='neondb' user='neondb\_owner' password='npg\_soDn7mHl2xJf' host='ep-red-haze-a85ugmii-pooler.eastus2.azure.neon.tech' port='5432' sslmode='require'")

c = conn.cursor()

# Создаем таблицы для хранения данных

c.execute('''CREATE TABLE IF NOT EXISTS expenses (

    id SERIAL PRIMARY KEY,

    amount REAL,

    category TEXT,

    date TEXT

)''')

c.execute('''CREATE TABLE IF NOT EXISTS categories (

    id SERIAL PRIMARY KEY,

    name TEXT UNIQUE

)''')

c.execute('''CREATE TABLE IF NOT EXISTS finance\_info (

    id SERIAL PRIMARY KEY,

    balance REAL,

    expenses\_total REAL

)''')

conn.commit()

# Глобальные переменные для баланса и расходов

balance = 0.0

expenses\_total = 0.0

def load\_finance\_info():

    global balance, expenses\_total

    c.execute("SELECT \* FROM finance\_info WHERE id = 1")

    row = c.fetchone()

    if row:

        balance, expenses\_total = row[1], row[2]

    else:

        balance, expenses\_total = 0.0, 0.0

        c.execute("INSERT INTO finance\_info (balance, expenses\_total) VALUES (%s, %s)", (balance, expenses\_total))

        conn.commit()

def save\_finance\_info():

    global balance, expenses\_total

    c.execute("UPDATE finance\_info SET balance = %s, expenses\_total = %s WHERE id = 1", (balance, expenses\_total))

    conn.commit()

def update\_display():

    global balance, expenses\_total

    lbl\_balance.config(text=f"Общий баланс: {balance:.2f} сом")

    lbl\_expenses.config(text=f"Общие траты: {expenses\_total:.2f} сом")

def update\_balance():

    def save\_balance():

        global balance

        new\_balance = entry\_balance.get()

        try:

            balance = float(new\_balance)

            save\_finance\_info()  # Сохраняем изменения в базе

            update\_display()

            balance\_window.destroy()

        except ValueError:

            messagebox.showwarning("Ошибка", "Некорректный ввод!")

    balance\_window = tk.Toplevel(root)

    balance\_window.title("Редактировать баланс")

    balance\_window.geometry("380x200")

    ttk.Label(balance\_window, text="Введите новый баланс:", font=("Arial", 16)).pack(pady=16)

    entry\_balance = ttk.Entry(balance\_window)

    entry\_balance.pack(pady=10)

    ttk.Button(balance\_window, text="Сохранить", command=save\_balance).pack(pady=10)

def add\_category():

    def save\_category():

        category = entry\_category.get()

        if category:

            try:

                c.execute("INSERT INTO categories (name) VALUES (%s)", (category,))

                conn.commit()

                category\_window.destroy()

                update\_category\_list()

            except psycopg2.IntegrityError:

                messagebox.showwarning("Ошибка", "Такая категория уже существует!")

        else:

            messagebox.showwarning("Ошибка", "Введите название категории!")

    category\_window = tk.Toplevel(root)

    category\_window.title("Добавить категорию")

    category\_window.geometry("380x200")

    ttk.Label(category\_window, text="Введите категорию:", font=("Arial", 16)).pack(pady=16)

    entry\_category = ttk.Entry(category\_window)

    entry\_category.pack(pady=10)

    ttk.Button(category\_window, text="Добавить", command=save\_category).pack(pady=10)

def delete\_category():

    selected\_item = category\_tree.selection()

    if not selected\_item:

        messagebox.showwarning("Ошибка", "Выберите категорию для удаления!")

        return

    item = category\_tree.item(selected\_item)

    category\_id = item["values"][0]

    c.execute("DELETE FROM categories WHERE id = %s", (category\_id,))

    conn.commit()

    update\_category\_list()

# Глобальная переменная для хранения списка категорий

category\_tree = None

def update\_category\_list():

    """Обновляет список категорий в дереве"""

    global category\_tree

    if category\_tree is None:

        return  # Если окно категорий ещё не открыто, выходим из функции

    # Очищаем текущие записи в Treeview

    category\_tree.delete(\*category\_tree.get\_children())

    # Запрашиваем список категорий из базы данных

    c.execute("SELECT id, name FROM categories")

    rows = c.fetchall()

    # Заполняем дерево категориями

    for row in rows:

        category\_tree.insert("", "end", values=row)

def update\_table():

    tree.delete(\*tree.get\_children())

    c.execute("SELECT \* FROM expenses")

    for row in c.fetchall():

        tree.insert("", "end", values=row)

def delete\_expense():

    selected\_item = tree.selection()

    if not selected\_item:

        messagebox.showwarning("Ошибка", "Выберите запись для удаления!")

        return

    item = tree.item(selected\_item)

    expense\_id = item["values"][0]

    c.execute("DELETE FROM expenses WHERE id = %s", (expense\_id,))

    conn.commit()

    update\_table()

def add\_expense():

    def save\_expense():

        global balance, expenses\_total

        amount = entry\_amount.get()

        category = category\_var.get()

        date = datetime.now().strftime("%Y-%m-%d")

        if not amount or not category:

            messagebox.showwarning("Ошибка", "Введите сумму и выберите категорию!")

            return

        try:

            amount = float(amount)

            balance -= amount

            expenses\_total += amount

            save\_finance\_info()  # Сохраняем изменения в базе

            c.execute("INSERT INTO expenses (amount, category, date) VALUES (%s, %s, %s)", (amount, category, date))

            conn.commit()

            update\_table()

            update\_display()

            expense\_window.destroy()

        except ValueError:

            messagebox.showwarning("Ошибка", "Некорректная сумма!")

    expense\_window = tk.Toplevel(root)

    expense\_window.title("Добавить расход")

    expense\_window.geometry("500x300")

    ttk.Label(expense\_window, text="Введите сумму:").pack(pady=5)

    entry\_amount = ttk.Entry(expense\_window)

    entry\_amount.pack(pady=5)

    ttk.Label(expense\_window, text="Выберите категорию:").pack(pady=5)

    category\_var = tk.StringVar()

    c.execute("SELECT name FROM categories")

    categories = [row[0] for row in c.fetchall()]

    category\_menu = ttk.Combobox(expense\_window, textvariable=category\_var, values=categories, state="readonly")

    category\_menu.pack(pady=5)

    ttk.Button(expense\_window, text="Добавить", command=save\_expense).pack(pady=10)

def edit\_expense():

    selected\_item = tree.selection()

    if not selected\_item:

        messagebox.showwarning("Ошибка", "Выберите запись для редактирования!")

        return

    item = tree.item(selected\_item)

    expense\_id, amount, category, date = item["values"]

  def save\_changes():

        new\_amount = entry\_amount.get()

        new\_category = category\_var.get()

        try:

            new\_amount = float(new\_amount)

            c.execute("UPDATE expenses SET amount = %s, category = %s WHERE id = %s", (new\_amount, new\_category, expense\_id))

            conn.commit()

            update\_table()

            edit\_window.destroy()

        except ValueError:

            messagebox.showwarning("Ошибка", "Некорректная сумма!")

    edit\_window = tk.Toplevel(root)

    edit\_window.title("Редактировать расход")

    edit\_window.geometry("300x200")

    ttk.Label(edit\_window, text="Введите сумму:").pack(pady=5)

    entry\_amount = ttk.Entry(edit\_window)

    entry\_amount.insert(0, amount)

    entry\_amount.pack(pady=5)

    ttk.Label(edit\_window, text="Выберите категорию:").pack(pady=5)

    category\_var = tk.StringVar(value=category)

    c.execute("SELECT name FROM categories")

    categories = [row[0] for row in c.fetchall()]

    category\_menu = ttk.Combobox(edit\_window, textvariable=category\_var, values=categories, state="readonly")

    category\_menu.pack(pady=5)

    ttk.Button(edit\_window, text="Сохранить", command=save\_changes).pack(pady=10)

root = tk.Tk()

root.title("Мои финансы")

root.geometry("600x700")

root.configure(bg="#DCC6E0")

lbl\_title = ttk.Label(root, text="МОИ ФИНАНСЫ", font=("Arial", 18, "bold"), background="#DCC6E0", foreground="#4A235A")

lbl\_title.pack(pady=10)

lbl\_balance = ttk.Label(root, text="Общий баланс: 0.00 сом", font=("Arial", 14, "bold"), background="#DCC6E0", foreground="#4A235A")

lbl\_balance.pack(pady=5)

lbl\_expenses = ttk.Label(root, text="Общие траты: 0.00 сом", font=("Arial", 14, "bold"), background="#DCC6E0", foreground="#4A235A")

lbl\_expenses.pack(pady=5)

frame\_buttons = tk.Frame(root, bg="#DCC6E0")

frame\_buttons.pack(pady=10)

btn\_edit\_balance = ttk.Button(frame\_buttons, text="Редактировать баланс", command=update\_balance)

btn\_edit\_balance.pack(side="left", padx=10)

btn\_category\_list = ttk.Button(frame\_buttons, text="Категории", command=lambda: open\_category\_window())

btn\_category\_list.pack(side="left", padx=10)

# Функция для отображения окна с категориями

def open\_category\_window():

    """Открывает окно с категориями"""

    global category\_tree

    category\_window = tk.Toplevel(root)

    category\_window.title("Категории")

    category\_window.geometry("380x400")

    # Вставляем кнопки внутри окна категорий

    frame\_buttons = tk.Frame(category\_window, bg="#DCC6E0")

    frame\_buttons.pack(pady=10)

    btn\_add\_category = ttk.Button(frame\_buttons, text="Добавить категорию", command=add\_category)

    btn\_add\_category.pack(side="left", padx=10)

    btn\_delete\_category = ttk.Button(frame\_buttons, text="Удалить категорию", command=delete\_category)

    btn\_delete\_category.pack(side="left", padx=10)

    # Создание списка категорий

    category\_tree = ttk.Treeview(category\_window, columns=("ID", "Категория"), show="headings", height=10)

    category\_tree.heading("ID", text="ID")

    category\_tree.heading("Категория", text="Категория")

    category\_tree.pack(pady=10)

    update\_category\_list()  # Обновляем список категорий после создания окна

# Основное окно для расходов

tree = ttk.Treeview(root, columns=("ID", "Сумма", "Категория", "Дата"), show="headings")

tree.heading("ID", text="ID")

tree.heading("Сумма", text="Сумма")

tree.heading("Категория", text="Категория")

tree.heading("Дата", text="Дата")

tree.pack(pady=10)

frame\_buttons = tk.Frame(root, bg="#DCC6E0")

frame\_buttons.pack(pady=10)

btn\_add\_expense = ttk.Button(frame\_buttons, text="Добавить расход", command=add\_expense)

btn\_add\_expense.pack(side="left", padx=10)

btn\_edit = ttk.Button(frame\_buttons, text="Редактировать расход", command=edit\_expense)

btn\_edit.pack(side="left", padx=10)

btn\_delete = ttk.Button(frame\_buttons, text="Удалить расход", command=delete\_expense)

btn\_delete.pack(side="left", padx=10)

load\_finance\_info()

update\_display()

update\_table()

update\_category\_list()

root.mainloop()

conn.close()

c.close()