

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Пензенский государственный университет»
(ФГБОУ ВО «Пензенский государственный университет»)
кафедра «Математическое обеспечение и применение ЭВМ»

«УТВЕРЖДАЮ»

Зав. кафедрой Козлов А.Ю.

"__" _____ 2023 г.

Отчет
по производственной практике по получению профессиональных умений и
опыта профессиональной деятельности
на тему «Изучение языка программирования Python»

Автор работы	Левин М.В.
Специальность	09.03.04 «Программная инженерия»
Группа	20ВП1
Руководитель практики	к.т.н., доцент Казаков Б.В.
Работа защищена «__» _____ 2023 г.	Оценка _____

Пенза 2023 г.

Реферат

Отчет по производственной практике содержит 53 листа, 23 рисунка, 3 использованных источника.

PYTHON, SQLITE, SQL, ЗАПРОС, БАЗА ДАННЫХ, ТАБЛИЦА, ООП, НАСЛЕДОВАНИЕ, ПОЛИМОРФИЗМ

Объект разработки – программные средства для изучения и практического применения языка программирования Python.

Цель работы – ознакомление с основами языка Python, выполнение первых четырех лабораторных работ из методички Рубцова Т.П., получение практических навыков программирования на Python.

Технология разработки – структурная разработка программ с использованием языка Python и интегрированной среды разработки (IDE).

Результаты работы – овладение основами синтаксиса Python, выполнение четырех лабораторных работ, разработка программных решений на языке Python, приобретение навыков работы с IDE и отладчиком, понимание принципов разработки и отладки программ.

Оглавление

ВВЕДЕНИЕ	4
Лабораторная работа №1. Создание типа данных «класс».....	5
Лабораторная работа №2. Наследование и полиморфизм.....	11
Лабораторная работа №3. Работа с базами данных.....	19
Лабораторная работа №4. Связанные структуры данных	44
Заключение по практике.....	52
Использованные источники	53

ВВЕДЕНИЕ

Изучение языка программирования Python становится все более популярным среди начинающих и опытных программистов. Python обладает простым и понятным синтаксисом, что делает его идеальным выбором для тех, кто только начинает свой путь в программировании. Однако, несмотря на свою простоту, Python является мощным и универсальным языком, который может использоваться для разработки различных типов приложений, включая веб-сайты, научные вычисления, анализ данных, искусственный интеллект, игры и многое другое.

Целью данной работы является ознакомление с основами языка программирования Python и приобретение практических навыков в его использовании. Мы будем изучать основные концепции и синтаксис языка Python, включая работу с переменными, операторами, условными выражениями, циклами, функциями и работу с файлами и базой данных.

В процессе изучения будут выполнены следующие лабораторные работы из методички Рубцова Т.П.:

1. Создание типа данных «класс»
2. Наследование и полиморфизм
3. Работа с базами данных
4. Связанные структуры данных

Для каждой лабораторной работы мы разработаем и реализуем программные решения, которые позволят нам применить полученные знания на практике.

Лабораторная работа №1. Создание типа данных «класс»

Задание

1. Создать класс с полями, указанными в индивидуальном задании.
2. Реализовать в классе методы:
 - a. конструктор по умолчанию;
 - b. деструктор для освобождения памяти (с сообщением об уничтожении объекта);
 - c. функции обработки данных, указанные в индивидуальном задании;
 - d. функцию формирования строки информации об объекте.
3. Создать проект для демонстрации работы: сформировать объекты со значениями-константами и с введенными с клавиатуры значениями полей объекта. В основной ветке программы создайте три объекта класса. Вывести результаты работы на экран.

Таблица 1 – задание к лабораторной работе № 1

№ варианта	Класс-родитель и его поля	Функция-метод 1 обработки данных	Функция-метод 2 обработки данных
14	Товар: наименование, цена товара в рублях, изготовитель	Пересчитать цену товара в евро	Увеличить цену товара в евро, если название товара содержит слово «Samsung».

Ход выполнения

1. Создадим класс Product (товар) с указанными в задании полями [1]:

Листинг 1.1. – класса с указанными полями

```
class Product:
    def __init__(self):
        self.name = ''
        self.priceInRuble = 0.00
        self.manufacturer = ''
```

2. Реализуем указанные в задании методы. Конструктор по умолчанию показан на предыдущем листинге (листинг 1). Код для каждого из оставшихся подпунктов указан в соответствующих листингах 1.2. – 1.4.

Листинг 1.2. – деструктор класса

```
def __del__(self):
    print(f'Товар [{self.name}] УНИЧТОЖЕН !!!')
```

Листинг 1.3. - функции обработки данных

```
EURO_RATE = 100.42

def recalcPriceToEuro(self) -> float:
    return round(self.priceInRuble / Product.EURO_RATE, 2)

def increasePriceForSamsung(self, moneyInEuro: float):
    if 'Samsung' in self.name:
        self.priceInRuble += moneyInEuro * Product.EURO_RATE
```

Метод **recalcPriceToEuro()** пересчитывает цену товара в евро. Для этого цена, указанная в рублях делится на константу **EURO_RATE**, которая обозначает курс евро к рублю. Полученный результат округляется до двух знаков после запятой.

Метод **increasePriceForSamsung(moneyInEuro: float)** увеличивает цену товара в евро, если название товара содержит слово «Samsung». Для этого к цене, указанной в рублях, прибавляется величина **moneyInEuro** умноженная

на константу **EURO_RATE**, если в названии товара содержится строка «Samsung».

Листинг 1.4. - функция формирования строки информации об объекте

```
def __str__(self) -> str:
    s = f'Товар: {self.name};'
    s += f'Цена: {self.priceInRuble}Р;'
    s += f'Изготовитель: {self.manufacturer}'
    return f'{{{s}}}'
```

На листинге 1.5. приведен весь код для класса Product со всеми необходимыми комментариями.

Листинг 1.5. – весь код класса Product [1]

```
class Product:
    EURO_RATE = 100.42
    '''курс евро к рублю'''

    def __init__(self):
        self.name = ''
        """наименование товара"""
        self.priceInRuble = 0.00
        """цена товара в рублях"""
        self.manufacturer = ''
        """изготовитель товара"""

    def recalcPriceToEuro(self) -> float:
        """Пересчитать цену товара в евро"""
        return round(self.priceInRuble / Product.EURO_RATE, 2)

    def increasePriceForSamsung(self, moneyInEuro: float):
        """
            Увеличить цену товара в евро,
            если название товара содержит
            слово «Samsung».

            Параметры:
            moneyInEuro (float): сумма в евро, на которую нужно увеличить
        """
        if 'Samsung' in self.name:
            self.priceInRuble += moneyInEuro * Product.EURO_RATE

    def __str__(self) -> str:
        s = f'Товар: {self.name};'
        s += f'Цена: {self.priceInRuble}Р;'
        s += f'Изготовитель: {self.manufacturer}'
        return f'{{{s}}}'
```

```
def __del__(self):
    print(f'Товар [{self.name}] УНИЧТОЖЕН !!!')
```

3. Создадим главный файл для демонстрации работы. Код приведен в листинге 1.6.

Листинг 1.6. – код главного файла main.py

```
from Product import Product

if __name__ == '__main__':
    # создание товаров
    cpu = Product()
    cpu.name = 'Процессор Intel Core i5 12400F, LGA 1700, OEM'
    cpu.priceInRuble = 15590
    cpu.manufacturer = 'Intel'

    motherboard = Product()
    motherboard.name = 'Материнская плата ASROCK B660 STEEL LEGEND, LGA 1700, Intel B660, ATX, Ret'
    motherboard.priceInRuble = 15360
    motherboard.manufacturer = 'ASROCK'

    print('Введите необходимые параметры для создание товара:')

    phone = Product()
    phone.name = input('Название телефона: ')
    phone.priceInRuble = float(input('Цена (₽) телефона: '))
    phone.manufacturer = input('Изготовитель телефона: ')

    products = [cpu, motherboard, phone]

    print()

    # Вывод всех товаров
    print('Созданные товары:', '=' * 50, sep='\n')
    for product in products:
        print(product)
    print('=' * 50, '\n')

    # Вывод цен в евро для всех товаров
    print('Цены товаров в евро (€):', '=' * 50, sep='\n')
    for product in products:
        print(f'[{product.name}] = {product.recalcPriceToEuro()}€')
    print('=' * 50, '\n')

    INCREASE_VALUE_IN_EURO = 50

    # Увеличение цены для Sumsung'ов
    for product in products:
        product.increasePriceForSamsung(INCREASE_VALUE_IN_EURO)
```



```

# Вывод цен в евро после увеличение цен
print(f'Цены товаров в евро (€) после увелечение цены Sumsung\'ов на
{INCREASE_VALUE_IN_EURO}€:', '=' * 50, sep='\n')
for product in products:
    print(f'[{product.name}] = {product.recalcPriceToEuro()}€')
print('=' * 50, '\n')

```

Результат работы показан на рисунке 1.

```

marti@DESKTOP-9GPTH0M MINGW64 /d/Projects/Python/PGU_Practice (main)
$ python Laba1/main.py
Введите необходимые параметры для создание товара:
Название телефона: Samsung S7
Цена (P) телефона: 25999
Изготовитель телефона: Samsung

Созданные товары:
=====
{Товар: Процессор Intel Core i5 12400F, LGA 1700, OEM;Цена: 15590P;Изготовитель: Intel}
{Товар: Материнская плата ASROCK B660 STEEL LEGEND, LGA 1700, Intel B660, ATX, Ret;Цена: 15360P;Изготовитель: ASROCK}
{Товар: Samsung S7;Цена: 25999.0P;Изготовитель: Samsung}
=====

Цены товаров в евро (€):
=====
[Процессор Intel Core i5 12400F, LGA 1700, OEM] = 155.25€
[Материнская плата ASROCK B660 STEEL LEGEND, LGA 1700, Intel B660, ATX, Ret] = 152.96€
[Samsung S7] = 258.9€
=====

Цены товаров в евро (€) после увелечение цены Sumsung'ов на 50€:
=====
[Процессор Intel Core i5 12400F, LGA 1700, OEM] = 155.25€
[Материнская плата ASROCK B660 STEEL LEGEND, LGA 1700, Intel B660, ATX, Ret] = 152.96€
[Samsung S7] = 308.9€
=====

Товар [Материнская плата ASROCK B660 STEEL LEGEND, LGA 1700, Intel B660, ATX, Ret] УНИЧТОЖЕН !!!
Товар [Процессор Intel Core i5 12400F, LGA 1700, OEM] УНИЧТОЖЕН !!!
Товар [Samsung S7] УНИЧТОЖЕН !!!

```

Рисунок 1 – результат работы

Заключение

В процессе выполнения работы был создан класс Product (товар) с указанными в задании полями и методами. Был создан основной файл main.py, в котором создаются три объекта типа Product, у двух из которых поля заполняются константными значениями, а для последнего вводятся с клавиатуры. И демонстрируется работа для всех указанных в задании методов. В результате чего были получены знания и навыки по работе с классами в языке программирования Python.

Лабораторная работа №2. Наследование и полиморфизм

Задание

1. На основании предложенной предметной области спроектировать 3-4 класса, используя механизм наследования. Для каждого класса использовать отдельный модуль.
2. Предусмотреть у класса наличие полей, методов и свойств. Названия членов класса должны быть осмысленны и снабжены комментариями.
3. Один из наследников должен перегружать метод родителя.
4. Один из классов должен содержать виртуальный метод, который переопределяется в одном наследнике и не переопределяется в другом.
5. Продемонстрировать работу всех объявленных методов.
6. Продемонстрировать вызов конструктора родительского класса при наследовании.

Таблица 2 - варианты заданий к лабораторной работе № 2

Вариант	Задание
Вариант 14**	Написать программу, в которой описана иерархия классов: функция от одной переменной (экспонента, гиперболический синус, гиперболический косинус). Базовый класс должен иметь методы получения значения функции для данного значения переменной, а также создания экземпляра класса, представляющего собой производную текущего экземпляра. Продемонстрировать работу всех методов классов всех классов.

Ход выполнения работы

Создадим абстрактный класс для функция от одной переменной и назовем его SingleVariableFunc. На листинге 2.1. показан конструктор данного класса.

Листинг 2.1. – конструктор класса SingleVariableFunc

```
def __init__(self, name: str, desc: str):
    print('[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()')
    self.__name = name
    self.__desc = desc
```

Конструктор принимает два параметра: name, desc - они обозначают имя и описании функции соответственно.

Создадим свойство для класса, которое будет иметь только геттер. Код показан на листинге 2.2.

Листинг 2.2. – свойство класса SingleVariableFunc

```
@property
def name(self):
    return f'f(x) := {self.__name} - {self.__desc}'
```

Данное свойство возвращает полное имя функции, которое представляет собой конкатенацию двух строк: имя и описание функции.

Добавим в абстрактный класс метод общий для всех его потомков, но которой может быть ими переопределен. Код показан на листинге 2.3.

Листинг 2.3. – метод с реализацией в абстрактном классе SingleVariableFunc

```
def getUsefulInfo(self) -> str:
    return 'Нет полезной информации :('
```

Данный метод будет возвращать строку, представляющую собой некоторую полезную информацию о функции. По умолчанию её не будет, но потомки смогут переопределить данный метод.

Добавим в базовый класс абстрактные методы для получения значения функции для данного значения переменной, а также для создания экземпляра класса, представляющего собой производную текущего экземпляра. Код показан на листинге 2.4.

Листинг 2.4. – абстрактные методы класса SingleVariableFunc

```
@abstractmethod
def calc(self, x: float) -> float:
    """
        Вычислить значение функции

        Параметры
        x (float): Значение независимой переменной
    """

@abstractmethod
def getDerivative(self):
    """
        Получить экземпляр класса,
        представляющего собой производную
        текущего экземпляра
    """
```

Весь код базового класса SingleVariableFunc со всеми необходимыми комментариями показан на листинге 2.5.

Листинг 2.5. – весь код для класса SingleVariableFunc [1]

```
from abc import ABCMeta, abstractmethod

class SingleVariableFunc:
    __metaclass__ = ABCMeta

    def __init__(self, name: str, desc: str):
        print('[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()')
        self.__name = name
        """имя функции"""
        self.__desc = desc
        """описание функции"""

    @property
    def name(self):
        '''Полное имя функции (имя + словесное описание)'''
        return f'f(x) := {self.__name} - {self.__desc}'

    def getUsefulInfo(self) -> str:
        """
```

```

        Получить полезную
        информацию по функции
    """
    return 'Нет полезной информации :('

@abstractmethod
def calc(self, x: float) -> float:
    """
        Вычислить значение функции

        Параметры
        x (float): Значение независимой переменной
    """

@abstractmethod
def getDerivative(self):
    """
        Получить экземпляр класса,
        представляющего собой производную
        текущего экземпляра
    """

```

Создадим три класса: **HyperbolicSine**, **HyperbolicCosine**, **Exponent** - и унаследуем их от ранее созданного класса **SingleVariableFunc**. Они представляют собой гиперболический синус, гиперболический косинус и экспоненту соответственно. Код для этих классов приведен на листингах 2.6 - 2.8.

Листинг 2.6. – код для класса **HyperbolicSine**

```

class HyperbolicSine(SingleVariableFunc):
    def __init__(self):
        super().__init__(name='sinh(x)', desc='Гиперболический синус')

    def calc(self, x: float) -> float:
        return math.sinh(x)

    def getDerivative(self):
        return HyperbolicCosine()

    def getUsefulInfo(self) -> str:
        return 'Вид функции  $\sinh(x) = (\exp(x) - \exp(-x)) / 2$ '

```

Листинг 2.7. – код для класса **HyperbolicCosine**

```

class HyperbolicCosine(SingleVariableFunc):
    def __init__(self):
        super().__init__(name='cosh(x)', desc='Гиперболический косинус')

```

```

def calc(self, x: float) -> float:
    return math.cosh(x)

def getDerivative(self):
    return HyperbolicSine()

def getUsefulInfo(self) -> str:
    return 'Вид функции  $\cosh(x) = (\exp(x) + \exp(-x)) / 2$ '

```

Листинг 2.8. – код для класса Exponent

```

class Exponent(SingleVariableFunc):
    def __init__(self):
        super().__init__(name='exp(x)', desc='Экспонента')

    def calc(self, x: float) -> float:
        return math.exp(x)

    def getDerivative(self):
        return Exponent()

```

Класс HyperbolicSine и HyperbolicCosine перегружают метод родителя – **getUsefulInfo**, а класс Exponent использует реализацию по умолчанию.

Создадим главный файл, в котором продемонстрируем как работают созданные классы. Код приведен на листинге 2.9.

Листинг 2.9. – код главного файла main.py [1]

```

from HyperbolicFunctions import HyperbolicSine, HyperbolicCosine
from Exponent import Exponent
from SingleVariableFunc import SingleVariableFunc

def calc(func: SingleVariableFunc, x: float):
    print(f'ФУНКЦИЯ: {func.name}\n')

    print(f'Значения в точке x = {x}', func.calc(x), sep='\n')
    print()

    dfunc = func.getDerivative() # type: SingleVariableFunc

    print(f'Производная функции: {dfunc.name}')
    print()

    print(f'Значения в точке x = {x} для функции производной', dfunc.calc(x),
sep='\n')
    print()

```

```

    print('Полезная информация', func.getUsefulInfo(), sep='\n')

if __name__ == '__main__':
    x = 0.2

    sh = HyperbolicSine()
    calc(sh, x)

    print('*' * 75)

    ch = HyperbolicCosine()
    calc(ch, x)

    print('*' * 75)

    exp = Exponent()
    calc(exp, x)

```

Дополнительно создается метод `calc(func: SingleVariableFunc, x: float)`, который принимает два параметра. Первый из них представляют собой любого наследника класса `SingleVariableFunc`, а второй параметр – точка в которой нужно вычислить значение для переданной функции, а также для её производной.

Результат работы приведен на рисунке 2.


```

marti@DESKTOP-9GPTH0M MINGW64 /d/Projects/Python/PGU_Practice (main)
$ python Laba2/main.py
[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()
ФУНКЦИЯ:  $f(x) := \sinh(x)$  – Гиперболический синус

Значения в точке  $x = 0.2$ 
0.201336002541094

[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()
Производная функции:  $f(x) := \cosh(x)$  – Гиперболический косинус

Значения в точке  $x = 0.2$  для функции производной
1.020066755619076

Полезная информация
Вид функции  $\sinh(x) = (\exp(x) - \exp(-x)) / 2$ 
*****
[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()
ФУНКЦИЯ:  $f(x) := \cosh(x)$  – Гиперболический косинус

Значения в точке  $x = 0.2$ 
1.020066755619076

[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()
Производная функции:  $f(x) := \sinh(x)$  – Гиперболический синус

Значения в точке  $x = 0.2$  для функции производной
0.201336002541094

Полезная информация
Вид функции  $\cosh(x) = (\exp(x) + \exp(-x)) / 2$ 
*****
[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()
ФУНКЦИЯ:  $f(x) := \exp(x)$  – Экспонента

Значения в точке  $x = 0.2$ 
1.2214027581601699

[INFO]: ВЫЗОВ БАЗОВОГО КОНСТРУКТОРА SingleVariableFunc()
Производная функции:  $f(x) := \exp(x)$  – Экспонента

Значения в точке  $x = 0.2$  для функции производной
1.2214027581601699

Полезная информация
Нет полезной информации :(

```

Рисунок 2 – результат работы

Заключение

В процессе работы была создана иерархия классов: функция от одной переменной (экспонента, гиперболический синус, гиперболический косинус). Был выделен абстрактный класс, от которого унаследовались все остальные. В нем были созданы абстрактные методы, реализация которых была определена в классах наследниках, а также был создан обычный метод, который может быть перегружен в наследниках. Некоторые классы перегрузили данный метод, другие – нет. В результате работы были получены знания и навыки работы с ООП (наследование и полиморфизм) в языке программирования Python.

Лабораторная работа №3. Работа с базами данных

Задание

1. Представьте таблицы (согласно вашему варианту) в виде структур языка Python
2. Реализуйте в консоли интерфейс по добавлению, удалению, изменению данных. Имейте в виду, что связанные операции (удаление, добавление, изменение) для связанных таблиц, должны изменять данных во всех связанных структурах.
3. Реализуйте функционал по сохранению данных в файлы формата .csv и считыванию информации из файлов

Таблица 3 - варианты заданий к лабораторной работе №3

№ варианта	Выведите следующую информацию в консоль построчно:	Посчитайте и выведите результат:
14	Для каждого товара: «номер товара, «название товара», «стоимость товара», «цвет товара»	Для каждого цвета: количество товаров.



Ход выполнения

В качестве базы данных будем использовать **SQLite** – компактную встраиваемую СУБД. Для создания таблиц, указанных в задании, необходимо выполнить sql скрипт, который показан на листинге 3.1. [2]

Листинг 3.1. – sql скрипт для создания таблиц

```
CREATE TABLE IF NOT EXISTS color(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    color_name nvarchar(128) NOT NULL
);

CREATE TABLE IF NOT EXISTS type_product(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    type_name nvarchar(128) NOT NULL
);

CREATE TABLE IF NOT EXISTS product(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    product_name nvarchar(128) NOT NULL,
    price REAL NOT NULL,
    type_id INTEGER NOT NULL,
    availability INTEGER NOT NULL DEFAULT 0 CHECK(availability = 0 OR
availability = 1),
    color_id INTEGER NOT NULL,

    FOREIGN KEY (type_id) REFERENCES type_product (id) ON UPDATE CASCADE ON
DELETE CASCADE,
    FOREIGN KEY (color_id) REFERENCES color (id) ON UPDATE CASCADE ON DELETE
CASCADE
);
```

Для вызова данного скрипта из кода на Python была написана функция, показанная на листинге 3.2. [3]

Листинг 3.2. – выполнение sql скрипта из кода

```
def createSchemaSQLite(dbPath: str, scriptPath: str):
    conn = sqlite3.connect(dbPath)
    cur = conn.cursor()
    with open(scriptPath) as file:
        sql = file.read()
        cur.executescript(sql)
```

Первый параметр указывает на местоположение базы данных, второй указывает на местоположение sql скрипта.

Создадим в Python коде классы для соответствующих таблиц. Код для этих классов показан на листингах 3.3. – 3.5. [1]

Листинг 3.3. – код для класса Color (цвет)

```
class Color:
    def __init__(self, colorName: str):
        self.id = -1
        self.colorName = colorName

    def __str__(self) -> str:
        s = f'id: {self.id};'
        s += f'название: {self.colorName};'
        return f'Color {{{s}}}'
```

Листинг 3.4. – код для класса TypeProduct (тип товара)

```
class TypeProduct:
    def __init__(self, typeName: str):
        self.id = -1
        self.typeName = typeName

    def __str__(self) -> str:
        s = f'id: {self.id};'
        s += f'название: {self.typeName};'
        return f'TypeProduct {{{s}}}'
```

Листинг 3.5. – код для класса Product (товар)

```
from entity.Color import Color
from entity.TypeProduct import TypeProduct

class Product:
    def __init__(
        self,
        productName: str,
        price: float,
        typeProduct: TypeProduct,
        availability: int,
        color: Color):
        self.id = -1
        self.productName = productName
        self.price = price
        self.typeProduct = typeProduct
```

```

self.availability = availability
self.color = color

def __str__(self) -> str:
    s = f'id: {self.id};'
    s += f'название: {self.productName};'
    s += f'цена: {self.price};'
    s += f'тип: {self.typeProduct.typeName};'
    s += f'наличие в магазине: {self.availability};'
    s += f'цвет: {self.color.colorName}'
    return f'Product {{{s}}}'

```

Для удобства создания подключения (connection) к базе данных напишем класс, при создании которого необходимо будет указать местоположение базы данных. Код приведен на листинге 3.6.

Листинг 3.6. – код класса SQLiteConnector

```

import sqlite3

class SQLiteConnector:
    def __init__(self, path: str):
        self.path = path

    def connect(self) -> sqlite3.Connection:
        return sqlite3.connect(self.path)

```

Создадим класс ColorRepository, который будет общаться с таблицей Color в базе данных. В конструкторе он будет принимать экземпляр класса SQLiteConnector для подключения к базе. Он будет содержать следующие методы:

- getAll() – получение всех цветов;
- getById(colorId: int) – получение цвета по его идентификатору
- add(color: Color) – добавление цвета
- delete(colorId: int) – удаление цвета по его идентификатору
- update(updatedColor: Color) – обновление цвета

Код для класса ColorRepository с комментариями приведен в листинге 3.7.

Листинг 3.7. – код для класса ColorRepository

```

from data.SQLiteConnector import SQLiteConnector
from entity.Color import Color

class ColorRepository:
    def __init__(self, connector: SQLiteConnector):
        self.connector = connector

    def getAll(self) -> list[Color]:
        '''получить все цвета'''
        conn = self.connector.connect()
        cur = conn.cursor()
        cur.execute("""SELECT * FROM color""")

        colors = []
        for row in cur.fetchall():
            color = Color(row[1])
            color.id = row[0]
            colors.append(color)

        return colors

    def getById(self, colorId: int) -> Color:
        '''получить цвет по id'''
        conn = self.connector.connect()
        cur = conn.cursor()
        cur.execute("""SELECT * FROM color WHERE id = ?""", (colorId, ))
        result = cur.fetchone()

        if result is None:
            return None

        receivedId, colorName = result
        color = Color(colorName)
        color.id = receivedId
        return color

    def add(self, color: Color) -> int:
        '''добавить цвет'''
        if color.id == -1:
            conn = self.connector.connect()
            cur = conn.cursor()

            cur.execute("""
                INSERT INTO color
                (color_name)
                VALUES (?)""", (color.colorName, ))

            conn.commit()
            return cur.lastrowid

        return -1

```

```

def delete(self, colorId: int):
    '''удалить цвет по id'''
    conn = self.connector.connect()
    cur = conn.cursor()
    cur.execute("PRAGMA foreign_keys = ON")
    cur.execute("""DELETE FROM color WHERE id = ?""", (colorId, ))
    conn.commit()

def update(self, updatedColor: Color):
    '''обновить цвет'''
    if updatedColor.id != -1:
        conn = self.connector.connect()
        cur = conn.cursor()
        cur.execute("PRAGMA foreign_keys = ON")
        cur.execute("""
            UPDATE color SET
                color_name = ?
            WHERE id = ?""", (updatedColor.colorName, updatedColor.id, ))
        conn.commit()

```

В каждом из методов создается подключение к базе данных и выполняется одна из DML операций: SELECT, INSERT, UPDATE, DELETE – для получения, вставки, обновления или удаления соответственно.

Создадим класс `TypeProductRepository`, который будет общаться с таблицей `TypeProduct` в базе данных. В конструкторе он будет принимать экземпляр класса `SQLiteConnector` для подключения к базе. Он будет содержать следующие методы:

- `getById(typeId: int)` – получение типа товара по его идентификатору
- `add(typeProduct: TypeProduct)` – добавление типа товара
- `delete(typeProductId: int)` – удаление типа товара по его идентификатору
- `update(updatedTypeProduct: TypeProduct):` – обновление типа товара

Код для класса `TypeProductRepository` с комментариями приведен в листинге 3.8.

Листинг 3.8. – код для класса TypeProductRepository

```

from data.SQLiteConnector import SQLiteConnector
from entity.TypeProduct import TypeProduct

class TypeProductRepository:
    def __init__(self, connector: SQLiteConnector):
        self.connector = connector

    def getById(self, typeId: int):
        '''получить тип товара по id'''
        conn = self.connector.connect()
        cur = conn.cursor()
        cur.execute("""
            SELECT * FROM type_product
            WHERE id = ?""", (typeId, ))
        result = cur.fetchone()

        if result is None:
            return None

        receivedId, typeName = result
        typeProduct = TypeProduct(typeName)
        typeProduct.id = receivedId
        return typeProduct

    def add(self, typeProduct: TypeProduct) -> int:
        '''добавить тип товара'''
        if typeProduct.id == -1:
            conn = self.connector.connect()
            cur = conn.cursor()
            cur.execute("""
                INSERT INTO type_product
                (type_name)
                VALUES (?)""", (typeProduct.typeName, ))
            conn.commit()
            return cur.lastrowid

        return -1

    def delete(self, typeProductId: int):
        '''удалить тип товара по id'''
        conn = self.connector.connect()
        cur = conn.cursor()
        cur.execute("PRAGMA foreign_keys = ON")
        params = (typeProductId, )
        cur.execute("""DELETE FROM type_product WHERE id = ?""", params)
        conn.commit()

    def update(self, updatedTypeProduct: TypeProduct):
        '''обновить тип товара'''
        if updatedTypeProduct.id != -1:
            conn = self.connector.connect()
            cur = conn.cursor()

```

```

cur.execute("PRAGMA foreign_keys = ON")

params = (updatedTypeProduct.typeName, updatedTypeProduct.id, )
cur.execute("""
    UPDATE type_product SET
        type_name = ?
    WHERE id = ?""", params)

conn.commit()

```

Создадим класс `ProductRepository`, который будет общаться с таблицей `ProductRepository` в базе данных. В конструкторе он будет принимать экземпляр класса `SQLiteConnector` для подключения к базе. Он будет содержать следующие методы:

- `getAll()` – получение всех товаров
- `getId(productId: int)` – получение товара по его идентификатору
- `add(product: Product)` – добавление товара
- `delete(productId: int)` – удаление товара по его идентификатору
- `update(updatedProduct: Product):` – обновление товара

Код для класса `ProductRepository` с комментариями приведен в листинге 3.9.

Листинг 3.9. – код для класса `ProductRepository`

```

from entity.Product import Product
from entity.TypeProduct import TypeProduct
from entity.Color import Color
from data.SQLiteConnector import SQLiteConnector

class ProductRepository:
    def __init__(self, connector: SQLiteConnector):
        self.connector = connector

    def getAll(self) -> list[Product]:
        '''получить все товары'''
        conn = self.connector.connect()
        cur = conn.cursor()

        cur.execute("""
            SELECT
                p.id AS 'product_id',

```

```

        p.product_name AS 'product_name',
        p.price AS 'product_price',
        tp.id AS 'type_id',
        tp.type_name AS 'type_name',
        p.availability AS 'product_availability',
        c.id AS 'color_id',
        c.color_name AS 'color_name'
    FROM product p
    JOIN type_product tp ON p.type_id = tp.id
    JOIN color c ON p.color_id = c.id"""

    allProducts = []
    for row in cur.fetchall():
        receivedId, productName, price, typeProductId, typeProductName,
availability, colorId, colorName = row

        typeProduct = TypeProduct(typeProductName)
        typeProduct.id = typeProductId

        color = Color(colorName)
        color.id = colorId

        product = Product(productName, price, typeProduct, availability,
color)

        product.id = receivedId

        allProducts.append(product)

    return allProducts

def getById(self, productId: int) -> Product:
    '''получить товар по id'''
    conn = self.connector.connect()
    cur = conn.cursor()

    cur.execute("""
        SELECT
            p.id AS 'product_id',
            p.product_name AS 'product_name',
            p.price AS 'product_price',
            tp.id AS 'type_id',
            tp.type_name AS 'type_name',
            p.availability AS 'product_availability',
            c.id AS 'color_id',
            c.color_name AS 'color_name'
        FROM product p
        JOIN type_product tp ON p.type_id = tp.id
        JOIN color c ON p.color_id = c.id
        WHERE p.id = (?)""", (productId, ))

    result = cur.fetchone()

    if result is None:
        return None

```

```

        receivedId, productName, price, typeProductId, typeProductName,
        availability, colorId, colorName = result

        typeProduct = TypeProduct(typeProductName)
        typeProduct.id = typeProductId

        color = Color(colorName)
        color.id = colorId

    color)
    product = Product(productName, price, typeProduct, availability,
    product.id = receivedId

    return product

def add(self, product: Product) -> int:
    '''добавить товар'''
    if product.id == -1:
        conn = self.connector.connect()
        cur = conn.cursor()

        typeId = product.typeProduct.id
        if typeId == -1:
            cur.execute("""
                INSERT INTO type_product
                (type_name)
                VALUES (?)""", (product.typeProduct.typeName, ))
            typeId = cur.lastrowid

        colorId = product.color.id
        if colorId == -1:
            cur.execute("""
                INSERT INTO color
                (color_name)
                VALUES (?)""", (product.color.colorName, ))
            colorId = cur.lastrowid

        params = (product.productName, product.price, typeId,
        product.availability, colorId, )
        cur.execute("""
            INSERT INTO product
            (product_name, price, type_id, availability, color_id)
            VALUES
            (?, ?, ?, ?, ?)""", params)

        conn.commit()
        return cur.lastrowid

    return -1

def delete(self, productId: int):
    '''удалить товар по id'''
    conn = self.connector.connect()
    cur = conn.cursor()

```

```

cur.execute("""DELETE FROM product WHERE id = ?""", (productId, ))
conn.commit()

def update(self, updatedProduct: Product):
    '''обновить товар'''
    if updatedProduct.id != -1 and updatedProduct.typeProduct.id != -1
and updatedProduct.color.id != -1:
        conn = self.connector.connect()
        cur = conn.cursor()

        paramsForColor = (updatedProduct.color.colorName,
updatedProduct.color.id, )
        cur.execute("""
            UPDATE color SET
                color_name = ?
            WHERE id = ?""", paramsForColor)

        paramsForTypeProduct = (updatedProduct.typeProduct.typeName,
updatedProduct.typeProduct.id, )
        cur.execute("""
            UPDATE type_product SET
                type_name = ?
            WHERE id = ?""", paramsForTypeProduct)

        paramsForProduct = (
            updatedProduct.productName,
            updatedProduct.price,
            updatedProduct.typeProduct.id,
            updatedProduct.availability,
            updatedProduct.color.id,
            updatedProduct.id,)

        cur.execute("""
            UPDATE product SET
                product_name = ?,
                price = ?,
                type_id = ?,
                availability = ?,
                color_id = ?
            WHERE id = ? """, paramsForProduct)

        conn.commit()

```

Так как класс Product (товар) содержит поля типа Color (цвет) и TypeProduct (тип товара), то при выполнении всех команд кроме удаления нужно также делать дополнительные запросы к соответствующим таблицам.

Создадим функцию, которая для каждого цвета вернет количество товаров с этим цветом. Код приведен на листинге 3.10.

Листинг 3.10. – получение количества товаров для каждого цвета

```
from data.SQLiteConnector import SQLiteConnector

def countProductsForEachColor(
    connector: SQLiteConnector) -> list[tuple[str, int]]:
    '''посчитать количество товаров для каждого цвета'''
    conn = connector.connect()
    cur = conn.cursor()
    cur.execute("""
        SELECT
            c.color_name as 'color_name',
            count(1) AS 'count'
        FROM product p
        JOIN color c ON p.color_id = c.id
        GROUP BY c.color_name
    """)
    return cur.fetchall()
```

Данная функцию выполняет sql запрос с объединением таблиц Product и Color и делает группировку по названию цвета. В качестве агрегирующей функции используется **count** – подсчет строк.

Создание консольного интерфейса

Консольный интерфейс будет основываться на командах. Для его создания создадим класс Argument, представляющий собой аргумент для команды. Код приведен на листинге 3.11.

Листинг 3.11. – код для класса Argument

```
class Argument:
    def __init__(self, name: str, description: str):
        '''
            Параметры:
                name (str): имя аргумента

                description (str): описание аргумента
        '''
        self.name = name
        self.description = description

    def __str__(self) -> str:
        return f'{self.name} - {self.description}'
```

Создадим базовый класс команды BaseCommand. Код приведен на листинге 3.12.

```
from data.SQLiteConnector import SQLiteConnector
from abc import ABCMeta, abstractmethod

class BaseCommand:
    __metaclass__ = ABCMeta

    def __init__(self):
        self.name = ''
        self.description = ''
        self.args = []

    @abstractmethod
    def handle(self, params: list[str]):
        """Обработчик команды"""

    def help(self) -> str:
        """Подробная информация по команде"""
        params = list(map(lambda arg: f'[{arg.name}]', self.args))
        s = f'{self.description}: {self.name} {" ".join(params)}\n'
        if len(self.args) != 0:
            for arg in self.args:
                s += f' * {arg}\n'

        return s

    def __str__(self) -> str:
        params = list(map(lambda arg: f'[{arg.name}]', self.args))
        s = f'{self.description}: {self.name} {" ".join(params)}\n'
        return s
```

Данный класс имеет следующие поля:

- name – название команды;
- description – описание команды;
- args – список необходимых аргументов. Элементы списка являются экземплярами класса Argument.

Данный класс имеет следующие методы:

- handle(params: list[str]) - абстрактный метод, который является обработчиком этой команды. Он принимает список строк, представляющий собой аргументы.

- `help()` – подробная информация по команде

Создадим следующие классы команд и унаследуем их от базового класса команды – `BaseCommand`:

- `GetColorByIdCommand` - команда по получению цвета по `id`
- `AddColorCommand` - команда по добавлению цвета
- `DeleteColorCommand` - команда по удалению цвета
- `UpdateColorCommand` - команда по обновлению цвета
- `GetTypeProductByIdCommand` - команда по получению типа товара по `id`
- `AddTypeProductCommand` - команда по добавлению типа товара
- `DeleteTypeProductCommand` - команда по удалению типа товара
- `UpdateTypeProductCommand` - команда по обновлению типа товара
- `GetProductByIdCommand` - команда по получению товара по `id`
- `GetAllProductsCommand` - команда по получению всех товаров
- `AddProductCommand` - команда по добавлению товара
- `DeleteProductCommand` - команда по удалению товара
- `UpdateProductCommand` - команда по обновлению товара
- `CountProductsForEachColorCommand` - команда для подсчета количества товаров для каждого цвета
- `SaveColorsToCsvCommand` - команда для сохранения всех цветов в csv файл
- `LoadAllColorsFromCsvCommand` - команда для загрузки всех цветов из csv файла

Во всех командах, связанных по работе с цветом (`Color`), в методе `handle` вызывается соответствующий метод в классе `ColorRepository`. Аналогично, для типа товара (`TypeProduct`) вызывается соответствующий метод в классе `TypeProductRepository`, для товара (`Product`) вызывается соответствующий

метод в классе `ProductRepository`. В качестве примера возьмем класс `UpdateColorCommand`, его код приведен в листинге 3.13.

Листинг 3.13. – код для класса `UpdateColorCommand`

```
from data.repository.ColorRepository import ColorRepository
from consoleAPI.Argument import Argument
from consoleAPI.BaseCommand import BaseCommand

class UpdateColorCommand(BaseCommand):
    '''Команда по обновлению цвета'''
    def __init__(self, colorRepository: ColorRepository):
        super().__init__()
        self.colorRepository = colorRepository
        self.name = 'update_color'
        self.description = 'Обновление цвета'
        self.args = [
            Argument(name='prev_id', description='идентификатор цвета,
который нужно обновить'),
            Argument(name='new_name', description='новое название')
        ]

    def handle(self, params: list[str]):
        if len(params) != len(self.args):
            print('Передано неверное кол-во параметров')
            return

        color = self.colorRepository.getById(params[0])
        color.colorName = params[1]
        self.colorRepository.update(color)
        print(f'Цвет обновлен: {color}\n')
```

Экземпляр класса `ColorRepository` передается через конструктор. В теле конструктора указывается имя команды – `'update_color'`, описание команды – `'Обновление цвета'` и список необходимых аргументов, каждый из которых также содержит имя и описание.

В методе `handle` проверяется, что количество передаваемых аргументов соответствует ожидаемым. После чего с помощью поля класса типа `ColorRepository` вызывается метод `getById`, который возвращает из базы данных цвет по идентификатору, у полученного цвета меняется название, и вызывается метод `update`, который обновляет в базе данных соответствующий цвет.

Код всех остальных команд, за исключением: `CountProductsForEachColorCommand`, `LoadAllColorsFromCsvCommand`, `SaveColorsToCsvCommand` – аналогичен.

В команде **`CountProductsForEachColorCommand`** вызывает ранее написанная функция `countProductsForEachColor(connector: SQLiteConnector)`

В команде **`LoadAllColorsFromCsvCommand`** вызывается функция, код которой приведен на листинге 3.14. Данная функция загружает простые объекты из .csv файла

Листинг 3.14. – код для функции `loadSimpleObjectsFromCsv`

```
def loadSimpleObjectsFromCsv(path: str) -> list[dict]:
    """
        Загрузить простые объекты из csv файла

        Параметры:
            path (str): путь до csv файла

        Return:
            список объектов, считанных из csv файла
    """
    myFile = Path(path)
    if myFile.is_file():
        with open(path, encoding='utf-8') as rFile:
            file_reader = csv.reader(rFile, delimiter = ",")
            keys, rows = [], []
            count = 0
            for row in file_reader:
                if count == 0: keys = row
                else: rows.append(row)
                count += 1

            return [dict(zip(keys, row)) for row in rows]

    return None
```

Результат этой функции переводится из списка словарей в список с экземплярами класса `Color`.

В команде **`SaveColorsToCsvCommand`** вызывается функция, код которой приведен на листинге 3.15. Данная функция сохраняет простые объекты в .csv файл

Листинг 3.15. – код для функции saveColorsToCsvCommand

```
def saveSimpleObjectsInCsv(path: str, datas: list):
    """
        Сохранить простые объекты в csv файл

        Параметры:
            path (str): путь до csv файла

            datas (list): список объектов для сохранения
    """
    with open(path, mode='a+', encoding='utf-8') as file:
        for data in datas:
            keys = [
                attr
                for attr in dir(data)
                if not attr.startswith('__')]

            values = [
                str(getattr(data, attr))
                for attr in dir(data)
                if not attr.startswith('__')]

            fileWriter = csv.writer(file, delimiter = ',',
lineterminator='\r')
            file.seek(0)
            tmp = file.readline()[:-1]
            if tmp != ','.join(keys):
                fileWriter.writerow(keys)

            fileWriter.writerow(values)
```

Напишем основной файл программы main.py. Его код приведен на листинге 3.16.

```
if __name__ == '__main__':
    DB_PATH = './Laba3/db/laba3_db.db'
    connector = SQLiteConnector(DB_PATH)
    colorRepo = ColorRepository(connector)
    typeProductRepo = TypeProductRepository(connector)
    productRepo = ProductRepository(connector)

    # регистрация допустимых команд
    commands = [
        GetColorByIdCommand(colorRepo),
        AddColorCommand(colorRepo),
        DeleteColorCommand(colorRepo),
        UpdateColorCommand(colorRepo),

        GetTypeProductByIdCommand(typeProductRepo),
        AddTypeProductCommand(typeProductRepo),
        DeleteTypeProductCommand(typeProductRepo),
```

```

UpdateTypeProductCommand(typeProductRepo),

GetProductByIdCommand(productRepo),
GetAllProductsCommand(productRepo),
AddProductCommand(productRepo, colorRepo, typeProductRepo),
DeleteProductCommand(productRepo),
UpdateProductCommand(productRepo, colorRepo, typeProductRepo),

CountProductsForEachColorCommand(connector),
SaveColorsToCsvCommand(colorRepo),
LoadAllColorsFromCsvCommand()
]

# Вывод базовых команд
print('help - справка по всем командам')
print('help [command] - справка по команде')
print('q - выход из программы')

# цикл, запрашивающий ввод команды от пользователя
while True:
    print('> ', end='')
    parts = input().split(' ')

    if len(parts) == 0:
        print('Ошибка!!!')
        continue

    commandName = parts[0]

    if len(parts) == 1 and commandName == 'help':
        for command in commands:
            print(command)
        continue

    if len(parts) == 2 and commandName == 'help':
        for command in filter(lambda command: command.name == parts[1],
commands):
            print(command.help())
        continue

    if len(parts) == 1 and commandName == 'q':
        break

    filteredCommands = list(filter(lambda x: x.name == commandName,
commands))

    if len(filteredCommands) == 0:
        print('неизвестная команда')
        continue

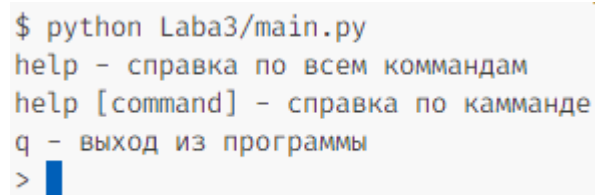
    command = filteredCommands[0] # type: BaseCommand
    command.handle(parts[1:])

```

В начале создается экземпляр класса `SQLiteConnector`. С помощью него создаются классы по работе с базой данных: `ColorRepository`, `TypeProductRepository`, `ProductRepository`. После чего создается список допустимых команд.

После создания необходимых объектов на экран выводится список с основными командами (`help`, `help [command]`, `q`). После чего у пользователя запрашивают строку. Введенная строка разбивается на слова по пробелу. Первое слово – название команды, все остальные – аргументы команды. По имени команды из списка выбирается нужная и у нее вызывается обработчик – метод `handle`.

Вид после запуска (рис. 3)



```
$ python Laba3/main.py
help - справка по всем командам
help [command] - справка по камманде
q - выход из программы
> 
```

Рисунок 3 – вид после запуска программы

Вид после ввода команды `help` (рис. 4)

```

$ python Laba3/main.py
help - справка по всем командам
help [command] - справка по команде
q - выход из программы
> help
Получение цвета по id: get_color_by_id [id]

Добавление цвета: add_color [name]

Удаление цвета: delete_color [id]

Обновление цвета: update_color [prev_id] [new_name]

Получение типа товара по id: get_type_product_by_id [id]

Добавление типа товара: add_type_product [name]

Удаление типа товара: delete_type_product [id]

Обновление типа товара: update_type_product [prev_id] [new_name]

Получение товара по id: get_product_by_id [id]

Получение всех товаров: get_all_products

Добавление товара: add_product [name] [price] [type_id] [availability] [color_id]

Удаление товара: delete_product [id]

Обновление товара: update_product [prev_id] [new_name] [new_price] [new_type_id] [new_availability] [new_color_id]

посчитать количество товаров для каждого цвета: count_products_for_each_color_command

Сохранить все цвета в csv файл: save_all_colors [path]

Загрузить все цвета из csv файла: load_all_colors [path]

```

Рисунок 4 – вид после ввода команды help

Вид после ввода команды help update_product (рис. 5)

```

> help update_product
Обновление товара: update_product [prev_id] [new_name] [new_price] [new_type_id] [new_availability] [new_color_id]
* prev_id - идентификатор товара, который нужно обновить
* new_name - новое название товара
* new_price - новая цена товара
* new_type_id - новый id типа товара
* new_availability - новое значение для 'наличие в магазине'
* new_color_id - новый id цвета

>

```

Рисунок 5 - Вид после ввода команды help update_product

Добавим тестовых данных в базу данных. После добавления таблицы будут выглядеть следующим образом (рис. 6 - 8):

	id	color_name
1	1	black
2	2	white
3	3	green
4	10	silver
5	13	pink

Рисунок 6 – данные в таблицы color

	id	type_name
1	1	Электроника
2	2	Телефоны
3	3	Аксессуары

Рисунок 7 – данные в таблице type_product

	id	product_name	price	type_id	availability	color_id
1	1	Беспроводные наушники A6RDots TWS	5750	1	0	3
2	2	Батарейки 2032, 3В, 2шт	205	1	1	1
3	3	Samsung S7	25999.67	2	1	1
4	4	IPhone 14 Pro MAX	149990.57	2	1	2
5	5	Кабельный органайзер	359	3	1	1
6	6	Чехол для хранения наручных часов	550	3	1	3

Рисунок 8 – данные в таблице product

Выведем все товары в консоль (рис. 9)

```
> get_all_products
Product {id: 1;название: Беспроводные наушники A6RDots TWS;цена: 5750.0;тип: Электроника;наличие в магазине: 0;цвет: green}

Product {id: 2;название: Батарейки 2032, 3В, 2шт;цена: 205.0;тип: Электроника;наличие в магазине: 1;цвет: black}

Product {id: 3;название: Samsung S7;цена: 25999.67;тип: Телефоны;наличие в магазине: 1;цвет: black}

Product {id: 4;название: IPhone 14 Pro MAX;цена: 149990.57;тип: Телефоны;наличие в магазине: 1;цвет: white}

Product {id: 5;название: Кабельный органайзер;цена: 359.0;тип: Аксессуары;наличие в магазине: 1;цвет: black}

Product {id: 6;название: Чехол для хранения наручных часов;цена: 550.0;тип: Аксессуары;наличие в магазине: 1;цвет: green}
```

Рисунок 9 – получение всех товаров

Добавим еще данные в базу данных с помощью консольного приложения (рис. 10).

```
> add_color lightgreen
Добавлен цвет: Color {id: 14;название: lightgreen;}

> add_type_product Стройматериалы
Добавлен тип: TypeProduct {id: 4;название: Стройматериалы;}

> add_product древесная_плитка 599 4 1 14
Передано неверное кол-во параметров
> add_product древесная_плитка 599 4 1 14
Добавлен товар: Product {id: 13;название: древесная_плитка;цена: 599;тип: Стройматериалы;наличие в магазине: 1;цвет: lightgreen}
```

Рисунок 10 – добавление данных в базу данных

В результате в базе появились соответствующие строки (рис. 11 - 13)

6	14	lightgreen
---	----	------------

Рисунок 11 – добавленный цвет

4	4	Стройматериалы
---	---	----------------

Рисунок 12 – добавленный тип товара

7	13	древесная_плитка	599	4	1	14
---	----	------------------	-----	---	---	----

Рисунок 13 – добавленный товар

Удалим только что добавленный цвет (рис. 14). Так как товар связан с цветом, то после удаления цвета, удалится и сам товар (рис. 15 - 16).

```
> delete_color 14
Цвет удален
```

Рисунок 14 – команда по удалению цвета

	id	color_name
1	1	black
2	2	white
3	3	green
4	10	silver
5	13	pink

Рисунок 15 – таблица после удаления цвета «lightgreen»

	id	product_name	price	type_id	availability	color_id
1	1	Беспроводные наушники A6RDots TWS	5750	1	0	3
2	2	Батарейки 2032, 3В, 2шт	205	1	1	1
3	3	Samsung S7	25999.67	2	1	1
4	4	iPhone 14 Pro MAX	149990.57	2	1	2
5	5	Кабельный органайзер	359	3	1	1
6	6	Чехол для хранения наручных часов	550	3	1	3

Рисунок 16 – таблица после удаления товара «древесная_плитка»

Выполним команду по подсчету количества товара для каждого цвета (рис. 17).

```
> count_products_for_each_color_command
black: 3
green: 2
white: 1
```

Рисунок 17 - подсчет количества товара для каждого цвета

Сохраним все цвета из базы данных в .csv файл (рис. 18 - 19).

```
> save_all_colors ./Laba3/all_colors.csv
Цвета сохранены в файл ./Laba3/all_colors.csv
```

Рисунок 18 – выполнение команды по сохранению цветов в .csv файл

Preview 'all_colors.csv' X		
	Color Name▼	Id ▼
	black	1
	white	2
	green	3
	silver	10
	pink	13

Рисунок 19 – результат сохранения всех цветов в .csv файл

Загрузим все цвета из .csv файла (рис. 20).

```
> load_all_colors ./Laba3/all_colors.csv  
Color {id: 1;название: black;}  
Color {id: 2;название: white;}  
Color {id: 3;название: green;}  
Color {id: 10;название: silver;}  
Color {id: 13;название: pink;}  
> █
```

Рисунок 20 – загрузка всех цветов из .csv файла

Заключение

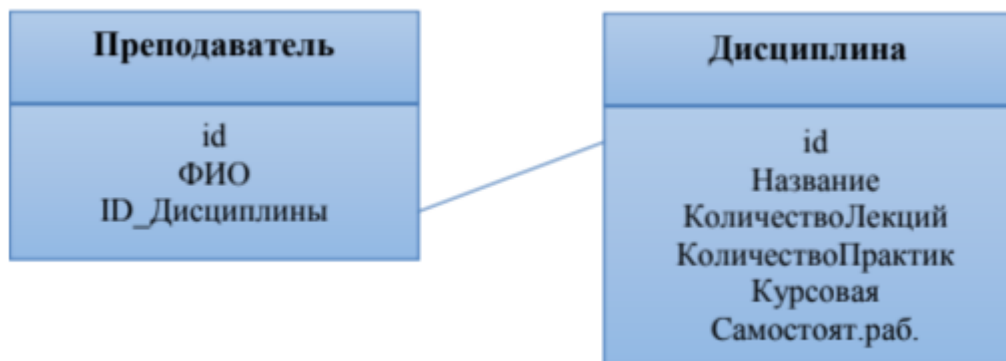
В процессе работы была создана SQLite база данных с указанными в задании таблицами. Для каждой из таблиц был создан соответствующий класс в Python. Для работы с таблицами в базе данных были созданы отдельные классы репозитории, которые и выполняли все SQL инструкции. Консольный интерфейс был реализован на командах. Для каждой команды был создан отдельный класс с обработчиком (метод `handle`). Была написана дополнительная функция, которая считает количество товаров для каждого цвета. Была написана функция, которая сохраняет цвета в .csv файл, а также функция, считывающая цвета из .csv файла.

В результате работы были получены знания и навыки по работе с базой данных в Python, а также работе с .csv файлами.

Лабораторная работа №4. Связанные структуры данных

Задание

1. Пусть дана база данных (приведена ниже). Используйте нужные структуры данных для ее хранения. Заполните БД. Выведите список всех преподавателей.
2. Для БД из задания 1 выведите список всех дисциплин преподавателя «Иванов И.И.», по которым предусмотрена курсовая работа и самостоятельная работа.
3. Для БД из задания 1 выведите все дисциплины, в чьем названии встречается буква «П».



Ход выполнения

В качестве базы данных будем использовать **SQLite** – компактную встраиваемую СУБД. Для создания таблиц, указанных в задании, необходимо выполнить sql скрипт, который показан на листинге 4.1. [2]

Листинг 4.1. – sql скрипт для создания таблиц

```
CREATE TABLE IF NOT EXISTS subject(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name NVARCHAR(128) NOT NULL,
    count_lectures INTEGER NOT NULL,
    count_practices INTEGER NOT NULL,
    there_is_coursework INTEGER NOT NULL DEFAULT 0 CHECK(there_is_coursework
= 0 OR there_is_coursework = 1),
    there_is_selfwork INTEGER NOT NULL DEFAULT 0 CHECK(there_is_selfwork = 0
OR there_is_selfwork = 1)
);

CREATE TABLE IF NOT EXISTS teacher(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    FIO NVARCHAR(128) NOT NULL,
    subject_id INTEGER NOT NULL,

    FOREIGN KEY (subject_id) REFERENCES subject (id) ON UPDATE CASCADE ON
DELETE CASCADE
);
```

Для вызова данного скрипта из кода на Python была написана функция, показанная на листинге 4.2. [3]

Листинг 3.2. – выполнение sql скрипта из кода

```
def createSchemaSQLite(dbPath: str, scriptPath: str):
    conn = sqlite3.connect(dbPath)
    cur = conn.cursor()
    with open(scriptPath) as file:
        sql = file.read()
        cur.executescript(sql)
```

Первый параметр указывает на местоположение базы данных, второй указывает на местоположение sql скрипта.

Создадим в Python коде классы для соответствующих таблиц. Код для этих классов показан на листингах 4.3. – 4.4.

Листинг 4.3. – код для класса Subject

```

class Subject:
    def __init__(
        self,
        name: str,
        count_lectures: int,
        count_practices: int,
        there_is_coursework: int,
        there_is_selfwork: int):
        self.id = -1
        self.name = name
        self.count_lectures = count_lectures
        self.count_practices = count_practices
        self.there_is_coursework = there_is_coursework
        self.there_is_selfwork = there_is_selfwork

    def __str__(self) -> str:
        s = f'id: {self.id};'
        s += f'Название: {self.name};'
        s += f'КоличествоЛекций: {self.count_lectures};'
        s += f'КоличествоПрактик: {self.count_practices};'
        s += f'Курсовая: {self.there_is_coursework};'
        s += f'Самостоятельная: {self.there_is_selfwork};'
        return f'Дисциплина: {{{s}}}'

```

Листинг 4.4. – код для класса Teacher

```

class Teacher:
    def __init__(
        self,
        FIO: str,
        subjectId: int):
        self.id = -1
        self.FIO = FIO
        self.subjectId = subjectId

    def __str__(self) -> str:
        s = f'id: {self.id};'
        s += f'ФИО: {self.FIO};'
        s += f'ID_Дисциплины: {self.subjectId}'
        return f'Преподаватель: {{{s}}}'

```

Для удобства создания подключения (connection) к базе данных напишем класс, при создании которого необходимо будет указать местоположение базы данных. Код приведен на листинге 4.5.

Листинг 4.5. – код класса SQLiteConnector

```
import sqlite3

class SQLiteConnector:
    def __init__(self, path: str):
        self.path = path

    def connect(self) -> sqlite3.Connection:
        return sqlite3.connect(self.path)
```

Напишем функцию, которая выводит список всех дисциплин для преподавателя с указанными ФИО, по которым предусмотрена курсовая работа и самостоятельная работа. Код приведен на листинге 4.6.

Листинг 4.6. – код функции getSubjectsWithCourseworkAndSelfwork

```
from SQLiteConnector import SQLiteConnector
from entity.Subject import Subject

def getSubjectsWithCourseworkAndSelfwork(
    connector: SQLiteConnector,
    teacherFIO: str) -> list[Subject]:
    ...
    Получить все дисциплин для преподавателя,
    по которым предусмотрена курсовая работа
    и самостоятельная работа
    ...
    conn = connector.connect()
    cur = conn.cursor()

    params = (teacherFIO, 1, 1, )
    cur.execute("""
        SELECT
            s.id,
            s.name,
            s.count_lectures,
            s.count_practices,
            s.there_is_coursework,
            s.there_is_selfwork
        FROM teacher t
        JOIN subject s ON t.subject_id = s.id
        WHERE
            t.FIO = ? and
            s.there_is_coursework = ? and
            s.there_is_selfwork = ?""", params)

    subjects = []
    for row in cur.fetchall():
        subject = Subject(row[1], int(row[2]), int(row[3]), int(row[4]),
            int(row[5]))
```

```

        subject.id = int(row[0])
        subjects.append(subject)

    return subjects

```

Данная функция делает sql запрос, объединяющий таблицы teacher и subject, и фильтрует результат по ФИО преподавателя, а также по наличию курсовой и самостоятельной работ.

Напишем функцию, которая выводит все дисциплины, в чьем названии встречается строка с указанным шаблоном. Код приведен на листинге 4.7.

Листинг 4.7. – код функции getSubjectWithLike

```

from entity.Subject import Subject
from SQLiteConnector import SQLiteConnector

def getSubjectWithLike(
    connector: SQLiteConnector,
    like: str) -> list[Subject]:
    ...
    Получить все дисциплины, в чьем названии
    встречается строка с указанным шаблоном
    ...

    conn = connector.connect()
    cur = conn.cursor()
    cur.execute(f'SELECT * FROM subject WHERE name LIKE \'{like}\'' )

    subjects = []
    for row in cur.fetchall():
        subject = Subject(row[1], int(row[2]), int(row[3]), int(row[4]),
int(row[5]))
        subject.id = int(row[0])
        subjects.append(subject)

    return subjects

```

Данная функция выполняет sql запрос к таблице subject и фильтрует строки по названию предмета используя шаблон.

Добавим данные в базу данных. В результате получим следующие таблицы с данными (рис. 21 - 22).

	id	FIO	subject_id
1	1	Иванов И.И.	1
2	2	Смирнова М.О.	2
3	3	Петров А.А.	3
4	4	Смирнова Е.А.	4
5	5	Иванов А.П.	5
6	6	Петрова О.С.	6
7	7	Козлова Н.И.	7
8	8	Сидоров Д.П.	8
9	9	Васильева Е.С.	9
10	10	Иванов И.И.	6
11	11	Иванов И.И.	9

Рисунок 21 – данные в таблице teacher

	id	name	count_lectur	count_pract	there_is_coi	there_is_sel
1	1	Математика	3	2	1	0
2	2	Физика	4	3	0	1
3	3	Химия	2	2	1	1
4	4	Русский язык	5	4	0	1
5	5	История	3	2	1	0
6	6	Литература	4	3	1	1
7	7	Биология	3	2	1	1
8	8	География	2	3	0	1
9	9	Английский язык	4	4	1	1

Рисунок 22 – данные в таблице subject

Напишем главный файл программы main.py. Его код приведен на листинге 4.8.

Листинг 4.8. – код главного файла main.py

```

if __name__ == '__main__':
    DB_PATH = './Laba4/db/laba4_db.db'
    connector = SQLiteConnector(DB_PATH)

    ...
    Получить список всех дисциплин преподавателя «Иванов И.И.»,
    по которым предусмотрена курсовая работа и
    самостоятельная работа.
    ...
    res1 = getSubjectsWithCourseworkAndSelfwork(connector, 'Иванов И.И.')
    for row in res1: print(row)

```

```
print('=' * 75)

# Получить все дисциплины, в чьем названии встречается буква «т».
res2 = getSubjectWithLike(connector, '%т%')
for row in res2: print(row)
```

В начале создает экземпляр класса SQLiteConnector для создания подключений к базе данных. С его помощью вызываются ранее написанные функции. Результат работы показан на рисунке 23.

```
marti@DESKTOP-9GPTH0M MINGW64 /d/Projects/Python/PGU_Practice (main)
$ python Laba4/main.py
Дисциплина: {id: 6;Название: Литература;КоличествоЛекций: 4;КоличествоПрактик: 3;Курсовая: 1;Самостоятельная: 1;}
Дисциплина: {id: 9;Название: Английский язык;КоличествоЛекций: 4;КоличествоПрактик: 4;Курсовая: 1;Самостоятельная: 1;}
=====
Дисциплина: {id: 1;Название: Математика;КоличествоЛекций: 3;КоличествоПрактик: 2;Курсовая: 1;Самостоятельная: 0;}
Дисциплина: {id: 5;Название: История;КоличествоЛекций: 3;КоличествоПрактик: 2;Курсовая: 1;Самостоятельная: 0;}
Дисциплина: {id: 6;Название: Литература;КоличествоЛекций: 4;КоличествоПрактик: 3;Курсовая: 1;Самостоятельная: 1;}
```

Рисунок 23 – результат работы

Заключение

В процессе работы была создана SQLite база данных с указанными в задании таблицами. Для каждой из таблиц был создан соответствующий класс в Python. Были реализованы следующие функции:

- `getSubjectsWithCourseworkAndSelfwork` – получение всех дисциплин преподавателя с указанным ФИО, по которым предусмотрена курсовая работа и самостоятельная работа.
- `getSubjectWithLike` – получение всех дисциплины, в чьем названии встречается строка с указанным шаблоном

В результате работы были получены знания и навыки по работе со связанными структурами данных.

Заключение по практике

В ходе работы были изучены основы языка Python, включая синтаксис, переменные, операторы, условные выражения, циклы, функции, работа с файлами и SQLite базой данных. Были выполнены четыре лабораторные работы, в которых решались задачи различной сложности с использованием Python. В процессе работы была использована методичка Рубцова Т.П. «Лабораторный практикум по программированию на языке Python». Для разработки программ использовалась интегрированная среда разработки (IDE), что позволило эффективно создавать и отлаживать программные решения.

В результате выполнения работы были достигнуты поставленные цели, получены навыки программирования на языке Python, а также понимание основных принципов разработки и отладки программ.

Использованные источники

1. Python [Электронный ресурс]. URL: <https://metanit.com/python/tutorial>
(дата обращения 10.07.2023)
2. SQLite [Электронный ресурс]. URL: <https://metanit.com/sql/sqlite>
(дата обращения 10.07.2023)
3. Работа с SQLite в Python [Электронный ресурс]. URL: <https://metanit.com/python/database>