Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

# ОТЧЕТ

## По лабораторной работе №7

## Дисциплины «Объектно-ориентированное программирование»

Выполнил:

Пустяков Андрей Сергеевич

3 курс, группа ИВТ-б-о-22-1,

09.03.01 «Информатика и вычислительная техника (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем», очная форма обучения

_____

(подпись)

Руководитель практики:

Воронкин Р. А., доцент департамента цифровых и робототехнических систем и электроники института перспективной инженерии

_____

(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2024 г.

Тема: Основы работы с Tkinter.

Цель: приобрести навыки построения графического интерфейса пользователя GUI с помощью пакета Tkinter языка программирования Python версии 3.x.

Ход работы:

Выполнение заданий:

Задание 7.

Необходимо написать простейший калькулятор, состоящий из двух текстовых полей, куда пользователь вводит числа, и четырех кнопок «+», «-», «*», «/». Результат вычисления должен отображаться в метке. Если арифметическое действие выполнить невозможно (например, если были введены буквы, а не числа), то в метке должно появляться слово "ошибка".

Код программы, реализующей данный калькулятор:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Напишите простейший калькулятор, состоящий из двух текстовых полей,
# куда пользователь вводит числа, и четырех кнопок "+", "-", "*", "/".
# Результат вычисления должен отображаться в метке. Если арифметическое
# действие выполнить невозможно (например, если были введены буквы, а
# не числа), то в метке должно появляться слово "ошибка".

import tkinter as tk
from typing import Union


class Calculator:
    """
    Класс калькулятора, реализующий базовые арифметические операции.
    """

    def calculate(self, first: float, second: float, operation: str) ->
Union[float, str]:
        """
        Выполняет арифметические операции (сложение, вычитание, умножение,
деление).

        :param first: Первое число.
        :param second: Второе число.
        :param operation: Строка, обозначающая требуемую операцию: "+", "-",
"*", "/".
        :return: Результат вычисления (float или "ошибка" при делении на ноль
или неверном типе операции).
        """

        if operation == "+":
            return first + second
        elif operation == "-":
            return first - second
```

```
        elif operation == "*":
            return first * second
        elif operation == "/":
            # Проверяем деление на ноль
            return first / second if second != 0 else "ошибка"
        return "ошибка"


class CalculatorApp:
    """
    Класс, отвечающий за графический интерфейс калькулятора на базе Tkinter.
    """

    def __init__(self, root: tk.Tk, calculator: Calculator) -> None:
        """
        Инициализирует приложение, создавая все необходимые элементы
интерфейса.

        :param root: Окно (экземпляр tk.Tk), в котором будут размещаться все
элементы.
        :param calculator: Экземпляр класса Calculator, выполняющий
арифметические операции.
        """

        self.root = root
        self.calculator = calculator
        self.create_widgets()

    def create_widgets(self) -> None:
        """
        Создаёт и размещает на форме все виджеты: поля ввода, кнопки операций
и метку для результата.
        """

        # Поля ввода для двух чисел
        self.entry1 = tk.Entry(self.root, width=10)
        self.entry1.grid(row=0, column=0, columnspan=2, padx=5, pady=5)

        self.entry2 = tk.Entry(self.root, width=10)
        self.entry2.grid(row=1, column=0, columnspan=2, padx=5, pady=5)

        # Кнопки операций
        btn_add = tk.Button(self.root, text="+", width=5, command=lambda:
self.calculate("+"))
        btn_add.grid(row=2, column=0, columnspan=2, padx=5, pady=2)

        btn_sub = tk.Button(self.root, text="-", width=5, command=lambda:
self.calculate("-"))
        btn_sub.grid(row=3, column=0, columnspan=2, padx=5, pady=2)

        btn_mul = tk.Button(self.root, text="*", width=5, command=lambda:
self.calculate("*"))
        btn_mul.grid(row=4, column=0, columnspan=2, padx=5, pady=2)

        btn_div = tk.Button(self.root, text="/", width=5, command=lambda:
self.calculate("/"))
        btn_div.grid(row=5, column=0, columnspan=2, padx=5, pady=2)

        # Метка для отображения результата
        self.result_label = tk.Label(self.root, text="", font=("Arial", 14))
        self.result_label.grid(row=6, column=0, columnspan=2, pady=10)

    def calculate(self, operation: str) -> None:
        """
```

```
        Получает значения из полей ввода, вызывает метод calculate() у
объекта Calculator
        и отображает результат в метке. Если введены неверные данные, выводит
"ошибка".

        :param operation: Строка, обозначающая требуемую операцию: "+", "-",
"*", "/".
        :return: None
        """

        try:
            num1 = float(self.entry1.get())
            num2 = float(self.entry2.get())
            result = self.calculator.calculate(num1, num2, operation)
            self.result_label.config(text=str(result))
        except ValueError:
            self.result_label.config(text="ошибка")


def main() -> None:
    """
    Главная функция: создаёт окно, инициализирует приложение калькулятора и
запускает главный цикл.
    """

    root = tk.Tk()
    root.title("Калькулятор")

    # Создаем экземпляры калькулятора и приложения
    calculator = Calculator()
    app = CalculatorApp(root, calculator)

    # Запускаем основной цикл обработки событий
    root.mainloop()


if __name__ == "__main__":
    main()
```

Результаты работы данной программы при указании некоторых вещественных чисел и выбора некоторой математической операции в окне интерфейса (рис. 1).

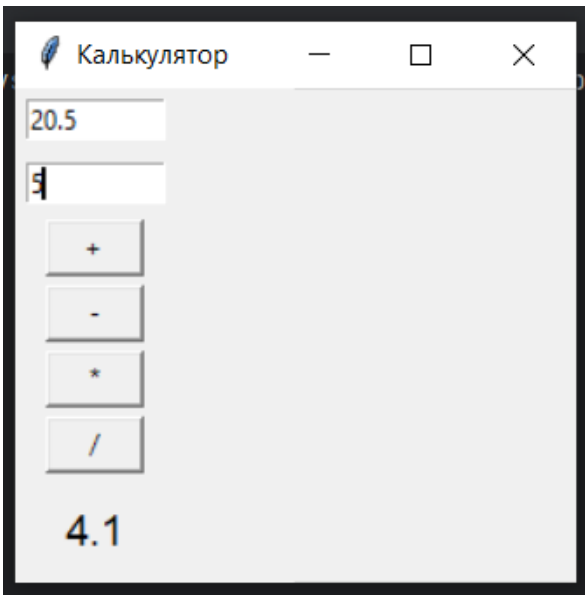


Рисунок 1 – Результаты работы программы задания 7

Для тестирования данного задания был написан unit-тест, тестирующий некоторые методы программы на корректность вычисления результата и выпадения исключения и сообщения об ошибке (рис. 2).

```
PS C:\Users\Andrey\Desktop\ООП\Лабораторная_работа_7\Object-Oriented_Programming_laboratory_work_7> pytest -v
============================= test session starts ==============================
cachedir: .pytest_cache
rootdir: C:\Users\Andrey\Desktop\ООП\Лабораторная_работа_7\Object-Oriented_Programming_laboratory_work_7
configfile: pytest.ini
plugins: anyio-4.2.0
collected 7 items

tests/test_task_1.py::TestCalculator::test_addition PASSED               [ 14%]
tests/test_task_1.py::TestCalculator::test_division PASSED               [ 28%]
tests/test_task_1.py::TestCalculator::test_division_by_zero PASSED       [ 42%]
tests/test_task_1.py::TestCalculator::test_invalid_input PASSED          [ 57%]
tests/test_task_1.py::TestCalculator::test_invalid_operation PASSED      [ 71%]
tests/test_task_1.py::TestCalculator::test_multiplication PASSED         [ 85%]
tests/test_task_1.py::TestCalculator::test_subtraction PASSED            [100%]

============================== 7 passed in 0.10s ==============================
```

Рисунок 2 – Успешные результаты тестирования программы

Задание 8.

Необходимо написать программу, состоящую из семи кнопок, цвета которых соответствуют цветам радуги. При нажатии на ту или иную кнопку в текстовое поле должен вставляться код цвета, а в метку – название цвета. Коды цветов в шестнадцатеричной кодировке: #ff0000 – красный, #ff7d00 – оранжевый, #ffff00 – желтый, #00ff00 – зеленый, #007dff – голубой, #0000ff – синий, #7d00ff – фиолетовый.

Код программы, реализующей интерфейс с цветными кнопками:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Напишите программу, состоящую из семи кнопок, цвета которых соответствуют
# цветам радуги. При нажатии на ту или иную кнопку в текстовое поле должен
```

```python
# вставляться код цвета, а в метку – название цвета. Коды цветов в
шестнадцатеричной
# кодировке: #ff0000 – красный, #ff7d00 – оранжевый, #ffff00 – желтый,
# 00ff00 – зеленый, #007dff – голубой, #0000ff – синий, #7d00ff – фиолетовый.

import tkinter as tk
from typing import Dict


class ColorManager:
    """
    Класс для управления цветовыми данными.
    """

    def __init__(self, colors: Dict[str, str]) -> None:
        """
        Инициализирует ColorManager, принимая на вход словарь соответствия:
        название цвета -> шестнадцатеричный код.

        :param colors: Словарь кодов цветов.
        """

        self.colors = colors

    def get_color_code(self, color_name: str) -> str:
        """
        Возвращает код цвета по его названию. Если цвет не найден,
        возвращает строку "Неизвестный цвет".

        :param color_name: Название цвета.
        :return: Строка с шестнадцатеричным кодом цвета.
        """

        return self.colors.get(color_name, "Неизвестный цвет")


class ColorApp:
    """
    Класс, представляющий графическое приложение для отображения цветов
радуги.
    """

    def __init__(self, root: tk.Tk, color_manager: ColorManager) -> None:
        """
        Инициализирует класс ColorApp.

        :param root: Главное окно приложения.
        :param color_manager: Экземпляр класса ColorManager.
        """

        self.root: tk.Tk = root
        self.color_manager: ColorManager = color_manager
        self.create_widgets()

    def create_widgets(self) -> None:
        """
        Создаёт и размещает на форме все виджеты: метку для кода цвета,
        текстовое поле для названия цвета и кнопки, соответствующие цветам
радуги.
        """

        # Метка для отображения кода цвета
        self.color_label: tk.Label = tk.Label(self.root, text="",
font=("Arial", 14))
```

```
        self.color_label.grid(row=0, column=0, padx=5, pady=5)

        # Текстовое поле для отображения названия цвета
        self.color_entry: tk.Entry = tk.Entry(self.root, width=20,
font=("Arial", 14))
        self.color_entry.grid(row=1, column=0, padx=5, pady=5)

        # Создание кнопок для каждого цвета из словаря ColorManager
        for idx, (color_name, color_code) in
enumerate(self.color_manager.colors.items()):
            button: tk.Button = tk.Button(
                self.root,
                text=color_name,    # Текст на кнопке: название цвета
                bg=color_code,      # Цвет фона кнопки: шестнадцатеричный код
                width=20,
                command=lambda code=color_code, name=color_name:
self.set_color(code, name)
            )
            # Размещение кнопки ниже предыдущих виджетов
            button.grid(row=2 + idx, column=0, padx=5, pady=5)

    def set_color(self, color_code: str, color_name: str) -> None:
        """
        Устанавливает название цвета в текстовое поле и код цвета в метку.

        :param color_code: Шестнадцатеричный код цвета (например, "#ff0000").
        :param color_name: Название цвета (например, "Красный").
        """
        self.color_entry.delete(0, tk.END)       # Очищаем текстовое поле
        self.color_entry.insert(0, color_name)   # Вставляем название цвета в
текстовое поле
        self.color_label.config(text=color_code)  # Обновляем метку,
показывая код цвета


def main() -> None:
    """
    Основная функция приложения. Создаёт окно, инициализирует ColorManager и
ColorApp,
    а затем запускает главный цикл обработки событий Tkinter.
    """
    # Словарь соответствий цвета радуги и их hex-кодов
    colors: Dict[str, str] = {
        "Красный":   "#ff0000",
        "Оранжевый": "#ff7d00",
        "Желтый":    "#ffff00",
        "Зеленый":   "#00ff00",
        "Голубой":   "#007dff",
        "Синий":     "#0000ff",
        "Фиолетовый": "#7d00ff"
    }

    root: tk.Tk = tk.Tk()
    root.title("Цвета радуги")

    # Создаём экземпляр ColorManager, который управляет данными о цветах
    color_manager: ColorManager = ColorManager(colors)

    # Создаём и запускаем приложение
    app: ColorApp = ColorApp(root, color_manager)
    root.mainloop()
```

```
if __name__ == "__main__":
    main()
```

Результаты работы программы, интерфейс для выбора цвета и вывод его кодировки (рис. 3).

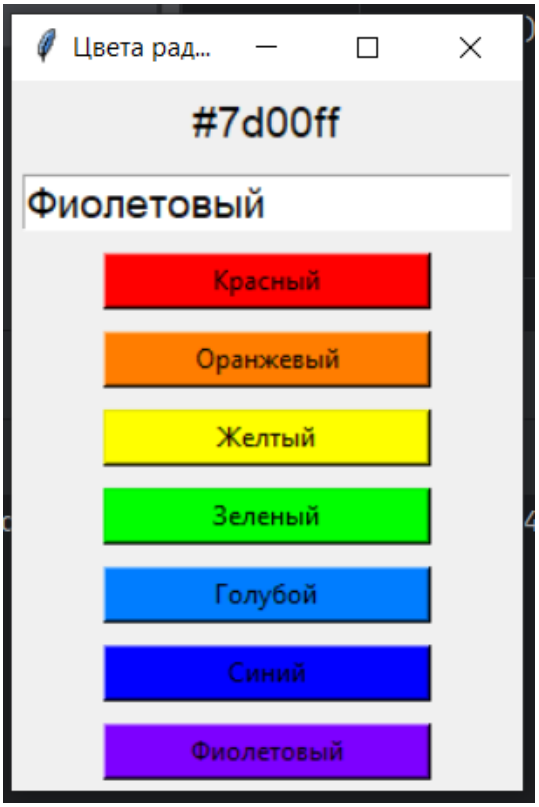


Рисунок 3 – Результаты работы программы задания 8

Для тестирования программы вывода кодировок цветов был создан unit-тесть тестирующие некоторые методы на корректность данных. Результаты тестирования программы задания 8 (рис. 4).

```
tests/test_task_2.py::TestColorManager::test_get_invalid_color_code PASSED
tests/test_task_2.py::TestColorManager::test_get_valid_color_code PASSED

============================================================================== 9 passed in 0.11s ====
PS C:\Users\Andrey\Desktop\ООП\Лабораторная_работа_7\Object-Oriented_Programming_laboratory_work_7>
```

Рисунок 4 – Успешные результаты тестирования задания 8

Задание 9.

Необходимо переписать программу задания 8 так, чтобы кнопки цветов располагались горизонтально.

Код программы, реализующей данный интерфейс:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```python
# Необходимо программу из пункта 8 так, чтобы интерфейс
# выглядел чуть другим образом, а именно, чтобы кнопки
# располагались горизонтально

import tkinter as tk
from typing import Dict


class ColorManager:
    """
    Класс для управления набором цветов.
    Хранит соответствия названий цветов их шестнадцатеричным кодам.
    """

    def __init__(self, colors: Dict[str, str]) -> None:
        """
        Инициализирует ColorManager словарём, в котором ключ — это название
цвета,
        а значение — его шестнадцатеричный код.

        :param colors: Словарь вида кодировок цветов.
        """

        self.colors = colors

    def get_color_code(self, color_name: str) -> str:
        """
        Возвращает шестнадцатеричный код цвета по его названию.

        :param color_name: Название цвета.
        :return: Шестнадцатеричный код цвета или #ffffff, если цвет не
найден.
        """

        return self.colors.get(color_name, "#ffffff")


class ColorApp:
    """
    Класс приложения, позволяющего выбрать цвет из набора и отобразить
название
    и шестнадцатеричный код выбранного цвета.
    """

    def __init__(self, root: tk.Tk, color_manager: ColorManager) -> None:
        """
        Инициализирует приложение, создавая необходимые элементы интерфейса
        и связывая их с обработчиками.

        :param root: Главное окно приложения, экземпляр tk.Tk.
        :param color_manager: Экземпляр ColorManager.
        """

        self.root: tk.Tk = root
        self.color_manager: ColorManager = color_manager
        self.create_widgets()

    def create_widgets(self) -> None:
        """
        Создаёт и размещает элементы интерфейса.
        """

        # Метка, отображающая название текущего цвета
```

```
        self.label: tk.Label = tk.Label(self.root, text="желтый", width=20,
font=("Arial", 14))
        self.label.pack(pady=5)

        # Текстовое поле для отображения кода цвета
        self.color_code_entry: tk.Entry = tk.Entry(self.root, width=10,
justify="center", font=("Arial", 14))
        self.color_code_entry.insert(0,
self.color_manager.get_color_code("желтый"))
        self.color_code_entry.pack(pady=5)

        # Создаём горизонтальные кнопки для каждого цвета из словаря
        for color_name, color_code in self.color_manager.colors.items():
            btn: tk.Button = tk.Button(
                self.root,
                bg=color_code,
                width=4,
                height=2,
                command=lambda c=color_name: self.update_color(c)
            )
            # Размещаем кнопки слева направо в одной строке
            btn.pack(side=tk.LEFT, padx=2, pady=10)

    def update_color(self, selected_color: str) -> None:
        """
        Обработчик нажатия кнопки, обновляет текстовое поле и метку
        в соответствии с выбранным цветом.

        :param selected_color: Название выбранного цвета.
        """

        # Получаем код выбранного цвета из ColorManager
        color_code = self.color_manager.get_color_code(selected_color)

        # Обновляем текстовое поле кодом цвета
        self.color_code_entry.delete(0, tk.END)
        self.color_code_entry.insert(0, color_code)

        # Обновляем метку названием цвета
        self.label.config(text=selected_color)


def main() -> None:
    """
    Основная функция программы.
    """

    # Словарь с набором цветов (название_цвета -> шестнадцатеричный_код)
    colors: Dict[str, str] = {
        "красный": "#ff0000",
        "оранжевый": "#ff7f00",
        "желтый": "#ffff00",
        "зеленый": "#00ff00",
        "голубой": "#00ffff",
        "синий": "#0000ff",
        "фиолетовый": "#7f00ff"
    }

    root: tk.Tk = tk.Tk()
    root.title("Цветовая палитра")

    # Создаём экземпляр ColorManager
    color_manager: ColorManager = ColorManager(colors)
```

```
    # Создаём экземпляр приложения и запускаем цикл обработки событий
    app: ColorApp = ColorApp(root, color_manager)
    root.mainloop()


if __name__ == "__main__":
    main()
```

Результаты работы программы, интерфейс вывода информации о кодировке цвета, в котором кнопки цветов располагаются горизонтально (рис. 5).

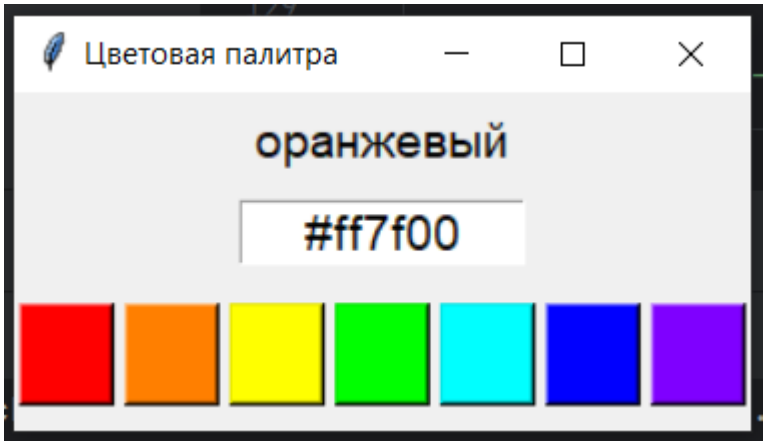


Рисунок 5 – Результаты работы программы задания 9

Для тестирования некоторых методов программы задания 9 был написан похожий на предыдущий unit-тест. Результаты тестирования программы задания 9 (рис. 6).

```
tests/test_task_3.py::TestColorManager::test_get_invalid_color_code PASSED
tests/test_task_3.py::TestColorManager::test_get_valid_color_code PASSED

============================================================================== 11 passed in 0.06s =
PS C:\Users\Andrey\Desktop\ООП\Лабораторная_работа_7\Object-Oriented_Programming_laboratory_work_7>
```

Рисунок 6 – Успешные результаты тестирования программы задания 9

Задание 10.

Необходимо написать программу, состоящую из однострочного и многострочного текстовых полей и двух кнопок «Открыть» и «Сохранить». При клике на первую должен открываться на чтение файл, чье имя указано в поле класса Entry, а содержимое файла должно загружаться в поле типа Text.

При клике на вторую кнопку текст, введенный пользователем в экземпляр Text, должен сохраняться в файле под именем, которое пользователь указал в однострочном текстовом поле.

Файлы будут читаться и записываться в том же каталоге, что и файл скрипта, если указывать имена файлов без адреса.

Код программы задания 10:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Необходимо написать программу, состоящую из однострочного и многострочного
# текстовых полей и двух кнопок "Открыть" и "Сохранить". При клике на первую
# должен открываться на чтение файл, чье имя указано в поле класса Entry, а
# содержимое файла должно загружаться в поле типа Text. При клике на вторую
# кнопку текст, введенный пользователем в экземпляр Text, должен сохраняться
# в файле под именем, которое пользователь указал в однострочном текстовом
поле.
# Файлы будут читаться и записываться в том же каталоге, что и файл скрипта,
# если указывать имена файлов без адреса. Для выполнения практической работы
# вам понадобится функция open языка Python и методы файловых объектов чтения
# и записи.

import tkinter as tk
from tkinter import messagebox
from typing import Optional


class FileManager:
    """
    Класс для работы с файлами: открытие, сохранение, чтение.
    """

    def open_file(self, filename: str) -> Optional[str]:
        """
        Открывает файл с именем filename и возвращает содержимое файла в виде
строки.

        :param filename: Путь к файлу, который нужно открыть.
        :return: Содержимое файла или None, если файл не удалось открыть.
        """

        try:
            with open(filename, 'r', encoding='utf-8') as file:
                return file.read()
        except FileNotFoundError:
            messagebox.showerror("Ошибка", f"Файл {filename} не найден!")
        except Exception as e:
            messagebox.showerror("Ошибка", f"Не удалось открыть файл:
{str(e)}")
        return None

    def save_file(self, filename: str, content: str) -> None:
        """
        Сохраняет переданное содержимое content в файл с именем filename.

        :param filename: Путь к файлу, в который требуется сохранить данные.
        :param content: Строка с содержимым, которое нужно записать в файл.
        :return: None
```

```
        """

        try:
            with open(filename, 'w', encoding='utf-8') as file:
                file.write(content)
            messagebox.showinfo("Успех", f"Файл {filename} успешно
сохранён!")
        except Exception as e:
            messagebox.showerror("Ошибка", f"Не удалось сохранить файл:
{str(e)}")


class FileEditorApp:
    """
    Класс для создания интерфейса простого текстового редактора, который
    позволяет открывать и сохранять файлы.
    """

    def __init__(self, root: tk.Tk, file_manager: FileManager) -> None:
        """
        Инициализирует приложение, создавая необходимые виджеты и
        связывая их с методами класса FileEditorApp.

        :param root: Главное окно приложения, экземпляр tk.Tk.
        :param file_manager: Экземпляр класса FileManager.
        """

        self.root: tk.Tk = root
        self.file_manager: FileManager = file_manager
        self.create_widgets()

    def create_widgets(self) -> None:
        """
        Создаёт и располагает на форме все необходимые элементы интерфейса.
        """

        # Поле ввода имени файла
        self.filename_entry: tk.Entry = tk.Entry(self.root, width=40,
font=("Arial", 14))
        self.filename_entry.grid(row=0, column=0, padx=10, pady=10)

        # Кнопка "Открыть"
        open_button: tk.Button = tk.Button(
            self.root, text="Открыть", width=15, font=("Arial", 14),
command=self.open_file
        )
        open_button.grid(row=0, column=1, padx=10, pady=10)

        # Кнопка "Сохранить"
        save_button: tk.Button = tk.Button(
            self.root, text="Сохранить", width=15, font=("Arial", 14),
command=self.save_file
        )
        save_button.grid(row=0, column=2, padx=10, pady=10)

        # Фрейм для текстового поля и полосы прокрутки
        frame: tk.Frame = tk.Frame(self.root)
        frame.grid(row=1, column=0, columnspan=3, padx=10, pady=10)

        # Многострочное текстовое поле
        self.text_field: tk.Text = tk.Text(frame, width=60, height=15,
font=("Arial", 14))
        self.text_field.grid(row=0, column=0, padx=5, pady=5)
```

```
        # Полоса прокрутки для текстового поля
        scrollbar: tk.Scrollbar = tk.Scrollbar(frame,
command=self.text_field.yview)
        scrollbar.grid(row=0, column=1, sticky="ns", pady=5)

        # Связываем текстовое поле с полосой прокрутки
        self.text_field.config(yscrollcommand=scrollbar.set)

    def open_file(self) -> None:
        """
        Получает из поля ввода имя файла, открывает его с помощью
FileManager,
        и, в случае успеха, загружает содержимое в текстовое поле.
        """

        filename = self.filename_entry.get()
        content = self.file_manager.open_file(filename)
        if content is not None:
            self.text_field.delete(1.0, tk.END)
            self.text_field.insert(tk.END, content)

    def save_file(self) -> None:
        """
        Получает из поля ввода имя файла и сохраняет в него текущее
содержимое
        многострочного текстового поля (self.text_field) с помощью
FileManager.
        """

        filename = self.filename_entry.get()
        content = self.text_field.get(1.0, tk.END)
        self.file_manager.save_file(filename, content)


def main() -> None:
    """
    Основная функция, создаёт главное окно, инициализирует FileManager и
FileEditorApp,
    а затем запускает главный цикл обработки событий Tkinter.
    """

    root: tk.Tk = tk.Tk()
    root.title("Редактор файлов")

    # Создаём экземпляр FileManager
    file_manager: FileManager = FileManager()

    # Создаём и запускаем приложение
    app: FileEditorApp = FileEditorApp(root, file_manager)
    root.mainloop()


if __name__ == "__main__":
    main()
```

Для примера в данной программе был открыт файл в той же директории, что и модуль программы, а именно файл задания 7 (рис. 7).

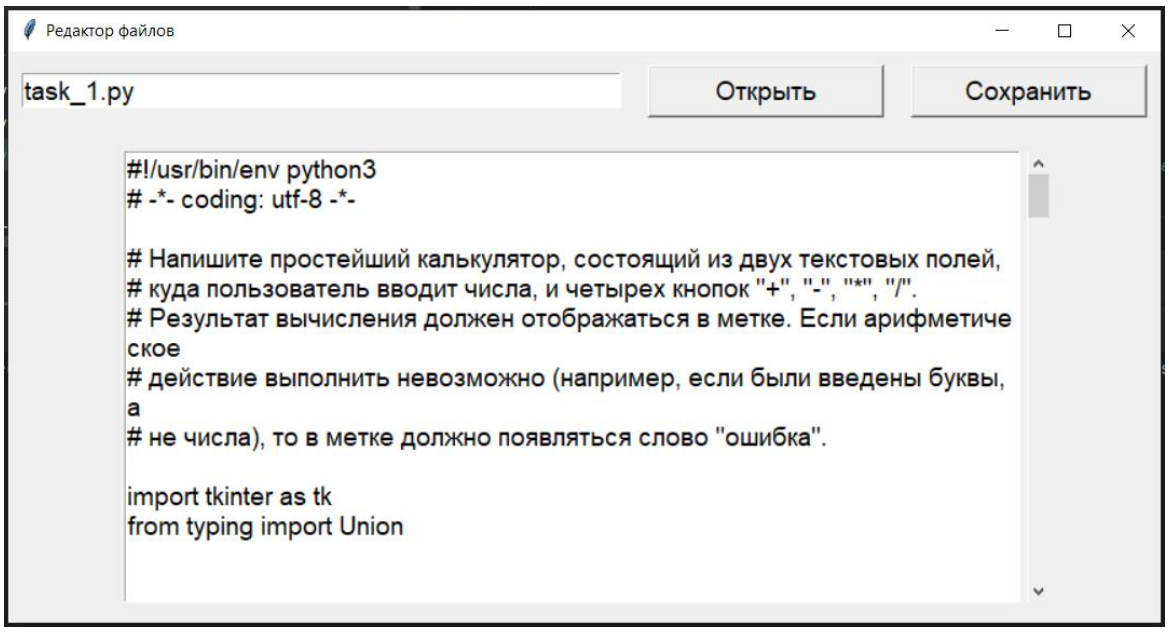


Рисунок 7 – Результаты работы программы задания 10

Для тестирования пользовательского ввода пути в программе задания 10 был написан unit-тест, предусматривающий различные варианты ввода пути пользователем (рис. 8).

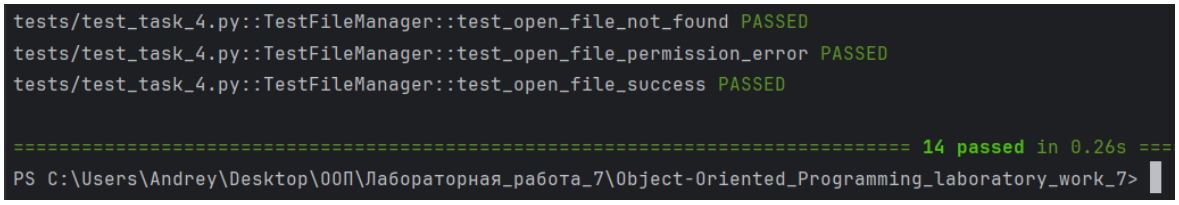


Рисунок 8 – Успешные результаты тестирования программы задания 10

Задание 11.

Виджеты «Radiobatton» и «Checkbutton» поддерживают большинство свойств оформления внешнего вида, которые есть у других элементов графического интерфейса. При этом у «Radiobutton» есть особое свойство «indicatoron». По умолчанию он равен единице, в этом случае радиокнопка выглядит как нормальная радиокнопка. Однако если присвоить этой опции ноль, то виджет «Radiobutton» становится похожим на обычную кнопку по внешнему виду. Но не по смыслу.

Необходимо написать программу, в которой имеется несколько объединенных в группу радиокнопок, индикатор которых выключен (indicatoron = 0). Если какая-нибудь кнопка включается, то в метке должна

отображаться соответствующая ей информация. Обычных кнопок в окне быть не должно.

Код программы задания 11:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Виджеты Radiobatton и Checkbutton поддерживают большинство свойств
# оформления внешнего вида, которые есть у других элементов графического
# интерфейса. При этом у Radiobutton есть особое свойство indicatoron.
# По-умолчанию он равен единице, в этом случае радиокнопка выглядит как
# нормальная радиокнопка. Однако если присвоить этой опции ноль, то виджет
# Radiobutton становится похожим на обычную кнопку по внешнему виду.
# Но не по смыслу.
# Напишите программу, в которой имеется несколько объединенных в группу
# радиокнопок, индикатор которых выключен ( indicatoron=0 ). Если какая-
нибудь
# кнопка включается, то в метке должна отображаться соответствующая ей
информация.
# Обычных кнопок в окне быть не должно.

import tkinter as tk
from typing import Dict


class ContactApp:
    """
    Класс приложения, демонстрирующего работу радиокнопок, внешне
    похожих на обычные кнопки, но сохранивших поведение Radiobutton.
    """

    def __init__(self, root: tk.Tk, contacts: Dict[str, str]) -> None:
        """
        Инициализирует приложение, принимая на вход основное окно и
        словарь с контактными данными.

        :param root: Главное окно приложения.
        :param contacts: Словарь контактов.
        """

        self.contacts: Dict[str, str] = contacts
        # StringVar для хранения выбранного контакта
        self.selected_contact: tk.StringVar = tk.StringVar(value="")

        # Создаём и размещаем виджеты интерфейса
        self.create_widgets(root)

    def create_widgets(self, root: tk.Tk) -> None:
        """
        Создаёт основные элементы графического интерфейса.
        Фрейм с радиокнопками и метку для отображения информации.

        :param root: Главное окно приложения.
        """

        # Фрейм для радиокнопок
        frame: tk.Frame = tk.Frame(root)
        frame.pack(side=tk.LEFT, padx=10, pady=10)

        # Создание «кнопок»-радиокнопок с отключённым индикатором
        for name in self.contacts.keys():
```

```
            rb: tk.Radiobutton = tk.Radiobutton(
                frame,
                text=name,                          # Текст, отображаемый на
кнопке
                variable=self.selected_contact,  # Общая переменная для всех
радиокнопок
                value=name,                         # Значение, присваиваемое
этой переменной при выборе
                indicatoron=0,                      # Отключает индикатор, делая
радиокнопку похожей на обычную кнопку
                width=10,
                command=lambda n=name: self.show_info(n)
            )
            rb.pack(pady=5)

        # Метка для отображения информации о выбранном пункте
        self.info_label: tk.Label = tk.Label(
            root,
            text="",
            width=30,
            anchor="w",
            font=("Arial", 14)
        )
        self.info_label.pack(side=tk.LEFT, padx=10, pady=10)

    def show_info(self, name: str) -> None:
        """
        Обработчик переключения радиокнопок. По названию контакта получает
        соответствующую информацию и отображает её в метке.

        :param name: Ключ (название контакта).
        """

        info: str = self.contacts.get(name, "")
        self.info_label.config(text=info)


def main() -> None:
    """
    Основная функция, создаёт окно Tk, инициализирует словарь с контактами
    и запускает приложение ContactApp в главном цикле событий.
    """
    contacts: Dict[str, str] = {
        "Вася": "+7 928 822 76 78",
        "Петя": "+7 933 995 65 90",
        "Маша": "+7 926 906 81 11"
    }

    # Создаём главное окно
    root: tk.Tk = tk.Tk()
    root.title("Контакты")

    # Запускаем приложение
    app: ContactApp = ContactApp(root, contacts)

    # Главный цикл обработки событий
    root.mainloop()


if __name__ == "__main__":
    main()
```

Результаты работы программы, вывод номеров телефонов людей в списке контактов по нажатию соответствующих кнопок (рис. 9).

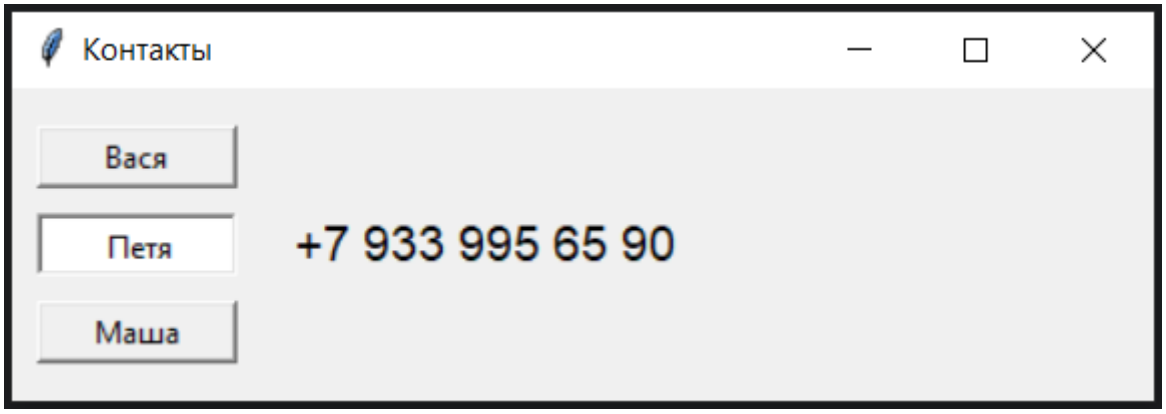


Рисунок 9 – Результаты работы программы задания 11

Для тестирования программы задания 11 был создан unit-тест для проверки работоспособности программы, проверки соответствия кнопки и ее значения и начального состояния приложения (рис. 10).

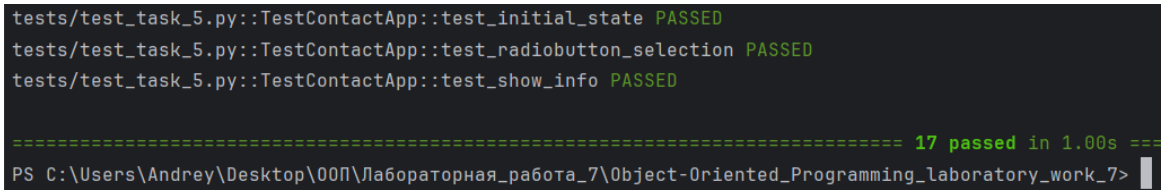


Рисунок 10 – Успешные результаты тестирования программы задания 11

Ссылка не репозиторий данной лабораторной работы:

https://github.com/AndreyPust/Object-Oriented_Programming_laboratory_work_7.git

Ответы на контрольные вопросы:

1. Какие существуют средства в стандартной библиотеке Python для построения графического интерфейса пользователя?

В стандартной библиотеке Python для построения графического интерфейса пользователя (GUI) существует несколько инструментов:

– Tkinter: наиболее популярный и основной инструмент для создания GUI.

– Pygame: используется для разработки игр, но также подходит для создания графического интерфейса.

– curses: для создания текстовых интерфейсов в терминале (не графический, но часто используется для CLI).

– IDLE: это встроенная среда разработки, использующая Tkinter для создания интерфейса.

– Turtle: используется для обучения программированию через рисование и графику.

2. Что такое Tkinter?

Tkinter – это стандартная библиотека для создания графических интерфейсов пользователя в Python. Tkinter является оберткой вокруг библиотеки Tk, которая представляет собой набор инструментов для создания оконных приложений. Tkinter предоставляет доступ к множеству виджетов (кнопки, метки, текстовые поля и т.д.), которые можно использовать для создания GUI.

3. Какие требуется выполнить шаги для построения графического интерфейса с помощью Tkinter?

Основные шаги для создания графического интерфейса с использованием Tkinter:

– импортировать Tkinter;

– создать основное окно приложения (root = tk.Tk());

– добавить виджеты (например, кнопки, метки, текстовые поля) в окно;

– разместить виджеты в окне (с помощью методов pack(), grid(), place());

– запустить главный цикл обработки событий (root.mainloop()).

4. Что такое цикл обработки событий?

Цикл обработки событий (или главный цикл) — это механизм, который позволяет программе реагировать на действия пользователя, такие как нажатия кнопок, движение мыши, изменения в полях ввода и т.д. Цикл обработки событий постоянно ожидает событий и вызывает соответствующие обработчики (функции) для этих событий.

Главный цикл запускается методом «root.mainloop()» и продолжается до тех пор, пока окно не будет закрыто.

5. Каково назначение экземпляра класса Tk при построении графического интерфейса с помощью Tkinter?

Экземпляр класса Tk является главным окном вашего приложения. Это окно, которое будет содержать все виджеты, с которыми взаимодействует пользователь. Он инициализирует внутренние механизмы Tkinter, включая главный цикл событий, и управляет отображением всех элементов интерфейса.

6. Для чего предназначены виджеты Button, Label, Entry и Text?

Button: Кнопка, которая реагирует на действия пользователя (например, клики). С помощью неё можно вызвать функцию или выполнить команду.

Label: Метка (ярлык), отображающая текст или изображение, используется для вывода информации.

Entry: Однострочное текстовое поле, в котором пользователь может ввести данные.

Text: Многострочное текстовое поле, в котором можно редактировать большой объем текста (с возможностью прокрутки).

7. Каково назначение метода pack() при построении графического интерфейса пользователя?

Метод pack() используется для размещения виджетов в окне. Он автоматически размещает виджеты в контейнере (например, в окне или фрейме) по определенным правилам. Виджеты могут быть размещены вертикально или горизонтально, в зависимости от параметров.

8. Как осуществляется управление размещением виджетов с помощью метода pack()?

Метод pack() имеет несколько параметров для управления размещением виджетов:

– side: определяет сторону контейнера (например, TOP, BOTTOM, LEFT, RIGHT).

– fill: устанавливает, как виджет должен растягиваться по оси (например, X, Y, BOTH).

– expand: если установлено в True, виджет будет расширяться, чтобы занять доступное пространство.

9. Как осуществляется управление полосами прокрутки в виджете Text?

Для управления полосами прокрутки в виджете Text используется виджет «Scrollbar». Полоса прокрутки связывается с виджетом Text через параметр «yscrollcommand».

10. Для чего нужны тэги при работе с виджетом Text?

Тэги в Text используются для выделения или изменения атрибутов текста (например, цвет, стиль, шрифт). С помощью тэгов можно управлять стилями текста в определенных областях виджета Text.

11. Как осуществляется вставка виджетов в текстовое поле?

Вставка виджетов (например, кнопок или меток) в текстовое поле осуществляется с помощью метода «window_create()». С помощью этого метода можно вставить объект Tkinter (например, виджет) в определенную позицию в тексте.

12. Для чего предназначены виджеты Radiobutton и Checkbutton?

Radiobutton: предназначен для выбора одного из нескольких вариантов. Все радиокнопки в одной группе взаимодействуют друг с другом, и только одна из них может быть выбрана одновременно.

Checkbutton: предназначен для выбора нескольких вариантов (можно поставить несколько флажков одновременно). Каждый флажок, может быть, как выбран, так и не выбран.

13. Что такое переменные Tkinter и для чего они нужны?

Переменные Tkinter (например, StringVar, IntVar, BooleanVar) — это особые объекты, которые используются для связи данных между виджетами и программой. Они позволяют отслеживать изменения в виджетах и обновлять их состояние.

14. Как осуществляется связь переменных Tkinter с виджетами Radiobutton и Checkbutton?

Для связи переменных с виджетами Radiobutton и Checkbutton используется параметр variable. Переменная хранит текущее состояние виджета (выбран/не выбран или значение радиокнопки).

Вывод: в ходе выполнения лабораторной работы были приобретены навыки по работе с классами данных при написании программ с помощью языка программирования Python версии 3.x.