**ИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное бюджетное образовательное учреждение   
высшего образования

**«Сибирский государственный университет науки и технологий   
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

Языки программирования

|  |
| --- |
| Наследование |

Руководитель А.В. Проскурин

подпись, дата инициалы, фамилия

Обучающийся БПИ22-02, 221219040 К.В. Трифонов

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2023 г.

# ЦЕЛЬ РАБОТЫ

Получение практических навыков разработки и отладки программ c использованием механизма наследования.

# порядок выполнения работы

1. Ознакомиться с общей постановкой задачи.

2. Ознакомится с вариантом задания – соответствует вашему номеру в списке группы (при нехватке заданий вариант задания вычисляется как номер\_в\_списке\_группы - количество\_заданий).

3. Разработать классы согласно варианту задания.

4. Подготовить отчет по лабораторной работе. Отчет должен включать в себя:

• титульный лист;

• цель лабораторной работы;

• постановку задачи;

• текст программы с комментариями;

• демонстрацию работы программы (Снимки экрана при выполнении действий программы с описанием).

• краткие ответы на контрольные вопросы;

• выводы по лабораторной работе.

5. Защитить лабораторную работу перед преподавателем.

# Задания

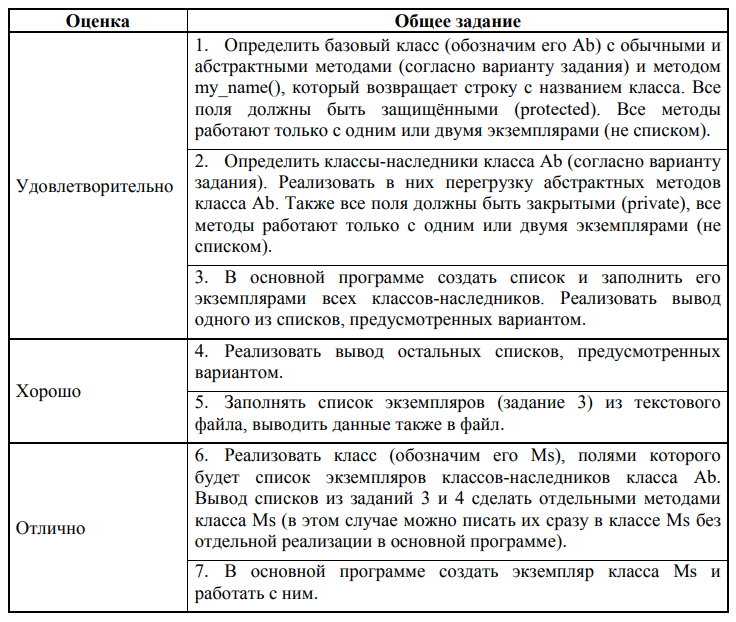
Необходимо разработать программу на языке Python, реализующую задачу ранее решенную на языке С++ в лабораторной работе №2.

Необходимо разработать программу, в которой будет реализовано несколько классов согласно варианту задания.

В реализуемой программе необходимо предусмотреть возможность демонстрации работы методов созданных классов.

Организацию взаимодействия с пользователем реализовать на основе меню.

В зависимости от оценки, на которую вы претендуете, необходимо выполнить следующие задания (для каждой следующей оценки нужно выполнить **ВСЕ** предыдущие задания, если обратное не сказано явно):



**Вариант №21.** Создать класс Line (Линия) с чистым виртуальным методом нахождения длины линии. На его основе создать классы:

- **ColoredLine** (Цветная линия) с полем «цвет»;

- **PolyLine** (Полилиния) с массивом длин участков ломаной линии (кроме первого). Для массива указателей на объекты этих классов предусмотреть возможность:

- вывода характеристик объектов;

- вывода объектов, длина которых находится в заданном диапазоне;

- вывода массива, отсортированного по длине линий.

Создать класс Picture, содержащий массив указателей на объекты этих классов. Написать демонстрационную программу, в которой будут использоваться все методы классов.

# ХОД РАБОТЫ

**Текст программы:**

**Содержимое файла lines.py:**

from abc import ABC, abstractmethod

*class* Line(ABC):

*def* \_\_init\_\_(*self*, *length* = 0):

*self*.\_length = *float*(length)

*#@abstractmethod*

*def* Lenght(*self*):

        return *self*.\_length

*def* my\_name(*self*):

        return *str*(*type*(*self*).\_\_name\_\_)

*def* from\_string(*self*,*string*):

*self*.\_length = *float*(string)

*def* to\_string(*self*):

        return *f*"{*self*.\_length}"

    Определение базового класса **Line** и методов возврата длины, имени класса, строки с данными и заполнения полей из строки (метод **Length()** предполагается как чистый по заданию (а класс абстрактный), однако в лаб №2-21 после согласования он был полностью определён).

*class* ColoredLine(*Line*):

*def* \_\_init\_\_(*self*, *length* = 0, *color* = "0 0 0"):

*super*().\_\_init\_\_(length)

*self*.\_color = color

*def* get\_color(*self*):

        return *self*.\_color

*def* from\_string(*self*,*string*):

        length, color = map(*str*, string.strip().split("   "))

*self*.\_length = *float*(length)

*self*.\_color = color

*def* to\_string(*self*):

        return *f*"{*self*.\_length}   {*self*.\_color}"

Определение производного класса **ColoredLine** с дополнительным текстовым полем цвета и методом возврата поля цвета. Были перегружены методы ввода/вывода из строки (Длина из базового класса отделяется в строке от остальных данных тройным пробелом “\_\_\_“)

*class* PolyLine(*Line*):

*def* \_\_init\_\_(*self*, *lengths* = [0,0]):

*super*().\_\_init\_\_(*float*(lengths[0]))

*self*.\_segment\_lengths = map(*float*,lengths[1:])

*def* Lenght(*self*):

        return *self*.\_length + sum(*self*.\_segment\_lengths)

*def* from\_string(*self*,*string*):

        length, ls = string.strip().split("   ")

*self*.\_length = *float*(length)

*self*.\_segment\_lengths = *list*(map(*float*,ls.split()))

*def* to\_string(*self*):

        return *f*"{*self*.\_length}   {' '.join(map(*str*,*self*.\_segment\_lengths))}"

Определение производного класса **PolyLine** с дополнительным полем списка участков кривой линии. Были перегружены методы определения длины (теперь он возвращает сумму длины поля из базового класса и всех участков кривой из списка), ввода/вывода из строки (Длина из базового класса отделяется в строке от остальных данных тройным пробелом “\_\_\_“).

**Определение класса Picture, содержащего список линий:**

*class* Picture:

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

*self*.\_objects = []

*def* add\_from\_console(*self*):

        sw = *int*(input("""Выберите тип линии:

1) Line

2) ColoredLine

3) PolyLine

"""))

        match sw:

            case 1:

                obj = Line()

                obj.from\_string(input("Введите длину: "))

*self*.\_objects.append(obj)

            case 2:

                obj = ColoredLine()

                l = input("Введите длину: ")

                col = input("Введите цвет: ")

                obj.from\_string(*f*"{l}   {col}")

*self*.\_objects.append(obj)

            case 3:

                obj = PolyLine()

                l = input("Введите длину: ")

                ls = input("Введите длины участков через пробел: ")

                obj.from\_string(*f*"{l}   {ls}")

*self*.\_objects.append(obj)

        Метод add\_from\_console позволяет заполнить 1 объект и сразу же добавить его в список объектов. Пользователь выбирает из меню нужный тип линии.

*def* add(*self*, *obj*):

*self*.\_objects.append(obj)

Метод добавления объекта к списку.

*def* get(*self*):

        print("Содержимое picture:")

        for obj in *self*.\_objects:

            print(*f*"{*type*(obj).\_\_name\_\_}:\nПараметры: {obj.to\_string()}")

Вывод в консоль списка объектов (название класса и его полей).

*def* get\_in\_range(*self*, *min\_length*, *max\_length*):

        for obj in *self*.\_objects:

            if *float*(min\_length) <= obj.Lenght() <= *float*(max\_length):

                print(*f*"{*type*(obj).\_\_name\_\_}:\nПараметры: {obj.to\_string()}")

Вывод в консоль списка линий в диапазоне длины. Сравнение осуществляется при помощи метода Length().

*def* get\_sorted\_by\_length(*self*):

        sorted\_objects = sorted(*self*.\_objects, *key*=*lambda* *x*: x.Lenght())

        for obj in sorted\_objects:

            print(*f*"{*type*(obj).\_\_name\_\_}:\nПараметры: {obj.to\_string()}")

    Вывод отсортированной копии списка в консоль.

*def* read(*self*, *filename*):

        with open(filename, 'r') as file:

            for string in file:

                if string == "\n":

                    continue

                else:

                    ty,meta = string.split("|")

                if ty == "Line":

                    obj = Line()

                    obj.from\_string(meta)

                elif ty == "ColoredLine":

                    obj = ColoredLine()

                    obj.from\_string(meta)

                elif ty == "PolyLine":

                    obj = PolyLine()

                    obj.from\_string(meta)

                else:

                    print("Найден неизвестный тип в файле:", ty)

                    continue

*self*.\_objects.append(obj)

Чтение из файла, предполагается, что название класса отделено от его данных разделителем “|”, если будет найден неизвестный тип, то строка будет пропущена. Затем строка после “|” передаётся в метод from\_string одного из объектов.

*def* write(*self*, *filename*):

        with open(filename, 'w') as file:

            for obj in *self*.\_objects:

                file.write(*f*"{obj.my\_name()}|{obj.to\_string()}\n")

Запись в файл, название класса отделяется от его данных при помощи разделителя “|”.

**Содержимое файла main.py:**

from Lines import \*

if \_\_name\_\_ == "\_\_main\_\_":

    p = Picture()

    while True:

        sw = *int*(input("""Главное меню:

1) Добавить объект с консоли

2) Добавить из файла

3) Вывести в консоль

4) Вывести в файл

5) Вывести в диапазоне длины

6) Вывести в порядке возрастания длины

0) <<< Выход

"""))

        match sw:

            case 1:

                p.add\_from\_console()

            case 2:

                p.read("input.txt")

            case 3:

                p.get()

            case 4:

                p.write("output.txt")

            case 5:

                l1, l2 = input("Введите диапазон длины через пробел:").split()

                p.get\_in\_range(l1,l2)

            case 6:

                p.get\_sorted\_by\_length()

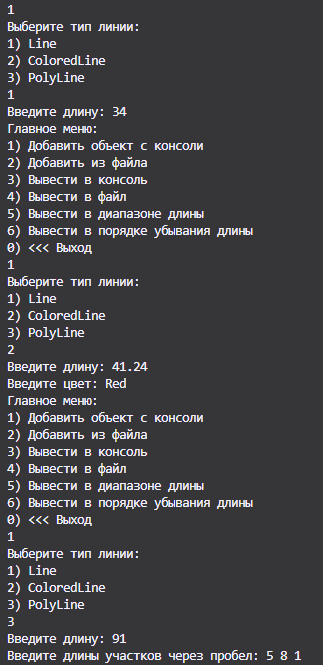
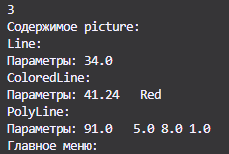
            case 0:

                break

Меню вызова методов класса Picture, для выбора используется оператор match-case. Для чтения используется файл input.txt, для записи – output.txt.

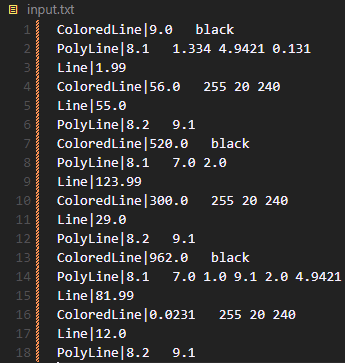
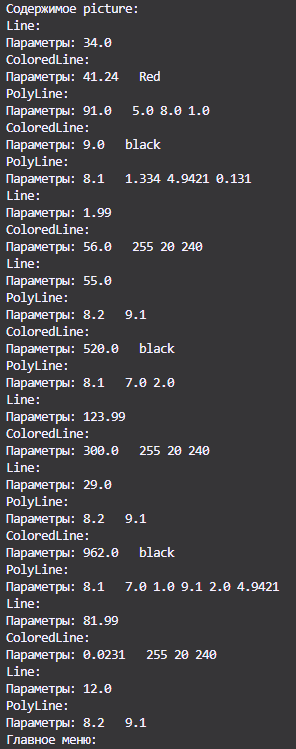
# Тестирование

Заполнение списка объектов 3 различными объектами и вывод их в консоль:

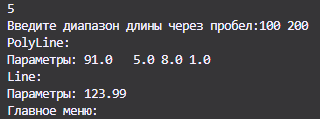
 

Чтение файла input.txt:

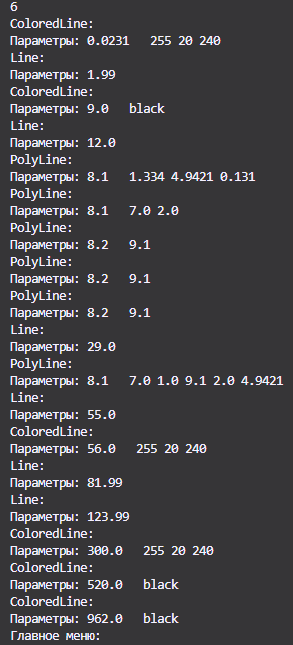
Содержимое файла input.txt и результат обработки файла:

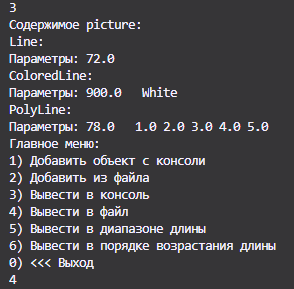
Вывод объектов в диапазоне длины:



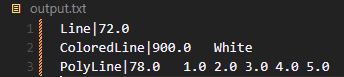
Вывод списка объектов в порядке возрастания их длины.



Вывод в файл списка объектов, состоящего из 3 различных линий:



Содержимое файла output.txt:



# ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ

1. **В чем состоит сущность механизма наследования как принципа ООП?**
   * Наследование в объектно-ориентированном программировании позволяет создавать новые классы на основе существующих, унаследовав их свойства и методы.
2. **Как объявляется производный класс?**
   * Производный класс объявляется с использованием ключевого слова **class**, за которым следует имя производного класса, а в скобках указывается имя базового класса.
3. **Как осуществляется доступ к полям базового класса из производного при наследовании? Какие при этом существуют договорѐнности?**
   * Доступ к полям базового класса из производного осуществляется через обращение к ним через экземпляр производного класса. Согласно принципам инкапсуляции, поля базового класса могут быть закрытыми (private), и их доступ осуществляется через методы.
4. **Как происходит перегрузка методов при наследовании?**
   * Перегрузка методов при наследовании позволяет в производном классе предоставить свою реализацию метода с тем же именем, что и у базового класса.
5. **Каким образом выполняется инициализация наследуемых элементов?**
   * Инициализация наследуемых элементов происходит при вызове конструктора производного класса. Конструктор производного класса может вызывать конструктор базового класса с помощью ключевого слова **super()**.
6. **В каком порядке вызываются конструкторы класса при создании экземпляра производного класса?**
   * При создании экземпляра производного класса вызывается сначала конструктор базового класса, затем конструктор производного класса.
7. **Что такое абстрактный класс?**
   * Абстрактный класс в Python создается с использованием модуля **abc** и содержит хотя бы один абстрактный метод. Экземпляры абстрактного класса не могут быть созданы напрямую.
8. **Что такое множественное наследование?**
   * Множественное наследование позволяет классу наследовать свойства и методы от нескольких базовых классов. Python поддерживает множественное наследование, что означает, что класс может наследовать от нескольких классов.

# ВЫВОД

Наследование способствует повторному использованию кода, структурирует программу и улучшает понимание и её поддерживаемость. При наследовании важными аспектами являются доступ к элементам базового класса, перегрузка методов, инициализация наследуемых элементов, порядок вызова конструкторов, а также возможность создания абстрактных классов и использование множественного наследования в тех случаях, когда это оправдано. Все эти аспекты обеспечивают гибкость и расширяемость кода, что является важным принципом объектно-ориентированного программирования.