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

Институт перспективной инженерии

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №5
дисциплины «Объектно-ориентированное программирование»
Вариант 7

Выполнил:
Мотовилов Вадим Борисович
3 курс, группа ИВТ-б-о-23-2,
09.03.01 «Информатика и вычислительная техника»,
направленность (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем», очная форма обучения

_____
(подпись)

Руководитель практики:
Воронкин Р.А. ктн доцент департамента перспективной инженерии

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2026 г.

Тема: Классы данных в языке Python.

Цель работы: приобретение навыков по работе с классами данных при написании программ с помощью языка программирования Python версии 3.x.

Порядок выполнения работы:

Задание 1.

Создайте датакласс `Player` с полями: имя, амплуа, забитые голы.
Выведите игроков указанного амплуа и отсортируйте их по голам по убыванию.

Код программы:

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

from dataclasses import dataclass, field
from typing import List
from enum import Enum

class Position(Enum):
    """Перечисление для амплуа игроков"""
    GOALKEEPER = "вратарь"
    DEFENDER = "защитник"
    MIDFIELDER = "полузащитник"
    FORWARD = "нападающий"

@dataclass
class Player:
    """Датакласс для представления игрока"""
    name: str
    position: Position
    goals: int = 0

    def __post_init__(self):
        """Проверка корректности данных после инициализации"""
        if self.goals < 0:
            raise ValueError("Количество голов не может быть отрицательным")

@dataclass
class Team:
    """Контейнер для хранения и управления игроками команды"""
    name: str
    players: List[Player] = field(default_factory=list)

    def add_player(self, player: Player) -> None:
        """Добавить игрока в команду"""
        self.players.append(player)

    def add_players(self, *players: Player) -> None:
        """Добавить несколько игроков в команду"""
        for player in players:
```

```
        self.add_player(player)

    def get_players_by_position(self, position: Position) -> List[Player]:
        """Получить игроков указанного амплуа"""
        return [player for player in self.players if player.position == position]

    def get_top_scorers(self, position: Position = None, limit: int = None) -> List[Player]:
        """
        Получить игроков, отсортированных по голам по убыванию

        Args:
            position: фильтр по амплуа (опционально)
            limit: ограничение количества результатов (опционально)
        """
        # Фильтруем по амплуа если указано
        if position:
            filtered_players = self.get_players_by_position(position)
        else:
            filtered_players = self.players.copy()

        # Сортируем по голам по убыванию
        sorted_players = sorted(filtered_players, key=lambda p: p.goals, reverse=True)

        # Ограничиваем количество если указано
        if limit:
            return sorted_players[:limit]
        return sorted_players

    def get_total_goals(self) -> int:
        """Получить общее количество голов команды"""
        return sum(player.goals for player in self.players)

    def get_goals_by_position(self) -> dict:
        """Получить количество голов по амплуа"""
        result = {}
        for position in Position:
            players = self.get_players_by_position(position)
            result[position.value] = sum(p.goals for p in players)
        return result

    def get_best_scorer(self) -> Player:
        """Получить лучшего бомбардира команды"""
        if not self.players:
            raise ValueError("В команде нет игроков")
        return max(self.players, key=lambda p: p.goals)

    def __len__(self) -> int:
        """Количество игроков в команде"""
        return len(self.players)

    def __str__(self) -> str:
        """Строковое представление команды"""
```

```
        return f"Команда '{self.name}' ({len(self)} игроков, голов: {self.get_total_goals()})"

def demonstrate_team_operations():
    """Демонстрация работы с командой и игроками"""
    team = Team("Молния")
    print(f"Создана {team}\n")

    players = [
        Player("Алексей Иванов", Position.FORWARD, 15),
        Player("Игорь Петров", Position.FORWARD, 10),
        Player("Сергей Сидоров", Position.FORWARD, 8),
        Player("Дмитрий Козлов", Position.MIDFIELDER, 5),
        Player("Антон Николаев", Position.MIDFIELDER, 3),
        Player("Владимир Михайлов", Position.MIDFIELDER, 7),
        Player("Артем Федоров", Position.DEFENDER, 2),
        Player("Павел Сергеев", Position.DEFENDER, 1),
        Player("Евгений Васильев", Position.DEFENDER, 0),
        Player("Константин Алексеев", Position.GOALKEEPER, 0),
    ]

    team.add_players(*players)
    print(f"Добавлено игроков: {len(team)}\n")

    print("Все игроки команды:")
    for i, player in enumerate(team.players, 1):
        print(f"  {i}. {player.name} - {player.position.value}, голов: {player.goals}")

    print("\nНападающие (FORWARD):")
    forwards = team.get_players_by_position(Position.FORWARD)
    for i, player in enumerate(forwards, 1):
        print(f"  {i}. {player.name}, голов: {player.goals}")

    print("\nВсе игроки, отсортированные по голам (по убыванию):")
    top_scorers = team.get_top_scorers()
    for i, player in enumerate(top_scorers, 1):
        print(f"  {i}. {player.name} ({player.position.value}) - {player.goals} голов")

    print("\nНападающие, отсортированные по голам (по убыванию):")
    top_forwards = team.get_top_scorers(Position.FORWARD)
    for i, player in enumerate(top_forwards, 1):
        print(f"  {i}. {player.name} - {player.goals} голов")

    print("\nТОП-3 бомбардира команды:")
    top_3 = team.get_top_scorers(limit=3)
    for i, player in enumerate(top_3, 1):
        print(f"  {i}. {player.name} ({player.position.value}) - {player.goals} голов")

    print("\nГолы по амплуа:")
    goals_by_position = team.get_goals_by_position()
    for position, goals in goals_by_position.items():
        print(f"  {position}: {goals} голов")
```

```
    print("\nЛучший бомбардир команды:")
    try:
        best = team.get_best_scorer()
        print(f"   {best.name} ({best.position.value}) - {best.goals} голов")
    except ValueError as e:
        print(f"   Ошибка: {e}")

    print(f"\nОбщая статистика команды:")
    print(f"    Всего игроков: {len(team)}")
    print(f"    Всего голов: {team.get_total_goals()}")
    print(f"    Среднее голов на игрока: {team.get_total_goals() / len(team):.1f}")

    print("\nПолузащитники (MIDFIELDER), отсортированные по голам:")
    midfielders = team.get_top_scorers(Position.MIDFIELDER)
    if midfielders:
        for i, player in enumerate(midfielders, 1):
            print(f"   {i}. {player.name} - {player.goals} голов")
    else:
        print("   Полузащитников нет в команде")

    print("\nСоздание новой пустой команды:")
    empty_team = Team("Новички")
    print(f"   {empty_team}")

    try:
        empty_team.get_best_scorer()
    except ValueError as e:
        print(f"   Попытка получить лучшего бомбардира: {e}")

    print(f"   Нападающие в новой команде:
{len(empty_team.get_players_by_position(Position.FORWARD))} игроков")

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

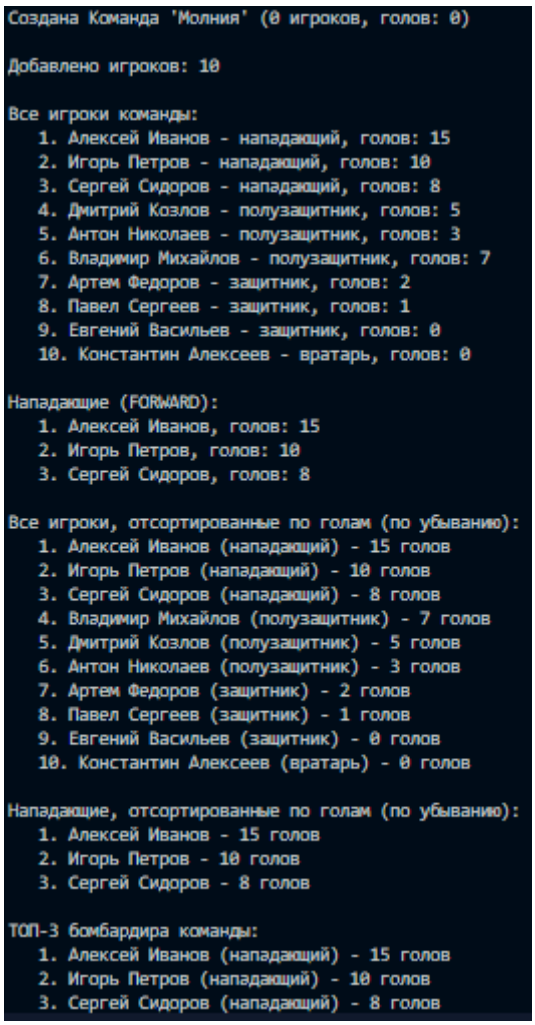

Рисунок 1. Пример работы кода

Задание 2.

Создайте `Range(Generic[T])` с полями `start: T` и `end: T`.
В `__post_init__` проверяйте, что `start < end`.
Используйте `order=True`, чтобы контейнер можно было сравнивать.

Код программы:

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

from dataclasses import dataclass, field
from typing import TypeVar, Generic, List, Optional, Any, Iterator
from functools import total_ordering

# Определяем TypeVar с ограничением сравнения
T = TypeVar("T", bound="Comparable")

class Comparable:
    """Класс-маркер для типов, поддерживающих сравнение"""

    def __lt__(self, other: Any) -> bool:
        if not isinstance(other, type(self)):
```

```
            return NotImplemented
        return self._compare_to(other) < 0

    def _compare_to(self, other: Any) -> int:
        """Метод, который должен быть реализован в наследниках"""
        raise NotImplementedError("Должен быть реализован в наследниках")

@dataclass(order=True, frozen=True)
class Range(Generic[T]):
    """
    Универсальный диапазон значений от start до end (не включая end)

    Атрибуты:
        start: начальное значение диапазона
        end: конечное значение диапазона
    """

    start: T
    end: T

    def __post_init__(self) -> None:
        """Проверка, что start < end"""
        if self.start >= self.end:
            raise ValueError(
                f"start ({self.start}) должен быть меньше end ({self.end})"
            )

    @property
    def length(self) -> Any:
        """Длина диапазона (разность end - start)"""
        try:
            return self.end - self.start
        except TypeError:
            # Для типов, не поддерживающих вычитание
            return None

    def contains(self, value: T) -> bool:
        """Проверяет, содержится ли значение в диапазоне [start, end)"""
        return self.start <= value < self.end

    def overlaps(self, other: "Range[T]") -> bool:
        """Проверяет, пересекается ли диапазон с другим диапазоном"""
        return not (self.end <= other.start or self.start >= other.end)

    def intersection(self, other: "Range[T]") -> Optional["Range[T]"]:
        """Возвращает пересечение двух диапазонов или None если нет пересечения"""
        if not self.overlaps(other):
            return None

        start = max(self.start, other.start)
        end = min(self.end, other.end)
```

```
        return Range(start, end)

    def union(self, other: "Range[T]") -> List["Range[T]"]:
        """Возвращает объединение двух диапазонов"""
        if not self.overlaps(other) and not self.is_adjacent_to(other):
            return [self, other]

        start = min(self.start, other.start)
        end = max(self.end, other.end)

        return [Range(start, end)]

    def is_adjacent_to(self, other: "Range[T]") -> bool:
        """Проверяет, являются ли диапазоны смежными"""
        return self.end == other.start or self.start == other.end

    def split(self, point: T) -> List["Range[T]"]:
        """Разделяет диапазон в указанной точке"""
        if not self.contains(point):
            raise ValueError(f"Точка {point} не находится в диапазоне {self}")

        return [Range(self.start, point), Range(point, self.end)]

    def __contains__(self, value: T) -> bool:
        """Поддержка оператора 'in'"""
        return self.contains(value)

    def __len__(self) -> Optional[int]:
        """Длина диапазона для целочисленных типов"""
        if isinstance(self.start, int) and isinstance(self.end, int):
            return self.end - self.start
        return None

    def __iter__(self) -> Iterator[T]:
        """Итерация по значениям диапазона (только для int)"""
        if isinstance(self.start, int) and isinstance(self.end, int):
            current = self.start
            while current < self.end:
                yield current
                current += 1
        else:
            raise TypeError(
                "Итерация поддерживается только для целочисленных диапазонов"
            )

    def __str__(self) -> str:
        """Строковое представление"""
        return f"[{self.start}, {self.end})"

# Реализация конкретных типов для демонстрации
@dataclass(order=True, frozen=True)
class ComparableInt(Comparable):
```

```
    """Целое число с поддержкой сравнения"""

    value: int

    def _compare_to(self, other: "ComparableInt") -> int:
        return self.value - other.value

    def __add__(self, other: Any) -> "ComparableInt":
        if isinstance(other, ComparableInt):
            return ComparableInt(self.value + other.value)
        elif isinstance(other, int):
            return ComparableInt(self.value + other)
        return NotImplemented

    def __sub__(self, other: Any) -> "ComparableInt":
        if isinstance(other, ComparableInt):
            return ComparableInt(self.value - other.value)
        elif isinstance(other, int):
            return ComparableInt(self.value - other)
        return NotImplemented

    def __str__(self) -> str:
        return str(self.value)

@dataclass(order=True, frozen=True)
class ComparableFloat(Comparable):
    """Вещественное число с поддержкой сравнения"""

    value: float

    def _compare_to(self, other: "ComparableFloat") -> int:
        if self.value < other.value:
            return -1
        elif self.value > other.value:
            return 1
        else:
            return 0

    def __str__(self) -> str:
        return f"{self.value:.2f}"

# Контейнер для работы с коллекцией диапазонов
@dataclass
class RangeContainer(Generic[T]):
    """Контейнер для хранения и работы с множеством диапазонов"""

    name: str
    ranges: List[Range[T]] = field(default_factory=list)

    def add_range(self, range_obj: Range[T]) -> None:
        """Добавить диапазон в контейнер"""
        self.ranges.append(range_obj)
```

```
def add_ranges(self, *ranges: Range[T]) -> None:
    """Добавить несколько диапазонов"""
    for r in ranges:
        self.add_range(r)

def find_containing_range(self, value: T) -> Optional[Range[T]]:
    """Найти диапазон, содержащий указанное значение"""
    for r in self.ranges:
        if value in r:
            return r
    return None

def merge_overlapping_ranges(self) -> "RangeContainer[T]":
    """Объединить пересекающиеся диапазоны"""
    if not self.ranges:
        return RangeContainer(self.name + "_merged", [])

    # Сортируем диапазоны по start
    sorted_ranges = sorted(self.ranges, key=lambda x: x.start)
    merged = [sorted_ranges[0]]

    for current in sorted_ranges[1:]:
        last = merged[-1]

        # Если пересекаются или смежные - объединяем
        if last.overlaps(current) or last.is_adjacent_to(current):
            new_start = min(last.start, current.start)
            new_end = max(last.end, current.end)
            merged[-1] = Range(new_start, new_end)
        else:
            merged.append(current)

    return RangeContainer(self.name + "_merged", merged)

def get_total_coverage(self) -> Optional[Range[T]]:
    """Получить общий охватывающий диапазон"""
    if not self.ranges:
        return None

    starts = [r.start for r in self.ranges]
    ends = [r.end for r in self.ranges]

    return Range(min(starts), max(ends))

def filter_by_value(self, value: T) -> List[Range[T]]:
    """Получить все диапазоны, содержащие указанное значение"""
    return [r for r in self.ranges if value in r]

def get_gaps(self) -> List[Range[T]]:
    """Получить все промежутки между диапазонами"""
    if len(self.ranges) < 2:
```

```
            return []

        sorted_ranges = sorted(self.ranges, key=lambda x: x.start)
        gaps = []

        for i in range(len(sorted_ranges) - 1):
            current = sorted_ranges[i]
            next_range = sorted_ranges[i + 1]

            if current.end < next_range.start:
                gaps.append(Range(current.end, next_range.start))

        return gaps

    def __len__(self) -> int:
        """Количество диапазонов в контейнере"""
        return len(self.ranges)

    def __str__(self) -> str:
        """Строковое представление"""
        ranges_str = "\n  ".join(str(r) for r in self.ranges)
        return f"RangeContainer '{self.name}' ({len(self)} диапазонов):\n  {ranges_str}"

def demonstrate_range_operations() -> None:
    """Демонстрация работы с универсальными диапазонами"""

    print("=== Демонстрация универсального класса Range ===\n")

    # 1. Работа с целыми числами
    print("1. Диапазоны целых чисел:")
    int_range1 = Range(ComparableInt(1), ComparableInt(10))
    int_range2 = Range(ComparableInt(5), ComparableInt(15))
    int_range3 = Range(ComparableInt(20), ComparableInt(25))

    print(f"   Range 1: {int_range1}")
    print(f"   Range 2: {int_range2}")
    print(f"   Range 3: {int_range3}")
    print(f"   Длина Range 1: {int_range1.length}")
    print(f"   5 в Range 1? {ComparableInt(5) in int_range1}")
    print(f"   15 в Range 1? {ComparableInt(15) in int_range1}")
    print(f"   Range 1 и Range 2 пересекаются? {int_range1.overlaps(int_range2)}")
    print(f"   Пересечение Range 1 и Range 2: {int_range1.intersection(int_range2)}")

    # 2. Работа с вещественными числами
    print("\n2. Диапазоны вещественных чисел:")
    float_range1 = Range(ComparableFloat(1.5), ComparableFloat(5.5))
    float_range2 = Range(ComparableFloat(3.0), ComparableFloat(7.0))

    print(f"   Float Range 1: {float_range1}")
    print(f"   Float Range 2: {float_range2}")
    print(f"   3.14 в Float Range 1? {ComparableFloat(3.14) in float_range1}")
    print(f"   Пересечение: {float_range1.intersection(float_range2)}")
```

```
    # 3. Сравнение диапазонов
    print("\n3. Сравнение диапазонов:")
    print(
        f"   Range [1, 5) < Range [2, 6)? {Range(ComparableInt(1), ComparableInt(5)) <
Range(ComparableInt(2), ComparableInt(6))}"
    )
    print(
        f"   Range [1, 5) == Range [1, 5)? {Range(ComparableInt(1), ComparableInt(5)) ==
Range(ComparableInt(1), ComparableInt(5))}"
    )

    # 4. Работа с контейнером
    print("\n4. Работа с контейнером диапазонов:")
    container = RangeContainer("TestContainer")
    container.add_ranges(int_range1, int_range2, int_range3)

    print(container)
    print(f"   Всего диапазонов: {len(container)}")

    # 5. Поиск в контейнере (ИСПРАВЛЕННЫЙ КОД)
    print("\n5. Поиск в контейнере:")
    test_values = [ComparableInt(3), ComparableInt(12), ComparableInt(22)]
    for val in test_values:
        found = container.find_containing_range(val)
        if found is not None:  # Явная проверка на None
            print(f"   Значение {val} находится в диапазоне {found}")
        else:
            print(f"   Значение {val} не найдено ни в одном диапазоне")

    # 6. Объединение диапазонов
    print("\n6. Объединение пересекающихся диапазонов:")
    merged_container = container.merge_overlapping_ranges()
    print(merged_container)

    # 7. Общий охват и промежутки
    print("\n7. Общий охват и промежутки:")
    total_coverage = container.get_total_coverage()
    print(f"   Общий охват: {total_coverage}")

    gaps = container.get_gaps()
    if gaps:
        print(f"   Промежутки между диапазонами: {gaps}")
    else:
        print("   Промежутков нет")

    # 8. Фильтрация
    print("\n8. Фильтрация диапазонов по значению:")
    value_to_find = ComparableInt(7)
    containing_ranges = container.filter_by_value(value_to_find)
    print(f"   Диапазоны, содержащие {value_to_find}: {containing_ranges}")
```

```
    # 9. Разделение диапазона
    print("\n9. Разделение диапазона:")
    try:
        split_result = int_range1.split(ComparableInt(5))
        print(f"   Разделение {int_range1} в точке 5: {split_result}")
    except ValueError as e:
        print(f"   Ошибка при разделении: {e}")

    # 10. Проверка на смежность
    print("\n10. Проверка на смежность:")
    range_a = Range(ComparableInt(1), ComparableInt(5))
    range_b = Range(ComparableInt(5), ComparableInt(10))
    range_c = Range(ComparableInt(10), ComparableInt(15))

    print(f"   {range_a} смежен с {range_b}? {range_a.is_adjacent_to(range_b)}")
    print(f"   {range_a} смежен с {range_c}? {range_a.is_adjacent_to(range_c)}")

    # 11. Объединение смежных диапазонов
    print("\n11. Объединение смежных диапазонов:")
    adjacent_container = RangeContainer("Adjacent")
    adjacent_container.add_ranges(range_a, range_b, range_c)
    print("   До объединения:")
    print(f"   {adjacent_container}")
    print("   После объединения:")
    merged_adjacent = adjacent_container.merge_overlapping_ranges()
    print(f"   {merged_adjacent}")

    # 12. Итерация по целочисленному диапазону
    print("\n12. Итерация по целочисленному диапазону:")
    iter_range = Range(ComparableInt(1), ComparableInt(6))
    print(f"   Значения в диапазоне {iter_range}: ", end="")
    for val in iter_range:
        print(f"{val} ", end="")
    print()

    # 13. Обработка ошибок
    print("\n13. Обработка ошибок:")
    try:
        bad_range = Range(ComparableInt(10), ComparableInt(1))  # start > end
    except ValueError as e:
        print(f"   Ошибка при создании диапазона: {e}")

    try:
        bad_split = int_range1.split(ComparableInt(15))  # точка вне диапазона
    except ValueError as e:
        print(f"   Ошибка при разделении: {e}")

    print("\n=== Демонстрация завершена ===")

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

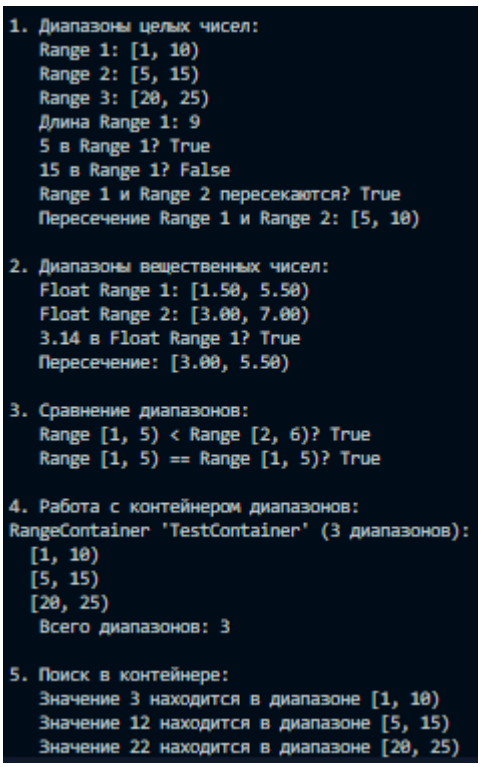

Рисунок 2. Пример работы кода

Вывод: приобрел навыки по работе с классами данных при написании программ с помощью языка программирования Python версии 3.x.