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 №5
**дисциплины**
**«Объектно-ориентированное программирование»**
**Вариант № 3**

Выполнил:
Левашев Тимур Рашидович
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)

Проверил:
Доцент департамента цифровых, робототехнических систем и электроники института перспективной инженерии Воронкин Р.А

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2025 г.

**Тема работы:** “Классы данных в Python”.

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

**Порядок выполнения работы:**

Ссылка на Git репозиторий: https://github.com/mazy99/oop_pract_5

1. **Пример 1:** Данный пример демонстрирует использование декоратора @dataclass для упрощённого создания классов-контейнеров данных в Python.

Листинг программы:

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


from dataclasses import dataclass


@dataclass
class DataClassCard:
    rank: str
    suit: str


def main():
    queen_of_hearts = DataClassCard("Q", "Hearts")
    print(queen_of_hearts.rank)
    print(queen_of_hearts)
    print(queen_of_hearts == DataClassCard("Q", "Hearts"))


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

```
Q
DataClassCard(rank='Q', suit='Hearts')
True
```

Рисунок 1 – Результат выполнения программы

2. **Пример 2:** В данном примере демонстрируется использование функции make_dataclass для динамического создания дата-класса Position.

Такой класс автоматически получает конструктор и удобное строковое представление, аналогично декоратору @dataclass.

Листинг прогарммы:

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

from dataclasses import make_dataclass

Position = make_dataclass(
    "Position",
    [
        ("name", str),
        ("lat", float),
        ("lon", float),
    ],
)


def main():

    moscow = Position("Moscow", 55.7558, 37.6176)
    print("Пример 1:", moscow)

    moscow.lat = 55.75
    print("Пример 2 (изменение lat):", moscow)

    cities = [
        Position("Paris", 48.8566, 2.3522),
        Position("Tokyo", 35.6895, 139.6917),
        Position("New York", 40.7128, -74.0060),
    ]

    print("\nПример 3 (список точек):")
    for city in cities:
        print(" •", city)

    sorted_cities = sorted(cities, key=lambda c: c.name)
    print("\nПример 4 (сортировка по имени):")
    for city in sorted_cities:
        print(" •", city)


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

```
Пример 1: Position(name='Moscow', lat=55.7558, lon=37.6176)
Пример 2 (изменение lat): Position(name='Moscow', lat=55.75, lon=37.6176)

Пример 3 (список точек):
  • Position(name='Paris', lat=48.8566, lon=2.3522)
  • Position(name='Tokyo', lat=35.6895, lon=139.6917)
  • Position(name='New York', lat=40.7128, lon=-74.006)

Пример 4 (сортировка по имени):
  • Position(name='New York', lat=40.7128, lon=-74.006)
  • Position(name='Paris', lat=48.8566, lon=2.3522)
  • Position(name='Tokyo', lat=35.6895, lon=139.6917)
(oop-pract-5) PS C:\учеба\ООП\5 лаба\oop_pract_5>
```

Рисунок 2 – Результат выполнения программы

3. **Пример 3:** В данном примере используется декоратор @dataclass с параметром frozen=True, который делает экземпляры класса Position неизменяемыми (аналогично кортежам).

Листинг программы:

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

from dataclasses import dataclass


@dataclass(frozen=True)
class Position:
    name: str
    lon: float = 0.0
    lat: float = 0.0


def main():

    pos = Position("Oslo", 10.8, 59.9)
    print("Город:", pos.name)
    print("Долгота:", pos.lon)
    print("Широта:", pos.lat)

    pos2 = Position("Oslo", 10.8, 59.9)
    print("\nПример 2 (сравнение):", pos == pos2)

    locations = {
        pos: "Столица Норвегии",
        Position("Berlin", 13.4, 52.5): "Столица Германии",
    }
    print("\nПример 3 (использование как ключа словаря):")
```

```
    for key, value in locations.items():
        print(f"  {key} -> {value}")

    print("\nПример 4 (неизменяемость):")
    try:
        pos.name = "New Name"
    except Exception as exc:
        print("  Ошибка при попытке изменить объект:", exc)


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

```
Город: Oslo
Долгота: 10.8
Широта: 59.9

Пример 2 (сравнение): True

Пример 3 (использование как ключа словаря):
  Position(name='Oslo', lon=10.8, lat=59.9) -> Столица Норвегии
  Position(name='Berlin', lon=13.4, lat=52.5) -> Столица Германии

Пример 4 (неизменяемость):
  Ошибка при попытке изменить объект: cannot assign to field 'name'
```

Рисунок 3 – Результат выполнения программы

4. **Пример 4:** В данном примере демонстрируется использование механизма наследования в сочетании с декоратором @dataclass. Базовый класс Position содержит общие поля для географической точки: название, долготу и широту. Класс Capital наследует эти поля и расширяет структуру дополнительным атрибутом country, описывающим страну, столицей которой является данная точка.

Листинг программы:

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


from dataclasses import dataclass


@dataclass
class Position:
    name: str
```

```
    lon: float = 0.0
    lat: float = 0.0


@dataclass
class Capital(Position):
    country: str = ""


def main():
    cap1 = Capital("Осло", 10.8, 59.9, "Норвегия")
    cap2 = Capital("Москва", 37.6, 55.8, "Россия")
    cap3 = Capital("Париж", 2.35, 48.85, "Франция")

    pos1 = Position("Эйфелева башня", 2.2945, 48.8584)
    pos2 = Position("Большой каньон", -112.1129, 36.1069)
    pos3 = Position("Стадион Сан-Сиро", 9.2773, 45.4781)

    capitals = [cap1, cap2, cap3]
    positions = [pos1, pos2, pos3]

    print("Столицы:")
    for cap in capitals:
        print(cap)

    print("\nПрочие позиции:")
    for pos in positions:
        print(pos)


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

```
Столицы:
Capital(name='Осло', lon=10.8, lat=59.9, country='Норвегия')
Capital(name='Москва', lon=37.6, lat=55.8, country='Россия')
Capital(name='Париж', lon=2.35, lat=48.85, country='Франция')

Прочие позиции:
Position(name='Эйфелева башня', lon=2.2945, lat=48.8584)
Position(name='Большой каньон', lon=-112.1129, lat=36.1069)
Position(name='Стадион Сан-Сиро', lon=9.2773, lat=45.4781)
```

Рисунок 4 – Результат выполнения программы

5. **Задание 1:** Создайте дата класс Product с полями: название, цена, категория. Выведите список товаров определенной категории, введенной пользователем.

Во всех предлагаемых задачах требуется описать данные с помощью механизма `dataclass` в Python, чтобы представить каждую сущность — книгу, студента, поезд, товар или любой другой объект, указанный в условии — в виде простой и удобной структуры. Каждый датакласс должен содержать набор полей, чётко описывающих характеристики этой сущности, а также их типы. Для хранения множества таких объектов нужно создать отдельный датакласс-контейнер с полем-коллекцией, использующим `default_factory=list`, чтобы правильно инициализировать пустой список. Контейнер должен обеспечивать добавление новых объектов и хранение всех созданных элементов. В зависимости от задачи контейнер также должен уметь выполнять операции поиска, фильтрации, сортировки или группировки данных по заданным критериям. Методы контейнера должны возвращать отфильтрованные списки, агрегированные данные или отсортированные коллекции. Логика обработки должна находиться в самом контейнере, чтобы разделить представление данных и операции над ними. Для каждого задания необходимо продемонстрировать работу программы: создать несколько объектов, поместить их в контейнер и выполнить требуемые операции. Реализация должна быть аккуратной и опираться на типизацию. Все действия над данными должны использовать созданные структуры, а не работать напрямую со словарями или произвольными списками. Важно, чтобы решение каждой задачи оставалось модульным: датакласс описывает структуру, контейнер — работу с набором объектов, а основная часть программы — демонстрацию возможностей. Такой подход позволяет получить чистый, понятный и расширяемый код.

Рисунок 5 – Условия задания 1

Листинг программы:

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


from dataclasses import dataclass, field


@dataclass(frozen=True)
class Product:
    name: str
    price: float
    category: str


@dataclass
class Product_list:
    products: list[Product] = field(default_factory=list)

    def add(self, name: str, price: float, category: str) -> None:
        new_product = Product(name=name, price=price, category=category)
        self.products.append(new_product)
        self.products.sort(key=lambda product: product.name)

    def get_products_by_category(self, category: str) -> str:
        filtered_products = [
            product for product in self.products if product.category == category
```

```
        ]
        if not filtered_products:
            return f"В категории '{category}' нет продуктов."
        return f"Продукты в категории '{category}':\n" + "\n".join(
            f"- {product.name}: {product.price} ({product.category})"
            for product in filtered_products
        )

    def __str__(self) -> str:
        try:
            if not self.products:
                raise ValueError(
                    "Списко продуктов пуст, добавьте хотя бы один продукт."
                )
            return "Список продуктов:\n" + "\n".join(
                f"- {product.name}: {product.price} ({product.category})"
                for product in self.products
            )
        except Exception as ex:
            return f"Ошибка: {ex}"
```

Листинг вызова программы:

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


from product import Product_list

if __name__ == "__main__":

    print("=== Создание списка и добавление продуктов ===")
    plist = Product_list()

    plist.add("Хлеб", 45.0, "Еда")
    plist.add("Молоко", 80.5, "Еда")
    plist.add("Шампунь", 250.0, "Быт")
    plist.add("Яблоки", 120.0, "Еда")

    print(plist)
    print()

    print("=== Фильтрация по категории ===")
    print(plist.get_products_by_category("Еда"))
    print()

    print("=== Пустая категория ===")
    print(plist.get_products_by_category("Техника"))
    print()
```

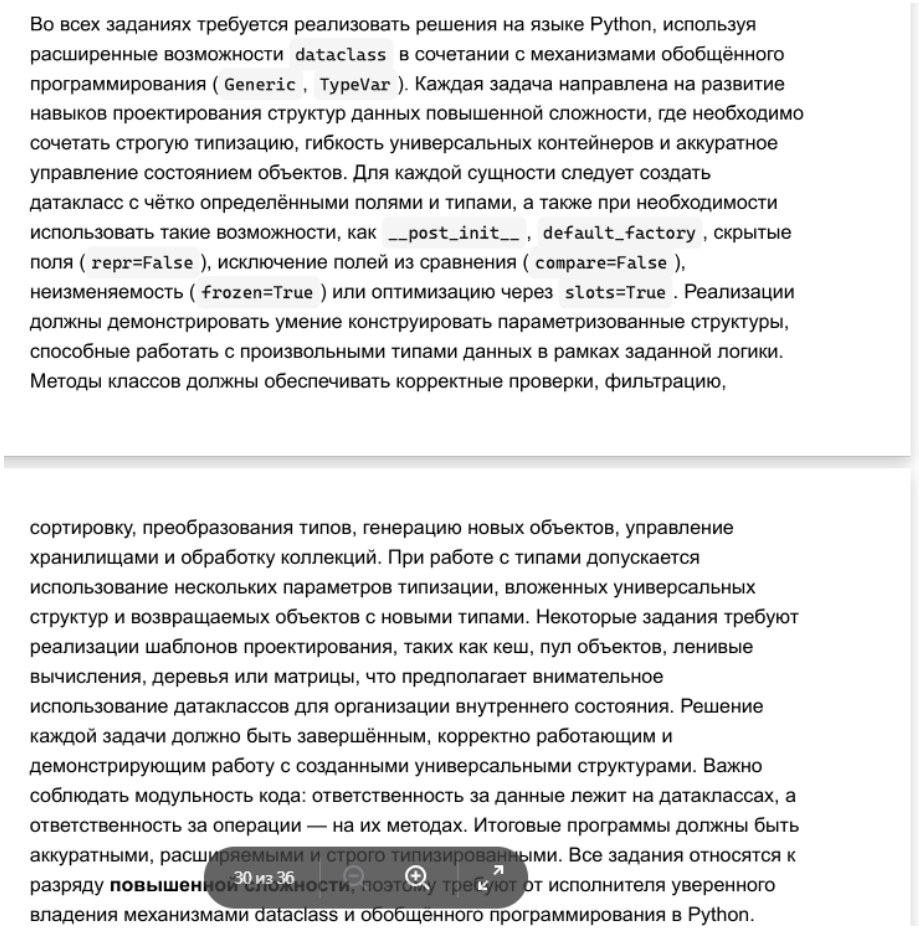
```
print("=== Проверка обработки пустого списка ===")
empty_list = Product_list()
print(empty_list)
```

```
=== Создание списка и добавление продуктов ===
Список продуктов:
- Молоко: 80.5 (Еда)
- Хлеб: 45.0 (Еда)
- Шампунь: 250.0 (Быт)
- Яблоки: 120.0 (Еда)

=== Фильтрация по категории ===
Продукты в категории 'Еда':
- Молоко: 80.5 (Еда)
- Хлеб: 45.0 (Еда)
- Яблоки: 120.0 (Еда)

=== Пустая категория ===
В категории 'Техника' нет продуктов.

=== Проверка обработки пустого списка ===
Ошибка: Списко продуктов пуст, добавьте хотя бы один продукт.
```

Рисунок 6 – Результат выполнения программы

6. **Задание 2:** Создайте класс LimitedStorage(Generic[T]) с коллекцией items: list[T] и полем capacity: int.

В add() запрещайте добавление элемента, если хранилище заполнено. Используйте __post_init__ для проверки, что capacity > 0.



Во всех заданиях требуется реализовать решения на языке Python, используя расширенные возможности `dataclass` в сочетании с механизмами обобщённого программирования ( `Generic` , `TypeVar` ). Каждая задача направлена на развитие навыков проектирования структур данных повышенной сложности, где необходимо сочетать строгую типизацию, гибкость универсальных контейнеров и аккуратное управление состоянием объектов. Для каждой сущности следует создать датакласс с чётко определёнными полями и типами, а также при необходимости использовать такие возможности, как `__post_init__` , `default_factory` , скрытые поля ( `repr=False` ), исключение полей из сравнения ( `compare=False` ), неизменяемость ( `frozen=True` ) или оптимизацию через `slots=True` . Реализации должны демонстрировать умение конструировать параметризованные структуры, способные работать с произвольными типами данных в рамках заданной логики. Методы классов должны обеспечивать корректные проверки, фильтрацию,

сортировку, преобразования типов, генерацию новых объектов, управление хранилищами и обработку коллекций. При работе с типами допускается использование нескольких параметров типизации, вложенных универсальных структур и возвращаемых объектов с новыми типами. Некоторые задания требуют реализации шаблонов проектирования, таких как кеш, пул объектов, ленивые вычисления, деревья или матрицы, что предполагает внимательное использование датаклассов для организации внутреннего состояния. Решение каждой задачи должно быть завершённым, корректно работающим и демонстрирующим работу с созданными универсальными структурами. Важно соблюдать модульность кода: ответственность за данные лежит на датаклассах, а ответственность за операции — на их методах. Итоговые программы должны быть аккуратными, расширяемыми и строго типизированными. Все задания относятся к разряду **повышенной сложности**, поэтому требуют от исполнителя уверенного владения механизмами dataclass и обобщенного программирования в Python.

30 из 36

Рисунок 7 – Условия задания 2

Листинг программы:

```
#!/usr/bin/env python3

# -*- coding: utf-8 -*


from dataclasses import dataclass, field
from typing import Generic, TypeVar

T = TypeVar("T")


@dataclass
class LimitedStorage(Generic[T]):

    items: list[T] = field(default_factory=list, repr=False)
    capacity: int = 1

    def __post_init__(self) -> None:
        if self.capacity <= 0:
            raise ValueError("Ёмкость хранилища должна быть больше нуля")
        if len(self.items) > self.capacity:
            raise ValueError("Начальный список элементов превышает ёмкость
хранилища")
```

```
    def add(self, item: T) -> None:
        if len(self.items) >= self.capacity:
            raise OverflowError(
                "Хранилище заполнено— невозможно добавить новый элемент"
            )
        self.items.append(item)

    def len(self) -> int:
        return len(self.items)

    def is_full(self) -> bool:
        return len(self.items) == self.capacity

    def __repr__(self) -> str:
        return f"Ограниченное Хранилище\
    (вместимость={self.capacity}, элементы={self.items})"
```

Листинг вызова программы:

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


from storage_package.limited_storage import LimitedStorage

if __name__ == "__main__":

    print("=== Создание с неверной ёмкостью ===")
    try:
        s = LimitedStorage[int](capacity=-3)
    except ValueError as e:
        print("Ошибка при создании:", e)
    print()

    print("=== Переполнение хранилища ===")
    s = LimitedStorage[int](capacity=2)
    try:
        s.add(10)
        s.add(20)
        s.add(30)
    except OverflowError as e:
        print("Ошибка при добавлении:", e)
    print()

    print("=== Корректная работа хранилища ===")
    s = LimitedStorage[int](capacity=3)
    s.add(10)
    s.add(20)
```

```
    print("Текущее состояние:", s)
    print("Заполнено?", s.is_full())

    s.add(30)
    print("После добавления третьего элемента — заполнено?", s.is_full())
    print("Итоговое состояние:", s)
```

```
=== Создание с неверной ёмкостью ===
Ошибка при создании: Ёмкость хранилища должна быть больше нуля

=== Переполнение хранилища ===
Ошибка при добавлении: Хранилище заполнено— невозможно добавить новый элемент

=== Корректная работа хранилища ===
Текущее состояние: Ограниченное Хранилище    (вместимость=3, элементы=[10, 20])
Заполнено? False
После добавления третьего элемента — заполнено? True
Итоговое состояние: Ограниченное Хранилище    (вместимость=3, элементы=[10, 20, 30])
```

Рисунок 8 – Результат выполнения программы

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