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 13

Выполнил:
Рябинин Егор Алексеевич
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 Роман Александрович

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2025 г

Тема: Классы данных в Python.

Цель: приобретение навыков по работе с классами данных при написании программ с помощью языка программирования Python версии 3.13.3.

Порядок выполнения работы:

Ссылка на репозиторий:

https://github.com/bohemiaaaaa/Lab5_Object-oriented-programming

Задание №1. 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.

13. Учёт заказов

Создайте датакласс Order с полями: номер заказа, клиент, сумма. Выведите заказы с суммой выше заданной.

Листинг программы:

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


from dataclasses import dataclass, field
from typing import List


@dataclass
class Order:
    # Датакласс для представления заказа
    order_id: int
    client: str
    amount: float


@dataclass
class OrderContainer:
    # Контейнер для хранения и обработки заказов
    _orders: List[Order] = field(default_factory=list, repr=False)

    def add_order(self, order_id: int, client: str, amount: float) -> None:
        # Добавление нового заказа в контейнер
        new_order = Order(order_id=order_id, client=client, amount=amount)
        self._orders.append(new_order)

    def get_orders_above_amount(self, min_amount: float) -> List[Order]:
        # Получение заказа с суммой выше заданной
        result = [order for order in self._orders if order.amount >
min_amount]
        return result

    def get_all_orders(self) -> List[Order]:
        # Получение всех заказов
        return self._orders.copy()

    def display_orders(self, orders_list: List[Order] = None) -> None:
        # Вывод списка заказов
        if orders_list is None:
            orders_list = self._orders

        if not orders_list:
            print("Список заказов пуст.")
            return

        print("\n" + "=" * 60)
        print(f"{'№ заказа':<12} {'Клиент':<25} {'Сумма':<10}")
        print("=" * 60)

        for order in orders_list:
            print(f"{order.order_id:<12} {order.client:<25}
```

```
{order.amount:<10.2f}")
        print("=" * 60)
```

Листинг программы:

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


from order_tracking import OrderContainer


def main() -> None:
    container = OrderContainer()

    container.add_order(1001, "Иванов А.А.", 12500.50)
    container.add_order(1002, "Петров Б.Б.", 8500.00)
    container.add_order(1003, "Сидоров В.В.", 21000.75)
    container.add_order(1004, "Кузнецов Г.Г.", 5500.25)
    container.add_order(1005, "Смирнов Д.Д.", 18000.00)

    print("=" * 60)
    print("Все заказы:")
    container.display_orders()

    min_amount = 10000.00
    print(f"\nЗаказы с суммой выше {min_amount:.2f}:")
    filtered_orders = container.get_orders_above_amount(min_amount)
    container.display_orders(filtered_orders)

    min_amount2 = 15000.00
    print(f"\nЗаказы с суммой выше {min_amount2:.2f}:")
    filtered_orders2 = container.get_orders_above_amount(min_amount2)
    container.display_orders(filtered_orders2)


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

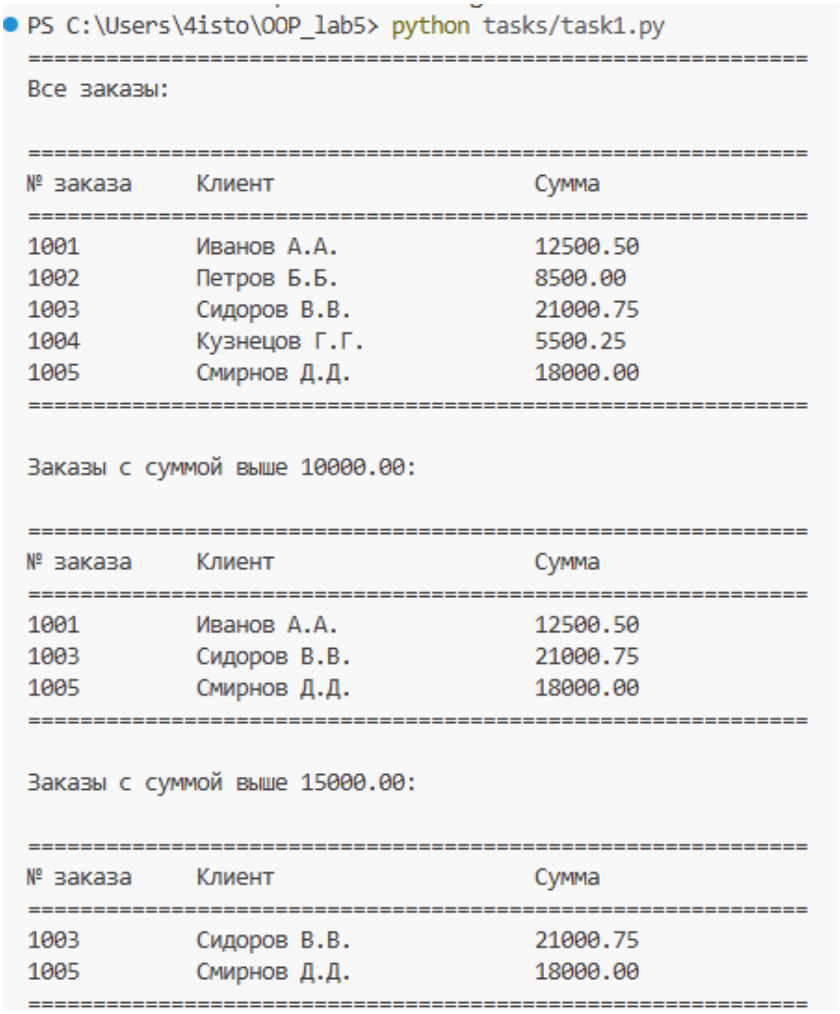

Рисунок 1 – Результат работы программы

Задание №2. 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 обобщённого программирования в Python.

13. Ленивая инициализация значения

Создайте `Lazy(Generic[T])` с полем `factory: Callable[[], T]` и `_value: T | None`. Добавьте метод `get()`, который вызывает `factory()` при первом обращении.

Листинг программы:

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


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

T = TypeVar("T")


@dataclass
class Lazy(Generic[T]):
    # Класс для ленивой инициализации значения
    factory: Callable[[], T] = field(repr=False)
    _value: Optional[T] = field(default=None, init=False, repr=False,
compare=False)
```

```
    def get(self) -> T:
        # Получение значения, инициализируя его при первом обращении
        if self._value is None:
            self._value = self.factory()
        return self._value
```

Листинг программы:

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


from lazy_package.lazy_init import Lazy


def main() -> None:
    print("=" * 60)
    print("Демонстрация Lazy(Generic[T])")
    print("=" * 60)

    # Пример 1: Базовая демонстрация
    print("\n1. Ленивая строка:")
    lazy_string = Lazy[str](factory=lambda: "Рябинин" + " Егор")
    print("   Объект создан, factory ещё не вызвана")
    print(f"   Первый вызов get(): {lazy_string.get()}")
    print(f"   Второй вызов get(): {lazy_string.get()} (кэшировано)")

    # Пример 2: С подсчётом вызовов factory
    print("\n2. Проверка однократного вызова factory:")
    call_count = 0

    def counting_factory() -> str:
        nonlocal call_count
        call_count += 1
        return f"factory вызвана {call_count} раз"

    lazy_counter = Lazy[str](factory=counting_factory)
    result1 = lazy_counter.get()
    result2 = lazy_counter.get()
    print(f"   Результат 1: {result1}")
    print(f"   Результат 2: {result2}")
    print(f"   call_count = {call_count} (factory вызвана 1 раз)")

    # Пример 3: С числом (вычисление в factory)
    print("\n3. Ленивое число с вычислением в factory:")
    lazy_number = Lazy[int](factory=lambda: 10 + 32)
    print(f"   Результат get(): {lazy_number.get()}")


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

```
PS C:\Users\4isto\OOP_lab5> python tasks/task2.py
================================================
Демонстрация Lazy(Generic[T])
================================================

1. Ленивая строка:
   Объект создан, factory ещё не вызвана
   Первый вызов get(): Рябинин Егор
   Второй вызов get(): Рябинин Егор (кэшировано)

2. Проверка однократного вызова factory:
   Результат 1: factory вызвана 1 раз
   Результат 2: factory вызвана 1 раз
   call_count = 1 (factory вызвана 1 раз)

3. Ленивое число с вычислением в factory:
   Результат get(): 42
```

Рисунок 2 – Результат работы программы

Результат работы тестов для написанных программ:

```
PS C:\Users\4isto\OOP_lab5> pytest
============================= test session starts =
platform win32 -- Python 3.11.0, pytest-8.3.5, pluggy-1.6.0 -- C:\Program Files\Python311\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\4isto\OOP_lab5
configfile: pyproject.toml
testpaths: tests
plugins: anyio-4.9.0
collected 22 items

tests/test_task1.py::TestOrder::test_order_creation PASSED
tests/test_task1.py::TestOrder::test_order_repr PASSED
tests/test_task1.py::TestOrderContainer::test_empty_container PASSED
tests/test_task1.py::TestOrderContainer::test_add_order PASSED
tests/test_task1.py::TestOrderContainer::test_get_all_orders_returns_copy PASSED
tests/test_task1.py::TestOrderContainer::test_get_orders_above_amount_empty PASSED
tests/test_task1.py::TestOrderContainer::test_get_orders_above_amount_none PASSED
tests/test_task1.py::TestOrderContainer::test_get_orders_above_amount_all PASSED
tests/test_task1.py::TestOrderContainer::test_get_orders_above_amount_some PASSED
tests/test_task1.py::TestOrderContainer::test_get_orders_above_amount_exact_boundary PASSED
tests/test_task1.py::test_integration PASSED
tests/test_task2.py::TestLazy::test_lazy_creation PASSED
tests/test_task2.py::TestLazy::test_get_first_call PASSED
tests/test_task2.py::TestLazy::test_get_multiple_calls PASSED
tests/test_task2.py::TestLazy::test_different_types PASSED
tests/test_task2.py::TestLazy::test_factory_with_computation PASSED
tests/test_task2.py::TestLazy::test_none_value_initial PASSED
tests/test_task2.py::TestLazy::test_repr_hides_internals PASSED
tests/test_task2.py::TestLazy::test_compare_ignores_value PASSED
tests/test_task2.py::TestLazy::test_same_factory_different_instances PASSED
tests/test_task2.py::TestLazy::test_factory_returns_none PASSED
tests/test_task2.py::test_lazy_integration PASSED

============================== 22 passed in 0.07s ==
```

Рисунок 3 – Результат работы тестов

Контрольные вопросы:

1. Что такое dataclass и для чего он используется в Python?

Dataclass – это декоратор в Python, который автоматически генерирует специальные методы для классов, предназначенных в основном для хранения данных. Он используется для создания простых классов-структур, которые автоматически получают методы __init__, __repr__, __eq__ и другие, уменьшая объем шаблонного кода.

2. Как создать класс данных в языке Python?

Для создания класса данных используется декоратор @dataclass. Сначала импортируется dataclass из модуля dataclasses, затем класс объявляется с этим декоратором, а внутри класса определяются поля с аннотациями типов.

3. Какие методы по умолчанию реализует класс данных?

По умолчанию dataclass генерирует методы __init__ для инициализации полей, __repr__ для строкового представления объекта и __eq__ для сравнения объектов по значениям полей.

4. Как создать неизменяемый класс данных?

Для создания неизменяемого класса данных нужно установить параметр frozen=True в декораторе @dataclass(frozen=True). Это сделает все поля объекта доступными только для чтения после создания.

5. Какие методы автоматически генерирует dataclass при создании класса?

При создании класса dataclass автоматически генерирует методы __init__, __repr__, __eq__, а при использовании параметра order=True также генерирует методы сравнения __lt__, __le__, __gt__, __ge__.

6. Какой параметр декортатора @dataclass позволяет сделать объект неизменяемым?

Параметр frozen=True в декораторе @dataclass позволяет сделать объект неизменяемым.

7. Что делает параметр order=True в объявлении dataclass?

Параметр order=True включает автоматическую генерацию методов сравнения (__lt__, __le__, __gt__, __ge__), позволяя сравнивать объекты класса между собой.

8. Для чего используется функция field() и в каких случаях она необходима?

Функция field() используется для настройки отдельных полей датакласса. Она необходима когда нужно задать специальные параметры для поля, такие как default_factory, repr, compare или metadata.

9. Как работает default_factory и чем он отличается от обычного значения по умолчанию?

default_factory принимает функцию без аргументов, которая вызывается для создания значения по умолчанию каждый раз при создании нового экземпляра. В отличие от обычного значения по умолчанию, которое создается один раз и разделяется между всеми экземплярами, default_factory создает новое значение для каждого экземпляра.

10. Почему использование default_factory=list предпочтительнее, чем items: list = []?

Использование default_factory=list предпочтительнее, потому что items: list = [] создает один общий список для всех экземпляров класса, что приводит к неожиданному разделению данных между объектами. default_factory создает новый список для каждого экземпляра.

11. Что такое метод __post_init__ и для каких целей он применяется?

Метод __post_init__ вызывается автоматически после __init__ и используется для выполнения дополнительной инициализации или проверок, например для вычисления производных полей или валидации данных.

12. Как исключить поле из генерации метода __repr__ в dataclass?

Чтобы исключить поле из __repr__, нужно использовать параметр repr=False в функции field(): field(repr=False).

13. Как исключить поле из сравнения (==, <, >) в dataclass?

Чтобы исключить поле из сравнения, нужно использовать параметр compare=False в функции field(): field(compare=False).

14. Что произойдёт, если попытаться изменить поле объекта при frozen=True?

При попытке изменить поле объекта с frozen=True будет вызвано исключение dataclasses.FrozenInstanceError.

15. Как можно создать копию датакласса с частичной заменой полей?

Для создания копии с частичной заменой полей используется функция replace() из модуля dataclasses: replace(existing_object, field1=new_value1, field2=new_value2).

16. Что делает функция asdict() и когда её используют?

Функция asdict() преобразует объект датакласса в словарь. Её используют когда нужно сериализовать объект в JSON или передать данные в формате словаря.

17. Как dataclass взаимодействует с типами из модуля typing (например list[int], Optional[X])?

Dataclass полностью поддерживает аннотации типов из модуля typing. Поля могут быть объявлены с использованием Generic, Optional, List и других типов, которые используются для проверки типов, но не влияют на выполнение кода в runtime.

18. Как сделать так, чтобы датакласс использовал меньше памяти при большом числе объектов?

Чтобы датакласс использовал меньше памяти, можно добавить __slots__ = True или явно определить __slots__, что предотвращает создание словаря __dict__ для каждого экземпляра.

19. Может ли dataclass наследоваться от обычного класса и наоборот?

Dataclass может наследоваться от обычного класса, и обычный класс может наследоваться от датакласса. Однако при наследовании датаклассов нужно учитывать порядок и значения по умолчанию полей.

20. Что такое InitVar и для каких полей он применяется?

InitVar – это специальный тип для полей, которые передаются в __init__ и доступны в __post_init__, но не становятся атрибутами экземпляра. Используется для передачи временных данных при инициализации.

21. В каком виде хранятся аннотации типов внутри датакласса?

Аннотации типов хранятся в атрибуте __annotations__ класса и используются dataclass для определения полей и их типов.

22. Что будет, если определить __init__ вручную внутри датакласса?

Если определить __init__ вручную, то dataclass не будет генерировать автоматический метод __init__, что лишает преимуществ автоматической инициализации полей.

23. Какие ситуации требуют использования slots=True, и какие преимущества это даёт?

Slots=True используется когда нужно уменьшить потребление памяти и ускорить доступ к атрибутам при большом количестве объектов. Преимущества: экономия памяти, более быстрый доступ к атрибутам, но теряется возможность динамического добавления атрибутов.

Вывод: в ходе лабораторной работы были приобретены навыки по работе с классами данных при написании программ с помощью языка программирования Python версии 3.13.3.