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»

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: Аннотация типов.

Цель: приобрести навыки по работе с аннотациями типов при написании программ с помощью языка программирования Python версии 3.x. Рассмотреть вопрос контроля типов переменных и функций с использованием комментариев и аннотаций. Рассмотреть описание PEP`ов, регламентирующих работу с аннотациями, и изучить работу с инструментом «mypy» для анализа Python кода.

Ход работы:

Проработка примеров лабораторной работы:

Пример 1.

Необходимо для примера лабораторной работы 4.4 предусмотреть аннотации типов. Код программы примера 1:

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

import logging
import sys
import xml.etree.ElementTree as ET
from dataclasses import dataclass, field
from datetime import date
from typing import List


# Класс пользовательского исключения в случае, если неверно
# введен номер года.
class IllegalYearError(Exception):

    def __init__(self, year, message="Illegal year number"):
        self.year = year
        self.message = message
        super(IllegalYearError, self).__init__(message)

    def __str__(self):
        return f"{self.year} -> {self.message}"


# Класс пользовательского исключения в случае, если введенная
# команда является недопустимой.
class UnknownCommandError(Exception):

    def __init__(self, command, message="Unknown command"):
        self.command = command
        self.message = message
        super(UnknownCommandError, self).__init__(message)

    def __str__(self):
        return f"{self.command} -> {self.message}"


@dataclass(frozen=True)
```

```
class Worker:
    name: str
    post: str
    year: int


@dataclass
class Staff:
    workers: List[Worker] = field(default_factory=lambda: [])

    def add(self, name: str, post: str, year: int) -> None:
        # Получить текущую дату.
        today = date.today()

        if year < 0 or year > today.year:
            raise IllegalYearError(year)

        self.workers.append(Worker(name=name, post=post, year=year))

        self.workers.sort(key=lambda worker: worker.name)

    def __str__(self) -> str:
        # Заголовок таблицы.
        table = []
        line = "+-{}-+-{}-+-{}-+-{}-+".format("-" * 4, "-" * 30, "-" * 20, "-
" * 8)
        table.append(line)
        table.append("| {:^4} | {:^30} | {:^20} | {:^8} |".format("№",
"Ф.И.О.", "Должность", "Год"))
        table.append(line)

        # Вывести данные о всех сотрудниках.
        for idx, worker in enumerate(self.workers, 1):
            table.append("| {:>4} | {:<30} | {:<20} | {:>8} |".format(idx,
worker.name, worker.post, worker.year))

        table.append(line)

        return "\n".join(table)

    def select(self, period: int) -> List[Worker]:
        # Получить текущую дату.
        today = date.today()

        result: List[Worker] = []
        for worker in self.workers:
            if today.year - worker.year >= period:
                result.append(worker)

        return result

    def load(self, filename: str) -> None:
        with open(filename, "r", encoding="utf8") as fin:
            xml = fin.read()

        parser = ET.XMLParser(encoding="utf8")
        tree = ET.fromstring(xml, parser=parser)

        self.workers = []
        for worker_element in tree:
            name, post, year = None, None, None

            for element in worker_element:
                if element.tag == "name":
```

```
                    name = element.text
                elif element.tag == "post":
                    post = element.text
                elif element.tag == "year":
                    year = int(element.text)  # type: ignore

                if name is not None and post is not None and year is not
None:
                    self.workers.append(Worker(name=name, post=post,
year=year))

    def save(self, filename: str) -> None:
        root = ET.Element("workers")
        for worker in self.workers:
            worker_element = ET.Element("worker")

            name_element = ET.SubElement(worker_element, "name")
            name_element.text = worker.name

            post_element = ET.SubElement(worker_element, "post")
            post_element.text = worker.post

            year_element = ET.SubElement(worker_element, "year")
            year_element.text = str(worker.year)

            root.append(worker_element)

        tree = ET.ElementTree(root)
        with open(filename, "wb") as fout:
            tree.write(fout, encoding="utf8", xml_declaration=True)


if __name__ == "__main__":
    # Выполнить настройку логгера.
    logging.basicConfig(filename="workers4.log", level=logging.INFO)

    # Список работников.
    staff = Staff()

    # Организовать бесконечный цикл запроса команд.
    while True:
        try:
            # Запросить команду из терминала.
            command = input(">>> ").lower()

            # Выполнить действие в соответствие с командой.
            if command == "exit":
                break

            elif command == "add":
                # Запросить данные о работнике.
                name = input("Фамилия и инициалы? ")
                post = input("Должность? ")
                year = int(input("Год поступления? "))

                # Добавить работника.
                staff.add(name, post, year)
                logging.info(f"Добавлен сотрудник: {name}, {post}, "
f"поступивший в {year} году.")

            elif command == "list":
                # Вывести список.
                print(staff)
                logging.info("Отображен список сотрудников.")
```

```
            elif command.startswith("select "):
                # Разбить команду на части для выделения номера года.
                parts = command.split(maxsplit=1)
                # Запросить работников.
                selected = staff.select(int(parts[1]))

                # Вывести результаты запроса.
                if selected:
                    for idx, worker in enumerate(selected, 1):
                        print("{:>4}: {}".format(idx, worker.name))

                    logging.info(f"Найдено {len(selected)} работников со "
f"стажем более {parts[1]} лет.")

                else:
                    print("Работники с заданным стажем не найдены.")
                    logging.warning(f"Работники со стажем более {parts[1]}
лет не найдены.")

            elif command.startswith("load "):
                # Разбить команду на части для имени файла.
                parts = command.split(maxsplit=1)
                # Загрузить данные из файла.
                staff.load(parts[1])
                logging.info(f"Загружены данные из файла {parts[1]}.")

            elif command.startswith("save "):
                # Разбить команду на части для имени файла.
                parts = command.split(maxsplit=1)
                # Сохранить данные в файл.
                staff.save(parts[1])
                logging.info(f"Сохранены данные в файл {parts[1]}.")

            elif command == "help":
                # Вывести справку о работе с программой.
                print("Список команд:\n")
                print("add - добавить работника;")
                print("list - вывести список работников;")
                print("select <стаж> - запросить работников со стажем;")
                print("load <имя_файла> - загрузить данные из файла;")
                print("save <имя_файла> - сохранить данные в файл;")
                print("help - отобразить справку;")
                print("exit - завершить работу с программой.")

            else:
                raise UnknownCommandError(command)

        except Exception as exc:
            logging.error(f"Ошибка: {exc}")
            print(exc, file=sys.stderr)
```

В данном коде была проведена проверка типов с помощью утилиты «mypy» (рис. 1).

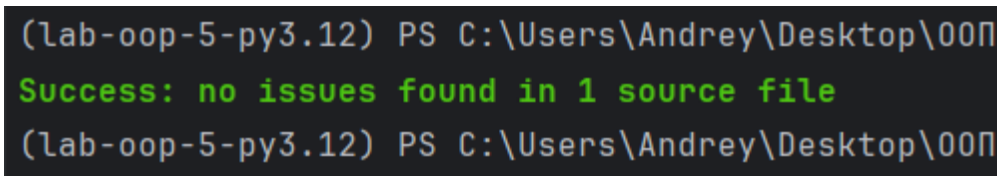

Рисунок 1 – Успешная проверка на соответствие типов

Выполнение индивидуальных заданий:

Вариант 25

Задание 1.

Необходимо выполнить индивидуальное задание 2 лабораторной работы 2.19, но с использованием аннотации типов и проверить программу на соответствие типов с помощью утилиты «mypy». Код программы лабораторной работы 2.19 с использованием аннотации типов, в том числе и для списков сложной структуры:

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

import json
import logging
import os
import sys
from typing import List, Dict


class UnknownCommandError(Exception):
    """
    Класс пользовательского исключения в случае,
    если введенная команда является недопустимой.
    Сообщения об этой ошибке записываются в журнал (лог-файл) "trains.log".
    """

    def __init__(self, command: str, message: str = "Неизвестная команда") ->
None:
        """
        Конструктор класса пользовательского исключения.
        При создании экземпляра исключения возможна запись в лог
        для сохранения информации об ошибочной команде.

        :param command: Неверная команда;
        :param message: Сообщение об ошибке.
        """

        self.command = command
        self.message = message
        super().__init__(message)

    def __str__(self) -> str:
        """
        Метод вывода сообщения об ошибке (магический метод).

        :return: Неверная команда и сообщение об ошибке.
        """
```

```
        return f"{self.command} -> {self.message}"


class TrainManager:
    """
    Класс для управления списком поездов. Предоставляет методы для:
    - добавления поезда (add_train),
    - отображения всех поездов (list_trains),
    - выборки поездов по пункту назначения (select_trains),
    - загрузки поездов из JSON (load_from_json),
    - сохранения поездов в JSON (save_to_json).

    Все основные действия (добавление, загрузка, сохранение) сопровождаются
записью
    в журнал (лог-файл) "trains.log".
    """

    def __init__(self) -> None:
        """
        Инициализировать пустой список поездов.
        """
        self.trains: List[Dict[str, str]] = []

    def add_train(
        self,
        departure_point: str,
        number_train: str,
        time_departure: str,
        destination: str
    ) -> None:
        """
        Добавить информацию о поезде в список.
        Добавленный поезд — это словарь  такой структурой:
        {
          "departure_point": <пункт отправления>,
          "number_train": <номер поезда>,
          "time_departure": <время отправления>,
          "destination": <пункт назначения>
        }
        После добавления список поездов упорядочивается по времени
отправления.

        :param departure_point: Пункт отправления;
        :param number_train: Номер поезда;
        :param time_departure: Время отправления;
        :param destination: Пункт назначения.

        После успешного добавления поезда информация вносится в лог-файл.
        """

        self.trains.append(
            {
                "departure_point": departure_point,
                "number_train": number_train,
                "time_departure": time_departure,
                "destination": destination,
            }
        )
        self.trains.sort(key=lambda train: train["time_departure"])
        logging.info(
            f"Добавлен поезд: пункт отправления={departure_point}, "
            f"№={number_train}, время={time_departure}, "
            f"пункт назначения={destination}"
```

```
        )

    def list_trains(self) -> List[Dict[str, str]]:
        """
        Вернуть текущий список поездов.

        :return: Список поездов.
        """

        return self.trains

    def select_trains(self, point_user: str) -> List[Dict[str, str]]:
        """
        Выбрать поезда, пункт назначения которых совпадает с point_user.
        Результат выборки логгируется (указывается, сколько найдено поездов).

        :param point_user: Пункт назначения (введенный пользователем).
        :return: Список таких поездов (может быть пустым).
        """

        point_user = point_user.lower()
        selected = [train for train in self.trains if
train["destination"].lower() == point_user]
        logging.info(
            f"Выполнен поиск поездов по пункту назначения='{point_user}'. "
            f"Найдено {len(selected)} поезд(а)."
        )
        return selected

    def load_from_json(self, filename: str) -> None:
        """
        Загрузить список поездов из указанного файла в формате JSON.
        Если файл отсутствует список остаётся пустым или прежним.
        При успешной загрузке записывается соответствующее сообщение в лог.
        Если файл не существует, в лог также добавляется предупреждение.

        :param filename: Имя файла для загрузки.
        """

        if os.path.exists(filename):
            with open(filename, "r", encoding="utf-8") as f:
                self.trains = json.load(f)
            logging.info(f"Данные успешно загружены из файла: {filename}.")
        else:
            logging.warning(f"Файл {filename} не найден. Загрузка не
выполнена.")

    def save_to_json(self, filename: str) -> None:
        """
        Сохранить текущий список поездов в указанный файл в формате JSON.
        Если файл отсутствует, создается новый с таким же именем.
        При успешном сохранении выполняется запись в лог-файл.

        :param filename: Имя файла для сохранения.
        """

        if not filename.endswith(".json"):
            filename += ".json"

        with open(filename, "w", encoding="utf-8") as f:
            json.dump(self.trains, f, ensure_ascii=False, indent=4)

        logging.info(f"Данные сохранены в файл: {filename}.")
```

```
def print_trains(trains: List[Dict[str, str]]) -> None:
    """
    Напечатать таблицу поездов в табличном формате.
    Если список пуст, выводится сообщение о пустом списке.

    :param trains: Список поездов.
    """

    if not trains:
        print("Список поездов пуст или ничего не найдено.")
        return

    line = "+-{}-+-{}-+-{}-+-{}-+-{}-+".format("-" * 4, "-" * 20, "-" * 13,
"-" * 18, "-" * 20)
    print(line)
    print(
        "| {:^4} | {:^20} | {:^13} | {:^18} | {:^20} |".format(
            "№", "Пункт отправления", "№ поезда", "Время отправления", "Пункт
назначения"
        )
    )
    print(line)
    for idx, train in enumerate(trains, 1):
        print(
            "| {:>4} | {:<20} | {:<13} | {:>18} | {:<20} |".format(
                idx, train["departure_point"], train["number_train"],
train["time_departure"], train["destination"]
            )
        )
        print(line)


def main() -> None:
    """
    Главная функция, организующая цикл взаимодействия с пользователем.
    Также решается проблема `теневых имен` (одинаковые имена в основном коде
и методах).
    Все введённые пользователем команды, а также возникшие ошибки,
    регистрируются в лог-файле "trains.log".
    """

    logging.basicConfig(
        filename="trains.log",
        level=logging.INFO,
        format="%(asctime)s [%(levelname)s] %(message)s",
    )

    manager = TrainManager()
    logging.info("Программа запущена.")

    while True:
        try:
            command = input(">>> ").strip().lower()
            logging.info(f"Введена команда: '{command}'")

            if command == "exit":
                logging.info("Программа завершена по команде 'exit'.")
                print("Программа завершена.")
                break

            elif command == "add":
                departure_point = input("Пункт отправления? ")
                number_train = input("Номер поезда? ")
```

```
                time_departure = input("Время отправления? ")
                destination = input("Пункт назначения? ")
                manager.add_train(departure_point, number_train,
time_departure, destination)
                print("Поезд добавлен.")

            elif command == "list":
                trains_list = manager.list_trains()
                print_trains(trains_list)
                logging.info(f"Выведен список из {len(trains_list)}
поезд(ов).")

            elif command.startswith("select "):
                parts = command.split(maxsplit=1)
                point_user = parts[1]
                selected = manager.select_trains(point_user)
                print_trains(selected)

            elif command.startswith("load "):
                parts = command.split(maxsplit=1)
                filename = parts[1]
                manager.load_from_json(filename)
                print(f"Данные загружены из файла {filename}.")

            elif command.startswith("save "):
                parts = command.split(maxsplit=1)
                filename = parts[1]
                manager.save_to_json(filename)
                print(f"Данные сохранены в файл {filename}.")

            elif command == "help":
                print("Список доступных команд:")
                print("add  - добавить поезд;")
                print("list - вывести список всех поездов;")
                print("select <пункт_назначения> - вывести поезда по пункту
назначения;")
                print("load <имя_файла> - загрузить данные из файла JSON;")
                print("save <имя_файла> - сохранить данные в файл JSON;")
                print("help - показать справку;")
                print("exit - завершить работу.")

            else:
                raise UnknownCommandError(command)

        except Exception as exc:
            logging.error(f"Произошла ошибка: {exc}")
            print(f"Ошибка: {exc}", file=sys.stderr)


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

Данный код был проверен на соответствие типов с помощью утилиты «mypy» (рис. 2).

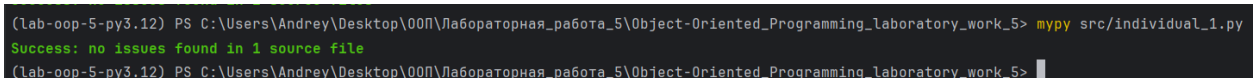

Рисунок 2 – Успешная проверка программы утилитой «mypy»

Ссылка на репозиторий данной лабораторной работы:

https://github.com/AndreyPust/Object-Oriented_Programming_laboratory_work_5.git

Ответы на контрольные вопросы:

1. Для чего нужны аннотации типов в языке Python?

Аннотации типов помогают сделать код более понятным, читаемым и проверяемым. Они позволяют разработчикам указывать, какого типа данные ожидаются для переменных, параметров и возвращаемых значений функций. Это упрощает чтение кода, облегчает отладку и работу над проектом в команде, а также позволяет инструментам статического анализа (например, mypy) выявлять возможные ошибки типов до запуска программы.

2. Как осуществляется контроль типов языке Python?

Python — это язык с динамической типизацией, где типы данных определяются и проверяются во время выполнения программы. Аннотации типов в Python не влияют на исполнение кода и не заставляют язык следить за соответствием типов. Однако контроль типов можно выполнять с помощью сторонних инструментов, таких как mypy, Pyright, Pylint и других анализаторов, которые проверяют соответствие типов с учетом аннотаций и помогают выявить ошибки.

3. Какие существуют предложения по усовершенствованию Python для работы с аннотациями типов?

Python постоянно развивается, и сообщество активно предлагает улучшения для работы с типами. Несколько важных предложений:

– PEP 484 — Введение стандартных аннотаций типов и базовых коллекций (например, List, Dict).

– PEP 585 — Позволяет использовать встроенные типы коллекций (например, list[int], dict[str, int]), начиная с Python 3.9.

– PEP 563 — Отложенная аннотация типов с использованием строк для улучшения производительности.

– PEP 563 и PEP 649 — Внесли изменения в правила вычисления аннотаций, позволяя указывать типы в виде строк для их отложенной интерпретации.

– PEP 544 — Протоколы, которые обеспечивают поддержку структурной типизации в Python.

– PEP 604 — Введение операторов объединения типов (int | str для обозначения Union).

4. Как осуществляется аннотирование параметров и возвращаемых значений функций?

Для аннотирования параметров и возвращаемых значений функции используют следующую синтаксис:

```
def функция(параметр: Тип) -> ТипВозвращаемогоЗначения:
# Тело функции
pass
```

5. Как выполнить доступ к аннотациям функций?

Доступ к аннотациям функции можно получить через специальное свойство «annotations», которое содержит аннотации параметров и возвращаемого значения функции в виде словаря.

6. Как осуществляется аннотирование переменных в языке Python?

Для аннотирования переменных в Python используют двоеточие (:) после имени переменной и указывают тип данных. Python не заставляет придерживаться указанных типов, но аннотации делают код более понятным и позволяют инструментам проверки типов выполнять анализ типов.

7. Для чего нужна отложенная аннотация в языке Python?

Отложенная аннотация (введенная в PEP 563) позволяет использовать строки для указания типов в аннотациях, чтобы отложить их интерпретацию до момента, когда они будут реально использоваться. Это полезно для улучшения производительности, особенно когда типы зависят от других модулей, которые еще не загружены. Она также решает проблемы с циклическими зависимостями типов.

Вывод: в ходе выполнения лабораторной работы были приобретены навыки по работе с аннотациями типов при написании программ с помощью языка программирования Python версии 3.x. Был рассмотрен вопрос контроля типов переменных и функций с использованием комментариев и аннотаций. Было изучено описание PEP-ов, регламентирующих работу с аннотациями, и представлены примеры работы с инструментом «mypy» для анализа Python кода.