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:
Мотовилов Вадим Борисович
3 курс, группа ИВТ-б-о-22-1,
09.03.01 «Информатика и вычислительная техника»,
направленность (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем», очная форма обучения

_____
(подпись)

Проверил:
Воронкин Р.А

_____
(подпись)

Отчет защищен с оценкой _____ Дата защиты_____

Ставрополь, 2024 г.

Тема: Аннотация типов

Цель: приобретение навыков по работе с аннотациями типов при написании программ с помощью языка программирования Python версии 3.x. Рассмотрен вопрос контроля типов переменных и функций с использованием комментариев и аннотаций. Приведено описание PEP-ов, регламентирующих работу с аннотациями, и представлены примеры работы с инструментом mypy для анализа Python кода.

Порядок выполнения работы:

1. Создал новый репозиторий, клонировал его, в нем создал ветку developer и перешел на нее. Ссылка на гит: https://github.com/AkselSukub/OOP_5
2. Выполнил индивидуальное задание №1:

> Выполнить индивидуальное задание 2 лабораторной работы 2.19, добавив аннтотации типов.
> Выполнить проверку программы с помощью утилиты *mypy*.

Код индивидуального задания №1:

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

import argparse
import bisect
import sys
from io import StringIO
from pathlib import Path

from colorama import Fore, Style

class TreeNode:
    def __init__(self, state: Path) -> None:
        self.state = state
        self.children: list["TreeNode"] = []

    def add_child(self, child: "TreeNode") -> None:
        bisect.insort(self.children, child)

    def __repr__(self) -> str:
        return f"<{self.state}>"

    def __len__(self) -> int:
        return len(self.children)

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, TreeNode):
            return NotImplemented
        return self.state == other.state and self.children == other.children
```

```
    def __lt__(self, other: "TreeNode") -> bool:
        return len(self) < len(other)

class Tree:
    def __init__(self, directory: Path, args: argparse.Namespace) -> None:
        self.directory = directory
        self.args = args
        self.root = TreeNode(directory)
        self.dir_count = 0
        self.file_count = 0
        self.full = False
        self.generate_tree(self.root, 0)

    def expand(self, node: TreeNode) -> None:
        try:
            for child in node.state.iterdir():
                if not self.__should_include(child):
                    continue
                self.__increment_counts(child)
                if self.full:
                    break
                node.add_child(TreeNode(child))
        except PermissionError:
            pass

    def generate_tree(self, node: TreeNode, level: int) -> None:
        if self.full or level == self.args.max_depth:
            return
        self.expand(node)
        for child in node.children:
            if child.state.is_dir():
                self.generate_tree(child, level + 1)

    def __should_include(self, child: Path) -> bool:
        if not self.args.a and child.name.startswith((".", "__")):
            return False
        if self.args.d and child.is_file():
            return False
        return True

    def __increment_counts(self, child: Path) -> None:
        if child.is_dir():
            self.dir_count += 1
        else:
            self.file_count += 1
        if self.dir_count + self.file_count >= 200:
            self.full = True

    def __format_tree(self, node: TreeNode, branch: str = "") -> str:
        result = StringIO()
        for i, child in enumerate(node.children):
            item = f"{branch}{'└── ' if i == len(node.children) - 1 else '├── '}"
            name = (
                child.state.name
                if not self.args.f
                else child.state.relative_to(self.directory)
            )
            color = Fore.GREEN if child.state.is_file() else Fore.YELLOW
            result.write(f"{item}{color}{name}{Style.RESET_ALL}\n")
            new_branch = f"{branch}{'    ' if i == len(node.children) - 1 else '│   '}"
            result.write(self.__format_tree(child, new_branch))
        return result.getvalue()
```

```
    def __str__(self) -> str:
        header = f"{Fore.BLUE}{self.root.state.name}{Style.RESET_ALL}\n"
        body = self.__format_tree(self.root)
        footer = f"\n{Fore.YELLOW}Directories: {self.dir_count}, "
        footer += f"{Fore.GREEN}Files: {self.file_count}{Style.RESET_ALL}"
        if self.full:
            footer += f"{Fore.RED}\nOutput limited to 200 elements.{Style.RESET_ALL}"
        return header + body + footer

def main(command_line: list[str] | None = None) -> None:
    """
    Главная функция программы.
    """
    parser = argparse.ArgumentParser()
    parser.add_argument("-a", action="store_true", help="All files are printed.")
    parser.add_argument("-d", action="store_true", help="Print directories only.")
    parser.add_argument("-f", action="store_true", help="Print relative path.")
    parser.add_argument(
        "-m",
        "--max_depth",
        type=int,
        default=None,
        help="Max depth of directories.",
    )
    parser.add_argument(
        "-i",
        action="store_true",
        help="Tree does not print the indentation lines."
        " Useful when used in conjunction with the -f option.",
    )
    parser.add_argument("directory", nargs="?", default=".", help="Directory to scan.")
    args = parser.parse_args(command_line)

    try:
        directory = Path(args.directory).resolve(strict=True)
    except FileNotFoundError:
        print(f"Directory '{Path(args.directory).resolve()}' does not exist.")
        sys.exit(1)

    tree = Tree(directory, args)
    print(tree)

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

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 усовершествованию 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 работы были приобретены навыки по работе с аннотациями типов при написании программ с помощью языка программирования Python версии 3.x. Рассмотрен вопрос контроля типов переменных и функций с использованием комментариев и аннотаций. Приведено описание PEP-ов, регламентирующих работу с аннотациями, и представлены примеры работы с инструментом mypy для анализа Python кода.