

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 ____

Выполнил:
Епифанов Алексей Александрович
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 Роман Александрович

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

Ставрополь, 2024 г.

Тема: Аннотация типов

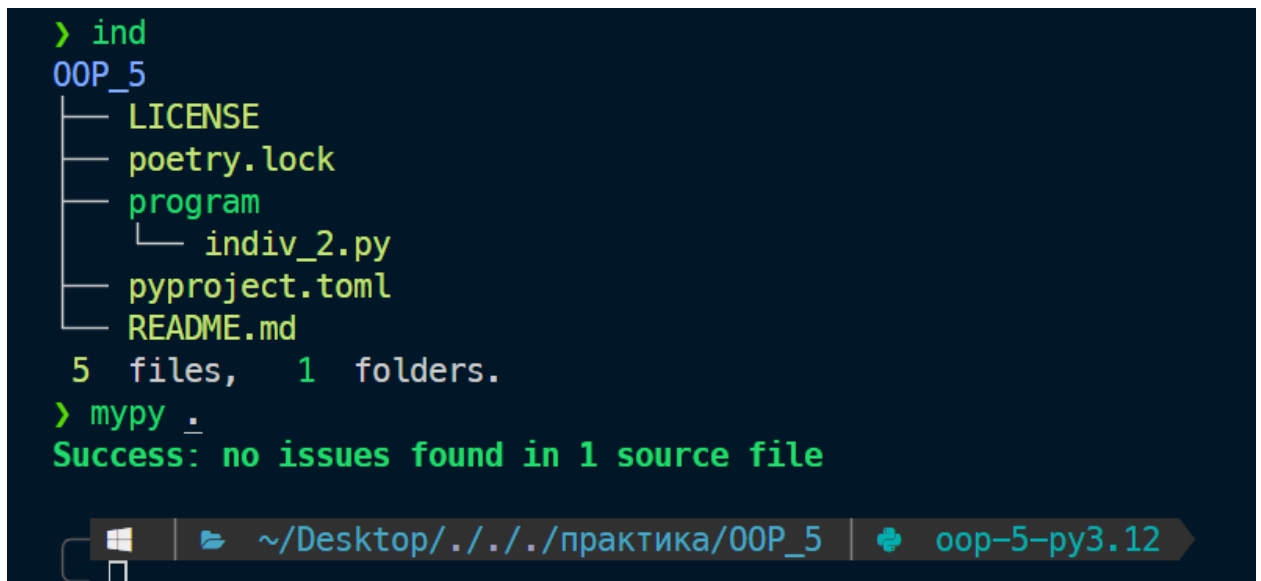
Цель: приобретение навыков по работе с аннотациями типов при написании программ с помощью языка программирования Python версии 3.x. Рассмотрен вопрос контроля типов переменных и функций с использованием комментариев и аннотаций. Приведено описание PEP'ов, регламентирующих работу с аннотациями, и представлены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

Порядок выполнения работы:

1. Создал новый репозиторий, клонировал его, в нем создал ветку developer и перешел на нее.

Ссылка на гитхаб: https://github.com/alexeiepip/OOP_5.git

2. Выполнил индивидуальное задание вариант 9: Выполнить индивидуальное задание 2 лабораторной работы 2.19, добавив аннотации типов. Выполнить проверку программы с помощью утилиты муру.



```
> ind
OOP_5
├── LICENSE
├── poetry.lock
├── program
│   └── indiv_2.py
├── pyproject.toml
└── README.md
5 files, 1 folders.
> mypy .
Success: no issues found in 1 source file
```

~/Desktop/./././практика/OOP_5 | oop-5-py3.12

Рисунок 1. Результат работы программы индивидуального задания

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

```
import argparse
import sys
from pathlib import Path
from typing import Any

from colorama import Fore, Style
```

```

def print_tree(
    tree: dict[Path, Any], lines: bool, level: int = 0, levels: list[bool] = []
) -> None:
    """
    Отрисовка дерева каталогов в виде иерархической структуры.
    """
    if not tree:
        return

    for i, (node, child) in enumerate(tree.items()):
        if i == len(tree) - 1 and level != 0:
            levels[level - 1] = False
        if not lines:
            branch = "".join(" | " if lev else " " for lev in levels[:level])
            branch += "└─ " if i == len(tree) - 1 else "├─ "
        else:
            branch = ""
        if level == 0:
            # Синий цвет для корневой папки
            print(Fore.BLUE + str(node) + Style.RESET_ALL)
        else:
            # Для файлов: зеленый цвет, для папок: желтый цвет
            color = Fore.GREEN if child is not None else Fore.YELLOW
            print(branch + color + str(node) + Style.RESET_ALL)

        print_tree(child, lines, level + 1, levels + [True])

```

```

def tree(directory: Path, args: argparse.Namespace) -> None:
    """
    Создание структуры дерева каталогов в виде словаря.
    """

```

```

    sw = False
    files = 0
    folders = 0
    folder_tree: dict[Path, Any] = {}
    count = 0

    path_list: list[Path] = []
    all_files = args.a
    max_depth = args.max_depth
    only_dir = args.d
    counter = 0
    for path in directory.rglob("*"):
        try:
            if counter < 100000:
                counter += 1
            else:
                sw = True
                break
            if len(path_list) >= 1000:
                break
            if (
                max_depth
                and len(path.parts) - len(directory.parts) > max_depth
            ):
                continue
            if only_dir and not path.is_dir():
                continue
            if (not all_files) and (
                any(part.startswith(".") for part in path.parts)
            ):

```

```

        continue
    path_list.append(path)
except PermissionError:
    pass
path_list.sort()

for path in path_list:
    count += 1
    relative_path = path.relative_to(directory)
    parts = relative_path.parts

    if args.f:
        path_work = relative_path
    else:
        path_work = Path(relative_path.name)
    current_level = folder_tree
    p = Path()
    for part in parts[:-1]:
        if args.f:
            p = p / part
        else:
            p = Path(part)
        current_level = current_level[p]

    if path.is_dir():
        current_level[path_work] = current_level.get(path_work, {})
        folders += 1
    else:
        current_level[path_work] = None
        files += 1
    if folders + files >= 1000:
        sw = True
        break

print_tree({Path(directory.name): folder_tree}, args.i)

if sw:
    if folders + files < 1000:
        str_1 = "Вывод ограничен временем"
    else:
        str_1 = "Вывод ограничен по длине: 1000 элементов"
    print(Fore.RED, str_1, Style.RESET_ALL)
print(
    Fore.YELLOW,
    files,
    Style.RESET_ALL,
    "files, ",
    Fore.GREEN,
    folders,
    Style.RESET_ALL,
    "folders.",
)

```

```

def main(command_line: 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(directory, args)

if __name__ == "__main__":
    main()

```

Ответы на контрольные вопросы:

1. Для чего нужны аннотации типов в языке Python?

Для повышения информативности исходного кода и возможности его анализа с помощью сторонних инструментов важно уделить внимание контролю типов переменных. Одной из наиболее актуальных тем в этом контексте является именно типизация. Несмотря на то что Python является языком с динамической типизацией, время от времени возникает необходимость в контроле типов, что может значительно улучшить читаемость и надежность кода.

2. Как осуществляется контроль типов в языке Python?

Для статического анализа типов используются аннотации (type hints) и инструменты, такие как `mypy`. Они позволяют проверять соответствие типов до запуска программы, но сами по себе не влияют на выполнение кода.

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

Использование отложенных аннотаций, использование инструментов для анализа.

4. Как осуществляется аннотирование параметров и возвращаемых значений функций?

имя_аргумента: аннотация, def имя_функции() -> тип.

5. Как выполнить доступ к аннотациям функций?

Доступ к использованным в функции аннотациям можно получить через атрибут `__annotations__`.

6. Как осуществляется аннотирование переменных в языке Python?

`var = value # type: annotation`

`var: annotation; var = value`

`var: annotation = value`

7. Для чего нужна отложенная аннотация в языке Python?

PEP 563 — Postponed Evaluation of Annotations. Данный PEP вступил в силу с выходом Python 3.7. У подхода работы с аннотация до этого PEP'а был ряд проблем, связанных с тем, что определение типов переменных (в функциях, классах и т.п.) происходит во время импорта модуля, и может сложиться такая ситуация, что тип переменной объявлен, но информации об этом типе ещё нет, в таком случае тип указывают в виде строки – в кавычках. В PEP 563 предлагается использовать отложенную обработку аннотаций, это позволяет определять переменные до получения информации об их типах и ускоряет выполнение программы, т.к. при загрузке модулей не будет тратиться время на проверку типов – это будет сделано перед работой с переменными.

Вывод: в ходе выполнения данной работы были приобретены навыки по работе с аннотациями типов при написании программ с помощью языка программирования Python версии 3.x.