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 №4.5**

**дисциплины «Объектно-ориентированное программирование»**

**Вариант 9**

|  |  |  |  |
| --- | --- | --- | --- |
|  | | Выполнил:  Иващенко Олег Андреевич  3 курс, группа ИВТ-б-о-22-1,  09.03.02 «Информационные и вычислительные машины», направленность (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | |
|  | | Руководитель практики:  Воронкин Роман Александрович, доцент департамента цифровых, робототехнических систем и электроники  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (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:

Индивидуальное задание. Выполнить индивидуальное задание 2 лабораторной работы 2.19, добавив аннотации типов. Выполнить проверку программы с помощью утилиты mypy.

Индивидуальное задание 2 в лабораторной работе 2.19 представляло собой написание консольной программы, которая при указании пути выводит список всех документов и каталогов на указанной глубине (уровне).

В программе используется единственный метод *tree*, который принимает параметры:

* path – str, указание пути к директории;
* level – int, уровень глубины для вывода;
* max\_levels – int, максимальный уровень глубины;
* show\_hidden – bool, флаг для отображения скрытых файлов.

Данный метод не возвращает ничего, вывод осуществляется прям в нём. Наша задача – сделать аннотацию типов для принимаемых аргументов. В методе *main* имеется строка, которая вызывает метод tree с указанием всех параметров, установленные при запуске программы в ключах. Код программа приведён в листинге 1.

Листинг 1 – Код программы индивидуального задания 1 (individual.py)

|  |
| --- |
| #!/usr/bin/env python3  # -\*- coding: utf-8 -\*-  """  Выполнить индивидуальное задание 2 лабораторной работы 2.19, добавив аннотации  типов. Выполнить проверку программы с помощью утилиты mypy. Индивидуальное  задание 2 в лабораторной работе 2.19 представляло собой написание консольной  программы, которая при указании пути выводит список всех документов и  каталогов на указанной глубине (уровне).  """  import os  import argparse  from typing import Optional  def tree(path: str, level: int, max\_levels: int, show\_hidden: bool) -> None:  """  Вывод списка каталогов и файлов по указанному пути,  аналогично утилите tree в ОС Linux.  Аргументы:  path (str): Путь к директории.  level (int): Уровень глубины для вывода.  max\_levels (int): Максимальный уровень глубины.  show\_hidden (bool): Флаг для отображения скрытых файлов.  """  if level > max\_levels:  return  for element in os.listdir(path):  if not show\_hidden and element.startswith('.'):  continue  dir\_path = os.path.join(path, element)  if os.path.isdir(dir\_path):  print(' ' \* level + f'/{element}')  tree(dir\_path, level + 1, max\_levels, show\_hidden)  else:  print(' ' \* level + element)  def main() -> None:  """  Основная функция для обработки аргументов командной строки  и вызова функции tree.  """  parser = argparse.ArgumentParser()  parser.add\_argument(  'directory',  nargs='?',  default='.',  help="Директория"  )  parser.add\_argument(  '-l',  '--level',  type=int,  default=float('inf'),  help="Максимальный уровень глубины"  )  parser.add\_argument(  '-a',  '--all',  action='store\_true',  help="Вывод скрытых файлов"  )  parser.add\_argument(  '--author',  nargs='?',  help="Вывод автора программы"  )  args = parser.parse\_args()  if args.author:  print("> Автор работы: Иващенко О.А.\n")  return  path = os.path.abspath(args.directory)  if not os.path.exists(path):  print("Указанного каталога не существует")  return  if not os.path.isdir(path):  print(f"Ошибка: {path} - не каталог")  return  print(f'Список файлов в каталоге {path}')  tree(path, 0, args.level, args.all)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Для проверки правильности вводимых типов данных будет использоваться утилита *mypy*, установленная в окружение. Попробуем выполнить программу при правильном указании всех данных (рисунок 1).

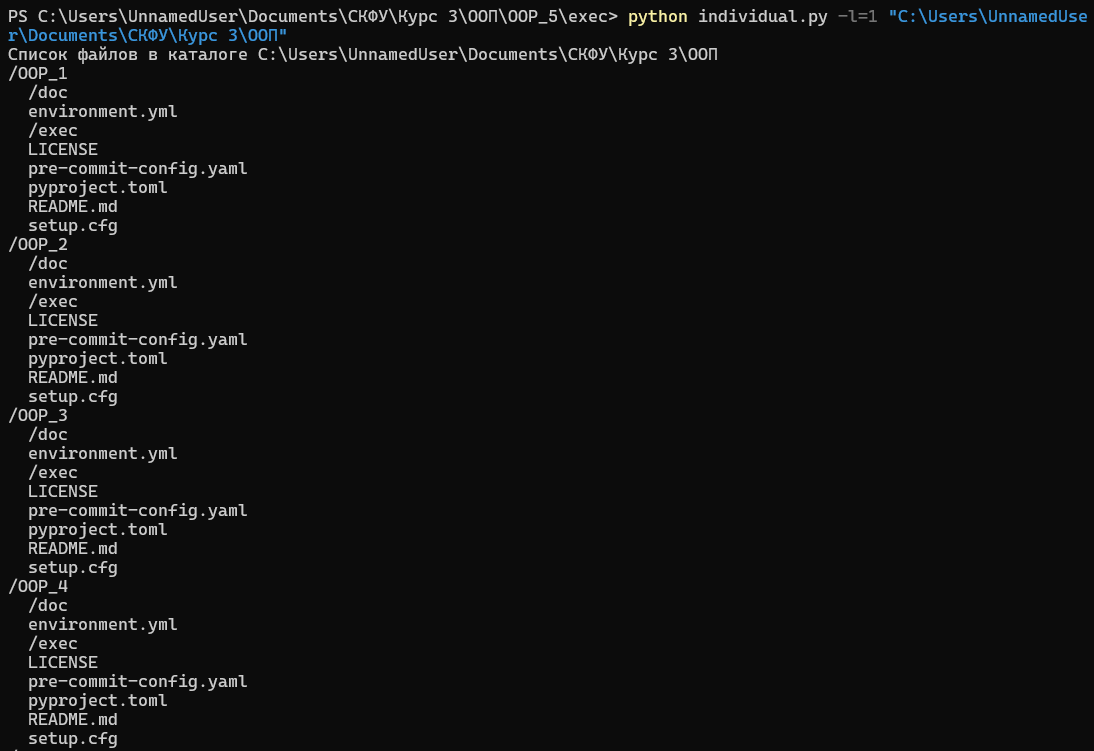


Рисунок 1 – Результат выполнения программы

Как видно из рисунка 1, программа успешно выполняется. Теперь попробуем проверить эту программу при помощи утилиты *mypy*. В командной строке Anaconda Powershell Prompt, запущенной из каталога с программой, вводим команду *mypy <имя\_файла\_программы>*, после чего видим запись об успешном завершении проверки (рисунок 2).



Рисунок 2 – Успешное завершение проверки

Теперь изменим строку метода *main*, в которой вызывается метод, намеренно введя неправильные типы данных в качестве аргументов метода *tree*. Результат проверки *mypy* представлен на рисунке 3.

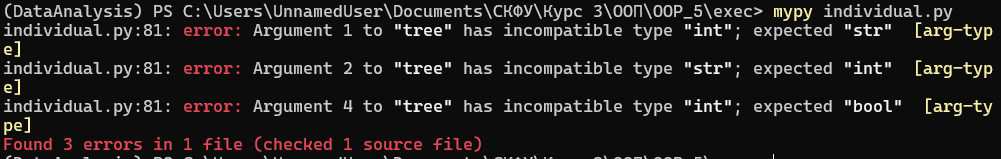


Рисунок 3 – Завершение проверки с ошибками

Ответы на контрольные вопросы:

1. Для чего нужны аннотации типов в языке Python?

Аннотации типов нужны для повышения информативности исходного кода, а также для получения возможности с помощью сторонних инструментов производить его анализ. Одной из наиболее востребованных является контроль типов переменных. Несмотря на то, что Python – язык с динамической типизацией, иногда возникает необходимость в контроле типов.

1. Как осуществляется контроль типов в языке Python?

Во-первых, указать в комментарии об определённом типе данных для условной переменной; во-вторых, использовать специальный инструмент, который выполнит соответствующую проверку (таким инструментом является mypy).

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

PEP 3107 – Function Annotations. В нём описывается синтаксис использования аннотаций в функциях Python. Важным является то, что аннотации не имеют никакого семантического значения для интерпретатора Python и предназначены только для анализа сторонними приложениями. Аннотировать можно аргументы функции и возвращаемое ей значение.

PEP 484 – Type Hints. В нём представлены рекомендации по использованию аннотаций типов. Аннотация типов упрощает статический анализ кода, рефакторинг, контроль типов в рантайме и кодогенерацию, использующую информацию о типах. В рамках данного документа, определены следующие варианты работы с аннотациями: использование аннотаций в функциях согласно PEP 3107, аннотация типов переменных через комментарии в формате *# type: type\_name* и использование stub-файлов.

PEP 526 – Syntax for Variable Annotations. Приводится описание синтаксиса для аннотации типов переменных (базируется на PEP 484), использующего языковые конструкции, встроенные в Python.

PEP 563 – Postponed Evaluation of Annotations. Данный PEP вступил в силу с выходом Python 3.7. У подхода работы с аннотациями 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 сложится такая ситуация, что тип переменной объявлен, но информации об этом типе ещё нет, в таком случае указываются в виде строки – в кавычках. В PEP 563 предлагается использовать отложенную обработку аннотаций, это позволяет определять переменные для получения информации об их типах и ускоряет выполнение программа, т.к. при загрузке модулей не будет тратиться время на проверку типов - это будет сделано перед работой с переменными.

1. Как осуществляется аннотирование параметров и возвращаемых значений функций?

В функциях мы можем аннотировать аргументы и возвращаемое значение. Выглядеть это может так:

*def repeater(s: str, n: int) -> str:*

*return s \* n*

Аннотация для аргумента определяется через двоеточие после его имени.

*имя\_аргумента: аннотация*

Аннотация, определяющая тип возвращаемого функцией значения, указывается после её имени с использованием символов “->”.

*def имя\_функции() -> тип*

Для лямбд аннотации не поддерживаются.

1. Как выполнить доступ к аннотациям функций?

Доступ к использованным в функции аннотациям можно получить через атрибут \_\_annotations\_\_, в котором аннотации представлены в виде словаря, где ключами являются атрибуты, а значениями – аннотации. Возвращаемое функцией значение хранится в записи с ключом return.

Содержимое *repeater.\_\_annotations\_\_*:

*{‘n’: int, ‘return’: str, ‘s’: str}*

1. Как осуществляется аннотирование переменных в языке Python?

Можно использовать один из трёх способов создания аннотированных переменных:

*var = value # type: annotation*

*var: annotation; var = value*

*var: annotation = value*

Пример:

*name = “John” # type: str*

*name: str; name = “John”*

*name: str = “John”*

1. Для чего нужна отложенная аннотация в языке Python?

До выхода Python 3.7 определение типов в аннотациях происходило во время импорта модуля, что приводило к проблеме. Например, если выполнить следующий код:

*class Rectangle:*

*def \_\_init\_\_(self, height: int, width: int, color: Color) -> None:*

*self.height = height*

*self.width = width*

*self.color = color*

То возникает ошибка *NameError: name ‘Color’ is not defined*. Она связана с тем, что переменная color имеет тип Color, который пока ещё не объявлен.

В таком случае мы можем указать тип Color в кавычка, но это будет не удобно:

*def \_\_init\_\_(self, height: int, width: int, color: “Color”) -> None:*

Эту проблему можно решить, воспользовавшись отложенной обработкой аннотаций из Python 3.7:

*from \_\_future\_\_ inport annotations*

*class Rectangle:*

*def \_\_init(self, height: int, width: int, colot: Colot) -> None:*

*self.height = height*

*self.width = width*

*self.color = color*

*class Color:*

*def \_\_init\_\_(self, r: int, g: int, b: int) -> None:*

*self.R = r*

*self.G = g*

*self.B = b*

*rect = Rectable(1, 2, Color(255, 255, 255))*

Вывод: В процессе 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, приведено описание PEP’ов, регламентирующих работу с аннотациями, и представлены примеры работы с инструментом mypy для анализа Python кода. Выполнено индивидуальное задание.

Ссылка на репозиторий GitHub:

[IUnnamedUserI/OOP\_5: Объектно-ориентированное программирование. Лабораторная работа №5](https://github.com/IUnnamedUserI/OOP_5)