МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРОКАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Кафедра инфокоммуникаций

Институт цифрового развития

ОТЧЁТ

по лабораторной работе

Дисциплина: «Объектно – ориентированное программирование»

Выполнил: студент 3 курса
группы ИВТ-б-о-21-1
Мальцев Николай Артемович

Ставрополь 2023

Аннотация типов

Цель работы: 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.

Ход работы:

Индивидуальное задание.

Выполнить индивидуальное задание 2 лабораторной работы 2.19, добавив аннотации типов.

Листинг программы:

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
from abc import ABC, abstractmethod


# Создаем абстрактный класс Pair
class Pair(ABC):
    @abstractmethod
    def __add__(self, other) -> "Pair":
        pass

    @abstractmethod
    def __sub__(self, other) -> "Pair":
        pass

    @abstractmethod
    def __mul__(self, other) -> "Pair":
        pass

    @abstractmethod
    def __truediv__(self, other) -> "Pair":
        pass

    @abstractmethod
    def __str__(self) -> str:
        pass


# Создаем класс Money, который наследуется от Pair
class Money(Pair):
    def __init__(self, amount: float):
        self.amount = amount
```

```
    def __add__(self, other: "Money") -> "Money":
        if isinstance(other, Money):
            return Money(self.amount + other.amount)
        else:
            raise TypeError("Unsupported operand type")

    def __sub__(self, other: "Money") -> "Money":
        if isinstance(other, Money):
            return Money(self.amount - other.amount)
        else:
            raise TypeError("Unsupported operand type")

    def __mul__(self, other: float) -> "Money":
        if isinstance(other, (int, float)):
            return Money(self.amount * other)
        else:
            raise TypeError("Unsupported operand type")

    def __truediv__(self, other: float) -> "Money":
        if isinstance(other, (int, float)):
            return Money(self.amount / other)
        else:
            raise TypeError("Unsupported operand type")

    def __str__(self) -> str:
        return str(self.amount)


# Создаем класс Fraction, который наследуется от Pair
class Fraction(Pair):
    def __init__(self, numerator: int, denominator: int):
        self.numerator = numerator
        self.denominator = denominator

    def __add__(self, other: "Fraction") -> "Fraction":
        if isinstance(other, Fraction):
            common_denominator = self.denominator * other.denominator
            new_numerator = (self.numerator * other.denominator) + (other.numerator
* self.denominator)
            return Fraction(new_numerator, common_denominator)
        else:
            raise TypeError("Unsupported operand type")

    def __sub__(self, other: "Fraction") -> "Fraction":
        if isinstance(other, Fraction):
            common_denominator = self.denominator * other.denominator
            new_numerator = (self.numerator * other.denominator) - (other.numerator
* self.denominator)
            return Fraction(new_numerator, common_denominator)
        else:
            raise TypeError("Unsupported operand type")

    def __mul__(self, other: float) -> "Fraction":
        if isinstance(other, (int, float)):
```

```
            return Fraction(int(self.numerator * other), self.denominator)
        else:
            raise TypeError("Unsupported operand type")

    def __truediv__(self, other: float) -> "Fraction":
        if isinstance(other, (int, float)):
            return Fraction(self.numerator, int(self.denominator * other))
        else:
            raise TypeError("Unsupported operand type")

    def __str__(self) -> str:
        return f"{self.numerator}/{self.denominator}"


# Основной код программы
if __name__ == "__main__":
    money1 = Money(100)
    money2 = Money(50)
    print(money1 + money2)
    print(money1 - money2)
    print(money1 * 2)
    print(money1 / 2)
    fraction1 = Fraction(1, 2)
    fraction2 = Fraction(3, 4)
    print(fraction1 + fraction2)
    print(fraction1 - fraction2)
    print(fraction1 * 2)
    print(fraction1 / 2)
```

Выполнить проверку программы с помощью утилиты mypy.

```
PS C:\Users\Николай Мальцев\OneDrive\Рабочий стол\ООП\Практика\Lab_4.5> pip install mypy
Requirement already satisfied: mypy in c:\users\николай мальцев\onedrive\рабочий стол\ооп\практика\lab_4.5\env\
Requirement already satisfied: typing-extensions>=4.1.0 in c:\users\николай мальцев\onedrive\рабочий стол\ооп\п
Requirement already satisfied: mypy-extensions>=1.0.0 in c:\users\николай мальцев\onedrive\рабочий стол\ооп\пра
[notice] A new release of pip available: 22.3.1 -> 23.3.2
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\Николай Мальцев\OneDrive\Рабочий стол\ООП\Практика\Lab_4.5> python -m  mypy ".\Tasks\Ind_Task_2.py"
Success: no issues found in 1 source file
PS C:\Users\Николай Мальцев\OneDrive\Рабочий стол\ООП\Практика\Lab_4.5>
```

Рисунок 1. Установка утилиты и проверка программы

1. Для чего нужны аннотации типов в языке Python?

Аннотации типов в языке Python представляют собой способ указать ожидаемый тип данных для аргументов функций, возвращаемых значений функций и переменных. Вот несколько причин, по которым аннотации типов могут быть полезны:

1. Документация: Аннотации типов могут служить документацией для кода, помогая другим разработчикам понять ожидаемые типы данных в функциях и методах.

2. Поддержка инструментов статического анализа: Аннотации типов

могут использоваться инструментами статического анализа кода, такими как Mypy, Pyre или Pyright, чтобы проверять соответствие типов данных во время компиляции или анализа кода.

3. Улучшение читаемости: Аннотации типов могут помочь улучшить читаемость кода, особенно в случае сложных или больших проектов, где явное указание типов данных может помочь понять назначение переменных и результатов функций.

4. Интеграция с IDE: Некоторые интегрированные среды разработки (IDE), такие как PyCharm, могут использовать аннотации типов для предоставления подсказок о типах данных и автоматической проверки соответствия типов.

2. Как осуществляется контроль типов в языке Python?

В языке Python контроль типов данных может осуществляться несколькими способами:

1. Аннотации типов: Как уже упоминалось, в Python можно использовать аннотации типов для указания ожидаемых типов данных для аргументов функций, возвращаемых значений функций и переменных. Это позволяет документировать ожидаемые типы данных и использовать инструменты статического анализа кода для проверки соответствия типов.

2. Использование инструментов статического анализа: Существуют сторонние инструменты, такие как Mypy, Pyre и Pyright, которые могут использоваться для статической проверки соответствия типов данных в Python-коде. Эти инструменты могут обнаруживать потенциальные ошибки типов данных и предоставлять рекомендации по улучшению кода.

3. Вручную проверять типы данных: В Python можно вручную выполнять проверку типов данных с помощью условных операторов и функций, таких как isinstance(). Например, можно написать условие для проверки типа данных перед выполнением определенной операции.

4. Использование аннотаций типов в комбинации с декораторами: В Python можно использовать декораторы, такие как @overload из модуля functools, для реализации перегрузки функций с разными типами аргументов.

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

Предложения по усовершенствованию работы с аннотациями типов в Python включают расширение поддержки аннотаций типов, улучшение интеграции с инструментами статического анализа, улучшение документации и рекомендаций, а также разработку стандартной библиотеки типов. Эти изменения могут сделать работу с аннотациями типов более мощной и удобной для разработчиков.

4. Как осуществляется аннотирование параметров и возвращаемых значений функций?

В Python аннотирование параметров и возвращаемых значений функций осуществляется с использованием двоеточия и указания типа данных после имени параметра или перед знаком "->" для возвращаемого значения. Например:

```
def greet(name: str) -> str:
    return "Hello, " + name
```

В этом примере name: str указывает, что параметр name должен быть строкой, а -> str указывает, что функция возвращает строку.

5. Как выполнить доступ к аннотациям функций?

В Python можно получить доступ к аннотациям функций с помощью специального атрибута__annotations__. Этот атрибут содержит словарь, в котором ключами являются имена параметров или "return" (для возвращаемого значения), а значениями - указанные типы данных.

Пример:

```
def greet(name: str) -> str:
    return "Hello, " + name
print(greet.__annotations__)
```

Этот код выведет на экран словарь с аннотациями функции greet:

{'name': <class 'str'>, 'return': <class 'str'>}

Таким образом, вы можете получить доступ к аннотациям функции и использовать их в своем коде, например, для проверки типов данных или для документирования функций.

6. Как осуществляется аннотирование переменных в языке Python?

В Python переменные можно аннотировать с использованием синтаксиса аннотаций типов. Это позволяет указать ожидаемый тип данных для переменной, хотя интерпретатор Python не выполняет никакой проверки типов во время выполнения.

7. Для чего нужна отложенная аннотация в языке Python?

Отложенная аннотация в Python (Delayed Evaluation Annotation) позволяет создавать аннотации типов, используя строковые литералы вместо ссылок на фактические классы. Это может быть полезно в случаях, когда требуется аннотировать типы данных, которые еще не определены или недоступны в момент написания аннотации.

Отложенные аннотации могут быть полезны при работе с циклическими зависимостями между классами или модулями, при использовании динамически загружаемых модулей или при аннотации типов в коде, который будет выполняться на разных версиях Python.