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

Классы данных в Python

Цель работы: приобретение навыков по работе с классами данных при написании программ с помощью языка программирования Python версии 3.x.

Ход работа:

Индивидуальное задание.

Выполнить индивидуальное задание лабораторной работы 4.5, использовав классы данных, а также загрузку и сохранение данных в формат XML.

Код программы:

```
from dataclasses import dataclass
import xml.etree.ElementTree as ET
from abc import ABC, abstractmethod


@dataclass
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

    def to_xml(self, element_name: str) -> str:
        root = ET.Element(element_name)
        for key, value in self.__dict__.items():
            child = ET.Element(key)
            child.text = str(value)
            root.append(child)
        return ET.tostring(root, encoding="utf-8").decode()

    @classmethod
    def from_xml(cls, xml_string: str) -> "Pair":
        root = ET.fromstring(xml_string)
        kwargs = {child.tag: child.text for child in root}
```

```
        return cls(**kwargs)


@dataclass
class Money(Pair):
    amount: float

    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)

    def to_xml(self) -> str:
        return super().to_xml("Money")

    @classmethod
    def from_xml(cls, xml_string: str) -> "Money":
        return super().from_xml(xml_string)


@dataclass
class Fraction(Pair):
    numerator: int
    denominator: int

    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}"

    def to_xml(self) -> str:
        return super().to_xml("Fraction")

    @classmethod
    def from_xml(cls, xml_string: str) -> "Fraction":
        return super().from_xml(xml_string)


if __name__ == "__main__":
    money1 = Money(100)
    money2 = Money(50)
    print(money1 + money2)
    print(money1 - money2)
    print(money1 * 2)
    print(money1 / 2)

    # Сериализация и десериализация для Money
    money_xml = money1.to_xml()
    money_from_xml = Money.from_xml(money_xml)
    print(f"Deserialized Money: {money_from_xml}")

    # Сохранение XML в файл
    with open("money.xml", "w") as money_file:
        money_file.write(money_xml)
```

```
fraction1 = Fraction(1, 2)
fraction2 = Fraction(3, 4)
print(fraction1 + fraction2)
print(fraction1 - fraction2)
print(fraction1 * 2)
print(fraction1 / 2)

# Сериализация и десериализация для Fraction
fraction_xml = fraction1.to_xml()
fraction_from_xml = Fraction.from_xml(fraction_xml)
print(f"Deserialized Fraction: {fraction_from_xml}")

# Сохранение XML в файл
with open("fraction.xml", "w") as fraction_file:
    fraction_file.write(fraction_xml)
```

Контрольные вопросы:

1. Как создать класс данных в языке Python?

В Python создание класса данных осуществляется с использованием ключевого слова class. Вот пример простого класса данных:

```
class Person:
    def__init__(self, name, age):
        self.name = name
        self.age = age


# Создание экземпляра класса
person1 = Person("Иван", 25)


# Доступ к атрибутам экземпляра класса
print(person1.name) # Выведет: Иван
print(person1.age)   # Выведет: 25
```

В этом примере мы создаем класс Person, который имеет атрибуты name и age. Метод__init является конструктором класса и используется для инициализации атрибутов при создании экземпляра класса. При создании экземпляра класса Person мы передаем значения для атрибутов name и age.

Доступ к атрибутам экземпляра класса осуществляется с использованием точки (например, person1.name).

Это только простейший пример класса данных. В Python классы могут содержать методы (функции, связанные с классом), наследование, статические методы, свойства и многое другое.

2. Какие методы по умолчанию реализует класс данных?

В Python класс данных может реализовывать несколько встроенных методов по умолчанию, которые позволяют определить специальное поведение объекта. Некоторые из этих методов включают:

1. __init__(self, ...): Конструктор класса, который вызывается при создании нового экземпляра класса.

2. __str__(self): Метод, который возвращает строковое представление объекта. Он вызывается, когда объект передается функции str() или когда объект используется в строковом контексте.

3. __repr__(self): Метод, который возвращает представление объекта, которое может быть использовано для его воссоздания. Он вызывается, когда объект передается функции repr() или когда объект используется в интерактивной оболочке Python.

4. __eq__(self, other): Метод для сравнения объектов на равенство (используется оператор ==).

5. __lt__(self, other), __le___(self, other), __gt___(self, other), __ge___(self, other): Методы для сравнения объектов (используются операторы <, <=, >, >=).

6. __hash__(self): Метод для вычисления хэш-значения объекта, используемого в словарях и множествах.

7. __getattr__(self, name),__setattr__(self, name, value): Методы для перехвата доступа к атрибутам объекта.

8. __del__(self): Метод, который вызывается при удалении объекта.

Это только небольшой набор методов по умолчанию, которые могут быть реализованы в классе данных. В Python есть еще много других "магических" методов, которые позволяют определить специальное поведение объектов.

3. Как создать неизменяемый класс данных?

В Python неизменяемый класс данных можно создать, используя неизменяемые типы данных в качестве атрибутов класса, и предоставляя только методы для чтения значений атрибутов, но не для их изменения. Вот пример создания неизменяемого класса данных:

```
class ImmutableData:
    def__init__(self, value1, value2):
```

```
        self._value1 = value1    # Префикс "_" обозначает "приватный" атрибут
        self._value2 = value2

    def get_value1(self):
        return self._value1

    def get_value2(self):
        return self._value2
```

В этом примере атрибуты value1 и value2 являются приватными (по соглашению обозначены префиксом _), и доступ к ним осуществляется только через методы get_value1 и get_value2. Таким образом, значения атрибутов не могут быть изменены напрямую извне.

Пример использования:

```
data = ImmutableData(10, 20)
print(data.get_value1()) # Выведет: 10
print(data.get_value2())  # Выведет: 20

# Попытка изменить значение атрибута вызовет ошибку
data._value1 = 100  # AttributeError: can't set attribute
```

Этот подход позволяет создать неизменяемый класс данных, в котором значения атрибутов не могут быть изменены после создания экземпляра класса.