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

Кафедра инфокоммуникаций

Отчет по лабораторной работе №4.3
Наследование и полиморфизм в языке Python
по дисциплине «Объектно-ориентированное программирование»

Выполнил студент группы ИВТ-б-о-21-1
Лысенко И.А. « »_____20__г.
Подпись студента_____
Работа защищена « »_____20__г.
Проверил Воронкин Р.А. _____
(подпись)

Ставрополь 2024

Цель работы: приобретение навыков по созданию иерархии классов при написании программ с помощью языка программирования Python версии 3.x.

Ход работы:

1. Создал репозиторий на GitHub: https://github.com/IsSveshuD/OOP_Lab_4.3.git .
2. Проработал примеры 1 и 2:

```
3/4                                  C:\Users\user\AppData\
Введите обыкновенную дробь: 3/5      I have 3 sides
3/5                                  I have 4 sides
27/20                                I have 5 sides
3/20                                 I have 6 sides
9/20
4/5                                  Process finished with
```

Рисунок 1 – Примеры 1 и 2

3. Выполнил индивидуальное задание 1.

```python
# Python program showing
# abstract base class work


from abc import ABC, abstractmethod


class Triad(ABC):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    @abstractmethod
    def increase(self):
        pass

    @abstractmethod
    def display(self):
        pass

    @abstractmethod
    def compare(self, other):
        pass


class Date(Triad):
    def __init__(self, day, month, year):
        super().__init__(day, month, year)
```

```
    def increase(self):
        self.a += 1
        self.b += 1
        self.c += 1

    def display(self):
        print(f"Дата: {self.a}/{self.b}/{self.c}")

    def compare(self, other):
        if self.a == other.a and self.b == other.b and self.c == other.c:
            return "Даты равны"
        else:
            return "Даты не равны"

    def __eq__(self, other):
        return self.a == other.a and self.b == other.b and self.c == other.c

    def __gt__(self, other):
        return (self.c, self.b, self.a) > (other.c, other.b, other.a)

    def __lt__(self, other):
        return (self.c, self.b, self.a) < (other.c, other.b, other.a)

    def __ge__(self, other):
        return (self.c, self.b, self.a) >= (other.c, other.b, other.a)

    def __le__(self, other):
        return (self.c, self.b, self.a) <= (other.c, other.b, other.a)

    def __ne__(self, other):
        return not self.__eq__(other)

if __name__ == '__main__':
    date1 = Date(17, 3, 2001)

    date1.display()
    date1.increase()
    date1.display()

    date2 = Date(17, 3, 2001)
    print(date1.compare(Date(17, 3, 2001)))

    print(date1 == date2)
    print(date1 > date2)
    print(date1 < date2)
    print(date1 >= date2)
    print(date1 <= date2)
    print(date1 != date2)
```

Рисунок 2 – Индивидуальное задание1

4. Получил следующий результат работы индивидуального задания 1.

3

```
C:\Users\user\AppDa
Дата: 17/3/2001
Дата: 18/4/2002
Даты не равны
False
True
False
True
False
True
```

Рисунок 3 – Результат работы задания 1

5. Выполнил индивидуальное задание 1.

```python
from abc import ABC, abstractmethod


class Triad(ABC):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    @abstractmethod
    def increase(self):
        pass

    @abstractmethod
    def display(self):
        pass


class Date(Triad):
    def __init__(self, day, month, year):
        super().__init__(day, month, year)

    def increase(self):
        self.a += 1
        self.b += 1
        self.c += 1

    def display(self):
        print(f"Дата: {self.a}/{self.b}/{self.c}")
```

```python
class Time(Triad):
    def __init__(self, hour, minute, second):
        super().__init__(hour, minute, second)

    def increase(self):
        self.a += 1
        self.b += 1
        self.c += 1

    def display(self):
        print(f"Время: {self.a}:{self.b}:{self.c}")


if __name__ == '__main__':
    date = Date(17, 3, 2001)
    time = Time(12, 30, 45)

    date.display()
    date.increase()
    date.display()

    time.display()
    time.increase()
    time.display()
```

Рисунок 4 – Индивидуальное задание 2

6. Получил следующий результат работы индивидуального задания 2:

```
C:\Users\user\App
Дата: 17/3/2001
Дата: 18/4/2002
Время: 12:30:45
Время: 13:31:46
```

Рисунок 5 – Результат работы задания 2.

Ответы на вопросы:

1. Что такое наследование как оно реализовано в языке Python?

Наследование в объектно-ориентированном программировании (ООП) – это способ создания нового класса на основе уже существующего класса.

Класс, который наследует свойства и методы от другого класса, называется производным классом или подклассом, а класс, от которого наследуют, называется базовым классом или суперклассом.

В Python наследование реализуется очень просто. Вот пример:

```
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        pass
class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
# Пример использования наследования
dog = Dog("Buddy")
print(dog.name) # Выведет "Buddy"
print(dog.speak()) # Выведет "Woof!"
cat = Cat("Whiskers")
print(cat.name) # Выведет "Whiskers"
print(cat.speak()) # Выведет "Meow!"
```

В этом примере классы Dog и Cat наследуют от базового класса Animal.

Это означает, что классы Dog и Cat наследуют атрибуты и методы, определенные в классе Animal, такие как конструктор init и метод speak. При этом, классы-наследники могут переопределить методы базового класса, чтобы адаптировать их под свои нужды.

2. Что такое полиморфизм и как он реализован в языке Python?

Полиморфизм – это принцип объектно-ориентированного программирования, который позволяет объектам различных типов обрабатываться с использованием общего интерфейса. Это означает, что объекты могут проявлять различное поведение в зависимости от их типа или класса.

В Python полиморфизм может быть достигнут несколькими способами:

Полиморфизм параметров функций: Функции могут принимать аргументы различных типов, и их поведение может зависеть от типа переданных объектов.

```
def print_type(obj):
    print(type(obj))
    print_type(5) # <class 'int'>
    print_type("Hello") # <class 'str'>
```

Полиморфизм методов: Различные классы могут предоставлять свои собственные реализации методов с одинаковыми именами, что позволяет использовать их полиморфически.

```
class Dog:
    def speak(self):
        return "Woof!"
class Cat:
    def speak(self):
        return "Meow!"
    def animal_sound(animal):
        return animal.speak()
dog = Dog()
cat = Cat()
print(animal_sound(dog)) # Woof!
print(animal_sound(cat)) # Meow!
```

Полиморфизм через магические методы (dunder-методы): Магические методы, такие как len, add, eq, и другие, позволяют объектам поддерживать стандартные операции. Это делает их полиморфными по отношению к встроенным функциям и операторам.

```
class Circle:
    def init(self, radius):
        self.radius = radius
    def add(self, other):
        return Circle(self.radius + other.radius)
circle1 = Circle(3)
circle2 = Circle(5)
result_circle = circle1 + circle2
print(result_circle.radius) # 8
```

Все эти методы реализуют идею полиморфизма в Python, позволяя объектам различных типов взаимодействовать с использованием общих интерфейсов.

3. Что такое "утиная" типизация в языке программирования Python?

"Утиная" типизация (duck typing) в языке программирования Python относится к философии, согласно которой тип или класс объекта не важен, пока объект поддерживает необходимые методы или свойства. Принцип "утиной" типизации формулируется как "если это выглядит как утка, плавает как утка и крякает как утка, то это, вероятно, и есть утка".

Это означает, что в Python важнее не явное указание типов данных, а возможность объекта выполнять определенные действия. Например, если объект имеет методы, необходимые для выполнения определенной операции, то этот объект может использоваться в контексте этой операции, независимо от его фактического типа или класса.

Вот пример "утиной" типизации в Python:

```
class Duck:
```

```
    def quack(self):
        print("Quack!")
class Person:
    def quack(self):
        print("I'm quacking like a duck!")
    def in_the_forest(entity):
        entity.quack()
duck = Duck()
person = Person()
in_the_forest(duck) # Выведет "Quack!"
in_the_forest(person) # Выведет "I'm quacking like a duck!"
```

В этом примере функция intheforest принимает объект и вызывает метод quack у этого объекта. Объекты Duck и Person имеют разные типы, но оба поддерживают метод quack, поэтому они могут быть использованы в контексте этой функции.

4. Каково назначение модуля abc языка программирования Python?

Модуль abc (Abstract Base Classes) в языке программирования Python предназначен для создания абстрактных базовых классов (ABC).

Абстрактный базовый класс в Python - это класс, который может содержать абстрактные методы, то есть методы, которые не имеют реализации, но должны быть переопределены в производных классах.

Назначение модуля abc заключается в том, чтобы обеспечить механизм определения интерфейсов и структур данных, которые должны быть реализованы в производных классах. Это помогает в организации кода, улучшает его читаемость и обеспечивает единообразие интерфейсов.

Пример использования модуля abc для создания абстрактного базового класса:

```
from abc import ABC, abstractmethod
class Shape(ABC):
```

```
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimeter(self):
        pass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius * self.radius
    def perimeter(self):
        return 2 * 3.14 * self.radius
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    def perimeter(self):
        return 2 * (self.width + self.height)
# Пример использования абстрактного базового класса
circle = Circle(5)
print(circle.area()) # Выведет площадь круга
print(circle.perimeter()) # Выведет периметр круга
rectangle = Rectangle(4, 6)
print(rectangle.area()) # Выведет площадь прямоугольника
print(rectangle.perimeter()) # Выведет периметр прямоугольника
```

В этом примере класс Shape является абстрактным базовым классом, который содержит абстрактные методы area() и perimeter(). Классы Circle и Rectangle являются производными классами, которые переопределяют эти методы. Таким образом, модуль abc позволяет создавать структуры, которые обеспечивают единообразие интерфейсов для классов, реализующих определенную функциональность.

5. Как сделать некоторый метод класса абстрактным?

В Python абстрактный метод класса может быть создан с использованием декоратора @abstractmethod из модуля abc (Abstract Base Classes). Этот декоратор позволяет определить метод как абстрактный, то есть метод, который должен быть переопределен в производных классах.

Вот пример того, как сделать метод класса абстрактным:

```
from abc import ABC, abstractmethod
class AbstractClass(ABC):
    @abstractmethod
    def abstract_method(self):
        pass
class ConcreteClass(AbstractClass):
    def abstract_method(self):
        print("Implementing the abstract method")
# Этот код будет работать
instance = ConcreteClass()
instance.abstract_method() # Выведет "Implementing the abstract method"
# Этот код вызовет ошибку, так как мы пытаемся создать экземпляр абстрактного класса
# abstract_instance = AbstractClass()
```

В этом примере метод abstract_method в классе AbstractClass определен как абстрактный с помощью декоратора @abstractmethod. Затем в

производном классе ConcreteClass этот метод переопределяется с конкретной реализацией.

6. Как сделать некоторое свойство класса абстрактным?

В Python абстрактное свойство класса можно создать с использованием модуля abc (Abstract Base Classes) и декоратора @property в сочетании с абстрактным методом. Это позволяет определить свойство как абстрактное, то есть свойство, которое должно быть переопределено в производных классах.

Вот пример того, как сделать свойство класса абстрактным:

```
from abc import ABC, abstractmethod
class AbstractClass(ABC):
    @property
    @abstractmethod
    def abstract_property(self):
        pass
class ConcreteClass(AbstractClass):
    @property
    def abstract_property(self):
        return "Implementing the abstract property"
# Этот код будет работать
instance = ConcreteClass()
print(instance.abstract_property) # Выведет "Implementing the abstract property"
# Этот код вызовет ошибку, так как мы пытаемся создать экземпляр абстрактного класса
# abstract_instance = AbstractClass()
```

В этом примере свойство abstract_property в классе AbstractClass определено как абстрактное с помощью декоратора @property в сочетании с @abstractmethod. Затем в производном классе ConcreteClass это свойство переопределяется с конкретной реализацией.

7. Каково назначение функции isinstance?

Функция isinstance() в Python используется для проверки принадлежности объекта к определенному типу или классу. Она возвращает True, если объект является экземпляром указанного класса или его подкласса, и False в противном случае. Назначение функции isinstance() заключается в том, чтобы предоставить способ динамической проверки типа объекта во время выполнения программы. Это может быть полезно для выполнения различных действий в зависимости от типа объекта или для обеспечения корректной обработки различных типов данных.

Вывод: в результате выполнения лабораторной работы были приобретены навыки по созданию иерархии классов при написании программ с помощью языка программирования Python.