МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет информационных технологий и робототехники

Кафедра «Программное обеспечение информационных систем и технологий»

**Отчет по лабораторной работе №4**

**Вариант 12**

По дисциплине: «Скриптовые языки программирования»

**«Объектно-ориентированное программирование»**

Выполнил:

студент группы

10702423

Татур Е. Ю.

Проверил**:**

ст.преподаватель

Козловская Ю.Б.

Минск 2024

**Цель работы**

Продемонстрировать принципы объектно-ориентированного программирования (ООП), такие как инкапсуляция, наследование и полиморфизм, а также показать, как можно расширять функциональность классов и оптимизировать выполнение функций с использованием механизмов наследования, перегрузки операторов, декораторов и управления состоянием объектов через атрибуты и методы.

**Задания**

1. Напишите программу, создающую класс MinVoda (для определения типа минеральной воды), принимающую один аргумент при инициализации (отвечающий за тип выбираемой минеральной воды: столовая или лечебная).

В этом классе реализуется метод show\_my\_drink () и на печать выводится фраза «Минеральная вода – {Тип минеральной воды}» при указании типа минеральной воды. В противном случае на печать выводится фраза «Обычная минеральная вода».

1. Напишите программу, которая создает класс Triangle (представляющий собой треугольник). Конструктор этого класса должен принимать на вход три числа (длины сторон треугольника), также у этого класса должен быть метод, возвращающий периметр.

Надо наследовать класс EquilateralTriangle, представляющий равносторонний треугольник от класса Triangle. Переопределить конструктор этого класса с использованием конструктора базового класса. Конструктор класса EquilateralTriangle должен принимать на вход одно число (длину стороны). Программа должна возвращать следующие значения в зависимости от ситуации:

* Можно построить треугольник!;
* Это равносторонний треугольник!;
* Периметр треугольника равен = ;
* Треугольник не сделать из этого.

1. Напишите программу, реализующую перегрузку оператора + для класса Point, который реализует точку на плоскости. В классе должны быть реализованы следующие методы:

* \_init\_(),который перегружает конструктор класса. Конструктор класса вызывается при создании экземпляра класса;
* getXY(), который возвращает значение координат точки (х;у) в виде списка;
* \_add\_(),который реализует перегрузку оператора сложения. Реализация предусматривает суммирование координат по осям Х и У;
* Show(), который выводит координаты точки на экран.

1. Пусть задан класс Point, описывающий точку х;у на координатной плоскости. Используя механизм наследования, расширьте возможности класса Point путем добавления нового атрибута цвета, реализовав при этом подкласс PointColor. В классе Point реализуйте следующие атрибуты:
   * координаты точки;
   * метод инициализации, который получает два параметра – координаты точки х;у;
   * метод вычисления расстояния от точки до начала координат;
   * метод getPoint(), который возвращает точку в виде списка;

В подклассе PointColor реализуйте следующие атрибуты:

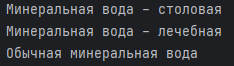
* + цвет точки color;
  + метод начальной инициализации, который получает параметры координаты точки и цвет;
  + метод доступа к цвету color с именем getColor().

1. Напишите декоратор, оптимизирующий работу декорируемой функции. Декоратор должен сохранять результат работы функции на ближайшие три запуска и вместо выполнения функции возвращать сохраненный результат. После трех запусков функция должна вызываться вновь, а результат работы функции – вновь кешироваться.

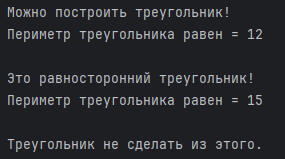
Рекомендации: создайте в декоратере переменную-кеш (промежуточный буфер с быстрым доступом к нему), сохраните в нем результат выполнения декорируемой функции. Создайте в декораторе переменную, хранящую счетчик запросов. Пока значение счетчика ниже предельного, отдавайте результат, сохраненный в кеше. Когда число запросов к функции превысит предел и надо будет снова высчитывать результат выполнения функции, сбросьте счетчик, выполните декорируемую функцию и заново сохраните результат ее выполнения в переменную-кеш.

**Ход выполнения работы**

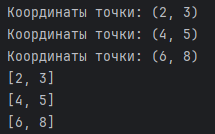
**Результат работы**



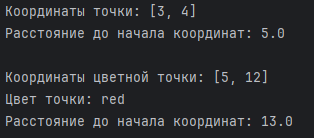
*Рисунок 1 – Результат выполненного задания 1.*



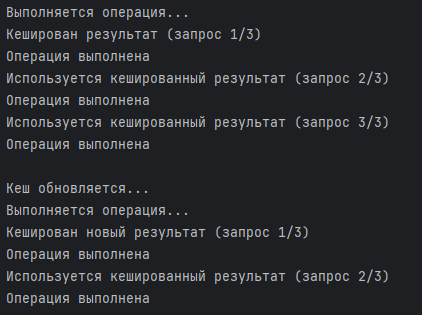
*Рисунок 2 – Результат выполненного задания 2.*



*Рисунок 3 – Результат выполненного задания 3.*



*Рисунок 4 – Результат выполненного задания 4.*



*Рисунок 5 – Результат выполненного задания 5.*

**Листинг**

**ex1.py:**

class MinVoda:  
 def \_\_init\_\_(self, water\_type = None):  
 self.water\_type = water\_type  
  
 def show\_my\_drink(self):  
 if self.water\_type in ['столовая', 'лечебная']:  
 print(f"Минеральная вода – {self.water\_type}")  
 else:  
 print("Обычная минеральная вода")  
  
# Примеры использования класса  
if \_\_name\_\_ == "\_\_main\_\_":  
 voda1 = MinVoda("столовая")  
 voda1.show\_my\_drink() # Вывод: Минеральная вода – Столовая  
  
 voda2 = MinVoda("лечебная")  
 voda2.show\_my\_drink() # Вывод: Минеральная вода – Лечебная  
  
 voda3 = MinVoda("обычная")  
 voda3.show\_my\_drink() # Вывод: Обычная минеральная вода

**ex2.py:**

class Triangle:  
 def \_\_init\_\_(self, a, b, c):  
 self.a = a  
 self.b = b  
 self.c = c  
  
 def can\_form\_triangle(self):  
 # Проверка условия существования треугольника  
 return self.a + self.b > self.c and self.a + self.c > self.b and self.b + self.c > self.a  
  
 def perimeter(self):  
 if self.can\_form\_triangle():  
 return self.a + self.b + self.c  
 else:  
 return None  
  
 def show\_triangle\_info(self):  
 if self.can\_form\_triangle():  
 print("Можно построить треугольник!")  
 print(f"Периметр треугольника равен = {self.perimeter()}\n")  
 else:  
 print("Треугольник не сделать из этого.\n")  
  
# Класс EquilateralTriangle, наследующий Triangle  
class EquilateralTriangle(Triangle):  
 def \_\_init\_\_(self, side\_length):  
 # Использование конструктора базового класса  
 super().\_\_init\_\_(side\_length, side\_length, side\_length)  
  
 def show\_triangle\_info(self):  
 if self.can\_form\_triangle():  
 print("Это равносторонний треугольник!")  
 print(f"Периметр треугольника равен = {self.perimeter()}\n")  
 else:  
 print("Треугольник не сделать из этого.\n")  
  
# Примеры использования  
# Обычный треугольник  
triangle = Triangle(3, 4, 5)  
triangle.show\_triangle\_info()  
  
# Равносторонний треугольник  
equilateral\_triangle = EquilateralTriangle(5)  
equilateral\_triangle.show\_triangle\_info()  
  
# Треугольник с невозможными сторонами  
invalid\_triangle = Triangle(1, 2, 3)  
invalid\_triangle.show\_triangle\_info()

**ex3.py:**

class Point:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def getXY(self):  
 # Возвращает координаты в виде списка  
 return [self.x, self.y]  
  
 def \_\_add\_\_(self, other):  
 # Перегрузка оператора сложения  
 if isinstance(other, Point):  
 return Point(self.x + other.x, self.y + other.y)  
 return NotImplemented  
  
 def show(self):  
 # Выводит координаты точки  
 print(f"Координаты точки: ({self.x}, {self.y})")  
  
# Примеры использования  
point1 = Point(2, 3)  
point2 = Point(4, 5)  
  
# Сложение точек  
result\_point = point1 + point2  
  
# Вывод результата  
point1.show() # Вывод: Координаты точки: (2, 3)  
point2.show() # Вывод: Координаты точки: (4, 5)  
result\_point.show() # Вывод: Координаты точки: (6, 8)  
  
# Получение координат в виде списка  
print(point1.getXY()) # Вывод: [2, 3]  
print(point2.getXY()) # Вывод: [4, 5]  
print(result\_point.getXY()) # Вывод: [6, 8]

**ex4.py:**

import math  
  
# Базовый класс Point  
class Point:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def distance\_to\_origin(self):  
 # Вычисляет расстояние от точки до начала координат (0, 0)  
 return math.sqrt(self.x \*\* 2 + self.y \*\* 2)  
  
 def getPoint(self):  
 # Возвращает координаты точки в виде списка [x, y]  
 return [self.x, self.y]  
  
# Подкласс PointColor, наследующий Point  
class PointColor(Point):  
 def \_\_init\_\_(self, x, y, color):  
 # Вызов конструктора базового класса  
 super().\_\_init\_\_(x, y)  
 self.color = color  
  
 def getColor(self):  
 # Возвращает цвет точки  
 return self.color  
  
# Примеры использования  
point = Point(3, 4)  
print("Координаты точки:", point.getPoint()) # Вывод: [3, 4]  
print("Расстояние до начала координат:", point.distance\_to\_origin()) # Вывод: 5.0  
  
colored\_point = PointColor(5, 12, "red")  
print("\nКоординаты цветной точки:", colored\_point.getPoint()) # Вывод: [5, 12]  
print("Цвет точки:", colored\_point.getColor()) # Вывод: red  
print("Расстояние до начала координат:", colored\_point.distance\_to\_origin()) # Вывод: 13.0

**ex5.py:**

def cache\_decorator(func):  
 cache = None # Переменная для хранения кешированного результата  
 call\_count = 0 # Счетчик вызовов  
 max\_calls = 3 # Число вызовов, после которых кеш обновляется  
  
 def wrapper(\*args, \*\*kwargs):  
 nonlocal cache, call\_count  
 if call\_count == 0:  
 cache = func(\*args, \*\*kwargs) # Вычисление нового результата  
 call\_count = 1 # Сброс счетчика запросов  
 print(f"Кеширован результат (запрос {call\_count}/{max\_calls})")  
 elif call\_count < max\_calls:  
 call\_count += 1  
 print(f"Используется кешированный результат (запрос {call\_count}/{max\_calls})")  
 else:  
 print("\nКеш обновляется...")  
 cache = func(\*args, \*\*kwargs) # Вычисление нового результата  
 call\_count = 1 # Сброс счетчика запросов  
 print(f"Кеширован новый результат (запрос {call\_count}/{max\_calls})")  
 return cache  
  
 return wrapper  
  
# Пример использования декоратора  
@cache\_decorator  
def expensive\_calculation():  
 print("Выполняется операция...")  
 return "Операция выполнена"  
  
# Тестирование  
print(expensive\_calculation()) # Первый запуск, выполняется операция  
print(expensive\_calculation()) # Второй запуск, используется кеш  
print(expensive\_calculation()) # Третий запуск, используется кеш  
print(expensive\_calculation()) # Четвертый запуск, кеш обновляется  
print(expensive\_calculation()) # Пятый запуск, используется кеш

**Контрольные вопросы**

1. В чем суть объектно-ориентированного программирования?

**Суть объектно-ориентированного программирования (ООП)** заключается в организации кода вокруг объектов, которые объединяют данные и методы для работы с этими данными. Это позволяет создавать более структурированные и модульные программы, упрощая их поддержку и развитие, в отличие от процедурного программирования, где функции и данные разделены.

1. Что представляет собой класс?

**Класс** — это шаблон или модель для создания объектов определённого типа. Он описывает структуру объектов (набор полей) и определяет методы, с помощью которых можно взаимодействовать с этими объектами. Класс служит основой для создания экземпляров (объектов) в программе.

1. Что такое объект класса?

**Объект класса** — это конкретный экземпляр класса, созданный на основе его определения. Каждый объект имеет свои уникальные значения свойств, но использует методы, определенные в классе.

1. Что представляет собой объект класса?

**Объект класса** представляет собой конкретную реализацию абстрактного типа данных, описанного классом. Он хранит данные (свойства) и может выполнять действия (методы), определенные в классе, что позволяет ему функционировать как независимый элемент программы.

1. В чем суть метода класса?

**Метод класса** — это функция, определенная внутри класса, которая описывает поведение объектов этого класса. Методы могут использовать данные объекта и изменять их состояние, а также выполнять различные операции.

1. В чем суть перегрузки операторов?

**Перегрузка операторов** — это возможность переопределять стандартное поведение операторов (например, +, -, \*, /) для пользовательских типов данных (классов). Это позволяет использовать привычные операторы для работы с объектами так же, как с примитивными типами данных.

1. Какие существуют методы перегрузки операторов?

Методы перегрузки операторов включают специальные функции, которые начинаются с двойного подчеркивания (например, \_\_add\_\_ для сложения). Каждый оператор имеет свой соответствующий метод:

* \_\_add\_\_ — для +
* \_\_sub\_\_ — для -
* \_\_mul\_\_ — для \*
* \_\_truediv\_\_ — для /

1. Какие выделяют методы перегрузки операторов?

Методы перегрузки операторов, как правило, делятся на два типа:

* Унарные операторы: работают с одним объектом (например, -x).
* Бинарные операторы: работают с двумя объектами (например, x + y)

1. Что такое наследование? В чем суть наследования в Python?

**Наследование** — это механизм ООП, позволяющий создавать новый класс на основе существующего. Новый класс (дочерний) наследует свойства и методы родительского класса, что способствует повторному использованию кода и расширению функциональности. В Python наследование реализуется через указание родительского класса при определении нового класса.

1. Какие существуют виды наследования?

Виды наследования включают:

* Простое наследование: один дочерний класс от одного родительского.
* Множественное наследование: один дочерний класс от нескольких родительских.
* Многоуровневое наследование: дочерний класс наследует от родительского класса, который сам является дочерним по отношению к другому классу.
* Гибридное наследование: комбинация различных видов наследования

1. Для чего нужна проверка принадлежности к классу/типу?

**Проверка принадлежности к классу/типу** необходима для обеспечения корректности работы программы и предотвращения ошибок во время выполнения. Она позволяет определить, является ли объект экземпляром определенного класса или его подкласса.

1. Что такое декоратор? В чем суть его работы?

**Декоратор** — это специальная функция в Python, которая позволяет изменять или расширять поведение других функций или методов без изменения их исходного кода. Декораторы применяются перед определением функции и могут принимать аргументы.

1. Как используют декораторы в реальных приложениях?

Использование декораторов в реальных приложениях включает:

* Логирование вызовов функций.
* Проверку прав доступа.
* Кэширование результатов функций для повышения производительности.
* Измерение времени выполнения функций

1. В чем суть свойств класса?

**Свойства класса** представляют собой специальные методы доступа к атрибутам объекта (геттеры и сеттеры), позволяющие управлять доступом к данным и обеспечивать инкапсуляцию. Свойства позволяют контролировать изменение значений атрибутов и выполнять дополнительные действия при их установке или получении.