**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«РОССИЙСКАЯ АКАДЕМИЯ НАРОДНОГО ХОЗЯЙСТВА**

**И ГОСУДАРСТВЕННОЙ СЛУЖБЫ**

**ПРИ ПРЕЗИДЕНТЕ РОССИЙСКОЙ ФЕДЕРАЦИИ»**

**НИЖЕГОРОДСКИЙ ИНСТИТУТ УПРАВЛЕНИЯ – филиал РАНХиГС**

**Отчет по программированию №5**

**Выполнил: студент группы Иб-321**

**Комышков Владислав Дмитриевич**

[Задание 1 2](#_Toc1103388357)

[Ход 3](#_Toc738667848)

[Итог 5](#_Toc752733589)

[Задание 2 5](#_Toc1261854985)

[Ход 6](#_Toc782806094)

[Итог 8](#_Toc988911455)

[Задание 4 8](#_Toc933957135)

[Ход 9](#_Toc1425153168)

[Итог 10](#_Toc53004273)

[Задание 5 12](#_Toc1242181851)

[Ход 13](#_Toc978263638)

[Итог 21](#_Toc112493314)

# Задание 1

Текст задания: На основании анализа схемы соотношения классов (Рис 1.1), реализовать соответствующий скрипт, реализующий эту схему с учётом наследования и полиморфизма.

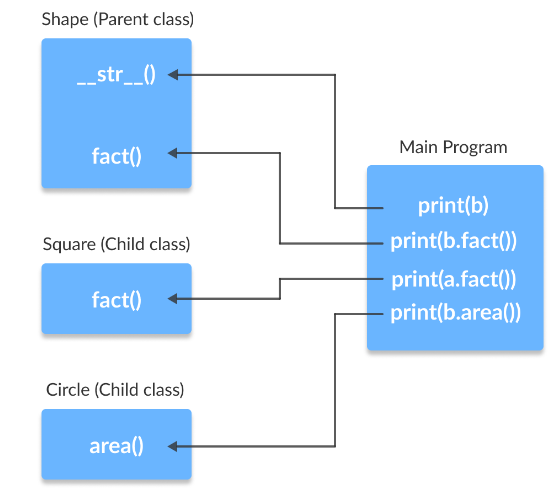


Рис 1.1 - схема к заданию

## Ход

Создаем базовый класс Shape (рис 1.2) с методами представленными в схеме на рисунке 1.1.

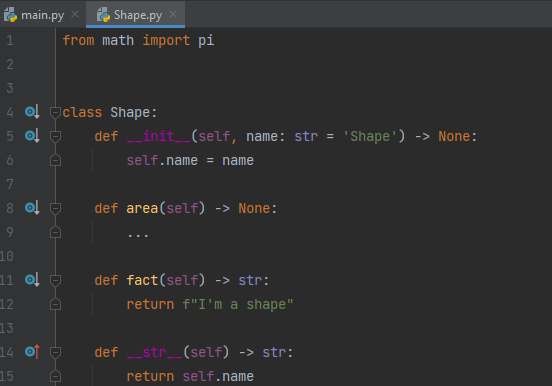


Рис 1.2 - класс Shape

Далее создаем дочерние классы Square и Circle (рис 1.3) с соответствующими схеме (рис 1.1) методами.

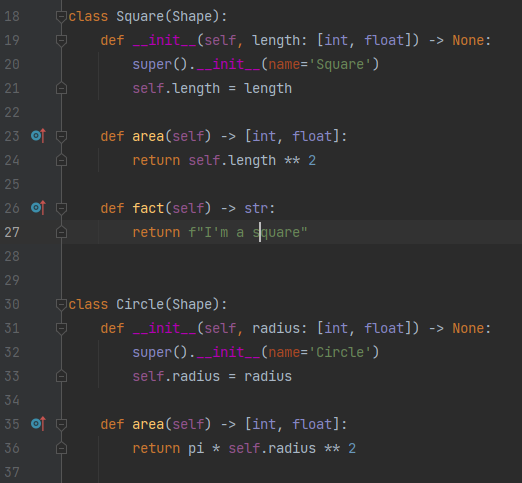


Рис 1.3 - класс Shape

Реализуем скрипт согласно схеме (рис 1.1). Скрипт изображен на рисунке 1.4.

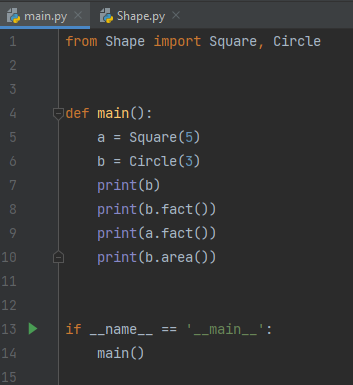


Рис 1.4 - скрипт main.py

Запускаем скрипт и смотрим результат (рис 1.5).

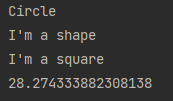


Рис 1.5 - результат скрипта main.py

### Итог

На основании схемы, представленной в задании (рис 1.1), с учётом наследования и полиморфизма, мы реализовали скрипт и проверили его работу.

# Задание 2

Текст задания: Реализовать описанный в этом разделе вариант полиморфизма с использованием абстрактного метода.

Проверить (показать на примере) исключение NotImplementedError в действии.

## Ход

Создаем классы Vector2D и Vector3D (рис 2.1). В них реализуем методы возвращающие заданные параметры.

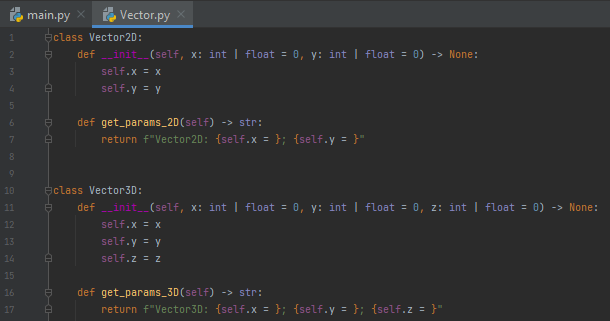


Рис 2.1 - классы Vector2D и Vector3D

Напишем скрипт. Создадим объекты принадлежащие к обоим классам, объекты поместим в список. В цикле, на каждой итерации, будем вызывать метод класса Vector2D, чтобы вызвать ошибку. Посмотрим на результат (рис 2.2).

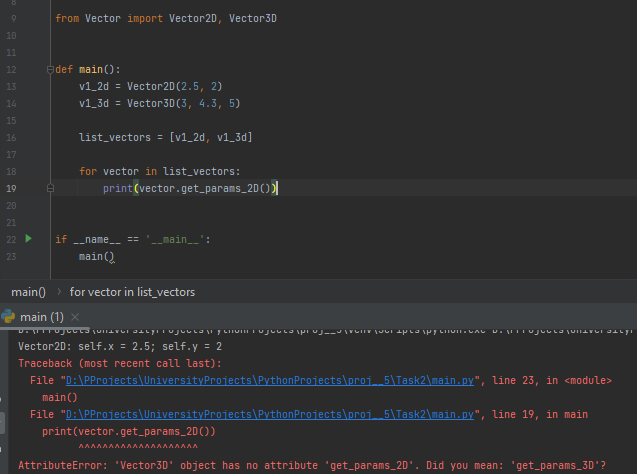


Рис 2.2 - скрипт main.py и результат

Как видим функция сработала для первого объекта, далее вышла ошибка. Она говорит нам о том что мы пытаемся вызвать несуществующий метод класса.

Теперь изменим имена методов классов на схожие и создадим еще один класс Vector4D, в котором специально забудем прописать такой же метод (Рис 2.3).

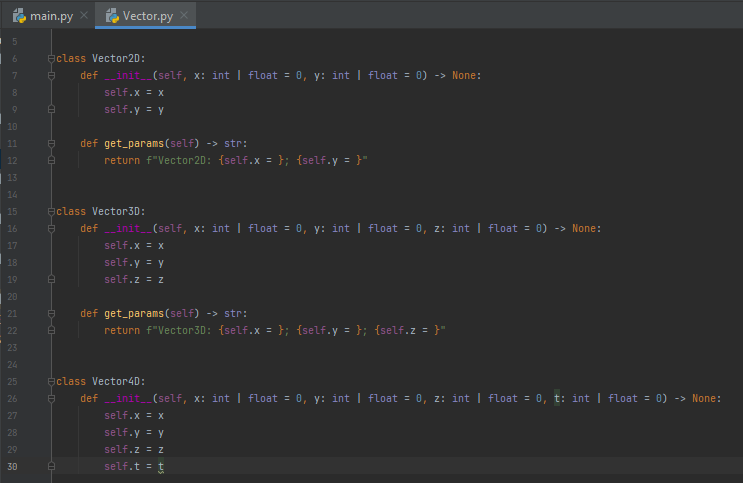


Рис 2.3 - классы Vector2D, Vector3D и Vector4D

Добавим в наш скрипт еще один объект принадлежащий классу Vector4D. Внесем объект в список. В цикле, на каждой итерации, будем вызывать метод, который есть у классов Vector2D и Vector3D. Результат работы скрипта представлен на рисунке 2.4.

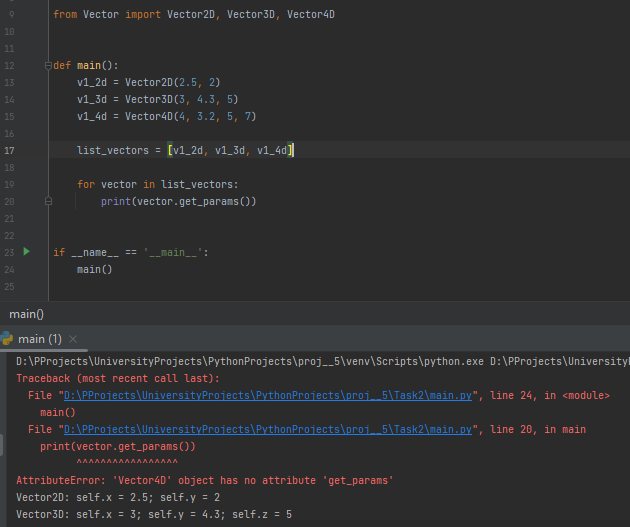


Рис 2.4 - скрипт main.py и результат

Как мы видим метод был вызван в тех классах в которых он есть.

Создадим базовый класс Vector и пропишем у него все тот же метод, но в теле метода будем вызывать ошибку **NotImplementedError**, которая будет информировать нас о то что мы забыли объявить метод в дочернем классе (рис 2.5).

Запустим наш скрипт снова и посмотрим на результат.

Мы получили ошибку с нашим сообщением.

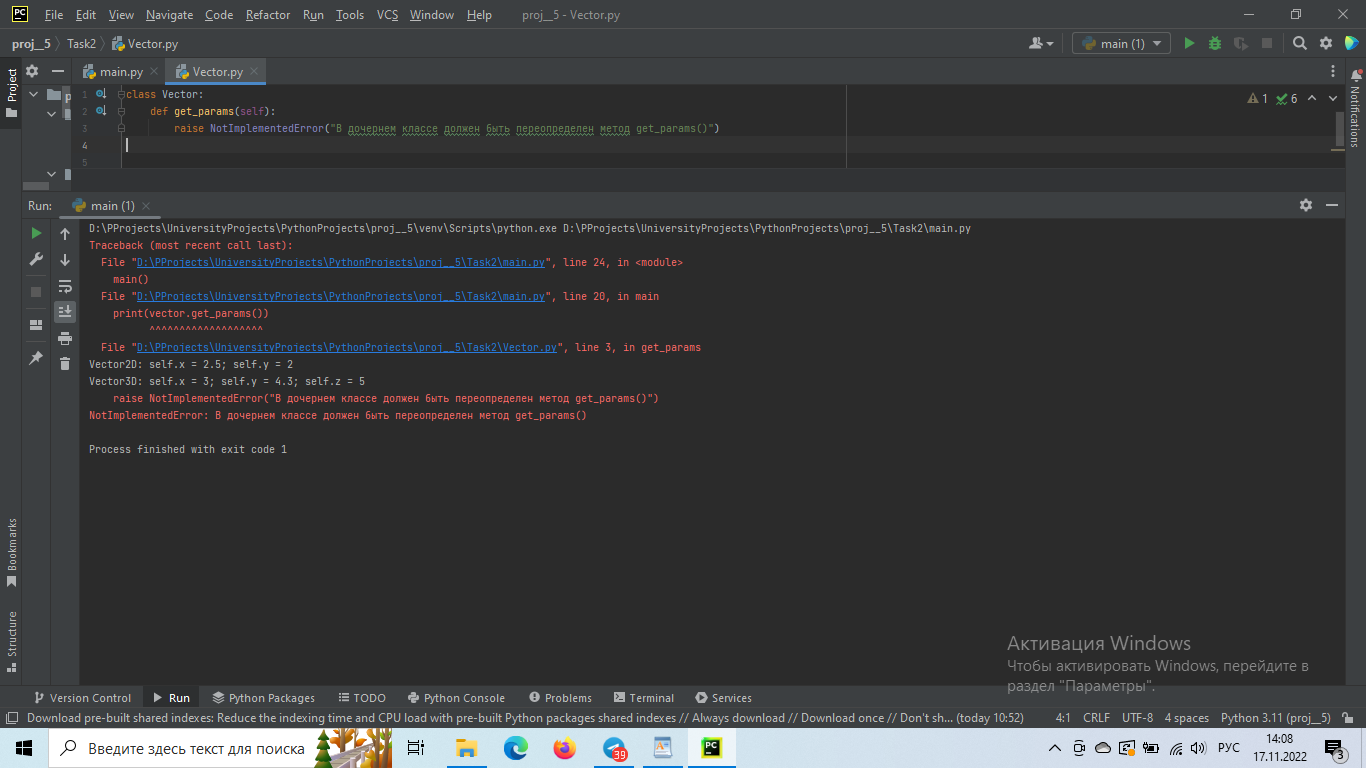


Рис 2.5 - базовый класс Vector и результат работы скрипта main.py

### Итог

Мы реализовали описанный в разделе вариант полиморфизма с использованием абстрактного метода и проверили исключение NotImplementedError в действии.

# Задание 4

Текст задания: Реализовать приведённые в задании текст с выводом результата на печать в консоли и дать объяснение полученных результатов.

## Ход

Нам даны базовый класс **Base** и дочерний **Sub**. Мы создаем объекты соответстующие классам и с помощью функции **isinstance** сравниваем их (рис 3.1).

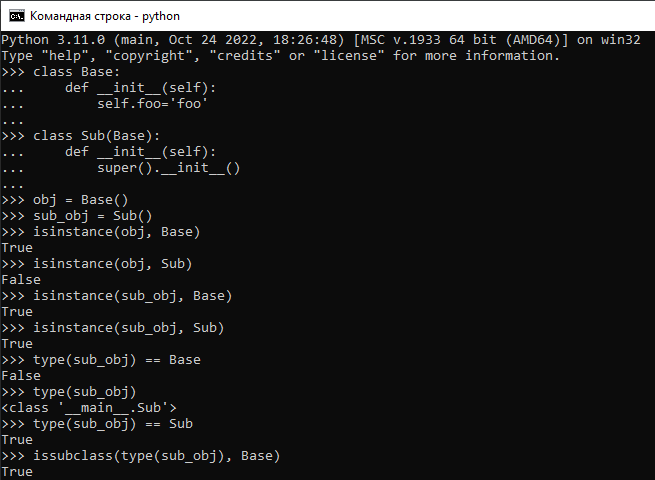


Рис 3.1 - консоль

Функция **isinstance** позволяет проверить принадлежность экземпляра к классу.

Объект **obj** принадлежит базовому классу Base, **sub\_obj** принадлежит дочернему классу **Sub**, а он в свою очередь отсылается к базовому классу **Base**, поэтому при проверке принадлежности **sub\_obj** к **Base** мы получаем **True**.

Функция **type** в python возвращет точный класс из которого был передан аргумент, поэтому при попытке сравнить тип **sub\_obj** с **Base** мы получаем **False**, то есть по сути мы пытаемся сравнивнить **<class ‘\_\_main\_\_.Sub’>** и **<class ‘\_\_main\_\_.Base’>**.

Используя функцию **issubclass** для проверки отношения класса **Base** к типу **sub\_obj** мы получим **True**, так как **sub\_obj** относится к классу **Sub**, который является дочерним классом **Base**.

Следующий текст который мы реализуем представлен на рисунке 3.2.

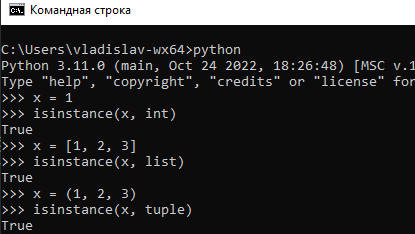


Рис 3.2 - консоль

Сейчас мы просто проверям принадлежность содержимого переменной **x** к разным типам, что соответственно возвращает **True**.

На рисунке 3.3 представлен способ проверки принадлежности строки к одному из типов. Так как в кортеже типов присутствует **str**, мы получаем **True**.

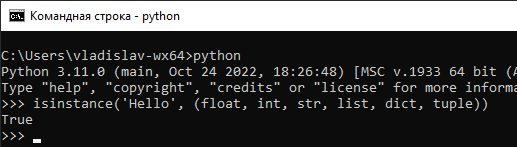


Рис 3.3 - консоль

Следующий код задания представлен ниже (рис 3.4).

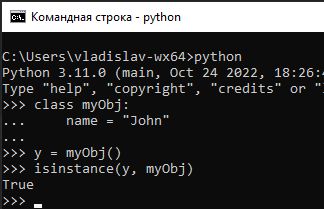


Рис 3.4 - консоль

Мы знаем что переменная после присвоения отсылается к определенному месту в памяти. В данном случае **y** отсылается к объекту **myObj**, то есть к месту его нахождения в памяти, поэтому при проверке принадлежности экземпляра к классу, мы получаем **True**.

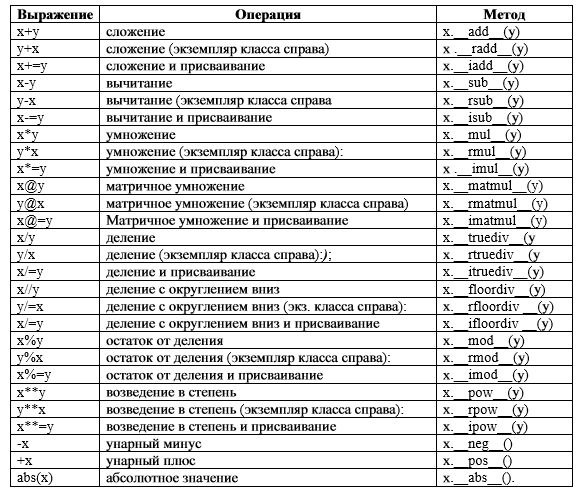
### Итог

Мы реализовали приведённый в задании текст и дали объяснение полученным результатам.

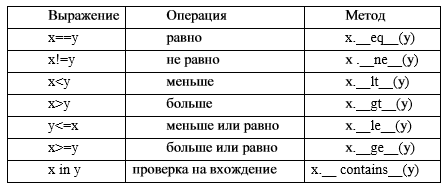
# Задание 5

Текст задания: Реализовать (в скрипте) и объяснить работу обеспечения полиморфизма методов, показанных на рисунке в лекции.

Реализовать 3-4 варианта обеспечения полиморфизма методов, показанных в таблице 1, не реализованных в предыдущем пункте задания.

  
Таблица 1

Реализовать обеспечение полиморфизма методов сравнения, показанных в таблице 2.

  
Таблица 2

## Ход

Создадим схожий класс, как в лекции.

Наш класс будет называться **Player** (рис 4.1), то есть игрок, который будет иметь атрибут **hp** (health points - очки жизни). Также класс будет иметь те же магические методы, что и на рисунке представленном в лекции.

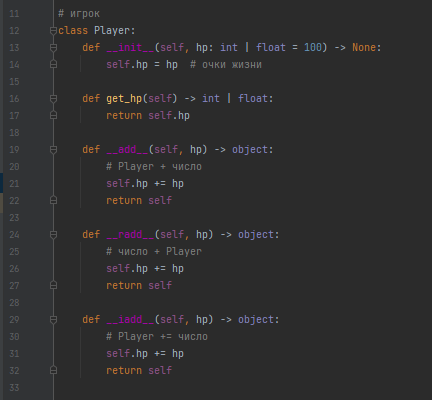


Рис 4.1 - класс Player

Суть магических методов в том, чтобы экземпляры класса могли, по средствам операторов, взаимодействовать с объектами других типов или экземплярами других классов. Все зависит от инструкций которые мы зададим. Операторы в свою очередь обладают способностью полеморфизма, то есть могут обрабатывать разные типы данных.

В нашем случае (рис 4.1), мы имеем возможность с помощью операторов сложения, прибавлять очки жизни нашему персонажу.

Добавим, соответственно, вожможность эти очки отнимать, используя магические методы представленные на рисунке 4.2.

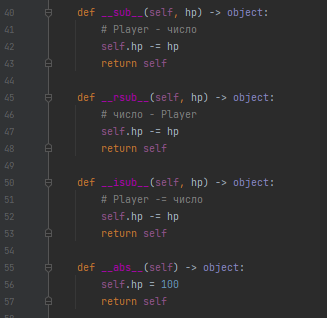


Рис 4.2 - класс Player

Чтобы фиксировать кол-во очков, пропишем метод, который будет их возвращать.

Реализуем скрипт для демонстрации работы магических методов (рис 4.3).

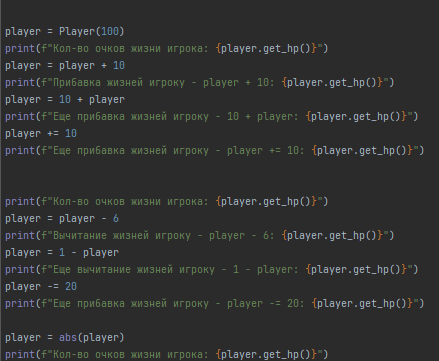


Рис 4.3 - скрипт main.py

Запустим скрипт и проверим его работу (рис 4.4).

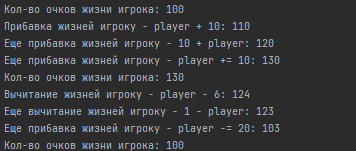


Рис 4.4 - результат скрипта main.py

Скрипт отработал без ошибок.

Далее создадим класс врага - **Enemy** (рис 4.9). Враг также будет иметь очки жизни.

Мы реализуем методы позволяющие сравнивать кол-во жизней нашего игрока с кол-ом жизней врага или другого игрока (рис 4.6 - 4.9). Также нам необходимо иметь возможность сравнить жизни игрока с числом, для этого в методе будем проверять - является ли объект, с которым мы сравниваем игрока, врагом или игроком:

* Если да, то мы сравниваем их очки жизни
* Если нет, сравниваем объект с числом жизней игрока

Для того чтобы узнать сравниваем мы игрока с другим игроком или врагом, создадим базовый класс персонажа - **Character** (рис 4.5). При вызове метода сравнения мы будем проверять является ли экземпляр класса дочерним классом **Character**.

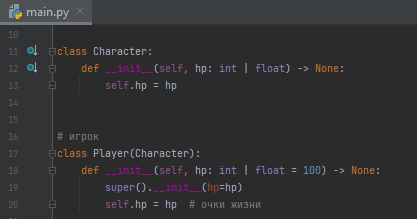


Рис 4.5 - базовый класс Character и дочерний класс Player

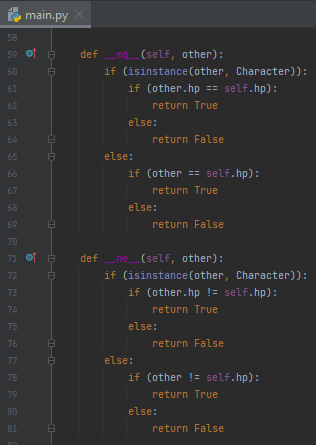


Рис 4.6 - дочерний класс Player

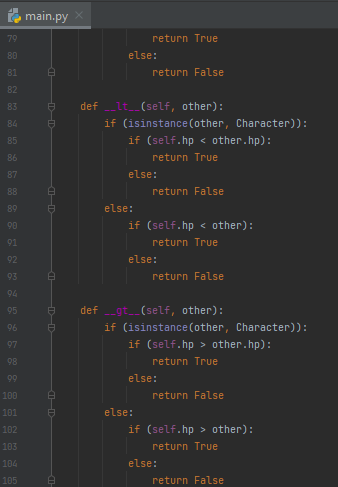


Рис 4.7 - дочерний класс Player

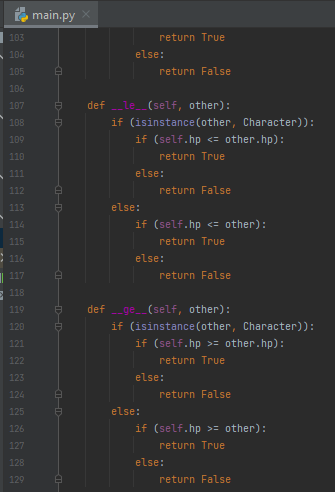


Рис 4.8 - дочерний класс Player

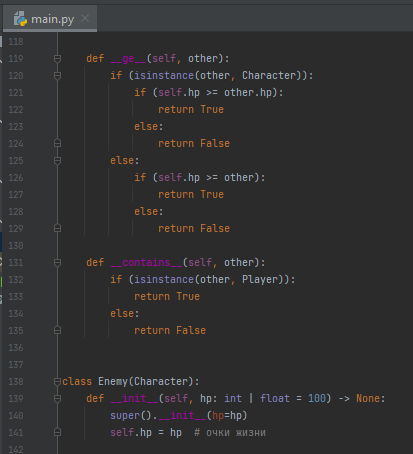


Рис 4.9 - дочерниый класс Player и дочерний класс Enemy

Напишем скрипт для проверки работы методов сравнения (рис 4.10).

Создадим объект enemy, для сравнения очков жизни с игроком.

Далее будем сравнивать жизни игрока с числами.

Результат работы скрипта на рисунке 4.11.

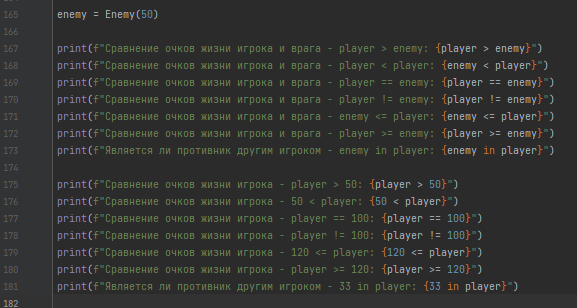


Рис 4.10 - скрипт main.py

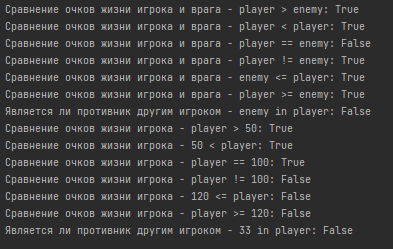


Рис 4.11 - результат скрипта main.py

Все работает корректно.

### Итог

Взяв за пример скрипт из лекции, мы объяснили работу обеспечения полиморфизма методов, а также реализовали несколько вариантов обеспечения полиморфизма методов, показаных в таблице 1 и таблице 2.