# **Полиморфизм и абстрактные методы**

Курс по Python ООП: <https://stepik.org/a/116336>

[Смотреть материал на видео](https://www.youtube.com/watch?v=fzUI3NyJflw&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

На этом занятии я хочу затронуть следующий важный вопрос ООП: что такое полиморфизм и как он реализуется на Python? Вначале вспомним, что

*Полиморфизм – это возможность работы с совершенно разными объектами (языка Python) единым образом.*

Кажется, пока не особо понятно? Поэтому давайте, как всегда, постигнем суть этого подхода на конкретном примере.

Вначале я продемонстрирую пример, где мы увидим один недостаток, который как раз исправляется с помощью полиморфизма. Предположим, у нас есть два класса Rectangle и Square:

**class** Rectangle:     **def** \_\_init\_\_(self, w, h):         self.w = w         self.h = h       **def** get\_rect\_pr(self):         **return** 2\*(self.w+self.h)     **class** Square:     **def** \_\_init\_\_(self, a):         self.a = a       **def** get\_sq\_pr(self):         **return** 4\*self.a

И в них объявлены геттеры get\_rect\_pr() и get\_sq\_pr() для получения периметра соответствующих фигур: прямоугольника и квадрата. Далее, мы можем создать экземпляры этих классов и вывести в консоль значения периметров:

r1 = Rectangle(1, 2) r2 = Rectangle(3, 4)**print**(r1.get\_rect\_pr(), r2.get\_rect\_pr())   s1 = Square(10) s2 = Square(20)**print**(s1.get\_sq\_pr(), s2.get\_sq\_pr())

Все отлично, все работает. Но, теперь предположим, что все эти объекты помещаются в коллекцию:

geom = [r1, r2, s1, s2]

которую можно легко перебрать с помощью цикла for и где бы мы хотели получить значение периметра для каждой фигуры:

**for** g **in** geom:     **print**(g.get\_rect\_pr())

Как вы понимаете, когда в цикле очередь дойдет до объекта s1, возникнет ошибка, т.к. в классе Square отсутствует метод get\_rect\_pr(). Конечно, зная, что в коллекции находятся объекты Rectangle и Square, можно было бы в цикле записать проверку:

**for** g **in** geom:     **if** isinstance(g, Rectangle):         **print**(g.get\_rect\_pr())     **else**:         **print**(g.get\_sq\_pr())

и все заработает. Но у такого кода мало гибкости и, например, при добавлении еще одного класса:

**class** Triangle:     **def** \_\_init\_\_(self, a, b, c):         self.a = a         self.b = b         self.c = c       **def** get\_tr\_pr(self):         **return** self.a + self.b + self.c

Получим снова ошибку:

t1 = Triangle(1,2,3) t2 = Triangle(4,5,6) geom = [r1, r2, s1, s2, t1, t2]

Конечно, в цикле for можно дополнительно проверить на соответствие классам Square и Triangle, но красоты и гибкости нашей программе это не придаст. Вот как раз здесь очень хорошо применим подход, который и называется полиморфизмом. Мы договоримся в каждом классе создавать методы с одинаковыми именами, например,

get\_pr()

Тогда в цикле будем просто обращаться к этому методу и получать периметры соответствующих фигур:

**for** g **in** geom:     **print**( g. get\_pr() )

И это логично, так как каждая ссылка списка ведет на соответствующий объект класса и далее через нее происходит прямой вызов метода get\_pr(). Это и есть пример полиморфизма, когда к разным объектам мы обращаемся через индекс единого списка geom (единый интерфейс), а затем, вызываем геттер get\_pr() соответствующего объекта.

Мало того, мы можем сформировать этот список, сразу создавая в нем объекты соответствующих классов:

geom = [Rectangle(1, 2), Rectangle(3, 4),         Square(10), Square(20),         Triangle(1, 2, 3), Triangle(4, 5, 6)         ]

Мне кажется, так программа выглядит несколько приятнее и читабельнее.

## **Абстрактные методы**

Но у нашей реализации есть один существенный недостаток. Что если мы забудем в каком-либо классе определить метод get\_pr(), например, в Triangle. Тогда, очевидно, программа приведет к ошибке. Как можно было бы этого избежать? Один из вариантов определить базовый класс для классов геометрических примитивов и в нем прописать реализацию геттера get\_pr(), используемую по умолчанию, например, так:

**class** Geom:     **def** get\_pr(self):         **return** -1

А все остальные классы унаследовать от него:

**class** Rectangle(Geom): ...     **class** Square(Geom): ...     **class** Triangle(Geom): ...

Теперь, после запуска программы, для треугольников будем получать значения -1.

Но и это не самое лучшее решение. Все же, нам бы хотелось, чтобы каждый дочерний класс имел бы обязательную реализацию метода get\_pr(). Для этого в геттере get\_pr() мы будем генерировать специальное исключение NotImplementedError, следующим образом:

**class** Geom:     **def** get\_pr(self):         **raise** NotImplementedError("В дочернем классе должен быть переопределен метод get\_pr()")

И если в каком-либо дочернем классе не будет определен метод get\_pr(), то вызовется метод базового класса и выдаст ошибку NotImplementedError, которая будет сигнализировать о том, что метод не переопределен.

Запустим программу и действительно видим это сообщение при попытке вызвать get\_pr() для объектов Triangle. Причем, видя ошибку NotImplementedError, мы понимаем, что она связана именно с необходимостью  переопределения get\_pr(), а не с чем-то другим. В этом плюс такого подхода.

В языках программирования методы, которые обязательно нужно переопределять в дочерних классах и которые не имеют своей собственной реализации называют **абстрактными**. Конечно, в языке Python нет чисто абстрактных методов. Здесь мы лишь выполнили имитацию их поведения, заставляя программиста определять геттер get\_pr() в дочерних классах, самостоятельно генерируя исключение NotImplementedError.

Итак, из этого занятия я, думаю, вы поняли, что из себя представляет полиморфизм и как он реализуется на Python. Также узнали, как можно определять методы, которые ведут себя подобно абстрактным с необходимостью их переопределения в дочерних классах.

Курс по Python ООП: <https://stepik.org/a/116336>