**Аннотация**

*Этот урок посвящен приемам применения наследования: расширению и переопределению методов. Обсуждается и множественное наследование.*

**Расширение методов**

Вернемся к иерархии классов геометрических фигур. И заодно рассмотрим способ, как отразить эту иерархию, представленную в виде картинки, кодом на языке Python.

class Shape:

def describe(self):

# Атрибут \_\_class\_\_ содержит класс или тип объекта self

# Атрибут \_\_name\_\_ содержит строку,

# в которой написано название класса или типа

print(f"Класс: {self.\_\_class\_\_.\_\_name\_\_}")

from math import pi

class Circle(Shape):

def \_\_init\_\_(self, radius):

self.r = radius

def area(self):

return pi \* self.r \*\* 2

def perimeter(self):

return 2 \* pi \* self.r

class Rectangle(Shape):

def \_\_init\_\_(self, a, b):

self.a = a

self.b = b

def area(self):

return self.a \* self.b

def perimeter(self):

return 2 \* (self.a + self.b)

Давайте отнаследуем класс Square от класса Rectangle.

class Square(Rectangle):

pass

side = 5

sq = Square(side, side)

print(sq.area())

print(sq.perimeter())

25

20

Поскольку мы никак не «заполнили» код класса Square, он будет иметь те же самые методы, что были у класса Rectangle. Но это не очень удобно. Мы хотим, чтобы конструктор класса Square принимал на вход один аргумент (длину стороны). Однако конструктор класса Rectangle принимает на вход два аргумента (ширину и высоту). Как быть?

Пока что мы сделали эту логику вручную, с помощью переменной side. Но, коль скоро мы программируем в объектно-ориентированном стиле, давайте «спрячем» (инкапсулируем) эту логику внутрь класса. А именно: мы немного модифицируем конструктор класса Square так, чтобы он принимал на вход только одно число, которое будет передаваться в качестве первого и второго аргумента конструктору базового класса.

**Расширение метода**

Такая процедура, когда метод производного класса дополняет аналогичный метод базового класса, называется **расширением метода**, а в коде это выглядит следующим образом:

class Square(Rectangle):

def \_\_init\_\_(self, size):

print('Создаем квадрат')

super().\_\_init\_\_(size, size)

Функция super() возвращает специальный объект, который делегирует («передает») вызовы методов (в данном случае — метода \_\_init\_\_) от производного класса к базовому. Эту функцию можно вызывать в любом методе класса — в частности, в конструкторе.

Фактически фраза super().\_\_init\_\_(size, size) звучит так: «Вызови метод \_\_init\_\_ у моего базового (родительского) класса».

Давайте проверим, что произойдет, если мы создадим объект класса Square и вызовем методы area() и perimeter():

sq = Square(2)

print(sq.area())

print(sq.perimeter())

print(sq.a)

Создаем квадрат

4

8

2

Как видим, методы area() и perimeter() отработали корректно, и нам не пришлось переписывать эти методы заново — они были полностью наследованы от базового класса, а при создании экземпляра класса была выведена строка, которая при создании элементов базового класса не выводится.

Кроме того, от базового класса унаследовались поля a и b.

Заметим, что расширение можно использовать для любого метода класса, а не только для конструктора \_\_init\_\_. Например:

import datetime

class Greeter:

def greet(self):

print("Good news, everyone")

class GreeterWithDate(Greeter):

def greet(self):

print(datetime.datetime.now())

super().greet()

g = GreeterWithDate()

g.greet()

**Использование методов наследников в базовом классе**

На протяжении этого урока нам пару паз потребовалось вывести на экран небольшое «описание» фигуры — ее периметр и площадь. Поскольку все фигуры имеют для этого общий интерфейс (методы perimeter() и area() соответственно), можно, например, написать универсальную (т. е. **полиморфную**) функцию для этого:

def describe\_shape(shape):

print(f"Периметр: {shape.perimeter()}\nПлощадь: {shape.area()}")

describe\_shape(sq)

Но есть одно неудобство. Что, если на вход этой функции подать переменную неправильного типа? Программа завершится с ошибкой:

describe\_shape(5)

-------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-12-398f18afe0b6> in <module>()

----> 1 describe\_shape(5)

<ipython-input-10-fafe33c63281> in describe\_shape(shape)

1 def describe\_shape(shape):

----> 2 print(f"Периметр: {shape.perimeter()}\nПлощадь: {shape.area()}")

AttributeError: 'int' object has no attribute 'perimeter'

Конечно, внутри describe\_shape можно добавить необходимые проверки, но есть более правильное решение — нужно добавить соответствующий метод в базовый класс. В нашем случае можно просто немного дополнить метод describe класса Shape:

class Shape:

def describe(self):

# Добавим еще и название класса

print(f"Класс: {self.\_\_class\_\_.\_\_name\_\_}\n"

f"Периметр: {self.perimeter()}\n"

f"Площадь: {self.area()}")

Обратите внимание: у класса Shape нет методов perimeter() и area(), поэтому метод describe() не будет работать для объектов этого класса. Но у всех производных классов эти методы есть, поэтому для них все сработает правильно:

sq = Square(3)

sq.describe()

Создаем квадрат

Класс: Square

Периметр: 12

Площадь: 9

**Переопределение методов**

Давайте «починим» метод describe() для класса Shape. Будем считать, что у «абстрактной» фигуры площадь и периметр не определены (т. е. равны None):

class Shape:

def describe(self):

print(f"Класс: {self.\_\_class\_\_.\_\_name\_\_}\n"

f"Периметр: {self.perimeter()}\n"

f"Площадь: {self.area()}")

def area(self):

return None

def perimeter(self):

return None

А как теперь будет работать метод describe() для производных классов? У какого класса он будет вызывать методы area() и perimeter() — у производного или у базового?

Давайте вспомним, что по сути представляет собой наследование классов в Python: если мы вызовем метод у производного класса, сперва ищется метод этого класса, а если его там нет, такой же поиск выполняется в его базовом классе. Значит, поведение производных классов измениться не должно.

Давайте убедимся в этом:

shape = Shape()

circle = Circle(5)

rectangle = Rectangle(3, 4)

square = Square(5)

shape.describe()

circle.describe()

rectangle.describe()

square.describe()

Создаем квадрат

Класс: Shape

Периметр: None

Площадь: None

Класс: Circle

Периметр: 31.41592653589793

Площадь: 78.53981633974483

Класс: Rectangle

Периметр: 14

Площадь: 12

Класс: Square

Периметр: 20

Площадь: 25

Итак, методы perimeter() и area() есть в базовом классе, но в производных классах они реализованы по-другому.

**Переопределение методов**

Это называется **переопределением методов**. В отличие от расширения методов, в данном случае метод area() базового класса **не используется** при реализации метода area() производного класса; то же самое относится и к методу perimeter().

**Множественное наследование**

**Множественное наследование**

Python предоставляет возможность наследоваться сразу от нескольких классов. Такой механизм называется **множественное наследование**, и он позволяет вызывать в производном классе методы разных базовых классов.

Рассмотрим пример:

class Base1:

def tic(self):

print("tic")

class Base2:

def tac(self):

print("tac")

class Derived(Base1, Base2):

pass

d = Derived()

d.tic() # метод, наследованный от Base1

d.tac() # метод, наследованный от Base2

Множественное наследование на практике используется достаточно редко (хотя все же используется), поскольку при его использовании возникают закономерные вопросы:

* Что, если названия каких-то методов в базовых классах совпадают?
* Какой из них будет вызван из производного класса?

Хотя в языке и зафиксирован порядок разрешения таких конфликтов (в общем случае классы просматриваются слева направо — подробнее в [документации](https://docs.python.org/3/tutorial/classes.html#multiple-inheritance)), эта особенность все равно может привести к ошибкам при использовании множественного наследования.

В нашей иерархии классов геометрических фигур можно привести следующий пример множественного наследования. Мы знаем, что квадрат является не только прямоугольником, но и правильным многоугольником. В любой правильный многоугольник, например, можно вписать окружность, а в произвольный прямоугольник нельзя. Давайте напишем отдельный класс RegularPolygon для правильных многоугольников:

from math import tan, pi

class RegularPolygon:

def \_\_init\_\_(self, side, n):

self.side = side # длина стороны

self.n = n # число сторон

def inscribed\_circle\_radius(self):

'''Радиус вписанной окружности'''

return self.side / (2 \* tan(pi / self.n))

Квадрат можно отнаследовать от прямоугольника и правильного многоугольника. Обратите внимание на конструктор класса Square:

class Square(Rectangle, RegularPolygon):

def \_\_init\_\_(self, a):

# Приходится явно вызывать конструкторы базовых классов

Rectangle.\_\_init\_\_(self, a, a)

RegularPolygon.\_\_init\_\_(self, a, 4)

s = Square(5)

s.describe() # метод класса Rectangle

print(s.inscribed\_circle\_radius()) # метод класса RegularPolygon