ООП. Наследование

План урока

1. Объяснение понятия наследования
2. Синтаксис наследования в Python
3. Наследование методов
4. Расширение методов
5. Использование методов наследников в базовом классе
6. Переопределение методов
7. Множественное наследование

Аннотация

В этом уроке объясняется понятие наследования, показывается его связь с уже известными понятиями объектно-ориентированного программирования (инкапсуляция и полиморфизм), описывается синтаксис и семантика наследования в Python. Мы рассмотрим приемы применения наследования: расширение и переопределение методов. Также обсуждается множественное наследование.

Мы уже знакомы с основами объектно-ориентированного программирования. Мы умеем определять классы и создавать объекты. Также мы знаем, что хорошо продуманный набор методов (иначе говоря, **интерфейс**) позволяет добиться **инкапсуляции** (т.е. скрытия информации о внутреннем устройстве объекта) и **полиморфизма** (т.е. возможности писать код, работающий одинаково с разными классами).

Часто бывает так, что классы в программе имеют не только общий интерфейс, но и похожую реализацию. Для примера рассмотрим несколько классов, представляющих геометрические фигуры: круг (`Circle`) и прямоугольник (`Rectangle`). Пусть интерфейс этих классов пока состоит из единственного метода `area()`, возвращающего площадь фигуры.

```
from math import pi


class Circle:
    def __init__(self, radius):
        self.r = radius

    def area(self):
        return pi * self.r**2


class Rectangle:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def area(self):
        return self.a * self.b
```

Допустим теперь, что нам нужно реализовать класс для квадрата (`Square`). Конечно, мы можем сделать это непосредственно:

```
class Square:
    def __init__(self, a):
        self.a = a

    def area(self):
        return self.a * self.a
```

Но мы знаем, что квадрат -- это частный случай прямоугольника, у которого ширина равна высоте. Чем нам это может помочь?

Представьте, что мы хотим добавить в интерфейс наших классов еще один метод `perimeter()` для вычисления периметра. Сейчас нам нужно добавить его во все три класса -- но это лишняя работа, ведь периметр квадрата вычисляется так же, как и периметр прямоугольника. А любая лишняя работа не только отнимает время у программиста, но и увеличивает вероятность допустить ошибку. Конечно, вы вряд ли ошибетесь в написании метода `perimeter()` для квадрата, ведь это займет одну строку; но в реальности методы классов могут занимать десятки строк (в плохих программах -- сотни), а логика работы этих методов гораздо сложнее.

Резюмируем наши наблюдения:

- Класс `Square` является частным случаем класса `Rectangle`.
- Если бы имелся способ явно запрограммировать это отношение, то наш код получился более коротким и, что важно, *согласованным* (если периметр прямоугольника вычисляется правильно, то периметр квадрата *автоматически* вычисляется правильно). Такой способ, конечно, существует -- он и называется "наследование".

Итак, дадим определение:

Наследование -- это механизм, позволяющий запрограммировать отношение вида "класс `B` является частным случаем класса `A`". В этом случае класс `A` также называется **базовым** классом, а `B` -- **производным** классом.

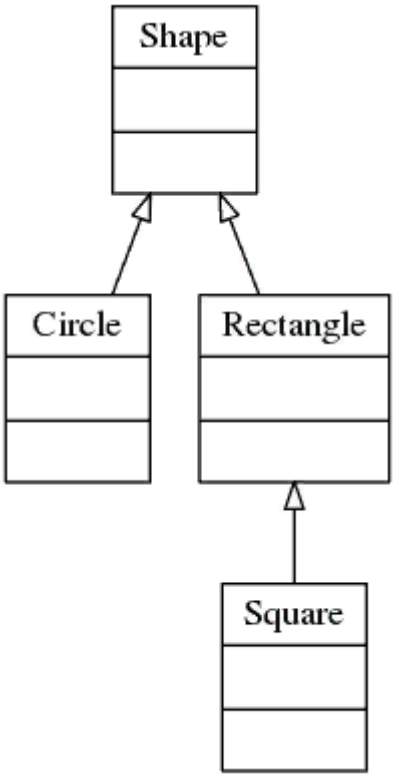
Наследование является способом переиспользования кода между классами без необходимости нарушения инкапсуляции. Это достигается за счет того, что производный класс может пользоваться атрибутами и методами базового класса (иными словами, производный класс "наследует" атрибуты и методы базового класса).

Наследование -- важная концепция объектно-ориентированного программирования наряду с инкапсуляцией и полиморфизмом.

Используя отношение "частный случай", можно строить иерархии классов. Добавим к нашим классам еще один класс, представляющий произвольную геометрическую фигуру (`Shape`):

```
class Shape:
    pass
```

Тогда иерархия классов будет выглядеть так (заодно посмотрим, как можно вывести изображение с помощью Python):



В этом и следующем уроках мы узнаем, как отразить эту иерархию в коде.

Чтобы наследовать класс `B` от класса `A` (то есть, чтобы запрограммировать отношение "класс `B` является частным случаем класса `A`"), нужно написать так:

```
class A:
    pass


class B(A):
    pass
```

Вопрос: Что происходит, когда класс `B` наследуется от класса `A` (т.е. когда мы пишем `class B(A)`)?

Ответ: Меняется процедура поиска методов и атрибутов в классе `B`.

Разберем, как именно.

Мы знаем, что у "простого" объекта (который ни от кого не наследуется) нельзя вызвать несуществующий метод или прочитать несуществующий атрибут:

```
class C:
    def foo(self):
        print('foo')


c = C()
c.foo()  # ok
c.bar()  # error
```

```
foo

---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
<ipython-input-4-de63cbd4e47c> in <module>()
      6 c = C()
      7 c.foo()  # ok
----> 8 c.bar()  # not ok

AttributeError: 'C' object has no attribute 'bar'
```

Но если класс наследован от другого класса, то проверка существования метода (или атрибута) осуществляется так:

- сперва метод ищется в исходном (производном) классе
- если его там нет, он ищется в базовом классе
- предыдущие шаги повторяются до тех пор, пока метод не будет найден, или пока процедура не дойдет до класса, который ни от кого не наследуется

А это означает, что производному классу доступны не только собственные методы, но и методы базового класса. В этом случае говорят, что производный класс "наследует" методы базового класса.

```
class BaseC:
    def bar(self):
        print('bar')


class C(BaseC):
    def foo(self):
        print('foo')


c = C()
c.foo()  # ok    -- этот метод есть в производном классе
c.bar()  # ok    -- этот метод есть в базовом классе
c.baz()  # error -- этого метода нет ни в производном, ни в базовом классе
```

Как видим, в производные классы можно не только добавлять новые методы, но и пользоваться методами базового класса. В этом и состоит польза наследования методов -- оно позволяет обойтись без дублирования кода и при этом не нарушает инкапсуляцию.

Рассмотрим механизм наследования подробнее на примере нашей иерархии геометрических фигур.

Для начала исследуем, как производные классы могут пользоваться методами базового класса. Для этого реализуем в классе `Shape` метод `describe`, который будет печатать название собственного класса:

```
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("Класс: {}".format(self.__class__.__name__))
```

Отнаследуем от `Shape` классы `Circle` и `Rectangle` и убедимся, что метод `describe` работает и для них тоже:

```
from math import pi


class Circle(Shape):
    def __init__(self, radius):
        self.r = radius

    def area(self):
        return pi * self.r**2


class Rectangle(Shape):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def area(self):
        return self.a * self.b


shape = Shape()
shape.describe()

circle = Circle(1)
circle.describe()

rectangle = Rectangle(1, 2)
rectangle.describe()
```

```
Класс: Shape
Класс: Circle
Класс: Rectangle
```

Теперь рассмотрим, как добавить в производный класс новый метод, которого нет в базовом классе.

А именно, добавим в класс `Circle` метод `square`, который решает знаменитую задачу **квадратуры круга** -- возвращает квадрат (в нашем случае -- объект `Rectangle` с равными сторонами), который по площади равен площади исходного круга. Математики много столетий бились над задачей квадратуры круга и в итоге доказали, что такое построение нельзя выполнить с помощью циркуля и линейки.

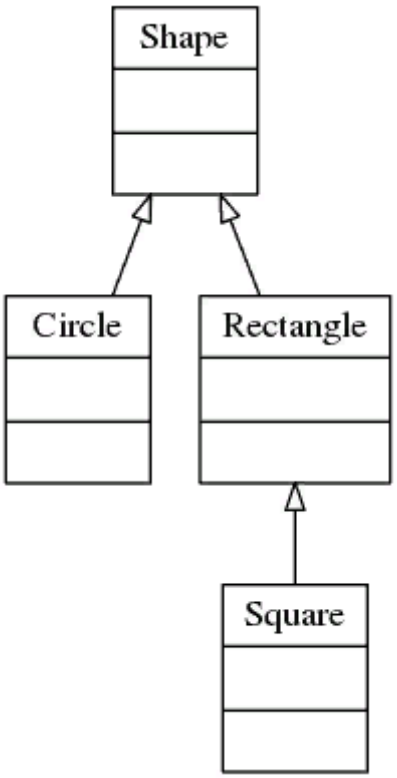
Зато это можно сделать с помощью Python:

```
class Circle(Shape):
    def __init__(self, radius):
        self.r = radius

    def area(self):
        return pi * self.r**2

    def square(self):
        side = pi**0.5 * self.r
        return Rectangle(side, side)


circle = Circle(1)
square = circle.square()

print("Площадь круга:    {}".format(circle.area()))
print("Площадь квадрата: {}".format(square.area()))
print("Радиус круга: {}".format(circle.r))
print("Длина стороны квадрата: {}".format(square.a))
```

```
Площадь круга:    3.141592653589793
Площадь квадрата: 3.1415926535897927
Радиус круга: 1
Длина стороны квадрата: 1.7724538509055159
```

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("Класс: {}".format(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 унаследуем 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`, то 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, то 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.

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 полностью отнаследовались 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 и поля `a` и `b`.

Заметим, что расширение можно использовать для любого метода класса, а не только для конструктора `__init__`.

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), то можно, например, написать универсальную (т.е. **полиморфную**) функцию для этого:

```
def describe_shape(shape):
    print("Периметр: {}\nПлощадь: {}".format(shape.perimeter(), 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("Периметр: {}\nПлощадь: {}".format(shape.perimeter(), 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):
        print("Класс: {}\nПериметр: {}\nПлощадь: {}".format(
            # Добавим еще и название класса
            self.__class__.__name__,
            self.perimeter(),
            self.area()))
```

Обратите внимание, что 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("Класс: {}\nПериметр: {}\nПлощадь: {}".format(
            self.__class__.__name__, self.perimeter(), 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, то сперва ищется метод этого класса, а если его там нет, то 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. Это называется **переопределением методов**. В отличии 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

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 Base1
```

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?
- И 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 еще 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
```