# **Наследование в объектно-ориентированном программировании**

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

[Смотреть материал на видео](https://www.youtube.com/watch?v=7WVYqjdMa6U&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

На этом занятии мы начнем изучение важной темы в ООП – наследовании. Давайте предположим, что у нас в программе объявлены два независимых класса:

**class** Geom:     name = 'Geom'     **class** Line:     **def** draw(self):         **print**("Рисование линии")

Как вы понимаете, если сейчас создать экземпляр класса Geom:

g = Geom()

то мы сможем обратиться к свойству name:

**print**(g.name)

но не сможем вызвать метод draw() класса Line, так как это совершенно другой, независимый класс. И то же самое для объекта Line:

l = Line() l.draw()

Можно вызвать его метод draw(), но нельзя обратиться к свойству name другого класса Geom. То есть, эти классы, пространства имен совершенно независимы между собой. Но, при необходимости, мы можем установить связь между ними и, например, сделать так, чтобы открытые атрибуты класса Geom были доступны в классе Line. Записывается такая связка, следующим образом:

**class** Line(Geom):     **def** draw(self):         **print**("Рисование линии")

В круглых скобках мы указали класс Geom, который собираемся расширять классом Line и все, что доступно в классе Geom автоматически становится доступным и в классе Line. В частности, мы теперь можем совершенно спокойно обратиться к свойству name через объект класса Line:

**print**(l.name)

Такая конструкция, когда один класс определяется на основе другого, называется **наследованием**. Причем, класс Geom называется **родительским** или **базовым**, а класс Line – **подклассом родительского** или **дочерним классом**.

Графически иерархию наследования классов принято изображать в виде прямоугольников, соединенных стрелками, причем стрелка направлена от дочернего к базовому классу:

Для чего может понадобиться наследование? Давайте добавим в класс Line метод set\_coords для записи координат в объект-линию:

**def** set\_coords(self, x1, y1, x2, y2):         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2

А, затем, определим еще один подобный класс для прямоугольников:

**class** Rect(Geom):     **def** set\_coords(self, x1, y1, x2, y2):         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2       **def** draw(self):         **print**("Рисование прямоугольника")

Как видите, у нас здесь налицо дублирование кода. И оно будет быстро нарастать с увеличением классов для различных геометрических фигур. Чтобы этого не было, мы можем общее для всех дочерних классов (от Geom) вынести в базовый и, в частности, записать в нем метод set\_coords. В итоге получим, следующие определения классов:

**class** Geom:     name = 'Geom'       **def** set\_coords(self, x1, y1, x2, y2):         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2     **class** Line(Geom):     **def** draw(self):         **print**("Рисование линии")     **class** Rect(Geom):     **def** draw(self):         **print**("Рисование прямоугольника")

В результате мы устранили дублирование кода и можем вызывать метод set\_coords из базового класса через объекты дочерних классов:

l = Line() r = Rect() l.set\_coords(1, 1, 2, 2) r.set\_coords(1, 1, 2, 2)

А иерархия наследования принимает вид:

Давайте теперь внимательнее посмотрим, как отрабатывает метод set\_coords базового класса Geom. Во-первых, когда происходит его вызов, например, через объект класса Line:

l.set\_coords(1, 1, 2, 2)

то метод с таким названием сначала ищется в классе Line и если не находится, то поиск продолжается в базовых классах. В данном случае – это класс Geom. В нем этот метод обнаруживается и запускается.

Во-вторых, при вызове метода set\_coords() из базового класса его параметр self будет ссылаться на объект, через который этот метод был вызван. То есть, на объект класса Line. Вот это очень важный момент! Параметр self в базовых классах может ссылаться не только на объекты этого же класса, но и на объекты производных (дочерних) от него классов, как в нашем случае – на объект класса Line. Почему это важно? Смотрите, формально сейчас мы можем в методе set\_coords() вызвать метод дочернего класса draw():

**def** set\_coords(self, x1, y1, x2, y2):         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2         self.draw()

И после запуска программы увидим, что сначала была нарисована линия, а затем, прямоугольник. Потому что сначала вызывался метод set\_coords() через объект класса Line, а потом – через объект класса Rect. Но такой вызов метода draw() таит в себе потенциальную ошибку. Если мы создадим объект базового класса:

g = Geom()

а, затем, вызовем все тот же метод set\_coords():

g.set\_coords(0, 0, 0, 0)

то при выполнении программы получим ошибку, что метод draw() не определен. Действительно, теперь параметр self метода set\_coords() ссылается на объект базового класса Geom, а в нем метод draw отсутствует. Отсюда и получается такая ошибка. Поэтому, при реализации методов в классах следует придерживаться одного простого правила: внутри них обращаться только к разрешенным атрибутам либо внутри текущего класса, либо базовых классов. Но не дочерних.

Я уберу вызов метода draw() в методе set\_coords() и теперь никаких ошибок у нас не возникает.

Похожим образом происходит и обращение к свойствам базовых классов. Сейчас в классе Geom прописано свойство name и, как мы видели, к нему можно совершенно спокойно обращаться из объектов дочерних классов:

**print**(l.name)**print**(r.name)

То есть, если указанное свойство не находится в соответствующем дочернем классе, то поиск продолжается в базовых. Однако, если свойство name прописать непосредственно в дочернем классе, например, Line:

**class** Line(Geom):     name = 'Line'       **def** draw(self):         **print**("Рисование линии")

то для объектов-линий оно будет взято непосредственно из класса Line, а для прямоугольников, по-прежнему, из базового класса Geom.

Кстати, когда мы определяем какой-либо существующий атрибут в дочернем классе – это называется **переопределением**. То есть, мы переопределили атрибут name, который присутствует в базовом классе Geom и интегрированная среда нам об этом сигнализирует стрелочкой вверх.

То же самое происходит и при переопределении методов. Давайте добавим в класс Geom метод draw():

**def** draw(self):         **print**("Рисование примитива")

и видим, что у этого метода в базовом классе нарисована стрелочка вниз, а в дочерних – стрелочка вверх. Это и есть обозначения переопределения методов.

Работают они очевидным образом. Если draw() вызвать из дочерних объектов:

l.draw() r.draw()

то будут вызваны соответствующие методы дочерних классов. А если мы вызываем draw() из базового класса:

g = Geom() g.draw()

то будет вызван метод базового класса. Разумеется, если в каком-либо дочернем классе убрать метод draw(), например, в прямоугольнике, то для него будет вызван метод уже базового класса.

На этом мы завершим наше первое знакомство с механизмом наследования. Надеюсь все описанные моменты вам здесь были понятны и если это так, то переходите к следующему занятию, где будем продолжать изучение этой важной темы.

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