# **Наследование. Функция super() и делегирование**

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

[Смотреть материал на видео](https://www.youtube.com/watch?v=BTV9esoCwEE&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

Мы продолжаем изучение темы «наследование». Это занятие я начну с простой, но важной терминологии. Сморите, если у нас имеется некий базовый класс Geom и мы создаем дочерний класс Line, в котором дополнительно прописан метод draw(), то это называется **расширением** (extended) базового класса:

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

Как правило, дочерние создаются именно для расширения функциональности базовых классов. Однако, если в классе Geom также прописать метод draw():

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

то теперь класс Line лишь **переопределяет** (overriding) поведение базового класса, не меняя его принцип функционирования. Поэтому, когда говорят о расширении, то подразумевают добавление новых атрибутов в дочерних классах, а при переопределении (обычно методов) – изменение поведения уже существующего функционала.

## **Функция super() и делегирование**

Давайте теперь пропишем инициализатор в базовом классе Geom (метод draw уберем):

**class** Geom:     name = 'Geom'       **def** \_\_init\_\_(self):         **print**("инициализатор Geom")

А ниже создадим экземпляр класса Line:

l = Line()

После запуска программы увидим ожидаемое поведение – был автоматически вызван инициализатор базового класса. В действительности, здесь происходит следующая последовательность вызовов магических методов. Сначала вызывается \_\_call\_\_(), который, в свою очередь, последовательно вызывает метод \_\_new\_\_() для создания экземпляра класса, а затем, метод \_\_init\_\_() для его инициализации. Так вот, все эти методы вызываются из дочернего класса Line. Если какой-либо из них не находится, то поиск продолжается в родительских классах в порядке иерархии наследования. Например, метод \_\_new\_\_() в данном случае будет взят из метакласса type, который неявно вызывается при создании классов (подробнее о метаклассах мы еще будем говорить). А вот метод \_\_init\_\_() мы прописали в классе Geom, поэтому будет вызван именно он. Причем, параметр self в этом методе будет ссылаться на созданный объект класса Line. Об этом мы с вами уже говорили и это следует помнить. Параметр self в методах класса всегда ссылается на объект, из которого метод был вызван.

Отлично, я думаю в целом схема вызова методов в момент создания экземпляров классов, понятна. И в соответствии с ней, если мы определим инициализатор в классе Line, то именно он и должен вызваться. Давайте это сделаем:

**class** Line(Geom):     **def** \_\_init\_\_(self):         **print**("инициализатор Line")       **def** draw(self):         **print**("Рисование линии")

Запустим программу и теперь видим, что действительно, вызывается именно метод \_\_init\_\_ класса Line. Я перепишу его со следующими параметрами:

**def** \_\_init\_\_(self, x1, y1, x2, y2):         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2

Это координаты начала и конца линии на плоскости. Соответственно, при создании объектов этого класса, мы теперь должны передавать аргументы:

l = Line(0, 0, 10, 20)

Все работает и никаких проблем у нас нет. Но, давайте теперь добавим еще один класс Rect для прямоугольников:

**class** Rect(Geom):     **def** \_\_init\_\_(self, x1, y1, x2, y2, fill=None):         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2         self.fill = fill       **def** draw(self):         **print**("Рисование прямоугольника")

Смотрите, у нас получилось дублирование кода. Это очень нехорошо. Но мы знаем, как это можно поправить. Давайте общее этих методов вынесем в базовый класс Geom:

**class** Geom:     name = 'Geom'       **def** \_\_init\_\_(self, x1, y1, x2, y2):         **print**(f"инициализатор Geom для {self.\_\_class\_\_}")         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2

В дочернем классе Line уберем инициализатор, т.к. он полностью повторяется в Geom, а класс Rect запишем в виде:

**class** Rect(Geom):     **def** \_\_init\_\_(self, x1, y1, x2, y2, fill=None):         **print**("инициализатор Rect")         self.fill = fill       **def** draw(self):         **print**("Рисование прямоугольника")

Ниже создадим экземпляры обоих классов:

l = Line(0, 0, 10, 20) r = Rect(1, 2, 3, 4)

После запуска увидим следующее:

инициализатор Geom для <class '\_\_main\_\_.Line'>  
инициализатор Rect

О чем это говорит? Для класса Line был вызван инициализатор в базовом классе Geom, а для класса Rect не вызывался – только инициализатор самого класса. И это логично, так как метод \_\_init\_\_() был найден в Rect и дальше цепочка поиска не продолжалась. Но нам же нужно при создании примитивов также вызывать инициализатор и базового класса Geom. Как это сделать? Конечно, мы могли бы явно указать имя базового класса Geom и вызвать через него магический метод \_\_init\_\_() в инициализаторе класса Rect:

**def** \_\_init\_\_(self, x1, y1, x2, y2, fill=None):         Geom.\_\_init\_\_(self, x1, y1, x2, y2)         **print**("инициализатор Rect")         self.fill = fill

Но явно указывать имена базовых классов не лучшая практика, так как имена и иерархия наследования могут меняться. Поэтому в Python для обращения к базовому классу используется специальная функция super():

**def** \_\_init\_\_(self, x1, y1, x2, y2, fill=None):         super().\_\_init\_\_(x1, y1, x2, y2)         **print**("инициализатор Rect")         self.fill = fill

Она возвращает ссылку на, так называемый, объект-посредник, через который происходит вызов методов базового класса.

Теперь, при запуске программы мы видим, что был вызван инициализатор сначала класса Geom, а затем, для Rect. Такое обращение к переопределенным методам базового класса с помощью функции super() называется **делегированием**. То есть, мы делегировали вызов инициализатора класса Geom, чтобы он создал в нашем объекте локальные свойства с координатами углов прямоугольника. Причем, вызов метода \_\_init\_\_() базового класса лучше делать в первой же строчке, чтобы он случайно не переопределял какие-либо локальные свойство в дочернем классе. Например, если в базовом \_\_init\_\_() дополнительно прописать:

**def** \_\_init\_\_(self, x1, y1, x2, y2):         **print**(f"инициализатор Geom для {self.\_\_class\_\_}")         self.x1 = x1         self.y1 = y1         self.x2 = x2         self.y2 = y2         self.fill = 0

А в дочернем его вызвать в последнюю очередь:

**class** Rect(Geom):     **def** \_\_init\_\_(self, x1, y1, x2, y2, fill=None):         **print**("инициализатор Rect")         self.fill = fill         super().\_\_init\_\_(x1, y1, x2, y2)

то, очевидно, свойство fill будет неявно переопределено при вызове \_\_init\_\_() базового класса:

**print**(r.\_\_dict\_\_)

Подобные причины и рекомендуют делать вызов инициализатора базового класса в первой же строчке:

**class** Rect(Geom):     **def** \_\_init\_\_(self, x1, y1, x2, y2, fill=None):         super().\_\_init\_\_(x1, y1, x2, y2)         **print**("инициализатор Rect")         self.fill = fill

Теперь у нас нет проблем с определением локального свойства fill.

Надеюсь, из этого занятия вы узнали о понятиях расширения классов и переопределения их методов, разобрались с работой функции super(), а также делегированием методов, то есть их вызовом из базового класса через функцию super().

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