# **Множественное наследование**

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

[Смотреть материал на видео](https://www.youtube.com/watch?v=9YPooWY6x9o&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

Мы продолжаем изучать тему наследования. В языке Python допускается множественное наследование, когда один дочерний класс образуется сразу от нескольких базовых, согласно синтаксису:

class A(base1, base2, …, baseN):  
…

Давайте посмотрим, как это работает и для чего вообще нужно. Признаюсь честно, я не сразу смог придумать учебный пример, где можно было бы показать целесообразность применения множественного наследования. То есть, он применяется не так часто, как обычное наследование от одного класса. Но, тем не менее это тоже важный механизм и некоторые подходы к программированию его активно используют. Например, идея миксинов (mixins) в Python реализуется через множественное наследование. И учебный пример я решил связать именно с ними, а заодно показать еще один прием в программировании.

Давайте представим, что мы делаем интернет-магазин по продаже товаров, например, ноутбуков, дисков, процессоров и т.п. Каждый товар будет определяться своим классом, а общим (базовым) для них всех будет класс Goods – товары. Также у каждого товара обязательно будут поля:

* name – наименование
* weight – вес
* price – цена

Для примера запишем в нашей программе базовый класс Goods:

**class** Goods:     **def** \_\_init\_\_(self, name, weight, price):         **print**("init MixinLog")         self.name = name         self.weight = weight         self.price = price       **def** print\_info(self):         **print**(f"{self.name}, {self.weight}, {self.price}")

И один дочерний класс для ноутбуков:

**class** NoteBook(Goods):     **pass**

Затем, мы можем создать объект для ноутбука:

n = NoteBook("Acer", 1.5, 30000)

и распечатать информацию о нем:

n.print\_info()

Пока ничего нового здесь нет. Но потом, к нам подходит тимлид и говорит:

- Дорогой сеньор, добавь, пожалуйста, возможность логирования товаров магазина.

И как бы вы поступили на месте этого сеньора? Плохой сеньор начнет прописывать логику логирования либо непосредственно в базовом классе Goods, либо уровнем выше (в иерархии наследования). А хороший воспользуется идеей миксинов. Для этого он создаст еще один класс, который можно назвать:

**class** MixinLog:     ID = 0       **def** \_\_init\_\_(self):         **print**("init MixinLog")         self.ID += 1         self.id = self.ID       **def** save\_sell\_log(self):         **print**(f"{self.id}: товар продан в 00:00 часов")

Этот класс работает совершенно независимо от классов Goods и Notebook и лишь добавляет функционал по логированию товаров с использованием их id. Такие независимые базовые классы и получили название миксинов – примесей.

Добавим этот класс в цепочку наследования:

**class** NoteBook(Goods, MixinLog):     **pass**

а ниже вызовем метод save\_sell\_log():

n.save\_sell\_log()

И видим ошибку. Очевидно, она связана с тем, что у второго класса MixinLog не был вызван инициализатор. Почему так произошло? Как мы уже знаем, при создании объектов инициализатор ищется сначала в дочернем классе, но так как его там нет, то в первом базовом Goods. Он там есть, выполняется и на этом инициализация нашего объекта NoteBook завершается. Однако, нам нужно также взывать инициализатор и второго базового класса MixinLog. В данном случае, сделать это можно с помощью объекта-посредника super(), которая и делегирует вызов метода \_\_init\_\_ класса MixinLog:

**class** Goods:     **def** \_\_init\_\_(self, name, weight, price):         super().\_\_init\_\_()         **print**("init Goods")         self.name = name         self.weight = weight         self.price = price …

Теперь, после запуска программы, мы видим, что оба инициализатора сработали и ошибок никаких нет.

Но откуда функция super() «знает», что нужно обратиться ко второму базовому классу MixinLog, а, скажем, не к базовому классу object, от которого неявно наследуются все классы верхнего уровня? В Python существует специальный алгоритм обхода базовых классов при множественном наследовании. Сокращенно, он называется:

MRO – Method Resolution Order

И говорит, в каком порядке обходить базовые классы:

Мы можем увидеть эту цепочку обхода базовых классов, если распечатать специальную коллекцию \_\_mro\_\_ любого класса:

**print**(NoteBook.\_\_mro\_\_)

В консоли появится следующая последовательность:

(<class '\_\_main\_\_.NoteBook'>, <class '\_\_main\_\_.Goods'>, <class '\_\_main\_\_.MixinLog'>, <class 'object'>)

То есть, методы сначала ищутся в самом классе NoteBook, затем, в классе Goods, далее идет класс MixinLog и от него уже идет к классу object. Это цепочка обхода для нашего конкретного примера. При другой иерархии наследования эта цепочка может быть другой, но одно всегда неизменно –первый базовый класс, указанный при наследовании, выбирается первым (после дочернего, разумеется). И это важный момент. Вы всегда можете быть уверены, что инициализатор первого базового класса сработает в первую очередь. Почему это важно? Смотрите, при создании объекта NoteBook мы передаем ему три аргумента. Эти три аргумента, затем, передаются в инициализатор. И так как первым будет вызван инициализатор класса Goods, то мы уверены, что эти аргументы будут переданы именно в него, а не в какой-то другой инициализатор других базовых классов. И какая бы цепочка наследования у нас ни была, все равно первым будет вызываться метод \_\_init\_\_ класса Goods, потому что он записан первым. Это гарантирует работоспособность нашей программы при разных иерархиях множественного наследования.

Ради интереса, давайте поменяем местами базовые классы:

**class** NoteBook(MixinLog, Goods):     **pass**

И мы сразу получаем ошибку, что в метод \_\_init\_\_ передаются четыре аргумента, а он принимает только один, так как здесь отрабатывает инициализатор уже класса MixinLog. Так что порядок следования базовых классов при множественном наследовании имеет важное значение. Первым должен идти «основной» класс и у него, как правило, инициализатор имеет несколько параметров. А далее, записываются классы, у которых, опять же, как правило, инициализаторы имеют только параметр self. Это второй важный момент. Когда мы собираемся использовать множественное наследование, то структуру классов следует продумывать так, чтобы инициализаторы вспомогательных классов имели только один параметр self, иначе будут сложности их использования. В чем они состоят?

Давайте для примера пропишем в инициализаторе класса MixinLog один параметр p1

**class** MixinLog:     **def** \_\_init\_\_(self, p1):         super().\_\_init\_\_(1, 2)         …

И объявим еще один класс миксинов, где в инициализаторе два параметра:

**class** MixinLog2:     **def** \_\_init\_\_(self, p1, p2):         super().\_\_init\_\_()         **print**("init MixinLog 2")

В каждом методе \_\_init\_\_ мы также делаем делегированный вызов инициализатора следующего базового класса. А цепочка наследования будет такой:

**class** NoteBook(Goods, MixinLog, MixinLog2):     **pass**

Сейчас при запуске у нас не возникает никаких ошибок, так как последовательность MRO имеет вид:

(<class '\_\_main\_\_.NoteBook'>, <class '\_\_main\_\_.Goods'>, <class '\_\_main\_\_.MixinLog'>, <class '\_\_main\_\_.MixinLog2'>, <class 'object'>)

То есть, мы знаем, что функция super() в классе Goods вызовет метод \_\_init\_\_ класса MixinLog с одним дополнительным параметром, а затем, метод \_\_init\_\_ класса MixinLog2 с двумя дополнительными параметрами. И мы «жестко» это прописали. Но, как вы понимаете, если хотя бы немного изменится цепочка наследования, например, так:

**class** NoteBook(Goods, MixinLog2, MixinLog):     **pass**

то все нарушится и получим ошибки. Чтобы в программах при множественном наследовании не возникало проблем с зависимостью последовательности наследования дополнительных базовых классов, их инициализаторы следует создавать с одним параметром self и в каждом из них прописывать делегированный вызов инициализатора следующего класса командой:

super().\_\_init\_\_()

Тогда точно никаких особых проблем при использовании множественного наследования не возникнет.

Последнее, что я хочу отметить на этом занятии, это вызов методов с одинаковыми именами из базовых классов. Давайте предположим, что в классе MixinLog имеется метод print\_info с тем же именем, что и в классе Goods:

**def** print\_info(self):         **print**("print\_info класса MixinLog")

Понятно, что если сейчас его вызвать через объект класса NoteBook:

n.print\_info()

то мы обратимся к методу класса Goods, так как он записан первым в цепочке наследования и в соответствии с алгоритмом обхода MRO он будет найден первым. Но что если мы хотим вызвать этот метод из второго базового класса MixinLog? Как поступить? Сделать это можно двумя способами. Либо напрямую вызвать этот метод через класс MixinLog:

MixinLog.print\_info(n)

Обратите внимание, что в этом случае нам обязательно нужно указать первым аргументом ссылку на объект класса NoteBook. Либо, определить какой-либо метод в классе NoteBook (пусть он называется также):

**class** NoteBook(Goods, MixinLog):     **def** print\_info(self):         MixinLog.print\_info(self)

И тогда будет вызываться метод именно второго базового класса MixinLog.

Обычно, если нужно делать такие подмены, то есть, из конкретного дочернего класса вызывать метод другого (не первого) базового класса, то создают метод в дочернем классе с тем же именем, а затем, явно указывают нужный базовый класс.

Вот так работает множественное наследование в языке Python.

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