**Магический метод \_\_new\_\_. Пример паттерна Singleton**

[Смотреть материал на видео](https://www.youtube.com/watch?v=-xoT6rntpK0&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E" \t "_blank)

На этом занятии мы познакомимся с еще одним магическим методом \_\_new\_\_, который вызывается непосредственно перед созданием объекта класса. Я напомню, что другой магический метод \_\_init\_\_ вызывается после создания объекта (о нем мы говорили на предыдущем занятии).

Здесь у вас может сразу возникнуть вопрос, зачем нужно было определять два разных метода, которые последовательно вызываются при создании экземпляров классов? Разве не достаточно одного \_\_init\_\_, чтобы выполнять начальную инициализацию объекта? Конечно, нет. В практике программирования встречаются самые разнообразные задачи и иногда нужно что-то делать и до создания объектов. Например, реализация известного паттерна Singleton в Python, как раз делается через метод \_\_new\_\_ и мы с ним позже познакомимся.

А для начала нам нужно познакомиться с работой самого метода \_\_new\_\_. Давайте добавим его в наш класс Point. Я его перепишу в сокращенной форме:

**class** Point:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**print**("вызов \_\_new\_\_ для " + str(cls))

**def** \_\_init\_\_(self, x=0, y=0):

**print**("вызов \_\_init\_\_ для " + str(self))

        self.x = x

        self.y = y

Смотрите, здесь записан метод \_\_new\_\_, у которого первым идет обязательный параметр cls – это ссылка на текущий класс Point, а затем, указываются коллекции из фактических и формальных параметров, которые может принимать данная функция. Это стандартное определение метода \_\_new\_\_ в классах. В теле функции я просто сделал вывод сообщения и переменной cls.

Если теперь попробовать создать экземпляр класса:

pt = Point(1, 2)

то мы в консоли увидим только одно сообщение от метода \_\_new\_\_. То есть, второй метод \_\_init\_\_ не был вызван и, кроме того, если мы распечатаем переменную pt:

**print**(pt)

то увидим значение None, то есть, объект не был создан. Почему так произошло? В Python магический метод \_\_new\_\_ должен возвращать адрес нового созданного объекта. А в нашей программе он ничего не возвращает, то есть, значение None, что эквивалентно отказу в создании нового объекта. Именно поэтому переменная pt принимает значение None.

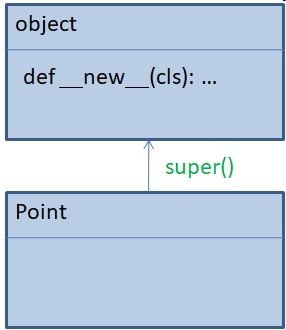
Хорошо, давайте адрес нового объекта. Но откуда мы его возьмем? Для этого можно вызвать аналогичный метод базового класса и делается это, следующим образом:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**print**("вызов \_\_new\_\_ для " + str(cls))

**return** super().\_\_new\_\_(cls)

Здесь функция super() возвращает ссылку на базовый класс и через нее мы вызываем метод \_\_new\_\_ с одним первым аргументом. Но, подождите! Что это за базовый класс? Мы наш класс Point ни от какого класса не наследовали? Да и вообще еще не изучали тему наследования! Да, поэтому, забегая вперед, скажу, что, начиная с версии Python 3, все классы автоматически и неявно наследуются от базового класса object:



И уже из этого базового класса мы вызываем метод \_\_new\_\_. Кстати, если метод \_\_new\_\_ не прописывать в классе Point, то будет автоматически запускаться версия базового класса. То есть, этот метод всегда вызывается при создании нового объекта. При необходимости, мы можем его переопределять, добавляя новую логику его работы. И то же самое относится ко всем магическим методам. Они всегда существуют у всех классов. Но переопределяем мы лишь те, что необходимо, а остальные работают по умолчанию. В этом сила базового класса object. В нем уже существует программный код, общий для всех классов языка Python. Иначе, нам пришлось бы его каждый раз прописывать заново.

Итак, теперь мы знаем откуда берется и вызывается магический метод \_\_new\_\_. Запустим программу и видим в консоли, что были вызваны оба метода \_\_new\_\_ и \_\_init\_\_ нашего класса Point, а также был успешно сформирован новый объект.

Возможно, здесь у вас остался один вопрос: а зачем нужны списки параметров \*args, \*\*kwargs в методе \_\_new\_\_? Мы, вроде, их нигде не используем? В действительности, здесь хранятся дополнительные параметры, которые мы можем указывать при создании объекта. Например, строчка:

pt = Point(1, 2)

создает объект с двумя числовыми значениями, то есть, \*args будет содержать эти два числа. По идее, мы можем реализовать в методе \_\_new\_\_ какую-либо логику с учетом значений этих аргументов. Но, в данном случае, просто игнорируем. Используем их дальше в методе \_\_init\_\_ при инициализации объекта. То есть, аргументы 1 и 2 передаются и в метод \_\_new\_\_ и в метод \_\_init\_\_.

**Пример паттерна Singleton (учебный)**

Думаю, вы в целом теперь представляете себе работу магического метода \_\_new\_\_, но остается вопрос: зачем все же он нужен? В качестве ответа я приведу пример очень известного паттерна проектирования под названием Singleton. Этот паттерн будет представлен в учебном варианте, то есть, мы его реализуем не полностью, т.к. пока отсутствуют достаточные знания.

Итак, давайте предположим, что мы разрабатываем класс для работы с БД. В частности, через него можно будет подключаться к СУБД, читать и записывать информацию, закрывать соединение:

**class** DataBase:

**def** \_\_init\_\_(self, user, psw, port):

        self.user = user

        self.psw = psw

        self.port = port

**def** connect(self):

**print**(f"соединение с БД: {self.user}, {self.psw}, {self.port}")

**def** close(self):

**print**("закрытие соединения с БД")

**def** read(self):

**return** "данные из БД"

**def** write(self, data):

**print**(f"запись в БД {data}")

И далее полагаем, что в программе должен существовать только один экземпляр этого класса в каждый момент ее работы. То есть, одновременно два объекта класса DataBase быть не должно. Чтобы это обеспечить и гарантировать, как раз и используется паттерн Singleton. Реализуем его для класса DataBase.

Я пропишу в нем специальный атрибут (на уровне класса):

\_\_instance = None

который будет хранить ссылку на экземпляр этого класса. Если экземпляра нет, то атрибут будет принимать значение None. А, затем, чтобы гарантировать создание строго одного экземпляра, добавим в класс магический метод \_\_new\_\_:

**def** \_\_new\_\_(cls, \*args, \*\*kwargs):

**if** cls.\_\_instance **is** None:

            cls.\_\_instance = super().\_\_new\_\_(cls)

**return** cls.\_\_instance

Работает этот метод очевидным образом. Мы проверяем атрибут класса \_\_instance. Причем, для обращения к нему используем параметр cls – ссылку на текущий класс. Подробнее я еще освещу этот момент. Далее, проверяем, если значение равно None, то вызываем метод \_\_new\_\_ базового класса и тем самым разрешаем создание объекта. Иначе, просто возвращаем ссылку на ранее созданный экземпляр. Как видите, все достаточно просто.

И пропишем еще один магический метод – финализатор \_\_del\_\_, который будет обнулять атрибут \_\_instance перед уничтожением объекта, чтобы мы могли, при необходимости, создать новый.

Все, простейший вариант паттерна Singleton готов. Правда он имеет один изъян. Смотрите, если попробовать создать два экземпляра:

db = DataBase('root', '1234', 80)

db2 = DataBase('root2', '5678', 40)

**print**(id(db), id(db2))

то их id ожидаемо будут равны. То есть, ссылки db и db2 действительно ведут на один объект. Но, если выполнить метод:

db.connect()

db2.connect()

то увидим значения: 'root2', '5678', 40 – аргументы при повторном создании класса. По идее, если объект не создается, то и локальные свойства его также не должны меняться. Почему так произошло? Все просто. Мы здесь действительно видим первый объект. Но при повторном вызове DataBase() также был вызван магический метод \_\_init\_\_ с новым набором аргументов и локальные свойства изменили свое значение. Конечно, мы можем здесь поставить «костыль» (как говорят в программисты) и дополнительно в классе прописать флаговый атрибут, например:

\_\_is\_exist = False

специально для метода \_\_init\_\_, чтобы не выполнять его если объект уже создан. Но я даже не буду дописывать такую программу. Слишком уж костыльно получается. Правильнее было бы здесь переопределить еще один магический метод \_\_call\_\_, о котором мы еще будем говорить. А пока оставим нашу реализацию паттерна Singleton в таком виде.

Я, надеюсь, что из этого занятия вы поняли, как работает магический метод \_\_new\_\_ и зачем он нужен. Если все это понятно, то переходите к следующему занятию.