# **Магические методы \_\_add\_\_, \_\_sub\_\_, \_\_mul\_\_, \_\_truediv\_\_**

[Смотреть материал на видео](https://www.youtube.com/watch?v=OMMQ1ZNKK6Q&list=PLA0M1Bcd0w8zPwP7t-FgwONhZOHt9rz9E)

Раз уж мы затронули тему магических методов, то доведем ее до конца и познакомимся с наиболее употребительными из них. На этом занятии мы будем рассматривать методы для работы с арифметическими операторами:

* \_\_add\_\_() – для операции сложения;
* \_\_sub\_\_() – для операции вычитания;
* \_\_mul\_\_() – для операции умножения;
* \_\_truediv\_\_() – для операции деления.

Как всегда, объяснить работу этих методов проще всего на конкретном примере. Предположим, что мы создаем класс для работы со временем. Его экземпляры будут хранить часы, минуты и секунды текущего дня. Начальной точкой отсчета будет 00:00 часов ночи. Время будем хранить в виде секунд с максимальным значением 86400 – число секунд в одном дне. Поэтому перед присвоением в инициализаторе будем брать остаток от деления на это значение:

**class** Clock:     \_\_DAY = 86400   *# число секунд в одном дне*       **def** \_\_init\_\_(self, seconds: int):         **if** **not** isinstance(seconds, int):             **raise** TypeError("Секунды должны быть целым числом")         self.seconds = seconds % self.\_\_DAY

Обратите внимание как записан параметр seconds. После него стоит двоеточие и указан ожидаемый тип данных. Эта нотация языка Python подсказывает программисту, какой тип данных следует передавать в качестве seconds. Конечно, мы можем передавать и другие типы данных, строгого ограничения здесь нет, это лишь пометка для программиста и не более того. Поэтому далее внутри инициализатора мы делаем проверку, что параметр seconds должен являться целым числом. Если это не так, генерируем исключение TypeError.

Далее в этом же классе пропишем метод get\_time для получения текущего времени в виде форматной строки:

**def** get\_time(self):         s = self.seconds % 60            *# секунды*         m = (self.seconds // 60) % 60    *# минуты*         h = (self.seconds // 3600) % 24  *# часы*         **return** f"{self.\_\_get\_formatted(h)}:{self.\_\_get\_formatted(m)}:{self.\_\_get\_formatted(s)}"       @classmethod     **def** \_\_get\_formatted(cls, x):         **return** str(x).rjust(2, "0")

Здесь дополнительно определен метод класса для форматирования времени (добавляется незначащий первый ноль, если число меньше 10).

Далее, мы можем воспользоваться этим классом, например, так:

c1 = Clock(1000)**print**(c1.get\_time())

Если же нам понадобиться изменить время в объекте c1, то сейчас это можно сделать через локальное свойство seconds:

c1.seconds = c1.seconds + 100

Мы добавили 100 секунд. Но было бы здорово если бы это изменение можно было бы прописать вот так:

c1 = c1 + 100

Конечно, при запуске программы возникнет ошибка, так как оператор сложения не работает с экземплярами класса Clock. Однако, это можно поправить, если добавить в наш класс магический метод \_\_add\_\_. Я запишу его в следующем виде:

**def** \_\_add\_\_(self, other):         **if** **not** isinstance(other, int):             **raise** ArithmeticError("Правый операнд должен быть типом int")           **return** Clock(self.seconds + other)

И теперь, при запуске программы, все работает так, как и задумывалось. Давайте разберем этот момент подробнее.

Вначале у нас есть объект класса Clock со значением секунд 1000. Затем, арифметическая операция c1 = c1 + 100 фактически означает выполнение команды:

с1 = c1.\_\_add\_\_(100)

В результате, активируется метод \_\_add\_\_ и параметр other принимает целочисленное значение 100. Проверка проходит и формируется новый объект класса Clock со значением секунд 1000+100 = 1100. Этот объект возвращается методом \_\_add\_\_ и переменная c1 начинает ссылаться на этот новый экземпляр класса. На прежний уже не будет никаких внешних ссылок, поэтому он будет автоматически удален сборщиком мусора.

Вас может удивить сложность процессов, когда нам всего лишь нужно прибавить 100 секунд к уже имеющемуся значению. Но эта сложность оправданна. Чтобы это понять, мы расширим функционал оператора сложения и допустим, что можно складывать два разных объекта класса Clock, следующим образом:

c1 = Clock(1000) c2 = Clock(2000) c3 = c1 + c2 **print**(c3.get\_time())

Конечно, если сейчас запустить программу, то увидим ошибку ArithmeticError, так как параметр other не соответствует целому числу. Поправим это и немного изменим реализацию метода \_\_add\_\_:

**def** \_\_add\_\_(self, other):         **if** **not** isinstance(other, (int, Clock)):             **raise** ArithmeticError("Правый операнд должен быть типом int или объектом Clock")           sc = other **if** isinstance(other, int) **else** other.seconds         **return** Clock(self.seconds + sc)

Теперь, в программе можно складывать и отдельные целые числа и объекты классов Clock. Видите, как это удобно! Кроме того, мы можем прописывать и более сложные конструкции при сложении, например, такие:

c1 = Clock(1000) c2 = Clock(2000) c3 = Clock(3000) c4 = c1 + c2 + c3 **print**(c4.get\_time())

И она сработала благодаря тому, что метод \_\_add\_\_ возвращает каждый раз новый экземпляр класса Clock. Детальнее все выглядит так.

Сначала идет сложение объектов c1 + c2, в результате формируется новый объект класса Clock со значением секунд 1000 + 2000 = 3000. Пусть на этот класс ведет внутренняя переменная t1. Затем, для этого нового объекта вызывается снова метод \_\_add\_\_ и идет сложение с объектом t1 + c3. Получаем еще один объект с числом секунд 6000. На этот объект, как раз и будет ссылаться переменная c4, а объект с t1 будет автоматически уничтожен сборщиком мусора.

Если бы мы не создавали экземпляры классов Clock в методе \_\_add\_\_ и не возвращали их, то конструкцию с двумя сложениями было бы невозможно реализовать.

Еще одним важным нюансом работы оператора сложения для объектов классов, является порядок их записи. Мы всегда прописывали его в виде:

c1 = c1 + 100

то есть, сначала шел объект, а затем, число. Если записать наоборот, то возникнет ошибка:

c1 = 100 + c1

и это очевидно, так как здесь, фактически идет вызов метода:

100.\_\_add\_\_(c1)

но он не существует для объекта int и экземпляров класса Clock. Как выйти из этой ситуации? Очень просто. Язык Python предоставляет нам специальный набор магических методов с добавлением буквы r:

\_\_radd\_\_()

Он автоматически вызывается, если не может быть вызван метод \_\_add\_\_(). Давайте добавим его определение в наш класс Clock:

**def** \_\_radd\_\_(self, other):         **return** self + other

Смотрите, мы здесь записали команду сложения текущего объекта класса Clock с параметром other, который может быть или числом или тоже объектом класса Clock. В свою очередь будет вызван метод \_\_add\_\_, но с правильным порядком типов данных, поэтому сложение пройдет без ошибок.

Наконец, у всех магических методов, связанных с арифметическими операторами, есть еще одна модификация с первой буквой i:

\_\_iadd\_\_()

Она вызывается для команды:

c1 += 100

Если запустить сейчас программу, то никаких ошибок не будет и отработает метод \_\_add\_\_(). Но в методе \_\_add\_\_ создается новый объект класса Clock, тогда как при операции += этого делать не обязательно. Поэтому я добавлю еще один магический метод \_\_iadd\_\_ в наш класс Clock:

**def** \_\_iadd\_\_(self, other):         **print**("\_\_iadd\_\_")         **if** **not** isinstance(other, (int, Clock)):             **raise** ArithmeticError("Правый операнд должен быть типом int или объектом Clock")           sc = other **if** isinstance(other, int) **else** other.seconds         self.seconds += sc           **return** self

Смотрите, мы здесь не создаем нового объекта, а меняем число секунд в текущем. Это логичнее, так как вызывать цепочкой операцию += не предполагается и, кроме того, она изменяет (по смыслу) состояние текущего объекта. Поэтому я посчитал правильным добавить этот магический метод.

Вот мы с вами подробно рассмотрели работу одного арифметического магического метода \_\_add\_\_() с его вариациями \_\_radd\_\_() и \_\_iadd\_\_(). По аналогии используются и все остальные подобные магические методы:

|  |  |  |  |
| --- | --- | --- | --- |
| Оператор | Метод оператора | Оператор | Метод оператора |
| x + y | \_\_add\_\_(self, other) | x += y | \_\_iadd\_\_(self, other) |
| x - y | \_\_sub\_\_(self, other) | x -= y | \_\_isub\_\_(self, other) |
| x \* y | \_\_mul\_\_(self, other) | x \*= y | \_\_imul\_\_(self, other) |
| x / y | \_\_truediv\_\_(self, other) | x /= y | \_\_itruediv\_\_(self, other) |
| x // y | \_\_floordiv\_\_(self, other) | x //= y | \_\_ifloordiv\_\_(self, other) |
| x % y | \_\_mod\_\_(self, other) | x %= y | \_\_imod\_\_(self, other) |

Предлагаю вам в качестве самостоятельного задания добавить некоторые из них в класс Clock по аналогии с методами \_\_add\_\_(). При этом обращайте внимание на недопустимость дублирования кода. Оно произойдет, если вы «в лоб» будете прописывать другие арифметические магические методы. Подумайте, как все сделать правильно.

На этом мы завершим это занятие по магическим методам. На следующем продолжим эту тему и поговорим о реализации операторов сравнения.