* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок ООП №3](https://lyceum.yandex.ru/courses/123/groups/568/lessons/669)

**Определение операторов**

**План урока**

1

[Специальные методы](https://lyceum.yandex.ru/courses/123/groups/568/lessons/669/materials/1654#1)

2

[Переопределение функции print()](https://lyceum.yandex.ru/courses/123/groups/568/lessons/669/materials/1654#2)

3

[Метод \_\_repr\_\_](https://lyceum.yandex.ru/courses/123/groups/568/lessons/669/materials/1654#3)

4

[Класс «Вектор на плоскости»](https://lyceum.yandex.ru/courses/123/groups/568/lessons/669/materials/1654#4)

5

[Другие специальные методы](https://lyceum.yandex.ru/courses/123/groups/568/lessons/669/materials/1654#5)

**Аннотация**

*Почти любой оператор Python можно определить и для типов данных, которые мы сами создаём с помощью классов. Это делается с помощью специальных методов. О них и пойдет речь в этом уроке.*

**1. Специальные методы**

На предыдущем занятии мы обсудили полиморфизм на примере оператора **+**. Оператор **+** работает для многих встроенных типов данных: чисел, строк, списков, кортежей. Однако возможность определять операторы есть не только у встроенных типов данных.

**Важно**

Специальные методы имеют для интерпретатора особое значение. Имена специальных методов и их смысл определены создателями языка: создавать новые нельзя, можно только реализовывать существующие. Названия всех специальных методов начинаются и заканчиваются на два подчёркивания.

Пример такого метода — уже знакомый нам \_\_init\_\_. Он предназначен для инициализации экземпляров и автоматически вызывается интерпретатором после создания экземпляра объекта.

Остальные специальные методы также вызываются в строго определённых ситуациях. Большинство из них отвечает за реализацию операторов. Так, например, всякий раз, когда интерпретатор встречает запись вида x + y, он заменяет её на x.\_\_add\_\_(y), и для реализации сложения нам достаточно определить в классе экземпляра **x** метод **\_\_add\_\_**.

**class** Time:

**def** \_\_init\_\_(self, minutes, seconds):

self.minutes = minutes

self.seconds = seconds

**def** \_\_add\_\_(self, other):

m = self.minutes + other.minutes

s = self.seconds + other.seconds

m += s // 60

s = s % 60

**return** Time(m, s)

**def** info(self):

**return** '{}:{}'.format(self.minutes, self.seconds)

t1 = Time(5, 50)

**print**(t1.info()) *# 5:50*

t2 = Time(3, 20)

**print**(t2.info()) *# 3:20*

t3 = t1 + t2

**print**(t3.info()) *# 9:10*

Обратите внимание, что в методе **\_\_add\_\_** мы создаём новый экземпляр с результатом сложения, а не изменяем уже существующий. Для арифметических операторов мы будем поступать так почти всегда, ведь при выполнении z = x + y ни x, ни y изменяться не должны. Должен создаваться новый объект z с результатом операции.

Кстати, именно поэтому в некоторых случаях запись a = a + b отличается от a += b. В первом случае вызывается метод **\_\_add\_\_**, а во втором **\_\_iadd\_\_**, для чисел эти методы работают одинаково, а для списков — нет.

Так как объекты класса **Time** относятся к изменяемым, при вызове **\_\_iadd\_\_** должен изменяться сам объект. Давайте добавим этот метод в наш класс и посмотрим, как он работает.

**class** Time:

**def** \_\_init\_\_(self, minutes, seconds):

self.minutes = minutes

self.seconds = seconds

**def** \_\_add\_\_(self, other):

m = self.minutes + other.minutes

s = self.seconds + other.seconds

m += s // 60

s = s % 60

**return** Time(m, s)

**def** \_\_iadd\_\_(self, other):

m = self.minutes + other.minutes

s = self.seconds + other.seconds

m += s // 60

s = s % 60

self.minutes = m

self.seconds = s

**return** self

**def** info(self):

**return** '{}:{}'.format(self.minutes, self.seconds)

t1 = Time(5, 50)

**print**(t1.info()) *# 5:50*

t2 = Time(3, 20)

**print**(t2.info()) *# 3:20*

**print**(id(t1))

t1 += t2

**print**(t1.info()) *# 9:10*

**print**(id(t1)) *# id объекта не поменяется*

Обратите внимание, что мы не просто изменяем атрибуты объекта в методе **\_\_iadd\_\_**, но и после всех преобразований возвращаем self — сам объект (если мы ничего не вернём, то в переменной окажется None).

**2. Переопределение функции print()**

Другой специальный метод позволяет избавиться от вызовов метода **info** перед передачей данных в print.

**\_\_str\_\_**

Перед выводом аргументов на печать функция print преобразует их в строки с помощью функции **str**. Но функция str делает это не сама, а вызывает метод **\_\_str\_\_** своего аргумента. Так что вызов str(x) эквивалентен x.\_\_str\_\_().

Если мы сейчас попытаемся распечатать экземпляры **Time** просто с помощью print(t1), то получим что-то вроде:

<\_\_main\_\_.Time object at 0x7fa021586f98>

Это сработала реализация метода **\_\_str\_\_** по умолчанию из класса **object**. Дело в том, что при создании класса можно указать так называемый суперкласс, от которого наш класс получит всю функциональность. Такой процесс называется **наследованием**. Об этом механизме мы поговорим на следующем уроке.

**Важно**

Если суперкласс не указать, то по умолчанию наследуется класс **object**, содержащий некоторую базовую функциональность, в том числе метод **\_\_str\_\_**.

Если мы определим в своём классе собственный метод **\_\_str\_\_**, он заменит тот, что был унаследован от **object**.

Давайте это сделаем:

**class** Time:

**def** \_\_init\_\_(self, minutes, seconds):

self.minutes = minutes

self.seconds = seconds

**def** \_\_add\_\_(self, other):

m = self.minutes + other.minutes

s = self.seconds + other.seconds

m += s // 60

s = s % 60

**return** Time(m, s)

**def** \_\_str\_\_(self):

**return** '{}:{}'.format(self.minutes, self.seconds)

t1 = Time(5, 50)

**print**(t1) *# 5:50*

t2 = Time(3, 20)

**print**(t2) *# 3:20*

t3 = t1 + t2

**print**(t3) *# 9:10*

**3. Метод \_\_repr\_\_**

Кроме метода **\_\_str\_\_**, который предназначен для выдачи информации об экземпляре для пользователей в «человеческом» виде, часто определяется метод **\_\_repr\_\_**. Метод **\_\_repr\_\_** внутри себя вызывает функцию **repr**, предназначенную для выдачи полной информации об объекте для программиста. Она часто применяется при отладке. Для нашего класса **Time** этот метод мог бы выглядеть так:

**class** Time:

... методы \_\_init\_\_, \_\_add\_\_, \_\_str\_\_ ...

**def** \_\_repr\_\_(self):

**return** 'Time({}, {})'.format(self.minutes, self.seconds)

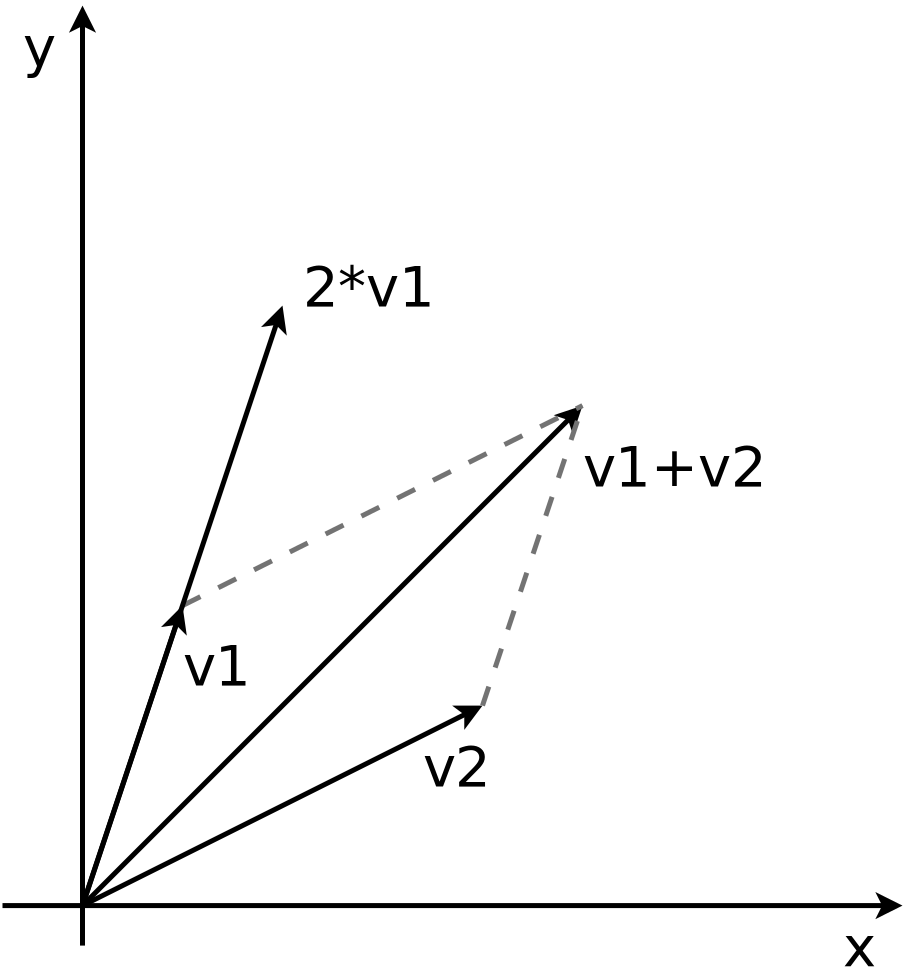
t1 = Time(5, 50)

**print**(t1) *# 5:50*

**print**(repr(t1)) *# Time(5, 50)*

Как видно, здесь метод **\_\_repr\_\_** выдаёт строку, которую можно скопировать и вставить в исходный код на Python, чтобы получить выражение, которое заново сконструирует такой же объект.

**4. Класс «Вектор на плоскости»**



Двумерные векторы — очень полезный и важный геометрический объект. Векторы любой нужной размерности уже есть в библиотеке **Numpy**, которую вы могли изучить в дополнительных материалах курса, но если бы мы захотели реализовать двумерный вектор самостоятельно, то можно было бы сделать это, например, так:

**class** MyVector:

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

self.x = x

self.y = y

**def** \_\_add\_\_(self, other):

**return** MyVector(self.x + other.x, self.y + other.y)

**def** \_\_sub\_\_(self, other):

**return** MyVector(self.x - other.x, self.y - other.y)

**def** \_\_mul\_\_(self, other):

**return** MyVector(self.x \* other, self.y \* other)

**def** \_\_rmul\_\_(self, other):

**return** MyVector(self.x \* other, self.y \* other)

**def** \_\_str\_\_(self):

**return** 'MyVector({}, {})'.format(self.x, self.y)

v1 = MyVector(-2, 5)

v2 = MyVector(3, -4)

v\_sum = v1 + v2

**print**(v\_sum) *# MyVector(1, 1)*

v\_mul = v1 \* 1.5

**print**(v\_mul) *# MyVector(-3.0, 7.5)*

v\_rmul = -2 \* v1

**print**(v\_rmul) *# MyVector(4, -10)*

В этом примере определены методы **\_\_add\_\_** и **\_\_sub\_\_** для реализации классических операций сложения и вычитания векторов. Метод **\_\_mul\_\_** реализует операцию умножения вектора на число, а метод **\_\_rmul\_\_** — операцию умножения числа на вектор. Для преобразования в строку используется уже знакомый нам метод **\_\_str\_\_**.

**5. Другие специальные методы**

Специальных методов слишком много, чтобы рассмотреть их все на этом уроке. Мы приведем лишь небольшой их список.

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| \_\_add\_\_(self, other) | Сложение (x + y). Будет вызвано: x.\_\_add\_\_(y) |
| \_\_sub\_\_(self, other) | Вычитание (x - y) |
| \_\_mul\_\_(self, other) | Умножение (x \* y) |
| \_\_truediv\_\_(self, other) | Деление (x / y) |
| \_\_floordiv\_\_(self, other) | Целочисленное деление (x // y) |
| \_\_mod\_\_(self, other) | Остаток от деления (x % y) |
| \_\_divmod\_\_(self, other) | Частное и остаток (divmod(x, y)) |
| \_\_radd\_\_(self, other) | Сложение (y + x). Будет вызвано: y.\_\_radd\_\_(x) |
| \_\_rsub\_\_(self, other) | Вычитание (y - x) |
| \_\_lt\_\_(self, other) | Сравнение (x < y). Будет вызвано: x.\_\_lt\_\_(y) |
| \_\_eq\_\_(self, other) | Сравнение (x == y). Будет вызвано: x.\_\_eq\_\_(y) |
| \_\_len\_\_(self) | Возвращение длины объекта |
| \_\_getitem\_\_(self, key) | Доступ по индексу (или ключу) |
| \_\_call\_\_(self[, args...]) | Вызов экземпляра класса как функции |

Однако, найти полную документацию по специальным методам в Интернете сравнительно легко. Если вам нужно реализовать тот или иной оператор, то для начала поищите соответствующий ему специальный метод на **втором листе** вот этой [шпаргалки](https://yastatic.net/s3/lyceum/content/resources/abregepython-english.pdf).

Если вы не нашли необходимой информации, то мы рекомендуем очень подробную статью с [длинным и обстоятельным описанием](https://habr.com/post/186608/).

Ну и конечно же, никто не должен забывать про официальную документацию на сайте [python.org](https://docs.python.org/3/reference/datamodel.html#special-method-names).

[Справка](https://yandex.ru/support/lyceum-students)

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты