Определение операторов

Аннотация

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. 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.

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.

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

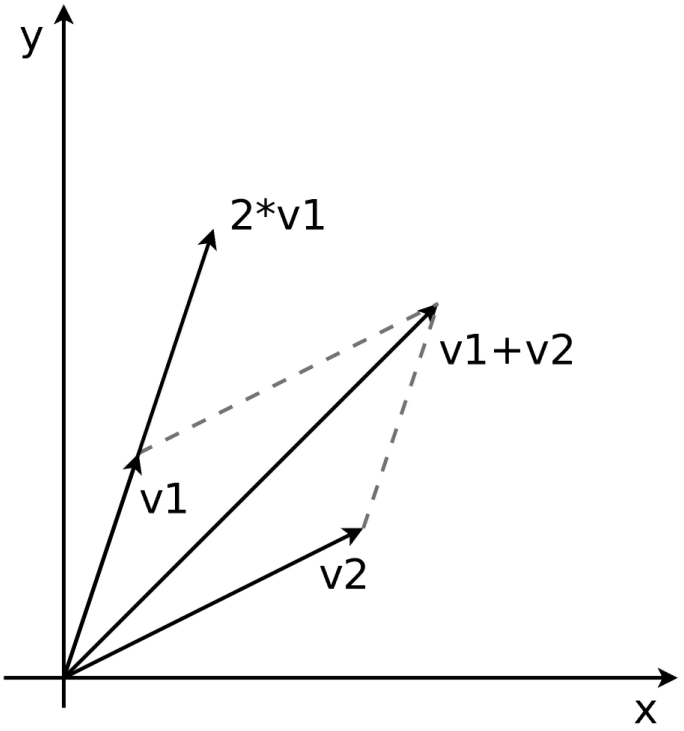
```
class Time:
    def __init__(self, minutes, seconds):
        self.minutes = minutes
        self.seconds = seconds

    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
```

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.

Рассмотрим пример класса "Вектор на плоскости"



Двумерные векторы — очень полезный и важный геометрический объект. Векторы любой нужной размерности уже есть в библиотеке **Numpy**, которую мы уже изучали, 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__**.
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))
__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

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 (http://anytask.s3.yandex.net/materials/32/abregepython-english.pdf).

> *Эта шпаргалка на английском языке, но она очень лаконичная.*

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

Ну и конечно же, никто не должен забывать про официальную документацию на сайте python.org (https://docs.python.org/3/).