Руководство по магическим методам

<http://www.rafekettler.com/magicmethods.html>

Источник: https://habr.com/post/186608/

Это перевод 1.17 версии [руководства](http://www.rafekettler.com/magicmethods.html) от Rafe Kettler.

Содержание

1. Вступление
2. Конструирование и инициализация
3. Переопределение операторов на произвольных классах
   * Магические методы сравнения
   * Числовые магический методы
4. Представление своих классов
5. Контроль доступа к атрибутам
6. Создание произвольных последовательностей
7. Отражение
8. Вызываемые объекты
9. Менеджеры контекста
10. Абстрактные базовые классы
11. Построение дескрипторов
12. Копирование
13. Использование модуля pickle на своих объектах
14. Заключение
15. Приложение 1: Как вызывать магические методы
16. Приложение 2: Изменения в Питоне 3

Вступление

Что такое магические методы? Они всё в объектно-ориентированном Питоне. Это специальные методы, с помощью которых вы можете добавить в ваши классы «магию». Они всегда обрамлены двумя нижними подчеркиваниями (например, \_\_init\_\_ или \_\_lt\_\_). Ещё, они не так хорошо документированны, как хотелось бы. Все магические методы описаны в документации, но весьма беспорядочно и почти безо всякой организации. Поэтому, чтобы исправить то, что я воспринимаю как недостаток документации Питона, я собираюсь предоставить больше информации о магических методах, написанной на понятном языке и обильно снабжённой примерами. Надеюсь, это руководство вам понравится. Используйте его как обучающий материал, памятку или полное описание. Я просто постарался как можно понятнее описать магические методы.

Конструирование и инициализация.

Всем известен самый базовый магический метод, \_\_init\_\_. С его помощью мы можем инициализировать объект. Однако, когда я пишу x = SomeClass(), \_\_init\_\_ не самое первое, что вызывается. На самом деле, экземпляр объекта создаёт метод \_\_new\_\_, а затем аргументы передаются в инициализатор. На другом конце жизненного цикла объекта находится метод \_\_del\_\_. Давайте подробнее рассмотрим эти три магических метода:

* \_\_new\_\_(cls, [...)  
  Это первый метод, который будет вызван при инициализации объекта. Он принимает в качестве параметров класс и потом любые другие аргументы, которые будут переданы в \_\_init\_\_. \_\_new\_\_используется весьма редко, но иногда бывает полезен, в частности, когда класс наследуется от неизменяемого (immutable) типа, такого как кортеж (tuple) или строка. Я не намерен очень детально останавливаться на \_\_new\_\_, так как он не то чтобы очень часто нужен, но этот метод очень хорошо и детально описан в [документации](http://www.python.org/download/releases/2.2/descrintro/#__new__).
* \_\_init\_\_(self, [...)  
  Инициализатор класса. Ему передаётся всё, с чем был вызван первоначальный конструктор (так, например, если мы вызываем x = SomeClass(10, 'foo'), \_\_init\_\_ получит 10 и 'foo' в качестве аргументов. \_\_init\_\_ почти повсеместно используется при определении классов.
* \_\_del\_\_(self)  
  Если \_\_new\_\_ и \_\_init\_\_ образуют конструктор объекта, \_\_del\_\_ это его деструктор. Он не определяет поведение для выражения del x (поэтому этот код не эквивалентен x.\_\_del\_\_()). Скорее, он определяет поведение объекта в то время, когда объект попадает в сборщик мусора. Это может быть довольно удобно для объектов, которые могут требовать дополнительных чисток во время удаления, таких как сокеты или файловыве объекты. Однако, нужно быть осторожным, так как нет гарантии, что \_\_del\_\_ будет вызван, если объект продолжает жить, когда интерпретатор завершает работу. Поэтому \_\_del\_\_ не может служить заменой для хороших программистских практик (всегда завершать соединение, если закончил с ним работать и тому подобное). Фактически, из-за отсутствия гарантии вызова, \_\_del\_\_ не должен использоваться почти никогда; используйте его с осторожностью!  
    
  **Замечание от переводчика:** [svetlov](https://habrahabr.ru/users/svetlov/) [отмечает](https://habr.com/post/186608/#comment_6492862), что здесь автор ошибается, на самом деле \_\_del\_\_ всегда вызывается по завершении работы интерпретатора.

Соединим всё вместе, вот пример \_\_init\_\_ и \_\_del\_\_ в действии:

from os.path import join

class FileObject:

'''Обёртка для файлового объекта, чтобы быть уверенным в том, что файл будет закрыт при удалении.'''

def \_\_init\_\_(self, filepath='~', filename='sample.txt'):

*# открыть файл filename в filepath в режиме чтения и записи*

self.file = open(join(filepath, filename), 'r+')

def \_\_del\_\_(self):

self.file.close()

del self.file

Переопределение операторов на произвольных классах

Одно из больших преимуществ использования магических методов в Питоне то, что они предоставляют простой способ заставить объекты вести себя по подобию встроенных типов. Это означает, что вы можете избежать унылого, нелогичного и нестандартного поведения базовых операторов. В некоторых языках обычное явление писать как-нибудь так:

if instance.equals(other\_instance):

*# do something*

Вы, конечно, можете поступать так же и в Питоне, но это добавляет путаницы и ненужной многословности. Разные библиотеки могут по разному называть одни и те же операции, заставляя использующего их программиста совершать больше действий, чем необходимо. Используя силу магических методов, мы можем определить нужный метод (\_\_eq\_\_, в этом случае), и так точно выразить, что мы *имели в виду*:

if instance == other\_instance:

*#do something*

Это одна из сильных сторон магических методов. Подавляющее большинство из них позволяют определить, что будут делать стандартные операторы, так что мы можем использовать операторы на своих классах так, как будто они встроенные типы.

Магические методы сравнения

В Питоне уйма магических методов, созданных для определения интуитивного сравнения между объектами используя операторы, а не неуклюжие методы. Кроме того, они предоставляют способ переопределить поведение Питона по-умолчанию для сравнения объектов (по ссылке). Вот список этих методов и что они делают:

* \_\_cmp\_\_(self, other)  
  Самый базовый из методов сравнения. Он, в действительности, определяет поведение для всех операторов сравнения (>, ==, !=, итд.), но не всегда так, как вам это нужно (например, если эквивалентность двух экземпляров определяется по одному критерию, а то что один больше другого по какому-нибудь другому). \_\_cmp\_\_ должен вернуть отрицательное число, если self < other, ноль, если self == other, и положительное число в случае self > other. Но, обычно, лучше определить каждое сравнение, которое вам нужно, чем определять их всех в \_\_cmp\_\_. Но \_\_cmp\_\_может быть хорошим способом избежать повторений и увеличить ясность, когда все необходимые сравнения оперерируют одним критерием.
* \_\_eq\_\_(self, other)  
  Определяет поведение оператора равенства, ==.
* \_\_ne\_\_(self, other)  
  Определяет поведение оператора неравенства, !=.
* \_\_lt\_\_(self, other)  
  Определяет поведение оператора меньше, <.
* \_\_gt\_\_(self, other)  
  Определяет поведение оператора больше, >.
* \_\_le\_\_(self, other)  
  Определяет поведение оператора меньше или равно, <=.
* \_\_ge\_\_(self, other)  
  Определяет поведение оператора больше или равно, >=.

Для примера расммотрим класс, описывающий слово. Мы можем сравнивать слова лексиграфически (по алфавиту), что является дефолтным поведением при сравнении строк, но можем захотеть использовать при сравнении какой-нибудь другой критерий, такой, как длина или количество слогов. В этом примере мы будем сравнивать по длине. Вот реализация:

class Word(str):

'''Класс для слов, определяющий сравнение по длине слов.'''

def \_\_new\_\_(cls, word):

*# Мы должны использовать \_\_new\_\_, так как тип str неизменяемый*

*# и мы должны инициализировать его раньше (при создании)*

if ' ' in word:

print "Value contains spaces. Truncating to first space."

word = word[:word.index(' ')] *# Теперь Word это все символы до первого пробела*

return str.\_\_new\_\_(cls, word)

def \_\_gt\_\_(self, other):

return len(self) > len(other)

def \_\_lt\_\_(self, other):

return len(self) < len(other)

def \_\_ge\_\_(self, other):

return len(self) >= len(other)

def \_\_le\_\_(self, other):

return len(self) <= len(other)

Теперь мы можем создать два Word (при помощи Word('foo') и Word('bar')) и сравнить их по длине. Заметьте, что мы не определяли \_\_eq\_\_ и \_\_ne\_\_, так как это приведёт к странному поведению (например, Word('foo') == Word('bar') будет расцениваться как истина). В этом нет смысла при тестировании на эквивалентность, основанную на длине, поэтому мы оставляем стандартную проверку на эквивалентность от str.  
Сейчас, кажется, удачное время упомянуть, что вы не должны определять каждый из магических методов сравнения, чтобы полностью охватить все сравнения. Стандартная библиотека любезно предоставляет нам класс-декторатор в модуле functools, который и определит все сравнивающие методы, от вас достаточно определить только \_\_eq\_\_ и ещё один (\_\_gt\_\_, \_\_lt\_\_ и т.п.) Эта возможность доступна начиная с 2.7 версии Питона, но если это вас устраивает, вы сэкономите кучу времени и усилий. Для того, чтобы задействовать её, поместите @total\_ordering над вашим определением класса.

Числовые магические методы

Точно так же, как вы можете определить, каким образом ваши объекты будут сравниваться операторами сравнения, вы можете определить их поведение для числовых операторов. Приготовтесь, друзья, их много. Для лучшей организации, я разбил числовые магические методы на 5 категорий: унарные операторы, обычные арифметические операторы, отражённые арифметические операторы (подробности позже), составные присваивания и преобразования типов.

Унарные операторы и функции

Унарные операторы и функции имеют только один операнд — отрицание, абсолютное значение, и так далее.

* \_\_pos\_\_(self)  
  Определяет поведение для унарного плюса (+some\_object)
* \_\_neg\_\_(self)  
  Определяет поведение для отрицания(-some\_object)
* \_\_abs\_\_(self)  
  Определяет поведение для встроенной функции abs().
* \_\_invert\_\_(self)  
  Определяет поведение для инвертирования оператором ~. Для объяснения что он делает смотри [статью в Википедии о бинарных операторах](http://en.wikipedia.org/wiki/Bitwise_operation#NOT).
* \_\_round\_\_(self, n)  
  Определяет поведение для встроенной функции round(). n это число знаков после запятой, до которого округлить.
* \_\_floor\_\_(self)  
  Определяет поведение для math.floor(), то есть, округления до ближайшего меньшего целого.
* \_\_ceil\_\_(self)  
  Определяет поведение для math.ceil(), то есть, округления до ближайшего большего целого.
* \_\_trunc\_\_(self)  
  Определяет поведение для math.trunc(), то есть, обрезания до целого.

Обычные арифметические операторы

Теперь рассмотрим обычные бинарные операторы (и ещё пару функций): +, -, \* и похожие. Они, по большей части, отлично сами себя описывают.

* \_\_add\_\_(self, other)  
  Сложение.
* \_\_sub\_\_(self, other)  
  Вычитание.
* \_\_mul\_\_(self, other)  
  Умножение.
* \_\_floordiv\_\_(self, other)  
  Целочисленное деление, оператор //.
* \_\_div\_\_(self, other)  
  Деление, оператор /.
* \_\_truediv\_\_(self, other)  
  *Правильное* деление. Заметьте, что это работает только когда используется from \_\_future\_\_ import division.
* \_\_mod\_\_(self, other)  
  Остаток от деления, оператор %.
* \_\_divmod\_\_(self, other)  
  Определяет поведение для встроенной функции divmod().
* \_\_pow\_\_  
  Возведение в степень, оператор \*\*.
* \_\_lshift\_\_(self, other)  
  Двоичный сдвиг влево, оператор <<.
* \_\_rshift\_\_(self, other)  
  Двоичный сдвиг вправо, оператор >>.
* \_\_and\_\_(self, other)  
  Двоичное И, оператор &.
* \_\_or\_\_(self, other)  
  Двоичное ИЛИ, оператор |.
* \_\_xor\_\_(self, other)  
  Двоичный xor, оператор ^.

Отражённые арифметические операторы

Помните как я сказал, что собираюсь остановиться на отражённой арифметике подробнее? Вы могли подумать, что это какая-то большая, страшная и непонятная концепция. На самом деле всё очень просто. Вот пример:

some\_object + other

Это «обычное» сложение. Единственное, чем отличается эквивалентное отражённое выражение, это порядок слагаемых:

other + some\_object

Таким образом, все эти магические методы делают то же самое, что и их обычные версии, за исключением выполнения операции с other в качестве первого операнда и self в качестве второго. В большинстве случаев, результат отражённой операции такой же, как её обычный эквивалент, поэтому при определении \_\_radd\_\_ вы можете ограничиться вызовом \_\_add\_\_ да и всё. Заметьте, что объект слева от оператора (other в примере) не должен иметь обычной неотражённой версии этого метода. В нашем примере, some\_object.\_\_radd\_\_ будет вызван только если в other не определён \_\_add\_\_.

* \_\_radd\_\_(self, other)  
  Отражённое сложение.
* \_\_rsub\_\_(self, other)  
  Отражённое вычитание.
* \_\_rmul\_\_(self, other)  
  Отражённое умножение.
* \_\_rfloordiv\_\_(self, other)  
  Отражённое целочисленное деление, оператор //.
* \_\_rdiv\_\_(self, other)  
  Отражённое деление, оператор /.
* \_\_rtruediv\_\_(self, other)  
  Отражённое *правильное* деление. Заметьте, что работает только когда используется from \_\_future\_\_ import division.
* \_\_rmod\_\_(self, other)  
  Отражённый остаток от деления, оператор %.
* \_\_rdivmod\_\_(self, other)  
  Определяет поведение для встроенной функции divmod(), когда вызывается divmod(other, self).
* \_\_rpow\_\_  
  Отражённое возведение в степерь, оператор \*\*.
* \_\_rlshift\_\_(self, other)  
  Отражённый двоичный сдвиг влево, оператор <<.
* \_\_rrshift\_\_(self, other)  
  Отражённый двоичный сдвиг вправо, оператор >>.
* \_\_rand\_\_(self, other)  
  Отражённое двоичное И, оператор &.
* \_\_ror\_\_(self, other)  
  Отражённое двоичное ИЛИ, оператор |.
* \_\_rxor\_\_(self, other)  
  Отражённый двоичный xor, оператор ^.

Составное присваивание

В Питоне широко представлены и магические методы для составного присваивания. Вы скорее всего уже знакомы с составным присваиванием, это комбинация «обычного» оператора и присваивания. Если всё ещё непонятно, вот пример:

x = 5

x += 1 *# другими словами x = x + 1*

Каждый из этих методов должен возвращать значение, которое будет присвоено переменной слева (например, для a += b, \_\_iadd\_\_ должен вернуть a + b, что будет присвоено a). Вот список:

* \_\_iadd\_\_(self, other)  
  Сложение с присваиванием.
* \_\_isub\_\_(self, other)  
  Вычитание с присваиванием.
* \_\_imul\_\_(self, other)  
  Умножение с присваиванием.
* \_\_ifloordiv\_\_(self, other)  
  Целочисленное деление с присваиванием, оператор //=.
* \_\_idiv\_\_(self, other)  
  Деление с присваиванием, оператор /=.
* \_\_itruediv\_\_(self, other)  
  *Правильное* деление с присваиванием. Заметьте, что работает только если используется from \_\_future\_\_ import division.
* \_\_imod\_(self, other)  
  Остаток от деления с присваиванием, оператор %=.
* \_\_ipow\_\_  
  Возведение в степерь с присваиванием, оператор \*\*=.
* \_\_ilshift\_\_(self, other)  
  Двоичный сдвиг влево с присваиванием, оператор <<=.
* \_\_irshift\_\_(self, other)  
  Двоичный сдвиг вправо с присваиванием, оператор >>=.
* \_\_iand\_\_(self, other)  
  Двоичное И с присваиванием, оператор &=.
* \_\_ior\_\_(self, other)  
  Двоичное ИЛИ с присваиванием, оператор |=.
* \_\_ixor\_\_(self, other)  
  Двоичный xor с присваиванием, оператор ^=.

Магические методы преобразования типов

Кроме того, в Питоне множество магических методов, предназначенных для определния поведения для встроенных функций преобразования типов, таких как float(). Вот они все:

* \_\_int\_\_(self)  
  Преобразование типа в int.
* \_\_long\_\_(self)  
  Преобразование типа в long.
* \_\_float\_\_(self)  
  Преобразование типа в float.
* \_\_complex\_\_(self)  
  Преобразование типа в комплексное число.
* \_\_oct\_\_(self)  
  Преобразование типа в восьмеричное число.
* \_\_hex\_\_(self)  
  Преобразование типа в шестнадцатиричное число.
* \_\_index\_\_(self)  
  Преобразование типа к int, когда объект используется в срезах (выражения вида [start:stop:step]). Если вы определяете свой числовый тип, который может использоваться как индекс списка, вы должны определить \_\_index\_\_.
* \_\_trunc\_\_(self)  
  Вызывается при math.trunc(self). Должен вернуть своё значение, обрезанное до целочисленного типа (обычно long).
* \_\_coerce\_\_(self, other)  
  Метод для реализации арифметики с операндами разных типов. \_\_coerce\_\_ должен вернуть Noneесли преобразование типов невозможно. Если преобразование возможно, он должен вернуть пару (кортеж из 2-х элементов) из self и other, преобразованные к одному типу.

Представление своих классов

Часто бывает полезно представление класса в виде строки. В Питоне существует несколько методов, которые вы можете определить для настройки поведения встроенных функций при представлении вашего класса.

* \_\_str\_\_(self)  
  Определяет поведение функции str(), вызванной для экземпляра вашего класса.
* \_\_repr\_\_(self)  
  Определяет поведение функции repr(), вызыванной для экземпляра вашего класса. Главное отличие от str() в целевой аудитории. repr() больше предназначен для машинно-ориентированного вывода (более того, это часто должен быть валидный код на Питоне), а str()предназначен для чтения людьми.
* \_\_unicode\_\_(self)  
  Определяет поведение функции unicode(), вызыванной для экземпляра вашего класса. unicode()похож на str(), но возвращает строку в юникоде. Будте осторожны: если клиент вызывает str() на экземпляре вашего класса, а вы определили только \_\_unicode\_\_(), то это не будет работать. Постарайтесь всегда определять \_\_str\_\_() для случая, когда кто-то не имеет такой роскоши как юникод.
* \_\_format\_\_(self, formatstr)  
  Определяет поведение, когда экземпляр вашего класса используется в форматировании строк нового стиля. Например, "Hello, {0:abc}!".format(a) приведёт к вызову a.\_\_format\_\_("abc"). Это может быть полезно для определения ваших собственных числовых или строковых типов, которым вы можете захотеть предоставить какие-нибудь специальные опции форматирования.
* \_\_hash\_\_(self)  
  Определяет поведение функции hash(), вызыванной для экземпляра вашего класса. Метод должен возвращать целочисленное значение, которое будет использоваться для быстрого сравнения ключей в словарях. Заметьте, что в таком случае обычно нужно определять и \_\_eq\_\_ тоже. Руководствуйтесь следующим правилом: a == b подразумевает hash(a) == hash(b).
* \_\_nonzero\_\_(self)  
  Определяет поведение функции bool(), вызванной для экземпляра вашего класса. Должна вернуть True или False, в зависимости от того, когда вы считаете экземпляр соответствующим True или False.
* \_\_dir\_\_(self)  
  Определяет поведение функции dir(), вызванной на экземпляре вашего класса. Этот метод должен возвращать пользователю список атрибутов. Обычно, определение \_\_dir\_\_ не требуется, но может быть жизненно важно для интерактивного использования вашего класса, если вы переопределили \_\_getattr\_\_ или \_\_getattribute\_\_ (с которыми вы встретитесь в следующей части), или каким-либо другим образом динамически создаёте атрибуты.
* \_\_sizeof\_\_(self)  
  Определяет поведение функции sys.getsizeof(), вызыванной на экземпляре вашего класса. Метод должен вернуть размер вашего объекта в байтах. Он главным образом полезен для классов, определённых в расширениях на C, но всё-равно полезно о нём знать.

Мы почти закончили со скучной (и лишённой примеров) частью руководства по магическим методам. Теперь, когда мы рассмотрели самые базовые магические методы, пришло время перейти к более продвинутому материалу.

Контроль доступа к атрибутам

Многие люди, пришедшие в Питон из других языков, жалуются на отсутствие настоящей инкапсуляции для классов (например, нет способа определить приватные атрибуты с публичными методами доступа). Это не совсем правда: просто многие вещи, связанные с инкапсуляцией, Питон реализует через «магию», а не явными модификаторами для методов и полей. Смотрите:

* \_\_getattr\_\_(self, name)  
  Вы можете определить поведение для случая, когда пользователь пытается обратиться к атрибуту, который не существует (совсем или пока ещё). Это может быть полезным для перехвата и перенаправления частых опечаток, предупреждения об использовании устаревших атрибутов (вы можете всё-равно вычислить и вернуть этот атрибут, если хотите), или хитро возвращать AttributeError, когда это вам нужно. Правда, этот метод вызывается только когда пытаются получить доступ к несуществующему атрибуту, поэтому это не очень хорошее решение для инкапсуляции.
* \_\_setattr\_\_(self, name, value)  
  В отличии от \_\_getattr\_\_, \_\_setattr\_\_ решение для инкапсуляции. Этот метод позволяет вам определить поведение для присвоения значения атрибуту, независимо от того существует атрибут или нет. То есть, вы можете определить любые правила для любых изменений значения атрибутов. Впрочем, вы должны быть осторожны с тем, как использовать \_\_setattr\_\_, смотрите пример нехорошего случая в конце этого списка.
* \_\_delattr\_\_  
  Это то же, что и \_\_setattr\_\_, но для удаления атрибутов, вместо установки значений. Здесь требуются те же меры предосторожности, что и в \_\_setattr\_\_ чтобы избежать бесконечной рекурсии (вызов del self.name в определении \_\_delattr\_\_ вызовет бесконечную рекурсию).
* \_\_getattribute\_\_(self, name)  
  \_\_getattribute\_\_ выглядит к месту среди своих коллег \_\_setattr\_\_ и \_\_delattr\_\_, но я бы не рекомендовал вам его использовать. \_\_getattribute\_\_ может использоваться только с классами нового типа (в новых версиях Питона все классы нового типа, а в старых версиях вы можете получить такой класс унаследовавшись от object). Этот метод позволяет вам определить поведение для *каждого* случая доступа к атрибутам (а не только к несуществующим, как \_\_getattr\_\_(self, name)). Он страдает от таких же проблем с бесконечной рекурсией, как и его коллеги (на этот раз вы можете вызывать \_\_getattribute\_\_ у базового класса, чтобы их предотвратить). Он, так же, главным образом устраняет необходимость в \_\_getattr\_\_, который в случае реализации \_\_getattribute\_\_ может быть вызван только явным образом или в случае генерации исключения AttributeError. Вы конечно можете использовать этот метод (в конце концов, это ваш выбор), но я бы не рекомендовал, потому что случаев, когда он действительно полезен очень мало (намного реже нужно переопределять поведение при получении, а не при установке значения) и реализовать его без возможных ошибок очень сложно.

Вы можете запросто получить проблему при определении любого метогда, управляющего доступом к атрибутам. Рассмотрим пример:

def \_\_setattr\_\_(self, name, value):

self.name = value

*# это рекурсия, так как всякий раз, когда любому атрибуту присваивается значение,*

*# вызывается \_\_setattr\_\_().*

*# тоесть, на самом деле это равнозначно self.\_\_setattr\_\_('name', value).*

*# Так как метод вызывает сам себя, рекурсия продолжится бесконечно, пока всё не упадёт*

def \_\_setattr\_\_(self, name, value):

self.\_\_dict\_\_[name] = value *# присваивание в словарь переменных класса*

*# дальше определение произвольного поведения*

Ещё раз, мощь магических методов в Питоне невероятна, а с большой силой приходит и большая ответственность. Важно знать, как правильно использовать магические методы, ничего не ломая.  
  
Итак, что мы узнали об управлении доступом к атрибутам? Их не нужно использовать легкомысленно. На самом деле, они имеют склонность к чрезмерной мощи и нелогичности. Причина, по которой они всё-таки существуют, в удволетворении определённого желания: Питон склонен не запрещать плохие штуки полностью, а только усложнять их использование. Свобода первостепенна, поэтому вы на самом деле можете делать всё, что хотите. Вот пример использования методов контроля доступа (заметьте, что мы используем super, так как не все классы имеют атрибут \_\_dict\_\_):

class AccessCounter(object):

'''Класс, содержащий атрибут value и реализующий счётчик доступа к нему.

Счётчик увеличивается каждый раз, когда меняется value.'''

def \_\_init\_\_(self, val):

super(AccessCounter, self).\_\_setattr\_\_('counter', 0)

super(AccessCounter, self).\_\_setattr\_\_('value', val)

def \_\_setattr\_\_(self, name, value):

if name == 'value':

super(AccessCounter, self).\_\_setattr\_\_('counter', self.counter + 1)

*# Не будем делать здесь никаких условий.*

*# Если вы хотите предотвратить изменение других атрибутов,*

*# выбросьте исключение AttributeError(name)*

super(AccessCounter, self).\_\_setattr\_\_(name, value)

def \_\_delattr\_\_(self, name):

if name == 'value':

super(AccessCounter, self).\_\_setattr\_\_('counter', self.counter + 1)

super(AccessCounter, self).\_\_delattr\_\_(name)]

Создание произвольных последовательностей

В Питоне существует множество способов заставить ваши классы вести себя как встроенные последовательности (словари, кортежи, списки, строки и так далее). Это, безусловно, мои любимые магические методы, из-за до абсурда высокой степени контроля, которую они дают и той магии, от которой с экземплярами ваших классов вдруг начинает прекрасно работать целое множество глобальных функций. Но, до того как мы перейдём ко всяким хорошим вещам, мы должны знать о протоколах.

Протоколы

Теперь, когда речь зашла о создании собственных последовательностей в Питоне, пришло время поговорить о *протоколах*. Протоколы немного похожи на интерфейсы в других языках тем, что они предоставляют набор методов, которые вы должны реализовать. Однако, в Питоне протоколы абсолютно ни к чему не обязывают и не требуют обязательно реализовать какое-либо объявление. Наверное, они больше похожи на руководящие указания.  
  
Почему мы заговорили о протоколах? Потому, что реализация произвольных контейнерных типов в Питоне влечёт за собой использование некоторых из них. Во-первых, протокол для определения неизменяемых контейнеров: чтобы создать неизменяемый контейнер, вы должны только определить \_\_len\_\_ и \_\_getitem\_\_ (продробнее о них дальше). Протокол изменяемого контейнера требует того же, что и неизменяемого контейнера, плюс \_\_setitem\_\_ и \_\_delitem\_\_. И, наконец, если вы хотите, чтобы ваши объекты можно было перебирать итерацией, вы должны определить \_\_iter\_\_, который возвращает итератор. Этот итератор должен соответствовать протоколу итератора, который требует методов \_\_iter\_\_(возвращает самого себя) и next.

Магия контейнеров

Без дальнейшего промедления, вот магические методы, используемые контейнерами:

* \_\_len\_\_(self)  
  Возвращает количество элементов в контейнере. Часть протоколов для изменяемого и неизменяемого контейнеров.
* \_\_getitem\_\_(self, key)  
  Определяет поведение при доступе к элементу, используя синтаксис self[key]. Тоже относится и к протоколу изменяемых и к протоколу неизменяемых контейнеров. Должен выбрасывать соответствующие исключения: TypeError если неправильный тип ключа и KeyError если ключу не соответствует никакого значения.
* \_\_setitem\_\_(self, key, value)  
  Определяет поведение при присваивании значения элементу, используя синтаксис self[nkey] = value. Часть протокола изменяемого контейнера. Опять же, вы должны выбрасывать KeyError и TypeError в соответсвующих случаях.
* \_\_delitem\_\_(self, key)  
  Определяет поведение при удалении элемента (то есть del self[key]). Это часть только протокола для изменяемого контейнера. Вы должны выбрасывать соответствующее исключение, если ключ некорректен.
* \_\_iter\_\_(self)  
  Должен вернуть итератор для контейнера. Итераторы возвращаются в множестве ситуаций, главным образом для встроенной функции iter() и в случае перебора элементов контейнера выражением for x in container:. Итераторы сами по себе объекты и они тоже должны определять метод \_\_iter\_\_, который возвращает self.
* \_\_reversed\_\_(self)  
  Вызывается чтобы определить поведения для встроенной функции reversed(). Должен вернуть обратную версию последовательности. Реализуйте метод только если класс упорядоченный, как список или кортеж.
* \_\_contains\_\_(self, item)  
  \_\_contains\_\_ предназначен для проверки принадлежности элемента с помощью in и not in. Вы спросите, почему же это не часть протокола последовательности? Потому что когда \_\_contains\_\_не определён, Питон просто перебирает всю последовательность элемент за элементом и возвращает True если находит нужный.
* \_\_missing\_\_(self, key)  
  \_\_missing\_\_ используется при наследовании от dict. Определяет поведение для для каждого случая, когда пытаются получить элемент по несуществующему ключу (так, например, если у меня есть словарь d и я пишу d["george"] когда "george" не является ключом в словаре, вызывается d.\_\_missing\_\_("george")).

Пример

Для примера, давайте посмотрим на список, который реализует некоторые функциональные конструкции, которые вы могли встретить в других языках (Хаскеле, например).

class FunctionalList:

'''Класс-обёртка над списком с добавлением некоторой функциональной магии: head,

tail, init, last, drop, take.'''

def \_\_init\_\_(self, values=None):

if values is None:

self.values = []

else:

self.values = values

def \_\_len\_\_(self):

return len(self.values)

def \_\_getitem\_\_(self, key):

*# если значение или тип ключа некорректны, list выбросит исключение*

return self.values[key]

def \_\_setitem\_\_(self, key, value):

self.values[key] = value

def \_\_delitem\_\_(self, key):

del self.values[key]

def \_\_iter\_\_(self):

return iter(self.values)

def \_\_reversed\_\_(self):

return FunctionalList(reversed(self.values))

def append(self, value):

self.values.append(value)

def head(self):

*# получить первый элемент*

return self.values[0]

def tail(self):

*# получить все элементы после первого*

return self.values[1:]

def init(self):

*# получить все элементы кроме последнего*

return self.values[:-1]

def last(self):

*# получить последний элемент*

return self.values[-1]

def drop(self, n):

*# все элементы кроме первых n*

return self.values[n:]

def take(self, n):

*# первые n элементов*

return self.values[:n]

Теперь у вас есть полезный (относительно) пример реализации своей собственной последовательности. Существуют, конечно, и куда более практичные реализации произвольных последовательностей, но большое их число уже реализовано в стандартной библиотеке (с батарейками в комплекте, да?), такие как Counter, OrderedDict, NamedTuple.

Отражение

Вы можете контролировать и отражение, использующее встроенные функции isinstance() и issubclass(), определив некоторые магические методы. Вот они:

* \_\_instancecheck\_\_(self, instance)  
  Проверяет, является ли экземлпяр членом вашего класса (isinstance(instance, class), например.
* \_\_subclasscheck\_\_(self, subclass)  
  Проверяет, является наследуется ли класс от вашего класса (issubclass(subclass, class)).

Может показаться, что вариантов полезного использования этих магических методов немного и, возможно, это на самом деле так. Я не хочу тратить слишком много времени на магические методы отражения, не особо они и важные, но они отражают кое-что важное об объектно-ориентированном программировании в Питоне и о Питоне вообще: почти всегда существует простой способ что-либо сделать, даже если надобность в этом «что-либо» возникает очень редко. Эти магические методы могут не выглядеть полезными, но если они вам когда-нибудь понадобятся, вы будете рады вспомнить, что они есть (и для этого вы читаете настоящее руководство!).

Вызываемые объекты

Как вы наверное уже знаете, в Питоне функции являются объектами первого класса. Это означает, что они могут быть переданы в функции или методы так же, как любые другие объекты. Это невероятно мощная особенность.  
  
Специальный магический метод позволяет экземплярам вашего класса вести себя так, как будто они функции, тоесть вы сможете «вызывать» их, передавать их в функции, которые принимают функции в качестве аргументов и так далее. Это другая удобная особенность, которая делает программирование на Питоне таким приятным.

* \_\_call\_\_(self, [args...])  
  Позволяет любому экземпляру вашего класса быть вызванным как-будто он функция. Главным образом это означает, что x() означает то же, что и x.\_\_call\_\_(). Заметьте, \_\_call\_\_ принимает произвольное число аргументов; то есть, вы можете определить \_\_call\_\_ так же как любую другую функцию, принимающую столько аргументов, сколько вам нужно.

\_\_call\_\_, в частности, может быть полезен в классах, чьи экземпляры часто изменяют своё состояние. «Вызвать» экземпляр может быть интуитивно понятным и элегантным способом изменить состояние объекта. Примером может быть класс, представляющий положение некоторого объекта на плоскости:

class Entity:

'''Класс, описывающий объект на плоскости. "Вызываемый", чтобы обновить позицию объекта.'''

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

self.x, self.y = x, y

self.size = size

def \_\_call\_\_(self, x, y):

'''Изменить положение объекта.'''

self.x, self.y = x, y

*# чик...*

Менеджеры контекста

В Питоне 2.5 было представлено новое ключевое слово вместе с новым способом повторно использовать код, ключевое слово with. Концепция менеджеров контекста не являлась новой для Питона (она была реализована раньше как часть библиотеки), но в [PEP 343](http://www.python.org/dev/peps/pep-0343/) достигла статуса языковой конструкции. Вы могли уже видеть выражения с with:

with open('foo.txt') as bar:

*# выполнение каких-нибудь действий с bar*

Менеджеры контекста позволяют выполнить какие-то действия для настройки или очистки, когда создание объекта обёрнуто в оператор with. Поведение менеджера контекста определяется двумя магическими методами:

* \_\_enter\_\_(self)  
  Определяет, что должен сделать менеджер контекста в начале блока, созданного оператором with. Заметьте, что возвращаемое \_\_enter\_\_ значение и есть то значение, с которым производится работа внутри with.
* \_\_exit\_\_(self, exception\_type, exception\_value, traceback)  
  Определяет действия менеджера контекста после того, как блок будет выполнен (или прерван во время работы). Может использоваться для контроллирования исключений, чистки, любых действий которые должны быть выполнены незамедлительно после блока внутри with. Если блок выполнен успешно, exception\_type, exception\_value, и traceback будут установлены в None. В другом случае вы сами выбираете, перехватывать ли исключение или предоставить это пользователю; если вы решили перехватить исключение, убедитесь, что \_\_exit\_\_ возвращает True после того как всё сказано и сделано. Если вы не хотите, чтобы исключение было перехвачено менеджером контекста, просто позвольте ему случиться.

\_\_enter\_\_ и \_\_exit\_\_ могут быть полезны для специфичных классов с хорошо описанным и распространённым поведением для их настройки и очистки ресурсов. Вы можете использовать эти методы и для создания общих менеджеров контекста для разных объектов. Вот пример:

class Closer:

'''Менеджер контекста для автоматического закрытия объекта вызовом метода close

в with-выражении.'''

def \_\_init\_\_(self, obj):

self.obj = obj

def \_\_enter\_\_(self):

return self.obj *# привязка к активному объекту with-блока*

def \_\_exit\_\_(self, exception\_type, exception\_val, trace):

try:

self.obj.close()

except AttributeError: *# у объекта нет метода close*

print 'Not closable.'

return True *# исключение перехвачено*

Пример использования Closer с FTP-соединением (сокет, имеющий метод close):

>>> from magicmethods import Closer

>>> from ftplib import FTP

>>> with Closer(FTP('ftp.somesite.com')) as conn:

... conn.dir()

...

*# output omitted for brevity*

>>> conn.dir()

*# long AttributeError message, can't use a connection that's closed*

>>> with Closer(int(5)) as i:

... i += 1

...

Not closable.

>>> i

6

Видите, как наша обёртка изящно управляется и с правильными и с неподходящими объектами. В этом сила менеджеров контекста и магических методов. Заметьте, что стандартная библиотека Питона включает модуль [contextlib](http://docs.python.org/library/contextlib.html), который включает в себя contextlib.closing() — менеджер контекста, который делает приблизительно то же (без какой-либо обработки случая, когда объект не имеет метода close()).

Абстрактные базовые классы

Смотри <http://docs.python.org/2/library/abc.html>.

Построение дескрипторов

Дескрипторы это такие классы, с помощью которых можно добавить свою логику к событиям доступа (получение, изменение, удаление) к атрибутам других объектов. Дескрипторы не подразумевается использовать сами по себе; скорее, предполагается, что ими будут владеть какие-нибудь связанные с ними классы. Дескрипторы могут быть полезны для построения объектно-ориентированных баз данных или классов, чьи атрибуты зависят друг от друга. В частности, дескрипторы полезны при представлении атрибутов в нескольких системах исчисления или каких-либо вычисляемых атрибутов (как расстояние от начальной точки до представленной атрибутом точки на сетке).  
  
Чтобы класс стал дескриптором, он должен реализовать по крайней мере один метод из \_\_get\_\_, \_\_set\_\_ или \_\_delete\_\_. Давайте рассмотрим эти магические методы:

* \_\_get\_\_(self, instance, instance\_class)  
  Определяет поведение при возвращении значения из дескриптора. instance это объект, для чьего атрибута-дескриптора вызывается метод. owner это тип (класс) объекта.
* \_\_set\_\_(self, instance, value)  
  Определяет поведение при изменении значения из дескриптора. instance это объект, для чьего атрибута-дескриптора вызывается метод. value это значение для установки в дескриптор.
* \_\_delete\_\_(self, instance)  
  Определяет поведение для удаления значения из дескриптора. instance это объект, владеющий дескриптором.

Теперь пример полезного использования дескрипторов: преобразование единиц измерения.

class Meter(object):

'''Дескриптор для метра.'''

def \_\_init\_\_(self, value=0.0):

self.value = float(value)

def \_\_get\_\_(self, instance, owner):

return self.value

def \_\_set\_\_(self, instance, value):

self.value = float(value)

class Foot(object):

'''Дескриптор для фута.'''

def \_\_get\_\_(self, instance, owner):

return instance.meter \* 3.2808

def \_\_set\_\_(self, instance, value):

instance.meter = float(value) / 3.2808

class Distance(object):

'''Класс, описывающий расстояние, содержит два дескриптора для футов и

метров.'''

meter = Meter()

foot = Foot()

Копирование

В Питоне оператор присваивания не копирует объекты, а только добавляет ещё одну ссылку. Но для коллекций, содержащих изменяемые элементы, иногда необходимо полноценное копирование, чтобы можно было менять элементы одной последовательности, не затрагивая другую. Здесь в игру и вступает [copy](http://docs.python.org/library/copy.html). К счастью, модули в Питоне не обладают разумом, поэтому мы можем не беспокоиться что они вдруг начнут бесконтрольно копировать сами себя и вскоре линуксовые роботы заполонят всю планету, но мы должны сказать Питону как правильно копировать.

* \_\_copy\_\_(self)  
  Определяет поведение copy.copy() для экземпляра вашего класса. copy.copy() возвращает *поверхностную копию* вашего объекта — это означает, что хоть сам объект и создан заново, все его данные ссылаются на данные оригинального объекта. И при изменении данных нового объекта, изменения будут происходить и в оригинальном.
* \_\_deepcopy\_\_(self, memodict={})  
  Определяет поведение copy.deepcopy() для экземпляров вашего класса. copy.deepcopy()возвращает *глубокую копию* вашего объекта — копируются и объект *и* его данные. memodict это кэш предыдущих скопированных объектов, он предназначен для оптимизации копирования и предотвращения бесконечной рекурсии, когда копируются рекурсивные структуры данных. Когда вы хотите полностью скопировать какой-нибудь конкретный атрибут, вызовите на нём copy.deepcopy() с первым параметром memodict.

Когда использовать эти магические методы? Как обычно — в любом случае, когда вам необходимо больше, чем стандартное поведение. Например, если вы пытаетесь скопировать объект, который содержит кэш как словарь (возможно, очень большой словарь), то может быть вам и не нужно копировать весь кэш, а обойтись всего одним в общей памяти объектов.

Использование модуля pickle на своих объектах

Pickle это модуль для сериализации структур данных Питона и он может быть невероятно полезен, когда вам нужно сохранить состояние какого-либо объекта и восстановить его позже (обычно, в целях кэширования). Кроме того, это ещё и отличный источник переживаний и путаницы.  
  
Сериализация настолько важна, что кроме своего модуля (pickle) имеет и свой собственный *протокол* и свои магические методы. Но для начала о том, как сериализовать с помощью pickle уже существующие типы данных (спокойно пропускайте, если вы уже знаете).

Вкратце про сериализацию

Давайте погрузимся в сериализацию. Допустим, у вас есть словарь, который вы хотите сохранить и восстановить позже. Вы должны записать его содержимое в файл, тщательно убедившись, что пишете с правильным синтаксисом, потом восстановить его, или выполнив exec(), или прочитав файл. Но это в лучшем случае рискованно: если вы храните важные данные в тексте, он может быть повреждён или изменён множеством способов, с целью обрушить вашу программу или, вообще, запустить какой-нибудь опасный код на вашем компьютере. Лучше использовать pickle:

import pickle

data = {'foo': [1, 2, 3],

'bar': ('Hello', 'world!'),

'baz': True}

jar = open('data.pkl', 'wb')

pickle.dump(data, jar) *# записать сериализованные данные в jar*

jar.close()

И вот, спустя несколько часов, нам снова нужен наш словарь:

import pickle

pkl\_file = open('data.pkl', 'rb') *# открываем*

data = pickle.load(pkl\_file) *# сохраняем в переменную*

print data

pkl\_file.close()

Что произошло? Точно то, что и ожидалось. data как-будто всегда тут и была.  
  
Теперь, немного об осторожности: pickle не идеален. Его файлы легко испортить случайно или преднамеренно. Pickle, может быть, безопаснее чем текстовые файлы, но он всё ещё может использоваться для запуска вредоносного кода. Кроме того, он несовместим между разными версиями Питона, поэтому если вы будете распространять свои объекты с помощью pickle, не ожидайте что все люди смогут их использовать. Тем не менее, модуль может быть мощным инструментом для кэширования и других распространённых задач с сериализацией.

Сериализация собственных объектов.

Модуль pickle не только для встроенных типов. Он может использоваться с каждым классом, реализующим его протокол. Этот протокол содержит четыре необязательных метода, позволяющих настроить то, как pickle будет с ними обращаться (есть некоторые различия для расширений на C, но это за рамками нашего руководства):

* \_\_getinitargs\_\_(self)  
  Если вы хотите, чтобы после десериализации вашего класса был вызыван \_\_init\_\_, вы можете определить \_\_getinitargs\_\_, который должен вернуть кортеж аргументов, который будет отправлен в \_\_init\_\_. Заметьте, что этот метод работает только с классами старого стиля.
* \_\_getnewargs\_\_(self)  
  Для классов нового стиля вы можете определить, какие параметры будут переданы в \_\_new\_\_ во время десериализации. Этот метод так же должен вернуть кортеж аргументов, которые будут отправлены в \_\_new\_\_.
* \_\_getstate\_\_(self)  
  Вместо стандартного атрибута \_\_dict\_\_, где хранятся атрибуты класса, вы можете вернуть произвольные данные для сериализации. Эти данные будут переданы в \_\_setstate\_\_ во время десериализации.
* \_\_setstate\_\_(self, state)  
  Если во время десериализации определён \_\_setstate\_\_, то данные объекта будут переданы сюда, вместо того чтобы просто записать всё в \_\_dict\_\_. Это парный метод для \_\_getstate\_\_: когда оба определены, вы можете представлять состояние вашего объекта так, как вы только захотите.
* \_\_reduce\_\_(self)  
  Если вы определили свой тип (с помощью Python's C API), вы должны сообщить Питону как его сериализовать, если вы хотите, чтобы он его сериализовал. \_\_reduce\_\_() вызывается когда сериализуется объект, в котором этот метод был определён. Он должен вернуть или строку, содержащую имя глобальной переменной, содержимое которой сериализуется как обычно, или кортеж. Кортеж может содержать от 2 до 5 элементов: вызываемый объект, который будет вызван, чтобы создать десериализованный объект, кортеж аргументов для этого вызываемого объекта, данные, которые будут переданы в \_\_setstate\_\_ (опционально), итератор списка элементов для сериализации (опционально) и итератор словаря элементов для сериализации (опционально).
* \_\_reduce\_ex\_\_(self, protocol)  
  Иногда полезно знать версию протокола, реализуя \_\_reduce\_\_. И этого можно добиться, реализовав вместо него \_\_reduce\_ex\_\_. Если \_\_reduce\_ex\_\_ реализован, то предпочтение при вызове отдаётся ему (вы всё-равно должны реализовать \_\_reduce\_\_ для обратной совместимости).

Пример

Для примера опишем грифельную доску (Slate), которая запоминает что и когда было на ней записано. Впрочем, конкретно эта доска становится чистой каждый раз, когда она сериализуется: текущее значение не сохраняется.

import time

class Slate:

'''Класс, хранящий строку и лог изменений. И забывающий своё значение после

сериализации.'''

def \_\_init\_\_(self, value):

self.value = value

self.last\_change = time.asctime()

self.history = {}

def change(self, new\_value):

*# Изменить значение. Зафиксировать последнее значение в истории.*

self.history[self.last\_change] = self.value

self.value = new\_value

self.last\_change = time.asctime()

def print\_changes(self):

print 'Changelog for Slate object:'

for k, v in self.history.items():

print '%s\t %s' % (k, v)

def \_\_getstate\_\_(self):

*# Намеренно не возвращаем self.value or self.last\_change.*

*# Мы хотим "чистую доску" после десериализации.*

return self.history

def \_\_setstate\_\_(self, state):

self.history = state

self.value, self.last\_change = None, None

Заключение

Цель этого руководства донести что-нибудь до каждого, кто его читает, независимо от его опыта в Питоне или объектно-ориентированном программировании. Если вы новичок в Питоне, вы получили ценные знания об основах написания функциональных, элегантных и простых для использования классов. Если вы программист среднего уровня, то вы, возможно, нашли несколько новых приятных идей и стратегий, несколько хороших способов уменьшить количество кода, написанного вами и вашими клиентами. Если же вы Питонист-эксперт, то вы обновили некоторые свои, возможно, подзабытые знания, а может и нашли парочку новых трюков. Независимо от вашего уровня, я надеюсь, что это путешествие через специальные питоновские методы было поистине магическим (не смог удержаться).

Дополнение 1: Как вызывать магические методы

Некоторые из магических методов напрямую связаны со встроенными функциями; в этом случае совершенно очевидно как их вызывать. Однако, так бывает не всегда. Это дополнение посвящено тому, чтобы раскрыть неочевидный синтаксис, приводящий к вызову магических методов.

| **Магический метод** | **Когда он вызывается (пример)** | **Объяснение** |
| --- | --- | --- |
| \_\_new\_\_(cls [,...]) | instance = MyClass(arg1, arg2) | \_\_new\_\_ вызывается при создании экземпляра |
| \_\_init\_\_(self [,...]) | instance = MyClass(arg1, arg2) | \_\_init\_\_ вызывается при создании экземпляра |
| \_\_cmp\_\_(self, other) | self == other, self > other, etc. | Вызывается для любого сравнения |
| \_\_pos\_\_(self) | +self | Унарный знак плюса |
| \_\_neg\_\_(self) | -self | Унарный знак минуса |
| \_\_invert\_\_(self) | ~self | Побитовая инверсия |
| \_\_index\_\_(self) | x[self] | Преобразование, когда объект используется как индекс |
| \_\_nonzero\_\_(self) | bool(self), if self: | Булевое значение объекта |
| \_\_getattr\_\_(self, name) | self.name # name не определено | Пытаются получить несуществующий атрибут |
| \_\_setattr\_\_(self, name, val) | self.name = val | Присвоение любому атрибуту |
| \_\_delattr\_\_(self, name) | del self.name | Удаление атрибута |
| \_\_getattribute\_\_(self, name) | self.name | Получить любой атрибут |
| \_\_getitem\_\_(self, key) | self[key] | Получение элемента через индекс |
| \_\_setitem\_\_(self, key, val) | self[key] = val | Присвоение элементу через индекс |
| \_\_delitem\_\_(self, key) | del self[key] | Удаление элемента через индекс |
| \_\_iter\_\_(self) | for x in self | Итерация |
| \_\_contains\_\_(self, value) | value in self, value not in self | Проверка принадлежности с помощью in |
| \_\_call\_\_(self [,...]) | self(args) | «Вызов» экземпляра |
| \_\_enter\_\_(self) | with self as x: | with оператор менеджеров контекста |
| \_\_exit\_\_(self, exc, val, trace) | with self as x: | with оператор менеджеров контекста |
| \_\_getstate\_\_(self) | pickle.dump(pkl\_file, self) | Сериализация |
| \_\_setstate\_\_(self) | data = pickle.load(pkl\_file) | Сериализация |

Надеюсь, эта таблица избавит вас от любых вопросов о том, что за синтаксис вызова магических методов.

Дополнение 2: Изменения в Питоне 3

Опишем несколько главных случаев, когда Питон 3 отличается от 2.x в терминах его объектной модели:

* Так как в Питоне 3 различий между строкой и юникодом больше нет, \_\_unicode\_\_ исчез, а появился \_\_bytes\_\_ (который ведёт себя так же как \_\_str\_\_ и \_\_unicode\_\_ в 2.7) для новых встроенных функций построения байтовых массивов.
* Так как деление в Питоне 3 теперь по-умолчанию «правильное деление», \_\_div\_\_ больше нет.
* \_\_coerce\_\_ больше нет, из-за избыточности и странного поведения.
* \_\_cmp\_\_ больше нет, из-за избыточности.
* \_\_nonzero\_\_ было переименовано в \_\_bool\_\_.
* next у итераторов был [переименован](http://www.python.org/dev/peps/pep-3114/) в \_\_next\_\_.