Магические методы и перегрузка операторов
Магический метод это такой метод, который вызывается функциями стандартной библиотеки Python, либо самим интерпретатором в неявном виде и является особенностью реализации. Магические методы практически никогда не требуется вызывать напрямую. Имя магического метода начинается и заканчивается двумя подчеркиваниями. Один из уже известных нам магических методов это __init__.

Операторы также реализуются в виде магический методов. Так, если a и b это экземпляры нашего класса, то при написании выражения a + b вызовется следующая конструкция: a.__add__(b).

Бинарные математические операторы
Обычная версия (__add__) срабатывает, если для левого операнда он определен. Обратная версия (__radd__) срабатывает, если для левого операнда обычная версия не определена (или возвращает объект NotImplemented), но определена обратная версия для правого операнда. Версия составного присваивания (__iadd__) срабатывает, если используется выражение составного присваивания с указанным оператором. Обычная и обратная версии не должны изменять свои аргументы и должны возвращать результат как новый объект. Версия составного присваивания должна изенять свой левый аргумент.

Выражение	Тип левого операнда	Тип правого операнда	Магический метод
`a + b`	Наш класс	Что угодно	`__add__(self, right)`
`a + b`	Что угодно	Наш класс	`__radd__(self, left)`
`a += b`	Наш класс	Что угодно	`__iadd__(self, right)`
`a - b`	Наш класс	Что угодно	`__sub__(self, right)`
`a - b`	Что угодно	Наш класс	`__rsub__(self, left)`
`a -= b`	Наш класс	Что угодно	`__isub__(self, right)`
`a * b`	Наш класс	Что угодно	`__mul__(self, right)`
`a * b`	Что угодно	Наш класс	`__rmul__(self, left)`
`a *= b`	Наш класс	Что угодно	`__imul__(self, right)`
`a @ b` (умножение матриц)	Наш класс	Что угодно	`__matmul__(self, right)`
`a @ b`	Что угодно	Наш класс	`__rmatmul__(self, left)`

`a @= b`	Наш класс	Что угодно	`__imatmul__(self, right)`
`a / b`	Наш класс	Что угодно	`__truediv__(self, right)`
`a / b`	Что угодно	Наш класс	`__rtruediv__(self, left)`
`a /= b`	Наш класс	Что угодно	`__itruediv__(self, right)`
`a // b`	Наш класс	Что угодно	`__floordiv__(self, right)`
`a // b`	Что угодно	Наш класс	`__rfloordiv__(self, left)`
`a //= b`	Наш класс	Что угодно	`__ifloordiv__(self, right)`
`a % b`	Наш класс	Что угодно	`__mod__(self, right)`
`a % b`	Что угодно	Наш класс	`__rmod__(self, left)`
`a %= b`	Наш класс	Что угодно	`__imod__(self, right)`
`a ** b` или `pow(a, b)`	Наш класс	Что угодно	`__pow__(self, right)`
`a ** b` или `pow(a, b)`	Что угодно	Наш класс	`__rpow__(self, left)`
`a **= b`	Наш класс	Что угодно	`__ipow__(self, right)`
`a << b`	Наш класс	Что угодно	`__lshift__(self, right)`
`a << b`	Что угодно	Наш класс	`__rlshift__(self, left)`
`a <<= b`	Наш класс	Что угодно	`__ilshift__(self, right)`
`a >> b`	Наш класс	Что угодно	`__rshift__(self, right)`
`a >> b`	Что угодно	Наш класс	`__rrshift__(self, left)`
`a >>= b`	Наш класс	Что угодно	`__irshift__(self, right)`
`a & b`	Наш класс	Что угодно	`__and__(self, right)`
`a & b`	Что угодно	Наш класс	`__rand__(self, left)`
`a &= b`	Наш класс	Что угодно	`__iand__(self, right)`
`a | b`	Наш класс	Что угодно	`__or__(self, right)`
`a | b`	Что угодно	Наш класс	`__ror__(self, left)`
`a |= b`	Наш класс	Что угодно	`__ior__(self, right)`
`a ^ b`	Наш класс	Что угодно	`__xor__(self, right)`

`a ^ b`	Что угодно	Наш класс	`__rxor__(self, left)`
`a ^= b`	Наш класс	Что угодно	`__ixor__(self, right)`

Бинарные логические операторы

Если реализован метод __eq__, но не реализован метод __ne__, метод __ne__ получает автоматическую реализацию, вызывающую метод __eq__ и инвертирующую его результат. Остальные выражения автоматически не определяются.

Выражение	Тип левого операнда	Тип правого операнда	Магический метод
`a == b`	Наш класс	Что угодно	`__eq__(self, right)`
`a != b`	Наш класс	Что угодно	`__ne__(self, right)`
`a < b`	Наш класс	Что угодно	`__lt__(self, right)`
`a > b`	Наш класс	Что угодно	`__gt__(self, right)`
`a <= b`	Наш класс	Что угодно	`__le__(self, right)`
`a >= b`	Наш класс	Что угодно	`__ge__(self, right`

Унарные математические операторы

Выражение	Магический метод
`+a`	`__pos__(self)`
`-a`	`__neg__(self)`
`~a`	`__invert__(self)`

Особые математические функции

Выражение	Магический метод	Примечание
`abs(a)`	`__abs__(self)`	
`round(a)` или `round(a, ndigits)`	`__round__(self)` или `__round(self, ndigits)`	
`trunc(a)`	`__trunc__(self)`	Если для типа не

		определен оператор int(), вызывается этот метод
`floor(a)`	`__floor__(self)`	
`ceil(a)`	`__ceil__(self)`	

Приведение к встроенным типам

Выражение	Магический метод	Примечание
`str(a)` или `print(a)`	`__str__(self)`	Этот метод должен возвращать “красивое” представление объекта, по-умолчанию используется при вызове print()
`bytes(a)`	`__bytes__(self)`	
`bool(a)` или `if a:` или `if !a:`	`__bool__(self)`	Если этот метод реализован, проверки на истинность и ложность вызывают его
`int(a)`	`__int__(self)`	
`float(a)`	`__float__(self)`	
`complex(a)`	`__complex__(self)`	
`some_list[a]`	`__index__(self)`	Вызывается при попытке использовать объект как целочисленный индекс; также этот метод вызывается при обращении к int(), float(), complex(), если соответствующие методы не реализованы.

Отладочное представление

Используется print(), если метод __str__() не реализован. Должен вернуть строку в формате ИмяКласса(значение_поля1, значение_поля2, …) или <ИмяКласса произвольное описание>:

`__repr__(self)`

Вызов как функция
Используется для создания т.н. функторов – классов, экземпляры которых могут вызываться как функции, но иметь дополнительные параметры помимо параметров функции за счет хранения их в полях класса.

```
__call__(self, param1, param2, …)
```

Используется:

```
a = MyClass()
a(1, 3.14, “test”)#в зависимости от списка параметров __call__
```

Определение коллекции
Если определен метод __len__(), проверка на истинность и ложность объекта вызывают его. Объект будет ложен, если __len__() вернет 0, и истиннен в любом другом случае.

Метод __missing__() вызывается методом __getitem__() классом dict и доступен для переопределения его наследниками.

Если метод __contains__() не определен, он будет реализован автоматически при помощи __iter__() или __getitem__().

Выражение	Магический метод
`len(a)`	`__len__(self)`
`x = a[index]`	`__getitem__(self, key)`
`a[index] = x`	`__setitem__(self, key, value)`
`del a[index]`	`__delitem__(self, key)`
`x = a[non_existent_key]`	`__missing__(self, key)`
`if x in a:` или `if x not in a:`	`__contains__(self, value)`
`for x in a:`	`__iter__(self)`

Конструирование объекта
Задачей статического магического метода __new__(cls, остальные параметры конструктора) является определение экземпляра класса, возвращаемого выражением конструктора. Если нужно создать новый экземпляр, достаточно вызвать реализацию __new__, унаследованную от класса object. В противном случае нужно вернуть уже существующий экземпляр.

В случае, если __new__ вернул экземпляр класса cls, в реализации конструктора по-умолчанию, будет вызван метод __init__(self, остальные параметры конструктора), где self это экземпляр, возвращенный методом __new__.

Метаклассы

Так как типы данных в Python также являются объектами, они должны иметь собственный тип данных. Таким образом, типом всех классов в Python является метакласс `type` или любой его наследник. Метакласс `type` сам является классом и тип этого класса тоже `type` (сам метакласс собственного типа). Для создания собственного метакласса необходимо унаследовать от `type`.

Собственные метаклассы используются для тонкой настройки создания экземпляров классов и в регулярном кодировании не требуются. Прежде чем использовать метаклассы, рассмотрите альтернативные способы решения вашей задачи.

Создание класса вручную

Допустим, мы хотим создать класс следующего вида:

```
class Foo(Bar):
    def baz(self, x):
        return x
```

При помощи метакласса `type` это будет выглядеть следующим образом:

```
def baz_impl(self, x):
    return x

Foo = type("Foo", (Bar,), {"baz": baz_impl})
```

Здесь первый параметр это имя класса, второй – кортеж, содержащий непосредственных предков класса, третий – словарь, содержащий атрибуты класса, такие как методы и статические поля класса.

Простая настройка метакласса

Можно создать функцию, которая выполнит необходимые преобразования имени класса, его предков и/или атрибутов и вызовет type() с измененными параметрами:

```
def simple_metaclass(name, parents, attrs):
    # изменить нужное
    return type(name, parents, attrs)
```

Настройка метакласса как класса

```
class SimpleMetaclass(type):
    def __new__(cls, name, parents, attrs):
        # изменить нужно
        return type(name, parents, attrs)
```

Настройка конструирования класса
Выражение-конструктор это всего лишь вызов магического метода __call__() у экземпляра метакласса (т.е. класса).

Если нужно создать новый экземпляр, достаточно вызвать реализацию __call__() от type.

```
class NotSoSimpleMetaclass(type):
    def __call__(self, constructor_param):
        if needs_to_construct():
            return super().__call__(constructor_param)
        return get_existing_instance()
```

Задать метакласс классу можно следующим образом:

```
class Test(metaclass=NotSoSimpleMetaclass):
    pass
```

Стоит помнить, что метаклассы наследуются и у класса может быть только один метакласс. Так, нельзя унаследовать от двух классов, определяющих разные собственные метаклассы.

Мы не рекомендуем создавать собственные метаклассы в любых целях (если только вы не понимаете отчетливо, что метакласс это именно то, что вам нужно). Данный раздел включен в курс только потому, что один из шаблонов проектирования, изучаемый впоследствии, реализуется при помощи метакласса.