A Guide to Python's Magic Methods

Rafe Kettler

Version 1.17

A PDF version of this guide can be obtained from my site or Github. The magic methods guide has a git repository at http://www.github.com/RafeKettler/magicmethods. Any issues can be reported there, along with comments, (or even contributions!).

Table of Contents

Introduction

This guide is the culmination of a few months' worth of blog posts. The subject is **magic methods**.

What are magic methods? They're everything in object-oriented Python. They're special methods that you can define to add "magic" to your classes. They're always surrounded by double underscores (e.g. `__init__` or `__lt__`). They're also not as well documented as they need to be. All of the magic methods for Python appear in the same section in the Python docs, but they're scattered about and only loosely organized. There's hardly an example to be found in that section (and that may very well be by design, since they're all detailed in the *language reference*, along with boring syntax descriptions, etc.).

So, to fix what I perceived as a flaw in Python's documentation, I set out to provide some more plain-English, example-driven documentation for Python's magic methods. I started out with weekly blog posts, and now that I've finished with those, I've put together this guide.

I hope you enjoy it. Use it as a tutorial, a refresher, or a reference; it's just intended to be a user-friendly guide to Python's magic methods.

Construction and Initialization

Everyone knows the most basic magic method, `__init__`. It's the way that we can define the initialization behavior of an object. However, when I call `x = SomeClass()`, `__init__` is not the first thing to get called. Actually, it's a method called `__new__`, which actually creates the instance, then passes any arguments at creation on to the initializer. At the other end of the object's lifespan, there's `__del__`. Let's take a closer

look at these 3 magic methods:

`__new__(cls, [...)`
: `__new__` is the first method to get called in an object's instantiation. It takes the class, then any other arguments that it will pass along to `__init__`. `__new__` is used fairly rarely, but it does have its purposes, particularly when subclassing an immutable type like a tuple or a string. I don't want to go in to too much detail on `__new__` because it's not too useful, but it is covered in great detail in the Python docs.

`__init__(self, [...)`
: The initializer for the class. It gets passed whatever the primary constructor was called with (so, for example, if we called `x = SomeClass(10, 'foo')`, `__init__` would get passed `10` and `'foo'` as arguments. `__init__` is almost universally used in Python class definitions.

`__del__(self)`
: If `__new__` and `__init__` formed the constructor of the object, `__del__` is the destructor. It doesn't implement behavior for the statement `del x` (so that code would not translate to `x.__del__()`). Rather, it defines behavior for when an object is garbage collected. It can be quite useful for objects that might require extra cleanup upon deletion, like sockets or file objects. Be careful, however, as there is no guarantee that `__del__` will be executed if the object is still alive when the interpreter exits, so `__del__` can't serve as a replacement for good coding practices (like always closing a connection when you're done with it. In fact, `__del__` should almost never be used because of the precarious circumstances under which it is called; use it with caution!

Putting it all together, here's an example of `__init__` and `__del__` in action:

```
from os.path import join

class FileObject:
    '''Wrapper for file objects to make sure the file gets closed on deletion.'''

    def __init__(self, filepath='~', filename='sample.txt'):
        # open a file filename in filepath in read and write mode
        self.file = open(join(filepath, filename), 'r+')

    def __del__(self):
        self.file.close()
        del self.file
```

Making Operators Work on Custom Classes

One of the biggest advantages of using Python's magic methods is that they provide a simple way to make objects behave like built-in types. That means you can avoid ugly, counter-intuitive, and nonstandard ways of performing basic operators. In some languages, it's common to do something like this:

```
if instance.equals(other_instance):
    # do something
```

You could certainly do this in Python, too, but this adds confusion and is unnecessarily verbose. Different libraries might use different names for the same operations, making the client do way more work than necessary. With the power of magic methods, however, we can define one method (`__eq__`, in this case), and say what we *mean* instead:

```
if instance == other_instance:
    #do something
```

That's part of the power of magic methods. The vast majority of them allow us to define meaning for operators so that we can use them on our own classes just like they were built in types.

Comparison magic methods

Python has a whole slew of magic methods designed to implement intuitive comparisons between objects using operators, not awkward method calls. They also provide a way to override the default Python behavior for comparisons of objects (by reference). Here's the list of those methods and what they do:

`__cmp__(self, other)`
: `__cmp__` is the most basic of the comparison magic methods. It actually implements behavior for all of the comparison operators (<, ==, !=, etc.), but it might not do it the way you want (for example, if whether one instance was equal to another were determined by one criterion and and whether an instance is greater than another were determined by something else). `__cmp__` should return a negative integer if `self < other`, zero if `self == other`, and positive if `self > other`. It's usually best to define each comparison you need rather than define them all at once, but `__cmp__` can be a good way to save repetition and improve clarity when you need all comparisons implemented with similar criteria.

`__eq__(self, other)`
: Defines behavior for the equality operator, `==`.

`__ne__(self, other)`
: Defines behavior for the inequality operator, `!=`.

`__lt__(self, other)`
: Defines behavior for the less-than operator, `<`.

`__gt__(self, other)`
: Defines behavior for the greater-than operator, `>`.

`__le__(self, other)`
: Defines behavior for the less-than-or-equal-to operator, `<=`.

`__ge__(self, other)`
: Defines behavior for the greater-than-or-equal-to operator, `>=`.

For an example, consider a class to model a word. We might want to compare words lexicographically (by the alphabet), which is the default comparison behavior for strings, but we also might want to do it based on some other criterion, like length or number of syllables. In this example, we'll compare by length. Here's an implementation:

```
class Word(str):
    '''Class for words, defining comparison based on word length.'''

    def __new__(cls, word):
        # Note that we have to use __new__. This is because str is an immutable
        # type, so we have to initialize it early (at creation)
        if ' ' in word:
            print "Value contains spaces. Truncating to first space."
            word = word[:word.index(' ')] # Word is now all chars before first space
        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)
```

Now, we can create two `Word`s (by using `Word('foo')` and `Word('bar')`) and compare them based on length. Note, however, that we didn't define `__eq__` and `__ne__`. This is because this would lead to some weird behavior (notably that `Word('foo') == Word('bar')` would evaluate to true). It wouldn't make sense to test for equality based on length, so we fall back on `str`'s implementation of equality.

Now would be a good time to note that you don't have to define every comparison magic method to get rich comparisons. The standard library has kindly provided us with a class decorator in the module

`functools` that will define all rich comparison methods if you only define `__eq__` and one other (e.g. `__gt__`, `__lt__`, etc.) This feature is only available in Python 2.7, but when you get a chance it saves a great deal of time and effort. You can use it by placing `@total_ordering` above your class definition.

Numeric magic methods

Just like you can create ways for instances of your class to be compared with comparison operators, you can define behavior for numeric operators. Buckle your seat belts, folks, there's a lot of these. For organization's sake, I've split the numeric magic methods into 5 categories: unary operators, normal arithmetic operators, reflected arithmetic operators (more on this later), augmented assignment, and type conversions.

Unary operators and functions

Unary operators and functions only have one operand, e.g. negation, absolute value, etc.

`__pos__(self)`
: Implements behavior for unary positive (e.g. `+some_object`)

`__neg__(self)`
: Implements behavior for negation (e.g. `-some_object`)

`__abs__(self)`
: Implements behavior for the built in `abs()` function.

`__invert__(self)`
: Implements behavior for inversion using the `~` operator. For an explanation on what this does, see the Wikipedia article on bitwise operations.

`__round__(self, n)`
: Implements behavior for the built in `round()` function. `n` is the number of decimal places to round to.

`__floor__(self)`
: Implements behavior for `math.floor()`, i.e., rounding down to the nearest integer.

`__ceil__(self)`
: Implements behavior for `math.ceil()`, i.e., rounding up to the nearest integer.

`__trunc__(self)`
: Implements behavior for `math.trunc()`, i.e., truncating to an integral.

Normal arithmetic operators

Now, we cover the typical binary operators (and a function or two): +, -, * and the like. These are, for the most part, pretty self-explanatory.

`__add__(self, other)`
: Implements addition.

`__sub__(self, other)`
: Implements subtraction.

`__mul__(self, other)`
: Implements multiplication.

`__floordiv__(self, other)`
: Implements integer division using the `//` operator.

`__div__(self, other)`
: Implements division using the `/` operator.

`__truediv__(self, other)`
: Implements *true* division. Note that this only works when `from __future__ import division` is in effect.

`__mod__(self, other)`
Implements modulo using the `%` operator.

`__divmod__(self, other)`
Implements behavior for long division using the `divmod()` built in function.

`__pow__`
Implements behavior for exponents using the `**` operator.

`__lshift__(self, other)`
Implements left bitwise shift using the `<<` operator.

`__rshift__(self, other)`
Implements right bitwise shift using the `>>` operator.

`__and__(self, other)`
Implements bitwise and using the `&` operator.

`__or__(self, other)`
Implements bitwise or using the `|` operator.

`__xor__(self, other)`
Implements bitwise xor using the `^` operator.

Reflected arithmetic operators

You know how I said I would get to reflected arithmetic in a bit? Some of you might think it's some big, scary, foreign concept. It's actually quite simple. Here's an example:

```
some_object + other
```

That was "normal" addition. The reflected equivalent is the same thing, except with the operands switched around:

```
other + some_object
```

So, all of these magic methods do the same thing as their normal equivalents, except the perform the operation with other as the first operand and self as the second, rather than the other way around. In most cases, the result of a reflected operation is the same as its normal equivalent, so you may just end up defining `__radd__` as calling `__add__` and so on. Note that the object on the left hand side of the operator (`other` in the example) must not define (or return `NotImplemented`) for its definition of the non-reflected version of an operation. For instance, in the example, `some_object.__radd__` will only be called if `other` does not define `__add__`.

`__radd__(self, other)`
Implements reflected addition.

`__rsub__(self, other)`
Implements reflected subtraction.

`__rmul__(self, other)`
Implements reflected multiplication.

`__rfloordiv__(self, other)`
Implements reflected integer division using the `//` operator.

`__rdiv__(self, other)`
Implements reflected division using the `/` operator.

`__rtruediv__(self, other)`
Implements reflected *true* division. Note that this only works when `from __future__ import division` is in effect.

`__rmod__(self, other)`

Implements reflected modulo using the `%` operator.

`__rdivmod__(self, other)`
Implements behavior for long division using the `divmod()` built in function, when `divmod(other, self)` is called.

`__rpow__`
Implements behavior for reflected exponents using the `**` operator.

`__rlshift__(self, other)`
Implements reflected left bitwise shift using the `<<` operator.

`__rrshift__(self, other)`
Implements reflected right bitwise shift using the `>>` operator.

`__rand__(self, other)`
Implements reflected bitwise and using the `&` operator.

`__ror__(self, other)`
Implements reflected bitwise or using the `|` operator.

`__rxor__(self, other)`
Implements reflected bitwise xor using the `^` operator.

Augmented assignment

Python also has a wide variety of magic methods to allow custom behavior to be defined for augmented assignment. You're probably already familiar with augmented assignment, it combines "normal" operators with assignment. If you still don't know what I'm talking about, here's an example:

```
x = 5
x += 1 # in other words x = x + 1
```

Each of these methods should return the value that the variable on the left hand side should be assigned to (for instance, for `a += b`, `__iadd__` might return `a + b`, which would be assigned to `a`). Here's the list:

`__iadd__(self, other)`
Implements addition with assignment.

`__isub__(self, other)`
Implements subtraction with assignment.

`__imul__(self, other)`
Implements multiplication with assignment.

`__ifloordiv__(self, other)`
Implements integer division with assignment using the `//=` operator.

`__idiv__(self, other)`
Implements division with assignment using the `/=` operator.

`__itruediv__(self, other)`
Implements *true* division with assignment. Note that this only works when `from __future__ import division` is in effect.

`__imod__(self, other)`
Implements modulo with assignment using the `%=` operator.

`__ipow__`
Implements behavior for exponents with assignment using the `**=` operator.

`__ilshift__(self, other)`
Implements left bitwise shift with assignment using the `<<=` operator.

`__irshift__(self, other)`
Implements right bitwise shift with assignment using the `>>=` operator.

`__iand__(self, other)`
Implements bitwise and with assignment using the `&=` operator.

`__ior__(self, other)`
Implements bitwise or with assignment using the `|=` operator.

`__ixor__(self, other)`
Implements bitwise xor with assignment using the `^=` operator.

Type conversion magic methods

Python also has an array of magic methods designed to implement behavior for built in type conversion functions like `float()`. Here they are:

`__int__(self)`
Implements type conversion to int.

`__long__(self)`
Implements type conversion to long.

`__float__(self)`
Implements type conversion to float.

`__complex__(self)`
Implements type conversion to complex.

`__oct__(self)`
Implements type conversion to octal.

`__hex__(self)`
Implements type conversion to hexadecimal.

`__index__(self)`
Implements type conversion to an int when the object is used in a slice expression. If you define a custom numeric type that might be used in slicing, you should define `__index__`.

`__trunc__(self)`
Called when `math.trunc(self)` is called. `__trunc__` should return the value of `self truncated to an integral type (usually a long).

`__coerce__(self, other)`
Method to implement mixed mode arithmetic. `__coerce__` should return `None` if type conversion is impossible. Otherwise, it should return a pair (2-tuple) of `self` and `other`, manipulated to have the same type.

Representing your Classes

It's often useful to have a string representation of a class. In Python, there's a few methods that you can implement in your class definition to customize how built in functions that return representations of your class behave.

`__str__(self)`
Defines behavior for when `str()` is called on an instance of your class.

`__repr__(self)`
Defines behavior for when `repr()` is called on an instance of your class. The major difference between `str()` and `repr()` is intended audience. `repr()` is intended to produce output that is mostly machine-readable (in many cases, it could be valid Python code even), whereas `str()` is intended to be human-readable.

`__unicode__(self)`
Defines behavior for when `unicode()` is called on an instance of your class. `unicode()` is like `str()`, but it returns a unicode string. Be wary: if a client calls `str()` on an instance of your class and you've only defined `__unicode__()`, it won't work. You should always try to define `__str__()` as well in case someone doesn't have the luxury of using unicode.

`__format__(self, formatstr)`
Defines behavior for when an instance of your class is used in new-style string formatting. For instance, `"Hello, {0:abc}!".format(a)` would lead to the call `a.__format__("abc")`. This can be useful for defining your own numerical or string types that you might like to give special formatting options.

`__hash__(self)`
Defines behavior for when `hash()` is called on an instance of your class. It has to return an integer, and its result is used for quick key comparison in dictionaries. Note that this usually entails implementing `__eq__` as well. Live by the following rule: `a == b` implies `hash(a) == hash(b)`.

`__nonzero__(self)`
Defines behavior for when `bool()` is called on an instance of your class. Should return True or False, depending on whether you would want to consider the instance to be True or False.

`__dir__(self)`
Defines behavior for when `dir()` is called on an instance of your class. This method should return a list of attributes for the user. Typically, implementing `__dir__` is unnecessary, but it can be vitally important for interactive use of your classes if you redefine `__getattr__` or `__getattribute__` (which you will see in the next section) or are otherwise dynamically generating attributes.

`__sizeof__(self)`
Defines behavior for when `sys.getsizeof()` is called on an instance of your class. This should return the size of your object, in bytes. This is generally more useful for Python classes implemented in C extensions, but it helps to be aware of it.

We're pretty much done with the boring (and example-free) part of the magic methods guide. Now that we've covered some of the more basic magic methods, it's time to move to more advanced material.

Controlling Attribute Access

Many people coming to Python from other languages complain that it lacks true encapsulation for classes (e.g. no way to define private attributes and then have public getter and setters). This couldn't be farther than the truth: it just happens that Python accomplishes a great deal of encapsulation through "magic", instead of explicit modifiers for methods or fields. Take a look:

`__getattr__(self, name)`
You can define behavior for when a user attempts to access an attribute that doesn't exist (either at all or yet). This can be useful for catching and redirecting common misspellings, giving warnings about using deprecated attributes (you can still choose to compute and return that attribute, if you wish), or deftly handing an `AttributeError`. It only gets called when a nonexistent attribute is accessed, however, so it isn't a true encapsulation solution.

`__setattr__(self, name, value)`
Unlike `__getattr__`, `__setattr__` is an encapsulation solution. It allows you to define behavior for assignment to an attribute regardless of whether or not that attribute exists, meaning you can define custom rules for any changes in the values of attributes. However, you have to be careful with how you use `__setattr__`, as the example at the end of the list will show.

`__delattr__(self, name)`
This is the exact same as `__setattr__`, but for deleting attributes instead of setting them. The same precautions need to be taken as with `__setattr__` as well in order to prevent infinite recursion (calling `del self.name` in the implementation of `__delattr__` would cause infinite recursion).

`__getattribute__(self, name)`
After all this, `__getattribute__` fits in pretty well with its companions `__setattr__` and `__delattr__`. However, I don't recommend you use it. `__getattribute__` can only be used with new-style classes (all classes are new-style in the newest versions of Python, and in older versions you can make a class new-style by subclassing `object`. It allows you to define rules for whenever an attribute's value is accessed. It suffers from some similar infinite recursion problems as its partners-in-crime (this time you call the base class's `__getattribute__` method to prevent this). It also mainly obviates the need for `__getattr__`, which only gets called when `__getattribute__` is implemented if it is called explicitly or an `AttributeError` is raised. This method can be used (after all, it's your

choice), but I don't recommend it because it has a small use case (it's far more rare that we need special behavior to retrieve a value than to assign to it) and because it can be really difficult to implement bug-free.

You can easily cause a problem in your definitions of any of the methods controlling attribute access. Consider this example:

```
def __setattr__(self, name, value):
    self.name = value
    # since every time an attribute is assigned, __setattr__() is called, this
    # is recursion.
    # so this really means self.__setattr__('name', value). Since the method
    # keeps calling itself, the recursion goes on forever causing a crash

def __setattr__(self, name, value):
    self.__dict__[name] = value # assigning to the dict of names in the class
    # define custom behavior here
```

Again, Python's magic methods are incredibly powerful, and with great power comes great responsibility. It's important to know the proper way to use magic methods so you don't break any code.

So, what have we learned about custom attribute access in Python? It's not to be used lightly. In fact, it tends to be excessively powerful and counter-intuitive. But the reason why it exists is to scratch a certain itch: Python doesn't seek to make bad things impossible, but just to make them difficult. Freedom is paramount, so you can really do whatever you want. Here's an example of some of the special attribute access methods in action (note that we use `super` because not all classes have an attribute `__dict__`):

```
class AccessCounter(object):
    '''A class that contains a value and implements an access counter.
    The counter increments each time the value is changed.'''

    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)
        # Make this unconditional.
        # If you want to prevent other attributes to be set, raise 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)
```

Making Custom Sequences

There's a number of ways to get your Python classes to act like built in sequences (dict, tuple, list, string, etc.). These are by far my favorite magic methods in Python because of the absurd degree of control they give you and the way that they magically make a whole array of global functions work beautifully on instances of your class. But before we get down to the good stuff, a quick word on requirements.

Requirements

Now that we're talking about creating your own sequences in Python, it's time to talk about *protocols*. Protocols are somewhat similar to interfaces in other languages in that they give you a set of methods you must define. However, in Python protocols are totally informal and require no explicit declarations to implement. Rather, they're more like guidelines.

Why are we talking about protocols now? Because implementing custom container types in Python involves using some of these protocols. First, there's the protocol for defining immutable containers: to make an immutable container, you need only define `__len__` and `__getitem__` (more on these later). The mutable container protocol requires everything that immutable containers require plus `__setitem__` and `__delitem__`. Lastly, if you want your object to be iterable, you'll have to define `__iter__`, which returns an iterator. That iterator must conform to an iterator protocol, which requires iterators to have methods called `__iter__` (returning itself) and `next`.

The magic behind containers

Without any more wait, here are the magic methods that containers use:

`__len__(self)`
Returns the length of the container. Part of the protocol for both immutable and mutable containers.

`__getitem__(self, key)`
Defines behavior for when an item is accessed, using the notation `self[key]`. This is also part of both the mutable and immutable container protocols. It should also raise appropriate exceptions: `TypeError` if the type of the key is wrong and `KeyError` if there is no corresponding value for the key.

`__setitem__(self, key, value)`
Defines behavior for when an item is assigned to, using the notation `self[nkey] = value`. This is part of the mutable container protocol. Again, you should raise `KeyError` and `TypeError` where appropriate.

`__delitem__(self, key)`
Defines behavior for when an item is deleted (e.g. `del self[key]`). This is only part of the mutable container protocol. You must raise the appropriate exceptions when an invalid key is used.

`__iter__(self)`
Should return an iterator for the container. Iterators are returned in a number of contexts, most notably by the `iter()` built in function and when a container is looped over using the form `for x in container:`. Iterators are their own objects, and they also must define an `__iter__` method that returns `self`.

`__reversed__(self)`
Called to implement behavior for the `reversed()` built in function. Should return a reversed version of the sequence. Implement this only if the sequence class is ordered, like list or tuple.

`__contains__(self, item)`
`__contains__` defines behavior for membership tests using `in` and `not in`. Why isn't this part of a sequence protocol, you ask? Because when `__contains__` isn't defined, Python just iterates over the sequence and returns `True` if it comes across the item it's looking for.

`__missing__(self, key)`
`__missing__` is used in subclasses of `dict`. It defines behavior for whenever a key is accessed that does not exist in a dictionary (so, for instance, if I had a dictionary `d` and said `d["george"]` when `"george"` is not a key in the dict, `d.__missing__("george")` would be called).

An example

For our example, let's look at a list that implements some functional constructs that you might be used to from other languages (Haskell, for example).

```
class FunctionalList:
    '''A class wrapping a list with some extra functional magic, like head,
    tail, init, last, drop, and 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):
        # if key is of invalid type or value, the list values will raise the error
        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):
        # get the first element
        return self.values[0]
    def tail(self):
        # get all elements after the first
        return self.values[1:]
    def init(self):
        # get elements up to the last
        return self.values[:-1]
    def last(self):
        # get last element
        return self.values[-1]
    def drop(self, n):
        # get all elements except first n
        return self.values[n:]
    def take(self, n):
        # get first n elements
        return self.values[:n]
```

There you have it, a (marginally) useful example of how to implement your own sequence. Of course, there are more useful applications of custom sequences, but quite a few of them are already implemented in the standard library (batteries included, right?), like `Counter`, `OrderedDict`, and `NamedTuple`.

Reflection

You can also control how reflection using the built in functions `isinstance()` and `issubclass()` behaves by defining magic methods. The magic methods are:

`__instancecheck__(self, instance)`
: Checks if an instance is an instance of the class you defined (e.g. `isinstance(instance, class)`.

`__subclasscheck__(self, subclass)`
: Checks if a class subclasses the class you defined (e.g. `issubclass(subclass, class)`).

The use case for these magic methods might seem small, and that may very well be true. I won't spend too much more time on reflection magic methods because they aren't very important, but they reflect something important about object-oriented programming in Python and Python in general: there is almost always an easy way to do something, even if it's rarely necessary. These magic methods might not seem useful, but if you ever need them you'll be glad that they're there (and that you read this guide!).

Callable Objects

As you may already know, in Python, functions are first-class objects. This means that they can be passed to functions and methods just as if they were objects of any other kind. This is an incredibly powerful feature.

A special magic method in Python allows instances of your classes to behave as if they were functions, so that you can "call" them, pass them to functions that take functions as arguments, and so on. This is another powerful convenience feature that makes programming in Python that much sweeter.

`__call__(self, [args...])`
Allows an instance of a class to be called as a function. Essentially, this means that `x()` is the same as `x.__call__()`. Note that `__call__` takes a variable number of arguments; this means that you define `__call__` as you would any other function, taking however many arguments you'd like it to.

`__call__` can be particularly useful in classes whose instances that need to often change state. "Calling" the instance can be an intuitive and elegant way to change the object's state. An example might be a class representing an entity's position on a plane:

```
class Entity:
    '''Class to represent an entity. Callable to update the entity's position.'''

    def __init__(self, size, x, y):
        self.x, self.y = x, y
        self.size = size

    def __call__(self, x, y):
        '''Change the position of the entity.'''
        self.x, self.y = x, y

    # snip...
```

Context Managers

In Python 2.5, a new keyword was introduced in Python along with a new method for code reuse, the `with` statement. The concept of context managers was hardly new in Python (it was implemented before as a part of the library), but not until PEP 343 was accepted did it achieve status as a first class language construct. You may have seen with statements before:

```
with open('foo.txt') as bar:
    # perform some action with bar
```

Context managers allow setup and cleanup actions to be taken for objects when their creation is wrapped with a `with` statement. The behavior of the context manager is determined by two magic methods:

`__enter__(self)`
Defines what the context manager should do at the beginning of the block created by the `with` statement. Note that the return value of `__enter__` is bound to the *target* of the `with` statement, or the name after the `as`.

`__exit__(self, exception_type, exception_value, traceback)`
Defines what the context manager should do after its block has been executed (or terminates). It can be used to handle exceptions, perform cleanup, or do something always done immediately after the action in the block. If the block executes successfully, `exception_type`, `exception_value`, and `traceback` will be `None`. Otherwise, you can choose to handle the exception or let the user handle it; if you want to handle it, make sure `__exit__` returns `True` after all is said and done. If you don't want the exception to be handled by the context manager, just let it happen.

`__enter__` and `__exit__` can be useful for specific classes that have well-defined and common behavior for setup and cleanup. You can also use these methods to create generic context managers that wrap other objects. Here's an example:

```
class Closer:
    '''A context manager to automatically close an object with a close method
    in a with statement.'''

    def __init__(self, obj):
        self.obj = obj

    def __enter__(self):
        return self.obj # bound to target

    def __exit__(self, exception_type, exception_val, trace):
        try:
           self.obj.close()
        except AttributeError: # obj isn't closable
           print 'Not closable.'
           return True # exception handled successfully
```

Here's an example of `Closer` in action, using an FTP connection to demonstrate it (a closable socket):

```
>>> 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
```

See how our wrapper gracefully handled both proper and improper uses? That's the power of context managers and magic methods. Note that the Python standard library includes a module contextlib that contains a context manager, `contextlib.closing()`, that does approximately the same thing (without any handling of the case where an object does not have a `close()` method).

Abstract Base Classes

See http://docs.python.org/2/library/abc.html.

Building Descriptor Objects

Descriptors are classes which, when accessed through either getting, setting, or deleting, can also alter other objects. Descriptors aren't meant to stand alone; rather, they're meant to be held by an owner class. Descriptors can be useful when building object-oriented databases or classes that have attributes whose values are dependent on each other. Descriptors are particularly useful when representing attributes in several different units of measurement or representing computed attributes (like distance from the origin in a class to represent a point on a grid).

To be a descriptor, a class must have at least one of `__get__`, `__set__`, and `__delete__` implemented. Let's take a look at those magic methods:

`__get__(self, instance, owner)`
: Define behavior for when the descriptor's value is retrieved. `instance` is the instance of the owner object. `owner` is the owner class itself.

`__set__(self, instance, value)`
: Define behavior for when the descriptor's value is changed. `instance` is the instance of the owner

class and `value` is the value to set the descriptor to.

`__delete__(self, instance)`
Define behavior for when the descriptor's value is deleted. `instance` is the instance of the owner object.

Now, an example of a useful application of descriptors: unit conversions.

```
class Meter(object):
    '''Descriptor for a meter.'''

    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):
    '''Descriptor for a foot.'''

    def __get__(self, instance, owner):
        return instance.meter * 3.2808
    def __set__(self, instance, value):
        instance.meter = float(value) / 3.2808

class Distance(object):
    '''Class to represent distance holding two descriptors for feet and
    meters.'''
    meter = Meter()
    foot = Foot()
```

Copying

Sometimes, particularly when dealing with mutable objects, you want to be able to copy an object and make changes without affecting what you copied from. This is where Python's `copy` comes into play. However (fortunately), Python modules are not sentient, so we don't have to worry about a Linux-based robot uprising, but we do have to tell Python how to efficiently copy things.

`__copy__(self)`
Defines behavior for `copy.copy()` for instances of your class. `copy.copy()` returns a *shallow copy* of your object -- this means that, while the instance itself is a new instance, all of its data is referenced -- i.e., the object itself is copied, but its data is still referenced (and hence changes to data in a shallow copy may cause changes in the original).

`__deepcopy__(self, memodict={})`
Defines behavior for `copy.deepcopy()` for instances of your class. `copy.deepcopy()` returns a *deep copy* of your object -- the object *and* its data are both copied. `memodict` is a cache of previously copied objects -- this optimizes copying and prevents infinite recursion when copying recursive data structures. When you want to deep copy an individual attribute, call `copy.deepcopy()` on that attribute with `memodict` as the first argument.

What are some use cases for these magic methods? As always, in any case where you need more fine-grained control than what the default behavior gives you. For instance, if you are attempting to copy an object that stores a cache as a dictionary (which might be large), it might not make sense to copy the cache as well -- if the cache can be shared in memory between instances, then it should be.

Pickling Your Objects

If you spend time with other Pythonistas, chances are you've at least heard of pickling. Pickling is a serialization process for Python data structures, and can be incredibly useful when you need to store an object and retrieve it later (usually for caching). It's also a major source of worries and confusion.

Pickling is so important that it doesn't just have its own module (`pickle`), but its own *protocol* and the magic methods to go with it. But first, a brief word on how to pickle existing types(feel free to skip it if you already know).

Pickling: A Quick Soak in the Brine

Let's dive into pickling. Say you have a dictionary that you want to store and retrieve later. You couldwrite it's contents to a file, carefully making sure that you write correct syntax, then retrieve it using either `exec()` or processing the file input. But this is precarious at best: if you store important data in plain text, it could be corrupted or changed in any number of ways to make your program crash or worse run malicious code on your computer. Instead, we're going to pickle it:

```
import pickle

data = {'foo': [1, 2, 3],
        'bar': ('Hello', 'world!'),
        'baz': True}
jar = open('data.pkl', 'wb')
pickle.dump(data, jar) # write the pickled data to the file jar
jar.close()
```

Now, a few hours later, we want it back. All we have to do is unpickle it:

```
import pickle

pkl_file = open('data.pkl', 'rb') # connect to the pickled data
data = pickle.load(pkl_file) # load it into a variable
print data
pkl_file.close()
```

What happens? Exactly what you expect. It's just like we had `data` all along.

Now, for a word of caution: pickling is not perfect. Pickle files are easily corrupted on accident and on purpose. Pickling may be more secure than using flat text files, but it still can be used to run malicious code. It's also incompatible across versions of Python, so don't expect to distribute pickled objects and expect people to be able to open them. However, it can also be a powerful tool for caching and other common serialization tasks.

Pickling your own Objects

Pickling isn't just for built-in types. It's for any class that follows the pickle protocol. The pickle protocol has four optional methods for Python objects to customize how they act (it's a bit different for C extensions, but that's not in our scope):

`__getinitargs__(self)`
: If you'd like for `__init__` to be called when your class is unpickled, you can define `__getinitargs__` , which should return a tuple of the arguments that you'd like to be passed to `__init__` . Note that this method will only work for old-style classes.

`__getnewargs__(self)`
: For new-style classes, you can influence what arguments get passed to `__new__` upon unpickling. This method should also return a tuple of arguments that will then be passed to `__new__` .

`__getstate__(self)`
: Instead of the object's `__dict__` attribute being stored, you can return a custom state to be stored when the object is pickled. That state will be used by `__setstate__` when the object is unpickled.

`__setstate__(self, state)`
: When the object is unpickled, if `__setstate__` is defined the object's state will be passed to it instead of directly applied to the object's `__dict__` . This goes hand in hand with `__getstate__` : when both are defined, you can represent the object's pickled state however you want with whatever you want.

`__reduce__(self)`

When defining extension types (i.e., types implemented using Python's C API), you have to tell Python how to pickle them if you want them to pickle them. `__reduce__()` is called when an object defining it is pickled. It can either return a string representing a global name that Python will look up and pickle, or a tuple. The tuple contains between 2 and 5 elements: a callable object that is called to recreate the object, a tuple of arguments for that callable object, state to be passed to `__setstate__` (optional), an iterator yielding list items to be pickled (optional), and an iterator yielding dictionary items to be pickled (optional).

`__reduce_ex__(self)`

`__reduce_ex__` exists for compatibility. If it is defined, `__reduce_ex__` will be called over `__reduce__` on pickling. `__reduce__` can be defined as well for older versions of the pickling API that did not support `__reduce_ex__`.

An Example

Our example is a `Slate`, which remembers what its values have been and when those values were written to it. However, this particular slate goes blank each time it is pickled: the current value will not be saved.

```
import time

class Slate:
    '''Class to store a string and a changelog, and forget its value when
    pickled.'''

    def __init__(self, value):
        self.value = value
        self.last_change = time.asctime()
        self.history = {}

    def change(self, new_value):
        # Change the value. Commit last value to history
        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):
        # Deliberately do not return self.value or self.last_change.
        # We want to have a "blank slate" when we unpickle.
        return self.history

    def __setstate__(self, state):
        # Make self.history = state and last_change and value undefined
        self.history = state
        self.value, self.last_change = None, None
```

Conclusion

The goal of this guide is to bring something to anyone that reads it, regardless of their experience with Python or object-oriented programming. If you're just getting started with Python, you've gained valuable knowledge of the basics of writing feature-rich, elegant, and easy-to-use classes. If you're an intermediate Python programmer, you've probably picked up some slick new concepts and strategies and some good ways to reduce the amount of code written by you and clients. If you're an expert Pythonista, you've been refreshed on some of the stuff you might have forgotten about and maybe picked up a few new tricks along the way. Whatever your experience level, I hope that this trip through Python's special methods has been truly magical (I couldn't resist the final pun).

Appendix 1: How to Call Magic Methods

Some of the magic methods in Python directly map to built-in functions; in this case, how to invoke them is fairly obvious. However, in other cases, the invocation is far less obvious. This appendix is devoted to exposing non-obvious syntax that leads to magic methods getting called.

Magic Method	When it gets invoked (example)	Explanation
`__new__(cls [,...])`	`instance = MyClass(arg1, arg2)`	`__new__` is called on instance creation
`__init__(self [,...])`	`instance = MyClass(arg1, arg2)`	`__init__` is called on instance creation
`__cmp__(self, other)`	`self == other`, `self > other`, etc.	Called for any comparison
`__pos__(self)`	`+self`	Unary plus sign
`__neg__(self)`	`-self`	Unary minus sign
`__invert__(self)`	`~self`	Bitwise inversion
`__index__(self)`	`x[self]`	Conversion when object is used as index
`__nonzero__(self)`	`bool(self)`	Boolean value of the object
`__getattr__(self, name)`	`self.name # name doesn't exist`	Accessing nonexistent attribute
`__setattr__(self, name, val)`	`self.name = val`	Assigning to an attribute
`__delattr__(self, name)`	`del self.name`	Deleting an attribute
`__getattribute__(self, name)`	`self.name`	Accessing any attribute
`__getitem__(self, key)`	`self[key]`	Accessing an item using an index
`__setitem__(self, key, val)`	`self[key] = val`	Assigning to an item using an index
`__delitem__(self, key)`	`del self[key]`	Deleting an item using an index
`__iter__(self)`	`for x in self`	Iteration
`__contains__(self, value)`	`value in self`, `value not in self`	Membership tests using `in`
`__call__(self [,...])`	`self(args)`	"Calling" an instance
`__enter__(self)`	`with self as x:`	`with` statement context managers
`__exit__(self, exc, val, trace)`	`with self as x:`	`with` statement context managers
`__getstate__(self)`	`pickle.dump(pkl_file, self)`	Pickling
`__setstate__(self)`	`data = pickle.load(pkl_file)`	Pickling

Hopefully, this table should have cleared up any questions you might have had about what syntax invokes which magic method.

Appendix 2: Changes in Python 3

Here, we document a few major places where Python 3 differs from 2.x in terms of its object model:

- Since the distinction between string and unicode has been done away with in Python 3, `__unicode__` is gone and `__bytes__` (which behaves similarly to `__str__` and `__unicode__` in 2.7) exists for a new built-in for constructing byte arrays.
- Since division defaults to true division in Python 3, `__div__` is gone in Python 3
- `__coerce__` is gone due to redundancy with other magic methods and confusing behavior
- `__cmp__` is gone due to redundancy with other magic methods
- `__nonzero__` has been renamed to `__bool__`